TIBCO FTL®
Concepts

Software Release 5.2
March 2017
Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO, Two-Second Advantage, TIB, Information Bus, eFTL and Rendezvous are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

Enterprise Java Beans (EJB), Java Platform Enterprise Edition (Java EE), Java 2 Platform Enterprise Edition (J2EE), and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle Corporation in the U.S. and other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

THIS 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 THIS SOFTWARE VERSION ON A SPECIFIC OPERATING SYSTEM PLATFORM.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

Copyright © 2010–2017 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figures</td>
<td>4</td>
</tr>
<tr>
<td>About this Product</td>
<td>5</td>
</tr>
<tr>
<td>TIBCO Documentation and Support Services</td>
<td>6</td>
</tr>
<tr>
<td>FTL Concepts: a Bottom-Up Approach</td>
<td>8</td>
</tr>
<tr>
<td>Messaging Model</td>
<td>9</td>
</tr>
<tr>
<td>One-to-Many Publishing</td>
<td>10</td>
</tr>
<tr>
<td>Transports</td>
<td>11</td>
</tr>
<tr>
<td>Transport Types</td>
<td>12</td>
</tr>
<tr>
<td>Endpoints</td>
<td>13</td>
</tr>
<tr>
<td>One Endpoint Many Subscribers</td>
<td>13</td>
</tr>
<tr>
<td>One Endpoint Many Publishers</td>
<td>14</td>
</tr>
<tr>
<td>Distinct Endpoints in One Program</td>
<td>15</td>
</tr>
<tr>
<td>Content Matchers</td>
<td>16</td>
</tr>
<tr>
<td>One-to-One Communication</td>
<td>17</td>
</tr>
<tr>
<td>Request/Reply Pattern</td>
<td>18</td>
</tr>
<tr>
<td>Persistence: Stores and Durables</td>
<td>19</td>
</tr>
<tr>
<td>Stores and Durables</td>
<td>20</td>
</tr>
<tr>
<td>Store Examples</td>
<td>20</td>
</tr>
<tr>
<td>One Store</td>
<td>20</td>
</tr>
<tr>
<td>Two Stores</td>
<td>21</td>
</tr>
<tr>
<td>Shared Durable</td>
<td>22</td>
</tr>
<tr>
<td>Direct-Path Requirement and Merging</td>
<td>23</td>
</tr>
<tr>
<td>Formats</td>
<td>25</td>
</tr>
<tr>
<td>Application Endpoints</td>
<td>27</td>
</tr>
<tr>
<td>Variant Configurations</td>
<td>28</td>
</tr>
<tr>
<td>Realm</td>
<td>29</td>
</tr>
<tr>
<td>Concept Review</td>
<td>30</td>
</tr>
<tr>
<td>From Concepts to Usage</td>
<td>31</td>
</tr>
</tbody>
</table>
Figures

Publishers, Subscribers and Message Streams .......................................................... 9
One-to-Many Publishing ......................................................................................... 10
Transports Carry Separate Message Streams ......................................................... 11
Transports of Different Types ............................................................................... 12
Endpoints ............................................................................................................... 13
Separation of Responsibilities .............................................................................. 14
Adding Another Publisher ................................................................................... 14
Different Messaging Requirements ...................................................................... 15
One-to-One Communication, Sending to an Inbox ........................................... 17
Handshake to Arrange Inbox Communication .................................................... 18
Store Backs a Transport ...................................................................................... 20
Stores Isolate Separate Message Streams ........................................................... 22
Shared Durable Distributes Messages to Subscribers .......................................... 22
Network of Endpoints and Direct Paths ............................................................... 23
Messages: Data & Metadata ................................................................................. 25
Transport Variants ................................................................................................. 28
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 leverage the power of Information Bus® technology to enable truly event-driven IT environments. To find out more about how TIBCO FTL software and other TIBCO products are powered by TIB® technology, please visit us at www.tibco.com.
TIBCO Documentation and Support Services

