



# **TIBCO FTL® - Enterprise Edition**

## **Getting Started**

Version 7.0.1 | December 2024

# Contents

---

<b>Contents</b>	<b>2</b>
<b>About this Product</b>	<b>3</b>
<b>Getting Started</b>	<b>4</b>
Setting Up the FTL Environment	4
Starting the FTL Server and Realm Service	5
<b>Running the Sample Applications</b>	<b>7</b>
Publish-Subscribe example	7
Persistent Publish-Subscribe Example	9
Queuing	13
Request/Reply	18
Performance Measurement (Peer to Peer)	20
Performance Measurement (Persistence)	25
<b>Next Steps</b>	<b>30</b>
<b>Command Reference</b>	<b>31</b>
<b>TIBCO Documentation and Support Services</b>	<b>32</b>
<b>Legal and Third-Party Notices</b>	<b>34</b>

# 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](http://www.tibco.com).

# Getting Started

---

Use this document to get started using FTL. In this guide, you use your own hardware and the samples provided to:

- Send and receive messages
- Use persistence for guaranteed message delivery
- Demonstrate queuing and request/reply
- Collect performance statistics



**Tip:** For an overview of FTL and its functions, see the FTL [Concepts](#) guide.

## Setting Up the FTL Environment

Initialize the FTL environment in at least two terminals.

1. Open the first terminal.
2. To run the setup script from the FTL package, navigate to the samples directory in the FTL samples directory.
  - Linux and macOS in `/opt/tibco/ftl/<n.n>/samples`: Include the preceding `'.'` to ensure the environment variables are set in your current shell.

```
$ . ./setup
```

- Windows, run the setup batch file in the `C:\tibco\ftl\<n.n>\samples` folder:

```
>setup
```

3. Repeat these steps for each terminal.

# Starting the FTL Server and Realm Service

The FTL server needs to be running before you can run applications. It provides services including the FTL realm service. The realm configuration defines the applications and connectivity parameters used to connect FTL-based applications for message flow. All cooperating FTL applications are part of a specific FTL realm.

To run the sample applications in this guide, the FTL software download includes pre-defined configurations for the FTL server and the FTL realm. As you gain more experience with TIBCO FTL, you can modify the configuration to try other settings and options (such as using different transports to connect applications).



**Important:** The configuration provided is not appropriate for a production environment.

## Start the FTL Server

The command starts a single, non-redundant FTL server running (`ftls_1`) on the local system at a specific port (`localhost port 8080`). The supplied `ftlstart` script supports either one or three servers (`ftls_1`, `ftls_2`, and `ftls_3`), so be sure to use `ftls_1` for this example.

The `ftlstart` script also initializes the FTL realm configuring using `tibrealm.json` from the `samples/scripts` directory. This realm configuration is used with the performance samples later.

1. Load the configuration. You can use any port available for the host and the sample applications. From `samples/scripts` run:

- Linux and macOS:

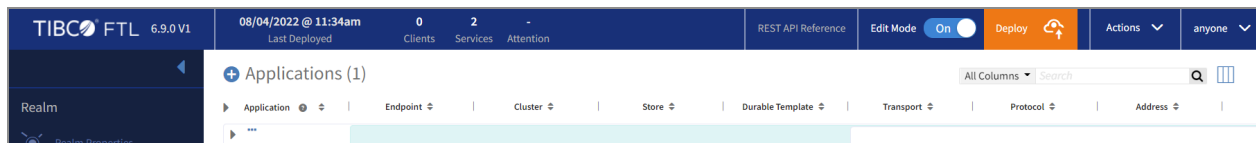
```
$ ./ftlstart ftls_1@localhost:8080
```

- Windows:

```
>ftlstart ftls_1@localhost: 8080
```

2. Confirm a message like this is returned:  
FTL Server configuration file has been saved as `<filename>.yaml`.

You can see the FTL UI by going to <http://localhost:8080>. If you see the FTL log in screen, accept the **User ID** anyone and click **SIGN IN**.



**Important:** If you do not see a log in screen, check the log file `ftls_1.log` at `$HOME/ftl-server/ftls_1`. If the port is not available to you, you see the message: Address already in use.

