



TIBCO FTL® - Enterprise Edition

Development

Version 7.1.2 | February 2026

Contents

Contents	2
About this Product	8
Product Overview	9
Package Contents	11
Directory Structure Reference	11
Sample Programs Reference	11
Brief Definitions of Key Concepts	14
Messaging Concepts	14
Infrastructure Concepts	15
Administrative Concepts	17
Information for Developers	18
API Reference Documentation	18
Sample Programs	18
C Programmer's Checklist	18
Java Programmer's Checklist	19
.NET Framework and .NET Core Programmer's Checklist	20
Go Programmer's Checklist	22
Restrictions on Names	23
Reserved Names	23
Length Limit	23
Destinations	23
FTL Development Overview	24
Destinations Model	27

Destination Types	28
Destination Name Syntax	35
Destination Matching and Wildcards	38
Statically Bound and Unbound Publishers	38
Topic Selectors (Match Filters)	40
Content Matching Model	43
Application Design	44
Structuring Programs	44
Structuring Programs in Go	46
Application Instance Identifier	47
Callbacks and Recursive Dispatch	47
Inline Mode	48
Request/Reply	48
Recovery of a Disabled Process Restart versus Reopen the Realm	49
Clean-Up	50
Consolidation with Loose Coupling	51
Delivery	52
Inbound Message Delivery	52
Content Matchers	53
Inline Mode	57
Direct Publishers and Subscribers	62
Use Cases for Direct Publishers and Subscribers	64
Programming with Direct Publishers	65
Programming with Direct Subscribers	67
Multithreading with Direct Publishers and Subscribers	68
Coordination Forms	69
IPv4 and IPv6	70
Endpoints	71
Abilities	72

Endpoints and Stores	73
Send: Blocking versus Non-Blocking	75
Messages	76
Message Objects in Program Code	76
Message Origin	76
Message Ownership	76
Message Access	78
Message Mutability	78
Copy of a Message	78
Messages and Thread-Safety	79
Field Values and Types	79
Field Data Type Reference	79
Storage of Field Values within Messages	81
DateTime	81
Formats: Managed, Built-In, and Dynamic	82
Format Names	83
Field Names	84
Flexibility in Formats and Messages	84
Field Access: Names and References	85
Reusing Message Objects	85
Built-In Formats Reference	86
Opaque Format	87
Keyed Opaque Format	87
String Encoding	87
DateTime Values Printable String Form	90
Go: Marshal Method for FTL Messages	90
Logs	99
Log Levels	99
Log Level Tuning via API	99
Log Output Targets	102

Log Content and Form	104
Advisory Messages	105
Advisory Message Common Fields Reference	105
Subscribing to Advisory Messages	106
Advisory Messages Catalog	107
DATALOSS FAILOVER_LOSS	109
DATALOSS SENDER_DISCARD	110
DATALOSS QUEUE_LIMIT_EXCEEDED	112
DATALOSS INCOMPLETE_MESSAGE	114
DATALOSS TPORT_DATALOSS	115
DATALOSS RECONNECT_LOSS	116
DATALOSS STORE_DISCARD_DATALOSS	118
DATALOSS UPSTREAM_LOSS	119
DATALOSS DIRECT_SUB_LOSS	121
ORDINAL_UPDATE	122
GROUP_STATUS	123
RESOURCE_AVAILABLE	124
RESOURCE_UNAVAILABLE	125
RETRANSMISSION RETRANSMISSION_REQUEST	126
RETRANSMISSION RETRANSMISSION_SENT	128
RETRANSMISSION RETRANSMISSION_SUPPRESSED	129
SUBSCRIBER_FORCE_CLOSE	130
LOCK_LOST	131
LICENSE	132
LICENSE SERVER_UNREACHABLE	133
LICENSE EXPIRED	134
LICENSE EXPIRING_IMMINENTLY	135
LICENSE EXPIRING_SOON	136
Notifications	138
Notifications Catalog	138

CLIENT_DISABLED	138
Groups of Applications	140
Introduction to Groups	140
Simple Fault Tolerance with Groups	140
Groups with More than Two Roles	142
Weights for Group Members	144
Groups Principles of Operation	146
Group Service	146
Group Members	146
Interactions within a Group	147
Side Effects of Groups	147
Techniques for Programming with Groups	148
Program Structure for Groups	148
Activation Interval	148
Disconnect	149
Group Status	149
Group Observer	150
Programmer's Checklist Addenda for Groups	151
Persistence: Stores and Durables	152
Persistence Concepts	152
Publishers	163
Publisher Mode and Send Policy	163
Maximum Message Size	165
Subscribers	166
Acknowledgment of Messages	166
Acknowledgment Mode	166
Delivery Count	167
Message ID	167
Message Destination	168
Message Browsing	168

No-Local Message Delivery	169
Retention Time	169
Rewinding a Subscription	170
Key-Value Maps	171
Locks	172
TIBCO Documentation and Support Services	175
Legal and Third-Party Notices	177

About this Product

TIBCO® is proud to announce the latest release of TIBCO FTL® software.

This release is the latest in a long history of TIBCO products that use the power of Information Bus® technology to enable truly event-driven IT environments. TIBCO FTL software is part of TIBCO Messaging®. To find out more about TIBCO Messaging software and other TIBCO products, please visit us at www.tibco.com.

Product Overview

TIBCO FTL® software is a messaging infrastructure product. It features high speed, structured data messages, and clearly defined roles for application developers and application administrators. TIBCO FTL software can achieve low message latency with consistent performance.

Components

TIBCO FTL software consists of these main components.

API Libraries

Developers use APIs in application programs.

FTL Server

The FTL server is a multi-purpose executable component that can consolidate several services into one server process: realm service, persistence service, transport bridge service, group service, authentication service, and eFTL service.

Combining services can simplify usage, especially in cloud-based installations.

FTL Services

An FTL server can provide one or more of the following services, as specified in its configuration file.

Realm Service

Applications obtain operational metadata from a realm service. The realm service funnels operational metrics from application clients to external monitoring points.

Authentication Service

The realm service can authenticate users against a separate authentication service or an internal authentication service.

Persistence Service

Persistence services store messages for durable subscribers.

Transport Bridge Service

Bridge services forward messages between two endpoints.

eFTL Service

eFTL services forward messages between eFTL clients and FTL clients. With eFTL, the messaging backbone can communicate with a variety of client devices, applications, and other things—smartphones, tablets, browsers, sensors.

Other Messaging Systems

TIBCO FTL can interoperate with the other component systems of TIBCO Messaging software, such as monitoring using [TIBCO® Messaging Monitor for TIBCO FTL®](#).

Package Contents

Directory Structure Reference

The TIBCO FTL installation directory contains the subdirectories in this table.

TIBCO FTL Directories

Directory	Description
bin	DLL files (for Windows). FTL server executable.
bin/modules	The FTL server manages the executable services of other related products, such as TIBCO eFTL. To enable this functionality, product installers for those products place their service components into the <i>FTL</i> product tree under this directory.
include	C API header files (in subdirectories).
lib	Library files (for UNIX). Java .jar files.
samples	Sample programs. For more detail, see Sample Programs Reference .
scripts	Postinstallation scripts for Linux platforms.

Sample Programs Reference

Sample programs serve as models for your own programs. You can use sample programs to experiment with TIBCO FTL software, test for correct installation, and demonstrate performance in your network environment.

Sample Programs

For instructions on running the sample programs, and for descriptions of the sample programs, see the readme files in each sub-directory of the `src` directory.

Each sample program prints a usage summary in response to the `-h` command line option.

Files

Files in Samples Directory

File	Description
<code>readme.txt</code>	Instructions to compile and run sample programs.
<code>setup</code>	Script that defines environment variables for the samples, such as <code>PATH</code> , <code>LD_LIBRARY_PATH</code> and <code>CLASSPATH</code> . Start with this script.

Directories

Samples Directories

Directory	Description
<code>bin</code>	Executable sample programs, precompiled from C sources.
<code>config</code>	More sample configuration files, for use as instructional models, but not related to the sample programs.
<code>docker</code>	Docker resources and information.
<code>helm</code>	Sample to deploy FTL on a Kubernetes cluster with the default configuration.
<code>jaas</code>	Files that configure JAAS authentication and authorization for the sample scripts.
<code>scripts</code>	Sample scripts to start and stop the FTL server. Before running the sample programs, you <i>must</i> start the FTL server with this script. Sample configuration file for the FTL server.

Directory	Description
	The <code>readme.txt</code> file contains instructions for using these samples.
<code>src</code>	Sample program source code in C, Java, and .NET. You may use these samples as models for your application programs. The <code>readme.txt</code> file contains instructions for compiling.
<code>yaml</code>	FTL server samples. The <code>readme.txt</code> file contains instructions.

Brief Definitions of Key Concepts

These brief definitions are reminders of the key concepts of TIBCO FTL software.

For a more intuitive picture of TIBCO FTL software and its concepts, please read in TIBCO FTL [Concepts](#) as your first introduction to this product.

Messaging Concepts

Application programs (clients) use TIBCO FTL software to communicate by sending messages to one another. Programs send messages using *publisher* objects, and receive messages using *subscriber* objects.

As of release 7.1.0, FTL offers a *destinations* messaging model. Publishers send to destinations, and subscribers receive messages from a destination.

As an alternative, for minimum messaging overhead, FTL also offers a *content matching* model. Subscribers use content matchers to express interest in messages with particular contents.

FTL typically uses a message broker architecture. For low latency, FTL can be configured such that messages travel directly between peer application programs.

Messages and Fields

A *message* is a structured unit of data. The structure is flexible, and developers tailor the structure to the needs of the application.

Each message consists of a set of named *fields*. Each field can contain a value of a specific data type.

Topics

In the destinations model, publishers can send messages to *topics*. This is a form of one-to-many communication, where messages can reach potentially many subscribers.

Queues

In the destinations model, publishers can send messages to *queues*. This is a form of one-to-one communication, where messages reach subscribers on a specific queue.

Key-Value Map

In the broker-based architecture, client programs can store data in a *map* as a key-value pair. This is suitable for data that is long-lived, as opposed to a message that is meant to be consumed once.

One-to-Many Communication

In the content matching model, publishers can send messages using one-to-many communication. These messages can reach potentially many subscribers.

A subscriber can use a content matcher to receive only those messages in which a particular set of fields contain specific values.

One-to-One Communication

In the content matching model, for efficient one-to-one communication, publishers can also send messages that can reach one specific inbox subscriber. The inbox data structure uniquely identifies an inbox subscriber.

Inbox subscribers cannot use content matchers. An inbox subscriber receives all messages directed to it.

Infrastructure Concepts

Applications and FTL servers communicate through various infrastructure components.

Destination

In the destinations model, messages are sent to and received from destinations. Each destination is managed by FTL server.

Endpoint

In the content matching model, an endpoint is the connection point for publishers and subscribers in communicating programs. Application architects and developers determine the set of endpoints that an application requires. Administrators formally define those endpoints. Programs create publisher and subscriber objects using those endpoints.

Transport

In the content matching model, a transport is the underlying medium that moves message data between endpoints. For example, TCP transports and multicast transports can move messages among programs connected by a network, and a shared memory transport can move messages among separate program processes on a multi-core hardware platform.

Affiliated Servers and Sites

For larger-scope messaging systems, FTL servers at different geographic sites can cooperate to distribute services. These affiliated servers are typically arranged as a primary site with one or more satellite sites. Each primary or satellite site can consist of multiple cooperating core and auxiliary servers. A third type of site, Disaster Recovery (DR) site, serves as a complete backup/standby site in the system.

Persistence Stores

In the broker-based architecture, all messages belong to a persistence store. A persistence store is a set of configuration parameters for this group of messages, relating to delivery assurance, routing, storage limits, and more. Stores can be replicated and/or written to disk within a persistence cluster to provide persisted messaging.

Persistence Clusters

Multiple persistence services can form a persistence cluster, where the persistence services form a quorum with an active (leader) persistence service, and standby (replica) persistence services. A persistence cluster manages a set of persistence stores.

Administrative Concepts

A *realm* is a namespace that defines resources and configurations available to application programs, such as destinations. FTL servers persist the realm definition. FTL clients connect to FTL server to retrieve the realm definition.

Administrators can use the following types of interfaces to define the realm:

- FTL Administrative GUI (Graphical User Interface)
- REST API (either via a command line tool like curl, or the Swagger extension on the GUI: REST API Reference)
- YAML file configuration options, which are read at FTL server startup
- FTL server command-line options (typically within a script)

For the destinations model, realm definition parameters fall into the following categories:

- Realm Properties: General and common settings for the entire realm.
- Destinations: Configuration for destinations used by publishers and subscribers. Destinations are mapped to persistence stores.
- Persistence: Configuration of stores, persistence clusters, and zones.

For the content matching model, additional parameters may be needed:

- Applications: A set of endpoints that a client program may use to send or receive messages.
- Transports: Note that transports are not visible to application developers, rather, they are strictly the responsibility of the administrator. From the developer's perspective, these details are hidden behind endpoints.
- Bridges: A service that forwards messages among sets of transport buses.
- Formats: Definitions of message formats

When configuration changes are edited and stored, they then can be deployed, per a controlled deployment procedure. When this procedure completes, FTL server stores the new definition and notifies all running FTL clients.

Information for Developers

API Reference Documentation

Reference documentation for the TIBCO FTL API is available only in HTML format.

The documentation set includes API reference resources for each supported programming language.

Sample Programs

The directory `TIBCO_HOME/ftl/version/samples` includes sample programs that you can use as models for programming, compiling, linking, and running.

For more information, see [Sample Programs Reference](#).

C Programmer's Checklist

Use this checklist when developing C programs that use the TIBCO FTL API.

When using the group facility, see also [Programmer's Checklist Addenda for Groups](#).

Environment

Use the setup script in the `samples` directory to set environment variables.

Code

Include the header file `ftl.h` (which in turn includes other header files).

Compile

Compile programs with an ANSI-compliant C compiler.

Use the compiler flags in the Makefile in the `samples/src/c` directory.

Link

On UNIX platforms, use the linker flags `-ltib -ltibutil`.

On Windows platforms, link the corresponding C library files, `tib.lib` and `tibutil.lib`.

Run

On Windows platforms, the `PATH` variable must include the `bin` directory to use the TIBCO FTL DLL files.

On UNIX platforms, the `LD_LIBRARY_PATH` variable must include the `lib` directory to use the TIBCO FTL library files.

Ensure that the TIBCO FTL realm server is running, and that each client process can connect to it.

Java Programmer's Checklist

Use this checklist when developing programs that use the TIBCO FTL Java API.

When using the group facility, see also [Programmer's Checklist Addenda for Groups](#).

Environment

The `CLASSPATH` must include an archive file that contains the TIBCO FTL classes (`tibftl.jar` from the `lib` directory). Use the `setup` script in the `samples` directory to set environment variables.

Code

Java programs must import this package:

```
import com.tibco.ftl.*;
```

Compile

TIBCO FTL classes require Java 1.8 (or later) 64-bit.

The CLASSPATH must include an archive file that contains the TIBCO FTL classes (tibftl.jar from the lib directory).

Run

TIBCO FTL classes requires Java 1.8 (or later) 64-bit.

On Windows platforms, the PATH variable must include the bin directory to use the TIBCO FTL DLL files.

On other UNIX platforms, either the LD_LIBRARY_PATH variable or the java.library.path variable must include the lib directory to use the TIBCO FTL library files.

Ensure that the FTL server is running, and that each client process can connect to it.

.NET Framework and .NET Core Programmer's Checklist

Use this checklist when developing programs that use the TIBCO FTL .NET API.

When using the group facility, see also [Programmer's Checklist Addenda for Groups](#).

Environment

Set up the environment for .NET Framework 4.7.2 or .NET Core 6. Ensure that it is current with the latest updates.

TIBCO FTL classes require Microsoft Visual Studio 2017 for development.

TIBCO FTL software requires 64-bit support (in the operating system and .NET Framework/Core).

Use the setup script in the samples directory to set environment variables.

For .NET Framework, ensure that the PATH environment variable includes the .NET Framework SDK (%windir%\Microsoft.NET\Framework64\dotnet_version).

For .NET Core, ensure that LD_LIBRARY_PATH (UNIX), PATH (Windows) or DYLD_LIBRARY_PATH (Mac OS) points to TIBCO_HOME/ftl/<version>/lib (UNIX, Mac OS) or TIBCO_HOME\ftl\<version>\bin (Windows).

Also for .NET Core, ensure that the dotnet driver version 2.1 is installed.

Code

To simplify coding, programs can include this statement:

```
using TIBCO.FTL;
```

Compile

TIBCO FTL classes require .NET Framework 4.7.2 or .NET Core 6.

To use the TIBCO FTL classes, programs must reference the assembly TIBCO.FTL. If the .NET framework is properly installed, then installing TIBCO FTL registers the assembly with the Windows general assembly cache (GAC).

The assembly TIBCO.FTL is compiled with platform target x64. You must compile applications that use TIBCO.FTL accordingly.

Run

TIBCO FTL classes require .NET Framework 4.7.2 or .NET Core 6.

To use the TIBCO FTL classes, application processes must access the assembly TIBCO.FTL, which installs into the Windows general assembly cache (GAC).

The PATH variable must include the bin directory to use the TIBCO FTL DLL files.

You must arrange appropriate .NET security for your applications. The base library calls unmanaged code, and requires full trust.

Ensure that the FTL server is running, and that each client process can connect to it.

Go Programmer's Checklist

Use this checklist when developing programs that use the TIBCO FTL Go API.

When using the group facility, see also [Programmer's Checklist Addenda for Groups](#).

Environment

Use the setup script in the `samples` directory to set environment variables.

Ensure that the Go compiler is in the `PATH` environment variable.

Code

Include the package `tibco.com/ftl`.

If your program uses the groups API feature, include the package `tibco.com/ftl/group`.

Compile and Link

Compile programs with a Go compiler, version 1.12.1 or later.

On the compiler command line, set `CGO_CFLAGS` to `<FTL_HOME>/include`. For example:

```
CGO_CFLAGS="-I /opt/tibco/ftl/7.1/include"
```

On the compiler command line, set `CGO_LDFLAGS` to `<FTL_HOME>/lib`, and include all of the required libraries. For example:

```
CGO_LDFLAGS="-L /opt/tibco/ftl/7.1/lib -lssl -lcrypto -ltib -ltibutil -ltibgroup"
```

Run

On Windows platforms, the `PATH` variable must include the `bin` directory to use the TIBCO FTL DLL files.

On UNIX platforms, the `LD_LIBRARY_PATH` variable must include the `lib` directory to use the TIBCO FTL library files.

Ensure that the FTL server is running, and that each client process can connect to it.

Restrictions on Names

Names and other literal strings must conform to the restrictions in the following topics.

Reserved Names

All names that begin with an underscore character (`_`) are reserved. It is illegal for programs to define fields, endpoints, applications, transports, formats, durables, stores, clusters, or zones with these reserved names.

You may use the underscore character elsewhere, that is, where it is *not* the *first* character of a name.

Length Limit

All names are limited to a maximum length of 256 bytes.

This limit includes names of fields, endpoints, applications, transports, formats, durables, stores, clusters, or zones. It also extends to string values in content matchers.

If you use UTF-8 characters that are larger than one byte, remember that the limit restricts the number of bytes in the name, rather than the number of characters.

Destinations

In addition to the above, there are additional restrictions on destination strings.

Destinations are limited to a total length of 249 characters.

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

See also [Destination Name Syntax](#).

FTL Development Overview

FTL offers two messaging models:

- **Destinations:** Client programs send and receive messages from topic and queue destinations. Key-value maps are also available for storage of long-lived data.
- **Content matching:** Subscribers express interest in messages based on the content of the message. (For example, subscribers may express interest in messages where certain fields are present and set to certain values.) Key-value maps are also available for storage of long-lived data.

Development of FTL applications is driven by the choice of messaging model. New users of FTL are encouraged to use the destinations model, which is likely to be easier to use and more intuitive. The content matching model is designed for minimum overhead, since in the content matching model, messages do not need to carry a destination.

In both cases, the administrator must ensure that appropriate resources are provisioned in the FTL realm configuration. The FTL realm configuration determines how FTL applications interact (for example, where messages for a particular destination are sent and how they are stored). See the FTL Configuration Overview topic in *TIBCO FTL® Administration*.

For concrete code examples, see the Running the Sample Applications in *TIBCO FTL® Getting Started*. API documentation is available for all supported languages.

For information specific to the destinations model, see [Destinations Model](#).

For information specific to the content matching model, see [Content Matching Model](#).

In addition, a number of concepts are the same for the destinations model and the content matching model.

- Before messaging can start, FTL applications need to retrieve the FTL realm configuration from an FTL server. To do so, program code specifies a list of FTL servers and passes the list to the realm connect call. The result of this call is a realm object, which encapsulates the FTL realm configuration and the connection to FTL server.
 - The list of FTL server URLs can be a single URL (localhost:8080) or a set of URLs

separated by the pipe (“|”) character (localhost:8080|localhost:9090|localhost:10100). The FTL library tries each URL until one succeeds. The URLs must all belong to the same cluster of FTL servers.

- The realm object is required when creating all publishers and subscribers.
- The realm object (and the encapsulated connection) remain valid until the FTL application is ready to shut down.
- If FTL server requires authentication or TLS, refer to *Securing FTL: Overview in TIBCO FTL® Administration*.
- FTL subscribers typically receive messages asynchronously. FTL applications first create an object called an event queue, and then associate the subscriber object and a message callback with the event queue.
 - One or more application threads (that is threads that run program code) invoke the dispatch method of the event queue, typically in a loop.
 - An internal thread in the FTL library waits for messages to arrive on the network for the subscriber. When a message arrives, the FTL library puts the message on the event queue and wakes up the application thread.
 - The application thread (still in the dispatch call) dequeues the message and executes the message callback for the subscriber.
 - The application thread returns from the event queue dispatch call. Typically, applications immediately begin a new dispatch call.
 - The net effect is that the work of receiving a message (done by the FTL library) and the work of processing the message (done by program code) happen on different threads, possibly in parallel.
 - Note that there is no relationship between an event queue and a queue destination. The event queue is entirely local to an FTL client program.
- FTL applications send and receive FTL message objects. See [Messages](#).
- The FTL library can log certain events or unexpected conditions. See [Logs](#).
- The FTL library can issue advisories for consumption by program code. See [Advisory Messages](#).
- The FTL library can issue out-of-band notifications to program code. See

Notifications.

- FTL applications can join groups to create a fault-tolerant set of applications. See [Groups of Applications](#).
- FTL applications can send and receive messages in a broker-based pattern, where FTL server is responsible for receiving messages from publishers, storing messages, and delivering messages to subscribers. When configured appropriately, this pattern offers replication and persistent storage of messages, fault tolerance, and delivery assurance. The destinations model always uses a broker-based pattern, while the content matching model can optionally use a broker-based pattern. See [Persistence: Stores and Durables](#).
- Instead of (or along with) sending and receiving messages, FTL applications can use key-value maps as simple databases, where the key is a string, and the value is an FTL message. Key-value maps are managed by FTL server, and can be replicated for fault tolerance and durability. Locks can be used with maps to implement operations such as atomic test-and-set in a distributed environment. See [Key-Value Maps](#).

Destinations Model

In the destinations model, for the realm connect call, you specify the following. Do not specify an application name; supply NULL instead.

A list of FTL server URLs as described in [FTL Development Overview](#).

Optional properties such as a client label, or properties for authentication and TLS. For authentication and TLS, see *Securing FTL: Overview* in *TIBCO FTL® Administration*.

Messages are sent to and received from destinations. The destinations model always uses a broker-based messaging pattern, where destinations are managed by FTL server.

Specifically, messages sent or received from a destination belong to a persistence store. A persistence store is a set of configuration parameters that determine durability and delivery assurance for messages (for example, persistent versus non-persistent messages), among other things. A persistence store is managed by a persistence cluster. A persistence cluster is a set of persistence services that cooperate for replication, storage, and fault tolerance. Persistence services run as part of the FTL server process.

Administrators must define destinations in the FTL realm configuration. At a minimum, administrators must specify which persistence store is associated with each destination. For more details, see *FTL Configuration Overview* in *TIBCO FTL® Administration*.



Note: The default FTL configuration includes a set of pre-defined destinations. Administrators may use these, or delete them and define their own.

Destinations come in three types: topics, queues, and maps. See [Destination Types](#). Client programs must obey syntax rules when specifying destination strings in program code. See [Destination Name Syntax](#).

Destinations in the FTL realm configuration are typically wildcards. However, destinations specified in program code are not wildcards. Instead, FTL matches destinations in program code to destinations in the FTL realm configuration. See [Destination Matching and Wildcards](#).

Publishers send messages to topic or queue destinations. Developers must choose whether to specify the destination when creating a publisher (for maximum performance) or when sending a message (for maximum flexibility). See [Statically Bound and Unbound Publishers](#).

Subscribers receive messages from topic or queue destinations. Recall that, to process messages, FTL client programs must create an event queue and a message callback, as described in [FTL Development Overview](#).

Note that subscribers must be created on a specific destination, not a wildcard destination. However, to simplify message processing, you may add several subscribers on different destinations to the same event queue using the same message callback. Program code may retrieve the destination of a message directly from the message object in the message callback.

Subscribers on topics must specify a type for the subscription (standard, shared, or lastvalue). Subscription types are described in [Topics](#). Subscribers on topics must also generally specify a durable name; if no durable name is specified, the subscription will be destroyed when the FTL client program closes the subscriber, disconnects from FTL server, or exits.

