TIBCO Enterprise Message Service™

User Guide

Version 10.3.0 | February 2024

Contents

About this Product

TIBCO is proud to announce the latest release of TIBCO Enterprise Message Service™ software.

This release is the latest in a long history of TIBCO products that leverage the power of the Information Bus® technology to enable truly event-driven IT environments. To find out more about how TIBCO Enterprise Message Service software and other TIBCO products are powered by TIB® technology, please visit us at www.tibco.com.

TIBCO Enterprise Message Service software lets application programs send and receive messages according to the Jakarta Messaging protocol. It also integrates with TIBCO FTL and TIBCO Rendezvous.

TIBCO EMS software is part of TIBCO® Messaging.

Product Editions

TIBCO Messaging is available in a community edition and an enterprise edition.

TIBCO Messaging - Community Edition is ideal for getting started with TIBCO Messaging, for implementing application projects (including proof of concept efforts), for testing, and for deploying applications in a production environment. Although the community license limits the number of production clients, you can easily upgrade to the enterprise edition as your use of TIBCO Messaging expands.

The community edition is available free of charge. It is a full installation of the TIBCO Messaging software, with the following limitations and exclusions:

- Users may run up to 100 connections in a production environment. A connection is as defined in the Jakarta Messaging Specification and established between an application built with the TIBCO Enterprise Message Service Client Libraries and an instance of the TIBCO Enterprise Message Service Server.
- Users do not have access to TIBCO Support, but you can use TIBCO Community as a resource (https://community.tibco.com).
- Available on Red Hat Enterprise Linux Server, Microsoft Windows & Windows Server and Apple macOS.

TIBCO Messaging - Community Edition has the following additional limitations and exclusions:

- Excludes Fault Tolerance of the server.
- Excludes Unshared State Failover.
- Excludes Routing of messages between servers.
- Excludes JSON configuration files.
- Excludes EMS OSGi bundle.
- Excludes grid store and FTL store types.

TIBCO Messaging - Enterprise Edition is ideal for all application development projects, and for deploying and managing applications in an enterprise production environment. It includes all features presented in this documentation set, as well as access to TIBCO Support.

Overview

The following sections contain a general overview of Jakarta Messaging and TIBCO Enterprise Message Service concepts.

Jakarta Messaging Overview

Jakarta Messaging is a Java framework specification for messaging between applications. This specification was developed to supply a uniform messaging interface among enterprise applications.

Using a message service allows you to integrate the applications within an enterprise. For example, you may have several applications: one for customer relations, one for product inventory, and another for raw materials tracking. Each application is crucial to the operation of the enterprise, but even more crucial is communication between the applications to ensure the smooth flow of business processes. Message-oriented-middleware (MOM) creates a common communication protocol between these applications and allows you to easily integrate new and existing applications in your enterprise computing environment.

The Jakarta Messaging framework (an interface specification, not an implementation) is designed to supply a basis for MOM development. TIBCO Enterprise Message Service implements Jakarta Messaging and integrates support for connecting with TIBCO FTL and TIBCO Rendezvous. This chapter describes the concepts of Jakarta Messaging and its implementation in TIBCO Enterprise Message Service. For more information on Jakarta Messaging requirements and features, see the following sources:

- Jakarta Messaging specification, available through https://jakarta.ee/specifications/messaging/2.0.
- *Java Message Service* by Richard Monson-Haefel and David A. Chappell, O'Reilly and Associates, Sebastopol, California, 2001.

Jakarta Messaging Compliance

TIBCO Enterprise Message Service 10.3 has passed the Eclipse Foundation Technology Compatibility Kit (TCK) tests for Jakarta Messaging 2.0 and 3.0. Therefore, Enterprise Message Service 10.3 is compliant with the Jakarta Messaging 2.0 and 3.0 specifications, assuming the following requirements are met:

- All EMS software must be run on a supported operating system. Supported systems are listed in the readme file.
- The EMS software must be properly installed to include correct versions of software the EMS is dependent on.
- The EMS server configuration parameter `jms_2_0_compliance` must be set to `true`.

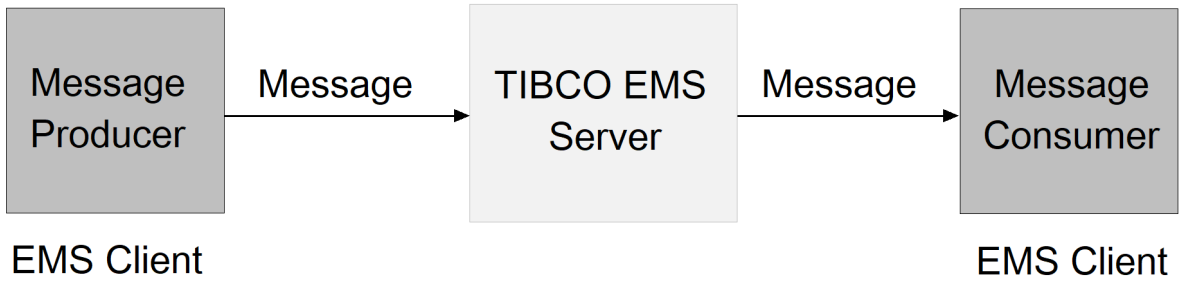
Note: Jakarta Messaging 2.0 TCK tests were run using Open Java Development Kit 8, as the corresponding version of the TCK does not support later Java versions.
Jakarta Messaging 3.0 TCK tests were run using Open Java Development Kit 11.

Jakarta Messaging Message Models

Jakarta Messaging is based on creation and delivery of messages. Messages are structured data that one application sends to another.

The creator of the message is known as the *producer* and the receiver of the message is known as the *consumer*. The TIBCO EMS server acts as an intermediary for the message and manages its delivery to the correct destination. The server also provides enterprise-class functionality such as fault-tolerance, message routing, and communication with TIBCO FTL and TIBCO Rendezvous.

The following image illustrates an application producing a message, sending it by way of the server, and a different application receiving the message.



Jakarta Messaging supports these messaging models:

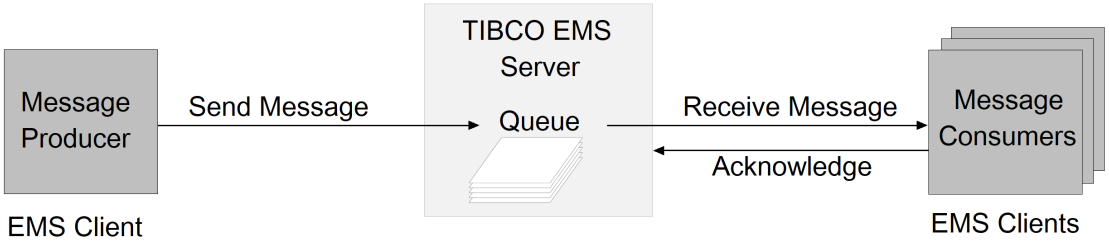
- Point-to-Point (queues)
- Publish and Subscribe (topics)

Point-to-Point

Point-to-point messaging has one producer and one consumer per message. This style of messaging uses a *queue* to store messages until they are received. The message producer sends the message to the queue; the message consumer retrieves messages from the queue and sends acknowledgment that the message was received.

More than one producer can send messages to the same queue, and more than one consumer can retrieve messages from the same queue. The queue can be configured to be exclusive, if desired. If the queue is exclusive, then all queue messages can only be retrieved by the first consumer specified for the queue. Exclusive queues are useful when you want only one application to receive messages for a specific queue. If the queue is not exclusive, any number of receivers can retrieve messages from the queue. Non-exclusive queues are useful for balancing the load of incoming messages across multiple receivers. Regardless of whether the queue is exclusive or not, only one consumer can ever consume each message that is placed on the queue.

The following image illustrates point-to-point messaging using a non-exclusive queue. Each message consumer receives a message from the queue and acknowledges receipt of the message. The message is taken off the queue so that no other consumer can receive it.

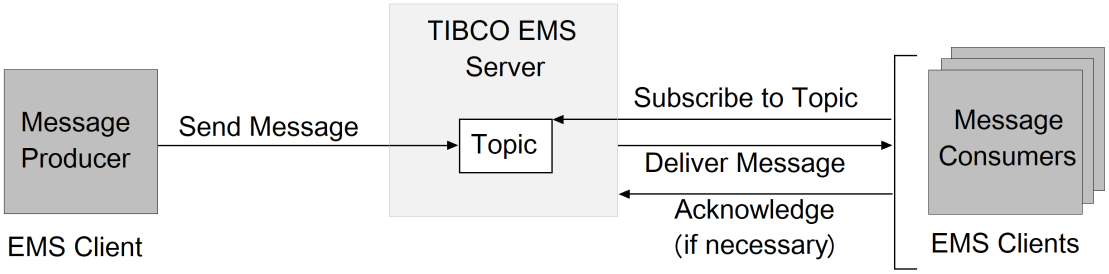


Publish and Subscribe

In a publish and subscribe message system, producers address messages to a *topic*. In this model, the producer is known as a *publisher* and the consumer is known as a *subscriber*.

Many publishers can publish to the same topic, and a message from a single publisher can be received by many subscribers. Subscribers subscribe to topics, and all messages published to the topic are received by all subscribers to the topic. This type of message protocol is also known as *broadcast* messaging because messages are sent over the network and received by all interested subscribers, similar to how radio or television signals are broadcast and received.

The following image illustrates publish and subscribe messaging. Each message consumer subscribes to a topic. When a message is published to that topic, all subscribed consumers receive the message.



Durable Subscribers for Topics

By default, subscribers only receive messages when they are active. If messages arrive on the topic when the subscriber is not available, the subscriber does not receive those

messages.

The EMS APIs allow you to create durable subscribers to ensure that messages are received, even if the message consumer is not currently running. Messages for durable subscriptions are stored on the server as long as durable subscribers exist for the topic, or until the message expiration time for the message has been reached, or until the storage limit has been reached for the topic. Durable subscribers can receive messages from a durable subscription even if the subscriber was not available when the message was originally delivered.

When an application restarts and recreates a durable subscriber with the same ID, all messages stored on the server for that topic are delivered to the durable subscriber.

See Create a Message Consumer for details on how to create durable subscribers.

Shared Subscriptions for Topics

Shared subscriptions allow an application to share the work of receiving messages on a topic among multiple message consumers.

When multiple consumers share a subscription, only one consumer in the group receives each new message. This is similar in function to a queue; however, there are no restrictions placed on the type of consumers to the topic, meaning that a topic can have a mix of shared and not shared, durable and non-durable consumers. When a message is published to the topic, the same message goes to all the matching subscriptions.

Shared subscriptions are created with a specific name, and optionally a client ID. Consumers sharing the subscription specify this name when subscribing to the topic. If the shared subscription type is durable, it persists and continues to accumulate messages until deleted. If the shared subscription type is non-durable, it persists only so long as subscribers exist.

For example, the topic `foo` might have the following subscriptions:

- not shared, non-durable subscription
- not shared, durable subscription
- shared, non-durable subscription called `mySharedSub` with three shared consumers
- shared, durable subscription called `myDurableSharedSub` with two shared consumers

If a message is received on `foo`, each of the above four subscriptions receive that same message. For the shared subscriptions `mySharedSub` and `myDurableSharedSub`, the message is delivered to only one of its respective shared consumers.

If the shared consumers of the shared durable subscription `myDurableSharedSub` are closed, then the shared durable subscription continues to exist and accumulate messages until it is deleted, or until the application creates a new durable shared consumer named `myDurableSharedSub` to resume this subscription. If the shared consumers of `mySharedSub` are all closed, the subscription is removed from topic foo.

See Create a Message Consumer for details on how to create shared subscriptions.

EMS Destination Features

TIBCO Enterprise Message Service allows you to configure destinations to enhance the functionality of each messaging model.

The EMS destination features allow you to:

- Set a `secure` mode for access control at the queue or topic level, so that some destinations may require permission and others may not. See Destination Control.
- Set threshold limits for the amount of memory used by the EMS server to store messages for a topic or a queue and fine-tune the server's response to when the threshold is exceeded. See `flowControl` and `overflowPolicy`.
- Route messages sent to destinations to other servers. See Routes.
- Create bridges between destinations of the same or different types to create a hybrid messaging model for your application. This can be useful if your application requires that you send the same message to both a topic and a queue. For more information on creating bridges between destinations and situations where this may be useful, see Destination Bridges.
- Control the flow of messages to a destination. This is useful when message producers send messages much faster than message consumers can receive them. For more information on flow control, see Flow Control.
- Exchange messages with TIBCO FTL and TIBCO Rendezvous. Queues can receive messages. Topics can either receive or send messages. See Interoperation with TIBCO FTL and Interoperation with TIBCO Rendezvous.
- Set queues to be exclusive or non-exclusive. Only one receiver can receive messages

from an exclusive queue. More than one receiver can receive messages from non-exclusive queues. See `exclusive`.

- Specify a redelivery policy for queues. When messages must be redelivered, you can specify a property on the queue that determines the maximum number of times a message should be redelivered. See `maxRedelivery`.
- Trace and log all messages passing through a destination. See `trace`.
- Include the user name of the message producer in the message. See `sender_name` and `sender_name_enforced`.
- Administrator operations can use wildcards in destination names. The wildcard destination name is the parent, and any names that match the wildcard destination name inherit the properties of the parent. See Wildcards.
- Use the `store` property to cause messages sent to a destination to be written to a store file. Set the destination store to `store=$sys.failsafe` to direct the server to write messages to the file synchronously and guarantee that messages are not lost under any circumstances. See `store` for more information.
- Specify that a consumer is to receive batches of messages in the background to improve performance. Alternatively, you can specify that queue receivers are to only receive one message at a time. See `prefetch` for more information.

Client APIs

Java applications use the `javax.jms` package to send or receive Jakarta Messaging messages. This is a standard set of interfaces, specified by the Jakarta Messaging specification, for creating the connection to the EMS server, specifying the type of message to send, and creating the destination (topic or queue) on which to send or receive messages. You can find a description of the `javax.jms` package in *TIBCO Enterprise Message Service Java API Reference* included in the online documentation.

Because EMS implements the Jakarta Messaging standard, you can also view the documentation on these interfaces along with the Jakarta Messaging specification at https://jakarta.ee/specifications/messaging/2.0.

TIBCO Enterprise Message Service includes parallel APIs for other development environments. See the following for more information:

- *TIBCO Enterprise Message Service C & COBOL API Reference*

- *TIBCO Enterprise Message Service .NET API Reference* (online documentation)

Sample Code

EMS includes several example programs that illustrate the various features of EMS.

You may wish to view these example programs when reading about the corresponding features in this manual. The examples are included in the samples subdirectory of the EMS installation directory.

For more information about running the examples, see Getting Started.

TIBCO Rendezvous Java Applications

EMS includes a Java class that allows pure Java TIBCO Rendezvous applications to connect directly with the EMS server.

For more information see Pure Java Rendezvous Programs.

Administration

EMS provides mechanisms for administering server operations and creating objects that are managed by the server, such as ConnectionFactories and Destinations.

Administration functions can be issued either using the command-line administration tool or by creating an application that uses the administration API (either Java or .NET). The command-line administration tool is described in EMS Administration Tool. The administration APIs are described in the online documentation.

The administration interfaces allow you to create and manage administered objects such as ConnectionFactories, Topics, and Queues. EMS clients can retrieve references to these administered objects by using Java Naming and Directory Interface (JNDI). Creating static administered objects allows clients to use these objects without having to implement the objects within the client.

Administering the Server

EMS has several administration features that allow you to monitor and manage the server. The following table provides a summary of administration features and details where in the documentation you can find more information.

Feature	More Information
Configuration files allow you to specify server characteristics.	Configuration Files
Administration tool provides a command line interface for managing the server.	EMS Administration Tool
Authentication and permissions can restrict access to the server and to destinations. You can also specify who can perform administrative activities with administrator permissions.	Authentication and Permissions
Configure log files to provide information about various server activity.	Monitor Server Activity
The server can publish messages when various system events occur. This allows you to create robust monitoring applications that subscribe to these system monitor topics.	Monitor Server Activity
The server can provide various statistics at the desired level of detail.	Monitor Server Activity

User and Group Management

EMS provides facilities for creating and managing users and groups locally for the server. The EMS server can also use an external system, such as an LDAP server (using JAAS modules) for authenticating users and storing group information.

See Authentication and Permissions for more information about configuring EMS to work with external systems for user and group management.

Using TIBCO Hawk

You can use TIBCO Hawk® for monitoring and managing the EMS server. See TIBCO Hawk documentation for more information.

Modes, Roles, and States

The *mode* of an EMS server is determined by its configuration, and dictates how it operates in its environment. If a fault tolerant mode is selected, two EMS servers are required and each operates in a defined *role*. How an EMS server is operating at any given moment can be determined by viewing its fault tolerant *state*.

For example, an EMS server operating in fault tolerant mode can play either a primary or secondary role. Once both EMS servers in the fault tolerant pair have been started, one of the two servers will be in the active state while its peer will be in the standby state. In the event of a failover, the server that was standby becomes active.

Modes

By default, the EMS server operates in standalone mode. However, it can also be configured to run in a fault tolerant mode:

- Standalone — the default EMS server mode.
- Classic Fault Tolerant — configured through the `ft_active` parameter if using file-based stores or grid stores and through the FTL configuration if using FTL stores.

Roles

Each server operating in a fault tolerant mode has a distinct role: primary or secondary.

These roles are implicit for EMS servers started using `tibemsd.conf` files. They are explicit for EMS servers started using a JSON configuration file. For JSON-configured servers, the primary server is the EMS server started without the `-secondary` command line parameter, while the secondary server is started with it. In the `.conf` files, each server in the fault tolerant pair has a distinct `tibemsd.conf` file.

Note: Although EMS servers using FTL stores are JSON-configured, the primary and secondary roles for a fault-tolerant EMS server pair are still implicit.

States

The state of the EMS server tells you about its current operations.

Use the `info` or `show state` command in the administration tool to determine the state of the EMS server.

State	Description
`active`	The server is fully operational and ready to service clients.
`standby`	The server is in classic fault tolerant mode and is ready to take over should its peer fail.

Security

For communication security between servers and clients, and between servers and other servers, you must explicitly configure TLS within EMS.

Transport Layer Security (TLS) is a protocol for transmitting encrypted data over the Internet or an internal network. TLS works by using public and private keys to encrypt data that is transferred over the TLS connection.

EMS supports TLS between the following components:

- between an EMS client and the EMS server
- between the administration tool and the EMS server
- between the administration APIs and the EMS server
- between routed servers
- between fault-tolerant servers (not applicable when using FTL stores)

See TLS Protocol for more information about TLS support in EMS.

Fault Tolerance

You can configure EMS servers as primary and secondary servers to provide fault tolerance for your environment. The primary and secondary servers act as a pair, one of them starting out in the active state and the other in the standby state. The active server accepts client connections and performs the work of handling messages, while the standby server acts as a backup in case of failure. If the active server fails, the standby server assumes operation and becomes the active server.

See Fault Tolerance for more information about the fault-tolerance features of EMS.

Routing

EMS provides the ability for servers to route messages between each other. Topic messages can be routed across multiple hops, provided there are no cycles (that is, the message can not be routed to any server it has already visited). Queue messages can travel at most one hop to any other server from the server that owns the queue.

EMS stores and forwards messages in most situations to provide operation when a route is not connected.

See Routes for more information about the routing features of EMS.

Integrating with Third-Party Products

EMS allows you to work with third-party naming/directory service products or with third-party application servers.

Transaction Support

TIBCO Enterprise Message Service can integrate with Jakarta Transactions API (JTA) compliant transaction managers. EMS implements all interfaces necessary to be JTA compliant.

The EMS C API is compliant with the X/Open XA specification. The EMS .NET API supports Microsoft Distributed Transaction Coordinator (MS DTC) with .NET Framework. Transactions

created using MSDTC in a .NET Framework client are seen as XA transactions in C and Java clients.

Containerization

TIBCO Enterprise Message Service supports containerization.

Refer to the corresponding TIBCO Community pages for specific solutions involving environments such as Docker, Kubernetes, OpenShift, etc.

Messages

The following sections provide an overview of EMS messages.

EMS Extensions to Jakarta Messaging Messages

The Jakarta Messaging specification details a standard format for the header and body of a message. Properties are provider-specific and can include information on specific implementations or enhancements to Jakarta Messaging functionality. See EMS Message Properties for the list of message properties that are specific to EMS.

In addition to the EMS message properties, EMS provides a select number of extensions to Jakarta Messaging. These are:

- The Jakarta Messaging standard specifies two delivery modes for messages, PERSISTENT and `NON_PERSISTENT`. EMS also includes a `RELIABLE_DELIVERY` mode that eliminates some of the overhead associated with the other delivery modes. See `RELIABLE_DELIVERY`.
- For consumer sessions, you can specify a `NO_ACKNOWLEDGE` mode so that consumers do not need to acknowledge receipt of messages, if desired. EMS also provides an `EXPLICIT_CLIENT_ACKNOWLEDGE` and `EXPLICIT_CLIENT_DUPS_OK_ACKNOWLEDGE` mode that restricts the acknowledgment to single messages. See Message Acknowledgement.
- EMS extends the `MapMessage` and `StreamMessage` body types. These extensions allow EMS to exchange messages with TIBCO Rendezvous, which contains certain features not available within the Jakarta Messaging `MapMessage` and `StreamMessage`.

  TIBCO Enterprise Message Service adds these two extensions to the `MapMessage` and `StreamMessage` body types:

  - You can insert another `MapMessage` or `StreamMessage` instance as a submessage into a `MapMessage` or `StreamMessage`, generating a series of nested messages, instead of a flat message.
  - You can use arrays as well as primitive types for the values.

    These extensions add considerable flexibility to the `MapMessage` and

`StreamMessage` body types. However, they are extensions and therefore not compliant with Jakarta Messaging specifications. Extended messages are tagged as extensions with the vendor property tag `JMS_TIBCO_MSG_EXT`.

For more information on compatibility with Rendezvous messages, see Message Body.

Jakarta Messaging Message Structure

Jakarta Messaging messages have a standard structure.

The Jakarta Messaging message structure includes the following sections:

- Header (required)
- Properties (optional)
- Body (optional)

Jakarta Messaging Message Header Fields

The header contains predefined fields that contain values used to route and deliver messages.

Header Field	Set by	Comments
`JMSDestination`	send or publish method	Destination to which message is sent
`JMSDeliveryMode`	send or publish method	Persistent or non-persistent message. The default is persistent. EMS extends the delivery mode to include a `RELIABLE_DELIVERY` mode.
`JMSExpiration`	send or publish method	Length of time that message will live before expiration. If set to 0, message does not expire. The time-to-live is specified in milliseconds.

Header Field	Set by	Comments
		If the server `expiration` property is set for a destination, it will override the `JMSExpiration` value set by the message producer.
`JMSDeliveryTime`	send or publish method	Read-only field. If the message producer has a delivery delay set, then the time returned here after calling the send method represents the earliest time when the EMS server will deliver the message to consumers. Once the message has been received, it carries that same value. This value is calculated by adding the delivery delay value held by the message producer to the time the message was sent. For transactions, the delivery time is calculated using the time the client sends the message, not the time the transaction is committed. For more information, see Delivery Delay.
`JMSPriority`	send or publish method	Uses a numerical ranking, between 0 and 9, to define message priority as normal or expedited. Larger numbers represent higher priority. See Message Priority for more information.
`JMSMessageID`	send or publish method	Value uniquely identifies each message sent by a provider.
`JMSTimestamp`	send or publish method	Timestamp of time when message was handed off to a provider to be sent. Message may actually be sent later than this timestamp.
`JMSCorrelationID`	message client	This ID can be used to link messages, such as linking a response message to a request message. Entering a value in this field is optional. The JMS Correlation ID has a recommended maximum of 4 KB. Higher values may result in the message being rejected.

Header Field	Set by	Comments
`JMSReplyTo`	message client	A destination to which a message reply should be sent. Entering a value for this field is optional.
`JMSType`	message client	Message type identifier.
`JMSRedelivered`	Jakarta Messaging provider	If this field is set, it is possible that the message was delivered to the client earlier, but not acknowledged at that time.

EMS Message Properties

In the properties area, applications, vendors, and administrators on Jakarta Messaging systems can add optional properties. The properties area is optional, and can be left empty. The Jakarta Messaging specification describes the Jakarta Messaging message properties. This section describes the message properties that are specific to EMS.

TIBCO-specific property names begin with `JMS_TIBCO`. Client programs may use the TIBCO-specific properties to access EMS features, but not for communicating application-specific information among client programs.

The EMS properties are summarized in the following table and described in more detail in subsequent sections.

Property	Description	More Info
`JMS_TIBCO_CM_PUBLISHER`	Correspondent name of an RVCM sender for messages imported from TIBCO Rendezvous.	Import RVCM
`JMS_TIBCO_CM_SEQUENCE`	Sequence number of an RVCM message imported from TIBCO Rendezvous.	Import RVCM
`JMS_TIBCO_COMPRESS`	Allows messages to be	Message

Property	Description	More Info
	compressed for more efficient storage.	Compression
JMS_TIBCO_DISABLE_SENDER	Specifies that the user name of the message sender should not be included in the message, if possible.	Including the Message Sender
JMS_TIBCO_IMPORTED	Set by the server when the message has been imported from TIBCO FTL or TIBCO Rendezvous.	Import (for FTL) Import (for RV)
JMS_TIBCO_MSG_EXT	Extends the functionality of the MapMessage and StreamMessage body types to include submessages or arrays.	EMS Extensions to Jakarta Messaging Messages Import (for RV)
JMS_TIBCO_MSG_TRACE	Specifies the message should be traced from producer to consumer.	Message Tracing
JMS_TIBCO_PRESERVE_UNDELIVERED	Specifies the message is to be placed on the undelivered message queue if the message must be removed.	Undelivered Message Queue
JMS_TIBCO_SENDER	Contains the user name of the message sender.	Including the Message Sender

Undelivered Message Queue

If a message could not be delivered for one of the reasons below, the server checks the message's JMS_TIBCO_PRESERVE_UNDELIVERED property. If that property is set to true, the server moves the message to the undelivered message queue, $sys.undelivered. Otherwise, the message is deleted by the server.

The server will examine the JMS_TIBCO_PRESERVE_UNDELIVERED property of the message if any of the following conditions are met:

- the message has expired
- the message has exceeded the value specified by the maxRedelivery property on a queue
- the message had a delivery delay that has expired and was sent to a destination that has reached its maxmsgs limit and also has overflowPolicy=rejectIncoming

$sys.undelivered is a system queue that is always present and cannot be deleted. To make use of it, the application that sends or publishes the message must set the boolean JMS_TIBCO_PRESERVE_UNDELIVERED property to true before sending or publishing the message.

You can only set the undelivered property on individual messages, there is no way to set the undelivered message queue as an option at the per-topic or per-queue level.

You should create a queue receiver to receive and handle messages as they arrive on the undelivered message queue. If you wish to remove messages from the undelivered message queue without receiving them, you can purge the $sys.undelivered queue with the administration tool, using the purge queue command described under Command Listing. You can also remove messages using the administrative API included with TIBCO Enterprise Message Service.

Note that $sys.undelivered ignores the global destination property setting. Messages in the undelivered message queue are not routed to other servers.

Filtering Messages in the Undelivered Message Queue

You can filter messages in the undelivered message queue by destination using a selector. Note that this is an exception to the Jakarta Messaging Specification that is made only for messages in the undelivered message queue. In the undelivered message queue, the JMSDestination header field can be used in a selector the same way that a supported header field or any other message property with a string value is used.

The expected value of the JMSDestination field depends on the original message destination type and name:

```
JMSDestination operator 'Topic|Queue[destination_name]'
```

For example:

```
JMSDestination='Queue[A]'
JMSDestination='Topic[B7]'
JMSDestination NOT LIKE 'Queue[A]'
JMSDestination LIKE 'Queue[A]'
JMSDestination LIKE 'Q%'
JMSDestination IS NOT NULL
JMSDestination IN ('Queue[H]','Queue[J]')
JMSDestination NOT IN ('Topic[H]','Topic[J]')
JMSDestination='Queue[A]' OR JMSDestination='Queue[B]'
```

Including the Message Sender

Within a message, EMS can supply the user name given by the message producer when a connection is created. The `sender_name` and `sender_name_enforced` server properties on the destination determine whether the message producer's user name is included in the sent message.

When a user name is included in a message, a message consumer can retrieve that user name by getting the string message property named `JMS_TIBCO_SENDER`.

When the `sender_name` property is enabled and the `sender_name_enforced` property is not enabled on a destination, message producers can specify that the user name is to be left out of the message. Message producers can specify the `JMS_TIBCO_DISABLE_SENDER` boolean property for a particular message, and the message producer's user name will not be included in the message. However, if the `sender_name_enforced` property is enabled, the `JMS_TIBCO_DISABLE_SENDER` property is ignored and the user name is always included in the message.

Jakarta Messaging Message Bodies

A Jakarta Messaging message has one of several types of message bodies, or no message body at all.

The types of messages are described in the following table.

Message Type	Contents of Message Body
Message	This message type has no body. This is useful for simple event

Message Type	Contents of Message Body
	notification.
TextMessage	A `java.lang.String`.
MapMessage	A set of name/value pairs. The names are `java.lang.String` objects, and the values are Java primitive value types or their wrappers. The entries can be accessed sequentially by enumeration or directly by name. The order of entries is undefined. When EMS is exchanging messages with Rendezvous, you can generate a series of nested MapMessages, as described in EMS Extensions to Jakarta Messaging Messages.
BytesMessage	A stream of uninterrupted bytes. The bytes are not typed; that is, they are not assigned to a primitive data type.
StreamMessage	A stream of primitive values in the Java programming language. Each set of values belongs to a primitive data type, and must be read sequentially. When EMS is exchanging messages with Rendezvous, you can generate a series of nested StreamMessages, as described in EMS Extensions to Jakarta Messaging Messages.
ObjectMessage	A serializable object constructed in the Java programming language.

Maximum Message Size

EMS supports messages up to a maximum size of 2 GB. However, we recommend that application programs use smaller messages, since messages approaching this maximum size will strain the performance limits of most current hardware and operating system platforms.

Message Priority

The Jakarta Messaging specification includes a `JMSPriority` message header field in which senders can set the priority of a message, as a value in the range [0,9]. EMS *does* support

message priority (though it is optional, and other vendors might not implement it).

When the EMS server has several messages ready to deliver to a consumer client, and must select among them, then it delivers messages with higher priority before those with lower priority.

However, priority ordering applies only when the server has a *backlog* of deliverable messages for a consumer. In contrast, when the server has only one message at a time to deliver to a consumer, then the priority ordering feature will not be apparent.

You can set default message priority for the Message Producer, as described in Configure a Message Producer. The default priority can be overridden by the client when sending a message, as described in Send Messages.

Also refer to Jakarta Messaging Specification, chapter 3.4.10.

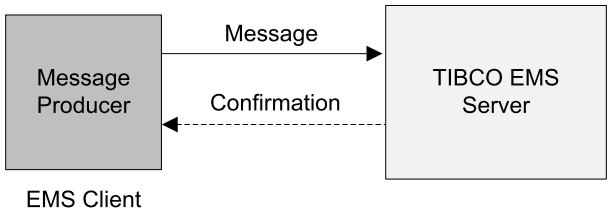
Message Delivery Modes

The `JMSDeliveryMode` message header field defines the delivery mode for the message. Jakarta Messaging supports `PERSISTENT` and `NON_PERSISTENT` delivery modes for both topic and queue. EMS extends these delivery modes to include a `RELIABLE_DELIVERY` mode.

You can set the default delivery mode for the Message Producer, as described in Configure a Message Producer. This default delivery mode can be overridden by the client when sending a message, as described in Send Messages.

PERSISTENT

When a producer sends a `PERSISTENT` message, the producer must wait for the server to reply with a confirmation. The message is persisted on disk by the server. This delivery mode ensures delivery of messages to the destination on the server in almost all circumstances. However, the cost is that this delivery mode incurs two-way network traffic for each message or committed transaction of a group of messages.



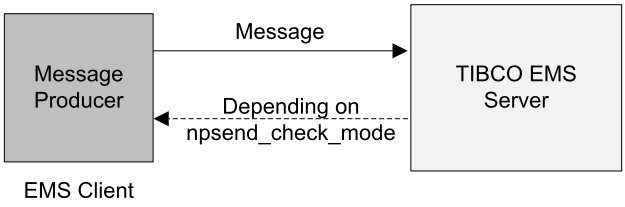
NON_PERSISTENT

Sending a `NON_PERSISTENT` message omits the overhead of persisting the message on disk to improve performance.

If `authorization` is disabled on the server, the server does not send a confirmation to the message producer.

If `authorization` is enabled on the server, the default condition is for the producer to wait for the server to reply with a confirmation in the same manner as when using `PERSISTENT` mode.

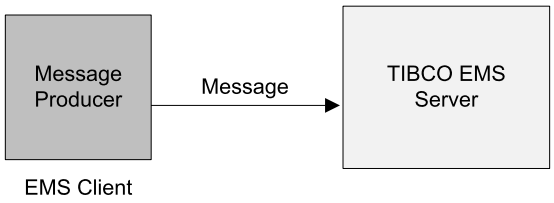
Regardless of whether `authorization` is enabled or disabled, you can use the `npsend_check_mode` parameter in the `tibemsd.conf` file to specify the conditions under which the server is to send confirmation of `NON_PERSISTENT` messages to the producer. See the description for `npsend_check_mode` for details.



RELIABLE_DELIVERY

EMS extends the Jakarta Messaging delivery modes to include reliable delivery. Sending a `RELIABLE_DELIVERY` message omits the server confirmation to improve performance regardless of the authorization setting.

Also see `authorization`.



When using `RELIABLE_DELIVERY` mode, the server never sends the producer a receipt confirmation or access denial and the producer does not wait for it. Reliable mode decreases the volume of message traffic, allowing higher message rates, which is useful for messages containing time-dependent data, such as stock price quotations.

When you use the reliable delivery mode, the client application does not receive any response from the server. Therefore, all publish calls will always succeed (not throw an exception) unless the connection to the server has been terminated.

In some cases a message published in reliable mode may be disqualified and not handled by the server because the destination is not valid or access has been denied. In this case, the message is not sent to any message consumer. However, unless the connection to the server has been terminated, the publishing application will not receive any exceptions, despite the fact that no consumer received the message.

How EMS Manages Persistent Messages

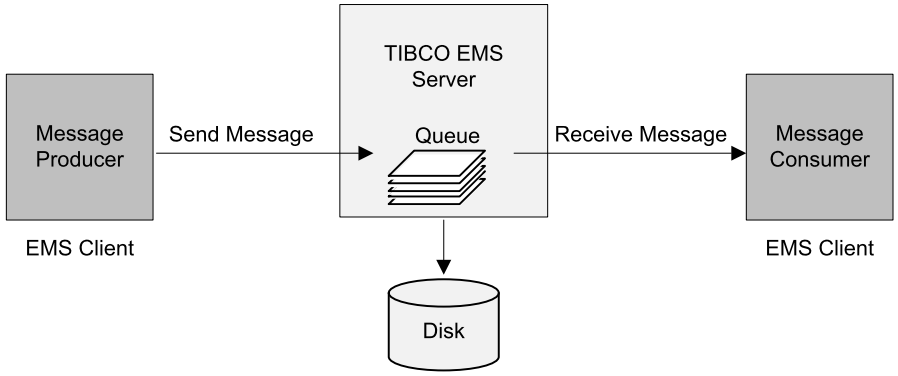
Jakarta Messaging defines two message delivery modes, `PERSISTENT` and `NON_PERSISTENT`, and EMS defines a `RELIABLE_DELIVERY` mode.

For more information see Message Delivery Modes.

`NON_PERSISTENT` and `RELIABLE_DELIVERY` messages are never written to persistent storage. `PERSISTENT` messages are written to persistent storage when they are received by the EMS server.

Persistent Messages Sent to Queues

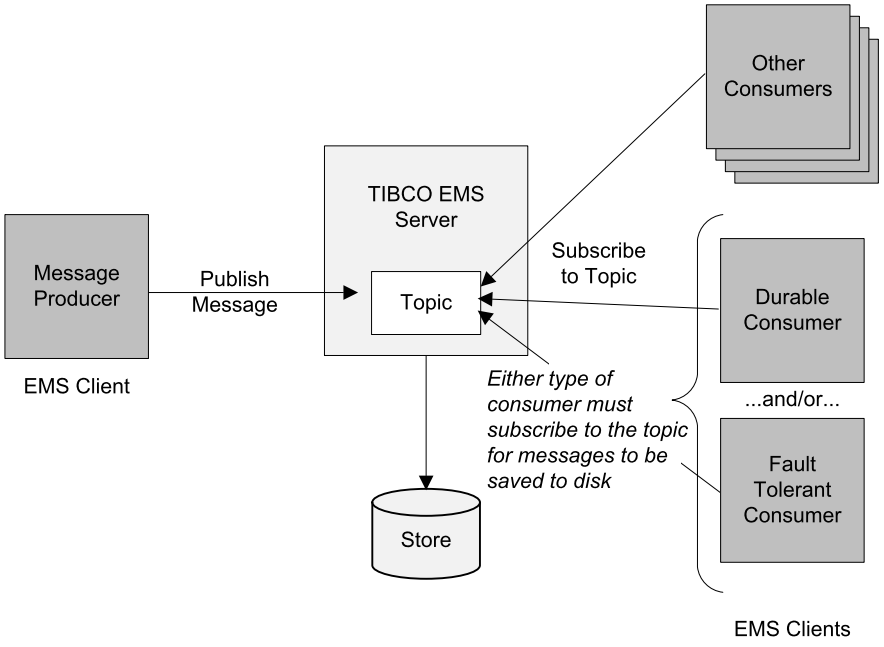
Persistent messages sent to a queue are always written to disk. Should the server fail before sending persistent messages to subscribers, the server can be restarted and the persistent messages will be sent to the subscribers when they reconnect to the server.



Persistent Messages Published to Topics

Persistent messages published to a topic are written to disk *only* if that topic has at least one durable subscriber or one subscriber with a fault-tolerant connection to the EMS server.

In the absence of a durable subscriber or subscriber with a fault-tolerant connection, there are no subscribers that need messages resent in the event of a server failure. In this case, the server does not needlessly save persistent messages. This improves performance by eliminating the unnecessary disk I/O to persist the messages.



This behavior is consistent with the Jakarta Messaging specification because durable subscribers to a topic cause published messages to be saved. Additionally, subscribers to a topic that have a fault-tolerant connection need to receive messages from the new active server after a failover. However, non-durable subscribers without a fault-tolerant connection that re-connect after a server failure are considered newly created subscribers and are not entitled to receive any messages created prior to the time they are created (that is, messages published before the subscriber re-connects are not resent).

Persistent Messages and Synchronous File Storage

When using file-based stores or FTL stores, persistent messages received by the EMS server are by default written asynchronously to disk. This means that, when a producer sends a persistent message, the server does not wait for the write-to-disk operation to complete before returning control to the producer.

Should the server fail before completing the write-to-disk operation, the producer has no way of detecting the failure to persist the message and taking corrective action.

You can set the mode parameter to sync for a given file-based store or FTL store to specify that persistent messages for the topic or queue be synchronously written to disk. When `mode = sync`, the producer remains blocked until the write-to-disk operation is completed. In the case of FTL stores, the producer will continue to remain blocked until

the message has been replicated to the other EMS servers in the cluster and written to disk. This parameter is not relevant in the case of grid stores, where the disk access mode is always synchronous.

When using file-based stores, the EMS server writes persistent messages to a store file. To prevent two servers from using the same store file, each server restricts access to its store file for the duration of the server process. For details on how EMS manages access to shared file-based stores, see Managing Access to Shared File-Based Stores.

When using FTL stores, the EMS server stores persistent messages in FTL. See Persistence with FTL Stores for more information.

Store Messages in Multiple Stores

The EMS server writes `PERSISTENT` messages to disk while waiting for confirmation of receipt from the subscriber. Messages are persisted to a *store*. The EMS server can write messages to different types of *stores*: file-based stores, grid stores, and FTL stores.

By default, the EMS server writes persistent messages to file-based stores. There are three default stores, as described in Default Stores. You can configure the system to change the default stores' properties, and also to store persistent messages to one or more store files, filtering them by destination. Stores are defined in the `stores.conf` configuration file, and associated with a destination using the `store` destination property.

Stores have properties that allow you to control how the server manages them. For example:

- When using file-based stores:
  - Preallocate disk space for the store file.
  - Truncate the file periodically to relinquish disk space.
  - Specify whether messages are written synchronously or asynchronously.
- When using grid stores:
  - Specify the rate at which the store contents are scanned.
- When using FTL stores:
  - Specify whether messages are persisted synchronously or asynchronously.

With the multiple stores feature, you can configure your messaging application to store messages in different locations for each application, or create separate stores for related

destinations. For example, you can create one store for messages supporting Marketing, and one for messages supporting Sales. Because stores are configured in the server, they are transparent to clients.

When using the multiple stores feature, all stores must be of the same type. Configuring multiple store types in the same server is not supported.

The EMS Administration Tool allows administrators to review the system's configured stores and their settings by using the `show stores` and `show store` commands.

Store Types

TIBCO Enterprise Message Service allows you to configure several different types of stores, described here.

File-Based Stores

The EMS server stores persistent messages in file-based stores. You can use the default store files, or create your own file-based stores. You direct the EMS server to write messages to these store files by associating a destination with a store.

File-based stores are enabled by default, and the server automatically defines three default stores, described below. You do not need to do anything in order to use the default stores.

The section Configuring File-Based Stores describes how to change store settings or create custom stores.

Grid Stores

The EMS server can store messages in a TIBCO ActiveSpaces data grid. See Grid Stores for a full description of this feature.

FTL Stores

The EMS server can store messages in a TIBCO FTL server cluster. See FTL Stores for a full description of this feature.

Default Stores

The EMS server defines these default stores, and writes persistent messages and meta data to them:

- `$sys.nonfailsafe`—Persistent messages without a store property designation are written to `$sys.nonfailsafe` by default. The server writes messages to this store using asynchronous I/O calls.
- `$sys.failsafe`—Associate a destination with this store to write messages synchronously. The server writes messages to this store using synchronous I/O calls.
- `$sys.meta`—The server writes state information about durable subscribers, fault-tolerant connections, and other metadata in this store.

The EMS server creates these default stores as file-based stores automatically, and no steps are required to enable or deploy them. However, you can change the system configuration to customize the default store file settings, or even override the default store settings to either point to different file location, or write to a grid store or FTL store.

If the EMS server is started as a service under an FTL server, or with grid store command line parameters and the default stores are not present in the configuration, the server automatically creates the three default stores as FTL stores or grid stores respectively.

Configuring File-Based Stores

This section describes the basic steps required to configure file-based stores.

For information on grid stores and FTL stores, see Configuring and Deploying Grid Stores and Configuring and Deploying FTL Stores. Settings for creating and configuring multiple stores are managed in the EMS server, and are transparent to clients. To configure the multiple stores feature, follow these steps:

Procedure

1. Setup and configure stores in the `stores.conf` file.

   Stores are created and configured in the `stores.conf` file. Each store must have a

unique name. The stores are configured through parameters.

File-based stores have two required parameters, type and file, which determine that the store is a file-based store, and set its location and filename. Optional parameters allow you to determine other settings, including how messages are written to the file, the minimum size of the file, and whether the EMS server attempts to truncate the file.

2. Associate destinations with the configured stores.

   Messages are sent to different stores according to their destinations. Destinations are associated with specific stores with the `store` parameter in the `topics.conf` and `queues.conf` files.

When using file-based stores, you can also change store associations dynamically using the `setprop topic` or `setprop queue` command in the EMS Administration Tool.

Multiple destinations can be mapped to the same store, either explicitly or using wildcards. Even if no stores are configured, the server sends persistent messages that are not associated with a store to default stores. See Default Stores for more information.

For details about the store parameter, see store.

Character Encoding in Messages

Character encodings are named sets of numeric values for representing characters. For example, ISO 8859-1, also known as Latin-1, is the character encoding containing the letters and symbols used by most Western European languages.

If your applications are sending and receiving messages that use only English language characters (that is, the ASCII character set), you do not need to alter your programs to handle different character encodings. The EMS server and application APIs automatically handle ASCII characters in messages.

Character sets become important when your application is handling messages that use non-ASCII characters (such as the Japanese language). Also, clients encode messages by default as UTF-8. Some character encodings use only one byte to represent each character, but UTF-8 can potentially use between one and four bytes to represent the same character. For example, the Latin-1 is a single-byte character encoding. If all strings in your messages contain only characters that appear in the Latin-1 encoding, you can potentially improve performance by specifying Latin-1 as the encoding for strings in the message.

EMS clients can specify a variety of common character encodings for strings in messages. The character encoding for a message applies to strings that appear in any of the following places within a message:

- property names and property values
- `MapMessage` field names and values
- data within the message body

The EMS client APIs (Java, .NET and C) include mechanisms for handling strings and specifying the character encoding used for all strings within a message. The following sections describe the implications of string character encoding for EMS clients.

Note: Nearly all character sets include unprintable characters. EMS software does not prevent programs from using unprintable characters. However, messages containing unprintable characters (whether in headers or data) can cause unpredictable results if you instruct EMS to print them. For example, if you enable the message tracing feature, EMS prints messages to a trace log file.

Supported Character Encodings

Each message contains the name of the character encoding used to encode strings within the message. This character encoding name is one of the canonical names for character encodings contained in the Java specification.

You can obtain a list of canonical character encoding names from the java.sun.com website.

Java and .NET clients use these canonical character encoding names when setting or retrieving the character encoding names.

Sending Messages

When a client sends a message, the message stores the character encoding name used for strings in that message. Java clients represent strings using Unicode. A message created by a Java client that does not specify an encoding will use UTF-8 as the named encoding within the message.

UTF-8 uses up to four bytes to represent each character, so a Java client can improve performance by explicitly using a single-byte character encoding, if possible.

Java clients can globally set the encoding to use with the `setEncoding` method or the client can set the encoding for each message with the `setMessageEncoding` method. For more information about these methods, see *TIBCO Enterprise Message Service Java API Reference*.

Typically, C clients manipulate strings using the character encoding of the machine on which they are running. The EMS C client library itself does not do any encoding or decoding of characters. When sending a message, an EMS C client application can use `tibemsMsg_SetEncoding` to put information into the message describing the encoding used. When receiving a message in an EMS C client application, the encoding can be retrieved using `tibemsMsg_GetEncoding`. Use a third party library to do the actual decoding based on the retrieved encoding information.

Message Compression

The TIBCO Enterprise Message Service client can compress the body of a message before sending the message to the server. EMS supports message compression/decompression across client types (Java, C and C#). For example, a Java producer may compress a message and a C consumer may decompress it.

About Message Compression

Message compression is especially useful when messages will be stored on the server (persistent queue messages, or topics with durable subscribers). Setting compression ensures that messages will take less memory space in storage. When messages are compressed and then stored, they are handled by the server in the compressed form. Compression assures that the messages will usually consume less space on disk and will be handled faster by the EMS server.

The compression option only compresses the body of a message. Headers and properties are never compressed. It is best to use compression when the message bodies will be large and the messages will be stored on a server.

When messages will not be stored, compression is not as useful. Compression normally takes time, and therefore the time to send or publish and receive compressed messages is

generally longer than the time to send the same messages uncompressed. There is little purpose to message compression for small messages that are not be stored by the server.

Setting Message Compression

Message compression is specified for individual messages. That is, message compression, if desired, is set at the message level. TIBCO Enterprise Message Service does not define a way to set message compression at the per-topic or per-queue level.

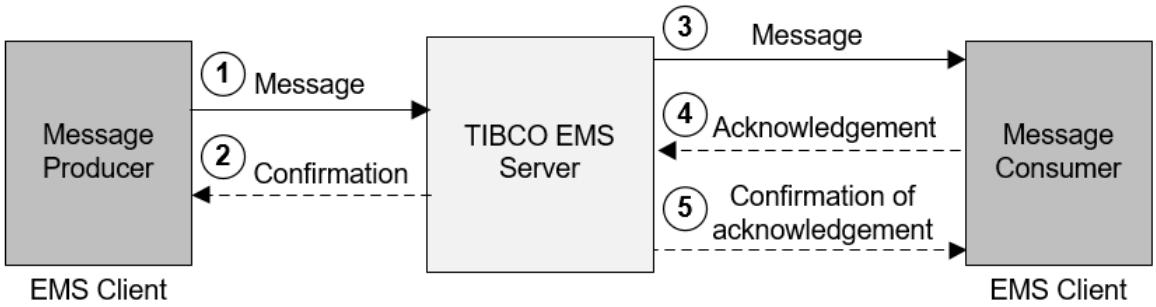
To set message compression, the application that sends or publishes the message must access the message properties and set the boolean property `JMS_TIBCO_COMPRESS` to `true` before sending or publishing the message.

Compressed messages are handled transparently. The client code only sets the `JMS_TIBCO_COMPRESS` property. The client does not need to take any other action. The client automatically decompresses any compressed messages it receives.

Message Acknowledgment

The interface specification for Jakarta Messaging requires that message delivery be guaranteed under many, but not all, circumstances.

The following figure illustrates the basic structure of message delivery and acknowledgment.



The following describes the steps in message delivery and acknowledgment:

1. A message is sent from the message producer to the machine on which the EMS server resides.

2. For persistent messages, the EMS server sends a confirmation to the producer that the message was received.
3. The server sends the message to the consumer.
4. The consumer sends an acknowledgment to the server that the message was received. A session can be configured with a specific session mode that specifies how the consumer-to-server acknowledgment is handled. These session modes are described below.
5. In many cases, the server then sends a confirmation of the acknowledgment to the consumer.

The Jakarta Messaging specification defines three levels of acknowledgment for non-transacted sessions:

- `CLIENT_ACKNOWLEDGE` specifies that the consumer is to acknowledge all messages that have been delivered so far by the session. When using this mode, it is possible for a consumer to fall behind in its message processing and build up a large number of unacknowledged messages.
- `AUTO_ACKNOWLEDGE` specifies that the session is to automatically acknowledge consumer receipt of messages when message processing has finished.
- `DUPS_OK_ACKNOWLEDGE` specifies that the session is to "lazily" acknowledge the delivery of messages to the consumer. "Lazy" means that the consumer can delay acknowledgment of messages to the server until a convenient time; meanwhile the server might redeliver messages. This mode reduces session overhead. Should Jakarta Messaging fail, the consumer may receive duplicate messages.

EMS extends the Jakarta Messaging session modes to include:

- `NO_ACKNOWLEDGE`
- `EXPLICIT_CLIENT_ACKNOWLEDGE`
- `EXPLICIT_CLIENT_DUPS_OK_ACKNOWLEDGE`

Warning: The Simplified Jakarta Messaging API introduced in JMS 2.0 supports the session modes defined in the Jakarta Messaging specification: `CLIENT_ACKNOWLEDGE`, `AUTO_ACKNOWLEDGE`, `DUPS_OK_ACKNOWLEDGE`, and `SESSION_TRANSACTED`. However, it does not support the EMS extended session modes.

The session mode is set when creating a Session, as described in Create a Session.

NO_ACKNOWLEDGE

NO_ACKNOWLEDGE mode suppresses the acknowledgment of received messages.

After the server sends a message to the client, all information regarding that message for that consumer is eliminated from the server. Therefore, there is no need for the client application to send an acknowledgment to the server about the received message. Not sending acknowledgments decreases the message traffic and saves time for the receiver, therefore allowing better utilization of system resources.

Note: Sessions created in no-acknowledge receipt mode cannot be used to create durable subscribers.

Also, queue receivers on a queue that is routed from another server are not permitted to specify NO_ACKNOWLEDGE mode.

EXPLICIT_CLIENT_ACKNOWLEDGE

EXPLICIT_CLIENT_ACKNOWLEDGE is like CLIENT_ACKNOWLEDGE except it acknowledges only the individual message, rather than all messages received so far on the session.

One example of when EXPLICIT_CLIENT_ACKNOWLEDGE would be used is when receiving messages and putting the information in a database. If the database insert operation is slow, you may want to use multiple application threads all doing simultaneous inserts. As each thread finishes its insert, it can use EXPLICIT_CLIENT_ACKNOWLEDGE to acknowledge only the message that it is currently working on.

EXPLICIT_CLIENT_DUPS_OK_ACKNOWLEDGE

EXPLICIT_CLIENT_DUPS_OK_ACKNOWLEDGE is like DUPS_OK_ACKNOWLEDGE except it 'lazily' acknowledges only the individual message, rather than all messages received so far on the session.

Message Selectors

A message selector is a string that lets a client program specify a set of messages, based on the values of message headers and properties. A selector *matches* a message if, after substituting header and property values from the message into the selector string, the string evaluates to true. Consumers can request that the server deliver only those messages that match a selector.

The syntax of selectors is based on a subset of the SQL92 conditional expression syntax.

Identifiers

Identifiers can refer to the values of message headers and properties, but not to the message body. Identifiers are case-sensitive.

Basic Syntax

An identifier is a sequence of letters and digits, of any length, that begins with a letter. As in Java, the set of letters includes _ (underscore) and `$` (dollar).

Illegal

Certain names are exceptions, which cannot be used as identifiers. In particular, `NULL`, `TRUE`, `FALSE`, `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, `IS`, and `ESCAPE` are defined to have special meaning in message selector syntax.

Value

Identifiers refer either to message header names or property names. The type of an identifier in a message selector corresponds to the type of the header or property value. If an identifier refers to a header or property that does not exist in a message, its value is `NULL`.

Literals

String Literal

A string literal is enclosed in single quotes. To represent a single quote within a literal, use two single quotes; for example, `'literal''s'`. String literals use the Unicode character encoding. String literals are case sensitive. The server has a limit of 32,767 string literals in a selector string.

Exact Numeric Literal

An exact numeric literal is a numeric value without a decimal point, such as `57`, `-957`, and `+62`; numbers of `long` are supported.

Approximate Numeric Literal

An approximate numeric literal is a numeric value with a decimal point (such as `7.`, `-95.7`, and `+6.2`), or a numeric value in scientific notation (such as `7E.3` and `-57.9E2`); numbers in the range of `double` are supported. Approximate literals use floating-point literal syntax of the Java programming language.

Boolean Literal

The boolean literals are `TRUE` and `FALSE` (case insensitive).

Internal computations of expression values use a 3-value boolean logic similar to SQL. However, the final value of an expression is always either `TRUE` or `FALSE`, but never `UNKNOWN`.

Expressions

Selectors as Expressions

Every selector is a conditional expression. A selector that evaluates to `true` matches the message; a selector that evaluates to `false` or unknown does not match.

Arithmetic Expression

Arithmetic expressions are composed of numeric literals, identifiers (that evaluate to numeric literals), arithmetic operations, and smaller arithmetic expressions.

Conditional Expression

Conditional expressions are composed of comparison operations, logical operations, and smaller conditional expressions.

Order of Evaluation

Order of evaluation is left-to-right, within precedence levels. Parentheses override this order.

Operators

Case Insensitivity

Operator names are case-insensitive.

Logical Operators

Logical operators in precedence order: `NOT`, `AND`, `OR`.

Comparison Operators

Comparison operators: `=`, `>`, `>=`, `<`, `<=`, `<>` (not equal).

These operators can compare only values of comparable types. (Exact numeric values and approximate numerical values are comparable types.) Attempting to compare incomparable types yields `false`. If either value in a comparison evaluates to `NULL`, then the result is unknown (in SQL 3-valued logic).

Comparison of string values is restricted to `=` and `<>`. Two strings are equal if and only if they contain the same sequence of characters.

Comparison of boolean values is restricted to `=` and `<>`.

Arithmetic Operators

Arithmetic operators in precedence order:

- `+`, `-` (unary)
- `*`, `/` (multiplication and division)
- `+`, `-` (addition and subtraction)

Arithmetic operations obey numeric promotion rules of the Java programming language.

Between Operator

arithmetic-expr1 `[NOT] BETWEEN` *arithmetic-expr2* `AND` *arithmetic-expr3*

The `BETWEEN` comparison operator includes its endpoints. For example:

- `age BETWEEN 5 AND 9` is equivalent to `age >= 5 AND age <= 9`
- `age NOT BETWEEN 5 AND 9` is equivalent to `age < 5 OR age > 9`

String Set Membership

identifier `[NOT] IN (`*string-literal1*`,` *string-literal2*`, ...)`

The *identifier* must evaluate to either a string or `NULL`. If it is `NULL`, then the value of this expression is unknown. You can use a maximum of 32,767 string-literals in the string set.

Pattern Matching

identifier `[NOT] LIKE` *pattern-value* `[ESCAPE` *escape-character*`]`

The *identifier* must evaluate to a string.

The *pattern-value* is a string literal, in which some characters bear special meaning:

- _ (underscore) can match any single character.
- `%` (percent) can match any sequence of zero or more characters.
- *escape-character* preceding either of the special characters changes them into ordinary characters (which match only themselves).

Null Header or Property

identifier `IS NULL`

This comparison operator tests whether a message header is null, or a message property is absent.

identifier `IS NOT NULL`

This comparison operator tests whether a message header or message property is non-null.

White Space

White space is any of the characters space, horizontal tab, form feed, or line terminator—or any contiguous run of characters in this set.

Performance

In order to efficiently handle queue consumers with a selector when there is a large backlog of messages in the queue, message headers and properties are cached in the memory of the server for the queue. The caching begins for a given queue the first time a queue consumer with a selector is created.

This may result in an increase of the memory footprint of the server when such queue consumers are created. Both new incoming messages and messages already existing in the backlog are optimized through the server cache. If the server is restarted and a fault

tolerant consumer on the queue is restored, then all recovered messages in that queue are optimized.

Data Type Conversion

The following table summarizes legal data type conversions. The symbol X in the following table indicates that a value written into a message as the row type can be extracted as the column type. This table applies to all message values—including map pairs, headers and properties except as noted below.

	bool	byte	short	char	int	long	float	double	string	byte[]
bool	X								X	
byte		X	X		X	X			X	
short			X		X	X			X	
char				X					X	
int					X	X			X	
long						X			X	
float							X	X	X	
double								X	X	
string	X	X	X		X	X	X	X	X	
byte[]										X

Note:

- Message properties cannot have byte array values.
- Values written as strings can be extracted as a numeric or boolean type only when it is possible to parse the string as a number of that type.

Sending Messages Synchronously and Asynchronously

TIBCO Enterprise Message Service supports two modes of sending messages:

- **Synchronous** sending blocks the application thread until the entire send is complete.
- **Asynchronous** sending offloads the notification of the success or failure to another thread, thereby increasing performance in certain situations.

Each sending mode has certain benefits. The following sections describe the benefits of the different modes.

Sending Synchronously

Because synchronous sending does not have the overhead involved in asynchronous sending, it yields better performance in most cases. Synchronous sending is also the best choice when sending the following types of messages:

- **Non-Persistent Messages**

  When high performance is a concern, use synchronous sending for non-persistent or reliable messages. Although asynchronous sending of non-persistent messages is supported, it is generally not recommended.

- **Transactions**

  Typically, it makes sense for applications to use synchronous sending when using transactions. Sending messages within a transaction does not require a response from the server, so higher throughput can be obtained sending synchronously within a transaction.

  Synchronous sending simplifies a transaction; coordination of asynchronous send notifications and committing or rolling back a transaction introduces complexity to the application.

See Send Messages for details.

Sending Asynchronously

The message producer can send messages asynchronously by registering a *completion listener* to monitor message send success or failure.

Operating in a thread separate from that of the message producer, the completion listener manages the response to a successful or failed send, leaving the message producer free to perform other operations. See Create a Completion Listener for Asynchronous Sending for details.

Asynchronous sending can increase performance in certain circumstances. One of the best uses for asynchronous sending is when sending persistent messages. High level outgoing message throughput can be obtained when sending non-transacted persistent messages.

There are other considerations for the application programmer when sending messages asynchronously. These considerations are described below.

Concurrent Message Use

For simplicity, it is suggested that application programmers create a new message for every asynchronous send call. If concurrent message use is acceptable in an application, messages may be reused when sending asynchronously, but generally it is not recommended due to the complexity it may add.

Warning: During asynchronous sends, the application programmer should be very aware of concurrent message usage between the application and the thread handling completion listeners. The message passed to the completion listener is the same message passed to the MessageProducer send method, which means modification of that particular message is reflected in both the application thread and the thread invoking the completion listener.

For example, if a TextMessage is asynchronously sent with the text of `foo`, and then the same message object's text is subsequently set to `bar`, it is conceivable that when the completion listener is invoked the message will contain bar even though it contained `foo` at the time it was sent.

Memory Use

Application programmers should be aware that some additional memory is used by the EMS server when asynchronously sending. Memory use increases if the performance of completion listeners is slower than overall application send rates.

Fault Tolerant Failovers

Because send notifications are handled in a separate thread when messages are sent asynchronously, it is possible to receive messages out of order after a fault tolerant switch.

For example, consider an application that sends messages A, B, and C. Message A succeeds, Message B fails, but message C succeeds immediately after reconnect to the fault tolerant server. The application may not know message B failed before message C was sent. Message consumers could conceivably receive messages in the order of A, C, B; it is up to the application to appropriately handle this situation.

Receiving Messages Synchronously and Asynchronously

The EMS APIs allow for both synchronous or asynchronous message consumption. For synchronous consumption, the message consumer explicitly invokes a receive call on the topic or queue.

When synchronously receiving messages, the consumer remains blocked until a message arrives. See Receive Messages for details.

The consumer can receive messages asynchronously by registering a *message listener* to receive the messages. When a message arrives at the destination, the message listener delivers the message to the message consumer. The message consumer is free to do other operations between messages. See Create a Message Listener for Asynchronous Message Consumption for details.

Destinations

Destinations for messages can be either Topics or Queues. A destination can be created statically in the server configuration files, or dynamically by a client application.

Servers connected by routes exchange messages sent to temporary topics. As a result, temporary topics are ideal destinations for reply messages in request/reply interactions.

Destination Overview

The following table summarizes the differences between static, dynamic, and temporary destinations. The sections that follow provide more detail.

Aspect	Static	Dynamic	Temporary
Purpose	Static destinations let administrators configure EMS behavior at the enterprise level. Administrators define these administered objects, and client programs use them—relieving program developers and end users of the responsibility for correct configuration.	Dynamic destinations give client programs the flexibility to define destinations as needed for short-term use.	Temporary destinations are ideal for limited-scope uses, such as reply subjects.
Scope of Delivery	Static destinations support concurrent use. That is, several client processes (and in several threads within a process) can create local objects denoting the destination, and consume messages from it.	Dynamic destinations support concurrent use. That is, several client processes (and in several threads within a process) can create local objects denoting the destination, and consume messages from it.	Temporary destinations support only local use. That is, only the client connection that created a temporary destination can consume messages from it. However, servers connected by routes do exchange messages sent to temporary topics.
Creation	Administrators create static destinations using EMS server administration tools or administration API.	Client programs create dynamic destinations, if permitted by the server configuration.	Client programs create temporary destinations.
Lookup	Client programs lookup static destinations by name. Successful lookup returns a local object representation of the destination.	Not applicable.	Not applicable.
Duration	A static destination remains in the server until an administrator explicitly deletes it.	A dynamic destination remains in the server as long as at least one client actively uses it. The server automatically deletes it (at a convenient time) when all applicable conditions are true: • **Topic or Queue** All client programs that access the destination have disconnected. • **Topic** No offline durable subscribers exist for the topic. • **Queue** Queue, no messages are stored in the queue.	A temporary destination remains in the server either until the client that created it explicitly deletes it, or until the client disconnects from the server.

Destination Names

A destination name is a string divided into elements, each element separated by the dot character (.). The dot character allows you to create multi-part destination names that categorize destinations.

For example, you could have an accounting application that publishes messages on several destinations. The application could prefix all messages with ACCT, and each element of the name could specify a specific component of the application. ACCT.GEN_LEDGER.CASH, ACCT.GEN_LEDGER.RECEIVABLE, and ACCT.GEN_LEDGER.MISC could be subjects for the general ledger portion of the application.

Separating the subject name into elements allows applications to use wildcards for specifying more than one subject. See Wildcards for more information. The use of wildcards in destination names can also be used to define "parent" and "child" destination relationships, where the child destinations inherit the properties from its parents. See Inheritance of Properties.

Static Destinations

Configuration information for static destinations is stored in configuration files for the EMS server. Changes to the configuration information can be made in a variety of ways. To manage static destinations, you can edit the configuration files using a text editor, you can use the administration tool, or you can use the administration APIs.

Clients can obtain references to static destinations through a naming service such as JNDI or LDAP. See Creating and Modifying Destinations for more information about how clients use static destinations.

Dynamic Destinations

Dynamic destinations are created on-the-fly by the EMS server, as required by client applications. Dynamic destinations do not appear in the configuration files and exist as long as there are messages or consumers on the destination. A client cannot use JNDI to lookup dynamic queues and topics.

When you use the `show queues`or `show topics` command in the administration tool, you see dynamic topics and queues have an asterisk (*) in front of their name in the list of topics or queues. If a property of a queue or topic has an asterisk (*) character in front of its name, it means that the property was inherited from the parent queue or topic and cannot be changed.

See Dynamically Create Topics and Queues for details on topics and queues can be dynamically created by the EMS server.

Temporary Destinations

TIBCO Enterprise Message Service supports temporary destinations as defined in Jakarta Messaging specification and its API.

Servers connected by routes exchange messages sent to temporary topics. As a result, temporary topics are ideal destinations for reply messages in request/reply interactions.

For more information on temporary queues and topics, refer to the Jakarta Messaging documentation described in Third Party Documentation.

Destination Bridges

You can create server-based bridges between destinations of the same or different types to create a hybrid messaging model for your application. This allows all messages delivered to one destination to also be delivered to the bridged destination. You can bridge between different destination types, between the same destination type, or to more than one destination. For example, you can create a bridge between a topic and a queue or from a topic to another topic.

See Destination Bridges for more information about destination bridging.

Destination Name Syntax

TIBCO Enterprise Message Service places few restrictions on the syntax and interpretation of destination names. System designers and developers have the freedom to establish their own conventions when creating destination names. The best destination names reflect the structure of the data in the application itself.

Structure

A destination name is a string divided into elements, each element separated by the dot character (.). The dot character allows you to create multi-part destination names that categorize destinations.

Empty strings ("") are not permitted in destination names. Likewise, elements cannot incorporate the dot character by using an escape sequence.

Although they are not prohibited, we recommend that you do not use tabs, spaces, or any unprintable character in a destination name. You may, however, use wildcards. See Wildcards for more information.

Length

Destinations are limited to a total length of 249 characters. However, some of that length is reserved for internal use. The amount of space reserved for internal use varies according to the number of elements in the destination; destinations that include the maximum number of elements are limited to 196 characters.

A destination can have up to 64 elements. Each element cannot exceed 127 characters. Dot separators are not included in element length.

Destination Name Performance Considerations

When designing destination naming conventions, remember these performance considerations:

- Shorter destination names perform better than long destination names.
- Destinations with several short elements perform better than one long element.
- A set of destinations that differ early in their element lists perform better than subjects that differ only in the last element.

Special Characters in Destination Names

These characters have special meanings when used in destination names:

Char	Char Name	Special Meaning
.	Dot	Separates elements within a destination name.
>	Greater-than	Wildcard character, matches one or more trailing elements.
*	Asterisk	Wildcard character, matches one element.

For more information on wildcard matching, see Wildcards * and >.

Examples of Destination Names

These examples illustrate the syntax for destination names.

Examples of Destination Names
Valid Examples
`NEWS.LOCAL.POLITICS.CITY_COUNCIL`
`NEWS.NATIONAL.ARTS.MOVIES.REVIEWS`
`CHAT.MRKTG.NEW_PRODUCTS`
`CHAT.DEVELOPMENT.BIG_PROJECT.DESIGN`
`News.Sports.Baseball`
`finance`
`This.long.subject_name.is.valid.even.though.quite.uninformative`
Invalid Examples
`News..Natural_Disasters.Flood` (null element)
`WRONG.` (null element)

Examples of Destination Names
`.TRIPLE.WRONG..` (three null elements)
`News.Tennis.Stats.Roger\.Federer` (backslash in the element Roger will be included in the element name, and will not escape the dot)

Destination Properties

The following section contain a description of properties for topics and queues.

You can set the destination properties directly in the `topics.conf` or `queues.conf` file or by means of the `setprop topic` or `setprop queue` command in the EMS Administration Tool.

The following table lists the properties that can be assigned to topics and queues. The sections that follow describe each property.

Property	Topic	Queue
`exclusive`	No	Yes
`expiration`	Yes	Yes
`export`	Yes	No
`flowControl`	Yes	Yes
`global`	Yes	Yes
`import`	Yes	Yes
`maxbytes`	Yes	Yes
`maxmsgs`	Yes	Yes
`maxRedelivery`	No	Yes

Property	Topic	Queue
overflowPolicy	Yes	Yes
prefetch	Yes	Yes
redeliveryDelay	No	Yes
secure	Yes	Yes
sender_name	Yes	Yes
sender_name_enforced	Yes	Yes
store	Yes	Yes
trace	Yes	Yes

exclusive

The `exclusive` property is available for queues only (not for topics), and cannot be used with global queues.

When `exclusive` is set for a queue, the server sends all messages on that queue to one consumer. No other consumers can receive messages from the queue. Instead, these additional consumers act in a *standby* role; if the primary consumer fails, the server selects one of the standby consumers as the new primary, and begins delivering messages to it.

You can set exclusive using the form:

```
    exclusive
```

Non-Exclusive Queues & Round-Robin Delivery

By default, `exclusive` is not set for queues and the server distributes messages in a round-robin—one to each receiver that is ready. If any receivers are still ready to accept additional messages, the server distributes another round of messages—one to each receiver that is still ready. When none of the receivers are ready to receive more messages, the server waits until a queue receiver reports that it can accept a message.

This arrangement prevents a large buildup of messages at one receiver and balances the load of incoming messages across a set of queue receivers.

expiration

If an `expiration` property is set for a destination, the server honors the overridden expiration period and retains the message for the length of time specified by the expiration property.

However, the server overrides the `JMSExpiration` value set by the producer in the message header with the value 0 and therefore the consuming client does not expire the message.

You can set the `expiration` property for any queue and any topic using the form:

```
expiration=time[msec|sec|min|hour|day]
```

where *time* is the number of seconds. Zero is a special value that indicates messages to the destination never expire.

You can optionally include time units, such as msec, sec, min, hour or day to describe the *time* value as being in milliseconds, seconds, minutes, hours, or days, respectively. For example:

```
expiration=10min
```

means 10 minutes.

When a message expires it is either destroyed or, if the `JMS_TIBCO_PRESERVE_UNDELIVERED` property on the message is set to true, the message is placed on the undelivered queue so it can be handled by a special consumer. See Undelivered Message Queue for details.

All machines running EMS servers must be synchronized using NTP. If you use grid stores or FTL stores, all machines running ActiveSpaces and FTL processes must also be synchronized using NTP. Machines running EMS clients do not need to synchronized. For information about how non-synchronized client machines are handled, refer to the `clock_sync_interval` parameter.

export

The `export` property allows messages published by a client to a topic to be exported to the external systems with configured transports.

You can set export using the form:

```
export="list"
```

where *list* is one or more transport names, as specified by the [*transport_name*] ids in the `transports.conf` file. Multiple transport names in the list are separated by commas.

For example:

```
export="RV1,RV2"
```

You can configure transports for TIBCO FTL or Rendezvous reliable and certified messaging protocols. You can specify the name of one or more transports of the same type in the export property.

You must purchase, install, and configure the external system (for example, Rendezvous) before configuring topics with the export property. Also, you must configure the communication parameters to the external system by creating a named transport in the `transports.conf` file.

For complete details about external message services, see:

- Interoperation with TIBCO FTL
- Interoperation with TIBCO Rendezvous

flowControl

The `flowControl` property specifies the target maximum size the server can use to store pending messages for the destination. Should the number of messages exceed the maximum, the server will slow down the producers to the rate required by the message consumers.

This is useful when message producers send messages much more quickly than message consumers can consume them. Unlike the behavior established by the `overflowPolicy` property, `flowControl` never discards messages or generates errors back to producer.

You can set `flowControl` using the form:

```
flowControl=size[KB|MB|GB]
```

where *size* is the maximum number of bytes of storage for pending messages of the destination. If you specify the `flowControl` property without a value, the target maximum is set to 256KB.

You can optionally include a KB, MB or GB after the number to specify kilobytes, megabytes, or gigabytes, respectively. For example:

```
flowControl=1000KB
```

Means 1000 kilobytes.

The `flow_control` parameter in `tibemsd.conf` file must be set to enabled before the value in this property is enforced by the server. See Flow Control for more information about flow control.

global

Messages destined for a topic or queue with the `global` property set are routed to the other servers that are participating in routing with this server.

You can set `global` using the form:

```
global
```

For further information on routing between servers, see Routes.

import

The `import` property allows messages published by an external system to be received by a EMS destination (a topic or a queue), as long as the transport to the external system is configured.

You can set `import` using the form:

```
import="list"
```

where *list* is one or more transport names, as specified by the [*NAME*] ids in the `transports.conf` file. Multiple transport names in the list are separated by commas. For example:

```
import="RV1,RV2"
```

You can configure transports for TIBCO FTL or Rendezvous reliable and certified messaging protocols. You can specify the name of one or more transports of the same type in the import property.

You must purchase, install, and configure the external system (for example, Rendezvous) before configuring topics with the import property. Also, you must configure the communication parameters to the external system by creating a named transport in the `transports.conf` file.

For complete details about external message services, see:

- Interoperation with TIBCO FTL
- Interoperation with TIBCO Rendezvous

maxbytes

Topics and queues can specify the `maxbytes` property in the form:

```
maxbytes=value[KB|MB|GB]
```

where *value* is the number of bytes. For example:

```
maxbytes=1000
```

Means 1000 bytes.

You can optionally include a KB, MB or GB after the number to specify kilobytes, megabytes, or gigabytes, respectively. For example:

```
maxbytes=1000KB
```

Means 1000 kilobytes.

For queues, `maxbytes` defines the maximum size (in bytes) that the queue can store, summed over all messages in the queue. Should this limit be exceeded, messages will be rejected by the server and the message producer send calls will return an error (see also `overflowPolicy`). For example, if a receiver is off-line for a long time, then the queue size could reach this limit, which would prevent further memory allocation for additional messages.

If `maxbytes` is zero, or is not set, the server does not limit the memory allocation for the queue.

You can set both `maxmsgs` and `maxbytes` properties on the same queue. Exceeding either limit causes the server to reject new messages until consumers reduce the queue size to below these limits.

> **Warning:** If the `maxbytes` limit is not set on a destination, the server still checks to see if that destination's memory footprint is growing beyond a threshold. If so, a warning is logged. For more details, see `large_destination_memory` and `large_destination_count`.

For topics, `maxbytes` limits the maximum size (in bytes) that the topic can store for delivery to each durable or non-durable online subscriber on that topic. That is, the limit applies separately to each subscriber on the topic. For example, if a durable subscriber is off-line for a long time, pending messages accumulate until they exceed `maxbytes`; when the subscriber consumes messages (freeing storage) the topic can accept additional messages for the subscriber. For a non-durable subscriber, `maxbytes` limits the number of pending messages that can accumulate while the subscriber is online.

> **Warning:** Under certain conditions, because of the pipelined nature of message processing or the requirements of transactional messaging, the `maxbytes` limit can be slightly exceeded. You may see message totals that are marginally larger than the set limit.

When a destination inherits different values of this property from several parent destinations, it inherits the smallest value.

> **Note:** You can further protect against consumers that receive messages without acknowledging them using the parameter `disconnect_non_acking_consumers`.

maxmsgs

Topics and queues can specify the `maxmsgs` property in the form:

```
maxmsgs=value
```

where *value* defines the maximum number of messages that can be waiting in a queue. When adding a message would exceed this limit, the server does not accept the message into storage, and the message producer's send call returns an error (but see also `overflowPolicy`).

If `maxmsgs` is zero, or is not set, the server does not limit the number of messages in the queue.

> **Warning:** If the `maxmsgs` limit is not set on a destination, the server still checks to see if that destination's memory footprint is growing beyond a threshold. If so, a warning is logged. For more details, see `large_destination_memory` and `large_destination_count`.

You can set both `maxmsgs` and `maxbytes` properties on the same queue. Exceeding either limit causes the server to reject new messages until consumers reduce the queue size to below these limits.

> **Warning:** Under certain conditions, because of the pipelined nature of message processing or the requirements of transactional messaging, the `maxmsgs` limit can be slightly exceeded. You may see message totals that are marginally larger than the set limit.

> **Note:** You can further protect against consumers that receive messages without acknowledging them using the parameter `disconnect_non_acking_consumers`.

maxRedelivery

The `maxRedelivery` property specifies the number of attempts the server should make to deliver a message sent to a queue.

Set maxRedelivery using the form:

```
maxRedelivery=count
```

where *count* is an integer between 2 and 255 that specifies the maximum number of times a message can be delivered to receivers. A value of zero disables `maxRedelivery`, so there is no maximum.

Once the server has attempted to deliver the message the specified number of times, the message is either destroyed or, if the `JMS_TIBCO_PRESERVE_UNDELIVERED` property on the message is set to true, the message is placed on the undelivered queue so it can be handled by a special consumer. See Undelivered Message Queue for details.

For messages that have been redelivered, the `JMSRedelivered` header property is set to true and the `JMSXDeliveryCount` property is set to the number of times the message has been delivered to the queue. If the server restarts, the current number of delivery attempts in the `JMSXDeliveryCount` property is not retained.

Note: In the event of an abrupt exit by the client, the `maxRedelivery` count can be mistakenly incremented. An abrupt exit prevents the client from communicating with the server; for example, when the client exits without closing the connection or when the client application crashes. If a client application exits abruptly, the EMS server counts all messages sent to the client as delivered, even if they were not presented to the application.

overflowPolicy

Topics and queues can specify the `overflowPolicy` property to change the effect of exceeding the message capacity established by either `maxbytes` or `maxmsgs`.

Set the `overflowPolicy` using the form:

```
overflowPolicy=default|discardOld|rejectIncoming
```

If `overflowPolicy` is not set, then the policy is default.

The effect of `overflowPolicy` on the `maxbytes` and `maxmsgs` behaviors differs depending on whether you set it on a topic or a queue, so the impact of each `overflowPolicy` value is described separately for topics and queues.

If wildcards are used in the .conf file the inheritance of the `overflowPolicy` policy from multiple parents works as follows:

- If a child destination has a non-default `overflowPolicy` policy set, then that policy is used and it does not inherit any conflicting policy from a parent.
- If a parent has `OVERFLOW_REJECT_INCOMING` set, then it is inherited by the child destination over any other policy.
- If no parent has `OVERFLOW_REJECT_INCOMING` set and a parent has `OVERFLOW_DISCARD_OLD` policy set, then that policy is inherited by the child destination.
- If no parent has the `OVERFLOW_REJECT_INCOMING` or `OVERFLOW_DISCARD_OLD` set, then the default policy is used by the child destination.

default

For topics, `default` specifies that messages are sent to each subscriber in turn. If the `maxbytes` or `maxmsgs` setting has been reached for a subscriber, that subscriber does not receive the message. No error is returned to the message producer.

For queues, `default` specifies that new messages are rejected by the server and an error is returned to the producer if the established `maxbytes` or `maxmsgs` value has been exceeded.

Note: When delivery delay is enabled for a topic, the behavior of `overflowPolicy=default` mimics that of a queue. That is, when `maxbytes` or `maxmsgs` has been reached, new messages are rejected by the server and an error is returned to the producer.

discardOld

For topics, `discardOld` specifies that, if any of the subscribers have an outstanding number of undelivered messages on the server that are over the message limit, the oldest messages are discarded before they are delivered to the subscriber.

The `discardOld` setting impacts subscribers individually. For example, you might have three subscribers to a topic, but only one subscriber exceeds the message limit. In this case, only the oldest messages for the one subscriber are discarded, while the other two subscribers continue to receive all of their messages.

When messages for a topic or queue exceed the `maxbytes` or `maxmsgs` value, the oldest messages are silently discarded. No error is returned to the producer.

rejectIncoming

For topics, `rejectIncoming` specifies that, if *any* of the subscribers have an outstanding number of undelivered messages on the server that are over the message limit, all new messages are rejected and an error is returned to the producer.

For queues, `rejectIncoming` specifies that, if messages on the queue have exceeded the `maxbytes` or `maxmsgs` value, all new messages are rejected and an error is returned to the producer. (This is the same as the `default` behavior.)

Examples

To discard messages on `myQueue` when the number of queued messages exceeds 1000, enter:

```
setprop queue myQueue maxmsgs=1000,overflowPolicy=discardOld
```

To reject all new messages published to `myTopic` when the memory used by undelivered messages for any of the topic subscribers exceeds 100KB, enter:

```
setprop topic myTopic maxbytes=100KB,overflowPolicy=rejectIncoming
```

prefetch

The message consumer portion of a client and the server cooperate to regulate fetching according to the `prefetch` property. The `prefetch` property applies to both topics and queues.

You can set `prefetch` using the form:

```
    prefetch=value
```

where *value* is one of the values in prefetch Values.

prefetch Values

The following table lists values used with the `prefetch` property.

Value	Description
2 or more	The message consumer automatically fetches messages from the server. The message consumer never fetches more than the number of messages specified by *value*. See Automatic Fetch Enabled for details.
1	The message consumer automatically fetches messages from the server—initiating fetch only when it does not currently hold a message.
none	Disables automatic fetch. That is, the message consumer initiates fetch only when the client calls `receive`—either an explicit synchronous call, or an implicit call (in an asynchronous consumer). This value cannot be used with topics or global queues. See Automatic Fetch Disabled for details.
0	The destination inherits the `prefetch` value from a parent destination with a matching name. If it has no parent, or no destination in the parent chain sets a value for `prefetch`, then the default value is 5 queues and 64 for topics. When a destination does not set any value for `prefetch`, then the default value is 0 (zero; that is, inherit the `prefetch` value). See Inheritance for details.

Note: If both `prefetch` and `maxRedelivery` are set to a non-zero value, then there is a potential to lose prefetched messages if one of the messages exceeds the `maxRedelivery` limit. For example, `prefetch=5` and `maxRedelivery=4`. The first message is redelivered 4 times, hits the `maxRedelivery` limit and is sent to the undelivered queue (as expected). However, the other 4 pre-fetched messages are also sent to the undelivered queue and are not processed by the receiving application. The work around is to set `prefetch=none`, but this can have performance implications on large volume interfaces.

Background

Delivering messages from the server destination to a message consumer involves two independent phases—fetch and accept.

- The *fetch* phase is a two-step interaction between a message consumer and the server.
    - The message consumer initiates the fetch phase by signaling to the server that it is ready for more messages.
    - The server responds by transferring one or more messages to the client, which stores them in the message consumer.
- In the *accept* phase, client code takes a message from the message consumer.

The receive call embraces both of these phases. It initiates fetch when needed and it accepts a message from the message consumer.

To reduce waiting time for client programs, the message consumer can *prefetch* messages—that is, fetch a batch of messages from the server, and hold them for client code to accept, one by one.

acl.conf

This file defines all permissions on topics and queues for all users and groups.

The format of the file is:

```
TOPIC=topic USER=user PERM=permissions
TOPIC=topic GROUP=group PERM=permissions
QUEUE=queue USER=user PERM=permissions
QUEUE=queue GROUP=group PERM=permissions
ADMIN USER=user PERM=permissions
ADMIN GROUP=group PERM=permissions
```

Parameter Name	Description
TOPIC	Name of the topic to which you wish to add permissions.

Parameter Name	Description
QUEUE	Name of the queue to which you wish to add permissions.
ADMIN	Specifies that you wish to add administrator permissions.
USER	Name of the user to whom you wish to add permissions.
GROUP	Name of the group to which you wish to add permissions. The designation `all` specifies a predefined group that contains all users.
PERM	Permissions to add. The permissions which can be assigned to queues are `send`, `receive` and `browse`. The permissions which can be assigned to topics are `publish`, `subscribe` and `durable` and `use_durable`. The designation `all` specifies all possible permissions. For information about these permissions, refer to When Permissions Are Checked and Inheritance of Permissions. Administration permissions are granted to users to perform administration activities. See Administrator Permissions for more information about administration permissions.

Example

```
ADMIN USER=sys-admins PERM=all
TOPIC=foo USER=user2 PERM=publish,subscribe
TOPIC=foo GROUP=group1 PERM=subscribe
```

Automatic Fetch Enabled

To enable automatic fetch, set `prefetch` to a positive integer. Automatic fetch ensures that if a message is available, then it is waiting when client code is ready to accept one. It can improve performance by decreasing or eliminating client idle time while the server transfers a message.

However, when a queue consumer prefetches a group of messages, the server does not deliver them to other queue consumers (unless the first queue consumer's connection to the server is broken).

Note: A positive `prefetch` must be configured in order to use receiveNoWait function calls.

Automatic Fetch Disabled

To disable automatic fetch, set `prefetch=none`.

Even when `prefetch=none`, a queue consumer can still hold a message. For example, a `receive` call initiates fetch, but its timeout elapses before the server finishes transferring the message. This situation leaves a fetched message waiting in the message consumer. A second `receive` call does not fetch another message; instead, it accepts the message that is already waiting. A third `receive` call initiates another fetch.

Notice that a waiting message still belongs to the queue consumer, and the server does not deliver it to another queue consumer (unless the first queue consumer's connection to the server is broken). To prevent messages from waiting in this state for long periods of time, code programs either to call `receive` with no timeout, or to call it (with timeout) repeatedly and shorten the interval between calls.

Note: Automatic fetch cannot be disabled for global queues or for topics.

Inheritance

When a destination inherits the `prefetch` property from parent destination with matching names, these behaviors are possible:

- When all parent destinations set the value none, then the child destination inherits the value none.
- When any parent destination sets a non-zero numeric value, then the child destination inherits the *largest* value from among the entire parent chain.
- When none of the parent destinations sets any non-zero numeric value, then the child destination uses the default value (which is 5).

redeliveryDelay

When `redeliveryDelay` is set, the EMS server waits the specified interval before returning an unacknowledged message to the queue.

When a previously delivered message did not receive a successful acknowledgment, the EMS server waits the specified redelivery delay before making the message available again in the queue. This is most likely to occur in the event of a transaction rollback, session or message recovery, session or connection close, or the abrupt exit of a client application. However, note that the delay time is not exact, and in most situations will exceed the specified `redeliveryDelay`.

Note: The redelivery delay is not available for routed queues.

The value can be specified in seconds, minutes, or hours. The value may be in the range of 15 seconds and 8 hours.

You can set `redeliveryDelay` using the form:

```
redeliveryDelay=time[sec|min|hour]
```

where *time* is the number of seconds. Zero is a special value that indicates no redelivery delay.

You can optionally include time units, such as sec, min, or hour describe the *time* value as being in seconds, minutes, or hours, respectively. For example:

```
redeliveryDelay=30min
```

specifies a redelivery delay of 30 minutes.

During the delay interval, messages are placed in the `$sys.redelivery.delay` queue. This queue can be browsed, but it cannot be consumed from or purged. However, purging the queue from which the delayed message came, or removing the message using its message ID, immediately removes that message from `$sys.redelivery.delay`.

Note: While a message is on the `$sys.redelivery.delay queue`, it is not on the queue from which it came and so it is not included in statistical message counts. This includes `maxmsgs`, `maxbytes`, `flowControl`, and so on.

secure

When the `secure` property is enabled for a destination, it instructs the server to check user permissions whenever a user attempts to perform an operation on that destination.

You can set secure using the form:

```
secure
```

If the `secure` property is not set for a destination, the server does not check permissions for that destination and any authenticated user can perform any operation on that topic or queue.

Note: The `secure` property is independent of TLS—it controls basic authentication and permission verification within the server. To configure secure communication between clients and server, see TLS Protocol.

The server `authorization` property acts as a master switch for checking permissions. That is, the server checks user permissions on secure destinations only when the `authorization` property is enabled. To enforce permissions, you must *both* enable the authorization configuration parameter, and set the `secure` property on each affected destination.

See Authentication and Permissions for more information on permissions and the secure property.

sender_name

The `sender_name` property specifies that the server may include the sender's user name for messages sent to this destination.

You can set `sender_name` using the form:

```
sender_name
```

When the `sender_name` property is enabled, the server takes the user name supplied by the message producer when the connection is established and places that user name into the `JMS_TIBCO_SENDER` property in the message.

The message producer can override this behavior by specifying a property on a message. If a message producer sets the `JMS_TIBCO_DISABLE_SENDER` property to true for a message,

the server overrides the sender_name property and does not add the sender name to the message.

If authentication for the server is turned off, the server places whatever user name the message producer supplied when the message producer created a connection to the server. If authentication for the server is enabled, the server authenticates the user name supplied by the connection and the user name placed in the message property will be an authenticated user. If TLS is used, the TLS connection protocol guarantees the client is authenticated using the client's digital certificate.

sender_name_enforced

The sender_name_enforced property specifies that messages sent to this destination *must* include the sender's user name. The server retrieves the user name of the message producer using the same procedure described in the sender_name property above.

However, unlike, the sender_name property, there is no way for message producers to override this property.

You can set sender_name_enforced using the form:

```
sender_name_enforced
```

If the sender_name property is also set on the destination, this property overrides the sender_name property.

Note: In some business situations, clients may not be willing to disclose the user name of their message producers.
If this is the case, these clients may wish to avoid sending messages to destinations that have the sender_name or sender_name_enforced properties enabled.

In these situations, the operator of the EMS server should develop a policy for disclosing a list of destinations that have these properties enabled. This will allow clients to avoid sending messages to destinations that would cause their message producer usernames to be exposed.

store

The `store` property determines where messages sent to this destination are stored. Messages may be stored in a file, in a TIBCO ActiveSpaces data grid, or in a TIBCO FTL server cluster.

See Store Messages in Multiple Stores for more information on using and configuring multiple stores.

Warning: When using the `setprop` or `addprop` commands to change the store settings for a topic or queue, note that existing messages are not migrated to the new store. As a result, stopping the EMS server and deleting the original store may result in data loss, if a destination still had messages in the original store.

Set the store property using this form:

```
store=name
```

where *name* is the name of a store, as defined in the `stores.conf` file.

For example, this will send all messages sent to the destination `giants.games` to the store named `baseball`; messages sent to all other destinations will be stored in `everythingelse`:

```
> store=everythingelse
giants.games store=baseball
```

Only one store is allowed for each destination. If there is a conflict, for example if overlapping wildcards cause a topic to inherit multiple store properties, the topic creation will fail.

Note: This parameter cannot be used without first enabling this feature in the `tibemsd.conf` file. The `stores.conf` file must also exist, but can be left empty if the only store names that are associated with destinations are the default stores. These rules apply when using a JSON configuration file as well.

See Store Messages in Multiple Stores for more information.

trace

The `trace` property specifies that tracing should be enabled for this destination.

You can set `trace` using the form:

```
trace = [body]
```

Specifying `trace` (without `=body`), generates trace messages that include the message sequence, message ID, and message size. Specifying `trace=body` generates trace messages that include the message body. See Message Tracing for more information about message tracing.

Temporary Destination Properties

Temporary destinations, both topics and queues, support the following properties:

- `maxbytes`
- `maxmsgs`
- `overflowPolicy`

Temporary destinations tend to be short-lived objects by nature. Applications have no control over destination names for temporary topics and queues. For these reasons, you cannot directly set the above supported properties on temporary destinations.

However, EMS defines a special temporary destination wildcard that can be used to assign properties and values to temporary topics and queues by way of inheritance.

The temporary destination wildcard is defined as `$TMP$.>`, and can be used for both topics and queues. All properties set on topics using the wildcard are inherited by all temporary topics. Similarly, all properties set on queues using the wildcard are inherited by all temporary queues.

Although the same wildcard is used for both destination types, property values assigned using the wildcard are not shared between topics and queues. That is, you can assign one `overflowPolicy` to all temporary topics, and a different `overflowPolicy` to all temporary queues.

Properties can also be set on the `$TMP$.>` temporary destination wildcard through a variety of ways:

- Using the following `tibemsadmin` commands:
  - `create topic $TMP$.>` [*properties*]
  - `create queue $TMP$.>` [*properties*]
  - `addprop topic $TMP$.>` [*properties*]
  - `addprop queue $TMP$.>` [*properties*]
  - `setprop topic $TMP$.>` [*properties*]
  - `setprop queue $TMP$.>` [*properties*]
- In the `topics.conf` and `queues.conf` configuration files.
- In the JSON configuration file.

Topics

Properties set on the `$TMP$.>` topic are immediately and directly inherited by all existing temporary topics and all temporary topics created in the future.

Queues

Properties set on the `$TMP$.>` queue are immediately and directly inherited by all existing temporary queues and all temporary queues created in the future.

Usage Notes

The temporary destination wildcard `$TMP$.>` can *only* be used to set properties on temporary topics or queues through inheritance.

- `$TMP$.>` cannot be used to send or receive messages.
- `$TMP$.>` cannot be used as the source or target of a destination bridge.
- You cannot create a durable subscription on the temporary topic wildcard `$TMP$.>`.
- You cannot use `$TMP$.>` to import or export messages from TIBCO FTL or Rendezvous.
- `$TMP$.>` never inherits any properties from other destination wildcards. For example, `$TMP$.>` does not inherit from the wildcard >.

Creating and Modifying Destinations

Destinations are typically "static" administered objects that can be stored in a JNDI or LDAP server. Administered objects can also be stored in the EMS server and looked up using the EMS implementation of JNDI.

This section describes how to use the EMS Administration Tool to create and modify destination objects in EMS. For more information, see EMS Administration Tool.

You create a queue using the `create queue` command and a topic using the `create topic` command. For example, to create a new queue named `myQueue`, enter:

```
create queue myQueue
```

To create a topic named `myTopic`, enter:

```
create topic myTopic
```

The queue and topic data stored on the EMS server is located in the `queues.conf` and `topics.conf` files, respectively. You can use the `show queues` and `show topics` commands to list all of the queues and topics on your EMS server and the show queue and show topic commands to show the configuration details of specific queues and topics.

A queue or topic may include optional properties that define the specific characteristics of the destination. These properties are described in Destination Properties and they can be specified when creating the queue or topic or modified for an existing queue or topic using the `addprop queue`, `addprop topic`, `setprop queue`, `setprop topic`, `removeprop queue`, and `removeprop topic` commands.

For example, to discard messages on *myQueue* when the number of queued messages exceeds 1000, you can set an `overflowPolicy` by entering:

```
addprop queue myQueue maxmsgs=1000,overflowPolicy=discardOld
```

To change the `overflowPolicy` from `discardOld` to `rejectIncoming`, enter:

```
addprop queue myQueue overflowPolicy=rejectIncoming
```

The `setprop queue` and `setprop topic` commands remove properties that are not explicitly set by the command. For example, to change `maxmsgs` to 100 and to *remove* the `overflowPolicy` parameter, enter:

```
setprop queue myQueue maxmsgs=100
```

Creating Secure Destinations

By default, all authenticated EMS users have permissions to perform any action on any topic or queue.

You can set the `secure` property on a topic or queue and then use the `grant topic` or `grant queue` command to specify which users and/or groups are allowed to perform which actions on the destination.

The `secure` property requires that you enable the `authorization` property on the EMS server.

For example, to create a secure queue, named `myQueue`, to which only users "joe" and "eric" can send messages and "sally" can receive messages, in the EMS Administration Tool, enter:

```
set server authorization=enabled
create queue myQueue secure
grant queue myQueue joe send
grant queue myQueue eric send
grant queue myQueue sally receive
```

See Authentication and Permissions for more information.

Wildcards

You can use wildcards when specifying statically created destinations in `queues.conf` and `topics.conf`.

The use of wildcards in destination names can be used to define "parent" and "child" destination relationships, where the child destinations inherit the properties and permissions from its parents. You must first understand wildcards to understand the inheritance rules described in Inheritance.

Wildcards * and >

To understand the rules for inheritance of properties, it is important to understand the use of the two wildcards, * and >.

- The wildcard > by itself matches any destination name.
- When > is mixed with text, it matches one or more trailing elements. For example:

  ```
  foo.>
  ```

  Matches `foo.bar`, `foo.boo`, `foo.boo.bar`, and `foo.bar.boo`.
- The wildcard * means that any token can be in the place of *. For example:

  ```
  foo.*
  ```

  Matches `foo.bar` and `foo.boo`, but not `foo.bar.boo`.

  ```
  foo.*.bar
  ```

  Matches `foo.boo.bar`, but not `foo.bar`.

Overlapping Wildcards and Disjoint Properties

Some destination properties are disjoint, and the server allows that property to be set only once for each destination. If an existing destination includes a value for a disjoint property and you attempt to assign a different value, the action will fail.

Overlapping wildcard destinations can cause conflicts with disjoint properties. For example, consider the following configuration of the store property:

```
topic.sample.>         store=$sys.failsafe
topic.sample.quotes.* store=$sys.nonfailsafe
```

The topic `topic.sample.quotes.tibx` would be assigned both stores, `$sys.failsafe` and `$sys.nonfailsafe`. Therefore, the wildcard topics `topic.sample.>` and `topic.sample.quotes.*` cannot coexist. Their creation would fail.

EMS currently has only one disjoint property: `store`.

Wildcards in Topics

TIBCO Enterprise Message Service enables you to use wildcards in topic names in some situations.

- You can subscribe to wildcard topics.

  If you subscribe to a topic containing a wildcard, you will receive any message published to a matching topic. For example, if you subscribe to `foo.*` you will receive messages published to a topic named `foo.bar`.

  You can subscribe to a wildcard topic (for example `foo.*`), whether or not there is a matching topic in the configuration file (for example, `foo.*`, `foo.>`, or `foo.bar`). However, if there is no matching topic name in the configuration file, no messages will be published on that topic.

- You cannot publish to wildcard topics.
- If `foo.bar` is not in the configuration file, then you can publish to `foo.bar` if `foo.*` or `foo.>` exists in the configuration file.
- On routed topic messages, subscribers must specify a topic that is a direct subset (or equal) of the configured global topic. For more information, see Wildcards.

Wildcards in Queues

TIBCO Enterprise Message Service enables you to use wildcards in queue names in some situations. You can neither send to nor receive from wildcard queue names. However, you can use wildcard queue names in the configuration files.