# Running the Sample Applications

---

This guide covers several common messaging scenarios:

- [Publish-Subscribe example](#)
- [Persistent Publish-Subscribe Example](#)
- [Queuing](#)
- [Request/Reply](#)
- [Performance Measurement \(Peer to Peer\)](#) (messaging throughput and latency between two applications)
- [Performance Measurement \(Persistence\)](#)

## Port Designations

If you change the port, which the FTL server is using, change the port numbers in the example commands to match.

If you are running more than one FTL server, you can do one of the following:

- Specify any one of the host:port values for the FTL servers on the command line
- Specify all of the host:port values on the command line separated by the “|” character, for example:  
`localhost:8080|localhost:9090|localhost:10100.`  
The application connects to a FTL server automatically, and falls over to another server in the event of no response or a disconnection.

## Publish-Subscribe example

A subscriber receives any associated publisher messages as they are published. If a publisher publishes a message and there are no associated subscribers (or a specific subscriber has not yet been started), the messages are not received by the subscribers. Messages are not stored as they are in the [Persistent Publish-Subscribe Example](#) in the next section. For more details, see the FTL Administration guide [Publish-Subscribe Models](#).

Use the `tibftlsend` sample application as a publisher to send messages that are received by the subscriber in the `tibftlrecv` sample application. Each sample application sends or receives a single message then exits. The applications are solid, well structured, and well-documented examples to start your own development from. The examples demonstrate good practices in areas like error and exception handling.



**Important:** These and other sample applications are not meant to be used in production.

Before you start, open two terminals. Make sure you have run the setup command and started the FTL server. See the [Command Reference](#).

Perform these steps:

1. To receive messages, from `samples/bin`, run:

- In Linux and macOS, the command is:

```
$. /tibftlrecv localhost:8080
```

- In Windows, the command is:

```
>tibftlrecv localhost:8080
```

The host is ready to receive messages sent by the publisher.

```
C:\tibco\ftl\6.9\samples\bin>tibftlrecv localhost:8080
#
# tibftlrecv
# (FTL) TIBCO FTL Version 6.9.0 V1
#
# Client name tibftlrecv_42944
#
Waiting for message(s)
```

2. In the other terminal, from `samples/bin`, run:

- In Linux and macOS, the command is:

```
$. ./tibftlsend localhost:8080
```

- In Windows, the command is:



```
>tibftlsend localhost:8080
```

## Check Your Result

In the terminal running `tibftlsend`, you will see the following output:

```
#
# tibftlsend
# (FTL) TIBCO FTL Version <n.n>V<n>
#
# Client name tibftlsend_40314
#
Sending 'hello world' message
```

In the terminal running `tibftlrecv`, you'll see the following output which is a string dump of the message:

```
#
Received message
{string:type="hello", string:message="hello world earth"}
```

In many of these examples, you can see some text in the output such as:

```
Client name tibftlxxxx #####
```

FTL dynamically generates these human-readable strings to identify each client in an FTL messaging realm. If you use the status display capabilities of the FTL user interface, you will see these numbers displayed as the **Client Label**. For details on Client IDs and client labels, see [Client Status Details](#). In this case, there are two fields, `type` and `message`, both of type `string`. The use of typed and named message fields allows content-based addressing using [Content Matchers](#).

## Persistent Publish-Subscribe Example

Persistence is the potential to store a message after sending it and before it is received so off line subscribers can consume it at a later time. Publishers and subscribers are completely decoupled. Subscribers can recover messages which may have gotten lost between the publisher and the subscriber due to network failure, resource issues along the

delivery path, or any other failure. For details about persistence, see the FTL Concept guide, [Persistence: Stores and Durables](#).

The sample programs from `samples\bin\advanced` are `tibsendex` (the publisher) and `tibrecvex` (the subscriber).

The following command line options are used for persistent message delivery:

- `-c` option: Count of messages informing `tibrecvex` (subscriber) to exit after receiving `n` messages or `tibsendex` (publisher) to send `n` messages.
- `-d` option: Durable name that specifies a durable name, which is used to express interest in a particular message stream by a subscriber.
- `-e` option: Endpoint to use that has been configured for persistence.

Perform these steps:

1. Open two terminals. Make sure you have run the setup command and started the FTL server. See the [Command Reference](#).
2. In a terminal, start the `tibrecvex` sample application located in the `/samples/bin/advanced` directory, using the `-c`, `-d`, and `-e` options:

- Linux and macOS:

```
$. /tibrecvex -a default -c 10 -d myDr -e persistent  
localhost:8080
```

- Windows:

```
>tibrecvex -a default -c 10 -d myDr -e persistent  
localhost:8080
```

3. In the other terminal, start the `tibsendex` sample application located in the `/samples/bin/advanced` directory using the `-c` and `-e` options.