Subscribers on topics can specify selectors (match filters). When a match filter is specified, the subscriber does not receive all messages published to the topic; instead the subscriber only receives messages that match the constraints in the match filter. See [Topic Selectors \(Match Filters\)](#).

Key-value maps are created on map destinations. In this case, the destination simply serves as a name for the map. Maps are analogous to tables in a database.

Destination Types

Topics

Topics can be used to implement a publish-and-subscribe messaging system.

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.

Subscribers

In FTL, when an application program creates a subscriber on a topic, FTL will match the application's destination with a configured destination. See [Destination Matching and Wildcards](#) for a description of the destination matching algorithm. If no configured destination matches, the subscribe call will fail.

FTL then retrieves the persistence cluster and store from the configured destination. Recall that a configured destination is always associated with a single persistence store and cluster. (A persistence cluster is a group of persistence services, running in FTL servers, that cooperate to manage a persistence store. A persistence store is a group of messages that share a configuration, including parameters for replication and delivery assurance.)

The persistence cluster for that topic then creates a subscription on behalf of the client program. On success, the subscribe call will return.

Durables

All topic subscriptions are durable subscriptions. This means that the subscriber's interest for messages on this topic will be maintained even if the subscriber is disconnected or offline. If the subscriber is not available, the persistence cluster will store messages sent to that topic, according to the configuration of the persistence store, and deliver them to the subscriber at the next opportunity.

The durable subscription is identified by the durable name supplied by the application program. If the subscriber restarts or reconnects, it will use the durable name to resume consuming messages on its topic. In a given persistence store, no two durables can have the same name, even if they represent subscriptions on different topics.

There can be multiple durable subscriptions (using different durable names) on the same topic. A message is stored until acknowledged by all subscribers.

Note that a few things can modify the above behavior:

- Use of the `durable_ttl` destination property. See [Destination Details Panel](#).
- If the application program does not supply a durable name, the persistence cluster will delete the subscription when the subscriber disconnects, or when the application program closes the subscriber.
- Client programs may issue the unsubscribe call once all subscribers are closed, which instructs the persistence cluster to destroy the durable subscription identified by the durable name.

When the persistence cluster destroys a subscription, all pending messages for that subscription are discarded.

To avoid accumulation of messages, when a subscription is no longer needed, client programs should issue the unsubscribe call. Administrators can also delete durables on any topic.

Durable Types

When subscribing on a topic, the application program must specify the type of the subscription.

- **Standard:** The durable subscription is associated with one subscriber. All messages sent to the topic are delivered to all standard subscriptions on the topic. Creating a second subscriber with the same durable name will cause the first subscriber to be closed, and messages will be delivered to the second subscriber.
 - If you want to have redundant subscribers on a standard durable, consider using the application group feature. See [Groups of Applications](#).
- **Shared:** The durable subscription is associated with one or more subscribers. All messages sent to the topic are delivered to one of the subscribers. Shared subscriptions allow multiple client programs to parallelize the work of consuming messages on a topic.
- **Lastvalue:** The durable subscription is associated with one or more subscribers. When a subscriber starts, it receives the last message sent on its topic (rather than all messages). Subsequent messages published to the topic are delivered to each online subscriber on that topic. Lastvalue subscriptions allow subscribers to initialize to the most recent state for a topic, and then receive updates for that topic.
 - Unlike standard and shared subscriptions, for lastvalue subscriptions the persistence cluster stores only the last message sent to the topic. Publishing a message on the topic will cause the previous message to be replaced.
 - When an application program creates a lastvalue subscriber on a topic, the persistence cluster will store the last message sent to that topic or to any siblings of that topic (foo.1, foo.2, foo.3, etc). This is to allow applications to maintain lastvalue state when the set of topics is not known ahead of time (e.g., ticker symbols).
 - For example, if a lastvalue subscriber is created on foo.1, then a second lastvalue subscriber, using the same durable name, created on foo.2 would receive the last message published to foo.2. However, messages published to bar or foo.1.2 would not be stored.

It is possible to create standard, shared, and lastvalue subscriptions (using different durable names) on the same topic.

Message Acknowledgment

Lastvalue subscribers do not acknowledge messages. The most recent message is replaced whenever a new message is published to the topic.

Standard and shared subscribers do acknowledge messages. A message is not deleted by the persistence cluster until it has been acknowledged for all subscriptions on the topic.

- If the application program specifies explicit acknowledgment mode, then the application program must issue an acknowledge call for each message.
- Otherwise, FTL will automatically issue the acknowledge call once the subscriber's dispatch callback returns, for all messages processed by the callback.

Client programs may specify the acknowledgment mode.

- Sync acks: Acknowledgment calls are synchronous. The call does not return until the persistence cluster confirms that it has received the acknowledgment. This type of acknowledgment minimizes duplicates.
- Async acks: Acknowledgment calls are asynchronous. FTL will send the acknowledgment in the background on a timer. Duplicates can occur if the application program fails before the acknowledgment is sent.

Publishers

When an application program publishes a message to a topic, FTL will identify the configured destination, persistence store, and persistence cluster in the same way as it does for subscribers.

FTL will send the message to the indicated persistence cluster. If there are subscriptions on the topic, the message will be delivered to those subscribers. The message will be stored until acknowledged by all subscribers.

If there are no subscriptions on the topic, the message will be discarded.

The persistence store determines whether messages are replicated. A persistence store can be replicated ("persistent"), in which case all members of a persistence cluster store a copy of the message, or non-replicated ("non-persistent"), in which case the message is stored only in the memory of the persistence cluster leader.

The persistence store determines the degree of delivery assurance. If the persistence store is configured with publisher mode `store_confirm_send`, the send call will not return until the persistence cluster has stored the message (or discarded the message in the case of no interest). If the persistence store is configured with publisher mode `store_send_noconfirm`,

the send call will return immediately, regardless of whether the persistence cluster received the message.

Queues

Queues can be used to implement a point-to-point messaging system.

In a point-to-point messaging system, there is 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.

Queues are named by destination. There can only be one queue per queue destination.

Subscribers

In FTL, when an application program creates a subscriber on a queue, FTL will match the application's destination with a configured destination. See [Destination Matching and Wildcards] for a description of the destination matching algorithm. If no configured destination matches, the subscribe call will fail.

FTL then retrieves the persistence cluster and store from the configured destination. Recall that a configured destination is always associated with a single persistence store and cluster. (A persistence cluster is a group of persistence services, running in FTL servers, that cooperate to manage a persistence store. A persistence store is a group of messages that share a configuration, including parameters for replication and delivery assurance.)

If the queue does not already exist, the persistence cluster will create it. On success, the subscribe call returns.

Multiple subscribers may be created on the same queue. In this case messages are divided between the subscribers for processing.

Message Acknowledgment

Queue subscribers acknowledge messages. A message is not deleted by the persistence cluster until it has been acknowledged by a subscriber.

- If the application program specifies explicit acknowledgment mode, then the application program must issue an acknowledge call for each message.
- Otherwise, FTL will automatically issue the acknowledge call once the subscriber's

dispatch callback returns, for all messages processed by the callback.

Client programs may specify the acknowledgment mode.

- Sync acks: Acknowledgment calls are synchronous. The call does not return until the persistence cluster confirms that it has received the acknowledgment. This type of acknowledgment minimizes duplicates.
- Async acks: Acknowledgment calls are asynchronous. FTL will send the acknowledgment in the background on a timer. Duplicates can occur if the application program fails before the acknowledgment is sent.

Publishers

When an application program publishes a message to a queue, FTL will identify the configured destination, persistence store, and persistence cluster in the same way as it does for subscribers.

FTL will send the message to the indicated persistence cluster. If the queue does not exist, it will be created by the persistence cluster. The message will be stored even if subscribers have never existed for the queue.

The persistence store determines whether messages are replicated. A persistence store can be replicated (“persistent”), in which case all members of a persistence cluster store a copy of the message, or non-replicated (“non-persistent”), in which case the message is stored only in the memory of the persistence cluster leader.

The persistence store determines the degree of delivery assurance. If the persistence store is configured with publisher mode `store_confirm_send`, the send call will not return until the persistence cluster has stored the message (or discarded the message in the case of no interest). If the persistence store is configured with publisher mode `store_send_noconfirm`, the send call will return immediately, regardless of whether the persistence cluster received the message.

Queue Deletion

Only administrators can delete queues. Deleting a queue will discard all pending messages in the queue.

Maps

Maps can be used for long-term storage of data that is meant to be read repeatedly, rather than consumed once.

For example, if there is state associated with an application program, that state could be stored in a map. Client programs could read that state on startup or at any convenient time.

Maps store key-value pairs. The key is a string determined by the application. The value is an FTL message, containing whatever fields the application needs. Applications may get, set, or delete keys from the map.

In the map case, the destination is just a name for the map. There can only be one map per map destination.

Map Creation

In FTL, when an application program creates a map object on a map destination, FTL will match the application's destination with a configured destination. See [Destination Matching and Wildcards](#) for a description of the destination matching algorithm. If no configured destination matches, the subscribe call will fail.

FTL then retrieves the persistence cluster and store from the configured destination. Recall that a configured destination is always associated with a single persistence store and cluster. (A persistence cluster is a group of persistence services, running in FTL servers, that cooperate to manage a persistence store. A persistence store is a group of messages that share a configuration, including parameters for replication and delivery assurance.)

If the map does not already exist, the persistence cluster will create it. On success, the map creation call returns.

Multiple client programs can create map objects on the same map destination. They all refer to the same map at the persistence cluster, with the same set of key-value pairs.

Map Keys

Calls to get, put, or delete keys are always synchronous. The call does not return until the persistence cluster either responds with the value (in case of get), or confirms that the key has been updated or deleted.

The persistence store determines whether key-value pairs are replicated. A persistence store can be replicated ("persistent"), in which case all members of a persistence cluster store a copy of the key-value pair, or non-replicated ("non-persistent"), in which case the key-value pair is stored only in the memory of the persistence cluster leader.

Map Deletion

Client programs may issue a call to delete a map. This causes the persistence services to destroy all key-value pairs in the map.

Administrators may also delete maps.

Destination Name Syntax

FTL 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. Elements cannot incorporate the dot character by using an escape sequence.

Prefixes in the first element denote special destinations. If a prefix is used, the prefix cannot be the entirety of the first element (the first element must contain one or more characters following the prefix).

- Destinations where the first element begins with “Q:” are queue destinations.
- Destinations where the first element begins with “M:” are map destinations.
- Destinations where the first element begins with an underscore (_) are reserved by FTL. You may not configure reserved destinations or use them in client programs.
- All other destinations are topic destinations.

Although they are not prohibited, we recommend that you do not use tabs, spaces, or any unprintable character in a destination name.

You may use wildcards when configuring destinations. You may not use wildcards in client programs. Wildcards may only be the last element of a destination. See [Destination Matching and Wildcards](#) for more information.

Length

Destinations are limited to a total length of 249 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 Sequences in Destination Names

These characters or sequences of characters have special meanings when used in destination names:

Sequence	Name	Special Meaning
.	Dot	Separates elements within a destination name.
>	Greater-than	Wildcard character, matches one or more trailing elements.
*	Asterisk	Wildcard character, matches exactly one trailing element.
_	Underscore	When prefixed to the first element, denotes a destination reserved by FTL. May not be used in configuration or applications.
Q:	Queue prefix	When prefixed to the first element, denotes a queue destination.
M:	Map prefix	When prefixed to the first element, denotes a map destination.

For more information on wildcard matching, see [Destination Matching and Wildcards](#).

Examples of Destination Names

These examples illustrate the syntax for 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
- abc.>
- abc.*
- >
- *
- Q:abc
- Q:abc.>
- Q:>
- M:123.456
- M:123.*
- M:*

Invalid examples:

- News..Natural_Disasters.Flood (null element)
- WRONG. (null element)
- .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)
- Q: (queue/map prefix cannot be the entire first element)
- M:.abc (queue/map prefix cannot be the entire first element)
- _News (leading underscore is reserved by FTL)
- A.*.B (wildcards must be the last element)
- >.B (wildcards must be the last element)

Destination Matching and Wildcards

Whenever an application program uses a topic, queue, or map destination, FTL must match that destination to a configured destination. The configured destination contains subscription properties, access control lists, and the name of the persistence store and cluster.

When matching a destination to the configuration, only destinations of the same type are considered. For example, if an application specifies a topic destination, that destination is only matched against topic destinations in the configuration, not queue destinations or map destinations.

If the application's destination is exactly equal to a configured destination, then it matches that destination.

However, configured destinations are usually wildcards, representing a set of possible destinations. There are two wildcard characters, ">" and "*", with the following semantics. Recall that wildcards may only be used as the last element of a destination.

The wildcard > by itself matches any destination name (except for destinations reserved by FTL).

When > is used as the last element, it matches one or more trailing elements. For example: foo.> matches foo.bar, foo.boo, foo.boo.bar, and foo.bar.boo.

When used as the last element, 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.

When wildcards are used in the configuration, it is possible for the application's destination to match multiple configured destinations. See [Inheritance](#) for more details on overlapping destinations.

Statically Bound and Unbound Publishers

When creating a destination publisher, you have two choices:

- Specify a queue or topic destination name. The results in the creation of a *statically bound* publisher.
- Specify NULL for the destination name. This results in the creation of an *unbound* publisher.

Statically Bound Publishers

Statically bound publishers only send messages to the queue or topic destination that was specified in the publisher creation call. As such they are appropriate for client programs that send to a small number of well-known destinations.

For statically bound publishers, do not specify a destination in the send call. Specify NULL instead.

Statically bound publishers offer the maximum performance, since the FTL library does not need to consult the configuration or perform initialization steps during the send call. For example, overhead related to establishing the connection to the persistence cluster or initializing the publisher occurs during the publisher creation call, not during the send call.

Statically bound publishers also remove the need for client code to “remember” the destination during the send call, or even be aware of what the destination is.

Unbound Publishers

Unbound publishers can send messages to any topic or queue destination. As such, they are highly flexible, and are appropriate for client programs that send to a large number of destinations.

For unbound publishers, you must specify a destination in every send call.

The FTL library may need to lookup the configuration for a destination and perform initialization steps during the send call. This can happen if a destination in the send call is associated with a persistence store that has not been used by any previous destination. The affected send call will exhibit higher latency than normal.

After startup costs are paid, unbound publishers can perform well if the following conditions are met:

- The client program re-uses publisher objects. Destroying and creating a new publisher object may result in more startup costs. Therefore, code that creates a publisher object to send one message, and then immediately destroys the publisher object, is strongly discouraged.
- The number of persistence stores associated with the various destinations used by the client program is small. Client programs may use many different destinations; however, administrators should avoid creating many persistence stores for those destinations. In many cases, destinations with similar requirements regarding durability and delivery assurance (persistent vs. non-persistent) can share one

persistence store.

Topic Selectors (Match Filters)

By default, subscribers on a topic receive all messages sent to a topic. As an alternative, subscribers on topics can specify a selector, or *match filter*, when creating a subscription on a topic. Selectors are not supported for queues.

A match filter is a string that specifies a set of constraints on the fields in a message and their values. If a match filter is specified, then the subscriber receives messages sent to the topic only if the content of the message satisfies the constraints in the match filter. If a message is sent to a topic, but the message does not satisfy the constraints of any match filter of any subscription on the topic, then the message is discarded because none of the subscribers is interested in the message.

Match filters only operate on the fields of a message. They do not operate on the destination or other properties of the message.

Match strings are a logical AND, a logical OR, or both.

- Match on all fields (a logical AND match): The match string matches a message if and only if every field specification in the match string correctly matches the presence, absence, or value of a corresponding field in the message. For example, a subscriber with the following match filter would receive messages only if the bank is ABCBank and the type is debits. The subscriber would not receive the message if the publisher were sending a message with bank ABCBank and type of credits.

```
{“bank”: “ABCBank”, “type”: “debits”}
```

which can be read as:

```
("bank" is "ABCBank") AND ("type" is "debits")
```

- Match on one or more fields (a logical OR match): The match string matches a message if any field in the match string matches the presence, absence, or value of a corresponding field in the message. For example, a subscriber with the following interest would receive messages if the bank is ABCBank and the type is debits or credits.

```
[{"type": "debits"}, {"type": "credits"}]
```

or this shorter syntax:

```
{"type": ["debits", "credits"]}
```

which can be read as (any bank):

("type" is "debits" OR "type" is "credits")

- A combination of AND and OR matching. For example:

```
{"bank": "ABCBank", "type": ["debits", "credits"]}
```

which can be read as:

("bank" is "ABCBank") AND ("type" is "debits" OR "type" is "credits")

Match String Semantics

In the match string, each JSON “key” represents a field name, and each JSON “value” represents an assertion about that field.

- If the match string specifies a field with boolean value *true*, that field must be present in the message in order to match.
- If the match string specifies a field with boolean value *false*, that field must be absent from the message in order to match.
- If the match string specifies a field with either a string or long integer value, that field must be present in the message with that value.
- When comparing a field name or a value, all comparisons must be exact. Matching does not support wildcards nor regular expressions.

Match String Syntax

Construct matchers on this syntactic template in JSON format.

1. Logical AND Syntax

field:value pairs are in curly braces and separated by commas. Each field name can appear at most once.

```
{"fieldname1":"value1" , ... , "fieldnameN":"valueN"}
```

2. Logical OR Syntax

A logical OR can be expressed in one of two ways:

As a list of possible values for a given field.

```
{"f1": ["v1", "v2"]}
```

As a list of valid matchers, which need not be related to each other, and each could be a logical OR.

```
[{"f1": "v1"}, {"f1": "v2"}, ..., {"f1": "vn", "f2": "w1"}]  
[{"f1": ["v1", "v2"]}, [{"f2": "w1"}, {"f2": "w2"}]]
```

3. Other Syntax Rules

Logical AND as well as logical OR syntax follow these rules:

- Enclose field names and string values in double-quote characters. You may precede quote characters with an escape character, as needed.
- Do not enclose boolean tokens in double-quote characters.
- Values can be long integers, strings, or the special boolean tokens true or false.
- When value is a string, its maximum length is 256 characters (see [Restrictions on Names]).
- Whitespace is ignored, except within double-quote characters.

Content Matching Model

In the content matching model, for the realm connect call, you specify:

- A list of FTL server URLs as described in [FTL Development Overview](#).
- An application name.
- Optional properties such as a client label, or properties for authentication and TLS. For authentication and TLS, see *Securing FTL: Overview* in *TIBCO FTL® Administration*.

The application name must be defined in the FTL realm configuration. An application is a list of endpoints, defined by the administrator. When an FTL client program specifies an application name in the realm connect call, it gains access to all endpoints defined in that application.

 **Note:** The default FTL realm configuration includes an application named “default”, which contains a basic set of endpoints for broker-based messaging.

When defining an application, administrators also define the endpoints in that application. The configuration of an endpoint describes how an FTL client program should send and receive messages over the network.

- If an endpoint is associated with a persistence store, messages will be sent to and received from the persistence cluster that manages the persistence store. This is a broker-based messaging pattern; persistence clusters are provided by FTL server.
- If an endpoint is associated with one or more peer-to-peer transports, messages will be exchanged directly between FTL client programs, without involving FTL server. The transport describes the protocol to use for communication (TCP, multicast, and so on) along with the network addresses to use (host, port, and so on).

Publishers are created on endpoints. If the endpoint is associated with a persistence store, messages are sent to the persistence cluster that manages the persistence store. If the endpoint is associated with peer-to-peer transports, messages are sent over the peer-to-peer transports associated with the endpoint.

Subscribers are created on endpoints. If the endpoint is associated with a persistence store, messages are received from the persistence cluster that manages the persistence store. If the endpoint is associated with peer-to-peer transports, messages are received from each

transport. Recall that, to process messages, FTL client programs must create an event queue and a message callback, as described in [FTL Development Overview](#).

In general, if the endpoint is associated with a persistence store, the administrator must also associate the endpoint with a durable template, which describes how a durable subscription should be configured (for example, sync or async acks).

i Note: In advanced configurations, it is possible for an endpoint to be associated with both a persistence store and peer-to-peer transports, in which case the persistence store serves as a backup for the transport. See [Persistence: Stores and Durables in FTL Administration].

By default, subscribers receive every message sent via a persistence store or peer-to-peer transport. In general you will need to segregate traffic for unrelated FTL client programs. There are a few possibilities:

- The administrator may define multiple persistence stores or transports for use by different FTL client programs.
- Subscribers may use a content matcher to filter out unnecessary messages. See [Delivery](#) for the semantics of content matchers.
- Inbox subscribers are a type of subscriber that only receive messages sent to the subscriber's inbox. An inbox is an object that can be included as a field in FTL messages. The replying application receives an FTL message, retrieves the inbox from the FTL message, and sends a reply to the inbox.

Key-value maps are created on endpoints. In this case the endpoint must be associated with a persistence store, since the map is managed by a persistence cluster. The administrator must associate the endpoint with a last-value durable template.

Application Design

Structuring Programs

These steps outline the main structural components of most application programs that communicate using TIBCO FTL.

i Note: For Go program structure, see [Structuring Programs in Go](#).

Procedure

Task A Initializing TIBCO FTL Objects

1. Create a realm object by connecting to an FTL server. (C programs must first open the top-level FTL object, and then a realm object.)

Task B Defining Callbacks

2. Define callbacks to process inbound messages.
3. Define callbacks to process advisory messages, as needed.
4. Define callbacks to handle out-of-band notifications.
5. Define callbacks for timer events, timer completion, and queue completion, as needed.

Task C Sending Messages

6. Define methods to construct outbound messages.
7. Instantiate endpoints as publisher objects.
8. Arrange to call the send methods of publishers.

Programs usually call send methods in the context of a data-generation loop, or in the context of message callbacks, or both. (You can use timer callbacks to implement a data-generation loop.)

Task D Receiving Messages

9. Instantiate endpoints as subscriber objects. (With a durable, supply a name property.)
10. Create event queues, and add each subscriber to an event queue.
11. Start a loop to dispatch event queues.

Task E Recovery and Clean-Up

12. Recover from administrative disable (see [Recovery of a Disabled Process Restart versus Reopen the Realm](#)).
13. Exit cleanly (see [Clean-Up](#)).

Structuring Programs in Go

Application programs in the Go language differ in several ways from programs in other languages. These steps outline the main structural components of most Go application programs that communicate using TIBCO FTL.

Summary of Differences

- Go does not support FTL timer events.
- Queue objects do not require explicit dispatch.
- Subscribe calls are methods of event queue objects, and these methods automatically add the new subscriber objects to the queue.
- When an inbound message arrives, the queue places the message on a subscriber channel, where your message processing methods can receive it.
- FTL inline mode is not relevant to Go programs.

Procedure

Task A Initializing TIBCO FTL Objects

1. Create a realm object by connecting to an FTL server.

Task B Processing Events

2. Create channels for inbound messages, and define goroutines to process messages from those channels.
3. Create channels for advisory messages, and define goroutines to process advisories, as needed.
4. Create a channel for out-of-band notifications, and define a goroutine to process notifications.
5. Launch the goroutines that poll message channels.

Task C Sending Messages

6. Define structs for marshaling data to and from messages.
7. Define methods to construct outbound messages.
8. Instantiate endpoints as publisher objects.
9. Arrange to call the send methods of publishers.

Programs usually call send methods in the context of a data-generation loop, or during message processing, or both.

Task D Receiving Messages

10. Create an event queue object.
11. Create subscriber objects.

Supply a channel as an argument. (The goroutines that process messages on the channel are already running.)

With a durable, supply a name property.

Task E Recovery and Clean-Up

12. Recover from administrative disable (see [Recovery of a Disabled Process Restart versus Reopen the Realm](#)).
13. Exit cleanly (see [Clean-Up](#)).

Application Instance Identifier

Administrators can arrange transports to implement endpoints in different ways, and select among them at run time using an *application instance identifier*.

Application programs can obtain that identifier from the command line, a configuration file, or an environment variable, and then supply that identifier as a property value in the realm connect call.

Design this ability into all application programs from the outset, even if you do not expect to use it. Coordinate with administrators to agree upon a standard way for programs to obtain identifiers. See the "Durable Coordination Form" in the FTL [Product Guides](#) list.

Callbacks and Recursive Dispatch

When designing application callbacks, do not recursively dispatch the same event queue.

Illegal Dispatch of queue A invokes a callback that dispatches queue A.

Legal Dispatch of queue A invokes a callback that dispatches queue B.

Illegal Dispatch of queue A invokes a callback that dispatches queue B, which invokes a callback that dispatches queue A again.

Inline Mode

Programs that receive time-sensitive messages can use inline mode to favor low latency over high throughput. Inline mode reduces inbound message latency by consolidating transport I/O and message callback processing into one thread.

For a complete description of inline mode, its usage, requirements and restrictions, see [Inline Mode](#).

For background information to help you better understand inline mode, read all of [Delivery](#).

Do *not* use inline mode unless callbacks *always* return quickly (see [Callback Restrictions with Inline Mode](#)).

When specifying inline mode, programmers and administrators must coordinate to avoid illegal state exceptions. See:

- [Transport Restrictions with Inline Mode](#),
- "Application Coordination Form" in the FTL [Product Guides](#) list

Request/Reply

Request/reply is an important communication pattern. Proper use of this pattern places requirements on the runtime environment.

- The replying process must be operational before the requesting process sends requests.
- The transports that carry the requests and replies must have already established operational communications before either process sends a request or a reply.
- The replying process must reply to each message as received directly from the requesting process, using the `SendReply` call.