For example, if the queue configuration file includes a line:

```
foo.*
```

then users can dynamically create queues `foo.bar`, `foo.bob`, and so forth, but not `foo.bar.bob`.

Wildcards and Dynamically Created Destinations

The EMS server may dynamically create destinations on behalf of its clients. The use of wildcards in the `.conf` files can be used to control the allowable names of dynamically

created destinations.

The same basic wildcard rules apply to dynamically created destinations as described above for static destinations.

Examples

- If the `queues.conf` file contains:

  ```
  >
  ```

  The EMS server can dynamically create a queue with any name.

- If the `topics.conf` file contains only:

  ```
  foo.>
  ```

  The EMS server can dynamically create topics with names like `foo.bar`, `foo.boo`, `foo.boo.bar`, and `foo.bar.boo`.

- If the `queues.conf` file contains only:

  ```
  foo.*
  ```

  The EMS server can dynamically create queues with names like `foo.bar` and `foo.boo`, but not `foo.bar.boo`.

- If the `topics.conf` file contains only:

  ```
  foo.*.bar
  ```

  The EMS server can dynamically create topics with names like `foo.boo.bar`, but not `foo.bar`.

Inheritance

The following sections describe the inheritance of properties and permissions.

The Wildcards, Destination Properties, and Authentication and Permissions sections provide useful information in this context

Inheritance of Properties

All destination properties are inheritable for both topics and queues. This means that a property set for a "wildcarded" destination is inherited by all destinations with matching names.

For example, if you have the following in your `topics.conf` file:

```
foo.* secure

foo.bar

foo.bob
```

Topics `foo.bar` and `foo.bob` are `secure` topics because they inherit secure from their parent, `foo.*`. If your EMS server were to dynamically create a `foo.new` topic, it too would have the secure property.

The properties inherited from a parent are *in addition* to the properties defined for the child destination.

For example, if you have the following in your `topics.conf` file:

```
foo.* secure
foo.bar sender_name
```

Then `foo.bar` has both the `secure` and `sender_name` properties.

In the above example, there is no way to make topic `foo.*` secure without making `foo.bar` secure. In other words, EMS does not offer the ability to remove inherited properties. However, for properties that are assigned values, you can override the value established in a parent.

For example, if you have the following in your `queues.conf` file:

```
foo.* maxbytes=200
foo.bar maxbytes=2000
```

The `foo.bar` queue has a `maxbytes` value of 2000.

When there are multiple ancestors for a destination, the destination inherits the properties from all of the parents. For example:

```
> sender_name
foo.* secure
foo.bar trace
```

The `foo.bar` topic has the `sender_name`, `secure` and `trace` properties.

When there are multiple parents for a destination that contain conflicting property values, the destination inherits the smallest value. For example:

```
> maxbytes=2000
foo.* maxbytes=200
foo.bar
```

The `foo.bar` topic has a `maxbytes` value of 200.

Property inheritance is powerful, but can be complex to understand and administer. You must plan before assigning properties to topics and queues. Using wildcards to assign properties must be used carefully. For example, if you enter the following line in the `topics.conf` file:

```
> store=mystore
```

you make every topic store messages, regardless of additional entries. This might require a great deal of memory for storage and greatly decrease the system performance.

Inheritance of Permissions

Inheritance of permissions is similar to inheritance of properties. If the parent has a permission, then the child inherits that permission.

For example, if Bob belongs to GroupA, and GroupA has publish permission on a topic, then Bob has `publish` permission on that topic.

Permissions for a single user are the union of the permissions set for that user, and of all permissions set for every group in which the user is a member. These permission sets are additive. Permissions have positive boolean inheritance. Once a permission right has been granted through inheritance, it can not be removed.

All rules for wildcards apply to inheritance of permissions. For example, if a user has permission to publish on topic `foo.*`, the user also has permission to publish on `foo.bar` and `foo.new`.

For more information on wildcards, refer to Wildcards. For more information on permissions, refer to User Permissions.

Destination Bridges

Some applications require the same message to be sent to more than one destination, possibly of different types.

For example, an application may send messages to a queue for distributed load balancing. That same application, however, may also need the messages to be published to several monitoring applications. Another example is an application that publishes messages to several topics. All messages however, must also be sent to a database for backup and for data mining. A queue is used to collect all messages and send them to the database.

An application can process messages so that they are sent multiple times to the required destinations. However, such processing requires significant coding effort in the application. EMS provides a server-based solution to this problem. You can create bridges between destinations so that messages sent to one destination are also delivered to all bridged destinations.

Bridges are created between one destination and one or more other destinations of the same or of different types. That is, you can create a bridge from a topic to a queue or from a queue to a topic. You can also create a bridge between one destination and multiple destinations. For example, you can create a bridge from topic `a.b` to queue `q.b` and topic `a.c`.

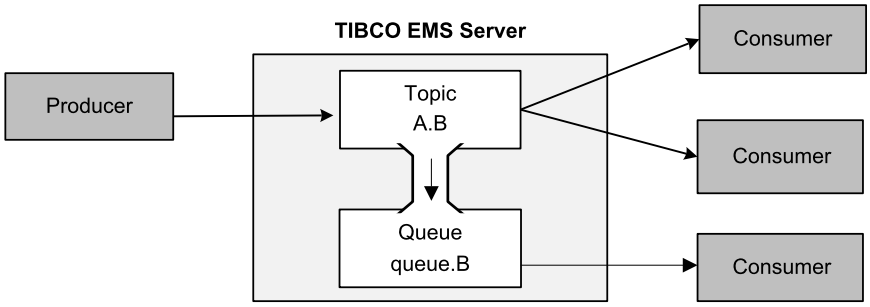
When specifying a bridge, you can specify a particular destination name, or you can use wildcards. For example, if you specify a bridge on topic `foo.*` to queue `foo.queue`, messages delivered to any topic matching `foo.*` are sent to `foo.queue`.

> **Note:** Because global topics are routed between servers and global queues are limited to their neighbors, in most cases the best practice is to send messages to a topic and then bridge the topic to a queue.

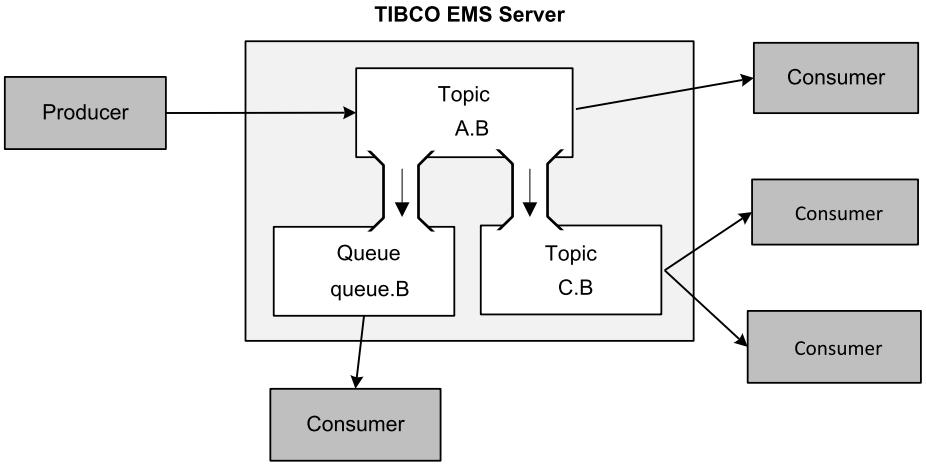
When multiple bridges exist, using wildcards to specify a destination name may result in a message being delivered twice. For example, if the queues `Q.1` and `Q.>` are both bridged to `QX.1`, the server will deliver two copies of sent messages to `QX.1`.

The following figures illustrate example bridging scenarios.

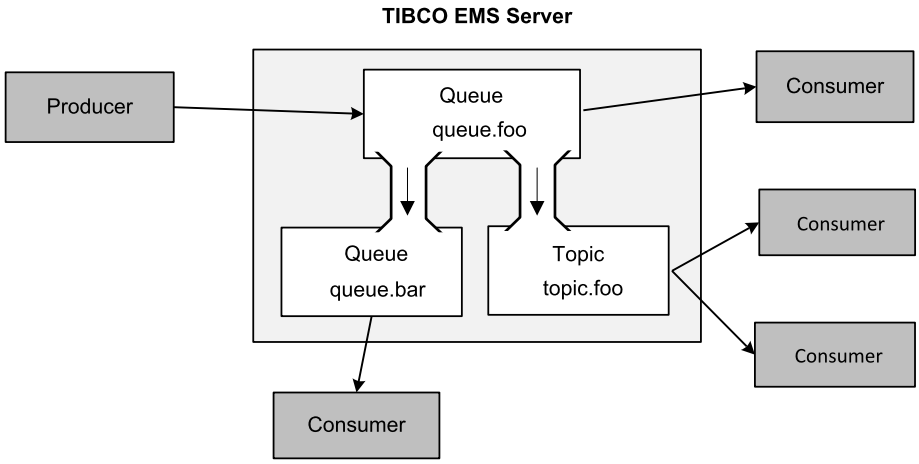
Bridging a topic to a queue:



Bridging a topic to multiple destinations:



Bridging a queue to multiple destinations:



Note: When a bridge exists between two queues, the message is delivered to both queues. The queues operate independently; if the message is retrieved from one queue, that has no effect on the status of the message in the second queue.

Bridges are not transitive. That is, messages sent to a destination with a bridge are only delivered to the specified bridged destinations and are not delivered across multiple bridges. For example, topic `A.B` has a bridge to queue `Q.B`. Queue `Q.B` has a bridge to topic `B.C`. Messages delivered to `A.B` are also delivered to `Q.B`, but not to `B.C`.

The bridge copies the source message to the target destination, which assigns the copied message a new message identifier. Note that additional storage may be required, depending on the target destination store parameters.

Create a Bridge

Bridges are configured using the `bridges.conf` configuration file.

You specify a bridge using the following syntax:

```
[destinationType:destinationName]
    destinationType=destinationToBridgeTo selector="msg-selector"
```

where *destinationType* is the type of the destination (either `topic` or `queue`), *destinationName* is the name of the destination from which you wish to create a bridge, *destinationToBridgeTo* is the name of the destination you wish to create a bridge to, and `selector="`*msg-selector*`"` is an optional message selector to specify the subset of messages the destination should receive.

Each *destinationName* can specify wildcards, and therefore any destination matching the pattern will have the specified bridge. Each *destinationName* can specify more than one *destinationToBridgeTo*.

For example, the bridges illustrated in the images Bridging a topic to a queue and Bridging a topic to multiple destinations would be specified as the following in `bridges.conf`:

```
[topic:A.B]
   queue=queue.B
   topic=C.B
```

Specifying a message selector on a bridged destination is described in the following section.

Note: Deleting the source destination or a target destination of a bridge is prohibited. The server prevents you from deleting the source destination, however it does not prevent you from deleting a target destination. Regardless, prior to deleting a destination that is the source or target of a bridge, you must first remove the bridge.

Select the Messages to Bridge

By default, all messages sent to a destination with a bridge are sent to all bridged destinations. This can cause unnecessary network traffic if each bridged destination is only interested in a subset of the messages sent to the original destination. You can optionally specify a message selector for each bridge to determine which messages are sent over that bridge.

Message selectors for bridged destinations are specified as the selector property on the bridge. The following is an example of specifying a selector on the bridges defined in the previous section:

```
[topic:A.B]
   queue=queue.B
   topic=C.B selector="urgency in('medium', 'high')"
```

For detailed information about message selector syntax, see the documentation for the Message class in the relevant EMS API reference document.

Access Control and Bridges

Message producers must have access to a destination to send messages to that destination. However, a bridge automatically has permission to send to its target destination. Special configuration is not required.

Transactions

When a message producer sends a message within a transaction, all messages sent across a bridge are part of the transaction. Therefore, if the transaction succeeds, all messages are delivered to all bridged destinations. If the transaction fails, no consumers for bridged destinations receive the messages.

If a message cannot be delivered to a bridged destination because the message producer does not have the correct permissions for the bridged destination, the transaction cannot complete, and therefore fails and is rolled back.

Flow Control

In some situations, message producers may send messages more rapidly than message consumers can receive them. The pending messages for a destination are stored by the server until they can be delivered, and the server can potentially exhaust its storage capacity if the message consumers do not receive messages quickly enough.

To avoid this, EMS allows you to control the flow of messages to a destination. Each destination can specify a target maximum size for storing pending messages. When the target is reached, EMS blocks message producers when new messages are sent. This effectively slows down message producers until the message consumers can receive the pending messages.

Enable Flow Control

The `flow_control` parameter in `tibemsd.conf` enables and disables flow control globally for the EMS server.

When `flow_control` is `disabled` (the default setting), the server does not enforce any flow control on destinations. When `flow_control` is `enabled`, the server enforces any flow control settings specified for each destination. See Configuration Files for more information about working with configuration parameters.

When you wish to control the flow of messages on a destination, set the `flowControl` property on that destination. The `flowControl` property specifies the target maximum size of stored pending messages for the destination. The size specified is in bytes, unless you specify the units for the size. You can specify KB, MB, or GB for the units. For example, `flowControl = 60MB` specifies the target maximum storage for pending messages for a destination is 60 Megabytes.

Enforce Flow Control

The value specified for the `flowControl` property on a destination is a target maximum for pending message storage. When flow control is enabled, the server may use slightly more or less storage before enforcing flow control, depending upon message size, number of message producers, and other factors.

Setting the `flowControl` property on a destination but specifying no value causes the server to use a default value of 256KB.

When the storage for pending messages is near the specified limit, the server blocks all new calls to send a message from message producers. The calls do not return until the storage has decreased below the specified limit, or the `flowControl` limit is increased. Once message consumers have received messages and the pending message storage goes below the specified limit, the server allows the send message calls to return to the caller and the message producers can continue processing.

Flow Control in the Absence of Consumers

The server enforces flow control on destinations regardless of the presence of consumers.

Note: Prior to release 8.4, if there was no message consumer for a destination, the server would not enforce flow control for the destination. That is, if a queue had no started receiver, the server did not enforce flow control for that queue. Also, if a topic had inactive durable subscriptions or no current subscriber, the server did not enforce flow control for that topic. For topics, if flow control was set on a specific topic (for example, `foo.bar`), then flow control was enforced as long as there were subscribers to that topic or any parent topic (for example, if there were subscribers to `foo.*`).

This behavior can be restored by setting the `flow_control_only_with_active_consumer` property but note that this property and the corresponding behavior are deprecated and will be removed in a future release.

Routes and Flow Control

For global topics where messages are routed between servers, flow control can be specified for a topic on either the server where messages are produced or the server where messages are received. Flow control is not relevant for queue messages that are routed to another server.

If the `flowControl` property is set on the topic on the server receiving the messages, when the pending message size limit is reached, messages are not forwarded by way of the route until the topic subscriber receives enough messages to lower the pending message size below the specified limit.

If the `flowControl` property is set on the topic on the server sending the messages, the server may block any topic `publishers` when sending new messages if messages cannot be sent quickly enough by way of the route. This could be due to network latency between the routed servers or it could be because flow control on the other server is preventing new messages from being sent.

Destination Bridges and Flow Control

Flow control can be specified on bridged destinations.

If you wish the flow of messages sent over the bridge to slow down when receivers on the bridged-to destination cannot process the messages quickly enough, you must set the `flowControl` property on both destinations on either side of the bridge.

Flow Control, Threads and Deadlock

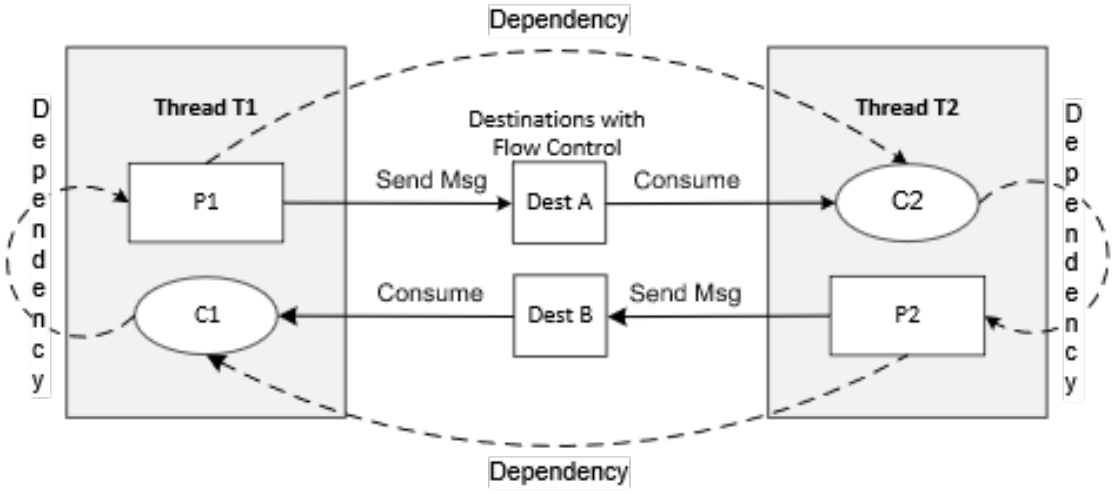
When using flow control, you must be careful to avoid potential deadlock. When flow control is in effect for a destination, producers to that destination can block waiting for flow control signals from the destination's consumers. If any of those consumers are within the same thread of program control, a potential for deadlock exists.

Namely, the producer will not unblock until the destination contains fewer messages, and the consumer in the blocked thread cannot reduce the number of messages.

The simplest case to detect is when producer and consumer are in the same session (sessions are limited to a single thread). But more complex cases can arise. Deadlock can even occur across several threads, or even programs on different hosts, if dependencies link them. For example, consider the situation in the following image that illustrates a flow control deadlock across two threads:

- Producer P1 in thread T1 has a consumer C2 in thread T2.
- Producer P2 in T2 has a consumer C1 in T1.
- Because of the circular dependency, deadlock can occur if either producer blocks its thread waiting for flow control signals.

The dependency analysis is analogous to mutex deadlock. You must analyze your programs and distributed systems in a similar way to avoid potential deadlock.



Delivery Delay

The delivery delay feature allows the message producer to specify the earliest time at which a message should be delivered to consumers. This is done by using the `setDeliveryDelay()` method to set the minimum length of time that must elapse after a message is sent before the EMS server may deliver the message to a consumer.

Whenever a message is sent to destination *dest* with a non-zero delivery delay for the first time, the server dynamically creates a queue named `$sys.delayed.q.`*dest* when *dest* is a queue, or `$sys.delayed.t.`*dest* when *dest* is a topic.

`$sys.delayed` queues support browsing and purging but do not support other permissions such as receive or send. They inherit destination limits, security, and storage selection properties from *dest*. However, note that a `$sys.delayed.t` queue created for a topic that has the `secure` property cannot be browsed.

Note that the `$sys.delayed` queue corresponding to a destination takes any `maxmsgs` property setting from the destination. That is, if *dest* has property `maxmsgs` set to *X*, its `$sys.delayed` queue also has `maxmsgs` set to X. This doubles the number of messages that can potentially be held for *dest* in the server.

If the `maxmsgs` limit has been reached and the destination has the property `overflowPolicy=rejectIncoming`, when the delivery delay expires for a message one of two things can happen. If the message has the `JMS_TIBCO_PRESERVE_UNDELIVERED` set to `true`, it is put on the `$sys.undelivered` queue. Otherwise, the message is discarded.

Note that, when delivery delay is enabled for a topic, the behavior of `overflowPolicy=default` mimics that of a queue. That is, when `maxbytes` or `maxmsgs` has been reached, new messages are rejected by the server and an error is returned to the producer.

Getting Started

This following topics provide a quick introduction to setting up a simple EMS configuration and running some sample client applications to publish and subscribe users to a topic.

About the Sample Clients

The EMS sample clients were designed to allow you to run TIBCO Enterprise Message Service with minimum start-up time and coding.

The *EMS_HOME*/samples directory contains several subdirectories. The c, cs, and java subdirectories contain the C, .NET and Java sample clients.

In this chapter, you will compile and run the Java sample clients. For information on how to run the C and .NET sample clients, see the readme files in their respective directories.

The *EMS_HOME*/samples/java directory contains the following sets of files:

- Sample clients for TIBCO Enterprise Message Service implementation.
- The JNDI subdirectory contains sample clients that use the JNDI lookup technique.
- The tibrv subdirectory contains sample clients that demonstrate the interoperation of TIBCO Enterprise Message Service with TIBCO Rendezvous applications.
- The admin subdirectory contains samples that illustrate the use of the administration API.

The *EMS_HOME*/samples/c directory contains sample clients.

On Windows platforms only, the *EMS_HOME*\samples\cs directory contains two sets of files:

- Sample clients for TIBCO Enterprise Message Service implementation.
- The admin subdirectory contains samples that illustrate the use of the administration API.

In this chapter, you will use some of the sample clients in the *EMS_HOME*/samples/java directory. For information on compiling and running the other sample clients, see the Readme files in their respective folders.

Compiling the Sample Java Clients

To compile and run the sample Java clients you need to execute "setup" script, which is located in the *EMS_HOME*/samples/java directory.

On Windows systems, the setup file is setup.bat.

On UNIX systems, the setup file is setup.sh.

Procedure

1. Make sure you have JDK 1.8 or greater installed and that you've added the bin directory to your PATH variable.
2. Open a command line or console window, and navigate to the EMS_HOME/samples/java directory.
3. Open the correct setup script file and verify that the TIBEMS_ROOT environment variable identifies the correct pathname to your EMS_HOME directory. For example, on a Windows system this might look like:

   ```
   > set TIBEMS_ROOT=C:\tibco\ems\10.3
   ```

4. Enter setup to set the environment and classpath:

   ```
   > setup
   ```

5. Compile the samples:

   ```
   > javac -d . *.java
   ```

   This compiles all the samples in the directory, except for those samples in the JNDI, tibrv, and admin subdirectories.

   If the files compile successfully, the class files will appear in the EMS_HOME/samples/java directory. If they do not compile correctly, an error message appears.

Creating Users with the EMS Administration Tool

In this example, you will create topics and users using the EMS Administration Tool. You must first start the EMS server before starting the EMS administration tool.

Follow these steps to start the EMS server and to use the administration tool to create two new users.

Note: All of the parameters you set using the administration tool in this chapter can also be set by editing the configuration files described in Configuration Files. You can also programmatically set parameters using the C, .NET, or Java APIs. Parameters set programmatically by a client are only set for the length of the session.

Procedure

1. **Start the EMS server**

   Start the EMS server as described in Starting and Stopping the EMS Server.

2. **Start the Administration Tool and Connect to the EMS Server**

   a. Start the EMS administration tool as described in Starting the EMS Administration Tool.

   b. After starting the administration tool, connect it to the EMS server.

   To connect the EMS administration tool to the EMS server, execute one of the following commands:

- If you are using TIBCO Enterprise Message Service on a single computer, type **connect** in the command line of the Administration tool:

  ```
  > connect
  ```

  You will be prompted for a login name. If this is the first time you've used the EMS administration tool, follow the procedure described in When You First Start tibemsadmin.

  Once you have logged in, the screen will display:

  connected to `tcp://localhost:7222`

```
tcp://localhost:7222>
```

- If you are using TIBCO Enterprise Message Service in a network, use the connect server command as follows:

  > `connect` [*server URL*] [*user-name*] [*password*]

  For more information on this command, see `connect`.

  For further information on the administration tool, see Starting the EMS Administration Tool and Command Listing.

3. **Create Users**

   Once you have connected the administration tool to the server, use the `create user` command to create two users.

   In the administration tool, enter:

   ```
   tcp://localhost:7222> create user user1
   tcp://localhost:7222> create user user2
   ```

   The tool will display messages confirming that `user1` and `user2` have been `created`.

   You have now created two users. You can confirm this with the `show users` command:

   ```
   tcp://localhost:7222> show users
     User Name         Description
      user1
      user2
   ```

   For more information on the `create user` command, refer to `create user`.

Point-to-Point Messaging Example

This section demonstrates how to use point-to-point messaging, as described in Point-to-Point.

Creating a Queue

In the point-to-point messaging model, client send messages to and receive messages from a queue.

To create a new queue in the administration tool, use the `create queue` command to create a new queue named `myQueue`:

```
tcp://localhost:7222> create queue myQueue
```

For more information on the `create queue` command, refer to `create queue`. For more information on the commit command, see `commit` and `autocommit`.

Starting the Sender and Receiver Clients

Procedure

1. Open two command line windows and in each window navigate to the `EMS_HOME/samples/java` folder.
2. In each command line window, enter setup to set the environment and classpath:

   ```
   > setup
   ```

3. In the first command line window, execute the `tibjmsMsgProducer` application to direct user1 to place some messages to the `myQueue` queue:

   ```
   > java tibjmsMsgProducer -queue myQueue -user user1 Hello User2
   ```

4. In the second command line window, execute the `tibjmsMsgConsumer` client to direct `user2` to read the messages from the message queue:

   ```
   > java tibjmsMsgConsumer -queue myQueue -user user2
   ```

   The messages placed on the queue are displayed in the receiver's window.

   Note: Messages placed on a queue by the sender are persistent until acknowledged by the receiver, so you can start the sender and receiver clients in any order.

Publish and Subscribe Messaging Example

In this section, you will execute a message producer client and two message consumer clients that demonstrate the publish/subscribe messaging model described in Publish and Subscribe. This example is not intended to be comprehensive or representative of a robust application.

To execute the client samples, you must give them commands from within the sample directory that contains the compiled samples. For this exercise, open three separate command line windows and navigate to the *EMS_HOME*/samples/java directory in each window.

For more information on the samples, refer to the readme file within the sample directory. For more information on compiling the samples, refer to Compiling the Sample Java Clients.

Creating a Topic

In the publish/subscribe model, you publish and subscribe to topics.

To create a new topic in the administration tool, use the create topic command to create a new topic named myTopic:

```
tcp://localhost:7222> create topic myTopic
```

For more information on the create topic command, refer to create topic. For more information on the commit command, see commit and autocommit.

Starting the Subscriber Clients

You start the subscribers first because they enable you to observe the messages being received when you start the publisher.

Procedure

To start user1 as a subscriber:

1. In the first command line window, navigate to *EMS_HOME*/samples/java.

2. Enter setup to set the environment and classpath:

```
> setup
```

3. Execute the `tibjmsMsgConsumer` client to assign `user1` as a subscriber to the `myTopic` topic:

```
> java tibjmsMsgConsumer -topic myTopic -user user1
```

   The screen will display a message showing that `user1` is subscribed to `myTopic`.

To start `user2` as a subscriber:

4. In the second command line window, navigate to the *EMS_HOME*`/samples/java` folder.
5. Enter setup to set the environment and classpath:

```
> setup
```

6. Execute the `tibjmsMsgConsumer` application to assign `user2` as a subscriber to the `myTopic` topic:

```
> java tibjmsMsgConsumer -topic myTopic -user user2
```

   The screen will display a message showing that `user2` is subscribed to `myTopic`.

> **Note:** The command windows do not return to the prompt when the subscribers are running.

Starting the Publisher Client and Sending Messages

Setting up the publisher is very similar to setting up the subscriber. However, while the subscriber requires the name of the topic and the user, the publisher also requires messages.

To start the publisher:

Procedure

1. In the third command line window, navigate to the *EMS_HOME*`/samples/java` folder.

2. Enter `setup` to set the environment and classpath:

```
> setup
```

3. Execute the `tibjmsMsgProducer` client to direct `user1` to publish some messages to the `myTopic` topic:

```
> java tibjmsMsgProducer -topic myTopic -user user1 hello user2
```

where 'hello' and 'user2' are separate messages.

Note: In this example, `user1` is both a publisher and subscriber.

Result

The command line window will display a message stating that both messages have been published:

```
Publishing on topic 'myTopic'
Published message: hello
Published message: user2
```

After the messages are published, the command window for the publisher returns to the prompt for further message publishing.

Note: Note that if you attempt to use the form:

```
java tibjmsMsgProducer -topic myTopic -user user1
```

without adding the messages, you will see an error message, reminding you that you must have at least one message text.

The first and second command line windows containing the subscribers will show that each subscriber received the two messages:

```
Subscribing to destination: myTopic

Received message: TextMessage={ Header={ JMSMessageID={ID:EMS-
SERVER.16C5B5C81B3CB4:1}
JMSDestination={Topic[myTopic]} JMSReplyTo={null} JMSDeliveryMode={PERSISTENT}
```

```
JMSRedelivered={false} JMSCorrelationID={null} JMSType={null} JMSTimestamp={Thu
Mar 07 18:18:01 CST 2019} JMSDeliveryTime={Thu Mar 07 18:18:01 CST 2019}
JMSExpiration={0} JMSPriority={4} } Properties={ JMSXDeliveryCount={Integer:1}
} Text={hello} }
Received message: TextMessage={ Header={ JMSMessageID={ID:EMS-
SERVER.16C5B5C81B3CB4:2}
JMSDestination={Topic[myTopic]} JMSReplyTo={null} JMSDeliveryMode={PERSISTENT}
JMSRedelivered={false} JMSCorrelationID={null} JMSType={null} JMSTimestamp={Thu
Mar 07 18:18:01 CST 2019} JMSDeliveryTime={Thu Mar 07 18:18:01 CST 2019}
JMSExpiration={0}
JMSPriority={4} } Properties={ JMSXDeliveryCount={Integer:1} } Text={user2} }
```

Creating a Secure Topic

In this example, you make `myTopic` into a secure topic and grant `user1` permission to publish to the `myTopic` and `user2` permission to subscribe to `myTopic`.

Adding the secure Property to the Topic

When the secure property is added to a topic, only users who have been assigned a certain permission can perform the actions allowed by that permission. For example, only users with publish permission on the topic can publish, while other users cannot publish.

If the secure property is not added to a topic, all authenticated users have all permissions (publish, subscribe, create durable subscribers) on that topic.

For more information on the secure property, see the section about `secure`. For more information on topic permissions, see Authentication and Permissions,.

To enable server authorization and add the `secure` property to a topic, do the following steps:

Procedure

1. In each subscriber window, enter Control-C to stop each subscriber.
2. In the administration tool, use the `set server` command to enable the `authorization` property:

   ```
   tcp://localhost:7222> set server authorization=enabled
   ```

The `authorization` property enables checking of permissions set on destinations.

3. Enter the following command to add the `secure` property to a topic named `myTopic`:

```
tcp://localhost:7222> addprop topic myTopic secure
```

For more information on the `set server` command, refer to `set server`. For more information on the `addprop topic` command, refer to `addprop topic`.

Granting Topic Access Permissions to Users

To see how permissions affect the ability to publish and receive messages, grant publish permission to `user1` and subscribe permission to the `user2`.

Use the `grant topic` command to grant permissions to users on the topic `myTopic`.

In the administration tool, enter:

```
tcp://localhost:7222> grant topic myTopic user1 publish
tcp://localhost:7222> grant topic myTopic user2 subscribe
```

For more information on the `grant topic` command, refer to `grant topic`.

Starting the Subscriber and Publisher Clients

Start the subscribers, as described in Starting the Subscriber Clients. Note that you cannot start `user1` as a subscriber because `user1` has permission to publish, but not to subscribe. As a result, you receive an exception message including the statement:

```
Operation not permitted.
```

`User2` should start as a subscriber in the same manner as before.

You can now start `user1` as the publisher and send messages to `user2`, as described in Starting the Publisher Client and Sending Messages.

Creating a Durable Subscriber

As described in Publish and Subscribe, subscribers, by default, only receive messages when they are active. If messages are published when the subscriber is not available, the subscriber does not receive those messages. You can create durable subscriptions, where subscriptions are stored on the server and subscribers can receive messages even if it was inactive when the message was originally delivered.

In this example, you create a durable subscriber that stores messages published to topic `myTopic` on the EMS server.

To start `user2` as a durable subscriber:

Procedure

1. In the a command line window, navigate to the *EMS_HOME*`/samples/java` folder.
2. Enter setup to set the environment and classpath:

```
> setup
```

3. Execute the `tibjmsDurable` application to assign `user2` as a durable subscriber to the `myTopic` topic:

```
> java tibjmsDurable -topic myTopic -user user2
```

4. In the administration tool, use the show durables command to confirm that `user2` is a durable subscriber to `myTopic`:

```
tcp://localhost:7222> show durables
  Topic Name  Durable     User  Msgs  Size
* myTopic     subscriber  user2    0   0.0 Kb
```

5. In the subscriber window, enter Ctrl+C to stop the subscriber.
6. In another command line window, execute the `tibjmsMsgProducer` client, as described in Starting the Publisher Client and Sending Messages:

```
> java tibjmsMsgProducer -topic myTopic -user user1 hello user2
```

7. Restart the subscriber:

```
> java tibjmsDurable -topic myTopic -user user2
```

The stored messages are displayed in the subscriber window.

8. Enter Ctrl+C to stop the subscriber and then unsubscribe the durable subscription:

```
> java tibjmsDurable -unsubscribe
```

The subscriber is no longer durable and any additional messages published to the `myTopic` topic are lost.

Running the EMS Server

To use TIBCO Enterprise Message Service with your applications, the TIBCO Enterprise Message Service Server must be running.

Starting and Stopping the EMS Server

The server and the clients work together to implement TIBCO Enterprise Message Service. The server implements all types of message persistence and no messages are stored on the client side. The following topics describe how to start and stop the EMS Server.

Types of Configuration Files

You can choose to have the TIBCO Enterprise Message Service server store configuration settings in a single JSON-based configuration file. This file holds the entire configuration of the server without the need for sub-files. Furthermore, a single JSON configuration file holds the configuration settings for a pair of fault-tolerant servers. JSON-based configuration files use the .json file extension.

The JSON configuration standard was introduced in an earlier version of EMS. Prior to that release, the configuration of the server could only be stored in a set of text-based configuration files with names ending in .conf. The main configuration file name defaults to tibemsd.conf and a set of sub-files such as queues.conf hold information on specific types of configuration items. These configuration files are described in the present book.

A server can be started either with a set of .conf files or with a single .json file. However, servers using stores of type `as` or `ftl` can only be configured with a JSON-based configuration. In this particular case, the configuration is hosted in either TIBCO ActiveSpaces or TIBCO FTL.

You can convert a text-based server configuration to a single tibemsd.json file using the tibemsconf2json utility, which is described in the Conversion of Server Configuration Files to JSON section.

Starting the EMS Server Using a Sample Configuration

To start the EMS server from the command line using sample configuration files, navigate to `EMS_HOME/samples/config` and perform the following steps:

Procedure

1. Create a local directory called `datastore` (for example, `/opt/tibco/ems/samples/config/datastore`)
2. Execute the command:

   ```
   tibemsd -config tibemsd.conf
   ```

Starting the EMS Server Using JSON Configuration

This section and the next describe the steps to start the TIBCO Enterprise Message Service server using the JSON configuration file when using file-based stores or grid stores. For details on starting the server with FTL stores, see Configuring and Deploying FTL Stores.

Procedure

1. From the command line, navigate to *EMS_HOME*`/bin`.
2. Enter the following command and option:

   ```
   tibemsd -config json-file-path
   ```

   where *json-file-path* is the path to your JSON configuration file. For example:

   ```
   tibemsd -config /tibemsconfig/tibemsd.json
   ```

   If the server is unable to find the JSON configuration file at *`json-file-path`*, it automatically creates a default JSON configuration file at that location.

   When started using the JSON configuration, the server silently ignores any unknown parameters. For example, no configuration errors are thrown if the `tibemsd.json` file contains an obsolete parameter.

Note: For information on converting `.conf` configuration files to JSON configuration files, see Conversion of Server Configuration Files to JSON.

Starting Fault Tolerant Server Pairs

With a JSON-based configuration, fault tolerant pairs share a single JSON configuration file. Primary and secondary server roles are determined when the servers are started.

Start the primary EMS server as usual. Start the secondary server using the -secondary flag. For example, where the JSON configuration file is `tibemsd.json`:

- To start the primary server:

  ```
  tibemsd -config tibemsd.json
  ```

- To start the secondary server:

  ```
  tibemsd -config tibemsd.json -secondary
  ```

Starting the EMS Server Using Options

To start the EMS server from the command line using options:

Procedure

1. Navigate to the `samples` subdirectory.

   Sample EMS server configuration files are located in `EMS_HOME/samples/config`. For more information, see 'Installing TIBCO Enterprise Message Service' in *TIBCO Enterprise Message Service Installation*.

   The EMS server dynamically loads the OpenSSL and compression shared libraries. If the `tibemsd` executable is executed from the `samples` directory, it automatically locates these libraries. If the server is moved elsewhere, the shared library directory must be moved as well.

2. Start the `tibemsd`

   Type `tibemsd` [*options*]

   where *options* are described in `tibemsd Options`. The command options to `tibemsd` are similar to the parameters you specify in `tibemsd.conf`, and the command options

override any value specified in the parameters. See tibemsd.conf for more information about configuration parameters.

tibemsd Options

The `tibemsd` options override any value specified in the parameters.

Note: A number of these options are unsupported when using FTL stores. See the Unsupported tibemsd Options section for details.

Option	Description
`-config` *config file name*	*config file name* is the name of the main configuration file for `tibemsd` server. Default is `tibemsd.conf`. For example, to start an EMS server using the default JSON configuration file, use: `tibemsd -config tibemsd.json` If using grid stores, this option has a different purpose. See Server Command-Line Options for Grid Stores for more information.
`-trace` *items*	Specifies the trace items. These items are not stored in the configuration file. The value has the same format as the value of `log_trace` parameter specified with `set server` command of the administration tool; see Trace Messages for the Server.
`-secondary`	Specifies the secondary server in a fault tolerant pair. This option is only valid for EMS servers started using JSON config.
`-module_path`	Specifies a directory or directories that contain external shared library files such as those of

Option	Description
	FTL, ActiveSpaces, and Rendezvous. This parameter is only relevant when using FTL or Rendezvous transports or grid stores.
`-ssl_password` *string*	Private key password.
`-ssl_trace`	Print the certificates loaded by the server and do more detailed tracing of TLS-related situation.
`-ssl_debug_trace`	Turns on tracing of TLS connections.
`-ft_active` *active_url*	URL of the active server. If this server can connect to the active server, it will act as a standby server. If this server cannot connect to the active server, it will become the active server.
`-ft_heartbeat` *seconds*	Heartbeat signal for the active server, in seconds. Default is 3.
`-ft_activation` *seconds*	Activation interval (maximum length of time between heartbeat signals) which indicates that active server has failed. Set in seconds: default is 10. This interval should be set to at least twice the heartbeat interval.
`-forceStart`	Causes the server to delete corrupted messages in the stores, allowing the server to start even if it encounters errors. Note that using this option causes data loss, and it is important to backup store data before using `-forceStart`. See Error Recovery Policy for more information.
Grid Store Options	Options required for starting the EMS server

Option	Description
	with grid stores. See Server Command-Line Options for Grid Stores for more information.

Stopping the EMS Server

You can stop the EMS server by using the `shutdown` command from the EMS Administration Tool. For more information, see `shutdown`.

Running the EMS Server as a Windows Service

Some situations require the EMS server to start automatically. You can satisfy this requirement by registering it with the Windows service manager. The `emsntsrg` utility facilitates registry.

emsntsrg

The `emsntsrg` utility registers or unregisters the EMS server as a Windows service.

Syntax

```
emsntsrg /i [/a]|[/d] service_name emsntsct_directory service_directory [arguments] [suffix]
emsntsrg /r [service_name] [suffix]
```

Remarks

Some situations require the EMS server processes to start automatically. You can satisfy this requirement by registering these with the Windows service manager. This utility facilitates registry.

Restrictions

You must have administrator privileges to change the Windows registry.

Location

Locate this utility program as an executable file in the EMS `bin` directory.

Parameter	Description
`/i`	Insert a new service in the registry (that is, register a new service).
`/a`	Automatically start the new service. Optional with `/i`. You can use either `/a` or `/d` but not both.
`/d`	Automatically start the new service with a delay. Optional with `/i`. You can use either `/a` or `/d` but not both.
`/?`	Display usage.
service_name	Insert or remove a service with this base name. When inserting a service, this parameter is required, and must be `tibemsd`. When removing a service, this parameter is optional. However, if it is present, it must be `tibemsd`.
emsntsct_directory	Use this directory pathname to specify the location of the `emsntsct.exe` executable. The `emsntsrg` utility registers the `emsntsct.exe` program as a windows service. The `emsntsct.exe` program then invokes the associated `tibemsd`. By default, `emsntsct.exe` is located in *`EMS_HOME`*`\bin`. This parameter is only required when installing a service.
service_directory	Use this directory pathname to locate the service executable,

Parameter	Description
	`tibemsd`. Required.
arguments	Supply command line arguments. Optional with `/i`. Enclose the entire arguments string in double quote characters.
suffix	When registering more than one instance of a service, you can use this suffix to distinguish between them in the Windows services applet. Optional.
`/r`	Remove a service from the registry.

Register

To register `tibemsd` as a Windows service, run the utility with this command line:

```
emsntsrg /i [/a]|[/d] tibemsd emsntsct_directory tibemsd_directory
[arguments] [suffix]
```

- Example 1

  This simple example registers one `tibemsd` service:

  ```
  emsntsrg /i tibemsd C:\tibco\ems\10.3\bin C:\tibco\ems\10.3\bin
  ```

- Example 2

  This example registers a service with command line arguments:

  ```
  emsntsrg /i tibemsd C:\tibco\ems\10.3\bin C:\tibco\ems\10.3\bin "-
  trace DEFAULT"
  ```

- Example 3

  This pair of example commands registers two `tibemsd` services with different configuration files. In this example, the numerical suffix and the configuration directory both reflect the port number that the service uses.

```
emsntsrg /i tibemsd C:\tibco\ems\10.3\bin C:\tibco\ems\10.3\bin
"-config C:\tibco\ems\10.3\7222\tibemsd.conf" 7222
emsntsrg /i tibemsd C:\tibco\ems\10.3\bin C:\tibco\ems\10.3\bin
"-config C:\tibco\ems\10.3\7223\tibemsd.conf" 7223
```

Notice these aspects of this example:

- When installing `tibemsd`, if you supply a `-config` argument, the service process finds the directory containing the main configuration file (`tibemsd.conf`), and creates all secondary configuration files in that directory. In this example, each service uses a different configuration directory.
- When you register several EMS services, you must avoid configuration conflicts. For example, two instances of `tibemsd` cannot listen on the same port.

Remove

To unregister a service, run the utility with this command line:

```
emsntsrg /r [service_name] [suffix]
```

Both parameters are optional. If the *service_name* is present, it must be `tibemsd`. To supply the *suffix* parameter, you must also supply the *service_name*. When both parameters are absent, the utility removes the services named `tibemsd`.

Command Summary

To view a command line summary, run the utility with this command line:

```
emsntsrg
```

Windows Services Applet

The Windows services applet displays the name of each registered service. For EMS services, it also displays this additional information:

- The suffix (if you supply one)
- The process ID (PID)—when the service is running

Error Recovery Policy

During startup the EMS server can encounter a number of errors while it recovers information from the stores.

Potential errors include:

- Low-level file errors. For example, corrupted disk records.
- Low-level object-specific errors. For example, a record that is missing an expected field.
- Inter-object errors. For example, a session record with no corresponding connection record.

When the EMS server encounters one of these errors during startup, the recovery policy is:

- By default, the server exits startup completely when a corrupt disk record error is detected. Because the state can not be safely restored, the server can not proceed with the rest of the recovery. You can then examine your configuration settings for errors. If necessary, you can then copy the store and configuration files for examination by TIBCO Support.
- You can direct the server to delete bad records by including the `-forceStart` command line option. This prevents corruption of the server runtime state.
- The server exits if it runs out of memory during startup.

It is important to backup all stores before restarting the server with the `-forceStart` option, because data will be lost when the problematic records are deleted. To back up file-based stores, you can simply create a copy of the store files. For grid stores and FTL stores, you will need to back up the associated ActiveSpaces or FTL deployment. Refer to the *TIBCO ActiveSpaces Administration* and *TIBCO FTL Administration* product guides for instructions on creating backups for these products.

Keep in mind that different type of records are stored in the stores. The most obvious are the persistent Jakarta Messaging Messages that your applications have sent. However, other internal records are also stored. If a consumer record used to persist durable subscriber state information were to be corrupted and later deleted with the `-forceStart` option, all Jakarta Messaging messages that were persisted (and valid in the sense that they were not corrupted) would also be lost because the durable subscription itself would not be recovered.

When running in this mode, the server still reports any errors found during the recovery, but problematic records are deleted and the recovery proceeds. This mode may report

more issues than are reported without the `-forceStart` option, because without that flag the server stops with the very first error.

> **Warning:** We strongly recommended that you make a backup of all the stores before restarting the server with the `-forceStart` option. The backup is useful when doing a postmortem analysis to find out what records were deleted with the `-forceStart` option.

Security Considerations

> **Warning:** This section highlights information relevant to secure deployment. We recommend that all administrators read this section.

Secure Environment

To ensure secure deployment, EMS administration must meet certain criteria.

These criteria include:

- **Correct Installation**: EMS is correctly installed and configured.
- **Physical Controls**: The computers where EMS is installed are located in areas where physical entry is controlled to prevent unauthorized access. Only authorized administrators have access, and they cooperate in a benign environment.
- **Domain Control**: The operating system, file system and network protocols ensure domain separation for EMS, to prevent unauthorized access to the server, its configuration files, LDAP servers, etc.
- **Benign Environment**: Only authorized administrators have physical access or domain access, and those administrators cooperate in a benign environment.

Destination Security

Three interacting factors affect the security of destinations (that is, topics and queues). In a secure deployment, you must properly configure all three of these items.

- The server's `authorization` parameter (see Authorization Parameter, below)
- The secure property of individual destinations (see `secure`)
- The ACL permissions that apply to individual destinations (see Authentication and Permissions)

Authorization Parameter

The `authorization` parameter of the server acts as a master switch for checking permissions for connection requests and operations on secure destinations.

The default value of this parameter is `disabled`—the server does not check any permissions, and allows all operations. For secure deployment, you must enable this parameter.

Admin Password

For ease in installation and initial testing, the default setting for the admin password is no password at all. Until you set an actual password, the user admin can connect without a password. Once the administrator password has been set, the server always requires it.

To configure a secure deployment, the administrator must change the admin password immediately after installation; see Assigning a Password to the Administrator.

Connection Security

When `authorization` is enabled, the server requires a name and password before users can connect. Only authenticated users can connect to the server. The form of authentication can be either an X.509 certificate or a username and password (or both).

When `authorization` is disabled, the server does not check user authentication; user (non-admin) connections are allowed. However, even when `authorization` is disabled, admin connections must still supply the correct password to connect to the server.

Even when `authorization` is enabled, the administrator (`admin`) may explicitly allow anonymous user connections, which do not require password authorization. To allow these connections, create a user with the name `anonymous` and no password.

Note: Creating the user `anonymous` does not mean that `anonymous` has all permissions. Individual topics and queues can still be secure, and the ability to use these destinations (either sending or receiving) is controlled by the access control list of permissions for those destinations. The user `anonymous` can access only non-secure destinations.

Nonetheless, this feature (anonymous user connections) is outside the tested configuration of EMS security certification.

For more information on destination security, refer to the destination property `secure`, and Create Users.

Communication Security

For communication security between servers and clients, and between servers and other servers, you must explicitly configure TLS within EMS .

TLS communication requires software to implement TLS on both server and client. The EMS server includes the OpenSSL implementation. Java client programs use JSSE (part of the Java environment). JSSE is not a part of the EMS product. C client programs can use the OpenSSL library shipped with EMS.

For more information, see TLS Protocol

Sources of Authentication Data

The server uses only one source of X.509 certificate authentication data, namely, the server parameter `ssl_server_trusted` (its value is set in EMS an configuration file). The server can use three sources of secure password authentication data:

- Local data from the EMS configuration files.
- External data from an LDAP server (using provided JAAS LoginModules).
- A user-supplied JAAS LoginModule.

You must safeguard the security of EMS configuration files and LDAP servers.

For more information, see `ssl_server_trusted`.

Timestamp

The administration tool can either include or omit a timestamp associated with the output of each command.

To ensure a secure deployment, you must explicitly enable the timestamp feature. Use the following administration tool command:

```
time on
```

Passwords

> **Warning:** Passwords are a significant point of vulnerability for any enterprise. We recommend enforcing strong standards for passwords.
>
> For security equivalent to single DES (an industry minimum), security experts recommend passwords that contain 8–14 characters, with at least one upper case character, at least one numeric character, and at least one punctuation character.

EMS software does not automatically enforce such standards for passwords. You must enforce such policies within your organization.

Audit Trace Logs

Audit information is output to log files (and `stderr`), and is configured by the server parameters `log_trace` and `console_trace`.

For more information on these parameters, see Tracing and Log File Parameters.

The `DEFAULT` setting includes `+ADMIN`, so all administrative operations produce audit output. For further details, see Server Tracing Options.

Audit information in log files always includes a time stamp.

Administrators can read and print the log files for audit review using tools (such as text editors) commonly available within all IT environments. EMS software does not include a special tool for audit review.

Managing Access to Shared File-Based Stores

To prevent two EMS servers from using the same store file, each server restricts access to its store file for the duration of the server process. This section describes how EMS manages locked file-based stores. Shared store locking for grid stores and FTL stores is handled directly by ActiveSpaces and FTL, respectively.

Windows

On Windows platforms, servers use the standard Windows `CreateFile` function, supplying `FILE_SHARE_READ` as the `dwShareMode` (third parameter position) to restrict access to other servers.

UNIX

On UNIX platforms, servers use the standard `fcntl` operating system call to implement cooperative file locking:

```
struct flock fl;
int err;

fl.l_type = F_WRLCK;
fl.l_whence = 0;
fl.l_start = 0;
fl.l_len = 0;

err = fcntl(file, F_SETLK, &fl);
```

To ensure correct locking, we recommend checking the operating system documentation for this call, since UNIX variants differ in their implementations.

Performance Tuning

By default, the TIBCO Enterprise Message Service server has the following general thread architecture:

- A single thread to process network traffic.
- One thread for each store.

- Additional threads for various background tasks such as expiring messages, connecting routes, and so on.

Setting Thread Affinity for Increased Throughput

If the default behavior of the EMS server cannot provide the required throughput and the EMS server machine has multiple cores, you can assign specific cores to the EMS threads that handle network traffic and stores.

For instance, with a 4-core machine, you can use the `processor_ids` parameter to assign core 0 and core 1 to handle network traffic. You can also use the store configuration `processor_id` parameter to assign core 2 to handle the `$sys.failsafe store`. This configuration causes the EMS server to create two threads that handle network traffic, and sets the affinity of them to core 0 and core 1 respectively. It also sets the affinity of the thread handling the store `$sys.failsafe` to core 2. No affinity is set for other threads.

Increasing Network Threads without Setting Thread Affinity

If you want to increase the number of network threads without assigning them to specific cores, use the `network_thread_count` parameter.

The `network_thread_count` lets the EMS server control the number of network threads and also lets the administrator control the thread affinity externally (for example, by using the Linux `taskset` command).

If you set the thread affinity externally, we recommend that you avoid setting any thread affinity in the EMS server for either network traffic or stores.

The EMS server ignores the `network_thread_count` if the `processor_ids` parameter is also specified.

Determine Core Allocation

The phrase "less is more" summarizes the best practices for EMS performance tuning.

- When the EMS server does not set thread affinity, the operating system can better

schedule EMS server threads to react to changing workloads on the machine. Also examine if the application is making efficient use of the API before changing the default behavior. For example, when performing persistent messaging operations, consider using multiple threads in the applications (each with its own session) or consider using local transactions to batch sends and acknowledgments.

- Use the minimum number of threads to handle network traffic. Specifying a single thread may yield sufficient performance improvements over the default behavior, so start testing affinity there. Using excessive numbers of threads leads to greater thread contention for global data structures, which can reduce throughput and waste machine resources. Excessive numbers can also lead to more unbalanced connection assignments. TIBCO tests have shown that three (or four under some workloads) is the maximum useful number for network traffic.
- Specifying thread affinity to specific cores can provide the highest performance but can also lead to a configuration that does not react well to changing workloads. If you specify thread affinity for network traffic for persistent messaging, also set thread affinity for stores in order to prevent contention between threads handling those tasks.

Transparent Huge Pages

The Transparent Huge Pages (THP) feature of Linux does not have a significant impact on the performance of EMS.

Network I/O Connections

When a client connects to the EMS server, the EMS server assigns it to one of the threads handling network traffic based on which of those threads have the fewest existing connections. This balances the total number of connections evenly across those threads.

Note that if all the connections to one thread are closed, the EMS server does not move existing connections from other threads in order to rebalance them.

Also note that the EMS server does not account for the traffic generated by those connections. For instance, the EMS server could assign ten connections to one thread and ten connections to another thread but still have an unbalanced state if the first ten connections account for 90% of all network traffic to the EMS server.

Other Considerations

- When assigning cores for EMS use, ensure that the Operating System does not schedule those cores for other processes.
- Assign cores on the same die if possible. This reduces cache sharing between dies. High levels of cache sharing between dies reduces memory performance.
- Hyper-threads are not real cores. Disable hyper-threading if possible. Do not assign cores to the EMS server such that it sets affinity for two "cores" that are actually sharing the same physical core by hyper-threading.

Using the EMS Administration Tool

The following sections give an overview of the commands in use in the administration tool for TIBCO Enterprise Message Service.

Starting the EMS Administration Tool

The EMS Administration Tool is located in your *EMS_HOME*/bin directory and is a stand-alone executable named tibemsadmin on UNIX and tibemsadmin.exe on Windows platforms.

The EMS server must be started as described in Running the EMS Server before you start the EMS Administration Tool.

Note: When a system uses shared configuration files in the .conf format, actions performed using the tibemsadmin tool take effect only when connected to the active server.

When a system uses a shared configuration file in the .json format, most commands in the tibemsadmin tool are unavailable when connected to a server that is not in the active state. In such a situation, the only commands available are show connections, show state, shutdown, and rotatelog. In the particular case of systems configured to use FTL stores, non-active servers also support the activate_dr_site, setup_dr_site, show server, and show config commands.

Additionally, if the tibemsadmin tool is connected to the standby server, it will be disconnected when a failover occurs.

Options for tibemsadmin

Type tibemsadmin -help to display information about tibemsadmin startup parameters. All tibemsadmin parameters are optional.

Option	Description
`-help or -h`	Print the help screen.
`-script script-file`	Execute the specified text file containing `tibemsadmin` commands then quit. Any valid `tibemsadmin` command described in this chapter can be executed. Line breaks within the file delimit each command. That is, every command must be contained on a single line (no line breaks within the command), and each command is separated by a line break.
`-server server-url`	Connect to specified server.
`-user user-name`	Use this user name to connect to the server. If the server is configured for OAuth 2.0 authentication, use this user name to obtain an OAuth 2.0 access token as part of the Authentication Using OAuth 2.0.
`-password password`	Use this password to connect to the server. If the server is configured for OAuth 2.0 authentication, use this password to obtain an OAuth 2.0 access token as part of the Authentication Using OAuth 2.0.
`-pwdfile password-file`	Use the clear-text password in the specified file to connect to the server. If both `-pwdfile` and `-password` options are given, the password specified through the `-password` option takes precedence.
`-oauth2_access_token`	The OAuth 2.0 access token to use to authenticate with the EMS server. If an access token is directly provided via this option, none of the other `-oauth2_` options should be specified.
`-oauth2_server_url`	The HTTP(S) URL of the OAuth 2.0 authorization server that will issue the access tokens to be used to authenticate with the EMS server.

Option	Description
`-oauth2_client_id`	The OAuth 2.0 client ID to use when connecting to the OAuth 2.0 authorization server. This parameter is required regardless of the grant type to be used for requesting access tokens.
`-oauth2_client_secret`	The OAuth 2.0 client secret to use when connecting to the OAuth 2.0 authorization server. This parameter is required regardless of the grant type to be used for requesting access tokens.
`-oauth2_server_trust_file`	A file containing one or more PEM-encoded public certificates that can be used to validate the OAuth 2.0 authorization server's identity. This parameter is only required if establishing an HTTPS connection to the authorization server.
`-oauth2_disable_verify_hostname`	If set, the name in the CN field of the OAuth 2.0 authorization server’s certificate will not be verified. This parameter is optional. Hostname verification is performed by default.
`-oauth2_expected_hostname`	The name that is expected in the CN field of the OAuth 2.0 authorization server's certificate. This parameter is optional and only relevant when `-oauth2_disable_verify_hostname` is not set to `true`.
`-ignore`	Ignore errors when executing script file. This parameter only ignores errors in command execution but not syntax errors in the script.
`-mangle [password]`	Mangle the password and quit. Mangled string in the output can be set as a value of one of these passwords from the configuration files: • server password

Option	Description
	• server TLS password
`-ssl_trusted filename`	File containing trusted certificate(s). This parameter may be entered more than once if required.
`-ssl_identity filename`	File containing client certificate and (optionally) extra issuer certificate(s), and the private key.
`-ssl_issuer filename`	File containing extra issuer certificate(s) for client-side identity.
`-ssl_key filename`	File containing the private key.
`-ssl_password password`	Private key or PKCS#12 password. If the password is required, but has not been specified, it will be prompted for.
`-ssl_pwdfile password-file`	Use the private key or PKCS12 password in the specified file to connect to the server. If both `-ssl_pwdfile` and `-ssl_password` options are given, the password specified through the `-ssl_password` option takes precedence.
`-ssl_noverifyhost`	Do not verify the server's certificate. Server certificate verification is enabled by default.
`-ssl_noverifyhostname`	Do not verify hostname against the name on the certificate.
`-ssl_hostname name`	Name expected in the certificate sent by the host.
`-ssl_trace`	Show loaded certificates and certificates sent by the host.
`-ssl_debug_trace`	Show additional tracing, which is useful for debugging.

Note: When a command specifies `-user` and `-password`, that information is not stored for later use. It is only used to connect to the server specified in the same command line. The user name and password entered on one command line are not reused with subsequent connect commands entered in the script file or interactively.

Examples

```
tibemsadmin -server "tcp://host:7222"
tibemsadmin -server "tcp://host:7222" -user admin -password
secret
```

Some options are needed when you choose to make a TLS connection. For more information on TLS connections, refer to TLS Protocol.

When You First Start tibemsadmin

The administration tool has a default user with the name `admin`. This is the default user for logging in to the administration tool.

To protect access to the server configuration, you must assign a password to the user `admin`.

Assigning a Password to the Administrator

Procedure

1. Log in and connect to the administration tool, as described directly above.
2. Use the `set password` command to change the password:

```
set password admin password
```

Result

When you restart the administration tool and type `connect`, the administration tool now requires your password before connecting to the server.

For further information about setting and resetting passwords, refer to `set password`.

Naming Conventions

These rules apply when naming users, groups, topics or queues:

- $ is illegal at the beginning of the queue or topic names—but legal at the beginning of user and group names.
- A user name cannot contain colon (":") character.
- Space characters are permitted in a description field—if the entire description field is enclosed in double quotes (for example, "`description field`").
- Both * and > are wildcards, and cannot be used in names except as wildcards.

  For more information about wildcards, see Wildcards.
- Dot separates elements within a destination name (**foo.bar.***) and can be used only for that purpose.

Name Length Limitations

The following length limitations apply for these parameter names:

- Destination name — cannot exceed 249 characters. For more information on topic and queue naming conventions, see Destination Name Syntax.
- Username — cannot exceed 255 characters. The `username` parameter is described in `users.conf`.
- Group name — cannot exceed 255 characters. The `group-name` parameter is described in `groups.conf`.
- Client ID — cannot exceed 255 characters.
- Connection URL — cannot exceed 1000 characters.

  Note: For more information on Client ID and Connection URL, see factories.conf.
- Passwords — cannot exceed 4096 characters. This length limitation applies to

passwords used by the `tibemsd` to authenticate connecting clients or servers.

Command Listing

The command line interface of the administration tool allows you to perform a variety of functions. Note that when a system uses shared configuration files, the actions performed using the administration tool take effect only when connected to the active server.

Note: Many of the commands listed below accept arguments that specify the names of users, groups, topics or queues. For information about the syntax and that apply to these names, see Naming Conventions.

TLS aspects are addressed in TLS Protocol.

The following is an alphabetical listing of the commands including command syntax and a description of each command.

activate_dr_site

```
activate_dr_site
```

This command is only available when using FTL stores with Disaster Recovery configured.

Inform the FTL servers at the disaster recovery site that their site of operations is now the new primary site.

This command should be issued to one of the EMS servers in the fault-tolerant pair at the disaster recovery site. It should only ever be used if the primary site becomes unavailable.

add member

```
add member group_name user_name [,user2,user3,...]
```

Add one or more users to the group. User names that are not already defined are added to the group as external users; see Administration Commands and External Users and Groups.

addprop factory

```
addprop factory factory-name properties ...
```

Adds properties to the factory. Property names are separated by spaces.

See `factories.conf` for the list of factory properties.

Example

```
addprop factory MyTopicFactory ssl_trusted=cert1.pem
ssl_trusted=cert2.pem ssl_verify_host=disabled
```

addprop queue

```
addprop queue queue-name properties,...
```

Adds properties to the queue. Property names are separated by commas.

For information on properties that can be assigned to queues, see Destination Properties.

addprop route

```
addprop route route-name prop=value[ prop-value...]
```

Adds properties to the route.

Destination (topic and queue) properties must be separated by commas but properties of routes and factories are separated with spaces.

You can set the `zone_name` and `zone_type` parameters when creating a route, but you cannot subsequently change them.

For route properties, see Configure Routes and Zones.

For the configuration file `routes.conf`, see `routes.conf`.

addprop topic

```
addprop topic topic_name properties,...
```

Adds properties to the topic. Property names are separated by commas.

For information on properties that can be assigned to topics, see Destination Properties.

autocommit

```
autocommit [on|off]
```

When autocommit is set to **on**, the changes made to the configuration files are automatically saved to disk after each command. When autocommit is set to **off**, you must manually use the `commit` command to save configuration changes to the disk.

By default, autocommit is set to **on** when interactively issuing commands.

Entering **autocommit** without parameters displays the current setting of autocommit (`on` or `off`).

> **Note:** Regardless of the autocommit setting, the EMS server acts on each admin command immediately making it part of the configuration. The autocommit feature only determines when the configuration is written to the files.

commit

```
commit
```

Commits all configuration changes into files on disk.

compact

```
compact store-name max-time
compact
```

Compacts a specified store of type `file`, or all stores of type `ftl`. Compaction is not available for stores of type `as`.

For stores of type `file`:

- Since compaction for file-based stores can be a lengthy operation and it blocks other operations, a time limit (in seconds) must be specified for the operation through the *max-time* parameter. Note that *max-time* must be a number greater than zero.
- If truncation is not enabled for the store file, the `compact` command does not reduce the file size. Enable truncation using the `file_truncate` parameter in the `stores.conf` file. See `stores.conf` for more information.
- We recommend compacting the store files only when the `Used Space` usage is 30% or less (see `show store`).

For stores of type `ftl`:

- FTL stores are designed to automatically compact in the background when the underlying FTL deployment reaches a certain threshold of unused disk space. The `compact` command can be used to manually trigger this compaction process.
- The *store-name* and *max-time* arguments are not supported for the `compact` command when using FTL stores. The compaction process occurs at the FTL level and automatically affects all FTL stores; and since the compaction process occurs asynchronously over time, a time limit is not required.

connect

```
connect [server-url {admin|user_name} password]
```

Connects the administration tool to the server. Any administrator can connect. An administrator is either the admin user, any user in the `$admin` group, or any user that has administrator permissions enabled. See Administrator Permissions for more information about administrator permissions.

server-url is usually in the form:

```
protocol://host-name:port-number
```

for example:

```
tcp://myhost:7222
```

The protocol can be `tcp` or `ssl`.

If a user name or password are not provided, the user is prompted to enter a user name and password, or only the password, if the user name was already specified in the command.

You can enter `connect` with no other options and the administrative tool tries to connect to the local server on the default port, which is `7222`.

create bridge

```
create bridge source=type:dest_name target=type:dest_name [selector=msg-selector]
```

Creates a bridge between destinations.

type is either topic or queue.

For further information, see `bridges.conf`.

create durable

```
create durable topic-name durable-name [property, ... ,property]
```

Creates a static durable subscriber.

For descriptions of parameters and properties, and information about conflict situations, see `durables.conf`.

create factory

```
create factory factory_name factory_parameters
```

Creates a new connection factory.

For descriptions of factory parameters, see `factories.conf`.

create group

```
create group group_name "description"
```

Creates a new group of users.

Initially, the group is empty. You can use the `add member` command to add users to the group.

create jndiname

```
create jndiname new_jndiname topic|queue|jndiname name
```

Creates a JNDI name for a topic or queue, or creates an alternate JNDI name for a topic that already has a JNDI name.

The following example will create new JNDI name FOO referring the same object referred by JNDI name BAR

```
create jndiname FOO jndiname BAR
```

create queue

```
create queue queue_name [properties]
```

Creates a queue with the specified name and properties. The possible queue properties are described in Destination Properties. Properties are listed in a comma-separated list, as described in `queues.conf`.

create route

```
create route name url=URL [properties ...]
```

Creates a route.

The *name* must be the name of the other server to which the route connects.

The local server connects to the destination server at the specified URL. If you have configured fault-tolerant servers, you may specify the URL as a comma-separated list of URLs.

The route properties are listed in `routes.conf` and are specified as a space-separated list of parameter name and value pairs.

You can set the `zone_name` and `zone_type` parameters when creating a route, but you cannot subsequently change them.

If a passive route with the specified *name* already exists, this command promotes it to an *active-active* route; see Active and Passive Routes.

For additional information on route parameters, see Configure Routes and Zones.

create rvcmlistener

```
create rvcmlistener transport_name cm_namesubject
```

Registers an RVCM listener with the server so that any messages exported to a `tibrvcm` transport (including the first message sent) are guaranteed for the specified listener. This causes the server to perform the TIBCO Rendezvous call `tibrvcmTransport_AddListener`.

The parameters are:

- *transport_name* — the name of the transport to which this RVCM listener applies.
- *cm_name* — the name of the RVCM listener to which topic messages are to be

exported.

- *subject* — the RVCM subject name that messages are published to. This should be the same name as the topic names that specify the `export` property.

For more information, see `tibrvcm.conf` and Rendezvous Certified Messaging (RVCM) Parameters.

create topic

```
create topic topic_name [properties]
```

Creates a topic with specified name and properties. See Destination Properties for the list of properties. Properties are listed in a comma-separated list, as described in `topics.conf`.

create user

```
create user user_name ["user_description"] [password=password]
```

Creates a new user. Following the user name, you can add an optional description of the user in quotes. The password is optional and can be added later using the `set password` command.

Note: User names cannot contain colon (:) characters.

delete all

```
delete all users|groups|topics|queues|durables [topic-name-pattern|queue-name-pattern]
```

If used as `delete all users|groups|topics|queues|durables` without the optional parameters, the command deletes all users, groups, topics, or queues (as chosen).

If used with a topic or queue, and the optional parameters, such as those seen below, the command deletes all topics and queues that match the topic or queue name pattern.

```
delete all topics|queues topic-name-pattern|queue-name-pattern
```

delete bridge

```
delete bridge source=type:dest_name target=type:dest_name
```

Delete the bridge between the specified source and target destinations.

type is either `topic` or `queue`.

See Destination Bridges for more information on bridges.

delete connection

```
delete connection connection-id
```

Delete the named connection for the client. The connection ID is shown in the first column of the connection description printed by `show connection`.

delete durable

```
delete durable durable-name [client ID]
```

Delete the named durable subscriber.

When both the durable name and the client ID are specified, the server looks for a durable named *clientID*:*durable-name* in the list of durables. If a matching durable subscriber is not found, the administration tool prints an error message including the fully qualified durable name.

See also, Conflicting Specifications.

delete factory

```
delete factory factory-name
```

Delete the named connection factory.

delete group

```
delete group group-name
```

Delete the named group.

delete jndiname

```
delete jndiname jndiname
```

Delete the named JNDI name. Notice that deleting the last JNDI name of a connection factory object will remove the connection factory object as well.

See The EMS Implementation of JNDI for more information.

delete message

```
delete message messageID
```

Delete the message with the specified message ID.

delete queue

```
delete queue queue-name
```

Delete the named queue.

delete route

```
delete route route-name
```

Delete the named route.

delete rvcmlistener

```
delete rvcmlistener transport_name cm_namesubject
```

Unregister an RVCM listener with the server so that any messages being held for the specified listener in the RVCM ledger are released. This causes the server to perform the TIBCO Rendezvous call `tibrvcmTransport_RemoveListener`.

The parameters are:

- *transport_name* — the name of the transport to which this RVCM listener applies.
- *cm_name* — the name of the RVCM listener to which topic messages are exported.
- *subject* — the RVCM subject name that messages are published to. This should be the same name as the topic names that specify the export property.

For more information, see `tibrvcm.conf` and Rendezvous Certified Messaging (RVCM) Parameters.

delete topic

```
delete topic topic-name
```

Delete the named topic.

delete user

```
delete user user-name
```

Delete the named user.

disconnect

```
disconnect
```

Disconnect the administrative tool from the server.

echo

```
echo [on|off]
```

Echo controls the reports that are printed into the standard output. When `echo` is off the administrative tool only prints errors and the output of queries. When `echo` is `on`, the administrative tool report also contains a record of successful command execution.

Choosing the parameter on or off in this command controls `echo`. If `echo` is entered in the command line without a parameter, it displays the current echo setting (`on` or `off`). This command is used primarily for scripts.

The default setting for echo is `on`.

exit

```
exit  (aliases: quit, q, bye, end)
```

Exit the administration tool.

The administrator may choose the `exit` command when there are changes in the configuration have which have not been committed to disk. In this case, the system will prompt the administrator to use the `commit` command before exiting.

grant queue

```
grant queue queue-name user=name | group=namepermissions
```

Grants specified permissions to specified user or group on specified queue. The name following the queue name is first checked to be a group name, then a user name.

Specified permissions are added to any existing permissions. Multiple permissions are separated by commas. Enter **all** in the *permissions* string if you choose to grant all possible user permissions.

User permissions are:

- `receive`
- `send`
- `browse`

For more information on queue permissions, see User Permissions.

Destination-level administrator permissions can also be granted with this command. The following are administrator permissions for queues.

- `view`
- `create`
- `delete`
- `modify`
- `purge`

For more information on destination permissions, see Destination-Level Permissions.

grant topic

```
grant topic topic-name user=name | group=namepermissions
```

Grants specified permissions to specified user or group on specified topic. The name following the topic name is first checked to be a group name, then a user name.

Specified permissions are added to any existing permissions. Multiple permissions are separated by commas. Enter **all** in the *permissions* string if you choose to grant all possible permissions.

Topic permissions are:

- `subscribe`
- `publish`
- `durable`
- `use_durable`

For more information on topic permissions, see User Permissions.

Destination-level administrator permissions can also be granted with this command. The following are administrator permissions for topics.

- `view`
- `create`
- `delete`
- `modify`
- `purge`

For more information on destination permissions, see Destination-Level Permissions.

grant admin

```
grant admin user=name | group=name admin_permissions
```

Grant the named global administrator permissions to the named user or group. For a complete listing of global administrator permissions, see Global Administrator Permissions.

help

```
help (aliases: h, ?)
```

Display help information.

Enter `help` commands for a summary of all available commands.

Enter `help`*command* for help on a specific command.

info

```
info (alias: i)
```

Shows server name and information about the connected server.

jaci clear

```
jaci clear
```

Empties the JACI permission cache of all entries.

jaci resetstats

```
jaci resetstats
```

Resets all statistics counters for the JACI cache to zero.

jaci showstats

```
jaci showstats
```

Prints statistics about JACI cache performance.

purge all queues

```
purge all queues [pattern]
```

Purge all or selected queues.

When used without the optional pattern parameter, this command erases all messages in all queues for all receivers.

When used with the *pattern* parameter, this command erases all messages in all queues that fit the pattern (for example: `foo.*`).

purge all topics

```
purge all topics [pattern]
```

Purge all or selected topics.

When used without the optional pattern parameter, this command erases all messages in all topics for all subscribers.

When used with the *pattern* parameter, this command erases all messages in all topics that fit the pattern (for example: `foo.*`).

purge durable

```
purge durable durable-name [client ID]
```

Purge all messages in the topic for the named durable subscriber.

When both the durable name and the client ID are specified, the server looks for a durable named *clientID:durable-name* in the list of durables.

purge queue

```
purge queue queue-name
```

Purge all messages in the named queue.

purge topic

```
purge topic topic-name
```

Purge all messages for all subscribers on the named topic.

remove member

```
remove member group-name user-name[,user2,user3,...]
```

Remove one or more named users from the named group.

removeprop factory

```
removeprop factory factory-name properties
```

Remove the named properties from the named factory. See Connection Factory Parameters for a list of properties.

removeprop queue

```
removeprop queue queue-name properties
```

Remove the named properties from the named queue.

removeprop route

```
removeprop route route-name properties
```

Remove the named properties from the named route.

You cannot remove the URL.

You can set the `zone_name` and `zone_type` parameters when creating a route, but you cannot subsequently change them.

For route parameters, see Configure Routes and Zones.

For the configuration file `routes.conf`, see `routes.conf`.

removeprop topic

```
removeprop topic topic-name properties
```

Remove the named properties from the named topic.

resume route

```
resume route route-name
```

Resumes sending messages to named route, if messages were previously suspended using the `suspend route` command.

revoke admin

```
revoke admin user=name | group=name permissions
```

Revoke the specified global administrator permissions from the named user or group. See Authentication and Permissions, for more information about administrator permissions.

revoke queue

```
revoke queue queue-name user=name | group=name permissions
revoke queue queue-name * [user | admin | both]
```

Revoke the specified permissions from a user or group for the named queue.

User and group permissions for queues are `receive`, `send`, `browse`, and `all`. Administrator permissions for queues are `view`, `create`, `delete`, `modify`, and `purge`.

If you specify an asterisk (*), all user-level permissions on this queue are removed. You can use the optional `admin` parameter to revoke all administrative permissions, or the `both` parameter to revoke all user-level and administrative permissions on the queue.

For more information, see Authentication and Permissions.

revoke topic

```
revoke topic topic-name user=name | group=name permissions
revoke topic topic-name * [user | admin | both]
```

Revoke the specified permissions from a user or group for the named topic.

User and group permissions for topics are `subscribe`, `publish`, `durable`, `use_durable`, and `all`. Administrator permissions for topics are `view`, `create`, `delete`, `modify`, and `purge`.

If you specify an asterisk (*), all user-level permissions on this topic are removed. You can use the optional `admin` parameter to revoke all administrative permissions, or the `both` parameter to revoke all user-level and administrative permissions on the topic.

For more information, see Authentication and Permissions.

rotatelog

```
rotatelog
```

Force the current log file to be backed up and truncated. The server starts writing entries to the newly empty log file.

The backup file name is the same as the current log file name with a sequence number appended to the filename. The server queries the current log file directory and determines what the highest sequence number is, then chooses the next highest sequence number for the new backup name. For example, if the log file name is `tibems.log` and there is already a `tibems.log.1` and `tibems.log.2`, the server names the next backup `tibems.log.3`.

This command is not supported if the EMS server is using FTL stores.

save_and_exit

```
save_and_exit
```

When using in-memory replication with FTL stores, save the state of each server to disk and exit. Each server's state is written to a file named `<FTL server name>.state` in its FTL store-specific data directory. If a state file already exists in the directory, the server will first rename it as `<FTL server name>.state.backup` before writing the new state file.

This command should be used when all FTL servers have to be shut down. The YAML configuration of the FTL server cluster will need to be altered to instruct each of the servers to load its saved state upon restart.

This command is only relevant when using FTL stores with `in_memory_replication` enabled. See Shutting down and Restarting an In-Memory Cluster for more details.

set password

```
set password user-name [password]
```

Set the password for the named user.

If you do not supply a password in the command, the server prompts you to type one.

- To reset a password, type:

  ```
  set password user-name
  ```

  Type a new password at the prompt.

- To remove a password, use this command without supplying a password, and press the **Enter** key at the prompt (without typing a password).

> **Warning:** Passwords are a significant point of vulnerability for any enterprise. We recommend enforcing strong standards for passwords.
>
> For security equivalent to single DES (an industry minimum), security experts recommend passwords that contain 8–14 characters, with at least one upper case character, at least one numeric character, and at least one punctuation character.

set server

```
set server parameter=value [parameter=value ...]
```

The `set server` command can control many parameters. Multiple parameters are separated by spaces. The following table describes the parameters you can set with this command.

Parameter	Description
`password [= string]`	Sets server password used by the server to connect to other routed servers. If the value is omitted it is prompted for by the administration tool. Entered value will be stored in the main server configuration file in mangled form (but not encrypted). To reset this password, enter the empty string twice at the prompt.
`authorization=enabled|disabled`	Sets the `authorization` mode in the `tibemsd.conf` file. After a transition from disabled to enabled, the server checks ACL permissions for all subsequent requests. While the server

Parameter	Description
	requires valid authentication for existing producers and consumers, it does not retroactively reauthenticate them; it denies access to users without valid prior authentication.
`log_trace=`*`trace-items`*	Sets the trace preference on the file defined by the `logfile` parameter. If `logfile` is not set, the values are stored but have no effect. The value of this parameter is a comma-separated list of trace options. For a list of trace options and their meanings, see Server Tracing Options. You may specify trace options in three forms: • plain A trace option without a prefix character replaces any existing trace options. • + A trace option preceded by + adds the option to the current set of trace options. • - A trace option preceded by - removes the option from the current set of trace options. **Examples** The following example sets the trace log to only show messages about access control violations. `log_trace=ACL` The next example sets the trace log to show all default trace messages, in

Parameter	Description
	addition to TLS messages, but ADMIN messages are not shown. `log_trace=DEFAULT,-ADMIN,+SSL`
`console_trace=`*`console-trace-items`*	Sets trace options for output to `stderr`. The values are the same as for `log_trace`. However, console tracing is independent of log file tracing. If `logfile` is defined, you can stop console output by specifying: `console_trace=-DEFAULT` Note that important error messages (and some other messages) are always output, overriding the trace settings. **Examples** See `log_trace` above.
`client_trace={enabled|disabled} [target=`*`location`*`] [filter=`*`value`*`]`	Administrators can trace a connection or group of connections. When this property is enabled, the client generates trace output for opening or closing a connection, message activity, and transaction activity. This type of tracing does not require restarting the client program. The client sends trace output to *location*, which may be either `stderr` (the default) or `stdout`. You can specify a filter to selectively trace specific connections. The *filter* can be user, `connid` or `clientid`. The *value* can be a user name or ID (as appropriate to the filter).

Parameter	Description
	When the filter and value clause is absent, the default behavior is to trace all connections. Setting this parameter using the administration tool does not change its value in the configuration file `tibemsd.conf`.
`max_msg_memory=`*`value`*	Maximum memory the server can use for messages. For a complete description, see `max_msg_memory` in `tibemsd.conf`. Specify units as `KB`, `MB` or `GB`. The minimum value is `8MB`. Zero is a special value, indicating no limit. Lowering this value will not immediately free memory occupied by messages.
`msg_swapping=enabled|disabled`	Enables or disables the ability to swap messages to disk.
`track_message_ids=enabled|disabled`	Enables or disables tracking messages by MessageID.
`track_correlation_ids=` `enabled|disabled`	Enables or disables tracking messages by CorrelationID.
`ssl_password[=`*`string`*`]`	This sets a password for TLS use only. Sets private key or PKCS#12 file password used by the server to decrypt the content of the server identity file. The password is stored in mangled form.
`ft_ssl_password[=`*`string`*`]`	This sets a password for TLS use with Fault Tolerance.

Parameter	Description
	Sets private key or PKCS#12 file password used by the server to decrypt the content of the FT identity file. The password is stored in mangled form. This parameter is not supported when the EMS server is using FTL stores.
`server_rate_interval=`*num*	Sets the interval (in seconds) over which overall server statistics are averaged. This parameter can be set to any positive integer greater than zero. Overall server statistics are always gathered, so this parameter cannot be set to zero. By default, this parameter is set to 1. Setting this parameter allows you to average message rates and message size over the specified interval.
`statistics=enabled|disabled`	Enables or disables statistic gathering for producers, consumers, destinations, and routes. By default this parameter is set to disabled. Disabling statistic gathering resets the total statistics for each object to zero.
`rate_interval=`*num*	Sets the interval (in seconds) over which statistics for routes, destinations, producers, and consumers are averaged. By default, this parameter is set to 3 seconds. Setting this parameter to zero disables the average calculation.
`detailed_statistics=` `NONE| PRODUCERS,CONSUMERS,ROUTES`	Specifies which objects should have detailed statistic tracking. Detailed statistic tracking is only appropriate for routes,

Parameter	Description
	producers that specify no destination, or consumers that specify wildcard destinations. When detailed tracking is enabled, statistics for each destination are kept for the object. Setting this parameter to NONE disables detailed statistic tracking. You can specify any combination of PRODUCERS, CONSUMERS, or ROUTES to enable tracking for each object. If you specify more than one type of detailed tracking, separate each item with a comma.
`statistics_cleanup_interval=`*num*	Specifies how long (in seconds) the server should keep detailed statistics if the destination has no activity. This is useful for controlling the amount of memory used by detailed statistic tracking. When the specified interval is reached, statistics for destinations with no activity are deleted.
`max_stat_memory=`*num*	Specifies the maximum amount of memory to use for detailed statistic gathering. If no units are specified, the amount is in bytes, otherwise you can specify the amount using KB, MB, or GB as the units. Once the maximum memory limit is reached, the server stops collecting detailed statistics. If statistics are deleted and memory becomes available, the server resumes detailed statistic gathering.

setprop factory

```
setprop factory factory-name properties ...
```

Set the properties for a connection factory, overriding any existing properties. Multiple properties are separated by spaces. See Connection Factory Parameters for the list of the properties that can be set for a connection factory.

setprop queue

```
setprop queue queue-name properties, ...
```

Set the properties for a queue, overriding any existing properties. Any properties on a queue that are not explicitly specified by this command are removed.

Multiple properties are separated by commas. See Destination Properties for the list of the properties that can be set for a queue.

setprop route

```
setprop route route-name properties ...
```

Set the properties for a route, overriding any existing properties. Any properties on a route that are not explicitly specified by this command are removed.

You can set the `zone_name` and `zone_type` parameters when creating a route, but you cannot subsequently change them.

Multiple properties are separated by spaces. For route parameters, see routes.conf and Configure Routes and Zones.

setprop topic

```
setprop topic topic-name properties
```

Set topic properties, overriding any existing properties. Any properties on a topic that are not explicitly specified by this command are removed.

Multiple properties are separated by commas. See Destination Properties for the list of the properties that can be set for a topic.

setup_dr_site

```
setup_dr_site url_list
```

This command is only available when using FTL stores with Disaster Recovery configured.

Establish a new Disaster Recovery (DR) site after the original DR site has become the new primary site.

This command must be issued to the active EMS server of the FTL server cluster at the designated new DR site.

The *url_list* is a pipe-separated list of URLs of the FTL server cluster at the primary site. Each URL should be of the form:

```
<FTL server name>@<host>:<port>
```

show bridge

```
show bridge topic|queue bridge_source
```

Display information about the configured bridges for the named topic or queue. The *bridge_source* is the name of the topic or queue established as the source of the bridge.

The following is example output for this command:

```
Target Name    Type Selector
queue.dest        Q
topic.dest.1      T "urgency in ('high', 'medium')"
topic.dest.2      T
```

The names of the destinations to which the specified destination has configured bridges are listed in the Target Name column. The type and the message selector (if one is defined) for the bridge are listed in the Type and Selector column.

show bridges

```
show bridges [type=topic|queue] [pattern]
```

Shows a summary of the destination bridges that are currently configured. The type option specifies the type of destination established as the bridge source. For example, `show bridges topic` shows a summary of configured bridges for all topics that are established as a bridge source. The *pattern* specifies a pattern to match for source destination names. For example `show bridges foo.*` returns a summary of configured bridges for all source destinations that match the name `foo.*`. The `type` and *pattern* are optional.

The following is example output for this command:

```
  Source Name        Queue Targets  Topic Targets
Q queue.source                   1              1
T topic.source                   1              2
```

Destinations that match the specified pattern and/or type are listed in the Source Name column. The number of bridges to queues for each destination is listed in the Queue Targets column. The number of bridges to topics for each destination is listed in the Topic Targets column.

show config

```
show config
```

Shows the configuration parameters for the connected server. The output includes:

- configuration files
- server database
- server JVM
- listen ports
- configuration settings
- message tracking
- server tracing parameters

- statistics settings
- fault-tolerant setup
- external transport setup
- server TLS setup

show consumer

```
show consumer consumerID
```

Shows details about a specific consumer. The *consumerID* can be obtained from the show consumers output.

show consumers

```
show consumers  [topic=name | queue=name] [durable] [user=name]
[connection=id] [sort=conn|user|dest|msgs] [full]
```

Shows information about all consumers or only consumers matching specified filters. Output of the command can be controlled by specifying the `sort` or `full` parameter. If the `topic` or `queue` parameter is specified, then only consumers on destinations matching specified queue or topic are shown. The `user` and/or `connection` parameters show consumers only for the specified user or connection. Note that while the queue browser is open, it appears as a consumer in the EMS server.

The `durable` parameter shows only durable topic subscribers and queue receivers, but it does not prevent queue consumers to be shown. To see only durable topic consumers, use:

```
show consumers topic=> durable
```

The `sort` parameter sorts the consumers by either connection ID, user name, destination name, or number of pending messages. The full parameter shows all columns listed below and can be as wide as 120-140 characters or wider. Both topic and queue consumers are shown in separate tables, first the topic consumers and then the queue consumers.

Note: When connected to an EMS 8.0 or higher server, this command no longer displays offline durable subscribers. In order to see offline durables, use the command `show durables` or `show subscriptions`.

show consumers (description of output fields)

Heading	Description
`Id`	Consumer ID.
`Conn`	Consumer's connection ID. If performed on an EMS 7.x or earlier server, this field displays '-' to indicate a disconnected durable topic subscriber.
`Sess`	Consumer's session ID. If performed on an EMS 7.x or earlier server, this field displays '-' to indicate a disconnected durable topic subscriber.
`T`	Consumer type character which can be one of: For topic consumer: • T - non-durable topic subscriber. • D - durable topic subscriber. • R - system-created durable for a routed topic. • P - proxy subscriber on route's temporary topic. For queue consumer: • Q - regular queue receiver. • q - inactive queue receiver. • P - system-created receiver on global queue for user receiver created in one of routes.
`Topic/Queue`	Name of the subscription topic or queue.
`Name`	(Topics Only.) Durable or shared subscription name. This column is shown

Heading	Description
	for topic consumers if at least one consumer is a durable or shared consumer.
SAS[NMBS]	Description of columns: • S - '+' if consumer's connection started, '-' otherwise. • A - mode of consumer's session, values are: ◦ N - no acknowledge ◦ A - auto acknowledge ◦ D - dups_ok acknowledge ◦ C - client acknowledge ◦ T - session is transactional ◦ X - XA or MS DTC session ◦ Z - connection consumer • S - '+' if consumer has a selector, '-' otherwise. • N - (TOPICS ONLY) '+' if subscriber is "NoLocal." • B - (QUEUES ONLY) '+' if consumer is a queue browser. • S - (TOPICS ONLY) '+' if this is a shared consumer.
Pre	Prefetch value of the consumer's destination.
Pre Dlv	Number of prefetch window messages delivered to consumer
Msgs Sent	Current number of messages sent to consumer which are not yet acknowledged by consumer's session.
Size Sent	Combined size of unacknowledged messages currently sent to consumer. Value is rounded and shown in bytes, (K)ilobytes, (M)egabytes or (G)igabytes.
Pend Msgs	(Topics Only.) Total number of messages pending for the topic consumer.
Pend Size	(Topics Only.) Combined size of messages pending for the topic consumer.

Heading	Description
	Value is rounded and shown in bytes, (K)ilobytes, (M)egabytes or (G)igabytes.
`Uptime`	Uptime of the consumer.
`Last Sent`	Approximate time elapsed since last message was sent by the server to the consumer. Value is approximate with precision of 1 second.
`Last Ackd`	Approximate time elapsed since last time a message sent to the consumer was acknowledged by consumer's session. Value is approximate with precision of 1 second.
`Total Sent`	Total number of messages sent to consumer since it was created. This includes resends due to session recover or rollback.
`Total Acked`	Total number of messages sent to the consumer and acknowledged by consumer's session since consumer created.

show connections

```
show connections [type=q|t|s] [host=hostname] [user=username] [version]
[address] [counts] [full]
```

Show connections between clients and server. The table show connections (description of output fields) describes the output.

The `type` parameter selects the subset of connections to display as shown in the following table. The `host` and `user` parameters can further narrow the output to only those connections involving a specific host or user. When the `version` flag is present, the display includes the client's version number.

If the `address` parameter is specified, then the IP address is printed in the output table. If the `counts` parameter is specified, then number of producers, consumers and temporary destinations are printed. Specifying the `full` parameter prints all of the available information.

Type	Description
`type=q`	Show queue connections only.
`type=t`	Show topic connections only.
`type=s`	Show system connections only.
absent	Show queue and topic connections, but not system connections.

show connections (description of output fields)

Heading	Description
`L`	The type of client. Can be one of the following: • J — Java client • C — C client • # — C# client • - — unknown system connection
`Version`	The EMS version of the client.
`ID`	Unique connection ID. Each connection is assigned a unique, numeric ID that can be used to delete the connection.
`FSXT`	Connection type information. The `F` column displays whether the connection is fault-tolerant. • - — not a fault-tolerant connection, that is, this connection has no alternative URLs • + — fault-tolerant connection, that is, this connection has alternative URLs The S column displays whether the connection uses TLS. • - — connection is not TLS • + — connection is TLS The X column displays whether the connection is an XA or MS DTC

Heading	Description
	transaction. • - — connection is not XA or MS DTC • + — connection is either an XA or MS DTC connection The T column displays the connection type. • C — generic user connection • T — user TopicConnection • Q — user QueueConnection • A — administrative connection • R — system connection to another route server • F — system connection to the fault-tolerant server
`S`	Connection started status, + if started, - if stopped.
`IP Address`	Shows client IP address. The address or `full` parameter must be specified to display this field.
`Port`	The ephemeral port used by the client on the client machine. The address or `full` parameter must be specified to display this field.
`Host`	Connection's host name. (If the name is not available, this column displays the host's IP address.)
`Address`	Connection's IP address. If you supply the keyword `address`, then the table includes this column.
`User`	Connection user name. If a user name was not provided when the connection was created, it is assigned the default user name `anonymous`.
`ClientID`	Client ID of the connection.
`Sess`	Number of sessions on this connection.

Heading	Description
Prod	Number of producers on this connection. The counts or full parameter must be specified to display this field.
Cons	Number of consumers on this connection. The counts or full parameter must be specified to display this field.
TmpT	Number of temporary topics created by this connection. The counts or full parameter must be specified to display this field.
TmpQ	Number of temporary queues created by this connection. The counts or full parameter must be specified to display this field.
Uncomm	Number of messages in uncommitted transactions on the connection. The counts or full parameter must be specified to display this field.
UncommSize	The combined size, in bytes, of messages in uncommitted transactions on the connection. The counts or full parameter must be specified to display this field.
Uptime	Time that the connection has been in effect.

show db

```
show db
```

Print a summary of the server's databases. Databases are also printed by `show stores`, the preferred command. This command is only supported when using file-based stores.

See `show store` for details about a specific database.

show durable

```
show durable durable-name [client ID]
```

Show information about a durable subscriber.

When both the durable name and the client ID are specified, the server looks for a durable named *clientID:durable-name* in the list of durables.

show durable (description of output field)

Heading	Description
Durable Subscriber	Fully qualified name of the durable subscriber. This name concatenates the client ID (if any) and the subscription name (separated by a colon).
Subscription name	Full name of the durable subscriber.
Shared	`yes` if this is a shared durable subscription, no otherwise.
Client ID	Client ID of the subscriber's connection.
Topic	The topic from which the durable subscription receives messages.
Type	`dynamic`—created by a client `static`—configured by an administrator
Status	`online` `offline`
Username	Username of the durable subscriber (that is, of the client's connection). If the durable subscriber is currently offline, the value in this column is `offline`.
Consumer ID	This internal ID number is not otherwise available outside the server.
No Local	`enabled`—the subscriber does not receive messages sent from its local connection (that is, the same connection as the subscriber).

Heading	Description
	`disabled`—the subscriber receives messages from all connections.
Selector	The subscriber receives only those messages that match this selector.
Pending Msgs	Number of all messages in the topic. (This count includes the number of delivered messages.)
Delivered Msgs	Number of messages in the topic that have been delivered to the durable subscriber, but not yet acknowledged.
Pending Msgs Size	Total size of all pending messages

show durables

```
show durables [pattern]
```

If a pattern is not entered, this command shows a list of all durable subscribers on all topics.

If a pattern is entered (for example `foo.*`) this command shows a list of durable subscribers on topics that match that pattern.

show durable (description of output fields)

Heading	Description
Topic Name	Name of the topic. An asterisk preceding this name indicates a dynamic durable subscriber. Otherwise the subscriber is static (configured by an administrator).
Durable	Full name of the durable subscriber.
Shared	`Y` to indicate that this is a shared durable subscription, `N` otherwise.

Heading	Description
User	Name of the user of this durable subscriber. If the durable subscriber is currently `offline`, the value in this column is offline. If this is a shared durable subscription, the value of this column is `shared`. For users defined externally, there is an asterisk in front of the user name.
Msgs	Number of pending messages
Size	Total size of pending messages

For more information, see Destination Properties.

show factory

```
show factory factory-name
```

Shows properties of specified factory.

show factories

```
show factories [generic|topic|queue]
```

Shows all factories. You can refine the listed output by specifying only generic, topic, or queue factories be listed.

show jndiname

```
show jndiname jndi-name
```

Shows the object that the specified name is bound to by the JNDI server.

show jndinames

```
show jndinames [type]
```

The optional parameter *type* can be:

- destination
- topic
- queue
- factory
- topicConnectionFactory
- queueConnectionFactory

When type is specified only JNDI names bound to objects of the specified type are shown. When type is not specified, all JNDI names are shown.

show group

```
show group group-name
```

Shows group name, description, and number of members in the group.

For groups defined externally, there is an asterisk in front of the group name. Only external groups with at least one currently connected user are shown.

show groups

```
show groups
```

Shows all user groups.

For groups defined externally, there is an asterisk in front of the group name.

show members

```
show members group-name
```

Shows all user members of specified user group.

show message

```
show message messageID
```

Shows the message for the specified message id.

This command requires that tracking by message ID be turned on using the `track_message_ids` configuration parameter.

show messages

```
show messages correlationID
```

Shows the message IDs of all messages with the specified correlation ID set as `JMSCorrelationID` message header field. You can display the message for each ID returned by this command by using the `show message` *messageID* command.

This command requires that tracking by correlation ID be turned on using the `track_correlation_ids` configuration parameter.

show parents

```
show parents user-name
```

Shows the user's parent groups. This command can help you to understand the user's permissions.

show queue

```
show queue queue-name
```

Shows the details for the specified queue.

Note: If the queue is a routed queue, specify only the name of the queue (do not specify the server using the *queue-name@server* form).

show queue (description of output fields)

Heading	Description
Queue	Full name of the queue.
Type	`dynamic`—created by a client `static`—configured by an administrator
Properties	A list of property names that are set on the queue, and their values. For an index list of property names, see Destination Properties.
JNDI Names	A list of explicitly assigned JNDI names that refer to this queue.
Bridges	A list of bridges from this queue to other destinations.
Receivers	Number of consumers on this queue.
Pending Msgs	Number of all messages in the queue, followed by the number of persistent messages in parenthesis. These counts include the number of delivered messages.
Delivered Msgs	Number of messages in the queue that have been delivered to a consumer, but not yet acknowledged.
Pending Msgs Size	Total size of all pending messages, followed by the size of all persistent messages in parenthesis.

show queues

```
show queues [pattern-name [notemp|static|dynamic] [first=n|next=n|last=n]]
```

If a *pattern-name* is not entered, this command shows a list of all queues.

If a *pattern-name* is entered (for example `foo.*` or `foo.>`) this command shows a list of queues that match that pattern. See Wildcards * and > for more information about using wildcards.

You can further refine the list of queues that match the pattern by using one of the following parameters:

- `notemp` — do not show temporary queues
- `static` — show only static queues
- `dynamic` — show only dynamic queues

When a *pattern-name* is entered, you can also cursor through the list of queues using one of the following commands, where *n* is whole number:

- `first=`*n* — show the first *n* queues
- `next=`*n* — show the next *n* queues
- `last=`*n* — show the next *n* queues and terminate the cursor

The cursor examines *n* queues and displays queues that match the *pattern-name*. Because it does not traverse the full list of queues, the cursor may return zero or fewer than *n* queues. To find all matching queues, continue to use next until you receive a `Cursor complete` message.

A * appearing before the queue name indicates a dynamic queue.

show queues (description of output fields)

Heading	Description
Queue Name	Name of the queue. If the name is prefixed with an asterisk (*), then the queue is temporary or was created dynamically. Properties of dynamic and temporary queues cannot be changed.
SNFGXIBCT	Prints information on the topic properties in the order

Heading	Description
	(S)ecure (N)sender_name or sender_name_enforced (F)ailsafe (G)lobal e(X)clusive (I)mport (B)ridge (C)flowControl (T)race The characters in the value section show: - Property not present + Property is present, and was set on the queue itself * Property is present, and was inherited from another queue Note that inherited properties cannot be removed.
Pre	Prefetch value. If the value is followed by an asterisk (*), then it is inherited from another queue or is the default value.
Rcvrs	Number of currently active receivers
All Msgs	
Msgs	Number of pending messages
Size	Total size of pending messages
Persistent Msgs	
Msgs	Number of pending persistent messages
Size	Total size of pending persistent messages

For more information, see Destination Properties.

show route

```
show route route-name
```

Shows the properties (URL and TLS properties) of a route.

show routes

```
show routes
```

Shows the properties (URL and TLS properties) of all created routes.

These commands print the information described in the following table.

