TIBCO FTL® - Enterprise Edition

Concepts

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.

FTL Introduction and Components

TIBCO FTL® is a messaging infrastructure middleware product. It features high speed and reliable structured data message exchange. There are clearly defined roles for application developers and application administrators. TIBCO FTL has the flexibility to achieve low transfer latency, high delivery reliability, or a customized balance of these.

To get started working in FTL, see the Getting Started activities.

The main components of TIBCO FTL are:

- **FTL Server**: The FTL server is a multi-purpose executable component that consolidates several services into one server process. These services include:
    - **Realm Service**: Used as a unified communication fabric within and across networks which supplies application programs with communication configuration data.
    - **Transport Bridge Service**: Used to forward messages among multiple transport types.
    - **Persistence Service**: Used to manage stores and durable subscriptions (durables).
    - **Group Service**: Used to coordinate operating roles among application processes and fault-tolerant operation to ensure continuated operation when one or more components fail.
    - **Authentication Service**: Used to authenticate the identity of, then authorize, client programs, administrative tools, and affiliated servers, typically as a prerequisite to allowing access or communication.
    - **eFTL Service**: Used to integrate mobile and browser platforms into an enterprise messaging solution.
- **Application Programming Interface (API)**: Developers use the API to build applications that interface with TIBCO FTL.
- **User Interface**: Administrators will find the FTL server's graphical user interface (GUI) useful for monitoring the state of FTL services and clients.

These tutorial lessons are meant for experienced application developers, using Linux operating systems. The goal is to provides a full picture of TIBCO FTL software and its capabilities. The lessons begin with a simplified view of TIBCO FTL® messaging, and adds details and features with requirements they satisfy or the problems they solve.

The terms we define along the way are basic concepts, and not necessarily complete functional descriptions. For details about the concepts, see TIBCO FTL Product Documentation, *TIBCO FTL Development* and *TIBCO FTL Administration*.

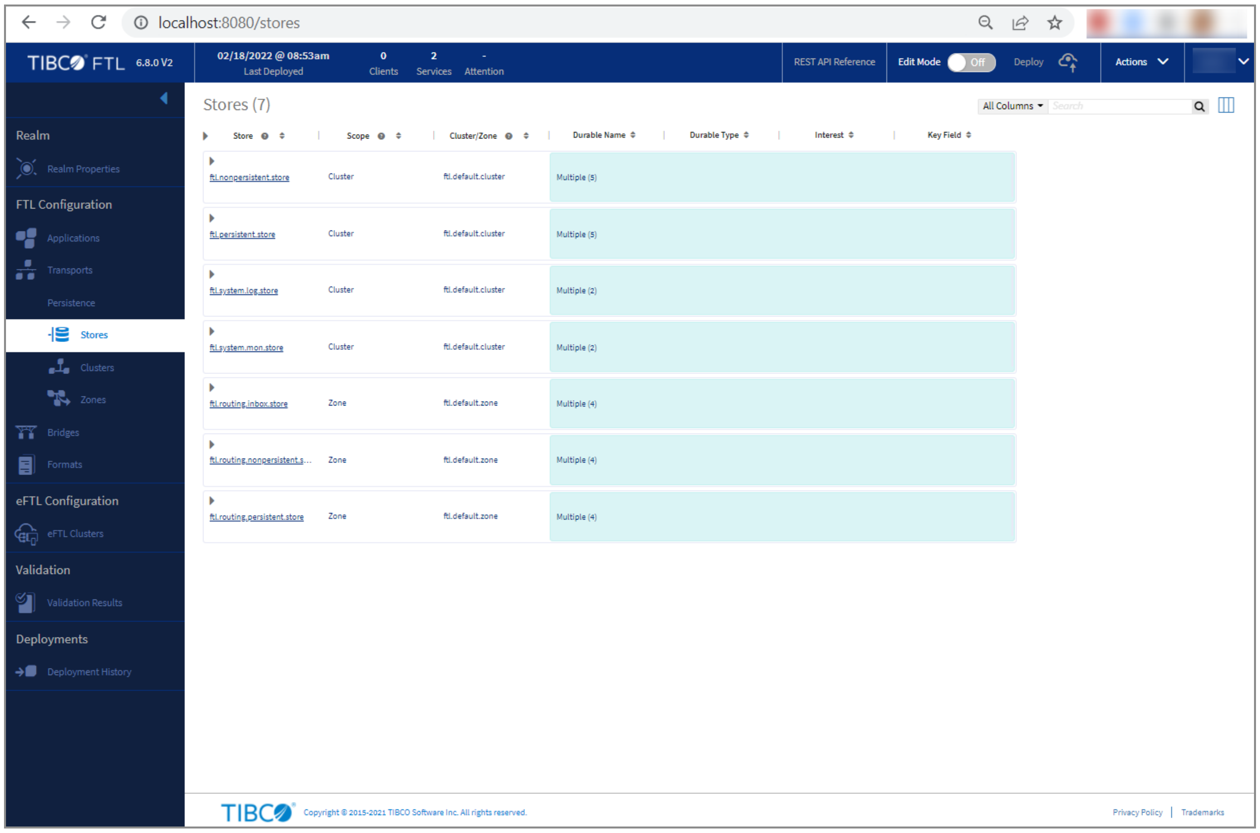
FTL Graphical User Interface (GUI)

Administrators will find the FTL server's GUI useful.

- The top bar of the GUI displays the state of FTL services and clients at a glance. It also includes the **REST API Reference** in Swagger, **Edit Mode** (**On** or **Off**), **Deploy** if **Edit Mode** is **On**, **Actions**, and **Sign Out**.
- Status tables summarize the current state of FTL clients and services.
- The left column menu of the GUI is a table of contents into the realm definition and its history. Each item in this menu accesses a page where you can view and modify an aspect of the realm.
- A grid summarizes information about a category of objects in the realm. Rows present objects as rectangles. Columns present details of an object.

For more information, see the TIBCO FTL documentation, Administration guide.

Figure 1: FTL GUI



localhost:8080/stores

TIBCO FTL 6.8.0 V2 | 02/18/2022 @ 08:53am Last Deployed | 0 Clients | 2 Services | - Attention | REST API Reference | Edit Mode Off | Deploy | Actions

Realm
- Realm Properties

FTL Configuration
- Applications
- Transports
- Persistence
  - Stores
  - Clusters
  - Zones
- Bridges
- Formats

eFTL Configuration
- eFTL Clusters

Validation
- Validation Results

Deployments
- Deployment History

Stores (7)

Store	Scope	Cluster/Zone	Durable Name	Durable Type	Interest	Key Field
ftl.nonpersistent.store	Cluster	ftl.default.cluster	Multiple (5)			
ftl.persistent.store	Cluster	ftl.default.cluster	Multiple (5)			
ftl.system.log.store	Cluster	ftl.default.cluster	Multiple (2)			
ftl.system.mon.store	Cluster	ftl.default.cluster	Multiple (2)			
ftl.routing.inbox.store	Zone	ftl.default.zone	Multiple (4)			
ftl.routing.nonpersistent.s...	Zone	ftl.default.zone	Multiple (4)			
ftl.routing.persistent.store	Zone	ftl.default.zone	Multiple (4)			

Copyright © 2015-2021 TIBCO Software Inc. All rights reserved. Privacy Policy | Trademarks

Messaging Overview

In TIBCO FTL, *Publishers* send messages and *subscribers* receive messages. A program can send messages, receive messages, or both. The terms *publisher* and *subscriber* refer to TIBCO FTL objects that perform these two roles within programs. Administrators can configure user permissions to publish, subscribe, or allow other functions. Entitlements can be configured for destinations to prevent users from exchanging sensitive data.

A *message stream* is a sequence of messages. A publisher is the source of the message stream, and a subscriber expresses interest in that stream and receives messages.

A message only carries data. A message *format* defines the metadata, which is the set of fields that a message can contain. The format is available to cooperating programs before they exchange messages. This approach reduces the message size, transport bandwidth, and processing time. TIBCO FTL messages do not carry destination information.

Content Matchers

Subscriber objects can specify interest in messages based on their *content* made up of message fields and their values. A *content matcher* is a pattern-matching template for message fields and values and is used to select a subset of messages from a message stream. A subscriber can have one content matcher to *express interest* in the subset of messages that the content matcher specifies.

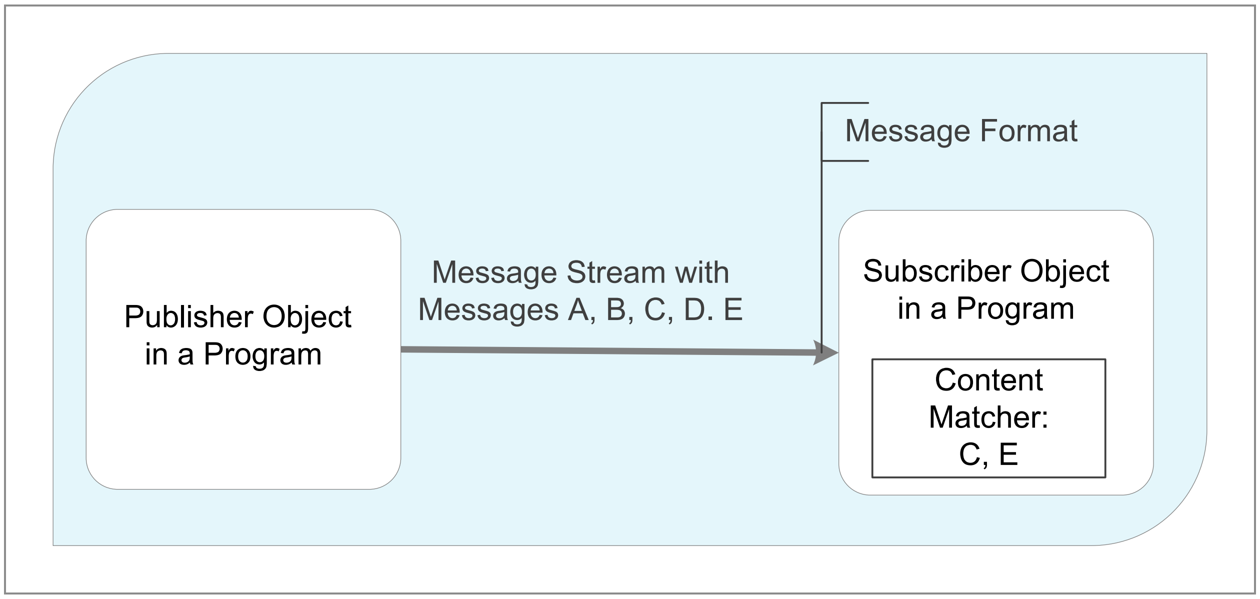
Messages can match on all values (an AND match) or on one or more of the values (an OR match). For example, a subscriber could receive only messages in which the `Symbol` field contains the value `TIBX` and `FOO`. Or, a subscriber could receive only messages in which the `Symbol` field contains the value `TIBX` and `FOO`.

A subscriber without a content matcher does not filter out any messages.

Example

In the following diagram, a program's publisher object transmits a message stream with messages A, B, C, D, and E. The subscriber object has expressed interested in the message stream and receives the stream. The subscriber uses the message format to interpret the contents of the messages. The subscriber's content matcher is only interested in messages C and E, so only messages C and E are consumed by the subscriber.