Design programs to ensure robust behavior in situations where these requirements are *not* satisfied.

For example, a program that sends a request and then waits indefinitely for a reply is fragile. If the replying process is not available, or the transport's communications are not operating, then the request message does not reach its intended recipient, and the requesting process does not receive any information about this failure.

A more robust program would repeat the request at intervals until it receives a reply, or until it exceeds some reasonable limit.

Recovery of a Disabled Process Restart versus Reopen the Realm

When the FTL server disables an application process, the application can recover in one of two ways: either create a new objects and resume operation, or exit and restart.

The appropriate course of action depends on business factors and program design factors:

- *Process restart cost.* The program's start sequence might be time-consuming, or use resources heavily. For example, a start sequence might load a large data set into process memory, or establish several database connections. Seek to avoid that restart cost by re-using the existing context and state.
- *Downtime cost.* Business factors might require minimal interruption of an application process.
- *Parallel activities.* An application might do some activities which do not require TIBCO FTL communication. It could be appropriate for those activities to continue non-stop operation while the program creates a new realm object. For example, a program could control machinery and also send sensor data to other applications. In this example the controller thread must not stop, even if sensor messages temporarily stop flowing.
- *Simplicity.* Unless any of the preceding factors prohibit it, in many cases the simplest and fastest action is to exit and restart the program.
- *Fresh start.* An application's state could become stale while communications are suspended. Returning to a valid state might be more complicated or time-consuming than starting fresh.

Disabled Advisory

When the FTL server disables an application process, the process receives a [CLIENT_DISABLED](#) advisory. This advisory signals the need for recovery action.

Clean-Up

Design programs to behave appropriately when they exit, and when they create a new realm object and its associated objects. (For example, creating new realm objects could be a recovery path after the FTL server disables the client process.)

Proper clean-up serves two goals:

- The FTL server and administrators can distinguish application processes that exit cleanly from processes that exit abnormally. When a process cleans up, you can infer that it exited deliberately, rather than unexpectedly.
- Clean-up manages the resources within a process that closes and reopens the realm but does not exit.

Program Exit

When a program exits, it is important to close the realm object. This step closes the FTL server's connection to the program process. The FTL server stops monitoring the process.

When a program exits, it is not necessary to close or destroy other objects associated with the realm object.

Nonetheless, in C programs, you must close the top-level FTL object as well as the realm object.

Close and Reopen the Realm

In contrast, the clean-up requirements are different when a program does not completely exit, but instead creates a new realm object. In this situation, programs must do these steps, in order:

Close or destroy objects associated with the defunct realm object: including messages, property objects, field reference objects, content matchers, subscribers, publishers, inboxes, and event queues.

Close the defunct realm object.

Open a new realm object (that is, connect to the FTL server).

Create objects associated with the new realm.

Consolidation with Loose Coupling

With process transports, program threads within a process can communicate with one another by publishing and subscribing to messages. The main use case for process transports is consolidation of several application processes into one process, to boost performance by decreasing message latency.

To understand the concept of consolidation, consider a purely administrative first step toward the goal of decreasing latency. The administrator can run cooperating application processes on a single host computer, where they can communicate using a shared memory transport.

A second step is analogous, but involves programming. The developer can consolidate the loosely-coupled modules of a distributed application into a single program, where they run in separate threads that communicate using a process transport.

When consolidating application modules, you may retain the same endpoint structure as before consolidation. That is, if two modules each used a separate endpoint name before consolidation, the consolidated program can continue to use the same two endpoint names, and the administrator can bind them both to the same process transport. (Notice that this endpoint structure reflects the loose coupling among the module threads.)

Tip: Avoid Copying Messages

For best performance with a process transport, avoid the need to copy messages:

- Publish messages to only one subscriber, in one receiving thread. Publishing a message to two or more subscribers in two or more receiving threads requires a separate copy of the message for each receiving thread.
- Set the publisher property to release messages to send, and set the subscriber property to release messages to callback.

Delivery

Inbound Message Delivery

These six phases define the precise terminology that describes the delivery of an inbound message:

Phases of Inbound Message Delivery

	Phase	Description
1	<i>Receive</i>	A transport presents a message to a subscriber.
2	<i>Match</i>	If a subscriber has a content matcher, it filters the message. A matching message advances to the next phase. A message that does not match is discarded.
3	<i>Distribute</i>	The library adds the message to the event queue associated with the subscriber.
4	<i>Dispatch</i>	A dispatch call in your program removes messages from the event queue, and invokes the appropriate callback.
5	<i>Process</i>	Your callback gets data from the message and uses it.
6	<i>Release</i>	The callback returns, releasing the message and its resources.

Phases, Pipelines, and Threads

Consider the six delivery phases as two pipelines. Event queues serve as a stopover point between the two pipelines:

- The *receive pipeline* includes the receive, match, and distribute phases.
- The *dispatch pipeline* includes the dispatch, process, and release phases.

Specific threads do the work that moves a message through each pipeline. Each thread requires CPU resources to do that work.

- One or more *receive threads* drive the receive pipeline. Inbound data arrives on a

transport's data source.

- In the receive phase, receive threads pump messages out of those data sources.
- In the match phase, receive threads filter messages through the content matcher.
- In the distribute phase, receive threads place messages on the appropriate event queues.
- One or more *dispatch threads* drive the dispatch pipeline.
 - In the dispatch phase, dispatch threads pump messages out of event queues.
 - In the process phase, dispatch threads process messages using application callbacks.
 - In the release phase, dispatch threads reclaim message resources.

The functional definition of a dispatch thread is any program thread that calls an event queue's dispatch method. Your application program determines the number of dispatch threads required to dispatch and process messages from all its event queues in a timely and efficient manner. (Developers: record the number of dispatch threads on each programs' application coordination form.)

With regular event queues, receive threads are internal to the TIBCO FTL library, and the library determines the number of receive threads. In contrast, with inline event queues, the application's dispatch thread also serves the functional role of a receive thread, as it combines all six phases into a single pipeline (see [Inline Mode](#)).

Both pipelines begin by polling for available messages: the receive phase polls a transport's data source, and the dispatch phase polls an event queue. For the details of polling, see "Receive Spin Limit" in TIBCO FTL [Administration](#).

Content Matchers

A content matcher filters the inbound stream of messages for a subscriber.

Content matchers operate during the match phase of message delivery. (Also see [Subscriber Interest](#).)

A subscriber can have at most one content matcher. A subscriber without a content matcher distributes all inbound messages to its event queue: it does not filter out any

messages. In contrast, a subscriber with a content matcher distributes only the messages that match it.

An inbox subscriber cannot have a content matcher. An inbox subscriber always receives all the messages sent to its inbox.

Each content matcher has a *match string*, which defines its filtering action. The match string specifies a set of field names paired with corresponding values. Fields can be matched as true (value present), false (value absent), a string value, or a long value. (see [Match Semantics](#)).

Content matchers can be constructed to match on the following which removes the need to construct multiple subscribers.

- Match on all fields (a logical AND match): The match string matches a message if and only if every field specification in the match string correctly matches the presence, absence, or value of a corresponding field in the message. For example, a subscriber with the following interest would receive messages only if the bank is ABCBank and the type is debits. The subscriber would not receive the message if the publisher were sending a message with bank ABCBank and type of credits.

```
{“bank”: “ABCBank”, “type”: “debits”}
```

which can be read as:

("bank" is "ABCBank") AND ("type" is "debits")

- Match on one or more fields (a logical OR match): The match string matches a message if *any* field in the match string matches the presence, absence, or value of a corresponding field in the message. For example, a subscriber with the following interest would receive messages if the bank is ABCBank and the type is debits or credits.

```
[{"type": "debits"}, {"type": "credits"}]
```

or this shorter syntax:

```
{"type": ["debits", "credits"]}
```

which can be read as (any bank):

("type" is "debits" OR "type" is "credits")



Caution: A logical OR match is not permitted on the key field in the matcher used by a last-value durable subscriber. A subscriber must specify exactly one string value for the key field.

- A combination of AND and OR matching For example:

```
{“bank”: “ABCBank”, “type”: [“debits”, “credits”]}
```

 which can be read as:

("bank" is "ABCBank") AND ("type" is "debits" OR "type" is "credits")

i Note: There are some restrictions on last-value durables with regard to the key field. See [Last-Value Durables](#), "Content Matchers".

Match Semantics

Content matchers interpret match strings using these simple rules.

- If the match string specifies a field with boolean token `true`, that field must be *present* in the message in order to match.
- If the match string specifies a field with boolean token `false`, that field must be *absent* from the message in order to match.
- If the match string specifies a field with either a string or long integer value, that field must be present in the message with that value.
- When comparing a field name or a value, all comparisons must be exact. Matching does not support wildcards nor regular expressions.

Match String Syntax

Construct matchers on this syntactic template in JSON format.

Logical AND Syntax

field:value pairs are in curly braces and separated by commas). Each field name can appear at most once.

```
{"fieldname1":"value1" , ... , "fieldnameN":"valueN"}
```

Logical OR Syntax

A logical OR can be expressed in one of two ways:

1. As a list of possible values for a given field.

```
{"f1": ["v1", "v2"]}
```

2. As a list of valid matchers, which need not be related to each other, and each could be a logical OR.

```
[{"f1": "v1"}, {"f1": "v2"}, ..., {"f1": "vn", "f2": "w1"}]
```

```
[{"f1": ["v1", "v2"]}, [{"f2": "w1"}, {"f2": "w2"}]]
```

Other Syntax Rules

Logical AND as well as logical OR syntax follow these rules:

- Enclose field names and string values in double-quote characters. You may precede quote characters with an escape character, as needed.
- Do *not* enclose boolean tokens in double-quote characters.
- Values can be long integers, strings, or the special boolean tokens `true` or `false`.
- When *value* is a string, its maximum length is 256 characters (see [Restrictions on Names](#)).
- Whitespace is ignored, except within double-quote characters.

Content Matcher: Match String Examples

```
{"Symbol":"TIBX"}
{"Symbol":"TIBX","Order":"Buy","Price":"Market"}
{"Item":"Book","Title":"The Power of Now"}
{"Answer":42}
{"MustHave":true,"MustNotHave":false,"MustBeZero":0}
```

Content Matcher Performance and Architecture

For best performance, match against no more than 10 fields. (Fewer is faster.)

If your application requires regular expression matches, numeric value ranges, or more than 10 match fields, then the best practice is to narrow the field of messages with a content matcher, and then code your callback to refine the filtering even further.

Subscriber Interest

Whenever possible, publishers send messages only to interested subscribers. That is, publishers send only those messages that match at least one subscriber content matcher. This behavior can reduce bandwidth consumption, and can enhance subscriber performance in some situations.

To support this feature, subscribers send content matchers to publishers, which apply them to *outbound* messages.

Inline Mode

Programs that receive time-sensitive messages can use inline mode to favor low latency over high throughput. Inline mode reduces inbound message latency by consolidating transport I/O and message callback processing into one thread.

Before using inline mode, you must thoroughly understand the important details in this topic and the topics that follow.

Background

The term *inline mode* refers to the execution thread of the six phases of message delivery (see [Inbound Message Delivery](#)).

In regular operation the first three phases are separate from the last three phases. That is, the receive, match, and distribute phases occur in an internal receive thread, separate from program code. Meanwhile, the dispatch, process, and release phases occur in a dispatch thread, starting with a dispatch API call within application program code. The event queue is a loose coupling between these two triplets of phases: namely, the receive and dispatch pipelines.

In contrast, inline mode co-locates all six phases in the same thread. That is, the entire phase stack operates as a single I/O pipeline from the underlying transport, through the application program's dispatch call, to message processing and release. All six phases occur within one program thread, at the application's dispatch call. This tactical optimization eliminates a source of latency, namely, context-switching overhead between the distribute and dispatch phases. However, this tight coupling introduces several restrictions and potential pitfalls, which become the responsibility of the application programmer.

Callback Restrictions with Inline Mode

Inline mode requires that callbacks *always* return quickly. Otherwise, long callbacks could delay message I/O, which would defeat the purpose of inline mode.

For example, the negative consequences of long callbacks could include a large I/O backlog. As a result, the receiving program could lose inbound messages. Furthermore, message buffers at the publishing end of a transport could fill up. To prevent overflow at these buffers, the transport could block programs at the publishing end, delaying their outbound messages.

Transport Restrictions with Inline Mode

Inline mode restricts underlying transports. Conformance with the following restriction must be ensured.



Restriction: When a transport is associated with an *inline* event queue, it cannot be associated with any other event queue.

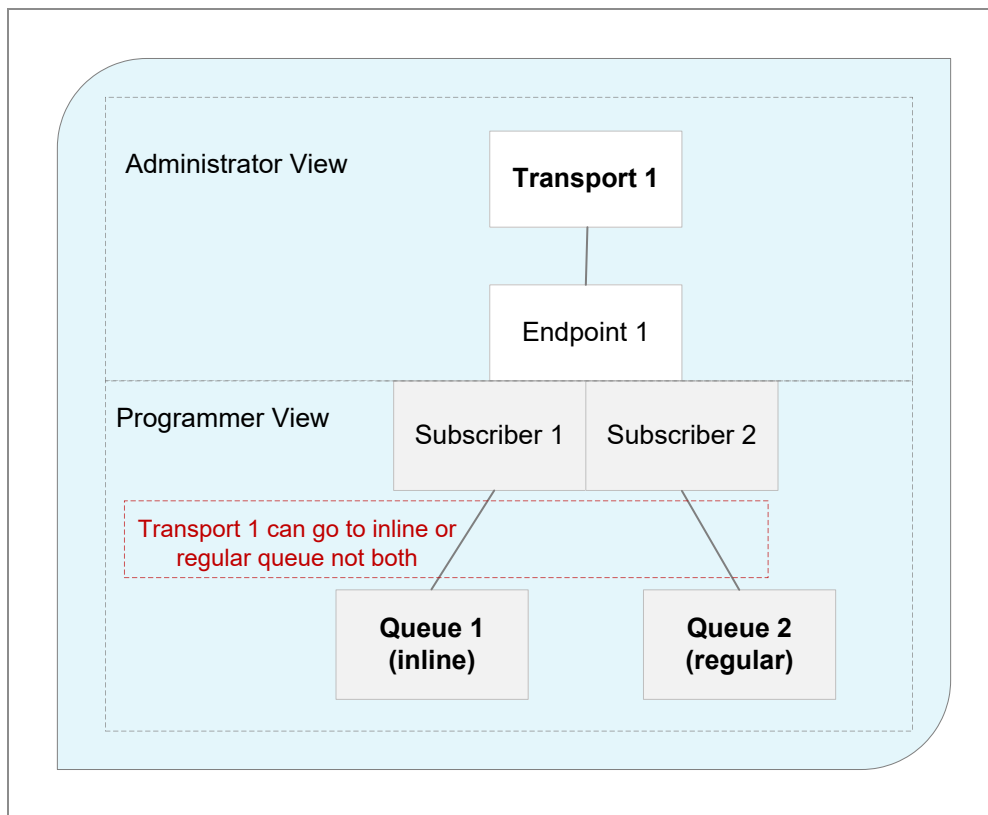
Violating the restriction results in an illegal state exception. The add subscriber API call throws the exception when the program attempts to add a subscriber that would violate the principle.

When specifying inline mode, programmers must coordinate with administrators to avoid illegal state exceptions.

Programmer Violation

The first diagram illustrates a violation attributable to programmer error. The programmer created two subscribers (S1 and S2) from a single endpoint (E1), and adds them to two event queues, at least one of which (Q1) is an inline queue. This situation would associate the underlying transport (T1) of the subscribers with two incompatible queues (Q1 and Q2), violating the restriction.

Figure 1: Illegal Inline Queue, Simple Case



To rectify the violation, either add both subscribers to a single inline event queue (Q1), or to two *ordinary* event queues, neither of which use inline mode.

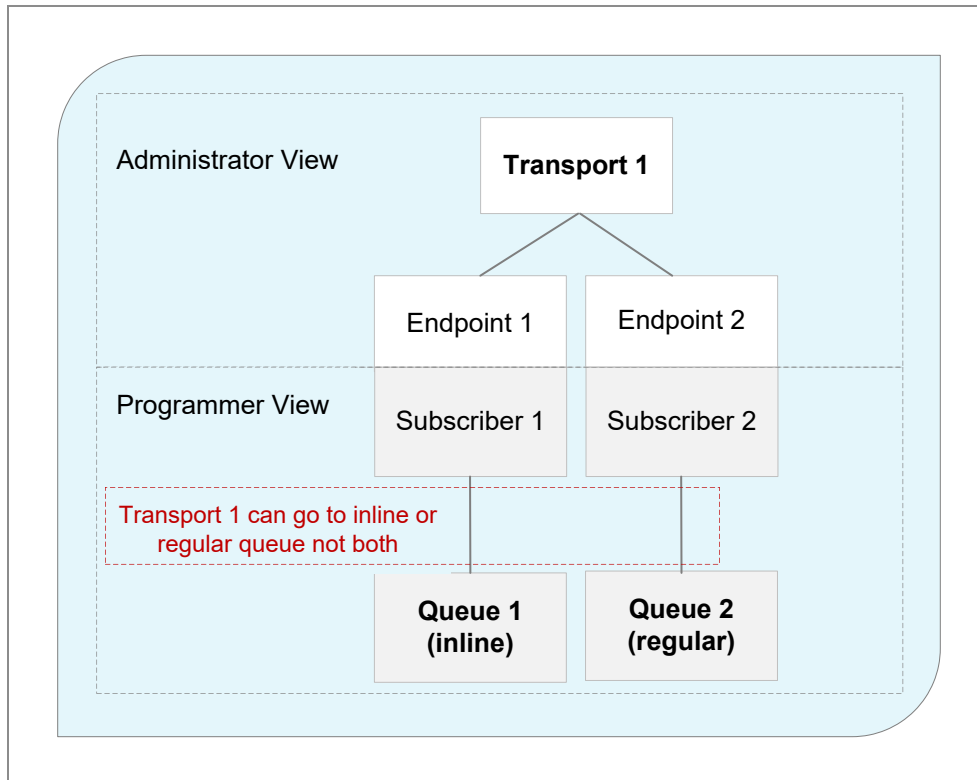
Hidden Violation

While it would appear straightforward for programmers to avoid the violation of the simple case of the first diagram, the situation is more complicated because responsibilities are divided between programmers and administrators.

From a programmer's perspective, transports are ordinarily hidden behind endpoint names. Conversely, administrators can ordinarily employ transports to implement endpoints without understanding the specifics of subscribers and queues in application programs. However, inline mode tightly couples transport I/O to program code, which in turn blurs the usual separation of responsibilities between programmer and administrator. As a result, the restriction requires that programmers and administrators coordinate whenever a program uses inline event queues.

Consider the following example. The situation in the second diagram might appear correct from the separate perspectives of the programmer and the administrator, but combining their perspectives reveals a hidden violation.

Figure 2: Illegal Inline Queue, Hidden Violation



The programmer instantiates two subscribers (S1 and S2) on two separate endpoints (E1 and E2), and associates them with separate queues (Q1 and Q2). Q1 is an inline queue.

Meanwhile, the administrator employs a single transport (T1) to implement the two endpoints (E1 and E2).

Because T1 is associated with Q1, which is an inline queue, this arrangement violates the principle. When the program adds S2 to Q2, the add subscriber call throws an exception, whether or not Q2 uses inline mode.

By coordinating their information, the programmer and administrator can cooperate to avoid the violation in any of three ways:

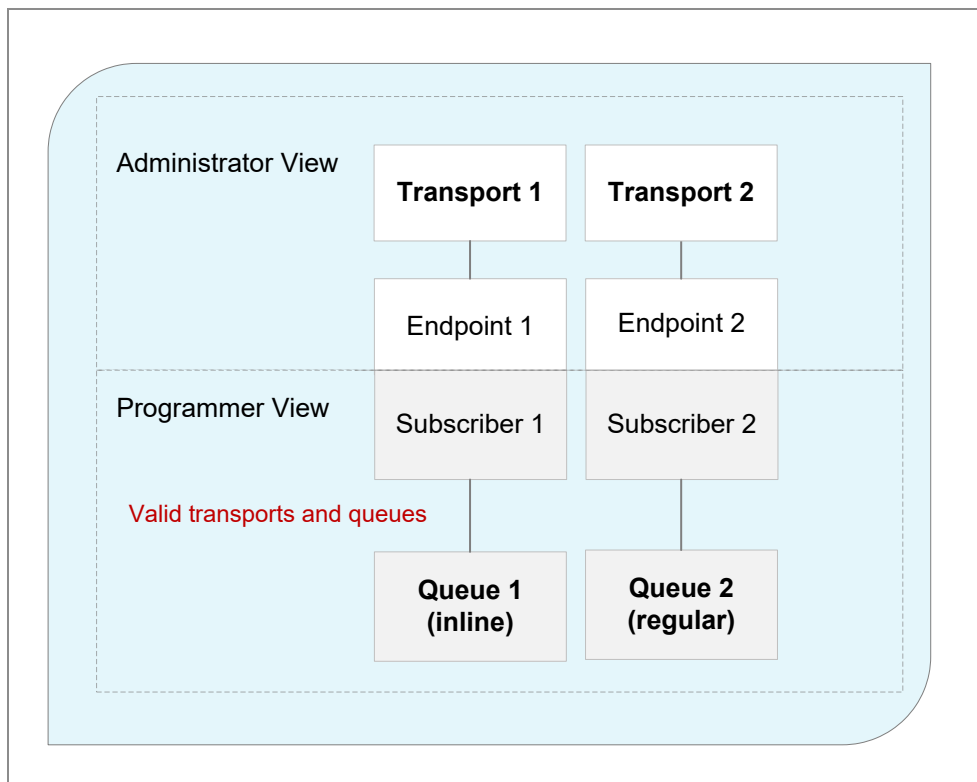
- Use ordinary queues instead of inline mode.
- Use separate transports to implement the two endpoints E1 and E2 (see [Separate Transports to Correct a Violation](#), which follows).

- Add subscribers S1 and S2 to only one inline queue (see [Single Queue to Correct a Violation](#), which follows).

Separate Transports to Correct a Violation

In the third diagram, transport T1 is associated with only one inline event queue, Q1. This arrangement does *not* violate the restriction.

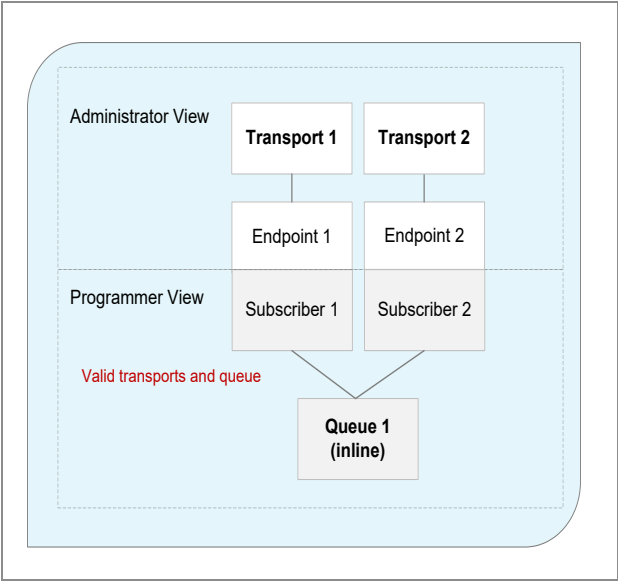
Figure 3: Legal Inline Queue, Separate Transports



Single Queue to Correct a Violation

In the fourth diagram, even though two subscribers (S1 and S2) share the same underlying transport (T1), they are associated with the same inline event queue (Q1). This arrangement does *not* violate the restriction.

Figure 4: Legal Inline Queue, Single Queue



Direct Publishers and Subscribers

Direct publishers and subscribers use direct memory access to achieve the fastest data transfer and lowest latency.

A publishing application writes data directly into memory that is accessible to subscribing applications. Subscribing applications read data directly from memory written by the publishing application.

The implementation minimizes or eliminates sources of latency.

The following table summarizes the differences between regular and direct publishers and subscribers. Subsequent topics contain greater detail.

Aspect	TIBCO FTL Publishers and Subscribers	Direct Publishers and Subscribers
Messages	Applications exchange message objects.	Applications exchange data in buffers.

Aspect	TIBCO FTL Publishers and Subscribers	Direct Publishers and Subscribers
Structuring Data	Applications structure message objects using fields that contain typed data.	Applications structure data within buffers using an array of sizes.
Copying Data	The library copies message data as needed.	The library eliminates copying data wherever possible. The sending application populates a data buffer, and receiving applications have direct access to that same data buffer.
Memory Management	Applications can create and destroy publisher and subscriber objects. The library and application programs can both create and destroy message objects.	Applications create direct publisher and subscriber objects, but cannot reclaim their memory. Application programs neither allocate data buffers, nor free them.
Subscribers and Threads	Subscribers are thread-safe.	Applications must not use a direct subscriber in more than one thread. Instead, applications can create a separate direct subscriber for each thread.
Delivery	The library distributes inbound messages through event queues.	Applications dispatch inbound data buffers directly to callback methods.
Content Matchers	Applications can use content matchers to select a sub-stream of messages.	Applications receive the entire data stream. Selectivity based on data content is not available.