Documentation for this and other TIBCO products is available on the TIBCO Documentation site. This site is updated more frequently than any documentation that might be included with the product. To ensure that you are accessing the latest available help topics, visit:

https://docs.tibco.com

Product-Specific Documentation

Documentation for TIBCO products is not bundled with the software. Instead, it is available on the TIBCO Documentation site. To access the documentation web page for this product from a local software installation, open the following file:

TIBCO_HOME/release_notes/TIB_ftl_5.2.0_docinfo.html

TIBCO_HOME is the top-level directory in which TIBCO products are installed.

- On Windows platforms, the default TIBCO_HOME is C:\tibco.
- On UNIX platforms, the default TIBCO_HOME is /opt/tibco.

The following documents for this product can be found on the TIBCO Documentation site.

**TIBCO FTL® Documentation Set**

- **TIBCO FTL Concepts** *Everyone reads this booklet first*, for an intuitive introduction to the fundamental concepts of FTL software.
- **TIBCO FTL Development** Application developers and architects read this manual to understand concepts relevant in any supported programming language.
- **TIBCO FTL API Reference** Application developers use this HTML documentation to learn the details of the FTL API in specific programming languages.
- **TIBCO FTL Administration** Administrators read this manual to learn how to use the realm server and its interfaces, and how to define a realm. Developers can also benefit from understanding FTL software from an administrator’s perspective.
- **TIBCO FTL Monitoring** Administrators read this manual to learn about monitoring and metrics. Developers read this manual to learn how an application can subscribe to the stream of monitoring data.
- **TIBCO FTL Installation** Read this manual before installing or uninstalling the product.
- **TIBCO FTL Glossary** The glossary contains brief definitions of key terms used in all other parts of the documentation set.
- **TIBCO FTL Release Notes** Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.

**TIBCO eFTL™ Documentation Set**

TIBCO eFTL software is documented separately. Administrators use the FTL realm 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 TIBCO Support

For comments or problems with this manual or the software it addresses, contact TIBCO Support:

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support
If you already have a valid maintenance or support contract, visit this site:

https://support.tibco.com

Entry to this site requires a user name and password. If you do not have a user name, you can request one.

**How to Join TIBCOmmunity**

TIBCOmmunity is an online destination for TIBCO customers, partners, and resident experts. It is a place to share and access the collective experience of the TIBCO community. TIBCOmmunity offers forums, blogs, and access to a variety of resources. To register, go to the following web address:

https://www.tibcommunity.com
FTL Concepts: a Bottom-Up Approach

This booklet begins with a simplified view of TIBCO FTL® messaging, and gradually adds details and features in conjunction with requirements they satisfy or the problems they solve. In this way, we build a full picture of TIBCO FTL software and its capabilities.

The terms we define along the way are basic concepts, and not necessarily complete functional descriptions. For details about these concepts, and the information you need to use them, see TIBCO FTL Development and TIBCO FTL Administration.
Messaging Model

To illustrate the fundamental principles of TIBCO FTL software, we begin with the simplest form of its messaging model.

A message is data, structured as a set of named fields. Fields contain values.

TIBCO FTL software transfers messages. Transferring a message involves two distinct roles:

- **Publishers** send messages.
- **Subscribers** receive messages.

**Publishers, Subscribers and Message Streams**

An application can send messages, receive messages, or both. The terms *publisher* and *subscriber* refer to TIBCO FTL objects that perform these two roles within applications.

A message stream is a sequence of messages.

- A *publisher* is the source of a stream of messages.
- A *subscriber* expresses interest in a stream of messages.
One-to-Many Publishing

The primary data-transfer model in TIBCO FTL software is *one-to-many publishing*. A publisher produces a stream of messages, and every subscriber to that stream can receive every message in the stream.

