TIBCO FTL® - Enterprise Edition

Shifting to TIBCO FTL

Version 7.0.1 | December 2024

Contents

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.

Shifting from EMS to FTL

Application developers, architects, and administrators familiar with TIBCO Enterprise Message Service™ can be confused when shifting to TIBCO FTL® because these two messaging products use similar terminology for concepts and features that can sometimes be quite different. This manual maps between the two paradigms while highlighting the differences.

Another point of confusion is that TIBCO FTL supports a message broker paradigm (like EMS), and also supports peer-to-peer messaging (which is unlike EMS).

Shifting from Rendezvous to FTL

Application developers, architects, and administrators familiar with TIBCO Rednezvous® can be confused when shifting to TIBCO FTL® because these two messaging products use different paradigms for addressing. They also assign different meanings to some terminology.

Pay special attention to these topics:

- Content-Based Addressing
- Endpoints, Publishers, Subscribers, and Transports
- FTL Delivery

Another point of confusion is that TIBCO FTL supports peer-to-peer messaging (which is like Rendezvous), and also supports a message broker paradigm (which is like EMS).

Message Form and Content

Messages are simpler in TIBCO FTL than they are in TIBCO EMS.

Differences

- In EMS a message has three parts, its body, header, and properties.

  In FTL a message consists of field and value pairs, which are analgous to the field and values in the body of an EMS map message. An FTL message does not carry headers or properties.

- EMS messages can take several forms, indicated by JMS body types.

  FTL does not distinguish messages by body type.

Message Format

In FTL every message has a format. A *format* is a set of field names and types that characterize the structure of a message.

Simple Use Case

In the simple use case, you can supply any non-null format name when you create a message. Subsequently, you can safely ignore that name. Nonetheless, it is good practice to adhere consistently to one of the following conventions for format names:

- Use only one format name for all messages within a suite of communicating applications.
- Use a small number of format names, and use the same format name for all messages with similar structure (that is, field names and types).

However, if you use the JMS BytesMessage type, consider using a built-in opaque format in FTL; see "Built-In Formats Reference" in TIBCO FTL Development.

Format Optimizations

Adhering to one of those conventions from the start ensures that, later, you can transition easily to a more complex use case, in which FTL uses formats to automatically optimize messages. These optimizations can result in smaller messages and faster lookup of fields. When the need for message optimizations becomes apparent, see the following topics:

- "Formats" in TIBCO FTL Concepts
- "Formats: Managed, Built-In, and Dynamic" in TIBCO FTL Development
- "Formats Grid" in TIBCO FTL Administration
- "Defining Formats" in TIBCO FTL Administration

JMS Message Types

TIBCO FTL uses only one type of message - unlike JMS, which uses several message types. Yet you can use FTL messages to emulate most of the JMS message types.

The structure of an FTL message is analogous to that of a JMS MapMessage. Other JMS message types are either conveniences or optimizations.

To emulate JMS message types, use the FTL equivalents in the following table.

JMS	FTL
Message A message without a body, for event notification.	Include only those fields that are required for content-based addressing.
TextMessage A message containing only a string.	Include one field, with a string value.
MapMessage A set of name/value pairs.	Include all the name/value pairs as fields.
BytesMessage A stream of bytes.	Include one opaque field with the bytes as its value. For optimized performance use either the opaque or keyed opaque built-in formats.

JMS	FTL
StreamMessage	No direct equivalent in TIBCO FTL. Encode within an opaque.
ObjectMessage	No direct equivalent in TIBCO FTL.

Messaging Paradigms

TIBCO FTL provides two messaging paradigms: a message broker paradigm and a peer-to-peer messaging paradigm.

Message Broker

In the *message broker* paradigm, messages flow through a centralized store-and-forward server. This paradigm is familiar to user of TIBCO Enmterprise Message Service (EMS) and other JMS systems.

In TIBCO FTL, you can arrange for the FTL server to fill the role of message broker.

Brokered messages travel in two hops, from the publisher to the message broker, and then from broker to subscribers.

Peer-to-Peer Messaging

In peer-to-peer communication, messages can flow *directly* from a publisher object in one program to subscriber objects in other programs.

Peer-to-peer messages travel immediately from publisher to subscriber, in only one hop. When developing applications that require the lowest latency, use peer-to-peer messaging.

FTL Server and Services

The role of the FTL server in TIBCO FTL software is broader than the role of the TIBCO EMS server, providing several important services.

The FTL server provides a *realm service*, which supplies application programs with configuration data.

The FTL server can provide an optional *persistence service*, which can act as a message broker, or implement various qualities of service for message delivery.

The FTL server can provide an optional *bridge service*, which can extend the range of transports.

