Enterprise Transport API Java Edition 3.9.0.L2

OPEN SOURCE PERFORMANCE TOOLS GUIDE

Document Version: 3.9.0.L2
Date of issue: May 2025
Document ID: ETAJ390L2PETOO.250

Contents

List of Figures

List of Tables

1 Introduction

1.1 About this Manual

This guide introduces the Enterprise Transport API Java Edition of the performance suite. It presents an overview of how the performance suite applications work with the LSEG Real-Time Distribution System, how the applications themselves work, and how the application tests are run. It also provides an overview of the basic concepts of writing performant Enterprise Transport API applications, as well as configuring both the applications and the Enterprise Transport API for optimal performance.

The authors include Enterprise Transport API architects and developers who encountered and resolved many of issues you might face. Several of its authors have designed, developed, and maintained the Enterprise Transport API product and other LSEG products which leverage it. As such, this document is concise and addresses realistic scenarios and use cases.

This guide documents the general design and usage of the tools provided for measuring the performance of the Enterprise Transport API Java Edition. It describes how features of the API are used to send and receive data with high throughput and low latency. This information applies both when the API is directly connected to itself as well as through LSEG Real-Time Distribution System components, such as the LSEG Real-Time Advanced Distribution Hub and LSEG Real-Time Advanced Distribution Server.

1.2 Audience

This document is written to help programmers using the Enterprise Transport API to take advantage of its features to achieve high throughput and low latency with their applications. The information detailed herein assumes that the reader is a user or a member of the programming staff involved in the design, code, and test phases for applications that will use the Transport API. It is assumed that you are familiar with the data types, operational characteristics, and user requirements of real-time data delivery networks, and that you have experience developing products using the Java programming language in a networked environment. It is assumed that the reader has read the *Transport API Java Developer's Guide* to have a basic familiarity with the Enterprise Transport API Transport and the interaction models of Open Message Model Consumers, Open Message Model Interactive Providers, and Open Message Model Non-Interactive Providers.

1.3 Programming Language

Enterprise Transport API Java is written to the Java language. All code samples in this document and all example applications provided with the product are written in Java.

1.4 Acronyms and Abbreviations

ACRONYM	DEFINITION
ADH	LSEG Real-Time Advanced Distribution Hub
ADS	LSEG Real-Time Advanced Distribution Server
API	Application Programming Interface
CPU	Central Processing Unit
DMM	Domain Message Model
EMA	Enterprise Message API
RTSDK	LSEG Real-Time Software Developer Kit
ETA	Enterprise Transport API
OMM	Open Message ModelS
RDM	Domain Model
RFA	Robust Foundation API
RTT	Round Trip Time; in reference to the API's latency monitoring feature.

Table 1: Acronyms and Abbreviations

1.5 References

- *Enterprise Transport API Java Edition Developers Guide*
- *Enterprise Transport API Java Edition LSEG Domain Model Usage Guide*
- *Enterprise Transport API Java Edition Value Added Components Developers Guide*

1.6 Documentation Feedback

While we make every effort to ensure the documentation is accurate and up-to-date, if you notice any errors, or would like to see more details on a particular topic, you have the following options:

- Send us your comments via email at ProductDocumentation@lseg.com.
- Add your comments to the PDF using Adobe's **Comment** feature. After adding your comments, submit the entire PDF to LSEG by clicking **Send File** in the **File** menu. Use the ProductDocumentation@lseg.com address.

1.7 Document Conventions

1.7.1 Typographic

This document uses the following types of conventions:

- Java classes, methods, in-line code snippets, and types are shown in **`Courier New`** font.
- Parameters, filenames, tools, utilities, and directories are shown in **Bold** font.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.
- Longer code examples are shown in Courier New font against a gray background. For example:

```
/* decode contents into the filter list object */
if ((retVal = filterList.decode(decIter)) >= CodecReturnCodes.SUCCESS)
{
    /* create single filter entry and reuse while decoding each entry */
    FilterEntry filterEntry = CodecFactory.createFilterEntry();
```

1.7.2 Diagrams

Diagrams that depict a component in a performance scenario use the following format. The grey box represents one physical machine, whereas blue or white boxes represent processes running on that machine.

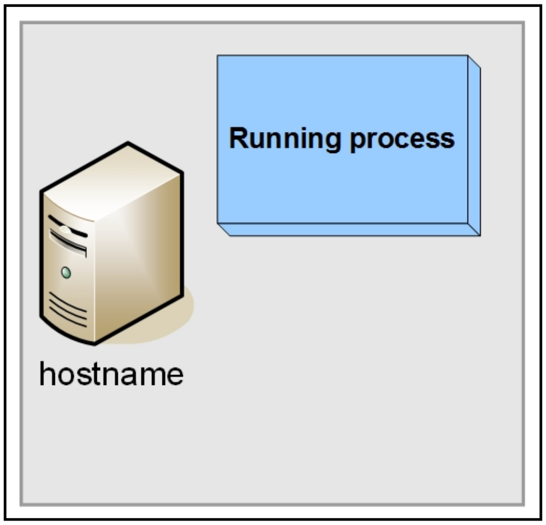

Figure 1. Running Performance Example and Host Notation

Diagrams that depict the interaction between components on a network use the following notation:

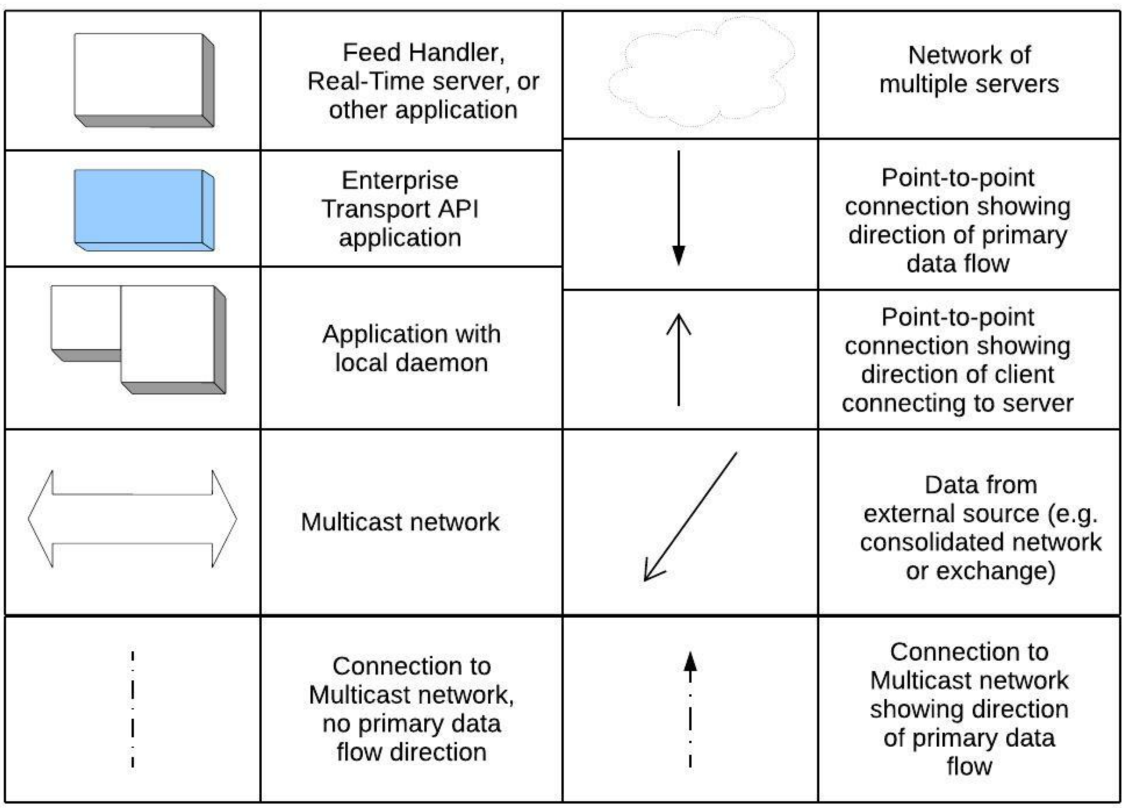

Figure 2. Network Diagram Notation

2 Open Source Performance Tool Suite Overview

2.1 Overview

The general idea behind the Open Source Performance Tool Suite is to provide a consistent set of platform test applications that look and behave consistently across the LSEG Real-Time APIs. The tool suite covers the various Open Message Model-based API products and allows LSEG's internal and external clients to compare latency and throughput trade-offs of the various APIs and their differing functionality sets.

LSEG Real-Time Distribution System also offers the tools **testclient** and **testserver** for performance testing, focusing on throughput, latency, and capacity of LSEG Real-Time Distribution System components. The tool suite focuses on what can be done with each API and is meant to compliment other platform tools.

All tools in the suite are provided as buildable open-source and demonstrate best practice and coding for performance with their respective APIs. Future releases of API products will expand on these tests to include other areas of functionality (e.g., batch requesting, etc.). Clients can run these tools to determine performance results for their own environments, recreate LSEG-released performance numbers generated using these tools, and modify the open source to tune and tweak applications to best match their end-to-end needs.

These performance tools can generate reports comparing performance across all API products.

2.2 The Enterprise Transport API Performance Tool Suite

The Enterprise Transport API Java-based suite consists of an Open Message Model consumer, Open Message Model interactive provider, and Open Message Model non-interactive provider. These applications showcase optimal Open Message Model content consumption and providing within the LSEG Real-Time Distribution System. Additionally, the Enterprise Transport API provides a transport-only performance example which you can use to measure the performance of the Enterprise Transport API transport handling opaque, non-Open Message Model content. Source code is provided for all performance tool examples, so you can determine how functionality is coded and modify applications to suit your specific needs.