Heading	Description
Route	Name of the route.
T	Type of route: • A indicates an active route. • P indicates a passive route.
ConnID	Unique ID number of the connection from this server to the server at the other end of the route. A hyphen (-) in this column indicates that the other server is not connected.
URL	URL of the server at the other end of the route.
ZoneName	Name of the zone for the route.
ZoneType	Type of the zone: • m indicates a multi-hop zone. • 1 indicates a one-hop zone.

show rvcmtransportledger

```
show rvcmtransportledger transport_name [subject-or-wildcard]
```

Displays the TIBCO Rendezvous certified messaging (RVCM) ledger file entries for the specified transport and the specified subject. You can specify a subject name, use

wildcards to retrieve all matching subjects, or omit the subject name to retrieve all ledger file entries.

For more information about ledger files and the format of ledger file entries, see TIBCO Rendezvous documentation.

show rvcmlisteners

```
show rvcmlisteners
```

Shows all RVCM listeners that have been created using the `create rvcmlistener` command or by editing the `tibrvcm.conf` file.

show server

```
show server (aliases: info, i)
```

Shows server name and information about the connected server.

show stat

```
show stat consumers [topic=name|queue=name] [user=name]
          [connection=id] [total]
show stat producers [topic=name|queue=name] [user=name]
          [connection=id] [total]
show stat route name [topic=name|queue=name] [total] [wide]
show stat topic name [total] [wide]
show stat queue name [total] [wide]
```

Displays statistics for the specified item. You can display statistics for consumers, producers, routes, or destinations. Statistic gathering must be enabled for statistics to be displayed. Also, detailed statistics for each item can be displayed if detailed statistic tracking is enabled. Averages for inbound/outbound messages and message size are available if an interval is specified in the `rate_interval` configuration parameter.

The `total` keyword specifies that only total number of messages and total message size for the item should be displayed. The `wide` keyword displays inbound and outbound message statistics on the same line.

See Server Statistics for a complete description of statistics and how to enable/disable statistic gathering options.

Note: When connected to an EMS 8.0 or higher server, this command does not return statistics for offline durable subscribers.

show state

```
show state
```

Shows the state and a minimal subset of the information about the connected EMS server.

show store

```
show store store-name
```

Show the details of a specific store.

The *store-name* must be the exact name of a specific store.

This command prints a table of information described in the following table.

Heading	Description
Type	Type of store: • `file` indicates a file-based store. • `as` indicates a grid store. • `ftl` indicates an FTL store.
Message Count	The number of messages in the store.

Heading	Description
Swapped Count	The number of messages that have been swapped from process memory to the store.
Average Write Time	Average time in seconds a write call takes. (Not available for asynchronous file stores.)
Write Usage	The ratio between time spent within write calls and the time specified by the `server_rate_interval`. (Not available for asynchronous file stores.)
Access Mode	asynchronous—the server stores messages in the store using asynchronous I/O calls. synchronous—the server stores messages in the store using synchronous I/O calls.
Message Size	Total size of all messages in the store.
Swapped Size	The total size of swapped messages in the store.
Headings specific to file-based stores	
File	File name associated with this store file, as it is set by the file parameter in the `stores.conf` file.
Pre-allocation Minimum	The amount of disk space, if any, that is preallocated to this file.
Periodic Truncation	`enabled`—the EMS server occasionally truncates the store file, relinquishing unused disk space. `disabled`—the EMS server does not truncate the store file to relinquish unused disk space.
Destination Defrag Batch Size	The size of the batch used by the destination defrag feature.
File Size	The size of the store file, including unused allocated file space.

Heading	Description
Free Space	The amount of unused allocated file space.
Fragmentation	The level of fragmentation in the file.
Used Space	The amount of used space in the file.
Storage Write Rate	The number of bytes written per second.
Headings specific to grid stores	
Grid URL	The pipe-separated URLs of the data grid the store is connected to.
Grid Name	Name of the data grid the store is connected to.
Discard Scan Interval	The maximum length of time that the EMS server takes to examine all messages in the grid store. This interval is controlled with the `scan_iter_interval` store parameter. See `scan_iter_interval` for more information.
Discard Scan Interval Bytes	The bytes read and processed every Discard Scan Interval. This number is proportional to the grid store file size, and must be kept within the limits of your storage medium. See Understanding Grid Store Intervals for more information.
First Scan Finished	`true`—all the data in the store has been examined at least once since the EMS server startup. `false`—not all data has been examined since the EMS server last started. When `false`, certain server statistics (such as the Message Count field) may be underreported as a result of expired or purged messages still in the store. See Implications for Statistics for more information.

show stores

```
show stores
```

Print a list of the server's stores.

show subscriptions

```
show subscriptions [topic=name] [name=sub-name] [shared=only|none]
[durable=only|none] [sort=msgs|topic|name|cons|id]
```

This command prints information about all topic subscriptions, or only subscriptions matching specified filters. Command output is controlled using the sort parameter.

If `topic=`*name* is specified, then only subscriptions on destinations matching specified topic are shown. If `name=`*sub-name* is specified, then only subscriptions of that name are shown.

If `durable=only` is specified, then only durable subscriptions are shown.

If `durable=none` is specified, then only non-durable subscriptions are shown.

If `shared=only` is specified, then only shared subscriptions are shown.

If `shared=none` is specified, then only unshared subscriptions are shown.

The parameter `sort` allows you to specify how the command output is sorted in the output table. You can use to sort by number of pending messages, topic name, subscription name, number of consumers on that subscription, or the subscription's identifier.

show subscriptions (description of output fields)

Heading	Description
Id	The ID of the subscription.
T	The subscription type: • `T` — non-durable subscription • `D` — durable subscription
Topic	Name of the topic associated with the subscription.
Name	Name of the subscription (durable or shared name).

Heading	Description
	If this is an unshared non-durable subscription, this value is empty.
SS	Description of columns: • S - '+' if the subscription has a selector, '-' otherwise. • S - '+' if the subscription is shared, '-' otherwise.
Cons Count	The number of active consumers on this subscription. For an unshared non-durable subscription, the value is always 1. For a durable subscription, the value can be 0, meaning that there is no active consumer and the subscription is offline.
Pend Msgs	Total number of messages pending for the subscription.
Pend Size	Combined size of messages pending for the subscription. Value is rounded and shown in bytes, (K)ilobytes, (M)egabytes or (G)igabytes.
Uptime	The length of time, in hours, minutes, and seconds, since the subscription was created.

show topic

```
show topic topic-name
```

show topic (description of output fields)

Heading	Description
Topic	Full name of the topic.
Type	`dynamic`—created by a client `static`—configured by an administrator

Heading	Description
Properties	A list of property names that are set on the topic, and their values. For an index list of property names, see Destination Properties.
JNDI Names	A list of explicitly assigned JNDI names that refer to this topic.
Bridges	A list of bridges from this topic to other destinations.
Subscriptions	Number of subscriptions on this topic. (This count also includes durable subscriptions.)
Durable Subscriptions	The number of durable subscriptions on the topic.
Consumers	Number of active consumers on this topic. Note: When a durable consumer is offline, it is not included in the count reported here. However, if this command is performed on an EMS 7.x or earlier server, the count also includes offline durable consumers.
Durable Consumers	Number of active durable consumers on this topic. Note: When a durable consumer is offline, it is not included in the count reported here. However, if this command is performed on an EMS 7.x or earlier server, the count also includes offline durable consumers.
Pending Msgs	The total number of messages sent but not yet acknowledged by the consumer, followed by the number of persistent messages in parenthesis. These counts include copies sent to multiple subscribers.
Pending Msgs Size	Total size of all pending messages, followed by the size of all persistent messages in parenthesis.
The server accumulates the following statistics only when the administrator has enabled statistics. Otherwise these items are zero.	
Total Inbound	Cumulative count of all messages delivered to the topic.

Heading	Description
Msgs	
Total Inbound Bytes	Cumulative total of message size over all messages delivered to the topic.
Total Outbound Msgs	Cumulative count of messages consumed from the topic by consumers. Each consumer of a message increments this count independently of other consumers, so one inbound message results in *n* outbound messages (one per consumer).
Total Outbound Bytes	Cumulative total of message size over all messages consumed from the topic by consumers. Each consumer of a message contributes this total independently of other consumers.

show topics

```
show topics [pattern-name [notemp|static|dynamic] [first=n|next=n|last=n]]
```

If a *pattern-name* is not entered, this command shows a list of all topics.

If a *pattern-name* is entered (for example foo.* or foo.>) this command shows a list of topics that match that pattern. See Wildcards * and > for more information about using wildcards.

You can further refine the list of topics that match the pattern by using one of the following parameters:

- `notemp` — do not show temporary topics
- `static` — show only static topics
- `dynamic` — show only dynamic topics

When a *pattern-name* is entered, you can also cursor through the list of topics using one of the following commands, where *n* is whole number:

- `first=`*n* — show the first *n* topics
- `next=`*n* — show the next *n* topics

- `last=n` — show the next *n* topics and terminate the cursor

The cursor examines *n* topics and displays topics that match the *pattern-name*. Because it does not traverse the full list of topics, the cursor may return zero or fewer than *n* topics. To find all matching topics, continue to use next until you receive a `Cursor complete` message.

show topics (description of output fields)

Heading	Description
Topic Name	Name of the topic. If the name is prefixed with an asterisk (*), then the topic is temporary or was created dynamically. Properties of dynamic and temporary topics cannot be changed.
SNFGEIBCTM	Prints information on the topic properties in the order: (S)ecure (N)sender_name or sender_name_enforced (F)ailsafe (G)lobal (E)xport (I)mport (B)ridge (C)flowControl (T)race (M)ulticast The characters in the value section show: - Property not present + Property is present, and was set on the topic itself * Property is present, and was inherited from another topic Note that inherited properties cannot be removed.
Subs	Number of current subscriptions on the topic, including durable subscriptions. If this command is performed on an EMS 7.x or earlier server, the count reflects the number of *subscribers*, not the number of subscriptions.
Durs	Number of durable subscriptions on the topic. If this command is performed on an EMS 7.x or earlier server, the count reflects the number of durable *subscribers*, not the number of subscriptions.
All Msgs	The total number of messages sent but not yet acknowledged by the

Heading	Description
Msgs	consumer. This count includes copies sent to multiple subscribers. To see the count of actual messages (not multiplied by the number of topic subscribers) sent to all destinations, use the show server command.
Size	Total size of pending messages
Persistent Msgs	
Msgs	The total number of persistent messages sent but not yet acknowledged by the consumer
Size	Total size of pending persistent messages

For more information, see Destination Properties.

show transaction

```
show transaction XID
```

Shows a list of messages that were sent or received within the specified transaction. This command returns information on transactions in prepared, ended, and roll back states only. Transactions in a suspended or active state are not included.

show transaction (description of output fields)

Heading	Description
State	Transaction state: • **A** active • **E** ended • **R** rollback only • **P** prepared • **S** suspended Suspended transactions can be rolled back, but cannot be rolled forward

Heading	Description
	(committed).
Remaining time before timeout	The seconds remaining before the TX timeout is reached. For example, `3 sec`. This field is only applicable for transactions in State `ENDSUCCESS` or `ROLLBACKONLY`.
Messages to be consumed	
Message ID	The message ID of the message. null indicates the message ID could not be obtained or was disabled. If `track_message_ids` is not enabled, this field displays Disabled.
Type	The destination type to which the message was sent: • **`Q`** queue • **`T`** topic
Destination	The destination name to which the message was sent. null indicates that destination could not be found.
Consumer ID	The consumer ID of the Consumer that is consuming the message. Zero indicates that the consumer is offline.
Messages to be produced	
Message ID	The message ID of the message. null indicates the message ID could not be obtained or was disabled. If `track_message_ids` is not enabled, this field displays Disabled.
Type	The destination type to which the message was sent: • **`Q`** queue • **`T`** topic
Destination	The destination name to which the message was `sent. null` indicates that destination could not be found.
JMSTimestamp	The timestamp indicating the time at which the message was created.

show transactions

```
show transactions
```

Shows the XID for all client transactions that were created using the XA or MS DTC interfaces. Each row presents information about one transaction. The XID is the concatenation of the Format ID, GTrid Len, Bqual Len, and Data fields for a transaction. For example, if show transactions returns the row:

```
State  Format ID  GTrid Len  Bqual Len  Data
  E      0          6          2        branchid
```

then the XID is `0  6  2  branchid`.

Note that the spaces are required.

show transactions (description of output fields)

Heading	Description
State	Transaction state: • **A** active • **E** ended • **R** rollback only • **P** prepared • **S** suspended Suspended transactions can be rolled back, but cannot be rolled forward (committed).
Format ID	The XA transaction format identifier. `0 = OSI CCR` naming is used `>0 = some` other format is used -1 = NULL
GTrid Len	The number of bytes that constitute the global transaction ID.

Heading	Description
Bqual Len	The number of bytes that constitute the branch qualifier.
Data	The global transaction identifier (gtrid) and the branch qualifier (bqual).

show transport

```
show transport transport
```

Displays the configuration for the specified transport defined in `transports.conf`.

See Configure EMS Transports for TIBCO FTL and Configure EMS Transports for Rendezvous for details.

show transports

```
show transports
```

Lists all configured transport names in `transports.conf`.

show user

```
show user user-name
```

Shows user name and description. If no user name is specified, this command displays the currently logged in user.

For users defined externally, there is an asterisk in front of the user name.

show users

```
show users
```

Shows all users.

For users defined externally, there is an asterisk in front of the user name. Only currently connected external users are shown.

showacl admin

```
showacl admin
```

Shows all administrative permissions for all users and groups, but does not include administrative permissions on destinations.

showacl group

```
showacl group group-name [admin]
```

Shows all permissions set for a given group. Shows the group and the set of permissions. You can optionally specify admin to show only the administrative permissions for destinations or principals. Specifying showacl admin shows all administrative permissions for all users and groups (not including administrative permissions on destinations).

showacl queue

```
showacl queue queue-name [admin]
```

Shows all permissions set for a queue. Lists all entries from the `acl` file. Each entry shows the “grantee” (user or group) and the set of permissions. You can optionally specify `admin` to show only the administrative permissions for destinations or principals. Specifying `showacl admin` shows all administrative permissions for all users and groups (not including administrative permissions on destinations).

showacl topic

```
showacl topic topic-name [admin]
```

Shows all permissions set for a topic. Lists all entries from the `acl` file. Each entry shows the “grantee” (user or group) and the set of permissions. You can optionally specify `admin` to show only the administrative permissions for destinations or principals. Specifying `showacl admin` shows all administrative permissions for all users and groups (not including administrative permissions on destinations).

showacl user

```
showacl user user-name [admin | all | admin-all]
```

Shows the user and the set of permissions granted to the user for destinations and principals.

`showacl user` *username* — displays permissions granted directly to the user. (An administrator can use this form of the command to view own permissions, even without permissions to view any other user permissions.)

`showacl user` *username* `admin` — displays administrative permissions granted directly to the user.

`showacl user` *username* `all` — displays direct and inherited (from groups to which the user belongs) permissions.

`showacl user` *username* `admin-all` — displays all administrative permissions for a given user (direct and inherited)

Note: The output from this command displays inherited permissions prefixed with a '*'. Inherited permissions cannot be changed. An attempt to revoke an inherited permission for the principal user will not change the permission.

shutdown

```
shutdown
```

Shuts down currently connected server.

When issued to an EMS server that is using FTL stores, this command will shut down both the EMS server and the FTL server hosting it.

suspend route

```
suspend route route-name
```

Suspends outgoing messages to the named route.

Message flow can be recovered later using the command `resume route`.

time

```
time [on | off]
```

Specifying on places a timestamp before each command's output. By default, the timestamp is `off`.

timeout

```
timeout [seconds]
```

Show or change the current command timeout value. The timeout value is the number of seconds the Administration Tool will wait for a response from the server after sending a command.

By default, the timeout is 30 seconds. When `timeout` is entered with the optional *seconds* parameter, the timeout value is reset to the specified number of seconds. When entered without parameter, the current timeout value is returned.

transaction commit

```
transaction commit XID
```

Commits the transaction identified by the transaction ID. The transaction must be in the `ended` or `prepared` state. To obtain a transaction ID, issue the `show transactions` command, and cut and paste the XID into this command.

transaction rollback

```
transaction rollback XID
```

Rolls back the transaction identified by the transaction ID. The transaction must be in the `ended`, `rollback only`, or the `prepared` state. To obtain a transaction ID, issue the show transactions command, and cut and paste the XID into this command.

Note: Messages sent to a queue with `prefetch`=`none` and `maxRedelivery`=*number* properties are not received *number* times by an EMS application that receives in a loop and does an XA rollback after the XA prepare phase.

updatecrl

```
updatecrl
```

Immediately update the server's certificate revocation list (CRL).

whoami

```
whoami
```

Alias for the show user command to display the currently logged in user.

Configuration Files

This chapter describes configuring TIBCO Enterprise Message Service.

Location of Configuration Files

The installation process places a complete set of configuration files in `EMS_HOME/samples/config`. For deployment, we recommend copying files from this directory to a production configuration directory, and modifying those copies.

When selecting a production configuration directory, we recommend using a file system with regular backup commensurate with your need for reliability and disaster recovery. It is essential that the EMS server have both read and write privileges in the configuration directory.

Mechanics of Configuration

Configuration Files

The EMS server reads configuration files only once, when the server starts. It ignores subsequent changes to the configuration files. If you change a configuration file, use the `shutdown` command from the EMS Administration Tool to shut down the server and then restart the server as described in Running the EMS Server.

Administrative Requests

You can also change the server configuration with administrative requests, using either `tibemsadmin` (a command line tool), the Java or .NET administrative APIs, or TIBCO Administrator™ (a separate TIBCO product).

When the server validates and accepts an administrative request, it writes the change to the appropriate configuration file as well (overwriting any manual changes to that file). This

policy keeps configuration files current in case the server restarts (for example, in a fault-tolerant situation, or after a hardware failure).

Re-installing or updating EMS overwrites the files in the `bin/` and `samples/config/` directories. Do not use these directories to configure your deployment.

tibemsd.conf

The main configuration file controls the characteristics of the EMS server. This file is usually named `tibemsd.conf`, but you can specify another file name when starting the server.

You can find more information about starting the server in Running the EMS Server.

An example of the `tibemsd.conf` file is included in the *config-file-directory*`/cfmgmt/ems/data/` directory, where *config-file-directory* is specified during TIBCO Enterprise Message Service installation. You can edit this configuration file with a text editor. There are a few configuration items in this file that can be altered using the administration tool, but most configuration parameters must be set by editing the file (that is, the server does not accept changes to those parameters). See EMS Administration Tool for more information about using the administration tool.

Several parameters accept boolean values. In the description of the parameter, one specific set of values is given (for example, `enable` and `disable`), but all parameters that accept booleans can have the following values:

- `enable`, `enabled`, `true`, `yes`, `on`
- `disable`, `disabled`, `false`, `no`, `off`

Parameters that take multiple elements cannot contain spaces between the elements, unless the elements are enclosed in starting and ending double quotes. Parameters are limited to line lengths no greater than 256,000 characters in length.

The following table summarizes the parameters in `tibemsd.conf` according to category. The sections that follow provide more detail on each parameter.

Parameter	Description
Global System Parameters	
`always_exit_on_disk_error`	Enable or disable the server behavior to

Parameter	Description
	exit on any disk error.
authorization	Enable or disable server authorization.
auth_thread_count	Specifies the number of EMS server threads dedicated to authenticating incoming connections.
compliant_queue_ack	Guarantees that a message will not be redelivered after a client has successfully acknowledged its receipt from a routed queue.
disconnect_non_acking_consumers	Causes the server to review unacknowledged pending messages size and counts in consumers.
flow_control	Enable or disable flow control for destinations.
flow_control_only_with_active_consumer	Restore the flow control behavior that was enforced before release 8.4.
listen	Specifies the port on which the server is to listen for connections from clients.
max_msg_field_print_size	Limits the size of string fields in trace messages.
max_msg_print_size	Limits the size of the printed message of traced messages.
module_path	Specifies a directory or directories that contain external shared library files such as those of FTL, ActiveSpaces, and Rendezvous.
monitor_listen	Specifies the port on which the server is to

Parameter	Description
	listen for health check and Prometheus metrics requests.
network_thread_count	Specifies the number of network threads used by the EMS server.
npsend_check_mode	Specifies when the server is to provide confirmation upon receiving a NON_PERSISTENT message from a producer.
password	Password used to authenticate with other servers that have authorization enabled.
processor_ids	Specifies the processors to be used for network I/O traffic.
routing	Enable or disable routing functionality for this server.
secondary_monitor_listen	Specifies the port on which the server designated as secondary in a fault tolerant pair is to listen for health check and Prometheus metrics requests.
selector_logical_operator_limit	Limits the number of operators that the server reviews during selector evaluation.
server	Name of server.
startup_abort_list	Specifies conditions under which the server is to exit during its initialization sequence.
user_auth	Specifies the source of authentication information used to authenticate users attempting to access the EMS server.
xa_default_timeout	Specifies the TX timeout for XA transactions.

Parameter	Description
Storage File Parameter	
store	Specifies the directory in which the server stores data when using file-based stores.
Connection and Memory Parameters	
destination_backlog_swapout	Specifies the maximum number of messages per destination that are stored in the server before message swapping is enabled.
handshake_timeout	Specifies the amount of time that the EMS server waits for a connection to complete.
large_destination_count	Specifies the number of messages that an unbounded destination can gather before the server starts logging warnings about that destination's message count.
large_destination_memory	Specifies the size in memory that an unbounded destination can grow to before the server starts logging warnings about that destination's size.
max_client_msg_size	Sets a maximum size for incoming messages.
max_connections	Specifies the maximum number of simultaneous client connections to the server.
max_msg_memory	Specifies the maximum memory the server can use for messages.
msg_pool_block_size	Specifies the size of the pool to be pre-allocated by the server to store messages.

Parameter	Description
msg_swapping	Enable or disable message swapping.
prefetch_none_timeout_request_reply	Prevents the memory utilization of the server to grow in the context of a specific scenario that involves calling receive with a short timeout in a loop on a queue with prefetch set to none.
reserve_memory	Specifies the amount of memory to reserve for use in emergency situations.
socket_send_buffer_size	Sets the size of the send buffer used by clients when connecting to the EMS server.
socket_receive_buffer_size	Sets the size of the receive buffer used by clients when connecting to the EMS server.
Detecting Network Connection Failure Parameters	
active_route_connect_time	Specifies the interval at which an EMS server will attempt to connect or reconnect a route to another server.
client_heartbeat_server	Specifies the interval clients are to send heartbeats to the server.
clock_sync_interval	Periodically sends the EMS server's UTC time to clients.
server_timeout_client_connection	Specifies the period of time server will wait for a client heartbeat before terminating the client connection.
server_heartbeat_server	Specifies the interval this server is to send heartbeats to another server.
server_timeout_server_connection	Specifies the period of time this server will wait for a heartbeat from another server

Parameter	Description
	before terminating the connection to that server.
server_heartbeat_client	Specifies the interval this server is to send heartbeats to all of its clients.
client_timeout_server_connection	Specifies the period of time a client will wait for a heartbeat from the server before terminating the connection.
Fault Tolerance Parameters	
ft_active	Specifies the URL of the active server.
ft_heartbeat	Specifies the interval the active server is to send a heartbeat signal to the standby server to indicate that it is still operating.
ft_activation	Specifies the maximum length of time between heartbeat signals the standby server is to wait before assuming the active server has failed.
ft_reconnect_timeout	Specifies the maximum length of time the standby server is to wait for clients to reconnect after becoming the active server in a failover situation.
ft_ssl_identity	Specifies the server's digital certificate.
ft_ssl_issuer	Specifies the certificate chain member for the server.
ft_ssl_private_key	Specifies the server's private key.
ft_ssl_password	Specifies the password for private keys.
ft_ssl_trusted	Specifies the list of trusted certificates.

Parameter	Description
ft_ssl_verify_host	Specifies whether the fault-tolerant server should verify the other server's certificate.
ft_ssl_verify_hostname	Specifies whether the fault-tolerant server should verify the name in the CN field of the other server's certificate.
ft_ssl_expected_hostname	Specifies the name the server is expected to have in the CN field of the fault-tolerant server's certificate.
ft_ssl_ciphers	Specifies the cipher suites used by the server.
ft_oauth2_access_token_file	Specifies the path to a file containing the OAuth 2.0 access token to use to authenticate with the FT peer EMS server.
ft_oauth2_server_url	The HTTP(S) URL of the OAuth 2.0 authorization server that will issue the access tokens to be used to authenticate with the FT peer.
ft_oauth2_client_id	The OAuth 2.0 client ID to use when authenticating with the OAuth 2.0 authorization server. This parameter is required regardless of the grant type to be used for requesting access tokens.
ft_oauth2_client_secret	The OAuth 2.0 client secret to use when authenticating with the OAuth 2.0 authorization server. This parameter is required regardless of the grant type to be used for requesting access tokens.

Parameter	Description
ft_oauth2_grant_type	The grant type to use for requesting access tokens from the OAuth 2.0 authorization server. The type can be: • client_credentials–for Client Credentials Grant. • password–for Resource Owner Password Credentials Grant. If the password grant is specified, the server and password parameter values are used as the username and password for the grant. The default value of this parameter is client_credentials.
ft_oauth2_server_trust_file	A file containing one or more PEM-encoded public certificates that can be used to validate the OAuth 2.0 authorization server's identity. This parameter is only required if establishing an HTTPS connection to the authorization server.
ft_oauth2_disable_verify_hostname	If set, the EMS server will not verify the name in the CN field of the OAuth 2.0 authorization server's certificate. This parameter is optional. Hostname verification is performed by default.
ft_oauth2_expected_hostname	The name that the EMS server expects in the CN field of the OAuth 2.0 authorization server's certificate. This parameter is optional and only relevant when ft_oauth2_disable_

Parameter	Description
	`verify_hostname` is not set to `true`.
Message Tracking Parameters	
`track_message_ids`	Enable or disable message tracking by message ID.
`track_correlation_ids`	Enable or disable message tracking by correlation ID.
TIBCO FTL Transport Parameters	
`ftl_log_level`	Determines the trace level of FTL messages logged in the server when the EMS Server FTL trace item is enabled.
`ftl_trustfile`	Specifies the trust file for the EMS server to validate the TIBCO FTL server on a TLS connection.
`ftl_url`	Required. Specifies the URL at which the EMS server can connect to the TIBCO FTL server.
`ftl_username`	The username that the EMS server should use to authenticate itself when connecting to the TIBCO FTL server.
`ftl_password`	Specifies the password that the EMS server should use to authenticate itself when connecting to the TIBCO FTL server.
`ftl_oauth2_access_token_file`	Specifies the path to a file containing the OAuth 2.0 access token to use to authenticate with the FTL deployment.
`ftl_oauth2_server_url`	The HTTP(S) URL of the OAuth 2.0 authorization server that will issue the

Parameter	Description
	access tokens to be used to authenticate with the FTL deployment.
ftl_oauth2_client_id	The OAuth 2.0 client ID to use when authenticating with the OAuth 2.0 authorization server. This parameter is required regardless of the grant type to be used for requesting access tokens.
ftl_oauth2_client_secret	The OAuth 2.0 client secret to use when authenticating with the OAuth 2.0 authorization server. This parameter is required regardless of the grant type to be used for requesting access tokens.
ftl_oauth2_server_trust_file	A file containing one or more PEM-encoded public certificates that can be used to validate the OAuth 2.0 authorization server's identity. This parameter is only required if establishing an HTTPS connection to the authorization server.
ftl_oauth2_disable_verify_hostname	If set, the EMS server will not verify the name in the CN field of the OAuth 2.0 authorization server's certificate. This parameter is optional. Hostname verification is performed by default.
ftl_oauth2_expected_hostname	The name that the EMS server expects in the CN field of the OAuth 2.0 authorization server's certificate. This parameter is optional and only

Parameter	Description
	relevant when `ftl_oauth2_disable_verify_hostname` is not set to `true`.
`tibftl_transports`	Enable or disable the TIBCO FTL transports defined in `transports.conf` file.
TIBCO Rendezvous Transport Parameters	
`tibrv_transports`	Enable or disable the TIBCO Rendezvous transports defined in `transports.conf` file.
Tracing and Log File Parameters	
`client_trace`	Enable or disable client generation of trace output for opening or closing a connection, message activity, and transaction activity.
`console_trace`	Specifies the trace options for output to `stderr`.
`logfile`	Name and location of the server log file.
`log_trace`	Specifies the trace options on the file defined by the `logfile` parameter.
`logfile_max_count`	Specifies the maximum number of log files to be kept.
`logfile_max_size`	Specifies the maximum log file size before the log file is copied to a backup and then emptied.
`secondary_logfile`	Name and location of the server log file used by the server designated as secondary in a fault tolerant pair.
`trace_client_host`	Specifies whether the trace statements

Parameter	Description
	related to connections identify the host by its hostname, its IP address, or both.
Statistic Gathering Parameters	
server_rate_interval	Specifies the interval at which overall server statistics are averaged.
statistics	Enables or disables statistic gathering for producers, consumers, destinations, and routes.
rate_interval	Specifies the interval at which statistics for routes, destinations, producers, and consumers are averaged.
detailed_statistics	Specifies which objects should have detailed statistic tracking.
statistics_cleanup_interval	Specifies how long the server should keep detailed statistics if the destination has no activity.
max_stat_memory	Specifies the maximum amount of memory to use for detailed statistic gathering.
TLS Server Parameters	
ssl_server_ciphers	Specifies the cipher suites used by the server.
ssl_require_client_cert	Specifies if the server is to only accept TLS connections from clients that have digital certificates.
ssl_require_route_cert_only	Overrides ssl_require_client_cert to restrict requiring digital certificates to TLS connections only from routes.

Parameter	Description
ssl_use_cert_username	Specifies if a client's user name is to always be extracted from the CN field of the client's digital certificate.
ssl_cert_user_specname	Specifies a special username to identify which clients are to have their usernames taken from their digital certificates.
ssl_server_identity	Specifies the server's digital certificate.
ssl_server_key	Specifies the server's private key.
ssl_password	Specifies the password for private keys.
ssl_server_issuer	Specifies the certificate chain member for the server.
ssl_server_trusted	Specifies the list of CA root certificates the server trusts as issuers of client certificates.
ssl_crl_path	Specifies the pathname to the certificate revocation list (CRL) files.
ssl_crl_update_interval	Specifies the interval at which the server is to update its CRLs.
ssl_auth_only	Specifies whether the server allows clients to request the use of TLS only for authentication.
fips140-2	Enables the server for FIPS compliance.
Extensible Security Parameters	
jaas_config_file	Specifies the location of the JAAS configuration file used to run a custom authentication LoginModule.

Parameter	Description
jaas_login_timeout	Specifies the length of time, in milliseconds, that the server waits for the JAAS authentication module to execute and respond.
jaci_class	Specifies the name of the class that implements the extensible permissions interface.
jaci_timeout	Specifies the length of time, in milliseconds, that the server waits for the JACI permissions module to execute and respond.
security_classpath	Includes the JAR files and dependent classes used by the JAAS LoginModules and JACI modules.
OAuth 2.0 Authentication Parameters	
oauth2_server_validation_key	The PEM-encoded public key or JWKS to use for validating the OAuth 2.0 JWT access tokens presented by incoming connection requests.
oauth2_user_claim	The claim in the OAuth 2.0 access token that contains the EMS user to be associated with the incoming connection.
oauth2_group_claim	The claim in the OAuth 2.0 access token that contains the list of groups the EMS user belongs to. This parameter is optional. If not specified, the EMS server will assume that the user does not belong to any groups.
oauth2_audience	The expected value of the audience claim

Parameter	Description
	in OAuth 2.0 access tokens presented by incoming connections. This parameter is optional. If omitted, the EMS server will not validate the audience claim.
JVM Parameters	
`jre_library`	Enables the JVM in the EMS server.
`jre_option`	Passes command line options to the JVM at start-up.

Global System Parameters

always_exit_on_disk_error

Enable or disable the server behavior to exit on any disk error.

```
always_exit_on_disk_error = enable | disable
```

Defaults to disable.

authorization

Enable or disable server authorization.

```
authorization = enabled | disabled
```

Authorization is disabled by default. If you require that the server verify user credentials and permissions on secure destinations, you must enable this parameter.

See Enable Access Control for more information.

For example:

```
authorization = enabled
```

See Authentication and Permissions for more information about this parameter.

auth_thread_count

Specifies the number of EMS server threads dedicated to authenticating incoming connections.

```
auth_thread_count = threads
```

The `threads` count must be a positive integer. The default value is 4.

compliant_queue_ack

Guarantees that, once a client successfully acknowledges a message received from a routed queue, the message will not be redelivered. This is accomplished by the EMS server waiting until the message has been successfully acknowledged by the queue's home EMS server before sending the response to the client.

```
compliant_queue_ack = enable | disable
```

The `compliant_queue_ack` parameter is enabled by default. Because of the extra overhead incurred with compliant queue acknowledgments, you can disable this feature when performance is an issue. If compliant queue acknowledgment is disabled and a message is redelivered, the message's `JMSRedelivered` indicator will be set.

disconnect_non_acking_consumers

This parameter works in conjunction with the `maxbytes` and `maxmsgs` destination properties. In situations where consumers consume messages but do not acknowledge them, the messages are held in the server until they are confirmed. This can push the server above the set limits.

```
disconnect_non_acking_consumers = enabled | disabled
```

When enabled, `disconnect_non_acking_consumers` causes the server to check the number and size of pending messages sent to a consumer. If the `maxbytes` or `maxmsgs` limit is reached and the consumer has not acknowledged its messages, the server discards the messages sent to the consumer and disconnects the consumer's connection. This protects the server against applications that consume messages without ever acknowledging them.

Before enabling this property, ensure that the `maxbytes` and `maxmsgs` limits are set with reference to the `prefetch` setting, the size of the transaction (if transacted receive), or number of messages acknowledged when using client or explicit client acknowledgment mode. Otherwise the server may disconnect the consumer before it has a chance to acknowledge the messages.

When routes are deployed, all routed servers should use the same `disconnect_non_acking_consumers` setting. Additionally, if `maxbytes` or `maxmsgs` is set for a global destination, the same setting should be applied on all servers. The server does not discard or disconnect a routed consumer, since disconnecting the route may impact other well-behaved applications. Servers discard and disconnect their local consumers, which other servers involved are made aware of and discard messages for those remote consumers accordingly.

This parameter is disabled by default.

flow_control

Specifies whether flow control for destinations is enabled or disabled.

```
flow_control = enable | disable
```

By default, flow control is disabled. When flow control is enabled, the `flowControl` property on each destination specifies the target maximum storage for pending messages on the destination.

See Flow Control for more information about flow control.

flow_control_only_with_active_consumer

Restores the flow control behavior that was enforced before release 8.4. This property and the corresponding behavior are deprecated and will be removed in a future release.

```
flow_control_only_with_active_consumer = enable | disable
```

By default, this parameter is disabled. For more information, see Flow Control in the Absence of Consumers.

listen

Specifies the port on which the server is to listen for connections from clients.

```
listen=protocol://servername:port
```

For example:

```
listen=tcp://localhost:7222
```

If you are enabling TLS, for example:

```
listen=ssl://localhost:7222
```

You can use multiple listen entries if you have computers with multiple interfaces. For example:

```
listen=tcp://localhost:7222
listen=tcp://localhost:7224
```

If `localhost` is specified, or if the *servername* is not present, then the server uses every available interface. For example:

```
listen=tcp://7222
listen=ssl://7243
```

You can use an IP address instead of a host name. For example:

```
listen=tcp://192.168.10.107:7222
```

When specifying an IPv6 address, use square brackets around the address specification. For example:

```
listen=tcp://[2001:cafe::107]:7222
```

Note: This parameter is not supported when using FTL stores. See the Parameters Unsupported for FTL Stores section for details.

max_msg_field_print_size

Limits the size of string fields in trace messages. If a string field is larger than *size*, the field is truncated in the trace message.

```
max_msg_field_print_size = size [KB|MB|GB]
```

Specify signed 32-bit integer values as `KB`, `MB` or `GB`. The minimum permitted size is `1 KB`. By default, the field limit is `1 KB`.

max_msg_print_size

Limits the size of the printed message of traced messages. If the message is larger than *size*, the message is truncated.

```
max_msg_print_size = size [KB|MB|GB]
```

Specify signed 32-bit integer values as `KB`, `MB` or `GB`. The minimum permitted size is `8 KB`. By default, the field limit is `8 KB`.

module_path

```
module_path = shared-library-directory
```

where *shared-library-directory* is the absolute path to the directory containing any external library the server may need. This may include TIBCO FTL, ActiveSpaces, and Rendezvous libraries.

You can specify multiple directories (for example, to load TIBCO FTL, ActiveSpaces, and Rendezvous libraries). Separate paths using a colon (:) on UNIX platforms, or semicolon (;) on Windows platforms.

For example:

```
module_path = c:\tibco\ftl\6.10\bin
```

monitor_listen

Specifies the port on which the server is to listen for health check and Prometheus metrics requests.

```
monitor_listen=protocol://servername:port
```

For example:

```
monitor_listen = http://machine1:7220
```

If you are enabling TLS, for example:

```
monitor_listen = https://machine1:7220
```

When using `localhost` as the `servername`, the listen will only be accessible from the local machine. If you omit the `servername`, the listen will behave similarly to setting `localhost` in the server `listen` parameter.

For example:

```
monitor_listen = http://:7220
```

You can use an IP address instead of hostname.

For example:

```
monitor_listen = http://192.168.10.107:7220
```

When specifying an IPv6 address, use square brackets around the address specification.

For example:

```
monitor_listen = http://[2001:cafe::107]:7220
```

You can use only one `monitor_listen` entry at a time. For more information, see Server Health and Metrics.

Note: This parameter is not supported when using FTL stores. See the Parameters Unsupported for FTL Stores section for details.

network_thread_count

Specifies the number of network threads used by the EMS server.

```
network_thread_count = threads
```

The *threads* count can be any positive integer. The default value is 1.

When set, this parameter allows the EMS server to control the number of threads while still allowing the system administrator to control the thread affinity externally (for example, by using the Linux `taskset` command).

If you intend to set the thread affinity externally, we recommend that you avoid setting any thread affinity in the EMS server for either network traffic of stores.

The EMS server ignores this parameter if the `processor_ids` parameter is also specified.

npsend_check_mode

Specifies when the server is to provide confirmation upon receiving a `NON_PERSISTENT` message from a producer.

```
npsend_check_mode = [always | never | temp_dest | auth | temp_auth]
```

The `npsend_check_mode` parameter applies only to producers sending messages using `NON_PERSISTENT` delivery mode and non-transactional sessions.

Message confirmation has a great deal of impact on performance and should only be enabled when necessary. The circumstances in which a producer might want the server to send confirmation a `NON_PERSISTENT` message are:

- When `authorization` is enabled, so the producer can take action if permission to send the message is denied by the server.
- When sending to a temporary destination, so the producer can take action if the

message is sent to a temporary destination that has been destroyed.

- The message exceeded queue/topic limit (requires `rejectIncoming` policy for topics).
- Bridging of the message has failed.
- The server is out of memory or has encountered some other severe error.

The possible `npsend_check_mode` parameter modes are:

- `default` (no mode specified) - this means the server only provides confirmation of a `NON_PERSISTENT` message if `authorization` is enabled.
- `always` - the server always provides confirmation of a `NON_PERSISTENT` message.
- `never` - the server never provides confirmation of `NON_PERSISTENT` messages.
- `temp_dest` - the server provides confirmation of a `NON_PERSISTENT` message only when sending to a temporary destination.
- `auth` - the server provides confirmation of a `NON_PERSISTENT` message only if authorization was enabled when the connection was created.
- `temp_auth` - the server provides confirmation of a `NON_PERSISTENT` message if sending to a temporary destination or if authorization was enabled when the connection was created.

password

The password used when connecting to another EMS server that has authorization enabled.

```
password = password
```

For information on authorization between routed servers, see Routing and Authorization.

For information on authorization between fault tolerant server pairs, see Authorization and Fault-Tolerant Servers.

processor_ids

Setting this parameter causes the EMS Server to start as many network I/O threads as there are processor IDs specified in the list. Each network I/O thread is bound to the given processor ID, which means that the thread can execute only on that processor.

```
processor_ids = processor-id1,processor-id2,...
```

Note: Do not use this parameter if the default behavior provides sufficient throughput.

Specify the *processor-id* as an integer. Ask your system administrator for the valid processor IDs on the EMS Server host. Note that the IDs can be listed in any order. List IDs in a comma-separated list, with no spaces separating list items. For example:

```
processor_ids = 0,1,3,6
```

On startup, the parameter is parsed and the server refuses to start (regardless of the presence of the `startup_abort_list` parameter) if:

- The list is malformed. That is, if it contains invalid values such as non-numeric elements.
- The server is unable to bind a network I/O thread to a given processor ID. This can happen when the processor ID has been disabled, or the `tibemsd` process has been restricted by the system administrator to a set of processors that does not contain this processor ID. Additionally, the server cannot correctly bind the network I/O thread to the process ID if spaces are included in the parameter definition.

Note: Do not use hyper threading.

For instance, consider a machine with 24 processors, with 2 dies and processor IDs ranging from 0 to 5 and 12 to 17 on the first die, and 6 to 11 and 18 to 23 on the second die. In this example, you should specify processor IDs in either the 0 to 5 range, or the 6 to 11 range.

Specifying processor IDs 0 and 12 in the list would cause thrashing because two network I/O threads would be bound to the same processor (or core). Also, for optimal performance, processor IDs should be from the same die.

This parameter can be used in conjunction with the `stores.conf` parameter `processor_id`. For more information, see Performance Tuning.

routing

Enables or disables routing functionality for this server.

```
routing = enabled | disabled
```

For example:

```
routing = enabled
```

See Routes for more information about routing.

secondary_monitor_listen

Specifies the port on which the server designated as secondary in a fault tolerant pair is to listen for health check and Prometheus metrics requests.

```
secondary_monitor_listen = http://machine1:7220
```

If you are enabling TLS, for example:

```
secondary_monitor_listen = https://machine1:7220
```

If the `secondary_monitor_listen` is not set, the secondary server assumes the value of `monitor_listen`.

For more information, see `monitor_listen`.

> **Note:** This parameter is available only for JSON-configured EMS servers that are using file-based stores or grid stores.

selector_logical_operator_limit

Limits the number of operators that the server reviews during selector evaluation.

```
selector_logical_operator_limit = number
```

The server evaluates operators until reaching the specified *number* of false conditions. The server then stops evaluating further to protect itself from too many recursive evaluations. A very long selector clause, such as one including many OR conditions, can cause recursive selector evaluation and lead to a stack overflow in the EMS server.

number may be any positive integer. The default value is 5000. Zero is a special value, indicating no limit.

For example, if `selector_logical_operator_limit = 10` and the selector is:

```
a=1 or b=2 or c=3 or d=4 or e=5 or f=6 or g=7 or h=8 or i=9 or j=10 or
k=11 or l=12 or m=13 or n=14 or o=15 or p=16 or q=17 or r=18 or s=19 or
t=20 or u=21 or v=22 or w=23 or x=24 or y=25 or z=26
```

if the first 10 conditions are false, the server stops further evaluation.

server

Name of server.

```
server = serverName
```

Server names are limited to at most 64 characters, and may not include the dot character (.).

startup_abort_list

Specifies conditions that cause the server to exit during its initialization sequence.

```
startup_abort_list=[SSL,TRANSPORTS,CONFIG_FILES,CONFIG_ERRORS,
DB_FILES]
```

You may specify any subset of the conditions in a comma-separated list. The list cannot contain spaces between the elements, unless the elements are enclosed in starting and ending double quotes. If a space is included but not enclosed in quotation marks, the server ignores any conditions following the space.

Conditions that do not appear in the list are ignored by the server. The default is an empty list.

The conditions are:

- `SSL`—If TLS initialization fails, then it exits.
- `TRANSPORTS`—If any of the transports cannot be created as specified in the configuration files, then it exits.
- `CONFIG_FILES`—If any configuration file listed in `tibemsd.conf` does not exist, then it exits.
- `CONFIG_ERRORS`—If the server detects any errors while reading the config files, then it exits.

  Note that the `tibemsd` silently ignores any unknown parameters when it is started using the JSON configuration. For example, no configuration errors are thrown if the `tibemsd.json` file contains an obsolete parameter.
- `DB_FILES`—If the server cannot find one or more of its stores, then it exits. Stores include the default stores as well as any stores configured in the `stores.conf` configuration file.

  Note that if `DB_FILES` is *not* included in the `startup_abort_list` and the server cannot find a store, the server will create the missing store. For best results, do not include `DB_FILES` the first time a server is started, allowing it to create the stores. After initial startup or a major store configuration change (such as the addition of a new store), include `DB_FILES` in the list so that on restart the server will only start if all the configured stores are present.

user_auth

Specifies the authentication methods to be used by the EMS server.

```
user_auth = [local, jaas, oauth2]
```

This parameter can have one or more of the following values (separated by comma characters):

- `local`—authenticate incoming connection requests by validating the presented user credentials against locally defined user information (`users.conf` and `groups.conf`).
- `jaas`—authenticate incoming connection requests by validating the presented user credentials using a custom or provided JAAS authentication module, including LDAP support (see Extensible Authentication).

- `oauth2`—authenticate incoming connection requests by validating the presented OAuth 2.0 access token (see Authentication Using OAuth 2.0).

Each time the server receives a connection request, it attempts to authenticate it via each of the specified authentication methods in the order that this parameter specifies. The EMS server accepts successful authentication using any of the specified methods.

> **Note:** The `user_auth` setting does not affect authentication of the default administrator. The server always authenticates the admin user from the local configuration file. See Assigning a Password to the Administrator for more information.

xa_default_timeout

Specifies the default TX timeout, in seconds, for XA transactions. The default is 0, which specifies no timeout.

```
xa_default_timeout = seconds
```

The default timeout setting cannot be changed dynamically. However, you can specify a different transaction timeout for each individual XA resource using the API.

Storage File Parameter

The parameter described here configures file-based stores. For information about grid stores see Configuring and Deploying Grid Stores. For information about FTL stores, see Configuring and Deploying FTL Stores.

store

```
store = directory
```

Directory in which the server stores data files. For example:

```
store = /usr/tmp
```

Connection and Memory Parameters

The parameters described in the following topics affect the memory and connection management of the EMS server.

destination_backlog_swapout

Specifies the number of messages that may be stored in the server's memory before message swapping is enabled. The limit given is for each destination. For example, if the limit is 10,000 and you have three queues, the server can store up to 30,000 unswapped messages in memory.

```
destination_backlog_swapout = number
```

The specified *number* may be any positive value. When `destination_backlog_swapout` is `0`, the server attempts to immediately swap out the message.

By default, the limit for each destination is `1024` messages.

handshake_timeout

```
handshake_timeout = seconds
```

The amount of time that the EMS server waits for an incoming connection to complete depends on the `server_timeout_server_connection` and `server_timeout_client_connection` properties.

If either is specified, the connection handshake times out only after the duration mentioned in one of these properties. If both are specified, the largest of the two values is used. If neither is specified, you can set the period (in seconds) using `handshake_timeout`. The period specified must be a positive integer. If absent, the timeout defaults to 3 seconds. When the timeout is reached, the EMS server closes the connection and continues handling other clients.

The `handshake_timeout` server property, in addition to controlling the wait time for an incoming connection to complete, also controls:

- The amount of time that the server waits for an outgoing route connection to

complete;

- The amount of time the server waits for an incoming TLS connection to complete the TLS handshake. Note that this is independent from the wait time for the incoming TLS connection to complete.

large_destination_count

Specifies the number of messages that an unbounded destination (a destination without either of its `maxbytes` or `maxmsgs` properties set) can gather before the server starts logging warnings about that destination's message count.

```
large_destination_count = number
```

By default, `large_destination_count` is not set and the server establishes its own message count threshold. It can be set dynamically. Zero is a special value that disables the logging of the corresponding warning.

large_destination_memory

Specifies the size in memory that an unbounded destination (a destination without either of its `maxbytes` or `maxmsgs` properties set) can grow to before the server starts logging warnings about that destination's size.

```
large_destination_memory = size [KB|MB|GB]
```

By default, `large_destination_memory` is not set and the server establishes its own size threshold. It can be set dynamically. Zero is a special value that disables the logging of the corresponding warning.

max_client_msg_size

Maximum size allowed for an incoming message. This parameter setting instructs the server to reject incoming messages that are larger than the specified size limit.

```
max_client_msg_size = size [KB|MB|GB]
```

Specify whole numbers as KB, MB or GB. The maximum value is 2 GB. However, we recommend that the application programs use smaller messages, since messages approaching this maximum size will strain the performance limits of most current hardware and operating system platforms.

When omitted or zero, the EMS server accepts and attempts to process messages of any size.

Note: While using FTL stores, the default value of this parameter is set to 10 MB. The maximum value is still 2 GB.

max_connections

Maximum number of simultaneous client connections.

```
max_connections = number
```

Set to 0 to allow unlimited simultaneous connections.

max_msg_memory

Maximum memory the server can use for messages. This parameter lets you limit the memory that the server uses for messages, so server memory usage cannot grow beyond the system's memory capacity.

```
max_msg_memory = size [KB|MB|GB]
```

When msg_swapping is enabled, and messages overflow this limit, the server begins to swap messages from process memory to disk. Swapping allows the server to free process memory for incoming messages, and to process message volume in excess of this limit.

When the server swaps a message to disk, a small record of the swapped message remains in memory. If all messages are swapped out to disk, and their remains still exceed this memory limit, then the server has no room for new incoming messages. The server stops accepting new messages, and send calls in message producers result in an error. (This situation probably indicates either a very low value for this parameter, or a very high message volume.)

Specify units as KB, MB or GB. The minimum value is 8 MB. The default value of 0 (zero) indicates no limit.

For example:

```
max_msg_memory = 512MB
```

msg_pool_block_size

To lessen the overhead costs associated with malloc and free, the server pre-allocates pools of storage for messages. This parameter determines the behavior of these pools. Performance varies depending on operating system platform and usage patterns.

```
msg_pool_block_size = size
```

Note: Consult with your TIBCO support representative before using this parameter.

The *size* argument determines the approximate number of internal message structs that a block or pool can accommodate (not the number of bytes).

msg_pool_block_size instructs the server to allocate an *expandable* pool. Each time the server exhausts the pool, the server increases the pool by this size, as long as additional storage is available. The value may be in the range 32 to 65536.

When this parameter is not present, the default is msg_pool_block_size 128.

msg_swapping

This parameter enables and disables the message swapping feature (described above for max_msg_memory).

```
msg_swapping = enable | disable
```

The default value is enabled, unless you explicitly set it to disabled.

prefetch_none_timeout_request_reply

Prevents the memory utilization of the server to grow in the context of the following scenario.

The combination of a client consuming from a queue with prefetch set to `none` and calling receive with a short timeout in a loop can cause the memory utilization of the server to grow significantly. This can happen when the receive timeout is so short that the server doesn't have a chance to deliver a message to the consumer before being asked again, causing a backup of receive requests in the server.

To prevent this from happening, enable this parameter.

```
prefetch_none_timeout_request_reply = enable | disable
```

Defaults to disable.

> **Note:** Consult with your TIBCO support representative before using this parameter.

reserve_memory

When `reserve_memory` is non-zero, the EMS server allocates a block of memory for use in emergency situations to prevent the EMS server from being unstable in low memory situations.

```
reserve_memory = size
```

When the server process exhausts memory resources, it disables clients and routes from producing new messages, and frees this block of memory to allow consumers to continue operation (which tends to free memory).

The EMS server attempts to reallocate its reserve memory once the number of pending messages in the server has dropped to 10% of the number of pending messages that were in the server when it experienced the allocation error. If the server successfully reallocates memory, it begins accepting new messages.

The `reserve_memory` parameter only triggers when the EMS server has run out of memory and therefore is a reactive mechanism. The appropriate administrative action when an EMS server has triggered release of reserve memory is to drain the majority of the messages by

consuming them and then to stop and restart the EMS server. This allows the operating system to reclaim all the virtual memory resources that have been consumed by the EMS server. A trace option, `MEMORY`, is also available to help show what the server is doing during the period when it is not accepting messages.

Specify *size* in units of MB. When non-zero, the minimum block is 16MB. When absent, the default is zero.

> **Note:** There are a variety of limits that the user can set to prevent the EMS server from storing excessive messages, which can lead to situations where the EMS server runs out of memory.
> These include global parameters, such as `max_msg_memory`, as well as destination properties such as `maxbytes`. These limits should be used to prevent the `reserve_memory` mechanism from triggering.

socket_send_buffer_size

Sets the size (in bytes) of the send buffer used by clients when connecting to the EMS server.

```
socket_send_buffer_size = size [KB|MB|GB]
```

The specified *size* may be:

- any number greater than `512`
- `0` to use the default buffer size
- `-1` to skip the call for the specified buffer
- Optionally, specify units of `KB`, `MB`, or `GB` for units. If no units are specified, the file size is assumed to be in bytes.

When omitted, the server skips the call for the specified buffer. In this case, the operating system's auto-tuning controls buffering.

socket_receive_buffer_size

Sets the size (in bytes) of the receive buffer used by clients when connecting to the EMS server.

```
socket_receive_buffer_size = size [KB|MB|GB]
```

The specified *size* may be:

- any number greater than `512`
- `0` to use the default buffer size
- `-1` to skip the call for the specified buffer
- Optionally, specify units of `KB`, `MB`, or `GB` for units. If no units are specified, the file size is assumed to be in bytes.

When omitted, the server skips the call for the specified buffer. In this case, the operating system's auto-tuning controls buffering.

Detecting Network Connection Failure Parameters

This feature lets servers and clients detect network connection failures quickly. When these parameters are absent, or this feature is disabled, `tibemsd` closes a connection only upon the operating system notification.

active_route_connect_time

Specifies the interval (in seconds) at which an EMS server attempts to connect or reconnect a route to the another server. The default is 2 seconds.

```
active_route_connect_time = interval
```

client_heartbeat_server

In a server-to-client connection, clients send heartbeats to the server at this interval (in seconds).

```
client_heartbeat_server = interval
```

The `client_heartbeat_server` parameter must be specified when a `server_timeout_client_connection` is set.

The `client_heartbeat_server` interval should be no greater than one third of the `server_timeout_client_connection` limit.

This setting also ensures that garbage collection occurs on the connection. Collection is triggered by incoming messages and heartbeats. If the size of messages can vary widely or there is not a steady stream of message traffic, can use this parameter to ensure that collection occurs.

When omitted or zero, `client_heartbeat_server` is disabled.

clock_sync_interval

Periodically send the EMS server's Coordinated Universal Time (UTC) time to clients. This allows EMS clients to update their offset.

```
clock_sync_interval = seconds
```

The time specified, in seconds, determines the interval at which clock sync commands are sent from the server to its clients.

When omitted or zero, the EMS server sends the offset time only when the EMS client connects to the server. If `clock_sync_interval` is `-1`, the offset is never sent, not even on connect. Clients do not adjust their time values to match the server time.

server_timeout_client_connection

In a server-to-client connection, if the server does not receive a heartbeat for a period exceeding this limit (in seconds), it closes the connection.

```
server_timeout_client_connection = limit
```

We recommend setting this value to approximately 3 times the heartbeat interval, as it is specified in `client_heartbeat_server`.

Note: If you do not set the `client_heartbeat_server` parameter when a `server_timeout_client_connection` is specified, a configuration error is generated during startup. If `CONFIG_ERRORS` is part of the `startup_abort_list`, the server will not start. If not, the error is printed but the server starts, and clients will be disconnected after `server_timeout_client_connection` seconds.

Zero is a special value, which disables heartbeat detection in the server (although clients still send heartbeats).

server_heartbeat_server

In a server-to-server connection, this server sends heartbeats at this interval (in seconds). The two servers can be connected either by a route, or as a fault-tolerant pair.

```
server_heartbeat_server = interval
```

When using FTL stores, this parameter only affects server-to-server route connections. It has no effect on the behavior of fault-tolerance.

server_timeout_server_connection

In a server-to-server connection, if this server does not receive a heartbeat for a period exceeding this limit (in seconds), it closes the connection. This parameter applies to connections from other routes and to the standby server connection.

```
server_timeout_server_connection = limit
```

We recommend setting this value to approximately 3.5 times the heartbeat interval of the other server. When the other server or the network are heavily loaded, or when client programs send very large messages, we recommend a larger multiple.

Note: In a fault-tolerant configuration, the `server_timeout_server_connection` parameter has no effect on the standby server following a failover. The standby server activates only after the timeout set by the `ft_activation` parameter.

When using FTL stores, this parameter only affects server-to-server route connections. It has no effect on the behavior of fault-tolerance.

server_heartbeat_client

In a server-to-client connection, the server sends heartbeats to all clients at this interval (in seconds).

```
server_heartbeat_client = interval
```

When omitted or zero, the default is 5 seconds.

client_timeout_server_connection

In a server-to-client connection, if a client does not receive a heartbeat for a period exceeding this limit (in seconds), it closes the connection.

```
client_timeout_server_connection = limit
```

We recommend setting this value to approximately 3.5 times the heartbeat interval.

Zero is a special value, which disables heartbeat detection in the client (although the server still sends heartbeats).

Fault Tolerance Parameters

See Fault Tolerance for more information about these parameters.

The fault tolerance parameters that begin with the prefix `ft_ssl` are used to secure communications between pairs of fault tolerant servers. See TLS for additional information about this process.

Aside from `ft_reconnect_timeout`, none of the parameters in this section are applicable when using FTL stores. See Fault-Tolerance with FTL Stores for more details.

ft_active

Specifies the URL of the active server. If this server can connect to the active server, it will act as a standby server. If this server cannot connect to the active server, it will become the active server.

```
ft_active = URL
```

ft_heartbeat

Specifies the interval (in seconds) the server is to send a heartbeat signal to its peer to indicate that it is still operating.

```
ft_heartbeat = seconds
```

Default is 3 seconds.

ft_activation

Activation interval (maximum length of time between heartbeat signals) which indicates that server has failed.

```
ft_activation = seconds
```

Set in seconds: default is 10. This interval should be set to at least twice the heartbeat interval.

For example:

```
ft_activation = 60
```

See the `server_timeout_server_connection` parameter for more information on heartbeats.

ft_reconnect_timeout

The amount of time (in seconds) that a standby server waits for clients to reconnect (after it becomes the active server in a failover situation).

```
ft_reconnect_timeout = seconds
```

If a client does not reconnect within this time period, the server removes its state from the shared state files. The `ft_reconnect_timeout` time starts once the server has fully recovered the shared state, so this value does not account for the time it takes to recover the store files.

The default value of this parameter is 60.

ft_ssl_identity

The path to a file that contains the certificate in one of the supported formats. The supported formats are PEM, DER, or PKCS#12. A DER format file can only contain the certificate; it cannot contain both the certificate and a private key.

```
ft_ssl_identity = pathname
```

See File Names for Certificates and Keys for more information on file types for digital certificates.

ft_ssl_issuer

Certificate chain member for the server. Supply the entire chain, including the CA root certificate. The server reads the certificates in the chain in the order they are presented in this parameter.

```
ft_ssl_issuer = chain_member
```

The certificates must be in PEM, DER, PKCS#7, or PKCS#12 format. A DER format file can only contain a single certificate; it cannot contain a certificate chain. See File Names for Certificates and Keys for more information on file types for digital certificates.

ft_ssl_private_key

The server's private key. If it is included in the digital certificate in `ft_ssl_identity`, then this parameter is not needed.

```
ft_ssl_private_key = key
```

This parameter supports private keys in the following formats: PEM, DER, PKCS#12.

You can specify the actual key in this parameter, or you can specify a path to a file that contains the key. See File Names for Certificates and Keys for more information on file types for digital certificates.

ft_ssl_password

Private key or password for private keys.

```
ft_ssl_password = password
```

You can set passwords by way of the `tibemsadmin` tool. When passwords are set with this tool, the password is obfuscated in the configuration file. See EMS Administration Tool for more information about using `tibemsadmin` to set passwords.

ft_ssl_trusted

List of trusted certificates. This sets which Certificate Authority certificates should be trusted as issuers of the client certificates.

```
ft_ssl_trusted = trusted_certificates
```

The certificates must be in PEM, DER, or PKCS#7 format. You can either provide the actual certificates, or you can specify a path to a file containing the certificate chain. If using a DER format file, it can contain only a single certificate, not a certificate chain.

See File Names for Certificates and Keys for more information on file types for digital certificates.

ft_ssl_verify_host

Specifies whether the fault-tolerant server should verify the other server's certificate.

```
ft_ssl_verify_host = enabled | disabled
```

The values for this parameter are *`enabled`* or *`disabled`*.

By default, this parameter is `enabled`, signifying the server should verify the other server's certificate.

When this parameter is set to *`disabled`*, the server establishes secure communication with the other fault-tolerant server, but does not verify the server's identity.

ft_ssl_verify_hostname

Specifies whether the fault-tolerant server should verify the name in the CN field of the other server's certificate.

```
ft_ssl_verify_hostname = enabled | disabled
```

The values for this parameter are *enabled* and *disabled*. By default, this parameter is enabled, signifying the fault-tolerant server should verify the name of the connected host or the name specified in the `ft_ssl_expected_hostname` parameter against the value in the server's certificate. If the names do not match, the connection is rejected.

When this parameter is set to *disabled*, the fault-tolerant server establishes secure communication with the other server, but does not verify the server's name.

ft_ssl_expected_hostname

Specifies the name the server is expected to have in the CN field of the fault-tolerant server's certificate.

```
ft_ssl_expected_hostname = serverName
```

If this parameter is not set, the expected name is the hostname of the server.

This parameter is used when the `ft_ssl_verify_hostname` parameter is set to enabled.

ft_ssl_ciphers

Specifies the cipher suites used by the server; each suite in the list is separated by a colon (:). This parameter can use the OpenSSL name for cipher suites or the longer, more descriptive names.

```
ft_ssl_ciphers = cipherSuite
```

See Specify Cipher Suites for more information about the cipher suites available in EMS and the OpenSSL names and longer names for the cipher suites.

ft_oauth2_access_token_file

Specifies the path to a file containing an OAuth 2.0 access token to use to authenticate with the fault-tolerant peer EMS server.

```
ft_oauth2_access_token_file = pathname
```

If an access token is provided using this parameter, the EMS server will not attempt to obtain access tokens from an OAuth 2.0 authorization server even if `ft_oauth2_server_url` and other relevant parameters are set.

ft_oauth2_server_url

Specifies the HTTP or HTTPS URL of the OAuth 2.0 authorization server from which the EMS server will obtain access tokens for authenticating with its fault-tolerant peer EMS server.

```
ft_oauth2_server_url = http://hostname:port
```

If connecting to a secure OAuth 2.0 authorization server:

```
ft_oauth2_server_url = https://hostname:port
```

ft_oauth2_client_id

The OAuth 2.0 client ID to use when authenticating with the OAuth 2.0 authorization server. This parameter and `ft_oauth2_client_secret` are both required in order to obtain access tokens from the authorization server, regardless of the grant type to be used.

```
ft_oauth2_client_id = client_id
```

ft_oauth2_client_secret

The OAuth 2.0 client secret to use when authenticating with the OAuth 2.0 authorization server. This parameter and ft_oauth2_client_id are both required in order to obtain access tokens from the authorization server, regardless of the configured grant type.

```
ft_oauth2_client_secret = client_secret
```

ft_oauth2_grant_type

The grant type to use for requesting access tokens from the OAuth 2.0 authorization server.

The accepted values are:

- `client_credentials`–for Client Credentials grant.
- `password`–for Resource Owner Password Credentials grant.

```
ft_oauth2_grant_type = client_credentials|password
```

If using the resource owner password credentials grant type, the username and password included in the grant are the server name and server password configured through the `server` and `password` parameters.

If this parameter is not set, the client credentials grant type is used by default.

ft_oauth2_server_trust_file

Specifies the path to a file containing one or more PEM-encoded public certificates that can be used to validate a secure OAuth 2.0 authorization server's identity.

```
ft_oauth2_server_trust_file = trust_file
```

This parameter is only required if an HTTPS URL was specified for `ft_oauth2_server_url`.

ft_oauth2_disable_verify_hostname

When this parameter is enabled, the EMS server will not verify the hostname in the CN field of the OAuth 2.0 authorization server’s certificate.

```
ft_oauth2_disable_verify_hostname = enabled|disabled
```

This parameter is disabled by default.

ft_oauth2_expected_hostname

The name that the EMS server expects in the CN field of the OAuth 2.0 authorization server's certificate.

```
ft_oauth2_expected_hostname = hostname
```

If this parameter is not set, the expected name is the hostname of the authorization server.

This parameter is not relevant when the `ft_oauth2_disable_verify_hostname` parameter is set to `true`.

Message Tracking Parameters

The parameters described in the following topics configure the message tracking behavior of the EMS server.

track_message_ids

Tracks messages by message ID. Default is disabled.

```
track_message_ids = enabled | disabled
```

Enabling this parameter allows you to display messages using the `show message`*messageID* command in the administration tool.

track_correlation_ids

Tracks messages by correlation ID. Disabled by default.

```
track_correlation_ids = enabled | disabled
```

Enabling this parameter allows you to display messages using the `show messages`*correlationID* command in the administration tool.

TIBCO FTL Transport Parameters

The parameters listed here enable the EMS server to connect to a TIBCO FTL server using transports configured in the `transports.conf` file.

> **Note:** The EMS server creates a single FTL event queue that is used for all EMS transports for FTL configured in the `transports.conf` file.

For more information, see Interoperation with TIBCO FTL.

ftl_log_level

Optional. Determines the trace level of FTL messages logged in the server when the EMS Server FTL trace item is enabled.

```
ftl_log_level = level
```

When absent, the `ftl_log_level` defaults to `warn`.

For more details, see the TIBCO FTL documentation on logging.

ftl_trustfile

Optional. Specifies the trust file for the EMS server to validate the FTL server on a TLS connection.

```
ftl_trustfile = file name
```

The trust file must be the same as that used by other FTL clients to validate the FTL server.

> **Note:** For the trust file to be used, the `ftl_url` must start with `https://` instead of `http://`.

> **Note:** If the `ftl_url` starts with `https://` but a `ftl_trustfile` is not provided, a warning is logged that the connection is *not secure*.

Sets the `com.tibco.ftl.trust.type` and `com.tibco.ftl.trust.file` properties in the

following way:

- If `ftl_url` starts with `https://` and `ftl_trustfile` exists:

  `com.tibco.ftl.trust.type` is set to `TIB_REALM_HTTPS_CONNECTION_USE_SPECIFIED_TRUST_FILE` and

  `com.tibco.ftl.trust.file` is set to the contents of `ftl_trustfile`

- If `ftl_url` starts with `https://` and `ftl_trustfile` does not exist:

  `com.tibco.ftl.trust.type` is set to `TIB_REALM_HTTPS_CONNECTION_TRUST_EVERYONE` and `com.tibco.ftl.trust.file` is not set

  If the environment variable `TIB_FTL_TRUST_FILE` is set, the content of `ftl_trustfile` is ignored, and the content of the environment variable is used for validating the FTL server (if `ftl_url` starts with `https://`). For more details, see the TIBCO FTL documentation on realms.

ftl_url

Required. Specifies the URL at which the EMS server can connect to the TIBCO FTL server.

```
ftl_url = URL
```

For example, `ftl_url=http://localhost:5633`.

For more details, see the TIBCO FTL documentation on realms.

ftl_username

Optional. The username that the EMS server should use to authenticate itself when connecting to the TIBCO FTL server.

```
ftl_username = user
```

Sets the `com.tibco.ftl.client.username` property. For more details, see the TIBCO FTL documentation on realms.

When authenticating with FTL using OAuth 2.0, this parameter determines the grant type used to request access tokens from the OAuth 2.0 authorization server. If `ftl_username` is not set, the Client Credentials Grant type is used. If `ftl_username` is set, the

Resource Owner Password Credentials Grant type is used, with `ftl_username` and `ftl_password` serving as the username and password credentials.

ftl_password

Optional. The password that the EMS server should use to authenticate itself when connecting to the TIBCO FTL server. Note that the password can be stored in a mangled form.

```
ftl_password = password
```

Sets the `com.tibco.ftl.client.userpassword` property. For more details, see the TIBCO FTL documentation on realms.

This parameter may need to be set if authenticating with FTL using OAuth 2.0. See `ftl_username` for details.

ftl_oauth2_access_token_file

Specifies the path to a file containing an OAuth 2.0 access token to use to authenticate with the FTL deployment.

```
ftl_oauth2_access_token_file = pathname
```

If an access token is provided via this parameter, the EMS server will always use that access token for authentication with the FTL deployment. The server will not attempt to obtain access tokens from an OAuth 2.0 provider even if `ftl_oauth2_server_url` and other relevant parameters are set.

ftl_oauth2_server_url

Specifies the HTTP or HTTPS URL of the OAuth 2.0 authorization server from which the EMS server will obtain access tokens for authenticating with the FTL deployment.

```
ftl_oauth2_server_url = http://hostname:port
```

If connecting to a secure OAuth 2.0 authorization server:

```
ftl_oauth2_server_url = https://hostname:port
```

ftl_oauth2_client_id

The OAuth 2.0 client ID to use when authenticating with the OAuth 2.0 authorization server. This parameter and `ftl_oauth2_client_secret` are both required in order to obtain access tokens from the authorization server, regardless of the configured grant type.

```
ftl_oauth2_client_id = client_id
```

ftl_oauth2_client_secret

The OAuth 2.0 client secret to use when authenticating with the OAuth 2.0 authorization server. This parameter and `ftl_oauth2_client_id` are both required in order to obtain access tokens from the authorization server, regardless of the grant type to be used.

```
ftl_oauth2_client_secret = client_secret
```

ftl_oauth2_server_trust_file

Specifies the path to a file containing one or more PEM-encoded public certificates for validating a secure OAuth 2.0 authorization server's identity.

```
ftl_oauth2_server_trust_file = trust_file
```

This parameter is only required if an HTTPS URL was specified for `ftl_oauth2_server_url`.

ftl_oauth2_disable_verify_hostname

When this parameter is enabled, the EMS server will not verify the hostname in the CN field of the OAuth 2.0 authorization server’s certificate.

```
ftl_oauth2_disable_verify_hostname = enabled|disabled
```

This parameter is disabled by default.

ftl_oauth2_expected_hostname

The name that the EMS server expects in the CN field of the OAuth 2.0 authorization server's certificate.

```
ftl_oauth2_expected_hostname = hostname
```

If this parameter is not set, the expected name is the hostname of the authorization server.

This parameter is not relevant when the `ftl_oauth2_disable_verify_hostname` parameter is set to `true`.

tibftl_transports

Specifies whether the TIBCO FTL transports defined in `transports.conf` are enabled or disabled.

```
tibftl_transports = enabled | disabled
```

Unless you explicitly set this parameter to enabled, the default value is *`disabled`*—that is, all transports are *`disabled`* and will neither send messages to external systems nor receive messages from them.

Rendezvous Transport Parameters

For more information, see Interoperation With TIBCO Rendezvous.

tibrv_transports

Specifies whether TIBCO Rendezvous transports defined in `transports.conf` are enabled or disabled.

```
tibrv_transports = enabled | disabled
```

Unless you explicitly set this parameter to enabled, the default value is *disabled*—that is, all transports are disabled and will neither send messages to external systems nor receive message from them.

Tracing and Log File Parameters

See Monitor Server Activity, for more information about these parameters.

> **Note:** When using FTL stores, only the `client_trace`, `console_trace`, and `trace_client_host` parameters are applicable. All other log file related parameters are unsupported. See Logging With FTL Stores for details.

client_trace

Administrators can trace a connection or group of connections. When this property is enabled, the server instructs each client to generate trace output for opening or closing a connection, message activity, and transaction activity. This type of tracing does not require restarting the client program.

```
client_trace = {enabled|disabled} [target=location]
               [user|connid|clientid=value]
```

Each client sends trace output to *location*, which may be either `stderr` (the default) or `stdout`.

> **Note:** You can also direct client tracing output to a file, using the `tibems_SetTraceFile`, `Tibjms.setTraceFile` and `Tibems.SetTraceFile` in the C, Java and .NET libraries, respectively.

The default behavior is to trace all connections. You can specify either `user`, `connid` or `clientid` to selectively trace specific connections. The *value* can be a user name or ID (as appropriate).

Setting this parameter using the administration tool does not change its value in the configuration file `tibemsd.conf`; that is, the value does not persist across server restarts unless you set it in the configuration file.

console_trace

Sets trace options for output to `stderr`. The possible values are the same as for `log_trace`. However, console tracing is independent of log file tracing.

```
console_trace = traceOptions
```

If `logfile` is defined, you can stop console output by specifying:

```
console_trace=-DEFAULT
```

Note: Important error messages (and some other messages) are always output, overriding the trace settings.

logfile

Name and location of the server log file.

```
logfile = pathname
```

If the *pathname* contains spaces, it must be enclosed in double quotes.

By default, the logfile specified here is used by both servers in fault tolerant pair. Optionally, a JSON-configured server pair can set the `secondary_logfile` parameter to direct the server designated as secondary to write to a different file.

log_trace

Sets the trace preference on the file defined by the `logfile` parameter. If `logfile` is not set, the values have no effect.

```
log_trace = traceOptions
```

The value of this parameter is a comma-separated list of trace options. For a list of trace options and their meanings, see Server Tracing Options,.

You may specify trace options in three forms:

- plain A trace option without a prefix character replaces any existing trace options.
- + A trace option preceded by + adds the option to the current set of trace options.
- - A trace option preceded by - removes the option from the current set of trace options.

The following example sets the trace log to only show messages about access control violations.

```
log_trace=ACL
```

The next example sets the trace log to show all default trace messages, in addition to TLS messages, but ADMIN messages are not shown.

```
log_trace=DEFAULT,-ADMIN,+SSL
```

logfile_max_count

Specifies the maximum number of log files to be kept.

```
logfile_max_count = integer
```

Specify any number greater than 2.

When 0 or not specified, there is no limit to the number of log files kept.

logfile_max_size

Specifies the recommended maximum log file size before the log file is rotated. Set to 0 to specify no limit. Use KB, MB, or GB for units (if no units are specified, the file size is assumed to be in bytes).

```
logfile_max_size = size [KB|MB|GB]
```

The server periodically checks the size of the current log file. If it is greater than the specified size, the file is copied to a backup and then emptied. The server then begins writing to the empty log file until it reaches the specified size again.

Backup log files are named sequentially and stored in the same directory as the current log.

secondary_logfile

Name and location of the server log file used by the secondary EMS server in a fault tolerant pair. The EMS server designated as primary in the pair writes to the file specified by the `logfile` parameter.

```
secondary_logfile = pathname
```

If the `secondary_logfile` parameter is not set, the secondary server assumes the value of `logfile`.

If the *pathname* contains spaces, it must be enclosed in double quotes.

For more information, see `logfile`.

Note: This parameter is available only for JSON-configured EMS servers using file-based stores or grid stores.

trace_client_host

Trace statements related to connections can identify the host by its hostname, its IP address, or both. When absent, the default is `hostname`. The `both_with_port` option displays the ephemeral port used on the host as well as the IP address and hostname.

```
trace_client_host = [hostname|address|both|both_with_port]
```

Statistic Gathering Parameters

See Monitor Server Activity, for more information about these parameters.

server_rate_interval

Sets the interval (in seconds) over which overall server statistics are averaged.

```
server_rate_interval = seconds
```

This parameter can be set to any positive integer greater than zero.

Overall server statistics are always gathered, so this parameter cannot be set to zero. By default, this parameter is set to 1.

Setting this parameter allows you to average message rates and message size over the specified interval.

statistics

Enables or disables statistic gathering for producers, consumers, destinations, and routes. By default this parameter is set to disabled.

```
statistics = enabled | disabled
```

Disabling statistic gathering resets the total statistics for each object to zero.

rate_interval

Sets the interval (in seconds) over which statistics for routes, destinations, producers, and consumers are averaged.

```
rate_interval = seconds
```

By default, this parameter is set to 3 seconds. Setting this parameter to zero disables the average calculation.

detailed_statistics

Specifies which objects should have detailed statistic tracking.

```
detailed_statistics = NONE | [PRODUCERS,CONSUMERS,ROUTES]
```

Detailed statistic tracking is only appropriate for routes, producers that specify no destination, or consumers that specify wildcard destinations. When detailed tracking is enabled, statistics for each destination are kept for the object.

Setting this parameter to NONE disabled detailed statistic tracking. You can specify any combination of PRODUCERS, CONSUMERS, or ROUTES to enable tracking for each object. If you specify more than one type of detailed tracking, separate each item with a comma.

For example:

```
detailed_statistics = NONE
```

Turns off detailed statistic tracking.

```
detailed_statistics = PRODUCERS,ROUTES
```

Specifies detailed statistics should be gathered for producers and routes.

statistics_cleanup_interval

Specifies how long (in seconds) the server should keep detailed statistics if the destination has no activity.

```
statistics_cleanup_interval = seconds
```

This is useful for controlling the amount of memory used by detailed statistic tracking. When the specified interval is reached, statistics for destinations with no activity are deleted.

max_stat_memory

Specifies the maximum amount of memory to use for detailed statistic gathering.

```
max_stat_memory = size [KB|MB|GB]
```

If no units are specified, the amount is in bytes, otherwise you can specify the amount using KB, MB, or GB as the units.

Once the maximum memory limit is reached, the server stops collecting detailed statistics. If statistics are deleted and memory becomes available, the server resumes detailed statistic gathering.

TLS Server Parameters

See TLS Protocol for more information about these parameters.

ssl_server_ciphers

Specifies the cipher suites used by the server; each suite in the list is separated by a colon (:). This parameter must follow the OpenSSL cipher string syntax.

```
ssl_server_ciphers = cipherSuites
```

For example, you can enable the cipher suites for security level 2 with the following setting:

```
ssl_server_ciphers = @SECLEVEL=2
```

See Specify Cipher Suites for more information about the cipher suites available in EMS and the syntax for specifying them in this parameter.

ssl_require_client_cert

If this parameter is set to `enable`, the server only accepts TLS connections from clients that have digital certificates. Connections from clients without certificates are denied.

```
ssl_require_client_cert = enable | disable
```

If this parameter is set to *`disable`*, then connections are accepted from clients that do not have a digital certificate.

Whether this parameter is set to *`enable`* or *`disable`*, clients that do have digital certificates are always authenticated against the certificates supplied to the `ssl_server_trusted` parameter.

The default value is `disable`.

ssl_require_route_cert_only

This parameter overrides the `ssl_require_client_cert` parameter.

```
ssl_require_route_cert_only = enable | disable
```

If `ssl_require_route_cert_only` is set to *`enable`*, the server requires a digital certificate only for TLS connections coming from routes,
regardless of the value of `ssl_require_client_cert`. In this case, the server does not require a digital certificate for TLS connections coming from clients and from its fault-tolerant peer.

If `ssl_require_route_cert_only` is set to *`disable`*, whether the server requires a digital certificate for TLS connections coming from all sources (routes, clients, and fault-tolerant peer) still depends on the value of `ssl_require_client_cert`.

The default value is *`disable`*.

ssl_use_cert_username

If this parameter is set to enable, a client's user name is always extracted from the CN field of the client's digital certificate, if the digital certificate is specified.

```
ssl_use_cert_username = enable | disable
```

If a different username is provided through the connection factory or API calls, then that username is discarded. Only the username from the CN is used.

The CN field is either a username, an email address, or a web address.

Note: When `ssl_use_cert_username` is enabled, the username given by the CN becomes the only valid username. Any permissions associated with a different username, for example one assigned with an API call, are ignored.

ssl_cert_user_specname

This parameter is useful if clients are required to supply a username, but you wish to designate a special username to use when the client's username should be taken from the

client's digital certificate.

```
ssl_cert_user_specname = username
```

For example, you may wish all clients to specify their username when logging in. This means the `ssl_use_cert_username` parameter would be set to `disable`. The username is supplied by the user, and not taken from the digital certificate. However, you may wish one username to signify that the client logging in with that name should have the name taken from the certificate. A good example of this username would be `anonymous`. All clients logging in as `anonymous` will have their user names taken from their digital certificates.

The value specified by this parameter is the username that clients will use to log in when the username should be taken from their digital certificate. A good example of the value of this parameter would be `anonymous`.

Also, the value of this parameter is ignored if `ssl_use_cert_username` is set to `enable`, in which case all client `usernames` are taken from their certificates. This parameter has no effect for users that have no certificate.

ssl_server_identity

The server's digital certificate in PEM, DER, or PKCS#12 format. You can specify the path to a file that contains the certificate in one of the supported formats.

```
ssl_server_identity = certificate
```

This parameter must be specified if any TLS ports are listed in the `listen` parameter.

PEM and PKCS#12 formats allow the digital certificate to include the private key. If these formats are used and the private key is part of the digital certificate,
then setting `ssl_server_key` is optional.

For example:

```
ssl_server_identity = certs/server.cert.pem
```

ssl_server_key

The server's private key. If it is included in the digital certificate in `ssl_server_identity`, then this parameter is not needed.

```
ssl_server_key = private_key
```

This parameter supports private keys in the following formats: PEM, DER, PKCS#8, PKCS#12.

You must specify a path to a file that contains the key.

ssl_password

Private key or password for private keys. This password can optionally be specified on the command line when `tibemsd` is started.

```
ssl_password = password
```

If TLS is enabled, and the password is not specified with this parameter or on the command line, `tibemsd` will ask for the password upon startup.

You can set passwords by way of the `tibemsadmin` tool. When passwords are set with this tool, the password is obfuscated in the configuration file. See EMS Administration Tool for more information about using `tibemsadmin` to set passwords.

Note: Because connection factories do not contain the `ssl_password` (for security reasons), the EMS server uses the password that is provided in the "create connection" call for user authentication. If the create connection password is different from the `ssl_password`, the connection creation will fail.

ssl_server_issuer

Certificate chain member for the server. The server reads the certificates in the chain in the order they are presented in this parameter.

```
ssl_server_issuer = chain_member
```

The same certificate can appear in multiple places in the certificate chain.

The certificates must be in PEM, DER, PKCS#7, or PKCS#12 format. A DER format file can only contain a single certificate, it cannot contain a certificate chain.

See File Names for Certificates and Keys for more information on file types for digital certificates.

ssl_server_trusted

List of CA root certificates the server trusts as issuers of client certificates.

```
ssl_server_trusted = certificates
```

Specify only CA root certificates. Do not include intermediate CA certificates.

The certificates must be in PEM, DER, or PKCS#7 format. You can either provide the actual certificates, or you can specify a path to a file containing the certificate chain. If using a DER format file, it can contain only a single certificate, not a certificate chain.

For example:

```
ssl_server_trusted = certs\CA1_root.pem
ssl_server_trusted = certs\CA2_root.pem
```

See File Names for Certificates and Keys for more information on file types for digital certificates.

ssl_crl_path

A non-null value for this parameter activates the server's certificate revocation list (CRL) feature.

```
ssl_crl_path = pathname
```

The server reads CRL files from this directory. The directory should contain only CRL files. If other files are located in the *pathname* directory, TLS initialization will fail.

ssl_crl_update_interval

The server automatically updates its CRLs at this interval (in hours).

```
ssl_crl_update_interval = hours
```

When this parameter is absent, the default is 24 hours.

ssl_auth_only

When enabled, the server allows clients to request the use of TLS only for authentication (to protect user passwords).

```
ssl_auth_only = enable | disable
```

For an overview of this feature, see TLS Authentication Only.

When disabled, the server ignores client requests for this feature. When absent, the default value is `disabled`.

fips140-2

When `true`, the EMS server is enabled to run in FIPS 140-2 compliant mode. When `false` or excluded, the server is not FIPS compliant.

```
fips140-2 = true | false
```

For more information, see Enable FIPS Compliance.

HTTPS Server Parameters

See TLS Protocol for more information about these parameters.

monitor_ssl_identity

The digital certificate used for the Server Health and Metrics HTTPS listen in PEM, DER, or PKCS#12 format. You can specify the path to a file that contains the certificate in one of the supported formats.

```
monitor_ssl_identity = certificate
```

If this parameter is not specified, the identity file configured for the server's TLS listen is used in its place. If neither are specified, the server will fail to start up.

PEM, PKCS#12 formats allow the digital certificate to include the private key. If these formats are used and the private key is part of the digital certificate, then setting `monitor_ssl_key` is optional.

For example:

```
monitor_ssl_identity = certs/server.cert.pem
```

monitor_ssl_key

The private key used for the Server Health and Metrics HTTPS listen. If it is included in the digital certificate in `monitor_ssl_identity`, then this parameter is not needed.

```
monitor_ssl_key = private_key
```

If this parameter is not specified, the private key file configured for the server's TLS listen is used in its place. If neither are specified, the server will fail to start up.

This parameter supports private keys in the following formats: PEM, DER, PKCS#8, PKCS#12.

You must specify a path to a file that contains the key.

monitor_ssl_password

Password for the private key used for the Server Health and Metrics HTTPS listen.

```
monitor_ssl_password = password
```

If this parameter is not specified, the password for the private key file configured for the server's TLS listen is used in its place. If neither are specified, the server will fail to start up.

monitor_ssl_trusted

List of CA root certificates the server trusts as issuers of client certificates. This list only applies to incoming connections on the Server Health and Metrics HTTPS listen.

```
monitor_ssl_trusted = certificates
```

If this parameter is not specified, an attempt is made to use the list from the server's TLS listen.

Specify only CA root certificates. Do not include intermediate CA certificates.

The certificates must be in PEM or DER format. You can either provide the paths to certificates as individual `monitor_ssl_trusted` entries, or you can specify a path to a file containing the certificate chain. If using a DER format file, it can contain only a single certificate, not a certificate chain.

For example:

```
monitor_ssl_trusted = certs\CA1_root.pem
monitor_ssl_trusted = certs\CA2_root.pem
```

See File Names for Certificates and Keys for more information on file types for digital certificates.

OAuth 2.0 Parameters

See Authentication Using OAuth 2.0 for more information on using OAuth 2.0 in EMS.

oauth2_server_validation_key

Specifies the path to a file containing the PEM-encoded public key or JSON Web Key Set (JWKS) to use for validating the signature of an OAuth 2.0 JWT access token presented by an incoming connection request.

```
oauth2_server_validation_key = public_key|JWKS
```

oauth2_user_claim

Specifies the claim in the OAuth 2.0 access token that contains the EMS user to associate with the incoming connection request.

```
oauth2_user_claim = claim_name
```

oauth2_group_claim

Specifies the claim in the OAuth 2.0 access token that contains the list of groups that the EMS user belongs to.

```
oauth2_group_claim = claim_name
```

This parameter is optional. If not specified, the EMS server will assume that the incoming connection is associated with an EMS user that does not belong to any groups.

oauth2_audience

Specifies the expected value of the ‘aud’ claim in the access token presented by the incoming connection request.

```
oauth2_audience = audience
```

The EMS server can be configured to accept multiple audience values. For example:

```
oauth2_audience = ems_aud1
oauth2_audience = ems_aud2
```

This parameter is optional. If not specified, the JWT audience will not be verified.

Extensible Security Parameters

The extensible security feature allows you to write your own authentication and permissions modules for the server.

For more information on this feature, see Extensible Security.

jaas_config_file

Specifies the location of the JAAS configuration file used by the EMS server to run a custom authentication `LoginModule`.

```
jaas_config_file = file-name
```

For more information, see Loading the LoginModule in the EMS Server.

This parameter is required to enable the extensible security feature for authentication.

For example:

```
jaas_config_file = jaas.conf
```

jaas_login_timeout

Specifies the length of time, in milliseconds, that the EMS server will wait for the JAAS authentication module to execute and respond.

```
jaas_login_timeout = milliseconds
```

This timeout is used each time the server passes a username and password to the LoginModule. If the module does not return a response, the server denies authentication.

This parameter is optional. If it is not included, the default timeout is 10000 milliseconds.

For example:

```
jaas_login_timeout = 250
```

jaci_class

Specifies the name of the class that implements the extensible permissions interface.

```
jaci_class = class-name
```

The class must be written using the Java Access Control Interface (JACI). For more information about writing a custom application using JACI to grant permissions, see Permissions Module.

For example:

```
jaci_class = com.userco.auth.CustomAuthorizer
```

jaci_timeout

Specifies the length of time, in milliseconds, that the EMS server will wait for the JACI permissions module to execute and respond.

```
jaci_timeout = milliseconds
```

This timeout is used each time the server passes a destination, username, and action to the permissions module. If the module does not return a response, the server denies authorization.

This parameter is optional. If it is not included, the default timeout is 10000 milliseconds.

For example:

```
jaci_timeout = 250
```

security_classpath

Includes the JAR files and dependent classes used by the JAAS LoginModules and JACI modules.

```
security_classpath = classpath
```

This parameter is required to enable the extensible security feature for authentication and the extensible security feature for granting permissions.

For example:

```
security_classpath = .:/usr/local/custom/user_jaci_plugin.jar
```

JVM Parameters

These parameters enable and configure the Java virtual machine (JVM) in the EMS server.

For more information on how the JVM works in EMS, see Enable the JVM.

jre_library

Enables the JVM in the EMS server, where *path* is the absolute path to the JRE shared library file that is installed with the JRE.

```
jre_library = path
```

Depending on your platform, this could be `jvm.dll`, `libjvm.so`, `libjvm.dylib`, and so forth.

If this parameter is not included, the JVM is disabled by default.

If the *path* contains any spaces, the path must be enclosed in quotation marks.

For example:

```
jre_library =
"C:\Program Files\Java\jdk1.8.0_121\jre\bin\server\jvm.dll"
```

jre_option

Passes command line options to the JVM at start-up.

```
jre_option = JVMoption
```

The `jre_option` parameter can be used to define Java system properties, which are used by applications running in the JVM, such as extensible security modules.

You can use multiple `jre_option` entries in order to pass more than one options to the JVM. Permitted values for *JVMoption* include most JVM options that are defined by Sun Microsystems.

For example, this restricts the maximum heap size of the JVM to 256 megabytes:

```
jre_option = -Xmx256m
```

Using Other Configuration Files

In addition to the main configuration file, there are several other configuration files used for various purposes.

These configuration files can be edited by hand, but you can also use the administration tool or the administration APIs to modify some of these files. See EMS Administration Tool for more information about using the administration tool.

Configuration File	Description
`acl.conf`	Defines EMS access control lists.
`bridges.conf`	Defines bridges between destinations.
`durables.conf`	Defines static durable subscribers.
`factories.conf`	Defines the connection factories stored as JNDI names on the EMS server.
`groups.conf`	Defines EMS groups.
`jaas.conf`	Locates and loads the LoginModule.
`queues.conf`	Defines EMS Queues.
`routes.conf`	Defines routes between this and other EMS servers

Configuration File	Description
stores.conf	Defines the locations of the stores where the EMS server will store messages.
tibrvcm.conf	Defines the TIBCO Rendezvous certified messaging (RVCM) listeners for use by topics that export messages to a tibrvcm transport.
topics.conf	Defines EMS Topics.
transports.conf	Defines transports used by EMS to import messages from or export messages to TIBCO FTL and Rendezvous.
users.conf	Defines EMS users.

acl.conf

This file defines all permissions on topics and queues for all users and groups.

The format of the file is:

```
TOPIC=topic USER=user PERM=permissions
TOPIC=topic GROUP=group PERM=permissions
QUEUE=queue USER=user PERM=permissions
QUEUE=queue GROUP=group PERM=permissions
ADMIN USER=user PERM=permissions
ADMIN GROUP=group PERM=permissions
```

Parameter Name	Description
TOPIC	Name of the topic to which you wish to add permissions.
QUEUE	Name of the queue to which you wish to add permissions.
ADMIN	Specifies that you wish to add administrator permissions.

Parameter Name	Description
USER	Name of the user to whom you wish to add permissions.
GROUP	Name of the group to which you wish to add permissions. The designation `all` specifies a predefined group that contains all users.
PERM	Permissions to add. The permissions which can be assigned to queues are `send`, `receive` and `browse`. The permissions which can be assigned to topics are `publish`, `subscribe` and `durable` and `use_durable`. The designation `all` specifies all possible permissions. For information about these permissions, refer to When Permissions Are Checked and Inheritance of Permissions. Administration permissions are granted to users to perform administration activities. See Administrator Permissions for more information about administration permissions.

Example

```
ADMIN USER=sys-admins PERM=all
TOPIC=foo USER=user2 PERM=publish,subscribe
TOPIC=foo GROUP=group1 PERM=subscribe
```

bridges.conf

This file defines bridges between destinations.

See Destination Bridges for more information about destination bridges.

The format of the file is:

```
[destinationType:destinationName] # mandatory -- include brackets
destinationType=destinationToBridgeTo1 [selector="msg-selector"]
destinationType=destinationToBridgeTo2 [selector="msg-selector"]
...
```

The *destination-name* can be any specific destination or a wildcard pattern to match multiple destinations.

Parameter Name	Description
destinationType	The type of the destination. That is, `topic` or `queue`.
destinationName	The name of the destination.
destinationToBridgeTo	One or more names of destinations to which to create a bridge.
`selector="msg-selector"`	This optional property specifies a message selector to limit the messages received by the bridged destination. For detailed information about message selector syntax, see the 'Message Selectors' section in description for the Message class in *TIBCO Enterprise Message Service Java API Reference*.

Example

```
[topic:myTopic1]
  topic=myTopic2
  queue=myQueue1
```

durables.conf

This file defines static durable subscribers.

The file consists of lines with either of these formats:

```
topic-name  durable-name
  [route]
  [clientid=id]
  [nolocal]
  [selector="msg-selector"]
```

Parameter Name	Description
topic-name	The topic of the durable subscription.
durable-name	The name of the durable subscriber.
`route`	When present, the subscriber is another server, and the *durable-name* is the name of that server. When this property is present, no other properties are permitted.
`clientid=`*`id`*	The client ID of the subscriber's connection.
`nolocal`	When present, the subscriber does not receive messages published from its own connection.
`selector=` *`msg-selector`*	When present, this selector narrows the set of messages that the durable subscriber receives. For detailed information about message selector syntax, see the 'Message Selectors' section in description for the `Message` class in *TIBCO Enterprise Message Service Java API Reference*.

Example

```
topic1 dName1
topic2 dName2 clientid=myId,nolocal
topic3 dName3 selector="urgency in ('high','medium')"
topic4 Paris route
```

Conflicting Specifications

When the server detects an conflict between durable subscribers, it maintains the earliest specification, and outputs a warning. Consider these examples:

- A static specification in this file takes precedence over a new durable dynamically created by a client.
- An existing durable dynamically created by a client takes precedence over a new

static durable defined by an administrator.

- A static durable subscription takes precedence over a client attempting to dynamically unsubscribe (from the same topic and durable name).

Conflict can also arise because of wildcards. For example, if a client dynamically creates a durable subscriber for topic `foo.*`, and an administrator later attempts to define a static durable for topic `foo.1`, then the server detects this conflict and warns the administrator.

Configuration

To configure durable subscriptions in this file, we recommend using the create durable command in the `tibemsadmin` tool; see `create durable`.

If the `create durable` command detects an existing dynamic durable subscription with the same topic and name, it promotes it to a static subscription, and writes a specification to the file `durables.conf`.

factories.conf

This file defines the connection factories for the internal JNDI names.

The file consists of factory definitions with this format:

```
[factory-name] # mandatory -- square brackets included
  type = generic|xageneric|topic|queue|xatopic|xaqueue|
  url = url-string
  metric = connections | byte_rate
  clientID = client-id
  [connect_attempt_count|connect_attempt_delay|
  connect_attempt_timeout|reconnect_attempt_count|
  reconnect_attempt_delay|reconnect_attempt_timeout = value]
  [tls-prop = value]*
```

Parameter Name	Description
Mandatory Parameters	
These parameters are required. Values given to these parameters cannot be overridden using API calls.	

Parameter Name	Description
[*factory-name*]	[*factory-name*] is the name of the connection factory. Note that the square brackets [ ] DO NOT indicate that the *factory-name* is optional; they must be included around the name.
`type`	Type of the connection factory. The value can be: • `generic`: Generic connection • `xageneric`: Generic XA connection • `topic`: Topic connection • `queue`: Queue connection • `xatopic`: XA topic connection • `xaqueue`: XA queue connection

`url`	This string specifies the servers to which this factory creates connections:
	
	• A single URL specifies a unique server. For example:
	
	`tcp://host1:8222`
	
	• A pair of URLs separated by a comma specifies a pair of fault-tolerant servers. For example:
	
	`tcp://host1:8222,tcp://backup1:8222`
	
	• A set of URLs separated by vertical bars specifies a load balancing among those servers. For example:
	
	`tcp://a:8222|tcp://b:8222|tcp://c:8222`
	
	• You can combine load balancing with fault tolerance. For example:
	
	`tcp://a1:8222,tcp://a2:8222|tcp://b1:8222,tcp://b2:8222`
	
	This example defines two servers (a and b), each of which has a

Parameter Name	Description
	fault-tolerant backup. The client program checks the load on the active a server and the active b server, and connects to the one that has the smaller load. If it cannot connect to one of the active servers, the client attempts to connect to the standby server. For example, if it cannot connect to `b1`, it connects to `b2`. The connection URL cannot exceed 1000 characters. For cautionary information, see Load Balancing.
Optional Parameters These parameters are optional. The values of these parameters can be overridden using API calls.	
`metric`	The factory uses this metric to balance the load among a group of servers: • `connections`—Connect to the server with the fewest client connections. • `byte_rate`—Connect to the server with the lowest byte rate. Byte rate is a statistic that includes both inbound and outbound data. When this parameter is absent, the default metric is `connections`. For cautionary information, see Load Balancing.
`clientID`	The factory associates this client ID string with the connections that it creates. The client ID cannot exceed 255 characters in length.
`connect_attempt_count`	A client program attempts to connect to its server (or in fault-tolerant configurations, it iterates through its URL list) until it establishes its first connection to an EMS server. This property determines the maximum number of iterations. When absent, the default is 2.
`connect_attempt_delay`	When attempting a first connection, the client sleeps for this interval (in milliseconds) between attempts to connect to its server (or in fault-tolerant configurations, iterations through its URL list). When

Parameter Name	Description
	absent, the default is 500 milliseconds.
`connect_attempt_timeout`	When attempting to connect to the EMS server, you can set this connection timeout period to abort the connection attempt after a specified period of time (in milliseconds).
`reconnect_attempt_count`	After losing its server connection, a client program configured with more than one server URL attempts to reconnect, iterating through its URL list until it re-establishes a connection with an EMS server. This property determines the maximum number of iterations. When absent, the default is 4.
`reconnect_attempt_delay`	When attempting to reconnect, the client sleeps for this interval (in milliseconds) between iterations through its URL list. When absent, the default is 500 milliseconds.
`reconnect_attempt_timeout`	When attempting to reconnect to the EMS server, you can set this connection timeout period to abort the connection attempt after a specified period of time (in milliseconds).
tls-prop	TLS properties for connections that this factory creates. For further information on TLS, refer to Configure a Connection Factory.