The FTL server can provide an optional *eFTL service*, which can intergrate mobile and browser platforms into an enterprise messaging solution.

Message Broker Pattern

You can use the FTL server to implement a message broker, an intermediate store-and-forward service between publishers and subscribers.

The default configuration of an FTL server implements a message broker. That is, when you run one FTL server without supplying any configuration file, it automatically operates as a message broker. Moreover, when you run three FTL servers without supplying any configuration file, they automatically join together to operate as a message broker.

When application programs interact with the message broker, its default behavior uses a non-persistent store and standard durables. This behavior is equivalent, in EMS terminology, to non-persistent messaging through topics. (You can explicitly configure persistent messaging behavior and queue behavior.)

Realm Definition and Realm Service

The realm definition consists of configuration data for client programs. The realm service supplies that configuratin data to client processes, and funnels monitoring data and logging data from client processes.

Realm Definition

The FTL server provides the realm service, which is the central repository for configuration data required by its client processes, which include FTL application programs and services. This configuration data is called the *realm definition*. Supplying this realm definition to clients is the primary role of the realm service.

The realm definition includes configuration data for all the client processes in the realm. In the simplest use case, one FTL server supplies that configuration data to the clients. However, you can also arrange a family of affiliated FTL servers, for example, primary FTL servers on separate computers cooperating for fault tolerance, satellite FTL servers at WAN-connected sites, and disaster-recovery servers. A family of affiliated FTL servers all serve the same realm definition to clients.

Client process require configuration data to *start* running. In the peer-to-peer paradigm, clients can then *continue* communicating with other clients even if they temporarily disconnect from the FTL server. That is, they can communicate *without* using the FTL server as an intermediary. (In the message broker paradigm, in contrast, clients must maintain contact with the FTL server that acts as a message broker.)

Realm Monitoring and Logging

In a secondary role, the FTL server (through its realm service) receives client monitoring data and client logging data, and funnels it to external tools for analysis and display. If clients temporarily disconnect from the FTL server, they buffer their monitoring and logging data until they can reconnect.

Realm Server GUI and API

The FTL server GUI provides a browser-based visual interface to view and modify the realm definition, and to monitor and manage client processes.

The FTL server also provides a REST-style web API. You can use it to create administrative programs to configure, monitor, and manage clients.

Endpoints, Publishers, Subscribers, and Transports

Endpoints and *transports* are usually less familiar concepts than *publishers* and *subscribers* for new users shifting to TIBCO FTL from TIBCO EMS. This topic explains all four of these concepts within a three-layer framework.

Three layers of abstraction support the flow of FTL messages. These concepts are crucial for peer-to-peer messaging, in which messages flow directly from application program to application program, without an intervening store-and-forward server. Nonetheless, these concepts also apply in the message broker paradigm.

- *Endpoints* in the application layer
- *Publishers* and *subscribers* in the program layer
- *Transports* and *transport buses* in the transport layer

Summary Overview

This table briefly summarizes the main ideas. The sections that follow it present the ideas in greater depth.

	Application Layer	Program Layer	Transport Layer
Message Flow	Messages flow *between endpoints*.	Messages flow *frompublishersto subscribers*.	Messages flow *throughtransports*.
Key Concept	Messages enter and exit applications through named endpoints.	Programs create publishers and subscribers on endpoint names. Publisher send	At run time, programs automatically establish *transport buses* that can move messages from publishers to subscribers. Even when messages make an

	Application Layer	Program Layer	Transport Layer
		messages. Subscribers express interest in receiving messages.	intermediary hop through a message broker or persistence service, transport buses do the work of moving messages.
Your Task	Configure endpoints in the applications grid of the FTL server GUI.	Create publishers and subscribers in program code.	Define and configure transports in the transports grid of the FTL server GUI. Bind transports to endpoints in the applications grid of the FTL server GUI.

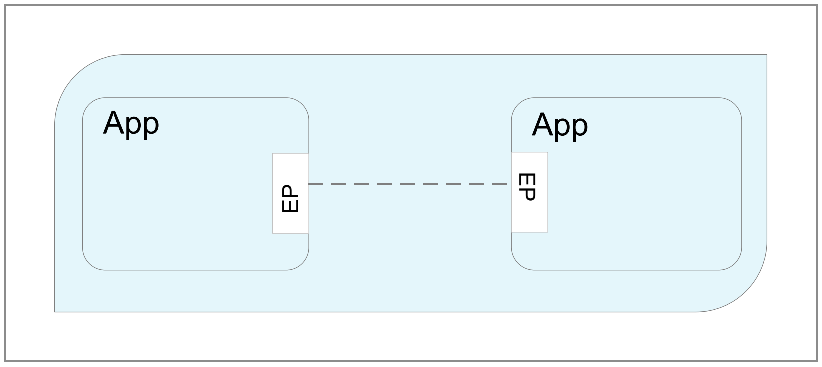
Endpoints

Endpoints are names. An endpoint denotes a place within an FTL program where messages can enter and exit the program.