Because applications from the LSEG Real-Time APIs are fully compatible and use similar methodologies, you can run them stand-alone within an API or mix them (e.g., a provider from Enterprise Transport API and a consumer from the Robust Foundation API).[1]

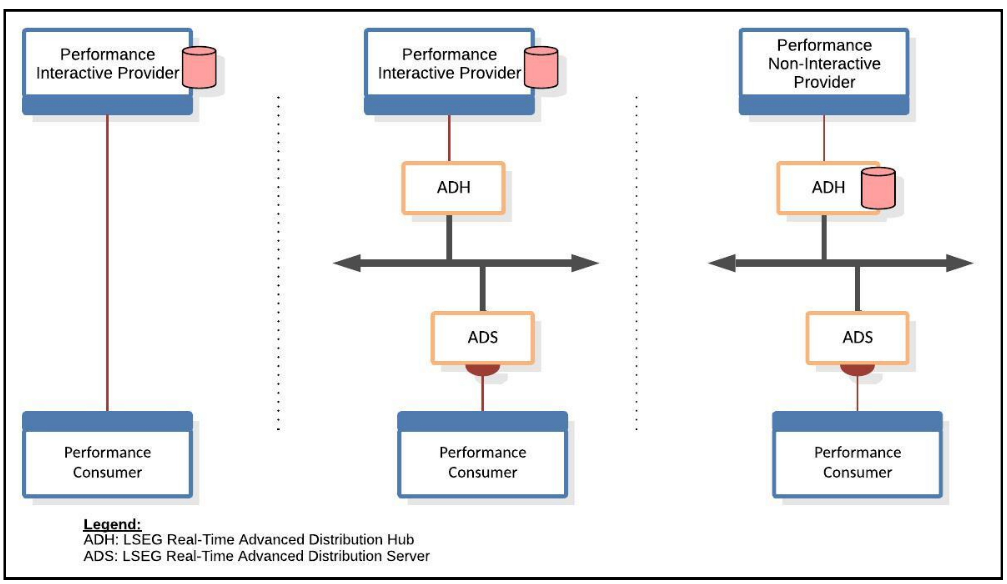

Figure 3. Three Connection Options for the Open Message Model-based Performance Tools

1. Tools from the Robust Foundation API C++ and Robust Foundation API Java APIs must be obtained from their respective distribution packages.

In a typical Open Message Model configuration, latency through the system is measured either one-way from a provider to consumer, or round-trip from a consumer, through the system, and back.[2] Latency information is encoded into a configurable number of update messages which are then distributed over the course of each second. The consumer receives update messages, and if the messages contain latency information, the consumer decodes them and measures the relative time taken to receive and process the message and its payload.

You can use the Enterprise Transport API transport-only performance tool (**TransportPerf**) to send non-Open Message Model content uni- or bi-directionally. Additionally, this application supports a “reflection” type mode used for round trip measurement. **TransportPerf** measures latency in all of these configurations, and records independent statistics for each instance of the application.

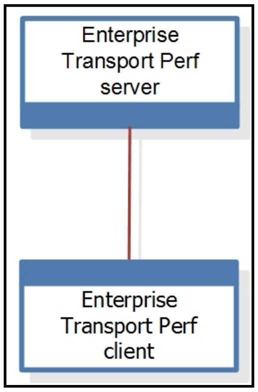

Figure 4. Enterprise Transport API Java Transport Perf

2. Without a microsecond-resolution synchronization of clocks across machines, the one-way measurement implies that the provider and consumer applications run on the same machine.

2.3 Package Contents

Performance examples are distributed as buildable source code with the Enterprise Transport API package. Each example is distributed in its own directory. The **PerfTools** root directory contains **build.gradle**. Each example project uses the **XML Pull Parser (XPP)** as a dependent library, which you must download from Maven Central.

For more information about examples and their operations, readers can refer to the appropriate application sections in this document. Readers can also refer to the **Javadoc** files and comments included in source.

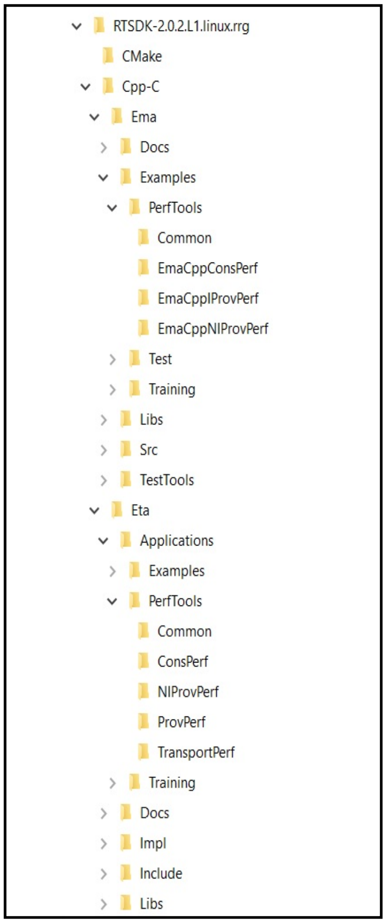

Figure 5. Directory Structure of the Performance Tools

2.3.1 XML Files

The **PerfTools** directory includes the following XML files:

- **350k.xml**: The list of 350,000 items loaded by the consumer (of content published by the non-interactive provider).
- **MsgFile.xml**: The default set of Open Message Model messages.

2.3.2 Building and Running

To build and run performance examples, use the following Gradle tasks: **`runETAPerfConsumer`**, **`runETAPerfNIProvider`**, **`runETAPerfProvider`**, and **`runETAPerfTransport`** (for details on running Gradle, refer to the *ESDK Java Edition Migration Guide*).

The **PerfTools** directory includes all necessary support files (**350k.xml**, **MsgData.xml**, **RDMFieldDictionary**, and **enumtype.def**).

2.4 What Gets Measured and Reported?

2.4.1 Latency

Each performance tool embeds timestamp information in its messages' payloads. The tool uses these timestamps to determine the overall time taken to send and process a message and its payload through the API and, where applicable, the LSEG Real-Time Distribution System. To ensure that the measurement captures end-to-end latency through the system, the timestamp is taken from the start of the sender's message and payload encoding, and is compared to the time at which the receiver completes its decoding of the message and payload.

When measuring performance, it is important to consider whether or not a particular component acts as a bottleneck on the system. Enterprise Transport API C applications and LSEG Real-Time Distribution System components provide higher throughput and lower latency than Enterprise Transport API Java and Robust Foundation API-based applications. In general, LSEG recommends that you use a Enterprise Transport API C performance tool to drive and calculate the performance of other non-Enterprise Transport API C-based performance tools. For example, if you want to test the performance of the Enterprise Transport API Java consumer, use the Enterprise Transport API C interactive or non-interactive provider to drive the publishing rather than an Enterprise Transport API Java providing application.

2.4.2 Throughput and Payload

These tools allow you to control the rate at which messages are sent as well as the content in each message. This allows you to measure throughput and latency using various rates and content, tailored to your specific needs.

2.4.3 Image Retrieval Time

The **ConsPerf** tool measures the overall time taken to receive a full set of images for items requested through the system. This time is measured from the start of the first request to the reception of the final expected image.

2.4.4 CPU and Memory Usage

Performance tools record a periodic sampling of CPU and Memory usage. This allows for consistent monitoring of resource use and can be used to determine the impact of various features and application modifications.

Java 7 (Oracle JDK) introduced **OperatingSystemMXBean** which is a platform-specific management interface for the operating system on which the Java virtual machine runs. The **`getCommittedVirtualMemorySize()`** method is used for memory usage and the **`getProcessCpuLoad()`** method is used for CPU usage.

- CPU Usage Calculation: The **`getProcessCpuLoad()`** method of **OperatingSystemMXBean** with the calculation **`getProcessCpuLoad()`** * 100 * N is used to determine CPU usage. N is the number of cores and is retrieved from the **`getAvailableProcessors()`** method of **OperatingSystemMXBean**.
- Memory Usage Calculation: The **`getCommittedVirtualMemorySize()`** method of **OperatingSystemMXBean** with the calculation **`getCommittedVirtualMemorySize()`** / 1048576.0 is used to determine memory usage.

2.4.5 Round Trip Time (RTT): Latency Monitoring

The **ProvPerf** tool can monitor an RTT message exchanged between the provider and consumer. To use RTT monitoring, you must enable the RTT feature on both the provider and the consumer. The RTT is measured using a separate type of **LoginMsg** and is separate from the type of latency monitoring described in Section 2.4.1.

2.5 Recorded Results and Output

The tools record their test results in the following files:

- Summary File
- Statistics File
- Latency File

2.5.1 Summary File

Each tool records the run's summary to a single file, including:

- The run's configuration
- Overall run results

If you use multiple threads, the file includes results for each thread as well as across all threads. For configuration details, refer to the chapter specific to the application that you use.

An example of recorded summary content for **ConsPerf** includes the average latency, update rate, and CPU/memory usage for the application's run time.

This summary information is output both to a file and to the console.

2.5.2 Statistics File

Each tool periodically records statistics relevant to that tool. For example, **ConsPerf** records:

- Latency statistics for updates (and, when so configured, posted content)
- Number of request messages sent and refresh messages received
- Number of update messages received
- Number of generic messages sent and received
- Latency statistics for generic messages (when so configured)

Each tool records these statistics on a per-thread basis. If the tool is configured to use multiple threads, the tool generates a file for each thread. For configuration details, refer to the chapter specific to the application that you use.

Each tool can configure statistics recording via the following options:

- **`writeStatsInterval`**: The interval (from 1 to *n*, in seconds) at which timed statistics are written to files and the console.
- **`noDisplayStats`**: Prevents writing periodic stats to console.

2.5.3 Latency File

You can configure **ConsPerf** and **TransportPerf** to record each individual latency measurement to a file. This is useful for creating plot or distribution graphs, ensuring that recorded latencies are consistent, and for troubleshooting purposes.

These latencies are recorded on a per-thread basis. If the tool is configured to use multiple threads, a file is generated for each thread.

For further details on configuring this behavior, refer to the chapter specific to the application that you use.

3 Latency Measurement Details

3.1 Time-slicing

All applications follow a similar model for controlling time: time is divided into small intervals, referred to as "ticks." During a run, each application has a main loop that runs an iteration once per tick. In this loop, the application performs some periodic action, and then waits until the next tick before starting the loop again.

For example, an application might observe the following loop:

1. Send out a burst of messages.
2. Wait until the time of the next tick. If network notification indicates that any connections have messages available, read them and continue waiting.

Applications can configure this rate using their respective **`-tickRate`** option. This determines how many ticks occur per second. For example, if you set the tick rate to 100, ticks occur at 10-millisecond intervals.

NOTE: **`-tickRate`** does not affect the Round Trip Time feature.