*One-to-Many Publishing*

Pub1 ➔ Sub1, Sub2

This model is analogous to radio broadcast: a radio station broadcasts a stream of audio data, and every radio receiver on the same frequency can receive that audio stream.
Messages travel from publishers to subscribers over transports. The publishers and subscribers on a transport all use the transport in common as a shared communication medium.

Each transport is a separate communication medium, insulated against crosstalk from other transports. The following diagram depicts this insulation using separate colors. Subscribers on transport T1 receive messages from Pub1 on T1, but not from Pub2 on transport T2, nor from publishers on any transport other than T1.

**Transports Carry Separate Message Streams**

![Diagram of transports and messages]

In the radio broadcast analogy, frequencies divide the airwaves into separate data streams. Each frequency is, in effect, a distinct communication medium that carries data to all devices tuned to that frequency. Radios tuned to frequency F1 receive the audio stream from the station that broadcasts on F1, but not from stations that broadcast on frequency F2, nor any other frequency.
Transport Types

The real world offers varying communication technologies, each appropriate in particular situations. Accordingly TIBCO FTL software offers different types of transports, which you can combine to fit the communication needs of your enterprise.

For example, when publishers and subscribers are co-located on the same host computer, they can use a shared memory transport. When publishers and subscribers are located on separate host computers, linked by a LAN, they can use a multicast transport.

The following diagram shows two such transports. Notice that the two transports carry identical message streams. The publisher sends each message once, but TIBCO FTL software transfers it over both transports.

*Transports of Different Types*
Endpoints

TIBCO FTL software uses the concept of an *endpoint* to simplify arrangements when multiple transports carry the same message stream. An endpoint abstracts all the transport details into a simple name. This abstraction insulates application programs from those details.

Transports are analogous to communications wiring behind the walls of an office building, while endpoints are like the plug and wall socket through which devices connect to that wiring. The designers and users of those devices do not need to know the specific communication technology behind the walls.

**One Endpoint Many Subscribers**

In this first example, the message stream from one publishing endpoint fans out to many subscribers that require different transport technologies. Endpoints help manage this complexity, and separate the responsibilities of programmers from the responsibilities of administrators.

The first diagram illustrates the role of endpoints, using the same situation as the diagram Transports of Different Types. Programs (blue) use endpoint names (black) to arrange publisher and subscriber objects (green triangles and circles). Administrators configure transports (red), and use them to connect endpoints (black).

Endpoints help to separate the responsibilities of programmers from the responsibilities of administrators. Continuing the example into the second diagram (below), the programmer must implement two application programs: the publishing program and the subscribing program. Within those programs, the programmer must arrange publisher and subscriber objects. The administrator determines where to deploy those programs, and configures the transports that allow them to communicate. Endpoints are a dividing line between these separate responsibilities, and a common ground between programmers and administrators.
Separation of Responsibilities

Notice that the preceding two diagrams present two views of the same situation.

To review, we have seen two sources of potential complexity: deploying programs in varying environments, and connecting them with different transport technologies. We also saw how TIBCO FTL software uses the concept of an endpoint to simplify this complexity.

One Endpoint Many Publishers

This second example extends the previous example by adding another publisher. Endpoints hide any additional complexity from programmers, and simplify the task of administrators.

The diagram (below) shows two host computers connected over a LAN. Two publishers publish messages: one publisher on each host. On each host, many subscribers receive messages. Every subscriber receives the messages from each of the publishers.

As in the previous example, programs co-located on the same host computer communicate over shared memory transports, while programs on separate hosts communicate over multicast transports. Observe the increased number of transports (red lines). Replicating this arrangement on additional hosts further multiplies the number of transport lines.