Aspect	TIBCO FTL Publishers and Subscribers	Direct Publishers and Subscribers
API Language Support	The API supports multiple programming languages.	The API supports only the C language.
Protection	The implementation protects against many common types of programming errors.	The implementation protects against fewer types of errors. For example, programmers must be careful to not overrun the length of data buffers, because the library does not guard against this error.
Publishers	Many publishers can send through an endpoint.	A direct shared memory bus supports only one direct publisher object.
Transport	Choose from a variety of transport types.	Only a direct shared memory transport can implement a direct publisher or subscriber endpoint.
Persistence	Available.	The persistence feature does not support direct publishers and subscribers.

Use Cases for Direct Publishers and Subscribers

Direct publishers and subscribers are appropriate only in very specialized use cases in which a few nanoseconds of latency can make a critical difference.

Direct publishers and subscribers are less flexible than regular TIBCO FTL communications. Direct publishers omit several features in exchange for greater speed and lower latency:

- The implementation has fewer protections against incorrect use of the API calls. For example, it does not check arguments nor data types. The implementation does not protect against writing beyond buffer boundaries.
- Data is in buffers, rather than in messages with typed fields.

- Event queues are not available.
- Content matchers are not available.
- Persistence is not available.
- C is the only supported programming language.

Direct shared memory transports support only one direct publisher per bus. This restriction precludes the request/reply pattern, which requires two-way communication. Nonetheless, you could use a separate bus for replies.

Programming with Direct Publishers

Create a direct publisher object. Code a *reserve-fill-send* loop. Close the direct publisher and exit.

Number of Publisher Objects

A suite of application processes that communicate over a direct shared memory bus may contain *only one* direct publisher object at a time. It is the shared responsibility of the application developers and the administrators to ensure this condition.

One Reserved Buffer

A direct publisher can reserve at most one buffer at a time.

- If direct publisher has already reserved a buffer in one thread, any subsequent reserve calls in other threads block until the direct publisher has sent the reserved buffer.
- Do *not* call reserve twice within the same thread without an intervening send call.

The Reserve-Fill-Send Loop

Avoid delays in the reserve-fill-send loop: as soon as the reserve call returns a buffer, fill it with data, and send it.

Procedure

1. Create a direct publisher object.

The object will exist in memory until the process terminates.

Programs may use a direct publisher in multiple threads. However, contention to reserve a buffer can decrease performance and increase latency.

The reserve-fill-send loop begins here.

2. Reserve a buffer.

Supply the total size of the data, and the number of data elements.

The reserve call returns a pointer to the buffer.

If the program specifies more than one data element, the reserve call also yield a size array.



Note: One reserve call can reserve a maximum block of 64 kilobytes. The data buffer *and* the size array must fit together within that maximum block.

3. Fill the buffer with data.

If another thread has already reserved a buffer with this direct publisher, this reserve call blocks until that other thread sends its buffer.

- **One Data Element:** Write the data into the buffer.
- **Multiple Data Elements:** Use a loop to write each data element into the buffer, and its corresponding size, in bytes, into the size array.

4. Send the reserved buffer.

The send reserved call makes the buffer available to direct subscribers.

After this call, the direct publisher is ready to reserve another buffer.

5. Loop to reserve the next buffer.

Clean up before the application process exits.

6. Close the direct publisher object.

Closing a direct publisher causes any blocked reserve calls to return. Closing also invalidates the direct publisher, which disallows any subsequent method calls.

However, closing a direct publisher does *not* free its memory.

Programming with Direct Subscribers

Create a direct subscriber object. Code a callback method. Code a *dispatch* loop. Close the direct subscriber.

Procedure

1. Code a direct callback method.

Within the callback, loop to unpack the data items.

Avoid introducing latency within the callback.

Programs can use the same callback with more than one direct subscriber object, and in more than one thread.

2. Create a direct subscriber object.

The object will exist in memory until the process terminates.

Programs may use a direct subscriber in multiple threads. However, contention to dispatch a direct subscriber object can decrease performance and increase latency.

3. Code a dispatch loop for the direct subscriber.

A dispatch loop repeatedly calls the dispatch method.

The dispatch method checks whether inbound data is available for a direct subscriber object.

- **Data** - If a data buffer is available, dispatch invokes the callback method, passing the buffer and a closure object as arguments to the callback. When the callback returns, the direct subscriber automatically acknowledges that it has received the buffer. Then the dispatch call returns.
- **No Data** - If a data buffer is not available, dispatch waits until data becomes available, or until the timeout expires.

You can supply a closure object that is specific to the direct subscriber object, or you can supply a global closure object common to more than one direct subscriber.

Clean up before the application process exits.

4. Close the direct subscriber object.

Closing a direct subscriber causes any executing dispatch calls to return immediately. Closing also invalidates the direct subscriber, which disallows any subsequent method calls.

However, closing a direct subscriber does *not* free its memory.

Multithreading with Direct Publishers and Subscribers

On hardware with multiple processor cores, multithreading can boost performance of applications that communicate using direct publishers and direct subscribers. You can use multithreading to implement a variety of communicating sub-applications within a single process, and to decrease context-switching latency as they communicate.

For best results, match the size of a multithreaded application program to the host computer hardware. Allot one processor core for each direct publisher and its reserve-fill-send loop, and one processor core for each direct subscriber object and its dispatch loop. Too few cores can increase latency.

i Note: A suite of application processes that communicate over a direct shared memory bus may contain *only one* direct publisher object at a time. It is the shared responsibility of the application developers and the administrators to ensure this condition.

Multiple Direct Subscribers

A program can create more than one direct subscriber on an endpoint. Each direct subscriber object receives every buffer published on the bus.

- You can use multiple direct subscribers to process the same inbound data stream with different callback semantics.
- You *cannot* use multiple direct subscribers to increase throughput by dividing an inbound data stream.

✓ Tip: In multithreaded application programs, use a separate direct subscriber object and a separate dispatch loop in each thread.

For best results, if an application program uses more than one direct subscriber object, create them all before starting the any of their dispatch loops.

**Tip:**

Programs may use a direct subscriber in multiple threads. However, contention to dispatch a direct subscriber object can decrease performance and increase latency.

Create only a finite number of direct subscriber objects. Re-use them as needed. Close them only when cleaning up to prepare for the process to exit.

Coordination Forms

Coordination forms help developers and administrators to agree upon application details and to document those details.

From the FTL [Product Guides](#) list, you can download the PDF forms listed below, rename the copies, and complete them online. Alternatively, you can print the downloaded forms and complete them on paper.

- Application Coordination Form
- Durable Coordination Form
- Endpoint Coordination Form
- Format Coordination Form

IPv4 and IPv6

TIBCO FTL software supports both IPv4 and IPv6.

TIBCO FTL components accept either IPv4 and IPv6 addressing in the following cases:

- FTL server GUI fields configuring these transport parameters:
 - RUDP transport: Host field
 - Static TCP transport: Host field
 - Dynamic TCP transport: Subnet Mask field
- Arguments to client API calls that specify client connections to FTL servers
- Values that specify ports, either in FTL server configuration files, or in FTL server administration utility command line arguments

All other cases accept only IPv4 addressing.

Notation

Specify IPv4 addresses in 4-part dot-decimal notation.

Specify IPv6 addresses in hexadecimal notation, with colon (:) separators, surrounded by square brackets. Square brackets *must* be present.

In situations where it is correct and meaningful to specify *star* (that is, asterisk, *) as a wildcard address, TIBCO FTL software interprets this wildcard as *both* an IPv4 wildcard *and* an IPv6 wildcard. (That is, it listens on all relevant interfaces in *both* protocols.)

You can specify the loopback address in IPv4 as either `localhost` or `127.0.0.1`, and in IPv6 as `[::1]` or an equivalent notation.

Endpoints

An endpoint is an abstraction that represents a set of publishers and subscribers in communicating programs.

Programming

Within an application program, an endpoint begins as a name. Program code uses the endpoint name to create endpoint instances, that is, publisher and subscriber objects. Then programs use the publishers to send messages, and the subscribers to receive messages.

Each subscriber object and each call to a publisher send method introduces an ability requirement (see [Abilities](#)). Application architects and developers record these requirements in endpoint coordination forms. See the "Endpoint Coordination Form" in the FTL [Product Guides](#) list.



Tip: For many applications or communicating application suites, a single endpoint name suffices.

Configuration

Administrators configure transport connectors to satisfy those ability requirements. Connectors bind transports to endpoints. Those transports implement the endpoints, transferring message data from publishers to subscribers.

Administrators can also view an endpoint as a complex entity, embracing four separate communication abilities. A transport connector can separately enable any subset of those four abilities.



Tip: The default application definition is a valid configuration for applications that use only one endpoint.

Abilities

An endpoint has four communication abilities. A separate aspect of the API embodies each of these abilities. The following table summarizes the four abilities.

Each of these four abilities and the API call that embodies it correspond to a distinct sub-stream of messages, called an *ability sub-stream*.

Endpoint Abilities: Definitions

Ability	API	Description
One-to-Many Receive	Subscriber object	Carries one-to-many messages inbound to the endpoint's subscribers in the application.
One-to-One Receive	Inbox subscriber object	Carries one-to-one messages inbound to the endpoint's inbox subscribers in the application.
One-to-Many Send	Send call of a publisher object	Carries one-to-many messages outbound from the endpoint's publishers in the application.
One-to-One Send	Send-to-inbox call of a publisher object	Carries one-to-one messages outbound from the endpoint's publishers in the application.

Notice that each ability combines two dimensions: *reach* and *direction*:

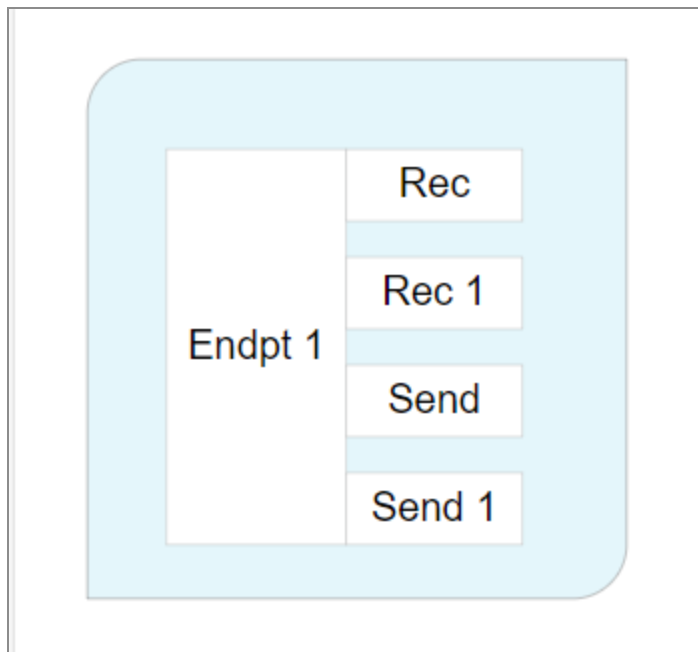
- *Reach*
 - *One-to-many abilities* carry messages that can reach *potentially many* subscribers.
 - *One-to-one abilities* carry messages that can reach *one* specific *inbox* subscriber.
- *Direction*
 - *Receive abilities* carry messages *inbound to* an application.
 - *Send abilities* carry messages *outbound from* an application.

Endpoint Abilities Matrix

	Inbound	Outbound
One-to-Many	Receive	Send
One-to-One	Receive Inbox	Send Inbox

Administrators can configure these abilities in the FTL server GUI (see [Endpoint Details Panel](#)).

Endpoint and its abilities are represented with the following shape. Here, Endpoint 1 has four abilities: Receive, Receive 1, Send, and Send 1.

Figure 5: Four Endpoint Abilities

Endpoints and Stores

When an application uses persistence stores and durables, endpoints play an additional role. Administrators can associate each application endpoint with at most one persistence store.

That persistence store serves all publisher and subscriber instances of the endpoint, in all process instances of the application.

Programs can supply either a durable name or a subscriber name to the create subscriber call, connecting the new subscriber object to a specific durable within the endpoint's persistence store.

Number of Endpoints

For many applications a single endpoint name suffices.

Minimizing the number of endpoints usually simplifies design, programming, and administration. Justify each additional endpoint based on requirements, for example:

- A functional requirement (such as forwarding)
- A business requirement to separate the data streams (such as security, or data priority)
- An administrative requirement to control the endpoints in different ways (such as efficiency, or flexibility)

Notice that *endpoint* is not analogous to the concept of *destination* in the Java Messaging System (JMS). Within a program, one endpoint name can yield several subscriber objects, each selecting a different subset of inbound messages. Furthermore, one endpoint can instantiate publishers as well as subscribers.

Tactics to Reduce Endpoints

If you think an additional endpoint might be necessary, consider whether these tactics might reduce that need, and simplify the situation:

- A program can create several publishers and subscribers from a single endpoint name.
- Administrators can define connectors to carry the four abilities of an endpoint in a way that either merges their sub-streams, or keeps them separate.
- Administrators can configure all application instance definitions to implement an endpoint using a common bus as a shared communication medium, or configure disjoint subsets of instances that use totally separate buses.

Send: Blocking versus Non-Blocking

Administrators determine the behavior of send calls when a transport bus is full. Program developers may advise administrators concerning the correct behavior.

Send calls in an application program place outbound messages on a bus, which is external to the program process. A transport mediates this I/O operation. Under normal circumstances, the I/O operation completes quickly, and the send call returns.

If the bus is *full*, that is, it cannot accept any more messages, then two behaviors are possible. Administrators select one of these two behaviors, and configure it in the transport definition.

Blocking and Non-Blocking Send

Behavior	Description
Blocking Send	When the bus is full, the send call blocks, and does not return until it has placed all the outbound message data on the bus.
Non-Blocking Send (Default Behavior)	When the bus is full, the transport buffers outbound messages in a backlog buffer in process memory until the bus can accept them. The send call returns. When the bus can accept more messages, the transport automatically moves them from its backlog buffer to the bus. If the backlog buffer overflows, the transport discards the oldest messages in the buffer to make room for new messages. The discarded messages do not arrive at slow receivers. The base library reports dataloss advisories to slow receivers.

Use [TIBCO FTL Endpoint Coordination Form](#) to document the correct behavior.

To configure this behavior in a transport definition, see “Transport Definition Reference.”

Messages

Message Objects in Program Code

The following categories describe messages and the ways that your program code can interact with them.

Message Origin

Origin is the way in which a message object comes into existence. The origin of a message within a program is either local or inbound:

- A message that program code creates using an explicit call is a *local* message.
- A message that a program receives through a subscriber is an *inbound* message.

Message Ownership

Either your program code owns a message, or the TIBCO FTL library owns it. Ownership depends in part upon message origin, but releasing a message can change its ownership.

Release of Message Ownership

Subscribers can release ownership of inbound messages. Such *released inbound* messages belong to your program code, effective upon dispatch to your callback.



Tip: Releasing messages is appropriate when a callback delegates message processing to a separate thread.

To enable release to callback behavior, the program must supply a property to the subscriber create call.

Message Ownership, Determining Factors

Origin	Released	Ownership
Local	No	Your program code owns a local message.
Inbound	No	The library owns an inbound message.
	Release to callback	Your program code owns an inbound message that the subscriber has released to the callback.

Destruction

The responsibility to destroy a message depends entirely on the current ownership of a message.

Message, Responsibility to Destroy

Ownership	Responsibility to Destroy
Program code	Program code
Library	Library

SubMessages

When program code owns a message that contains nested submessages, destroying the outer message does not automatically destroy submessages contained within its fields or arrays. To effectively reclaim storage, the program must first destroy all the inner submessages and then the outer message.

Message Access

Message origin and current ownership determine the time frame within which your program can safely access a message object and its data.

Message Access, Determining Factors

Origin	Released (Ownership)	Access
Local	No (program code owns)	Until your program destroys the message
Inbound	No (library owns)	Until your callback returns
	Released to callback (program code owns)	Until your program destroys the message

Message Mutability

Mutability indicates whether your program may modify a message:

- A program may modify a *mutable* message by adding, deleting, or changing field values.
- A program cannot modify an *immutable* message in any way.

Mutability depends entirely on the origin of a message.

Message Mutability Depends on Origin

Origin	Mutability
Local	A local message is mutable.
Inbound	An inbound message is immutable, even if the subscriber released it to the callback.

Copy of a Message

Programs can copy any message.

- The copy is local, even if the original message is inbound.
- Your program owns the copy, even if the library owns the original.
- Your program can pass the copy to another thread.
- The copy is mutable, because it is local. Modifying the copy does not affect the original.
- The copy, along with all of its data content, is completely independent of the original inbound message. Even if the library destroys the original, the copy remains accessible. (See also, [Storage of Field Values within Messages](#).)

Messages and Thread-Safety

Messages are *not* thread-safe. Ensure that only one thread at a time accesses a message and its data.

Field Values and Types

Messages contain data values in a set of fields. Structurally, a field pairs a name of type string with a value of a specific data type.

The format of a message defines the data type of each field that the message can contain.

Field Data Type Reference

A data type describes the value of a field within a message or a message format.

Administrators use the type names in the first column of the following table to define formats.

Programs denote the data type of message fields using the type constants in the second column of the table.

Field Data Types

Type Name (in Realm)	Type Constant (in Programs)	Description	Primitive
long	TIB_FIELD_TYPE_LONG	A long integer.	Yes
long_array	TIB_FIELD_TYPE_LONG_ARRAY	An array of long integers.	No
double	TIB_FIELD_TYPE_DOUBLE	A double-length floating point number.	Yes
double_array	TIB_FIELD_TYPE_DOUBLE_ARRAY	An array of double-length floating point numbers.	No
string	TIB_FIELD_TYPE_STRING	A UTF-8 string.	No
string_array	TIB_FIELD_TYPE_STRING_ARRAY	An array of UTF-8 strings.	No
opaque	TIB_FIELD_TYPE_OPAQUE	An opaque byte-array.	No
message	TIB_FIELD_TYPE_MESSAGE	A message (that is, the field's value is a submessage).	No
message_array	TIB_FIELD_TYPE_MESSAGE_ARRAY	An array of messages.	No
inbox	TIB_FIELD_TYPE_INBOX	An inbox.	No
datetime	TIB_FIELD_TYPE_DATETIME	A DateTime value.	No
datetime_array	TIB_FIELD_TYPE_DATETIME_ARRAY	An array of DateTime values.	No

Storage of Field Values within Messages

Values stored in a message field are part of the message, that is, they reside in memory associated with the message, and allocated by the library.

When a field's type is a primitive type, calls that get the field's value copy that value from the message. In contrast, when a field's type is non-primitive, calls that get the field's value return a pointer to memory within the message. (The fourth column of the table in [Field Data Type Reference](#) indicates primitive and non-primitive types.)

Programs must *not* modify non-primitive values that reside in memory associated with a message. For example, you must not set the value of a cell in a long array extracted from a message.

Furthermore, non-primitive values remain valid only for a limited time:

- A subsequent call that sets the value of that field invalidates a pointer returned by a preceding get call.
- Destroying the message invalidates all pointers returned by preceding get calls.

DateTime

Application programs can use the TIBCO FTL data type `DateTime` to represent date and time values with large range and nanosecond precision.

`DateTime` values combine two numeric components:

- The `sec` component represents whole seconds with up to 64 bits. Zero denotes the UNIX epoch: midnight entering January 1, 1970.
- The `nsec` component represents nanoseconds after the time that the `sec` component denotes. Although the data structure stores this component in a signed 64-bit integer, this component is always non-negative, between zero and 999,999,999 (inclusive).

For example, the value -1 seconds plus 999,999,998 nanoseconds represents December 31, 1969, 2 nanoseconds before midnight (that is, 2 nanoseconds before the epoch).

Formats: Managed, Built-In, and Dynamic

A *format* defines the set of fields that a message can contain: including field names and data types.

TIBCO FTL software supports three broad categories of format: managed, built-in, and dynamic formats. These categories have differing constraints and performance characteristics.

Dynamic Format

An application can define a format dynamically, as it constructs a message. That is, the sending program implicitly determines the set of fields and their data types as it sets field values.

When a program creates a message, and specifies an unrecognized format name (that is, neither built-in nor preloaded), then TIBCO FTL software automatically treats it as a dynamic format.

While easy to use, dynamic formats are inefficient because each message must be self-describing. That is, a dynamic format message necessarily includes its own format metadata. Expect larger message sizes, slower transmission, slower processing, and longer field access times.

Dynamic formats are useful for rapid prototyping projects. To improve performance, you can switch to defining the application's formats as managed formats when the project nears the production stage.

Unnamed Dynamic Format

A program can create a message with a dynamic format, but supply null as the format name. Such *unnamed dynamic formats* are single-use. Administrators cannot ever upgrade them to managed formats.

You can use unnamed dynamic formats as a convenience when you cannot determine the structure and content of a message in advance. Unnamed formats save you the effort of naming the formats and keeping track of the names.

The performance of unnamed dynamic formats is identical to named dynamic formats.

Managed Format

Application architects and developers determine the set of formats that an application needs. Administrators then define those managed formats globally.

Administrators make managed formats available to an application by including them as *preload formats* for the application.

Programs can use these managed formats to compose new messages, or to unpack inbound messages. A message with a managed format is small and fast. It does not carry format metadata because that metadata is available to applications from the FTL server.

For efficiency and for safety, administrators can use the `Manage All Formats` and `Dynamic Message Formats` parameters to restrict the formats available to some or all of the applications in a realm. That is, administrators can require that applications use only preload formats and built-in formats, and prohibit applications from using dynamic formats.

Built-In Format

TIBCO FTL software defines a set of highly optimized built-in formats, which are defined within the base library and always available to all applications. No administrative action is required to make them available.

The performance characteristics of built-in formats are similar to managed formats. When used properly, a built-in format can outperform a managed format.

For more information, see [Built-In Formats Reference](#).

Format Names

Format names must be unique within a realm, and ideally, throughout the network.

Format names cannot be the empty string.

Format names may contain space characters.

See also [Restrictions on Names](#).

Field Names

Field names must be unique within each format. (Nonetheless, two separate formats may use the same field name.)

Field names cannot be null, nor the empty string.

Field names may contain space characters.

See also [Restrictions on Names](#).

Flexibility in Formats and Messages

An application suite needs a set of message formats that embraces the various purposes for which it sends messages. Nonetheless, the simplicity of a smaller set of formats usually yields optimal results.

When a format defines many possible fields, and a program sets only a few field values in a message, the message remains small, bearing only those fields that the program actually sets.

At one extreme, you can define a single catch-all format with all possible fields. Application programs can use as few or as many fields as is appropriate, without incurring a performance penalty.

When designing the set of formats for an application suite, balance these competing principles:

- *Specificity* Define as many formats as the application needs. If an application sends messages for different purposes, and the content structure of those messages is different, then define separate and specific formats for those messages.
- *Compactness* Defining a small number of formats is a good practice. A large set of formats is more difficult to understand, to configure, to use properly, and to maintain as your application evolves. If a subset of the formats are similar, and their usage or meanings are similar, then consider merging them into one format that contains a *union* of their fields.

Field Access: Names and References

The APIs provide two versions of each message field accessor call: one accepts a field name, while the other accepts a field reference object.

Accessing a field by name involves string comparison operations, which can increase processing latency.

Access by field reference is more efficient than access by name. Field reference objects contain a field name, along with internal information that facilitates efficient access.

Programs can repeatedly use a field reference object to efficiently access a field, even across messages of different formats. For example, if formats A and B both have a field named `foo`, then a field reference object with field name `foo` accesses the correct field in messages of either format.

Reusing Message Objects

It is faster to reuse an existing message object than to destroy it and create a new one. Programs can gain efficiency by reusing message objects, whenever possible.

- If a program repeatedly sends the *exact same message content*, then the best practice is to create that message object only once, sending it many times.
- If a program sends many messages of the same format, with the same fields, but with *varying field values*, then the best practice is to reuse one message object, clearing and modifying individual field values appropriately before each send call.
- If a program sends many messages of the same format, but the *set of fields varies*, then the best practice is to reuse one message object, clearing all of its fields and setting new values before each send call.

A program that sends a limited variety of messages can combine these tactics for reusing message objects. For each send call, it can select an appropriate message from a set of pre-constructed message objects, and modify as needed. To determine the optimum way to combine these tactics, test performance empirically.

Built-In Formats Reference

Built-in formats are optimized for efficiency, and are always available. Administrators do not need to explicitly define them, nor specify them among an application's preload formats.

The C API defines global constants as listed in the following table.

The Java API defines these constants as fields of class `Message`.

The .NET API defines these constants as fields of class `FTL`.

Built-In Formats

Format Name Constant	Description
TIB_BUILTIN_MSG_FMT_OPAQUE	The message consists of a singleton field of type opaque. The constant TIB_BUILTIN_MSG_FMT_OPAQUE_FIELDNAME denotes the name of that singleton field. (The value of the constant is <code>_data</code>.) For the fastest and most efficient message transfer, limit the data payload to TIB_BUILTIN_MSG_FMT_OPAQUE_MAXSIZE (12000 bytes).
TIB_BUILTIN_MSG_FMT_KEYED_OPAQUE	This keyed opaque format enhances the opaque format with a key field of type string. Content matchers can match against the key field. The constant TIB_BUILTIN_MSG_FMT_KEY_FIELDNAME denotes the name of the string field. (The value of the constant is <code>_key</code>.) The constant value TIB_BUILTIN_MSG_FMT_KEY_MAXLEN is the maximum length, in bytes, of the key string field, including the null terminator character. The constant TIB_BUILTIN_MSG_FMT_OPAQUE_FIELDNAME denotes the name of the opaque data field. (The value of the constant is <code>_data</code>.) For the fastest and most efficient message transfer, limit the data payload to TIB_BUILTIN_MSG_FMT_OPAQUE_MAXSIZE (12000 bytes). Payload size includes the key, with its null terminator character, and the opaque data.