Applications adjust the message rate to fit the tick rate. For example, if an application wants to send 100,000 messages per second with a tick rate of 100 ticks per second, the application will send 1,000 messages per tick. Adjusting the tick rate affects the smoothness of message traffic by defining the amount of time between bursts:

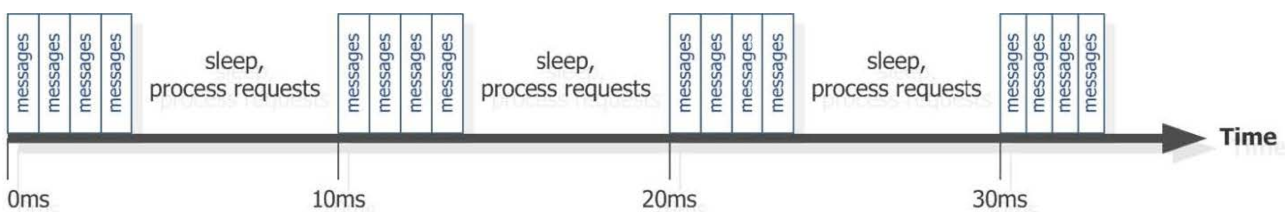

Figure 6. Time Slicing Algorithm

Depending on the tool, spare time in the tick might be used to perform other actions. For example, after **ProvPerf** or **NIProvPerf** sends an update burst, the remaining time is used to send outstanding refreshes:

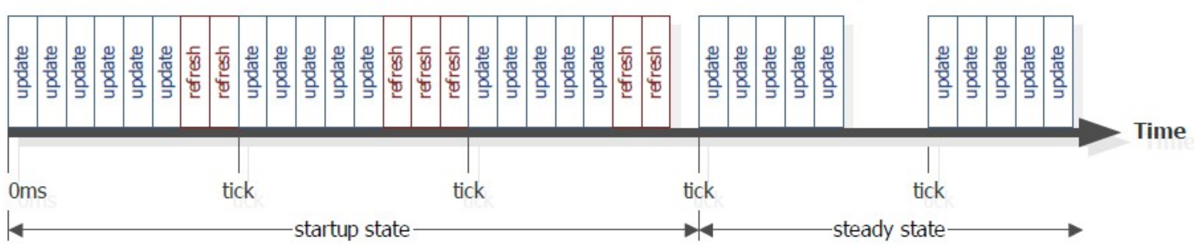

Figure 7. Refresh Publishing Algorithm

Applications always set tick times at fixed intervals as they progress, regardless of what the application does during the interval. For example, if the tick rate is 100 (i.e., 10 ms intervals), and the time of the previous tick was 40ms, then the times of the next ticks are 50 ms, 60 ms, etc... This helps maintain constant overall messaging rates: any irregularities in the timing of the current tick are corrected in subsequent ticks.

3.2 Latency

Latency is measured using timestamps embedded in the messages sent by each application. The receiving application compares this timestamp against the current time to determine the latency.

Each tool sends messages in bursts. To send timestamps, a message is randomly chosen from the message burst and the timestamp is embedded. When this message is received, the receiving application compares it to the current time to determine the latency.

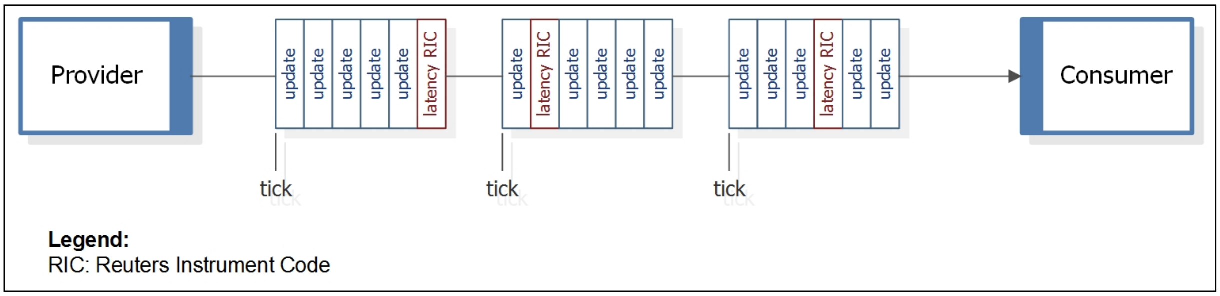

Figure 8. Latency Instrument Codes within a Tick

Timestamps are high-resolution and non-decreasing. Because the source of this time varies across platforms and might not be synchronized between multiple machines, update and generic message latency measurements require that the provider and consumer run on the same machine. Posting latency measurements do not require this, as **ConsPerf** generates both sending and receiving timestamps.

NOTE: Open Message Model performance tool timestamp information contains the number of microseconds since an epoch.[a] **TransportPerf** timestamps are provided with nanosecond granularity.

---

a. Windows uses `QueryPerformanceCounter()`, Solaris uses `gethrtime()`, Linux uses `clock_gettime()` with the monotonic clock. Enterprise Transport API Java uses **Stopwatch.GetTimestamp() / (Stopwatch.Frequency / 1000.0)** for microseconds.

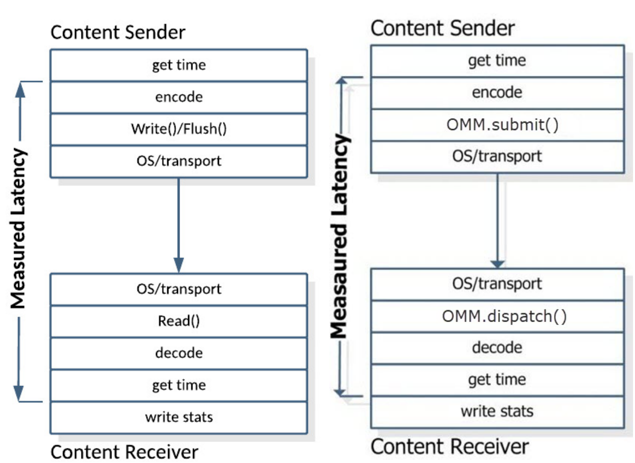

Figure 9. Timing Diagram for Latency Measurements

The standard latency measurement is initiated by the provider, which encodes a starting time into an update. This timestamp is included as a piece of data in the payload using a pre-determined latency Field IDentifier. On the consumer side, the application processes incoming updates and generic messages, decodes the payload, and looks for updates or generic messages which include the latency Field IDentifier (known as latency updates). After decoding a latency update or generic message, the consumer takes a second timestamp and compares the two, outputting the difference as the measured latency for that particular update or generic message.

3.3 Round Trip Latency

RTT is measured using timestamps that the provider application embeds in its messages. An application that receives a message with an RTT timestamp, responds according to its role:

- If application functions as a consumer: the application returns the message.
- If the application functions as a provider, the application calculates the RTT.

A provider application sends the RTT message within a predefined period of 5 seconds. After receiving an RTT message, the consumer immediately returns it to the provider application and executes any accompanying callback methods, if its RTT feature is enabled.

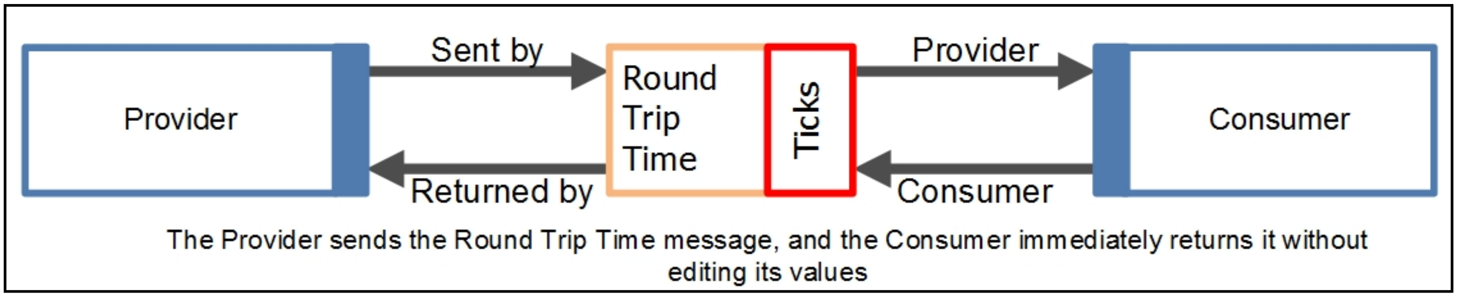

Figure 10. Round Trip Latency Process Diagram

Timestamps are high-resolution and non-decreasing. Because the source of this time varies across platforms and might not be synchronized between multiple machines, the RTT feature requires that the **ConsPerf** not modify RTT messages, instead immediately returning the message back it to provider. After receiving a returned message, the provider (**ProvPerf**) runs the round trip time calculation process.

The standard latency measurement is initiated by the provider, which encodes a starting time into a generic Login message. On the consumer side, the application processes the generic message and immediately returns it to the provider. On receipt, the provider decodes the message and executes the `LoginDomainCallback` method for RTT calculation. Afterwards, the provider reports its results to the console output.

4 Consumer Performance Tool

4.1 Overview

A typical Open Message Model consumer application requests content and processes responses to those requests. Thus, the performance consumer makes a large, configurable number of item requests and then processes refresh and update content corresponding to those requests. While processing, the performance consumer decodes all content and collects statistics regarding the count and latency of received messages.

The **ConsPerf** implements an OMM consumer using the Enterprise Transport API Java Edition. It connects to a provider (such as **ProvPerf** or LSEG Real-Time Distribution System), requests items, and processes the refresh and update messages it receives, calculating statistics such as update rate and latency. Additionally, the consumer can send post messages through the system at a configured rate, measuring the round-trip latency of posted content.

At startup, the consumer performs some administrative tasks, such as logging into the system, obtaining a source directory, and maybe requesting a dictionary. After the consumer is satisfied that the correct service is available and that the provider is accepting requests, the consumer begins requesting data. **ConsPerf** uses Enterprise Transport API Value-Add Administration Domain Representations to complete its start-up tasks. For more information, refer to the *Enterprise Transport API Developers Guide*.

4.2 Threading and Scaling

The Enterprise Transport API is designed to allow calls from multiple threads, such that applications can scale their work across multiple cores. Applications can leverage this feature by creating multiple threads to handle multiple connections through the Enterprise Transport API. As such, each application enables global locking when calling `rsslInitialize` `Transport.Initialize()`.

Configure **ConsPerf** for multiple threads using the `-threads` command-line option. When multiple threads are configured, each thread opens its own connection to the provider. **ConsPerf** divides its list of items among the threads (you can use the command line option, `-commonItemCount`, to request the same type and number of items on all connections).