- Linux and macOS:

```
$. /tibsendex -c 20 -a default -e persistent localhost:8080
```

- Windows:

```
>tibsendex -c 20 -a default -e persistent localhost:8080
```

## Check Your Result

You should see the `tibrecvex` application receive the first 10 messages that are sent and then exit similar to the output below.

```
#
# tibrecvex
#
# TIBCO FTL Version <n.n> V<n>
#
Invoked as: tibrecvex -a default -c 10 -d myDr -e persistent
localhost:8080
waiting for message(s)
received message 1
message:
type long: 1
type string: 'tibsend-persistent'
type opaque: size 6 data: 'opaque'
received message 2
message:
type long: 2
type string: 'tibsend-persistent'
type opaque: size 6 data: 'opaque'
received message 3
message:
type long: 3
type string: 'tibsend-persistent'
type opaque: size 6 data: 'opaque'
received message 4
message:
type long: 4
...
received message 8
message:
type long: 8
type string: 'tibsend-persistent'
type opaque: size 6 data: 'opaque'
received message 9
message:
type long: 9
type string: 'tibsend-persistent'
type opaque: size 6 data: 'opaque'
received message 10
```

```
message:
type long: 10
type string: 'tibsend-persist'
type opaque: size 6 data: 'opaque'
```

The tibsendex application messages contain three fields:

- type long: Containing a monotonically increasing integer value
- type string: Containing the fixed string tibsend-persistentendpoint
- type opaque: Containing 6 bytes corresponding to the ASCII characters spelling out “opaque”

The tibsendex application sent all 20 messages.

```
#
# tibsendex
#
# TIBCO FTL Version <n.n>    V<n>
#
Invoked as: /tibsendex -c 20 -a default -e persistent localhost:8080
localhost:8080
sending message 1
sending message 2
sending message 3
sending message 4
sending message 5
sending message 6
sending message 7
sending message 8
sending message 9
sending message 10
sending message 11
sending message 12
sending message 13
sending message 14
sending message 15
sending message 16
sending message 17
sending message 18
sending message 19
sending message 20
```

## Get the Next 10 Messages

The `tibsendex` application sent 20 messages. The 10 additional messages, numbered 11 through 20 are retained by an FTL store.

1. Run the `tibrecvex` application again using the same command line.
2. Check the result: You should receive the remaining 10 messages, even if the `tibsendex` application has completed sending all 20 messages and has exited.

If you see messages come in quickly at first and then slower, this receiver is getting previously sent messages quickly and, once those are delivered, subsequent messages are received in real-time based on the cadence of the publisher.

## Failure and Restart

In a production scenario, persistence guards against system, network, or FTL server failures. Multiple FTL servers are run across multiple systems to provide redundancy for the persistence service so failures do not compromise your message delivery.

If you want to see how the persistence service ensures message delivery across subscriber failures and restarts:

1. Increase the message count to a significantly higher number to give you time to stop `tibrecvex` before all of the messages have been delivered.
2. Stop `tibrecvex` before all the messages are received.
3. Restart it to receive the remaining messages.

## Queuing

Queuing and load balancing are important for an efficient messaging system.

A set of publishing applications sends messages to a queue from which subscribing applications retrieve messages. Run multiple subscribers to increase the throughput of your messaging system by increasing message consumption. Each subscriber acknowledges each message that it has finished processing. This causes the message to be removed from the queue. If an application stops or fails to acknowledge a message it has retrieved from the queue, the message is made available for other applications to retrieve. This ensures that all messages are processed. For more details, see [Event Queue and Messaging Dispatch](#).

In this example, you use these commands in `samples\bin`:

- `tibftlshared-producer`: To write messages to the queue (the publisher)
- `tibftlshared-consumer`: To read the messages in the queue (the subscriber)

Using a single producer and single consumer application, the consumer receives every message the producer sends in the order it was sent.

Perform these steps:

1. Open three terminals. In each, make sure you have run the `setup` command and started the FTL server. See the [Command Reference](#).
2. In one of the terminals, start the producer by running the `tibftlshared-producer` application located in the `samples/bin` directory.