Opaque Format

Applications can use the built-in opaque format to exchange messages that contain unstructured data. Opaque messages are very efficient, and can achieve very low-latency performance.

Opaque messages are an excellent choice for streaming data.

Application code can also embed structured data within opaque messages, however, application code is entirely responsible for the structure: the application must pack the data at the sender, and unpack the data at the receiver.

Content matchers *cannot* select opaque messages. (Recall that content matchers require an exact match on all specified field names and field values. Opaque values vary too much for content matchers.)

To optimize performance, built-in opaque formats limit the size of the data payload. If you need to send larger payloads, define a static format with an opaque field.

Keyed Opaque Format

When applications require efficient opaque messages, and also require content matching, you can use the keyed opaque built-in format. Keyed opaque messages are optimized: though less efficient than the opaque built-in format, they are more efficient than an equivalent managed format.

Applications can use content matchers to match against the key field.

To optimize performance, built-in opaque formats limit the combined size of the data payload and the key. If you need to send larger payloads, define a static format with an opaque field and a string key field.

String Encoding

To preserve interoperability throughout your enterprise, all strings must use UTF-8 encoding.

- When the TIBCO FTL Java and .NET libraries send messages, all strings are automatically UTF-8 encoded.

- C programs must treat strings in inbound messages as UTF-8 encoded strings.
- C programs must send only UTF-8 encoded strings.

i Note: Strings cannot include embedded null characters.

Message: Printable String Form

Programs can convert a message to a printable string. You can output the string to log files, or user output streams. The string does *not* contain sufficient information to reconstitute the original message.

Messages as Printable Strings

Element	Form	Detail
Message Nested message	<i>{field, field, field, field}</i>	Curly braces surround a list of fields, separated by commas. (For readability, you may add space characters.)
Field (with a value)	<i>data_type:field_name=value</i>	Data type, colon, field name, equals symbol, value.
Clear field (no value)		The output omits the field entirely.
Array	<i>[val, val, val, val]</i>	Square brackets surround array elements, which are separated by commas. (For readability, you may add space characters.)
Opaque	<i><length bytes></i>	Angle brackets surround the length of the opaque data.
String	<i>"chars"</i>	Double-quotes surround the characters of the string. This representation does not use escape characters.
DateTime	<i>yyyy-mm-dd hh:mm:ss.fracZ</i>	For more information, see DateTime Values Printable String Form .

Printable String Example

This example message string has added whitespace to increase readability. (Actual printable strings do not include vertical formatting or indenting.)

```
{long:myLong=44,
 opaque:myOpaque=<128 bytes>,
 message:myInner_msg=
   {double:a_double_field=5.300000,
    string:an_inner_string_field="the inner \"value\""},
 long_array:an_array=[11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
 message_array:a_msg_array=
   [{double:a_double_field=5.300021,
    string:an_inner_string_field="first inner string"},
   {double:a_double_field=123.4506,
    string:an_inner_string_field="second inner string"},
   {double:a_double_field=72.90927,
    string:an_inner_string_field="third inner string"},
   {double:a_double_field=17.17017,
    string:an_inner_string_field="fourth inner string"},
   {double:a_double_field=42.00000,
    string:an_inner_string_field="fifth inner string"}]}
```

Several aspects of this example are noteworthy:

- The second field, `myOpaque`, displays a token representing a byte-array. It does not attempt to display the actual bytes in the byte-array.
- The third field, `myInner_msg`, contains a nested submessage. Curly braces surround the submessage, which in turn has two fields.

The second of those two fields in the nested submessage contains a string. The string value is surrounded by double-quote character, and the string itself contains two double-quote characters (around the word `"value"`). Those inner double-quote characters are *not* preceded by an escape character.

- The fourth field, `an_array`, contains an array. The array values are surrounded by square brackets, and separated by commas.
- The fifth field, `a_msg_array`, contains an array of 5 messages. Each message contains two fields: a double and a string.

DateTime Values Printable String Form

The TIBCO FTL library calls print DateTime field values as strings. Double-quotes surround the characters of the string.

DateTime values print in UTC format, also known as Zulu time or GMT. The ISO-8601 standard requires appending a Z character in this notation.

For example, November 19, 2010 6:50:55 PM PST prints as:

```
2010-11-20 02:50:55.000000000Z
```

DateTime printing uses Common Era numbering for years. This system does not include a year zero. So for example, the time 1 second before 0001-01-01 00:00:00.000000000Z would print as -0001-12-31 23:59:59.000000000Z.

DateTime printing uses a proleptic Gregorian calendar. That is, when formatting dates earlier than the adoption of the Gregorian calendar, it projects the Gregorian calendar backward beyond its actual invention and adoption.

Go: Marshal Method for FTL Messages

The FTL Go language API implements the marshal and unmarshal interfaces to convert message datatype.

Methods

The `Message` type has two methods to conveniently import and export message fields:

- `Marshal` imports a Go structure or map into an FTL message.
- `Unmarshal` exports the fields of an FTL message to a Go structure or map.

FTL marshaling and unmarshaling support only those maps with key type string.

Marshaling ignores nil as the field value of a Go structure and as a slice element.

FTL Numeric Field Types

FTL messages store all Go language integral types as `TIB_FIELD_TYPE_LONG` fields, corresponding to the Go language type `int64`.

FTL messages store all Go language floating-point types as `TIB_FIELD_TYPE_DOUBLE` fields, corresponding to the Go language type `float64`.

These designations apply within the following tables.

- *IT* refers to any of the following integral types: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `uintptr`, `bool`.
- *FT* refers to any of the following floating-point types: `float32`, `float64`.

Conversion of Go Types to FTL Fields

Go Type	Marshals to FTL Message Field Type
IT	TIB_FIELD_TYPE_LONG
*IT	
FT	TIB_FIELD_TYPE_DOUBLE
*FT	
[]IT	TIB_FIELD_TYPE_LONG_ARRAY
*[]IT	
[]*IT	
*[]*IT	
[]FT	TIB_FIELD_TYPE_DOUBLE_ARRAY
*[]FT	
[]*FT	
*[]*FT	
string	TIB_FIELD_TYPE_STRING
*string	
[]string	TIB_FIELD_TYPE_STRING_ARRAY
*[]string	
[]*string	
*[]*string	

Go Type	Marshals to FTL Message Field Type
byte *byte Note: []*byte and []*byte are not allowed.	TIB_FIELD_TYPE_OPAQUE
time.Time *time.Time	TIB_FIELD_TYPE_DATETIME
[]time.Time *[]time.Time []*time.Time *[]*time.Time	TIB_FIELD_TYPE_DATETIME_ARRAY
structure pointer to structure	TIB_FIELD_TYPE_MESSAGE
embedded structure pointer to embedded structure	individual fields within the message
ftl.Message *ftl.Message	TIB_FIELD_TYPE_MESSAGE
[]ftl.Message *[]ftl.Message []*ftl.Message *[]*ftl.Message	TIB_FIELD_TYPE_MESSAGE_ARRAY
ftl.Inbox *ftl.Inbox	TIB_FIELD_TYPE_INBOX

Go Type	Marshals to FTL Message Field Type
map[string]	TIB_FIELD_TYPE_MESSAGE
Note: Each element of the map marshals into a field within the message, with the element key becoming the field name. The key type must be string.	

Conversion of FTL Fields to Go Types

FTL Message Field Type	Unmarshals to Go Type
TIB_FIELD_TYPE_LONG	IT
Supports strict conversion.	*IT
TIB_FIELD_TYPE_DOUBLE	FT
Supports strict conversion.	*FT
TIB_FIELD_TYPE_LONG_ARRAY	[]IT *[]IT []*IT *[]*IT The target struct must be a slice, not an array.
TIB_FIELD_TYPE_DOUBLE_ARRAY	[]FT *[]FT []*FT *[]*FT The target struct must be a slice, not an array.
TIB_FIELD_TYPE_STRING	string *string
TIB_FIELD_TYPE_STRING_ARRAY	[]string *[]string

FTL Message Field Type	Unmarshals to Go Type
	<code>[]*string</code> <code>*[]*string</code> The target struct must be a slice, not an array.
TIB_FIELD_TYPE_OPAQUE	<code>byte</code> <code>*byte</code>
TIB_FIELD_TYPE_DATETIME	<code>time.Time</code> <code>*time.Time</code>
TIB_FIELD_TYPE_DATETIME_ARRAY	<code>[]time.Time</code> <code>*[]time.Time</code> <code>[]*time.Time</code> <code>*[]*time.Time</code> The target struct must be a slice, not an array.
TIB_FIELD_TYPE_MESSAGE	<code>structure</code> <code>embedded structure</code> <code>ftl.Message</code> <code>*ftl.Message</code> <code>map[string]</code> <code>*map[string]</code>
TIB_FIELD_TYPE_MESSAGE_ARRAY	Array of structures <code>[]ftl.Message</code> <code>*[]ftl.Message</code> <code>[]*ftl.Message</code> <code>*[]*ftl.Message</code> <code>[]map[string]</code>

FTL Message Field Type	Unmarshals to Go Type
	<code>*[]map[string]</code> The target struct must be a slice, not an array.
TIB_FIELD_TYPE_INBOX	<code>ftl.Inbox</code> <code>*ftl.Inbox</code>

Field Tags in struct Definitions

For convenience, you can use field tags to guide marshaling and unmarshaling (similar to JSON processing). For example:

```
type myStructType struct {
    myFieldA    int `json:"my_field_A" ftl:"myFieldA"`
    myFieldB    int `json:"my_field_B" ftl:"myFieldB,strict"`
    myFieldC    int `json:"my_field_C"
    ftl:"myFieldC,zeromissing,omitzero"`
}
```

FTL software recognizes tags with this general form:

```
ftl:"<fieldname>[,<options>]"
```

- `<fieldname>` is the name of a field in an FTL message, corresponding to the tagged field of the Go struct definition.
- `<options>` is a comma-separated list of options to apply marshaling and unmarshaling the field (see the following table).
- Separate the individual tags with a space character. Each line of the example illustrates this with a space between the JSON tag and FTL tag.

Option	Description
<code>strict</code>	When unmarshaling, enforce strict conversion for numeric fields. In strict conversion, the sign of a numeric FTL field value must be compatible with the corresponding Go language field type. If it is not

Option	Description
	compatible, the unmarshal call returns an error. For example, consider the effect of unmarshaling an FTL message field with type <code>TIB_FIELD_TYPE_LONG</code>. The value 100 fits into any unsigned Go field, or into an <code>int8</code>. But the value -1000 does not fit, and strict conversion would reject it. Non-strict conversion could yield unpredictable values for values that do not fit.
<code>zeromissing</code>	When unmarshaling, supply zero values for missing fields. If the target Go struct definition specifies a field but a corresponding field does not exist in the source FTL message, then conversion sets the Go field to the zero value of its field type. When this option is absent, unmarshaling does not modify the target Go field unless the field is present in the FTL message.
<code>omitzero</code>	When marshaling, omit fields that contain the zero value. This option is analogous to <code>omitempty</code> in JSON marshaling. If the Go field contains the zero value for its type, then marshaling does not add a corresponding field to the FTL message. Zero values include <code>false</code>, <code>0</code>, <code>0.0</code>, <code>""</code>, a nil pointer or interface value, and any empty array, slice, map, structure, or string. When this option is absent, marshaling adds an FTL field corresponding to every Go field.
<code>format=<format_name></code>	Marshal a Go field value that is itself a nested structure to an FTL message with format <code><format_name></code>. (Use this option when your FTL application definition manages all formats.) When this option is absent, marshal a nested structure to an FTL message of unnamed dynamic format (that is, null format name).

Examples

The FTL go API provides a simple mechanism for marshaling structures into an FTL message, and for unmarshaling FTL messages into a structure. The use of field tags is

similar to that found in the `encoding/json` package. Note that structure fields must be exportable in order to be marshaled or unmarshaled. In simple terms, this means the field name must begin with an upper-case letter.

Marshal/unmarshal also supports the use of maps, in the form `map[string]interface{}`. In this case, the FTL field name corresponds to the map entry key, and the FTL field value corresponds to the map entry value.

The following example unmarshals message data into a Go structure:

```
type HelloMessage struct {
    Type string `ftl:"type"`
    Message string `ftl:"message"`
}

func main() {
    // Create an instance of the HelloMessage structure. Then unmarshal
    // the message into the structure, based on the tags attached to
    // each field in the structure.
    //
    // Note that a pointer to the structure (or map) must be passed to
    // the Unmarshal() function.
    helloMsg := HelloMessage{}
    if err = msg.Unmarshal(&helloMsg); err != nil {
        log.Fatal(ftl.ErrStack(err))
    }
    // Print the message contents, both as a formatted FTL message
    // (via the message String() function) and the as the
    // unmarshaled structure content (using go's %#v formatting directive).
    //
    log.Printf("message: %s\n", msg.String())
    log.Printf("Unmarshalled message: %#v\n", helloMsg)
}
```

The following example marshals Go structure data into a message:

```
func main() {
    msg, err := realm.NewMessage("helloworld")
    if err != nil {
        log.Fatal(ftl.ErrStack(err))
    }
    // Defer the destruction of the message.
    defer msg.Destroy()
    // Construct the message content into a map[string]interface{}.
    content := make(map[string]interface{})
    content["type"] = "hello"
}
```

```
    content["message"] = "hello world earth"
    // Marshal the message content into the FTL message. Note that a
    // pointer to the map must be passed to msg.Marshal().
    if err = msg.Marshal(&content); err != nil {
        log.Fatal(ftl.ErrStack(err))
    }
}
```

For more information about FTL message marshaling/unmarshaling, see the samples in [.../samples/src/golang/src/tibco.com/ftl-sample](#).

Logs

Developers, administrators, and TIBCO Support staff can use logging output for diagnostic purposes.

Application developers can set the log level. Administrators can subsequently change log level settings using the central logging facility.

Application developers can set the log output target.

Log Levels

The log level determines the level of detail and the quantity of log statements. Properly tuned logging can help diagnose unexpected behaviors. Excessively detailed logging can contribute to message latency and consume storage resources.

Application developers can set the log level using an API entry point. In addition, TIBCO FTL executable components accept the same log level specifications through command line arguments.

Log Level Tuning via API

You can tune the log level for all logging elements, or separately per individual elements.

Tuning for All Elements

In a log-level call, supply one of the string-valued constants in [Log Level Reference](#). For example (in C):

```
tib_SetLogLevel(tibEx e, TIB_LOG_LEVEL_WARN)
```

Alternatively, you can supply an **all** log element tag. See the next section.

Tuning for Selected Elements

Supply a string of this form:

```
"element:level; ... ;element:level"
```

Pair each element and its log level with a colon, and delimit the pairs with semicolons. For available elements, see [Log Element Tags Reference](#). For example (in C):

```
tib_SetLogLevel(tibEx e, "transports:warn;msg:debug")
```

When elements conflict, pairs that appear later in the list override earlier pairs. For example, `all:off;msg:warn` turns off logging for all elements, except for the messages (`msg`) element, which logs warnings and severe events, thus overriding the preceding `all` element.

Log Level Reference

These API constants and their string values denote the available log levels.

Log Level Constants and Values

Log Level Constant	String	Description
TIB_LOG_LEVEL_OFF	off	Disable all tracing.
TIB_LOG_LEVEL_SEVERE	severe	Output log entries only for severe events.
TIB_LOG_LEVEL_WARN	warn	Output log entries for warning and severe events.
TIB_LOG_LEVEL_INFO	info	Output log entries for information, warning, and severe events. If an application program does not explicitly set the log level, the library uses this default value.
TIB_LOG_	verbose	Output log entries for verbose, information, warning, and

Log Level Constant	String	Description
LEVEL_VERBOSE		severe events.
TIB_LOG_ LEVEL_DEBUG	debug	Output log entries for debug, verbose, information, warning, and severe events.



Note: The output from debug and verbose can result in very large log files. This level of detail is generally not useful unless TIBCO staff specifically requests it.

Log Element Tags Reference

These strings denote the available log element tags.

Log Element Tags

Element	Description
all	Tune logging for all elements, embracing all the categories in this table.
application	Tune logging related to the application program, such as the program's use of API calls, message formats, and content matchers.
IO	Admin only. Tune logging related to data I/O, such as message data entering or leaving the program, and message packetization.
RS	Admin only. Tune logging related to the realm server. - In FTL server client processes, this category includes client communication with the FTL server. - In FTL server processes, this category includes operations within the FTL server, such as communication with clients, with affiliated FTL servers, and with services.
transports	Tune logging related to transports, including peer connections, subscriber interest, message send operations, protocols.

Element	Description
store	Admin only. Tune logging related to persistence stores and durables, such as store and durable operations, quorum formation, and leader election.
msg	Tune logging related to messages. The debug level logs each outbound message send and each inbound message callback by printing a representation of the message.
broker	Admin only. Tune logging related to clients of the eFTL service. (This element is available <i>only</i> in the eFTL service.)
route	Admin only. Tune logging related to wide-area durables.
mux	Admin only. Tune logging related to the FTL server as it redirects HTTP requests to the services it provides.
api	Tune logging related to API calls.

Log Output Targets

Developers can select the output target for log statements. Application programs can use one of three mutually exclusive targets.

Log Targets

Log Target	Description
Default Log Target	Output all log statements to <code>stderr</code> .
Log Files in Rotation	Output all log statements to a set of rotating log files.
Log Callback	The program defines a callback, which intercepts all log statements.

Each program process may set its log target only once. Attempting to set it again is an error.

Default Log Target

If a program does not set a log target, TIBCO FTL software directs all log statements to `stderr`.

Log Files in Rotation

Rotating log files organize log output into a set of files, and limit the file system storage that log statements can consume.

The call that arranges rotating log files accepts a file prefix, the file size that triggers rotation, and the maximum number of files.

Log files receive filenames consisting of the prefix you supply, a dot (.) character, and an integer suffix. Suffix zero indicates the current log file. Suffix 1 indicates the next most recent. Higher suffixes indicate older files.

Rotating renames each file in the set by incrementing its numeric suffix. If rotating would exceed the maximum number of files, then it discards the oldest file. These file I/O operations can affect message latency.

The TIBCO FTL library requires that the total length of any log file name must be strictly less than 1024 characters. The host computer's operating system may require a smaller file name length.

Log Callback

Applications can define and register a log callback. Instead of directing log statements to the file system, the TIBCO FTL library invokes the callback to process each log statement (that is, to enter it into the application's log system).

This type of target can introduce overhead costs and potential latency.

**Warning:** Log Callback Restrictions

- Programmers must ensure that the callback always returns promptly. The log callback executes inline, that is, synchronously, and can affect message latency.
- Your callback *must* not call any TIBCO FTL API entry point.
- Your callback *must* be thread-safe.
- For best results, your callback must not acquire any lock that might require a long wait time.

Log Content and Form

The form and content of trace log output are *not* part of a defined interface: they could change in subsequent releases.



Warning: Avoid programming techniques that rely on parsing log output, or depend on log content.

Advisory Messages

TIBCO FTL software presents asynchronous *advisory messages* to application programs. Advisory messages can indicate errors, warnings, or other information.

In contrast with exceptions and error codes, which indicate success or failure within a specific TIBCO FTL call, asynchronous advisory messages notify programs of incidents that occur outside of the program's direct flow of control.

When it is not possible to communicate through an advisory message, TIBCO FTL software presents an *out-of-band notification* instead (see [Notifications](#)).

Programs must arrange to receive and process information through both of these mechanisms: advisories and notifications.

Advisory Message Common Fields Reference

Advisory messages contain fields that present information to programs. Some fields are present on all advisory messages. Other fields are present only when information is available. Still other fields are associated only with specific advisory messages. Programs can use content matchers to select subsets of advisory messages from the advisory endpoint.

Advisory Message, Common Fields

Field	Present	Description
name	Always	The value denotes the general category of the advisory.
reason	Only as needed	The value distinguishes among several possible reasons for the advisory.
severity	Always	The value classifies the advisory according to the ability of TIBCO FTL software to fulfill its function as a message carrier.

Field	Present	Description
module	Always	The value denotes the part of TIBCO FTL software to which the advisory pertains.
timestamp	Always	The <code>DateTime</code> value of this field indicates the time that the library generated the advisory. Although <code>DateTime</code> values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.
aggregation_count	Only for aggregate data	The value represents the number of aggregated incidents.
aggregation_time	Only for aggregate data	The value represents the length of the time interval, in seconds, for aggregated data.

Subscribing to Advisory Messages

To receive advisory messages, programs can create subscribers on a special endpoint named `_advisoryEndpoint`. (In each supported programming language, an API constant has the name of this endpoint as its value.)

Scope and Restrictions

i Note: The realm object scopes the advisories that a subscriber can receive on the special advisory endpoint, `_advisoryEndpoint`. That is, a subscriber to this endpoint receives only those advisories produced by other objects in the same program and the same realm as the advisory subscriber.

It is illegal to create publishers on the special advisory endpoint.

Procedure

1. Determine the set of advisory messages to receive.
2. Determine the action of your program in response to each advisory message.
3. Define message callbacks to process the advisory messages.

For example, an advisory callback might respond by logging the content of the advisory, by notifying an administrator, or by exiting.

4. Optional. Define content matchers.

A subscriber may receive all advisory messages without distinguishing among them, or it may select a specific subset of advisory messages using a content matcher.

5. Create subscribers on the special advisory endpoint.
6. Create a special event queue, dedicated only to dispatching and processing advisory messages.

This queue must never discard events.

7. Code a special dispatch thread within your program, dedicated only to dispatching the advisory event queue. Ensure that this thread always receives sufficient CPU cycles in a timely fashion, even when other dispatch threads might be stuck.
8. Add the advisory subscribers to the dedicated advisory event queue.

Advisory Messages Catalog

Application programs can receive the advisory messages in this table.

Advisory Message Catalog

Name and Reason	Description
<code>DATALOSS FAILOVER_LOSS</code>	Failover to a backup forwarding component, such as a transport bridge.
<code>DATALOSS INCOMPLETE_MESSAGE</code>	A transport could not reassemble the complete message.
<code>DATALOSS QUEUE_LIMIT_EXCEEDED</code>	An event queue discarded events.

Name and Reason	Description
DATALOSS SENDER_DISCARD	A transport discarded messages in accordance with its backlog settings (non-blocking send).
DATALOSS TPORT_DATALOSS	A transport lost data packets.
DATALOSS RECONNECT_LOSS	A connection-oriented transport became disconnected.
DATALOSS STORE_DISCARD_DATALOSS	A subscriber to a last-value durable has missed messages.
DATALOSS UPSTREAM_LOSS	A forwarding component detected an upstream dataloss advisory.
DATALOSS DIRECT_SUB_LOSS	A direct subscriber lost data.
RESOURCE_AVAILABLE	A resource is available.
RESOURCE_UNAVAILABLE	A resource is unavailable.
Groups	
ORDINAL_UPDATE	Change to a group ordinal.
Persistence Stores and Durables	
SUBSCRIBER_FORCE_CLOSE	A persistence store forced a subscriber object to close.
LOCK_LOST	The client program no longer holds a lock.
License	
LICENSE EXPIRED	License related advisory messages, when the license of FTLServer is expired
LICENSE EXPIRING_SOON	License related advisory messages, when the license of FTLServer is expiring soon (within 90 days)

Name and Reason	Description
LICENSE EXPIRING_IMMINENTLY	License related advisory messages, when the license of FTLServer is expiring imminently (within 7 days)
LICENSE SERVER_UNREACHABLE	License related advisory messages, when the license server is unreachable by the of FTLServer
Multicast Retransmission	
RETRANSMISSION RETRANSMISSION_REQUEST	A subscriber requested retransmission.
RETRANSMISSION RETRANSMISSION_SENT	A publisher sent a retransmission.
RETRANSMISSION RETRANSMISSION_SUPPRESSED	A subscriber suppressed a retransmission request.

DATALOSS FAILOVER_LOSS

This advisory reports potential dataloss. The reason is failover to a backup forwarding component.

Details

Only connection-oriented transports can detect this condition.

This advisory does not indicate the volume of lost data, but rather the number of data loss incidents. Each data loss incident represents the *potential* for lost messages. (The volume of lost data could be zero.)

Actions

Report this advisory to an administrator. Check the forwarding component.

Fields

Field	Description
name	DATALOSS
reason	FAILOVER_LOSS
endpoints	The string array value of this field lists the endpoints that could have lost data. Although data loss occurs primarily in a transport, its symptoms could affect all endpoints that use the transport, and by extension, any subscriber on those endpoints. Furthermore, transport names are meaningful to administrators, but usually not available to programmers. This advisory field reports the set of all endpoints through which the program could access the problematic transport, according to the configuration in the local realm object.
aggregation_count	The long value of this field reports the cumulative number of failover incidents during the time interval (see aggregation_time). The base library aggregates this count separately for each transport.
aggregation_time	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that aggregation_count reports. The time interval ends shortly before the timestamp.
timestamp	The DateTime value of this field indicates the time that the library generated the advisory. Although DateTime values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

DATALOSS SENDER_DISCARD

The advisory reports an unrecoverable dataloss. The reason is transport backlog overflow at the publisher, in accordance with the transport's backlog settings.

Details

This advisory reports only inbound dataloss to subscribing programs. It does *not* report outbound dataloss to the publishing program.

This advisory does not indicate the volume of lost data, but rather the number of dataloss incidents. Each dataloss incident represents *at least one* lost message, but could represent many lost messages.