The main thread monitors the other threads and collects and reports statistics from them.

4.2.1 Consumer Lifecycle

The lifecycle of **ConsPerf** is divided into the following sections:

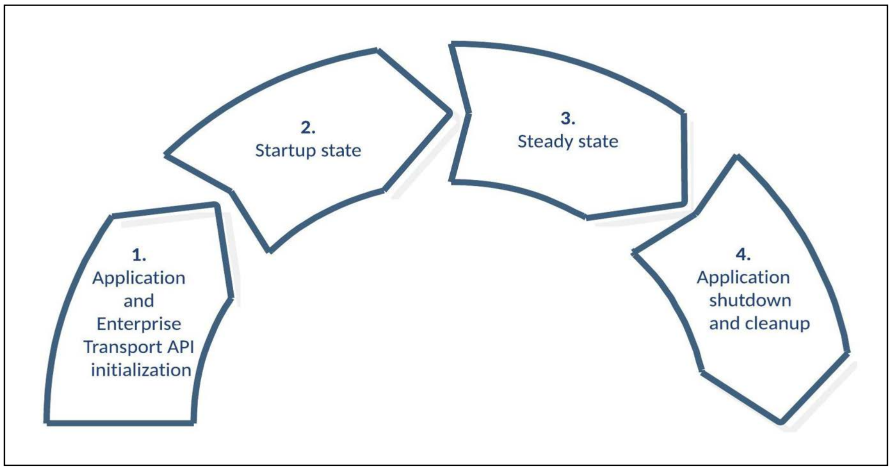

Figure 11. ConsPerf Lifecycle

1. Application and Enterprise Transport API Initialization.

   ConsPerf loads its configuration, initializes the Enterprise Transport API, loads its item list using the specified file, and starts the thread(s) which connect to the provider to perform the test.

   - The main thread periodically collects and writes statistics from the connection thread(s) until the test is over. All subsequent steps are performed by each thread.
   - Connection: the connection thread connects to the provider. If the connection fails, it continually attempts to reconnect until the connection succeeds. When the connection succeeds, the test begins and any subsequent disconnection ends the test.
   - Login: the connection thread provides its login requests and waits for the provider's response.
   - Directory: the connection thread opens a directory stream and searches for the configured service name.
   - Startup state: when the service is available, the "startup" phase of the performance measurement begins. During this phase, the connection thread continually performs the following actions:
     - Sends bursts of requests, until all desired items have been requested.
     - Reads from the transport, processing refresh, update, and generic message traffic from the provider.

   The "startup" phase continues until all items receive a refresh containing an Open/OK state. All latency statistics recorded up to this point are reported as "startup" statistics.

2. Steady state.

   The connection thread continually performs the following actions:

   - If configured for posting, the thread sends a burst of post messages.

- Reads from the transport, processing updates from the provider.
- If configured to do so, sends a burst of generic messages.

The “steady state” phase continues for the period of time specified in the command line. Latency statistics recorded during this phase are reported as “steady state” statistics.

3. Application shutdown and cleanup.

The connection thread disconnects and stops. The main thread collects all remaining information from the connection threads, cleans them up, and writes the final summary statistics. The main thread then uninitializes the Enterprise Transport API, any remaining resources, and exits.

4.2.2 Application Flow Diagram

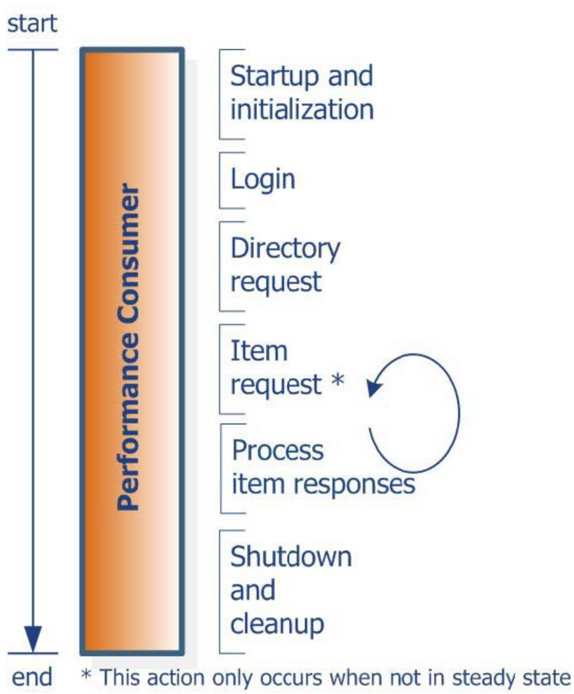

Figure 12. ConsPerf Application Flow

4.3 Latency Measurement

Provider applications encode the timestamp as part of their message payload. The initial timestamp is taken at the start of encoding, and added as field TIM_TRK_1 (**3902**) in Update messages and TIM_TRK_3 (3904) in Generic messages. When this field is detected, the **ConsPerf** gets the current time and computes the difference to measure latency.

When configured to do so via appropriate command line parameters, the Consumer application will encode timestamps as part of Generic messages payload. The timestamp is taken at the start of encoding and stored in the field TIM_TRK_3 (3904). The Performance Provider application can detect this field and calculate the latency by subtracting the received value from the current timestamp.

4.3.1 Consumer Latency

▶ Consumer Latency Measurement Sequence:

1. Read the message from the API (received via the underlying transport).
2. Decode the message.
3. Check whether the payload contains latency information, if so:
   - Get the current time (**t2**).
   - Calculate the difference between timestamps.
   - Store the result as part of the recorded output information.

4.3.2 Posting Latency

You can configure **ConsPerf** to send on-stream posts in which case the consumer periodically sends bursts of post messages for specified items in the item list file. You can also configure the tool to include latency information in its posts. When configured in this manner, **ConsPerf** adds latency information to random post messages. When the posted content returns on the stream, **ConsPerf** decodes the timestamp and measures the difference to determine posting latency.

▶ Posting Latency Measurement Sequence:

1. Get the current time (**t1**).
2. Obtain an output buffer using **`Channel.getBuffer()`**.
3. Encode the message, including the time (**t1**).
4. Pass the message to the API, which then passes it to the underlying transport.
5. When processing received content, check to see whether the payload contains latency information, if so:
   - Get the current time (**t2**).
   - Calculate the difference between timestamps.
   - Store the result in the recorded output information.

   The time at the start of encoding is encoded as a timestamp in the payload as field TIM_TRK_2 (**3903**). When the payload from the post returns from the platform, the consumer compares the timestamp to the current time to determine the posting latency.

4.4 ConsPerf Configuration Options

The following table describes available configuration options.

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-busyRead	None	Configures the application to continually read rather than use notifications.
-calcRWFJSONConversionLatency	(no argument)	Calculates the time spent on RWF to JSON conversion for WebSocket Transport and RWF. **NOTE:** This option works with WebSocket connection type only.
-commonItemCount	0	If multiple consumer threads are created (see **`-threads`**), each thread normally requests a unique set of items on its connection. This option specifies the number of common items to be requested by all connections.
-connType	socket	Specifies the consumer's connection type. **ConsPerf** supports the options: **`socket`**, **`websocket`**, and **`encrypted`**.
-delaySteadyStateCalc	0	Configures the time duration (in milliseconds), the consumer needs to wait to calculate the latency after receiving the last expected image.
-encryptedConnectionType	socket	Specifies the encrypted connection protocol. The Enterprise Transport API uses this option only when **`-connType`** is set to **`encrypted`**. Supported protocols include: • "**`socket`**" • "**`websocket`**" • "**`http`**"
-genericMsgLatencyRate	0	Controls the number of generic messages sent that contain latency information. This must be greater than the tick rate (see **`-tickRate`**) and less than to the total generic message rate (see **`-genericMsgRate`**).
-genericMsgRate	0	Controls the number of generic messages sent per second. This cannot be less than the tick rate, unless it is zero.
-h	localhost	Specifies the hostname to which the consumer connects.
-if	None	Configures **interfaceName** (a **ConnectOptions** parameter), which configures the network interface card (NIC) through which the consumer makes its connection. If your machine straddles networks, you can use this setting to force the consumer to use a particular network.
-itemCount	100000	Sets the total number of items requested by the consumer.
-itemFile	350k.xml	Configures the name of the item list file.
-keyfile	None	Specifies the keystore file directory path and name.
-keypasswd	None	Specifies the keystore password.
-latencyFile	None	Sets the name of the log file in which **ConsPerf** logs the latency retrieved from individual latency updates, generic messages, and posts. If a name is not specified, logging is disabled.
-maxOutputBuffer	50	Sets the maximum number of output buffers (configures **MaxOutputBuffers** in **BindOptions**). If you change **maxOutputBufs** from its default, **maxOutputBufs** must be equal to or greater than **outputBufs**.

Table 2: ConsPerf Configuration Options

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-msgFile	MsgData.xml	Configures the name of the file used by the consumer to determine the makeup of message payloads when posting (see **`-postingRate`**).
-noDisplayStats	(no argument)	Turns off printing statistics to the screen.
-outputBufs	5000	Configures **GuaranteedOutputBuffers** (a **ConnectOptions** parameter) which specifies the minimum guaranteed number of output buffers created for each channel. Setting this parameter to an appropriate size can aid performance if the consumer is posting messages (see **`-postingRate`**). You should configure enough buffers so that the provider does not run out of buffers while writing, but at the same time limit the number so as to conserve memory and optimize performance.
-p	14002	Specifies the port number to which the consumer connects.
-pl	""	Specifies a comma- or whitespace-delimited list of desired WebSocket sub-protocols in order of preference.
-postingLatencyRate	0	Controls the number of posts sent per second that contain latency information. This must be greater than the tick rate (see **`-tickRate`**) and less the total post message rate (see **`-postingRate`**).
-postingRate	0	Configures the consumer for posting. Sets the number of posting messages the consumer sends, per second. This cannot be less than the tick rate, unless it is zero (see **`-tickRate`**).
-primeJVM	(no argument)	Enables JVM priming. Accomplished by sending a snapshot request for all items before sending the actual streaming requests for the items. Latency measurements are only taken for updates so the refreshes from the snapshot requests are used to prime the JVM. This results in lower latency values in the start-up state.
-reactor	(no argument)	Use the Value Added Reactor instead of the Enterprise Transport API channel for sending and receiving. For details on Value Added Components, refer to the *Transport API Value Added Components Developers Guide*.
-recvBufSize	None	Sets the size (in bytes) of the system receive buffer. When unspecified, the OS setting is used.
-requestRate	500000	Sets the number of item requests sent (per second).
-sendBufSize	None	Sets the size (in bytes) of the system send buffer. When unspecified, the OS setting is used.
-securityProvider	SunJSSE	Allows to set security provider for encrypted connection. The library currently supports two options: • SunJSSE (default) • Conscrypt
-serviceName	DIRECT_FEED	Configures the name of the service used by the consumer to request items. The consumer begins requesting items whenever this service is found and appears ready.
-snapshot	(no argument)	Opens all items as snapshots, even if not specified in the item list file, and exits upon receiving all the solicited images. This is different from setting **`-steadyStateTime`** to **`0`** in that the requests are specifically made without the "STREAMING" **RequestMsg** flag.