Adding Another Publisher

The oval shapes in the diagram hint at the way endpoints can simplify this situation. Notice that the programmer’s responsibility is the same as in the previous example: implement the publishing...
program and the subscribing program. Only the administrator’s work has increased, though not as much as one might at first think. The administrator deploys the additional publishing program on Host 2, but the transport configuration remains the same as in the preceding example (refer back to the diagrams Endpoints and Separation of Responsibilities). TIBCO FTL software uses the configuration of the shared memory transport (T1) to automatically arrange two corresponding transport buses, T1a and T1b, one on each host computer. Similarly, TIBCO FTL software can reuse the configuration of the multicast transport (T2), integrating the new publisher into the existing transport bus.

To review, we have seen that endpoints simplify programming by insulating programs from the complexity of transports. We have also seen that endpoints can simplify the administrative task of arranging transports.

### Distinct Endpoints in One Program

In the two preceding examples, each program used only one endpoint. In this third example, we consider a reason that a program might need more than one endpoint.

An endpoint allows a program to access transports so a program can communicate over a rich set of technologies without specifying their arrangement. To a program, an endpoint is like a tap into a message stream. A program with more than one endpoint reflects the need to tap into more than one message stream.

#### Different Messaging Requirements

In the preceding diagram, a stock market data feed (Feed) sends a message stream (Inform) that consists of many small messages, produced at high speed, containing time-sensitive data such as trades, buy offers and sell offers. It is crucial that the subscribers (Trade) receive these messages with minimal latency, and act without delay.

As a result of those actions, Trade produces two message streams. Like the Inform stream, the Execute stream is time-sensitive.

In contrast, the Report stream is not time-sensitive. Its messages satisfy regulatory reporting requirements. It is crucial that subscribers (Rpt) receive and process these messages, but speed is not as important. However, it is critical that the Report stream not interfere with the Execute stream, nor slow it in any way.

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.
**Content Matchers**

Subscriber objects can specify interest in messages based on their content, that is, based on message fields and their values. TIBCO FTL messages do not carry explicit destination information.

In contrast, a TIBCO Rendezvous® message carries an explicit subject name, which directs it to subscribers on that subject. A JMS message carries an explicit destination: either a topic name or a queue name. But TIBCO FTL messages do not.

You do not need to establish in advance which fields of a TIBCO FTL message guide its delivery. You do not need to duplicate any message content in a subject or topic name. Indeed, duplication would be inefficient.

A content matcher selects a subset of messages from a message stream according to the fields and values in those messages. A subscriber uses a content matcher to express interest in the subset of messages that the content matcher specifies.

Without a content matcher, a subscriber receives all the messages that arrive at its endpoint. Using a content matcher, a subscriber receives only a subset of those messages: namely, those messages that fit the matcher’s specifications. For example, a subscriber could receive only messages in which the Symbol field contains the value TIBX.
One-to-One Communication

One-to-one communication lets a publisher send a message to one specific subscriber.

In the one-to-many publishing model, a publisher produces a stream of messages, and every subscriber to that stream can receive every message in the stream. However, sometimes a publisher needs to direct messages to a specific subscriber, even though many subscribers share the transport. Such situations require a different publishing model: one-to-one communication.

**One-to-One Communication, Sending to an Inbox**

In one-to-one communication, a publisher sends a message to a specific inbox subscriber: only that inbox subscriber receives the message. One-to-one communication can use the same transports as one-to-many communication.

In the preceding diagram, the program Pub sends a message stream over transport T1 (red). The programs Sub1 and Sub2 both receive that message stream, as one would expect with one-to-many publishing. Notice that the same transport, T1, also carries one-to-one messages (orange dashed line) to an inbox within Sub1, but Sub2 does not receive these one-to-one messages. Sub2 can have its own inbox, and Pub can send to it, but neither inbox receives messages directed to the other.
Request/Reply Pattern

Request/reply is a common interaction pattern, which combines both one-to-many and one-to-one communication.

In order to send a message to an inbox, a publisher must have a token representing that inbox, which it receives from the program that created the inbox. Cooperating programs can use a handshake sequence to arrange inboxes for one-to-one communication. The following diagram illustrates this pattern in a request/reply interaction.