Example

```
[north_america]
  type = topic
  url = tcp://localhost:7222,tcp://server2:7222
  clientID = "Sample Client ID"
  ssl_verify_host = disabled
```

Configuration

To configure connection factories in this file, we recommend using the `tibemsadmin` tool; see `create factory`.

Load Balancing

> **Warning: Do not specify load balancing in situations with durable subscribers.**
>
> If a client program that a creates durable subscriber connects to server A using a load-balanced connection factory, then server A creates and supports the durable subscription. If the client program exits and restarts, and this time connects to server B, then server B creates and supports a new durable subscription—however, pending messages on server A remain there until the client reconnects to server A.
>
> **Do not specify load balancing when your application requires strict message ordering.**
>
> Load balancing chooses from among multiple servers, which inherently violates strict ordering.

groups.conf

This file defines all groups. The format of the file is:

```
group-name1:"description"
    user-name1
    user-name2
group-name2:"description"
    user-name1
    user-name2
```

Group Parameters

Parameter Name	Description
group-name	The name of the group. The group name cannot exceed 255 characters in length.
description	A string describing the group.
user-name	One or more users that belong to the group.

Example

```
administrators: "TIBCO Enterprise Message Service administrators"
   admin
   Bob
```

jaas.conf

This file directs the TIBCO Enterprise Message Service server to the JAAS LoginModule.

See Loading the LoginModule in the EMS Server for more information about the `jaas.conf` file.

queues.conf

This file defines all queues.

The format of the file is:

```
[jndi-name1, jndi-name2, ...]queue-name property1, property2, ...
```

> **Note:** Note that, while including JNDI names is optional, the square brackets [ ] must be included around JNDI names if they are included. For more information about setting JNDI names, see `create jndiname`.

For example, you might enter:

```
test store=mystore,secure,prefetch=2
```

Only queues listed in this file or queues with names that match the queues listed in this file can be created by the applications (unless otherwise permitted by an entry in `acl.conf`). For example, if queue `foo.*` is listed in this file, queues `foo.bar` and `foo.baz` can be created by the application.

Properties of the queue are inherited by all static and dynamic queues with matching names. For example, if `test.*` has the property `secure`, then `test.1` and `test.foo` are also

secure. For information on properties that can be assigned to queues, see Destination Properties.

For further information on the inheritance of queue properties, refer to Wildcards * and > and Inheritance of Properties.

> **Note:** In the sample file, a > wildcard at the beginning of the file allows the applications to create valid queues with any name. A > at the beginning of the queue configuration file means that name-matching is not required for creation of queues.

Restrictions and rules on queue names are described in Destination Name Syntax.

routes.conf

This file defines routes between this TIBCO Enterprise Message Service server and other TIBCO Enterprise Message Service servers.

> **Note:** Routes may only be configured administratively, using the administration tool (see Using the EMS Administration Tool), or the administration APIs (see `com.tibco.tibjms.admin.RouteInfo` in the online documentation). Directly editing the `routes.conf` file causes errors.

The format of the file is:

```
[route-name] # mandatory -- square brackets included.
  url=url-string
  zone_name=zone_name
  zone_type=zone_type
  topic_prefetch=value
  [selector]*
  [tls-prop = value]*
  [oauth2-prop = value]*
```

Parameter Name	Description
[*route-name*]	[*route-name*] is the name of the passive server (at the other

Parameter Name	Description
	end of the route); it also becomes the name of the route. Note that the square brackets [ ] DO NOT indicate that the *route-name* is an option; they must be included around the name.
url	The URL of the server to and from which messages are routed.
zone_name	The route belongs to the routing zone with this name. When absent, the default value is `default_mhop_zone`. You can set this parameter when creating a route, but you cannot subsequently change it. For further information, see these sections: • Zone • Configure Routes and Zones
zone_type	The zone type is either `1hop` or `mhop`. When omitted, the default value is `mhop`. You can set this parameter when creating a route, but you cannot subsequently change it. The EMS server will refuse to start up if the zone type in the `routes.conf` file does not match the zone type already created in the `$sys.meta` file that holds the shared state for the primary and secondary server.
topic_prefetch	A `prefetch` value for the route. Setting a `prefetch` at the route level allows you to assign larger values for WAN routing functions. If `topic_prefetch` is not set, the route uses the `prefetch` value specified for the destination. If a `topic_prefetch` is set for the route and a different `prefetch` is set for the destination, the `topic_prefetch` value overrides the destination `prefetch`. See the `prefetch` destination property for valid settings.

Parameter Name	Description
selector	Topic selectors (for `incoming_topic` and `outgoing_topic` parameters) control the flow of topics along the route. For syntax and semantics, see Selectors for Routing Topic Messages.
tls-prop	TLS properties for this route. For further information on TLS, refer to TLS Parameters for Routes.
oauth2-prop	OAuth 2.0 authentication properties for this route. These properties will be used to obtain the OAuth 2.0 access tokens that will be used to authenticate with the server at the other end of the route. For further information on OAuth 2.0, refer to Authentication and Authorization for Routes.

Example

```
[test_route_2]
url = tcp://server2:7222
ssl_verify_host = disabled
```

stores.conf

This file defines the locations of stores where the EMS server will store messages or metadata (if the default `$sys.meta` definition is overridden). You can configure one or many stores in the `stores.conf` file.

Store parameters specific to file-based stores are described here. Grid store parameters are described in Configuring and Deploying Grid Stores and FTL store parameters are described in Configuring and Deploying FTL Stores.

The format of the file is:

```
[store_name] # mandatory -- square brackets included
  type=file
  file=name
  file_destination_defrag=size
  [file_minimum=value]
  [file_truncate=value]
  [mode=async|sync]
  [processor_id=processor id]
```

Parameter Name	Description
[*store_name*]	[*store_name*] is the name that identifies this store file configuration. Note that the square brackets [ ] DO NOT indicate that the *store_name* is an option; they must be included around the name.
type	Identifies the store type. This parameter is required for all store types. The type can be: • `file` — for file-based stores. • `as` — for grid stores. • `ftl` — for FTL stores.
file	The filename that will be used when creating this store file. This parameter is required for file stores. For example, `mystore.db`. The location for this file can be specified using absolute or relative path names. If no path separators are present, the file will be saved in the location specified by the `store` parameter in the `tibemsd.conf` file, if any is specified there.
mode	The mode determines whether messages will be written to the store synchronously

Parameter Name	Description
	or asynchronously. Mode is either: • async — the server stores messages in this file using asynchronous I/O calls. • sync — the server stores messages in this file using synchronous I/O calls. When absent, the default for file-based stores is async.
processor_id	When specified, the EMS Server binds the storage thread of this store to the specified processor. Do not use this parameter if the default behavior provides sufficient throughput. If no processor ID is specified for a store, the store is not bound to a specific processor. Specify the *processor-id* as an integer. This parameter has similar requirements, limitations, and benefits as the processor_ids parameter in tibemsd.conf. For use guidelines, see Performance Tuning.
file_destination_defrag	This parameter specifies a maximum batch size used by the destination defrag feature. Destination defrag improves store file performance by maintaining contiguous space for new messages, while improving server read performance. When persistent

Parameter Name	Description
	pending messages begin to accumulate in a queue, messages are grouped into a batch that is re-written to disk. Messages are written close together, allowing the server to read them more efficiently when later delivering the messages to consumers. Specify *size* in `bytes`, `KB`, `MB` or `GB`. The *size* should be set to a size that is known to be acceptable for the disk where the store points to. For instance, if it is set to 2MB, your disk must be able to write a 2MB batch efficiently. If `file_destination_defrag` is zero or absent, the destination defrag feature is disabled.
file_minimum	This parameter preallocates disk space for the store file. Preallocation occurs when the server first creates the store file. You can specify units of `MB` or `GB`. Zero is a special value, which specifies no minimum preallocation. Otherwise, the value specified must be greater than `4MB`. For example: `file_minimum = 32MB` If `file_truncate` is set to true, the `file_minimum` parameter prevents the EMS server from truncating the file below the set size. When this parameter is absent, there is no default minimum preallocation.

Parameter Name	Description
file_truncate	Determines whether the EMS server will occasionally attempt to truncate the store file, relinquishing unused disk space. When `file_truncate` is `true`, the store file can be truncated, but not below the size set in `file_minimum`. When this parameter is absent, the default is `true`, and the server will periodically attempt to truncate the store file.

Example

```
[my_sync]
  type = file
  file = /var/local/tibems/my_sync.db
  file_destination_defrag=2MB
  file_minimum = 10MB
  file_truncate = true
  mode = sync
```

tibrvcm.conf

This file defines the TIBCO Rendezvous certified messaging (RVCM) listeners for use by topics that export messages to a `tibrvcm` transport. The server preregisters these listeners when the server starts up so that all messages (including the first message published) sent by way of the `tibrvcm` transport are guaranteed. If the server does not preregister the RVCM listeners before exporting messages, the listeners are created when the first message is published, but the first message is not guaranteed.

The format of this file is

```
transport listenerName subjectName
```

Parameter Name	Description
transport	The name of the transport for this RVCM listener.
listenerName	The name of the RVCM listener to which topic messages are to be exported.
subjectName	The RVCM subject name that messages are published to. This should be the same name as the topic names that specify the `export` property.

Example

```
RVCM01 listener1 foo.bar
RVCM01 listener2 foo.bar.bar
```

topics.conf

This file defines all topics.

The format of the file is:

```
[jndi-name1, jndi-name2, ...]topic-name property1, property2, ...
```

> **Note:** Note that, while including JNDI names is optional, the square brackets [ ] must be included around JNDI names if they are included. For more information about setting JNDI names, see `create jndiname`.

For example, you might enter:

```
business.inventory global, import="FTL01,FTL02", export="FTL02",
maxbytes=1MB
```

Only topics listed in this file or topics with names that match the topics listed in this file can be created by the applications (unless otherwise permitted by an entry in `acl.conf`). For example, if topic `foo.*` is listed in this file, topics `foo.bar` and `foo.baz` can be created by the application.

Properties of the topic are inherited by all static and dynamic topics with matching names. For example, if `test.*` has the property secure, then `test.1` and `test.foo` are also secure. For information on properties that can be assigned to topics, see Destination Properties.

For further information on the inheritance of topic properties, refer to Wildcards * and > and Inheritance of Properties.

Restrictions and rules on topic names are described in Destination Name Syntax.

transports.conf

This file defines transports for importing messages from or exporting messages to TIBCO FTL and Rendezvous.

The format of the file is:

```
[transport_name] # mandatory -- square brackets included
   type = tibftl | tibrv | tibrvcm # mandatory
   [topic_import_dm = TIBEMS_PERSISTENT |
                      TIBEMS_NON_PERSISTENT |
                      TIBEMS_RELIABLE]
   [queue_import_dm = TIBEMS_PERSISTENT |
                      TIBEMS_NON_PERSISTENT |
                      TIBEMS_RELIABLE]
   [export_headers = true | false]
   [export_properties = true | false]
   transport-specific-parameters
```

Parameter Name	Description
[*transport_name*]	The name of the transport. Note that the square brackets [ ] DO NOT indicate that the *transport_name* is an option; they must be included around the name.
type	Transport type. • `tibftl` identifies TIBCO FTL transport • `tibrv` identifies TIBCO Rendezvous transport

Parameter Name	Description
	• `tibrvcm` identifies TIBCO Rendezvous Certified Messaging transport Each transport includes additional *transport-specific-parameters*.
topic_import_dm *queue_import_dm*	EMS sending clients can set the `JMSDeliveryMode` header field for each message. However, Rendezvous clients cannot set this header. Instead, these two parameters determine the delivery modes for all topic messages and queue messages that `tibemsd` imports on this transport. `TIBEMS_PERSISTENT | TIBEMS_NON_PERSISTENT | TIBEMS_RELIABLE` When absent, the default is `TIBEMS_NON_PERSISTENT`.
export_headers	When `true`, `tibemsd` includes Jakarta Messaging header fields in exported messages. When `false`, `tibemsd` suppresses Jakarta Messaging header fields in exported messages. When absent, the default value is `true`.
export_properties	When `true`, `tibemsd` includes Jakarta Messaging properties in exported messages. When `false`, `tibemsd` suppresses Jakarta Messaging properties in exported messages. When absent, the default value is `true`.
transport-specific-parameters	See Transport-specific Parameters.

Note: If you have multiple TIBCO Rendezvous transports configured in your `transports.conf` file, and if the EMS server fails to create a transport based on the last entry, the server will continue to traverse through the entries and attempt to create further transports.

Transport-specific Parameters

tibftl transports

If `type = tibftl`, the extended syntax is:

```
[endpoint = endpoint-name]
[import_subscriber_name = subscriber-name]
[import_match_string = {"fieldname1":value1,...,"fieldnameN":valueN}]
[export_format = format-name]
[export_constant = constant1,value1]
...
[export_constant = constantN,valueN]
```

See TIBCO FTL Parameters for descriptions.

tibrv transports

If `type = tibrv`, the extended syntax is:

```
[service = service]
[network = network]
[daemon = daemon]
[temp_destination_timeout = seconds]
[rv_queue_policy = [TIBRVQUEUE_DISCARD_NONE |
                    TIBRVQUEUE_DISCARD_FIRST |
                    TIBRVQUEUE_DISCARD_LAST]:max_msgs:qty_discard]
```

See Rendezvous Parameters for descriptions.

tibrvcm transports

If `type = tibrvcm`, the extended syntax is:

```
rv_tport = name # mandatory
[cm_name = name]
[ledger_file = file-name]
[sync_ledger = true | false]
[request_old = true | false]
[explicit_config_only = true | false]
[default_ttl = seconds]
[rv_queue_policy = [TIBRVQUEUE_DISCARD_NONE |
```

```
                    TIBRVQUEUE_DISCARD_FIRST |
                    TIBRVQUEUE_DISCARD_LAST]:max_msgs:qty_discard]
```

See Rendezvous Certified Messaging (RVCM) Parameters for descriptions.

Example

```
[FTL01]
  type = tibftl
  endpoint = EP1
  import_subscriber_name = sub1
  import_match_string = {"f1":"foo","f2":true}
  export_format = format-1
  export_constant = constant1,value1
  export_constant = constant2,value2
  export_constant = constant3,value3

[RV01]
  type = tibrv
  topic_import_dm = TIBEMS_RELIABLE
  queue_import_dm = TIBEMS_PERSISTENT
  service = 7780
  network = lan0
  daemon = tcp:host5:7885

[RVCM01]
  type = tibrvcm
  export_headers = true
  export_properties = true
  rv_tport = RV02
  cm_name = RVCMTrans1
  ledger_file = ledgerFile.store
  sync_ledger = true
  request_old = true
  default_ttl = 600

[RV02]
  type = tibrv
  topic_import_dm = TIBEMS_PERSISTENT
  queue_import_dm = TIBEMS_PERSISTENT
  service = 7780
  network = lan0
  daemon = tcp:host5:7885
  rv_queue_policy = TIBRVQUEUE_DISCARD_LAST:10000:100
```

users.conf

This file defines all users.

The format of the file is:

```
username:password:"description"
```

Parameter Name	Description
username	The name of the user. The username cannot exceed 255 characters in length.
password	Leave this item blank when creating a new user. For example: `bob::"Bob Smith"` There is one predefined user, the administrator. User passwords are not entered in this configuration file, and remain empty (and therefore *not* secure) until you set them using the administration tool; see Assigning a Password to the Administrator. You can also create users and assign passwords using API calls; see the API reference for the language you are working with.
description	A string describing the user.

Example

```
admin::"Administrator"
Bob::"Bob Smith"
Bill::"Bill Jones"
```

After the server has started and passwords have been assigned, the file will look like this:

```
admin:$1$urmKVgq78:"Administrator"
Bob:$2$sldfkj;lsafd:"Bob Smith"
Bill:$3$tyavmwq92:"Bill Jones"
```

Authentication and Permissions

The EMS server supports authentication of incoming connections through user and password validation, JAAS authentication modules, and OAuth 2.0.

The EMS server also supports access control (authorization) through enforcement of permissions on users and groups. EMS supports two basic levels of permissions: administrator and user.

Administrator permissions control the ability of a user to log in as an administrator to create, delete, or view the status of users, destinations, connections, factories, and so on. Administrators with the correct permissions can control user access to the EMS server by creating users, assigning passwords, and setting permissions.

User permissions apply to the activities a user can perform on each destination (topic and queue). Using permissions, you can control which users have permission to send, receive, or browse messages for queues. You can also control who can publish or subscribe to topics, or who can create durable subscriptions to topics. Permissions are stored in the access control list for the server.

Note: Authentication has some similar characteristics to Transport Layer Security (TLS). TLS allows for servers to require user authentication by way of the user's digital certificate. TLS does not, however, specify any access control at the destination level. TLS and the authentication and access control features described in this chapter can be used together or separately to ensure secure access to your system. See TLS Protocol for more information about TLS.

Setting up EMS Authentication and Access Control

The following procedure describes the general process for administrators to configure authentication and permissions and where to find more information on performing each step.

Procedure

1. Enable authentication and access control for the system. See Enable Authentication and Access Control.
2. Determine which destinations require access control, and enable access control for those destinations. See Destination Access Control .
3. Configure the list of authentication methods. See Authentication Methods.
4. Determine the names of the authorized users of the system and create usernames and passwords for these users. See Users.
5. Optionally, set up groups and assign users to groups. See Groups.
6. Determine which users need administrator permissions, and decide whether administrators can perform actions globally or be restricted to a subset of actions. See Administrator Permissions for more information.

Users and Groups

The following sections describe users and groups in EMS.

Users

Users are specific, named IDs that allow you to identify yourself to the server. When a client attempts to connect to the server, it presents a set of credentials to identify itself as a particular user. These credentials can either be a username and corresponding password, or an OAuth 2.0 access token containing a username (see Authentication Using OAuth 2.0).

Note: In special cases, you may wish to allow anonymous access to the server. In this case, a connect request does not have to supply a username or password. To configure the server to allow anonymous logins, you must create a user named `anonymous` and specify no password. Anonymous logins are not permitted unless the anonymous user exists.

Clients logging in anonymously are only able to perform the actions that the `anonymous` user has permission to perform.

Users are the basis for access control in the EMS server. Users can be assigned server-wide administrative permissions and destination-level permissions. These permissions dictate what actions the user can perform once connected to the EMS server.

There is one predefined user, `admin`, that performs administrative tasks, such as creating other users.

You can create and remove users and change passwords by specifying the users in the `users.conf` configuration file, using the `tibemsadmin` tool, or by using the administration APIs. For more information about specifying users in the configuration file, see users.conf. For more information about specifying users using the `tibemsadmin` tool, see EMS Administration Tool. For more information on the administration APIs, see the online documentation.

EMS can also obtain user information from an external directory (such as an LDAP server), or an OAuth 2.0 access token presented by the connecting client. Such externally-configured users must be defined and managed directly via the external directory or OAuth 2.0 provider.

Groups

Groups allow you to create classes of users. Groups make access control administration significantly simpler because you can grant and revoke permissions to large numbers of users with a single operation on the group.

Each user can belong to as many groups as necessary. A user's permissions are the union of the permissions of the groups the user belongs to, in addition to any permissions granted to the user directly.

You can create, remove, or add users to groups by specifying the groups in `groups.conf`, using the `tibemsadmin` tool, or by using the administration APIs. For more information about specifying groups in the configuration file, see `groups.conf`. For more information about specifying groups using the `tibemsadmin` tool, see EMS Administration Tool. For more information on the administration APIs, see the online documentation.

EMS can also obtain group information from an external directory (such as an LDAP server), or an OAuth 2.0 access token presented by the connecting client. Such externally-configured groups must be defined and managed directly using the external directory, or the OAuth 2.0 provider.

Note: EMS can retrieve both static and dynamic groups from an external directory.

Administration Commands and External Users and Groups

You can perform administrative commands on users and groups defined either locally (in the EMS server's local configuration files), or externally (in an external directory accessed through JAAS, or in an OAuth 2.0 provider). Furthermore, you can combine users and groups that are defined in different locations (for example, you can grant and revoke permissions for users and groups defined externally, or add externally-defined users to locally-defined groups).

Note: In order to combine users and groups in this manner, the `user_auth` configuration parameter must have at least two authentication methods specified. See Authentication Methods for details.

When you attempt to view users and groups using the `show user/s` or `show group/s` commands, any externally-defined users and groups have an asterisk next to their names. Externally-defined users and groups will only appear in the output of these commands in the following situations:

- an externally-defined user successfully authenticates
- a user belonging to an externally-defined group successfully authenticates
- an externally-defined user has been added to a locally-defined group
- permissions on a topic or queue have been granted to an externally-defined user or group

Therefore, not all externally-defined users and groups may appear when the show `user/s` or `show group/s` commands are executed. Only the users and groups that meet the above criteria at the time the command is issued will appear.

You can create users and groups with the same names as externally-defined users and groups. If a user or group exists in the server's configuration and is also defined externally, the local definition of the user takes precedence. Locally-defined users and groups will not have an asterisk by their names in the `show user/s` or `show group/s` commands.

You can also issue the `delete user` or `delete group` command to delete users and groups from the local server's configuration. The permissions assigned to the user or group are also deleted when the user or group is deleted. If you delete a user or group that is defined externally, this deletes the user or group from the server's memory and deletes any permissions assigned in the access control list, but it has no effect on the external definition of that user or group (for example, it will not be deleted from the external directory or OAuth 2.0 provider). The externally-defined user can once again log in, and the user is created in the server's memory and any groups to which the user belongs are also created. However, any permissions for the user or group have been deleted and therefore must be re-granted.

Enable Authentication and Access Control

Administrators can enable or disable authentication and access control for the server. Administrators can also enable and disable permission checking for specific destinations.

Server Access Control and Authentication

The authorization property in the main configuration file enables or disables authentication of incoming connections and the checking of permissions for all destinations managed by the server.

> ⚠ **Warning:** The default setting is `disabled`. For secure deployments, the administrator must explicitly set `authorization` to `enabled`.

When `authorization` is `disabled`, the server grants any connection request, and does not check permissions when a client accesses a destination (for example, publishing a message to a topic).

When `authorization` is `enabled`, the server grants connections only upon successful authentication. The server checks permissions for client operations involving secure destinations.

To enable `authorization`, either edit `tibemsd.conf` (set the `authorization` property to `enabled`, and restart the server). Or you can use the `tibemsadmin` tool to dynamically enable authorization with the following set server command:

```
set server authorization=enabled
```

Authorization does affect connections between fault-tolerant server pairs; see Authorization and Fault-Tolerant Servers.

Administrators must always log in with the correct administration username and password to perform any administrative function—even when `authorization` is `disabled`.

Destination Access Control

When authorization is enabled, the server verifies user and group permissions before allowing operations on destinations, such as sending a message or receiving a message. However, this destination access control is only applicable for those destinations that have the secure property enabled. All operations by applications on the destination with `secure` enabled are verified by the server according to the permissions listed in `acl.conf`. Destinations with `secure` disabled continue to operate without any restrictions.

Note: The `secure` property is independent of TLS-level security. The `secure` property controls only basic authentication and permission verification. It does not affect the security of communication between clients and server.

When a destination does not have the `secure` property set, any authenticated user can perform any actions on that topic or queue.

See Destination Properties for more information about destination properties.

Authentication Methods

The parameter user_auth in `tibemsd.conf` determines the authentication methods the EMS server can use to authenticate incoming connection requests - either from EMS clients or other EMS servers. This parameter can have one or more of the following values (separated by comma characters):

- `local`—authenticate incoming connection requests by validating the presented user credentials against locally defined user information (`users.conf` and `groups.conf`).
- `jaas`—authenticate incoming connection requests by validating the presented user

credentials using a custom or provided JAAS authentication module, including LDAP support (see Extensible Authentication).

- `oauth2`—authenticate incoming connection requests by validating the presented OAuth 2.0 access token (see Authentication Using OAuth 2.0).

Each time the server receives a connection request, it attempts to authenticate it via each of the specified authentication methods in the order that this parameter specifies. The EMS server accepts successful authentication using any of the specified methods.

Authentication Using OAuth 2.0

TIBCO EMS supports authentication of client connections via OAuth 2.0.

When connecting to an EMS server configured with OAuth 2.0 authentication, an EMS client must authenticate itself to the server by presenting an access token issued by an OAuth 2.0 authorization server. This access token must be a signed JSON Web Token (JWT) that includes a claim containing the user that the client will identify itself as to the server (see Users); and optionally, another claim containing the list of groups the user belongs to (see Groups).

The EMS server validates the access token's signature and claims and accepts or rejects the connection request accordingly. If this authentication process is successful, the EMS client will be allowed to connect as the EMS user specified in the access token. The server will enact access control for the client connection based on the permissions defined in the acl.conf file (and the Extensible Security permissions module, if applicable).

In order for an OAuth 2.0 authorization server to issue an access token with the expected claims, the relevant EMS user and group information must be made available to it. Depending on your OAuth 2.0 provider, there may be a number of options available for achieving this. For example, you may be able to define EMS users and groups directly in your provider, or you may be able to integrate your provider with an external authentication service such as LDAP. Refer to your OAuth 2.0 provider's documentation for instructions.

Obtaining an Access Token

EMS clients require an access token to connect to an EMS server configured with OAuth 2.0 authentication. Additionally, an EMS server itself may require an access token to connect to

another EMS server or a TIBCO messaging product that is configured for OAuth 2.0 authentication.

Note: EMS only supports access tokens in the form of signed JWTs with asymmetric validation keys. Unsigned JWTs, or JWTs with symmetric validation keys are not supported.

The EMS client APIs and the EMS server can be configured to use either the Client Credentials grant or Resource Owner Password Credentials grant for obtaining OAuth 2.0 access tokens.

Client Credentials Grant

In the Client Credentials grant, the EMS client (or EMS server) presents a client ID and client secret to the OAuth 2.0 authorization server. The authorization server uses these credentials to authenticate the EMS client and issues it an access token.

This is the preferred grant type for both the EMS client and EMS server.

Resource Owner Password Credentials Grant

In the resource owner password credentials grant, the EMS client (or EMS server) first authenticates itself to the OAuth 2.0 authorization server by presenting a client ID and client secret, then provides an EMS user and password for validation by the authorization server. If both authentication operations are successful, the authorization server issues an access token to the client.

Refresh tokens are supported with this grant type. If the authorization server issues a refresh token along with the requested access token, the EMS client (or EMS server) will use that refresh token instead of the grant for requesting the next access token. If it fails to obtain a new access token using the refresh token, it will try again using the grant.

This grant type has been deemed legacy and is expected to be removed in a future update to the OAuth 2.0 framework. It should only be used in situations where the client credentials grant type is not feasible.

Using Externally-Obtained Access Tokens

In addition to the above grant types, the EMS client APIs and the EMS server can be configured to use access tokens obtained via external means. If access tokens are directly configured in this manner, the EMS server and client APIs will not attempt to obtain access tokens using the OAuth 2.0 grants.

Note: The EMS client APIs additionally support user-defined callbacks for obtaining access tokens. This method of obtaining access tokens requires the use of new APIs. Existing EMS client applications will need to be modified to use the new APIs in order to make use of this method. Refer to the corresponding client API documentation for details.

User-defined callbacks are only available in the client APIs. This method of obtaining access tokens is not supported in the EMS server.

Access Token Expiration

OAuth 2.0 JWT access tokens can have an expiration time specified through the ‘exp’ claim. The EMS server enforces access token expiration by disconnecting the associated EMS client (or EMS server).

In order to minimize connection disruptions due to access token expirations, the EMS server periodically examines the access tokens of all connected EMS clients to check for upcoming expirations. If an access token is nearing the end of its lifetime, the EMS server will send a request for re-authentication to the client. This provides the client the opportunity to procure a new access token and authenticate with the EMS server again before the current access token expires.

If an EMS client (or EMS server) obtained its access token using one of the supported grant types, upon receiving a re-authentication request from the EMS server, it will automatically procure a new access token from the OAuth 2.0 authorization server and re-authenticate itself with the EMS server without any connection disruption.

If an EMS client obtained its access token through other means (see Access Token Expiration), it will need to be told how to handle the re-authentication request from the EMS server via a user-defined callback. Refer to the relevant EMS client API documentation for details.

Configure OAuth 2.0 in the EMS Server

To enable OAuth 2.0 authentication of incoming connections to the EMS server, add `oauth2` to the `user_auth` server parameter.

```
user_auth=oauth2
```

Note: The EMS server can be configured to use `oauth2` alongside other authentication methods. See Authentication Methods for details.

In order to validate the signature and claims of access tokens presented by incoming connections, the `oauth2_server_validation_key` and `oauth2_user_claim` server parameters must also be set. See OAuth 2.0 Parameters for descriptions of all required and optional OAuth 2.0 related server parameters.

Access Tokens for Outgoing Connections

Depending on its configuration, the EMS server may need to initiate an outgoing connection to a different EMS server or another TIBCO messaging product that is configured with OAuth 2.0 authentication. In these scenarios, the EMS server requires additional configuration relating to OAuth 2.0 access token procurement. Refer to the following sections for details.

- To configure OAuth 2.0 authentication between a fault-tolerant EMS server pair, see Authentication and Authorization for Fault-Tolerant Servers.
- To configure OAuth 2.0 authentication for routes between EMS servers, see Authentication and Authorization for Routes.
- To configure OAuth 2.0 authentication between the EMS server and an FTL deployment for FTL transports, see TIBCO FTL Transport Parameters.
- If using FTL stores, the configuration for OAuth 2.0 authentication in the EMS server differs. See tibemsd Service Parameters for more information.

Authenticate Administrative Connections

Administrative connections, such as those created by the EMS Administration Tool and the EMS administrative API, are authenticated differently than client connections.

When establishing an administrative connection, local authentication is always attempted first, regardless of the authentication methods specified through `user_auth`. If the local authentication attempt fails, authentication will proceed as per the `user_auth` parameter.

It is recommended that users making administrative connections to the EMS server are *not* defined both locally (`users.conf`) and externally (in an external directory, or an OAuth 2.0 provider). Administrative users should only be defined in one place.

An exception is the default administrative user, admin, which is always defined locally by the EMS server. If the default administrative user is to be defined elsewhere and authenticated through `jaas` or `oauth2` authentication methods, one can set an undisclosed password for the default administrative user in the EMS server's user configuration file (`users.conf`) so that local authentication of the admin user never succeeds, thus allowing the other authentication methods to be used instead.

Administrator Permissions

Administrators are a special class of users that can manage the EMS server. Administrators create, modify, and delete users, destinations, routes, factories, and other items. In general, administrators must be granted permission to perform administration activities when using the Administration Tool or administration API. Administrators can be granted global permissions (for example, permission to create users or to view all queues), and administrators can be granted permissions to perform operations on specific destinations (for example, purging a queue, or viewing properties for a particular topic).

> **Warning:** Administrator permissions control what administrators can view and change in the server only when using the Administration Tool or administration API. Administrator commands create entries in each of the configuration files (for example, `tibemsd.conf`, `acl.conf`, `routes.conf`, and so on).
>
> You should control access to the configuration files so that only certain system administrators can view or modify the configuration files. If a user can view or modify the configuration files, setting permissions to control which destination that user can manage would not be enforced when the user manually edits the files.
>
> Use the facilities provided by your Operating System to control access to the server's configuration files.

Administrators must be created using the administration tool, the administration APIs, or in the configuration files.

Predefined Administrative User and Group

There is a special, predefined user named `admin` that can perform any administrative action. You cannot grant or revoke any permissions to admin. You must assign a password for `admin` immediately after installation.

For more information about changing the `admin` password, see When You First Start tibemsadmin.

There is also a special group named `$admin` for system administrator users. When a user becomes a member of this group, that user receives the same permissions as the `admin` user. You cannot grant or revoke administrator permissions from any user that is a member of the `$admin` group. You should only assign the overall system administrator(s) to the `$admin` group.

Granting and Revoking Administration Permissions

You grant and revoke administrator permissions to users using the `grant` and `revoke` commands in `tibemsadmin`, or by means of the Java or .NET administration API. You can either grant global administrator permissions or permissions on specific destinations.

See Global Administrator Permissions for a complete list of global administrator permissions. See Destination-Level Permissions for a description of administrator permissions for destinations.

Global and destination-level permissions are granted and revoked separately using different administrator commands. See Command Listing for the syntax of the `grant` and `revoke` commands.

If a user has both global and destination-level administrator permissions, the actions that user can perform are determined by combining all global and destination-level administrator permissions granted to the user. For example, if an administrator is granted the `view-destination` permission, that administrator can view information about all destinations, even if the view permission is not granted to the administrator for specific destinations.

The `admin` user or all users in the `$admin` group can grant or revoke any administrator permission to any user. All other users must be granted the `change-admin-acl` permission and the `view-user` and/or the `view-group` permissions before they can grant or revoke administrator permissions to other users.

If a user has the `change-admin-acl` permission, that user can only grant or revoke permissions that have been granted to the user. For example, if user BOB is not part of the `$admin` group and he has only been granted the `change-admin-acl` and `view-user` permissions, BOB cannot grant any administrator permissions except the `view-user` or `change-admin-acl` permissions to other users.

Users have all administrator permissions that are granted to any group to which they belong. You can create administrator groups, grant administrator permissions to those groups, and then add users to each administrator group. The users will be able to perform any administrative action that is allowed by the permissions granted to the group to which the user belongs.

Any destination-level permission granted to a user or group for a wildcard destination is inherited for all child destinations that match the parent destination.

If protection permissions are set up, administrators can only grant or revoke permissions to other users that have the same protection permission as the administrator. See Protection Permissions for more information about protection permissions.

Enforcement of Administrator Permissions

An administrator can only perform actions for which the administrator has been granted permission. Any action that an administrator performs may be limited by the set of permissions granted to that administrator.

For example, an administrator has been granted the view permission on the `foo.*` destination. This administrator has not been granted the global view-destination permission. The administrator is only able to view destinations that match the `foo.*` parent destination. If this administrator is granted the global `view-acl` permission, the administrator is only able to view the access control list for destinations that match the `foo.*` parent. Any access control lists for other destinations are not displayed when the administrator performs the `showacl topic` or `showacl queue` commands.

If the administrative user attempts to execute a command without permission, the user may either receive an error or simply see no output. For example, if the administrator issues the `showacl queue bar.foo` command, the administrator receives a “Not authorized to execute command” error because the administrator is not authorized to view any destination except those that match `foo.*`.

Note: An administrator can always change his/her own password, even if the administrator is not granted the `change-user` permission.

An administrator can always view his/her own permissions by issuing the:

`showacl` *username*

command, even if the administrator is not granted the `view-acl` permission.

Global Administrator Permissions

Certain permissions allow administrators to perform global actions, such as creating users or viewing all queues.

The following table describes the global administrator permissions.

Permission	Allows Administrator To...
`all`	Perform all administrative commands.
`view-all`	View any item that can be administered (for example, users, groups, topics, and so on).
`change-acl`	Grant and revoke user-level permissions.
`change-admin-acl`	Grant and revoke administrative permissions.
`change-bridge`	Create and delete destination bridges.
`change-connection`	Delete connections.
`create-destination`	Create any destination.
`modify-destination`	Modify any destination.
`delete-destination`	Delete any destination.
`change-durable`	Delete durable subscribers.
`change-factory`	Create, delete, and modify factories.
`change-group`	Create, delete, and modify groups.
`change-message`	Delete messages stored in the server.
`change-route`	Create, delete, and modify routes
`change-server`	Modify server parameters.
`change-user`	Create, delete, and modify users.
`purge-destination`	Purge destinations.
`purge-durable`	Purge durable subscribers.
`shutdown`	Shut down the server.

Permission	Allows Administrator To...
`view-acl`	View user-level permissions.
`view-admin-acl`	View administrative permissions.
`view-connection`	View connections, producers and consumers.
`view-bridge`	View destination bridges.
`view-destination`	View destination properties and information.
`view-durable`	View durable subscribers. To view a durable subscriber, you must also have `view-destination` permission (because information about a durable subscriber includes information about the destination to which it subscribes.)
`view-factory`	View factories.
`view-group`	View all groups. Granting this permission implicitly grants view-user as well.
`view-message`	View messages stored in the server.
`view-route`	View routes.
`view-server`	View server configuration and information.
`view-user`	View any user.

Note: Any type of modification to an item requires that the user can view that item. Therefore, granting any create, modify, delete, change, or purge permission implicitly grants the permission to view the associated item.

Granting the view permissions is useful when you want specific users to only be able to view items. It is not necessary to grant the view permission if a user already has a permission that allows the user to modify the item.

Global permissions are stored in the `acl.conf` file, along with all other permissions. Global permissions in this file have the following syntax:

```
ADMIN USER=<username> PERM=<permission>
```

or

```
ADMIN GROUP=<groupname> PERM=<permission>
```

For example, if a user named BOB is granted the `view-user` global administration permission and the group `sys-admins` is granted the `change-acl` permission, the following entries are added to the `acl.conf` file:

```
ADMIN USER=BOB PERM=view-user
ADMIN GROUP=sys-admins PERM=change-acl
```

Destination-Level Permissions

Administrators can be granted permissions on each destination. Destination-level permissions control the administration functions a user can perform on a specific destination. Global permissions granted to a user override any destination-level permissions.

The typical use of destination-level administration permissions is to specify permissions on wildcard destinations for different groups of users. This allows you to specify particular destinations over which a group of users has administrative control. For example, you may allow one group to control all `ACCOUNTING.*` topics, and another group to control all `PAYROLL.*` queues.

The following table describes the destination-level administration permissions.

Permission	Allows Administrator To...
`view`	View information for this destination.
`create`	Create the specified destination. This permission is useful when used with wildcard destination names. This allows the user to create any destination that matches the specified parent.

Permission	Allows Administrator To...
`delete`	Delete this destination.
`modify`	Change the properties for this destination.
`purge`	Either purge this queue, if the destination is a queue, or purge the durable subscribers, if the destination is a topic with durable subscriptions.

Note: Any type of modification to an item requires that the user can view that item. Therefore, granting create, modify, delete, change, or purge implicitly grants the permission to view the associated item.

Granting the view permissions is useful when you want specific users to only be able to view items. It is not necessary to grant the view permission if a user already has a permission that allows the user to modify the item.

Administration permissions for a destination are stored alongside all other permissions for the destination in the `acl.conf` file. For example, if user `BOB` has publish and subscribe permissions on topic `foo`, and then `BOB` is granted view permission, the acl listing would look like the following:

```
TOPIC=foo USER=BOB PERM=publish,subscribe,view
```

Note: Both user and administrator permissions for a destination are stored in the same entry in the `acl.conf` file. This is for convenience rather than for clarity. User permissions specify the actions a client application can perform on a destination (publish, subscribe, send, receive, and so on). Administrator permissions specify what administrative commands the user can perform on the destination when using the administration tool or administration API.

Protection Permissions

Protection permissions allow you to group users into administrative domains so that administrators can only perform actions within their domain. An administrator can only perform administrative operations on a user that has the same protection permission as the user.

There are four protection permissions (`protect1`, `protect2`, `protect3`, and `protect4`) that allow you to create four groups of administrators. Protection permissions do not apply to the `admin` user or users in the `$admin` group — these users can perform any action on any user regardless of protection permissions.

To use protection permissions, grant one of the protection permissions to a set of users (either individually, or to a defined group(s)). Then, grant the same protection permission to the administrator that can perform actions on those users.

For example, there are four departments in a company: sales, finance, manufacturing, and system administrators. Each of these departments has a defined group and a set of users assigned to the group. Within the system administrators, there is one manager and three other administrators, each responsible for administering the resources of the other departments. The manager of the system administrators can perform any administrator action. Each of the other system administrators can only perform actions on members of the groups for which they are responsible.

The user name of the manager is `mgr`, the user names of the other system administrators are `admin1`, `admin2`, and `admin3`. The following commands illustrate the grants necessary for creating the example administration structure.

```
add member $admin mgr
grant admin sales protect1
grant admin admin1 protect1,all
grant admin manufacturing protect2
grant admin admin2 protect2,all
grant admin finance protect3
grant admin admin3 protect3,all
```

Note: You can grant a protection permission, in addition to the `all` permission. This signifies that the user has all administrator privileges for anyone who also has the same protection permission. However, if you revoke the `all` permission from a user, all permissions, including any protection permissions are removed from the access control list for the user.

An administrator is able to view users that have a different protection permission set, but the administrator can only perform actions on users with the same protection permission.

For example, `admin1` can perform any action on any user in the `sales` group, and can view any users in the `manufacturing` or `finance` groups. However, `admin1` is not able to grant permissions, change passwords, delete users from, or perform any other administrative action on users of the `manufacturing` or `finance` groups. The `mgr` user is able to perform

any action on any user, regardless of their protection permission because `mgr` is a member of the `$admin` group.

User Permissions

User permissions are stored in the access control list and determine the actions a user can perform on a destination. A user's permissions are the union of the permissions granted explicitly to that user along with any permissions the user receives by belonging to a group.

When granting user permissions, you specify the user or group to whom you wish to grant the permission, the name of the destination, and the permission(s) to grant. Granting permissions is an action that is independent from both the `authorization` server parameter, and the `secure` property of the relevant destinations. The currently granted permissions are stored in the access control file, however, the server enforces them only if the `authorization` is enabled, and only for secure destinations.

Note: When setting permissions for users and groups defined externally, user and group names are case-sensitive. Make sure you use the correct case for the name when setting the permissions.

User permissions can only be granted by an administrator with the appropriate permissions described in Administrator Permissions.

You assign permissions either by specifying them in the `acl.conf` file, using the `tibemsadmin` tool, or by using the administration APIs. When setting user permissions, you can specify either explicit destination names or wildcard destination names. See Inheritance of User Permissions for more information on wildcard destination names and permissions.

Queue and Topic Permissions

The permissions that can be granted to users to access queues and topics are listed in the following tables.

Queue Permission

Name	Description
`receive`	permission to create queue receivers
`send`	permission to create queue senders
`browse`	permission to create queue browsers

Topic Permission

Name	Description
`subscribe`	permission to create non-durable subscribers on the topic
`publish`	permission to publish on the topic
`durable`	permission to create, delete, or modify durable subscribers on the topic
`use_durable`	permission to use an existing durable subscriber on the topic, but *not* to create, delete, or modify the durable subscriber

Example of Setting User Permissions

The user bob has the following permission recorded in the acl.conf file:

```
USER=bob TOPIC=foo PERM=subscribe,publish
```

This set of permissions means that bob can subscribe to topic foo and publish messages to it, but bob cannot create durable subscribers to foo.

If bob is a member of group engineering and the group has the following entry in the acl file:

```
GROUP=engineering TOPIC=bar PERM=subscribe,publish
```

then bob can publish and subscribe to topics foo and bar.

If both the user bob and the group engineering have entries in the acl.conf file, then bob has permissions that are a union of all permissions set for bob directly and the permissions of the group engineering.

Inheritance of User Permissions

When you grant permissions to users for topics or queues with wildcard specifications, all created topics and queues that match the specification will have the same granted permissions as the permissions on the parent topic.

If there are multiple parent topics, the user receives the union of all parent topic permissions for any child topic. You can add permissions to a user for topics or queues that match a wildcard specification, but you cannot remove permissions.

For example, you can grant user Bob the browse permission on queue foo.*. The user Bob receives the browse permission on the foo.bar queue, and you can also grant Bob the send permission on the foo.bar queue. However, you cannot take away the inherited browse permission from Bob on the foo.bar queue.

See Wildcards for more information about wildcards in destination names.

Revoking User Permissions

Administrators can revoke permissions for users to create consumers on a destination. Without permission, the user cannot create new consumers for a destination—however, existing consumers of the destination continue to receive messages.

You can only revoke a permission that is granted directly. That is, you cannot revoke a permission from a user that the user receives from a group. Also, you cannot revoke a permission that is inherited from a parent topic. The `revoke` command in `tibemsadmin` can only remove items from specific entries in the `acl.conf` file. The revoke command cannot remove items that are inherited from other entries.

You can revoke permissions in several ways:

- Remove or edit entries in the `acl.conf` file.
- Use the revoke commands in `tibemsadmin`.
- Use the administration APIs.

When Permissions Are Checked

If permissions are enforced (that is, the `authorization` configuration property is set, and the `secure` property is set for the destination), the server checks them when a user attempts to perform an operation on a destination. For example, create a subscription to a topic, send a message to a queue, and so on. Since permissions can be granted or revoked dynamically, the server checks them each time an operation is performed on a destination (and each time a consumer or producer is created).

For specific (non-wildcard) destination names, permissions are checked when a user performs one of the following actions:

- creates a subscription to a topic
- attempts to become a consumer for a queue
- publishes or sends a message to a topic or queue
- attempts to create queue browser

A user cannot create or send a message to a destination for which he or she has not explicitly been granted the appropriate permission. So, before creating or sending messages to the destination, a user must be granted permissions on the destination.

However, for wildcard topic names (queue consumers cannot specify wildcards), permissions are not checked when users create non-durable subscriptions. Therefore, a user can create a subscription to topic `foo.*` without having explicit permission to create subscriptions to `foo.*` or any child topics. This allows administrators to grant users the desired permissions after the user's application creates the subscriptions. You may wish to allow users to subscribe to unspecific wildcard topics, then grant permission to specific topics at a later time. Users are not able to receive messages based on their wildcard subscriptions until permissions for the wildcard topic or one or more child topics are granted.

Attempts to perform an operation by a user who does not have the permission to perform it are traced in the server log file.

Note: When creating a durable subscriber, users must have the `durable` permission explicitly set for the topic they are subscribing to. For example, to create a durable subscriber to topic `foo.*`, the user must have been granted the durable permission to create durable subscriptions for topic `foo.*`. To subscribe an existing durable subscriber to a topic,
you must have either durable or `use_durable` permission set on that topic.

Example of Permission Checking

This example walks through a scenario for granting and revoking permissions to a user, and describes what happens as various operations are performed.

1. User bob is working with a EMS application that subscribes to topics and displays any messages sent to those topics.
2. User bob creates a subscription to `user.*`. This topic is the parent topic of each user. Messages are periodically sent to each user (for example, messages are sent to the topic `user.bob`). Because the same application is used by many users, the application creates a subscription to the parent topic.
3. User bob creates a subscription to topic `corp.news`. This operation fails because bob has not been granted access to that topic yet.
4. A message is sent to the topic `user.bob`, but the application does not receive the message because bob has not been granted access to the topic yet.
5. The administrator, as part of the daily maintenance for the application, grants access

to topics for new users. The administrator grants the `subscribe` permission to topic `user.bob` and `corp.*` to user `bob`. These grants occur dynamically, and user `bob` is now able to receive messages sent to topic `user.bob` and can subscribe to topic `corp.news`.

6. The administrator sends a message on the topic `user.bob` to notify bob that access has been granted to all `corp.*` topics.
7. The application receives the new message on topic `user.bob` and displays the message.
8. User `bob` attempts to create a subscription for topic `corp.news` and succeeds.
9. A message is sent to topic `corp.news`. User bob's application receives this message and displays it.
10. The administrator notices that bob is a contractor and not an employee, so the administrator revokes the subscribe permission on topic `corp.*` to user `bob`.

    The subscription to `corp.news` still exists for user `bob`'s application, but `bob` cannot create any new subscriptions to children of the `corp.*` topic.

Extensible Security

The following sections outline how to develop and implement custom authentication and permissions modules.

Overview of Extensible Security

The extensible security feature allows you to use your own authentication and permissions systems, in addition to the prebuilt JAAS modules included in EMS, to authenticate users and authorize them to perform actions such as publish and subscribe operations. Developing custom applications to grant authentication and permissions gives you more flexibility in architecting your system.

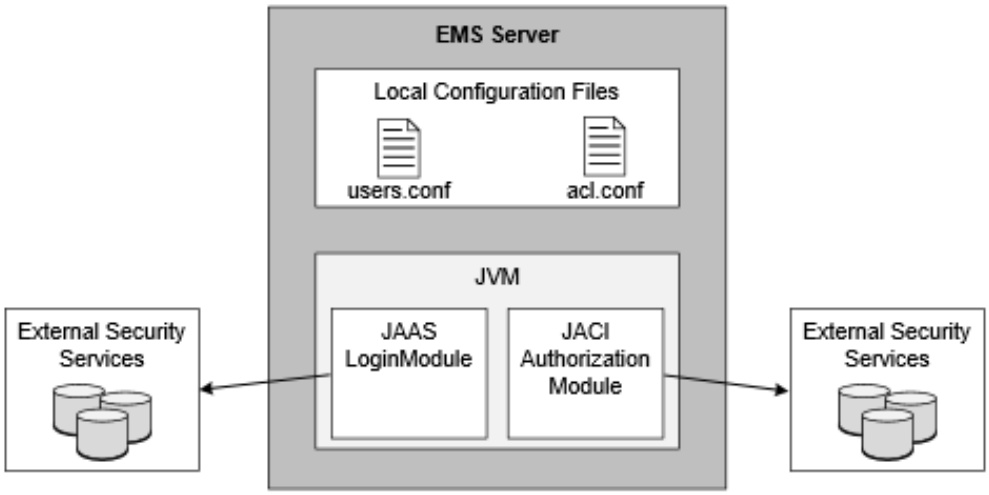
How Extensible Security Works

Extensible security works by allowing you to write your own authentication and permissions modules, which run in a Java virtual machine (JVM) in the EMS server. The modules connect to the server using the Java Authentication and Authorization Service (JAAS) for authentication modules, and the Java Access Control Interface (JACI) for permissions modules.

If the extensible security features are enabled when the EMS server starts, the server checks each user as it connects for authentication, and checks user permissions when they attempt to perform actions that require authorization.

Permission results are cached in the server for specified timeouts, and the permissions module is re-invoked when a cached permission expires. The server then replaces the old permission data with new data.

Extensible authentication and extensible permissions are enabled in the `tibemsd.conf` configuration file. Extensible security modules can connect to external security services, such as single sign on (SSO) servers or LDAP directories, which operate outside of the TIBCO Enterprise Message Service framework. Extensible security modules can work in tandem with the EMS `acl.conf` configuration file. The following figure shows the different security methods available in the server.



Extensible Authentication

The extensible authentication feature uses the Java virtual machine (JVM) and the Java Authentication and Authorization Service (JAAS) to allow you to run your own Java-based authentication module in the EMS server.

Your authentication module, or LoginModule, runs in the JVM within the EMS server, and is accessed by `tibemsd` using the JAAS interface. This is a flexible way to extend the security of your EMS application. The LoginModule can be used to augment existing authentication processes, or can be the sole method of authentication used by the EMS server.
The `user_auth` parameter in the main configuration file determines when the LoginModule is used.

Each time an EMS client attempts to create a connection to the server, the server will authenticate the client before accepting the connection. When extensible authentication is enabled, `tibemsd` passes user information to the LoginModule, which returns an allow or deny response.

If more than one authentication mechanism is enabled, when a user attempts to authenticate, the server seeks corresponding authentication information from each of the specified locations in the order determined by the `user_auth` parameter. The EMS server accepts successful authentication using any of the specified sources.

For example, if local authentication appears before JAAS authentication, the server will search for the provided username and password first in the `users.conf` file. If the user

does not exist there or if the provided username and password don't match, the EMS server passes those to the LoginModule, which allows or denies the connection attempt.

Enable Extensible Authentication

Extensible authentication is enabled in the EMS server, through parameters in the `tibemsd.conf` configuration file. The required parameters are:

- `authorization`—directs the server to verify user credentials and permissions on secure destinations.
- `user_auth`—directs the EMS server to use the LoginModule for authentication.
- `security_classpath`—specifies the JAR files and dependent classes used by the LoginModule.
- `jaas_config_file`—specifies the configuration file, usually `jaas.conf`, that loads the LoginModule. For more information, see the Example jaas.conf Configuration File.

Because the LoginModule runs in the Java virtual machine, you must also enable the JVM in the EMS server. See Enable the JVM for more information.

Prebuilt Authentication Modules

TIBCO Enterprise Message Service includes several supported JAAS authentication modules that offer flexible authentication for the EMS server. The source files of the prebuilt modules are provided in *EMS_HOME*`/src/java/jaas`, and provide an excellent template for developing custom modules. Multiple instances of any prebuilt JAAS module can be used in any stacked combination to suit the authentication requirements of your environment.

These modules are described in JAAS Authentication Modules.

Writing an Authentication Module

The LoginModule is a custom module that runs inside the EMS server within a JVM. The LoginModule is written using JAAS, a set of APIs provided by Sun Microsystems, and used to create plugable Java applications. JAAS provides the interface between your code and the EMS server. JAAS is a standard part of JRE, and is installed with EMS.

LoginModule Requirements

In order to implement extensible authentication, you must write a LoginModule implementing the JAAS interface.

There are some requirements for a LoginModule that will run in the EMS server:

- The LoginModule must accept the username and password from the EMS server by way of the `NameCallback` and `PasswordCallback` callbacks. The EMS server passes the username and password to the LoginModule using these callbacks, ignoring the prompt argument.
- If the username and password combination is invalid, the LoginModule must throw a `FailedLoginException`. The EMS server then rejects the corresponding connection attempt.
- The LoginModule must be thread-safe. That is, the LoginModule must be able to function both in a multi-threaded environment and in a single-threaded environment.
- The LoginModule should perform authentication only, by determining whether a username and password combination is valid. For information about custom permissions, see Extensible Permissions.
- The LoginModule, like the Permissions Module, should not perform long operations, and should return values quickly. As these modules become part of the EMS server's message handling process, slow operations can have a severe effect on performance.
- The LoginModule must be named `EMSUserAuthentication`.

More information about JAAS, including documentation of JAAS classes and interfaces, is available through http://java.sun.com/products/jaas/.

Load the LoginModule in the EMS Server

The EMS server locates and loads the LoginModule based on the contents of the configuration file specified by the jaas_config_file parameter in the `tibemsd.conf` file. Usually, the JAAS configuration file is named `jaas.conf`. This file contains the configuration information used to invoke the LoginModule.

The contents of the `jaas.conf` file should follow the JAAS configuration syntax, as documented at:
https://docs.oracle.com/javase/8/docs/api/javax/security/auth/login/Configuration.html

Note: The LoginModule in the JAAS configuration file must have the name `EMSUserAuthentication`.

Example jaas.conf Configuration File

```
EMSUserAuthentication {
com.tibco.tibems.tibemsd.security.example.FlatFileUserAuthLoginModule required
debug=true filename=jaas_users.txt;
};
```

Extensible Permissions

The extensible permissions feature uses the Java virtual machine (JVM) and the Java Access Control Interface (JACI) to allow you to run your own Java-based permissions module in the EMS server.

Your Permissions Module runs in the JVM within the EMS server, and connects to `tibemsd` using the JACI interface. Like the LoginModule, the Permissions Module provides an extra layer of security to your EMS application. It does not supersede standard EMS procedures for granting permissions. Instead, the module augments the existing process.

When a user attempts to perform an action, such as subscribing to a topic or publishing a message, the EMS server checks the `acl.conf` file, the Permissions Module, and cached results from previous Permissions Module queries, for authorization. This process is described in detail in Granting Permissions.

Cached Permissions

In order to speed the authorization process, the EMS server caches responses received from the Permissions Module in two pools, the *allow cache* and the *deny cache*. Before invoking the Permissions Module, the server first checks these caches for a cache entry matching the user's request.

What is Cached

Each cache entry consists of a username and action, and the authorization result response from the Permissions Module.

Properties of cache entries:

- The username is specific; the cached permission applies only to this user.
- The action is also specific. Only one action is included in each cache entry. Actions that require authorization are the same as those listed in the `acl.conf` file.
- The destination can include wildcards. That is, a single cache entry can determine the user's authorization to perform the action on multiple destinations.

If the response from the Permissions Module authorized the action, the permission is cached in the allow cache. If the action was denied, it is cached in the deny cache.

How Long Permissions are Cached

Permissions Module results also include timeouts, which determine how long the cache entry is kept in the cache before it expires.

When a timeout has expired, the entry is removed from the cache. Because these timeouts are assigned by the Permissions Module, you can control how often the Permissions Module is called, and therefore how much load it puts on the EMS server.

Warning: Long timeouts on permissions cache entries can increase performance, but they also lower the system's responsiveness to changes in permissions. Consider timeout lengths carefully when writing your Permissions Module.

Administer the Cache

You can view and reset cache statistics, as well as clear all cache entries.

These commands are available in the administration tool:

- `jaci showstats`
- `jaci resetstats`

- jaci clear

How Permissions are Granted

When an EMS client attempts to perform an action that requires permissions, the EMS server looks in several locations for authorization.

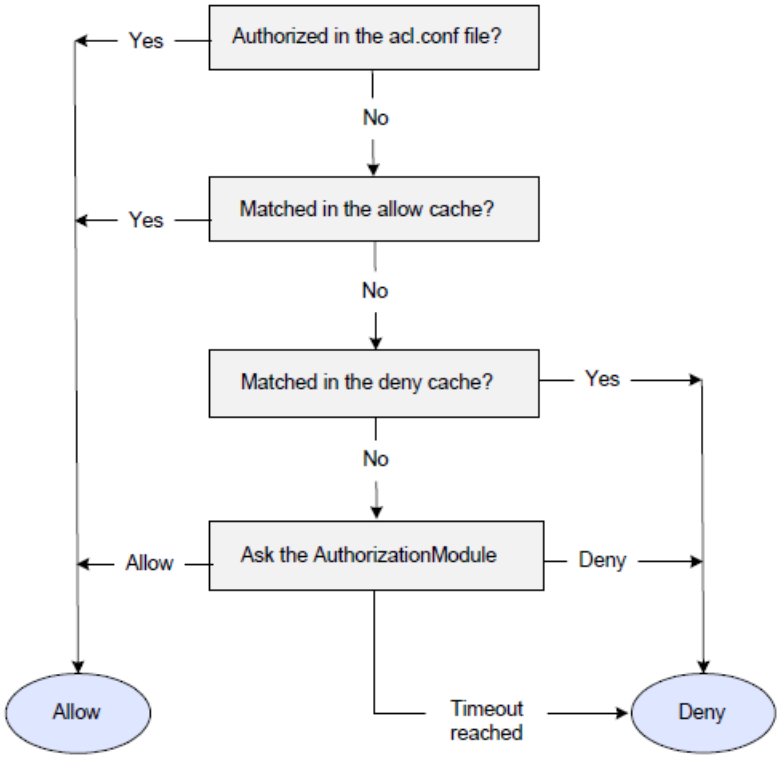
1. First, the server checks the acl.conf for authorization. This is the standard EMS mechanism for granting permissions, as is documented in Authentication and Permissions.
2. Next, the server checks the Permissions Module allow cache for authorization. If an entry matching the username, action, and destination exists in the cache, the request is allowed.

   Because destinations with wildcards can exist in the cache, an entry can have a wildcard destination that contains the requested destination. If that entry specifies the same username and action, the request is allowed. For more information on this topic, see Implications of Wildcards on Permissions below.
3. The server then checks the deny cache for a matching entry. If an entry exists in the deny cache, the request is denied.

   As in the allow cache, wildcards used in destinations can result in a cache entry with a destination that contains the requested destination. If that entry matches the username and action, the request is denied. For more information on this topic, see Implications of Wildcards on Permissions below.
4. Finally, if there are no matching entries in either cache, the server passes the username, action type, and destination to the Permissions Module, which returns an allow or deny authorization response. The response is also saved to the cache for the timeout specified in the response.

   If the Permissions Module does not respond to the request within the timeout specified by the jaci_timeout parameter in the tibemsd.conf file, the server denies authorization by default.

Actions that require permissions are the same as those listed in the acl.conf file, and include operations such as subscribe to a topic and publishing to a queue. Permissions are described in acl.conf. The following figure shows the decision tree the server follows when granting or denying permissions.



Note: In general, permissions are checked when a client initiates an operation. In the case of a browsing request, it's useful to note that the server reviews permissions only at certain points during the browsing operation.

The server checks for browsing permission when a client starts to browse a queue and whenever the client needs to refresh its list of browse-able messages. The client receives the list of messages from the server when it first begins browsing. The server refreshes the list and rechecks permissions whenever the client browses to the end of the current list.

Durable Subscribers

When a durable subscriber is disconnected from the EMS server, the server continues to accumulate messages for the client. However, while the client is disconnected, there is no user associated with the durable subscriber. Because of this, the server cannot immediately check permissions for a message that is received when the client is not connected.

When a user later reconnects to the server and resubscribes to the durable subscription, the server checks permissions for the subscribe operation itself, but all messages in the backlog are delivered to the consumer without additional permission checks.

Special Circumstances

There are some special circumstances under which the request, although it is not exactly matched in the `acl.conf` file, will be denied without reference to either the permissions cache or the Permissions Module. Any request will be denied if, in the `acl.conf`

- The username exists but is not associated with any destinations.
- The username exists and is associated with destinations, but not with the specific destination in the request.
- The username is part of a group, but the group is not associated with any destinations.
- The username is part of a group and the group is associated with destinations, but not with the specific destination in the request.

In general entries in the `acl.conf` file supersede entries in the Permissions Module, allowing you to optimize permission checks in well-defined static cases. When the `acl.conf` does not mention the user, the Permissions Module is fully responsible for permissions.

Implications of Wildcards on Permissions

A permission result from the Permissions Module can allow or deny the user authorization to perform the action on a range of destinations by including wildcards in the destination name.

For example, even though the application attempts to have user `mwalton` publish on topic `foo.bar.1`, the Permissions Module can grant permission to user `mwalton` to publish messages to the topic `foo.bar.*`. For as long as this authorization is cached, `mwalton` can also publish to the topics `foo.bar.baz` and `foo.bar.boo`, because `foo.bar.*` contains both those topics.

As long as a permission to perform an action on a destination is cached in the allow cache, the user will be authorized to perform that action, even if the permission is revoked in the external system used by the Permissions Module. This permission also extends to any destination contained by the authorized destination through the use of wildcards. The EMS server checks the allow cache for permissions before checking the deny cache and before sending an uncached permission request to the Permissions Module. In other words, the authorization status cannot be changed until the timeout on the cache entry expires and it is removed from the cache.

Similarly, an entry in the deny cache remains there until the timeout has expired and the entry is removed. Only then does the EMS server send the request to the Permissions Module, so that a change in status can take effect.

Overlapping wildcards can make this situation even more complex. For example, consider these three destinations:

```
foo.*.baz
foo.bar.*
foo.>
```

It might seem that, `if foo.*.baz` were in a cache, then `foo.bar.*` would match it and permissions for that destination would come from the cache. In fact, however, permissions could not be determined by the cache entry, because `foo.bar.*` intersects but is not a subset of `foo.*.baz`. That is, not every destination that matches `foo.bar.*` will also match `foo.*.baz`. The destination `foo.bar.boo`, for example, would be granted permissions by `foo.bar.*`, but not by `foo.*.baz`.

Since not all destinations that `foo.bar.*` matches will also match `foo.*.baz`, we say that `foo.*.baz` intersects `foo.bar.*`. The cache entry can determine a permission if the requested destination is a subset of the cache entry, but not if it is merely an intersection. In this case, permissions cannot be determined by the cache.

The destination foo.>, on the other hand, contains as subsets both `foo.bar.*` and `foo.*.baz`, because any destination name that matches either `foo.bar.*` or `foo.*.baz` will also match `foo.>`. If `foo.>` is in the cache, permissions will be determined by the cache.

Enable Extensible Permissions

Extensible permissions are enabled in the EMS server, through parameters in the `tibemsd.conf` configuration file.

The required parameters are:

- `authorization`—enables authorization.
- `jaci_class`—specifies the class that implements the Permissions Module.
- `security_classpath`—specifies the JAR files and dependent classes used by the Permissions Module.

The Permissions Module will be used to grant permissions only to those destinations that are defined as secure in the `topics.conf` and `queues.conf` configuration files. If there are

no topics or queues that include the `secure` property, then the Permissions Module will never be called because the server does not check permissions at all.

Because the Permissions Module runs in the Java virtual machine, you must also enable the JVM in the EMS server. See Enable the JVM for more information.

Permissions Module

The Permissions Module is a custom module that runs inside the EMS server within a JVM. The Permissions Module is written using JACI, a set of APIs developed by TIBCO Software Inc. that you can use to create a Java module that will authorize EMS client requests.

JACI provides the interface between your code and the EMS server. JACI is a standard component of EMS, and JACI classes and interfaces are documented in `com.tibco.tibems.tibemsd.security`.

Requirements

In order to implement extensible permissions, you must write a Permissions Module implementing the JACI interface.

There are some requirements for a Permissions Module that will run in the EMS server:

- The Permissions Module must implement the JACI `Authorizer` interface, which accepts information about the operation to be authorized.
- The Permissions Module must return a permission result, by way of the `AuthorizationResult` class. Permission results contain:
    - An `allowed` parameter, where true means that the request is allowed and false means the request is denied.
    - A timeout, which determines how long the permission result will be cached. Results can be cached for a time of up to 24 hours, or not at all.
    - The destination on which the user is authorized to perform the action. The destination returned can be more inclusive than the request. For example, if the user requested to subscribe to the topic `foo.bar`, the permission result can allow the user to subscribe to `foo.*`. If a destination is not included in the permission result, then the allow or deny response is limited to the originally requested destination.

    - The action type that the permission result replies to. For example, authorization to publish to the destination, or authorization to receive messages from a queue. Permissions can be granted to multiple action types, for example permission to publish and subscribe on `foo.>`. Note that the EMS server creates one cache entry for each action specified in the result.

- The Permissions Module must be thread-safe. That is, the Permissions Module must be able to function both in a multi-threaded environment and in a single-threaded environment.
- The Permissions Module, like the LoginModule, should not employ long operations, and should return values quickly. As these modules become part of the EMS server's message handling process, slow operations can have a severe effect on performance.

Documentation of JACI classes and interfaces is available through the `com.tibco.tibems.tibemsd.security` package.

The JVM in the EMS Server

The Java virtual machine (JVM) is a virtual machine on the Java platform, capable of running inside the EMS server.

Select independent Java modules can operate in the JVM and plug into the EMS server. The JVM is required to use the following TIBCO Enterprise Message Service features:

- Extensible Security—see Extensible Security.
- JAAS Authentication Modules —see JAAS Authentication Modules.

Enable the JVM

The Java virtual machine is enabled in the EMS server, through parameters in the `tibemsd.conf` configuration file.

The parameters that enable and configure the JVM are:

- `jre_library`—enables the JVM.
- `jre_option`—allows you to pass standard JVM options, defined by Sun Microsystems, to the JVM at start-up.

For more information about these parameters, see JVM Parameters and `tibemsd.conf`.

JAAS Authentication Modules

TIBCO provides several compiled and fully functional JAAS modules that can be used to enable LDAP and host-based authentication in the EMS server.

Overview of the JAAS Authentication Modules

The JAAS Authentication modules are LoginModules that use the JVM in the EMS server to authenticate connections to the EMS server.

Refer to Extensible Authentication for further information the use of JAAS in TIBCO Enterprise Message Service.

Prebuilt JAAS Modules

TIBCO Enterprise Message Service provides a number of JAAS modules that can be used with the EMS server. These default modules are very flexible, and offer a variety of configuration options to suit most needs.

An EMS server file, `tibemsd-jaas.conf`, that is preconfigured to use the prebuilt JAAS modules, is located with the other sample configuration files in the *EMS_HOME*`/samples/config` directory.

The module classes are found in *EMS_HOME*`/bin/tibemsd_jaas.jar`, and example module configuration files can be found in *EMS_HOME*`/samples/config/jaas` directory.

The default modules are:

- LDAP Simple Authentication — a simple user authentication scheme using LDAP. This module requires the fewest parameters and is easiest to configure.
- LDAP Authentication — a full featured user authentication scheme using LDAP. This module provides greater functionality and better performance than the LDAP Simple Authentication module.
- LDAP Group User Authentication — a full featured user authentication scheme using

LDAP. An extension of LDAP Authentication, this module also retrieves LDAP group membership information and passes it back into the EMS server, where it may be used for authorization. This modules provides the most functionality but generates more requests to the LDAP server.

- Host Based Authentication — authentication based on the hostname or IP of a user connection. The module is most often used in conjunction with other modules, or in situations where only specific network nodes may authenticate to the EMS server.

Custom JAAS Modules

The default JAAS modules included with your TIBCO Enterprise Message Service installation will accommodate most environments. However, sometimes specialized support for authentication is required.

To support this, well-documented source-code is provided for all of the EMS JAAS modules in the directory:

```
EMS_HOME/src/java/jaas
```

The readme.txt file in that directory contains instructions on compiling the source files.

Multiple JAAS Modules

The prebuilt JAAS modules support stacking, which provides great flexibility. Using multiple modules, you can direct the EMS server to check authentication using any arrangement of the modules.

A common example would stack the LDAP Authentication module with the Host Based Authentication module to authenticate a user by credentials and IP address. Another example would include stacking multiple LDAP Authentication modules to search different branches of an LDAP tree.

There are no restrictions on which or how many modules can be stacked.

For examples of stacking, see Using Multiple JAAS Modules.

Enabling Authentication Using JAAS Modules

The JAAS modules are designed to be simple to use.

A default EMS server configuration file, `tibemsd-jaas.conf`, is located with the other sample configuration files in the *EMS_HOME*`/samples/config` directory.

This file provides a default JAAS configuration that includes the security-related parameters required to use any of the TIBCO EMS JAAS modules. However, some additional steps are required to complete the configuration.

Procedure

1. Configure the JAAS Module

   Create a JAAS module configuration file with parameter values appropriate to your environment.

   If you are using one of the provided default modules, locate the configuration file for the desired module in the `EMS_HOME/samples/config/jaas` directory, and configure the module parameters for your environment. It is a good practice to copy this file along side your other EMS configuration files.

   The prebuilt JAAS modules and their parameters are described in Prebuilt JAAS Modules.

2. Configure the EMS Server Parameters

   The default *EMS_HOME*/samples/config/tibemsd-jaas.conf file is configured for JAAS. This file can be copied as `tibemsd.conf`, or the server can be started with the `-config` parameter to specify this file. See Starting the EMS Server Using Options for details.

   If you prefer to manually configure JAAS, then take the following steps to modify the main EMS server configuration file, `tibemsd.conf`:

   a. Set the `jre_library` parameter to enable the JVM. For more information, see The JVM in the EMS Server.

   b. Set the `security_classpath` parameter to include the following JAR files:

   ```
   EMS_HOME/bin/tibemsd_jaas.jar
   EMS_HOME/lib/tibjmsadmin.jar
   EMS_HOME/lib/tibjms.jar
   EMS_HOME/lib/jakarta.jms-api-2.0.3.jar
   ```

For example:

```
security_classpath = c:\tibco\ems\10.3\bin\tibemsd_
jaas.jar;c:\tibco\ems\10.3\lib\tibjmsadmin.jar;c:\tibco\ems\10
.3\lib\tibjms.jar;c:\tibco\ems\10.3\lib\jakarta.jms-api-
2.0.3.jar
```

c. Set the `jaas_config_file` to reference the JAAS module configuration file created in Step 1.

   For example:

   ```
   jaas_config_file = jaas_configuration.txt
   ```

d. Set the `user_auth` parameter to enable JAAS for LDAP authentication.

   For example:

   ```
   user_auth=jaas
   ```

Prebuilt JAAS Modules

This section provides detailed descriptions of the prebuilt JAAS modules.

Configuration files for these modules are provided in the *EMS_HOME*`/samples/config/jaas` directory.

For the LDAP modules, properties added in the JAAS configuration file that do not begin with `tibems` are passed into every LDAP context creation, allowing LDAP-specific parameters to be set in the JAAS configuration file.

Properties that must be set in the environment, such as TLS related properties, are configured through the `jre_option` parameter in the EMS server configuration. However, a TLS key store location can be set using the `tibems.ldap.truststore` parameter for convenience. See the parameter descriptions for each module type for details.

LDAP Simple Authentication

The LDAP Simple Authentication module implements a very basic form of LDAP authentication. The module validates all connections (users, routes, and so on) by

authenticating to the LDAP server. The authentication process uses the name and password that the application used when connecting to the EMS server.

The user name must be in the form of a distinguished name, unless a user name pattern is supplied through the `tibems.ldap.user_pattern` parameter. When a user pattern is supplied, the DN used for the lookup is that pattern string, with `%u` replaced with the name of the user.

Authentication Process

The simple authentication login module creates a local LDAP context, binding to the LDAP server as a particular user with credentials from the incoming connection. The result of the bind dictates authentication success or failure.

Implementation

The LDAP Simple Authentication module name is:
`com.tibco.tibems.tibemsd.security.jaas.LDAPSimpleAuthentication`

The JAAS configuration file entry for this login module should have a section similar to the following:

```
EMSUserAuthentication {
  com.tibco.tibems.tibemsd.security.jaas.LDAPSimpleAuthentication required
  tibems.ldap.url="ldap://ldapserver:389"
  tibems.ldap.user_pattern="CN=%u" ;
};
```

Parameters

The LDAP Simple Authentication Module parameters are listed in the following table.

Parameter	Description
debug	When set to `true`, enables debug output for the module. Enabling this parameter may aid in diagnosing configuration problems.

Parameter	Description
	Warning: Enabling the debug flag may create security vulnerabilities by revealing information in the log file. The default setting is `false`.
`tibems.ldap.operation_timeout`	The timeout, in milliseconds, set for LDAP connect and LDAP read operations. If not set, these two LDAP operations will follow their default behavior.
`tibems.ldap.truststore`	The key store that is used for TLS connections. On Windows, the trust store must use forward slashes or escape backslashes when specifying a path.
`tibems.ldap.url`	The location of the LDAP server. Specify a single URL or comma-separated list of URLs. Each URL must use the format described by RFC 2255. The server configuration can be defined as a single URL, or as a series of LDAP URLs representing the primary and backups servers. To configure a backup, provide a comma-separated list of URLs. For example: `ldap://localhost:389,ldap://localhost:489` The servers are attempted in the order listed. Should the first server in the list be unavailable or fail, the next URL is tried. Any number of backup servers may be specified. The default is `ldap://localhost:389`.
`tibems.ldap.user_pattern`	The user pattern to use with simple LDAP authentication. When a user pattern is supplied, the DN used for

Parameter	Description
	the lookup will be this pattern string entered here, with '%u' replaced with the name of the user. For example, `uid=%u;ou=People`. The default pattern is `CN=%u`.

LDAP Authentication

The LDAP Authentication login module is a more fully featured LDAP authentication module. This module validates all connections (users, routes, and so on) by authenticating to the LDAP server using the supplied credentials.

This EMS JAAS module keeps one lookup context open using a manager context, and then uses copies of that context to search for users. This allows the LDAP implementation to reuse the connection for subsequent searches, improving performance.

Authentication Process

This implementation queries LDAP, and optionally a user cache, to authenticate a user. A context with LDAP manager credentials is first used to look up a user and retrieve the complete distinguished name of the user's entry. If the user exists, a separate LDAP context is then created to authenticate the user. For performance reasons, the manager context, once created, exists for the lifetime of the module.

Should connectivity with the LDAP server break, multiple reconnection attempts may be made based on the parameters.

To increase performance, you can enable user caching. When enabled, a user is added to the user cache after being authenticated though LDAP. This allows for faster authentication on subsequent logins. If the user cache entry is found to be expired, the user is authenticated with LDAP again and the cache is updated.

Implementation

The LDAP Authentication module name is:
`com.tibco.tibems.tibemsd.security.jaas.LDAPAuthentication`.

The JAAS configuration file entry for this login module should have a section similar to the following:

```
EMSUserAuthentication {
 com.tibco.tibems.tibemsd.security.jaas.LDAPAuthentication required
 tibems.ldap.url="ldaps://ldapserver:391"
 tibems.ldap.truststore="/certificates/cacerts"
 tibems.ldap.user_base_dn="ou=Marketing,dc=company,dc=com"
 tibems.ldap.user_attribute="uid"
 tibems.ldap.scope="subtree"
 tibems.cache.enabled=true
 tibems.cache.user_ttl=600
 tibems.ldap.manager="CN=Manager"
 tibems.ldap.manager_password="password" ;
};
```

Parameters

The LDAP Authentication Module parameters are listed in the following table.

Parameter	Description
debug	When set to `true`, enables debug output for the module. Enabling this parameter may aid in diagnosing configuration problems. **Warning**: Enabling the debug flag may create security vulnerabilities by revealing information in the log file. The default setting is `false`.
tibems.ldap.operation_timeout	The timeout set for LDAP connect and LDAP read operations. The property is specified in milliseconds. If not set, these two LDAP operations will follow their default behavior.
tibems.ldap.truststore	The key store that is used for TLS connections. On Windows, the trust store must use forward slashes or escape backslashes when specifying a path.

Parameter	Description
`tibems.ldap.url`	The location of the LDAP server. Specify a single URL or comma-separated list of URLs. Each URL must use the format described by RFC 2255. The server configuration can be defined as a single URL, or as a series of LDAP URLs representing the primary and backups servers. To configure a backup, provide a comma-separated list of URLs. For example: `ldap://localhost:389,ldap://localhost:489` The servers are attempted in the order listed. Should the first server in the list be unavailable or fail, the next URL is tried. Any number of backup servers may be specified. The default is `ldap://localhost:389`.
`tibems.ldap.user_base_dn`	The base DN used for the LDAP search. For example: `ou=People,dc=TIBCO,dc=com`
`tibems.cache.enabled`	When true, enables caching of user information for better performance. The default is `false`.
`tibems.cache.instance`	A string that represents an instance of the user cache. When stacked login modules specify the same instance, they share the same user cache as a form of optimization. The default is a unique cache based on the values of the `tibems.ldap.url`, `tibems.ldap.user_base_dn`, and `tibems.ldap.user_attribute` parameters.
`tibems.cache.user_ttl`	Specifies the maximum time (in seconds) that cached LDAP data is retained before it is refreshed. The default is `60`.
`tibems.ldap.user_filter`	The filter used when searching for a user.

Parameter	Description
	If a more complex filter is needed, use this property to override the default. Any occurrence of `{0}` in the search string will be the user attribute, and `{1}` will be replaced with the user name. The default is `{0}={1}`.
`tibems.ldap.manager`	The distinguished name of the user that this module uses when binding to the LDAP server to perform a search. The specified user must have permissions to search LDAP for users under the entry specified by `tibems.ldap.user_base_dn`. The default is `CN=Manager`.
`tibems.ldap.manager_password`	The password used when binding to the LDAP server as the manager. This password may be mangled using the EMS Administration Tool.
`tibems.ldap.retries`	The number of times that the module should reattempt a connection if there is a communication failure with the LDAP server. If one or more backup severs are specified in `tibems.ldap.url`, this parameter determines the number of times the EMS server iterates through the list of backup LDAP servers. The default value is `0`, meaning no retries are attempted.
`tibems.ldap.retry_delay`	The module waits this number of milliseconds before retrying the connection to the LDAP server. The default is `1000`.
`tibems.ldap.scope`	The scope of the search. Valid values include: • `onelevel` • `subtree`

Parameter	Description
	• `object` The default is to use a one level search.
`tibems.ldap.user_attribute`	The attribute that is compared to the user name for the search. The default is `uid`.

LDAP Group User Authentication

The LDAP Group User Authentication module extends the full featured LDAP Authentication module and provides additional group information to the EMS server. This module validates all connections (users, routes, and so on) by authenticating to the LDAP server using the supplied credentials, and then updates the EMS server with any related group information found.

If caching is enabled, changes to group membership in the LDAP server are not reflected in EMS until the user's entry in the cache has expired.

Authentication Process

The Group User LDAP module authenticates a user just as the LDAP Authentication module does, but will make additional requests to garner group membership information from LDAP and update the EMS server for authorization purposes.

For example, consider a user "Joe", who belongs to the "Engineering" group in the LDAP server. When an application connects to the EMS server using Joe's credentials, the information that Joe belongs to the Engineering group is passed back up to the server after a successful authentication. If access controls are set up in EMS for the group Engineering, then Joe inherits those permissions.

Implementation

The LDAP Group User Authentication module name is:
`com.tibco.tibems.tibemsd.security.jaas.LDAPGroupUserAuthentication`

The JAAS configuration file entry for this module should have an entry similar to:

```
EMSUserAuthentication {
    com.tibco.tibems.tibemsd.security.jaas.LDAPGroupUserAuthentication required
    tibems.ldap.url="ldap://ldapserver:389"
    tibems.ldap.user_base_dn="ou=Marketing,dc=company,dc=com"
    tibems.ldap.user_attribute="uid"
    tibems.ldap.scope="subtree"
    tibems.ldap.group_base_dn="ou=Groups,dc=company"
    tibems.ldap.group_member_attribute="uniqueMember"
    tibems.ldap.dynamic_group_base_dn="ou=Groups,dc=company"
    tibems.ldap.dynamic_group_class="groupOfURLs"
    tibems.ldap.dynamic_group_member_attribute="uid"
    tibems.ldap.dynamic_group_filter="(objectClass=GroupOfURLs)"
    tibems.cache.enabled=true
    tibems.cache.user_ttl=600
    tibems.ldap.manager="CN=Manager"
    tibems.ldap.manager_password="password" ;
};
```

Parameters

In addition to all parameters available for the LDAP Authentication module, which are described in the following table, the following parameters are supported:

Parameter	Description
`tibems.ldap.group_attribute`	The attribute of a static LDAP group that contains the group name. Default is `cn`.
`tibems.ldap.group_base_dn`	The base path for the LDAP static group search. If null or not set, static groups are not searched.
`tibems.ldap.group_filter`	The filter used in the static group search. By default, a filter is created using the `ems_ldap.group_member_attribute` parameter. If a more complex filter is needed, use this property to override the default. Any occurrence of `{0}` in the search string is

Parameter	Description
	replaced with the group member attribute. Any occurrence of `{1}` is replaced with the user DN. `{2}` contains solely the user name for cases where the DN does not match group membership. Default is `{0}={1}`.
`tibems.ldap.group_member_attribute`	The attribute ID of a dynamic LDAP group object that specifies the name of members of the group. Default is `uniqueMember`.
`tibems.ldap.group_scope`	The scope of the static group search. Valid values include `onelevel`, `subtree`, and `object`. Default is to use a subtree search.
`tibems.ldap.dynamic_group_base_dn`	Base path for the LDAP dynamic group search. If null or not set, dynamic groups are not searched.
`tibems.ldap.dynamic_group_class`	The class name of a dynamic group. Default is `groupOfURLs`.
`tibems.ldap.dynamic_group_attribute`	The attribute of an LDAP dynamic group that contains the group name. Default is `cn`.
`tibems.ldap.dynamic_group_filter`	The filter used in the dynamic group search. By default, a filter is created using the `ems_ldap.dynamic_group_member_attribute` property. If a more complex filter is needed, use this property to override the default. Any occurrence of `{0}` is replaced with the group

Parameter	Description
	member property. Any occurrence of `{1}` is replaced with the DN of the user for cases where that may be required. A `{2}` in the search string is replaced with the user name. When using `tibems.ldap.dynamic_group_search_direct`, a simple filter should be used which matches all dynamic groups that may contain the user. For example, `(objectClass=GroupOfURLs)`. Default is `{0}={1}`.
`tibems.ldap.dynamic_group_member_attribute`	The attribute ID of a dynamic LDAP group object that specifies the name of members of the group. Default is `uniqueMember`.
`tibems.ldap.dynamic_group_member_url`	The attribute of a dynamic LDAP group object that specifies the URL generating the membership list. Default is `memberURL`.
`tibems.ldap.dynamic_group_scope`	The scope of the dynamic group search. Valid values include `onelevel`, `subtree`, and `object`. Default is to use a subtree search.
`tibems.ldap.dynamic_group_search_direct`	Changes the search algorithm used for determining membership of dynamic groups. Normally, LDAP servers automatically populate dynamic groups based on a configured search URL. However, some LDAP servers have issues where the generated

Parameter	Description
	attributes representing members of the groups are not properly returned by a search. When enabled, this parameter changes the group search algorithm to parse out a DN, scope, and filter from the search URL specified by the dynamic group and use those to search for a user. Use of this parameter is only recommended when it has been determined that dynamic group searches are not working. Default is `false`.
`tibems.ldap.backlink_group_base_dn`	The base path for the back-linked LDAP group search. By default, back-linked group searches are not enabled. If enabled, back-linked groups, including nested groups, are searched using back link parameters. To disable nested searches for back links, set `tibems.ldap.nested_groups_enabled` to false. Back link parameter defaults are set for use with Active Directory, the most commonly used LDAP server supporting back links.
`tibems.ldap.backlink_group_attribute`	The attribute that contains the groups an LDAP object (member or group) belongs to. Default is `memberOf`.
`tibems.ldap.backlink_group_rdn`	A back-link RDN that specifies the name portion of the DN representing the group. If the entire contents of the back link value is to be used as the group name, do not set this value.

Parameter	Description
	Default is `CN`.
`tibems.ldap.backlink_group_filter`	A back-link filter used by a group search to find groups the member belongs to. If nested groups are not used, then it is highly advisable to disable nested groups. Default is `(distinguishedName={1})`.
`tibems.ldap.backlink_group_scope`	The scope of the back link group search. Valid values include `onelevel`, `subtree`, and `object`. Default is to use a subtree search.

Host Based Authentication

The Host Based Authentication module authenticates a user based on the IP address or host name that is associated with their client connection during authentication.

When enabled, the IP address of the incoming connection is evaluated against a whitelist of IP addresses and/or IP masks. If any of the IP addresses or masks result in a match, IP authentication for the user is considered successful.

If an IP match is not found, then the host name of the incoming connection is compared with the configured whitelist of patterns, which may be specific host names or regular expressions. If the connection's host name evaluates to true with any of the patterns in the list, authentication is considered successful.

Either the host name *or* IP mask must match for authentication success.

Authentication Process

When a client connects to the EMS server, this module compares the IP address with the specified IP net/prefix list, if configured. If that is not successful, then the hostname is compared with the list of hostnames or domain names. Should none of the above succeed, authentication fails.

Warning: If hostname verification is configured, the module may do a DNS lookup. This could impact performance.

Implementation

The Host Based Authentication module name is:
`com.tibco.tibems.tibemsd.security.jaas.HostBasedAuthentication`

The JAAS configuration file entry for this login module should have a section similar to the following:

```
EMSUserAuthentication {
    com.tibco.tibems.tibemsd.security.jaas.HostBasedAuthentication required
    tibems.hostbased.accepted_hostnames="'production.*','.tibco.com'"
    tibems.hostbased.accepted_addresses"10.1.2.23, 10.100.0.0/16, 0:0:0:0:0:0:0:1"
};
```

Parameters

The Host Based Authentication Module parameters are listed in the following table.

Parameter	Description
debug	When set to `true`, enables debug output for the module. Enabling this parameter may aid in diagnosing configuration problems. **Warning**: Enabling the debug flag may create security vulnerabilities by revealing information in the log file. The default setting is `false`.
`tibems.hostbased.accepted_hostnames`	A comma delimited list of host names or patterns to compare with the incoming connection's host name, as known by the EMS server. A match results in successful authentication.

Parameter	Description
	Host names or domains can be explicitly specified, or any regular expression working with the Java Pattern class may be used. A domain may be used by beginning the string with a dot (.). Each host-name or pattern must be encapsulated by a single quote and separated by a comma. These entries are compared with the hostname associated with the IP of the connecting EMS client. **WARNING**: This could have a performance impact as a NIS or DNS lookup may be performed. If this property is not set, host names are not checked during authentication. For example: `'host1', '.tibco.com', '^.*_SERVER\\.tibco\\.com'`
`tibems.hostbased.accepted_addresses`	A comma delimited list of IP addresses or net/prefix (CIDR notation) masks to compare with the incoming connection's IP address. Both IPV4 and IPV6 are supported. Any match results in successful authentication. If this property is not set, IP address checking is disabled. For example: `10.1.2.23, 10.100.0.0/16, 0:0:0:0:0:0:0:1`

Connection Limit Authentication

The Connection Limit Authentication module limits the number of active connections a user can have at any one time.

Authentication Process

When a client connects, the user name is identified and then authenticated based on the number of connections open for that user. If the number of connections is less than the configured limit, the user is authenticated successfully, and the internal connection count is incremented. When a user disconnects, the internal connection count is decremented.

A client's user name can be specified as one of the following types: hostname, IP address, LDAP ID, or LDAP ID and hostname.

Note: If you plan on stacking this module with other JAAS modules, it is important to use this as the final JAAS module and to list all of the JAAS modules as 'requisite'. This ensures that the internal connection count of the Connection Limit Authentication module remains accurate.

Implementation

The Connection Limit Authentication module name is:
`com.tibco.tibems.tibemsd.security.jaas.ConnectionLimitAuthentication`

The JAAS configuration file entry for this login module should have a section similar to the following:

```
EMSUserAuthentication {
  com.tibco.tibems.tibemsd.security.jaas.ConnectionLimitAuthentication required
  tibems.connectionlimit.max_connections="5"
  tibems.connectionlimit.type="HOSTNAME" ;
};
```

Parameters

The Host Based Authentication Module parameters are listed in the following table.

Parameter	Description
debug	When set to `true`, enables debug output for the module. Enabling this parameter may aid in diagnosing

Parameter	Description
	configuration problems. **Warning**: Enabling the debug flag may create security vulnerabilities by revealing information in the log file. The default setting is `false`.
`tibems.connectionlimit.max_connections`	An integer to indicate the number of connections allowed per user.
`tibems.connectionlimit.type`	Identifies the type of user for an incoming connection. For example: "`HOSTNAME`", "`IP`", "`LDAPID`", or "`LDAPID@HOSTNAME`".

Using Multiple JAAS Modules

You can stack the provided JAAS modules to suit your environment and authentication needs. There are no restrictions on which or how many modules can be stacked.

To stack multiple JAAS modules, include the desired module configurations and JAAS flags in the same configuration file that is reference by the
JAAS configuration parameter, `jaas_config`.

The behavior and authentication requirements of the included modules are controlled by the module *Flag* value assigned to each module in the stack. For more information, see the Oracle `javax.security.auth.login.Configuration` Class documentation for information on using multiple JAAS modules.

Example: Two Authentication Requirements

In this example, a user is authenticated based on network location. If that succeeds, the user is then authenticated using LDAP credentials. Both must succeed for the user to be authenticated.

This behavior is controlled by the requisite *Flag*.

```
EMSUserAuthentication {
   com.tibco.tibems.tibemsd.security.jaas.HostBasedAuthentication requisite
   tibems.hostbased.accepted_addresses="10.98.48.45, ::1"
   tibems.hostbased.accepted_hostnames="'jsmith.*','.tibco.com'";
   com.tibco.tibems.tibemsd.security.jaas.LDAPSimpleAuthentication requisite
   tibems.ldap.user_pattern="uid=%u,ou=People,dc=tibco.com"
   tibems.ldap.url="ldap://localhost:389" ;
};
```

Example: One Authentication is Sufficient

In this example, a user is authenticated against multiple LDAP branches. If authentication fails in the first branch, the second is tried. Only one module instance needs to succeed for the user to be authenticated.

This behavior is controlled by the sufficient *Flag*.

```
EMSUserAuthentication {
    com.tibco.tibems.tibemsd.security.jaas.LDAPSimpleAuthentication sufficient
    tibems.ldap.user_pattern="uid=%u,ou=People,dc=Local"
    tibems.ldap.url="ldap://localhost:389" ;
    com.tibco.tibems.tibemsd.security.jaas.LDAPSimpleAuthentication sufficient
    tibems.ldap.user_pattern="uid=%u,ou=People,dc=Remote"
    tibems.ldap.url="ldap://localhost:389" ;
};
```

Migrating to the EMS JAAS Modules

Servers earlier than EMS 10.0 used to support user LDAP authentication within the EMS server through a set of server properties that started with `ldap_`. Migrating from this now unsupported authentication mechanism to the JAAS modules is relatively straightforward. Many of the parameters directly map to each other. Nevertheless, there are some differences and so care must still be taken.

The LDAP Group User Authentication module provides similar functionality to that of the pre-EMS 10.0 server. However, if group membership is not required for authentication, then the LDAP Authentication module is a better choice.

Former EMS Server LDAP Parameter to JAAS Module Parameter Mapping

When parameters have an exact equivalent, as indicated in the notes column, the same values from the Former EMS Server LDAP parameters can be used in the JAAS modules, except that the JAAS modules expect parameter values to be enclosed in quotes.

Former EMS Server LDAP Parameter	EMS JAAS Equivalent	Notes
`ldap_url`	`tibems.ldap.url`	Exact
`ldap_principal`	`tibems.ldap.manager`	Exact
`ldap_credential`	`tibems.ldap.manager_password`	Exact
`ldap_cache_enabled`	`tibems.cache.enabled`	Exact
`ldap_cache_ttl`	`tibems.cache.user_ttl`	Exact
`ldap_conn_type`	`tibems.ldap.url`	See ldap_conn_type below.
`ldap_tls_cacert_file`	`tibems.ldap.truststore`	See ldap_tls Parameters.
`ldap_tls_cacert_dir`	`tibems.ldap.truststore`	See ldap_tls Parameters.
`ldap_tls_cipher_suite`	N/A	See ldap_tls Parameters.
`ldap_tls_rand_file`	N/A	See ldap_tls Parameters.
`ldap_tls_cert_file`	`tibems.ldap.truststore`	See ldap_tls Parameters.
`ldap_tls_key_file`	`tibems.ldap.truststore`	See ldap_tls Parameters.
`ldap_user_class`	`tibems.ldap.user_filter`	See ldap_user_class and ldap_static_group_class.
`ldap_user_attribute`	`tibems.ldap.user_attribute`	Exact
`ldap_user_base_dn`	`tibems.ldap.user_base_dn`	Exact
`ldap_user_scope`	`tibems.ldap.scope`	Exact
`ldap_user_filter`	`tibems.ldap.user_filter`	See Filters.

Former EMS Server LDAP Parameter	EMS JAAS Equivalent	Notes
ldap_group_base_dn	tibems.ldap.group_base_dn	Exact
ldap_group_scope	tibems.ldap.group_scope	Exact
ldap_group_filter	tibems.ldap.group_filter	See Filters.
ldap_all_groups_filter	N/A	See Filters.
ldap_static_group_class	tibems.ldap.group_filter	See ldap_user_class and ldap_static_group_class.
ldap_static_group_attribute	tibems.ldap.group_attribute	Exact
ldap_static_group_member_filter	tibems.ldap.group_filter	See Filters.
ldap_static_member_attribute	tibems.ldap.group_member_attribute	Exact
ldap_dynamic_group_class	tibems.ldap.dynamic_group_class	Exact
ldap_dynamic_group_attribute	tibems.ldap.dynamic_group_attribute	Exact
ldap_dynamic_member_url_attribute	tibems.ldap.dynamic_group_member_url	Exact

Parameters Requiring Conversion

ldap_conn_type

The connection type is indirectly supported by the JAAS modules through the protocol portion of the LDAP URL.

- `ldap://` creates a TCP connection.
- `ldaps://` creates a TLS connection.

If the `startTLS` LDAP extension is required, additional JNDI parameters may be specified through the JAAS configuration. Alternately, you can customize the JAAS module. See Custom JAAS Modules for more information.

ldap_tls Parameters

The JAAS modules have the ability to pass any parameters to JNDI. It is up to the user to determine what java TLS parameters to pass to JNDI through the JAAS configuration.

In most cases, only a certificate key store is required. For convenience, the `tibems.ldap.truststore` parameter can be used to specify the store. Refer to Java documentation for additional information regarding the use of TLS.

Filters

Filters perform the same function in the JAAS modules as they do when LDAP authentication is configured within the EMS server, but the specification of the filter parameters is slightly different.

Be sure to substitute the EMS server's `%s` filters for the appropriate `{n}` JAAS module filter.

ldap_user_class and ldap_static_group_class

The `ldap_user_class` and `ldap_static_group_class` parameters are not necessary in the JAAS modules.

LDAP class names are specified in the filters, as in the following examples:

```
tibems.ldap_user_filter="(&({0}={1})(objectClass=uniqueMember))"
```

and

```
tibems.ldap.group_filter="(&({0}={1})(objectClass=groupofUniqueNames))"
```

Refer to the filter documentation to map various identifiers. For example, in converting the user filter, the former EMS server LDAP parameter, `%s` maps to `{1}` in the JAAS filter. Many group searches should work with a filter similar to:

```
(&{0}={1})(objectClass=<group class>)
```

However, dynamic groups do allow you to specify the class in order to mirror the search algorithm used by the former EMS server native LDAP functionality.

Dynamic Groups

Dynamic groups in LDAP should normally behave similarly to static groups in LDAP. However, some LDAP implementations require a modified search algorithm.

In order to perform this type of search with the JAAS modules, set the parameter:

```
tibems.ldap.dynamic_group_search_direct="true"
```

It is recommended this is parameter be enabled after you have determined that there is a problem, or when using an OpenLDAP server. In some cases, this is required in order to mirror the former EMS server native LDAP functionality.

Example

This section provides a walk through converting an existing set of pre-EMS 10.0 LDAP parameters using the LDAP Group User Authentication login module.

1. Set the `jre_library` parameter to enable the JVM.

   For more information, see The JVM in the EMS Server.

2. Set the `security_classpath`.

For example:

```
security_classpath =
c:\tibco\ems\10.3\bin\tibemsd_jaas.jar;
c:\tibco\ems\10.3\lib\tibjmsadmin.jar;
c:\tibco\ems\10.3\lib\tibjms.jar;c:\tibco\ems\10.3\lib\jakarta.jms-
api-2.0.3.jar
```

3. Enable JAAS for LDAP authentication by modifying the `user_auth` parameter. Remove `ldap` from the list of authentication sources, and verify that `jaas` is present.