Table 2: ConsPerf Configuration Options (Continued)

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-spTLSv1.2	(no argument)	Specifies for an encrypted connection to use TLS 1.2. Default has both 1.2 and 1.3 enabled.
-spTLSv1.3	(no argument)	Specifies for an encrypted connection to use TLS 1.3. Default has both 1.2 and 1.3 enabled.
-statsFile	ConsStats	Configures the base name that the consumer uses when writing its test statistics.
-steadyStateTime	300	Configures how long (in seconds) the consumer continues to run the test after receiving the last expected image. **`steadyStateTime`** has a second function: after beginning the test, if the consumer does not receive all expected images within this segment of time, the consumer times out. In this case, it exits and indicates that it did not reach steady state.
-summaryFile	ConsSummary.out	Configures the name of the file to which the consumer writes its test summary.
-threads	None	Sets the number of threads the consumer starts and the CPU core to which each thread binds. Each specified thread starts its own connection to the configured provider. Example: “1,3” creates two threads to make consumer connections, respectively bound to CPU cores 1 and 3.
-tickRate	1000	Sets the number of 'ticks' per second (the number of times per second the main loop of the consumer occurs). Adjusting the tick rate changes the size of request/post bursts; a higher tick rate results in smaller individual bursts, creating smoother traffic.
-tunnel	(no argument)	Opens a tunnel stream for exchanging basic messages. To use **`-tunnel`**, you must also use either **`-reactor`** or **`-watchlist`**. For details on creating and using tunnel streams, refer to the *Transport API Value Added Components Developers Guide*.
-tunnelAuth	(no argument)	Sets the consumer to enable authentication whenever opening tunnel streams.
-tunnelStreamBuffersUsed	(no argument)	Prints statistics for the tunnel stream’s buffers. This setting is disabled by default.
-tunnelStreamOutputBufs	5000	Sets the number of output tunnel buffers (i.e., this parameter configures **`guaranteedOutputBuffers`** in **`TunnelStreamOpenOptions`**).
-watchlist	(no argument)	Use the Value Added Reactor watchlist instead of the Enterprise Transport API channel for sending and receiving. For details on Value Added Components, refer to the *Transport API Value Added Components Developers Guide*.
-writeStatsInterval	5	Configures the frequency (in seconds) at which statistics are printed to the screen and statistics file.

Table 2: ConsPerf Configuration Options (Continued)

4.5 Input

ConsPerf requires the following files:

- Dictionary files to encode/validate fields in the message data. **RDMFieldDictionary** and **enumtype.def** are provided with the package.
- An XML file that describes refresh messages, update messages and generic messages. The package includes a default file (**350k.xml**).

For more details on input file information, refer to Chapter 9, Input File Details.

4.6 Output

ConsPerf records statistics during a test such as:

- Item requests sent and images received
- Image retrieval time
- The update rate
- The post message rate
- The generic message rate
- Latency statistics
- CPU and memory usage

For more details on output file information, refer to Chapter 10, Output File Details.

4.6.1 ConsPerf Summary File Sample

```
--- TEST INPUTS ---

  Steady State Time: 90
  Delay Steady State Time: 0
  Connection Type: socket
  Hostname: localhost
  Port: 14002
  Service: DIRECT_FEED
  Thread Count: 1
  Output Buffers: 5000
  Input Buffers: 15
  Send Buffer Size: 0(use default)
  Recv Buffer Size: 0(use default)
  High Water Mark: 0(use default)
  Interface Name: (use default)
  Tcp_NoDelay: Yes
  Username: (use system login name)
  Item Count: 100000
  Common Item Count: 0
  Request Rate: 500000
  Request Snapshots: No
  Posting Rate: 0
  Latency Posting Rate: 0
  Item File: 350k.xml
  Data File: MsgData.xml
  Summary File: ConsSummary.out
  Stats File: ConsStats
  Latency Log File: testlatency
  Latency Show JSON Conv: No
  Tick Rate: 1000
  Prime JVM: No
```

```
--- SUMMARY ---

Startup State Statistics:
  Sampling duration (sec): 1.222
  Latency avg (usec): 51661.7
  Latency std dev (usec): 63243.3
  Latency max (usec): 178689.0
  Latency min (usec): 388.0
  Avg update rate: 100930

Steady State Statistics:
  Sampling duration (sec): 93.079
  Latency avg (usec): 200.7
  Latency std dev (usec): 1898.5
  Latency max (usec): 57247.0
  Latency min (usec): 85.0
  Avg update rate: 100017

Overall Statistics:
  Sampling duration (sec): 94.301
  Latency avg (usec): 1024.5
  Latency std dev (usec): 10253.4
  Latency max (usec): 178689.0
  Latency min (usec): 85.0
  CPU/Memory samples: 19
  CPU Usage max (%): 26.93
  CPU Usage min (%): 0.00
  CPU Usage avg (%): 14.12
  Memory Usage max (MB): 2191.73
  Memory Usage min (MB): 2191.73
  Memory Usage avg (MB): 2191.73

Test Statistics:
  Requests sent: 100000
  Refreshes received: 100000
  Updates received: 9430069
  Image retrieval time (sec): 1.222
  Avg image rate: 81813
  Avg update rate: 100029
```

Code Example 1: ConsPerf Summary File Sample

4.6.2 ConsPerf Statistics File Sample

```
UTC, Latency updates, Latency avg (usec), Latency std dev (usec), Latency max (usec), Latency min (usec),
        Images, Update rate, Post Latency updates, Post Latency avg (usec), Post Latency std dev (usec),
        Post Latency max (usec), Post Latency min (usec), CPU usage (%), Memory(MB)
2013-02-04 22:27:18, 33, 23798.1, 49182.1, 178689.0, 98.0, 100000, 66793, 0, 0.0, 0.0, 0.0, 0.0, 60.00,
        2191.73
2013-02-04 22:27:23, 52, 138.1, 25.7, 203.0, 85.0, 0, 99866, 0, 0.0, 0.0, 0.0, 0.0, 17.65, 2191.73
2013-02-04 22:27:28, 52, 139.3, 29.8, 194.0, 100.0, 0, 99900, 0, 0.0, 0.0, 0.0, 0.0, 4.48, 2191.73
2013-02-04 22:27:33, 45, 143.0, 28.3, 185.0, 100.0, 0, 99900, 0, 0.0, 0.0, 0.0, 0.0, 3.64, 2191.73
2013-02-04 22:27:38, 51, 111.0, 13.3, 164.0, 96.0, 0, 99900, 0, 0.0, 0.0, 0.0, 0.0, 33.33, 2191.73
2013-02-04 22:27:43, 51, 120.1, 12.3, 151.0, 102.0, 0, 99920, 0, 0.0, 0.0, 0.0, 0.0, 4.55, 2191.73
2013-02-04 22:27:48, 53, 121.7, 22.3, 180.0, 99.0, 0, 99900, 0, 0.0, 0.0, 0.0, 0.0, 1.59, 2191.73
```

Code Example 2: ConsPerf Statistics File Sample

4.6.3 ConsPerf Latency File Sample

```
Upd, 4593840028874, 4593840207563, 178689
Upd, 4593840060824, 4593840222878, 162054
Upd, 4593840102796, 4593840238698, 135902
Upd, 4593840145860, 4593840248835, 102975
Upd, 4593840194855, 4593840259678, 64823
Upd, 4593840207845, 4593840262530, 54685
Upd, 4593840235895, 4593840268813, 32918
Upd, 4593840384817, 4593840396570, 11753
Upd, 4593840600799, 4593840614346, 13547
Upd, 4593840816808, 4593840817652, 844
Upd, 4593840880804, 4593840882774, 1970
Upd, 4593840885270, 4593840886078, 808
Upd, 4593840888376, 4593840889229, 853
Upd, 4593840994747, 4593840995135, 388
Upd, 4593841061762, 4593841074479, 12717
Upd, 4593841091891, 4593841099836, 7945
```

Code Example 3: ConsPerf Latency File Sample

4.6.4 ConsPerf Console Output Sample

```
005: Images: 100000, UpdRate:    66793, CPU:   0.00%, Mem: 2191.73MB
  Latency(usec): Avg:23798.1 StdDev:49182.1 Max:178689.0 Min:  98.0, Msgs: 33
  - Image retrieval time for 100000 images: 1.222s (81813 images/s)
010: Images:      0, UpdRate:    99866, CPU:   0.00%, Mem: 2191.73MB
  Latency(usec): Avg: 138.1 StdDev:  25.7 Max: 203.0 Min:  85.0, Msgs: 52
015: Images:      0, UpdRate:    99900, CPU:   0.00%, Mem: 2191.73MB
  Latency(usec): Avg: 139.3 StdDev:  29.8 Max: 194.0 Min: 100.0, Msgs: 52
020: Images:      0, UpdRate:    99900, CPU: 100.00%, Mem: 2191.73MB
  Latency(usec): Avg: 143.0 StdDev:  28.3 Max: 185.0 Min: 100.0, Msgs: 45
025: Images:      0, UpdRate:    99900, CPU:   0.00%, Mem: 2191.73MB
  Latency(usec): Avg: 111.0 StdDev:  13.3 Max: 164.0 Min:  96.0, Msgs: 51
030: Images:      0, UpdRate:    99920, CPU:   0.00%, Mem: 2191.73MB
  Latency(usec): Avg: 120.1 StdDev:  12.3 Max: 151.0 Min: 102.0, Msgs: 51
035: Images:      0, UpdRate:    99900, CPU:   0.00%, Mem: 2191.73MB
  Latency(usec): Avg: 121.7 StdDev:  22.3 Max: 180.0 Min:  99.0, Msgs: 53
```

Code Example 4: ConsPerf Console Output Sample

5 Interactive Provider Performance Tool

5.1 Overview

A typical interactive provider allows consuming applications, including LSEG Real-Time Distribution System, to connect. Once connected, consumers log in and request content. The interactive provider will respond, providing requested content when possible and a status indicating some type of failure when not possible. While a provider in a production environment might get its data from an external source or by performing a calculation on some other data, the performance provider generates its data internally.