Figure 2: Message from Publisher to Subscriber


Message Format
Publisher Object in a Program
Message Stream with Messages A, B, C, D. E
Subscriber Object in a Program
Content Matcher: C, E

Message Formats

Programmers determine the message format. Administrators create the format definitions which are used by programs at run time.

For example, in an order fulfillment program, a product request message could contain the following:

- A product name field with a string type and a value
- A warehouse bin field with an integer type and a value
- A quantity field with an integer type and a value

A program sending a message with the format definition can include values for any subset of the fields in the format definition. Receiving programs already have the format, including field names and field types, to unpack data values from messages correctly.

TIBCO FTL software does not limit the number of formats that a program can use for sending or receiving messages. Nonetheless, in practice, many programs exchange only a few types of messages.

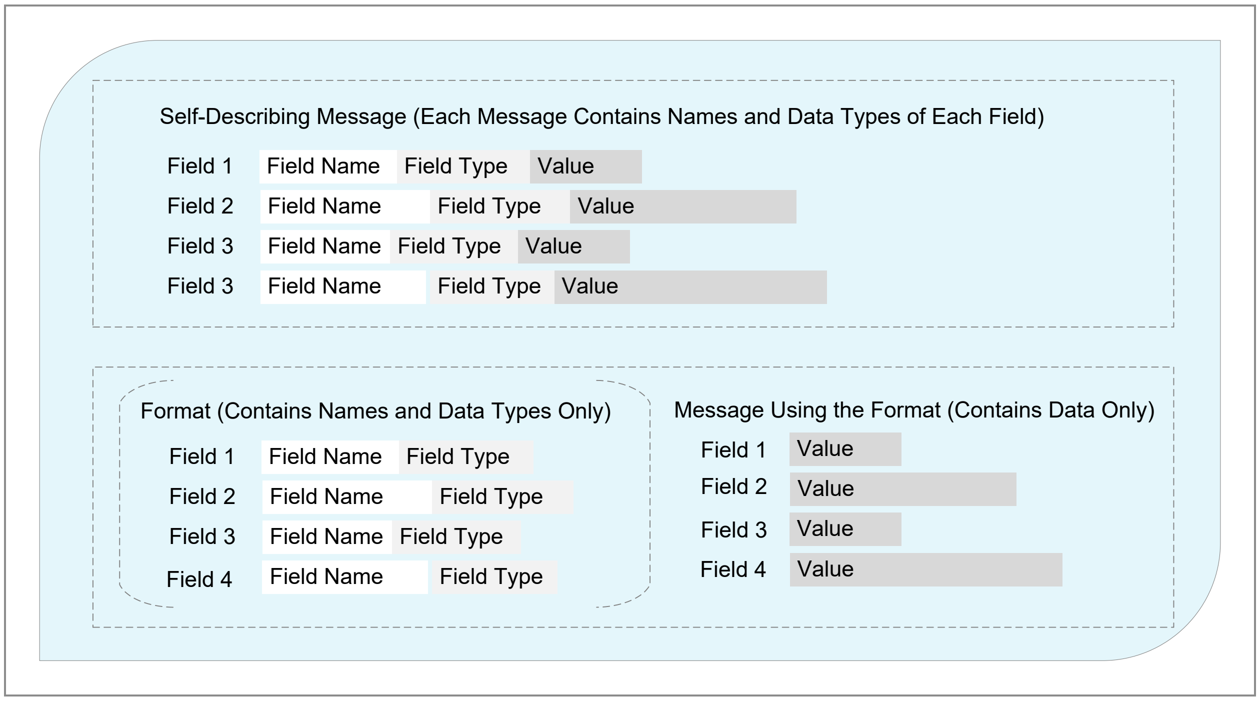
Benefit of Formatting versus Self Describing Message

The benefit of FTL's formatted messages versus self describing messages is shown in the following diagram.

- Self describing message: A publishing program assembles a message with any number of fields and any combination of the data types. Each message contains the name and data types of each field which enlarges each message and results in slower processing of messages.
- Message using a format: In FTL, the format is the set of fields that a message can contain, including field names and data types. The format is available to cooperating programs before they exchange messages. Messages only carry data, which reduces both the message size, transport bandwidth, and processing time.

In the following diagram, compare the complexity and length of one self-describing message at the top against the message with only data on the bottom right of the diagram.

Figure 3: Messages: Data and Metadata



Transports, Endpoints, and Brokers

The messaging stream is delivered using transports, endpoints, and brokers.

- **Transport**: Publishers send message streams to subscribers over a transport, which is a shared communication medium. TIBCO FTL supports a variety of transport types to use based on messaging requirements and environment. See Transport Types.
- **Broker**: A messaging broker is a intermediate *store* which temporarily holds messages. An FTL server can act as a broker.
- **Endpoint**: An endpoint is the connection point between a transport and the publisher, subscriber, or broker. An endpoint contains transport details. This insulates programs designers from those details and separates responsibilities between developers and administrators.
  Multiple publishers and many program processes may publish on one publishing endpoint. One subscriber may consume from more than one publishing endpoint. A program can have more than one endpoint to tap into more than one transport message stream. Messages are merged and neither the publisher nor the publishing program are identified to the subscriber. For details, see Message Merging.

Broker and Brokerless Messaging

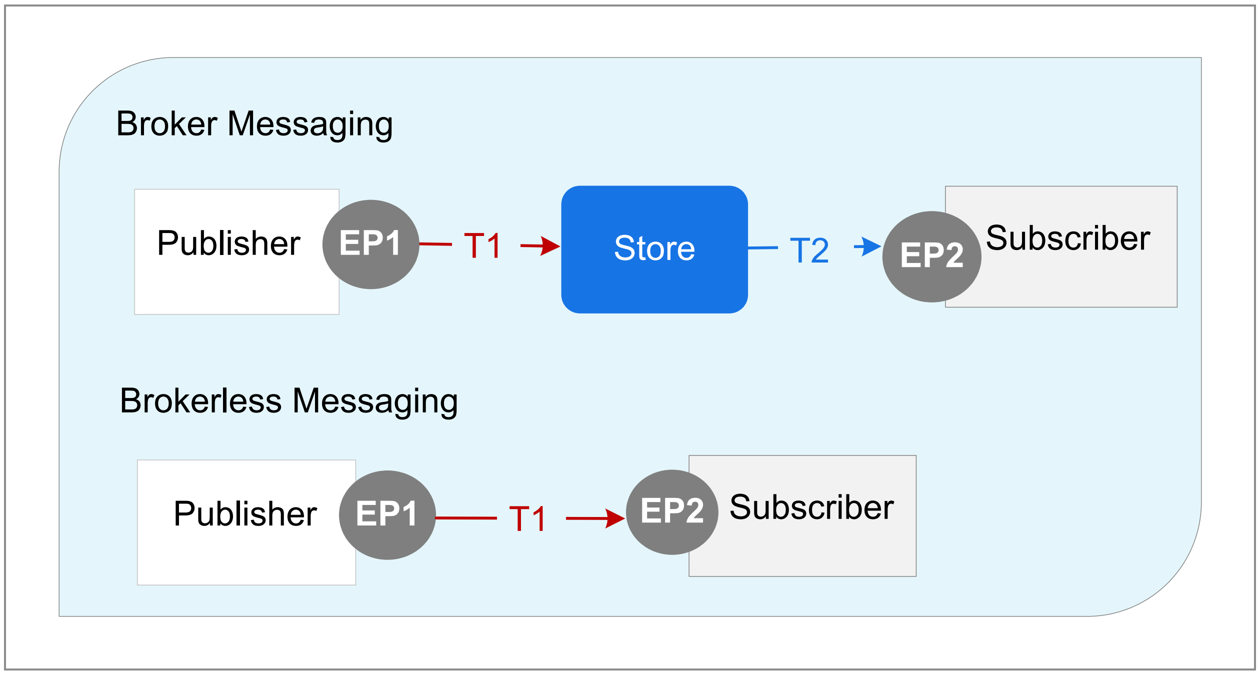
FTL may operate with *broker messaging* or *brokerless messaging*.

- **Broker Messaging**: In broker messaging, publishers send messages to the FTL server, and messages are held in a *store* until they are successfully sent to subscribers. This model allows for adjustment of reliability and throughput parameters, and provides flexible administration and monitoring.
- **Brokerless Messaging**: In brokerless messaging, publishers send messages directly to subscribers, with generally lower latency than when using a broker for storage. The FTL server still provides some management for the client programs.

Important: *Persistence* is when an odd number of stores are used in a stream to provide temporary message storage. Persistence is covered in more depth in the section Persistence: Stores and Durables.

In the following diagram, broker messaging flows from the publisher's endpoint (EP1) on transport T1 through a store to transport T2 arriving at the subscriber's endpoint (EP2). Brokerless messaging is transported from the publisher's endpoint (EP1) over transport T1 directly to the subscriber's endpoint (EP2).

Figure 4: Components of Broker Messaging and Brokerless Messaging



Transport Types

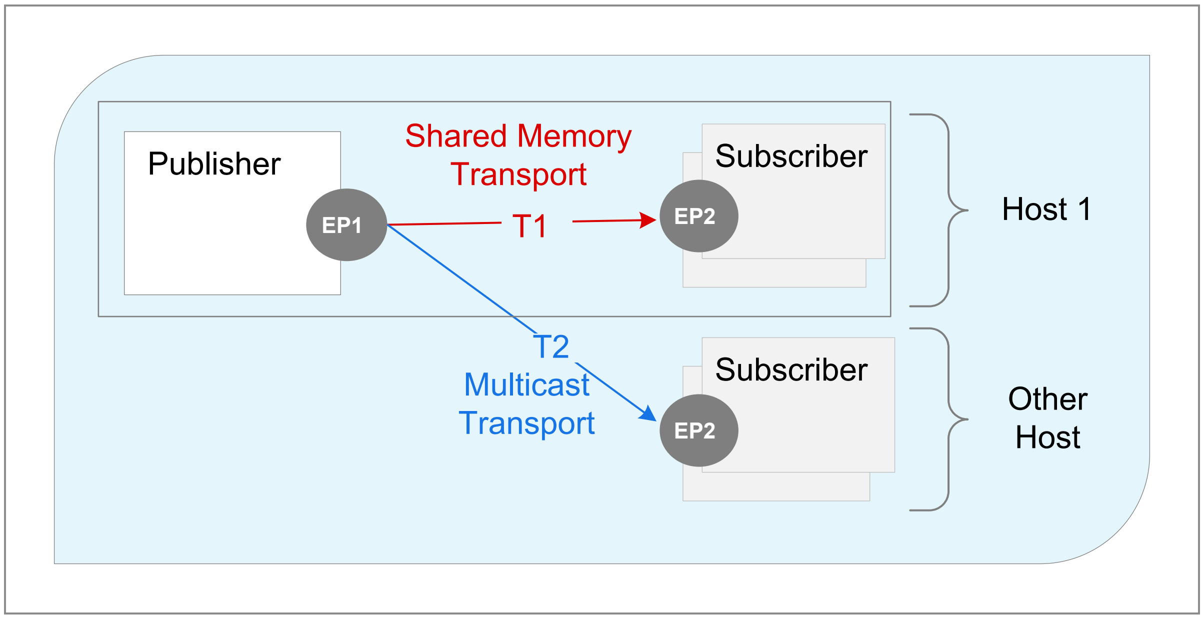
Each transport is a separate communication medium, insulated against crosstalk from other transports.