   For example:

   ```
   user_auth=jaas
   ```

4. Edit the provided `com.tibco.tibems.tibemsd.security.jaas.LDAPGroupUserAuthentication` module for your LDAP server configuration:

   a. Locate the sample configuration file `ems_ldap_with_groups.txt` in *EMS_HOME*`\samples\config\jaas`.

   b. Copy the file to a secure location, ideally alongside the other EMS server configuration files.

5. Set the `jaas_config_file` to reference the JAAS module configuration file created in Step 4 above.

   For example:

   ```
   jaas_config_file = ems_ldap_with_groups.txt
   ```

LDAP Parameters in the tibemsd.conf

Consider the following LDAP server configuration parameters in the EMS server configuration file, `tibemsd.conf`:

```
ldap_url                    = ldap://ldaphost:389
ldap_principal              = cn=Manager
ldap_credential             =$man$fPSdYgyVTQloUv36Km36AEOrARW
ldap_user_class             = person
ldap_user_attribute         = uid
ldap_user_base_dn           = "ou=People,dc=TIBCO"
```

```
ldap_user_scope              = subtree
ldap_user_filter             = "(&(uid=%s)(objectclass=person))"
ldap_group_base_dn           = "ou=Groups,dc=TIBCO"
ldap_group_scope             = subtree
ldap_group_filter            = "(&(cn=%s)(objectclass=groupOfUniqueNames))"
ldap_static_group_class      = groupOfUniqueNames
ldap_static_group_attribute  = cn
ldap_static_member_attribute = uniqueMember
ldap_cache_enabled           = FALSE
```

Mapped to LDAP Group User Authentication Module

The LDAP configuration parameters shown above map to the following JAAS configuration file:

```
EMSUserAuthentication  {
  com.tibco.tibems.tibemsd.security.jaas.LDAPGroupUserAuthentication required
  tibems.ldap.url="ldap://ldaphost:389"
  tibems.ldap.manager="cn=Manager"
  tibems.ldap.manager_password="$man$fPSdYgyVTQloUv36Km36AEOrARW"
  tibems.ldap.user_attribute="uid"
  tibems.ldap.user_base_dn="ou=People,dc=TIBCO"
  tibems.ldap.scope="subtree"
  tibems.ldap.user_filter="(&(uid={1})(objectclass=person))"
  tibems.ldap.group_base_dn="ou=Groups,dc=TIBCO"
  tibems.ldap.group_scope="subtree"
  tibems.ldap.group_filter="(&({0}={1}(objectclass=groupOfUniqueNames))"
  tibems.ldap.group_attribute="cn"
  tibems.ldap.group_member_attribute="uniqueMember"
  tibems.ldap.cache.enabled = "false" ;
};
```

Troubleshooting Problems in the JAAS Modules

In order to troubleshoot JAAS modules,

Procedure

1. Add JAAS to the EMS server trace options in the main server configuration file:

```
console_trace = DEFAULT,+JAAS,+JVM,+JVMERR
```

2. Enable debugging in the JAAS module itself, by setting the debug parameter to true:

```
EMSUserAuthentication {
    com.tibco.tibems.tibemsd.security.jaas.LDAPSimpleAuthentication required
    debug="true"
    tibems.ldap.url="ldap://ldapserver:389"
    tibems.ldap.user_pattern="CN=%u"
};
```

Warning: Note that enabling the debug flag may create security vulnerabilities by revealing information in the log file. This parameter should be enabled only for troubleshooting purposes.

Result

This will provide a list of parameters passed into LDAP, which is useful in identifying any mistyped parameters or default values that need to be changed. Verbose output is provided to help identify the problem.

When developing a custom JAAS module, it is possible for a runtime exception inside a JAAS method to cause the JAAS module to fail. In those cases, catching and printing exceptions to the default output stream provides valuable information.

Grid Stores

You can configure TIBCO Enterprise Message Service to store messages, state information, and configuration information in supported versions of TIBCO ActiveSpaces.

The following topics describe grid stores. For information about other store types, see Store Messages in Multiple Stores and FTL Stores.

Note: The EMS server supports grid stores only on Linux.

Grid Stores Overview

Grid stores are designed to achieve a minimal EMS server memory footprint and quick EMS server recovery time upon failover. When configured to use grid stores, the majority of server data is stored in an ActiveSpaces data grid and is read into memory only on-demand. A small portion of the information may be cached to speed up message processing, but the remainder is removed from memory once relevant operations are completed. This approach decouples the EMS server's memory usage and failover time from the size of its stores.

When using grid stores, persistent message data and state information are always written to the ActiveSpaces data grid. However, non-persistent data is still stored in memory in most cases. Non-persistent messages can be moved to the data grid if message swapping is enabled. When message swapping is enabled and the maximum message memory limit or the destination swap out threshold have been exceeded, non-persistent messages will be swapped from memory to the data grid. The impact of non-persistent messages on server memory can be reduced by setting a low value for `max_msg_memory` or `destination_backlog_swapout` parameters.

This storage-centered design of grid stores lends itself to quick server start-up times. As opposed to other store types, the entirety of each store's contents is not read upon server start-up or failover. Instead, the server continuously performs incremental scans of the grid stores in the background. This allows for much faster server recovery when the store sizes have grown very large.

The server scans through its grid stores incrementally in the background and discards stale data, such as purged and expired messages. As a result, purged and expired messages are not immediately removed, and may remain in a grid store longer than they would in a file-based or FTL store - although they are not delivered to consumers. The scanning behavior is determined by parameter settings in the store's configuration, and is further described in Understanding Grid Store Intervals.

A full background scan of the grid stores must be completed in order to obtain the correct overall statistics. Due to this, querying the server for a total pending message count before the grid stores have been fully scanned may return an inaccurate value. However, querying specific destinations, consumers or durables will return an accurate count. See Implications for Statistics for more information.

The latency costs in communicating with ActiveSpaces can make grid stores slower than file-based stores or FTL stores. The strength of grid stores lies with their scalability and consistent recovery time regardless of store size.

Fault-Tolerance with Grid Stores

An ActiveSpaces data grid deployment provides data persistence and replication capabilities that support Shared State fault-tolerance in EMS. For information on how to configure data replication in ActiveSpaces, see the TIBCO ActiveSpaces Concepts product guide.

When using grid stores, a fault-tolerant EMS server pair's configuration is stored within an ActiveSpaces data grid. This allows the configuration to be accessible to EMS servers running in separate machines or containers. Static configuration elements including JAAS modules, JAAS module configuration files, JACI modules and digital certificates are not stored in the data grid and must still be maintained manually.

Understanding Grid Store Intervals

Grid stores are designed to ensure a quick EMS server start-up time. To enable this functionality, the EMS server must continually monitor stores in the background. The server reads through grid stores incrementally and discards stale data, such as purged and expired messages.

In order to keep the background activity from degrading server performance, the examination is performed in increments. The length of these increments and the amount of

data processed each increment are controlled by two parameter settings. These parameters can be configured for each grid store.

The default parameter settings are optimized for best performance in most production environments (see Configuring Grid Stores for information about the default values). However, if the amount of data in a grid store grows significantly, the read rates associated with the background activity may begin to affect message transmission rates in the EMS server. If the EMS server performance is negatively affected by the size of the grid store, you can tune the grid store parameter values to spread grid store background activity over a longer period of time, thereby decreasing the associated read rates.

- `scan_target_interval`: the maximum amount of time allowed before each message in the store is examined.

  For example, if the `scan_target_interval` is 24 hours, each section of the grid store will be examined at least once every day. Because purged and expired messages are not removed from the grid store until they are examined by this background process, this means that it can take up to 24 hours before a message is removed from the queue following a purge command (making underlying storage space available for re-use).

- `scan_iter_interval`: the length of time between each increment of background activity.

  For example, if the `scan_iter_interval` is 10 seconds, the EMS server begins examining a new section of the grid store every 10 seconds. The amount of data read in each increment is dependent on the total size of the store and the length of the `scan_target_interval`. The server must examine enough data in each interval to fully traverse the store within the target interval.

Example

For example, assume that `scan_iter_interval` is 10 seconds, `scan_target_interval` is 1 day (86,400 seconds), and the grid store contains 9 GB of data. Every 10 seconds, the EMS server will examine about 1 MB of data. This produces an average read rate of about 100 KB/sec, which is unlikely to produce performance degradation with most modern storage mediums.

If EMS server performance does slow, you may need to increase the `scan_target_interval` value in order to spread the background activity over a longer period of time. You can monitor the settings for problems using the show store command and checking the ratio of "Discard Scan Interval Bytes" to "Discard Scan interval". For best results, this ratio should be kept below 20% of the system capacity. Adhering to this ratio will help ensure that the background activity does not occupy an excessive amount of system resources.

Implications for Statistics

The background monitoring and cleanup that occurs in the grid store also affects some key server statistics. Before the first scan has been completed for all grid stores, some message statistics reported by the server may be inaccurate.

For example, when the EMS server first starts, the "Pending Messages" and "Pending Message Size" counts reported by the info command in the administration tool can be understated, because the command only reports on messages it has scanned before the command is issued. Similarly, the "Message Count" and "Message Size" reported by the show store command may report a smaller number than actually exist in the store.

Once the first scan is complete, these counts can be considered accurate. To check the scan status on a grid store, use the show store command. The statistics returned include a "First scan finished" field, which reports the scan status since the last EMS server start time. When the value of this field is true, the server statistics can be considered accurate.

If it is important to acquire the correct values for these statistics sooner, you will need to decrease the `scan_target_interval`.

Configuring and Deploying Grid Stores

This section describes the steps required to configure and deploy grid stores. The ability to use grid stores is contingent upon the deployment of a TIBCO ActiveSpaces data grid or the availability of an existing TIBCO ActiveSpaces data grid.

Deploying a Simple TIBCO ActiveSpaces Data Grid

The instructions in this section outline the steps involved in running grid stores with a minimal ActiveSpaces data grid setup that is suitable for a development environment. For information on designing, configuring and deploying an ActiveSpaces data grid that is suitable for your production environment, refer to the TIBCO ActiveSpaces Concepts and TIBCO ActiveSpaces Administration product guides.

Before you begin

- TIBCO FTL must already be installed on the host machine that is to run the FTL

server.

- TIBCO ActiveSpaces must already be installed on all host machines that are to run the state keeper or node processes.

Overview

An ActiveSpaces data grid suitable for use with grid stores is composed of the following components:

- Realm service (runs as part of a TIBCO FTL server)
- Administrative daemon
- State keeper
- Node
- Proxy

Embedded Proxy

The ActiveSpaces proxy facilitates communication between the EMS server and the data grid. While the EMS server requires an external proxy process to connect to the data grid during startup, all subsequent communication between the server's grid stores and the data grid is carried out via an ActiveSpaces proxy embedded directly in the server. Using this embedded proxy improves grid store performance by eliminating any would-be server to proxy communication latency. Note that the embedded proxy must be included in the data grid definition. See the Defining the Data Grid and Component Processes section for details.

Starting a FTL server

This section describes the steps to bring up the FTL server.

Procedure

1. Navigate to an empty directory that can be used as the realm configuration data directory.

```
cd data_dir_1
```

The FTL server uses the current directory as the default location to store its working data files. When the FTL server detects an empty working directory, it begins with a default realm definition.

If you have already begun to configure the realm definition, then navigate to your existing data directory instead.

2. Start the FTL server executable.

```
tibftlserver -n <name>@<host>:<port>
```

Where,

<name> is a unique name for the FTL server, for example, ftl1.

<port> is any port not bound by another process.

ActiveSpaces component processes initiate contact with the FTL server at this address.

Defining the Data Grid and Component Processes

This section describes the steps to define the data grid and its component processes in the FTL server.

Procedure

1. In a text editor, start editing a script file.
2. Add the script command to create the data grid by using the syntax:

```
grid create statekeeper_count=1 copyset_size=1 grid_name
```

Where, *grid_name* is a unique name for this data grid.

3. Add the script commands to create the copyset, node, state keeper, external proxy and embedded proxy:

```
copyset create copyset_name
node create --copyset copyset_name node_name
keeper create keeper_name
proxy create proxy_name
proxy create _embedded_proxy
```

Where *copyset_name*, *node_name*, *keeper_name*, and *proxy_name* are any unique

name for each of these components.

> **Note:** Copysets and nodes in ActiveSpaces relate to horizontal data partitioning and data replication. When only a single copyset and node are defined as in this case, the data grid does not perform horizontal partitioning or replication.

4. Run the script using the tibdg administration tool to create the data grid.

```
tibdg -s script_file_path -r http://<host>:<port>
```

where `<host>` and `<port>` refer to the pipe-separated FTL server URLs.

Starting the Data Grid Processes

This section describes the steps to bring up the data grid component processes.

Procedure

1. Start the administrative daemon process

```
tibdgadmind -r http://<host>:<port>
```

2. Start the state keeper process.

```
tibdgkeeper -n keeper_name -r http://<host>:<port> -g grid_name
```

3. Start the node process

```
tibdgnode -n node_name -r http://<host>:<port> -g grid_name
```

4. Start the proxy process

```
tibdgproxy -n proxy_name -r http://<host>:<port> -g grid_name
```

For all of the above `<host>` and `<port>`, refer to the pipe-separated FTL server URLs.

keeper_name, *node_name*, *proxy_name* and *grid_name* refer to the names chosen in Defining the Data Grid and Component Processes

You can now run the following `tibdg` command to verify that all component processes are running and that the data grid is online:

```
tibdg -r http://<host>:<port> status
```

In a development environment, the FTL server, administrative daemon, state keeper, node, proxy, and EMS server processes can be started on the same host machine.

Connecting Multiple Servers to the Same Data Grid

There is no limit placed on the number of EMS servers using grid stores that can connect to a particular data grid. The grid store definitions in the data grid are differentiated based on the server name, meaning the only limitation in this respect is that servers with identical names - outside of a fault-tolerant pair - will not be able to use the same data grid.

Configuring Grid Stores

When using grid stores, the EMS server requires the configuration to be JSON-based. See Managing the JSON Configuration for details on how to create a JSON configuration file.

The following table describes the store parameters for grid stores.

Parameter Name	Description
Required Parameters	
`[store_name]`	`[store_name]` is the name that identifies this store configuration. Note that the square brackets [ ] DO NOT indicate that the `store_name` is an option; they must be included around the name.
`type=as`	Identifies the store type. This parameter is required for all store types. The type corresponding to grid stores is `as` (abbreviation of ActiveSpaces).

Parameter Name	Description
	Other available store types are as follows: • `file`—for file-based stores. • `ftl`—for FTL stores.
Optional Parameters	
`scan_iter_interval`	Determines the length of time between each interval of the store scan. The EMS server begins scanning a new section of the grid store at the time interval specified here. Specify time in units of msec, sec, min, hour or day to describe the time value as being in milliseconds, seconds, minutes, hours, or days, respectively. For example: `scan_iter_interval=100msec` By default, the server examines grid stores every 10 seconds. For more information, see Understanding Grid Store Intervals.
`scan_target_interval`	Controls the approximate length of time taken to complete a full scan of the grid store. Specify time in units of msec, sec, min, hour or day to describe the time value as being in milliseconds, seconds, minutes, hours, or days, respectively. For example: `scan_target_interval=12hour` By default, the scan interval is 24 hours. For more information, see Understanding Grid Store Intervals.

Note: Grid stores do not support an asynchronous write mode option as asynchronous writes are not supported by ActiveSpaces.

EMS does not support configuration of multiple store types in the same server. If using grid stores, all stores in the configuration must be of type `as`.

Managing the JSON Configuration

Creating and editing the JSON configuration can be done using the tibemsconf2json tool and TIBCO Messaging Manager, respectively.

Using the tibemsconf2json Tool

If the server configuration is first defined via the .conf configuration files, the tibemsconf2json tool can then be used to convert the .conf configuration files into a JSON configuration file. See Conversion of Server Configuration Files to JSON for more information.

The configuration for a grid store in stores.conf would be of the following format:

```
[store_name] # mandatory -- square brackets included.
type = as
[scan_iter_interval = time]
[scan_target_interval = time]
```

The tibemsconf2json tool cannot be used to make changes to the JSON configuration file after it has been generated.

Using the TIBCO Messaging Manager

Subsequent modifications to the JSON configuration should be done using TIBCO Messaging Manager. For details, refer to the TIBCO Messaging Manager documentation.

Server Configuration Upload/Download

When configured to use grid stores, the EMS server will first connect to the data grid and fetch the configuration information before beginning its start-up sequence. The JSON configuration must be available in the data grid prior to starting the EMS server. If no configuration is available in the data grid, the EMS server will start-up with a default configuration.

The tibemsjson2grid tool can be used to upload JSON configuration files to a specified data grid. It can also download JSON configuration files from a specified data grid.

Running the tibemsjson2grid Tool

The tibemsjson2grid tool is invoked from the command line. The tool is dependent on FTL and ActiveSpaces C client libraries, so the LD_LIBRARY_PATH environment variable must be set before running it.

```
export LD_LIBRARY_PATH=<AS_HOME>/lib:<FTL_HOME>/lib:$LD_LIBRARY_PATH
tibemsjson2grid options
```

tibemsjson2grid Options

The following table shows the options that are used with the tibemsjson2grid tool.

Option	Description
`-url` *url*	The pipe-separated URLs of the data grid the store is connected to.
`-name` *gridname*	The name of the data grid to connect to.
`-key` *uniquevalue*	The value passed to this parameter should be a unique value that identifies a specific JSON configuration. Uploading a JSON configuration with a non-unique combination `-key` and `-name` values will overwrite the existing configuration corresponding to that combination.
`-json` *pathname*	The absolute path to the JSON configuration file to be uploaded to the data grid. When the `-download` parameter is specified, the downloaded JSON configuration will be written to the value passed to this parameter.
`-download`	When specified, tibemsjson2grid will download the JSON configuration from the data grid and write it to the file passed to `-json`.

Option	Description
	If this parameter is not specified, the tool will default to uploading the JSON configuration to the data grid.
`-trustfile` *path*	Path to the plaintext file that contains the FTL server's public certificate. Required for TLS communication with a secure FTL server.
`-user` *user*	User name to use when connecting to an FTL server that has authentication enabled.
`-password` *password*	Password to use when connecting to an FTL server that has authentication enabled. To hide the password from casual observers, see the *Password Security* section of the *TIBCO FTL Administration* guide.

Examples

Example 1

Uploading configuration to the data grid:

```
tibemsjson2grid -url http://hostname:8080 -name devgrid -key uniquekey
-json tibemsd.json
```

Example 2

Downloading configuration from a data grid:

```
tibemsjson2grid -url http://hostname:8080 -name devgrid -key uniquekey
-json tibemsd.json -download
```

Example 3

Uploading configuration to a secure data grid:

```
tibemsjson2grid -url https://hostname:8080 -name devgrid -key uniquekey
-trustfile ftl-trust.pem -user user1 -password password -json
tibemsd.json
```

Server Command-Line Options for Grid Stores

When starting the EMS server with grid stores, the server must be pointed to the data grid where message data and configuration information will be stored and retrieved from. This is done via the following server command line options.

Option	Description
`-grid_url` *url*	The pipe-separated URLs of the data grid the store is connected to.
`-grid_name` *datagrid_name*	The name of the data grid to connect to.
`-config` *key*	The key specified while uploading the JSON configuration to the data grid.
`-module_path` *path_list*	List of paths to lib directories of ActiveSpaces and FTL installations.
`-grid_trust_file` *path*	Path to the plaintext file that contains the FTL server's public certificate. Required for TLS communication with a secure FTL server.
`-grid_user` *user*	User name to use when connecting to an FTL server that has authentication enabled.
`-grid_password` *password*	Password to use when connecting to an FTL server that has authentication enabled. To hide the password from casual observers, see the *Password Security* section of the *TIBCO FTL Administration* guide.

The syntax for starting the EMS server with grid stores is as follows:

```
tibemsd -grid_url <url> -grid_name <datagrid_name> -config <key>
-module_path <path_list> [-grid_trust_file <path>
-grid_user <user> -grid_password <password>]
```

Examples

Example 1

```
tibemsd -grid_url http://hostname:8080 -grid_name devgrid -config
uniquekey -module_path AS_HOME/lib:FTL_HOME/lib
```

Example 2

```
tibemsd -grid_url https://hostname:8080 -grid_name devgrid -config
uniquekey -module_path AS_HOME/lib:FTL_HOME/lib -grid_trust_file
ftl-trust.pem -grid_user user1 -grid_password password
```

FTL Stores

You can configure TIBCO Enterprise Message Service to store messages, state information, and configuration information in supported versions of TIBCO FTL.

The following topics describe FTL stores. For information about other store types, see Store Messages in Multiple Stores and Grid Stores.

Note: The FTL stores feature is only supported on Linux platforms.

FTL Stores Overview

Note: Release 10.2 of EMS introduced a redesigned version of FTL stores with improved performance capabilities and more robust fault-tolerance and disaster recovery features. Release 10.3 introduces asynchronous disk persistence. It is highly recommended that existing FTL stores deployments be migrated to the current release. Refer to the EMS 10.3 Release Notes for information on migrating deployments from EMS 10.1 and EMS 10.2.

FTL stores have a somewhat unique configuration and deployment model in comparison to other EMS storage types. When using FTL stores, the EMS server runs as a service within an FTL server – which is an umbrella process that launches and manages a number of messaging services that constitute a TIBCO FTL deployment. The integration of EMS with FTL in this manner reduces latency in communication between the EMS server and its FTL backend. Additionally, it also allows EMS to make use of FTL's disaster recovery capabilities.

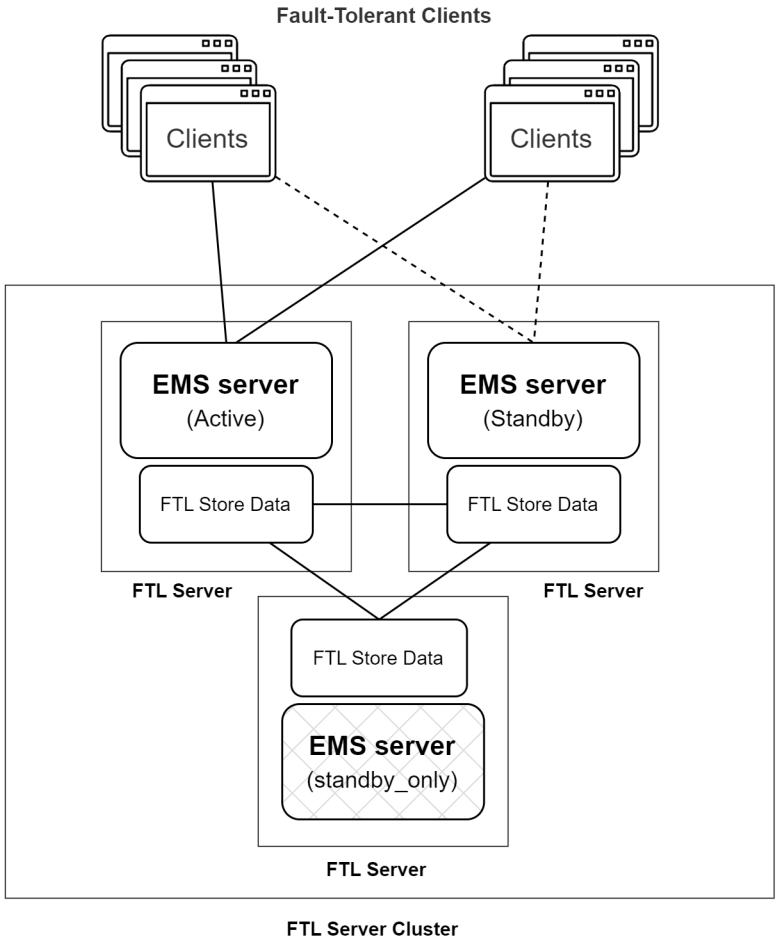
In terms of functionality, FTL stores are similar to file-based stores. When using FTL stores, all pending persistent message information and state information is maintained in EMS server memory and persisted on disk via FTL. Keeping this information in memory reduces the amount of disk reads required and facilitates faster message processing

As with file-based stores, an EMS server using FTL stores must recover all state information and pending message information before it can activate. Depending on the situation, this is

achieved either by reading the contents of all stores from disk via FTL, or by processing the data that has already been replicated to the EMS server by FTL.

Fault-Tolerance with FTL Stores

As a storage mechanism, a cluster of coordinating FTL servers provides data persistence and replication features that support Shared State fault-tolerance in EMS. The FTL server cluster must be formed of 3 constituent FTL servers in order to support EMS fault-tolerance with FTL stores. A single FTL server can only be configured to run a standalone EMS server with FTL stores.



Each of the three FTL servers in the cluster has its own EMS server running under it. One of those three EMS servers is configured as `standby_only`, meaning that it will always remain in the fault-tolerant standby state and can never transition to the active state. This server's

only purpose is to perform data replication and provide redundancy. The other two EMS servers comprise the actual fault-tolerant EMS server pair. One of these is chosen by the cluster to enter the active state and the other becomes its fault-tolerant peer in standby state.

A client making a fault-tolerant connection to these EMS servers would specify only the URLs of the fault-tolerant EMS server pair. The URL of the `standby_only` EMS server should not be used since that EMS server will never activate.

The fault-tolerant EMS server pair makes use of the connections between the FTL servers in the cluster for fault-tolerance related communication. This means that the specifics of fault-tolerance are handled by the FTL server cluster rather than the EMS servers themselves. The majority of EMS server parameters and options relating to configuration of fault-tolerance are not applicable when using FTL stores.

To provide data redundancy, the persistence capabilities of FTL are leveraged to consistently replicate FTL store data to at least a majority of EMS servers running within the FTL server cluster. By default, the FTL store data replicated to each server is also persisted on disk to provide an additional layer of redundancy (See Persistence with FTL Stores). In most fault-tolerant failover scenarios, the standby EMS server already has the required store data in memory and is able to activate with minimal recovery time.

When using FTL stores, the EMS server configuration is stored in the FTL server cluster. This allows the configuration to be accessible to EMS servers running in separate machines or containers. Static configuration elements including JAAS modules, JAAS module configuration files, JACI modules and digital certificates are not stored in the cluster and must still be maintained manually.

Deciding Between FTL Stores and File-Based Stores

When making the decision between FTL stores and file-based stores for a fault-tolerant EMS server deployment, there are three main criteria to consider.

Availability of Shared Storage

If a shared storage mechanism that meets the Support Criteria for shared state is available, file-based stores would be the preferred option. However, when such a storage mechanism is not available, or does not provide reasonable I/O speeds, FTL stores offer a strong,

performant alternative that fulfills all the data persistence and replication requirements needed for EMS fault-tolerance. A good example of this is in cloud-based deployments, where shared storage may not be an option, or may be prohibitively slow even when available. FTL stores would be able to leverage the more performant storage options offered by the cloud service provider to support a fault-tolerant deployment. Additionally, a cloud-based deployment of FTL stores would also be capable of deploying across multiple availability zones to provide a level of redundancy and availability that may not be achievable using shared storage.

Failover Time

For deployments where a large amount of backlogged persistent message data is expected, FTL stores can provide faster recovery time than file-based stores in most fault-tolerant failover scenarios. With smaller pending message backlogs, the fault-tolerant failover recovery times for both store types will be comparable.

Disaster Recovery

Another criteria to consider is whether disaster recovery is important for the deployment. When using FTL stores, EMS is able to leverage its integration with FTL to provide disaster recovery as an in-built feature. The EMS server does not have this capability when using file-based stores.

Configuring and Deploying FTL Stores

FTL stores have been designed such that an EMS user should be able to deploy them with minimal knowledge of TIBCO FTL. The following sections cover all information required for configuring and deploying EMS with FTL stores. If you would like to learn more about FTL servers or FTL in general, refer to the *TIBCO FTL Administration* product guide.

The ability to use FTL stores is contingent upon the deployment of an FTL server or FTL server cluster that has been configured to support FTL stores, as well as the availability of an EMS server configuration that has been configured with FTL stores.

Configuring the FTL Server Cluster

An individual FTL server or a cluster of coordinating FTL servers are configured via a YAML-based configuration file. The YAML below shows the basic template for an FTL server cluster consisting of 3 FTL servers configured to support FTL stores.

```
globals:
  core.servers:
    <name of FTL server #1>: <host>:<port>
    <name of FTL server #2>: <host>:<port>
    <name of FTL server #3>: <host>:<port>
servers:
  <name of FTL server #1>:
  - tibemsd:
      -listens: <comma-separated list of URLs for EMS server #1>
      exepath: EMS_HOME/bin/tibemsd
      -config_wait:
  <name of FTL server #2>:
  - tibemsd:
      -listens: <comma-separated list of URLs for EMS server #2>
      exepath: EMS_HOME/bin/tibemsd
      -config_wait:
  <name of FTL server #3>:
  - tibemsd:
      -listens: <comma-separated list of URLs for EMS server #3>
      exepath: EMS_HOME/bin/tibemsd
      -config_wait:
      -standby_only:
```

While this YAML template configures a cluster of 3 FTL servers, it can easily be adapted for a standalone FTL server by removing server entries from the servers section and from the `core.servers` list in the `globals` section.

In addition to the examples in the next section, sample YAML configuration files can be found under the *EMS_HOME/samples/config* directory of the EMS installation.

Sections in the FTL Server Cluster Configuration

Provided below is a brief description of each section of the YAML configuration file and a list of all required and optional parameters that can be included in those sections in the context of FTL stores. Parameters not mentioned in the following sections are not supported for use with FTL stores.

globals

The globals section contains parameters that directly affect the operation of the FTL servers in the cluster.

Parameter Name	Description
`core.servers`	This parameter is mandatory. A list of the names of the FTL servers in the cluster along with their location.
`tls.secure`	The password that was used to encrypt the keystore file. When this parameter is present, all communication between the FTL servers in the cluster will be encrypted. The value for this parameter should be of the form: `file:<path to keystore_password_file>` where `keystore_password_file` is a file containing the chosen password for authentication. See Initializing FTL Server Cluster Security for details.

`auth.url`	The URL of a flat file with the following contents:
	
	`admin: <password>, ftl-admin,ftl-internal`
	
	When present, authentication is enabled in the FTL server so that it requires and verifies username and password credentials from coordinating FTL servers in the cluster and from the tibemsjson2ftl and tibftladmin tools.
	
	The value for this parameter must be of the form:
	
	`file://<path to flat file>`
	
	This parameter must be set if the `tls.secure` parameter is specified.
	
	See Initializing FTL Server Cluster Security.

servers

The `servers` section must contain a list of all FTL servers in the cluster. For each server in the list, a sub-list of services whose behavior is to be configured can be specified. No service should be specified more than once for a given server.

The table below details the parameters available to configure each service.

Parameter Name	Description
`tibemsd` Service Parameters	
`exepath`	This parameter is mandatory and must be configured for each FTL server in the cluster. The path to the tibemsd executable. This executable is located at *`EMS_HOME`*`/bin/tibemsd`.
`-listens`	A comma-separated list of one or more listen URLs for the EMS server. This parameter must be used in place of the `listen` EMS server parameter when using FTL stores. If not specified, the EMS server will start with the default listen URL tcp://7222. Refer to the `listen` section for information about EMS server listen URL syntax.
`-config_wait`	When this parameter is specified for all `tibemsd` services, the EMS servers within the FTL server cluster will wait for an EMS configuration to become available in the cluster before starting up. This parameter does not accept a value. If this parameter is not specified for all `tibemsd` services, and an EMS configuration is not available in the cluster, the EMS servers will start with default configuration. See the Server Configuration Upload/Download section for instructions on uploading the EMS configuration to the FTL server cluster.

Parameter Name	Description
`-standby_only`	This parameter informs the FTL server that its EMS server is configured to be `standby_only`, meaning that it cannot ever transition to active state. This parameter does not accept a value. Only one of the 3 FTL servers in this section should have this parameter set for its `tibemsd` service. The selected FTL server will be the one whose EMS server will not be part of the fault-tolerant EMS server pair, and whose URL will not be used by clients. This parameter should not be specified if the configuration is for a standalone FTL server.
`-preferred_active`	Setting this parameter designates the EMS server as the preferred active server. In situations where either EMS server in the fault-tolerant pair could potentially enter the active state, the server configured with `-preferred_active`, will always be the one to activate. Configuring this parameter for an EMS server does not guarantee that the server will always be in the active state. The preferred active server will enter the standby state if its fault-tolerant peer EMS server is already in the active state.
`-store`	The path to the directory where the FTL server will write out FTL store-specific data. If not specified, FTL store data will be written to the current working directory by default.
`-monitor_listen`	The URL at which the EMS server will listen for health check and Prometheus metrics requests. This URL should follow the same syntax as described in the `monitor_listen` section.

Parameter Name	Description
`-oauth2_server_validation_key`	The path to the PEM-encoded public key or JWKS to use when validating the signature of OAuth 2.0 access tokens presented by incoming connections. This parameter should be used in place of the `oauth2_server_validation_key` EMS server parameter.
`-oauth2_audience`	The expected value of the 'aud' claim in OAuth 2.0 access tokens presented by incoming connections. This parameter should be used in place of the `oauth2_audience` EMS server parameter.
`load`	The path to the state file from which the FTL server will load its state information during startup. This parameter is only applicable in the context of restarting an FTL server cluster when `in_memory_replication` is enabled. See Shutting Down and Restarting an In-Memory Cluster for more information.
`<EMS server command line option>`	Any EMS server command line option can be included in this section. For example, `-ssl_trace`.
`realm` Service Parameters	
`data`	The general data directory for the FTL server. This directory will contain all non-FTL store specific data. If not specified, the default is the current working directory.
`drto`	When present, this FTL server cluster recognizes another given FTL server cluster as belonging to a disaster recovery site and attempts to connect to it. Supply a pipe-separated list of the URLs of the FTL

Parameter Name	Description
	servers in the disaster recovery site's FTL server cluster. Each URL should be of the form: `<FTL server name>@<host>:<port>` (You must also configure the disaster recovery FTL servers using the `drfor` parameter.)
`drfor`	When present, this FTL server cluster recognizes that it is in the disaster recovery site for a primary site FTL server cluster. Supply a pipe-separated list of URLs of the FTL servers in the primary site's FTL server cluster. Each URL should be of the form: `<FTL server name>@<host>:<port>` (You must also configure the primary site FTL servers using the `drto` parameter.)
`user`	The username that the FTL server clusters at the primary and DR sites will use to authenticate each other. If the authentication data for the primary and DR sites was created based on the steps in Initializing FTL Server Cluster Security, the value passed to this option should be `admin`. This parameter must be specified if setting up disaster recovery with secure FTL server clusters.
`password`	The password that the FTL server clusters at the primary and DR sites will use to authenticate each other. The value for this parameter should be of the form: `file:<path to password_file>` where `password_file` is a file containing the chosen

Parameter Name	Description
	password for authentication. See Initializing FTL Server Cluster Security for details. This parameter must be specified if setting up disaster recovery with secure FTL server clusters.
`ftlserver.properties` Parameters	
`logfile`	The prefix for the filenames of the rolling log files generated by the FTL server. If the prefix includes a directory path, the FTL server's log files will be generated under that directory. If not, the FTL server will generate its log files in the current directory. Any directories included in the prefix value must already exist. If this parameter is not specified, the FTL server will send log output to the console.
`max.log.size`	The maximum size of each FTL server log file in bytes.
`max.logs`	The maximum number of rolling log files that can be created.

Examples

Example 1
Configuration for a standalone FTL server that does not have security enabled and is using the default data directories.

```
globals:
  core.servers:
    ftls1: host1:8080
servers:
  ftls1:
  - tibemsd:
      -listens: tcp://host1:7222
```

```
    exepath: /opt/tibco/ems/10.3/bin/tibemsd
    -config_wait:
```

Example 2

Configuration for a secure FTL server cluster that specifies generic and FTL store-specific data directories, has logging configured and is also configured to replicate data to a disaster recovery site.

```
globals:
  core.servers:
    ftls1: host1:8080
    ftls2: host2:8085
    ftls3: host3:8090
  tls.secure: file:/opt/deployment/keystore_password_file
  auth.url: file:///opt/deployment/users.txt
servers:
  ftls1:
  - tibemsd:
      -listens: ssl://host1:7222
      exepath: /opt/tibco/ems/10.3/bin/tibemsd
      -store: /opt/deployment/ftls1/ftlstore_data
      -config_wait:
  - realm:
      data: /opt/deployment/ftls1/ftlserver_data
      drto: dr_ftls1@host4:8080|dr_ftls2@host5:8085|dr_ftls3@host6:8090
      user: admin
      password: file:/opt/deployment/password_file
  - ftlserver.properties:
      logfile: /opt/deployment/ftls1/logs/log
      max.log.size: 1048576
      max.logs: 100
  ftls2:
  - tibemsd:
      -listens: ssl://host2:7224
      exepath: /opt/tibco/ems/10.3/bin/tibemsd
      -store: /opt/deployment/ftls2/ftlstore_data
      -config_wait:
  - realm:
      data: /opt/deployment/ftls2/ftlserver_data
      drto: dr_ftls1@host4:8080|dr_ftls2@host5:8085|dr_ftls3@host6:8090
      user: admin
      password: file:/opt/deployment/password_file
  - ftlserver.properties:
      logfile: /opt/deployment/ftls2/logs/log
      max.log.size: 1048576
      max.logs: 100
```

```
ftls3:
- tibemsd:
    -listens: ssl://host3:7226
    exepath: /opt/tibco/ems/10.3/bin/tibemsd
    -store: /opt/deployment/ftls3/ftlstore_data
    -config_wait:
    -standby_only:
- realm:
    data: /opt/deployment/ftls3/ftlserver_data
    drto: dr_ftls1@host4:8080|dr_ftls2@host5:8085|dr_ftls3@host6:8090
    user: admin
    password: file:/opt/deployment/password_file
- ftlserver.properties:
    logfile: /opt/deployment/ftls3/logs/log
    max.log.size: 1048576
    max.logs: 100
```

Logging With FTL Stores

When using FTL stores, the `logfile` EMS server parameter and supporting parameters `log_trace`, `logfile_max_size` and `logfile_max_count` are unsupported. Consequently, the `rotatelog` administration command is also unsupported. Since the EMS server runs as a service under an FTL server, the EMS server console output is written out as part of the FTL server logs. The FTL server's logging capabilities should be used to capture both FTL and EMS output.

The `logfile`, `max.logs` and `max.log.size` YAML configuration parameters can be used to configure the FTL server to send its console output to a set of rolling log files. See the parameter descriptions in the `ftlserver.properties` Parameters section for more details.

Note that the trace options set via the `console_trace` EMS server parameter will determine which EMS server traces are seen in the FTL server logs.

For further information on FTL server logging, refer to the *TIBCO FTL Administration* guide.

Initializing FTL Server Cluster Security

This section details the steps to set up FTL server cluster security. Please note that both TLS-secured communication and authentication must be configured in order to enable security in the FTL server cluster. Deploying with only one of these is not supported.

Before you begin

- TIBCO FTL must already be installed on the machine where this procedure is to be performed.
- The clocks on all machines where FTL servers will be running must be synchronized.
- The YAML configuration must have been generated.

Procedure

1. Choose a password that will be used to encrypt the private key data, ie: the keystore password. Write the password to a file.

   ```
   echo <password> > keystore_password_file
   ```

2. Run the following command to generate the TLS data files.

   ```
   tibftlserver --init-security file:<path to keystore_password_file>
   ```

   This results in the generation of two files - the trust file which contains the public certificate and the keystore file which contains the encrypted private key data.

3. Copy the trustfile and keystore to the general data directory of each FTL server in the cluster. See Deploying the FTL Server Cluster for details on configuring the general data directory and FTL store-specific data directory for an FTL server.

4. Add the `tls.secure` parameter to the `globals` section of the YAML configuration file and specify the path to `keystore_password_file`.

   ```
   globals:
     # ...
     tls.secure: file:<absolute path to keystore_password_file>
   ```

Note: There are several other ways to pass the keystore password to the `tls.secure` parameter. See the *Password Security* section of the *TIBCO FTL Administration* product guide for details.

5. Choose a password to be used for authentication and create a new file containing the authentication data for the FTL server cluster.

   ```
   echo "admin: <password>, ftl-admin,ftl-internal" > users.txt
   ```

   If setting up disaster recovery, also write the password to a separate file.

```
echo <password> > password_file
```

6. Add the `auth.url` parameter to the `globals` section of the YAML configuration file and specify the path to the users.txt file.

```
globals:
  # ...
  auth.url: file://<absolute path to users.txt file>
```

Note: As an alternative to flat file based internal authentication, FTL also offers support for authentication via an external service. See the *Authentication Service* section of the *TIBCO FTL Administration* product guide for details.

With these steps completed, the FTL server cluster can be deployed by following the steps in the next section.

Deploying the FTL Server Cluster

Before you begin

- TIBCO FTL and TIBCO EMS must already be installed on each machine where an FTL server is to be started.
- The clocks on all machines where FTL servers will be running must be synchronized.
- The YAML configuration must have been generated.

Procedure

Repeat the following steps for each FTL server in the cluster.

1. Choose two existing empty directories or create two new directories to serve as the general FTL server data directory and the FTL store-specific data directory.
2. Configure the FTL server to use one of the chosen directories as its general data directory and the other as its FTL store-specific data directory. This can be done by adding the `data` and `-store` parameters under the corresponding server entry in the `servers` section of the YAML configuration.

```
servers:
  # ...
```

```
  <FTL server name>:
  # ...
  - realm:
      data: <absolute path to general FTL server data directory>
  - tibemsd:
      # ...
      -store: <absolute path to FTL store-specific data
directory>
```

If not specified, both the general data directory and the FTL store-specific data directory for the FTL server will default to the current working directory.

3. Run the FTL server executable.

```
tibftlserver -n <FTL server name> -c <path to YAML configuration>
```

Configuring FTL Stores in the EMS Server

This section lists the `stores.conf` parameters, `tibemsd.conf` parameters and EMS server options that are required, unique, unsupported, or which have differing behavior for FTL stores.

The EMS server only supports JSON-based configuration when using FTL stores. All .conf configuration files will need to be converted into a JSON-based configuration file. See Managing the JSON Configuration for details on how to do this.

stores.conf Parameters

The following table describes the `stores.conf` parameters required for FTL stores. Any `stores.conf` parameters not listed here are unsupported when using FTL stores.

Parameter Name	Description
`[store_name]`	`[store_name]` is the name that identifies this store configuration. Note that the square brackets [ ] DO NOT indicate that the store_name is an option; they must be included around the

Parameter Name	Description
	name.
`type=ftl`	Identifies the store type. This parameter is required for all store types. The type corresponding to FTL stores is `ftl`. Other available store types are as follows: • `file` – for file-based stores. • `as` – for grid stores.
`mode`	The `mode` determines whether messages persisted in the store will be written to disk synchronously or asynchronously by FTL. The value can be either: `async` - messages are persisted on disk using asynchronous I/O calls. `sync` - messages are persisted on disk using synchronous I/O calls. If not specified, the default for FTL stores is `async`. See Persistence with FTL Stores for more information.

EMS does not support configuration of multiple store types in the same server. If using FTL stores, all stores in the configuration must be of type `ftl`.

tibemsd.conf Parameters

The following table describes the tibemsd.conf parameters that are specific to FTL stores, unsupported with FTL stores, or which behave differently when used with FTL stores.

Parameter Name	Description
Parameters Specific to FTL Stores	
`in_memory_replication`	When enabled, data will be replicated to all EMS servers in the cluster, but will not be persisted to

Parameter Name	Description
	disk. Enabling this parameter will enhance the performance of FTL stores at the cost of disk persistence. See In-Memory Replication for more information.
`ftl_disk_preallocation`	This parameter reserves disk space for the FTL persistence underlying FTL stores. The value should be specified units of MB or GB. For example: `ftl_disk_preallocation = 20GB` This option should be considered if the FTL stores deployment experiences performance issues when there is a large pending message backlog. This option is not applicable if `in_memory_persistence` is enabled.
Parameters With Differing Behavior for FTL Stores	
`max_client_msg_size`	When using FTL stores, the default value of this parameter is set to 10 MB instead of being unbounded. Even though the maximum possible value is still 2 GB, we recommend that application programs use much smaller messages, since larger messages will strain the performance limits of most current hardware and operating system platforms.
`server_heartbeat_server` `server_timeout_server_connection`	When using FTL stores, these parameters only affect server-to-server route connections. They have no effect on the behavior of fault-tolerance. The specifics of fault-tolerance are handled directly by FTL.

Parameter Name	Description
Parameters Unsupported for FTL Stores	
All parameters with prefix `ft_` aside from `ft_reconnect_timeout`	In general, any parameters that configure fault-tolerance between EMS servers are ignored when using FTL stores. The specifics of fault-tolerance are handled directly by FTL. This means that all parameters that begin with the prefix `ft_` are ignored if a server is using FTL stores, with the one exception being the `ft_reconnect_timeout` parameter which is still honored.
`listen`	This parameter is ignored when present in the EMS server configuration. The EMS server `listen` URLs must be configured via the `-listens` parameter in the FTL server YAML configuration.
`module_path`	When using FTL stores, this parameter cannot be used to specify the FTL shared libraries to be loaded by the EMS server. The EMS server will always load the FTL shared libraries corresponding to the hosting FTL server.
`store`	This parameter may cause unexpected behavior if present in the EMS server configuration. The location of the FTL store data must be configured via the `-store` parameter in the FTL server YAML configuration.
`monitor_listen` `health_check_listen` (deprecated)	This parameter may cause unexpected behavior if present in the EMS server configuration. The `monitor_listen` parameter in the FTL server YAML configuration must be used to configure the port on which the EMS server listens for health

Parameter Name	Description
	check and Prometheus metrics requests.

`logfile`
`log_trace`
`logfile_max_count`
`logfile_max_size`
	When using FTL stores, the EMS server log file should not be configured. See the Logging With FTL Stores section for details on setting up FTL server logging.

`secondary_logfile`
`secondary_monitor_listen`
`secondary_health_check_listen`
(deprecated)
	These parameters are ignored when present in the EMS server configuration.
	
	When using FTL stores, the EMS server roles are implicit. Since these parameters are only valid when the secondary role is explicitly defined, they are unsupported for FTL stores.

Persistence with FTL Stores

FTL stores can be configured to persist data to disk synchronously, asynchronously, or only in-memory. Each configuration offers its own advantages in terms of performance and reliability.

Synchronous

If an FTL store is configured with `mode=sync`, all data sent to the store is synchronously written to disk via FTL.

Synchronous mode for FTL stores provides high reliability with guaranteed message persistence on disk. Message processing performance may be lower than other persistence modes due to the additional latency introduced by synchronous disk writes.

Asynchronous Replication

If an FTL store is configured with `mode=async`, all data sent to the store is persisted to disk asynchronously. Messages sent to a destination that is mapped to an asynchronous mode FTL store are synchronously replicated to at least a majority of EMS servers within the cluster to guarantee data persistence and redundancy in memory, then asynchronously written to disk via FTL.

Asynchronous mode for FTL stores can offer higher performance at the cost of guaranteed message persistence on disk. This persistence mode can be especially advantageous in deployments with slower disks.

If the `mode` parameter is not configured for a user-defined FTL store, it is set to asynchronous persistence mode by default.

In-Memory Replication

The `in_memory_replication` EMS server parameter can be used to configure the deployment to persist message data and state information only in memory and avoid all disk writes. This parameter applies to all FTL stores in the deployment and overrides the `mode` parameter of all stores.

In-memory persistence mode offers higher FTL store performance at the cost of disk persistence. This persistence model is only suggested for deployments with higher tolerance for message loss. Users are encouraged to measure whether the desired performance is achievable using synchronous or asynchronous persistence modes before considering in-memory persistence.

The procedure for shutting down the FTL server cluster differs when this parameter is enabled. See Shutting Down and Restarting an In-Memory Cluster for details.

Unsupported tibemsd Options

The table below lists the EMS server command-line options that are unsupported when using FTL stores. All unsupported EMS server command-line options are ignored when specified in the FTL server YAML configuration.

Option	Description
`-config`	When using FTL stores, the EMS server reads its configuration directly from the FTL server cluster. The tibemsjson2ftl tool must be used to upload the EMS server configuration to the FTL server cluster.
`-secondary`	When using FTL stores, the EMS server roles are implicit, which makes the `-secondary` option

Option	Description
	irrelevant.
`-ft_active` `-ft_activation`	When using FTL stores, the specifics of fault-tolerance are handled by the FTL server cluster.

Managing the JSON Configuration

Creating and editing the JSON configuration can be done using the tibemsconf2json tool and TIBCO Messaging Manager, respectively.

Using the tibemsconf2json Tool

The entirety of the EMS server configuration can first be defined via the .conf configuration files. The tibemsconf2json tool can then be used to convert the .conf configuration files into a JSON configuration file. See Conversion of Server Configuration Files to JSON for more information.

The configuration for an FTL store in stores.conf would be of the following format:

```
[store_name] # mandatory -- square brackets included.
type=ftl
[mode=async|sync]
```

The tibemsconf2json tool cannot be used to make changes to the JSON configuration file after it has been generated.

Using the TIBCO Messaging Manager

Subsequent modifications to the JSON configuration should be done using TIBCO Messaging Manager. For details, refer to the TIBCO Messaging Manager documentation.

Server Configuration Upload/Download

When configured to use FTL stores, the EMS server will first connect to the FTL server cluster and fetch the configuration information before beginning its start-up sequence. The

JSON configuration must be available in the FTL server cluster prior to starting the EMS server. If no configuration is available in the cluster, the EMS server will start-up with a default configuration.

The tibemsjson2ftl tool can be used to upload JSON configuration files to a specified FTL server cluster. It can also download JSON configuration files from a specified cluster.

Running the tibemsjson2ftl Tool

When the EMS server doesn't find an EMS server configuration in the FTL server cluster, its subsequent behavior is dependent on whether the `-config_wait` parameter has been set in the FTL server cluster configuration. If present, the EMS server will halt its start-up process until the configuration becomes available. This provides the user with a chance to upload their own EMS server configuration to the cluster. If the parameter is not present, the server will simply start with a default configuration.

The tibemsjson2ftl tool can be used to upload JSON-based EMS configuration files to a specified FTL server cluster. It can also download JSON configuration files from a specified cluster.

The tibemsjson2ftl tool is invoked from the command-line. The tool is dependent on the FTL C client libraries, so the LD_LIBRARY_PATH environment variable must be set before running it.

```
export LD_LIBRARY_PATH=<FTL_HOME>/lib:$LD_LIBRARY_PATH
tibemsjson2ftl options
```

tibemsjson2ftl Options

The following table shows the options that are used with the tibemsjson2ftl tool.

Option	Description

`-url` *url*	The pipe-separated list of URLs of the FTL server cluster to connect to.
	
	The URLs must be in one of these forms:
	
	`http://<host>:<port>`
	`https://<host>:<port>`

Option	Description
`-json` *pathname*	The absolute path to the JSON configuration file to be uploaded to the FTL server cluster. When the `-download` parameter is specified, the downloaded JSON configuration will be written to the value passed to this parameter.
`-download`	When specified, tibemsjson2ftl will download the JSON configuration from the FTL server cluster and write it to the file passed to `-json`. If this parameter is not specified, the tool will default to uploading the JSON configuration to the FTL server cluster.
`-trustfile` *path*	Path to the trust file created in the Initializing FTL Server Cluster Security section. Required when connecting to a secure FTL server cluster.
`-user` *user*	User name to use when connecting to an FTL server cluster that has authentication enabled. If the cluster's authentication data was created based on the steps in Initializing FTL Server Cluster Security, the value passed to this option should be admin.
`-password` *password*	Password to use when connecting to an FTL server cluster that has authentication enabled. This should be the same password written to the users.txt file in Initializing FTL Server Cluster Security. To hide the password from casual observers, see the Password Security section of the *TIBCO FTL Administration* guide.
`-oauth2_access_token` *token*	The OAuth 2.0 access token to use when connecting to an FTL server cluster configured with OAuth 2.0 authentication. This option is not required when connecting to an FTL server cluster that uses a different authentication method.

Examples

Example 1

Uploading configuration to an FTL server cluster:

```
tibemsjson2ftl -url http://hostname:8080 -json tibemsd.json
```

Example 2

Downloading configuration from an FTL server cluster:

```
tibemsjson2ftl -url http://hostname:8080 -json tibemsd.json -download
```

Example 3

Uploading configuration to a secure FTL server cluster:

```
tibemsjson2ftl -url https://hostname:8080 -json tibemsd.json -trustfile
ftl-trust.pem -user admin -password password
```

Shutting Down the FTL Server Cluster

This section describes the methods available for shutting down an individual FTL server or an entire FTL server cluster, thereby shutting down any EMS servers running under those FTL servers.

Note: Killing an EMS server process is not supported. Since each EMS server runs as a service within an FTL server, killing an EMS server process will simply cause its hosting FTL server to automatically restart it.

EMS Administration Tool

The tibemsadmin tool can be used to shut down an individual FTL server. When connected to an EMS server that is using FTL stores, the tibemsadmin `shutdown` command will shut

down both the EMS server and its hosting FTL server process.

See the Using the EMS Administration Tool section for a list of tibemsadmin options and usage instructions.

FTL Administration Tool

The FTL administration tool can be used to shut down an FTL server or FTL server cluster, thereby shutting down any EMS servers running under those FTL servers. This tool is available as part of the TIBCO FTL installation under *FTL_HOME/bin/tibftladmin*.

tibftladmin Options

The following table describes the tibftladmin options that are relevant to FTL stores.

Option	Description

`--ftlserver` *`url`*	URL of the FTL server to connect to. This URL can belong to any one of the FTL servers in the cluster.
	URL must be in one of these forms:
	`http://<host>:<port>`
	`https://<host>:<port>`

`--shutdown`	Shut down the FTL server process and all its services, including its associated EMS server.
`--shutdown_cluster`	Alternative to the `--shutdown` option. Shut down all FTL server processes in the cluster and all their services, including all associated EMS servers.
`--tls.trust.file` *path*	Path to the trust file created in the Initializing FTL Server Cluster Security section. Required when connecting to a secure FTL server cluster.
`--user` *user*	User name to use when connecting to an FTL server cluster that has authentication enabled.

Option	Description
	If the cluster's authentication data was created based on the steps in Initializing FTL Server Cluster Security, the value passed to this option should be `admin`.
`--password` *password*	Password to use when connecting to an FTL server cluster that has authentication enabled. This should be the same password written to the users.txt file in Initializing FTL Server Cluster Security. To hide the password from casual observers, see the Password Security section of the *TIBCO FTL Administration* guide.

Examples

Example 1

Shut down one of the FTL servers in the cluster:

```
tibftladmin --ftlserver http://hostname:8080 --shutdown
```

Example 2

Shut down the entire FTL server cluster:

```
tibftladmin --ftlserver http://hostname:8080 --shutdown_cluster
```

Example 3

Shut down a secure FTL server cluster:

```
tibftladmin --ftlserver http://hostname:8080 --shutdown --tls.trust.file
ftl-trust.pem --user admin --password password
```

Shutting Down and Restarting an In-Memory Cluster

When the `in_memory_replication` parameter is enabled, data is persisted only in-memory and is not written to disk. This means that performing a shutdown of the entire FTL server cluster by normal means would result in the loss of all FTL server state information – and thereby, all EMS server state information. To prevent this from happening, the cluster should only be shut down using the `save_and_exit` EMS administration tool command. This command will instruct each FTL server in the cluster to write its state to disk and then shut down. The state information for each FTL server will be written to a state file named `<FTL server name>.state` under its FTL store-specific data directory.

When the FTL server cluster needs to be restarted, the first step should be to modify the YAML configuration to instruct the cluster to load its saved state from disk upon startup. This can be done by adding the `load` parameter to the `tibemsd` service of each FTL server present in the YAML configuration. Each FTL server's `load` parameter must be supplied with the absolute path to the matching state file that was generated during the cluster shutdown.

```
servers:
  <name of FTL server #1>
  # ...
  - tibemsd
      # ...
      load: <path to state file of FTL server #1>
  <name of FTL server #2>
  # ...
  - tibemsd
      # ...
      load: <path to state file of FTL server #2>
  <name of FTL server #3>
  # ...
  - tibemsd
      # ...
      load: <path to state file of FTL server #3>
```

Disaster Recovery

When using FTL stores, the disaster recovery capabilities of TIBCO FTL are extended to EMS. Setting up a Disaster Recovery (DR) site of operations can minimize EMS server downtime in the event that the primary site of operations becomes disabled.

FTL's disaster recovery implementation works as follows. Whenever the FTL server cluster receives data that needs to be replicated among its constituent FTL servers, it immediately also forwards it to an identical FTL server cluster running in a remote DR site. This means that if the FTL server cluster at the primary site becomes unavailable, the cluster at the DR site, and thereby the EMS servers at the DR site, can pick up right where things left off. The only information that would have been lost is in-flight data that the FTL server cluster was in the process of replicating when the primary site went down. Data replication to the DR site is asynchronous and does not add latency to operations at the primary site.

Setting up the Disaster Recovery Site

FTL server clusters at both the primary site and DR site first need to be configured to support disaster recovery and then deployed.

Procedure

1. If the FTL server cluster at the primary site is already running, first shut it down using the EMS or FTL administration tool. See Shutting Down the FTL Server Cluster for more information.
2. Add the `drto` parameter to the YAML configuration of the FTL server cluster at the primary site. This parameter must be supplied with a pipe-separated list of URLs of all the FTL servers belonging to the cluster at the DR site. Each URL must be of the form `<FTL server name>@<host>:<port>`. If security is configured for the FTL servers at the primary site, the `user` and `password` parameters will also need to be added to the YAML configuration.

```
servers:
  <name of FTL server #1>
  # ...
  - realm:
      drto: <name of DR FTL server #1>@<host>:<port>|<name of DR FTL
server #2>@<host>:<port>|<name of DR FTL server #3>@<host>:<port>
      user: admin
      password: file:<path to password_file>
  <name of FTL server #2>
  # ...
  - realm:
      drto: <name of DR FTL server #1>@<host>:<port>|<name of DR FTL
server #2>@<host>:<port>|<name of DR FTL server #3>@<host>:<port>
      user: admin
      password: file:<path to password_file>
  <name of FTL server #3>
```

```
  # ...
  - realm:
      drto: <name of DR FTL server #1>@<host>:<port>|<name of DR FTL
server #2>@<host>:<port>|<name of DR FTL server #3>@<host>:<port>
      user: admin
      password: file:<path to password_file>
```

3. Start (or restart) the FTL server cluster at the primary site.
4. Copy the YAML configuration of the FTL server cluster at the primary site to the DR site. If security is configured at the primary site, also copy over the trustfile, keystore, keystore_password_file, password_file, and users.txt file. Rename the FTL servers in the YAML configuration file such that there are no repeated FTL server names between the primary and DR sites. Remove all occurrences of the `drto` parameter. Alter any URLs in the `core.servers` list and `-listens` parameters that overlap with URLs in the primary site. Also modify any invalid paths present in the configuration.

   Once the YAML has been modified for the DR site, add the `drfor` parameter. This parameter must be supplied with a pipe-separated list of URLs of the FTL servers in the primary site's cluster. Each URL must be of the form `<FTL server name>@<host>:<port>`. If security is configured for the FTL servers at the primary site, the `user` and `password` parameters will also need to be added to the YAML configuration.

```
servers:
  <name of DR FTL server #1>
  # ...
  - realm
      drfor: <name of FTL server #1>@<host>:<port>|<name of FTL server
#2>@<host>:<port>|<name of FTL server #3>@<host>:<port>
      user: admin
      password: file:<path to password_file>
  <name of DR FTL server #2>
  # ...
  - realm
      drfor: <name of FTL server #1>@<host>:<port>|<name of FTL server
#2>@<host>:<port>|<name of FTL server #3>@<host>:<port>
      user: admin
      password: file:<path to password_file>
  <name of DR FTL server #3>
  # ...
  - realm
      drfor: <name of FTL server #1>@<host>:<port>|<name of FTL server
```

```
#2>@<host>:<port>|<name of FTL server #3>@<host>:<port>
      user: admin
      password: file:<path to password_file>
```

5. Start up the FTL server cluster at the DR site. The primary site's FTL server cluster will connect to the DR site's FTL server cluster at this point.

Recovering After Disaster

In the event that the FTL server cluster – and thereby the EMS servers – at the primary site becomes unavailable, the FTL servers at the DR site will need to be notified that their site of operations has become the new primary.

This can be done by connecting the EMS admin tool to any one of the EMS servers and issuing the `activate_dr_site` command.

```
echo activate_dr_site > admin.script
tibemsadmin -server <EMS server URL> -script admin.script
```

Issuing this command will cause one of the EMS servers that is not configured as `-standby_only` to transition into active state.

Re-establishing a Disaster Recovery Site

Once the issues at the original primary site have been fixed, it can then be used as the new DR site for the current primary site. To do this the following steps will need to be performed.

Procedure

1. At the new DR site, start by deleting the general data directories and FTL store-specific data directories of the previous FTL server cluster to remove any residual artifacts.
2. Copy the YAML configuration from the FTL server cluster at the new primary site to the new DR site. If security is configured at the new primary site, also copy over the trustfile, keystore, keystore_password_file, password_file, and users.txt file. Rename the FTL servers in the YAML configuration file so that they have the exact same names that were used by the FTL servers in the original primary site. Alter any URLs in the `core.servers` list and `-listens` parameters that overlap with URLs in the new

primary site. Also modify any invalid paths present in the configuration.

Once the above changes have been made, replace the value of the `drfor` parameter with a pipe-separated list of URLs of the FTL servers in the new primary site cluster. Each URL must be of the form `<FTL server name>@<host>:<port>`. If security is configured for the FTL servers at the new primary site, the `user` and `password` parameters will also need to be added to the YAML configuration.

```
servers:
  <name of DR FTL server #1>
  # ...
  - realm
      drfor: <name of FTL server #1>@<host>:<port>|<name of FTL server
#2>@<host>:<port>|<name of FTL server #3>@<host>:<port>
      user: admin
      password: file:<path to password_file>
  <name of DR FTL server #2>
  # ...
  - realm
      drfor: <name of FTL server #1>@<host>:<port>|<name of FTL server
#2>@<host>:<port>|<name of FTL server #3>@<host>:<port>
      user: admin
      password: file:<path to password_file>
  <name of DR FTL server #3>
  # ...
  - realm
      drfor: <name of FTL server #1>@<host>:<port>|<name of FTL server
#2>@<host>:<port>|<name of FTL server #3>@<host>:<port>
      user: admin
      password: file:<path to password_file>
```

3. Start up the FTL server cluster at the new DR site.
4. The FTL server cluster at the new primary site will need to be informed that it needs to start replicating to a new DR site. This can be done by connecting the EMS admin tool to the active EMS server and issuing the `setup_dr_site` command. This command must be provided with a pipe-separated list of URLs of the FTL servers in the new DR site. Each URL must be of the form `<FTL server name>@<host>:<port>`.

```
echo setup_dr_site <name of DR FTL server #1>@<host>:<port>|<name of DR FTL server
#2>@<host>:<port>|<name of DR FTL server #3>@<host>:<port>
> admin.script
tibemsadmin -server <EMS server URL> -script admin.script
```

Developing an EMS Client Application

The following topics outline the development of EMS client applications in Java, C, and C#.

Jakarta Messaging Specification

EMS implements the Jakarta Messaging 2.0 and 3.0 specifications, which are backward compatible with earlier versions of the specification.

While the old JMS 1.0.2b interfaces are still supported, newly developed applications should use the JMS 2.0 or 1.1 interfaces instead. It is recommended to avoid using 1.0.2b interfaces, in particular due to their lack of flexibility. With these, an application initially written to work with topics has to be reworked if it needs to use queues, whereas an application based on the 1.1 or 2.0 APIs relies on a generic destination infrastructure that would not need to be altered significantly.

To get a better understanding and illustration of how the various Jakarta Messaging objects relate to each other, refer to the Jakarta Messaging Specification and to the samples client applications provided with EMS.

The code examples in this chapter illustrate the use of the JMS 2.0 interface.

Jakarta Messaging 3.0.0 Specification

Jakarta EE, formerly Java EE, has renamed the `javax.jms` package into `jakarta.jms` with the Jakarta EE 9 release. The Jakarta Messaging 3.0 specification reflects the corresponding change of namespace. EMS implements both Jakarta Messaging 2.0 (`javax.jms` namespace) and 3.0 (`jakarta.jms` namespace) in the form of two sets of jar files.

To run applications based on the Jakarta Messaging 2.0 API, include in your `CLASSPATH` the relevant jar files from the following list:

- `jakarta.jms-api-2.0.3.jar` (which replaces `jms-2.0.jar`)
- `tibjms.jar`
- `tibjmsadmin.jar`

- `tibjmsufo.jar`
- `tibrvjms.jar`

To run applications based on the Jakarta Messaging 3.0 API, primarily for use in the Jakarta EE 9+ world, use the following files instead:

- `jakarta.jms-api-3/jakarta.jms-api-3.0.0.jar`
- `jakarta.jms-api-3/jakarta.jms-tibjms.jar`
- `jakarta.jms-api-3/jakarta.jms-tibjmsadmin.jar`
- `jakarta.jms-api-3/jakarta.jms-tibjmsufo.jar`
- `jakarta.jms-api-3/jakarta.jms-tibrvjms.jar`

This applies to every mention of a Jakarta Messaging 2.0 based jar file throughout the present book.

Jakarta Messaging 2.0.3 Specification

The JMS specification has been renamed Jakarta Messaging specification as part of Jakarta EE, which has been moved to the Eclipse Foundation. The changes introduced in JMS 2.0.1 and Jakarta Messaging 2.0.2 and 2.0.3 are mostly cosmetic, such as the replacement of "JMS" with "Jakarta Messaging" in the API Javadoc, where appropriate.

JMS 2.0 Specification

The JMS 2.0 specification introduces several new features, including delivery delay, shared subscriptions, asynchronous sending and the Simplified API.

The Simplified API is offered in addition to the API originally provided with JMS 1.1, which is now called the Classic API. The Simplified API is less verbose than the Classic API, and introduces several important new objects:

- **JMSContext**

  Used to create messages, as well as JMS consumers and JMS producers. Each JMS context uses one session and one connection, but does not expose those. Additionally, multiple JMS context objects can share the same connection.

- **JMSConsumer**

A message consumer that has the ability to receive a message body without the need to use a Message object.

- **JMSProducer**

  Similar to an anonymous message producer, and provides a convenient API for configuring delivery options, message properties, and message headers.

Methods in the Simplified API throw unchecked exceptions rather than checked exceptions. For a sample showing the Simplified API in use, see the new Java sample file called `tibjmsJMSContextSendRecv.java`. This sample file demonstrates the Simplified API in the simplest possible way; for greater detail, refer to the Java API Reference Pages.

JMS 1.1 Specification

In the JMS 1.1 specification, applications using the point to point (queues) or publish and subscribe (topics) models use the same interfaces to create objects.

The specification refers to these interfaces as *common facilities* because these interfaces create objects that can be used for either topics or queues.

JMS 1.0.2b Specification

The JMS 1.0.2b specification defined specific interfaces for topics and for queues.

The JMS 1.0.2b interfaces have the same structure as the JMS 1.1 common facilities, but the interfaces are specific to topics or queues.

Sample Clients

TIBCO Enterprise Message Service includes several sample client applications that illustrate various features of EMS. You may wish to view these sample clients when reading about the corresponding features in this manual.

The samples are included in the *EMS_HOME*`/samples/java`, *EMS_HOME*`/samples/c`, and *EMS_HOME*`/samples/cs` subdirectories of the EMS installation directory. Each subdirectory includes a README file that describes how to compile and run the sample clients.

Getting Started walks through the procedures for setting up your EMS environment and running some of the sample clients.

Programmer Checklists

This section provides a checklist that outlines the steps for creating an EMS application in each language:

- Java Programmer's Checklist
- C Programmer's Checklist
- C# Programmer's Checklist

Java Programmer's Checklist

Install

- Install the EMS software release, which automatically includes the EMS jar files in the *EMS_HOME*/lib subdirectory.
- Add the full pathnames for the following jar files to your CLASSPATH:

```
jakarta.jms-api-2.0.3.jar
tibjms.jar
```

- Programs that use the unshared state failover API must add the following file to the CLASSPATH:

```
tibjmsufo.jar
```

- Programs that use OAuth 2.0 authentication must add the following file to the CLASSPATH:

```
json_simple-1.1.jar
```

- Programs that are set to operate in FIPS 140-2 compliant mode must add the

following files to the CLASSPATH:

```
bc-fips-1.0.2.4.jar
bctls-fips-1.0.17.jar
```

Note: All jar files listed in this section are located in the `lib` subdirectory of the TIBCO Enterprise Message Service installation directory.

Code

Import the following packages into your EMS application:

```
import javax.jms.*;
import javax.naming.*;
```

Compile

Compile your EMS application with the `javac` compiler to generate a `.class` file.

For example:

```
javac MyApp.java
```

generates a `MyApp.class` file.

Run

Use the `java` command to execute your EMS `.class` file.

For example:

```
java MyApp
```

C Programmer's Checklist

Developers of EMS C programs can use this checklist during the five phases of the development cycle.

Install

Install the EMS software release, which includes the EMS client libraries, binaries, and header files.

Code

Application programs must:

- Add *EMS_HOME*/include to the include path.
- Include the tibems.h header file:

```
#include <tibems/tibems.h>
```

- Programs that use the C administration API must also include the emsadmin.h header file:

```
#include <tibems/emsadmin.h>
```

- Programs that use the unshared state failover API must also include the tibufo.h header file:

```
#include <tibems/tibufo.h>
```

- Call tibems_Open() to initialize the EMS C API and tibems_Close() to deallocate the memory used by EMS when complete.

Compile and Link

- Compile programs with an ANSI-compliant C compiler.
- Link with the appropriate EMS C library files; see Link These Library Files.

See the samples/c/readme.txt file for details.

Run

- **UNIX**

The library path must include the *EMS_HOME*/lib directories (which contain the shared library files).

- **Windows**

  The PATH must include the ems\10.3\bin directory.

- **All Platforms**

  The application must be able to connect to a EMS server process (tibemsd).

Link These Library Files

EMS C programs must link the appropriate library files. The following sections describe which files to link for your operating system platform:

- UNIX
- Microsoft Windows

UNIX

Include *EMS_HOME*/lib in your library path.

Linker Flag	Description
-ltibems	All programs must link using this library flag.
-ltibemslookup	Programs that reference the defunct EMS LDAP lookup API must link using this library flag. This is provided so as not to break builds but will no longer provide the feature. Non-LDAP JNDI lookups are still supported.
-ltibemsadmin	Programs that use the C administration library must link using this library flag.
-ltibemsufo	Programs that use the unshared state failover library must link using this library flag.

Microsoft Windows

Library File	Description
Use the `/MT` compiler option.	
`tibems.lib`	All programs must link this library.
`tibemslookup.lib`	Programs that reference the defunct EMS LDAP lookup API must link using this library. This is provided so as not to break builds but will no longer provide the feature. Non-LDAP JNDI lookups are still supported.
`tibemsadmin.lib`	Programs that use the C administration library must link using this library.
`tibemsufo.lib`	Programs that use the C unshared state failover library must link using this library.

C# Programmer's Checklist

Developers of EMS C# programs can use this checklist during the four phases of the development cycle.

The EMS .NET client libraries are built to the .NET Standard 2.0 specification. They can be used to build both .NET Framework applications, which can only run on Windows, and .NET Core applications, which can run on both Windows and Linux.

Install

Install the EMS software release, which automatically includes the EMS assembly DLLs in the *EMS_HOME*`\bin` subdirectory.

Code

Import the correct EMS assembly (see the following table).

Version	DLL
.NET API	`TIBCO.EMS.dll`
.NET Administration API	`TIBCO.EMS.ADMIN.dll`
.NET Unshared State API	`TIBCO.EMS.UFO.dll`

Compile

Both .NET Framework and .NET Core applications can be built using the Microsoft `dotnet build` tool, C# project files (`*.csproj`) and, optionally, solution files (`*.sln`).

For example, to build a .NET Framework EMS application:

```
> dotnet build my-EMS-net-program.csproj -f net48
```

This will create a .NET Framework executable application: `my-EMS-net-program.exe`.

And to build a .NET Core EMS application:

```
> dotnet build my-EMS-net-core-program.csproj -f netcoreapp6.0
```

This will create a .NET Core DLL application: `my-EMS-net-core-program.dll`.

The `EMS_HOME/samples/cs` and `EMS_HOME/samples/cs/admin` directories contain sample C# project files (`*.csproj`) and solution files (`*.sln`) that are used to build the .NET Framework and .NET Core sample applications.

Run

The .NET Framework application built in the above example can be executed directly in the .NET Framework environment:

```
> my-EMS-net-program.exe
```

The .NET Core application built in the above example can be executed in the .NET Core runtime environment:

```
> dotnet my-EMS-net-core-program.dll
```

- In the .NET Framework environment, the EMS assembly must be in the global assembly cache (this location is preferred), or in the system path, or in the same directory as your program executable.
- In the .NET Framework environment, to automatically upgrade to the latest .NET assemblies, include the appropriate policy file in the global cache. See Automatic Upgrades Between Versions for more information.
- In the .NET Core environment, the EMS assembly must be in the same directory as your application executable.
- In both the .NET Framework and .NET Core environments, the application must be able to connect to a EMS server process (`tibemsd`).

Assembly Versioning in the Windows .NET Framework Environment

TIBCO Enterprise Message Service assembly DLLs are versioned using the format 1.0.*release*.*version*, where *release* is the EMS release number and *version* is an arbitrary value. For example, the assembly version number for software release 10.3.0 is similar to 1.0.1030.3.

Automatic Upgrades Between Versions

In order to allow for seamless upgrades between releases, the TIBCO Enterprise Message Service installation includes policy and configuration files that redirect existing applications from an older assembly to the newest assembly. There is a policy and configuration file for each EMS library:

- A `policy.1.0.`*assembly* file. For example, `policy.1.0.TIBCO.EMS.dll`. The policy file must be included in the global cache to enable automatic upgrades.
- An *assembly*`.config` file. For example, `TIBCO.EMS.dll.config`. The configuration file must be present when the related policy file is added to the global cache.

The following table shows the policy and configuration files for each EMS assembly.

Version	Files
.NET API	`policy.1.0.TIBCO.EMS.dll` `TIBCO.EMS.dll.config`
.NET Administration API	`policy.1.0.TIBCO.EMS.ADMIN.dll` `TIBCO.EMS.ADMIN.dll.config`
.NET Unshared State API	`policy.1.0.TIBCO.EMS.UFO.dll` `TIBCO.EMS.UFO.dll.config`

Enabling Updates

To enable automatic updates for a library, add the appropriate policy file to the global cache. Note that the related configuration file must be located in the directory with the policy file in order to add the policy file to the global cache.

Disabling Automatic Upgrades

If you do not want your older applications to automatically move to the newer version, do not include the policy DLL in the global cache. When the `policy.1.0.`*assembly* file is absent, the client application is not upgraded.

Running Multiple Clients from Different EMS Releases

To deploy two or more applications that are built with different TIBCO Enterprise Message Service releases:

- Build clients using the different .NET client assemblies.
- Include all desired versions of the .NET client assemblies in the global cache.
- Do not include the `policy` DLL in the global cache.

Excluded Features and Restrictions

This section summarizes features that are not available in the .NET library.

Feature	Framework	Core
Distributed transactions	Yes	No
XA protocols for external transactions managers	No	No
`ConnectionConsumer`, `ServerSession`, `ServerSessionPool`	No	No
LDAP JNDI Lookups	Yes	Windows only

Character Encoding

.NET programs represent strings within messages as byte arrays. Before sending an outbound message, EMS programs translate strings to their byte representation using an encoding, which the program specifies. Conversely, when EMS programs receive inbound messages, they reconstruct strings from byte arrays using the same encoding.

When a program specifies an encoding, it applies to all strings in message bodies (names and values), and properties (names and values). It does not apply to header names nor values. The method `BytesMessage.WriteUTF` always uses UTF-8 as its encoding.

- Outbound Messages

  Programs can determine the encoding of strings in outbound messages in three ways:

  - Use the default global encoding, UTF-8.
  - Set a non-default global encoding (for all outbound messages) using `Tibems.SetEncoding`.
  - Set the encoding for an individual message using `Tibems.SetMessageEncoding`.

- Inbound Messages

  An inbound message from another EMS client explicitly announces its encoding. A receiving client decodes the message using the proper encoding.

For more information about character encoding, see Character Encoding in Messages.

Connection Factories

A client must connect to a running instance of the EMS server to perform any Jakarta Messaging operations. A connection factory is an object that encapsulates the data used to define a client connection to an EMS server. The minimum factory parameters are the type of connection and the URL for the client connection to the EMS server.

A connection factory is either dynamically created by the application or obtained from a data store by means of a naming service, such as a Java Naming and Directory Interface (JNDI) server or a Lightweight Directory Access Protocol (LDAP) server.

Looking up Connection Factories

EMS provides a JNDI implementation that can be used to store connection factories. Java, C, and C# clients can use the EMS JNDI implementation to lookup connection factories.

You can also store connection factories in any JNDI-compliant naming service or in an LDAP server. Java clients can lookup connection factories in any JNDI-compliant naming service. C# clients can use LDAP servers but C clients cannot.

Look up Administered Objects Stored in EMS describes how to lookup a connection factory from an EMS server. How to create connection factories in a EMS server is described in Create and Modify Administered Objects in EMS.

Dynamically Creating Connection Factories

Normally client applications use JNDI to look up a Connection Factory object. However, some situations require clients to connect to the server directly. To connect to the EMS server directly, the application must dynamically create a connection factory.

The following examples show how to create a connection factory in each supported language for Jakarta Messaging connections. Each API also supports connection factories for Jakarta Messaging XA connections.

In each example, the `serverUrl` parameter in these expressions is a string defining the protocol and the address of the running instance of the EMS Server. The `serverUrl` parameter has the form:

```
serverUrl = protocol://host:port
```

The supported *protocols* are tcp and ssl. For example:

```
serverUrl = tcp://server0:7222
```

For a fault-tolerant connection, you can specify two or more URLs. For example:

```
serverUrl = tcp://server0:7222,tcp://server1:7344
```

See Configure Clients for Shared State Failover Connections for more information. For details on using TLS for creating secure connections to the server, see Configure TLS in EMS Clients and Create Connection Factories for Secure Connections.

- Java

  To dynamically create a `TibjmsConnectionFactory` object in a Java client:

  ```
  ConnectionFactory factory = new com.tibco.tibjms.TibjmsConnectionFactory
  (serverUrl);
  ```

  See the `tibjmsMsgProducer.java` sample client for a working example.

- C

  To dynamically create a `tibemsConnectionFactory` type in a C client:

  ```
  factory = tibemsConnectionFactory_Create();
  status = tibemsConnectionFactory_SetServerURL(factory, serverUrl);
  ```

  See the `tibemsMsgProducer.c` sample client for a working example.

- C# To dynamically create a `ConnectionFactory` object in a C# client:

  ```
  ConnectionFactory factory = new TIBCO.EMS.ConnectionFactory(serverUrl);
  ```

  See the `csMsgProducer.cs` sample client for a working example.

Set Connection Attempts, Timeout, and Delay Parameters

By default, a client will attempt to connect to the server two times with a 500 ms delay between each attempt.

A client can modify this behavior by setting new connection attempt count and delay values. There are also a number of factors that may cause a client to hang while attempting to create a connection to the EMS server, so you can set a connection timeout value to abort a connection attempt after a specified period of time. For best results, timeouts should be at least 500 milliseconds. EMS also allows you to establish separate count, delay and timeout settings for reconnections after a fault-tolerant failover, as described in Set Reconnection Failure Parameters.

The following examples establish a connection count of 10, a delay of 1000 ms and a timeout of 1000 ms.

- Java

  Use the `TibjmsConnectionFactory` object's `setConnAttemptCount()`, `setConnAttemptDelay()`, and `setConnAttemptTimeout()` methods to establish new connection failure parameters:

  ```
  factory.setConnAttemptCount(10);
  factory.setConnAttemptDelay(1000);
  factory.setConnAttemptTimeout(1000);
  ```

- C

  Use the `tibemsConnectionFactory_SetConnectAttemptCount` and `tibemsConnectionFactory_SetConnectAttemptDelay` functions to establish new connection failure parameters:

  ```
  status = tibemsConnectionFactory_SetConnectAttemptCount(factory, 10);
  status = tibemsConnectionFactory_SetConnectAttemptDelay(factory, 1000);
  status = tibemsConnectionFactory_SetConnectAttemptTimeout(factory, 1000);
  ```

- C#

  Use the `ConnectionFactory.SetConnAttemptCount`, `ConnectionFactory.SetConnAttemptDelay`, and `ConnectionFactory.SetConnAttemptTimeout` and methods to establish new connection failure parameters:

  ```
  factory.setConnAttemptCount(10);
  factory.setConnAttemptDelay(1000);
  factory.setConnAttemptTimeout(1000);
  ```

Connect to the EMS Server

A connection with the EMS server is defined by the Connection object obtained from a Connection Factory.

For more information, see Connection Factories.

A connection is a fairly heavyweight object, so most clients will create a connection once and keep it open until the client exits. Your application can create multiple connections, if necessary.

The following examples show how to create a Connection object.

- Java

  Use the `TibjmsConnectionFactory` object's `createConnection()` method to create a `Connection` object:

  ```
  Connection connection = factory.createConnection(userName, password);
  ```

  See the `tibjmsMsgProducer.java` sample client for a working example.

- C

  Use the `tibemsConnectionFactory_CreateConnection` function to create a connection of type `tibemsConnection`:

  ```
  tibemsConnection connection = NULL;
  status = tibemsConnectionFactory_CreateConnection(factory, &connection,
  userName, password);
  ```

  If there is no connection factory, a C client can use the `tibemsConnection_Create` function to dynamically create a `tibemsConnection` type:

  ```
  status = tibemsConnection_Create(&connection, serverUrl, NULL, userName,
  password);
  ```

  The `tibemsConnection_Create` function exists for backward compatibility, but the recommended procedure is that you create `tibemsConnection` objects from factories.

  See the `tibemsMsgProducer.c` sample client for a working example.

- C#

  Use the `ConnectionFactory.CreateConnection` method to create a `Connection` object:

```
Connection connection =factory.CreateConnection(userName, password);
```

See the `csMsgProducer.cs` sample client for a working example.

Start, Stop and Close a Connection

Before consuming messages, the Message Consumer client must "start" the connection. If you wish to temporarily suspend message delivery, you can "stop" the connection. When a client application exits, all open connections must be "closed."

See Create a Message Consumer for more details about Message Consumers.

Unused open connections are eventually closed, but they do consume resources that could be used for other applications. Closing a connection also closes any sessions created by the connection.

See the "start," "stop" and "close" methods for the Java `Connection` object, the C `tibemsConnection` type, and the C# `Connection` object.

Create a Session

A Session is a single-threaded context for producing or consuming messages. You create Message Producers or Message Consumers using Session objects.

A Session can be transactional to enable a group of messages to be sent and received in a single transaction. A non-transactional Session can define the acknowledge mode of message objects received by the session. See Message Acknowledgement for details.

- Java

  Use the `Connection` object's `createSession()` method to create a `Session` object.

  For example, to create a `Session` that uses the default `AUTO_ACKNOWLEDGE` session mode:

  ```
  Session session = connection.createSession();
  ```

  The EMS extended session modes, such as `NO_ACKNOWLEDGE`, require that you include the `com.tibco.tibjms.Tibjms` constant when you specify the EMS session mode. For

example, to create a `Session` that uses the `NO_ACKNOWLEDGE` session mode:

```
Session session =
connection.createSession(com.tibco.tibjms.Tibjms.NO_ACKNOWLEDGE);
```

See the `tibjmsMsgProducer.java` sample client for a working example.

- C

  Use the `tibemsConnection_CreateSession` function to create a session of type `tibemsSession`:

  ```
  tibemsSession session = NULL;
  status = tibemsConnection_CreateSession(connection,&session, TIBEMS_FALSE,
  TIBEMS_AUTO_ACKNOWLEDGE);
  ```

  See the `tibemsMsgProducer.c` sample client for a working example.

- C#

  Use the `Connection.CreateSession` method to create a `Session` object:

  ```
  Session session = connection.CreateSession(false, Session.AUTO_ACKNOWLEDGE);
  ```

  See the `csMsgProducer.cs` sample client for a working example.

Set an Exception Listener

All the APIs support the ability to set an exception listener on the connection that gets invoked when a connection breaks or experiences a fault-tolerant failover.

When the event is a disconnect, the exception handler can call various EMS methods without any problem. However, when the event is a fault-tolerant failover, the exception handler is not allowed to call any EMS method. To do so risks a deadlock. You can call the `setExceptionOnFTSwitch` method to receive an exception that contains the new server URL after a fault-tolerant failover has occurred.

The following examples demonstrate how to establish an exception listener for a connection.

- Java

  Implement an `ExceptionListener.onException` method, use the `Connection`

object's `setExceptionListener` method to register the exception listener, and call `Tibjms.setExceptionOnFTSwitch` to call the exception handler after a fault-tolerant failover:

```
public class tibjmsMsgConsumer
   implements ExceptionListener
{
.....
public void onException(JMSException e)
     {
       /* Handle exception */
     }
.....
connection.setExceptionListener(this);
com.tibco.tibjms.Tibjms.setExceptionOnFTSwitch(true);
.....
}
```

See the `tibjmsMsgConsumer.java` sample client for a working example (without the `setExceptionOnFTSwitch` call).

- C

  Define an `onException` function to handle exceptions,
  use the `tibemsConnection_SetExceptionListener` function to call `onException` when an error is encountered, and call `tibems_setExceptionOnFTSwitch` to call the exception handler after a fault-tolerant failover:

```
void onException(
  tibemsConnection    conn,
  tibems_status       reason,
  void*               closure)
 {
   /* Handle exception */
 }
 .....
status = tibemsConnection_SetExceptionListener(
                                       connection,
                                       onException,
                                       NULL);
tibems_setExceptionOnFTSwitch(TIBEMS_TRUE);
```

See the `tibemsMsgConsumer.c` sample client for a working example (without the `setExceptionOnFTSwitch` call).

- C#

  Implement an `IExceptionListener.OnException` method, set the `Connection` object's `ExceptionListener` property to register the exception listener, and call `Tibems.SetExceptionOnFTSwitch` to call the exception handler after a fault-tolerant failover:

```
  public class csMsgConsumer : IExceptionListener
  {
.....
  public void OnException(EMSException e)
      {
     /* Handle exception */
      }
.....
  connection.ExceptionListener = this;
  TIBCO.EMS.Tibems.SetExceptionOnFTSwitch(true);
.....
  }
```

  See the csMsgConsumer.cs sample client for a working example (without the `setExceptionOnFTSwitch` call).

Dynamically Create Topics and Queues

EMS provides a JNDI implementation that can be used to store topics and queues. Java, C, and C# clients can use the EMS JNDI implementation to lookup topics and queues.

You can also store topics and queues in any JNDI-compliant naming service or in an LDAP server. Java clients can lookup topics and queues in any JNDI-compliant naming service. C# clients can use LDAP servers but C clients cannot.

Look up Administered Objects Stored in EMS describes how to lookup topics and queues from an EMS server.

Clients can also create destinations as needed. If a client requests the creation of a destination that already exists, the existing destination is used. If the destination does not exist, and the specification of the `topics.conf`, `queues.conf`, or `acl.conf` files allow the destination, the server dynamically creates the new destination. The new destination inherits properties and permissions from its ancestors as described in Wildcards and Dynamically Created Destinations. The destination is managed by the server as long as clients that use the destination are running.

Note: Because dynamic destinations do not appear in the configuration files, a client cannot use JNDI to lookup dynamically created queues and topics.

The following examples show how to create destinations dynamically:

- Java

  Use the `Session` object's `create topic` method to create a topic as a `Destination` object:

  ```
  Destination topic = session.createTopic(topicName);
  ```

  Use the `Session` object's `createQueue()` method to create a queue as a `Destination` object:

  ```
  Destination queue = session.createQueue(queueName);
  ```

  See the `tibjmsMsgProducer.java` sample client for a working example.

- C

  Use the `tibemsTopic_Create` function to create a topic of type `tibemsDestination`:

  ```
  tibemsDestination topic = NULL;
  status = tibemsTopic_Create(&topic,topicName);
  ```

  Use the `tibemsQueue_Create` function to create a queue of type `tibemsDestination`:

  ```
  tibemsDestination queue = NULL;
  status = tibemsQueue_Create(&queue,queueName);
  ```

  See the `tibemsMsgProducer.c` sample client for a working example.

- C#

  Use the `Session.CreateTopic` method to create a `Topic` object:

  ```
  Destination topic = session.CreateTopic(topicName);
  ```

  Use the `Session.CreateQueue` method to create a `Queue` object:

  ```
  Destination queue = session.CreateQueue(queueName);
  ```

See the `csMsgProducer.cs` sample client for a working example.

Create a Message Producer

A *Message Producer* is an EMS client that either publishes messages to a topic or sends messages to a queue. When working with topics, a Message Producer is commonly referred to as a *Publisher*.

Optionally, when creating a Message Producer, you can set the destination to NULL and specify the destination when you send or publish a message, as described in Send Messages.

You must have send permission on a queue to create a message producer that sends messages to that queue. You must have durable permission on the topic to create a new durable subscriber for that topic, and have at least use_durable permission on the topic to attach to an existing durable subscriber for the topic. See User Permissions for details.

The following examples create a message producer that sends messages to the queue that was dynamically created in Dynamically Create Topics and Queues.

- Java

  Use the `Session` object's `createProducer()` method to create a `MessageProducer` object:

  ```
  MessageProducer QueueSender = session.createProducer(queue);
  ```

  See the `tibjmsMsgProducer.java` sample client for a working example.

- C

  Use the `tibemsSession_CreateProducer` function to create a message producer of type `tibemsMsgProducer`:

  ```
  tibemsMsgProducer QueueSender = NULL;
  status = tibemsSession_CreateProducer(session,&QueueSender,queue);
  ```

  See the `tibemsMsgProducer.c` sample client for a working example.

- C#

  Use the `Session.CreateProducer` method to create a `MessageProducer` object:

```
MessageProducer QueueSender = session.CreateProducer(queue);
```

See the `csMsgProducer.cs` sample client for a working example.

Configure a Message Producer

A message producer can be configured to generate messages with default headers and properties that define how those messages are to be routed and delivered.

Specifically, you can:

- Set the producer's default delivery mode.
- Set whether message IDs are disabled.
- Set whether message timestamps are disabled.
- Set the producer's default priority.
- Set the default length of time that a produced message should be retained by the message system.

For example, as described in the Message Delivery Modes, you can set the message deliver mode to either `PERSISTENT`, `NON_PERSISTENT`, or `RELIABLE_DELIVERY`.

- Java

  Use the `MessageProducer` object's `setDeliveryMode()` method to configure your Message Producer with a default delivery mode of `RELIABLE_DELIVERY`:

  ```
  QueueSender.setDeliveryMode(com.tibco.tibjms.Tibjms.RELIABLE_DELIVERY);
  ```

  To configure the Message Producer with a default delivery mode of `NON_PERSISTENT`:

  ```
  QueueSender.setDeliveryMode(javax.jms.DeliveryMode.NON_PERSISTENT);
  ```

  See the `tibjmsMsgProducerPerf.java` sample client for a working example.

Note: Delivery mode cannot be set by using the `Message.setJMSDeliveryMode()` method. According to the Jakarta Messaging specification, the publisher ignores the value of the `JMSDeliveryMode` header field when a message is being published.

- C

  Use the `tibemsMsgProducer_SetDeliveryMode` function to configure your Message Producer to set a default delivery mode for each message it produces to `RELIABLE_DELIVERY`:

  ```
  tibems_int deliveryMode = TIBEMS_RELIABLE;
  status tibemsMsgProducer_SetDeliveryMode(QueueSender, deliveryMode);
  ```

- C#

  Set the `DeliveryMode` on the `MessageProducer` object to `RELIABLE_DELIVERY`:

  ```
  QueueSender.DeliveryMode = DeliveryMode.RELIABLE_DELIVERY;
  ```

  See the `csMsgProducerPerf.cs` sample client for a working example.

Create a Completion Listener for Asynchronous Sending

TIBCO Enterprise Message Service provides APIs for a Message Producer to send messages either synchronously or asynchronously. For asynchronous sending, you need to implement a `CompletionListener` that serves as an asynchronous event handler for message send result notification.

A completion listener implementation has two methods: `onCompletion()` is invoked after a message has successfully been sent, and `onException()` is invoked if the send failed. These methods are invoked in a different thread from that in which the message was sent. You implement the methods to perform the desired actions when the application is notified of send success or failure. Your implementation should handle all exceptions, and it should not throw any exceptions.

Once you create a completion listener, you pass it as an argument into the `MessageProducer` send method, or into the JMSProducer `setAsync()` method. If passed into the JMSProducer `setAsync` method, the JMSProducer will always send asynchronously.

- Java

  Create an implementation of the CompletionListener interface, create a

CompletionListener and pass that into the appropriate send method:

```
/* create connection, session, producer, message */
TibjmsCompletionListener completionListener = new
  TibjmsCompletionListener();
msgProducer.send(destination, msg, completionListener);
```

Create a CompletionListener class and Implement the `onCompletion()` and `onException()` method to perform the desired actions when a message arrives:

```
class TibjmsCompletionListener implements CompletionListener
{
  public void onCompletion(Message msg)
  {
    /* Handle the send success case for the message */
  }
  public void onException(Message msg, Exception ex)
  {
    /* Handle the send failure case for the message */
  }
}
```

See the `tibjmsMsgProducer.java` sample client for a working example.

- C

  In C, Implement an `onCompletion()` function to perform the desired actions when a message is sent:

```
static void
onCompletion(tibemsMsg msg, tibems_status status, void* closure)
{
    if (status == TIBEMS_OK)
    {
        /* Handle the send success case for the message */
    }
    else
    {
        /* Handle the send failure case for the message */
    }
}
/* Create a connection, session, and producer. When sending, pass
 * the onCompletion() function as the tibemsMsgCompletionCallback
 */
status = tibemsMsgProducer_AsyncSend(producer, msg, onCompletion,
NULL);
```

```
```

See the `tibemsMsgProducer.c` sample client for a working example.

- C#

  Create an implementation of the `ICompletionListener` interface, create a `CompletionListener` and pass that into the appropriate send method.

```
EMSCompletionListener completionListener = new EMSCompletionListener();
producer.Send(destination, msg, completionListener);
```

  Create an implementation of the `IMessageListener` interface to perform actions when a message is sent:

```
class EMSCompletionListener : ICompletionListener
{
  public void OnCompletion(Message msg)
  {
    /* Handle the send success case for the message */
  }
  public void OnException(Message msg, Exception ex)
  {
    /* Handle the send failure case for the message */
  }
}
```

  See the `csMsgProducer.cs` sample client for a working example.

Create a Message Consumer

Message consumers are clients that receive messages published to a topic or sent to a queue. When working with topics, a Message Consumer is commonly referred to as a *Subscriber*.

A Message Consumer can be created with a "message selector" that restricts the consumption of message to those with specific properties. When creating a Message Consumer for topics, you can set a `noLocal` attribute that prohibits the consumption of messages that are published over the same connection from which they are consumed.

Carefully consider the message selectors that are used with queue consumers. Because messages that do not match a queue consumer's message selectors remains in the queue until it is retrieved by another consumer, a non-matching message can experience many

failed selectors. This is especially so when queue consumers connect, consume a message, and immediately disconnect.

As described in Durable Subscribers for Topics, messages published to topics are only consumed by active subscribers to the topic; otherwise the messages are not consumed and cannot be retrieved later. You can create a durable subscriber that ensures messages published to a topic are received by the subscriber, even if it is not currently running. For queues, messages remain on the queue until they are either consumed by a Message Consumer, the message expiration time has been reached, or the maximum size of the queue is reached.

The following examples create a Message Consumer that consumes messages from the queue and a durable subscriber that consumes messages from a topic. The queue and topic are those that were dynamically created in Dynamically Create Topics and Queues.

Note: The `createDurableSubscriber` method either creates a new durable subscriber for a topic or attaches the client to a previously created durable subscriber. A user must have durable permission on the topic to create a new durable subscriber for that topic. A user must have at least use_durable permission on the topic to attach to an existing durable subscriber for the topic. See User Permissions for details.

- Java

  Use the `Session` object's `createConsumer()` method to create a `MessageConsumer` object:

  ```
  MessageConsumer QueueReceiver = session.createConsumer(queue);
  ```

  See the `tibjmsMsgConsumer.java` sample client for a working example.

  The following `Session.createDurableSubscriber()` method creates a durable subscriber, named "MyDurable":

  ```
  TopicSubscriber subscriber = session.createDurableSubscriber
(topic,"myDurable");
  ```

  See the `tibjmsDurable.java` sample client for a working example.

  Shared Subscriptions

  Use the Session object's `createSharedConsumer()` method to create or add to a shared subscription:

```
MessageConsumer cons1 = session.createSharedConsumer(topic, "mySharedSub");
MessageConsumer cons2 = session.createSharedConsumer(topic, "mySharedSub");
```

`cons1` and `cons2` are two shared consumers on the same subscription called `mySharedSub`. If a message is published to the topic, then one of those two consumers will receive it. Note that shared consumers on a given subscription do not have to use the same session/connection.

Use the Session object's `createSharedDurableConsumer()` method to create or add to a shared durable subscription:

```
MessageConsumer cons1 = session.createSharedDurableConsumer(topic,
"myDurableSharedSub"); MessageConsumer cons2 =
session.createSharedDurableConsumer(topic, "myDurableSharedSub");
```

`cons1` and `cons2` are two shared durable consumers on the same durable subscription called `myDurableSharedSub`. If a message is published to the topic, then one of those two consumers will receive it. Note that shared durable consumers on a given subscription do not have to use the same session/connection.

- C

  Use the `tibemsSession_CreateConsumer` function to create a message consumer of type `tibemsMsgConsumer`:

  ```
     tibemsMsgConsumer QueueReceiver = NULL;
     status = tibemsSession_CreateConsumer(session, &QueueReceiver, queue,
  NULL, TIBEMS_FALSE);
  ```

  See the `tibemsMsgConsumer.c` sample client for a working example.