ProvPerf implements an OMM interactive provider using the Enterprise Transport API. It starts a server which allows OMM consumers to connect (either directly or through LSEG Real-Time Distribution System), and provides customizable refresh messages and update messages for requested items.

When a new connection is being established, the provider performs some administrative tasks, such as processing login messages, handling directory requests, and (optionally) providing a dictionary. This application uses the Enterprise Transport API Value Add Administration Domain Representations to complete these tasks. For more information, refer to the *Enterprise Transport API Developers Guide*.

5.2 Threading and Scaling

The Enterprise Transport API is designed to allow calls from multiple threads, such that applications can scale their work across multiple cores by creating multiple threads to handle multiple connections through the Enterprise Transport API. To support this multi-threading feature, each application must enable global locking when calling **`Transport.initialize()`**.

You can configure **ProvPerf** for multiple threads by using the **`-threads`** command-line option. When multiple threads are configured, consumer connections are balanced such that each thread receives an equal number of connections.

The main thread monitors the other threads and collects and reports statistics from them.

5.3 Provider Lifecycle

The lifecycle of **ProvPerf** is divided into the following sections:

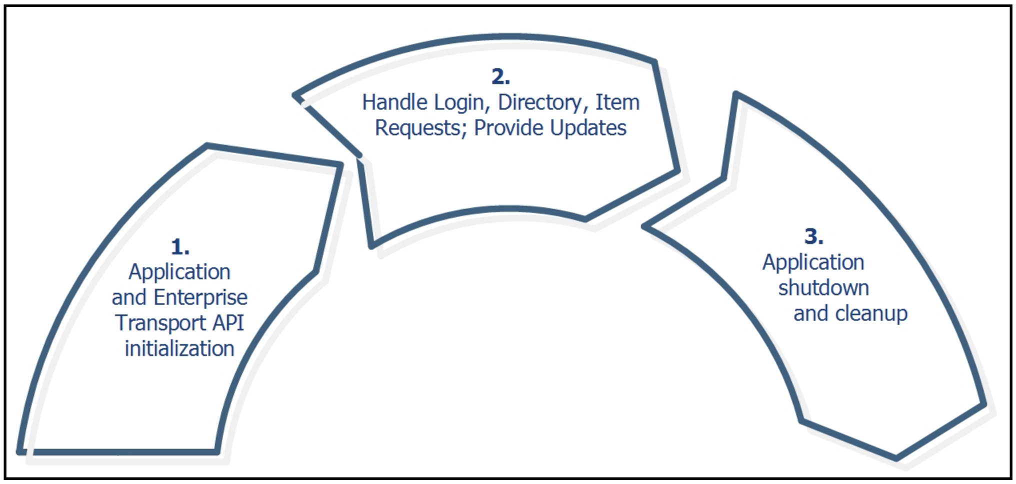

Figure 13. ProvPerf Application Flow

1. Application and Enterprise Transport API Initialization.

   ProvPerf loads its configuration, initializes the Enterprise Transport API, loads its sample message data using specified files, and starts one or more threads (as configured) to provide data to consumers.

   The main thread has two roles:

   • Accept consumer connections and pass them to one of the provider threads.

   • Periodically collect and write statistics from the connection thread(s) until the test is over.

2. Handle Login, Directory, and Item Requests; Provide Updates.

   - Add new connections passed from the main thread.
   - Send a burst of updates for items currently open on existing connections.
   - Send a burst of generic messages (if configured to do so).
   - Use available spare time to provide images for items that need them.
   - Use available spare time to read from the transport, processing any Login, Directory, or Item requests.

3. Shutdown and cleanup.

   The provider thread stops. The main thread collects any remaining data from the connection threads, cleans them up, and writes the final summary statistics. The main thread then cleans up the Enterprise Transport API and remaining resources, and exits.

   ProvPerf should run long enough to allow connected consumers to complete their measurements.

5.3.1 Application Flow Diagram

The following figure shows the flow of the **ProvPerf** application.

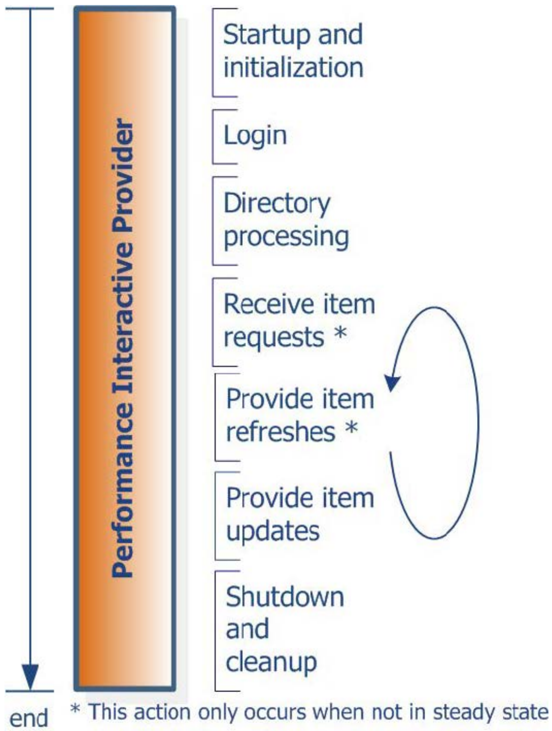

Figure 14. ProvPerf Application Flow

5.4 Latency Measurement

ProvPerf encodes the timestamp as part of its message payload. The timestamp is taken at the start of encoding and added as field TIM_TRK_1 (**3902**)

▶ Interactive Provider Latency Measurement Sequence:

1. Get the current time (**t1**).
2. Obtain an output buffer using `Channel.getBuffer()`
3. Encode the message, including time **t1**.
4. Pass the message to the API, which passes it to the underlying transport.
5. The consuming application receives the timestamp in the payload and compares it against the current time to calculate latency.

5.5 ProvPerf Configuration Options

ProvPerf uses the following command line options:

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-c	socket	Specifies the provider connection type. Available values are: • "socket" • "encrypted"
-directWrite	(no argument)	Sets whether to use the **WriteFlags.DIRECT_SOCKET_WRITE** flag when calling `Channel.write()`. This flag causes the write to attempt to bypass the Enterprise Transport API output queue, reducing latency but at a cost of throughput capability.
-genericMsgLatencyRate	0	Sets the number of generic messages sent (per second) that contain latency data. This number must be greater than the tick rate (see `-tickRate`) and less than the total generic message rate (see `-genericMsgRate`).
-genericMsgRate	0	Sets the number of generic messages sent per second. This number cannot be less than the tick rate (unless it is zero)
-highWaterMark	0	Configures the quantity of data (in bytes) that the Enterprise Transport API queues before automatically flushing it to the network. Adjusting this might provide a tradeoff of throughput vs. latency when writing large bursts of data. If set to 0, a default of 6144 is used.
-if	None	Sets **interfaceName** (a **BindOptions** parameter), which configures the NIC that the provider uses for its server. If your machine straddles networks, you can use this setting to force the provider to use a particular network.
-keyfile	None	Specifies the keystore file directory path and name.
-keypasswd	None	Specifies the keystore password.
-latencyFile	IProvLatency.out	Specifies the name of the log file in which **ProvPerf** logs the latency retrieved from individual latency updates, generic messages, and posts. If a name is not specified, logging is disabled.
-latencyUpdateRate	10	Sets the number of updates sent per second containing latency information.

Table 3: ProvPerf Configuration Options

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-maxEventsInPool	-1	Configures the maximum number of event objects in the event's pool. Set this large enough to minimize the allocation and releasing of memory (a small value might negatively impact performance).
-maxFragmentSize	6144	Sets **MaxFragmentSize** (a **BindOptions** parameter) which specifies the size of buffers. Messages less than this size are sent as a single buffer while larger messages will be fragmented and reassembled into multiple buffers by the Enterprise Transport API.
-maxPackCount	1	Sets the number of buffers packed when sending refresh and update bursts. A setting greater than **1** causes the provider to use the **`Channel.packBuffer()`** method to pack messages, raising the maximum throughput capability (at a potential cost to latency). When packing, the provider attempts to pack the number of buffers specified by **`-maxPackCount`** into a buffer of the length specified by **`-packBufSize`**. The provider periodically displays averages of how many messages are successfully packed.
-msgFile	MsgData.xml	Specifies the file that the provider uses to determine message content.
-noDisplayStats	(no argument)	Turns off printing statistics to the screen.
-packBufSize	6000	Configures the size (in bytes) of the buffer used for packing (used in conjunction with **`-maxPackCount`**). For more information, refer to Section 11.2.3 and the *Transport API Developers Guide*.
-openLimit	1000000	Configures the maximum number of items the provider allows each client to open.
-outputBufs	5000	Sets **GuaranteedOutputBuffers** (an **BindOptions** parameter) which specifies the minimum guaranteed number of output buffers created for each channel that a server accepts. Set this large enough so that the provider does not run out of buffers while writing, but not so large that it wastes memory and slows performance.
-p	14002	Specifies the port on which **ProvPerf** listens for connections.
-pl	""	Specifies a comma- or whitespace-delimited list of desired WebSocket sub-protocols in order of preference. By default, the Enterprise Transport API does not accept connections if protocols are not defined.
-reactor		Send and receive using the Value Added Reactor instead of the Enterprise Transport API channel. For details on using Value Added Components, refer to the *Transport API Value Added Components Developers Guide*.
-recvBufSize	None	Configures the size (in bytes) of the system's receive buffer size. When unspecified, the OS setting is used.
-refreshBurstSize	10	After the provider completes an update burst, it uses the time before the next burst to send any needed refreshes, monitoring the time to see whether the next tick time has been reached. This option configures how often the provider checks the time (in case checking is expensive for the system).
-runTime	360	Sets the length of time **ProvPerf** runs (in seconds).