At the application layer of abstraction, messages flow between endpoints. Messages exit the sending application program at its endpoint, and enter the receiving application program at its endpoint.

The following diagram depicts the endpoints (E).

Figure 1: Messages Flow between Endpoints



You can define applications and their endpoints in the applications grid of the FTL server GUI. Many programs require only a single endpoint, and can use the default application definition and its default endpoint.

Programs use endpoint names only to create publishers and subscribers. Programs can create several publishers and subscribers on a single endpoint.

Publishers and Subscribers

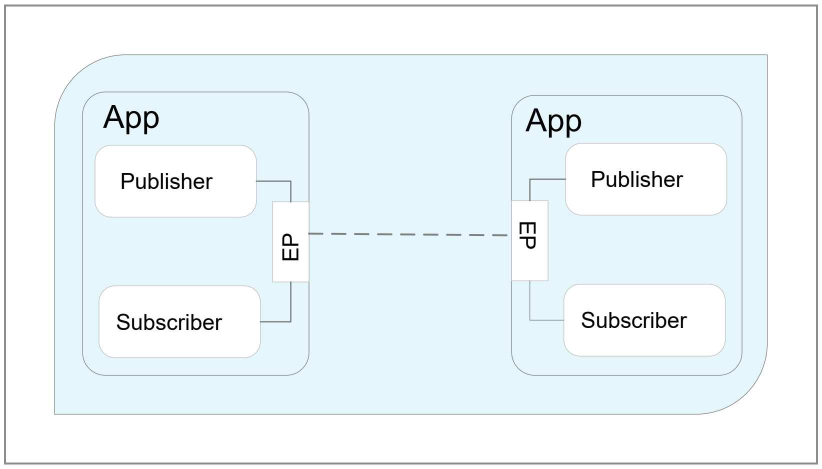
Publishers are objects within FTL programs that send messages. *Subscribers* are objects within FTL programs that express interest in receiving messages.

At the program layer of abstraction, messages flow from publishers to subscribers.

Programs create publishers and subscribers on endpoint names. That is, the calls that create a publisher or a subscriber accept an endpoint name as an argument. When a publisher sends messages, they go out through that endpoint. When a subscriber receives messages, they have entered through that endpoint.

The following diagram depicts the publishers (Pub) and subscribers (Sub) in purple.

Figure 2: Messages Flow from Publishers to Subscribers



The FTL server does not configure publisher and subscriber objects, nor even refer to them. They exist *only* within programs.

Transports

Transports are data pipes connecting endpoints to endpoints. Transports move messages between running programs.

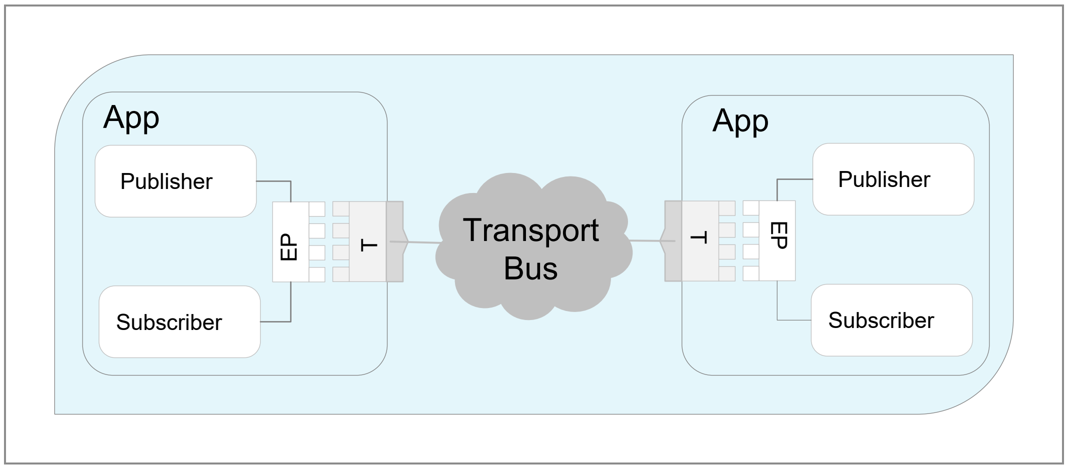
At the transport layer of abstraction, messages flow through transports.

At run time, programs automatically construct *transport buses* using configuration information supplied by the FTL server. Publishers and subscribers automatically use these transport buses to move messages directly from a publisher in one FTL client program to subscribers in other FTL client programs.