Using TIBCO FTL, you can use a combination of the following transport types to meet your organization's communication needs.

- **Dynamic TCP (DTCP)**: TCP-based connectivity without the need to specify explicit host and port configurations
- **Static TCP**: TCP-based connectivity using specific host and port configurations
- **Auto**: TCP connectivity to the FTL Server using only a transport name
- **Multicast**: UDP-based message delivery supporting potentially very large numbers of subscribers (requires multicast support to be enabled within the network)
- **Process**: High-speed connectivity within a single process
- **Shared Memory**: High-speed connectivity between programs running on the same host computer
- **Direct Shared Memory**: High-speed communication between programs running on same host computer (highest performance available with some API constraints)
- **Reliable UDP (RUDP)**: UDP-based datagrams connectivity (reliability of TCP with more user control and configuration options)

In the following diagram, a publisher and subscribers that are co-located on the same host use a shared memory transport (T1). The subscriber using a multicast transport (T2) is located on a separate host, local area network (LAN), or wide area network (WAN). The publisher sends a message once and TIBCO FTL transfers the message over both transports. The two transports carry identical message streams and the endpoints (EP1) have the same transport details.

Figure 5: Two Types of Transports



Message Merging

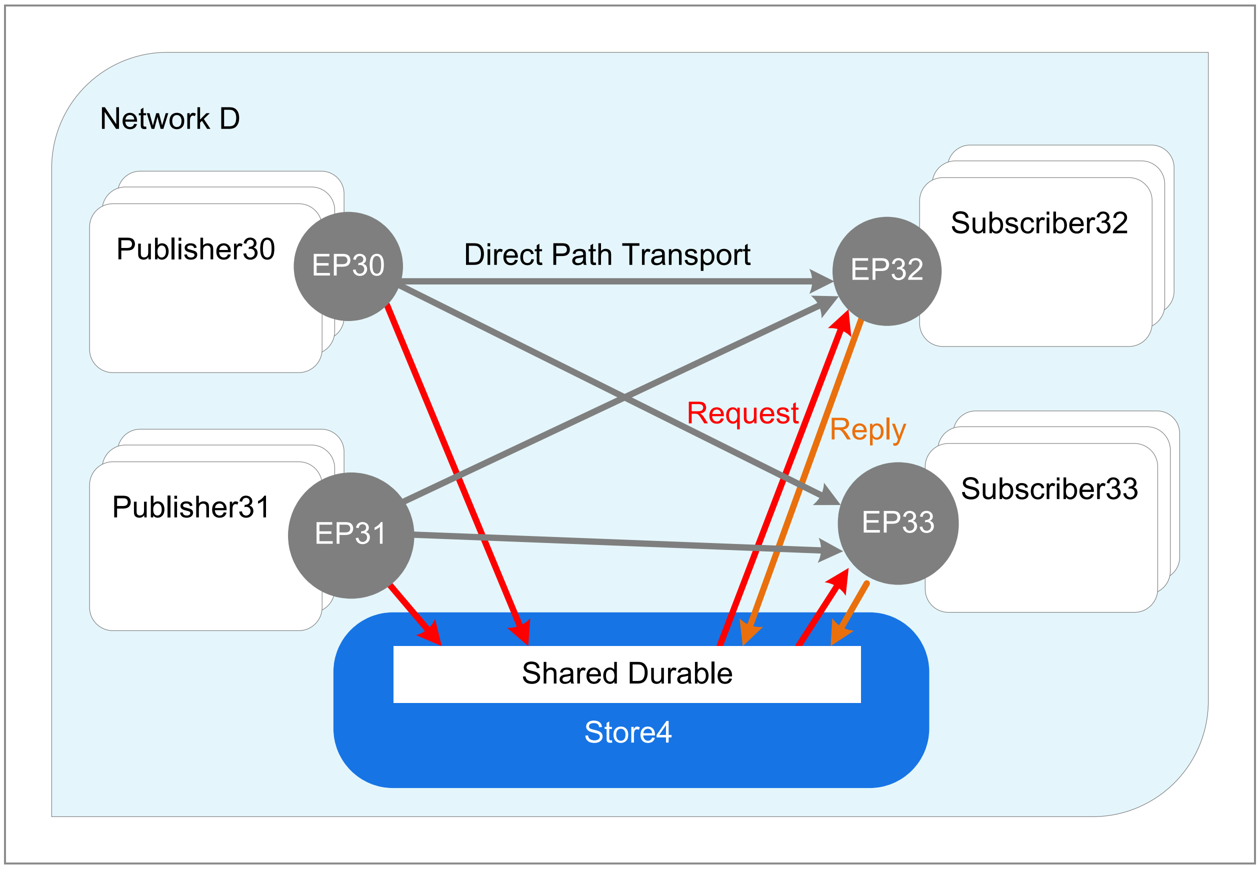
Many publishers and many program processes may publish on one endpoint and the messages are merged. Neither the publisher or the publisher program is identified to the subscriber.

Direct and Indirect Paths Example

In the following diagram, messages from all publishers are merged and are made available to all subscribers.

- Direct path: Network D uses of direct path transports (gray arrows) that connect every publishing endpoint to every subscribing endpoint. Each direct path transport merges message streams. Many publishers and many program processes may publish on Endpoint30. Messages received by the subscribers do not include identifying information for the publisher or the publishing program.
- Indirect path: Store4 merges the message streams from Endpoint30 and Endpoint31. Store4 backs messages from all publishers and makes them available to all subscribers. Store4 is an intermediate hop, part of several parallel, indirect paths in Network D. When a message eventually arrives at a subscriber, the message does not include identification of the publisher or the publishing program.

Figure 6: Network of Endpoints and Direct Paths



Publish-Subscribe Models

Publish-subscribe models allow distribution of information from publishers to interested subscribers in an efficient manner using appropriate transport types.

- **One-to-Many**: This is a common model in TIBCO FTL where a publisher produces a stream of messages and the subscribers can receive messages, sometimes selected by content matching.
- **Many-to-Many or One**: Two or more publishers publish messages and one or more subscribers receive messages.
- **One-to-One**: A publisher sends a message to a specific inbox subscriber and only that inbox subscriber receives the message.

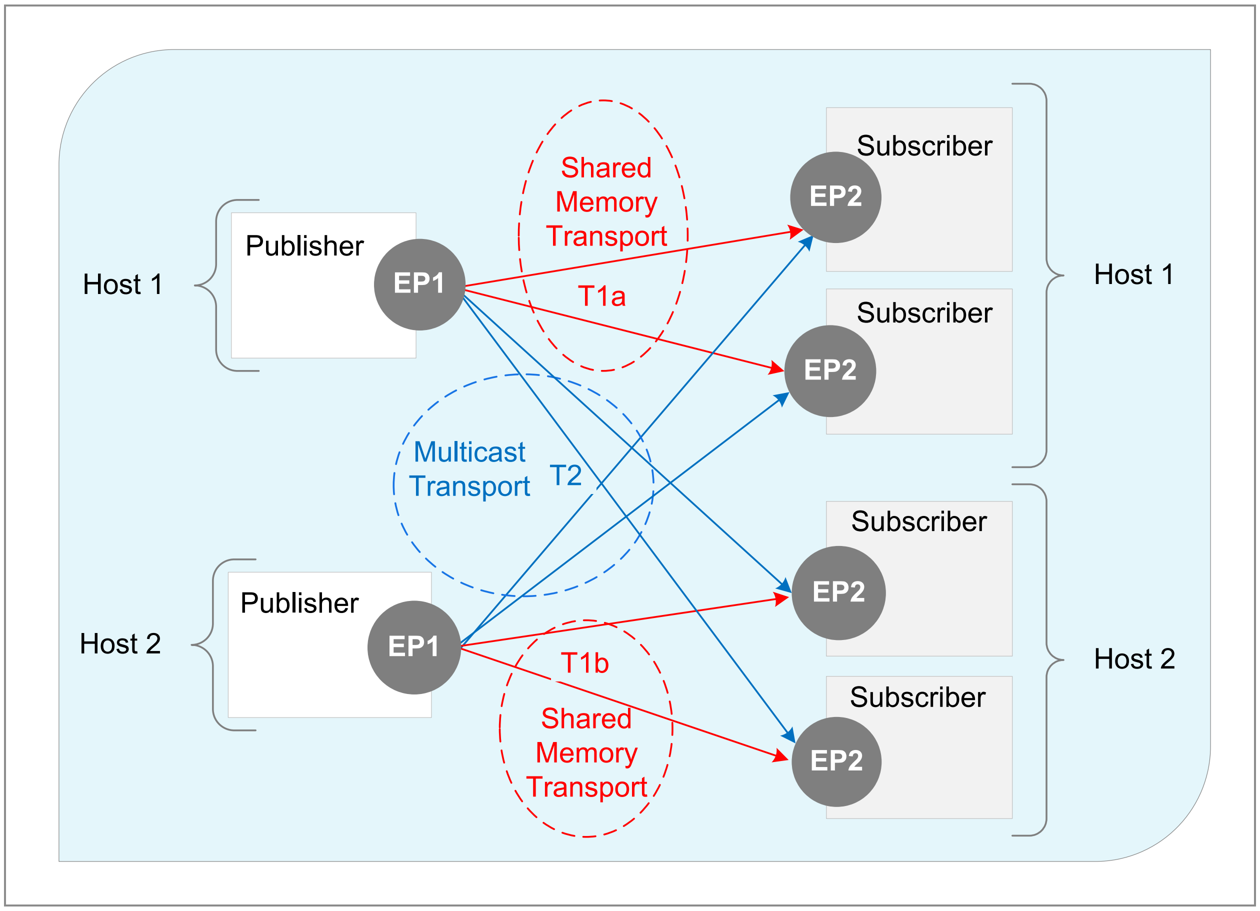
The following sections describe publish-subscribe models in more detail including examples to illustrate how transport types are handled.

One Publisher to Many Subscribers

In *one-to-many* communication, the message stream from one publishing endpoint can go to subscriber endpoints that require different transport types.

In the following diagram, a publisher and two subscribers are co-located on the same host using shared memory transports (T1). The subscribers located on a separate host use multicast transports (T2). The publisher sends a message once and TIBCO FTL transfers the message over all transports to the endpoints.