Table 3: ProvPerf Configuration Options (Continued)

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-securityProvider	SunJSSE	Allows to set security provider for encrypted connection. The library currently supports two options: • SunJSSE (default) • Conscrypt
-sendBufSize	None	Configures the size of the system's send buffer. When unspecified, the OS setting is used.
-serviceId	1	Specifies the ID of the provider's service.
-serviceName	DIRECT_FEED	Specifies the name of the provider's service.
-statsFile	ProvStats	Specifies the base name used to write the provider's test statistics.
-summaryFile	ProvSummary.out	Specifies the base file name used to write the provider's test summary.
-tcpDelay	(no argument)	Configures **tcp_nodelay** (an **BindOptions** parameter) which sets whether the underlying connection uses Nagle's algorithm, a method for batching data and optimizing network bandwidth. By default, performance tools disable Nagle's algorithm due to the increased latency from the buffering, but this option enables it, which can raise throughput capability.
-threads	None	Sets the number of threads that the provider starts. Each thread acts as a provider, and connected consumers are balanced among all threads.
-tickRate	1000	Sets the number of "ticks" (cycles completed by the provider's main loop) per second. Adjusting the tick rate changes the size of update bursts: higher tick rates result in smaller individual bursts, creating smoother traffic.
-tunnelStreamBuffersUsed	(no argument)	Prints statistics for the tunnel stream's buffers. This setting is disabled by default. For details on creating and using tunnel streams, refer to the *Transport API Value Added Components Developers Guide*.
-tunnelStreamOutputBufs	5000	Sets the number of output tunnel buffers (i.e., this parameter configures `guaranteedOutputBuffers` in `TunnelStreamOpenOptions`). For details on creating and using tunnel streams, refer to the *Transport API Value Added Components Developers Guide*.
-updateRate	100000	Configures the number of updates sent per second, per connection. **NOTE:** This cannot be less than the tick rate, unless it is zero.
-writeStatsInterval	5	Sets how often statistics are printed to the screen and statistics file (in seconds).

Table 3: ProvPerf Configuration Options (Continued)

5.6 Input Files

ConsPerf requires the following files:

- Dictionary files to encode/validate fields in the message data. **RDMFieldDictionary** and **enumtype.def** are provided with the package.
- An XML file that describes refresh messages, update messages and generic messages. The package includes a default file (**350k.xml**).

For more details on input file information, refer to Chapter 9, Input File Details.

5.7 Output

ProvPerf records statistics during a test such as:

- Item requests received
- Updates sent
- Posts received and reflected
- CPU and memory usage

For more detailed output file information, refer to Chapter 9, Output File Details 10.

5.7.1 ProvPerf Summary File Sample

```
--- TEST INPUTS ---

   Steady State Time: 90 sec
                Port: 14002
        Thread Count: 1
      Output Buffers: 5000
   Max Fragment Size: 6144
    Send Buffer Size: 0(use default)
    Recv Buffer Size: 0(use default)
      Interface Name: (use default)
         Tcp_NoDelay: Yes
           Tick Rate: 1000
   Use Direct Writes: No
     High Water Mark: 0(use default)
        Summary File: ProvSummary.out
Write Stats Interval: 5
          Stats File: ProvStats
       Display Stats: ProvStats
         Update Rate: 100000
 Latency Update Rate: 10
  Refresh Burst Size: 10
           Data File: MsgData.xml
             Packing: No
        Service Name: DIRECT_FEED
          Service ID: 1
           OpenLimit: 1000000

--- OVERALL SUMMARY ---

Overall Statistics:
  Image requests received: 100000
  Posts received: 0
  Updates sent: 6824846
  Posts reflected: 0
  CPU/Memory samples: 18
  CPU Usage max (%): 32.18
```

```
  CPU Usage min (%): 0.00
  CPU Usage avg (%): 24.21
  Memory Usage max (MB): 2191.73
  Memory Usage min (MB): 2191.73
  Memory Usage avg (MB): 2191.73
```

Code Example 5: ProvPerf Summary File Sample

5.7.2 ProvPerf Statistics File Sample

```
UTC, Requests received, Images sent, Updates sent, Posts reflected, CPU usage (%), Memory(MB)
2016-03-11 17:45:49, 0, 0, 0, 0, 50.00, 2191.73
2016-03-11 17:45:54, 100000, 13001, 133916, 0, 0.00, 2191.73
2016-03-11 17:45:59, 0, 86999, 497796, 0, 0.00, 2191.73
2016-03-11 17:46:04, 0, 0, 499588, 0, 0.00, 2191.73
2016-03-11 17:46:09, 0, 0, 499598, 0, 0.00, 2191.73
2016-03-11 17:46:14, 0, 0, 499508, 0, 0.00, 2191.73
2016-03-11 17:46:19, 0, 0, 499594, 0, 0.00, 2191.73
```

Code Example 6: ProvPerf Statistics File Sample

5.7.3 ProvPerf Console Output Sample

```
005: UpdRate:        0, CPU:   0.00%, Mem: 2191.73MB
010: UpdRate:    26783, CPU:   0.00%, Mem: 2191.73MB
 - Received 100000 item requests (total: 100000), sent 13001 images (total: 13001)
015: UpdRate:    99559, CPU:   0.00%, Mem: 2191.73MB
 - Received 0 item requests (total: 100000), sent 86999 images (total: 100000)
020: UpdRate:    99917, CPU:   0.00%, Mem: 2191.73MB
025: UpdRate:    99919, CPU:   0.00%, Mem: 2191.73MB
030: UpdRate:    99901, CPU:   0.00%, Mem: 2191.73MB
035: UpdRate:    99918, CPU:   0.00%, Mem: 2191.73MB
```

Code Example 7: ProvPerf Console Output Sample

6 Non-Interactive Provider Performance Tool

6.1 Overview

A ***Non-Interactive Provider*** publishes content regardless of consumer requests by connecting to an LSEG Real-Time Advanced Distribution Hub and publishing content to the LSEG Real-Time Advanced Distribution Hub cache. After login, a non-interactive provider publishes a service directory and then starts sending data for supported items.

NIProvPerf implements an Open Message Model non-interactive provider using the Enterprise Message API C# Edition for use with the LSEG Real-Time Advanced Distribution Hub on the LSEG Real-Time Distribution System. It connects and logs into an LSEG Real-Time Advanced Distribution Hub, publishes its service, and then provides images and updates.

When connecting, the non-interactive provider performs some administrative tasks, like processing system logins and publishing a directory refresh. The **NIProvPerf** uses Enterprise Transport API Value Add Administration Domain Representations to complete these tasks. For more information, refer to the *Enterprise Transport API Developers Guide*.

6.2 Threading and Scaling

You can configure **NIProvPerf** for multiple threads via the **`-threads`** command-line option. When you configure multiple threads, each thread opens its own connection to the LSEG Real-Time Advanced Distribution Hub, and the list of items is divided among all threads. You can use the **`-commonItemCount`** option to control the number of items that will be sent across all threads.

The main thread monitors the other threads and then collects and reports their statistics.

6.3 Non-Interactive Provider Lifecycle

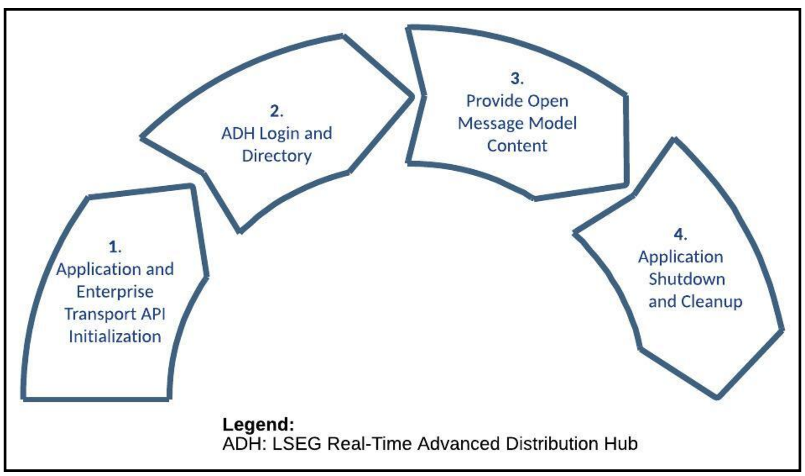

Figure 15. NIProvPerf Lifecycle

The lifecycle of **NIProvPerf** is divided into the following sections:

1. Application and Enterprise Transport API Initialization.

   In this phase **NIProvPerf**:

   - Loads its configuration.
   - Initializes the Enterprise Transport API.
   - Loads its dictionary, item list, and sample message data using the specified files.
   - Starts the thread(s) that will connect to the LSEG Real-Time Advanced Distribution Hub to perform the test.

- The main thread begins cycling: periodically collecting and writing statistics from the connection thread(s).
- Connection threads connect to the LSEG Real-Time Advanced Distribution Hub If a connection fails, the thread continually attempts to reconnect until the connection succeeds. Once the connection succeeds, the test begins and any subsequent disconnection ends the test.

2. LSEG Real-Time Advanced Distribution Hub Login and Directory.

The connection thread sends login requests and waits for the LSEG Real-Time Advanced Distribution Hub response. After a successful login, the connection thread publishes its service, followed by an item image to begin the publishing phase.

3. Provide Open Message Model content.

The connection thread begins providing the items specified in its item list, continually performing the following actions:

- Send a burst of updates for open items.
- If refreshes are needed, use spare time in the tick to send them.
- Using any spare time left, read from the transport and process incoming messages.

4. Application shutdown and cleanup.

The connection thread disconnects and stops. The main thread collects any remaining information from the connection threads, cleans them up, and writes the final summary statistics. The main thread then cleans up the Enterprise Transport API and any remaining resources and then exits.

Run **NIProvPerf** for a long enough period of time to allow for connected consumers to complete their measurements.