In its most simplified definition, a bus consists of a wire and protocol for using it. You can understand a transport bus in this way too. Its wire can include network objects, file objects, special hardware. FTL library code within the program processes implements a variety of protocols.

The following diagram depicts the parts of the transport bus in various orange colors. Notice that the transport bus is conceptually located outside and between the application program processes (the wire). In actuality, the bus also extends to transport objects (T) inside each of the processes (the protocol implementation).

Figure 3: Messages Flow through Transports



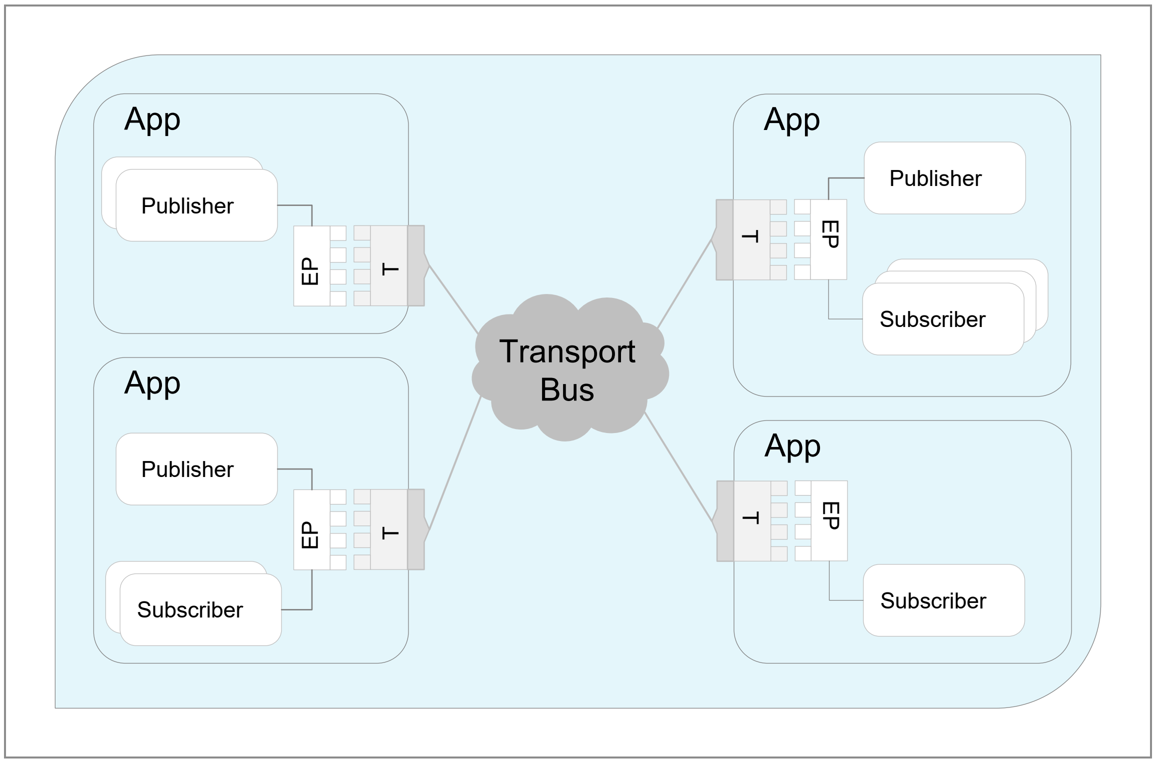
You can define and configure transports in the transports grid of the FTL server GUI. You can bind application endpoints to transport definitions in the applications grid of the FTL server GUI.

Transports are transparent to programs. Program code does not refer to transports. You can modify transport definitions in the FTL server without modifying programs. Modifying the transport definitions changes the way FTL moves messages among programs, but program code is independent of this change in behavior.

Transports are also transparent to programmers. Programmers cannot access transport objects, which are internal to the TIBCO FTL library. Programmers do not need to concern themselves with the details of transport definitions. The default transport definition is sufficient for many programs. During program development, you can run programs using the default definition. During deployment, testing, and production, application administrators can modify network behavior by substituting different transport definitions.

For simplicity, the preceding diagrams depict only two application programs, with one publisher and one subscriber in each program. However, FTL supports any number of communicating processes with any number of publisher and subscriber objects, as the following diagram depicts.

Figure 4: Generalized Messaging



FTL Delivery

TIBCO EMS and TIBCO FTL use different mechanisms to receive and process inbound messages in client programs. Understanding the differences can ease your shift to FTL.

FTL programs do these steps to receive messages:

1. Connect to the FTL server to obtain configuration information (the realm definition).
2. Create a *subscriber* object to express interest in messages.