For more information about backlog overflow, see [Send: Blocking versus Non-Blocking](#).

Actions

Report this advisory to an administrator.

Fields

Field	Description
name	DATALOSS
reason	SENDER_DISCARD
endpoints	The string array value of this field lists the endpoints that could have lost data. Although data loss occurs primarily in a transport, its symptoms could affect all endpoints that use the transport, and by extension, any subscriber on those endpoints. Furthermore, transport names are meaningful to administrators, but usually not available to programmers. This advisory field reports the set of all endpoints through which the program could access the problematic transport, according to the configuration in the local realm object.
aggregation_count	The long value of this field reports the cumulative number of dataloss incidents during the time interval (see <code>aggregation_time</code>). The base library aggregates this count separately for each transport.
aggregation_time	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that <code>aggregation_</code>

Field	Description
	count reports. The time interval ends shortly before the timestamp.
timestamp	The <code>DateTime</code> value of this field indicates the time that the library generated the advisory. Although <code>DateTime</code> values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

DATALOSS QUEUE_LIMIT_EXCEEDED

The advisory reports dataloss. The reason is overflow of an event queue.

Details

The queue has discarded some events, in accordance with its property values.

Diagnoses

- Lengthy processing in callbacks could delay prompt dispatch of the queue.
- The program could not process the volume of inbound messages from its subscribers.
- The program is starved for CPU cycles. Its host computer is too heavily loaded.
- The maximum events limit of the event queue is too low.

Actions

Consider whether this behavior is reasonable and expected.

Report this advisory to an administrator.

Fields

Field	Description
name	DATALOSS
reason	QUEUE_LIMIT_EXCEEDED
endpoints	The string array value of this field lists the endpoints that could have lost data. Although data loss occurs primarily in a transport, its symptoms could affect all endpoints that use the transport, and by extension, any subscriber on those endpoints. Furthermore, transport names are meaningful to administrators, but usually not available to programmers. This advisory field reports the set of all endpoints through which the program could access the problematic transport, according to the configuration in the local realm object.
aggregation_count	The long value of this field reports the cumulative number of events that the queue discarded during the time interval (see <code>aggregation_time</code>). The base library aggregates this count separately for each transport.
aggregation_time	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that <code>aggregation_count</code> reports. The time interval ends shortly before the <code>timestamp</code>.
timestamp	The <code>DateTime</code> value of this field indicates the time that the library generated the advisory. Although <code>DateTime</code> values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

DATALOSS INCOMPLETE_MESSAGE

Purpose

The advisory reports dataloss. The reason is an incomplete message.

The transport could not reassemble an inbound message because some data fragments did not arrive.

Diagnoses

- Network issues could interfere with message transmission.
- The sending process abruptly exited before transmitting all packets.
- When this advisory occurs alongside [DATALOSS SENDER_DISCARD](#), the problem could be within the sending process or a forwarding component.

Actions

Report this advisory to an administrator.

Fields

Field	Description
name	DATALOSS
reason	INCOMPLETE_MESSAGE
endpoints	The string array value of this field lists the endpoints that could have lost data. Although data loss occurs primarily in a transport, its symptoms could affect all endpoints that use the transport, and by extension, any subscriber on those endpoints. Furthermore, transport names are meaningful to administrators, but usually not available to programmers. This advisory field reports the set of all endpoints through which the program could access the problematic transport, according to the

Field	Description
	configuration in the local realm object.
aggregation_ count	The long value of this field reports the cumulative number of incomplete message incidents during the time interval (see <code>aggregation_time</code>). The base library aggregates this count separately for each transport.
aggregation_ time	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that <code>aggregation_count</code> reports. The time interval ends shortly before the <code>timestamp</code>.
timestamp	The <code>DateTime</code> value of this field indicates the time that the library generated the advisory. Although <code>DateTime</code> values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

DATALOSS TPORT_DATALOSS

The advisory reports an unrecoverable dataloss. The reason is a transport malfunction.

Details

This advisory reports only inbound data loss, not outbound.

This advisory does not indicate the volume of lost data, but rather the number of data loss incidents. Each data loss incident represents the *potential* for lost messages. (The volume of lost data could be zero.)

Actions

Report this advisory to an administrator.

Fields

Field	Description
name	DATALOSS
reason	TPORT_DATALOSS
endpoints	The string array value of this field lists the endpoints that could have lost data. Although data loss occurs primarily in a transport, its symptoms could affect all endpoints that use the transport, and by extension, any subscriber on those endpoints. Furthermore, transport names are meaningful to administrators, but usually not available to programmers. This advisory field reports the set of all endpoints through which the program could access the problematic transport, according to the configuration in the local realm object.
aggregation_count	The long value of this field reports the cumulative number of dataloss incidents during the time interval (see aggregation_time). The base library aggregates this count separately for each transport.
aggregation_time	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that aggregation_count reports. The time interval ends shortly before the timestamp.
timestamp	The DateTime value of this field indicates the time that the library generated the advisory. Although DateTime values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

DATALOSS RECONNECT_LOSS

The advisory reports potential dataloss. The reason is a temporary network disconnect between two connection-oriented transports.

Details

This advisory reports only inbound data loss, not outbound.

This advisory does not indicate the volume of lost data, but rather the number of data loss incidents. Each data loss incident represents the *potential* for lost messages. (The volume of lost data could be zero.)

Actions

Report this advisory to an administrator.

Fields

Field	Description
name	DATALOSS
reason	RECONNECT_LOSS
endpoints	The string array value of this field lists the endpoints that could have lost data. Although data loss occurs primarily in a transport, its symptoms could affect all endpoints that use the transport, and by extension, any subscriber on those endpoints. Furthermore, transport names are meaningful to administrators, but usually not available to programmers. This advisory field reports the set of all endpoints through which the program could access the problematic transport, according to the configuration in the local realm object.
aggregation_count	The long value of this field reports the cumulative number of disconnect incidents during the time interval (see aggregation_time). The base library aggregates this count separately for each transport.
aggregation_time	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that aggregation_count reports. The time interval ends shortly before the timestamp.

Field	Description
timestamp	The <code>DateTime</code> value of this field indicates the time that the library generated the advisory. Although <code>DateTime</code> values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

DATALOSS STORE_DISCARD_DATALOSS

The advisory reports potential data loss at a subscriber to a last-value durable. The reason is that the subscriber has either missed one or more messages, or disconnected from the persistence store.

Details

A subscriber to a last-value durable receives every message that updates that durable. However, if the subscriber cannot keep up with the stream of updates from publishers, the subscriber might miss messages, as the durable stores only the most recent message.

If the subscriber disconnects from the persistence store, it could miss messages during the interval in which it remains disconnected.

This advisory reports only inbound data loss, not outbound.

This advisory does not indicate the volume of lost data, but rather the number of data loss incidents. Each data loss incident represents the *potential* for lost messages. (The volume of lost data could be zero.)

Actions

Report this advisory to an administrator.

Fields

Field	Description
name	DATALOSS
reason	STORE_DISCARD_DATALOSS
endpoints	The string array value of this field lists the endpoints that could have lost data. Although data loss occurs primarily in a transport, its symptoms could affect all endpoints that use the transport, and by extension, any subscriber on those endpoints. Furthermore, transport names are meaningful to administrators, but usually not available to programmers. This advisory field reports the set of all endpoints through which the program could access the problematic transport, according to the configuration in the local realm object.
aggregation_count	The long value of this field reports the cumulative number of discard or disconnect incidents during the time interval (see <code>aggregation_time</code>). The base library aggregates this count separately for each transport.
aggregation_time	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that <code>aggregation_count</code> reports. The time interval ends shortly before the <code>timestamp</code>.
timestamp	The <code>DateTime</code> value of this field indicates the time that the library generated the advisory. Although <code>DateTime</code> values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

DATALOSS UPSTREAM_LOSS

The advisory reports potential dataloss. The reason is a dataloss advisory detected upstream at a forwarding component (such as a transport bridge).

Details

This advisory reports only inbound data loss, not outbound.

This advisory does not indicate the volume of lost data, but rather the number of data loss incidents. Each data loss incident represents the *potential* for lost messages. (The volume of lost data could be zero.)

Actions

Report this advisory to an administrator.

Fields

Field	Description
name	DATALOSS
reason	UPSTREAM_LOSS
endpoints	The string array value of this field lists the endpoints that could have lost data. Although data loss occurs primarily in a transport, its symptoms could affect all endpoints that use the transport, and by extension, any subscriber on those endpoints. Furthermore, transport names are meaningful to administrators, but usually not available to programmers. This advisory field reports the set of all endpoints through which the program could access the problematic transport, according to the configuration in the local realm object.
aggregation_count	The long value of this field reports the cumulative number of dataloss incidents during the time interval (see aggregation_time). The base library aggregates this count separately for each transport.
aggregation_time	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that aggregation_count reports. The time interval ends shortly before the timestamp.

Field	Description
timestamp	The DateTime value of this field indicates the time that the library generated the advisory. Although DateTime values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

DATALOSS DIRECT_SUB_LOSS

The advisory reports an unrecoverable data loss at a direct subscriber.

Details

This advisory reports only inbound data loss, not outbound.

This advisory does not indicate the volume of lost data, but rather the number of data loss incidents. Each data loss incident represents the *potential* for lost messages. (The volume of lost data could be zero.)

Actions

Report this advisory to an administrator.

Fields

Field	Description
name	DATALOSS
reason	DIRECT_SUB_LOSS
endpoints	The string array value of this field lists the endpoints that could have lost data. Although data loss occurs primarily in a transport, its symptoms could

Field	Description
	affect all endpoints that use the transport, and by extension, any subscriber on those endpoints. Furthermore, transport names are meaningful to administrators, but usually not available to programmers. This advisory field reports the set of all endpoints through which the program could access the problematic transport, according to the configuration in the local realm object.
aggregation_count	The long value of this field reports the cumulative number of data loss incidents during the time interval (see aggregation_time). The base library aggregates this count separately for each transport.
aggregation_time	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that aggregation_count reports. The time interval ends shortly before the timestamp.
timestamp	The DateTime value of this field indicates the time that the library generated the advisory. Although DateTime values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

ORDINAL_UPDATE

The advisory reports a change to a group ordinal.

Details

For background information, see [Groups of Applications](#).

Actions

Programs change the member's operating role and behavior according to the new ordinal.

Fields

Field	Description
name	ORDINAL_UPDATE
group	Group name. The name of the group to which the advisory pertains.
ordinal	Group member ordinal. The positive long value of the <code>ordinal</code> field represents the new ordinal of the group member. The value -1 indicates that the group object is disconnected from the group service. The group object automatically attempts to reconnect, and continues until the program explicitly destroys the object. Meanwhile, the group service could reassign the member's previous ordinal to another group member. Zero is a reserved value.

GROUP_STATUS

The advisory reports the status of a group and its members.

Group status messages differ in form and content, depending on the member and the state change. For more information, see [Group Status](#).

Fields of the Group Status Advisory Message

Field	Description
name	GROUP_STATUS
group	Group name. The name of the group to which the advisory pertains.
group_ server_	When present, the long value of this field indicates whether the application process has a functioning connection to the group service.

Field	Description
available	
group_member_status_list	When present, the value of this field is an array of submessages. Each submessage describes the status of one group member.

Fields of the Member Status Submessage

Field	Description
group_member_descriptor	The value of this field is a message that identifies a group member. Member programs can supply this optional member descriptor message when the program joins the group. If the member does not supply a descriptor, then this field is absent in the status submessage: the member is indistinguishable from any other member that omits a descriptor.
group_member_event	The long value of this field is an enumerated constant that indicates the connection status of the group member.

RESOURCE_AVAILABLE

The advisory reports that a resource is available.

Details

Persistence services are resources. For background information, see [Persistence: Stores and Durables](#).

Diagnosis

A resource that had been unavailable is now available. Operations that required that resource may proceed. The reason field indicates the resource.

Actions

Report this advisory to an administrator.

Fields

Field	Description
name	RESOURCE_AVAILABLE
reason	PERSISTENCE_STORE_AVAILABLE PERSISTENCE_STORE_AVAILABLE_AFTER_CLUSTER_DATALOSS
endpoints	The string array value of this field lists the endpoints that had been affected by a persistence service that had been unavailable but is now available again.
timestamp	The DateTime value of this field indicates the time that the library generated the advisory. Although DateTime values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

RESOURCE_UNAVAILABLE

The advisory reports that a resource is unavailable.

Details

Persistence services are resources. For background information, see [Persistence: Stores and Durables](#).

Diagnosis

A required resource is unavailable. The `reason` field indicates the resource.

- The persistence cluster lacks a quorum of available services for the store. Application clients must wait until a quorum forms.
- An insufficient number of persistence services are operating. Administrators must attend to FTL servers and persistence services.

Actions

Report this advisory to an administrator.

If this state continues for a long duration, investigate whether hardware or network issues are interfering with resource operation or communication.

Fields

Field	Description
name	RESOURCE_UNAVAILABLE
reason	PERSISTENCE_STORE_UNAVAILABLE
endpoints	The string array value of this field lists the endpoints affected by an unavailable persistence service.
timestamp	The <code>DateTime</code> value of this field indicates the time that the library generated the advisory. Although <code>DateTime</code> values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

RETRANSMISSION RETRANSMISSION_REQUEST

The advisory reports that a subscribing endpoint in the application process requested retransmission from a publisher.

Details

Retransmission is a normal occurrence for multicast transports. Retransmission advisories give applications access to data about network conditions.

When the subscribing endpoint of a multicast transport detects that it has missed one or more data packets, it requests that the publishing endpoint retransmit them. If possible, the publishing endpoint complies. Otherwise data loss occurs.

This advisory has severity `DEBUG`. It does not indicate impaired behavior. Applications can receive it *only* when the log level is set to `DEBUG` for the `transports` element. See also [Log Level Tuning via API](#).

Actions

This advisory contains information for debugging. Report this advisory to an administrator.

Fields

Field	Description
<code>name</code>	<code>RETRANSMISSION</code>
<code>reason</code>	<code>RETRANSMISSION_REQUEST</code>
<code>endpoints</code>	The string array value of this field lists the endpoints that detected missed data packets.
<code>aggregation_count</code>	The long value of this field reports the cumulative number of retransmission request incidents during the time interval (see <code>aggregation_time</code>). The base library aggregates this count separately for each transport.
<code>aggregation_time</code>	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that <code>aggregation_count</code> reports. The time interval ends shortly before the <code>timestamp</code>.
<code>timestamp</code>	The <code>DateTime</code> value of this field indicates the time that the library generated the advisory. Although <code>DateTime</code> values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

RETRANSMISSION RETRANSMISSION_SENT

The advisory reports that a publishing endpoint in the application process retransmitted data packets as requested as requested by a subscribing endpoint.

Details

Retransmission is a normal occurrence for multicast transports. Retransmission advisories give applications access to data about network conditions.

When the subscribing endpoint of a multicast transport detects that it has missed one or more data packets, it requests that the publishing endpoint retransmit them. If possible, the publishing endpoint complies. Otherwise data loss occurs.

This advisory has severity `DEBUG`. It does not indicate impaired behavior. Applications can receive it *only* when the log level is set to `DEBUG` for the transports element. See also [Log Level Tuning via API](#).

Actions

This advisory contains information for debugging. Report this advisory to an administrator.

Fields

Field	Description
name	RETRANSMISSION
reason	RETRANSMISSION_SENT
endpoints	The string array value of this field lists the endpoints that re-sent missed data packets.
aggregation_count	The long value of this field reports the cumulative number of data retransmission incidents during the time interval (see <code>aggregation_time</code>). The base library aggregates this count separately for each transport.
aggregation_time	The double floating point value, in seconds, of this field indicates the

Field	Description
	length of the time interval for aggregating the incidents that aggregation_count reports. The time interval ends shortly before the timestamp.
timestamp	The DateTime value of this field indicates the time that the library generated the advisory. Although DateTime values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

RETRANSMISSION RETRANSMISSION_SUPPRESSED

The advisory reports that a subscribing endpoint in the application process would have requested retransmission, but the request was suppressed. Suppression of retransmit requests can occur when a multicast transport enables retransmission control, and a subscriber misses packets in excess of a specified threshold.

Details

Retransmission is a normal occurrence for multicast transports. Retransmission advisories give applications access to data about network conditions.

When the subscribing endpoint of a multicast transport detects that it has missed one or more data packets, it requests that the publishing endpoint retransmit them. If possible, the publishing endpoint complies. Otherwise data loss occurs.

This advisory has severity WARN.

Actions

Report this advisory to an administrator.

Fields

Field	Description
name	RETRANSMISSION
reason	RETRANSMISSION_SUPPRESSED
endpoints	The string array value of this field lists the endpoints that suppressed retransmission requests.
aggregation_count	The long value of this field reports the cumulative number of retransmission request suppression incidents during the time interval (see <code>aggregation_time</code>). The base library aggregates this count separately for each transport.
aggregation_time	The double floating point value, in seconds, of this field indicates the length of the time interval for aggregating the incidents that <code>aggregation_count</code> reports. The time interval ends shortly before the <code>timestamp</code>.
timestamp	The <code>DateTime</code> value of this field indicates the time that the library generated the advisory. Although <code>DateTime</code> values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

SUBSCRIBER_FORCE_CLOSE

The advisory reports that the persistence store closed a durable subscriber.

Details

For background information, see [Persistence: Stores and Durables](#).

Diagnosis

Each durable can serve at most one subscriber object at a time. A durable subscriber connected to the persistence store, and it maps to a durable that already serves another subscriber object. The store resolves collisions in favor of the more recent subscriber, deactivating the older subscriber.

Actions

Close the subscriber object. Clean up application state associated with the subscriber. Report this advisory to an administrator.

Fields

Field	Description
name	SUBSCRIBER_FORCE_CLOSE
reason	DURABLE_SUBSCRIBER_COLLISION
endpoints	The string array value of this field contains the single endpoint that the closed subscriber had instantiated.
subscriber_name	Subscriber name of the closed durable subscriber.
timestamp	The <code>DateTime</code> value of this field indicates the time that the library generated the advisory. Although <code>DateTime</code> values can represent time with nanosecond precision, the actual resolution of the timestamp is only as fine-grained as the host computer's operating system supports.

LOCK_LOST

The advisory reports that the client program no longer holds a lock.

Details

For background information, see [LOCK_LOST](#).

Diagnosis

- Another client process has stolen the lock.
- This client process disconnected from the persistence service. All locks are invalid.

Actions

Request the lock again.

Fields

Field	Description
name	LOCK_LOST
reason	TIB_ADVISORY_REASON_LOCK_STOLEN TIB_ADVISORY_REASON_LOCK_LOST_ON_DISCONNECT
lock_ name	The string value of this field contains the name of the lock that was stolen or lost.

LICENSE

Application programs can receive the advisory messages in this table.

Name and Reason	Description
LICENSE EXPIRED	License-related advisory messages when the license of FTLServer is expired.
LICENSE EXPIRING_	License-related advisory messages when the license of FTLServer is

Name and Reason	Description
SOON	expiring soon (within 90 days).
LICENSE EXPIRING_IMMINENTLY	License-related advisory messages when the license of FTLServer is expiring imminently (within 7 days).
LICENSE SERVER_UNREACHABLE	License-related advisory messages when the license server is unreachable by the FTLServer.

LICENSE SERVER_UNREACHABLE

Details

This advisory indicates that the FTL Server is unable to reach the license server.

Actions

Report this advisory to an administrator. Restore connectivity to the TIBCO Activation Service to avoid potential disruption.

Fields

Field	Description
name	LICENSE
reason	SERVER_UNREACHABLE
license_is_active	True if the license is active, false otherwise.
license_max_activation_server_outage_seconds	The maximum number of seconds that the FTL server runs without connection to the Activation Server.
license_is_expiration_from_cache	True if the license expiration seconds is from the cached license, false otherwise.

Field	Description
server_name	Name of the FTL Server that is unable to reach the license server.
license_status	SERVER_UNREACHABLE
license_status_message	Message describing the status of the license
license_lease_expiration_seconds	Seconds until the license lease expires.
license_location	License server location

LICENSE EXPIRED

Details

This advisory indicates that the FTL Server has an expired license.

Actions

Report this advisory to an administrator. Request a new license immediately to avoid potential product shutdown.

Fields

Field	Description
name	LICENSE
reason	EXPIRED
license_is_active	True if the license is active, false otherwise

Field	Description
license_max_activation_server_outage_seconds	The maximum number of seconds that the FTL server runs without connection to the Activation Server
license_is_expiration_from_cache	True if the license expiration seconds is from the cached license, false otherwise
server_name	Name or URL of the FTL Server that has the expired license
license_lease_expiration_seconds	Seconds until the license lease expires

LICENSE EXPIRING_IMMINENTLY

Details

This advisory indicates that the FTL Server's license will expire imminently (within 7 days).

Actions

Report this advisory to an administrator. Request a new license immediately to avoid potential disruption.

Fields

Field	Description
name	LICENSE
reason	EXPIRING_IMMINENTLY
license_is_active	True if the license is active, false otherwise
license_max_activation_	The maximum number of seconds that the FTL server runs

Field	Description
server_outage_seconds	without connection to the Activation Server
license_is_expiration_from_cache	True if the license expiration seconds is from the cached license, false otherwise
server_name	Name of the FTL Server that has a license expiring imminently
license_status	The status of the license
license_status_message	Message describing the status of the license
license_lease_expiration_seconds	Seconds until the license lease expires
license_location	License file/server location

LICENSE EXPIRING_SOON

Details

This advisory indicates that the FTL Server's license will expire soon (within 90 days).

Actions

Report this advisory to an administrator. Request a new license soon.

Fields

Field	Description
name	LICENSE
reason	EXPIRING_SOON

Field	Description
license_is_active	True if the license is active, false otherwise
license_max_activation_server_outage_seconds	The maximum number of seconds that the FTL server runs without connection to the Activation Server
license_is_expiration_from_cache	True if the license expiration seconds is from the cached license, false otherwise
server_name	Name of the FTL Server that has a license expiring soon
license_status	The status of the license
license_status_message	Message describing the status of the license
license_lease_expiration_seconds	Seconds until the license lease expires
license_location	License file/server location

Notifications

In some situations, the base library must notify programs of conditions that preclude the use of event queues as the communications medium. Instead of sending an advisory, the library notifies the program through an out-of-band mechanism.

All application programs must define a notification handler callback to process these out-of-band notifications, and register that callback using the set notification handler method.

Notifications Catalog

Out-of-Band Notification Catalog

Name and Reason	Description
<code>CLIENT_DISABLED</code>	The FTL server disabled this application process.

CLIENT_DISABLED

This notification alerts the application code that the FTL server administratively disabled this client process.

Diagnoses

- An administrator modified the realm definition, but this application process could not accept the changes, and the administrator disabled it (along with other processes that could not accept the changes).
- An administrator disabled this process to solve a problem (for example, consuming too much network bandwidth).

The administrator can supply an optional string to indicate a reason for disabling client processes. The notification callback receives this reason string as an argument.

Actions

Your notification callback must arrange for either of these two recovery actions:

- Create a new realm object and associated objects, and resume operation.
- Exit and restart.

To determine the appropriate recovery action, see [Recovery of a Disabled Process Restart versus Reopen the Realm](#).

To do either recovery action, see [Clean-Up](#).

Groups of Applications

Applications can use the group facility to coordinate fault-tolerant operation, or to distribute operating roles among application processes.

Introduction to Groups

A set of application processes can join a group. Each member of a group receives an *ordinal*, that is, a number representing its current position within the group. An application can specify a weight in a group member process to influence the ordinal assigned. Application program logic uses the ordinal to select one of several possible operating roles.

A *group service* tracks the group members as processes join and leave the group, and as they disconnect from and reconnect to the FTL server. At each of these events, the group facility revises ordinals to restore a one-to-one mapping between n members and the positive integers $[1, n]$.

Developers are responsible for the names of groups and for the API calls that join groups. Administrators are responsible for configuring group communication.

Simple Fault Tolerance with Groups

Application programs can use the group facility to coordinate fault-tolerant operation.

In the following diagram, consider an application program that can operate in either of two roles, according to its application logic:

- A1 is the active role: the program subscribes to messages, processes each message, and sends another message in response.
- A2 is the standby role: the program subscribes to messages, but neither processes them nor sends responses.

Simple Fault Tolerance, Role Behavior

Ordinal	Role	Description
1	A1	Actively process messages
2 or greater	A2	Standby

In the following diagram, when a process instance of the program starts, it joins Group_A, and receives its ordinal. The first process to start (Process P1) receives ordinal 1, so according to its application logic, it enters role A1 to subscribe, receive, process, and send messages. The second process to join the group (Process P2) receives ordinal 2, so it enters the A2 standby role. As other processes join the group, they receive ordinals 3, 4, and 5 in sequence and enter the A2 standby role. These are Process P3 through Process P5 in the diagram. In the following Timeline table, time t1 describes this state.

Simple Fault Tolerance, Timeline

Timeline: Processes Join Group_A					
Time	Process P1	Process P2	Process P3	Process P4	Process P5
t1	Ord=1 Role= A1	Ord=2 Role=A2	Ord=3 Role=A2	Ord=4 Role=A2	Ord=5 Role=A2
t2	Ord=-1 Role=A2 (or exit)	Ord=1 Role= A1	Ord=2 Role=A2	Ord=3 Role=A2	Ord=4 Role=A2
t3	Ord=5 Role=A2	Ord=1 Role=A1	Ord=2 Role=A2	Ord=3 Role=A2	Ord=4 Role=A2

Time t2 in the previous diagram illustrates the state when Process P1 exits, or becomes disconnected from the group service. All group members receive new ordinals within the group, usually decrementing their existing ordinal. In particular, Process P2 receives ordinal 1, enters role A1, and begins processing messages and sending responses. If Process P1 is still running while disconnected from the group service, then the group facility assigns it ordinal -1 and attempts to reconnect to the group service. The program can either exit, or enter the standby role A2, according to its program logic.