The server program (Serv) creates a subscriber object (S) to receive request messages from clients.
A client program (Rqst) creates an inbox subscriber object (I) to receive replies from the server.
The client sends a request message, including a token representing inbox I.
The server uses the token to send a one-to-one reply message to inbox I at the client.

Handshake to Arrange Inbox Communication

Notice that the client publishes the request message one-to-many, but the server sends the reply one-to-one.
Persistence is the ability to temporarily store a message at an intermediary between publishers and subscribers.

Message delivery through a transport bus is immediate and transient:

- Transports carry messages immediately from publisher to subscribers. Messages do not pass through an intermediary, such as a server. Transports neither store messages in transit, nor forward them later.
- Messages are transient. Publisher objects and transports do not indefinitely retain messages after transmitting them to current subscribers.

These two qualities, immediacy and transience, promote speed and simplicity. However, sometimes we need a different quality, persistence, which is the potential to store a message between sending and receiving it.

We can introduce a store-and-forward intermediary to provide persistence. Consider the following aspects of its operation.

**Persistent Message Streams**

To receive a message through a transport, a subscriber must be present when the message is published, and the transport bus between publisher and subscriber must be working properly. To transcend the limitations of transient messages, a store-and-forward intermediary can maintain a persistent message stream, collecting and retaining messages from publishers. In a sense, it serves as a backup publisher for the benefit of subscribers.

**Persistent Message Interest**

A subscriber object expresses transient interest in a message stream. When the subscriber closes, its interest in those messages ceases. A subscriber misses any messages that pass through a transport bus after the subscriber closes or before the client program creates the subscriber. To transcend the limitations of transient interest, a store-and-forward intermediary can express interest that endures even when no subscriber object is present, maintaining persistent interest for the benefit of subscribers.

**Apportion Message Streams**

In one-to-many publishing, each subscriber expresses interest in a message stream, and receives every message in that stream through the transport bus. However, sometimes we might want to apportion a message stream among a set of cooperating subscribers, so that only one subscriber receives and processes each message, but acting together they process every message in the stream. A store-and-forward intermediary can facilitate this behavior.

**Last Value**

In some applications, a newly started application process needs only the most recent message from a message stream, rather than the entire stream. For example, in financial market ticker applications, the most recent price information about a stock is very important when an application starts or restarts, while previous messages containing out-of-date price information could be unimportant. In applications that monitor information technology environments, the current status of each resource is very important, and the status history of a resource could be unimportant.

A store-and-forward intermediary can express interest in a message stream, divide the message stream into sub-streams based on the string value of a key field, and cache the most recent message in each sub-stream. Applications can subscribe to individual sub-streams.
Key/Value Maps

A key/value map stores key/value pairs, in which each key is a string, and a key’s value is a message. Applications can use map methods to store and retrieve key/value pairs, and to iterate over the pairs in a map. Map methods provide a convenient way to use the store-and-forward intermediary as a simple database table.

Stores and Durables

As the phrase store-and-forward intermediary suggests, the intermediary provides its functionality in two phases. Two types of data structure, store and durable, facilitate these phases.

Store

A persistence store, or more briefly, a store, is a data structure that collects and stores a stream of messages from publishers.

Each store maintains exactly one message stream. It can collect that stream from one or more publishers. It can maintain that message stream for potentially many subscribers.

Durable

Within a store, durables maintain subscriber interest, forward messages, and track delivery acknowledgements.

- A standard durable represents the interest of one subscriber object. It strengthens delivery assurance for that subscriber, forwarding messages to the subscriber on request.
- A shared durable represents the identical interest of two or more cooperating subscriber objects. It apportions a message stream among those subscribers, forwarding each message to only one available subscriber. (That subscriber can reforward the message to a different subscriber if necessary.)
- A last-value durable represents the interest of potentially many independent subscriber objects. It divides its input message stream into a set of output sub-streams, storing only the most recent messages for each sub-stream. Upon subscribing, each new subscriber receives the most recent message. The durable continues to forward subsequent messages as they arrive.