   The subscriber uses the configuration information to automatically construct a *transport bus* that can transport messages from publishers (*not* from the realm server).
3. Create an *event queue* object to buffer inbound messages.
4. Add the subscriber to the event queue, with a callback method to process inbound messages.
5. Start a *dispatch loop* to remove messages from the event queue and present them to the callback method.

Differences

Ensure that you understand these crucial differences between the two messaging paradigms:

- The FTL server (through its realm service) supplies configuration information to application programs.
- In peer-to-peer messaging, messages do *not* flow through the FTL server. The FTL server neither stores messages from publishers, nor forwards messages to subscribers.

  In the message broker pattern, messages *do* flow through the FTL server.
- A *transport bus* is the physical mechanism that delivers messages among peers, or between client and broker. For example, a transport bus delivers a stream of messages directly from a publishing program to a subscribing program. The FTL server supplies the configuration of the transport bus, and the peers use that configuration to construct the bus.

Similarly, a transport bus delivers a stream of messages from a publishing program to the message broker, and another bus delivers a stream of messages from the broker to a subscriber.

EMS has no analogue for transport buses. You can imagine a transport bus as similar to a TCP connection from a publisher object to a subscriber object, though transport buses can also be more complex than these examples.

- Transport buses are automatic. Programs create subscriber or publisher objects, and those objects construct their transport buses without further explicit program calls. The FTL API does not refer to transports in any way; transports are completely hidden from program code.
- Messages arrive from a transport bus, and the subscriber buffers them in an *event queue*.

  You *cannot infer anything* about FTL event queues from your understanding of EMS destination queues. Despite the similarity of their names (that is, they are both called "queues"), they serve different purposes. FTL event queues are objects within client programs, *not* within a server.
- Adding an FTL subscriber object to an event queue associates the subscriber, the event queue, and a callback method. When a message arrives, the event queue stores it as a *message event*, which is a package that includes all the arguments that the callback method needs to process the message.
- To invoke the callback method, the receiving FTL program must *explicitly* dispatch message events from the event queue. Programs usually start a dispatch loop in a thread dedicated for this purpose.
- The event queue separates message arrival from message processing. Arrival and processing usually occur asynchronously, and in different threads. (The exception to this rule is a latency optimization called *inline mode*.)
- EMS offers either synchronous receive or asynchronous delivery.

  FTL offers only asynchronous delivery, as described in this topic.

Subscribers and Message Interest

The roles of a subscriber object in TIBCO FTL are different from the roles of a subscriber in TIBCO EMS. FTL subscriber objects express interest in a stream of messages, and act as the receiving end of a transport bus, and enqueues inbound messages for processing. However, unlike EMS subscribers, FTL subscribers do not actually process the information in messages.

Express Interest

FTL programs express interest in a stream of messages by creating a subscriber object on an endpoint. The create subscriber call accepts a *content matcher* argument, which specifies that message interest. After the subscriber object establishes its transport bus, the subscriber and the bus use the matcher to filter the stream of inbound messages. The resulting effect is that the subscriber object receives a stream of messages with content that matches the content matcher.

A subscriber with a null content matcher expresses interest in all messages published on the transport bus, without any filtering.

Establish a Transport Bus

Before any inbound messages can flow, a subscriber object must first establish a transport bus to carry messages. This low-level role can include protocol interactions and constructing internal machinery, all of which is invisible to application programs. Configuration data from the FTL server guides the subscriber in this role.

After the transport bus begins operating, the subscriber object repeatedly cycles through the following three message delivery phases.

Phase 1: Receive

A subscriber object receives a stream of inbound messages from its transports. This low-level role can include I/O operations and reassembling data into messages, all of which is transparent to application programs.

Phase 2: Match and Filter

A subscriber object uses its content matcher locally to test the content of each inbound message.

Notice that the program supplied the content matcher when it created the subscriber, and FTL software uses it repeatedly to automatically filter the message stream.

Phase 3: Distribute

A subscriber object distributes the filtered stream of inbound messages, that is, it places each message on an event queue.

This action concludes the subscriber's delivery responsibilities with respect to an individual message. Phases four, five, and six of message delivery are the responsibility of the application program. (See the topics that follow.)

The subscriber's role resumes with the receive phase, when another message arrives through the transport bus.

For precise definitions of all six phases of delivery, see "Inbound Message Delivery" in TIBCO FTL Development.

Key Points for Developers

- A program creates a subscriber object and supplies it with a content matcher.
- The content matcher specifies the subscriber's message interest, which determines the sub-stream of messages that the subscriber object receives.