Time T3 illustrates the state when Process P1 restarts, or reconnects to the group service. Process P1 receives the lowest unassigned ordinal, and operates in the corresponding role, A2. Notice that Process P1 does not resume with ordinal 1. Instead, P2 retains ordinal 1.

Groups with More than Two Roles

You can extrapolate fault-tolerant behavior to an application with *several* distinct operating roles.

For example, consider an application that uses roles for graceful degradation of service. Each role subscribes to a different subset of messages by using different content matchers, and processes matching messages.

In the following diagram:

- B1 is an active role: The program processes high priority messages with matcher M1, and ignores the rest. The member with ordinal 1 operates in this role.
- B2 is an active role: The program processes medium priority messages with matcher M2, and ignores the rest. The member with ordinal 2 operates in this role.
- B3 is an active role: The program processes low priority messages with matcher M3, and ignores the rest. The member with ordinal 3 operates in this role.
- B4 is the standby role: The program does not process any messages. Any member with ordinal 4 or greater operates in this role.

More than Two Roles, Role Behavior

Ordinal	Role	Description	Matcher
1	B1	Actively process high priority messages	M1
2	B2	Actively process medium priority messages	M2
3	B3	Actively process low priority messages	M3
4 or greater	B4	Standby	

The following diagram shows ordinals and roles based on the number of processes running. When only one instance of application B is running, it has ordinal 1, role B1, and it

processes messages with the highest priority. It does not devote any time to messages with medium or low priority.

When two instances are running, one instance, with ordinal 1, role B1, processes messages with high priority. The other instance, with ordinal 2, role B2, processes messages with medium priority. Neither devotes any time to messages with low priority.

When three instances are running, each instance processes messages with a different priority, and together they cover all the priorities. That is, one instance has ordinal 1, role B1, and processes messages with high priority. Meanwhile, the second instance, with ordinal 2, role B2, processes messages with medium priority. The third instance, with ordinal 3, role B3, processes messages with medium priority.

When four or more instances are running, the instances with ordinal greater than 3 idle in standby role B4, ready to process messages if an instance in any of the processes in the three active roles were to become unavailable.



Tip: Notice that when the process with ordinal k becomes unavailable, all processes with ordinals greater than k shift their roles. When developing programs, design program logic to make this shift smoothly and rapidly.

More than Two Roles, Timeline

Processes Running	Process P1	Process P2	Process P3	Process P4	Process P5
1	Ord=1 Role=B1				
2	Ord=1 Role=B1	Ord=2 Role=B2			
3	Ord=1 Role=B1	Ord=2 Role=B2	Ord=3 Role=B3		
4 or more	Ord=1 Role=B1	Ord=2 Role=B2	Ord=3 Role=B3	Ord=4 Role=B4 standby	Ord=5 Role=B4 standby

Weights for Group Members

An application can assign a weight to a fault tolerance group member process. The weight influences the FTL group service's assignment of the ordinal, which determines the operating role, such as 1 (active). Use weights to make decisions based on hardware speed, hardware reliability, load factors, or other operating environment factors. Programs can adjust weights based on system conditions.

Weights, Ordinals, and Status

When a process joins a group, a member of greater weight always outranks members of lower weights. The group member process with the greatest weight is assigned ordinal 1 and the status is active (the leader member of the group).

For example, in the following diagram, Member A runs on a computer that is much faster than Member B, so Member A is assigned the greatest weight (200). The FTL group service assigns Member A ordinal 1 which is active status and Member B (weight 100) ordinal 2 which is standby status. Members C and D run on equally fast computers with approximately equal load factors so they are assigned an equal weight of 50 each, which results in the assignment of ordinals 3 and 4 and standby status.

Weights Assigned to Member Processes

Member	Weight Assigned	Ordinal Assigned	Status
Member A	200	1	Active
Member B	100	2	Standby
Member C	50	3	Standby
Member D	50	4	Standby

Greater Weight Processes Join

When an inactive member with a greater weight re-joins a group, the member preempts the active member of a lower weight. For example, in the following diagram, when three processes are running, Member B has the greatest weight (100) and is the active member. When Member A (weight 200) joins the group and four processes are running, the FTL

group service re-assigns ordinal 1 (active) to Member A and sends an ordinal update to the other processes.

More than Two Roles, Timeline

Processes Running	Member B Weight 100	Member C Weight 50	Member D Weight 50	Member A Weight 200
3	Ord=1 Role=Active	Ord=2 Role=Standby	Ord=3 Role=Standby	
4 (Member A joins)	Ord=2 Role=Standby	Ord=3 Role=Standby	Ord=4 Role=Standby	Ord=1 Role=Active

Members with Equal Weight

Members of equal weight do not outrank each other and the order of equally weighted members joining the group is irrelevant to rank.

Considerations:

- If two members have equal weight, you cannot assume either member outranks the other or that either member will be the first to become the active member.
- An inactive member that is joining never preempts an active member with the same weight. For example, if Members Y and Z have equal weight, and Member Z is already active (ordinal 1), then Member Y does not preempt Member Z.

Adjusting Weights

In addition to specifying weight when a process joins a group, programs can adjust their weight at any time to reflect changing conditions. For example, when a load factor of a host computer changes, a member process might track the change and adjust its weight accordingly.

When a member's weight is adjusted, the FTL group service recomputes the ordinals members of the group. For large groups this re-computation can affect performance.

API Weight Settings

Weight is controlled via the FTL Group API. Considerations:

- Weights must be positive integers.
- Zero cannot be used as a weight and is reserved for future use.
- In groups where member weights are assigned, members without a weight are assigned a default weight of 100.
- In groups where no member weights are assigned, the member that joined first gets ordinal 1.

Groups Principles of Operation

Group Service

Group services coordinate all the members of a group.

Every FTL server automatically provides a group service. Programs use the group facility's API calls to interact with the group service, and to coordinate appropriate behavior.

Group Members

Each group has a unique *group name*, which identifies its members to the group service.

In each group, every member must be able to assume any role, according to its current ordinal. You can satisfy this requirement in a natural way if all the members are process instances of one application program, compiled from source code with identical functionality (though they may execute on different hardware and operating system platforms).

Member applications and the group service are all within the same realm, and usually run on host computers within a local area network.

The first member to join a group establishes heartbeat and timeout parameters for the entire group, based on the *activation interval* property that the first member supplies in the

join call. For consistent behavior, ensure that all members of a group supply the same activation interval.

An application can join several groups. Nonetheless, an application must *not* join a group a second time, that is, if it is already a member of that group.

Interactions within a Group

The *join* call establishes communication between a member and the group service. After the join call returns successfully, the group library continues exchanging heartbeats with the group service. (Heartbeats and other group protocols are not visible to your program code.)

The *leave* call explicitly ends the stream of heartbeats from a member. The group service immediately adjusts the ordinals of the remaining members.

Other events can *implicitly* interrupt the heartbeat stream. For example, the member process could abruptly exit, or network segmentation could separate the member from the group service. In these situations, the group service also adjusts the ordinals of the remaining members, but after a calculated delay (see [Activation Interval](#)).

When a member's ordinal changes, the group facility informs program code by raising an `ORDINAL_UPDATE` advisory message. Programs subscribe to this advisory, and application callback code must take appropriate action to change the member's operating behavior.

The group service tracks member heartbeats to determine the health of each member.

Individual members of a group do not exchange group protocol messages with one another, but only with the group service.

Side Effects of Groups

Group protocol traffic consumes network bandwidth. With fewer than 10 members, using the default activation interval, expect only minimal impact on message latency. More members or a smaller activation interval can increase the impact.

Techniques for Programming with Groups

Program Structure for Groups

Programs using the group facility follow this template.

1. Determine the group name and property values.

(See also, [Restrictions on Names](#).)

2. Subscribe to ordinal update advisories.

Content matchers can filter for these advisories by matching the string value `ORDINAL_UPDATE` in the name field, and the group name in the group field.

The ordinal field contains the group member's new ordinal.

The subscriber callback must change the operating role of the application process according to the new ordinal.

3. Join the group, using the join API call.
4. Explicitly leave the group, using the leave API call.

Programs must leave all groups before closing the realm object.

Activation Interval

When a member becomes disconnected, the group facility revises the ordinals of the remaining members, which respond by adjusting their operating roles. This process is not instantaneous: it takes time to *detect*, to *revise*, and to *adjust*.

You can supply the activation interval property as a guide for the group service during the detect and revise phases. The group service derives appropriate heartbeat and timeout intervals from this property, aiming to complete these two phases within the activation interval you supply. That is, when a member becomes disconnected, you can expect that all affected group members receive new ordinals in approximately this time.

In contrast, the length of the adjust phase depends on the application, and the time it requires to switch operating roles.

In the interim, the application processes as a group might miss some messages. You can implement strategies to compensate for this possibility, as appropriate to application requirements.

If you omit the activation interval property, its default value is 5 seconds.

Disconnect

Ordinal `-1` indicates that the group member is disconnected from the group service.

The group object automatically attempts to reconnect, and continues until the program explicitly destroys the object. Meanwhile, the group service could reassign the member's previous ordinal to another group member.

Appropriate program response depends on application requirements. For example, if it is not acceptable for two members to share the role, the program should respond by entering an idle or standby role. In contrast, if duplication of the role is acceptable, then the program could continue in that role while waiting to reconnect. Alternatively, the program could log an error and exit.

Group Status

Group status advisories report the state of group members.

Group members and observers can subscribe to receive group status advisories.

The group library presents members and observers with group status advisories, which can have one of three forms:

Total Status

A group status advisory that contains the group status of *all* existing member processes.

Change in Status

A group status advisory that contains the group status of *a subset* of members, including new members that joined the group, members that left the group, or members that became disconnected. Frequently, only one member at a time changes status, so the subset contains only that one member.

Server Unavailable

A group status advisory that indicates that the group service is unavailable or unreachable. This form does not contain the group status of any members.

When the state of the group changes, the group library presents members and observers with group status advisories. The form of the advisory depends on both the member and state change:

- When a *new member joins* a group, it receives a total status advisory, containing the status of all the members, excluding observers. All the other members and observers receive a change in status advisory.

When a *disconnected member reconnects*, this event results in the same set of status messages.

- When a *member leaves* a group, it does not receive a status advisory. All the other members and observers receive a change in status advisory.
- When a member becomes *disconnected* from the group service, it receives a server unavailable advisory. All the other members and observers also receive a change in status advisory.
- When a *new observer joins* a group, the group state does not change. The new observer receives a total status advisory, containing the status of all the other members, excluding any observers. Other members and observers do *not* receive a status advisory.
- When an *observer leaves* a group, or becomes *disconnected* from the group service, the group state does not change. The affected observer does *not* receive a status advisory, nor do any other members or observers.

For reference details about the group status advisory message, see [GROUP_STATUS](#).

Group Observer

An application program can join a group as an *observer*. Observers can monitor the group by subscribing to group status advisories, just as members can. However, the group service does not assign ordinals to observers: observers do not participate in the functional roles of the group.

When an observer joins or leaves a group, or becomes disconnected from the group service, the state of the group and its members does not change.

Programmer's Checklist Addenda for Groups

The items in this topic supplement the checklist items in [Information for Developers](#).

All Programming Languages

The FTL server and its group service must be available to application processes. For details, see TIBCO FTL [Administration](#).

C Programmer's Checklist

C programs must include the header file `tibgroup/group.h`.

On UNIX platforms, add the linker flag `-ltibgroup`.

On Windows platforms, link the corresponding C library file `tibgroup.lib`, and run using `tibgroup.dll`.

Java Programmer's Checklist

Java programs must import the group package:

```
import com.tibco.ftl.group.*;
```

The CLASSPATH must include the `lib` directory, which must contain the archive file `tibftlgroup.jar`.

.NET Programmer's Checklist

To simplify coding, programs can include this statement:

```
using TIBCO.FTL.GROUP;
```

The assembly `TIBCO.FTL.GROUP.dll` must be in the general assembly cache (GAC). If the .NET framework is properly installed, then installing TIBCO FTL registers the assembly with the GAC.

Persistence: Stores and Durables

Persistence Concepts

The key building blocks of FTL persistence are [Persistence Clusters](#) and [Persistence Stores](#).

In addition, when configuring persistence, you should expect to make at least these choices:

Will messages be persistent (preserved on server restart) or non-persistent (lost on server restart)? In FTL, “persistent” is referred to as “replicated”, and “non-persistent” is referred to as “non-replicated”. See [Replication](#).

How will FTL server store messages? In FTL, persistent (replicated) messages can be stored in memory for greater performance, or on disk for greater durability. Non-persistent (non-replicated) messages are always stored in memory. See [Storage](#).

What publisher mode will FTL clients use? In FTL, publishers may block in the send call until the server confirms that the message is stored. Or, for greater performance, publishers may return immediately from the send call, without knowing whether the message was stored successfully. See [Publisher Send Mode](#).

Persistence Clusters

A persistence cluster consists of a set of persistence services. A persistence service runs as part of an FTL server. You provision persistence services by editing the FTL server yaml configuration file. See [FTL Server and Services](#). You must also define persistence clusters and services in the FTL realm definition. See [Configuring Persistence](#) and [FTL Server GUI: Configuration](#).

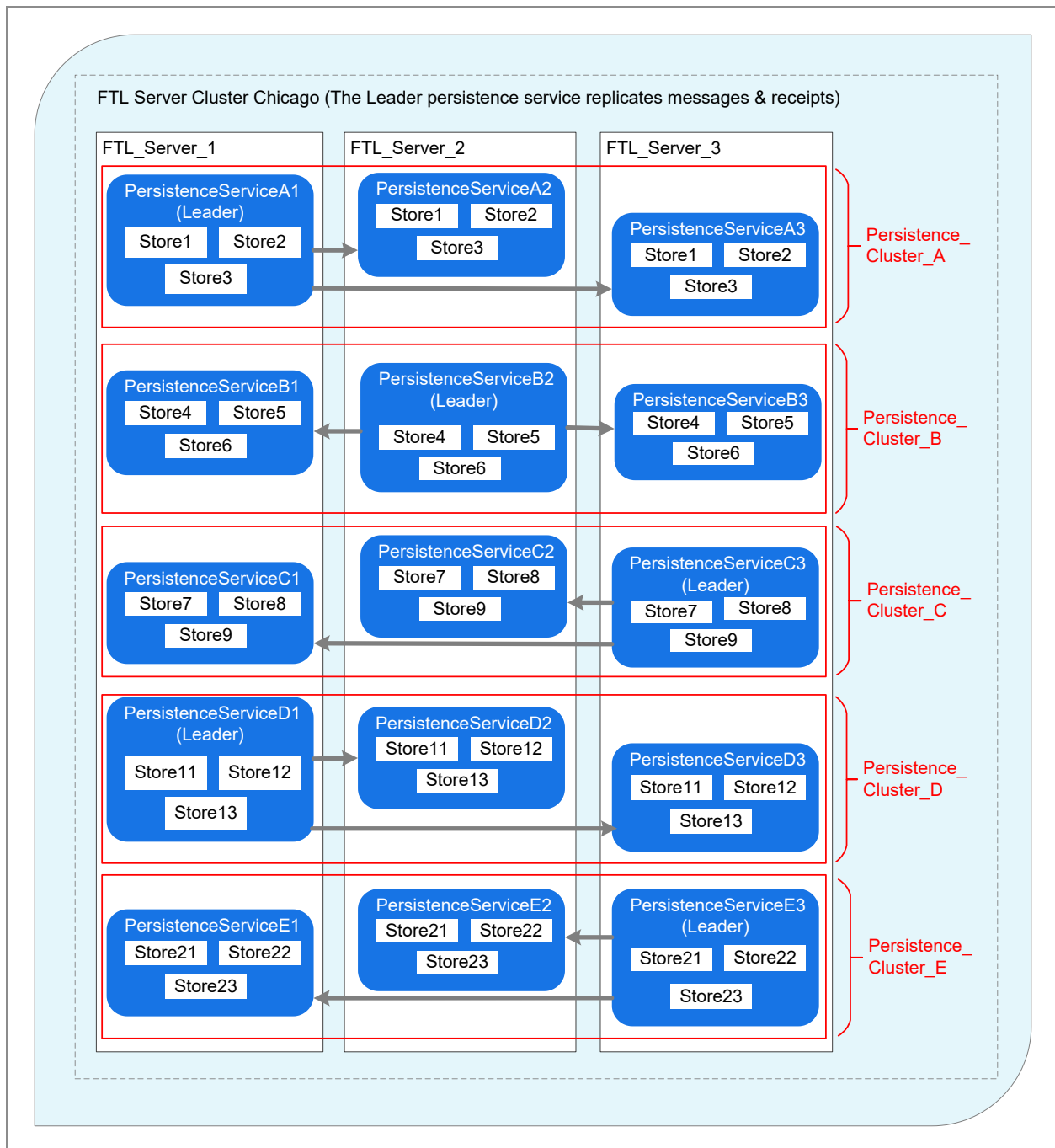
Alternatively, a default persistence cluster, consisting of a set of default persistence services, is automatically defined and provisioned when starting a cluster of FTL servers for the first time.

The persistence services in a cluster communicate with each other, choose a leader, and form a quorum. FTL clients always communicate with the leader. FTL clients cannot send or receive messages from the persistence cluster if there is no leader.

Persistence clusters provide fault tolerance. If the leader dies, or becomes unreachable on the network, another persistence service is chosen to be leader, a new quorum forms, and FTL clients fail over to the new leader. In general, to form a quorum, more than half of the persistence services in a cluster must be running and reachable. For example, 2 out of 3 services can usually form a quorum, but 1 out of 3 services cannot form a quorum. See [Persistence Services and Clusters](#) for more details about forming a quorum.

The leader replicates message and acknowledgment data to whichever members of the cluster are currently reachable and participating in the quorum. All members of the cluster store data either in memory or on disk. See [Storage](#). Messages are organized into persistence stores, as described in [Persistence Stores](#).

For example, in the following diagram, there are three FTL servers with five persistence services in each server. There are five persistence clusters of three members each.



Persistence Stores

A persistence cluster manages a set of persistence stores. A persistence store is a set of configuration parameters that determine how published messages are handled. For example, the persistence store determines whether:

- Messages are replicated to all persistence services. See [Replication](#).
- Messages are routed to other persistence clusters. See [Routing](#).
- Publishers block in the send call or return immediately. See [Publisher Send Mode](#).

Persistence stores can also be used to set limits on how much data is stored, either by the total number of messages or the total size of messages.

By default, a persistence store is owned by one persistence cluster. However, a persistence store can be routed, in which case its messages are distributed to all persistence clusters in a persistence zone. See [Routing](#).

Administrators choose which persistence stores an FTL client uses. The choice of store determines the persistence cluster. (In the routing case, the FTL client uses whichever persistence cluster is available locally.)

In the destinations messaging model, each configured destination is mapped to a persistence store.

In the content matching messaging model, each endpoint is mapped to a persistence store.

Replication

A persistence store may be either replicated (“persistent”) or non-replicated (“non-persistent”). See [Stores Grid](#).

In a replicated store, all message and acknowledgment data is replicated by the leader to the other members of the persistence cluster. Therefore, if the leader restarts or becomes unreachable, messages and acknowledgments are preserved. FTL clients fail over to the new leader and continue publishing and consuming messages.

If a synchronous publish or acknowledge call takes place in a replicated store, then the following will occur:

- The FTL client sends its request to the leader.
- The leader replicates the data to the other members of the cluster.
- The leader stores the data in memory or on disk, as described in [Storage](#).
- Concurrently, the leader waits for a majority of members to respond, indicating that the data is stored. Data may be stored in memory or on disk, as described in [\[Storage\]](#).
- The leader responds to the client.

In a non-replicated store, all message and acknowledgment data is stored only in the memory of the leader. Therefore, if the leader restarts or becomes unreachable, all pending messages are lost.

If a synchronous publish or acknowledge call takes place in a non-replicated store, then the following will occur:

- The FTL client sends its request to the leader.
- The leader immediately responds to the client.

Non-replicated stores offer reduced latency and improved throughput at the expense of message durability.

Publish and acknowledge calls are not always synchronous. See [Publisher Send Mode](#) and [Subscriber Acknowledgment](#).

Storage

In non-replicated stores, message and acknowledgment data is always stored in the memory of the leader.

In replicated stores, message and acknowledgment data may be stored in memory or on disk. This is a cluster-wide setting; see [Cluster Details Panel](#).

In-memory storage offers improved performance at the expense of durability. When a persistence service restarts, it must copy all message and acknowledgment data from other members of the cluster before rejoining the quorum. In addition, if a majority of the persistence services in the cluster fail simultaneously, data may be lost.

For storage on disk, you supply a data directory in the FTL server yaml configuration file; see [FTL Server and Services](#). FTL offers two modes for storage on disk:

- Sync: The leader responds to the client after a majority of members have written the data (message or acknowledgment) persistently to disk. This mode generally provides high durability at the cost of increased latency and lower throughput.
- Async: The leader responds to the client after a majority of members have instructed the OS to write the data (message or acknowledgment) to disk. The write may be lost if the OS crashes or the host computer loses power. This mode generally provides reduced latency and higher throughput, but messages could be lost if a majority of host computers fail simultaneously.

See [Disk-Based Persistence](#) for more details regarding on-disk storage, including compactions.

Message Swapping

If memory on FTL server's host computer is constrained, persistence services can swap messages to disk. Swapping reduces persistence service memory usage at the expense of additional disk I/O.

Messages in both non-replicated or replicated stores can be swapped to disk. This is a cluster-wide setting; see [Cluster Details Panel](#).

When messages are swapped to disk, the message body is freed from persistence service memory. However, message metadata is retained in persistence service memory. Therefore, for each message stored, some memory will be consumed. The best practice is to empirically determine the amount of memory needed for your desired message backlog. Then, size the environment accordingly.

If in-memory storage is enabled for the persistence cluster, then an asynchronous write will be issued as messages are published by FTL clients. When messages are delivered to consumers, a read may be required. There must be enough disk bandwidth to support the messaging rate.

If disk-based persistence is enabled for the cluster, then no additional write will occur. However, a read may be required when delivering messages to consumers.

Publisher Send Mode

Each persistence store has a publisher mode. The publisher mode may be `store_confirm_send` or `store_send_noconfirm`. See [Stores Grid](#).

If the publisher mode is `store_confirm_send`, then the publisher will not return from the send call until the persistence cluster leader confirms that the message has been stored. For example, if disk-based persistence is enabled, and the persistence store is replicated, the send call will not return until a majority of persistence services have written the message to disk. This publisher mode offers stronger delivery assurance at the expense of increased latency and lower throughput.

If performance with `store_confirm_send` is inadequate, consider increasing the number of publishers. Or, you may modify your application to use the non-inline send policy, which improves throughput at the expense of a somewhat more complex application. See [Publishers](#) for more information.

If the publisher mode is `store_send_noconfirm`, then the publisher will return from the send call regardless of whether the persistence cluster stored the message. The message is lost if the persistence cluster did not receive the message, or no leader was available at the

time. In addition, the client program is responsible for ensuring that it does not send messages too quickly. If it sends messages faster than the persistence cluster can replicate and/or store them, then messages will be lost. This publisher mode offers reduced latency and higher throughput at the cost of weaker delivery assurance and a potentially more complex application.

Topic Subscriptions

An FTL client's interest in a topic is represented by a durable. So long as the durable exists, the persistence cluster will store messages sent to the topic, to be delivered to subscribers connected to the durable.

To subscribe on a topic, FTL client programs pass a topic name and durable name to the subscribe call. There may be many durables on a given topic, each with a different durable name. The topic is mapped to a persistence store in the FTL realm definition.

When creating a subscriber on a topic, the client program must also choose what type of subscription to create. There are three subscription types:

- **Standard:** The durable is associated with one subscriber. All messages sent to the topic are delivered to all standard subscriptions on the topic. Creating a second subscriber with the same durable name will cause the first subscriber to be closed, and messages will be delivered to the second subscriber.
- **Shared:** The durable is associated with one or more subscribers. All messages sent to the topic are delivered to one of the subscribers on the durable. Shared subscriptions allow multiple client programs to parallelize the work of consuming messages on a topic.
- **Lastvalue:** The durable is associated with one or more subscribers. When a subscriber starts, it receives the last message sent on its topic (rather than all messages). Subsequent messages published to the topic are delivered to each online subscriber on that topic. Lastvalue subscriptions allow subscribers to initialize to the most recent state for a topic, and then receive updates for that topic.

See [Destination Concepts](#) and [Subscribers](#) for more information.

Queue Subscriptions

The persistence cluster will store messages sent to a queue, regardless of whether any subscribers have ever existed.

To subscribe to a queue, FTL client programs pass the queue name to the subscribe call. The queue is mapped to a persistence store in the FTL realm definition.

Queue subscriptions are shared, meaning multiple client programs may subscribe to the queue. Messages sent to the queue are delivered to any one of the subscribers.

See [Destination Concepts](#) and [Subscribers](#) for more information.

Content Subscriptions

This information does not apply to the destinations messaging model.