Store Examples

The following examples highlight different aspects of stores, and introduce related concepts.

One Store

The simplest example of persistence consists of one store with standard durables, as in the following diagram.

Store Backs a Transport

- In network A, program Pub1 publishes messages on endpoint EP1, while program Sub2 receives them on endpoint EP2.

We say that transport T1 is the direct path connecting the two endpoints.
• Store1 also receives the message stream from EP1, and retains each message until the subscriber object (S) in Sub2 acknowledges the message. If Sub2 were absent, or if the direct path failed, Store1 would retain unacknowledged messages, and retransmit them to Sub2 as needed.

We say that Store1 backs the network of endpoints: namely EP1 and EP2. This statement is equivalent to saying that Store1 backs all the direct paths in the network T1. We can also say that Store1 backs the message stream that flows from the publishing endpoint, EP1.

As we discuss a network of endpoints, our focus shifts among three aspects:

• the endpoints
• the direct paths connecting those endpoints (that is, the transports connecting the endpoints)
• the combined message stream that originates with the publishing endpoints, that is, the combined message streams from all of the publishers on those endpoints

Notice these additional features of this example:

• Store1 operates alongside the transport T1: it parallels the direct path, but it does not replace T1.
• Messages from EP1 flow simultaneously along T1 and into Store1.
• A message does not flow from Store1 to EP2 unless Sub2 has missed that message, and later becomes present to accept it.
• Store1 maintains interest in the message stream even when Sub2 is absent.
• In the diagram Pub1 represents potentially many program processes, all of which publish on endpoint EP1. Their message streams merge into Store1, just as they merge in transport T1. Neither Store1 nor the subscribers in program Sub2 distinguish the origin of any message.
• Sub2 also represents potentially many program processes, each of which subscribes to a separate standard durable. Each durable ensures that its subscriber receives the entire merged message stream that Store1 captures.

Two Stores

This second example of persistence illustrates the need for two stores. The following diagram depicts two separate networks of endpoints:

• Network A (which is familiar from the previous example) consists of endpoints EP1 and EP2, connected by direct-path transport T1.
• Network B consists of endpoints EP3 and EP4, connected by direct-path transport T2.
Stores Isolate Separate Message Streams

Network A

Pub1
EP1

Sub2
EP2

T1

Store1

Network B

Pub3
EP3

Sub4
EP4

T2

Store2

Notice these features of the example in the preceding diagram:

- Each network carries a separate message stream: the message stream of network A is separate and disjoint from the message stream of network B.
- Store1 backs network A, while Store2 backs network B. Each store backs only one network. Each network includes only one store.
- Each store collects only the message stream of the network that it backs. For example, messages from network B do not enter Store1, because we deliberately kept network A separate from network B.

Shared Durable

This example illustrates the operation of a persistence store with a shared durable. (In contrast, the preceding examples illustrated stores with standard durables.)

Shared Durable Distributes Messages to Subscribers

Network C

Pub10
EP

Sub11
EP

Sub20
EP

Store3

Notice these features of the example in the preceding diagram:

- Pub10 publishes messages through its endpoint. Notice that subscribers Sub11 through Sub20 do not receive those messages along a direct path. (Contrast the preceding examples of standard durables, which required a direct path.)
- Store3 collects the message stream from Pub10, which represents potentially many program processes.
Sub11 through Sub20 are all instances of the same program. They all subscribe to one shared durable within Store3. (Contrast the preceding examples, in which each program subscribed to a separate standard durable.)

The shared durable distributes the message stream to Sub11 through Sub20.

The subscribers receive their portion of the message stream, process the messages, and acknowledge completion back to the shared durable.

An individual message is processed by exactly one of the subscribers. Together, the subscribers process every message in the stream.