- Linux and macOS:

```
$/tibftlshared-producer localhost:8080
```

- Windows:

```
>tibftlshared-producer localhost:8080
```

The result shows the producer started publishing messages, each with an application-generated sequence number.

```
$ tibftlshared-producer localhost:8080
#
# tibftlshared-producer
# (FTL) TIBCO FTL Version <n.n> V<n>
#
# Client name tibftlshared-producer_58777
#
Sending message sequence 0
Sending message sequence 1
Sending message sequence 2
Sending message sequence 3
Sending message sequence 4
```

3. In another terminal, start the consumer by running the `tibftlshared-consumer` application located in the `samples/bin` directory:

- Linux and macOS:

```
$ ./tibftlshared-consumer localhost:8080
```

- Windows:

```
>tibftlshared-consumer localhost:8080
```

The result shows a message that has two fields:

- string containing the application name of the producer who sent this message
- long containing the sequence number from that specific producer

Because only one producer and one consumer is running, the sequence numbers are in order.

```
$ tibftlshared-consumer localhost:8080
#
# tibftlshared-consumer
# (FTL) TIBCO FTL Version <n.n> V<n>
#
# Client name tibftlshared-consumer_58804
#
Waiting for message(s)
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=0}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=1}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=2}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=3}
```

Received message

```
{string:sender="tibftlshared-producer_58777", long:sequence=4}
```

4. In the third terminal, start the second consumer so the output of the two consumers doesn't get interspersed.

## Check Your Result

Typical output from each consumer follows. In this example, each consumer requests a single message, reads from the same queue, and is serviced in a round-robin fashion. The consumers dump the message as a string and acknowledge it. This results in each consumer receiving every other message.

### *Consumer 1*

```
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=219}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=221}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=223}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=225}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=227}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=229}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=231}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=233}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=235}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=237}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=239}
```

### *Consumer 2*

```
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=220}
Received message
```



```

{string:sender="tibftlshared-producer_58777", long:sequence=222}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=224}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=226}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=228}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=230}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=232}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=234}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=236}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=238}
Received message
{string:sender="tibftlshared-producer_58777", long:sequence=240}

```

Typically, you do not see the alternating cadence in the example. A consumer controls how many messages it processes at a time. It can take different amounts of time for a consumer to process and acknowledge the messages.

## Adding More Producers

The following output is from one of three consumers running with messages coming from two producers. The two producers are identified by their different application names:

- tibftlshared-producer\_59154
- tibftlshared-producer\_59163

```

Received message
{string:sender="tibftlshared-producer_59154", long:sequence=258}
Received message
{string:sender="tibftlshared-producer_59163", long:sequence=148}
Received message
{string:sender="tibftlshared-producer_59154", long:sequence=261}
Received message
{string:sender="tibftlshared-producer_59163", long:sequence=151}
Received message
{string:sender="tibftlshared-producer_59154", long:sequence=264}
Received message

```

```
{string:sender="tibftlshared-producer_59163", long:sequence=154}
Received message
{string:sender="tibftlshared-producer_59154", long:sequence=267}
Received message
{string:sender="tibftlshared-producer_59163", long:sequence=157}
```

You can try different combinations of producers and consumers. Stop and restart them in different scenarios and see how the delivery cadence changes.

## Request/Reply

In a request/reply scenario, one application sends a message to another and expects a reply message such as a response code, query result, or receipt confirmation. For more details, see the FTL Development guide, [Request/Reply](#) and the FTL Concepts Guide, [Request/Reply with Inbox Token](#).

The `tibftlrequest` and `tibftlreply` applications are used in this example.

Perform these steps:

1. Open two terminals. Make sure you have run the setup command and started the FTL server. See the [Command Reference](#).
2. In one of the terminals, start the `tibftlreply` application which is in the `samples\bin` directory. The start command and output follows.

- Linux and macOS:

```
$/tibftlreply localhost:8080
```

- Windows:

```
>tibftlreply localhost:8080
```

3. Check the reply.

```
#
# tibftlreply
# (FTL) TIBCO FTL Version <n.n> V<n>
```

```
#
# Client name tibftlreply_44316
#
Waiting for message(s)
```

4. In the other terminal, start `tibftlrequest` which is in the `samples\bin` directory to send a request to `tibftlreply`.