In the content matching messaging model, subscribers express interest by indicating what message fields, or combination of message fields, they expect to receive. See [FTL Configuration Overview](#).

To subscribe, FTL client programs pass a durable name, endpoint name, and content matcher to the subscribe call. The durable name identifies the subscription. The endpoint name is mapped to a persistence store in the FTL realm definition. The content matcher determines the names and values of certain expected fields in the message.

The endpoint name is also mapped to a dynamic durable template in the FTL realm definition. The administrator chooses the subscription type by configuring an appropriate durable template.

There are three types:

- **Standard:** The durable is associated with one subscriber. All messages sent to the persistence store that match the durable's interest are delivered to its subscriber. Creating a second subscriber with the same durable name will cause the first subscriber to be closed, and messages will be delivered to the second subscriber.
- **Shared:** The durable is associated with one or more subscribers. All messages sent to the persistence store that match the durable's interest are delivered to one of the subscribers on the durable. Shared subscriptions allow multiple client programs to parallelize the work of consuming messages on a topic.
- **Lastvalue:** The durable is associated with one or more subscribers. When a subscriber starts, it receives the last message sent that contains the key field and value that the subscriber indicated in its content matcher (rather than all messages). Subsequent messages published are delivered to each online subscriber on that key. Lastvalue subscriptions allow subscribers to initialize to the most recent state for a key, and then receive updates for that key. See [Stores for Last-Value Availability](#).

See [Persistence Stores for Content Matching](#) for more information.

Subscriber Acknowledgment

In general, when a subscriber receives a message, it must acknowledge the message after processing it. When a subscriber acknowledges a message, the persistence cluster will replicate and store the acknowledgment, just as it would for message data. If the message is no longer needed (there are no more subscribers), each persistence service in the cluster will delete the message from storage. Acknowledgments that are no longer needed will also be deleted from storage.

Lastvalue subscribers (topic-based or content-based) do not acknowledge messages. The most recent message is replaced whenever a new message is published.

All other subscribers do acknowledge messages.

- If the client program specifies explicit acknowledgment when creating a subscriber, then the client program must issue an acknowledge call for each message.
- Otherwise, the FTL library will automatically issue the acknowledge call once the subscriber's dispatch callback returns, for all messages processed by the callback.

Subscribers also have an acknowledgment mode (sync or async).

- Sync acks: Acknowledgment calls are synchronous. The call does not return until the persistence cluster confirms that it has stored the acknowledgment. This type of acknowledgment minimizes duplicates.
- Async acks: Acknowledgment calls are asynchronous. FTL will send the acknowledgment in the background on a timer. Duplicates can occur if the client program fails before the acknowledgment is sent.

Client programs may specify the acknowledgment mode (sync or async) when creating a topic or queue subscriber. If no acknowledgment mode is specified, the administrator can specify a default value in the destination configuration. See [Destination Details Panel](#).

Otherwise, if you are using the content matching messaging model, then administrators specify the acknowledgment mode (sync or async) when creating the dynamic durable template for the client program's endpoint.

Routing

Persistence stores may be routed or non-routed. Non-routed stores are owned by one persistence cluster. Message and acknowledgment data exists only at that persistence cluster.

You may wish to replicate messages to different geographic regions. However, locating the persistence services of a persistence cluster in different regions may result in poor performance due to higher network latencies or reduced network reliability. Instead, consider running one persistence cluster in each region and creating routes between them. In FTL, routes are implemented by creating routed (or “wide-area”) persistence stores.

Routed stores belong to one or more forwarding zones. A forwarding zone is a set of persistence clusters that connect to each other to exchange messages. See [Zones Grid](#).

The routed store exists at each persistence cluster in each forwarding zone to which the store belongs. When an FTL client interacts with a routed persistence store, it will use whichever persistence cluster exists at the FTL server(s) to which the FTL client connects. FTL client API calls return when the local persistence cluster confirms that the message is stored locally (not necessarily replicated across regions).

A message published at any one of the clusters may be forwarded to all other clusters in all forwarding zones to which the persistence store belongs. Forwarding zones can be “chained” together if a persistence cluster belongs to more than one forwarding zone.

Queues and maps cannot be created in routed stores.

Topic subscriptions of any type can be created in routed stores. A message sent to a topic at one persistence cluster will be forwarded to the other persistence clusters in the forwarding zone(s), provided that the other cluster has interest (has a subscription for the topic).

For users of the content matching message model: Content-based subscriptions of any type can be created in routed stores. A message sent to a persistence store at one persistence cluster will be forwarded to the other persistence clusters in the forwarding zone(s), provided that the other cluster has interest (has a matching subscription for the message).

In the following diagram, StoreS is a wide-area (routed) store and is assigned to Zone Z.

The FTL server clusters in Zone Z are: Rio, London, and Tokyo. Each cluster implements a StoreS Projection from StoreS. For example, a publisher in the London Cluster publishes a message to the StoreS Projection with Durable B and then the message is made available to Durable A (Rio Cluster) and Durable C (Tokyo Cluster), as long as they are interested subscribers. StoreT is not a wide-area store, so it remains functionally outside Zone Z and local to the Tokyo cluster.

Disaster Recovery

In the event of a disaster, all persistence services in a persistence cluster may go offline or lose their underlying disk storage. If this happens, FTL clients would be unable to proceed,

or stored messages might be lost.

In FTL, you may configure disaster recovery for each persistence cluster. If disaster recovery is configured, then message and acknowledgment data is replicated asynchronously to a disaster recovery site in a different location. FTL client API calls do not wait for data to be replicated to the disaster recovery site.



Note: Disaster recovery is not supported for the default persistence cluster.

To configure disaster recovery, you create a persistence cluster with two “sets” of persistence services. See [Clusters Grid](#).

One set is the “primary” set. The persistence services in the primary set choose a leader and form a quorum as usual. All FTL clients interact with the primary set.

The second set is the “standby” set. The persistence services in the standby set choose a leader to communicate with the primary set. The standby leader receives asynchronous data from the primary set and distributes it to the other persistence services in the standby set.

When a disaster occurs, you activate the “standby” set by changing its role to “primary”. You reconfigure DNS in your enterprise so that FTL clients reconnect to the former “standby”, now “primary” set. Since replication of data to the standby set was asynchronous, a tail end of the data may be lost.

See [Disaster Recovery](#) for full details on this procedure.

Monitoring and Logging

Persistence services can be managed via the FTL server GUI or the FTL server web API.

Monitoring metrics from the persistence service can be accessed via FTL server’s prometheus endpoint. The persistence service also logs various statistics periodically, such as disk usage and write rates. See [Persistence Monitoring and Management](#). See also [Catalog of Persistence Metrics](#).

For development and debugging purposes, tracing of individual messages can be enabled. See [Message Tracing](#).

Permissions

If you have enabled authentication at FTL server, you may optionally enable fine-grained permissions for destinations or persistence stores. See [Authentication and Authorization](#).

Publishers

Publisher Mode and Send Policy

There are two parameters that drive the behavior of publishers: the publisher mode and the send policy.

Administrators set the publisher's mode when configuring a store. The publisher mode is either `store_confirm_send`, which offers maximum delivery assurance, or `store_send_noconfirm`, which offers minimal delivery assurance. For details, see [Publisher Send Mode](#).

Applications may set the publisher's send policy when creating a publisher object. In addition, administrators may set a default send policy in the realm configuration.

The send policy may be inline or non-inline. In general inline send offers better latency, while non-inline send offers better throughput.

When the publisher mode is `store_send_noconfirm`, the send call returns immediately, regardless of the send policy. The application is not notified as to whether the message was persisted or lost. In this case, the non-inline send policy may offer somewhat improved throughput in exchange for higher latency.

When the publisher mode is `store_confirm_send`, the FTL library will attempt to persist the message for the publisher's retry duration. The retry duration can be set by the application. Administrators may set a default value in the realm configuration. See the [Administration](#) guide, [Realm Properties Details Panel](#), section [Persistence Retry](#).

If the publisher mode is `store_confirm_send`, and the send policy is inline, then the send call will block until the message is persisted, or the retry duration has elapsed. If the FTL library could not ensure that the message was persisted, an exception is raised by the send call. This is a synchronous send call, where performance is tied directly to the latency to the persistence service. It may be necessary to use the batched send call (`tibPublisher_SendMessages`), or multiple publishers in parallel, to achieve acceptable performance.

If the publisher mode is `store_confirm_send`, and the send policy is non-inline, the send call may return immediately, provided that the maximum batch count has not yet been reached. The application may set the maximum batch count. See [Configuring Publisher Send Policy](#).

When the send policy is non-inline, the FTL library attempts to persist the message in the background for the publisher's retry duration. If the latency to the persistence service is large, a non-inline send may offer a substantial improvement in throughput compared to

inline send. However, failures are signaled to the application asynchronously, and handling failures may require a more sophisticated application.

If using a non-inline send, it is critical that the application flushes or closes the publisher before exiting. If the application merely exits, the send call may not be complete, and the message may be lost.

i Note: Calls to set keys in a map (`tibMap_Set` and `tibMap_SetMultiple`) are always synchronous. That is, they behave as if the publisher mode is `store_confirm_send` and the send policy is inline.

Configuring Publisher Send Policy

Send policy is configurable via the realm properties and at the API level. The API property set for inline or non-inline sends during publisher creation takes precedence over the realm property.

Dynamic update of the send policy realm property is allowed. See [Realm Properties Details Panel](#). However it applies only to the newly created publishers.

Applications may override the send policy when creating a publisher object (`TIB_PUBLISHER_PROPERTY_INT_SEND_POLICY`).

When using non-inline send, the application may specify a maximum batch count (`TIB_PUBLISHER_PROPERTY_INT_MAX_BATCH_COUNT`) when creating the publisher. The maximum batch count is an upper limit on how many messages the FTL library may have that are not yet persisted (in other words, how many messages are in flight to the persistence service). If this limit is reached, the send call will block. This limit may also be used to determine which messages might have been lost in the event of a failure.

If using non-inline send, a situation may occur where an application closes a publisher shortly after making a send call. Since the send may not be complete, the FTL library will linger for some amount of time while the send is retried. If the message cannot be persisted within the linger period it may be lost. The application may override the publisher close linger period when creating the publisher (`TIB_PUBLISHER_PROPERTY_DOUBLE_PERSISTENCE_CLOSE_LINGER`).

FTL provides APIs that an application may use to determine the send policy and max batch count of a publisher object.

Detecting Failures with Non-Inline Send

When the send policy is non-inline, and a message cannot be persisted (perhaps because the retry duration has elapsed), the failure may occur in the background. The application will be notified, possibly after the fact, in one of two ways:

- An exception is generated by an ongoing or future send call
- An exception is generated by an ongoing or future flush call

However if the failure occurs during the publisher close linger period (for example, if a close call is made while the FTL library has messages that are not yet persisted), no exception is generated. Applications may wish to make a flush call before the close call.

When an exception is generated, the application must assume that up to maximum batch count messages might have been lost, in addition to the message that was passed to a failed send call. For example, if the maximum batch count is 100, and the send call fails, the application should assume that the current message and the previous 100 messages might have been lost. Applications may wish to keep information about the most recent messages for diagnostic or other purposes. Re-sending the most recent messages is possible, but may result in duplicates.

If the application does not make send or flush calls frequently, it may subscribe to an advisory for non-inline send loss (`TIB_ADVISORY_NAME_NON_INLINE_SEND_LOSS`). When this advisory is received, it is a hint to the application that an error has occurred in the background, and that a future flush or send call will generate an exception.

Maximum Message Size

A good rule of thumb is to limit the maximum size of persisted messages to 10 MB each. Messages that exceed the environment's maximum message size could disrupt the quorum of persistence services, and prevent them from reforming a quorum.

The actual maximum can vary depending on network and host hardware. To determine the actual maximum, test empirically in each deployment environment.

Administrators may enforce a maximum message size for a given persistence store. See [Store Detail Panel](#).

In advanced configurations, you may increase the backlog buffer size of all persistence service transports to accommodate large messages. See [Persistence Service Details Panel](#).

Subscribers

For an introduction, see [Topic Subscriptions](#), [Queue Subscriptions](#), or [Content Subscriptions](#).

Acknowledgment of Messages

A durable subscriber can acknowledge consuming a message to its durable in either of two ways. (Last-value subscribers do not acknowledge messages.)

Automatic Acknowledgment

With automatic acknowledgment, the base library automatically acknowledges the message when the application callback returns. This choice is appropriate when an application's callback completely processes the inbound message and then returns.

Explicit Acknowledgment

With explicit acknowledgment, the application program must use the acknowledge call to explicitly acknowledge each message. This choice is appropriate when an application's callback delegates processing of the inbound message to a separate thread. It could also be appropriate if the application must verify the success of each acknowledgment.

To enable explicit acknowledgment, programs pass a property to the subscriber create call. Otherwise, the default behavior is automatic acknowledgment.

The subscriber must acknowledge each message as received directly from the durable, using the Acknowledge call.



Note: The choice between these automatic and explicit acknowledgment is orthogonal to the choice of [Acknowledgment Mode](#).

Acknowledgment Mode

Durable subscribers acknowledge each message as they process it. (Last-value subscribers do not acknowledge messages.) However, the client library can transmit acknowledgments to the cluster in either of two modes: synchronous or asynchronous.

Synchronous

Each time the application program acknowledges a message, the client library immediately communicates the acknowledgment to the cluster, and waits for the cluster to confirm that it has safely recorded that acknowledgment.

Asynchronous

The client library gathers batches of acknowledgments, and communicates the batches to the cluster. (The application does not receive confirmation that the cluster has safely recorded the batch of acknowledgments.)

In some situations, batching can improve throughput and latency performance. However, if the client program fails or closes its subscription before acknowledgments can be transmitted to the persistence cluster, the acknowledgment might be lost, and duplicate messages might be delivered.

i Note: Acknowledgment mode as described here is orthogonal to the developer's choice between automatic acknowledgment versus an explicit acknowledge call.

Delivery Count

Standard topic durables, shared topic durables, and queues count the delivery attempts for each message. If the administrator sets an optional maximum delivery limit for a destination, the persistence cluster can discard a message after the number of failed delivery attempts exceeds that limit.

While this mechanism never discards a message before reaching the limit, it does not guarantee a discard at exactly the limit.

Program code can retrieve the delivery count from message objects in the subscriber's message callback. (In C API, `tibMessage_GetDeliveryCount`.)

Message ID

Standard topic durables, shared topic durables, and queues include a message ID when delivering a message.

For all messages in a given persistence store, the message ID is monotonically increasing. That is, under normal circumstances, the message ID of the first message to be delivered is strictly less than the message ID of the second message to be delivered, and so on. (Since a given subscriber is always associated with one persistence store, a particular subscriber will also observe monotonically increasing message IDs.)

If a subscriber observes message IDs that are not monotonically increasing, that could indicate that another subscriber on the shared topic durable or queue failed to process a message, and that message was re-delivered to the local subscriber.

Program code can retrieve the message ID from message objects in the subscriber's message callback. (In C API, `tibMessage_GetStoreLocalMessageId`.)

Message Destination

Subscribers on a topic or queue destination receive messages from that destination.

Program code can retrieve the message destination from message objects in the subscriber's message callback. (In C API, `tibMessage_GetDest`.)

i Note: Messages received from the internal monitoring, logging, or advisory destinations do not indicate their destination. The call to retrieve the message destination returns NULL. Program code can define separate message callbacks to handle these special destinations.

Message Browsing

FTL client programs can browse messages stored in a shared topic durable or queue. For example, the client could run analytics on the data or delete a message. Messages are browsed from oldest to newest. A matcher can be used to browse a subset of the messages. Browsing does not affect delivery of messages to subscribing clients. Messages can be delivered to other subscribing clients while simultaneously being browsed.

Messages released by a consumer while a browser is running are not browsed unless the browser is restarted or re-created. This may happen, for example, because a consumer failed while processing some messages.

In the event of leader failover or connection loss, browsers need to be restarted or closed.



Warning: Creating too many browsers is not recommended, as it can put a strain on the persistence service resources, including both CPU and memory usage.

No-Local Message Delivery

Durable subscriptions on standard topic durables may enable no-local message delivery. This is enabled either through a realm property (see [Realm Properties Details Panel]) or through the client API. (In C API, `TIB_SUBSCRIBER_PROPERTY_BOOL_NOLOCAL_MESSAGE_DELIVERY`.)

When no-local message delivery is enabled, messages sent from the same client (specifically, publishers created from the same realm object) are not delivered to the subscriber.

No-local message delivery may only be enabled for standard durables.



Note: No-local delivery is enforced only after the no-local subscriber is created. Any messages already stored in the durable will be delivered to a no-local subscriber, regardless of which client sent the messages. The best practice is to create subscribers before creating publishers.

Retention Time

You may configure a durable or durable template with a retention time. Only standard durables with prefetch and shared durables may have a retention time.

For more information, see [Durable Details Panel](#)

The retention time feature allows users to keep messages for a certain period. This would not affect the ability of consumers to receive and acknowledge messages normally. The messages may be replayed later by rewinding the durable, for example if some analysis is required, or lost data needs to be recovered. For more information, see [Rewinding a Subscription](#).

Administrators should be mindful of how much data is retained. A long retention time can greatly increase storage requirements.

When retention time is set, and a subscriber acknowledges a message, the persistence service does not always delete the message immediately. Instead, if the message's age, as measured from send time, is less than the retention time, the message is retained. Later, once the message's age exceeds the retention time, it is deleted permanently.

When retention time is set, and a message's age exceeds the retention time, but the message has not been acknowledged, the message is not deleted. The message is deleted immediately if it is acknowledged.

Acknowledged and retained messages cannot be browsed or delivered to subscribers. The administrator or the client application must rewind the durable to make those messages available again.

Administrators can monitor the number of acknowledged or unacknowledged messages. FTL server reports the total message count for each durable (which includes acknowledged and unacknowledged messages), as well as the unacked message count for each durable. For more information, see [Durables List](#).

Retention time does not override byte limits, message limits, delivery limits, or message TTL. Once any of those limits is reached, the message may be discarded, regardless of whether it has been acknowledged and regardless of whether the retention time has been set.

For example, if message TTL and retention time are both set, and the message's age is greater than the TTL, the message will be discarded, regardless of retention time. Similarly, if the message has been acknowledged, and its age is greater than the retention time, the message will be discarded, regardless of message TTL. Message TTL, byte limit, message limit, and delivery limit should be thought of as storage limits, orthogonal to retention time.

Rewinding a Subscription

When retention time is enabled, see, [Retention Time](#) an administrator or client application may rewind the durable. Rewinding a durable means that all messages present in the durable, including messages previously acknowledged, are now delivered to newly created subscribers (or browsers). This allows you to replay messages from a durable.

Administrators can rewind any durable through the REST API [POST persistence/clusters/<clus_name>/stores/<stor_name>/durables/<dur_name>](#) Or, administrators can rewind a durable through the user interface, see [Durables List](#).

Client applications can rewind any durable via the `tibRealm_RewindSubscription()` API, or equivalent language binding. If permissions are enabled for persistence, the client must have the subscribe permission to do so. For more information, see : [Required Permissions for API Calls](#)

Before rewinding a durable, all subscribers on the durable must be closed. Once rewind completes, subscribers can be created normally, possibly using the exact code as before to process and acknowledge messages.

Rewinding a durable clears the acknowledged status of all messages. However, rewinding a durable does not reset the message's age. For example, rewinding a durable does not cause an acknowledged message to be retained for any longer than normal, and it does not affect the behavior of message TTL.

Key-Value Maps

Programs can use maps to store key-value pairs in a persistence store.

A map behaves like a simple database table with two columns: key and value. The key is a string, and the value is a message.

Programs assign a name or destination to each map. A store may contain many maps, each with a unique name or destination.

API methods can do these operations:

- Create a map.
- Set a key-value pair or multiple key-value pairs.
- Get a key's value.
- Remove a key-value pair.
- Iterate over all the key-value pairs in a map, or some of the key-value pairs using a string matcher.
- Close a map object.
- Delete a map from the store.

In addition, most of these methods are available as locked operations. That is, you can use a lock to ensure that map operations in different processes do not interfere with one another (see [Locks](#)).

Locks

A lock is used to ensure exclusivity of a series of map operations in a persistence service.

If a series of client map API calls using a lock completes without an exception, then an application may be assured that no other writer using the same lock concurrently modified a map in the persistence service, and that the modifications will be visible to future readers using the same lock.

If a map API call using a lock fails, then nothing may be assumed about the result of that operation, and the lock must be returned or destroyed before trying again.

A typical use is to implement atomic test-and-set on a particular row of a map. See the sample program `tibftlmap-lock`.

Lock Name

When a program creates a lock object, it assigns a lock name. A persistence cluster may contain many locks, each with a unique name. Two lock objects with the same name represent the same lock, so two or more application processes can use the lock to cooperate.

Two lock objects with different names do not interact with each other or block each other in any way.

Programmers choose the name of a lock to indicate the purpose of the lock: for example, `ServerStatusMapLock` might lock an entire map that stores the states of server hardware within an enterprise, while `BlueLock` might lock the row of that map that describes the state of the server named Blue.

Methods of Lock Objects

API methods can do these operations:

- Create a lock.
- Return (that is, release) a lock that the process holds.
- Steal a lock that another process holds.
- Destroy a lock object to reclaim resources.

Methods with Locks and without Locks: Effective Use

Map operations are available in two forms: with lock and without lock. Methods that take a lock argument respect the state of that lock: that is, if another process holds the lock, then one of two outcomes will occur:

- If program code specified a retry duration when creating the lock, the call will block until the retry duration elapses. If the lock is returned by the other process before the retry duration elapses, the call can proceed. Otherwise, the call will raise an exception, and the requested operation (for example, get or set a key) will not occur.
- If program code did not specify a retry duration when creating the lock, the call will raise an exception, and the requested operation (for example, get or set a key) will not occur.

However, methods that do not take a lock argument do not respect the state of any lock: even if another process holds a lock for the map or for the key row, the method does not test the lock, and completes its map operation.

For locks to effectively prevent interference, all your application processes must access the map using only methods with locks. Lock objects with different names do not interact with each other or block each other in any way.

Pattern for Programming with Locks

1. Create a lock object.
2. Call one or more methods with that lock.

A program can call many methods with that lock, even within a loop, as with a map iterator.

While the process still holds the lock, these methods succeed. Otherwise they throw an exception. If an exception occurs, program code must return the lock before making another API call with the lock. This is necessary in order to clear the lock's broken state, and to make client code recognize that its view of the map may be out of date. See [Breaking a Lock](#).

3. Return the lock.

Breaking a Lock

If a client program holds a lock, the lock can be lost (broken) under the following conditions:

- The client program disconnects from the persistence cluster due to a network interruption.
- Another client program steals the lock. See below.

When either condition occurs, the ongoing or next API call with the lock will raise an exception. This indicates that the program's state is no longer valid (for example, another program may have acquired the lock and made changes to the map). Program code must return the lock, clearing its state, before attempting further API calls with the lock.

Locks and Stores

When a process acquires a lock, it associates the lock with a specific persistence cluster. When a process releases a lock, it cancels the association, making the lock available to use with any persistence cluster.

While a lock is associated with one persistence cluster, it is illegal to use it in a map operation on any other persistence cluster. Method calls that violate this restriction throw an exception.

Steal: Only to Circumvent a Blocked Lock

If a program process, A, holds a lock and does not return it in a timely fashion, the corresponding map or row remains inaccessible to other process that request that lock. To circumvent this blockage, another program process, B, can steal the lock.

When process A subsequently uses that lock in a map method call, the method throws an exception.

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

Documentation for TIBCO FTL® - Enterprise Edition is available on the [TIBCO FTL® - Enterprise Edition Product Documentation](#) page.

TIBCO eFTL™ Documentation Set

TIBCO eFTL software is documented separately. Administrators use the FTL server GUI to configure and monitor the eFTL service. For information about these GUI pages, see the documentation set for TIBCO eFTL software.

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 [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

SOME CLOUD SOFTWARE GROUP, INC. (“CLOUD SG”) SOFTWARE AND CLOUD SERVICES EMBED, BUNDLE, OR OTHERWISE INCLUDE OTHER SOFTWARE, INCLUDING OTHER CLOUD SG SOFTWARE (COLLECTIVELY, “INCLUDED SOFTWARE”). USE OF INCLUDED SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED CLOUD SG SOFTWARE AND/OR CLOUD SERVICES. THE INCLUDED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER CLOUD SG SOFTWARE AND/OR CLOUD SERVICES OR FOR ANY OTHER PURPOSE.

USE OF CLOUD SG SOFTWARE AND CLOUD SERVICES IS SUBJECT TO THE TERMS AND CONDITIONS OF AN AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER AGREEMENT WHICH IS DISPLAYED WHEN ACCESSING, DOWNLOADING, OR INSTALLING THE SOFTWARE OR CLOUD SERVICES (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH LICENSE AGREEMENT OR CLICKWRAP END USER AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE SAME TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of Cloud Software Group, Inc.

TIBCO, the TIBCO logo, the TIBCO O logo, FTL, eFTL, and Rendezvous are either registered trademarks or trademarks of Cloud Software Group, Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. You acknowledge that all rights to these third party marks are the exclusive property of their respective owners. Please refer to Cloud SG’s Third Party Trademark Notices (<https://www.cloud.com/legal>) for more information.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: <https://scripts.sil.org/OFL>

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

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.cloud.com/legal>.

Copyright © 2009-2026. Cloud Software Group, Inc. All Rights Reserved.