  The following `tibemsSession_CreateDurableSubscriber` function creates a durable subscriber, named "myDurable," of type `tibemsMsgConsumer`:

  ```
     tibemsMsgConsumer msgConsumer = NULL;
     status = tibemsSession_CreateDurableSubscriber(session, &msgConsumer,
  topic, "myDurable", NULL, TIBEMS_FALSE);
  ```

  See the `tibemsDurable.c` sample client for a working example.

- C#

  Use the `Session.CreateConsumer` method to create a `MessageConsumer` object:

```
MessageConsumer QueueReceiver = session.createConsumer(queue);
```

See the `csMsgConsumer.cs` sample client for a working example.

The following `Session.CreateDurableSubscriber` method creates a durable subscriber, named "MyDurable":

```
TopicSubscriber subscriber =
       session.CreateDurableSubscriber(topic, "myDurable");
```

See the `csDurable.cs` sample client for a working example.

Create a Message Listener for Asynchronous Message Consumption

EMS allows a Message Consumer to consume messages either synchronously or asynchronously. For synchronous consumption, the Message Consumer explicitly calls a receive method on the topic or queue. For asynchronous consumption, you can implement a *Message Listener* that serves as an asynchronous event handler for messages.

A Message Listener implementation has one method, `onMessage`, that is called by the EMS server when a message arrives on a destination. You implement the `onMessage` method to perform the desired actions when a message arrives. Your implementation should handle all exceptions, and it should not throw any exceptions.

Once you create a Message Listener, you must register it with a specific Message Consumer before calling the connection's `start` method to begin receiving messages.

A Message Listener is not specific to the type of the destination. The same listener can obtain messages from a queue or a topic, depending upon the destination set for the Message Consumer with which the listener is registered.

Note: The J2EE 1.3 platform introduced message-driven beans (MDBs) that are a special kind of Message Listener. See the J2EE documentation for more information about MDBs.

- Java

  Create an implementation of the `MessageListener` interface, create a

MessageConsumer, and use the MessageConsumer object's setMessageListener() method to register the Message Listener with the Message Consumer:

```
public class tibjmsAsyncMsgConsumer implements MessageListener
{
   /* Create a connection, session and consumer */
...
   MessageConsumer QueueReceiver =
                              session.createConsumer(queue);
   QueueReceiver.setMessageListener(this);
   connection.start();
}
```

Note: Do not use the Session.setMessageListener() method, which is used by application servers, rather than by applications.

Implement the onMessage() method to perform the desired actions when a message arrives:

```
public void onMessage(Message message)
{
     /* Process message and handle exceptions */
}
```

See the tibjmsAsyncMsgConsumer.java sample client for a working example.

- C

  Implement an onMessage() function to perform the desired actions when a message arrives:

```
void onMessage(tibemsMsgConsumer QueueReceiver,
          tibemsMsg message, void* closure)
{
     /* Process message and handle exceptions */
}
```

  In another function, that creates a tibemsMsgConsumer and uses the tibemsMsgConsumer_SetMsgListener function to create a message listener for the Message Consumer, specifying onMessage() as the callback function:

```
void run()
{
```

```
    tibemsMsgConsumer QueueReceiver = NULL;
    /* Create a connection, session and consumer */
    ...
    status = tibemsSession_CreateConsumer(session,
                 &QueueReceiver, queue, NULL, TIBEMS_FALSE);
    status = tibemsMsgConsumer_SetMsgListener(QueueReceiver,
                 onMessage, NULL);
    status = tibemsConnection_Start(connection);
}
```

See the `tibemsAsyncMsgConsumer.c` sample client for a working example.

- C#

  Create an implementation of the `IMessageListener` interface, use `Session.CreateConsumer` to create a `MessageConsumer`, and set the `MessageListener` property on the `MessageConsumer` object to register the Message Listener with the Message Consumer:

```
    public class csAsyncMsgConsumer : IMessageListener
    {
       /* Create a connection, session and consumer */
    ...
       MessageConsumer QueueReceiver =
                   session.CreateConsumer(queue);
       QueueReceiver.MessageListener = this;
       connection.Start();
    }
```

  Implement the `IMessageListener.OnMessage` method to perform the desired actions when a message arrives:

```
     public void OnMessage(Message message) {
         try
         {
             /* Process message and handle exceptions */
         }
     }
```

  See the `csAsyncMsgConsumer.cs and csAsyncMsgConsumerUsingDelegate.cs` sample clients for working examples.

Messages

Messages are a self-contained units of information used by Jakarta Messaging applications to exchange data or request operations.

Create Messages

As described in Jakarta Messaging Message Bodies , EMS works with the following types of messages:

- Messages with no body
- Text Messages
- Map Messages
- Bytes Messages
- Stream Messages
- Object Messages

There is a separate create method for each type of message.

The following examples show how to create a simple text message containing the string "Hello."

- Java

  Use the `Session` object's `createTextMessage()` method to create a `TextMessage`:

  ```
  TextMessage message = session.createTextMessage("Hello");
  ```

  See the `tibjmsMsgProducer.java` sample client for a working example.

- C

  Use the `tibemsTextMsg_Create` function to create a text message of type `tibemsTextMsg`:

  ```
  tibemsTextMsg message = "Hello";
  status = tibemsTextMsg_Create(&message);
  ```

  See the `tibemsMsgProducer.c` sample client for a working example.

- C#

  Use the `Session.CreateTextMessage` method to create text message of type `TextMessage`:

  ```
  TextMessage message = session.CreateTextMessage("Hello");
  ```

  See the `csMsgProducer.cs` sample client for a working example.

Set and Get Message Properties

Before a client sends a message, it can use a "set property" method to set the message properties. The client can check the message properties with a "get property" method.

For more information on message properties, see EMS Message Properties.

- Java

  Use the `Message` object's `setBooleanProperty()` method to set the `JMS_TIBCO_PRESERVE_UNDELIVERED` property to `true`:

  ```
  message.setBooleanProperty("JMS_TIBCO_PRESERVE_UNDELIVERED", true);
  ```

  Use the `getStringProperty()` method to get the user ID of the `JMS_TIBCO_SENDER`:

  ```
  userID = message.getStringProperty("JMS_TIBCO_SENDER");
  ```

- C

  Use the `tibemsMsg_SetBooleanProperty` function to set the `JMS_TIBCO_PRESERVE_UNDELIVERED` property to `true`:

  ```
  status = tibemsMsg_SetBooleanProperty(message,
   "JMS_TIBCO_PRESERVE_UNDELIVERED", true);
  ```

  Use the `tibemsMsg_GetStringProperty` function to get the user ID of the `JMS_TIBCO_SENDER`:

  ```
  char* userID = NULL;
  ```

  ```
  status = tibemsMsg_GetStringProperty(message,"JMS_TIBCO_SENDER", &userID);
  ```

- C#

  Use the `Message.SetBooleanProperty` method to set the `JMS_TIBCO_PRESERVE_UNDELIVERED` property to `true`:

  ```
  message.SetBooleanProperty("JMS_TIBCO_PRESERVE_UNDELIVERED", true);
  ```

  Use the `Message.GetStringProperty` method to get the user ID of the `JMS_TIBCO_SENDER`:

  ```
  string userID = message.GetStringProperty("JMS_TIBCO_SENDER");
  ```

Send Messages

Use a Message Producer client to send messages to a destination. You can either send a message to the destination specified by the Message Producer or, if the Message Producer specifies NULL as the destination, you can send a message to a specific destination.

In either case, you can optionally set the `JMSDeliveryMode`, `JMSExpiration`, and `JMSPriority` message header fields described in Jakarta Messaging Message Header Fields when sending each message.

The following examples show different ways to send a text message in each language:

- Send the message to the Message Producer, `QueueSender`, created in Create a Message Producer.
- Use a Message Producer with a NULL destination that sends the message to the topic created in Dynamically Create Topics and Queues.
- Use a Completion Listener, created in Create a Message Listener for Asynchronous Message Consumption, to send the message asynchronously.

See EMS Extensions to Jakarta Messaging Messages for more information about creating messages.

- Java

  Use the `MessageProducer` object's `send()` method to send a message to the destination specified by the `MessageProducer` object:

```
QueueSender.send(message);
```

Use the following form of the `send()` method to send a message to a specific destination:

```
MessageProducer NULLsender = session.createProducer(null);
....
NULLsender.send(topic, message);
```

Use the form of the `send()` method with a completion listener argument to send a message asynchronously:

```
QueueSender.send(message, completionListener);
```

See the `tibjmsMsgProducer.java` sample client for a working example.

- C

  Use the `tibemsMsgProducer_Send` function to send a message to the destination specified by the `tibemsMsgProducer`:

  ```
  status = tibemsMsgProducer_Send(QueueSender, message);
  ```

  Use the `tibemsMsgProducer_SendToDestination` function to send the message to a specific destination:

  ```
  status = tibemsMsgProducer_SendToDestination(NULLsender,
                                               topic, message);
  ```

  See the `tibemsMsgProducer.c` sample client for a working example.

  > **Note:** Unlike the Java and C# APIs, in the C API, you can use the `tibemsMsgProducer_SendToDestination` function to specify the destination regardless of whether a destination is in the `tibemsMsgProducer`.

- C#

  Use the `MessageProducer.Send` method to send a message to the destination specified by the `MessageProducer`:

```
QueueSender.Send(message);
```

Use the following form of the `MessageProducer.Send` method to send a message to a specific destination:

```
MessageProducer NULLsender = session.CreateProducer(NULL);
NULLsender.Send(topic, message);
```

See the `csMsgProducer.cs` sample client for a working example.

Receive Messages

A Message Consumer receives messages from a destination and acknowledges the receipt of messages using the mode established for the session, as described in Create a Session.

Before receiving messages, the Message Consumer must start the connection to the EMS server. Before exiting, the Message Consumer must close the connection.

The following examples start the connection created in Connect to the EMS Server; synchronously receive messages from the queue created in Dynamically Create Topics and Queues, and then close the connection.

Note: You can also implement a Message Listener for your Message Consumer to asynchronously receive messages, as described in Create a Message Listener for Asynchronous Message Consumption.

- Java

  Use the `Connection` object's `start()` method to start the connection:

  ```
  connection.start();
  ```

  Use the `MessageConsumer` object's `receive()` method to receive a message. This is typically used in a loop for the duration the client wishes to receive messages:

  ```
  Message message = QueueReceiver.receive();
  ```

  When the client has finished receiving messages, it uses the `Close()` method to close the connection:

```
connection.close();
```

See the `tibjmsMsgConsumer.java` sample client for a working example.

- C

  Use the `tibemsConnection_Start` function to start the connection:

  ```
  status = tibemsConnection_Start(connection);
  ```

  Use the `tibemsMsgConsumer_Receive` function to receive a message. This is typically used in a loop for the duration the client wishes to receive messages:

  ```
  tibemsMsg message = NULL;
  status = tibemsMsgConsumer_Receive(QueueReceiver,&message);
  ```

  When the client has finished receiving messages, use the `tibemsConnection_Close` function to close the connection:

  ```
  status = tibemsConnection_Close(connection);
  ```

  See the `tibemsMsgConsumer.c` sample client for a working example.

- C#

  Use the `Connection.Start` function to start the connection:

  ```
  connection.Start();
  ```

  Use the `MessageConsumer.Receive` function to receive a message. This is typically used in a loop for the duration the client wishes to receive messages:

  ```
  Message message = QueueReceiver.receive();
  ```

  When the client has finished receiving messages, use the `Connection.Close` function to close the connection:

  ```
  connection.Close();
  ```

  See the `csMsgConsumer.cs` sample client for a working example.

The EMS Implementation of JNDI

The EMS server provides a implementation of JNDI that enables you to lookup connection factories, topics and queues, which are collectively referred to as administered objects. Java clients can look up administered objects stored in EMS using standard JNDI calls. The C and C# APIs provide similar calls to look up object data in the EMS server.

How to create topics and queues is described in Creating and Modifying Destinations.

Create and Modify Administered Objects in EMS

You can create administered objects for storage in EMS using either the administration tool or the administration APIs, or directly in the configuration files. This section describes how to create administered objects using the administration tool.

To create a connection factory, use the `create factory` command in the EMS Administration Tool. For example, to create a generic connection factory, named *myFactory*, that establishes a TCP connection to port 7344 on *server1*, start the EMS Administration Tool and enter:

```
create factory myFactory generic URL=tcp://server1:7344
```

The connection factory data stored on the EMS server is located in the `factories.conf` file. You can use the `show factories` command to list all of the connection factories on your EMS server and the `show factory` command to show the configuration details of a specific connection factory.

A connection factory may include optional properties for balancing server load and establishing thresholds for attempted connections, as described in Connection Factory Parameters. These properties can be specified when creating the factory or modified for an existing factory using the `addprop factory`, `setprop factory`, and `removeprop factory` commands.

For example, to set the maximum number of connection attempts for the connection factory, *myFactory*, from the default value of 2 to 5, start the EMS Administration Tool and enter:

```
addprop factory myFactory connect_attempt_count=5
```

And to reset the value back to 2, enter:

```
setprop factory myFactory connect_attempt_count=2
```

Create Connection Factories for Secure Connections

This topic describes how to create a static connection factory for establishing a TLS connection.

Similar TLS parameters must be used when looking up the connection factory, as described in Perform Secure Lookups.

Connections that are to be secured using TLS identify the transport protocol as 'ssl' and may include any number of the TLS configuration parameters listed in TLS Server Parameters.

For example, to create a generic connection factory, named *mySecureFactory*, that establishes a TLS connection to port 7243 on *server1*, start the EMS Administration Tool and enter:

```
create factory mySecureFactory generic URL=ssl://server1:7243
```

To create a factory to set up a generic connection and check the server's certificate to confirm the name of the server is `myServer`, enter (all one line):

```
create factory MySSLFactory generic url=ssl://7243
ssl_verify_host=enabled ssl_expected_hostname=myServer ssl_trusted=
certs/server_root.cert.pem
```

To create a factory to set up a topic connection, check the server's certificate (but not the name inside the certificate), and to set the `ssl_auth_only` parameter so that TLS is only used by the client when creating the connection, enter (all one line):

```
create factory AnotherSSLFactory topic url=ssl://7243
ssl_verify_host=enabled ssl_verify_hostname=disabled ssl_trusted=
certs/server_root.cert.pem ssl_auth_only=enabled
```

Note: These samples assume that the certificate `server_root.cert.pem` is located in "certs" subdirectory of the directory where the server is running.

See TLS Protocol for details.

Create Connection Factories for Fault-Tolerant Connections

When connecting a fault-tolerant client to EMS, you must specify two or more EMS servers in your connection factory. When creating a connection factory for a fault-tolerant client, specify multiple server URLs in the `url` argument of the create factory command.

For example, to create a generic connection factory, named *myFtFactory*, that establishes TCP connections to port 7545 on the primary server, *server0*, and port 7344 on the secondary server, *server1*, start the EMS Administration Tool and enter (on one line):

```
create factory myFtFactory generic url=tcp://server0:7545,tcp://server1:7344
```

Should *server0* become unavailable, the client will connect to *server1*. See Fault Tolerance for details.

Look up Administered Objects Stored in EMS

You can lookup objects from an EMS server by name. All clients can lookup objects in the EMS naming service. Alternatively, Java applications can lookup objects in a third-party JNDI server, and C# clients can lookup objects in a third-party LDAP server.

To lookup administered objects stored in EMS, you need to create the initial context that identifies the URL of the naming service provider and any other properties, such as the username and password to authenticate the client to the service. The naming service provider URL has form:

```
tibjmsnaming://host:port
```

The following examples demonstrate how to access Jakarta Messaging administered objects when using TIBCO Enterprise Message Service. Each of these examples assume that

a connection factory, named `ConFac`, exists in the `factories.conf` file, a `topic.sample` topic exists in `topics.conf`, and a `queue.sample` queue exists in `queues.conf`.

- **Java**

  Create an `InitialContext` object for the initial context, which consists of the provider context factory and JNDI provider URL, as well as the username and password to authenticate the client to the EMS server:

  ```
  Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.tibco.tibjms.naming.TibjmsInitialContextFactory");
    env.put(Context.PROVIDER_URL,"tibjmsnaming://localhost:7222");
    env.put(Context.SECURITY_PRINCIPAL, "userName");
    env.put(Context.SECURITY_CREDENTIALS, "password");
  InitialContext jndiContext = new InitialContext(env);
  ```

  Look up a connection factory, named `ConFac`, and destinations, named `topic.sample` and `queue.sample`, from the initial context:

  ```
  ConnectionFactory factory =
        (javax.jms.ConnectionFactory)
        jndiContext.lookup("ConFac");
  javax.jms.Topic sampleTopic =
            (javax.jms.Topic)jndiContext.lookup("topic.sample");
  javax.jms.Queue sampleQueue =
            (javax.jms.Queue)jndiContext.lookup("queue.sample");
  ```

  See the `tibjmsJNDI.java` sample client located in the *EMS_HOME*`/samples/java/JNDI` directory.

- **C**

  Create a `tibemsLookupContext` object for the initial context, which consists of the JNDI provider URL and the username and password to authenticate the client to the EMS server:

  ```
  tibemsLookupContext* contextstatus = NULL;
  status = tibemsLookupContext_Create(
                              &context,
                              "tcp://localhost:7222",
                              "userName",
                              "password");
  ```

  Use the `tibemsLookupContext_LookupConnectionFactory` function to look up a connection factory, named `ConFac`, and use the `tibemsLookupContext_`

LookupDestination function to look up the destinations, named and queue.sample, from the initial context:

```
topic.sample
tibemsConnectionFactory factory = NULL;
tibemsDestination sampleTopic = NULL;
tibemsDestination sampleQueue = NULL;
status = tibemsLookupContext_Lookup(context,
                                    "ConFac",
                                    (void**)&factory);
status = tibemsLookupContext_Lookup(context,
                                    "sample.queue",
                                    (void**)&sampleQueue);
status = tibemsLookupContext_Lookup(context,
                                    "topic.sample,
                                    (void**)&sampleTopic);
```

- **C#**

  Create a ILookupContext object for the initial context, which consists of the JNDI provider URL and the username and password to authenticate the client to the EMS server.

```
Hashtable env = new Hashtable();
  env.Add(LookupContext.PROVIDER_URL,
          "tibjmsnaming://localhost:7222");
  env.Add(LookupContext.SECURITY_PRINCIPAL", "myUserName");
  env.Add(LookupContext.SECURITY_CREDENTIALS", "myPassword");
LookupContextFactory lcf = new LookupContextFactory();
ILookupContext searcher = lcf.CreateContext(
                           LookupContextFactory.TIBJMS_NAMING_CONT
                          EXT,
                          env);
```

  Use the ILookupContext.Lookup method to look up a connection factory, named ConFac, and destinations, named topic.sample and queue.sample, from the initial context:

```
ConnectionFactory factory =
      (ConnectionFactory) searcher.Lookup("ConFac");
Topic sampleTopic =
      (Topic)searcher.Lookup("topic.sample");
TIBCO.EMS.Queue sampleQueue =
      (TIBCO.EMS.Queue)searcher.Lookup("queue.sample");
```

Look Up Objects Using Full URL Names

Java clients can look up administered objects using full URL names. In this case, the `Context.URL_PKG_PREFIXES` property is used in place of the `Context.PROVIDER_URL` property.

For example:

```
Hashtable env = new Hashtable();
  env.put(Context.URL_PKG_PREFIXES,"com.tibco.tibjms.naming");
  env.put(Context.PROVIDER_URL,"tibjmsnaming://localhost:7222");
  env.put(Context.SECURITY_PRINCIPAL,"userName");
  env.put(Context.SECURITY_CREDENTIALS,"password");
jndiContext = new InitialContext(env);
```

When using full URL names, you can look up objects like the following example:

```
Topic sampleTopic = (javax.jms.Topic)jndiContext.lookup(
      "tibjmsnaming://emshost:7222/topic.sample");
Queue sampleQueue = (javax.jms.Queue)jndiContext.lookup(
      "tibjmsnaming://emshost:7222/queue.sample");
```

For further information on how to use full URL names, refer to the `tibjmsJNDIRead.java` example located in the *EMS_HOME*`/samples/java/JNDI` directory.

Perform Secure Lookups

TIBCO Enterprise Message Service client programs can perform secure JNDI lookups using the Transport Layer Security (TLS) protocol. To accomplish this, the client program must set TLS properties in the environment when the `InitialContext` is created. The TLS properties are similar to the TLS properties for the TIBCO Enterprise Message Service server.

See TLS Protocol for more information about using TLS in the TIBCO Enterprise Message Service server.

The following examples illustrate how to create an `InitialContext` that can be used to perform JNDI lookups using the TLS protocol.

- Java

In this example, the port number specified for the `Context.PROVIDER_URL` is set to the TLS listen port that was specified in the server configuration file `tibjsmd.conf`. The value for `TibjmsContext.SECURITY_PROTOCOL` is set to `ssl`. Finally, the value of `TibjmsContext.SSL_ENABLE_VERIFY_HOST` is set to "false" to turn off server authentication. Because of this, no trusted certificates need to be provided and the client will then not verify the server it is using for the JNDI lookup against the server's certificate.

```
Hashtable env = new Hashtable();
  env.put(Context.INITIAL_CONTEXT_FACTORY,
         "com.tibco.tibjms.naming.TibjmsInitialContextFactory");
  env.put(Context.PROVIDER_URL, tibjmsnaming://emshost:7223);
  env.put(Context.URL_PKG_PREFIXES, "com.tibco.tibjms.naming")
  env.put(TibjmsContext.SECURITY_PROTOCOL, "ssl");
  env.put(TibjmsContext.SSL_ENABLE_VERIFY_HOST,
         new Boolean("false"));
Context context = new InitialContext(env);
```

- C

  Create a `tibemsSSLParams` object and use the `tibemsSSLParams_SetIdentityFile` function to establish the client identity by means of a `pkcs12` file. Use the `tibemsLookupContext_CreateSSL` function to create a `tibemsLookupContext` object that uses a TLS connection for the initial context.

```
tibemsLookupContext*     context   = NULL;
tibemsConnection_Factory factory   = NULL;
tibemsSSLParams          sslParams = NULL;
tibems_status            status    = TIBEMS_OK;
sslParams = tibemsSSLParams_Create();
status = tibemsSSLParams_SetIdentityFile(
                                ssl_params,
                                "client_identity.p12",
                                TIBEMS_SSL_ENCODING_AUTO);
status = tibemsLookupContext_CreateSSL(
                                &context,
                                "tcp://localhost:7222",
                                "userName",
                                "password",
                                sslParams,
                                "pk_password");
```

- C#

  Create a `ILookupContext` object for the initial context over a TLS connection. The TLS

Store Info consists of a pkcs12 file that identifies the client and the client's password, which are stored in an `EMSSSLFileStoreInfo` object.

```
string ssl_identity = client_identity.p12;
string ssl_target_hostname = "server";
string ssl_password = "password";
EMSSSLFileStoreInfo StoreInfo = new EMSSSLFileStoreInfo();
  info.SetSSLClientIdentity(ssl_identity);
  info.SetSSLPassword(ssl_password.ToCharArray());
Hashtable env = new Hashtable();
  env.Add(LookupContext.PROVIDER_URL, "adc1.na.tibco.com:10636");
  env.Add(LookupContext.SECURITY_PRINCIPAL", "myUserName");
  env.Add(LookupContext.SECURITY_CREDENTIALS", "myPassword");
  env.Add(LookupContext.SECURITY_PROTOCOL, "ssl");
  env.Add(LookupContext.SSL_TARGET_HOST_NAME,
                       ssl_target_hostname);
  env.Add(LookupContext.SSL_STORE_TYPE,
                       EMSSSLStoreType.EMSSSL_STORE_TYPE_FILE);
  env.Add(LookupContext.SSL_STORE_INFO, StoreInfo);
```

Perform Fault-Tolerant Lookups

TIBCO Enterprise Message Service can perform fault-tolerant JNDI lookups. If the active server fails and the standby server becomes active, the JNDI provider automatically uses the new active server for JNDI lookups. You accomplish this by providing multiple URLs in the `Context.PROVIDER_URL` property when creating the `InitialContext`. Specify more than one URL separated by commas (,) in the property.

Example

The following illustrates setting up the `Context.PROVIDER_URL` property with the URLs of a primary EMS server on the machine named `emshost` and a secondary EMS server on the machine named `backuphost`.

```
env.put(Context.PROVIDER_URL,
"tibjmsnaming://emshost:7222,tibjmsnaming://backuphost:7222");
```

Assuming `emshost` starts out as active, if at any time it fails the JNDI provider automatically switches to the EMS server on the host `backuphost` for JNDI lookups. If `emshost` is repaired and restarted, it then becomes the standby EMS server.

Limitations of Fault-Tolerant JNDI Lookups

Fault-tolerant JNDI lookups do not occur in scenarios:

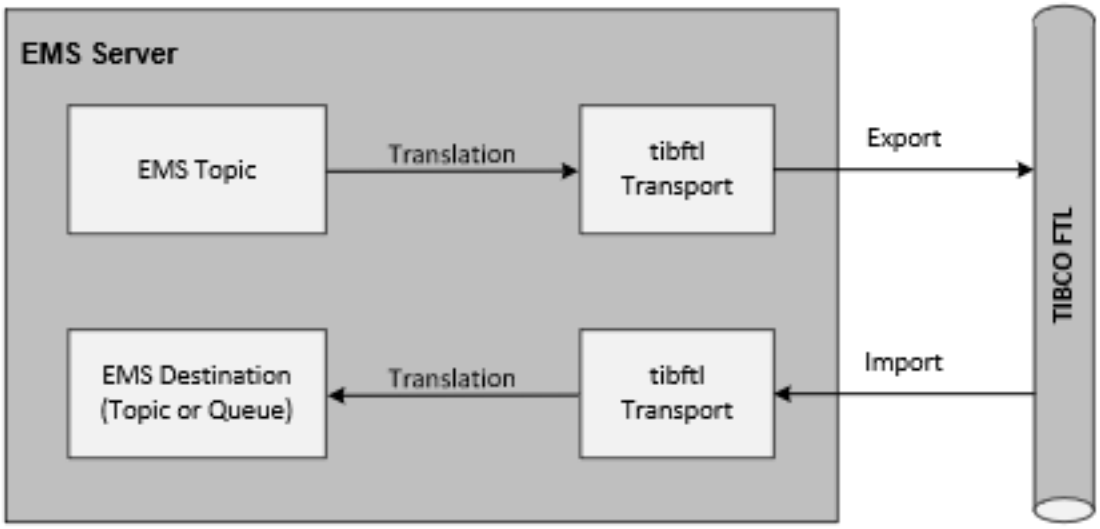
- When using full URL names in argument to the lookup method.
- When looking up an object that has been bound into a foreign naming/directory service such as LDAP.

Interoperation with TIBCO FTL

TIBCO Enterprise Message Service can exchange messages with supported versions of TIBCO FTL.

Scope

- EMS can import and export messages to TIBCO FTL through an EMS *topic*.
- EMS can import messages from TIBCO FTL to an EMS *queue* (but queues cannot export).



Warning: Do not configure EMS and FTL round-tripping. That is, do not send messages from EMS to FTL and then back to EMS, or the other way around.

Message Translation

EMS and TIBCO FTL use different formats for messages and their data.

When `tibemsd` imports or exports a messages, it translates the message and its data to the appropriate format; for details, see Message Translation.

Configuration

In classic EMS configuration, the `tibemsd` uses definitions and parameters in three configuration files to guide the exchange of messages with TIBCO FTL. In JSON-configured servers, all configuration options are in the same file.

Enabling

An EMS server is part of exactly one FTL realm, so all EMS transports for TIBCO FTL use the same TIBCO FTL realm. Thus, some parameters are shared for every EMS transport instance. These parameters are found in `tibemsd.conf`.

To enable EMS transports for TIBCO FTL, you must set these parameters in the configuration file `tibemsd.conf`:

- `tibftl_transports` globally enables or disables message exchange with TIBCO FTL. The default value is `disabled`. To use EMS transports for TIBCO FTL, you must explicitly set this parameter to `enabled`.
- `ftl_url` specifies the URL at which the EMS server is connected to the FTL server. For a TLS connection, this URL starts with `https://` rather than `http://`.
- `ftl_trustfile` specifies the trust file that is used to validate the FTL server on a TLS connection.
- `module_path` specifies the location of the TIBCO FTL shared library files. If the EMS server is configured to use FTL stores, the value of this parameter is ignored and the FTL shared library files corresponding to the hosting FTL server are used instead.

If connecting to a TIBCO FTL deployment that is configured for OAuth 2.0 authentication, the following parameters will also need to be set in `tibemsd.conf`:

- `ftl_oauth2_server_url` specifies the URL of the OAuth 2.0 authorization server from which access tokens will be obtained for authenticating with TIBCO FTL.
- `ftl_oauth2_client_id` specifies the OAuth 2.0 client ID to use when authenticating with the OAuth 2.0 authorization server to obtain access tokens.

- `ftl_oauth2_client_secret` specifies the OAuth 2.0 client secret to use when authenticating with the OAuth 2.0 authorization server to obtain access tokens.

Additional optional parameters can be used to further configure how the EMS server and FTL server interact. See TIBCO FTL Transport Parameters.

Transports

Transport definitions (in the configuration file `transports.conf`) specify the communication protocol between EMS and TIBCO FTL.

For more information, see Configure EMS Transports for TIBCO FTL.

Destinations

Destination definitions (in the configuration files `topics.conf` and `queues.conf`) can set the `import` and `export` properties to specify one or more EMS transport for TIBCO FTL.

- `import` instructs `tibemsd` to import messages that arrive on those transports from TIBCO FTL, and deliver them to the EMS destination. When a destination is configured to import a given `tibftl` transport and the optional `import_subscriber_name` transport property is not set, `tibemsd` creates a single default FTL durable for the transport with the following name: `<server name>:<transport name>`. For example, if `tibemsd.conf` has server = `EMS-SERVER` and `transports.conf` has a `[FTL1]` transport defined, the corresponding durable name comes as `EMS-SERVER:FTL1`. However, if `import_subscriber_name` is set, then `tibemsd` creates a FTL subscriber by that name instead of the aforementioned FTL durable.

  > **Note:** An FTL administrator can monitor the status of this default FTL durable and, if relevant, the FTL administrator can decide to configure it as static and alter its behavior.

- export instructs `tibemsd` to take messages that arrive on the EMS destination, and export them to TIBCO FTL using those EMS transports for TIBCO FTL. When a destination is configured to export a given `tibftl` transport, the EMS server creates a single FTL publisher for the transport.

For details, see Topics, and Queues.

Configure EMS Transports for TIBCO FTL

EMS transports mediate the flow of messages between TIBCO Enterprise Message Service and TIBCO FTL.

> **Note:** In TIBCO FTL, *transport* refers to the underlying mechanism that moves message data between FTL publishers and subscribers.
>
> In TIBCO Enterprise Message Service, a transport is a more narrowly defined concept, referring specifically to the connections between an EMS server and an external system.

The EMS server joins a TIBCO FTL realm as any other TIBCO FTL client would. EMS transport definitions (in the file `transports.conf`) configure the behavior of these connections.

All messages received from the transports for TIBCO FTL that are configured in the `transports.conf` file are processed in a single TIBCO FTL event queue.

After being dispatched from the TIBCO FTL event queue, all TIBCO FTL messages that are imported through an EMS transport are processed by the EMS server. The EMS server creates Jakarta Messaging message copies of the incoming TIBCO FTL messages and begins processing them as EMS messages. EMS transports for TIBCO FTL determine how the messages are converted to EMS messages.

If the EMS server cannot keep up with the rate of incoming TIBCO FTL messages, FTL could begin discarding messages before they have been successfully imported by EMS.

Requirements

To successfully deploy the EMS transport for TIBCO FTL, in the TIBCO FTL deployment, you must configure transports to be server-defined / of type Auto (see the TIBCO FTL documentation for more information on the server-defined 'Server' transport and on the 'Auto' transport type). Other types of transports are not supported.

EMS Transport for FTL Definitions

`transports.conf` contains zero or more transport definitions. Each definition begins with the name of a transport, surrounded by square brackets. Subsequent lines set the parameters of the transport.

Parameter	Description
`type`	Required. For all EMS transports for TIBCO FTL, the value must be `tibftl`.
TIBCO FTL Parameters The syntax and semantics of these parameters are identical to the corresponding parameters in TIBCO FTL clients. For full details, see the TIBCO FTL documentation set.	
`endpoint`	Optional. Specify a TIBCO FTL endpoint name. To define multiple transports that use the same TIBCO FTL endpoint, include the same `endpoint` name in each transport definition. If absent, the endpoint name defaults to the name of the EMS transport.
`import_subscriber_name`	Optional. The name of the FTL `subscriber` this EMS transport for FTL creates when it receives messages. If not set, then an EMS transport that receives FTL messages creates a default FTL `durable` instead. See Destinations for details on the FTL durable name.
`import_match_string`	Optional. Creates a content matcher object to filter messages. Specify content matchers using the syntax: `{"fieldname1":value1,...,"fieldnameN":valueN}` The following rules must be observed: • Field name and value declarations must conform to the match string syntax described in the TIBCO FTL documentation. • The `import_match_string` must be specified on a single line. No manual line breaks may be inserted. Spaces are not

Parameter	Description
	allowed. For example: `import_match_string = {"Item":"Book","Title":"Outliers","Stocked":true}`
`export_format`	Optional. Specifies a format name to be used when a message is created. If not provided, the EMS server passes NULL to the TIBCO FTL message create call, resulting in a dynamically formatted message.

`export_constant`	Optional. Defines fields that are always set to a constant value. Each line adds additional constants. For example:
	
	`export_constant = constant1,value1`
	`export_constant = constant2,value2`
	`export_constant = constant3,value3`

Example

These examples from `transports.conf` illustrate the syntax of EMS transport for FTL definitions.

```
[FTL1]
  type = tibftl
  endpoint = EP1
  import_subscriber_name = sub1
  import_match_string = {"f1":"foo","f2":true}
  export_format = format-1
  export_constant = constant1,value1
  export_constant = constant2,value2
  export_constant = constant3,value3

[FTL2]
  type = tibftl
```

Topics

Topics can both export and import messages. Accordingly, you can configure topic definitions (in the configuration file `topics.conf`) with `import` and `export` properties that specify one or more external transports:

import

`import` instructs `tibemsd` to import messages that arrive on those EMS transports from TIBCO FTL, and deliver them to the EMS destination. Each named `tibftl` transport can be named on only one EMS destination. That is, if the transport FTL01 is included on `import` property for destination `myTopics.Fiction`, it cannot also be added to the destination `myTopics.Nonfiction`.

> **Warning:** An EMS transport for TIBCO FTL may be specified as an import transport by only one destination. If the `topics.conf` configuration has a transport for TIBCO FTL included as an import transport by more than one destination, the server handles this as a configuration error.

export

`export` instructs `tibemsd` to take messages that arrive on the EMS destination, and export them to TIBCO FTL using the specified EMS transport for TIBCO FTL.

> **Note:** The EMS server never re-exports an imported message on the same topic.

(For general information about `topics.conf` syntax and semantics, see topics.conf. You can also configure topics using the administration tool command `addprop topic`.)

Example

For example, the following `tibemsadmin` commands configure the topic `myTopics.news` to `import` messages on the transports `FTL01` and `FTL02`, and to `export` messages on the transport `FTL02`.

```
addprop topic myTopics.news import="FTL01,FTL02"
addprop topic myTopics.news export="FTL02"
```

TIBCO FTL messages with subject `myTopics.news` arrive at `tibemsd` over the transports `FTL01` and `FTL02`. EMS clients can receive those messages by subscribing to `myTopics.news`.

EMS messages sent to `myTopics.news` are exported to TIBCO FTL over transport `FTL02`. TIBCO FTL clients of the corresponding daemons can receive those messages by subscribing to the endpoint associated with `myTopics.news` in the `FTL02` transport definition.

Import Only when Subscribers Exist

When a topic specifies `import` on a connected transport, `tibemsd` imports messages only when the topic has at least one subscriber.

For more information, see `import`.

Queues

Queues can `import` messages, but cannot `export` them.

Configuration

You can configure queue definitions (in the configuration file `queues.conf`) with the `import` property to specify one or more external transports.

`import` instructs `tibemsd` to import messages that arrive on those EMS transports from TIBCO FTL, and deliver them to the EMS destination.

(For general information about `queues.conf` syntax and semantics, see `queues.conf`. You can also configure queues using the administration tool command `addprop queue`.)

Example

For example, the following `tibemsadmin` command configures the queue `myQueue.in` to `import` messages on the EMS transports `FTL01` and `FTL02`.

```
addprop queue myQueue.in import="FTL01,FTL02"
```

TIBCO FTL messages with subject `myQueue.in` arrive at `tibemsd` over the transports `FTL01` and `FTL02`. EMS clients can receive those messages by subscribing to `myQueue.in`.

Import—Start and Stop

When a queue specifies `import` on a connected transport, `tibemsd` immediately begins importing messages to the queue, even when no receivers exist for the queue.

For static queues (configured by an administrator) `tibemsd` continues importing until you explicitly delete the queue. When the queue is deleted, the transport no longer imports messages.

Message Translation

The following topics describe how a message is translated by the EMS server when either imported from or exported to FTL.

Jakarta Messaging Header Fields

EMS supports the predefined JMS header fields.

For more information, see Jakarta Messaging Message Header Fields.

> **Note:** The `JMSTimestamp` Jakarta Messaging header field is a special case.
>
> The Jakarta Messaging header `JMSTimestamp` corresponds to the time when the message was created. If this header field is not present when the `tibemsd` receives the message, it sets `theJMSTimestamp` to the current time.

TIBCO FTL messages do not have destinations or subjects, or a mandatory set of predefined header fields. Instead, message fields and their values are set for individual messages.

If the `export_headers` is defined as `true` in the common EMS transport properties, the EMS server converts the Jakarta Messaging header fields and their values to TIBCO FTL fields and values and adds them to the outgoing message. This allows TIBCO FTL to use content matchers on the fields.

If the `export_headers` property is `false`, then the Jakarta Messaging header fields and their values are *not* included in the exported TIBCO FTL message. This includes the destination name. That is, if `export_headers = false` for the transport, then the message exported to TIBCO FTL will not contain the destination name.

When converting the Jakarta Messaging header fields to TIBCO FTL message fields, header fields are given the prefix `_emshdr:`. For example, the JMSDeliveryMode header field is assigned the field name `_emshdr:JMSDeliveryMode` when inserted into the TIBCO FTL message.

The following table presents the mapping of Jakarta Messaging header fields to TIBCO FTL message field name and types (that is, the name and type of the corresponding field in the exported message).

Jakarta Messaging Header Name	TIBCO FTL Field Name	FTL Field Type
`JMSDestination`	`_emsHdr:JMSDestination`	char*
`JMSDeliveryMode`	`_emsHdr:JMSDeliveryMode`	tibint64_t
`JMSPriority`	`_emsHdr:JMSPriority`	tibint64_t
`JMSMessageID`	`_emsHdr:JMSMessageID`	char*
`JMSTimestamp`	`_emsHdr:JMSTimestamp`	tibint64_t
`JMSCorrelationID`	`_emsHdr:JMSCorrelationID`	char*
`JMSType`	`_emsHdr:JMSType`	char*
`JMSDeliveryTime`	`_emsHdr:JMSDeliveryTime`	tibint64_t
`JMSExpiration`	`_emsHdr:JMSExpiration`	tibint64_t
`JMSRedelivered`	`_emsHdr:JMSRedelivered`	tibint64_t
JMSReplyTo	`_emsHdr:`JMSReplyTo	char*

Jakarta Messaging Property Fields

EMS supports the Jakarta Messaging property fields described in EMS Message Properties.

Import

When importing a TIBCO FTL message to an EMS message, `tibemsd` sets these EMS properties:

- `JMS_TIBCO_IMPORTED` gets the value `true`, to indicate that the message did not originate from an EMS client.
- `JMS_TIBCO_MSG_EXT` gets the value `true`, to indicate that the message *might* contain submessage fields or array fields.

Export

TIBCO FTL messages do not have destinations or subjects, or a mandatory set of predefined header fields. Instead, message fields and their values are set for individual messages.

If `export_properties` is defined as `true` in the common EMS transport properties, the EMS server converts the Jakarta Messaging properties and their values to TIBCO FTL fields and values and adds them to the outgoing message. This allows TIBCO FTL to use content matchers on the fields.

When converting the Jakarta Messaging properties to TIBCO FTL message fields, the property fields are given the prefix `_emsprop:`. For example the `JMS_TIBCO_SENDER` property would become the `_emsprop:JMS_TIBCO_SENDER field`.

The `tibemsd` server ignores any Jakarta Messaging property fields that are not set, or are set to null—it omits them from the exported message.

You can instruct `tibemsd` to exclude the properties fields from the exported message by setting the transport property `export_properties = false`.

Message Body

`tibemsd` can export messages with most Jakarta Messaging message body types to TIBCO FTL. However, Object messages and Stream messages cannot be exported. They are discarded with a warning.

`tibemsd` can import messages with any message format from TIBCO FTL.

For information about Jakarta Messaging body types, see Jakarta Messaging Message Bodies. For information about the structure of messages, see Jakarta Messaging Message Structure.

Import

When importing a TIBCO FTL message, `tibemsd` translates it to an EMS message body type based on the TIBCO FTL message format.

TIBCO FTL Message Format	EMS Message Type
FTL Message	Map Message
Built-in Opaque Format	Map Message with a bytes field, `_data`.
Keyed Opaque Format	Map Message with two fields: • `_key` (char*) • `_data` (bytes)

Export

When exporting an EMS message, `tibemsd` translates it to a TIBCO FTL message with the following structure:

- When `export_headers` is enabled on the EMS transport, Jakarta Messaging header fields are converted to TIBCO FTL message fields. See Jakarta Messaging Header Fields. When the transport parameter `export_headers` is `false`, these fields are omitted.
- When `export_properties` is enabled on the EMS transport, Jakarta Messaging

property fields are converted to TIBCO FTL message fields. See Jakarta Messaging Property Fields. When the transport parameter `export_properties` is `false`, these fields are omitted.

- When translating the data fields of an EMS message, the results depend on the Jakarta Messaging body type.

Jakarta Messaging Body Type	Export Translation
`MapMessage`	An FTL message of the format specified. If no format was specified, it is a dynamically formatted FTL message.
`BytesMessage`	An FTL message with one opaque field with the key of `_data`.
`TextMessage`	FTL message with a `_text` field.
`Message`	Empty FTL message.
`ObjectMessage`	Not converted. Messages with this Jakarta Messaging body type cannot be exported to TIBCO FTL.
`StreamMessage`	Not converted. Messages with this Jakarta Messaging body type cannot be exported to TIBCO FTL.

Message Fields

When `tibemsd` converts messages, it converts fields individually, based on field type. Some field types are equivalent between EMS and TIBCO FTL, while converting others may result in some information loss of data type, or require additional formatting.

The mapping of equivalent fields is bidirectional. These field types are equivalent in EMS and TIBCO FTL, and no additional formatting is required during conversion:

EMS Field Type	TIBCO FTL Field Type
tibems_long	tibint64_t
tibems_long array	tibint64_t array
tibems_double	tibdouble_t
tibems_double array	tibdouble_t array
char*	char*
MapMsg	Message
bytes	Opaque

Import

Not all TIBCO FTL field types are supported by EMS. When `tibemsd` imports a TIBCO FTL message, these fields are converted into EMS sub-messages as shown below.

TIBCO FTL Field Type	EMS Field Type	Map Message Field Name
Message Array	Sub-message with message fields named 0, 1, and so on.	_ftlMsgArray:fieldname
char* array	Sub-message with message fields named 0, 1, and so on.	_ftlStringArray:fieldname
tibDateTime	Sub-message with two fields: • s – long, representing seconds. • n – long, representing nanoseconds.	_ftlDateTime:fieldname
tibDateTime array	Sub-message containing tibDateTime equivalent sub-	_ftlDateTimeArray:fieldname

TIBCO FTL Field Type	EMS Field Type	Map Message Field Name
	messages. Each submessage contains two fields: • `s` – `long`, representing seconds. • `n` – `long`, representing nanoseconds.	
`tibInbox`	Discarded during conversion	N/A

Export

When exporting an EMS message, `tibemsd` translates it to a TIBCO FTL message. Not all field types that are supported by EMS map to TIBCO FTL. When `tibemsd` converts these fields, some information about data size is lost. The EMS fields are converted to TIBCO FTL fields as shown here:

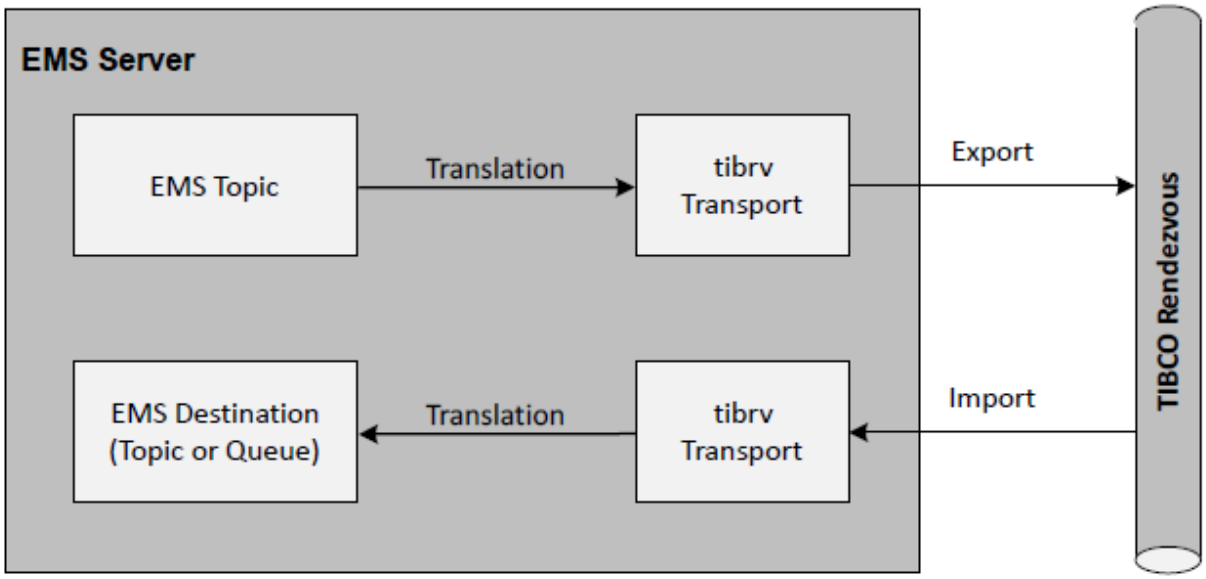
EMS Field Type	TIBCO FTL Field Type
`tibems_wchar`	`tibint64_t`
`tibems_byte`	`tibint64_t`
`tibems_short`	`tibint64_t`
`tibems_short_array`	`tibint64_t array`
`tibems_int`	`tibint64_t`
`tibems_int_array`	`tibint64_t array`
`tibems_float`	`tibdouble_t`
`tibems_float_array`	`tibdouble_t array`

Interoperation with TIBCO Rendezvous

TIBCO Enterprise Message Service can exchange messages with supported versions of TIBCO Rendezvous.

Scope

- EMS can import and export messages to an external system through an EMS *topic*.
- EMS can import messages from an external system to an EMS *queue* (but queues cannot export).



Message Translation

EMS and Rendezvous use different formats for messages and their data. When `tibemsd` imports or exports a messages, it translates the message and its data to the appropriate format.

For more information, see Message Translation.

Configuration

`tibemsd` uses definitions and parameters in four configuration files to guide the exchange of messages with Rendezvous.

Enabling

The parameter `tibrv_transports` (in the configuration file `tibemsd.conf`) globally enables or disables message exchange with Rendezvous. The default value is `disabled`. To use these transports, you must explicitly set this parameter to enabled.

The parameter `module_path` (in the configuration file `tibemsd.conf`) specifies the location of the Rendezvous shared library files.

Transports

Transport definitions (in the configuration file `transports.conf`) specify the communication protocol between EMS and the external system.

For more information, see Configure EMS Transports for Rendezvous.

Destinations

Destination definitions (in the configuration files `topics.conf` and `queues.conf`) can set the `import` and `export` properties to specify one or more transports:

- `import` instructs `tibemsd` to import messages that arrive on those transports from Rendezvous, and deliver them to the EMS destination.
- `export` instructs `tibemsd` to take messages that arrive on the EMS destination, and export them to Rendezvous via those transports.

For details, see Topics, and Queues.

RVCM Listeners

When exporting messages on a transport configured for certified message delivery, you can pre-register RVCM listeners in the file `tibrvcm.conf` .

For details, see tibrvcm.conf, and Certified Messages

Configure EMS Transports for Rendezvous

Transports mediate the flow of messages between EMS and TIBCO Rendezvous.

`tibemsd` connects to Rendezvous daemons in the same way as any other Rendezvous client would. Transport definitions (in the file `transports.conf`) configure the behavior of these connections. You must properly configure these transports.

Additionally, you must configure the parameter `module_path` (in the configuration file `tibemsd.conf`) to specify the location of the Rendezvous shared library files.

How Rendezvous Messages are Imported

The EMS server connects to the Rendezvous daemon as any other Rendezvous client would. Messages received from the Rendezvous daemon are stored in Rendezvous queues, then are dispatched to callbacks. The EMS server creates Jakarta Messaging message copies of the Rendezvous messages, and begins processing them as EMS messages. Transports determine how messages are imported.

Rendezvous messages that are imported through a transport are held in queues specific to that transport. Each transports is associated with a different Rendezvous queue, which holds as many Rendezvous messages as necessary. The number of pending messages in the queue will grow if the rate of incoming Rendezvous messages is greater than the rate at which the EMS server is able to process the corresponding EMS messages.

Depending on the import delivery mode defined for the transport, the EMS messages will be persisted on disk, which increases the likelihood of backlog in the Rendezvous queues, and which in turn results in a EMS process memory growth. This memory growth is not accounted for in any of the EMS server statistics.

Queue Limit Policies

In order to limit the number of pending messages in Rendezvous queues, a transport property allows you to set a queue limit policy, as you would for TIBCO Rendezvous client applications.

When the queue limit for the transport is reached, the Rendezvous library discards a set number of messages. The default policy is `TIBRVQUEUE_DISCARD_NONE`, which means that no message is ever discarded. Setting `TIBRVQUEUE_DISCARD_FIRST` or `TIBRVQUEUE_DISCARD_LAST` allows you to specify the maximum number of Rendezvous messages that can be pending in the queue before the discard policy that you have selected is applied. When the limit is reached, the number of messages discarded is based on the discard amount value.

When the limit is reached, Rendezvous messages are discarded, and so are not imported as EMS messages, regardless of the EMS import delivery mode. As stated above, a Rendezvous message becomes a EMS message only after it has been dispatched from the Rendezvous queue. If a queue limit is exceeded, reliable Rendezvous messages are lost.

Rendezvous certified messages are not lost, but the message flow is interrupted. The redelivery of the missed messages is handled automatically by the Rendezvous libraries, and can not be controlled by the EMS server.

Reaching a queue limit also generates a Rendezvous advisory that is logged (see RVADV log and console trace in the TIBCO Rendezvous documentation), indicating which transport reached its queue limit. This advisory goes into an independent, non limited, Rendezvous queue. If lots of advisories are generated, this internal queue may also grow, signaling that the limit policy is not appropriate for your environment.

Take care when setting a queue limit policy. In a controlled environment where the risk of Rendezvous producers overwhelming the EMS server is low, there is no need to set a queue limit policy.

Transport Definitions

`transports.conf` contains zero or more transport definitions. Each definition begins with the name of a transport, surrounded by square brackets. Subsequent lines set the parameters of the transport.

Parameter	Description
`type`	Required. For Rendezvous transports, the value must be either `tibrv` or `tibrvcm`.
Rendezvous Parameters Use these properties for either `tibrv` or `tibrvcm` transports. The syntax and semantics of these parameters are identical to the corresponding parameters in Rendezvous clients. For full details, see the Rendezvous documentation set.	
`service`	When absent, the default value is 7500.
`network`	When absent, the default value is the host computer's primary network.
`daemon`	When absent, the default value is an `rvd` process on the local host computer. When transporting messages between EMS and Rendezvous, the rvd process must be configured to run on the same host as the EMS daemon (`tibemsd`). To connect to a non-default daemon, supply *protocol*:*hostname*:*port*. You may omit any of the three parts. The default protocol is `tcp`. The default *hostname* is the local host computer. The default *port* is 7500.
Rendezvous Certified Messaging (RVCM) Parameters Use these properties only for `tibrvcm` transports. The syntax and semantics of these parameters are identical to the corresponding parameters in Rendezvous CM clients. For full details, see the Rendezvous documentation set.	
`cm_name`	The name of the correspondent RVCM listener transport.
`rv_tport`	Required. Each RVCM transport depends in turn upon an ordinary Rendezvous transport. Set this

Parameter	Description
	parameter to the name of a Rendezvous transport (type `tibrv`) defined in the EMS configuration file `transports.conf`.
`ledger_file`	Name for file-based ledger.
`sync_ledger`	`true` or `false`. If `true`, operations that update the ledger do not return until changes are written to the storage medium.
`request_old`	`true` or `false`. If `true`, this transport server requests unacknowledged messages sent from other RVCM senders while this transport was unavailable.
`default_ttl`	This parameter sets default CM time limit (in seconds) for all CM messages exported on this transport.
`explicit_config_only`	`true` or `false`. If `true`, `tibemsd` allows RVCM listeners to register for certified delivery only if they are configured in advance with the EMS server (either in `tibrvcm.conf` or using the `create rvcmlistener` command). That is, `tibemsd` ignores registration requests from non-configured listeners. If `false` (the default), `tibemsd` allows any RVCM listener to register.
EMS Parameters Use these properties for either `tibrv` or `tibrvcm` transports.	

`topic_import_dm`
 	
`queue_import_dm`
	EMS sending clients can set the `JMSDeliveryMode` header field for each message. However, Rendezvous clients cannot set this header. Instead, these two parameters determine the delivery modes for all topic messages and queue messages that `tibemsd` imports on this transport.

Parameter	Description
	`TIBEMS_PERSISTENT | TIBEMS_NON_PERSISTENT | TIBEMS_RELIABLE` When absent, the default is `TIBEMS_NON_PERSISTENT`.
`export_headers`	When `true`, `tibemsd` includes Jakarta Messaging header fields in exported messages. When `false`, `tibemsd` suppresses Jakarta Messaging header fields in exported messages. When absent, the default value is `true`.
`export_properties`	When `true`, `tibemsd` includes Jakarta Messaging properties in exported messages. When `false`, `tibemsd` suppresses Jakarta Messaging properties in exported messages. When absent, the default value is `true`.
`rv_queue_policy`	Set the queue limit policy for the Rendezvous queue used by the transport to hold incoming Rendezvous messages. This parameter has three parts: *policy*`:`*max_msgs*`:`*qty_discard* where *policy* is one of the queue limit policies described below, *max_msgs* is the maximum number of messages permitted in the queue before discard, and *qty_discard* is the number of messages that the EMS server discards when *max_msgs* is reached. The queue limit policies are: • `TIBRVQUEUE_DISCARD_NONE` — do not discard messages. Use this policy when the queue has no limit on the number of messages it

Parameter	Description
	can contain. • TIBRVQUEUE_DISCARD_FIRST — discard the first message in the queue. The first message in the queue is the oldest message, which if not discarded would be the next message dispatched from the queue. • TIBRVQUEUE_DISCARD_LAST — discard the last message in the queue. The last message is the most recent message received into the queue. For example, the following would cause the Rendezvous library to discard the 100 oldest messages in the queue when the total number of messages in the queue reached 10,000: `rv_queue_policy=TIBRVQUEUE_DISCARD_FIRST:10000:100` If the rv_queue_policy is not present, the default queue limit policy is TIBRVQUEUE_DISCARD_NONE.
temp_destination_timeout	Specifies the amount of time the server is to keep the temporary destination (created for the RV inbox) after its last use of the destination. This is useful for a multi-server configuration. For example, in a configuration in which rv-requester -> serverA -> serverB -> rv-responder, setting temp_destination_timeout=60 on serverB specifies that serverB is to hold the temporary destination for 60 seconds.

Example

These examples from transports.conf illustrate the syntax of transport definitions.

```
[RV01]
   type = tibrv
```

```
   topic_import_dm = TIBEMS_RELIABLE
   queue_import_dm = TIBEMS_PERSISTENT
   service = 7780
   network = lan0
   daemon = tcp:host5:7885

[RV02]
   type = tibrv
   service = 7890
   network = lan0
   daemon = tcp:host5:7995
   temp_destination_timeout = 60

[RVCM01]
   type = tibrvcm
   export_headers = true
   export_properties = true
   rv_tport = RV02
   cm_name = RVCMTrans1
   ledger_file = ledgerFile.store
   sync_ledger = true
   request_old = true
   default_ttl = 600
```

In the following two examples, `RVCM03` is an RVCM transport which does not define a queue limit policy, but references the RV transport `RV03`, which *does* have a queue limit policy. If Rendezvous messages are published to a subject that in EMS has the destination property `import=RVCM03`, no Rendezvous message will ever be discarded because each transport uses its own queue. Only messages that are imported directly through the `RV03` transport will potentially be discarded, should the queue limit of 10000 messages be reached.

```
[RV03]
   type = tibrv
   service = 7890
   network = lan0
   daemon = tcp:host5:7995
   rv_queue_policy = TIBRVQUEUE_DISCARD_LAST:10000:100

[RVCM03]
   type = tibrvcm
   rv_tport = RV03
   cm_name = RVCMTrans2
   ledger_file = ledgerFile2.store
   sync_ledger = true
```

```
request_old = true
default_ttl = 600
```

Topics

Topics can both export and import messages. Accordingly, you can configure topic definitions (in the configuration file `topics.conf`) with `import` and `export` properties that specify one or more external transports:

import

`import` instructs `tibemsd` to import messages that arrive on those transports from Rendezvous, and deliver them to the EMS destination.

export

`export` instructs `tibemsd` to take messages that arrive on the EMS destination, and export them to Rendezvous via those transports.

> **Note:** The EMS server never re-exports an imported message on the same topic.

(For general information about `topics.conf` syntax and semantics, see topics.conf. You can also configure topics using the administration tool command `addprop topic`.)

Example

For example, the following `tibemsadmin` commands configure the topic `myTopics.news` to import messages on the transports `RV01` and `RV02`, and to export messages on the transport `RV02`.

```
addprop topic myTopics.news import="RV01,RV02"
addprop topic myTopics.news export="RV02"
```

Rendezvous messages with subject `myTopics.news` arrive at `tibemsd` over the transports `RV01` and `RV02`. EMS clients can receive those messages by subscribing to `myTopics.news`.

EMS messages sent to `myTopics.news` are exported to Rendezvous over transport `RV02`. Rendezvous clients of the corresponding daemons can receive those messages by subscribing to `myTopics.news`.

Import Only when Subscribers Exist

When a topic specifies `import` on a connected transport, `tibemsd` imports messages only when the topic has has registered subscribers.

Wildcards

Wildcards in the `import` property obey EMS syntax and semantics (which is identical to Rendezvous syntax and semantics).

For more information, see Destination Name Syntax.

Certified Messages

You can import and `export` TIBCO Rendezvous certified messages (`tibrvcm` transport) to EMS topics. Rendezvous certified transports guarantee message delivery.

RVCM Ledger

`tibrvcm` transports can store information about subjects in a ledger file. You can review the ledger file using an administration tool command; see `show rvcmtransportledger`.

For more information about ledger files, see TIBCO Rendezvous documentation.

Subject Collisions

Subscribers to destinations that import from RVCM transports are subject to the same restrictions that direct RVCM listeners. These restrictions are described in the TIBCO Rendezvous documentation, and include subject collisions.

When importing messages from RV, the EMS server creates RVCM listeners using a single name for each transport. This can result in subject collisions if the corresponding EMS subscribers have overlapping topics.

Queues

Queues can `import` messages, but cannot `export` them.

See `import` and `export` for more information.

Configuration

You can configure queue definitions (in the configuration file `queues.conf`) with the `import` property that specify one or more external transports.

`import` instructs `tibemsd` to import messages that arrive on those transports from Rendezvous, and deliver them to the EMS destination.

(For general information about `queues.conf` syntax and semantics, see queues.conf. You can also configure queues using the administration tool command `addprop queue`.)

Example

The following `tibemsadmin` command configures the queue `myQueue.in` to `import` messages on the transports `RV01` and `RV02`.

```
addprop queue myQueue.in import="RV01,RV02"
```

Rendezvous messages with subject `myQueue.in` arrive at `tibemsd` over the transports `RV01` and `RV02`. EMS clients can receive those messages by subscribing to `myQueue.in`.

Import—Start and Stop

When a queue specifies `import` on a connected transport, `tibemsd` immediately begins importing messages to the queue, even when no receivers exist for the queue.

For static queues (configured by an administrator) `tibemsd` continues importing until you explicitly delete the queue.

Wildcards

Wildcards in the `import` property obey EMS syntax and semantics (not Rendezvous syntax and semantics).

For more information, see Destination Name Syntax.

EMS clients cannot subscribe to wildcard queues—however, you can define wildcards queues in the EMS server for the purpose of property inheritance. That is, you can configure a static queue named `foo.*` and set properties on it, so that child queues named `foo.bar` and `foo.baz` will both inherit those properties.

If you define a queue that imports `foo.*`, `tibemsd` begins importing all matching messages from Rendezvous. As messages arrive, `tibemsd` creates dynamic child queues (for example, `foo.bar` and `foo.baz`) and delivers the messages to them. Notices that `tibemsd` delivers messages to these dynamic child queues even when no consumers exist to drain them.

Import Issues

This section presents issues associated with importing messages to EMS from Rendezvous—whether on a topic or a queue.

Field Identifiers

When importing and translating Rendezvous messages, `tibemsd` is only able to process standard message field types that are identified by name in the Rendezvous program application. Custom fields and fields identified using a field identifier cannot be imported to EMS.

JMSDestination

When `tibemsd` imports and translates a Rendezvous message, it sets the `JMSDestination` field of the EMS message to the value of the Rendezvous subject.

Therefore, imported destination names must be unique. When a topic and a queue share the same name, at most one of them may set the `import` property. For example, if a topic `foo.bar` and a queue `foo.bar` are both defined, only one may specify the `import` property.

See `JMSDestination` for more information.

JMSReplyTo

When `tibemsd` imports and translates a Rendezvous message, it sets the `JMSReplyTo` field of the EMS message to the value of the Rendezvous reply subject, so that EMS clients can reply to the message.

Usually this value represents a Rendezvous subject. You must explicitly configure `tibemsd` to create a topic with a corresponding name, which exports messages to Rendezvous.

See `JMSReplyTo` for more information.

JMSExpiration

When `tibemsd` imports and translates a Rendezvous certified message, it sets the `JMSExpiration` field of the EMS message to the time limit of the certified message.

See `JMSExpiration` for more information.

If the message time limit is exceeded, the sender program no longer certifies delivery.

Note that if the `expiration` property is set for a destination, it will override the `JMSExpiration` value set by the message producer.

Guaranteed Delivery

Note: For full end-to-end certified delivery from Rendezvous to EMS, all three of these conditions must be true:

- Rendezvous senders must send labeled messages on RVCM transports. See the *TIBCO Rendezvous Concepts* manual for more information.
- The transport definition must set `topic_import_dm` or `queue_import_dm` (as appropriate) to `TIBEMS_PERSISTENT`.
- Either a durable queue or a subscriber for the EMS topic must exist.

Export Issues

This section presents issues associated with exporting messages from EMS to Rendezvous.

JMSReplyTo

Topics

Consider an EMS message in which the field `JMSReplyTo` contains a topic. When exporting such a message to Rendezvous, you must explicitly configure `tibemsd` to import replies from Rendezvous to that reply topic.

Temporary Topics

Consider an EMS message in which the field `JMSReplyTo` contains a temporary topic. When `tibemsd` exports such a message to Rendezvous, it automatically arranges to import replies to that temporary topic from Rendezvous; you do not need to configure it explicitly.

Certified Messages

RVCM Registration

When an RVCM listener receives its first labeled message, it registers to receive subsequent messages as certified messages. Until the registration is complete, it receives labeled messages as reliable messages. When exporting messages on a `tibrvcm` transport, we recommend either of two actions to ensure certified delivery for all exported messages:

- Create the RVCM listener before sending any messages from EMS clients.
- Pre-register an RVCM listener, either with the administration tool (see `create rvcmlistener`), or in the configuration file `tibrvcm.conf` (see `tibrvcm.conf`).

Guaranteed Delivery

For full end-to-end certified delivery to Rendezvous from EMS, the following condition must be true:

- EMS senders must send persistent messages.

Message Translation

The following topics describe how a message is translated by the EMS server when either imported from or exported to Rendezvous.

Jakarta Messaging Header Fields

EMS supports the predefined Jakarta Messaging header fields described in Jakarta Messaging Message Header Fields.

Special Cases

The following header fields are special cases:

- Jakarta Messaging header `JMSDestination` corresponds to Rendezvous subject.
- Jakarta Messaging header `JMSReplyTo` corresponds to Rendezvous reply subject.
- Jakarta Messaging header `JMSExpiration` corresponds to the time limit of the Rendezvous certified message.
- Jakarta Messaging header `JMSTimestamp` corresponds to the time when the message was created. If this header field is not present, when the `tibemsd` receives the message it sets the `JMSTimestamp` to the current time.

Import

When importing a Rendezvous message to an EMS message, `tibemsd` does not set any Jakarta Messaging header fields, except for the special cases noted above.

Export

When exporting an EMS message to a Rendezvous message, `tibemsd` groups all the Jakarta Messaging header fields into a single submessage within the Rendezvous message. The field `JMSHeaders` contains that submessage. Fields of the submessage map the names of Jakarta Messaging header fields to their values.

The `tibemsd` server ignores any Jakarta Messaging header fields that are not set, or are set to null—it omits them from the exported message.

You can instruct `tibemsd` to suppress the entire header submessage in the exported message by setting the transport property `export_headers = false`.

The following table shows the mapping of Jakarta Messaging header fields to Rendezvous data types (that is, the type of the corresponding field in the exported message).

Jakarta Messaging Header Name	Rendezvous Type
`JMSDeliveryMode`	`TIBRVMSG_U8`
`JMSDeliveryTime`	`TIBRVMSG_U64`
`JMSPriority`	`TIBRVMSG_U8`

Jakarta Messaging Header Name	Rendezvous Type
JMSTimestamp	TIBRVMSG_U64
JMSExpiration	TIBRVMSG_U64
JMSType	TIBRVMSG_STRING
JMSMessageID	TIBRVMSG_STRING
JMSCorrelationID	TIBRVMSG_STRING
JMSRedelivered	TIBRVMSG_BOOL
JMSDestination	send subject in TIBCO Rendezvous
JMSReplyTo	reply subject in TIBCO Rendezvous

Jakarta Messaging Property Fields

Import

When importing a Rendezvous message to an EMS message, `tibemsd` sets these EMS properties:

- `JMS_TIBCO_IMPORTED` gets the value `true`, to indicate that the message did not originate from an EMS client.
- `JMS_TIBCO_MSG_EXT` gets the value `true`, to indicate that the message *might* contain submessage fields or array fields.

Import RVCM

In addition to the two fields described above, when `tibemsd` imports a certified message on a `tibrvcm` transport, it can also set these properties (if the corresponding information is set in the Rendezvous message).

Property	Description
JMS_TIBCO_CM_PUBLISHER	A string value indicating the correspondent name of the TIBCO Rendezvous CM transport that sent the message (that is, the sender name).
JMS_TIBCO_CM_SEQUENCE	A long value indicating the CM sequence number of an RVCM message imported from TIBCO Rendezvous.

Export

When exporting an EMS message to a Rendezvous message, `tibemsd` groups all the Jakarta Messaging property fields into a single submessage within the Rendezvous message. The field `JMSProperties` contains that submessage. Fields of the submessage map the names of Jakarta Messaging property fields to their values.

The `tibemsd` server ignores any Jakarta Messaging property fields that are not set, or are set to null—it omits them from the exported message.

You can instruct `tibemsd` to suppress the entire properties submessage in the exported message by setting the transport property `export_properties = false`.

Message Body

`tibemsd` can export messages with any Jakarta Messaging message body type to TIBCO Rendezvous. Conversely, `tibemsd` can import messages with any message type from TIBCO Rendezvous.

For information about Jakarta Messaging body types, see Jakarta Messaging Message Bodies.

For information about the structure of messages, see Jakarta Messaging Message Structure.

Import

When importing a Rendezvous message, `tibemsd` translates it to an EMS message body type based on the presence of the field as seen in the following table.

Rendezvous Field	EMS Body Type
`JMSBytes`	`JMSBytesMessage`
`JMSObject`	`JMSObjectMessage`
`JMSStream`	`JMSStreamMessage`
`JMSText`	`JMSTextMessage`
None of these fields are present.	`JMSMapMessage`

Note: The field names `DATA` and `_data_` are reserved. We strongly discourage you from using these field names in EMS and Rendezvous applications, and especially when these two message transport mechanisms interoperate.

Note: Only standard Rendezvous fields identified by name can be imported into EMS. Custom fields and fields identified in the Rendezvous application by field identifiers cannot be imported.

Export

When exporting an EMS message, tibemsd translates it to a Rendezvous message with the following structure.

- The field `JMSHeaders` contains a submessage; see Jakarta Messaging Header Fields. When the transport parameter `export_headers` is `false`, this field is omitted.
- The field `JMSProperties` contains a submessage; see Jakarta Messaging Property Fields. When the transport parameter `export_properties` is `false`, this field is omitted.
- When translating the data fields of an EMS message, the results depend on the

Jakarta Messaging body type. The following table specifies the mapping.

Jakarta Messaging Body Type	Export Translation
`BytesMessage`	The message data translates to a byte array that contains the bytes of the original EMS message. The field `JMSBytes` receives this data. It has type `TIBRVMSG_OPAQUE`.
`TextMessage`	The message data translates to a UTF-8 string corresponding to the text of the original EMS message. The field `JMSText` receives this data. It has type `TIBRVMSG_STRING`.
`ObjectMessage`	The message data translates to a byte array containing the serialized Java object. The field `JMSObject` receives this data. It has type `TIBRVMSG_OPAQUE`.
`StreamMessage`	The message data translates to a byte array that encodes the objects in the original EMS message. The field `JMSStream` receives this data. It has type `TIBRVMSG_OPAQUE`.
`MapMessage`	The message data fields map directly to top-level fields in the Rendezvous message. The fields retain the same names as in the original EMS message. See also, EMS Extensions to Jakarta Messaging Messages.

Data Types

The mapping between EMS datatypes and Rendezvous datatypes is bidirectional, except for the Rendezvous types that have no corresponding EMS type (for these types the mapping is marked as unidirectional in the middle column).

EMS	Map	Rendezvous
Boolean		TIBRVMSG_BOOL
Byte		TIBRVMSG_I8
Short	<—	TIBRVMSG_U8
Short		TIBRVMSG_I16
Integer	<—	TIBRVMSG_U16
Integer		TIBRVMSG_I32
Long	<—	TIBRVMSG_U32
Long		TIBRVMSG_I64
Long	<—	TIBRVMSG_U64
Float		TIBRVMSG_F32
Double		TIBRVMSG_F64
Short	<—	TIBRVMSG_IPPORT16
Integer	<—	TIBRVMSG_IPADDR32
MapMessage		TIBRVMSG_MSG
Long	<—	TIBRVMSG_DATETIME
byte[]		TIBRVMSG_OPAQUE
java.lang.String		TIBRVMSG_STRING
byte[]	<—	TIBRVMSG_XML
byte[]	<—	TIBRVMSG_I8ARRAY

EMS	Map	Rendezvous
short[]	<—	TIBRVMSG_U8ARRAY
short[]		TIBRVMSG_I16ARRAY
int[]	<—	TIBRVMSG_U16ARRAY
int[]		TIBRVMSG_I32ARRAY
long[]	<—	TIBRVMSG_U32ARRAY
long[]		TIBRVMSG_I64ARRAY
long[]	<—	TIBRVMSG_U64ARRAY
float[]		TIBRVMSG_F32ARRAY
double[]		TIBRVMSG_F64ARRAY

Pure Java Rendezvous Programs

TIBCO Enterprise Message Service is shipped with the `tibrvjms.jar` file that you can include in your TIBCO Rendezvous applications. This JAR file includes the implementation of the `com.tibco.tibrv.TibrvJMSTransport` class. This class extends the `com.tibco.tibrv.TibrvNetTransport` class and allows your pure Java Rendezvous programs to communicate directly with the EMS server instead of through `rva`.

the application must include `tibrvjms.jar` and EITHER tibrvjweb.jar OR tibrvj.jar, but CANNOT include tibrvnative.jar

To use the `TibrvJMSTransport` class, your application must include `tibrvjms.jar` (included with EMS) and either `tibrvjweb.jar` or `tibrv.jar` (included with TIBCO Rendezvous). Your application *cannot* include `tibrvnative.jar`.

Note: You can use `TibrvJMSTransport` only in Rendezvous applications. This class is not intended for use in your EMS Java clients.

Both TIBCO Rendezvous and EMS must be purchased, installed, and configured before creating pure Java Rendezvous applications that use the `TibrvJMSTransport class`.

The `TibrvJMSTransport` class provides Rendezvous reliable communication only. Other types of communication, such as certified messaging, are not supported by this transport.

Applications using this transport can send messages to a topic on an EMS server that has the same topic name as the subject of the message. EMS topics receiving Rendezvous messages sent by way of the `TibrvJMSTransport` do not need to specify the `import` property. This transport cannot be used to send messages to Jakarta Messaging queues.

For more information about `TibrvNetTransport` and how to create use transports in TIBCO Rendezvous Java programs, see TIBCO Rendezvous documentation. For more information about the additional methods of TibrvJMSTransport, see the *TIBCO Enterprise Message Service Java API Reference*.

Monitor Server Activity

System administrators must monitor and manage the TIBCO Enterprise Message Service server. The logging, monitoring, and statistics facilities provided by the server allow system administrators to effectively view system activity and track system performance.

Server Health and Metrics

You can configure the TIBCO Enterprise Message Service server to service HTTP(S) GET requests for the current health and metrics of the server on a dedicated port.

This feature can be used to support the health check probes in Kubernetes. For more information refer to the Kubernetes documentation.

Configure the Monitor Listen

The `monitor_listen` configuration property in `tibemsd.conf` controls the interface and port the server will service HTTP(S) health check and Prometheus metrics requests on. If this property is not set, the server will not attempt to service these type of requests. This property cannot be set dynamically.

You can use only one `monitor_listen` and this listen should not conflict with other server listens.

These same restrictions apply to `secondary_monitor_listen` which is used by a server designated as secondary in a fault tolerant pair.

Whether a monitor listen uses HTTP or HTTPS is specified in the property itself. For information on how to configure explicit TLS properties for these monitor listens, refer to the Configure HTTPS in the Server section of the TLS Protocol chapter. If any of the corresponding properties is unset, the server attempts to use the corresponding TLS property applicable to the regular server listens in its place.

Health Check Response

A requestor can check whether the server is live or ready. An OK response to a liveness request means the server is up and running. An OK response on a readiness request means the server is in the active state while a BAD response means the server is not.

Liveness requests to the server should be HTTP(S) GET requests for the `path /isLive`. Readiness requests to the server should be HTTP(S) GET requests for the path `/isReady`.

For example: `http://machine:7220/isLive` and `http://machine:7220/isReady`.

Prometheus Metrics

You can configure TIBCO Enterprise Message Service to provide Prometheus-formatted metrics over an HTTP(S) listen port. This is the same port used to service health check responses.

- Metrics describing overall server state and the state of all queues and topics can be monitored at the path `/metrics`.
- Metrics describing the overall server state can be monitored at the path `/metrics/server`.
- Metrics describing the state of all queues and topics can be monitored at the paths `/metrics/queues` and `/metrics/topics`.
- Metrics describing individual destinations can be monitored at the paths `/metrics/queues/<`*queue name*`>` and `/metrics/topics/<`*topic name*`>`. In addition to naming an individual queue or topic, you can specify a wildcard to select metrics from a group of queues or topics. This type of request would yield the same set of individual metrics but just for the matching queues or topics. See Wildcards for more information.

Log Files and Tracing

You can configure the TIBCO Enterprise Message Service server to write a variety of information to the log. Several parameters and commands control where the log is located as well as what information is written to the log. The log can be written to a file, to the system console, or to both.

Configure the Log File

The `logfile` configuration parameter in `tibemsd.conf` controls the location and the name of the log file.

You can specify that the log file should be backed up and emptied after it reaches a maximum size. This allows you to rotate the log file and ensure that the log file does not grow boundlessly. The `logfile_max_size` configuration parameter allows you to specify the maximum size of the current log file. Set the parameter to 0 to specify no limit. Use KB, MB, or GB units.

Once the log file reaches its maximum size, it is copied to a file with the same name as the current log file except a sequence number is appended to the name of the backup file. On startup—and only on startup—the server queries the directory and determines the first available sequence number. It then uses the next sequence number when it needs to back up the current log file. By doing so, you can keep a continuous sequencing, as long as you retain the most recent log file (highest sequence number) between server restarts. Conversely, if you move or remove all log files before a server restart, then the sequencing will restart at 1.

For example, if the current log file is named `tibems.log`, the first copy is named `tibems.log.1`, the second is named `tibems.log.2`, and so on. Similarly, if the highest sequence number in use when the server starts is 19, or `tibemsd.log.19`, then the next backup file created will be named `tibemsd.log.20`. This is true even if you removed `tibemsd.log.19` and all other log files after the server started.

If `logfile_max_count` is specified, the server keeps at most the number of log files specified by that parameter, including the current log file. When the maximum number of log files has been reached and the server needs to back up the current log file, it deletes the oldest log file (the ones with smallest number). If you change the parameter setting, after the server is restarted, the next time it needs to rotate the log file it deletes however many of the lowest sequence numbered files required to reach the `logfile_max_count` maximum.

You can also dynamically force the log file to be backed up and truncated using the `rotatelog` command in `tibemsadmin`. See Command Listing for more information about the `rotatelog` command.

For other configuration parameters that affect the log file, see Tracing and Log File Parameters.

Trace Messages for the Server

The TIBCO Enterprise Message Service server can be configured to produce trace messages. These messages can describe actions performed for various areas of functionality (for example, Access Control, Administration, or Routing). These messages can also provide information about activities performed on or by the server, or the messages can provide warnings in the event of failures or illegal actions.

Trace messages can be sent to a log file, the console, or both. You configure tracing in the following ways:

- By configuring the `log_trace` and/or `console_trace` parameters in the `tibemsd.conf` file; see set server.
- By specifying the `-trace` option when starting the server
- By using the `set server` command when the server is running.

`log_trace` and `console_trace` can be used to configure what types of messages are to go to the log file and to the console.

Note: When you want trace messages to be sent to a log file, you must also configure the `logfile` configuration parameter. If you specify `log_trace`, and the `logfile` configuration parameter is not set to a valid file, the tracing options are stored, but they are not used until the server is started with a valid log file.

Server Tracing Options

When configuring log or console tracing, you have a variety of options for the types of trace messages that can be generated.

Specify tracing with a comma-separated list of trace options. You may specify trace options in three forms:

- plain: A trace option without a prefix character replaces any existing trace options.
- +: A trace option preceded by + adds the option to the current set of trace options.
- -: A trace option preceded by - removes the option from the current set of trace options.

Trace Option	Description
DEFAULT	Sets the trace options to the default set. This includes: • INFO • WARNING • ACL • LIMITS • ROUTE • ADMIN • RVADV • CONNECT_ERROR • CONFIG • MSG
ACL	Prints a message when a user attempts to perform an unauthorized action. For example, if the user attempts to publish a message to a secure topic for which the user has not been granted the publish permission.
ADMIN	Prints a message whenever an administration function is performed.
AUTH	Prints a message when the server authenticates an incoming connection with JAAS or OAuth 2.0.
CONFIG	Prints information about configuration files and their contents as the EMS server is starting up.
CONNECT	Prints a message when a user attempts to connect to the server.
CONNECT_ERROR	Prints a message when an error occurs on a connection.
DEST	Prints a message when a dynamic destination is created.
FLOW	Prints a message when the server enforces flow control or stops

Trace Option	Description
	enforcing flow control on a destination.
FTL	Prints trace messages related to TIBCO FTL transports.
INFO	Prints messages as the server performs various internal housekeeping functions, such as creating a configuration file, opening the persistent database files, and purging messages. Also prints a message when tracking by message ID is enabled or disabled.
JAAS	Prints messages related to any extensible security modules. Messages are printed when a username and password are passed to the LoginModule for authentication, and when a user and action are passed to the Permissions Modules for authorization.
JNDI	Prints a trace message for each JNDI lookup performed by a client, including the name and type of the object looked up and its return value.
JVM	Prints startup information about the JVM configuration, as well as any output from custom modules running in the JVM that uses `System.out`.
JVMERR	Prints output from custom modules running in the JVM that uses `System.err`.
LIMITS	Prints a message when a limit is exceeded, such as the maximum size for a destination.
LOAD	Prints the paths of any dynamically loaded libraries. The tibemsd can load FTL, ActiveSpaces, and Rendezvous libraries.
MEMORY	Prints a server trace information when reserve memory is triggered because of low server memory conditions.
MSG	Specifies that message trace messages should be printed. Message tracing is enabled/disabled on a destination or on an

Trace Option	Description
	individual message. If message tracing is not enabled for any messages or destinations, no trace messages are printed when this option is specified for log or console tracing. See Message Tracing for more information about message tracing.
OAUTH2	Prints messages related to OAuth 2.0 authentication.
OAUTH2_DEBUG	Prints detailed messages related to the OAuth 2.0 authentication process.
PRODCONS	Prints a message when a client creates or closes a producer or consumer.
ROUTE	Prints a message when routes are created or when a route connection is established.
ROUTE_DEBUG	Prints status and error messages related to the route.
RVADV	Prints TIBCO Rendezvous advisory messages whenever they are received.
SSL	Prints detailed messages of the TLS process, including certificate content.
SSL_DEBUG	Prints messages that trace the establishment of TLS connections.
TX	Prints a message when a client performs a transaction.
WARNING	Prints a message when a failure of some sort occurs, usually because the user attempts to do something illegal. For example, a message is printed when a user attempts to publish to a wildcard destination name.

Examples

The following example sets the trace log to only show messages about access control violations.

```
log_trace=ACL
```

The next example sets the trace log to show all default trace messages, in addition to TLS messages, but ADMIN messages are not shown.

```
log_trace=DEFAULT,-ADMIN,+SSL
```

Message Tracing

In addition to other server activity, you can trace messages as they are processed.

Trace entries for messages are only generated for destinations or messages that specify tracing should be performed. For destinations, you specify the `trace` property to enable the generation of trace messages. For individual messages, the `JMS_TIBCO_MSG_TRACE` property specifies that tracing should be performed for this message, regardless of the destination settings. The sections below describe the tracing properties for destinations and messages.

Message trace entries can be output to either the console or the log. The `MSG` trace option specifies that message trace entries should be displayed, and the `DEFAULT` trace option includes the `MSG` option. See Trace Messages for the Server for more information about specifying trace options.

You must set the tracing property on either destinations or messages and also set the `MSG` or `DEFAULT` trace option on the console or the log before you can view trace entries for messages.

Note: EMS tracing features do not filter unprintable characters from trace output. If your application uses unprintable characters within messages (whether in data or headers), the results of message tracing are unpredictable.

Enable Message Tracing for a Destination

The `trace` property on a destination specifies that trace entries are generated for that destination.

The trace property can optionally be specified as `trace=body`. Setting trace=body includes the message body in trace messages. The EMS server prints up to one kilobyte of a message string field, and up to a total message size of 8 KB. The trace message indicates if the full message is not printed.

Setting `trace` without the `body` option specifies that only the message sequence and message ID are included in the trace message.

When message tracing is enabled for a destination, a trace entry is output for each of the following events that occur in message processing:

- messages are received into a destination
- messages are sent to consumers
- messages are imported or exported to/from an external system
- messages are acknowledged
- messages are sent across a destination bridge
- messages are routed

Replies to request messages are traced only when the reply destination has the `trace` property. Similarly, replies to exported messages are only traced when the `trace` property is set.

Enable Message Tracing on a Message

You can enable tracing on individual messages by setting the `JMS_TIBCO_MSG_TRACE` property on the message.

The value of the property can be either null (Java/.NET null or NULL in C) or the string "body". Setting the property to null specifies only the message ID and message sequence will be included in the trace entries for the message. Setting the property to "body" specifies the message body will be included in the trace entries for the message.

When the `JMS_TIBCO_MSG_TRACE` property is set for a message, trace entries are generated for the message as it is processed, regardless of whether the `trace` property is set for any destinations the message passes through. Trace messages are generated for the message when it is sent by the producer and when it is received by the consumer.

Monitor Server Events

The TIBCO Enterprise Message Service server can publish topic messages for internal system events. For example, the server can publish a message when users connect or disconnect.

System event messages contain detail about the event stored in properties of the message. This section gives an overview of the monitoring facilities provided by the server. For a list of monitor topics and a description of the message properties for each topic, see Monitor Messages.

System Monitor Topics

The TIBCO Enterprise Message Service server can publish messages to various topics when certain events occur. There are several types of event classes, each class groups a set of related events. For example, some event classes are connection, admin, and route. Each event class is further subdivided into the events for each class. For example, the connection class has two events: connect and disconnect. These event classes are used to group the system events into meaningful categories.

All system event topic names begin with `$sys.monitor`. The remainder of the name is the event class followed by the event. For example, the server publishes a message to the topic `$sys.monitor.connection.disconnect` whenever a client disconnects from the server. The naming scheme for system event topics allows you to create wildcard subscriptions for all events of a certain class. For example, to receive messages whenever clients connect or disconnect, you would create a topic subscriber for the topic `$sys.monitor.connection.*`.

Monitor topics are created and maintained by the server. Monitor topics are not listed in the `topics.conf` file. Users can subscribe to monitor topics but cannot create them.

Monitor Messages

You can monitor messages processed by a destination as they are sent, received, or acknowledged.

You can also monitor messages that have prematurely exited due to expiration, being discarded, or a `maxRedelivery` failure.

The `$sys.monitor` topic for monitor messages has the following format:

```
$sys.monitor.D.E.destinationName
```

where *D* is the type of destination, *E* is the event you wish to monitor, and *destinationName* is the name of the destination whose messages you wish to monitor.

Message monitoring qualifiers

Possible values of *D* and *E* in message monitoring topics.

Qualifier	Value	Description
D	`T`	Destination to monitor is a topic. Include the message body in the monitor message as a byte array. Use the `createFromBytes()` method when viewing the monitor message to recreate the message body, if desired.
	`t`	Destination to monitor is a topic. Do not include the message body in the monitor message.
	`Q`	Destination to monitor is a queue. Include the message body in the monitor message as a byte array. Use the `createFromBytes()` method when viewing the monitor message to recreate the message body, if desired.
	`q`	Destination to monitor is a queue. Do not include the message body in the monitor message.
E	`s`	Monitor message is generated when a message is sent by the server to: • a consumer • a route • an external system by way of a transport
	`r`	Monitor message is generated when a message is received by the specified destination. This occurs when the message is: • Sent by a producer

Qualifier	Value	Description
		• Sent by a route • Forwarded from another destination by way of a bridge • Imported from transport to an external system
	a	Monitor message is generated when a message is acknowledged.
	p	Monitor message is generated when a message prematurely exits due to expiration, being discarded, or a `maxRedelivery` failure.
	*	Monitor message is generated when a message is sent, received, or acknowledged for the specified destination.

For example, `$sys.monitor.T.r.corp.News` is the topic for monitoring any received messages to the topic named `corp.News`. The message body of any received message is included in monitor messages on this topic. The topic `$sys.monitor.q.*.corp.*` monitors all message events (send, receive, acknowledge) for all queues matching the name `corp.*`. The message body is not included in this topic's messages.

The messages sent to this type of monitor topic include a description of the event, information about where the message came from (a producer, route, external system, and so on), and optionally the message body, depending upon the value of *D*. See Monitor Messages, for a complete description of the properties available in monitor messages.

You must explicitly subscribe to a message monitoring topic. That is, subscribing to `$sys.monitor.>` will subscribe to all topics beginning with `$sys.monitor`, but it does not subscribe you to any specific message monitoring topic such as `$sys.monitor.T.*.foo.bar`. However, if another subscriber generates interest in the message monitor topics, this subscriber will also receive those messages.

You can specify wildcards in the *destinationName* portion of the message monitoring topic to subscribe to the message monitoring topic for all matching destinations. For example, you can subscribe to `$sys.monitor.T.r.>` to monitor all messages received by all topics. For performance reasons, you may want to avoid subscribing to too many message monitoring topics. See Performance Implications of Monitor Topics for more information.

Description of Monitor Topics

Topic	Message Is Published When...
`$sys.monitor.admin.change`	The administrator has made a change to the configuration.
`$sys.monitor.connection.connect`	A user attempts to connect to the server.
`$sys.monitor.connection.disconnect`	A user connection is disconnected.
`$sys.monitor.connection.error`	An error occurs on a user connection.
`$sys.monitor.consumer.create`	A consumer is created.
`$sys.monitor.consumer.destroy`	A consumer is destroyed.
`$sys.monitor.flow.engaged`	Stored messages rise above a destination's limit, engaging the flow control feature.
`$sys.monitor.flow.disengaged`	Stored messages fall below a destination's limit, disengaging the flow control feature.
`$sys.monitor.limits.connection`	Maximum number of hosts or connections is reached.
`$sys.monitor.limits.queue`	Maximum bytes for queue storage is reached.
`$sys.monitor.limits.server`	Server memory limit is reached.
`$sys.monitor.limits.topic`	Maximum bytes for durable subscriptions is reached.
`$sys.monitor.producer.create`	A producer is created.
`$sys.monitor.producer.destroy`	A producer is destroyed.
`$sys.monitor.queue.create`	A dynamic queue is created.

Topic	Message Is Published When...
`$sys.monitor.route.connect`	A route connection is attempted.
`$sys.monitor.route.disconnect`	A route connection is disconnected.
`$sys.monitor.route.warning`	An issue worth warning about occurs on a route connection.
`$sys.monitor.route.error`	An error occurs on a route connection.
`$sys.monitor.route.interest`	A change in registered interest occurs on the route.
`$sys.monitor.server.info`	The server sends information about an event; for example, a log file is rotated.
`$sys.monitor.server.warning`	The active server detects a disconnection from the standby server.
`$sys.monitor.topic.create`	A dynamic topic is created.
`$sys.monitor.tx.action`	A local transaction commits or rolls back.
`$sys.monitor.xa.action`	An XA transaction commits or rolls back.
`$sys.monitor.`*`D`*`.`*`E`*`.`*`destination`*	A message is handled by a destination. The name of this monitor topic includes two qualifiers (*D* and *E*) and the name of the destination you wish to monitor. *D* signifies the type of destination and whether to include the entire message: • T — topic, include full message (as a byte array) into each event • t — topic, do not include full message into each event • Q — queue, include full message (as a byte

Topic	Message Is Published When...
	array) into each event • q — queue, do not include full message into each event *E* signifies the type of event: • r for receive • s for send • a for acknowledge • p for premature exit of message • * for all event types For example, `$sys.monitor.T.r.corp.News` is the topic for monitoring any received messages to the topic named `corp.News`. The message body of any received messages is included in monitor messages on this topic. The topic `$sys.monitor.q.*.corp.*` monitors all message events (send, receive, acknowledge) for all queues matching the name `corp.*`. The message body is not included in this topic's messages. The messages sent to this type of monitor topic include a description of the event, information about where the message came from (a producer, route, external system, and so on), and optionally the message body, depending upon the value of *D*. See Monitor Messages for more information about message monitoring.

Description of Topic Message Properties

Each monitor message can have a different set of these properties.

Property	Contents
conn_connid	Connection ID of the connection that generated the event.
conn_ft	Whether the client connection is a connection to a fault-tolerant server.
conn_hostname	Hostname of the connection that generated the event.
conn_ssl	Whether the connection uses the TLS protocol.
conn_type	Type of connection that generated the event. This property can have the following values: • Admin • Topic • Queue • Generic • Route • FT (connection to fault-tolerant server) • Unknown
conn_username	User name of the connection that generated the event.
conn_xa	Whether the client connection is an XA connection.
event_action	The action that caused the event. This property can have the values listed in Event Action Property Values.
event_class	The type of monitoring event (that is, the last part of the topic name without the $sys.monitor). For message monitoring, the value of this property is always set to message.
event_description	A text description of the event that has occurred.

Property	Contents
`event_reason`	The reason the event occurred (usually an error). The values this property can have are described in Event Reason Property Values.
`event_route`	For routing, the route that the event occurred on.
`message_bytes`	When the full message is to be included for message monitoring, this field contains the message as a byte array. You can use the `createFromBytes` method (in the various client APIs) to recover the message.
`mode`	Message delivery mode. This values of this property can be the following: • `persistent` • `non_persistent` • `reliable`
`msg_correlation_id`	JMS correlation ID.
`msg_id`	Message ID.
`msg_seq`	Message sequence number.
`msg_size`	Message size, in bytes.
`msg_timestamp`	Message timestamp.
`msg_expiration`	Message expiration.
`replyTo`	Message JMSReplyTo.
rv_reply	Message RV reply subject.
`source_id`	ID of the source object.
`source_name`	Name of the source object involved with the event. This

Property	Contents
	property can have the following values: • XID (global transaction ID) • message_id • connections (number of connections) • unknown (unknown name) • Any server property name • the name of the user, or anonymous
`source_object`	Source object that was involved with the event. This property can have the following values: • producer • consumer • topic • queue • permissions • durable • server • transaction • user • group • connection • message • jndiname • factory • file • limits (a limit, such as a memory limit) • route

Property	Contents
	• transport
source_value	Value of source object.
stat_msgs	Message count statistic for producer or consumer.
stat_size	Message size statistic for producer or consumer.
target_admin	Whether the target object is the admin connection.
target_created	Time that the consumer was created (in milliseconds since the epoch).
target_dest_name	Name of the target destination
target_dest_type	Type of the target destination.
target_durable	Name of durable subscriber when target is durable subscriber.
target_group	Group name that was target of the event
target_hostname	Hostname of the target object.
target_id	ID of the target object.
target_name	Name of the object that was the target of the event. This property can have the following values: • XID (global transaction ID) • message_id • connections (number of connections) • unknown (unknown name) • Any server property name • the name of the user, or anonymous

Property	Contents
`target_nolocal`	No Local flag when target is durable subscriber.
`target_object`	The general object that was the target of the event. This property can have the following values: • producer • consumer • topic • queue • durable • server • transaction • user • group • connection • message • jndiname • factory • file • limits (a limit, such as a memory limit) • route • transport
`target_selector`	Selector when the target is a consumer.
`target_subscription`	Subscription of the target object when target is durable subscriber.
`target_url`	URL of the target object.

Property	Contents
target_username	Username of the target object.
target_value	Value of the object that was the target of the event, such as the name of a topic, queue, and so on.

Event Action Property Values

Event Action Value	Description
accept	connection accepted
acknowledge	message is acknowledged
add	user added to a group
admin_commit	administrator manually committed an XA transaction
admin_rollback	administrator manually rolled back an XA transaction
commit	transaction committed
connect	connection attempted
create	something created
delete	something deleted
disconnect	connection disconnected
flow_engaged	stored messages rise above a destination's limit, engaging the flow control feature
flow_disengaged	stored messages fall below a destination's limit, disengaging the flow control feature
interest	registered interest for a route

Event Action Value	Description
modify	something changed
grant	permission granted
premature_exit	message prematurely exited
purge	topic, queue, or durable subscriber purged
receive	message posted into destination
remove	user removed from a group
resume	administrator resumed a route
revoke	permission revoked
rollback	transaction rolled back
rotate_log	log file rotated
send	message sent by server to another party
subscribe	subscription request
suspend	administrator suspended a route
txcommit	administrator manually committed a local transaction
txrollback	administrator manually rolled back a local transaction
xacommit	an application committed an XA transaction (2-phase)
xacommit_1phase	an application committed an XA transaction (1-phase)
xastart	an application started a new XA transaction

Event Action Value	Description
xastart_join	an application has joined (that is, added) a resource to an existing transaction
xastart_resume	an application resumed a suspended XA transaction
xaend_fail	an application ended an XA transaction, indicating failure
xaend_success	an application ended an XA transaction, indicating success
xaend_suspend	an application suspended an XA transaction
xaprepare	an application prepared an XA transaction
xarecover	an application called recover (to get a list of XA transactions)
xarollback	an application rolled back an XA transaction

Event Reason Property Values

Event Action Value	Description
accept	connection accepted
acknowledge	message is acknowledged
add	user added to a group
admin_commit	administrator manually committed an XA transaction
admin_rollback	administrator manually rolled back an XA transaction
commit	transaction committed
connect	connection attempted
create	something created

Event Action Value	Description
`delete`	something deleted
`disconnect`	connection disconnected
`flow_engaged`	stored messages rise above a destination's limit, engaging the flow control feature
`flow_disengaged`	stored messages fall below a destination's limit, disengaging the flow control feature
`interest`	registered interest for a route
`modify`	something changed
`grant`	permission granted
`premature_exit`	message prematurely exited
`purge`	topic, queue, or durable subscriber purged
`receive`	message posted into destination
`remove`	user removed from a group
`resume`	administrator resumed a route
`revoke`	permission revoked
`rollback`	transaction rolled back
`rotate_log`	log file rotated
`send`	message sent by server to another party
`subscribe`	subscription request
`suspend`	administrator suspended a route

Event Action Value	Description
`txcommit`	administrator manually committed a local transaction
`txrollback`	administrator manually rolled back a local transaction
`xacommit`	an application committed an XA transaction (2-phase)
`xacommit_1phase`	an application committed an XA transaction (1-phase)
`xastart`	an application started a new XA transaction
`xastart_join`	an application has joined (that is, added) a resource to an existing transaction
`xastart_resume`	an application resumed a suspended XA transaction
`xaend_fail`	an application ended an XA transaction, indicating failure
`xaend_success`	an application ended an XA transaction, indicating success
`xaend_suspend`	an application suspended an XA transaction
`xaprepare`	an application prepared an XA transaction
`xarecover`	an application called recover (to get a list of XA transactions)
`xarollback`	an application rolled back an XA transaction

View Monitor Topics

Monitor topics are similar to other topics. To view these topics, create a client application that subscribes to the desired topics.