Differences

- The roles of a *subscriber* are not entirely parallel between FTL and EMS. You cannot infer the behavior of one from the other.
- EMS programs can use either subscribers or consumers to receive messages.

  FTL programs use only subscribers. (Consumers do not exist as a separate concept in FTL.)
- EMS and FTL use the word *interest* to denote two separate concepts.

  In FTL, every subscriber expresses message interest.

  In EMS, the concept of interest is associated with routers and bridges.
- The filtering role of FTL content matchers is similar to that of EMS message selectors. However, their behavior is not similar. For more details, see Content-Based Addressing.

Event Queue and Message Dispatch

After a subscriber distributes a message to an event queue, the message is ready for processing. But processing does not occur until after the program explicitly dispatches the message from the event queue. Event queues have no analogue in TIBCO EMS.

Event Queue as a Separation

An event queue is like a transfer station for messages.

- A subscriber object shepherds a message through the first three phases of delivery, from the transport to the event queue.
- Thereafter, the application program shepherds the program through the last three phases of delivery, dispatching it from the event queue, processing it, and releasing it.

Between these two shepherds, a message makes a brief stop on an event queue.

A program that creates subscriber objects must also create at lest one event queue. The program must add each subscriber to an event queue.

Message Events

More precisely, a *message event* makes a brief stop on an event queue. A message event is a package that includes a message, the appropriate callback method to process the message, and all the arguments that the callback method requires. (This level precision is not important for developers. It is more common to say that the event queue contains messages, even though it actually contains message events.)

Dispatch

The dispatch call removes a message event from an event queue, and invokes the callback method on the message.

Application programs arrange a *dispatch thread* running a loop that dispatches messages.

It is good practice to dispatch each event queue using at least one dispatch thread. You can even dispatch a queue using several concurrent threads, unless concurrent processing of messages would yield incorrect application semantics.

Key Points for Developers

- A program creates an event queue.
- A program associates each subscriber with one event queue and one message processing callback.
- A program dispatches message events from the event queue for processing.
- Use a dispatch loop to repeatedly dispatch messages.
- You may start several dispatch loops in separate threads to take advantage of multi-core processors.

Callback and Message Processing

Programs process inbound messages using callback methods. Each dispatch call invokes a callback method. The callback method runs within the dispatch thread.

Callback Method

A program that creates a subscriber must define a callback method to process its messages. You can define a separate callback for each subscriber, or reuse a callback for several subscribers.

Adding a subscriber to an event queue also associates the subscriber with a callback method, which processes all the messages that arrive through that subscriber.

A callback method receives an array of messages to process. The array might contain only one message, or several messages.

Processing usually involves getting values from the fields of a message, and doing some action (for example, storing values or sending another message).

It is good practice to code callbacks to return promptly.

In a straightforward scenario, in which processing is quick, developers code the callback method to process the messages and return after completing the message processing.

In a complex scenario, in which processing could involve delays, developers can code the callback method to transfer processing to another thread. Processing continues in the other thread, even as the callback method returns and the dispatch thread dispatches the next message event.

These two scenarios imply different behavior in the final phase, release.

Processing

The processing phase begins when the dispatch call invokes the callback method on a message.

In either of the two scenarios, all processing remains completely within the control of application code.

Release

Release is the final phase of the delivery cycle. The inbound message is no longer needed, and can be destroyed to reclaim resources.

In the straightforward scenario, FTL library code owns the message, and automatically destroys the message after the callback returns.

In the complex scenario, a subscriber can release all of its messages to the callback. Application program code must explicitly destroy each such message after processing completes.

Key Points for Developers

- You can code as many different callback methods as you need.
- Every message is processed using only one callback method. You associate the callback with the subscriber when you add the subscriber to an event queue.
- Code callback methods to return promptly.
- A callback runs within a dispatch thread, within your program's control flow.
- Message processing can continue in other threads, also within your program's control flow.
- Message release can be automatic or explicit.

Content-Based Addressing

TIBCO FTL features *content-based addressing*, that is, subscribers select messages by their content. A *content matcher* specifies a set of fields. A subscriber receives only those messages with fields that match that specification. (Although the idea of filtering might seem familiar from TIBCO EMS message selectors, you *cannot* infer the behavior of FTL content matchers from the behavior of EMS message selectors.)

A null content matcher matches all messages. So a subscriber without a content matcher recieves all messages that arrive on its transport bus.

A subscriber can narrow the stream of messages it receives by specifying a set of field names and values. A content matcher can test any field name for its presence or absence in messages. Alternatively, a content matcher can test values within fields, but only for exact matches of string and numeric values.