- Linux and macOS:

```
$/tibftlrequest localhost:8080
```

- Windows:

```
>tibftlrequest localhost:8080
```

## Check Your Result

The reply returns quickly and you'll see output like the following.

### Requester

The `Sending request message:` line shows the message `tibftlrequest` sent to `tibftlreply`. Two string fields are defined, including a field type with the value `request`. After the replier responds the `Reply message received` line displays the response.

```
#
# tibftlrequest
# (FTL) TIBCO FTL Version <n.n> V<n>
#
# Client name tibftlrequest_44319
#
Sending request message: {string:type="request", string:client_
name="tibftlrequest_44319"}
Reply message received: {string:type="reply", string:client_
name="tibftlreply_44316"}
```

### Replier

The `tibftlreply` application looked for a field called `type` with a value of `request`. When the request is received, the application returns a message with the `type` field value `reply` and the `client_name` field with the client identifier of the `tibftlreply` application.

```
Received request: {string:type="request", string:client_name="tibftlrequest_110685"}  
Reply sent: {string:type="reply", string:client_name="tibftlreply_110663"}
```

A request can contain any number of fields with any type of content needed by an application.

## Performance Measurement (Peer to Peer)

FTL has sample applications to measure:

- Latency: The time delay between message generation and delivery
- Throughput: The number of messages processed in a given time

The samples can be run with any of the transports including shared memory, TCP, multicast, RUDP, and so on. By default, they run using the shared memory transport for best performance and least configuration. For more details, see the FTL Administration guide, [Transport Protocol Types](#).

### CPU Affinity Considerations

The applications discussed below may be run without considering CPU affinity. However, for best performance, the sender and receiver applications can be assigned to specific processors to improve cache performance and limit context switching.

To minimize latency, the latency sender and receiver are designed to receive messages from the transport and dispatch message callbacks from a single thread. This is accomplished through the use of inline event queues and publishers. For more information, see the FTL Development guide, [Inline Mode](#). As a rough rule of thumb, the latency sender and receiver may be assigned to one processor each.

To maximize throughput, the throughput receiver is designed to receive messages from the transport on one thread, and dispatch message callbacks from a second thread. (This is accomplished through the use of a non-inline event queue.) The sender application has a

dedicated sending thread. As a rough rule of thumb, the throughput sender and receiver may be assigned to two processors each.

In addition, when using the shared memory transport (which is the default in the sample realm configuration, `tibrealm.json`), the sender and receiver applications should be assigned to processors in a way that maximizes sharing of cache memory.

The following is just one example on Linux. Some experimentation will be required to achieve the best performance on a specific system.

```
taskset -c 1 ./tiblatrecv
taskset -c 2 ./tiblatsend

taskset -c 1,2 ./tibthrurecv
taskset -c 3,4 ./tibthrusend
```

## Latency Test

The `tiblatrecv` and `tiblatsend` applications work together to provide a latency value calculated by averaging the time needed to send 5 million messages from the sender to the receiver.

Perform these steps:

1. Open two terminals. Make sure you have run the setup command and started the FTL server. See the [Command Reference](#). Note that the `ftlstart` script automatically loads the realm configuration needed for this test.
2. From `samples\bin\advanced` in one terminal run:

- Linux and macOS:

```
$/tiblatrecv localhost:8080
```

- Windows:

```
>tiblatrecv localhost:8080
```

3. From `samples\bin\advanced` in the other terminal run:

- Linux and macOS:

```
$./tiblatsend localhost:8080
```

- Windows:

```
>tiblatsend localhost:8080
```

The `tiblatsend` terminal displays a summary of the total elapsed message delivery time, the number of messages sent, and an average representing the per message latency.

```
#
# tiblatsend
#
# TIBCO FTL Version <n.n.n>
Invoked as: tiblatsend
Calibrating tsc timer... done.
CPU speed estimated to be 2.20E+09 Hz
Sending 5000000 messages with payload size 16
Sampling latency every 5000000 messages.
Total time: 4.57E+00 sec. for 5000000 messages
One way latency: 456.62E-09 sec.
```

You can use these options with syntax for your platform:

- `count` to control the number of messages sent.
- `size` to change the size of the messages sent.
- `help` to see the complete set of command line options.

Try different values to see the impact on latency using your hardware.