Because monitor topics contain potentially sensitive system information, authentication and permissions are always checked when clients access a monitor topic. That is, even if authentication for the server is disabled, clients are not able to access monitor topics unless they have logged in with a valid username and password and the user has permission to view the desired topic.

The admin user and members of the $admin group have permission to perform any server action, including subscribing to monitor topics. All other users must be explicitly granted permission to view monitor topics before the user can successfully create subscribers for monitor topics. For example, if user BOB is not a member of the $admin group, and you wish to allow user BOB to monitor all connection events, you can grant BOB the required permission with the following command using the administration tool:

```
grant topic $sys.monitor.connection.* BOB subscribe
```

Bob's application can then create a topic subscriber for $sys.monitor.connect.* and view any connect or disconnect events.

Note: Topics starting with $sys.monitor do not participate in any permission inheritance from parent topics other than those starting with $sys.monitor (that is, *.* or *.> is not a parent of $sys.monitor).

Therefore, granting permission to a user to subscribe to > does not allow that user to subscribe to $sys.monitor topics. You must explicitly grant users permission to $sys.monitor topics (or parent topics, such as $sys.monitor.admin.*) for a user to be able to subscribe to that topic.

Monitor topics publish messages of type MapMessage. Information about the event is stored within properties in the message. Each system event has different properties. Monitor Messages, describes each of the monitor topics and the message properties for the messages published on that topic. Your application can receive and display all or part of a monitor message, just as it would handle any message sent to a topic. However, there are some ways in which monitor messages are handled differently from standard messages:

- Monitor messages cannot be routed to other servers.
- Monitor messages are not stored persistently on disk.
- Monitor messages are not swapped from process memory to disk.

You can have any number of applications that subscribe to monitor messages. You can create different applications that subscribe to different monitor topics, or you can create one application that subscribes to all desired monitor topics. Your topic subscribers can also use message selectors to filter the monitor messages so your application receives only the messages it is interested in.

Performance Implications of Monitor Topics

The TIBCO Enterprise Message Service server only generates messages for monitor topics that currently have subscribers. So, if no applications subscribe to monitor topics, no monitor messages are generated. Generating a monitor message does consume system resources, and therefore you should consider what kinds of monitoring your environment requires. System performance is affected by the number of subscribers for monitor topics as well as the frequency of messages for those topics.

For development and testing systems, monitoring all system events is probably desirable. Usually, development and testing systems do not have large message volumes, and monitoring can give you information about system problems.

For production systems, monitoring all events may have an adverse effect on system performance. Therefore, you should not create topic subscribers for `$sys.monitor.>` in your production system. Also, monitor events are likely to be added in future releases, so the number of monitor topics may grow. Subscriptions to monitor topics in production systems should always be limited to specific monitor topics or wildcard subscriptions to specific classes of monitor topics that are required.

Also, consider the frequency of messages to each monitor topic. System administration events, such as creating topics, routes, and changing permissions, do not occur frequently, so creating subscriptions for these types of events will most likely not have a significant effect on performance.

Also, using message selectors to limit monitor messages can improve performance slightly. The server does not send any messages that do not match a subscriber's message selector. Even though the message is not sent, the message is still generated. Therefore there is still system overhead for subscribers to a monitor topic, even if all messages for that topic do not match any subscriber's message selector filter.

Server Statistics

The TIBCO Enterprise Message Service server allows you to track incoming and outgoing message volume, message size, and message statistics for the server overall as well as for each producer, consumer, or route. You can configure the type of statistics collected, the interval for computing averages, and amount of detail for each type.

Statistic tracking can be set in the server's configuration file, or you can change the configuration dynamically using commands in the administration tool or by creating your own application with the administration APIs.

Statistics can be viewed using the administration tool, or you can create your own application that performs more advanced analysis of statistics using the administration APIs.

This section details how to configure and view statistics using the configuration files and administration tool commands. For more information about the administration APIs, see the description of `com.tibco.tibjms.admin` in the online documentation.

Note: The TIBCO Enterprise Message Service server tracks the number of incoming or outgoing messages, but only messages sent or received by a producer, consumer, or route are tracked. The server also sends system messages, but these are not included in the number of messages.

However, the server can add a small amount of data to a message for internal use by the server. This overhead is counted in the total message size, and you may notice that some messages have a greater message size than expected.

Overall Server Statistics

The server always collects certain overall server statistics. This includes the rate of inbound and outbound messages (expressed as number of messages per second), message memory usage, disk storage usage, and the number of destinations, connections, and durable subscriptions. Gathering this information consumes virtually no system resources, therefore these statistics are always available. You can view overall server statistics by executing the `show server` command.

The default interval for collecting overall server statistics is 1 second. You may wish to view average system usage statistics over a larger interval. The `server_rate_interval` configuration parameter controls the collection interval for server statistics. The parameter can be set in the configuration file or dynamically using the `set server` command. This parameter can only be set to positive integers greater than zero.

Enable Statistics Gathering

Each producer, consumer, destination, and route can gather overall statistics and statistics for each of its destinations. To enable statistic gathering, you must set the `statistics` parameter to enabled. This parameter can be specified in the configuration file, and it can be changed dynamically using the set server command.

The `statistics` parameter allows you to globally enable and disable statistic gathering. Statistics are kept in server memory for the life of each object. If you wish to reset the total statistics for all objects to zero, disable statistic gathering, then re-enable it. Server statistics are also reset when the server shuts down and restarts, or in the event of a fault-tolerant failover.

For each producer, consumer, destination, and route the total number of sent/received messages and total size of messages is maintained. Also, producers and consumers keep these statistics for each destination that they use to send or receive messages.

The rate of incoming/outgoing messages and message size is calculated over an interval. By default, the average is calculated every three seconds. You can increase or decrease this value by altering the `rate_interval` parameter. This parameter can be set in the configuration file or dynamically using the `set server` command. Setting this parameter to 0 disables the tracking of statistics over an interval—only the total statistics for the destination, route, producer, or consumer are kept.

Gathering total statistics for producers, consumers, destinations, and routes consumes few system resources. Under most circumstances, enabling statistic gathering and average calculations should not affect system performance.

Detailed Statistics

In some situations, the default statistic gathering may not be sufficient. For example, if a topic subscriber subscribes to wildcard topics, the total statistics for all topics that match the wildcard are kept. You may wish to get further detail in this case and track the statistics for each actual topic the subscriber receives.

The following situations may require detailed statistic gathering:

- Topic subscribers that subscribe to wildcard topics
- Message producers that do not specify a destination when they are created. These message producers can produce messages for any destination, and the destination name is specified when a message is sent.

- Routes can have incoming and outgoing messages on many different topics.

To enable detailed statistics, set the `detailed_statistics` parameter to the type of statistics you wish to receive. The parameter can have the following values:

- `NONE` — disables detailed statistic gathering.
- `CONSUMERS` — enables detailed statistics for topic subscribers with wildcard topic names.
- `PRODUCERS` — enables detailed statistics for producers that do not specify a destination when they are created.
- `ROUTES` — enables detailed statistics for routes.

You can set the `detailed_statistics` parameter to `NONE` or any combination of `CONSUMERS`, `PRODUCERS`, or `ROUTES`. To specify more than one type of detailed statistic gathering, provide a comma-separated list of values. You can set the `detailed_statistics` parameter in the configuration file or dynamically by using the set server command. For example, the following `set server` command enables detailed statistic tracking for producers and routes.

```
set server detailed_statistics = PRODUCERS,ROUTES
```

Collecting detailed statistics does consume memory, and can adversely affect performance when gathering a high volume of statistics. There are two parameters that allow you to control resource consumption when collecting detailed statistics. First, you can control the amount of time statistics are kept, and second you can set a maximum amount of memory for detailed statistic gathering. When application programs create many dynamic destinations, we recommend against gathering detailed statistics.

The `statistics_cleanup_interval` parameter controls how long detailed statistics are kept. This parameter can be set either in the configuration file or dynamically with the set server command. By default, statistics are kept for 15 seconds. For example, if there is a topic subscriber for the topic `foo.*`, and the subscriber receives a message on topic `foo.bar`, if no new messages arrive for topic `foo.bar` within 15 seconds, statistics for topic `foo.bar` are deleted for that consumer. You can set this parameter to 0 to signify that all detailed statistics are to be kept indefinitely. Of course, statistics for an object only exist as long as the object itself exists. That is, if a message consumer terminates, all detailed statistics for that consumer are deleted from memory.

The `max_stat_memory` parameter controls the amount of memory used by detailed statistics. This parameter can be set either in the configuration file or dynamically with the set server command. By default, this parameter is set to 0 which signifies that detailed

statistics have no memory limit. If no units are specified, the value of this parameter is in bytes. Optionally, you can specify units as KB, MB, or GB. When the specified limit is reached, the server stops collecting new statistics. The server will only resume collecting statistics if the amount of memory used decreases
(for example, if the `statistics_cleanup_interval` is set and old statistics are removed).

Display the Statistics

When statistic collecting is enabled, you can view statistics for producers, consumers, routes, and destinations using the `show stat` command in the administration tool.

The `show stat` command allows you to filter the statistics based on destination name, user name, connection ID, or any combination of criteria. You can optionally specify the `total` keyword to retrieve only the total statistics (this suppresses the detailed output). You can also optionally specify the "wide" keyword when displaying statistics for destinations or routes. This specifies that inbound and outbound message statistics should be displayed on the same line (the line can be 100 characters or more).

The following illustrates displaying statistics for a route where detailed statistic tracking is enabled.

```
tcp://server1:7322> show stat route B
Inbound statistics for route 'B':
                         Total Count      Rate/Second
Destination              Msgs   Size      Msgs   Size
<total>                   189   37.9 Kb     10    2.0 Kb
Topic: dynamic.0           38    7.6 Kb      2    0.4 Kb
Topic: dynamic.1           38    7.6 Kb      2    0.4 Kb
Topic: dynamic.2           38    7.6 Kb      2    0.4 Kb
Topic: dynamic.3           38    7.6 Kb      2    0.4 Kb
Topic: dynamic.4           37    7.4 Kb      2    0.4 Kb
Outbound statistics for route 'B':
                         Total Count      Rate/Second
Destination              Msgs   Size      Msgs   Size
<total>                  9538    1.9 MB     10    2.1 Kb
Topic: dynamic.0         1909  394.9 Kb      2    0.4 Kb
Topic: dynamic.1         1908  394.7 Kb      2    0.4 Kb
Topic: dynamic.2         1907  394.5 Kb      2    0.4 Kb
Topic: dynamic.3         1907  394.5 Kb      2    0.4 Kb
Topic: dynamic.4         1907  394.5 Kb      2    0.5 Kb
```

See `show stat` for more information and detailed syntax of the show stat command.

TLS Protocol

Transport Layer Security (TLS) is a protocol that provides secure authentication and transmits encrypted data over the Internet or an internal network.

The TLS protocol is complex, and this chapter is not a complete description of TLS. Instead, this chapter describes how to configure TLS in the TIBCO Enterprise Message Service server and in client applications that communicate with the server. For a more complete description of TLS, see the TLS specification at https://tools.ietf.org/html/rfc5246 and the article at https://en.wikipedia.org/wiki/Transport_Layer_Security.

Note: If end-to-end data security is necessary, note that it requires using an encrypted storage. For example for file stores, that would involve placing the EMS store files in an encrypted file system.

TLS Support in TIBCO Enterprise Message Service

TIBCO Enterprise Message Service supports the Transport Layer Security (TLS) protocol.

TLS uses public and private keys to encrypt data over a network connection to secure communication between pairs of components:

- between an EMS client and the `tibemsd` server
- between the `tibemsadmin` tool or API and the tibemsd server
- between MSGMX and the `tibemsd` server
- between two routed servers
- between two fault-tolerant servers (not applicable when using FTL stores)

TLS provides secure communication that works with other mechanisms for authentication available in the EMS server. When `authorization` is enabled in the server, the connection undergoes a two-phase authentication process. First, a TLS hand-shake between client and server initializes a secure connection. Second, the EMS server checks the credentials of the

client using the supplied username and password. If the connecting client does not supply a valid username and password combination, the connection fails, even if the TLS handshake succeeded.

Tip: When authorization is enabled, usernames and passwords are always checked, even on TLS secured connections.

Implementations

The TIBCO Enterprise Message Service server and the C client libraries use OpenSSL for TLS support.

For more information, see www.openssl.org.

EMS Java clients use JSSE (from Sun JavaSoft). JSSE is included in Java distributions.

EMS .NET Framework clients use the Microsoft implementation of TLS. The Microsoft implementation of TLS is compatible with OpenSSL. Certificates required by the client can either be stored in files or the Microsoft certificate store. However, Microsoft requires that the root certificate be installed in the Microsoft Certificate Store, even when certificate files are in use.

EMS distributions usually build and include the latest version of OpenSSL publicly available at the time of release. For exact version numbers see the Third Party Software License Agreements documented in the TIBCO Software Inc. End User License Agreement for TIBCO Enterprise Message Service.

Digital Certificates

Digital certificates are data structures that represent identities. EMS uses certificates to verify the identities of servers and clients. Though it is not necessary to validate either the server or the client for them to exchange data over TLS, certificates provide an additional level of security.

A digital certificate is issued either by a trusted third-party certificate authority, or by a security officer within your enterprise. Usually, each user and server on the network requires a unique digital certificate, to ensure that data is sent from and received by the correct party.

In order to support TLS, the EMS server must have a digital certificate. Optionally, EMS clients may also be issued certificates. If the server is configured to verify client certificates, a client must have a certificate and have it verified by the server. Similarly, an EMS client can be configured to verify the server's certificate. Once the identity of the server and/or client has been verified, encrypted data can be transferred over TLS between the clients and server.

A digital certificate has two parts—a public part, which identifies its owner (a user or server); and a private key, which the owner keeps confidential.

The public part of a digital certificate includes a variety of information, such as the following:

- The name of the owner, and other information required to confirm the unique identity of the subject. This information can include the URL of the web server using the digital certificate, or an email address.
- The subject's public key.
- The name of the certificate authority (CA) that issued the digital certificate.
- A serial number.
- The length of time the certificate will remain valid—defined by a start date and an end date.

The most widely-used standard for digital certificates is ITU-T X.509. TIBCO Enterprise Message Service supports digital certificates that comply with X.509 version 3 (X.509v3); most certificate authorities, such as Verisign and Entrust, comply with this standard.

Digital Certificate File Formats

TIBCO Enterprise Message Service supports the following file formats for digital certificates:

- PEM (Privacy Enhanced Mail)
- DER (Distinguished Encoding Rules)
- PKCS#7
- PKCS#12
- Java KeyStore (for client digital certificates)

Private Key Formats

TIBCO Enterprise Message Service supports the following file formats for private keys:

- PEM (Privacy Enhanced Mail)
- DER (Distinguished Encoding Rules)
- PKCS#8
- PKCS#12

The EMS server uses OpenSSL to read private keys. It does *not* read Java KeyStore files.

File Names for Certificates and Keys

For all parameters that specify the identity (digital certificate), private key, issuer (certificate chain), or trusted list of certificate authorities, valid files must be specified. Not all types of files are supported for clients and servers. The description of each parameter details which formats it supports.

The following table lists the valid types of files.

Extension	Description
`.pem`	PEM encoded certificates and keys (allows the certificate and private key to be stored together in the same file)
`.der`	DER encoded certificates
`.p8`	PKCS#8 file
`.p7b`	PKCS#7 file
`.p12`	PKCS12 file (allows the certificate and private key to be stored together in the same file)
`.jks`	Java KeyStore file

Certificates are located in the *EMS_HOME*`/samples/certs` directory. EMS is installed with some sample certificates and private keys that are used by the sample configuration files.

The sample certificates include:

- A root, self-signed certificate and corresponding private keys in encrypted PEM and PKCS8 formats:

```
server_root.cert.pem
server_root.key.pem
server_root.key.p8
```

- A server certificate and corresponding private keys in encrypted PEM and PKCS8 formats. This certificate is issued by `server_root.cert.pem` and is used by the server:

```
server.cert.pem
server.key.pem
server.key.p8
```

- A root, self-signed certificate and corresponding private key in encrypted PEM and PKCS8 formats.

```
client_root.cert.pem
client_root.key.pem
client_root.key.p8
```

- A client certificate and corresponding private key in encrypted PEM and PKCS8 formats. This certificate is issued by `client_root.cert.pem` and is used by the clients:

```
client.cert.pem
client.key.pem
client.key.p8
```

- A PKCS12 file that includes the `client.cert.pem` client certificate, the `client.key.pem` client private key, and the `client_root.cert.pem` issuer certificate:

```
client_identity.p12
```

Configure TLS in the Server

To use TLS, each instance of `tibemsd` must have a digital certificate and a private key. The server can optionally require a certificate chain or trusted certificates.

Set the server to listen for TLS connections from clients by using the `listen` parameter in `tibemsd.conf`. To specify that a port accept TLS connections, specify the TLS protocol in the listen parameter as follows:

```
listen = ssl://localhost:7243
```

TLS Parameters

Several TLS parameters can be set in `tibemsd.conf`. The minimum configuration is only one required parameter—`ssl_server_identity`. However, if the server's certificate file does not contain its private key, then you must specify it in `ssl_server_key`.

TLS Server Parameters provides a complete description of the TLS parameters that can be set in `tibemsd.conf`.

Command Line Options

The server accepts a few command-line options for TLS.

When starting `tibemsd`, you can specify the following options:

- `-ssl_trace`—enables tracing of loaded certificates. This prints a message to the console during startup of the server that describes each loaded certificate.
- `-ssl_debug_trace`—enables more detailed TLS tracing for debugging only; it is not for use in production systems.
- `-ssl_password`—specifies the private key password. Alternatively, you can specify this password in the `ssl_server_password` parameter in `tibemsd.conf`. If you do not supply a password using either of these methods, `tibemsd` will prompt for the password when it starts. For more information, see the description of the `ssl_password` configuration parameter.

Configure HTTPS in the Server

You can configure TIBCO Enterprise Message Service to provide health checks and Prometheus-formatted metrics over an HTTP(S) listen port. For more information about this feature, refer to the Server Health and Metrics.

To use HTTPS, each instance of `tibemsd` must have a digital certificate and a private key. The server can optionally be configured to handle trusted certificates.

Set the server to listen for HTTPS requests by using the `monitor_listen` parameter in tibemsd.conf. To specify that a port accepts HTTPS connections, specify the HTTPS protocol in the `monitor_listen` parameter as follows:

```
monitor_listen = https://servername:port
```

HTTPS Properties

There are a few HTTPS properties that can be set in tibemsd.conf. None are strictly required for the HTTPS listen to work as any unset parameter results in the matching server TLS parameter to be used in its place. For example, if `monitor_ssl_identity` is unset, the configured `ssl_server_identity` is used in its place. HTTPS Server Parameters provides a complete description of the TLS parameters that can be set in tibemsd.conf.

Configure TLS in EMS Clients

In basic TLS connections to the EMS server, with standard ciphers, EMS Java clients require no additional libraries or JAR files. The use of ciphers that use stronger encryption may require the installation of the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files into the JRE.

Client Digital Certificates

When client authentication with a digital certificate is required by the EMS server (see the description of the `ssl_require_client_cert` parameter in `tibemsd.conf`), the client may combine its client certificate and private key in a single file in one of the following formats:

- PKCS#12

- Java KeyStore

You can also store the private key file separately from the client certificate file. If this is the case, the certificate and private key must be stored in one of the following formats:

- PEM
- PKCS#8

The format of the client digital certificate and private key file depends on the TLS vendor used by the client. For more information about formats, see your TLS vendor's documentation.

Configure TLS

A client connecting to an EMS server can configure TLS characteristics in the following ways:

- Create a connection factory that specifies the appropriate TLS parameters and use JNDI to lookup the connection factory. The server URL in the connection factory must specify the TLS protocol, and the factory must specify appropriate TLS parameters.

  A preconfigured connection factory is the preferred mechanism in many situations. See Create Connection Factories for Secure Connections and Perform Secure Lookups for details on how to create a connection factory with TLS parameters in EMS.

- Dynamically create a connection factory, as described in Dynamically Creating Connection Factories and set the global TLS parameters locally using the `TibjmsSSL` class (Java), `tibemsSSLParams` type (C), or `EMSSSL` class (C#).

Specifying any TLS parameters within a connection factory causes all global TLS parameters set with the `TibjmsSSL` class, `tibemsSSLParams` type or `EMSSSL` class to be ignored.

Configure a Connection Factory

You can configure a connection factory using the administration tool or the administration APIs.

See EMS Administration Tool.

When configuring a connection factory, you can specify several TLS parameters, similar to the server parameters that you can configure in `tibemsd.conf`.

Note: When configuring a connection factory, EMS does not verify any file names specified in the TLS parameters. At the time the factory is retrieved using JNDI, the EMS server attempts to resolve any file references. If the files do not match the supported types or the files are not found, the JNDI lookup fails with a `ConfigurationException`.

Note: Because connection factories do not contain the `ssl_password` (for security reasons), the EMS server uses the password that is provided in the "create connection" call for user authentication. If the create connection password is different from the `ssl_password`, the connection creation will fail.

The following table describes the TLS parameters that can be set in a connection factory.

For more information about each parameter, see the description of the equivalent parameter in tibemsd.conf.

Parameter	Description
`ssl_vendor`	The vendor name of the TLS implementation that the client uses. Since software release 8.4.0, only one vendor (JSSE) is supported for the Java client, so use of this parameter is optional in that context.
`ssl_identity`	The client's digital certificate. For more information on file types for digital certificates, see File Names for Certificates and Keys.

`ssl_issuer`	Issuer's certificate chain for the client's certificate. Supply the entire chain, including the CA root certificate. The client reads the certificates in the chain in the order they are presented in this parameter.
	
	Example
	
	`ssl_issuer = certs\CA_root.pem`
	`ssl_issuer = certs\CA_child1.pem`
	`ssl_issuer = certs\CA_child2.pem`

Parameter	Description
	For more information on file types for digital certificates, see File Names for Certificates and Keys.
`ssl_private_key`	The client's private key. If the key is included in the digital certificate in `ssl_identity`, then you may omit this parameter. For more information on file types for digital certificates, see File Names for Certificates and Keys.
`ssl_trusted`	List of CA certificates to trust as issuers of server certificates. Supply only CA root certificates. For more information on file types for digital certificates, see File Names for Certificates and Keys.
`ssl_verify_host`	Specifies whether the client should verify the server's certificate. The values for this parameter are `enabled` or `disabled`. By default, this parameter is enabled, signifying the client should verify the server's certificate. When `disabled`, the client establishes secure communication with the server, but does not verify the server's identity.
`ssl_verify_hostname`	Specifies whether the client should verify the name in the CN field of the server's certificate. The values for this parameter are `enabled` and `disabled`. By default, this parameter is enabled, signifying the client should verify the name of the connected host or the name specified in the `ssl_expected_hostname` parameter against the value in the server's certificate. If the names do not match, the client rejects the connection. When `disabled`, the client establishes secure communication with the server, but does not verify the server's name.
`ssl_expected_hostname`	The name the client expects in the CN field of the server's

Parameter	Description
	certificate. If this parameter is not set, the expected name is the hostname of the server. The value of this parameter is used when the `ssl_verify_hostname` parameter is enabled.
`ssl_ciphers`	Specifies the cipher suites that the client can use. Supply a colon-separated list of cipher names. Names may be either OpenSSL names, or longer descriptive names. For more information, see Specify Cipher Suites.
`ssl_auth_only`	Specifies whether TLS should be used to encrypt all server-client communications, or only client authentication. When `enabled`, the client requests TLS be used only for authentication. The server then uses TCP communications for further data exchange. When `disabled` or absent, all communication between the client and server must be TLS encrypted. For an overview of this feature, see TLS Authentication Only.

Specify Cipher Suites

On the EMS server, specify cipher suites using the `ssl_server_ciphers` configuration parameter in `tibemsd.conf`.

For more information about server configuration files, see Configuration Files.

For clients connecting with a connection factory, specify cipher suites using the `ssl_ciphers` connection factory parameter. For more information, see Configure TLS in EMS Clients.

Syntax for Cipher Suites

EMS uses OpenSSL for TLS support. Therefore, the cipher suite names can be specified as the OpenSSL name for the cipher suite.

When specifying cipher suites, the usual way to specify more than one cipher suite is to separate each suite name with a colon (:) character. Alternatively, you can use spaces and commas to separate names.

Java Client Syntax

The syntax for specifying the list of cipher suites is different for Java clients than for any other location where cipher suites can be specified. For Java clients, you specify a qualifier (for example, + to add the suite) followed by the cipher suite name. Cipher suite names are case-sensitive. The following table describes the qualifiers you can use when specifying cipher suite names in a ConnectionFactory for Java clients.

Qualifier	Description
`+`	Add the cipher to the list of ciphers.
`-`	Remove the cipher from the list of ciphers.
`>`	Move the cipher to the end of the list.
`<`	Move the cipher to the beginning of the list.
`ALL`	All ciphers from the list (except null ciphers). You can use this keyword to add or remove all ciphers. At least one cipher suite must be present, otherwise the TLS connection fails to initialize. So, if you use `-ALL`, you must subsequently add the desired ciphers to the list.

This example specifies cipher suites in the `ssl_ciphers` connection factory parameter in a Java client:

```
-ALL:+ECDHE-RSA-AES256-GCM-SHA384:<ECDHE-RSA-AES128-GCM-SHA256
```

This example specifies cipher suites using Java names:

```
-ALL:+TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256:+TLS_ECDHE_RSA_WITH_AES_256_
GCM_SHA384:<SSL_RSA_WITH_3DES_EDE_CBC_SHA
```

Syntax for All Other Cipher Suite Specifications

For any cipher suite list that is not specified in a connection factory of a Java client, use the OpenSSL syntax. In particular, C clients and the `ssl_server_ciphers` configuration parameter require OpenSSL syntax.

While the full syntax of OpenSSL cipher suite selection is supported for TLSv1.2 cipher suites, we recommend using a simplified form based on the `SECLEVEL` directive. (See the OpenSSL documentation on `SECLEVEL` for details at https://www.openssl.org/docs/man3.0/man3/SSL_CTX_set_security_level.html.) For instance, a cipher specification consisting of only `@SECLEVEL=2` will yield a set of ciphers that is secure, but maintains a moderate level of backward compatibility.

In OpenSSL syntax, specifying a cipher suite name adds that cipher suite to the list. Each cipher suite name can be preceded by a qualifier. Cipher suite names are case-sensitive. The following table describes the qualifiers available using OpenSSL syntax.

> **Note:** The syntax described in the table below only applies to TLSv1.2 cipher suites. TLSv1.3 connections are subject only to the `SECLEVEL` restrictions. TLSv1.3 cipher suites always take priority over TLSv1.2 ones.

Qualifier	Description
/	When entered as the first item in the list, this option causes EMS to begin with an empty list, and add the ciphers that follow the slash. If the / does not prefix the cipher list, then EMS prefixes the cipher list with the OpenSSL cipher string DEFAULT. This modifier can only be used at the beginning of the list. If the / appears elsewhere, the syntax of the cipher suite list will be incorrect and cause an error.

Qualifier	Description
+	Moves the cipher to the end of the list. This qualifier is used to move an existing cipher. It can not be used to add a new cipher to the list.
-	Remove the cipher from the list of ciphers. When this option is used, the cipher can be added later on in the list of ciphers.
!	Permanently disable the cipher within the list of ciphers. Use this option if you wish to remove a cipher and you do not want later items in the list to add the cipher to the list. This qualifier takes precedence over all other qualifiers.
ALL	All ciphers from the list (except null ciphers). You can use this keyword to add or remove all ciphers. At least one cipher suite must be present or the TLS connection fails to initialize. So, after using `-ALL`, you should add at least one cipher to the list.

This example specifies cipher suites in the `ssl_server_ciphers` configuration parameter.

```
ssl_server_ciphers = -ALL:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-
GCM-SHA384
```

This example illustrates disabling `ECDHE-RSA-AES128-GCM-SHA256`, then adding all other ciphers:

```
ssl_server_ciphers = !ECDHE-RSA-AES128-GCM-SHA256:ALL
```

Default Cipher List

The EMS server and C client library use `DEFAULT` as their default cipher list. For details on the cipher suites corresponding to `DEFAULT` for a given version of OpenSSL, refer to the OpenSSL documentation.

Supported Cipher Suites

For a current list of supported cipher suites, run the `help ciphers` command in the `tibemsadmin` in tool.

Note that this list is only relevant to the release of *TIBCO Enterprise Message Service* that ships with the particular version of `tibemsadmin` that is running when the help command is issued

Supported Cipher Suites for the Server and C Clients

The EMS server and C client library support a subset of the cipher suites that OpenSSL supports.

Supported Cipher Suites for Java Clients

For Java clients, restrictions apply to some of the newer cipher suites. Using these may require adjustments to some of the following: JVM version, JVM vendor, JCE unlimited strength jurisdiction policy files, the `java.security` properties file and X509 certificate digital signature algorithms.

Note: Some updates of Java might deactivate compromised cipher suites. If absolutely required, refer to the Java documentation to reactivate them.

Supported Cipher Suites for .NET Clients

In general, the .NET client library supports the cipher suites that .NET supports. Refer to your MSDN documentation or contact Microsoft support for complete details on supported ciphers on specific .NET environments.

TLS Authentication Only

EMS servers can use TLS for secure data exchange (standard usage), or only for client authentication. This section describes the use of TLS for client authentication.

Motivation

Some applications require strong or encrypted authentication, but do not require message encryption.

In this situation, application architects could configure TLS with a null cipher. However, this solution incurs internal overhead costs of TLS calls, decreasing message speed and throughput.

For optimal performance, the preferred solution is to use TLS only to authenticate clients, and then avoid TLS calls thereafter, using ordinary TCP communications for subsequent data exchange. Message performance remains unaffected.

Preconditions

All the following preconditions must be satisfied to use TLS only for authentication:

- The server must explicitly enable the parameter `ssl_auth_only` in the `tibemsd.conf` configuration file.
- The client program must request a connection that uses TLS for authentication only. Clients can specify this request in factories by enabling the `ssl_auth_only` parameter, or by calling:
  - Java: `TibjmsSSL.setAuthOnly`
  - C: `tibemsSSLParams_SetAuthOnly`
  - C#: `EMSSSL.SetAuthOnly`

See Also

Server parameter `ssl_auth_only`

Client parameter `ssl_auth_only`

Enable FIPS Compliance

You can enable TIBCO Enterprise Message Service to run in compliance with Federal Information Processing Standard (FIPS), Publication 140-2.

Enable the EMS Server

> **Note:** The EMS server supports FIPS compliance only on the Linux and Windows platforms.

To enable FIPS 140-2 operations in the EMS server:

- Set the `fips140-2` parameter in the main configuration file to `true`.
- Ensure that incompatible parameters, listed below, are not included in the server configuration files.
- Ensure that the `ssl_server_ciphers` parameter for the EMS server is configured to use a supported cipher suite.

When `fips140-2` is enabled, on start-up the EMS server initializes in compliance with FIPS 140-2. If the initialization is successful, the EMS server prints a message indicating that it is operating in this mode. If the initialization fails, the server exits (regardless of the startup_abort_list setting).

Incompatible Parameters

In order to operate in FIPS compliant mode, you must not include these parameters in the `tibemsd.conf` file:

- `ssl_server_ciphers`
- `ft_ssl_ciphers`

These parameters cannot be included in the `routes.conf` file:

- `ssl_ciphers`

Enable EMS Clients

Java and C client applications can operate in FIPS compliance:

- **Java Clients**

Java clients that use the Bouncy Castle FIPS provider can operate in FIPS 140-2 compliant mode. To do so, perform both of the following:

- Set the TLS vendor to `bcfips` before calling any other EMS methods; refer to the EMS Java documentation for details.
- Start the JVM with `-Dorg.bouncycastle.fips.approved_only=true`

For backward compatibility reasons, an alternative to setting the TLS vendor to `bcfips` consists of setting the `com.tibco.security.FIPS` property to `true`.

If the `tibco.tibjms.ssl.debug.trace` property has been set to `true` and the Java client set to operate in FIPS 140-2 compliant mode, upon initializing the TLS infrastructure the client prints a message listing `BCFIPS` as the FIPS Provider.

- **C Clients**

  To enable FIPS 140-2 operations in the C client, load the required FIPS and base OpenSSL providers before calling any EMS functions. This can be done by setting the `OPENSSL_CONF` and `OPENSSL_MODULES` environment variables, as per the OpenSSL 3.0 documentation.

  For example:

  ```
  export OPENSSL_CONF=/opt/tibco/ems/10.3/lib/openssl-client.cnf
  export OPENSSL_MODULES=/opt/tibco/ems/10.3/lib
  ```

  If `tibemsSSL_SetDebugTrace(TIBEMS_TRUE)` has been called and FIPS 140-2 operations have been successfully enabled, upon establishing a new TLS connection the C client prints a message indicating that `fips` is among the OpenSSL providers that have been loaded.

Note: The Java and C clients support FIPS compliance only on the Linux and Windows platforms.

Fault Tolerance

The following sections describe the fault tolerance features of TIBCO Enterprise Message Service.

Fault Tolerance Overview

You can arrange TIBCO Enterprise Message Service servers for fault-tolerant operation by configuring a pair of servers—one primary and one secondary.

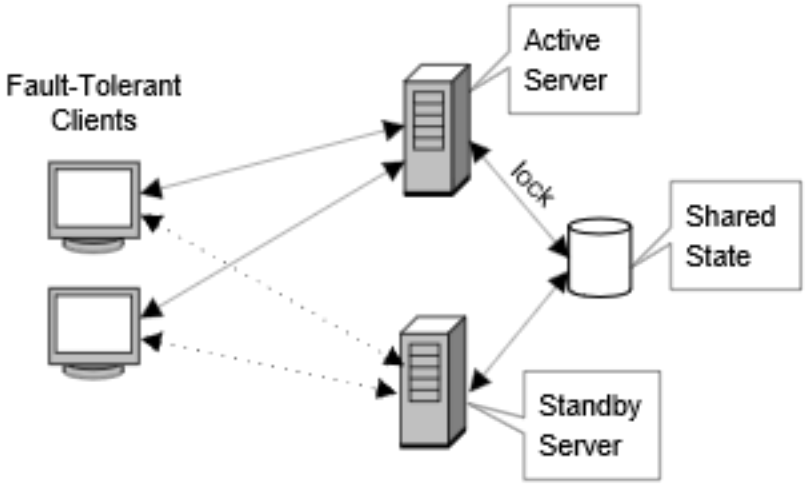
Upon startup, the first server to start reaches the active state and the other the standby state. The active server accepts client connections, and interacts with clients to deliver messages. If the active server fails, the standby server becomes active and resumes operation in its place.

Shared State

A pair of fault-tolerant servers can have access to shared state, which consists of information about clients and persistent messages.

Note: You cannot use more than two servers in a fault-tolerant configuration.

This information enables the standby server to properly assume responsibility for those clients and messages. The following image illustrates a fault-tolerant configuration of EMS.



Locking

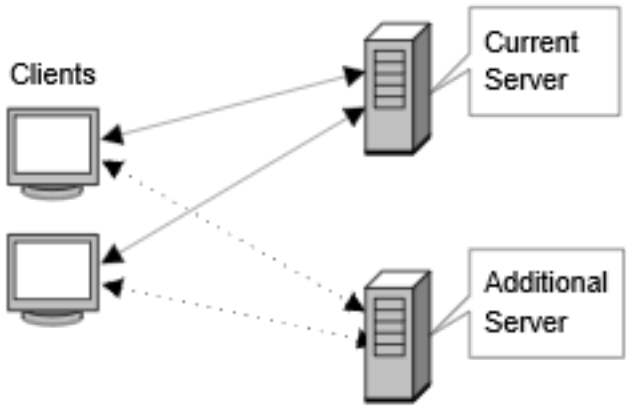
To prevent the standby server from assuming the role of the active server, the active server locks the shared state during normal operation. If the active server fails, the lock is released, and the standby server can obtain the lock and become active.

Unshared State Failover

You can also include additional servers that do not share state. As with shared state, the clients can automatically reconnect to additional servers.

However, unlike shared state, unshared state is controlled by the EMS client. As a result, it is up to client producers to catch failures on send that may occur during an unshared state failover, and to then resend the affected message. As this may lead to duplicate or out-of-order messages, the corresponding client consumers should be equipped to filter out duplicates and re-order messages if dictated by the application requirements.

The following image illustrates an unshared state fault-tolerant configuration of EMS.



Shared State Failover Process

This section presents details of the shared state failover sequence.

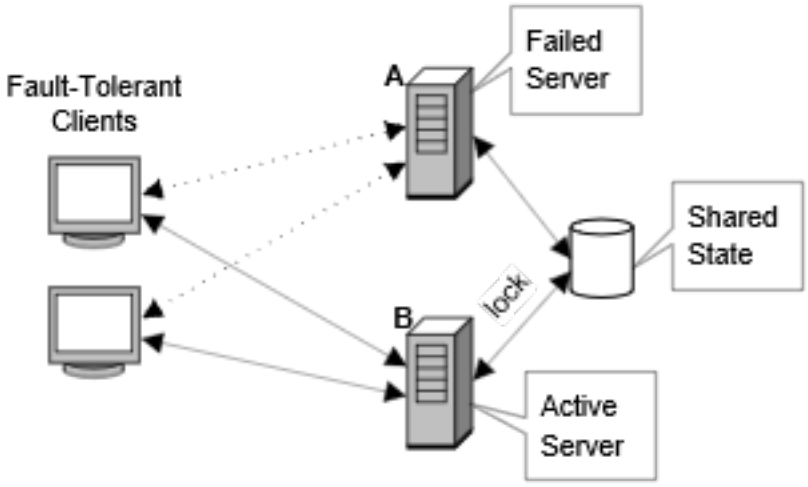
Detection

A standby server detects a failure of the active server in either of the following ways: Heartbeat Failure or Connection Failure.

- *Heartbeat Failure*—The active server sends heartbeat messages to the standby server to indicate that it is still operating. When a network failure stops the servers from communicating with each other, the standby server detects the interruption in the steady stream of heartbeats. For details, see Heartbeat Parameters.
- *Connection Failure*—The standby server can detect the failure of its TCP connection with the active server. When the active server process terminates unexpectedly, the standby server detects the broken connection.

Response

When a standby server (B) detects the failure of the active server (A), then B attempts to assume the role of active server. First, B obtains the lock on the current shared state. When B can access this information, it becomes the new active server.

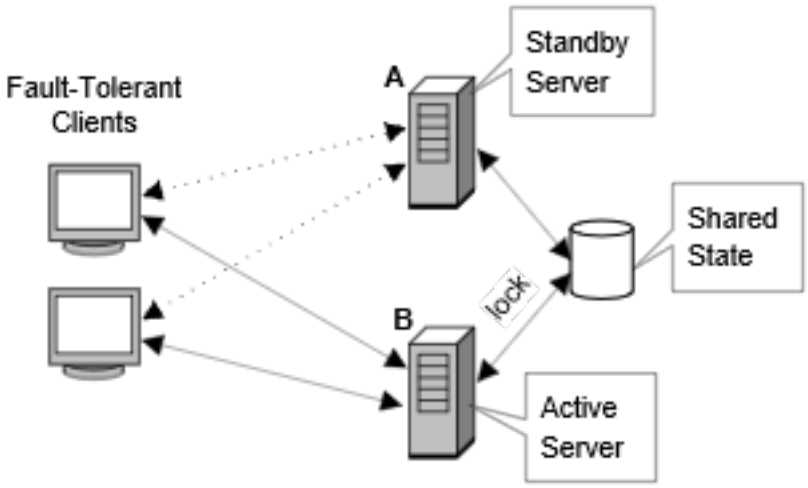


Lock Unavailable

If B cannot obtain the lock immediately, it alternates between attempting to obtain the lock (and become the active server), and attempting to reconnect to A (and resume as a standby server)—until one of these attempts succeeds.

Role Reversal

When B becomes the new active server, A can restart as a standby server, so that the two servers exchange roles.



Client Transfer

Clients of A that are configured to failover to standby server B automatically transfer to B when it becomes the new active server. B reads the client's current state from the shared storage to deliver any persistent messages to the client.

Client Notification

Client applications can receive notification when shared state failover occurs.

- Java

  To receive notification, Java client programs set the system property `tibco.tibjms.ft.switch.exception` to any value, and define an `ExceptionListener` to handle failover notification; see the class `com.tibco.tibjms.Tibjms` in *TIBCO Enterprise Message Service Java API Reference*.

- C

  To receive notification, C client programs call `tibems_setExceptionOnFTSwitch (TIBEMS_TRUE)` and register the exception callback in order to receive the notification that the reconnection was successful.

- C#

  To receive notification, .NET client programs call `Tibems.SetExceptionOnFTSwitch (true)`, and define an exception listener to handle failover notification; see the method `Tibems.SetExceptionOnFTSwitch` in *TIBCO Enterprise Message Service .NET API Reference*.

Message Redelivery

Qualified messages will be redelivered in a failover situation.

- Persistent

  When a failure occurs, messages with delivery mode `PERSISTENT`, that were not successfully acknowledged before the failure, are redelivered.

- Synchronous Mode

  When using durable subscribers, EMS guarantees that a message with `PERSISTENT` delivery mode and written to a `store` with the property `mode=sync`, will not be lost during a failure.

- Delivery Succeeded

  Any messages that have been successfully acknowledged or committed are not redelivered, in compliance with the Jakarta Messaging specification.

- Topics

  All topic subscribers continue normal operation after a failover.

Transactions

A (non-XA) transaction is considered active when at least one message has been sent or received by the session, and the transaction has not been successfully committed. An XA transaction is considered active when the XA start method is called.

After a failover, attempting to commit the active transaction results in a `javax.jms.TransactionRolledBackException`. Clients that use transactions must handle this exception, and resend any messages sent during the transaction. The standby server, upon becoming active, automatically redelivers any messages that were delivered to the session during the transaction that rolled back.

Queues

For queue receivers, any messages that have been sent to receivers, but have not been acknowledged before the failover, may be sent to other receivers immediately after the failover.

A receiver trying to acknowledge a message after a failover may receive the `javax.jms.IllegalStateException`. This exception signifies that the attempted acknowledgment is for a message that has already been sent to another queue receiver. This exception only occurs in this scenario, or when the session or connection have been closed. This exception cannot occur if there is only one receiver at the time of a failover, but it may occur for exclusive queues if more than one receiver was started for that queue.

When a queue receiver catches a `javax.jms.IllegalStateException`, the best course of action is to call the `Session.recover()` method. Your application program should also be prepared to handle redelivery of messages in this situation. All queue messages that can be redelivered to another queue receiver after a failover always have the header field `JMSRedelivered` set to true; application programs must check this header to avoid duplicate processing of the same message in the case of redelivery.

Note: Acknowledged messages are never redelivered (in compliance with the Jakarta Messaging specification). The case described above occurs when the application cannot acknowledge a message because of a failover.

Heartbeat Parameters

When the active server heartbeat stops, the standby server waits for its activation interval (elapsed time since it detected the most recent heartbeat); then the standby server retrieves information from shared storage and assumes the role of active server.

The default heartbeat interval is 3 seconds, and the default activation interval is 10 seconds. The activation interval must be at least twice the heartbeat interval. Both intervals are specified in seconds. You can set these intervals in the server configuration files. See Fault Tolerance Parameters for details.

Note: When using FTL stores, the heartbeat and activation intervals are managed by the FTL server cluster. See Fault-Tolerance with FTL Stores for more details.

Configuration Files

When an active server fails, its standby server assumes the status of the active server and resumes operation. Before becoming the active server, the standby server re-reads its configuration files.

If the two servers share configuration files, then the administrative changes to an active server carry over to its standby once the latter becomes active.

Note: When fault-tolerant servers share configuration files, you must limit configuration changes to the active server only. Separately reconfiguring the standby server can cause it to overwrite the shared configuration files; unintended misconfiguration can result.

Additionally, when a server that is a member of a fault-tolerant pair requires a restart, both servers must be restarted to activate the change. When the active server is shut down, the standby server does not reinitialize its properties (such as listens, heartbeats, timeouts, and so on) or stores during activation. It does reinitialize objects such as queues, topics, factories, routes, and so on.

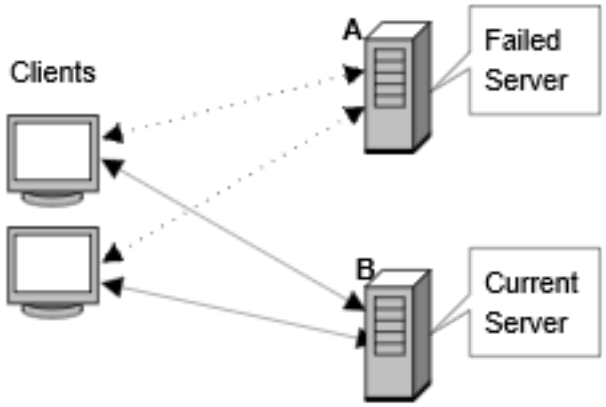
Unshared State Failover Process

The following topics detail the unshared state failover sequence of events.

To configure clients, see Configure Clients for Unshared State Failover Connections.

Detection

Unshared state failover is initiated by the EMS client. When a client setup for unshared state detects a lost connection to server (A), it attempts to connect to server (B), as defined in the connection factory.



Note: Unshared state is not limited to two servers. Unlike shared state failover, unshared state is controlled by the EMS client. The client can include more than two URLs in its list of additional servers.

Response

Clients with unshared state connections automatically connect to B after losing the connection to A.

When clients setup for unshared state detect lost connections to server A, they create new connections to server B. All runtime objects from the client's connection are recreated, including sessions, destinations, message producers, and message consumers.

Because unshared state is defined in the connection factory, B remains the current server as long as the connection is active. If the connection to B is lost, clients attempt to connect to another server defined in the connection factory

Message Loss

Because B does not have access to persistent messages that were not delivered or acknowledged prior to the failover, some messages may be lost or delivered out of order across the failover process. To prevent message loss, use shared state failover.

Unsupported Features

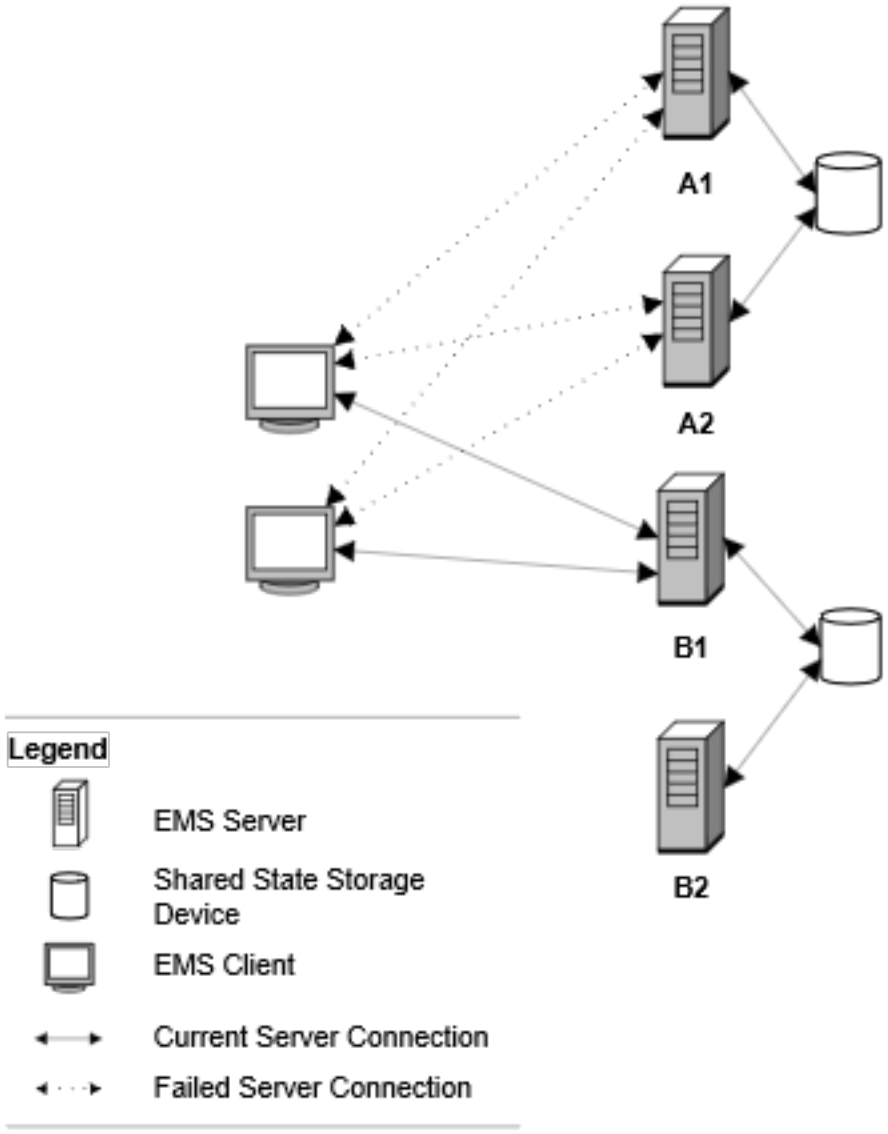
These features and Java classes are not supported with unshared state connections:

- XA transactions
- Durable topic subscribers
- ConnectionConsumer
- ServerSession
- ServerSessionPool
- QueueRequestor
- TopicRequestor

Dual State Failover

An unshared state connection factory can include shared-state server pairs in its list of backup servers. When both shared state and unshared state servers are included, the failover process is a combination of both types of failover.

The following image illustrates the dual state failover process.



In this example, servers A1 and A2 share state. Servers B1 and B2 also share state. However, A1 and A2 do not share state with B1 and B2.

The EMS clients created connections using a connection factory with A1, A2 + B1, B2. The initial server connections were with server A1. When the connection to A1 failed, the failover process proceeded as described in Shared State Failover Process, and the clients connect to A2.

A2 then failed, before A1 restarted. The clients next created connections to B1, recreating all runtime objects from the connection (as described above in Unshared State Failover

Process). B1 is now the active server. Because B1 and B2 share state, If B1 fails, B2 becomes the active server.

Shared State

For the most robust failover protection, the active server and standby server must share the same state.

Shared state includes the following categories of information:

- persistent message data (for queues and topics)
- client connections of the active server
- metadata about message delivery

During a failover, the standby server re-reads all shared state information.

Implement Shared State

If using file-based stores, we recommend that you implement shared state using shared storage devices. The shared state must be accessible to both the active and standby servers. If using grid stores or FTL stores, shared state is provided by the ActiveSpaces or FTL deployment.

Support Criteria

If your stores are file-based, there are several options available for implementing shared storage, using a combination of hardware and software. EMS requires that your storage solution guarantees the following listed criteria.

> ⚠ **Warning:** *Always* consult your shared storage vendor and your operating system vendor to ascertain that the storage solution you select satisfies all four criteria.

Criterion	Description
Write Order	The storage solution must write data blocks to shared storage in the same order as they occur in the data buffer. (Solutions that write data blocks in any other order (for example, to enhance disk efficiency) do *not* satisfy this requirement.)
Synchronous Write Persistence	Upon return from a synchronous write call, the storage solution guarantees that all the data have been written to durable, persistent storage.
Distributed File Locking	The EMS servers must be able to request and obtain an exclusive lock on the shared storage. The storage solution must *not* assign the locks to two servers simultaneously. (See Software Options.) EMS servers use this lock to determine the primary server.
Unique Write Ownership	The EMS server process that has the file lock must be the only server process that can write to the file. Once the system transfers the lock to another server, pending writes queued by the previous owner must fail.

Hardware Options

Consider these examples of commonly-sold hardware options for shared storage:

- Dual-Port SCSI device
- Storage Area Network (SAN)
- Network Attached Storage (NAS)

SCSI and SAN

Dual-port SCSI and SAN solutions generally satisfy the Write Order and Synchronous Write Persistence criteria. (The clustering software must satisfy the remaining two criteria.) As always, you must confirm all four requirements with your vendors.

NAS

NAS solutions require a CS (rather than a CFS) to satisfy the Distributed File Locking criterion (see below).

Some NAS solutions satisfy the criteria, and some do not; you must confirm all four requirements with your vendors.

NAS with NFS

When NAS hardware uses NFS as its file system, it is particularly difficult to determine whether the solution meets the criteria. Our research indicates the following conclusions:

- NFS v2 and NFS v3 definitely do *not* satisfy the criteria.
- NFS v4 with TCP *might* satisfy the criteria. Consult with the NAS vendor to verify that the NFS server (in the NAS) satisfies the criteria. Consult with the operating system vendor to verify that the NFS client (in the OS on the server host computer) satisfies the criteria. When both vendors certify that their components cooperate to guarantee the criteria, then the shared storage solution supports EMS.
- NFS over UDP is not supported under any circumstances.

For more information on how the EMS locks shared store files, see Managing Access to Shared File-Based Stores.

Software Options

Consider these examples of commonly-sold software options:

- Cluster Server (CS)

  A cluster server monitors the EMS server processes and their host computers, and ensures that exactly one server process is running at all times. If that server fails, the CS restarts it; if the CS fails to restart it, it starts the other server instead.

- Clustered File System (CFS)

  A clustered file system lets the two EMS server processes run simultaneously. It even lets both servers mount the shared file system simultaneously. However, the CFS assigns the lock to only one server process at a time. The CFS also manages operating system caching of file data, so the standby server has an up-to-date view of

the file system (instead of a stale cache).

With dual-port SCSI or SAN hardware, either a CS or a CFS might satisfy the Distributed File Locking criterion. With NAS hardware, only a CS can satisfy this criterion (CFS software generally does not). Of course, you must confirm all four requirements with your vendors.

Messages Stored in Shared State

Messages with `PERSISTENT` delivery mode are stored, and are available in the event of active server failure. Messages with `NON_PERSISTENT` delivery mode are not available if the active server fails.

For more information about recovery of messages during failover, see Message Redelivery.

Shared State Storage

By default, the `tibemsd` server creates three stores to store shared state.

- `$sys.failsafe`—This store holds persistent messages using synchronous I/O calls.
- `$sys.nonfailsafe`—This store stores messages using asynchronous I/O calls.
- `$sys.meta`—This store holds state information about durable subscribers, fault-tolerant connections, and other metadata.

These stores are fully customizable through parameters in the stores configuration file. More information about these stores and the default configuration settings are fully described in `stores.conf`.

When using file-based stores, to prevent two servers from using the same store file, each server restricts access to its store file for the duration of the server process. For more information on how the EMS manages shared store files, see Managing Access to Shared File-Based Stores. When using grid stores or FTL stores, this store exclusivity is handled by ActiveSpaces and FTL respectively.

Note: These default stores can be changed or modified. See Default Stores for more information.

Configure Fault-Tolerant Servers

The following topics describe how to configure fault-tolerant servers, for the shared state and the unshared state scenarios.

Shared State

To configure an EMS server as a fault-tolerant secondary when using file-based stores or grid stores, set the below parameters in its main configuration file (or on the server command line). The primary and secondary roles cannot be explicitly defined when using FTL stores.

- `server` Set this parameter to the same server name in the configuration files of both the primary server and the secondary server.
- `ft_active` In the configuration file of the primary server, set this parameter to the URL of the secondary server. In the configuration file of the secondary server, set this parameter to the URL of the active server.

When a server configured for fault tolerance starts, it attempts to connect to its peer server. If it establishes a connection to its peer, then it enters the standby state. If it cannot establish a connection to its peer, then it becomes the active server.

While a server is in the standby state, it does not accept connections from clients. To administer the standby server, the admin user can connect to it using the administration tool. Standby servers started with a JSON configuration file cannot be administered.

Authentication and Authorization for Fault-Tolerant Servers

> **Note:** Authentication and authorization for fault-tolerance is configured through FTL when the EMS servers are using FTL stores. See Configuring and Deploying FTL Stores for details.

EMS authentication and authorization interact with fault tolerance. If `authorization` is enabled and the two EMS Servers are configured for fault tolerance, then both servers in a fault-tolerant pair must be configured. The exact configuration requirements vary depending on the authentication method to be used.

User & Password

When using user credential based authentication (see the `local` or `jaas` Authentication Methods), a server will authenticate its fault-tolerant peer server by validating the user name and password provided as part of the connection request from the peer server.

This following files must be updated to perform `local` authentication:

- The `tibemsd.conf` file for each server must have the same server name and password (the server and password parameters must be the same on each server).
- The user name and password in the `users.conf` file for each server must match the values of the server and password parameters in the `tibemsd.conf` file.

> **Note:** If the two EMS Servers are not sharing a `users.conf` file, make sure that you create a user with the same name as the EMS Server, and set the user's password with the value of the "server" password.

For example, you have two EMS Servers (Server 1 and Server 2) that are named "EMS-SERVER" and are to use a password of "mySecret", but which do not share a `users.conf` file. To set the user names and passwords, start the EMS Administration Tool on each server, as described in EMS Administration Tool, and do the following.

From the active (Server 1), enter:

```
set server password=mySecret
create user EMS-SERVER password=mySecret
```

From the standby (Server 2), enter:

```
set server password=mySecret
create user EMS-SERVER password=mySecret
```

From the active (Server 1), enter:

```
set server authorization=enabled
```

From the standby (Server 2), enter:

```
set server authorization=enabled
```

For `jaas` authentication, the `tibemsd.conf` changes are still required. However, a JAAS authentication module must be configured in lieu of the `users.conf` changes. See JAAS Authentication Modules for details.

OAuth 2.0

When using `oauth2` authentication (see Authentication Methods) the EMS server will authenticate its fault-tolerant peer server by validating an OAuth 2.0 access token received as part of the peer server's connection request.

For OAuth 2.0 authentication to be successful between fault-tolerant peer servers, both servers will need to be configured to authenticate incoming connections via OAuth 2.0. Both servers will also need to be configured to procure OAuth 2.0 access tokens and include them in their connection requests to each other.

Follow the instructions in the Authentication Using OAuth 2.0 section to enable authentication of incoming connection requests in both servers.

The `ft_oauth2_server_url`, `ft_oauth2_client_id` and `ft_oauth2_client_secret` parameters must also be configured in both servers in order for them to obtain the OAuth 2.0 access tokens required to authenticate each other. Refer to Fault Tolerance Parameters for the full list of available parameters relating to OAuth 2.0 and fault-tolerance.

TLS

> **Note:** TLS security for fault-tolerance related communication is configured through FTL when using FTL stores. See Configuring and Deploying FTL Stores for details.

You can use TLS to secure communication between a pair of fault-tolerant servers.

Parameters in the main configuration file (tibemsd.conf) affect this behavior. The relevant parameters all begin with the prefix ft_ssl.

The server initializing a secure connection to another server uses the `ft_ssl` parameters to determine the properties of its secure connection to the other server. The receiving server validates the incoming connection against its own `ssl_` parameters. For more information about `ft_ssl` parameters, see Fault Tolerance Parameters. For more information about `ssl_` parameters, see TLS Server Parameters.

Also see TLS Protocol.

Reconnect Timeout

When a standby server assumes the role of the active server during failover, clients attempt to reconnect to the standby server (that is, the new active) and continue processing their current message state. Before accepting reconnects from the clients, the new active server reads its message state from the shared state files.

You can instruct the server to clean up state information for clients that do not reconnect before the time limit specified by the `ft_reconnect_timeout` configuration parameter. The `ft_reconnect_timeout` time starts once the server has fully recovered the shared state, so this value does not account for the time it takes to recover the store files. See `ft_reconnect_timeout` for details.

Unshared State

When configuring a fault tolerant pair that does not share state, you must ensure that both servers use identical configurations.

This is especially important for these configuration settings:

- **Destinations**

  Both servers must support the same destinations.

- **Routes**

  Messages must be able to arrive at the endpoints, using equivalent or identical routes across servers.

- **Access Control**

  Access control must be setup identically in both servers, so that the `users.conf`, `groups.conf`, and `acl.conf` file settings match.

- **TLS**

  When TLS is deployed, both servers must use the same certificate(s).

Fault Tolerance with a JSON Configuration

Note: This section is only applicable when using file-based stores or grid stores. To configure fault-tolerance when using FTL stores, see the Fault-Tolerance with FTL Stores and Configuring and Deploying FTL Stores sections.

When using a JSON configuration, the same JSON file is used to manage both servers in a fault tolerant pair. Primary and secondary server roles are determined when the servers are started.

All but two configuration settings are shared by both EMS servers: the `listen` and `ft_active` parameters are configured separately.

- The primary server, if elected active, listens for client connections on ports defined in the "primary_listens" node of the configuration. If elected standby, it listens for the secondary server on the URL that is flagged using the "ft_active" Boolean within the "secondary_listens" node.
- Conversely, the secondary server, if elected standby, listens for the primary server on the URL that is flagged using the "ft_active" Boolean within the "primary_listens" node. If elected active, it listens for client connections on ports defined in the "secondary_listens" node.

Configuring Fault Tolerance

To configure a fault tolerant server pair using a JSON configuration, refer to the TIBCO Messaging Manager documentation.

Configuration Errors

When an EMS server is started, the fault tolerance mechanism is triggered by the presence of a URL in the "secondary_listens" node of a primary tibemsd, or by that of a URL in the "primary_listens" node of a secondary `tibemsd`.

Once fault tolerance is triggered, the EMS server generates an error if it finds that the "ft_active" Boolean was not set for any URL in its peer's node. If `CONFIG_ERRORS` is present in the `startup_abort_list` parameter, the `tibemsd` aborts startup. Otherwise, the `tibemsd`

cancels fault tolerance and starts without checking its peer. This results in a file lock error for the EMS server that is started second.

Configure Clients for Shared State Failover Connections

When a failover occurs and the standby server takes the active state, clients attempt to reconnect to this server (that is, the new active server). To enable a client to reconnect, you must specify the URLs of both servers when creating a connection.

Specify multiple servers as a comma-separated list of URLs. Both URLs must use the same protocol (either `tcp` or `ssl`). For example, to identify the first server as `tcp://server0:7222`, and the second server as `tcp://server1:7344` (if first server is not available), you can specify:

```
serverUrl=tcp://server0:7222, tcp://server1:7344
```

The client attempts to connect to each URL in the order listed. If a connection to one URL fails, the client tries the next URL in the list. The client tries the URLs in sequence until all URLs have been tried. If the first failed connection was not the first URL in the list, the attempts wrap to the start of the list (so each URL is tried).

For information on how to lookup a fault-tolerance URL in the EMS naming service, see Perform Fault-Tolerant Lookups.

Note: The reconnection logic in the client is triggered by the specifying multiple URLs when connecting to a server. If no secondary server is present, the client must still provide at least two URLs (typically pointing to the same server) in order for it to automatically reconnect to the server when it becomes available after a failure.

Note: When messages are sent in non-persistent or reliable modes, the consumer does not normally wait for a server reply to its acknowledgements. However, a fault tolerant consumer does wait for a server reply (when using a session mode other than `DUPS_OK_ACKNOWLEDGE` or `EXPLICIT_CLIENT_DUPS_OK_ACKNOWLEDGE`). This is true for shared state configurations. Unshared state configurations, which tolerate lost, duplicated, and out-of-order messages during a failover, do not wait for server acknowledgements.

Specify More Than Two URLs

Even though there are only two servers (the primary and secondary servers), clients can specify more than two URLs for the connection.

For example, if each server has more than one listen address, a client can reconnect to the same server at a different address (that is, at a different network interface).

Set Reconnection Failure Parameters

EMS allows you to establish separate parameters for initial connection attempts and reconnection attempts.

How to set the initial connection attempt parameters is described in Set Connection Attempts, Timeout and Delay Parameters. This section describes the parameters you can establish for reconnection attempts following a fault-tolerant failover.

The reason for having separate connect and reconnect attempt parameters is that there is a limit imposed by the operating system to the number of connection attempts the EMS server can handle at any particular time. (For example, in UNIX, this limit is adjusted by the `ulimit` setting.) Under normal circumstances, each connect attempt is distributed so it is less likely for the server to exceed its maximum accept queue. However, during a fault-tolerant failover, all of the clients automatically try to reconnect to the new active server at approximately the same time. When the number of connections is large, it may require more time for each client to reconnect than for the initial connect.

By default, a client will attempt reconnection 4 times with a 500 ms delay between each attempt. You can modify these settings in the `factories.conf` file or by means of your client connection factory API, as demonstrated by the examples in this section.

The following examples establish a reconnection count of 10, a delay of 1000 ms and a timeout of 1000 ms.

- Java

  Use the `TibjmsConnectionFactory` object's `setReconnAttemptCount()`, `setReconnAttemptDelay()`, and `setReconnAttemptTimeout()` methods to establish new reconnection failure parameters.

  ```
  factory.setReconnAttemptCount(10);
  factory.setReconnAttemptDelay(1000);
  factory.setReconnAttemptTimeout(1000);
  ```

- C

  Use the `tibemsConnectionFactory_SetReconnectAttemptCount`, `tibemsConnectionFactory_SetReconnectAttemptDelay`, and `tibemsConnectionFactory_SetReconnectAttemptTimeout` functions to establish new reconnection failure parameters.

  ```
  status = tibemsConnectionFactory_SetReconnectAttemptCount(factory, 10);
  status = tibemsConnectionFactory_SetReconnectAttemptDelay(factory, 1000);
  status = tibemsConnectionFactory_SetReconnectAttemptTimeout(factory, 1000);
  ```

- C#

  Use the `ConnectionFactory.SetReconnAttemptCount`, `ConnectionFactory.SetReconnAttemptDelay`, and `ConnectionFactory.SetReconnAttemptTimeout` methods to establish new reconnection failure parameters:

  ```
  factory.setReconnAttemptCount(10);
  factory.setReconnAttemptDelay(1000);
  factory.setReconnAttemptTimeout(1000);
  ```

Configure Clients for Unshared State Failover Connections

> **Warning:** Unshared state failover is an extension of the Jakarta Messaging specification. Because state is not shared among servers, messages can be lost, duplicated, or delivered out-of-order across the failover process.

Unshared state connections are created differently from shared state connections in several important ways.

- For Java applications, a JAR file must be present in the environment `CLASSPATH` of the client.
- For C applications, a header file must be included and clients must link using the unshared state library.
- The connection must be created using an unshared state connection factory.
- The server URLs must be specified using unshared state syntax.

Include the Unshared State Library

- Java Applications

  Before creating the connection factory, ensure that the `CLASSPATH` includes the JAR file: `tibjmsufo.jar`

- C Applications

  Include the `tibemsufo.h` header file.

- C# Applications

  Include the `TIBCO.EMS.UFO.dll` file.

Create an Unshared State Connection Factory

To create unshared state connections, use the relevant methods:

- Java Applications

`java com.tibco.tibems.ufo` package.

- C Applications

  `tibemsufo` library and functions.

- C# Applications

  `TIBCO.EMS.UFO` package.

Methods called inside a MessageListener callback immediately return an EMSException indicating the connection has been terminated.

Connection Recovery

When an unshared state connection fails, the connection's `ExceptionListener` callback is invoked. To recover the connection—repair it so that it is connected to an active server—the client application calls the connection factory's `recoverConnection` method or `tibemsUFOConnectionFactory_RecoverConnection` function.

This must be performed in the `ExceptionListener` callback. The recover connection method blocks until the connection (and its related objects, including sessions, producers, and consumers) are fully recreated, or until it has failed in all its attempts to recreate these objects.

As long as the unshared state client has a valid connection, the API behaves the same as the standard EMS client. However, when the unshared state client's connection is broken, the API performs as follows:

1. Methods called inside a `MessageListener` callback immediately return a Java exception `ConnectionFailureException` or C status of `TIBEMS_SERVER_NOT_CONNECTED`.
2. Methods called elsewhere block until the connection is valid again.

Note that the connection is considered broken from the point where the underlying TCP/TLS connection fails, and until `recoverConnection` or `tibemsConnectionFactory_RecoverConnection` successfully returns.

Specify Server URLs

When a server connection is lost during an unshared state failover, clients attempt to reconnect to the second server. To enable a client to reconnect, you must specify the URLs

of both servers when creating a connection.

- **Unshared State**

  Specify multiple servers as a list of URLs separated by plus (+) signs. For example, to identify the first server as `tcp://server0:7222`, and the second server as `tcp://server1:7344`, you can specify:

  ```
  serverUrl=tcp://server0:7222+tcp://server1:7344
  ```

- **Dual State**

  To combine shared state server pairs with unshared state servers, use commas to separate the servers that share state, and plus (+) signs to separate servers that do not share state. For example, this line specifies server a1 and a2 as a fault-tolerant pair that share state, and servers `b1` and `b2` as a second pair with shared state:

  ```
  serverUrl=tcp://a1:8222,tcp://a2:8222+tcp://b1:8222,tcp://b2:8222
  ```

  Note that `a1` and `a2` do not share state with `b1` and `b2`.

The client attempts to connect to each URL in the order listed. If a connection to one URL fails, the client tries the next URL in the list. The client tries the URLs in sequence until all URLs have been tried. If the first failed connection was not the first URL in the list, the attempts wrap to the start of the list (so each URL is tried). If none of the attempts succeed, the connection fails.

Note: Server lookup functions do not permit unshared state syntax. That is, you cannot separate server URLs using the plus (+) symbol during a server lookup.

Set Connect Attempt and Reconnect Attempt Behavior

The effect of setting connect attempt and reconnect attempt properties at the application level is different when applied to unshared state connection factories.

If the EMS client is using a shared state connection factory, then the values specified by way of properties or API calls will be the values used during client connect and reconnect sequences. However, if the client is using an unshared state factory, then the application

layer values do not directly override the `connect_attempt_count` and `reconnect_attempt_count` properties set in the unshared state connection factory. Instead, the value specified at the application level is multiplied by the value in the connection factory to determine the resulting count. Also if the `connect_attempt_delay` and/or `reconnect_attempt_delay` are overridden at the application layer, the resulting actual delays can vary significantly from the override value.

For example, if the unshared state connection factory has a `connect_attempt_count` value of 5 and the Java system property `com.tibco.tibjms.connect.attempts` is set to 3 for the Java client, then the effective `connect_attempt_count` will be 15.

See Also

The connection factory connect attempt and reconnect attempt properties are documented in `factories.conf`.

The sections Set Connection Attempts, Timeout and Delay Parameters and Set Reconnection Failure Parameters describe the use of these settings.

Routes

The following sections describe routing of messages among TIBCO Enterprise Message Service servers.

Overview

TIBCO Enterprise Message Service servers can route messages to other servers.

- Topic messages can travel one hop or multiple hops (from the first server).
- Queue messages can travel only one hop to the home queue, and one hop from the home queue.

You can define routes using an administrative interface (that is, configuration files, `tibemsadmin`, or administration APIs).

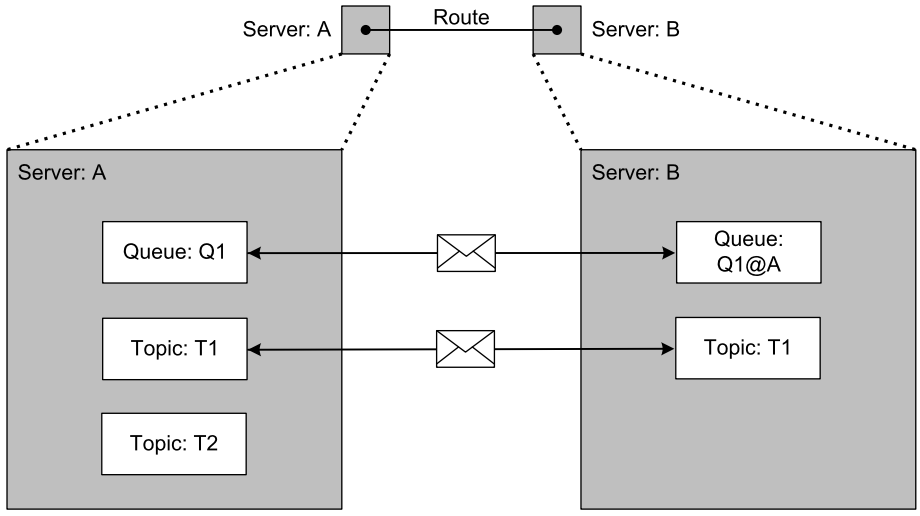
Route

Basic Operation

- Each *route* connects two TIBCO Enterprise Message Service servers.
- Each route forwards messages between corresponding destinations (that is, global topics with the same name, or explicitly routed queues) on its two servers.
- Routes are bidirectional; that is, each server in the pair forwards messages along the route to the other server.

For example, the compact view at the top of the following image denotes a route between two servers, A and B. The exploded view beneath it illustrates the behavior of the route. Each server has a global topic named T1, and a routed queue Q1; these destinations correspond, so the route forwards messages between them. In addition, server A has a

global topic T2, which does not correspond to any topic on server B. The route does not forward messages from T2.

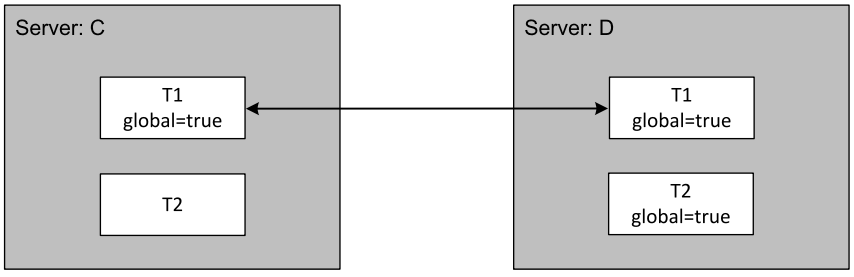


Global Destinations

Routes forward messages only between global destinations—that is, for topics the `global` property must be set on both servers (for queues, see Routed Queues).

For more information about destination properties, See Destination Properties.

The following image illustrates a route between two servers, C and D, with corresponding destinations T1 and T2. Notice that T1 is global on both C and D, so both servers forward messages across the route to the corresponding destination. However, T2 is not global on C, neither C nor D forward T2 messages to one another.



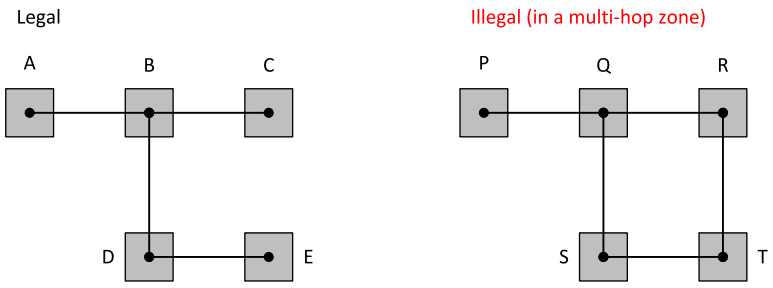
Unique Routing Path

It is illegal to define a set of routes that permit a message to reach a server by more than one path. TIBCO Enterprise Message Service servers detect illegal duplicate routes and report them as configuration errors.

The following image depicts two sets of routes. On the left, the routes connecting servers A, B, C, D and E form an acyclic graph, with only one route connecting any pair of servers; this configuration is legal (in any zone).

In contrast, the routing configuration on the right is illegal in a multi-hop zone. The graph contains redundant routing paths between servers Q and S (one direct, and one through R and T).

Note: Note that the configuration on the right is illegal only in a multi-hop zone; it is legal in a one-hop zone. For further information, see Zone.



Zone

Zones restrict the behavior of routes, so you can configure complex routing paths. Zones affect topic messages, but not queue messages.

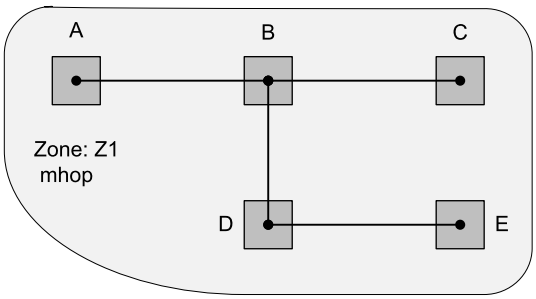
Basic Operation

A *zone* is a named set of routes. Every route belongs to a zone.

A zone affects the forwarding behavior of its routes:

- In a multi-hop (`mhop`) zone, topic messages travel along all applicable routes to all servers connected by routes within the zone.
- In a one-hop (`1hop`) zone, topic messages travel only one hop (from the first server).
- Queue messages travel only one hop, even within multi-hop zones.

For example, the following figure depicts a set of servers connected by routes within a multi-hop zone, Z1. If a client sends a message to a global topic on server B, the servers forward the message to A, C, D and E (assuming there are subscribers at each of those servers). In contrast, if Z1 were a one-hop zone, B would forward the message to A, C and D—but D would *not* forward it E.

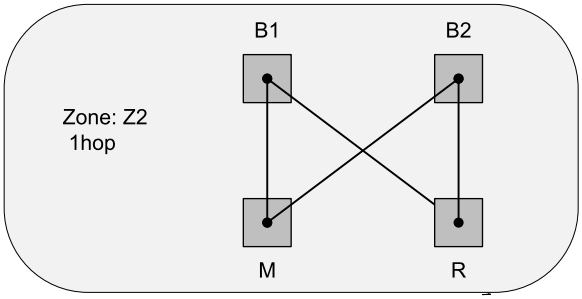


Eliminate Redundant Paths with a One-Hop Zone

The following image illustrates an enterprise with four servers:

- B1 and B2 serve producers at branch offices of an enterprise.
- M serves consumers at the main office, which process the messages from the branches.
- R serves consumers that record messages for archiving, auditing, and backup.

The goal is to forward messages from B1 and B2 to both M and R. The routing graph *seems* to contain a cycle—the path from B1 to M to B2 to R duplicates the route from B1 to R. However, since these routes belong to the one-hop zone Z2, it is impossible for messages to travel the longer path. Instead, this limitation results in the desired result—forwarding from B1 to M and R, and from B2 to M and R.

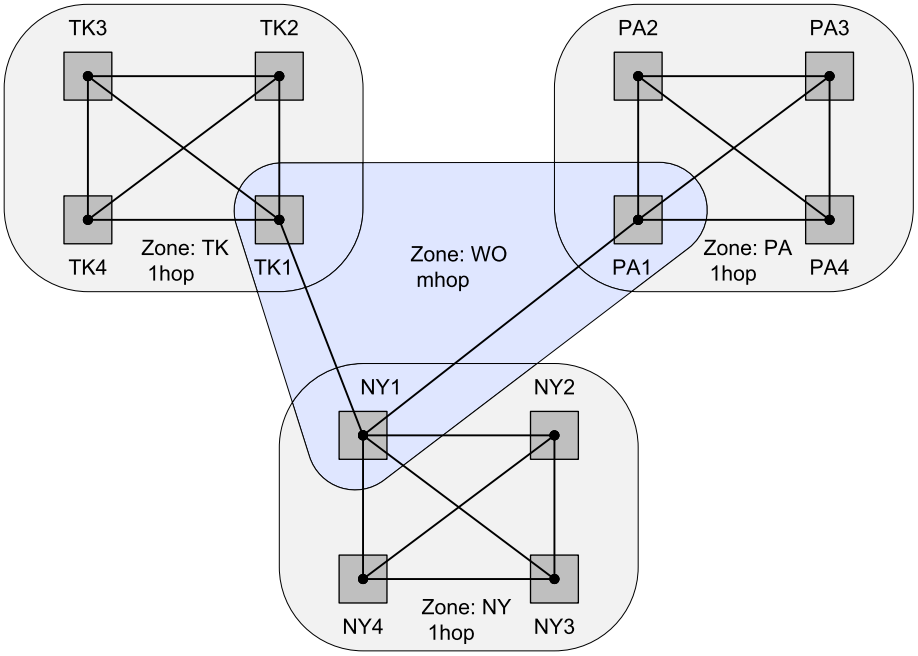


Overlapping Zones

A server can have routes that belong to several zones. When zones overlap at a server, the routing behavior within each zone does not limit routing in other zones. That is, when a forwarded message reaches a server with routes in several zones, the message crosses zone boundaries, and its hop count is reset to zero.

The following image illustrates an enterprise with one-hop zones connecting all the servers in each of several cities in a fully-connected graph. Zone TK connects all the servers in Tokyo; zone NY connects all the servers in New York; zone PA connects all the servers in Paris. In addition, the multi-hop zone WO connects one server in each city.

When a client of server TK3 produces a message, it travels one hop to each of the other Tokyo servers. When the message reaches TK1, it crosses into zone WO. TK1 forwards the message to NY1, which in turn forwards it to PA1. When the message reaches NY1, it crosses into zone NY (with hop count reset to zero); NY1 forwards it one hop to each of the other New York servers. Similarly, when the message reaches PA1, it crosses into zone PA (with hop count reset to zero); PA1 forwards it one hop to each of the other Paris servers.



Active and Passive Routes

A route connects two servers. You may configure a route at either or both of the servers.

Active-Passive Routes

When you configure a route at only one server, this asymmetry results in different perspectives on the route.

- A route is *active* from the perspective of the server where it is configured. This server actively initiates the connection to the other server, so we refer to it as the *active server*, or *initiating server*.
- A route is *passive* from the perspective of the other server. This server passively accepts connection requests from the active server, so we refer to it as the *passive server*.

A server can have both active and passive routes. That is, you can configure server S to initiate routes, and also configure other servers to initiate routes to S.

You can specify and modify the properties of an active route, but not those of a passive route. That is, properties of routes are associated with the server where the route is configured, and which initiates the connection.

Note: Defining a route specifies a zone as well (either implicitly or explicitly). The first route in the zone defines the type of the route; subsequent routes in the same zone must have the same zone type (otherwise, the server reports an error).

Active-Active Routes

Two servers can both configure an active route one to the other. This arrangement is called an *active-active configuration*.

For example, server A specifies a route to server B, and B specifies a route to A. Either server can attempt to initiate the connection. This configuration results in only one connection; it does not result in redundant routes.

You can promote an *active-passive* route to an *active-active* route. To promote a route, use this command on the passive server:

```
create route name url=url
```

The *url* argument is required, so that the server (where the route is being promoted) can connect to the other server if the route becomes disconnected. See `create route` for more information.

The promoted route behaves as a statically configured route—that is, it persists messages for durable subscribers, and stores its configuration in `routes.conf`, and administrators can modify its properties.

Configure Routes and Zones

You can create routes using the administration tool, or the administration APIs (see `com.tibco.tibjms.admin.RouteInfo` in the online documentation).

Syntax

To create a route using the administration tool, first connect to one of the servers, then use the `create route` command with the following syntax:

```
create route name url=URL zone_name=zone_name zone_type=1hop|mhop properties
```

- *name* is the name of the server at the other end of the route; it also becomes the name of the route.
- *URL* specifies the other server by its URL—including protocol and port.

  If your environment is configured for fault tolerance, the URL can be a comma-separated list of URLs denoting fault-tolerant servers. For more information about fault tolerance, see Fault Tolerance.
- *zone_name* specifies that the route belongs to the routing zone with this name. When absent, the default value is `default_mhop_zone` (this default yields backward compatibility with configurations from releases earlier than 4.0).
- The zone type is either `1hop` or `mhop`. When omitted, the default value is `mhop`.
- *properties* is a space-separated list of properties for the route. Each property has the syntax:

  ```
  prop_name=value
  ```

  For gating properties that control the flow of topics along the route, see Selectors for Routing Topic Messages.

  For properties that configure the Transport Layer Security (TLS) protocol for the route, see Routing and TLS.

  For properties that configure OAuth 2.0 authentication for the route, see Authentication.

Example

For example, these commands on server A would create routes to servers B and C. The route to B belongs to the one-hop zone `Z1`. The route to C belongs to the multi-hop zone `ZM`.

```
create route B url=tcp://B:7454 zone_name=Z1 zone_type=1hop
create route C url=tcp://C:7455 zone_name=ZM zone_type=mhop
```

Show Routes

You can display these routes using the `show routes` command in the administration tool:

```
show   routes
Route    T    ConnID   URL             Zone      T
B        A     3       tcp://B:7454    Z1        1
C        A      -      tcp://C:7455    ZM        m
```

- The Route column lists the name of the passive server.
- The `T` column indicates whether the route is active (`A`) or passive (`P`), from the perspective of server A.
- The `ConnID` column contains either an integer connection ID if the route is currently connected, or a dash (`-`) if the route is not connected.

Routes to Fault-Tolerant Servers

You can configure servers for fault tolerance. Client applications can specify the primary and secondary servers.

Once a client has connected to the active server, if its connection to the server fails, the client can connect to the standby server and resume operation. Similarly, a route specification can specify primary and secondary passive servers, so that if the route to the active-state server fails, the active-route server can connect to the standby-state server and resume routing.

Failover behavior for route connections is similar to that for client connections; see Configure Clients for Shared State Failover Connections.

Example

```
create route B url=tcp://B:7454,tcp://BBackup:7454 zone_name=Z1
zone_type=1hop
```

Routing and TLS

When configuring a route, you can specify TLS parameters for the connection. Although both participants in a TLS connection must specify a similar set of parameters, each server specifies this information in a different place.

- The passive server must specify TLS parameters in its main configuration file, `tibemsd.conf`.
- When an active server initiates a TLS connection, it sends the route's TLS parameters to identify and authenticate itself to the passive server. You can specify these parameters when creating the route, or you can specify them in the route configuration file, `routes.conf`.

You can configure the server to require a digital certificate only for TLS connections coming from routes, while not requiring such a certificate for TLS connections coming from clients or from its fault-tolerant peer.

For more information, see `ssl_require_route_cert_only`.

TLS Parameters for Routes

The following table lists parameters that you can specify in the `routes.conf` configuration file, or on the command line when creating a route. The parameters for configuring TLS between routed servers are similar to the parameters used to configure TLS between server and clients; see TLS Protocol.

Parameter	Description
`ssl_identity`	The server's digital certificate in PEM, DER, or PKCS#12 format. You can copy the digital certificate into the specification for this parameter, or you can specify the path to a file that contains the certificate in one of the supported formats. A DER format file can only contain the certificate; it cannot contain both the certificate and a private key. For more information, see File Names for Certificates and Keys.

Parameter	Description

`ssl_issuer`	Certificate chain member for the server. Supply the entire chain, including the CA root certificate. The server reads the certificates in the chain in the order they are presented in this parameter.
	
	The certificates must be in PEM, DER, PKCS#7 or PKCS#12 format. A DER format file can only contain a single certificate; it cannot contain a certificate chain.
	
	Example
	
	`ssl_issuer = certs\CA_root.pem`
	`ssl_issuer = certs\CA_child1.pem`
	`ssl_issuer = certs\CA_child2.pem`
	
	For more information, see File Names for Certificates and Keys.

`ssl_private_key`	The local server's private key. If the digital certificate in `ssl_identity` already includes this information, then you may omit this parameter. This parameter accepts private keys in PEM, DER and PKCS#12 formats. You can specify the actual key in this parameter, or you can specify a path to a file that contains the key. For more information, see File Names for Certificates and Keys.
`ssl_password`	Private key or password for private keys. You can set passwords using the `tibemsadmin` tool. When passwords are set with this tool, the password is obfuscated in the configuration file. For more information, see Using the EMS Administration Tool.
`ssl_trusted`	List of certificates that identify trusted certificate authorities.

Parameter	Description
	The certificates must be in PEM, DER or PKCS#7 format. You can either provide the actual certificates, or you can specify a path to a file containing the certificate chain. If using a DER format file, it can contain only a single certificate, not a certificate chain. For more information, see File Names for Certificates and Keys.
`ssl_verify_host`	Specifies whether the server must verify the other server's certificate. The values for this parameter are `enabled` and disabled. When omitted, the default is `enabled`, signifying the server must verify the other server's certificate. When this parameter is `disabled`, the server establishes secure communication with the other server, but does not verify the server's identity.
`ssl_verify_hostname`	Specifies whether the server must verify the name in the CN field of the other server's certificate. The values for this parameter are enabled and disabled. When omitted, the default is enabled, signifying the server must verify the name of the connected host or the name specified in the `ssl_expected_hostname` parameter against the value in the server's certificate. If the names do not match, the connection is rejected. When this parameter is `disabled`, the server establishes secure communication with the other server, but does not verify the server's name.
`ssl_expected_hostname`	Specifies the name expected in the CN field of the other server's certificate. If this parameter is not set, the default is the hostname of the other server. This parameter is relevant only when the `ssl_verify_hostname` parameter is enabled.

Parameter	Description
`ssl_ciphers`	Specifies a list of cipher suites, separated by colons (:). This parameter accepts both the OpenSSL name for cipher suites, or the longer descriptive names. For information about available cipher suites and their names, see Specify Cipher Suites.

Routed Topic Messages

A server forwards topic messages along routes only when the global property is defined for the topic.

Topic messages can traverse multiple hops.

When a route becomes disconnected (for example, because of network problems), the forwarding server stores topic messages. When the route reconnects, the server forwards the stored messages.

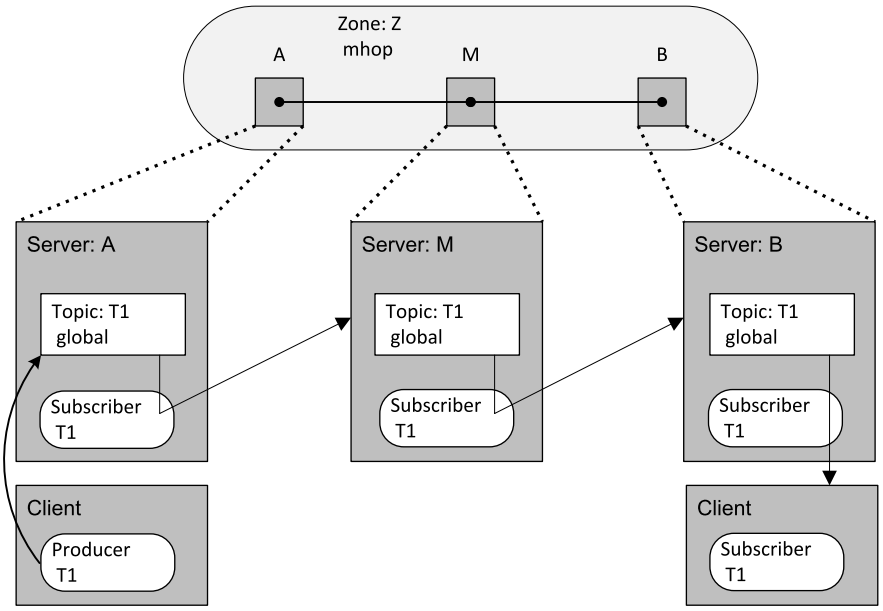
Servers connected by routes do exchange messages sent to temporary topics.

For more information, see `addprop topic` and `create topic`.

Registered Interest Propagation

To ensure forwarding of messages along routes, servers propagate their topic subscriptions to other servers.

For example, the top of the following image depicts an enterprise with three servers—A, M and B—connected by routes in a multi-hop zone. The bottom of the figure illustrates the mechanism at work within the servers to route messages from a producer client of server A, through server M, to server B and its subscriber client. Consider this sequence of events.



1. All three servers configure a global topic T1.
2. At bottom right of the above figure, a client of server B creates a subscriber to T1.
3. Server B, registers interest in T1 on behalf of the client by creating an internal subscriber object.
4. Because a route connects servers M and B, server B propagates its interest in T1 to server M. In response, M creates an internal subscriber to T1 on behalf of server B. This subscriber ensures that M forwards (that is, delivers) messages from topic T1 to B. Server B behaves as a client of server M.
5. Similarly, because a route connects servers A and M, server M propagates its interest in T1 to server A. In response, A creates an internal subscriber to T1 on behalf of server M. This subscriber ensures that A forwards messages from topic T1 to M. Server M behaves as a client of server A.
6. When a producer client of server A sends a message to topic T1, A forwards it to M. M accepts the message on its topic T1, and forwards it to B. B accepts the message on its topic T1, and passes it to the client.

Subscriber Client Exit

If the client of server B creates a *non-durable* subscriber to T1, then if the client process exits, the servers delete the entire sequence of internal subscribers. When the client

restarts, it generates a new sequence of subscribers; meanwhile, the client might have missed messages.

If the client of server B creates a *durable* subscriber to T1, then if the client process exits, the entire sequence of internal subscribers remains intact; messages continue to flow through the servers in store-and-forward fashion. When the client restarts, it can consume all the messages that B has stored in the interim.

Server Failure

In an active-active route between servers B and M, if B fails, then M retains its internal subscriber and continues to store messages for clients of B. When B reconnects, M forwards the stored messages.

In an active-passive route configured on B, if B fails, then M removes its internal subscriber and does not store messages for clients of B—potentially resulting in a gap in the message stream. When B reconnects, M creates a new internal subscriber and resumes forwarding messages.

In an active-passive route configured on A, if either server fails, then M retains its internal subscriber in the same way as an active-active route. However, B does not retain its internal state which it uses to suppress duplicate messages from A and can deliver messages to its consumers after they have consumed them. Therefore, if it is desirable to not lose messages and to not have duplicate messages, the route should be active-active.

Network Failure

If an active-passive connection between B and M is disrupted, M displays the same behavior as during a server failure.

maxbytes

Combining durable subscribers with routes creates a potential demand for storage—especially in failure situations. For example, if server B fails, then server M stores messages until B resumes. We recommend that you set the `maxbytes` or `maxmsgs` property of the topic (T1) on each server, to prevent unlimited storage growth (which could further disrupt operation).

Selectors for Routing Topic Messages

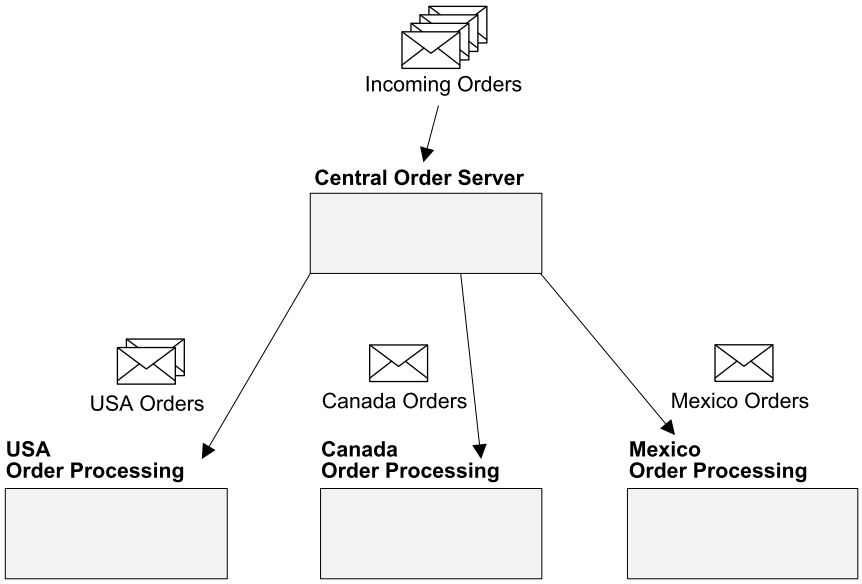
A server forwards a global topic message along routes to all servers with subscribers for that topic. When each of those other servers requires only a small subset of the messages, this policy could potentially result in a high volume of unwanted network traffic.

You can specify *message selectors* on routes to narrow the subset of topic messages that traverse each route.

> **Note:** Message selectors on routes are different from message selectors on individual subscribers, which narrow the subset of messages that the server delivers to the subscriber client.

Example

The following figure illustrates an enterprise with a central server for processing customer orders, and separate regional servers for billing those orders. For optimal use of network capacity, we configure topic selectors so that each regional server gets only those orders related to customers within its region.



Specifying Selectors

Specify message selectors for global topics as properties of routes. You can define these properties in two ways:

- Define selectors when creating the route (either in `routes.conf`, or with the

administrator command `create route`).

- Manipulate selectors on an existing route (using the `addprop`, `setprop`, or `removeprop` administrator commands).

Note: If you change the message selectors on a route, only incoming messages are evaluated against the new selectors. Messages pending in the server are re-evaluated only if the server is restarted.

Syntax

The message selector properties have the same syntax whether they appear in a command or in a configuration file:

```
incoming_topic=topicName selector="msg-selector"
outgoing_topic=topicName selector="msg-selector"
```

Note: The terms *incoming* and *outgoing* refer to the perspective of the active server—where the route is defined.

topicName is the name of a global topic.

msg-selector is a message selector string. For detailed information about message selector syntax, see the documentation for class Message in *TIBCO Enterprise Message Service Java API Reference*.

Example Syntax

As described in Example, an administrator might configure these routes on the central order server:

```
setprop route Canada outgoing_topic="orders" selector="country='Canada'"
setprop route Mexico outgoing_topic="orders" selector="country='Mexico'"
setprop route USA    outgoing_topic="orders" selector="country='USA'"
```

Those commands would create these entries in `routes.conf`:

```
[Canada]
url=ssl://canada:7222
outgoing_topic=orders selector="country='Canada'"
...
[Mexico]
url=ssl://mexico:7222
outgoing_topic=orders selector="country='Mexico'"
...
[USA]
url=ssl://usa:7222
outgoing_topic=orders selector="country='USA'"
...
```

Symmetry

`outgoing_topic` and `incoming_topic` are symmetric. Whether A specifies a route to B with `incoming_topic` selectors, or B specifies a route to A with `outgoing_topic` selectors, the effect is the same. That is, B sends only those messages that match the selector over the route.

Active-Active Configuration

In an active-active configuration, you may specify selectors on either or both servers. If you specify `outgoing_topic` selector S1 for topic T on server A, and `incoming_topic` selector S2 for T on server B, then the effective selector for T on the route from A to B is (S1 AND S2).

See also Active and Passive Routes.

Wildcards

You can specify wildcards in topic names. For each actual topic, the server uses logical AND to combine all the selectors that match the topic.