Figure 7: One Publisher to Many Subscribers



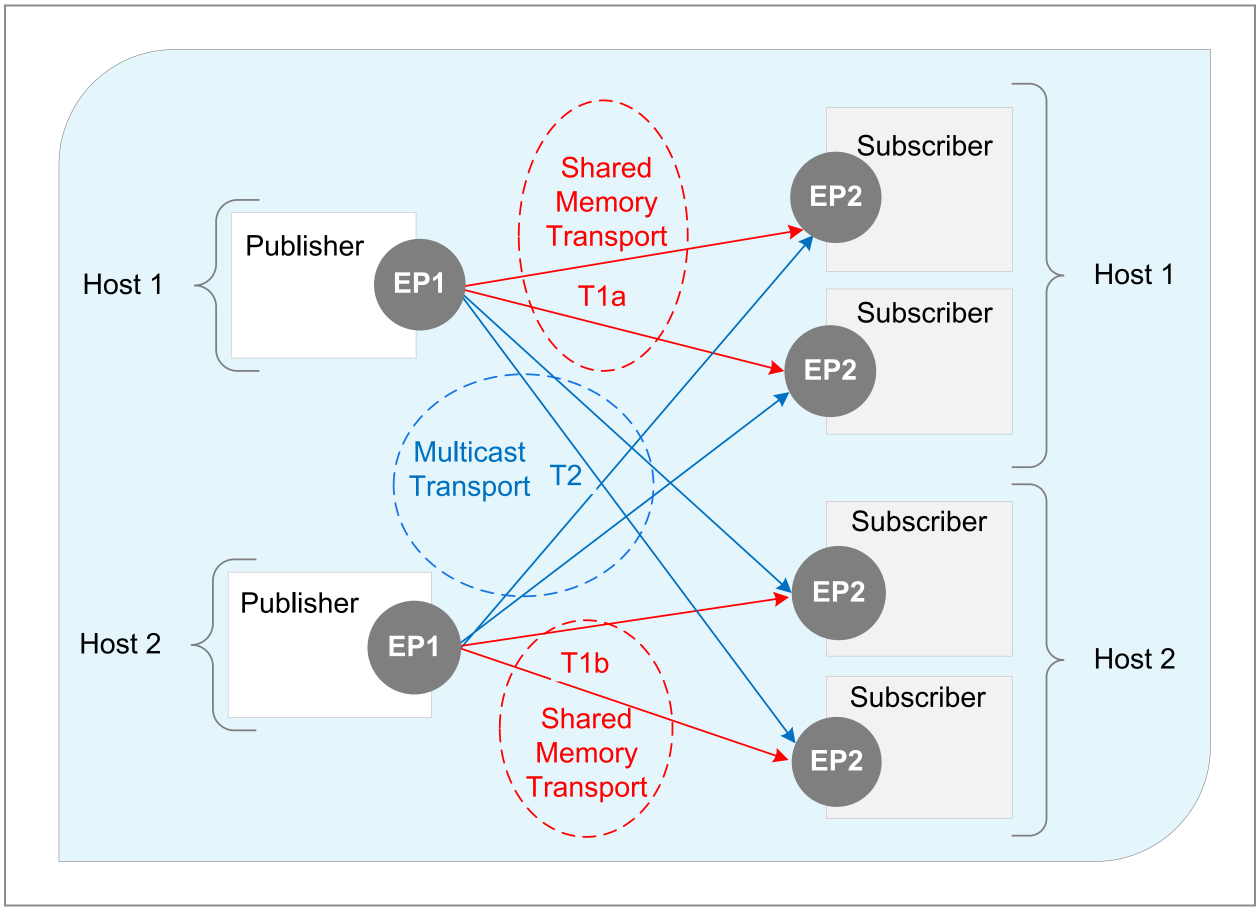
Many Publishers to Many Subscribers

In *many-to-many*, many publishers send messages to many subscribers and, potentially, every subscriber can receives messages from each of the publishers.

This example expands on the previous example by adding another publisher. In the following diagram, every subscriber receives messages from each of the publishers. As shown earlier, programs on the same host use shared memory transports and programs on separate hosts use multicast transports.

Although an increased number of transports is required, the programmer's responsibility is the same as in the previous example, which is to implement EP1 and EP2. The administrator deploys the additional publishing program on Host 2, but the transport configuration remains the same. TIBCO FTL uses the configuration of the shared memory transport (T1) to automatically arrange two shared memory transports: T1a on Host 1 and T1b on Host 2. Similarly, TIBCO FTL software reuses the configuration of the multicast transport (T2) and integrates the new publisher into the existing transport.

Figure 8: Adding Another Publisher

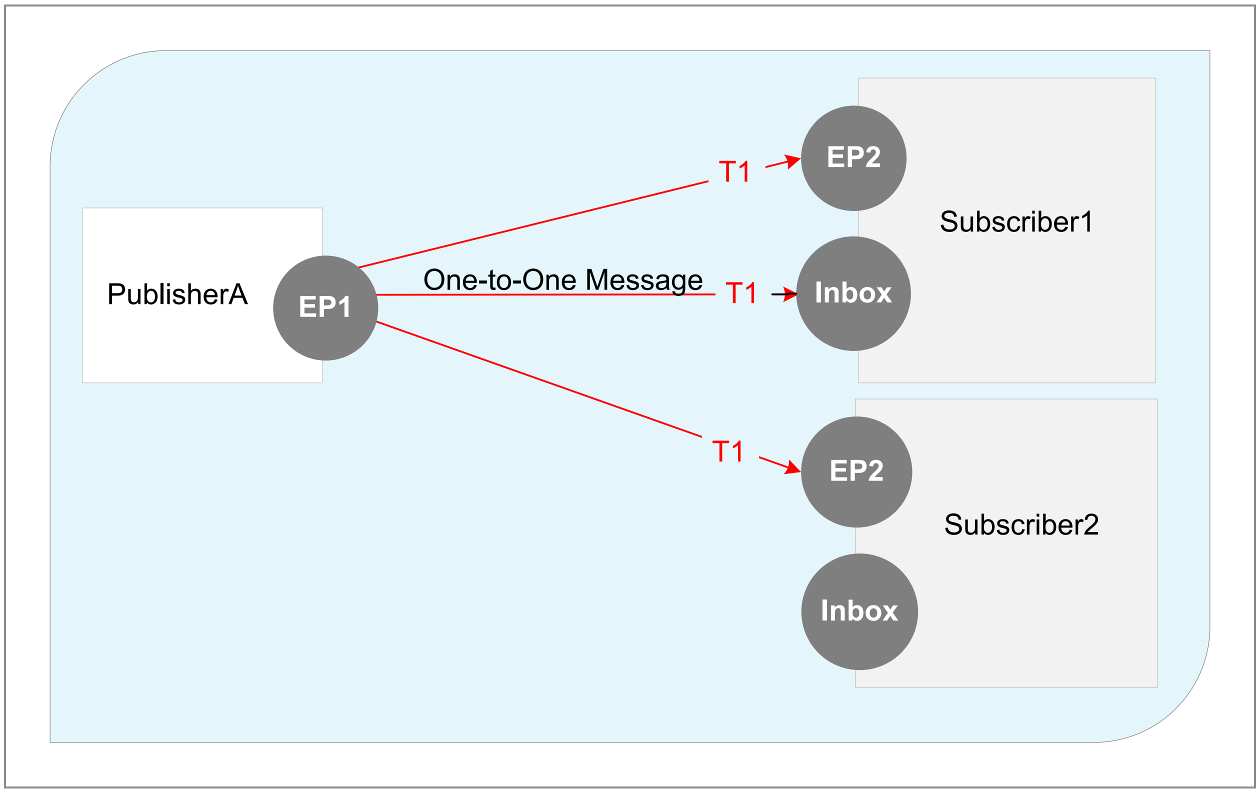


One-to-One Communication

One-to-one communication lets a publisher send a message to a specific *inbox subscriber* which is a kind of subscriber endpoint instance. Only that inbox subscriber receives the message, even when other subscribers share the transport.

In the following diagram, the two subscribers have *inboxes subscribers* within each program. PublisherA sends the same message stream over transport T1 to Subscriber1 and Subscriber2. PublisherA also sends a one-to-one message to Subscriber1's Inbox using the same T1 transport. Only Subscriber1 receives the message.

Figure 9: One-to-One Communication, Sending to an Inbox



Request/Reply with Inbox Token

In the *request/reply* communication model, cooperating programs use a token identifying an inbox for one-to-one communication to the inbox. The program that creates the inbox also creates the token and transmits the token, as needed, to request a message be sent to the inbox.

If inboxes are used with entitlements and permissions, requests must have publish and subscriber permissions. Replies require publish permission.

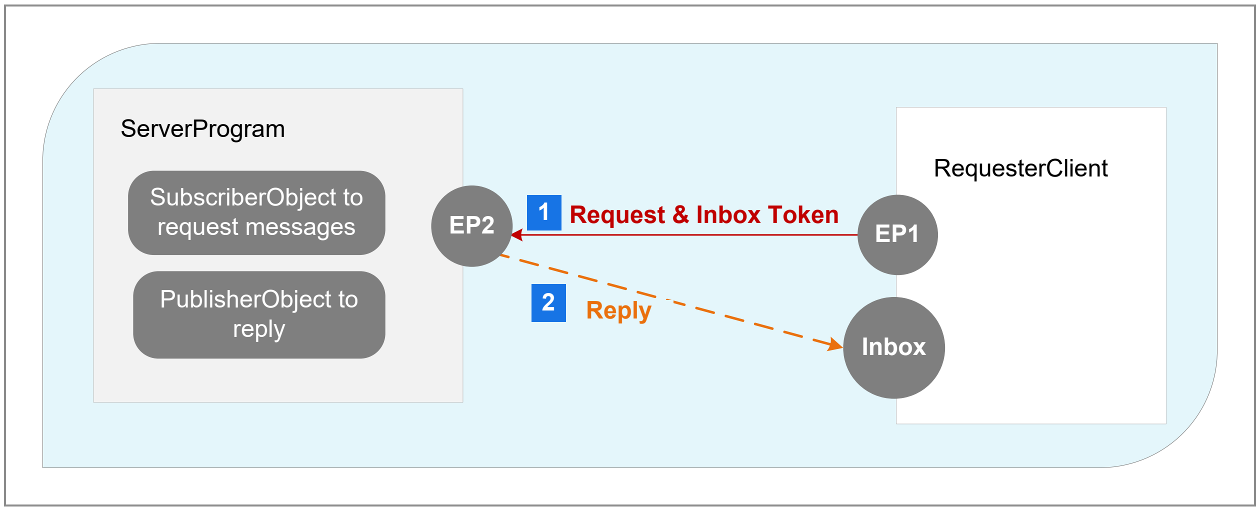
The following diagram illustrates a request/reply interaction.

- The ServerProgram creates a SubscriberObject to request messages from clients and a PublisherObject to reply.
- The RequesterClient program creates an Inbox subscriber object to receive replies from the ServerProgram.

The message flow follows.

1. The RequesterClient initiates the request from EP1 and includes the request and inbox token.
2. The ServerProgram uses the token to send a one-to-one reply message to the Inbox at the RequesterClient.