Direct-Path Requirement and Merging
This example returns for a closer look at two aspects of persistence stores: the direct-path requirement of standard durables, and the merging of publisher streams into one message stream.

Network of Endpoints and Direct Paths

Network D

The preceding diagram depicts a network of endpoints.

- Network D consists of endpoints EP30 through EP33, connected by direct-path transports.
- Store4 backs network D. That is, it backs all the direct-path transports.
- Actually, network D is not more topologically complex than network A in the earlier diagram Store Backs a Transport. Rather, we have drawn more detail to highlight the direct paths, and the merging of publisher message streams.

Direct-Path Requirement

Focus on the direct-path transports in the diagram:

- A direct-path transport connects every publishing endpoint to every subscribing endpoint.
- A store with standard durables cannot be the only path between a pair of endpoints. The store can parallel a direct path, but does not serve as a direct path.

Merging within a Network of Endpoints

Now focus on the publisher message streams in the diagram:

- Each direct path flowing from EP30 merges message streams from potentially many publishers in the potentially many program processes that publish on EP30. When a message arrives at a subscriber, it does not include information that identifies the publisher nor the publishing program.
- Similarly, Store4 merges the message streams from publishing endpoints EP30 and EP31. When message arrives at Store4, it does not include information that identifies the publishing endpoint.
• Although this example uses standard durables, merging would be identical if the store used shared durables.
Formats

Formats promote efficient message transfer by conserving transport bandwidth.

We have seen that a message is data, structured as a set of named fields, which contain values. Each message field contains data of a specific datatype. For simplicity and efficiency, TIBCO FTL messages support a limited set of data types. A publishing program can assemble a message with an arbitrary number of fields and any combination of the supported data types.

However, using arbitrary sets of fields has a cost: namely, each message must be fully self-describing. A self-describing message contains metadata: the names and data types of each field. Including the metadata enlarges each message, and results in slower processing of messages.

When the full flexibility of self-describing messages is not needed, TIBCO FTL software lets you achieve greater efficiency by abstracting the field metadata from a group of similar messages. We can define a message format, that is, a fixed set of field names and data types. Formats give cooperating programs the metadata to describe their messages, even before they exchange any messages. So a message that conforms to a format need not carry its own metadata, which reduces both message size and processing time.

Compare the self-describing message at the top of the following diagram against its more compact data at the bottom of the diagram.

Messages: Data & Metadata

<table>
<thead>
<tr>
<th>Field 1</th>
<th>Field Name</th>
<th>Field Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field 2</td>
<td>Field Name</td>
<td>Field Type</td>
<td>Value</td>
</tr>
<tr>
<td>Field 3</td>
<td>Field Name</td>
<td>Field Type</td>
<td>Value</td>
</tr>
<tr>
<td>Field 4</td>
<td>Field Name</td>
<td>Field Type</td>
<td>Value</td>
</tr>
</tbody>
</table>

Format (Metadata Only)

<table>
<thead>
<tr>
<th>Field 1</th>
<th>Field Name</th>
<th>Field Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field 2</td>
<td>Field Name</td>
<td>Field Type</td>
</tr>
<tr>
<td>Field 3</td>
<td>Field Name</td>
<td>Field Type</td>
</tr>
<tr>
<td>Field 4</td>
<td>Field Name</td>
<td>Field Type</td>
</tr>
</tbody>
</table>

Message Using Format (Data Only)

<table>
<thead>
<tr>
<th>Field 1</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field 2</td>
<td>Value</td>
</tr>
<tr>
<td>Field 3</td>
<td>Value</td>
</tr>
<tr>
<td>Field 4</td>
<td>Value</td>
</tr>
</tbody>
</table>

For example, in an order fulfillment application, a product request message could contain a product name field with a string value, a warehouse bin field with an integer value, and a quantity field with an integer value. Programmers determine the format. Administrators define the format accordingly. Programs automatically receive this format definition at runtime.