Note: However, routing of topic messages is only reliably supported when the subscriber's topic is a subset (or equal) of the configured global topic. Similarly, intersections are not supported. For example, if `topics.conf` contains `foo.*` and `foo.a*`, the following subscriptions are correct:

```
foo.*
foo.1
bar.a.b
```

The following subscriptions are *not* correct:

```
foo.>
bar.*.b
```

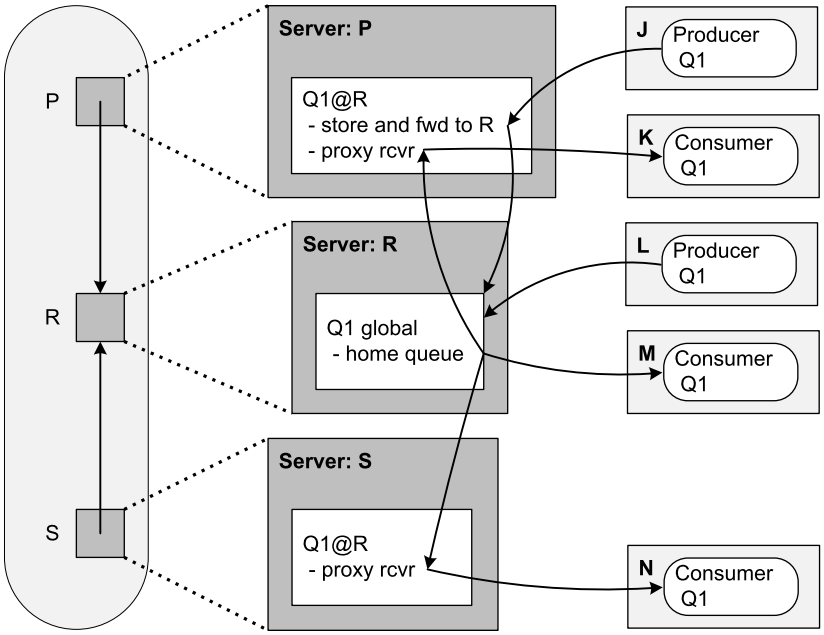
Routed Queues

With respect to routing, queues differ from topics in several respects.

These differences can be summarized as:

- Servers route queue messages between the queue owner and adjacent servers.
- The concept of zones and hops does not apply to queue messages (only to topic messages).

The left side of the following image depicts an enterprise with three servers—P, R and S—connected by routes. The remainder of the image illustrates the mechanisms that routes queue messages among servers (center) and their clients (right side).



Owner and Home

Server R defines a global queue named Q1. R is the *owner* of Q1.

Servers P and S define *routed queues* Q1@R. This designation indicates that these queues depend upon and reflect their *home queue* (that is, Q1 on server R). Notice that the designation Q1@R is only for the purpose of configuration; clients of P refer to the routed queue as Q1.

Example

When J sends a message to Q1, server P forwards the message to the home queue—Q1 on server R.

Now the message is available to receivers on all three servers, P, R and S—although only one client can consume the message. Either Q1 on P receives it on behalf of K; or Q1 on S receives it on behalf of N; or M receives it directly from the home queue.

Producers

From the perspective of producer clients, a routed queue stores messages and forwards them to the home queue. For example, when J sends a message to Q1 on server P, P forwards it to the queue owner, R, which delivers it to Q1 (the home queue).

The message is not available for consumers until it reaches the home queue. That is, client K cannot consume the message directly from server P.

If server R fails, or the route connection from P to R fails, P continues to store messages from J in its queue. When P and R resume communication, P delivers the stored messages to Q1 on R.

Similarly, routed queues do not generate an exception when the `maxbytes` and `maxmsgs` limits are exceeded in the routed server. Clients can continue to send messages to the queue after the limit is reached, and the messages will be stored in the routed server until the error condition is cleared.

Consumers

From the perspective of consumer clients, a routed queue acts as a proxy receiver. For example, when L sends a message to Q1 on server R, Q1 on P can receive it from R on behalf of K, and immediately gives it to K.

If server P fails, or the route connection from P to R fails, K cannot receive messages from Q1 until the servers resume communication. Meanwhile, M and N continue to receive messages from Q1. When P and R resume communication, K can again receive messages through Q1 on P.

Note: Receiving messages from a routed queue using either a small timeout (less than one second) or no wait can cause unexpected behavior. A small timeout value increases the chances that protocol messages may not be processed correctly between the routed servers. For example, queue receivers may not be correctly destroyed.

Configuration

You must explicitly configure each routed queue in `queues.conf`—clients cannot create routed queues dynamically.

> **Warning:** Dynamic routed queues are not supported. In a future release, the server will consider a routed queue with a wildcard as a misconfiguration and will fail to start when `startup_abort_list` includes `CONFIG_ERRORS`.

You may use the administration tool or administration API to configure routed queues; see `addprop queue` and `create queue`.

To configure a routed queue, specify the queue name and the server name of the queue owner; for example, on server P, configure:

```
Q1@R
```

> **Note:** It is legal to use this notation even for the home queue. The queue owner recognizes its own name, and ignores the location designation (@R).

It is illegal to configure a routed queue as `exclusive`.

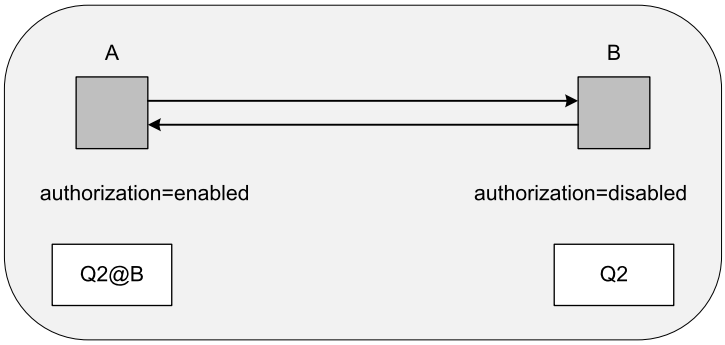
Browsing

Queue browsers cannot examine routed queues. That is, you can create a browser only on the server that owns the home queue.

Transactions

TIBCO Enterprise Message Service does not support transactional consumers on routed queues (through the use of XA or local transacted sessions).

Authentication and Authorization for Routes



In the above image, servers A and B both configure active routes to one another. This scenario will be used as an example to explain authentication and authorization for routes in the following sections.

Authentication

When a server's `authorization` parameter is enabled, other servers that actively connect to it must authenticate themselves by user name and password, or by providing an OAuth 2.0 access token, depending on the server's `user_auth` parameter (see Authentication Methods).

User & Password

When using the `local` or `jaas` authentication methods, a server will authenticate other servers connecting to it by validating the user name and password presented by the connecting servers.

In the scenario depicted in the image above:

- Because A enabled `authorization`:
    - If using `local` authentication, A must configure a user named B.
    - If using `jaas` authentication, A must be able to validate B's user credentials via

its JAAS authentication module.

- However, because B disabled `authorization`, A need not identify itself to B, and B need not configure, or be aware of a user named A.

Note: OAuth 2.0 authentication is not supported for active-passive route configurations.

OAuth 2.0

When using `oauth2` authentication, a server will authenticate other servers trying to connect to it by validating the OAuth 2.0 access token presented by the connecting servers.

In the scenario depicted in the image above:

- Because A enabled `authorization`, B must configure its route to provide an OAuth 2.0 access token when connecting to A.
- However, because B disabled `authorization`, A need not identify itself to B. A does not need to configure its route to provide an OAuth 2.0 access token when connecting to B.

The following table describes all required and optional route configuration parameters relating to the procurement of OAuth 2.0 access tokens. These parameters will need to be specified for any route that is connecting to an EMS server configured with OAuth 2.0 authentication.

Parameter	Description
`oauth2_access_token_file`	Specifies the path to a file containing an OAuth 2.0 access token to use for authenticating with the server on the other end of the route. If an access token is provided using this parameter, the EMS server will not attempt to obtain access tokens from an OAuth 2.0 authorization server even if `oauth2_server_url` and other relevant route configuration parameters are set.

Parameter	Description
`oauth2_server_url`	The HTTP(S) URL of the OAuth 2.0 authorization server that will issue the access token to be used for authenticating with the server on the other end of the route.
`oauth2_client_id`	The OAuth 2.0 client ID to use when authenticating with the OAuth 2.0 authorization server. This parameter and `oauth2_client_secret` are both required in order to obtain access tokens from the authorization server, regardless of the grant type to be used.
`oauth2_client_secret`	The OAuth 2.0 client secret to use when authenticating with the OAuth 2.0 authorization server. This parameter and `oauth2_client_id` are both required in order to obtain access tokens from the authorization server, regardless of the grant type to be used.
`oauth2_grant_type`	The grant type to use for requesting access tokens from the OAuth 2.0 authorization server. The type can be: • `client_credentials`–for Client Credentials Grant. • `password`–for Resource Owner Password Credentials Grant. If the `password` grant is specified, the `server` and `password` parameter values

Parameter	Description
	are used as the username and password for the grant. The default value of this parameter is `client_credentials`.
`oauth2_server_trust_file`	Specifies the path to a file containing one or more PEM-encoded public certificates that can be used to validate a secure OAuth 2.0 authorization server's identity. This parameter is only required if an HTTPS URL was specified for `oauth2_server_url`.
`oauth2_disable_verify_hostname`	If set, the EMS server will not verify the name in the CN field of the OAuth 2.0 authorization server's certificate. This parameter is optional and is disabled by default.
`oauth2_expected_hostname`	The name that the EMS server expects in the CN field of the OAuth 2.0 authorization server's certificate. This parameter is optional. When it is not set, the expected name is the hostname of the authorization server. This parameter is not relevant when the `oauth2_disable_verify_hostname` parameter is set to `true`.

See Authentication Using OAuth 2.0 more information about OAuth 2.0 authentication in EMS.

ACL

When routing a secure topic or queue, servers consult the ACL specification before forwarding each message. The servers must grant one another appropriate permissions to send, receive, publish or subscribe.

For example, in above image, you don’t need an ACL for messages to flow from A (where a producer is sending to) to B (where a consumer is consuming from) because B has authorization turned off and messages are being sent to and consumed from queues. However, if messages were to flow from B to A (producer connects to B and consumer connects to A), then server A's ACL should grant user B send permission on the queue Q2.

If we were to use topics in this example, then for messages to flow from A to B, you would need A to grant B the `subscribe` and `durable` permission on the topic (`global` on both servers). And for messages to flow from B to A, you would have to grant topic B `publish` permission on the topic.

Also see Authentication and Permissions.

Conversion of Server Configuration Files to JSON

The `tibemsconf2json` utility is provided to convert a set of text-based EMS server configuration files into a single JSON configuration file.

When using the utility, keep in mind that:

- If there are any unsupported parameters in the source configuration file, the `tibemsconf2json` utility issues a warning but continues converting.

  Review the *TIBCO Enterprise Message Service Release Notes* for details about any obsolete parameters that were removed from the current release.

- To convert a fault tolerant pair, use the *-secondaryconf* option to merge the two `tibemsd.conf` files of a fault tolerant pair of servers.

Syntax

To convert a EMS server configuration to JSON, use the command:

```
tibemsconf2json -conf source-file [-secondaryconf ft-source-file]
[-confencoding character-set-name] -json output-file | -console
```

where

- *source-file* is the path to the `tibemsd.conf` to be converted. Sub-file names and locations are derived from the content of the `tibemsd.conf` file. When converting servers in a fault tolerant pair, specify the configuration file for the primary server.
- *ft-source-file* is the path to the server configuration file for the second server in a fault tolerant pair. Specify this path with the `-secondaryconf` option to convert a fault tolerant pair.
- *character-set-name* is the name of the character set that was used to encode the *source-file* (and *ft-source-file*, if given). Any character encoding supported by the Java SE platform can be specified. Specify the encoding using the Canonical Name for java.lang API.

When omitted, the expected encoding is UTF-8. Note that the output JSON file is always encoded with UTF-8.

- *output-file* is the name and location of the new JSON file. This file must have the `.json` extension. For example, `tibemsd.json`. If no path is specified, the file is created in the current working directory.
- Alternately, specify `-console` to display the JSON output to the screen rather than saving to file.

The `tibemsconf2json` utility converts the `.conf` file to a JSON-based configuration.

If `-json` *output-file* is specified, the file is created and saved in the location specified, or the current working directory if no path is given. You can then start the EMS server using the JSON configuration, as documented in the Starting the EMS Server Using JSON Configuration section.

Convert a Fault Tolerant Pair

If a `-secondaryconf` *ft-source-file* is specified, the `tibemsconf2json` utility first converts the primary configuration to JSON, then uses the secondary configuration to complete the fault tolerant setup, deciding which one of the primary listen URLs must be marked as FT Active and adding extra secondary listen URLs, if any.

Note that the secondary configuration is used only for the purpose of completing the fault tolerant setup. With the only exception of the `logfile` property, any differences and discrepancies between the two initial sets of configuration files that are outside fault tolerance parameters are ignored.

Examples

Example 1

```
tibemsconf2json -conf EMS_HOME/samples/config/tibemsd.conf -json
tibemsd.json
```

Example 2

```
tibemsconf2json -conf EMS_HOME/samples/config/tibemsdft-1.conf
-secondaryconf EMS_HOME/samples/config/tibemsdft-2.conf -console
```

Monitor Messages

This section lists all the topics on which the server publishes messages for system events. The message properties for messages published on each topic are also described.

See Monitor Server Events for more information about monitor topics and messages.

Description of Monitor Topics

Topic	Message Is Published When...
`$sys.monitor.admin.change`	The administrator has made a change to the configuration.
`$sys.monitor.connection.connect`	A user attempts to connect to the server.
`$sys.monitor.connection.disconnect`	A user connection is disconnected.
`$sys.monitor.connection.error`	An error occurs on a user connection.
`$sys.monitor.consumer.create`	A consumer is created.
`$sys.monitor.consumer.destroy`	A consumer is destroyed.
`$sys.monitor.flow.engaged`	Stored messages rise above a destination's limit, engaging the flow control feature.
`$sys.monitor.flow.disengaged`	Stored messages fall below a destination's limit, disengaging the flow control feature.
`$sys.monitor.limits.connection`	Maximum number of hosts or connections is reached.
`$sys.monitor.limits.queue`	Maximum bytes for queue storage is reached.

Topic	Message Is Published When...
`$sys.monitor.limits.server`	Server memory limit is reached.
`$sys.monitor.limits.topic`	Maximum bytes for durable subscriptions is reached.
`$sys.monitor.producer.create`	A producer is created.
`$sys.monitor.producer.destroy`	A producer is destroyed.
`$sys.monitor.queue.create`	A dynamic queue is created.
`$sys.monitor.route.connect`	A route connection is attempted.
`$sys.monitor.route.disconnect`	A route connection is disconnected.
`$sys.monitor.route.warning`	An issue worth warning about occurs on a route connection.
`$sys.monitor.route.error`	An error occurs on a route connection.
`$sys.monitor.route.interest`	A change in registered interest occurs on the route.
`$sys.monitor.server.info`	The server sends information about an event; for example, a log file is rotated.
`$sys.monitor.server.warning`	The active server detects a disconnection from the standby server.
`$sys.monitor.topic.create`	A dynamic topic is created.
`$sys.monitor.tx.action`	A local transaction commits or rolls back.
`$sys.monitor.xa.action`	An XA transaction commits or rolls back.
`$sys.monitor.`*`D`*`.`*`E`*`.`*`destination`	A message is handled by a destination. The name of this monitor topic includes two qualifiers (*D*

Topic	Message Is Published When...
	and *E*) and the name of the destination you wish to monitor. *D* signifies the type of destination and whether to include the entire message: • T — topic, include full message (as a byte array) into each event • t — topic, do not include full message into each event • Q — queue, include full message (as a byte array) into each event • q — queue, do not include full message into each event *E* signifies the type of event: • r for receive • s for send • a for acknowledge • p for premature exit of message • * for all event types For example, `$sys.monitor.T.r.corp.News` is the topic for monitoring any received messages to the topic named `corp.News`. The message body of any received messages is included in monitor messages on this topic. The topic `$sys.monitor.q.*.corp.*` monitors all message events (send, receive, acknowledge) for all queues matching the name `corp.*`. The message body is not included in this topic's messages. The messages sent to this type of monitor topic include a description of the event, information about where the message came from (a producer, route, external system, and so on), and

Topic	Message Is Published When...
	optionally the message body, depending upon the value of *D*. See Monitor Messages for more information about message monitoring.

Description of Topic Message Properties

Each monitor message can have a different set of these properties.

Property	Contents
`conn_connid`	Connection ID of the connection that generated the event.
`conn_ft`	Whether the client connection is a connection to a fault-tolerant server.
`conn_hostname`	Hostname of the connection that generated the event.
`conn_ssl`	Whether the connection uses the TLS protocol.
`conn_type`	Type of connection that generated the event. This property can have the following values: • `Admin` • `Topic` • `Queue` • `Generic` • `Route` • `FT` (connection to fault-tolerant server) • `Unknown`
`conn_username`	User name of the connection that generated the event.

Property	Contents
conn_xa	Whether the client connection is an XA connection.
event_action	The action that caused the event. This property can have the values listed in Event Action Property Values.
event_class	The type of monitoring event (that is, the last part of the topic name without the $sys.monitor). For message monitoring, the value of this property is always set to message.
event_description	A text description of the event that has occurred.
event_reason	The reason the event occurred (usually an error). The values this property can have are described in Event Reason Property Values.
event_route	For routing, the route that the event occurred on.
message_bytes	When the full message is to be included for message monitoring, this field contains the message as a byte array. You can use the createFromBytes method (in the various client APIs) to recover the message.
mode	Message delivery mode. This values of this property can be the following: • persistent • non_persistent • reliable
msg_correlation_id	JMS correlation ID.
msg_id	Message ID.
msg_seq	Message sequence number.

Property	Contents
msg_size	Message size, in bytes.
msg_timestamp	Message timestamp.
msg_expiration	Message expiration.
replyTo	Message JMSReplyTo.
rv_reply	Message RV reply subject.
source_id	ID of the source object.
source_name	Name of the source object involved with the event. This property can have the following values: • XID (global transaction ID) • message_id • connections (number of connections) • unknown (unknown name) • Any server property name • the name of the user, or anonymous
source_object	Source object that was involved with the event. This property can have the following values: • producer • consumer • topic • queue • permissions • durable • server • transaction

Property	Contents
	• user • group • connection • message • jndiname • factory • file • limits (a limit, such as a memory limit) • route • transport
source_value	Value of source object.
stat_msgs	Message count statistic for producer or consumer.
stat_size	Message size statistic for producer or consumer.
target_admin	Whether the target object is the admin connection.
target_created	Time that the consumer was created (in milliseconds since the epoch).
target_dest_name	Name of the target destination
target_dest_type	Type of the target destination.
target_durable	Name of durable subscriber when target is durable subscriber.
target_group	Group name that was target of the event
target_hostname	Hostname of the target object.

Property	Contents
`target_id`	ID of the target object.
`target_name`	Name of the object that was the target of the event. This property can have the following values: • XID (global transaction ID) • message_id • connections (number of connections) • unknown (unknown name) • Any server property name • the name of the user, or anonymous
`target_nolocal`	No Local flag when target is durable subscriber.
`target_object`	The general object that was the target of the event. This property can have the following values: • producer • consumer • topic • queue • durable • server • transaction • user • group • connection • message • jndiname • factory

Property	Contents
	• file • limits (a limit, such as a memory limit) • route • transport
`target_selector`	Selector when the target is a consumer.
`target_subscription`	Subscription of the target object when target is durable subscriber.
`target_url`	URL of the target object.
`target_username`	Username of the target object.
`target_value`	Value of the object that was the target of the event, such as the name of a topic, queue, and so on.

Event Action Property Values

Event Action Value	Description
`accept`	connection accepted
`acknowledge`	message is acknowledged
`add`	user added to a group
`admin_commit`	administrator manually committed an XA transaction
`admin_rollback`	administrator manually rolled back an XA transaction
`commit`	transaction committed
`connect`	connection attempted

Event Action Value	Description
`create`	something created
`delete`	something deleted
`disconnect`	connection disconnected
`flow_engaged`	stored messages rise above a destination's limit, engaging the flow control feature
`flow_disengaged`	stored messages fall below a destination's limit, disengaging the flow control feature
`interest`	registered interest for a route
`modify`	something changed
`grant`	permission granted
`premature_exit`	message prematurely exited
`purge`	topic, queue, or durable subscriber purged
`receive`	message posted into destination
`remove`	user removed from a group
`resume`	administrator resumed a route
`revoke`	permission revoked
`rollback`	transaction rolled back
`rotate_log`	log file rotated
`send`	message sent by server to another party

Event Action Value	Description
subscribe	subscription request
suspend	administrator suspended a route
txcommit	administrator manually committed a local transaction
txrollback	administrator manually rolled back a local transaction
xacommit	an application committed an XA transaction (2-phase)
xacommit_1phase	an application committed an XA transaction (1-phase)
xastart	an application started a new XA transaction
xastart_join	an application has joined (that is, added) a resource to an existing transaction
xastart_resume	an application resumed a suspended XA transaction
xaend_fail	an application ended an XA transaction, indicating failure
xaend_success	an application ended an XA transaction, indicating success
xaend_suspend	an application suspended an XA transaction
xaprepare	an application prepared an XA transaction
xarecover	an application called recover (to get a list of XA transactions)
xarollback	an application rolled back an XA transaction

Event Reason Property Values

Event Action Value	Description
accept	connection accepted

Event Action Value	Description
acknowledge	message is acknowledged
add	user added to a group
admin_commit	administrator manually committed an XA transaction
admin_rollback	administrator manually rolled back an XA transaction
commit	transaction committed
connect	connection attempted
create	something created
delete	something deleted
disconnect	connection disconnected
flow_engaged	stored messages rise above a destination's limit, engaging the flow control feature
flow_disengaged	stored messages fall below a destination's limit, disengaging the flow control feature
interest	registered interest for a route
modify	something changed
grant	permission granted
premature_exit	message prematurely exited
purge	topic, queue, or durable subscriber purged
receive	message posted into destination
remove	user removed from a group

Event Action Value	Description
resume	administrator resumed a route
revoke	permission revoked
rollback	transaction rolled back
rotate_log	log file rotated
send	message sent by server to another party
subscribe	subscription request
suspend	administrator suspended a route
txcommit	administrator manually committed a local transaction
txrollback	administrator manually rolled back a local transaction
xacommit	an application committed an XA transaction (2-phase)
xacommit_1phase	an application committed an XA transaction (1-phase)
xastart	an application started a new XA transaction
xastart_join	an application has joined (that is, added) a resource to an existing transaction
xastart_resume	an application resumed a suspended XA transaction
xaend_fail	an application ended an XA transaction, indicating failure
xaend_success	an application ended an XA transaction, indicating success
xaend_suspend	an application suspended an XA transaction
xaprepare	an application prepared an XA transaction

Event Action Value	Description
`xarecover`	an application called recover (to get a list of XA transactions)
`xarollback`	an application rolled back an XA transaction

Error and Status Messages

This section lists all possible error messages that the server can output, alphabetized by category.

Key to this section

Category	
The category indicates the general class of error. This section is alphabetized by category.	
Description	The description explains the error category in more detail.
Resolution	The resolution indicates possible recovery actions that administrators should consider.
Errors	These strings represent all instances of the error, as they appear in EMS server code. Some categories have many error instances; others have only one. These strings can include formatting characters.

Error and Status Codes

Admin command failed	
Description	An admin tool or program using the admin API attempted an operation that failed for the given reason.
Resolution	The admin tool or admin API provides the failure reason. The user of the tool or API should examine the error and correct the syntax, parameter or configuration that is causing the failure.
Errors	Attempt by user %s to %s failed due to lack of permissions

Admin command failed	
	%s: create %s failed: conflicting zone: existing consumer has a different zone %s: create %s failed: detected duplicate durable subscription [%s] for topic [%s]. %s: create %s failed: illegal to use wildcard %s [%s]. %s: create %s failed: invalid %s [%s]. %s: create %s failed: invalid session id=%d. %s: create %s failed: invalid syntax of %s [%s]. %s: create %s failed: invalid temporary %s [%s]. %s: create %s failed: not allowed to create dynamic %s [%s]. Invalid consumer in recover one msg request. Invalid sequence number in recover one msg request.

Authentication error	
Description	The EMS server failed to authenticate the user or password.
Resolution	Ensure the user is defined to EMS by one of the methods allowed by the user_auth parameter in the main configuration file. The user is either specified by the application or in the TLS certificate. If the user is defined, reset the password and try again.
Errors	Unable to initialize connection, TLS username error. LDAP authentication failed for user '%s', status = %d - %s LDAP authentication failed for user '%s', LDAP server not found. LDAP authentication failed for user '%s', no password provided

Bad or missing value for command line parameter	
Description	An invalid value was supplied for a command line parameter.

Bad or missing value for command line parameter	
Resolution	Change the value of the named parameter to an acceptable value; for information about tibemsd command line parameters, see EMS documentation.
Errors	'%s' requires an integer argument. '%s' requires a positive integer argument. '%s' requires a string argument. Pathmap is only supported when using a JSON configuration file. Cannot open pathmap file '%s': file not found or permission denied. Invalid pathmap entry '%s'.

Basic initialization failed	
Description	tibemsd was unable to start.
Resolution	Correct the configuration or startup parameters and restart.
Errors	Unable to add admin user into admin group: error=(%d) %s Fault tolerant activation has to be greater than 2x heartbeat Server heartbeat client should be non-zero and no more than a third of the client timeout server connection Server heartbeat server should be non-zero and no more than a third of the server timeout server connection Client heartbeat server should be non-zero and no more than a third of the server timeout client connection Fault Tolerant configuration error, can't create loop. Fault tolerant connection failed, fault tolerant mode not supported on '%s'. Fault tolerant heartbeat has to be greater than 0 Initialization failed due to errors in configuration.

Basic initialization failed
Initialization failed due to errors in TLS.
Initialization failed due to errors with transports.
Initialization failed. Exiting.
Initialization has failed. Exiting.
Initialization of thread pool failed (%s). Exiting.
Startup aborted.
Server failed to read configuration.
Initialization failed: storage for '%s' not found.
Failure initializing storage thread: %s.
Ignoring condition %s in startup_abort_list: not supported on this platform.
Ignoring condition ALL in startup_abort_list: not supported on this platform. Using condition SSL instead.
Initialization failed due to errors with multicast.
Configuration error: dbstore_driver_name for store [%s] cannot be empty
Configuration error: dbstore_driver_url for store [%s] cannot be empty
Configuration error: dbstore_driver_dialect for store [%s] cannot be empty
Configuration error: dbstore_driver_username for store [%s] must be specified
Configuration error: dbstore_driver_password for store [%s] must be specified
Error Loading JVM: %s
Unknown Error Loading JVM
Trying JVM location: %s
Error Loading JVM: %s
Unknown Error Loading JVM
Unable to create default store '%s': %d - %s

Basic initialization failed	
	Configuration error: file=%s, line=%d: The parameter '%s' is not supported on this platform Unable to bind network IO thread: %d to Processor Id: %d. Exiting! Unable to bind storage thread for store '%s' to Processor Id: %d. Exiting! Unable to start Network IO Thread(s). Error: %d - %s Configuration error: Unsupported store type configured for store %s Configuration error: Mixing of grid, FTL, and/or file stores is not supported, please configure only one type. Configuration error: Missing ftlserver name in \drto\ spec of ftlserver yaml. Individual urls should look like ftlservername@host:port Configuration error: FTL server trust file, admin user and password are required for secure setup.

Commit failed due to prior failure or after fault-tolerant switch	
Description	A warning message indicating that the commit of a client application's transaction failed either because there were earlier errors when processing this transaction or because the transaction was started on the active server prior to a fault-tolerant failover.
Resolution	The client application should retry the transaction.
Errors	Commit failed due to prior failure or after fault-tolerant switch.

Compaction failed	
Description	Compaction of the store file failed.
Resolution	The most likely cause of this error is running out of memory. Shut down tibemsd and see remedies for Out of memory.
Errors	Compaction of store '%s' failed: %d (%s). Please shutdown and restart

Compaction failed	
	tibemsd. Compaction of store '%s' failed: %d (%s). Initialization of file_destination_defrag feature failed for queue '%s' (store '%s') due to an out of memory condition. Feature is disabled. file_destination_defrag of queue '%s' (store '%s') failed: %d (%s).

Configured durable differs from stored one	
Description	The durables configuration file specifies a durable with a given name and client identifier with attributes that are different from the identically named durable found in the meta.db file.
Resolution	Correct the durables configuration file to match the durable defined in the meta.db file or administratively delete the durable and re-define it.
Errors	Configured durable '%s' differs from durable in storage, storage version used.

Create of global routed topic failed: not allowed to create dynamic topic	
Description	A server received an interest notification from another server that does not match the allowed topics in its configuration.
Resolution	This only is printed when the trace includes ROUTE_DEBUG. If the server's topic definitions are as expected, this statement can be ignored or remove the ROUTE_DEBUG trace specification to prevent printing.
Errors	Create of global routed topic failed: not allowed to create dynamic topic [%s].

Create of routed queue failed: not allowed to create dynamic queue	
Description	A warning indicating that a tibemsd with a route to this daemon has a queue configured to be global but this daemon does not permit the creation of that queue dynamically.

Create of routed queue failed: not allowed to create dynamic queue	
Resolution	Add the specified queue or a pattern that includes it to this daemon if you want the queue to be accessible from this daemon, otherwise the warning can be ignored.
Errors	Create of routed queue failed: not allowed to create dynamic queue [%s].

Database record damaged	
Description	An error occurred reading one of the tibemsd store files.
Resolution	Send details of the error and the situation in which it occurred to TIBCO Support.
Errors	Server failed to recover state. Reverting incomplete batch for %s: %PRINTF_LLFMTd

Database Stores Setup Errors	
Description	In a database stores setup, errors occuring at runtime
Resolution	Check your database server vendor and database administrator for failures occuring during writes,deletes,reads of different records, for failures occuring during database store open check with the database administrator for permissions and the existence of the database. For failures occuring during a FT setup where all the stores are database stores, please check with the database server vendor or database administrator. In the case where both are active, we recommend shutting down both the servers and investigating the problem.
Errors	Unable to open store [%s]: [ESTATUS = %d, ERRSTR = %s] Failed to store message record in store [%s]: [ESTATUS = %d, ERRSTR = %s] Failed to write ack record in store [%s]: [ESTATUS = %d, ERRSTR = %s] Failed to write txn record in store [%s]: [ESTATUS = %d, ERRSTR = %s]

Database Stores Setup Errors
Failed to update txn record in store [%s]: [ESTATUS = %d, ERRSTR = %s]
No memory to create no hold list for valid msgs record
No memory to create hold list for valid msgs record
No memory to create held list for valid msgs record
Failed to write valid msg record in store [%s]: [ESTATUS = %d, ERRSTR = %s]
Failed to update msg record with record id [% PRINTF_LLFMT d] in store [%s]: [ESTATUS = %d, ERRSTR = %s]
Failed to delete %s record id = % PRINTF_LLFMT d : [ESTATUS = %d, ERRSTR = %s]
Failed to read message with store id = % PRINTF_LLFMT d: [ESTATUS = %d, ERRSTR = %s]
Failed to initialize dbstore [%s]: [ERRSTR = %s]
Failed to open store [%s], error = %s
Unable to restore %s records from store [%s]: [ESTATUS = %d, ERRSTR = %s]
Failed to delete meta record: [ESTATUS = %d, ERRSTR = %s]
Failed to beginTransaction: [ESTATUS = %d, ERRSTR = %s]
Failed to read message with store id = % PRINTF_LLFMT d: [ESTATUS = %d, ERRSTR = %s]
Store [%s] locked by server %s
Store [%s] cannot be locked by server %s
Failed to store txn record: [txn id = % PRINTF_LLFMT d, ESTATUS = %d]
Failed to update txn record: [txn record id = % PRINTF_LLFMT d, ESTATUS = %d]
Exception while processing msg from database store [%s], error = %d
Failed to write meta record: [ESTATUS = %d, ERRSTR = %s]

Database Stores Setup Errors	
	Failed to update meta record: [ESTATUS = %d, ERRSTR = %s]
	Failed to write connection record: error = %d
	Failed to write session record: error = %d
	Failed to write consumer record: error = %d
	Failed to write producer record: error = %d
	Failed to write zone record: error = %d
	Failed to update connection record: error = %d
	Failed to update consumer record: error = %d
	Failed to write purge record: [ESTATUS = %d, ERRSTR = %s]
	Commit Transaction Failed [ESTATUS = %d, ERRSTR = %s]
	No Memory to create lock manager: Store [%s] cannot be locked by server %s
	Could not find system record for store [%s]

Durable consumer was found in the store file for a route that does not exist	
Description	On server startup a durable consumer was found in the store file for a route that is not listed in the routes.conf file. This happens if the routes.conf file is manually edited.
Resolution	Make routing changes via administration tools.
Errors	Discarding durable '%s' for route '%s' because the route does not exist.

Dynamic Module Loading Errors	
Description	An error occurred when loading or using a shared library module.
Resolution	Module loading is affected by the presence of shared libraries in the module path. Use the +load tracing flag to get more information about how the server is loading modules. See the section on Starting the EMS Server for

Dynamic Module Loading Errors	
	more details.
Errors	Problem loading %s: %s Unknown problem loading %s. Loaded %s Problem binding %s: %s Unknown problem binding %s. Unable to locate %s Fatal error: Returned from exec(), errno = %d OpenSSL library version mismatch

Duplicate message detected	
Description	Warning generated when tibemsd receives a message with a message id that matches another message's message id.
Resolution	Only seen when message id tracking is enabled.
Errors	Detected duplicate %s message, messageID='%s'

Destination backlog growth detected	
Description	Warning generated when a destination appears to be growing an unwieldy backlog of messages.
Resolution	Consume or purge a large number of messages from that destination.
Errors	Destination growing very large: name=%s type=%s msg_count=%lld dest_ size=%lld (bytes) num_consumers=%d inbound_rate=%d (bytes/s) outbound rate=%d (bytes/s) Destination growing very large: name=%s type=%s msg_count=%lld dest_

Destination backlog growth detected	
	size=%lld (bytes) num_consumers=%d inbound_rate=statistics_disabled outbound_rate=statistics_disabled The server will attempt to trace warnings about destinations that are growing unbounded above %lld bytes or %lld messages. The server will attempt to trace warnings about destinations that are growing unbounded above %lld %s. Set server properties 'large_destination_memory' and 'large_destination_count' respectively to alter these thresholds. Set server property '%s' to alter this threshold.

Error in configuration file	
Description	The server encountered an invalid configuration statement in the specified configuration file on the specified line.
Resolution	Examine the appropriate configuration file and correct the syntax error.
Errors	Configuration warning: file=%s, line=%d: route '%s' does not have a user configured for authorization. TLS Configuration error: file=%s, line=%d: invalid certificate file name, unknown extension or invalid encoding specification Configuration error: file=%s, line=%d: illegal to specify %s for routed queue Configuration error: file=%s, line=%d: bad destination specification: %s Configuration warning: file=%s, line=%d: illegal to specify prefetch=none for global or routed queue. Prefetch reset to default. Configuration warning: file=%s, line=%d: illegal to specify prefetch=none for topic. Prefetch reset to default. Configuration error: file=%s, line=%d: ignored alias '%s' for %s '%s' because such alias already exists Configuration error: The specified file '%s' is empty or does not exist

Error in configuration file
Configuration error: file=%s, line=%d: both tibrv_export and tibrvcm_export are specified, ignoring tibrv_export
Configuration error: file=%s, line=%d: ignoring transport '%s' in %s list, transport not found
Configuration error: file=%s, line=%d: multiple bridge entries for the same destination '%s' are not allowed.
Configuration error: file=%s, line=%d: Ignoring durable, name cannot start with $sys.route, use route property instead.
Configuration error: file=%s, line=%d: Rendezvous transport not specified for Rendezvous CM transport '%s'
Configuration error: file=%s, line=%d: ignoring invalid max connections in the line, reset to %s
Configuration error: file=%s, line=%d: ignoring invalid max_client_msg_size in the line, reset to unlimited
Configuration error: file=%s, line=%d: value of %s out of range, reset to default
Configuration error: max_msg_field_print_size >= max_msg_print_size, resetting both to default
Configuration error: file=%s, line=%d: unable to create %s '%s': invalid destination name, invalid parameters or out of memory
Configuration error: file=%s, line=%d: value of db_pool_size too big or less than allowed minimum, reset to default value of %d bytes
Configuration error: file=%s, line=%d: Ignoring durable, route does not allow clientid, selector or nolocal.
Configuration error: file=%s, line=%d: Route '%s' does not exist for configured durable.
Configuration error: file=%s, line=%d: unable to process selector in route parameters, error=%s
Configuration error: file=%s, line=%d: both tibrv_import and tibrvcm_import

Error in configuration file
are specified, ignoring tibrv_import Configuration error: file=%s, line=%d: ignored route '%s' because route represents route to this server. Configuration error: file=%s, line=%d: ignoring invalid topic selector specifications in route parameters Configuration error: file=%s, line=%d: value of max_msg_memory less than allowed, reset to %dMB Configuration error: file=%s, line=%d: ignored alias '%s' for factory because such alias already exists Configuration error: file=%s, line=%d: invalid certificate file name, unknown extension or invalid encoding specification Configuration error: file=%s, line=%d: ignored route '%s' because route has invalid zone information. Configuration error: file=%s, line=%d: ignored route '%s' because route with such name or URL already exists. Configuration error: file=%s, line=%d: value of msg_pool_size invalid or too big or less than allowed minimum of %d, reset to default value of %d TLS Configuration error: file=%s, line=%d: invalid private key file name, unknown extension or invalid encoding specification Configuration conflict: file=%s, line=%d: value of msg_pool_block_size already set at line=%d. Ignoring msg_pool_size. Configuration error: file=%s, line=%d: bridge has no targets, unable to process Configuration error: file=%s, line=%d: Illegal to specify routed queue as a bridge source Configuration error: file=%s, line=%d: TMP.> cannot be bridge source or target destination Configuration error: file=%s, line=%d: A temporary destination cannot be bridge source or target destination

Error in configuration file
Configuration error: file=%s, line=%d: client_trace error: %s
Configuration error: file=%s, line=%d: %s
Configuration warning: monitor_listen is set more than once, using last value in the file
Configuration error: %smonitor_listen is malformed - %s
Configuration error: %smonitor_ssl_password not parseable or there were issues accessing specified file
Configuration error: file=%s, line=%d: duplicate specification of transport type
Configuration error: file=%s, line=%d: duplicate value
Configuration error: file=%s, line=%d: Ignoring durable, duplicate of earlier entry.
Configuration error: file=%s, line=%d: Ignoring durable, name is invalid.
Configuration error: file=%s, line=%d: Ignoring durable, name is missing or invalid.
Configuration error: file=%s, line=%d: Ignoring durable, topic is invalid.
Configuration error: file=%s, line=%d: Ignoring durable, topic is missing or invalid.
Configuration error: file=%s, line=%d: Ignoring durable, durable subscriptions not supported on temporary destination wildcard TMP.>.
Configuration error: file=%s, line=%d: error in the bridge description, unable to proceed.
Configuration error: file=%s, line=%d: error in permissions
Configuration error: file=%s, line=%d: error in the transport description, unable to proceed.
Configuration error: file=%s, line=%d: errors in line, some options may have been ignored
Error: unable to add bridge specified in file=%s, line=%d. Error=%s

Error in configuration file
Configuration error: file=%s, line=%d: Unable to create destination defined by the bridge source
Unable to create Rendezvous Certified transport '%s' because it references undefined Rendezvous transport '%s'
Configuration error: file=%s, line=%d: failed to create ACL entry, reason=%s
exit_on_nonretryable_disk_error: file=%s, line=%d: invalid boolean property value
consumed_msg_hold_time: file=%s, line=%d: invalid property value
active_route_connect_time: file=%s, line=%d: invalid property value
Fault tolerant reread error: file=%s, line=%d: invalid property value
Fault standby lock check error: file=%s, line=%d: invalid property value
Configuration error: file=%s, line=%d: ignored unknown permission '%s'
Configuration error: file=%s, line=%d: ignoring duplicate %s '%s' specified earlier
Configuration error: file=%s, line=%d: ignoring duplicate transport name '%s' in %s list
Configuration error: file=%s, line=%d: ignoring duplicate user
Configuration error: file=%s, line=%d: ignoring errors in permission line
Configuration error: file=%s, line=%d: ignoring invalid connect attempt count
Configuration error: file=%s, line=%d: ignoring invalid connect attempt delay
Configuration error: file=%s, line=%d: ignoring invalid connect attempt timeout
Configuration error: file=%s, line=%d: ignoring invalid disk statistic period
Configuration error: file=%s, line=%d: ignoring invalid entry syntax
Configuration error: file=%s, line=%d: ignoring invalid factory load balancing metric
Configuration error: file=%s, line=%d: ignoring invalid ft activation in the line

Error in configuration file
Configuration error: file=%s, line=%d: ignoring invalid ft heartbeat in the line
Configuration error: file=%s, line=%d: ignoring invalid ft reconnect timeout in the line
Configuration error: file=%s, line=%d: ignoring invalid line
Configuration error: file=%s, line=%d: ignoring invalid line in factory parameters
Configuration error: file=%s, line=%d: ignoring invalid line in route parameters
Configuration error: file=%s, line=%d: ignoring invalid line: invalid syntax in the line
Configuration error: file=%s, line=%d: ignoring invalid reconnect attempt count
Configuration error: file=%s, line=%d: ignoring invalid reconnect attempt delay
Configuration error: file=%s, line=%d: ignoring invalid reconnect attempt timeout
Configuration error: file=%s, line=%d: ignoring invalid value of %s
Configuration error: file=%s, line=%d: ignoring invalid value '%s' for property '%s'
Configuration error: file=%s, line=%d: ignoring unknown property '%s'
Configuration error: file=%s, line=%d: ignoring unrecognized property '%s'
Configuration error: file=%s, line=%d: ignoring user out of group context
Configuration error: file=%s, line=%d: illegal to use predefined name %s
Configuration error: file=%s, line=%d: Invalid clientid value
Configuration error: file=%s, line=%d: invalid value of db_pool_size, reset to default of %d bytes
Configuration error: file=%s, line=%d: invalid line syntax or line out of order

Error in configuration file
Configuration error: file=%s, line=%d: invalid value of max memory, reset to unlimited
Configuration error: file=%s, line=%d: invalid value of max_msg_memory, reset to unlimited
Configuration error: file=%s, line=%d: invalid property value
Configuration error: file=%s, line=%d: invalid property value, reset to default.
Configuration error: file=%s, line=%d: invalid password
Configuration error: file=%s, line=%d: invalid value of reserve_memory, reset to zero
Configuration error: file=%s, line=%d: invalid value of route_recover_interval, reset to default %d
Configuration error: file=%s, line=%d: invalid value of route_recover_count, line ignored
Configuration error: file=%s, line=%d: Invalid selector value
Configuration error: file=%s, line=%d: invalid syntax of %s, unable to continue.
Configuration error: file=%s, line=%d: invalid transport parameter '%s'
Configuration error: file=%s, line=%d: invalid transport type '%s'
Configuration error: file=%s, line=%d: invalid trace_client_host value
Configuration error: file=%s, line=%d: invalid trace_millisecond value
Configuration error: file=%s, line=%d: invalid value of %s, reset to unlimited
Configuration error: file=%s, line=%d: invalid value '%s'
Configuration error: file=%s, line=%d: invalid value '%s' for parameter '%s'
Configuration error: file=%s, line=%d: invalid value of '%s'
Configuration error: file=%s, line=%d: invalid value of %s
Configuration error: file=%s, line=%d: invalid value of %s, reset to 256MB

Error in configuration file
Configuration error: file=%s, line=%d: invalid value of %s, reset to default
Configuration error: file=%s, line=%d: line too long, ignoring it
Configuration error: file=%s, line=%d: maximum number of listen interfaces reached.
Configuration error: file=%s, line=%d: multiple principals specified, line ignored
Configuration error: file=%s, line=%d: multiple targets specified, line ignored
Configuration error: file=%s, line=%d: out of memory, unable to create Rendezvous transport
Configuration error: file=%s, line=%d: no permissions found in acl entry
Configuration error: file=%s, line=%d: no target found in acl entry
Configuration error: file=%s, line=%d: %s '%s' not found
Configuration error: No topic exists for configured durable '%s%s%s'.
failed to create durable '%s', exception: %s.
Configuration error: file=%s, line=%d: no valid user or group found in acl entry
Configuration conflict: file=%s, line=%d: Overriding value of msg_pool_size already set at line=%d.
Configuration warning: file=%s, line=%d: parameter '%s' is deprecated
Configuration warning: file=%s, line=%d: parameter '%s' is no longer supported
Configuration error: file=%s, line=%d: value of reserve_memory too small, reset to 16MB
Configuration error: file=%s, line=%d: ignoring invalid line in route parameters: invalid zone type, too long
Configuration error: file=%s, line=%d: ignoring invalid line in route parameters: invalid topic prefetch

Error in configuration file
Configuration error: file=%s, line=%d: ignoring invalid line in route parameters: zone name exceeding %d bytes
Configuration error: file=%s, line=%d: ignoring invalid line in route parameters: invalid OAuth 2.0 disable_verify_hostname
Routing Configuration error: file=%s, line=%d: invalid property value
Configuration warning: file=%s, line=%d: ignoring rvcmlistener, duplicate
Configuration error: file=%s, line=%d: ignoring rvcmlistener, first token is invalid
Configuration error: file=%s, line=%d: ignoring rvcmlistener, invalid destination
Configuration error: file=%s, line=%d: ignoring rvcmlistener, second token is invalid
Configuration error: file=%s, line=%d: ignoring rvcmlistener, third token is invalid
Configuration error: file=%s, line=%d: ignoring rvcmlistener, wildcards are not permitted
TLS Configuration error: file=%s, line=%d: duplicate value
TLS Configuration error: file=%s, line=%d: invalid property value
Configuration error: file=%s, line=%d: syntax error in the line, ignoring
Configuration error: file=%s, line=%d: syntax errors in line, line ignored
Topic '%s' not valid in configured durable '%s'.
%s%sNo client ID for%s unshared durable '%s'.
Configuration error: file=%s, line=%d: Unrecognized attribute
Configuration error: file=%s, line=%d: user '%s' not found, ignoring
Configuration error: file=%s, line=%d: value is invalid or less than minimum %d, reset to 0
Configuration error: file=%s, line=%d: value less than allowed minimum, reset

Error in configuration file
to 0 Configuration error: file=%s, line=%d: value of %s less than allowed minimum of %dKB, reset to unlimited Configuration error: file=%s, line=%d: Invalid value or value does not fall between %d and %d Configuration error: Invalid line: file=%s, line=%d Configuration error: Missing store header: file=%s, line=%d Configuration error: Mixed mode configuration: file=%s, line=%d Configuration error: Invalid store parameter: file=%s, line=%d Configuration error: Store definition failed Configuration error: Unrecognized store type requested. Configuration error: Filename for store '%s' cannot be empty. Configuration error: Store type '%s' is not supported on this platform. Configuration error: Grid store '%s' must not be async. Configuration error: Missing grid url for store '%s'. Configuration error: Missing grid name for store '%s'. Configuration error: Missing FTL server url for store '%s'. Configuration error: Missing FTL application name for store '%s'. Error occurred writing store definition into file. Configuration error: file=%s, line=%d: ignoring channel '%s' on topic '%s', channel does not exist Configuration error: file=%s, line=%d: ignoring channel '%s' on topic '%s', overlaps with channel '%s' on topic '%s' Configuration error: file=%s, line=%d: ignoring channel '%s', duplicate name Configuration error: file=%s, line=%d: ignoring channel '%s', address of '%s:%d' already defined

Error in configuration file
Configuration error: file=%s, line=%d: channel '%s', %s
Configuration error: file=%s, line=%d: channel '%s', no address specified.
Configuration error: file=%s, line=%d: channel '%s', invalid address syntax: port not specified.
Configuration error: file=%s, line=%d: channel '%s', invalid address: group must be in the range 224.0.0.0 to 239.255.255.255
Configuration error: file=%s, line=%d: channel '%s', interface must address a valid multicast-capable network interface.
Configuration error: file=%s, line=%d: channel '%s', invalid address: port must be in the range 1 to 65535
Configuration error: file=%s, line=%d: channel '%s', ttl must be in the range 1 to 255
Configuration error: file=%s, line=%d: channel '%s', priority must be in the range -5 to 5
Configuration error: file=%s, line=%d: channel '%s', maxrate must be less than 512MB
Configuration error: file=%s, line=%d: channel '%s', maxtime must be greater than 0
Configuration error: file=%s, line=%d: cannot store messages in: %s
Configuration error: file=%s, line=%d: cannot find store: %s
Required store param 'type' not specified for store '%s'
Configuration error: file=%s, line=%d: parameter does not match another parameter that defined store '%s' as 'file' type%s.
Configuration error: file=%s, line=%d: parameter does not match another parameter that defined store '%s' as 'dbstore' type%s.
Configuration error: file=%s, line=%d: parameter does not match another parameter that defined store '%s' as 'mstore' type%s.
Store '%s' already defined

Error in configuration file	
	Configuration error: Store with similar dbstore_driver_url exists, file=%s, line=%d Configuration error: duplicate file name %s for stores %s and %s Configuration warning: file=%s, line=%d: the discardAmount is too small for the selected RV Queue Limit Policy. It is recommended to have at least 10%% of the maxEvents Configuration error: file=%s, line=%d: the discardAmount is too big compared to the maxEvents value. Defaulting to TIBRVQUEUE_DISCARD_NONE policy Configuration error: file=%s, line=%d: maxEvents and discardAmount values must be strictly positive for an RV Queue Limit Policy other than TIBRVQUEUE_DISCARD_NONE. Defaulting to TIBRVQUEUE_DISCARD_NONE policy Configuration error: file=%s, line=%d: RV Queue Limit Policy '%s' unknown or not supported. Defaulting to TIBRVQUEUE_DISCARD_NONE policy Configuration error: file=%s, line=%d: Error parsing the RV Queue Limit Policy value '%s'. Defaulting to TIBRVQUEUE_DISCARD_NONE policy Configuration warning: file=%s, line=%d: The bridge's source destination '%s' is dynamic but has no parent. The bridge should either be removed or a static parent destination added Attempting to change the type of existing store '%s' from '%s' to '%s'. Store type changes are not permitted. Only stores of type 'file' are permitted. Unable to create store '%s' of type '%s'

Error writing commit request, errors already occurred in this transaction	
Description	A client application's attempt to commit a transaction failed because the server encountered an error during an operation associated with the transaction.
Resolution	Examine previous error statements to determine the cause of the operation failure and correct that before attempting the transaction again.

Error writing commit request, errors already occurred in this transaction	
Errors	Error writing commit request, errors already occurred in this transaction.

Error writing configuration file	
Description	tibemsd was unable to update one of its configuration files following a configuration change.
Resolution	Check that the user that started the tibemsd has permission to change the configuration files and that there is sufficient disk space on the device.
Errors	Error occurred saving acl information Error occurred saving bridges information Error occurred saving durables information Error occurred saving factories information Error occurred saving file '%s' Error occurred saving group information Error occurred saving %s information Error occurred saving main configuration file '%s' Error occurred saving routes information Error occurred saving tibrvcm information Error occurred while updating main configuration file '%s'. Configuration has not been saved. Error occurred writing bridges into file. Error occurred writing destination '%s' into file Error occurred writing factory into file. Error occurred writing group '%s' into file Error occurred writing into the file '%s'. Error occurred writing route into file.

Error writing configuration file	
	I/O error occurred saving bridge information I/O error occurred saving group information I/O error occurred saving route information I/O error occurred writing into file '%s' Configuration error: file=%s, line=%d: Ignoring property '%s' which is not supported in EMS Community Edition.

Error writing to store file	
Description	tibemsd was unable to write data to one of its store files.
Resolution	Ensure that the directory containing the store files is mounted and accessible to the tibemsd, and that there is free space available on the device
Errors	A %s I/O error occurred on file descriptor %d: %s - %d A %s I/O error occurred on file %s: %s - %d Failed writing block data to '%s': %s Failed writing message to '%s': I/O error or out of disk space. Failed writing purge state for queue '%s': I/O error or out of disk space. Failed writing purge state for topic consumer: I/O error or out of disk space. Exception trying to create confirm record, %s. Exception trying to create message from store: %s Exception trying to create transaction record. Exception trying to create valid messages record, %s. Exception trying to export message to RV. Failed writing message to '%s': %s. Exception writing transaction commit record: %s. Exception writing transaction rollback record: %s.

Error writing to store file	
	Exception writing transaction prepare record: %s. Failure deleting old version of transaction record: %s. Failed deleting '%s' record from %s: %s

Exceeded system resources.	
Description	The system resources are inadequate for timely processing of server activities.
Resolution	Increase the specified resource or reduce the workload on this server.
Errors	WARNING: Slow clock tick %d seconds, delayed messaging and timeouts may occur. System appears overloaded. WARNING: Connection timeouts delayed around %d seconds. Missed transfer of global lock before a slow operation was reported. Last offender grabbed lock around %d milliseconds ago. WARNING: Slow processing protocol message of type %s, lasted around %d milliseconds. WARNING: Slow completing processing protocol message of type %s, lasted around %d milliseconds. WARNING: Slow removing messages, lasted around %d milliseconds. This may have delayed connection timeouts. WARNING: Slow swapping out messages, lasted around %d milliseconds. This may have delayed connection timeouts. WARNING: Slow processing message (%s%s), lasted around %d milliseconds. WARNING: Slow processing message (%s ac=%d), lasted around %d milliseconds. WARNING: Slow processing event callback (%s), lasted around %d milliseconds. WARNING: Slow write to store (%s) lasted around %d milliseconds.

Exceeded system resources.	
	WARNING: A single %s store (%s) lasted around %d seconds. WARNING: Slow write to store (%s) (valid rec %g) lasted around %lld milliseconds. WARNING: Slow write to store (%s) (%d %s recs) lasted around %lld milliseconds. WARNING: Slow write to store (%s) (%lld tombstones and %lld msg recs size %lld) lasted around %lld milliseconds. WARNING: Slow write to store (%s) (removing %lld recs) lasted around %lld milliseconds.

Failed to open TCP port	
Description	tibemsd was unable to open the tcp port.
Resolution	Shutdown process that is using the port or change the value of the 'listen' parameter in the server's tibemsd.conf file to a port that is not in use.
Errors	Binding connection to TCP port %d failed:%d (%s).

File access error	
Description	tibemsd was unable to properly access the specified file.
Resolution	Check that the path name is correct and the directory exists, the user that started tibemsd has permission to read the specified directory and path, the file exists if it isn't one that the tibemsd can create, the file is not being used by another tibemsd or some other process.
Errors	Configuration file '%s' not found. Failed to create file '%s' failed to open file '%s'. failed to open log file '%s'.

File access error
Failed to read message from store.
Failed to rename file %s into %s: %s
Unable to open metadata file '%s', error '%s'.
Unable to open metadata file '%s', file may be locked.
Unable to open store file '%s', error '%s'.
Unable to open store file '%s', file may be locked.
Unable to preallocate storage file '%s'.
I/O error occurred reading from the file '%s'.
Exiting on non-retryable disk error: %d
Exiting on disk error: %d
Exception trying to read message from store.
Failure reading bridged msg from store %s: %d - %s
Exception trying to read message list from store.
Error during file close of '%s' - %d.
Unable to write to file '%s': Header record is empty
Unable to truncate file '%s': Length is less than the header record
Unable to open FT State Replication determination file '%s', error '%s'.
Unable to open FT State Replication determination file '%s', file may be locked.
Error upon accessing FT State Replication determination file '%s', invalid header CRC.
Unable to write to FT State Replication determination file '%s', error '%s'.
Exiting due to error while accessing the FT State Replication determination file.
Symbolic link '%s' is incorrect: %s.

FIPS 140-2 Mode Errors	
Description	An error occurred while starting or running the server in FIPS 140-2 compliant mode.
Resolution	Check the configuration of TLS related parameters to make sure that no incompatible ciphers or operations are requested.
Errors	Cannot specify ldap_tls_cipher_suite in FIPS 140-2 mode. Cannot specify ldap_tls_rand_file in FIPS 140-2 mode. Cannot specify TLS cipher list in FIPS 140-2 mode. Cannot specify random data source file in FIPS 140-2 mode. Cannot specify ssl_server_ciphers in FIPS 140-2 mode. The FIPS 140-2 mode is not supported on this platform.

FTL transport error	
Description	tibemsd encountered a FTL error.
Resolution	Correct the FTL transport in the EMS configuration and/or the Application in the FTL server.
Errors	Error setting FTL message constant '%s'. Constants (none). Constant (string) %s:\%s\ Constant (long) %s: %PRINTF_LLFMTd Exception Summary: %s Exception:\n%s The FTL application for this transport was administratively disabled. Please restart this server to re-enable this FTL transport. FTL Notification Type=%d: %s\n Failed to process FTL password.

FTL transport error
Connecting to the FTL server.
Setting a FTL discard policy is highly recommended.
Invalid FTL discard policy. Defaulting to \none\.
Global FTL Settings
FTL Server URLs: %s
Realm server Secondary URL: %s
Username: %s
Password: %s
Trust File: %s
Missing ftl_trustfile value. Trusting any FTL server without verifying trust in its certificate. This is not secure.
Log Level: %s
Application Name: %s
Discard Policy: %s
Discard Amount: %d
Discard Max Events: %d
Freeing FTL Global resources.
FTL Global resources freed.
FTL Advisory:
Unable to start FTL dispatcher thread.
Unable to initialize FTL Transport (%s). For more information, enable FTL tracing.
FTL Transport '%s' Ignoring unsupported FTL field type (%s).
Error importing FTL message. status = %s.
FTL Transport '%s': Skipping message (subscriber removed).

FTL transport error
Error setting FTL message field '%s'
Conversion from EMS bytes message to FTL message failed.
Conversion from EMS text message to FTL message failed.
Conversion from EMS data message to FTL message failed.
Conversion from an EMS Object message to a FTL message is not supported.
Conversion from an EMS Stream message to a FTL message is not supported.
Conversion from EMS message type (%d) to a FTL message is not supported.
Unable to set FTL fields from EMS Properties.
Unable to set FTL fields from EMS header values.
Unable to export message (%s).
FTL Transport Settings
Topic Import Delivery Mode: %s.
Queue Import Delivery Mode: %s.
Endpoint: %s
Import Parameters
Match String: %s
Subscriber Name: %s
Export Parameters
Format: %s
FTL Transport '%s' removed subscriber.
FTL Transport '%s' removing subscriber.
FTL Transport '%s': Error removing subscriber.
Error creating subscriber '%s' on endpoint '%s'
Creating FTL Transport '%s'

FTL transport error	
	Created FTL Transport '%s'
	Failed to create FTL transport '%s'
	FTL transport '%s' is creating a publisher.
	FTL Transport '%s' created a publisher.
	FTL transport '%s': Error creating publisher.
	Destroyed FTL transport '%s'
	Transport '%s' cannot subscribe to %s %s; already subscribed.
	FTL Transports cannot be imported on a wildcard destination.
	A FTL Transport can be used with only one destination.
	FTL Transport '%s' has subscribed to %s %s.
	FTL Transport '%s' failed to subscribe to %s %s.
	FTL Transport '%s' has unsubscribed from %s %s.
	FTL transport '%s' cannot be specified as an import by more than one destination.
	%s %s FTL Transport '%s' as an import for destination '%s'.
	%s %s FTL Transport '%s' as an export for destination '%s'.

Internal error	
Description	The server detected an internal inconsistency.
Resolution	Send the error statement and a description of the environment to TIBCO Support.
Errors	**Error** unable to process message, error = %s
	Admin user not found during initialization
	Error bridging transacted data message, '%s'.
	Error processing xa commit request, %s. connID=% PRINTF_LLFMT d %s

Internal error
Error processing xa end - transaction marked ROLLBACKONLY, %s. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s
Error processing xa prepare request, %s. connID=% PRINTF_LLFMT d %s
Error processing xa rollback request, %s. connID=% PRINTF_LLFMT d %s
Error decoding sequence data in xa rollback request. connID=% PRINTF_ LLFMT d %s
Error decoding sequence data in route ack response.
Unable to create internal session
Problem setting flow stall recover message on route queue:%s: %s
Failed to handle connection initialization: %s.
Problem trying to recover routed consumer for queue %s: setting recover message. Error: %s
Failed to send the flow stall recover request: %s.
Unable to handle transacted data message, '%s'.
Unable to handoff connection init message: %s.
Unable to initialize fault tolerant connection, remote server returned '%s'
Unable to process producer message, failed to add sender name, error=%s.
Unable to process sequence for message.
Unable to send recover ack on flow stall: %s
Handling of route flow stall recovery request from %s failed: unable to get message property %s: %s
Handling of route flow stall recovery request failed: Unable to get message properties:%s
Failed to send acknowledge to the stall recover request of server %s, will try later. Error: %s
failed to send recover ack on stalled flow: invalid consumer

Internal error
unable to create recovered connection, status: %s
Exception creating purge record.
Exception creating zone.
Exception creating zone: adding zone to state.
Exception in startup, exiting.
Exception preparing message for client send (%s): %s
Exception sending flow recover acknowledge
Exception sending routing information to %s - %s
Exception sending session init response
Exception sending queue acknowledge response to %s: %s
Exception trying to initialize connection.
Exception trying to initialize connection, can't send response: %s
Exception trying to initialize route.
Exception trying to initialize route '%s' configured durables: %s
Exception trying to process message, '%s'.
Exception trying to process message from store.
Failure queuing incoming message for processing: %s.
Failure queuing message for removal from system: %s.
Failure queuing message to add to dead queue: %s.
Failure discarding topic overflow: %s.
Failure processing system request.
Failure processing transaction message.
Failure bridging incoming message: %s.
Failed to pre-process bridge [%s:%s]: %d - %s

Internal error	
	Failure verifying uniqueness of routed message: %s.
	Failure scheduling message hold release: %s.
	Failure scheduling meta record delete: %s.
	Exception adding message write context: %s.
	%s: Failure processing multicast request: %s
	%s: Failure sending multicast request response: %s
	%s: Failure processing multicast status: %s
	%s: Failure sending multicast status response: %s
	%s: Failure sending multicast configuration: %s
	Failure sending multicast message on channel '%s': %s
	Failure enqueuing multicast message on channel '%s': %s
	Failure starting multicast engine: %s
	Failure starting multicast channel '%s': %s
	Failure posting multicast channel '%s' wake event: %s
	Failed preparing message for writing: %s
	Failed discarding local transaction: %s
	Abandoning transaction record due to IO failure.
	Error sending acknowledgment to route '%s': %s
	Error processing acknowledgments from route '%s': %s
	Failure starting delivery of delayed message seq = % PRINTF_LLFMTd: %s
	Failure moving failed delivery delayed message seq = % PRINTF_LLFMTd to dead queue: %s

Invalid connection	
Description	Warning indicating that tibemsd was attempting to reestablish delivery of

Invalid connection	
	messages across a route to another tibemsd but was unable to find the connection for that route.
Resolution	Either reduce the tibemsd's memory requirement by consuming messages or removing messages from its queues, or increase the amount of memory available to the tibemsd by shutting down other processes on the machine or increasing the machine's memory.
Errors	Recovery flow stall for destination %s failed: invalid route connection

Invalid destination	
Description	An application is attempting to use a destination name that is not valid.
Resolution	Alter application code to use an acceptable destination name.
Errors	%s: create %s failed: Not permitted to use reserved queue [%s]. %s: %s failed: illegal to use wildcard %s [%s]. %s: %s failed: %s [%s] not configured. At least one bridge is referencing %s [%s] as a target. This destination does not exist and there is no parent that would allow its dynamic creation. The destination has been forcefully created. To avoid this, the bridge(s) referencing this target should be destroyed. Use of '$' destination prefix is not supported [%s %s].

Invalid listen specification	
Description	The server could not parse the listen parameter in the tibemsd.conf file
Resolution	Correct the listen parameter to match the form [protocol]://[url] as specified in the manual.
Errors	Invalid listen specification: '%s'. Invalid request to create temporary destination.

Invalid session	
Description	tibemsd received a request that referred to a session that doesn't currently exist.
Resolution	Send details of the error and the situation in which it occurred to TIBCO Support.
Errors	Cannot find session for ack Cannot find session for ack range %s: destroy %s failed: invalid session id=%d. Unable to destroy session, invalid session. Invalid session in commit request. Invalid commit request. Invalid session trying to update(%d) tx record. Invalid session in commit transaction record. Invalid session in recover request. Invalid session in rollback request. Invalid session in xa end request. connID=% PRINTF_LLFMT d Invalid session in xa start request. connID=% PRINTF_LLFMT d

LDAP error - should always display LDAP error	
Description	An attempt to authenticate a client's userid and password using the external LDAP server failed.
Resolution	Examine the error code printed by the messaging server and consult the manual for the external LDAP server.
Errors	Filter '%s' contains an illegal type substitution character, only %%s is allowed Filter '%s' contains too many occurrences of %%s, max allowed is: %d

LDAP error - should always display LDAP error
Filter '%s' too long, max length is %d characters Invalid search scope: %s LDAP Configuration error: file=%s, line=%d: invalid property value LDAP is not present LDAP search resulted %d hits. Lookup of group '%s' produced incorrect or no results Missing LDAP URL Missing %s parameter Zero entries returned from getting attributes for group '%s': Failed adding user '%s' into LDAP user cache

LICENSE WARNING	
Description	The server detected a violation of its license.
Resolution	This error only occurs with the evaluation version of the server or in an embedded form. To correct this error either replace your evaluation version with a production version or contact the vendor who supplied the embedded version.
Errors	License violation: %s.

Missing configuration	
Description	An essential attribute has not been configured.
Resolution	Change the tibemsd.conf file so that a value for the attribute is provided.
Errors	Configuration error with metadata database. Configuration error with storage databases.

Missing transaction	
Description	A client application attempted to change the state of a transaction that the tibemsd does not have in its list of current transactions.
Resolution	Check tibemsd trace logs to see if the transaction had been committed or rolled back by an administrator, if not then check the client code to see if it or its transaction manager are calling the transaction operations in the correct order.
Errors	Cannot find transaction referred to transaction record update(%d) request. connID=% PRINTF_LLFMT d %s Cannot find transaction referred to in xa commit request. connID=% PRINTF_LLFMT d %s Cannot find transaction referred to in xa prepare request. connID=% PRINTF_LLFMT d %s Cannot find transaction referred to in xa rollback request. connID=% PRINTF_LLFMT d %s Received prepare request for transaction already prepared. connID=% PRINTF_LLFMT d %s Cannot find transaction referred to in xa start (resume) request. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s Cannot find transaction referred to in xa start (join) request. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s Cannot find transaction referred to in xa end request. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s

Multicast Channel Allotted Bandwith Exceeded.	
Description	Indicates that a multicast channel's allotted bandwidth has been exceeded.
Resolution	Either slow down the publisher(s), enable flow control, or increase the multicast channel's allotted bandwidth by increasing the channel's maxrate property or increasing the server's multicast_reserved_rate property.
Errors	Multicast channel \'%s\' has exceeded its allotted bandwidth

Multicast Daemon Status Codes and Errors	
Description	Errors occuring in the Multicast Daemon.
Resolution	Check the configuration of the Multicast Daemon and Server, as well as the health of the network.
Errors	Interface IP address: %s [%s] Connection created, connid=% PRINTF_LLFMT d Error: Unable to set channel property \'%s\'=% PRINTF_LLFMT d [%s] Created consumer consid=%PRINTF_LLFMTd connid=%PRINTF_LLFMTd topic=\'%s\' Multicast Daemon Id=%s Statistics enabled on a %d second interval. Statistics disabled. Rotating log from %s to %s Memory allocation error, possible data loss. Unrecoverable PGM error rc=%d, reason=%s Could not parse configuration file \'%s\' Interface IP address: %s Tracing enabled. Tracing disabled. refused new connection with existing ID % PRINTF_LLFMT d [%s] Connection destroyed, connid=%PRINTF_LLFMTd Error sending to consid=%PRINTF_LLFMTd connid=%PRINTF_LLFMTd from channel \'%s\': %s %s, status=%s Attached channel \'%s\' to consumer consid=%PRINTF_LLFMTd connid=%PRINTF_LLFMTd

Multicast Daemon Status Codes and Errors	
	Error attaching channel \'%s\' to consumer consid=%PRINTF_LLFMTd connid=%PRINTF_LLFMTd
	Detaching channel \'%s\' from consumer consid=%PRINTF_LLFMTd connid=%PRINTF_LLFMTd
	Destroying consumer consid=%PRINTF_LLFMTd connid=%PRINTF_LLFMTd
	Channel configuration from server does not match existing channel \'%s\'
	Ignoring additional PGM receiver created on group \'%s\', dport=%d, sport=%d, channel=%s
	Created channel: \'%s\'
	Error: %s is not a valid multicast-capable IP address. Use the -ifc command line parameter to specify a valid interface.

Multicast General Status Codes and Errors	
Description	General multicast errors that can occur in the Server and Multicast Daemon.
Resolution	Check the configuration of the Multicast Daemon and Server, as well as the health of the network.
Errors	PGM ERROR: %s - %s (%d)
	PGM ERROR: channel=\'%s\' - %s (%d)
	Error setting PGM parameter %s=%u: %s (%d)
	Error setting PGM parameter %s=\'%s\': %s (%d)
	Error getting PGM parameter \'%s\': %s (%d)
	Error getting PGM statistic \'%s\': %s (%d)
	Received an invalid EMS Message.
	Received a message spanning mulitple fragments.
	PGM Session was reset for channel \'%s\', PGM seqno=%PRINTF_LLFMTd, code=%c

Multicast General Status Codes and Errors	
	Stopped receiving on channel \'%s\'
	Started receiving on channel \'%s\'
	Error receiving on channel \'%s\'
	Stopped sending on channel \'%s\'
	Started sending on channel \'%s\'
	Error creating sender on channel \'%s\': %s

Grid and FTL store errors	
Description	An error occurred using a grid or FTL store.
Resolution	
Errors	Unable to step a statement (%s): %s: %d.
	Store %s: %s: bind fail: %d.
	Store %s: Fail retrieving msg interest info: %s.
	Store %s: Fail writing transaction record: %s.
	Store %s: Fail reading data: %s.
	Store %s: Fail reading topic message: %s.
	Store %s: Fail marking topic message non-pending: %s.
	Store %s: Fail reading next topic message: %s.
	Store %s: Fail reading queue message: %s.
	Store %s: Fail getting next queue message: %s.
	Store %s: Fail writing transaction info: %s.
	Store %s: Fail recording transaction msg: %s.
	Store %s: Fail recording transaction acks: %s.
	Store %s: Fail completing transaction: %s.

Grid and FTL store errors

Store %s: Invalid message interest for destination % PRINTF_LLFMT d.

Store %s: Invalid destination read.

Store %s: Failure restoring %s: %s.

Store %s: Failure restoring transaction msg: %s.

Store %s: Failure restoring transaction ack: %s.

Store %s: Failure resetting topic: %s.

Store %s: Correct functioning cannot be guaranteed due to mstore failure. Exiting.

Grid store failure at %s:%d: %s

Grid store failure: %s

FTL store failure at %s:%d: %s

FTL store failure processing map add: %s

FTL store failure discarding map state: %s

failure updating grid store lease: %s

Grid store ownership lost

Grid store ownership lease update delayed by %d msecs

Unable to acquire ownership of grid store %s.

Missing grid trust file value. Trusting any grid without verifying trust in its certificate. This is not secure.

Missing grid user name and/or user password value, so will attempt grid connection without credentials. This is not secure.

Unable to retrieve user name and/or user password value from file, so will attempt grid connection without credentials. This is not secure.

No memory available to process grid credentials. Aborting grid connect.

Missing FTL store trust file value. Trusting any FTL server without verifying trust in its certificate. This is not secure.

Grid and FTL store errors	
	Missing FTL store user name and/or user password value, so will attempt FTL store connection without credentials. This is not secure. Unable to retrieve user name and/or user password value from file, so will attempt FTL store connection without credentials. This is not secure. No memory available to process FTL store credentials. Aborting FTL store connect. Forced exit due to loss of primary status lost primary status; trying to reacquire ... regained primary status retrying FTL store op due to persistence error ftlserver shutdown requested: %s ftlserver requested save and exit: %s timeout initializing as service failure handling pserver event: %s All default stores are in synchronous mode

Out of memory	
Description	The server failed to allocate memory as it was attempting to perform an operation.
Resolution	Check how much memory the server process is using according to the operating system. Compare this with how much memory and swap space the host actually has. If there are sufficient memory and swap space, check the operating system limits on the server process to determine if this is the cause. If the limits are set to their maximum and this error occurs, reduce the load on this server by moving some topics and queues to another server.
Errors	%s trying to recreate persistent message. Error during routed queue configuration, can not create routed queue

Out of memory
consumer
Could not initialize monitor
Error: out of memory processing admin request
Error during route configuration, can not create routed queue consumer, err=%s
Configuration error - duplicate group: file=%s, line=%d: ignoring line
Unable to create admin group: out of memory during initialization
Error: unable to create alias for %s '%s': no memory
Error: unable to create alias: out of memory
Unable to create import event for %s '%s' on transport '%s'
Unable to create internal connection, error=(%d) %s
Unable to create internal connection: out of memory during initialization
Error: unable to create %s '%s': no memory
Error: unable to create route while parsing file=%s, line=%d.
Unable to create temporary destination, out of memory
Failed to create reserve memory. Exiting.
Failed writing message to '%s': No memory for operation.
Unable to process message imported on transport '%s'.
Fault Tolerant configuration, no memory!
Fault Tolerant error, %s.
No memory.
No memory: %s.
No memory authenticating user '%s'
No memory authenticating via LDAP

Out of memory
Out of memory while building admin response message
Out of memory while building JNDI response message
Out of memory creating global import event on transport '%s'
Out of memory creating import event for %s '%s' on transport '%s'
No memory creating stalled flows in destination
Out of memory during initialization
No memory for creating connection.
No memory generating dynamic route durable.
No memory in IO thread to create pool.
Out of memory while parsing bridges file
Out of memory while parsing factories file
Out of memory while parsing routes file
No memory performing routing operation.
Out of memory processing %s on %s '%s'
Out of memory processing administrative request
Out of memory processing message tracing
No memory processing purge record.
No memory while processing route interest
Out of memory processing transports
Out of memory processing transports configuration
Out of memory reading configuration.
Out of memory restoring routed consumer
Out of memory sending monitor message.
No memory sending topic routing information.

Out of memory
%s trying to add message to %s queue.
No memory trying to add message to system.
No memory trying to cleanup route.
No memory to create ack record.
No memory to create client connection
No memory to create configured durable '%s%s%s'.
No memory to create configured durables
No memory to create confirm record.
No memory to create connection.
No memory to create consumer.
No memory trying to create destination.
No memory to create destination for consumer or browser.
No memory trying to create global topic destination.
No memory to create message from store.
No memory trying to create message producer.
No memory to create producer.
No memory trying to create queue browser.
No memory trying to create response message.
No memory to create routed consumer
No memory to create routed queue consumers
No memory trying to create routed queue destination.
No memory trying to create routed tmp queue destination.
No memory to create session.
No memory trying to create tmp destination for consumer.

Out of memory
No memory trying to create transaction.
No memory to create valid messages record.
No memory restoring valid sequence number info.
No memory to create zone.
No memory trying to export message to RV.
No memory trying to export message to SS.
No memory trying to import message from RV%s.
No memory trying to import message from RVCM.
No memory trying to import message from SS. error=%s.
No memory trying to initialize connection.
No memory trying to initialize route connection.
No memory trying to parse configured durable.
No memory trying to process data message.
No memory trying to process queue message.
No memory to process route interest
No memory trying to process system request.
No memory trying to process topic consumer.
No memory trying to process topic message.
No memory trying to process xa end. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s
No memory trying to read message from store.
Route down while trying to recover routed consumer.
No memory trying to recover routed consumer.
No memory trying to recover one msg for routed consumer.

Out of memory
No memory trying to recover route stall.
No memory trying to recover route stall, will try again.
No memory to restore messages.
No memory to restore prepared transactions.
No memory trying to retrieve for queue browser.
No memory trying to send recover/rollback response.
out of memory trying to send topic interest to routes
No memory to set clientID for connection.
No memory trying to setup queue route configuration
No memory trying to setup route configuration
No memory trying to setup topic route configuration
Route recovery of destination %s on route from %s will fail: No memory
Route recovery of destination %s on route from %s will fail: No memory to create timer
Route recovery of destination %s on route from %s will fail: %s
Failed to initialize OpenSSL environment: out of memory
Out of memory queuing imported message for processing.
Out of memory gathering consumers for incoming message.
Out of memory scheduling message delete.
Out of memory preparing to write message.
Out of memory assembling list of message to store.
Out of memory processing route consumer.
Out of memory preparing message for writing.
Out of memory creating connection thread list.

Out of memory
Out of memory creating RV transport thread list.
Out of memory delaying bridged flow control response.
Out of memory preparing to delay flow control response.
Out of memory delaying one flow control response.
Out of memory delaying set of flow control responses.
Out of memory trying to clear message hold.
Out of memory trying to delete held message.
Unable to update the valid messages record. Error code: %d - %s.
No memory scheduling message delete completion, Error code: %d.
No memory to build msg properties.
No memory to create prop.
No memory to set prop.
No memory getting the list of delivered messages. The JMSXDeliveryCount property of some messages may no longer be accurate.
No memory getting the list of delivered messages from session % PRINTF_ LLFMT d. The JMSXDeliveryCount property of messages that were sent to this session may no longer be accurate.
No memory getting the list of delivered messages during rollback of transaction with xid: %s. The JMSXDeliveryCount property of messages that were rolled-back may no longer be accurate.
Out of memory discarding message.
Out of memory advancing queue pending.
Out of memory adding message to pending list.
Out of memory returning message to pending list.
Out of memory trying to re-queue after xa rollback.
Out of memory finalizing restored queue: %s.

Out of memory	
	Out of memory restoring queue flush state. Out of memory detaching message during queue purge. Out of memory removing message from queue. Out of memory retrieving message by correlation id. Out of memory scheduling cleanup of transaction ack: %s. Out of memory setting message all acked: %s. Out of memory cleaning up transaction: %s. Out of memory updating sent state on ack. Out of memory updating in-doubt state on ack. Out of memory removing message from system. Out of memory associating ack with data. Out of memory associating ack with transaction. Error setting mstore discard scan: %s. Out of memory recording modified topic. Out of memory re-queuing sent messages. No memory trying to resend delivered messages following an xa end NOTA. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s %s to create consumer on %s [%s] Failed to set delivery count in%smessage: status=%s Failure to create per-mstore delayed delivery state: %s.

Protocol error, incorrect XID in XA request	
Description	The tibemsd received an XA End instruction from the third party Transaction Manager which referred to a different transaction from the one currently in use by the session.

Protocol error, incorrect XID in XA request	
Resolution	Report this to the your Transaction Manager vendor.
Errors	Incorrect xid in xa end (0x%x) request. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s

Protocol error, transaction in incorrect state	
Description	A client application's attempt to start an XA transaction failed because the transaction already exists and is not in the correct state.
Resolution	This error is most likely caused by an external transaction manager that allowed two separate client applications to use the same XA transaction identifier (XID). Consult the manual for the transaction manager or report this to the transaction manager vendor.
Errors	Cannot process xa start for a session when another transaction is already active on that session. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s Cannot process xa start with TMNOFLAGS when the transaction is already active. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s All clients participating in the same global transaction must use the same protocol, connID=% PRINTF_LLFMT d Invalid xa start (resume) request: the session was not previously suspended. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s Error processing xa start - transaction marked ROLLBACKONLY. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s Error processing xa start request, %s. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s Invalid xa end (suspend) request: session already suspended or not started. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s Invalid xa end request: the session was neither associated with a transaction nor suspended. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s Error processing xa prepare - transaction marked ROLLBACKONLY, %s. connID=% PRINTF_LLFMT d %s

Protocol message format error	
Description	tibemsd received a message with either missing or incomplete data.
Resolution	Send details of the error and the situation in which it occurred to TIBCO Support.
Errors	Unable to confirm session, invalid request. Unable to create consumer, invalid destination. Unable to init session, invalid request. Unable to process msg for export. error=%s. Unable to recover consumer, invalid request. Unable to recover consumer, invalid session. Unable to recover one msg for consumer, invalid request. Unable to recover one msg for consumer, invalid sequence number. Unable to recover one msg for consumer, invalid session. Unable to serve the flow stall recover request from server %s, invalid request. Unable to start consumer, invalid consumer Unable to server the flow stall recover request from server %s, invalid consumer. Unable to unsubscribe consumer, invalid client request. %s: %s failed: illegal to use %s [%s] in standby state. Invalid flag in xa end request. connID=% PRINTF_LLFMT d sessID=% PRINTF_ LLFMT d %s Invalid flag in xa start request. connID=% PRINTF_LLFMT d sessID=% PRINTF_ LLFMT d %s Invalid request to delete temporary destination: %s. connID=% PRINTF_ LLFMT d Invalid request to delete temporary destination: not owner connection.

Protocol message format error
Invalid xid in commit request.
Invalid xid in commit transaction record.
Invalid xid trying to update(%d) transaction record.
Invalid xid in rollback request.
Invalid xid in rollback transaction record.
Invalid xid in xa commit request. connID=% PRINTF_LLFMT d
Invalid xid in xa end request. connID=% PRINTF_LLFMT d sessID=% PRINTF_ LLFMT d
Invalid xid in xa prepare request. connID=% PRINTF_LLFMT d
Invalid xid in xa rollback request. connID=% PRINTF_LLFMT d
Invalid xid in xa start request. connID=% PRINTF_LLFMT d sessID=% PRINTF_ LLFMT d
Malformed routed message
Problem decoding sequence data in confirm: %s.
Problem decoding sequence data in rollback.
Problem decoding sequence data in xa end. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d %s
%s:%s queue browser failed: queue name is missing in request message
Received admin request with replyTo not set
Received JNDI request with replyTo not set.
Received unexpected message type %d
No destination in incoming data message.
Invalid %s message

Protocol Sequence Error	
Description	A non-embedded java client is attempting to connect to a tibemsd that is part of an embedded Jakarta Messaging environment.
Resolution	Reconfigure the client to connect to a fully licensed tibemsd.
Errors	Invalid client connect detected. No closure.

Recovery errors	
Description	An error occurred during the recovery process.
Resolution	If you are not able to fix the problem and need to restart the system, make a backup of the store files and restart the server with the '-forceStart' command line parameter. The server will then attempt to start regardless of errors (except out-of-memory errors). In this mode, application messages and/or internal records causing problems (due to file corruption or other) will be deleted. Therefore, dataloss is likely to occur, so this command line parameter should be used with extreme caution and only after understanding the consequences. A copy of the store files can be sent to TIBCO Support for post-mortem analysis.
Errors	The recovery process stopped while processing a '%s' record (id=% PRINTF_ LLFMT d), error: %d - %s. Check the section 'Error Recovery Policy' from chapter 'Running the EMS Server' in the User Guide before attempting to restart the server The recovery process stopped while processing a '%s' record (id=% PRINTF_ LLFMT d) due to an out-of-memory condition. Ensure that the system can allocate sufficient memory to the EMS Server process before restarting it Unable to get the session's context handle for %s record: %d - %s Unable to get the list iterator for %s record Unable to get next element from list for %s record Unable to create %s object, no memory

Recovery errors

Error occured while processing %s record id=% PRINTF_LLFMT d (%s) - Unable to reconstruct message: %d - %s

Unable to recreate zone '%s': %d - %s

Unable to add zone '%s' to the system: %d - %s

Zone '%s' is defined as type '%s' in configuration but also is defined as type '%s' in meta.db

Unable to recreate connection id=% PRINTF_LLFMT d, client id=%s: %d - %s

Discarding session id=% PRINTF_LLFMT d because the connection id=% PRINTF_LLFMT d was not recovered. Recovery continues

Unable to recreate session id=% PRINTF_LLFMT d with connection id=% PRINTF_LLFMT d and client id=%s: %d - %s

Unable to recreate consumer id=% PRINTF_LLFMT d with connection id=% PRINTF_LLFMT d, session id=% PRINTF_LLFMT d, and client id=%s: invalid destination: %s

No memory to create destination for consumer id=% PRINTF_LLFMT d

Discarding consumer id=% PRINTF_LLFMT d on destination '%s' because connection id=% PRINTF_LLFMT d with client id=%s was not restored. Recovery continues

Discarding consumer id=% PRINTF_LLFMT d on destination '%s' and connection id=% PRINTF_LLFMT d with client id=%s because session id=% PRINTF_LLFMT d was not restored. Recovery continues

%s recreating consumer id=% PRINTF_LLFMT d

Failed to build import selectors for consumer id=% PRINTF_LLFMT d: %d - %s

Failed to read import selectors for routed consumer id=% PRINTF_LLFMT d: %d - %s

Discarding durable id=% PRINTF_LLFMT d (connection id=% PRINTF_LLFMT d, client id=%s) on destination '%s' because the durable name is not specified. Recovery continues

Unable to recreate producer id=% PRINTF_LLFMT d with connection id=%

Recovery errors
PRINTF_LLFMT d, session id=% PRINTF_LLFMT d, and client id=%s: invalid destination: %s
No memory to create destination for producer id=% PRINTF_LLFMT d
Discarding producer id=% PRINTF_LLFMT d on destination '%s' because connection id=% PRINTF_LLFMT d was not restored. Recovery continues
Discarding producer id=% PRINTF_LLFMT d on destination '%s' with connection id=% PRINTF_LLFMT d and client id=%s because session id=% PRINTF_LLFMT d was not restored. Recovery continues
Unable to recreate purge record: invalid destination: %s
Unable to recreate purge record for destination %s: %d - %s
Error creating message for transaction record: %d - %s
Error creating message's store structure for transaction record: %d - %s
Unable to recover transaction record: transaction id missing: %d - %s
Unable to recover transaction id=% PRINTF_LLFMT d: %d - %s
Unable to recover ack record (txid=% PRINTF_LLFMT d, consid=% PRINTF_ LLFMT d, seqid=% PRINTF_LLFMT d, location=%s): %d - %s
Unable to recover ack record, cannot create message: %d - %s
Unable to recover sequence numbers from valid record: %s
Unable to recover message, can not create lock: %d - %s
Unable to restore held message from store, (location=%s) no memory
Unable to restore message sequence=% PRINTF_LLFMT d: (location=%s) %d - %s
No memory to create destination for message
Inconsistency restoring routed message sequence=% PRINTF_LLFMT d
No memory to restore routed message sequence=% PRINTF_LLFMT d
Persisted message possibly corrupted: %s
Error creating message's store structure: %d - %s

Rejected attempt to connect via TLS to TCP port	
Description	A client application attempted to connect to the server's TCP port using the TLS protocol.
Resolution	Change the client application's URL from ssl to tcp or change the server's listen parameter from tcp to ssl. To activate a change of the server listen parameter requires a restart of the server.
Errors	Rejected attempt to connect via TLS to TCP port

Rejected attempt to connect via TCP to TLS port	
Description	A client application attempted to connect to the server's TLS port using the TCP protocol.
Resolution	Change the client application's URL from tcp to ssl or change the server's listen parameter from ssl to tcp. To activate a change of the server listen parameter requires a restart of the server.
Errors	Rejected attempt to connect via TCP to TLS port

Rejected connect from route: invalid cycle in route	
Description	The multi-hop route support of the server does not support configuring a cycle. However, it detected a configuration that would create a cycle.
Resolution	Remove one of the routes that creates the cycle.
Errors	[%s@%s]: rejected connect from route: invalid cycle in route: %s Illegal, route to '%s' creates a cycle. Terminate the connection Illegal, route to '%s' creates a cycle.

Rendezvous transport error	
Description	tibemsd encountered a Rendezvous error.
Resolution	See Rendezvous documentation for details of what the error means and how to remedy it.
Errors	Unable to create dispatcher for import event for %s '%s' on transport '%s', error is %s Unable to create inbox for import event for %s '%s' on transport '%s' Unable to create Rendezvous Certified transport '%s': %s Unable to create Rendezvous Certified transport '%s' because unable to create Rendezvous transport '%s' Unable to create Rendezvous transport '%s': %s Unable to create TIBCO Rendezvous Certified Listener for %s '%s' on transport '%s': %s Failed to confirm RVCM message: %d (%s) Failed to confirm RVCM message sequence % PRINTF_LLFMT u from cm sender '%s'. Error: %d (%s) Unable to store trackId % PRINTF_LLFMT d for RVCM message sequence % PRINTF_LLFMTu from cm sender '%s'. Error: %d (%s) Unable to retrieve trackId % PRINTF_LLFMT d. Error: %d (%s) A problem occurred while importing RVCM message sequence % PRINTF_ LLFMT u from cm sender '%s'. Expecting a redelivery Unable to queue the request type: %d. Transport '%s', destination '%s', CM Sender '%s', CM Sequence % PRINTF_LLFMT u . Error: %d (%s) Unable to queue the request type: %d. Transport '%s', destination '%s'. Error: %d (%s) Failed to disallow Rendezvous Certified Message listener '%s': %s Unable to export topic message, error=%s. Unable to pre-register certified listener '%s' on transport '%s': %s

Rendezvous transport error	
	Rendezvous send failed on transport '%s', error='%s' Unable to restart the CM Listener for %s '%s' (RVCM Transport '%s'). Error code: %d '%s' Unable to create the timer for the restart of the CM Listener for %s '%s' (RVCM Transport '%s'). Error code: %d '%s' Unable to stop the CM Listener for %s '%s' (RVCM Transport '%s'). Error code: %d '%s'

Restoring consumer failed	
Description	Seen when tibemsd starts up and detects that the zone for a route as specified in routes.conf has been changed.
Resolution	Either delete the route or change its zone back and restart the tibemsd.
Errors	Restoring consumer failed: Conflicting zone for route to [%s]: The route was initially zone \%s\ type %s, but now \%s\ type %s. Zone change not allowed while there are durable subscribers. Please delete the route first and create new one.

Running on reserve memory	
Description	Warnings indicating that the tibemsd has run out of memory and is now using its reserve memory
Resolution	Either reduce the tibemsd's memory requirement by consuming messages or removing messages from its queues, or increase the amount of memory available to the tibemsd by shutting down other processes on the machine or increasing the machine's memory.
Errors	Running on reserve memory, ignoring new message. Running on reserve memory, no more send requests accepted. Pending msg count = % PRINTF_LLFMT d Pending msg count = % PRINTF_LLFMT d

Runtime Error in Fault-Tolerant Setup	
Description	In a fault-tolerant setup, error occurs at runtime.
Resolution	Check the status of both servers (primary, secondary). In case of both active, the file store data may be corrupted already and we recommend shutting down both servers and investigating the situation.
Errors	Fault-tolerance error: Dual-Active server detected at: '%s' The active EMS server does not hold the lock on meta store The standby EMS server could not find the specified meta store. The active EMS server name is %s while the standby EMS server name is %s. The names must be the same A standby EMS server (%s) is already connected to the active EMS server Fault Tolerant error (%s), can't create connection to '%s'. Cannot determine which server should be active because both servers have been forced to start separately. Please force one of them to start (\forcestart\ tibemsadmin command). The other server will discard its data. This standby server is joining an active server and both have previously been forced to start. This server is discarding its data. Erasing content of store %s Internal error: Identical non-0 FT determination counters Store '%s' not defined on the active server, skipping it Store '%s' on the active server has a different file name than on the standby server Store '%s' is not present in the standby server configuration Error checking active server's configuration: %d - %s The configuration used on startup is incompatible with the one sent by the active server. Exiting!

TLS initialization failed	
Description	The server failed attempting to initialize the OpenSSL library.
Resolution	Examine the OpenSSL error and the EMS User Guide chapter describing the use of TLS.
Errors	Failed to process FT TLS password Failed to process TLS password Ignoring TLS listen port %s Failed to initialize TLS: can not load certificates and/or private key and/or CRL file(s) and/or ciphers. Failed to initialize OpenSSL environment: error=%d, message=%s. Failed to initialize TLS. Error=%s Failed to initialize TLS: unable to obtain password Failed to initialize TLS: server certificate not specified. Failed to initialize TLS: server private key not specified. Using secondary TLS password. Using secondary TLS identity. Using secondary TLS expected host name. Using secondary TLS private key.

Standby server '%s' disconnected	
Description	Lost connection with the standby fault-tolerant server.
Resolution	Determine if the standby server is running. If it is running, check for a network partition.
Errors	Standby server '%s' disconnected.

Store file format mismatch	
Description	The store files specified were created from a different version of EMS that is not supported by this version.
Resolution	Revert to use the version of EMS that created the store file or locate the store file conversion tool and use it to convert the store file to this version.
Errors	Unsupported store format: %s (%d)

System call error, should be errno-driven	
Description	A low-level system function has failed.
Resolution	Report the error to your system administrator and ask them to remedy the problem.
Errors	Accept() failed: too many open files. Please check per-process and system-wide limits on the number of open files. Accept() failed: %d (%s) Select() failed: %d (%s) %s%se=%p refs=%lu flags=%u type=%u id=%u subtype=%u q=%u cb_count=%u ioType=%u ioSrc=%d ioValid=%d ioVChecked=%d cb=%p free_cb=%p Epoll_wait() failed: %d (%s) Epoll_ctl() %s on fd %d failed: %d (%s) ioctl() on /dev/poll failed: %d (%s) write() %s update /dev/poll on fd %d failed: %d (%s) Cannot retrieve user name of the current process. Client connection not created, %s. Could not obtain hostname Could not resolve hostname '%s'. Possibly default hostname is not configured properly while multiple network interfaces are present.

System call error, should be errno-driven	
	Unable to listen for connections: %d (%s). Unable to open socket for listening: %d (%s). Closing connection from %s due to timeout, exceeded timeout of %d. Could not %s sequential file optimization: %d.

Transaction action while previous action is incomplete.	
Description	State-modifying action is requested on a transaction for which another such action is being processed.
Resolution	Send details of the error and the situation in which it occurred to TIBCO Support.
Errors	Cannot request second state change for transaction while the first request is in progress (%d, %d) %s. Unexpected request to roll xa txn forward with previous operation (%d) incomplete: %s. Unexpected request to roll xa txn back with previous operation (%d) incomplete: %s. Unexpected request to prepare xa txn with previous operation (%d) incomplete: %s. Unexpected request to commit xa txn with previous operation (%d) incomplete: %s. Unexpected request to commit session with previous operation (%d) incomplete.

Transaction timeout.	
Description	Transaction not completed before timeout. Offending transaction is discarded.

Transaction timeout.	
Resolution	Most likely, transaction manager error prevented it from advancing this transaction in a timely manner. Verify correct operation of the transaction manner.
Errors	Rollback due to timeout on unprepared transaction. connID=% PRINTF_LLFMT d %s

Unnecessary or duplicate message	
Description	tibemsd received a message with either missing or incomplete data.
Resolution	Send details of the error and the situation in which it occurred to TIBCO Support.
Errors	Error processing xa start request, %s. connID=% PRINTF_LLFMT d sessID=% PRINTF_LLFMT d Error trying to enter standby for '%s', %s.

Unrecognized option	
Description	The server's command line contains an unrecognized option.
Resolution	Run the server with the -help option and compare it with the command line containing the unrecognized option.
Errors	Unrecognized option: '%s'.

Monitoring issues	
Description	An issue has occurred related to the exposed monitor_listen or secondary_monitor_listen.
Resolution	Various.
Errors	Monitor: Failed to find %s %s: %s

Appliance State Replication Events.	
Description	A transition occurred in the State Replication feature.
Resolution	Refer to the section of the documentation pertaining to State Replication.
Errors	Transitioning to Server State: %s Transitioning to Server State: %s Forced exit to prevent dual-active servers Forced early exit - caught signal during server startup Site changed Disaster Recovery state: %s Restarging to complete upgrade

OAuth error - should always display OAuth error	
Description	An attempt to authenticate a client using the external OAuth server failed.
Resolution	Examine the error code printed by the messaging server and consult the manual for the external OAuth server.
Errors	The JWT token is expired. The JWT token is not yet valid. The JWT token is missing a required grant. A JWT token grant has the wrong value. The JWT token has an invalid audience. The JWT token has an invalid subject. The JWT token has an invalid issuer.

TIBCO Documentation and Support Services

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the Product Documentation website, mainly in HTML and PDF formats.

The Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

The following documentation for this product is available on the TIBCO Enterprise Message Service™ Product Documentation page:

- *TIBCO Enterprise Message Service™ Release Notes*
- *TIBCO Enterprise Message Service™ Installation*
- *TIBCO Enterprise Message Service™ User Guide*
- *TIBCO Enterprise Message Service™ C and COBOL Reference*
- *TIBCO Enterprise Message Service™ Java API Reference*
- *TIBCO Enterprise Message Service™ .NET API Reference*

Other TIBCO Product Documentation

When working with TIBCO Enterprise Message Service™, you may find it useful to read the documentation of the following TIBCO products:

- TIBCO® Messaging Manager
- TIBCO FTL®
- TIBCO ActiveSpaces®

- TIBCO Rendezvous®
- TIBCO® EMS Client for z/OS (CICS)
- TIBCO® EMS Client for z/OS (MVS)
- TIBCO® EMS Client for IBM i

How to Access Related Third-Party Documentation

When working with TIBCO Enterprise Message Service™, you may find it useful to read the documentation of the following third-party products:

- Jakarta Messaging™ Message specification, available through https://jakarta.ee/specifications/messaging/2.0.
- *Java™ Message Service* by Richard Monson-Haefel and David A. Chappell, O'Reilly and Associates, Sebastopol, California, 2001.
- Java™ Authentication and Authorization Service (JAAS) LoginModule Developer's Guide and Reference Guide, available through http://www.oracle.com/technetwork/java/javase/jaas/index.html.

How to Contact Support for TIBCO Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our product Support website.
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the our product Support website. If you do not have a username, you can request one by clicking **Register** on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to TIBCO Community.

Legal and Third-Party Notices

Cloud SG software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the “readme” file for the availability of a specific version of Cloud SG software on a specific operating system platform.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. CLOUD SG MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S), THE PROGRAM(S), AND/OR THE SERVICES DESCRIBED IN THIS DOCUMENT AT ANY TIME WITHOUT NOTICE.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "README" FILES.

This and other products of Cloud SG may be covered by registered patents. For details, please refer to the Virtual Patent Marking document located at https://www.tibco.com/patents.