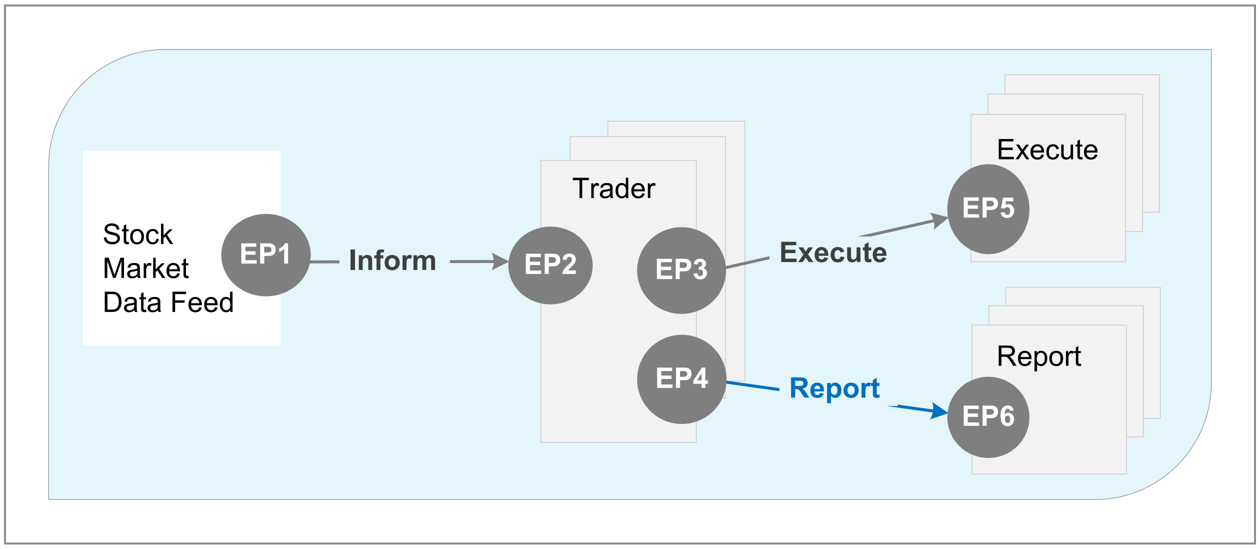
Figure 10: Request and Token for Inbox Communication



Distinct Endpoints in One Program

A program can have more than one endpoint to tap into more than one transport message stream. In the following diagram, a Stock Market Data Feed sends a message stream, Inform, that consists of many small messages, produced at high speed, containing time-sensitive data such as buy offers, sell offers, and executed trades. It is crucial that the Trader program receives these messages with minimal latency and acts without delay.

Figure 11: Different Messaging Requirements



As shown in the diagram, the Trader program produces two message streams:

- The Execute stream is time-sensitive for active trading.
- The Report stream is *not* time-sensitive and messages are used to satisfy regulatory requirements.

It is critical that the Report stream not slow the Execute stream.

The architecture of the Trade program reflects these requirements with three separate endpoints:

- EP2 for the inbound Inform stream
- EP3 for the outbound Execute stream
- EP4 for the outbound Report stream

The administrator deploys the programs and arranges transports to satisfy the performance requirements. The Inform and Execute streams require minimal latency, while the Report stream can use a slower transport technology.

Persistence: Stores and Durables

Persistence is when an odd number of stores are used in a stream to provide temporary message storage. Persistence allows applications to store a stream of messages even while the subscribers are off line, there by allowing subscribers to consume stored messages at a later time.

Persistence Stores

A *store* holds messages sent by a publisher until they are consumed by a subscriber. A *persistent store* is a store that is replicated to further ensure against loss of messages. Administrators configure stores within an FTL persistence service and tailor various aspects of persistence to meet the needs of applications.

Each store collects and maintains a message stream and can collect the stream from one or more publishers. The store can maintain that message stream for one or many subscribers.

A durable with a store expresses interest that endures even when no subscriber object is present, maintaining persistent interest for the benefit of subscribers.

Stores can be replicated and back each other. Non-replicated stores provide some level of persistence in cases where subscribers are temporarily unreachable.

A store can *apportion* a message stream among a set of cooperating subscribers where only one subscriber receives and processes each message. Acting together the subscribers process every message in the stream.

In a persistence store, a program uses a *key/value map* to stores key/value pairs where each key is a string and a key's value is a message. Programs can use map methods to store and retrieve key/value pairs, and to iterate over the pairs in a map. This is a way to use the store as a simple database table.

Durables

A *durable* is the data structure within a store that maintains subscriber interest, forwards messages, and tracks delivery acknowledgments.

There are different types of durables.

- **Standard Durable**: A *standard durable* represents the interest of *one* subscriber object. It forwards messages to the subscriber on request. The type of persistence for a standard durable depends on whether *prefetch* is enabled or disabled:
    - **Prefetch Enabled**: The store functions as a message broker, with no messages traveling peer-to-peer from publisher to subscriber.
    - **Prefetch Disabled**: Messages typically flow directly from publisher to subscriber and the store serves as a backup.
      For details, see Prefetch: Enabled Versus Disabled.
- **Shared Durable**: A *shared durable* represents the identical interest of *one or more* cooperating subscriber objects. It apportions a message stream among those subscribers, forwarding each message to only one available subscriber, and ensures that exactly one of those subscribers acknowledges each message in the message stream. That subscriber can re-forward the message to a different subscriber, if necessary. For details, see Standard Durable versus Shared Durable.
- **Last-Value Durable**: A *last-value* durable preserves only the most recent message for subscribers. It does not track message acknowledgments from subscribers. A last value durable divides its input message stream into output sub-streams based on a key field in each message and retains only one message for each output sub-stream. Programs can subscribe to individual sub-streams. Each new subscriber receives the most recent stored message and continues to receive the sub-stream of subsequent messages.
- **Dynamic Durable**: A *dynamic durable* uses a *dynamic durable template* used by subscribers to the durable to create a set of durables at run time. Programmers take responsibility for the number of durables and their names.
- **Static Durable:** A *static durable* is defined in advance of any subscribers and is defined in the realm configuration. Administrators control the number of durables and their names.
- **Wide-Area Durable**: A *wide-area durable* is part of a persistent store that is available to publishers and subscribers across different networks. For details, see Wide-Area Forwarding.

Examples

The following messaging examples highlight different aspects of stores, durables, endpoints, transports, and message streams.

Prefetch: Enabled Versus Disabled

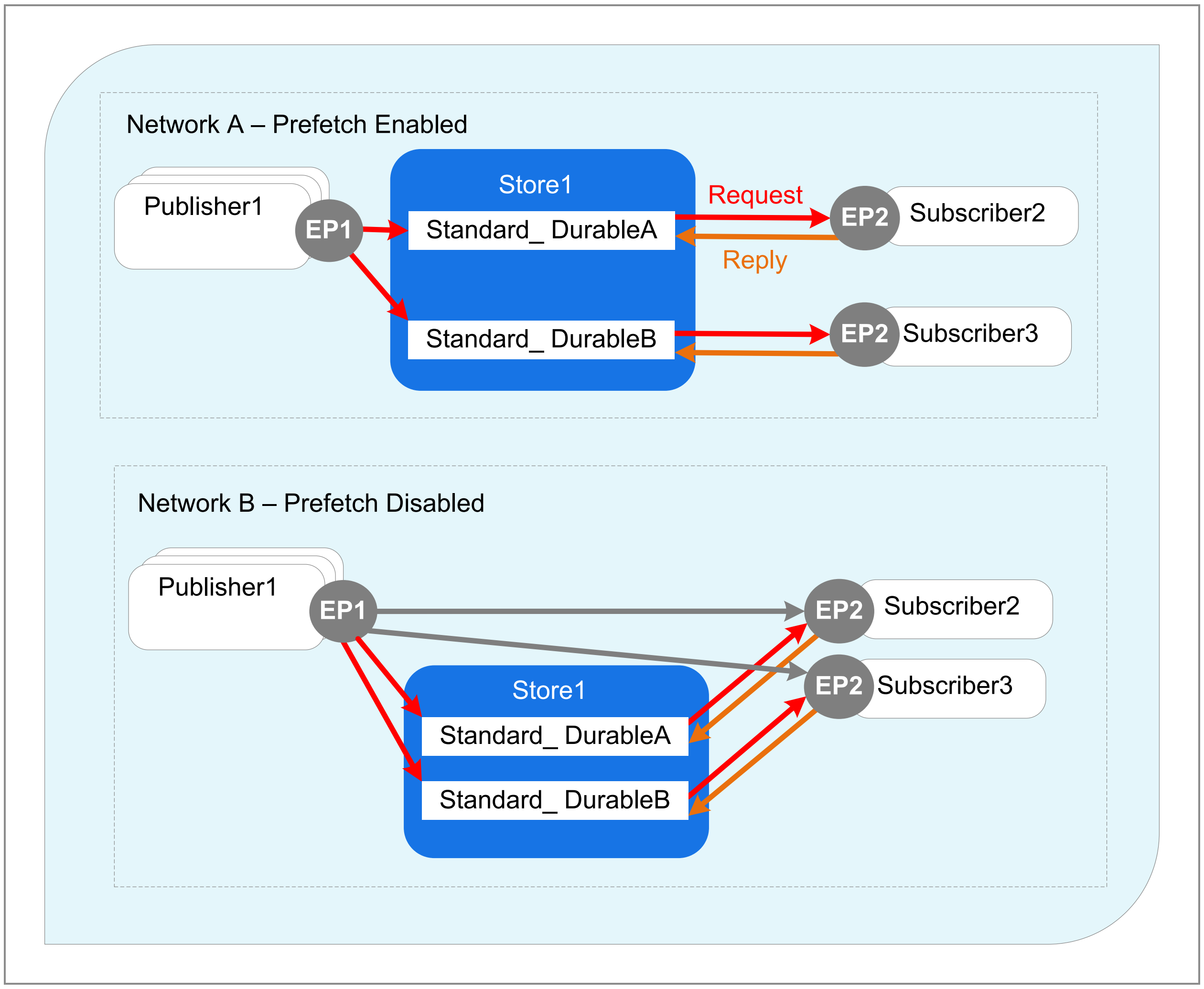
As covered earlier, a standard durable represents the interest of one subscriber object and forwards messages to the subscriber on request. The type of persistence for a standard durable depends on whether *prefetch* is enabled or disabled.

The following diagram illustrates two networks, Network A and Network B. In both networks, each publisher represents potentially many program processes, all of which publish on the same endpoint. Each subscriber represents potentially many program processes. There is one durable per subscriber. The store maintains interest in the message stream even when subscriber is absent.

In Network A, prefetch is enabled and the persistence store functions as a message broker, with no messages traveling from the publisher endpoint to the subscriber endpoint.

In Network B, prefetch is disabled and messages typically flow via a direct path transport from the publisher endpoint to the subscriber endpoint. The direct path transport is backed by the store that receives the message stream from the publisher endpoint and retains each message until the subscriber acknowledges (replies to) the message. The store retains unacknowledged messages if the subscriber is absent or the direct path fails and then retransmits the messages to the subscriber, as needed.