Example	Description
`{"alert":true}`	Match all messages that contain an alert field.
`{"maint_routine":false}`	Match all messages that do *not* pertain to routine maintenance.
`{"attn":"Jenny Pericles"}`	Match all messages that require the attention of Jenny Pericles.
`{"reorder":true,"reord_qty":false}`	Match all messages that indicate it's time to reorder, but do *not* specify the quantity to reorder.
`{"bug":true,"model":"421758"}`	Match all messages that contain bug reports against product model 421758.

Differences

- EMS message selectors filter messages based on the values of message headers and properties.

FTL content matchers filter messages based on the presence, absence, or values of message fields, that is, based on the *content* of the message.

- EMS message selectors specify messages using subset of SQL92 conditional expression syntax.

  FTL content matchers specify messages using JSON attribute/value pairs, combined with logical AND.

See Also

For complete details, see "Content Matchers" in TIBCO FTL Development, and its sub-topics.

A brief video presents the concepts of content-based addressing and content matchers. See FTL Content-Based Addressing: Rethinking Destinations.

Request/Reply Interactions

Request/reply interactions differ in several details. TIBCO FTL and TIBCO EMS use different objects and methods to achieve similar results.

Paradigm

The following steps distinguish code for request/reply interactions in FTL applications:

1. In the responding program, create a subscriber to receive request messages, and a publisher to send replies.
2. In the requesting program, create a publisher to send requests, and an *inbox subscriber* to receive the reply.

   Get the inbox object from within the inbox subscriber.
3. When composing request messages, include the inbox object as a field value.
4. Send the request message using the *send* method.
5. In the responding program's callback method, extract the inbox object from the request message.

   Send the reply message directly to that inbox using the *send to inbox* method.

For more information, see "Request and Reply" in TIBCO FTL Development.

For example code, see the sample programs `tibrequest` and `tibreply`.

Differences

- In place of an EMS message requestor, an FTL program uses an ordinary publisher to send requests, and an inbox subscriber to receive replies.

  That is, FTL uses separate objects, one to send the request message and another to subscribe to the reply message. In contrast, EMS consolidates them into one requestor object.
- In EMS the request method of the message requestor object is *synchronous*, in that it waits for a reply.

In FTL, replies arrive *asynchronously*, like any other inbound message.

- In EMS the `Reply To` header of the request message indicates the reply address.

  FTL messages do not contain header fields. The inbox object serves as the reply address.

  The requesting program explicitly embeds the inbox in a field of the request message (you choose the name of that field). The responding program extracts it and sends the reply to that inbox.

- In EMS you can link a reply message to its request using the correlation ID header.

  FTL messages do not contain header fields. You can explictly link a reply to a request with a unique ID token by including the token in a message field (you choose the name of that field). Use the token in the callback method of your requesting program to explictly correlate each reply with its request.

Persistence and Durability: Contrast FTL and EMS

TIBCO FTL and TIBCO EMS both include the concepts of message persistence and durable subscribers. However, the two products use similar terms for concepts that are not exactly parallel. To avoid confusion, understand the following terms and distinctions.

The FTL server provides a store-and-forward intermediary called the *persistence service*. Within it, objects called *durables* can subscribe to messages and store them. Different types of durables enable a variety of quality of service behaviors.

Persistence

In EMS, each message can be either *persistent*, *non-persistent*, or *reliable*. That is, the delivery mode is a property of the *message*. The EMS server writes persistent messages to disk.

In FTL, the term *persistent* does not apply to messages. Rather, *persistence* is the potential to store messages between the publishers that send them and the subscribers that receive them.

Persistence services are FTL services that can store messages. To use FTL persistence you must explicitly configure one or more FTL servers to provide persistence services.

Message Storage and Replication

The EMS server implements persistence by writing messages to disk.

The FTL persistence service does *not* write messages to disk. Instead, it replicates messages across a cluster of persistence services, each of which keeps messages in process memory. This strategy can reduce message latency.

Durable

In EMS, a subscription on a topic can be *durable* or *non-durable*. (A subscription to a queue is implicitly durable.) That is, durablility is a property of the *subscription*. The EMS server continues to store messages for a durable subscriber, even after the subscriber object has closed. When a client process recreates the durable subscriber, that subscriber can continue consuming the messages that the EMS server has stored in the interim.

In FTL, a *durable* is an object that combines three actions:

- It subscribes to messages, expressing message interest on behalf of *durable subscribers*, even when those subscribers are disconnected or closed.
- It can track acknowledgements that indicate that durable subscribers have received messages.
- It can send or resend messages to durable subscribers.

Durables are objects within a persistence service. You can define durables using the stores grid of the FTL server GUI.

Message Flow

In EMS, *all* messages flow through the EMS server, a message broker that can also provide message persistence.