## Throughput Test

Throughput measures the number of messages processed between applications in a given time. Larger and fewer messages typically reduces the impact of per-message overhead on messaging throughput.

The `tibthrurecv` and `tibthrusend` applications work together to provide a throughput value by measuring the time needed to send 5 million messages from the sender to the receiver.

Perform these steps:

1. Open two terminal and make sure you have run the `setup` command and started the FTL server. See the [Command Reference](#), if necessary. Note that the `ftlstart` script automatically loads the realm configuration needed for this test.
2. From `samples\bin\advanced` in one terminal run:

- Linux and macOS:

```
$. /tibthrurecv localhost:8080
```

- Windows:

```
>tibthrurecv localhost:8080
```

3. From `samples\bin\advanced` in the other terminal run:

- Linux and macOS:

```
$. /tibthrusend localhost:8080
```

- Windows:

```
>tibthrusend localhost:8080
```

The `tibthrusend` terminal displays a summary of:

- The number of messages sent (together with the number of batches and the batch size which can be set using the `--batchsize` command line option)
- The total send and receive times
- The aggregate message send and receive rates

Typical output from `tibthrusend` follows. Based on this run:

- The sender sent all 5 million messages in 2.92 seconds at an average rate of 1.72

million messages per second.

- The receiver received these messages in about the same amount of time at an average rate of 27 million bytes per second.

```
#
# tibthrusend
#
# TIBCO FTL Version <n.n.n>
#
Invoked as: tibthrusend http://localhost:8080
Calibrating tsc timer... done.
CPU speed estimated to be 2.20E+09 Hz

Sender Report:
Requested 5000000 messages.
Sending 5000000 messages (50000 batches of 100) with payload size 16
Sent 5.00E+06 messages in 2.92E+00 seconds. (1.71E+06 msgs/sec)

Receiver Report:
Received 5.00E+06 messages in 2.92E+00 seconds. (1.71E+06 messages per
second)
Received 80.00E+06 bytes in 2.92E+00 seconds. (27.38E+06 bytes per
second)
Messages per callback: min/max/avg/dev: 1.00E+00 / 100.00E+00 /
48.34E+00 / 29.37E+00
```

Performance is impacted by the performance and load of the system you are running on.

You can use these options with syntax for your platform:

- `count` to control the number of messages sent using `tibthrusend`. Keep these values relatively large in order to get representative results.
- `size` to specify the size of the messages in bytes. The default size of 16 bytes is a relatively small message and might not be representative of what your applications would be using.



- `help` to see the complete set of command line options for both `tibthrusend` and `tibthrurecv`.

Try different values for `--size` to see its impact on throughput using your hardware.

## Performance Measurement (Persistence)

FTL has sample applications to measure:

- Latency: The time delay between message generation and delivery
- Throughput: The number of messages processed in a given time

The samples can be run with any persistence store. For simplicity, the sample realm configuration, `tibrealm.json`, contains a definition of a persistence cluster with one persistence store and one persistence service, using DTCP transports. Since there is only one persistence service, this configuration does not provide fault tolerance or replication, and is not recommended for production use. See the `samples/yaml/persistence` directory for additional examples.

Persistence clusters require TCP-based transports. The DTCP transport provides maximum performance for persistence. For minimum latency it is possible to go a step further and set a receive spin limit for all persistence transports. However this requires dedicating significant CPU resources to the persistence service and all clients. Since this is not best for all systems, the sample realm configuration, `tibrealm.json` does not set a receive spin limit.

When configuring a persistence store, the strongest delivery assurance can be obtained by making the store replicated and using publisher mode `store_confirm_send`. The sample realm configuration, `tibrealm.json`, sets both these options. Configuring a store to be non-replicated, and/or to use publisher mode `store_send_noconfirm`, can offer better performance at the cost of weaker delivery guarantees. For more details, see the Administration guide, [Persistence: Stores and Durables](#).

**i Note:** When an application sends and receives messages on the same persistence store, by default the application will receive its own messages. (This is not true for peer-to-peer transports.) As a result, the sample applications used to measure performance in this section create content matchers to filter out locally generated messages.

## Latency Test

The `tiblatrecv1to1` and `tiblatsend1to1` applications work together to provide a latency value calculated by averaging the time needed to send 50 thousand messages from the sender to the receiver. If using a persistence store with publisher mode `store_confirm_send`, this includes the time required for the publisher to obtain a confirmation from the persistence service that the message is stored.