Figure 12: Prefetch Enabled and Prefetch Disabled



Standard Durable versus Shared Durable

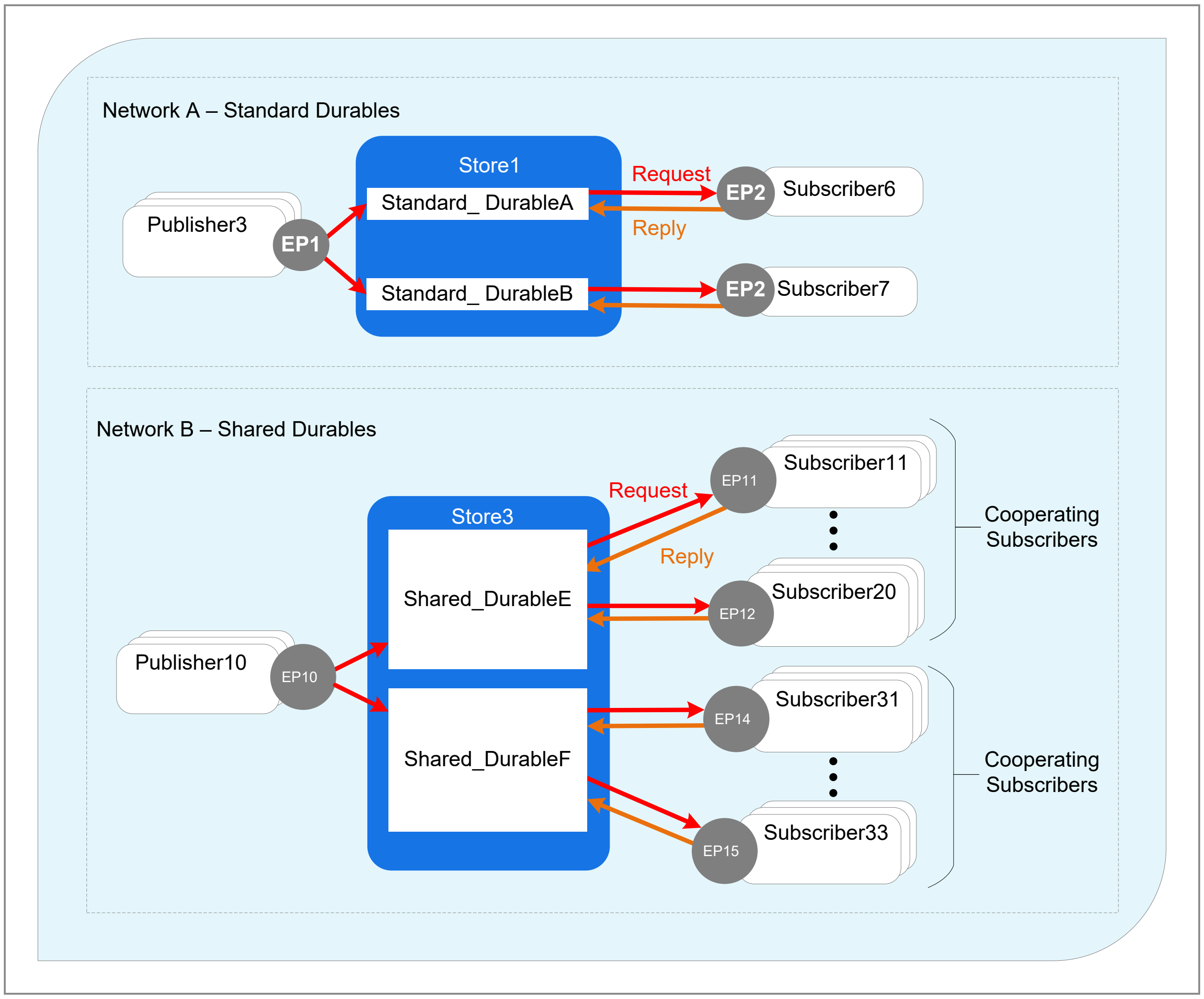
A standard durable serves one subscriber at a time. That subscriber receives the publisher's entire message stream.

A shared durable serves two or more subscribers. A shared durable apportions a message stream among its subscribers and ensures one of the subscribers receive and acknowledges each message.

In the following diagram, the store in Network A has standard durables which represent the interest of one subscriber object per durable. The standard durable collects the message stream from the publisher, which represents potentially many program processes. The standard durable forwards messages to the subscriber on request. The type of persistence for a standard durable depends on whether prefetch is enabled or disable, as detailed in Prefetch: Enabled Versus Disabled. Network A shows prefetch enabled, so there is no direct transport between the publisher and subscriber.

The store in Network B has a shared durable and a group of cooperating subscribers that subscriber to a single store. Subscriber11 through Subscriber20 are instances of the same program subscribing to Shared_DurableE and Subscriber 31 through 33 are instances of the same program subscribing to Shared_DurableF. The durables collect the message stream from the publisher, which represents potentially many program processes. The shared durables forward each message to only one available subscriber and ensure exactly one of those subscribers acknowledges each message. Together, the subscribers process every message in the stream. The subscriber can re-forward the message to a different subscriber, if necessary.

Figure 13: Shared Durable Distributes Messages to Cooperating Subscribers



Persistence Clusters

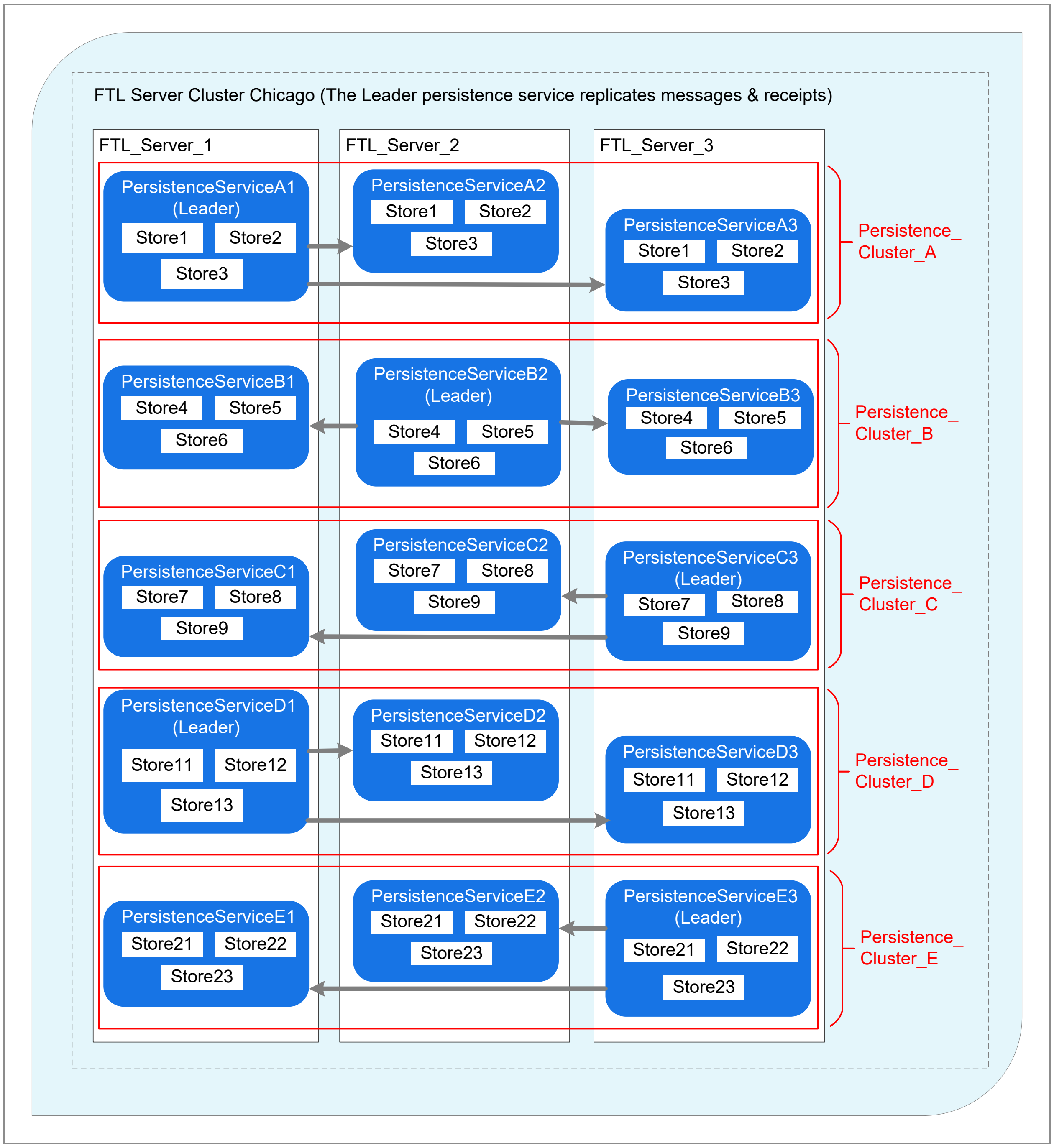
FTL servers run on dedicated host computers and can be organized in clusters. The persistence services cooperate in a *persistence cluster*. Persistent clusters replicate message data and acknowledgment data to ensure services continue to deliver messages to subscribers, even if some of the individual services become unavailable. An FTL server may run any number of persistence services. However, for good fault-tolerance, multiple persistence services in a given FTL server should not be members of the same persistence cluster. Each persistence cluster has one leader that is responsible for receiving messages and acknowledgments from clients and replicating them to other persistence services in the cluster. All client traffic is directed toward the leader.

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. Each persistence cluster has a leader to receive messages and acknowledgments and replicate them to other services in the persistence cluster. Each persistence cluster maintains a set of *persistence stores*. Persistence stores in different persistence clusters have different names. Replication of stores across the persistence cluster protects against hardware or network failures on a small scale. (However, this replication scheme cannot guarantee delivery after catastrophic failures, for example, simultaneous failure of all members of a cluster.)

A publisher publishes to a particular store at a particular persistence cluster, as determined by its endpoint. The message is retained by every durable in the store that is interested in the message (which might be none of them).

Administrators define any number of persistence clusters and services within the realm definition and monitor the health of a persistent cluster.

Figure 14: Persistent Clusters



Durables in Persistence Clusters

Durables are created within a store at a particular persistence cluster when a subscriber (on an endpoint configured for that store) connects to that cluster. The durable exists only at that persistence cluster. However, identical durables may be created at all clusters by separate subscribers at each cluster.

Save and Load a Persistence Cluster State

You can save the state of a persistence service's store to a file on any persistence cluster after suspending the cluster. Suspending the cluster prevents sending and receiving messages until the cluster is restarted. See "Saving and Loading Persistence State" in TIBCO FTL Administration.