When programs send messages with this format, they can include values for any subset of the fields in the format definition. Receiving programs already have the metadata they need to unpack data values from messages with this format.
TIBCO FTL software does not limit the number of formats that a program can use for sending or receiving messages. Nonetheless, in practice, many applications exchange only a few types of messages.
Application Endpoints

At the most abstract level, an FTL application is a collection of endpoints. FTL applications can send and receive message streams through their endpoints.

Programmer’s View

Programmers use endpoints to publish and subscribe to message streams. The program determines its publish and subscribe behavior with respect to its endpoints.

Administrator’s View

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 also determine a set of transports that connect application endpoints, allowing message streams to flow.

Administrators can also determine a set of stores to back application endpoints.

Yet this set of transports and stores is external to the program. Only the endpoints are visible to programmers.
Variant Configurations

Administrators can define application instances to configure varying arrangements of transports and stores for different environments.

When an application runs as a single process, the administrator determines one such set of transports and stores.

Sometimes, when an application runs as many processes, it nonetheless requires only one set of transports and stores. That is, the same set serves all the application processes in the same way.

However, at other times when an application runs as many processes, some of those processes require a different set of transports and stores than the others do.

For example, the following diagram returns to an earlier example, in which application Pub sends messages to application Sub.

Transport Variants

Pub runs as a single process on host 1. If Sub were to run as a single process on host 1 as well, then the administrator could arrange a very simple communication scheme, consisting only of shared memory transport $T_1$. Even if host 1 were to run many Sub processes, that same scheme could serve them all.

However, if we add Sub processes on other hosts, they cannot use $T_1$, because shared memory transports can operate only within a single host computer. Sub processes on other hosts require a different transport type, so the administrator must arrange a variant communication scheme for these processes.

The administrator must connect Pub to transports of both types. In contrast, Sub processes must use different variants. The correct variant depends on the host computer where the process runs.

In TIBCO FTL software we call such variants application instances. For each application, administrators can define one or more application instances. Each application process selects the appropriate application instance at run time based on environmental factors, such as the host name.
Realm

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 communication fabric. However, at other times we need development and test environments, which must not interfere with a production environment. Each of these environments needs a separate communication fabric, and we must *isolate* their communications from one another.

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

Reviewing the progression of concepts can help solidify your understanding of TIBCO FTL software. We began with simplified versions of the most important concepts of TIBCO FTL software. Yet while the simplified concepts promote understanding, they were insufficient to satisfy more complex messaging needs. Each new requirement added new concepts until we arrived at the full picture of TIBCO FTL software.

First we introduced simple messaging: publishers, subscribers and message streams. We explained FTL’s main paradigm: one-to-many publishing. We saw how transports carry message streams.

The need for different types of transports led us to the concept of endpoints, analogous to electrical terminals, where transports connect to application programs. We saw cases where a program might need more than one endpoint, each carrying a separate message stream.

We introduced content matchers as a mechanism for selecting messages from a message stream, rather than addressing messages to specific destinations.

The need for one-to-one messaging led us to the concept of inboxes. We saw how cooperating applications could use inboxes to communicate in a request/reply pattern.

We noted that message streams and subscriber interest are transient, but sometimes we need message streams that endure, which led us to persistence stores. We saw how stores back endpoints, or more generally, networks of endpoints. We also saw how shared durables can distribute a message stream among a set of cooperating subscribers. We saw how last-value durables can divide a message stream according to the value of a key field.

We saw that self-describing messages are easy to use, but less efficient than defining message formats.

Viewing an application as a collection of endpoints, we revisited the separation of responsibilities between programmer and administrator. The need for variant communication schemes, depending on execution environment, led to the concept of application instances.

Finally, the need for separate communication fabrics led us to the concept of a realm.
From Concepts to Usage

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

As we noted at the outset, the terms we have defined along the way 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 API. For more information, see *TIBCO FTL Development* and the API reference material.

- 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 configuration model. For more information, see *TIBCO FTL Administration*. 