**Note:** Unlike the samples used to measure peer-to-peer latency in the previous section, these samples report the round-trip time (sender to receiver and back again). The one-way latency can be obtained by dividing the round-trip time by 2.

Perform these steps:

1. Open two terminals. Make sure you have run the setup command and started the FTL server. See the [Command Reference](#). Note that the `ftlstart` script automatically loads the realm configuration needed for this test.
2. From `samples\bin\advanced` in one terminal run:

- Linux and macOS:

```
$/tiblatrecv1to1 localhost:8080
```

- Windows:

```
>tiblatrecv1to1 localhost:8080
```

3. From `samples\bin\advanced` in the other terminal run:

- Linux and macOS:

```
$/tiblatsend1to1 localhost:8080
```

- Windows:

```
>tiblatsend1to1 localhost:8080
```

The `tiblatsend1to1` terminal displays a summary of the number of messages sent and an average representing the per message latency.

```
#
# tiblatsend1to1
#
# TIBCO FTL Version <n.n.n>
Invoked as: ./tiblatsend1to1 http://localhost:8080
Calibrating tsc timer... done.
CPU speed estimated to be 3.19E+09 Hz
Sending 50000 messages with payload size 16.
Round-trip time: 109.60E-06 seconds.
```

You can use these options with syntax for your platform:

- `count` to control the number of messages sent.
- `size` to change the size of the messages sent.
- `help` to see the complete set of command line options.

Try different values to see the impact on latency using your hardware.

## Throughput Test

Throughput measures the number of messages processed between applications in a given time. Sending larger and fewer messages typically reduces the impact of per-message overhead on messaging throughput.

The `tibthrurecv1to1` and `tibthrusend1to1` applications work together to provide a throughput value by measuring the time needed to send 500 thousand messages from the sender to the receiver.



**Note:** The sample realm configuration, `tibrealm.json`, defines a persistence store with publisher mode `store_confirm_send`, which allows the persistence service to regulate the send rate of the publisher. If using a persistence store with publisher mode `store_send_noconfirm`, the application must regulate the send rate. For this purpose both `tibthrurecv1to1` and `tibthrusend1to1` have a `"--flow-control"` command line option.

Perform these steps:

1. Open two terminal and make sure you have run the setup command and started the FTL server. See the [Command Reference](#), if necessary. Note that the ftlstart script automatically loads the realm configuration needed for this test.
2. From samples\bin\advanced in one terminal run:

- Linux and macOS:

```
$. /tibthrurecv1to1 localhost:8080
```

- Windows:

```
>tibthrurecv1to1 localhost:8080
```

3. From samples\bin\advanced in the other terminal run:

- Linux and macOS:

```
$. /tibthrusend1to1 localhost:8080
```

- Windows:

```
>tibthrusend1to1 localhost:8080
```

The tibthrusend1to1 terminal displays a summary of:

- The number of messages sent (together with the number of batches and the batch size which can be set using the --batchsize command line option)
- The aggregate message send rate

Typical output from tibthrusend1to1 follows.

```
#
# tibthrusend1to1
#
# TIBCO FTL Version <n.n.n>
#
Invoked as: ./tibthrusend1to1 http://localhost:8080
Calibrating tsc timer... done.
```

```

CPU speed estimated to be 3.19E+09 Hz
Sender Report:
Requested 500000 messages.
Sending 500000 messages (5000 batches of 100) with payload size 16
Sent 500.00E+03 messages in 2.16E+00 seconds. (231.57E+03 messages
per second)
Sent 8.00E+06 bytes in 2.16E+00 seconds. (3.71E+06 bytes per
second)

```

Performance is impacted by the performance and load of the system you are running on.

You can use these options with syntax for your platform:

- `count` to control the number of messages sent using `tibthrusend1to1`. Keep these values relatively large in order to get representative results.
- `size` to specify the size of the messages in bytes. The default size of 16 bytes is a relatively small message and might not be representative of what your applications would be using.
- `help` to see the complete set of command line options for both `tibthrusend1to1` and `tibthrusend1to1`.

## Additional Tests to Try

Try different values for `--size` to see its impact on throughput using your hardware.

If your system has the capacity, try setting a receive spin limit of 10 milliseconds for all persistence transports.