Wide-Area Forwarding

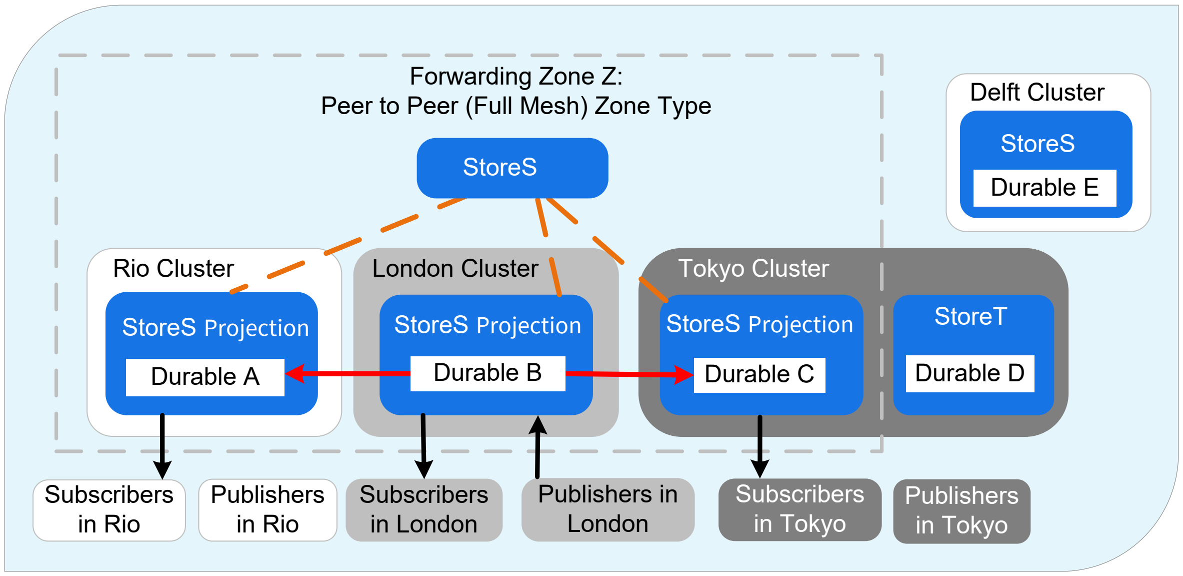
Client applications can publish and subscribe across network boundaries so persistence stores are not tied to a single location. Administrators define the following:

- **Wide-Area Stores**: *Wide-area stores* are accessed by publishers and subscribers in separate networks, connected by a wide-area network (WAN) link.
- **Wide-Area Durables**: *Wide-area durables* are accessible across a WAN link. Durables are created within a store at a particular persistence cluster when a subscriber connects to that cluster. If not a wide-area store, the durable exists only at the particular cluster. The difference with *wide-area stores* is that the durable's interest is propagated throughout the zone. So when a publisher publishes to a wide-area store, the message is retained by any matching durable at any persistence cluster in the zone.
- **Forwarding zones**: *Forwarding zones* delineate the scope of wide-area durables. A forwarding zone contains a set of persistence clusters, which cooperate to forward wide-area durables among all the persistence clusters in that zone.

In the following diagram, StoreS is a wide-area store and is assigned to Zone Z of one of several zone types. (Zone types are covered in the Administration guide.)

The FTL server clusters in Zone Z are: Rio, London, and Tokyo. Each cluster implements a StoreS Projection from StoreS. Application programs in Rio, London, and Tokyo can publish and subscribe to all stores in Zone Z. 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. Even though the Delft Cluster implements StoreS, the cluster does not forward StoreS messages in or out of the Delft Cluster. StoreT is not a wide-area store, so it remains functionally outside Zone Z and local to the Tokyo cluster.

Figure 15: Forwarding the Messages of Wide-Area Stores



Disk-Based Persistence and Message Swapping

You can control memory and disk usage with disk-based persistence and message swapping.

Use these approaches to manage a balance between memory use, messaging throughput (number of messages processed in a given time), and latency (time delay between message generation and delivery).

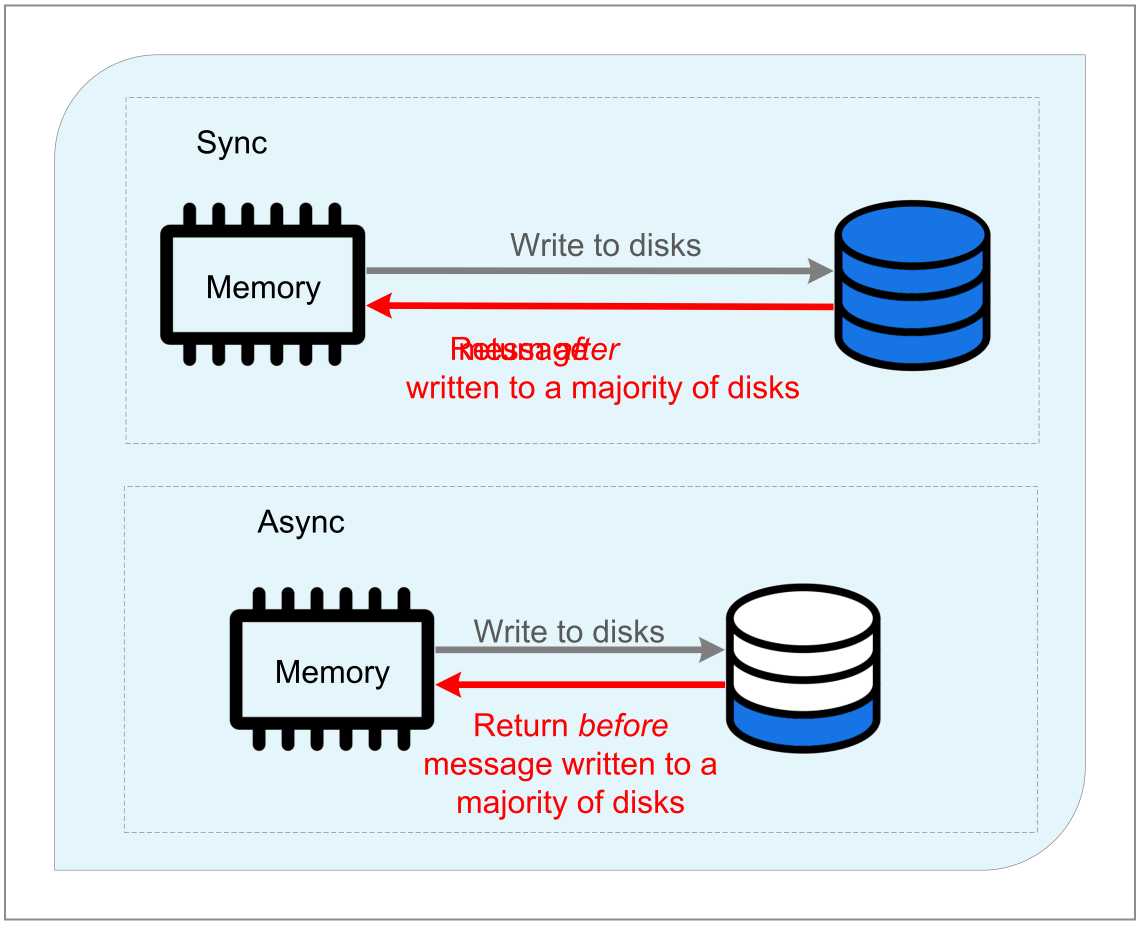
Disk-Based Persistence

Disk-based persistence is enabled on a by-cluster basis and, when enabled, only replicated stores perform disk persistence. Disk-based persistence is enabled in one of two modes, shown in the diagram that follows. For details, see TIBCO FTL Administration, Persistence Architecture, topic Disk-Based Persistence and Message Swapping.

- **Sync**: The client returns from the send call after the message has been written to a majority of disks. This mode generally provides consistent data and robustness, but at the cost of increased latency and lower throughput.

- **Async**: The client may return before the message has been written to a majority of disks. Calls to set or remove keys from a map may return before the set or remove has been written to disk by majority of the FTL servers.
  This mode generally provides less latency and more throughput, but messages could be lost if a majority of persistence services fail simultaneously.

Figure 16: Disk-Based Persistence: Sync and Async



Disk Persistence Enabled Backup and Compaction

Backup and online compaction cannot run at the same time. For details on disk space considerations, see TIBCO FTL® - Enterprise Edition Administration, Persistence Service Disk Capacity, section "Disk Space for Backup or Compaction".

Backup

A backup operation is available for persistence clusters that have disk persistence enabled. The backup saves a snapshot of the persisted state, corresponding roughly to the time the backup was started. Backups proceed in the background, and do not prevent sending and receiving messages while in progress. The stored messages and metadata can be automatically recovered on a full restart of the persistence cluster. See Disk Persistence Backup and Restore.

Compaction

The persistence service can compact its disk persistence files while running without interruption to active publishers or subscribers. Alternatively, you can use a utility to compact persistence files for a given persistence service when the persistence service is not running. For details about compaction options, see TIBCO FTL® - Enterprise Edition Administration:

- Compact Disk Persistence Files with Persistence Service Online
- Compact Disk Persistence Files with Persistence Service Offline

Saving and Loading a Persistence State

You can also save and load a persistence state for in-memory or disk-based clusters. Clients are interrupted. For details, see TIBCO FTL® - Enterprise Edition Administration, topics Saving and Loading Persistence State, section "Restoring Backups".

Message Swapping

Messages held in process memory can be swapped to disk when message swapping is enabled per persistence cluster. Both replicated and non-replicated stores swap when swapping is enabled. Messages are swapped when configured memory limits are exceeded. The limit can be set on a per-store or per-durable basis. Adjusting these limits helps you to manage a balance between memory use, messaging throughput, and latency.

Application Instances (Variant Configurations)

Administrators configure varying arrangements of transports and persistence stores for different environments. Variant communication schemes for processes are called *application instances*. For each application, administrators define one or more application instance. Each application process selects the appropriate application instance at run time based on environmental factors, such as the host name.

Application Instance Definition and Identifiers

The Administrator creates *application instance definitions* which consist of a name, a set of match parameters, and a set of connectors to implement each endpoint. Application instance definitions are used by application processes in a specific context, for example, a process that runs on a specific host computer or a process that supplies an *application instance identifier* which are used at run time for the connection.

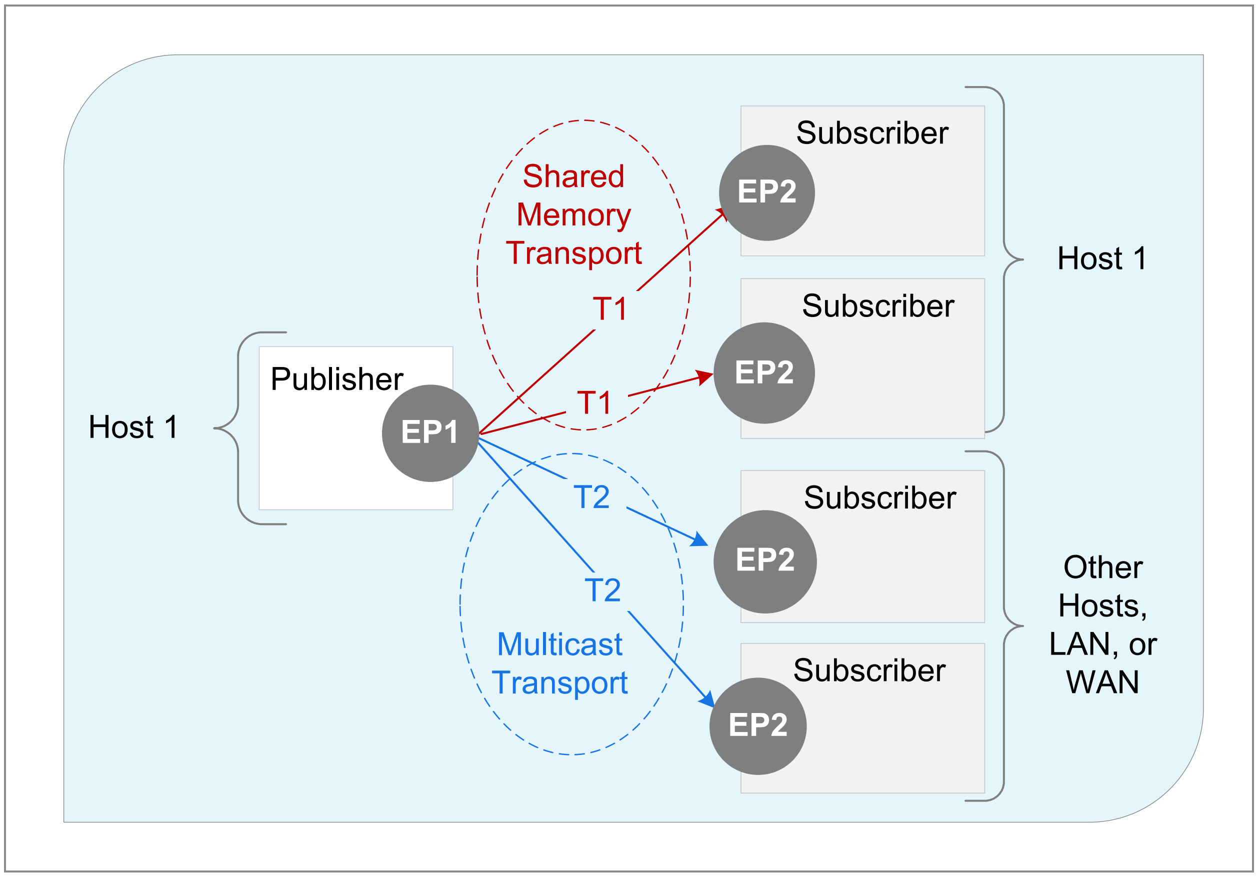
The number of transports and stores are based on whether an application runs as a single process or as many processes:

- When an application runs as a single process, the administrator determines one set of transports and stores.
- When an application runs as many processes, it may require either of the following:
    - Only one set of transports and stores to serve all the application processes in the same way.
    - A different set of transports and stores for different processes.

Example

In the following diagram, the application Publisher sends messages to application Subscriber. Publisher runs on Host 1 as a single process. If Subscriber ran only as a single process or multiple processes on Host 1, the administrator could arrange an *application* *instance* for only Shared Memory Transport T1. However, the Subscriber application processes on other hosts requiring a different transport, so the administrator arranged *application instance* Multicast Transport T2 (a variant communication scheme). The administrator connects Publisher to *both* transport types.

Figure 17: Transport Variants



Realm

In TIBCO FTL software, a *realm* represents one communication fabric, with its endpoints, applications, application instances, transports, stores, and message formats.

A *realm* embraces all the administrative definitions and configurations that enable communication among a set of cooperating programs.

Sometimes a small enterprise needs only one realm. However, at other times development and test environments are needed, which must not interfere with a production environment. Each of these environments needs a separate realm because communication must be isolated by environment.

The *realm definition* defines the realm and can include configurations for applications, endpoints, transports, transport abilities of an endpoint in an application instance, message formats, and other settings.

Programmer and Administrator Roles

An endpoint contains transport details and helps separate programmer and administrator responsibilities.

Programmer’s Role

Programmers use endpoints to publish and subscribe to message streams. The program uses the endpoint name so transport details can be omitted in the programs. The program determines its publish and subscribe behavior with respect to its endpoints. The transports and persistence stores are external to a program.

Administrator’s Role

Administrators can determine the following:

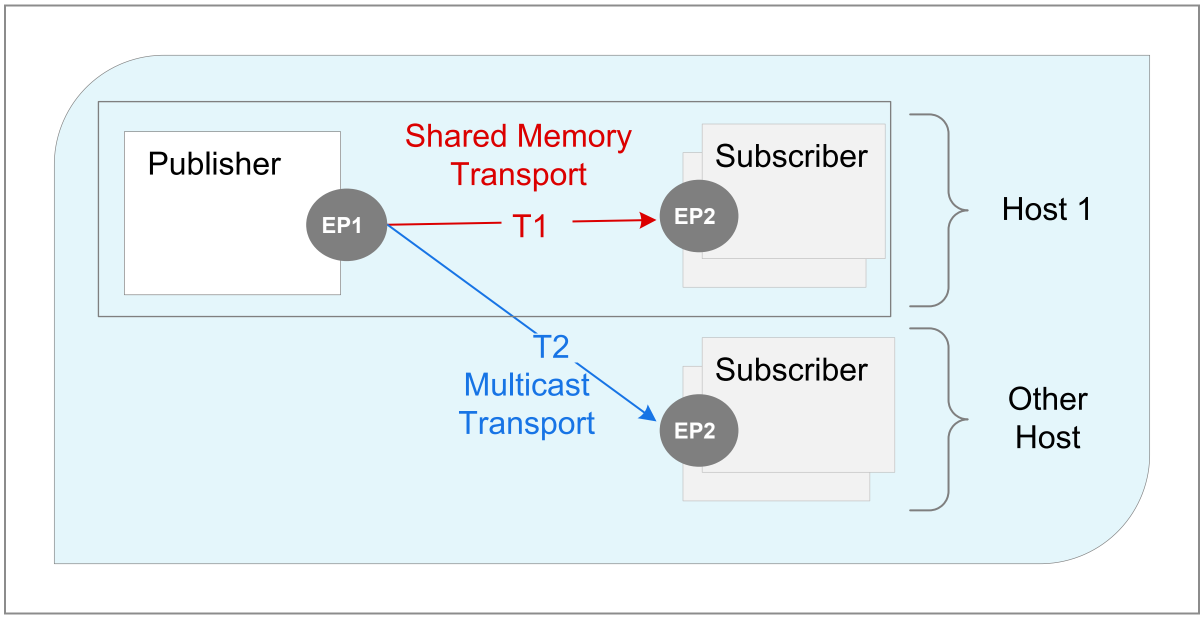
- Administrators determine the host computers where application processes run. An application could run as a single process: for example, a central repository application. Or an application could run as many processes: for example, a workstation interface application for stock market traders.
- Administrators configure the set of direct path transports that connect application endpoints, allowing message streams to flow.
- Administrators configure a set of persistence stores, providing an indirect path and backing application endpoints.

Transports and persistence stores are external to the programs and only the endpoints are visible to programmers.

Role Example

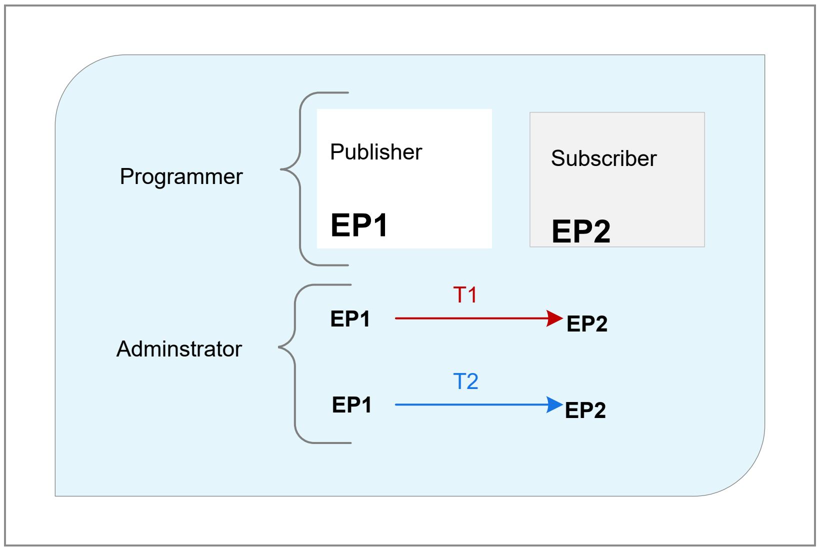
Recall the earlier diagram where each subscriber has a different type of transport for the same message.

Figure 18: Two Types of Transports



The following diagram represents the duties of the programmer and administrator. The programmer adds the endpoints (EP1 and EP2) in the publisher and subscriber programs. The Administrator configures the transports to work with EP1 and EP2.

Figure 19: Separation of Responsibilities



Coordination Forms

The following TIBCO FTL Forms are available to coordinate the work between programmers and administrators.

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

Concept Review

Review the concepts to solidify your understanding of TIBCO FTL software.

Messages are sent in *message streams* from *publishers* to *subscribers*. There are several publish-subscribe models:

- One Publisher to Many Subscribers (TIBCO FTL's main paradigm)
- Many Publishers to Many Subscribers
- One-to-One Communication utilizing an inbox
- Request/Reply with Inbox Tokenutilizing an inbox
- Distinct Endpoints in One Program

Message streams travel from publishers to subscribers over a *transport* which is a shared communication medium. TIBCO FTL supports a variety of transport types to use based on messaging requirements and environment. See Transport Types.

A messaging *broker* is a intermediate *persistence store* (or store) which temporarily holds messages. An FTL server can act as a broker. FTL may operate with broker messaging or brokerless messaging.

A *persistence store* (also called a *store*) temporarily holds a message stream sent by a publisher and held for the subscriber. Each store collects and maintains exactly one message stream and can collect the stream from one or more publishers. The store can maintain that message stream for one or many subscribers. Administrators can configure stores within a persistence service.

A *durable* is the data structure within a store that maintains subscriber interest, forwards messages, and tracks delivery acknowledgments. There are different types of durables. See Persistence: Stores and Durables.

An *endpoint* is the connection point between a transport and the publisher, subscriber, or broker. An endpoint contains transport details and insulates programs, and their designers/users, from those details. This enables separation of responsibilities between developers and administrators. Many publishers and many program processes may publish on one publishing endpoint. One subscriber may consume from more than one publishing endpoint. A program can have more than one endpoint to tap into more than one transport message stream

Messages travel from publishers to subscribers over a *transport* which is a shared communication medium. TIBCO FTL supports a variety of Transport Types to use based on messaging requirements and environment. Each transport is a separate communication medium, insulated against crosstalk from other transports.

A *content matcher* is a pattern matching template for message fields and values. A subscriber can use a content matcher to narrow its interest to a subset of a message stream.

A *format* defines the set of fields that a message can contain, including field names and data types. The format is available to cooperating programs before they exchange messages. Messages only carry data, which reduces both the message size, transport bandwidth, and processing time.

Viewing an application as a collection of endpoints, an *application instance definition* configures endpoints for use by application processes in a specific context, for example, a process that runs on a specific host computer or supplies an *application instance identifier*.

A *realm* embraces all the administrative definitions and configurations that enable communication among a set of cooperating programs.

See for a review of your role and helpful forms to coordinate between programmers and administrators.

From Concepts to Usage

Understanding the concepts is the first step to using TIBCO FTL software effectively. The next step depends on your role as either a programmer or an administrator.

The terms we have defined are basic concepts, and not necessarily complete functional descriptions.

These concepts become concrete in two different ways:

- For programmers, the concepts of application, endpoint, publisher, subscriber, inbox, message, content matcher and format all become concrete through the TIBCO FTL® - Enterprise Edition API. For more information, see TIBCO FTL® - Enterprise Edition Development and the API reference material in the FTL documentation.
- For administrators, the concepts of application, endpoint, transport, store, one-to-many publishing, one-to-one messaging, application instance, format and realm all become concrete through the TIBCO FTL realm definition. For more information, see TIBCO FTL® - Enterprise Edition Administration in the FTL documentation.

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.