In contrast, FTL software can also implement a peer-to-peer communication network, in which messages flow directly from publishing programs to subscribing programs. Messages need *not* flow through the FTL server. However, when persistence behaviors are required, messages do flow through an FTL server's persistence service, which stores messages from publishers and forwards messages to subscribers. The FTL server provides the persistence service, which is a client of the FTL server.

The durable type determines the message flow.

Durable Types: Standard, Shared, and Last-Value

Compare the delivery behavior and message flow of the different FTL durable types.

- In the absence of a durable, an FTL subscriber receives a message only when a transport bus connects it to the publisher. The message travels directly from publisher to subscriber.
- **Standard durables** implement delivery assurance.

  The message travels directly from publisher to subscriber, but also from publisher to durable (within a persistence service). The durable retains the message until the the subscriber acknowledges it. If the subscriber misses a message while temporarily disconnected from the transport bus, it can retrieve the missed message from the durable.

Delivery assurance with a standard durable is similiar to an EMS use case with persistent messages sent to a topic, and durable subscriptions on that topic.

- **Shared durables** implement apportioning of messages.

  The message travels from publisher to durable, and then from the durable to a subscriber. The durable retains each message until one subscriber acknowledges it.

  Apportioning messages with a shared durable is similar to an EMS use case with messages sent to a queue, and several subscriptions on that queue. However, an EMS queue distributes messages to subscribers in a round robin, while an FTL shared durable distributes messages using a load-sharing scheme that is not strictly round robin.

- **Last-value durables** implement last-value behavior with a key field.

  The message travels from publisher to durable, and then from the durable to subscribers. The durable uses the value in the key field to divide the messages into sub-streams, one for each key value. The durable stores only the most recent message for each key. A new subscriber specifies a key, and the durable forwards the last message with that key, and subsequent messages with that key (until the subscriber closes).

  Last-value behavior is not analogous to any EMS use case.

- **Map durables** implement key/value maps.

  Map durables are a specialized use of last-value durables.

  In addition to last-value durable semantics, programs can use the FTL map API to get and set the current value corresponding to a specific key.

Static and Dynamic Durables

In TIBCO FTL, you can use the stores grid in the FTL server GUI to define durables in two ways: static and dynamic.

Note: Dynamic durables are simpler to use than static durables.

- You can define a *dynamic durable template*, which enables durable subscribers within programs to create a set of durables at run time. Alternatively, you can select one of the built-in dynamic durable templates.
- You can define a *static durable* in advance of any subscribers.

You can configure both kinds of durable definitions using the stores grid in the FTL server GUI.

Durables and Content Matchers

A durable subscribes to messages, and can use a content matcher to express interest in a sub-stream of messages.

For dynamic durables, you can supply a content matcher (optional) as an argument to the create subscriber method.

For static durables, you can supply a content matcher (optional) in a durable definition. Use the stores grid in the FTL server GUI.

Arranging Persistence

Persistence in an FTL network requires the following actions.

These general steps outline the different tasks. For more detailed task instructions, see "Configuring Persistence" in TIBCO FTL Administration, and its sub-tasks.

Procedure

1. Configure definitions for a persistence cluster and its persistence services. Use the clusters grid of the FTL server GUI.
2. Configure definitions for a persistence store and its durables. Use the stores grid of the FTL server GUI.
3. Configure FTL servers to provide persistence services. Use the FTL server configuration file.
4. Start the persistence services.

   Start or restart the FTL servers. The FTL server reads the updated FTL server configuration file, and automatically starts the appropriate persistence services.

5. Create durable subscribers within application programs.

Arranging Message Brokers

Message brokers in an FTL network require the following actions.

Procedure

1. Configure FTL servers to provide persistence services. Use the FTL server configuration file.

   The default configuration file for the FTL server provisions a single, standalone message broker. If you need only one broker, you may skip this step.

   If you need multiple brokers, copy and modify the default configuration file as needed.

2. Start the brokers.

   Start the FTL server or servers, which automatically provide the configured message broker functionality.

3. Create durable subscribers within application programs.

FTL Can Emulate EMS Behaviors

You can use TIBCO FTL to emulate familiar TIBCO EMS behaviors.

- **EMS Topic** FTL standard durables behave similarly to EMS topics with durable subscriptions.
- **EMS Queue** FTL shared durables behave similarly to EMS queues.

  However, an EMS queue distributes messages to subscribers in a round robin, while an FTL shared durable distributes messages using a load-sharing scheme that is not strictly round robin.
- **Wildcard Subscriptions** Embed the destination name elements in a set of message fields. Subscribe using a content matcher to select messages with those fields. Omit wildcard elements from the content matcher.
- **Message Broker** The FTL server can behave as a message broker, emulating the EMS server.

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

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.