By default `tibthrusend1to1` uses the vectored send call, `tibPublisher_SendMessages`, which offers maximum performance. However if an application wishes to use the simple send call, `tibPublisher_Send`, the application may still obtain good performance by using the non-inline send policy, which allows the FTL library to send messages and receive confirmations from the persistence service in the background. The `tibthrusend1to1` application can demonstrate this with the `--single` command line option. For more details, see the Development guide, [Publisher Mode and Send Policy](#).

## Next Steps

---

The source code for the samples (and more) are included as part of the TIBCO FTL distribution in:

`samples/src/c`

`samples/src/dotnet (.NET)`

`samples/src/java`

`samples/golang/src/tibco.com/ftl-sample`

Feel free to review the source code, make modifications, rebuild the applications, and run the modified versions.

As a next step, use the [FTL Tutorials](#) for a step-by-step approach to building solid TIBCO FTL applications. You will use options and properties, exception handling, programming models, and the user interface for configuration and monitoring.

# Command Reference

---

You must run these commands before you use the sample applications. Replace <n.n> with your FTL version, such as 6.8.

## Windows Commands

### Setup the environment

```
C:\tibco\ftl\<n.n>\samples>setup.bat
```

### Start the FTL server

```
C:\tibco\ftl\<n.n>\samples\scripts>ftlstart ftls_1@localhost:8080
```

## Linux/macOS

### Setup the environment

```
/opt/tibco/ftl/<n.n>/samples$./setup
```

### Start the FTL server

```
/opt/tibco/ftl/<n.n>/samples/scripts$./ftlstart ftls_1@localhost:8080
```

# TIBCO Documentation and Support Services

---

For information about this product, you can read the documentation, contact TIBCO Support, and join [TIBCO Community](#).

## How to Access TIBCO Documentation

Documentation for TIBCO products is available on the [Product Documentation website](#), mainly in HTML and PDF formats.

The [Product Documentation website](#) is updated frequently and is more current than any other documentation included with the product.

## Product-Specific Documentation

Documentation for TIBCO FTL® - Enterprise Edition is available on the [TIBCO FTL® - Enterprise Edition Product Documentation](#) page.

## TIBCO eFTL™ Documentation Set

TIBCO eFTL software is documented separately. Administrators use the FTL server GUI to configure and monitor the eFTL service. For information about these GUI pages, see the documentation set for TIBCO eFTL software.

## How to Contact Support for TIBCO Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our [product Support website](#).
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the [product Support website](#). If you do not have a username, you can request one by clicking **Register** on the website.



## How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the [TIBCO Ideas Portal](#). For a free registration, go to [TIBCO Community](#).

# Legal and Third-Party Notices

---

SOME CLOUD SOFTWARE GROUP, INC. (“CLOUD SG”) SOFTWARE AND CLOUD SERVICES EMBED, BUNDLE, OR OTHERWISE INCLUDE OTHER SOFTWARE, INCLUDING OTHER CLOUD SG SOFTWARE (COLLECTIVELY, “INCLUDED SOFTWARE”). USE OF INCLUDED SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED CLOUD SG SOFTWARE AND/OR CLOUD SERVICES. THE INCLUDED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER CLOUD SG SOFTWARE AND/OR CLOUD SERVICES OR FOR ANY OTHER PURPOSE.

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

This document is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of Cloud Software Group, Inc.

TIBCO, the TIBCO logo, the TIBCO O logo, FTL, eFTL, and Rendezvous are either registered trademarks or trademarks of Cloud Software Group, Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. You acknowledge that all rights to these third party marks are the exclusive property of their respective owners. Please refer to Cloud SG’s Third Party Trademark Notices (<https://www.cloud.com/legal>) for more information.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: <https://scripts.sil.org/OFL>

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

Cloud SG software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the “readme” file for the availability of a specific version of Cloud SG software on a specific operating system platform.

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

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. CLOUD SG MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S), THE PROGRAM(S), AND/OR THE SERVICES DESCRIBED IN THIS DOCUMENT AT ANY TIME WITHOUT NOTICE.

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

This and other products of Cloud SG may be covered by registered patents. For details, please refer to the Virtual Patent Marking document located at <https://www.cloud.com/legal>.

Copyright © 2009-2024. Cloud Software Group, Inc. All Rights Reserved.