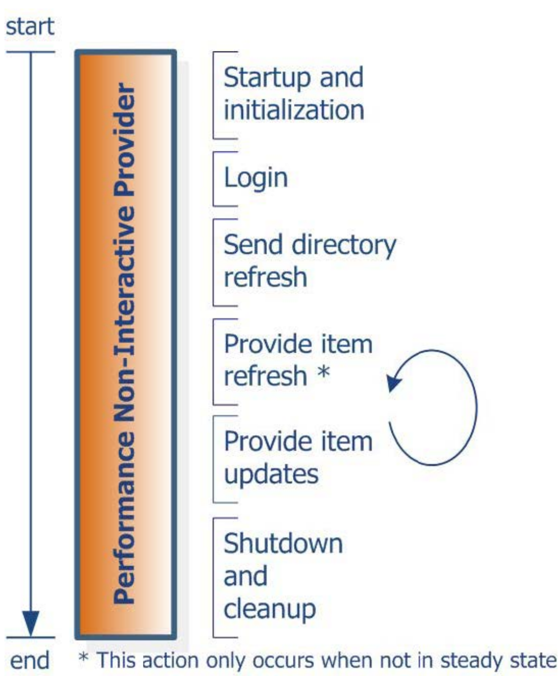

Figure 16. NIProvPerf Application Flow

6.4 Latency Measurement

The Enterprise Transport API is designed to allow calls from multiple threads, such that applications can scale their work across multiple cores. Applications can take advantage of this by creating multiple threads to handle multiple connections through the Enterprise Transport API. To support this multi-threading, each application enables global locking when calling `Transport.initialize()`.

NIProvPerf encodes a timestamp as part of its message payload. The timestamp is taken at the start of encoding and added as field TIM_TRK_1 (**3902**). Latency is measured after decodes the message and payload.

Non-Interactive Provider Latency Measurement Sequence:

1. Get the current time (**t1**).
2. Obtain an output buffer using `Channel.getBuffer()`.
3. Encode the message, including time **t1**.
4. Pass the message to the API, which passes it to underlying transport.
5. The consuming application receives a timestamp in the payload and compares it to the current time to calculate latency.

6.5 NIProvPerf Configuration Options

NIProvPerf uses the following command line options:

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-cert		Specifies the file containing the server certificate for encryption.
-commonItemCount	0	If multiple consumer threads are created (see `-threads`), each thread normally requests a unique set of items on its connection. This option specifies the number of common items to be requested by all connection.
-connType	socket	Specifies the connection type used by the application. Supported options are **socket**, **http**, **encrypted**, and **reliableMCast**.
-directWrite	(no argument)	Sets whether the **WriteFlags.DIRECT_SOCKET_WRITE** flag is used when calling `Channel.write()`. Using this flag causes the write to attempt to bypass the Enterprise Transport API output queue, reducing latency but at a cost of throughput capability.
-h	localhost	When using TCP socket connection types, specifies the hostname of the machine on which the LSEG Real-Time Advanced Distribution Server (to which the provider connects) runs.
-highWaterMark	0	Configures the Enterprise Transport API's "High-water Mark," the amount of data (in bytes) that the Enterprise Transport API queues before flushing it to the network. Adjusting this might provides a trade-off of throughput vs. latency when writing large bursts of data. If set to 0, a default of 6144 is used.
-if	None	Configures **interfaceName** (an **ConnectOptions** parameter), which configures the NIC that the provider uses for its server. On computers connected to multiple networks, you can use this parameter to force the provider to use the desired network.
-itemCount	100000	Sets the total number of items that the provider will publish.
-itemFile	350k.xml	Specifies the file that contains a list of items the provider will publish. For more details on input file information, refer to Section 8.2.

Table 4: NIProvPerf Configuration Options

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-key		Specifies the file containing the server private key for encryption.
-latencyUpdateRate	10	Sets the number of updates with latency information sent per second. **NOTE:** • This must be greater than the tick rate (see **`-tickRate`**) but less than the total update rate (see **`-updateRate`**).
-maxPackCount	1	Sets the number of buffers packed when sending refresh and update bursts. Specifying a count greater than 1 causes the provider to use the **`Channel.packBuffer()`** method to pack messages, raising maximum throughput capability but potentially having a negative affect on latency. When packing, the provider attempts to pack the number of buffers specified by **`-maxPackCount`** into a buffer of the length specified by **`-packBufSize`**. The provider periodically displays averages of how many messages are successfully packed.
-msgFile	MsgData.xml	Specifies the file that determines the provider's message content.
-noDisplayStats	(no argument)	Turns off printing statistics to the screen.
-packBufSize	6000	Specifies the size of the buffer (in bytes) used for packing. Used in conjunction with **`-maxPackCount`**.
-outputBufs	5000	Configures **GuaranteedOutputBuffers** (an **ConnectOptions** parameter), which sets the minimum guaranteed number of output buffers created for each channel. Set this value large enough so that the provider does not run out of buffers while writing but low enough so as not to waste memory and slow performance.
-p	14003	When using TCP socket connection types, specifies the port number the provider uses to connect to the LSEG Real-Time Advanced Distribution Server.
-ra	None	When using a reliable multicast connection, configures the multicast receive address.
-reactor		Send and receive using the Value Added Reactor instead of the Transport API channel. For details on using Value Added Components, refer to the *Transport Java Edition API Value Added Components Developers Guide*.
-recvBufSize	None	Configures the system receive buffer size. By default, the OS setting is used.
-refreshBurstSize	10	After the provider completes an update burst, it uses the time before the next burst to send any needed refreshes, monitoring the time to see whether it is time for the next tick time. This option configures how often the provider checks the time (in case checking is expensive for the system).
-rp	None	When using reliable multicast connection type, configures the multicast receive port.
-runTime	360	Sets the length of time for which **NIProvPerf** runs, in seconds.
-sa	None	When using reliable multicast connection type, configures the multicast send address.

Table 4: NIProvPerf Configuration Options (Continued)

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-securityProvider	SunJSSE	Allows to set security provider for encrypted connection. The library currently supports two options: • SunJSSE (default) • Conscrypt
-sendBufSize	None	Configures the system send buffer size. By default, the OS setting is used.
-serviceId	1	Specifies the provider's service ID.
-serviceName		Specifies the provider's service name.
-sp	None	When using reliable multicast connection type, configures the multicast send port.
-statsFile	NIProvStats	Specifies the base filename used to write the provider's test statistics.
-summaryFile	NIProvSummary.out	Specifies the base filename used to write the provider's test summary.
-tcpDelay	(no argument)	Configures **tcp_nodelay** (a **ConnectOptions** parameter), an option that sets whether the underlying connection uses Nagle's algorithm; a method for more efficiently using network packet headers by batching data. By default, **NIProvPerf** disables Nagle's algorithm due to the increased latency from the buffering, but **`-tcpDelay`** enables it, which can raise throughput capability.
-threads	None	Sets the number of threads started by the provider. Each thread opens its own connection to the LSEG Real-Time Advanced Distribution Server, and the list of items is divided among all threads.
-tickRate	1000	Sets the number of ticks per second (the number of cycles per second made by the provider's main loop). Adjusting the tick rate changes the size of update bursts; higher tick rates result in smaller individual bursts and smoother traffic.
-u	None	When using reliable multicast connection type, sets the unicast port.
-updateRate	100000	Sets the total number of updates sent per second, per connection. **NOTE:** This cannot be less than the tick rate, unless it is **0**.
-useServiceId	(no argument)	Turns on the usage of the service ID. See the **`-serviceId`** option.
-writeStatsInterval	5	Sets how often **NIProvPerf** prints statistics to the screen and statistics file.

Table 4: NIProvPerf Configuration Options (Continued)

6.6 Input Files

NIProvPerf requires the following files:

- An XML file that describes **NIProvPerf** message data. By default, the package includes the file: **MsgData.xml**.
- Dictionary files to validate fields present in the message data. By default, the package includes the **RDMFieldDictionary** and **enumtype.def** files.
- An XML file that describes the items that **NIProvPerf** should publish. By default, the package includes the file, **350k.xml**.

For more detailed input file information, refer to Chapter 8, Input File Details 9.

6.7 Output

NIProvPerf records statistics during a test, such as:

- The number of sent images
- The number of sent updates
- CPU and memory usage

For more detailed output file information, refer to Chapter 9 10.

6.7.1 NIProvPerf Summary File Sample

```
--- TEST INPUTS ---

  Run Time: 60 sec
  Connection Type: socket
  Hostname: oaklrh192
  Port: 14003
  Thread List: 1
  Output Buffers: 5000
  Max Fragment Size: 6144
  Send Buffer Size: 0(use default)
  Recv Buffer Size: 0(use default)
  High Water Mark: 0(use default)
  Interface Name: (use default)
  Username: (use system login name)
  Tcp_NoDelay: Yes
  Item Count: 100000
  Common Item Count: 0
  Tick Rate: 1000
  Use Direct Writes: No
  Summary File: NIProvSummary.out
  Stats File: NIProvStats
  Write Stats Interval: 5
  Display Stats: true
  Update Rate: 100000
  Latency Update Rate: 10
  Refresh Burst Size: 10
  Item File: 350k.xml
  Data File: MsgData.xml
  Packing: No
  Service ID: 1
  Service Name: NI_PUB

--- OVERALL SUMMARY ---

Overall Statistics:
  Images sent: 100000
  Updates sent: 5754945
  CPU/Memory samples: 12
  CPU Usage max (%): 27.41
  CPU Usage min (%): 0.00
  CPU Usage avg (%): 23.92
  Memory Usage max (MB): 2191.73
  Memory Usage min (MB): 2191.73
  Memory Usage avg (MB): 2191.73
```

Code Example 8: NIProvPerf Summary File Sample

6.7.2 NIProvPerf Statistics File Sample

```
UTC, Images sent, Updates sent, CPU usage (%), Memory (MB)
2013-03-11 18:20:37, 0, 42673, 361700, 66.67, 2191.73
2013-03-11 18:20:42, 0, 57327, 393174, 0.00, 2191.73
2013-03-11 18:20:47, 0, 0, 498371, 3.70, 2191.73
2013-03-11 18:20:52, 0, 0, 500400, 0.00, 2191.73
2013-03-11 18:20:57, 0, 0, 500100, 0.00, 2191.73
2013-03-11 18:21:02, 0, 0, 500200, 0.00, 2191.73
2013-03-11 18:21:07, 0, 0, 500100, 0.00, 2191.73
2013-03-11 18:21:12, 0, 0, 500200, 0.00, 2191.73
2013-03-11 18:21:17, 0, 0, 500100, 0.00, 2191.73
```

Code Example 9: NIProvPerf Statistics File Sample

6.7.3 NIProvPerf Console Output Sample

```
020: UpdRate:   100080, CPU:   0.00%, Mem: 2191.73MB
025: UpdRate:   100020, CPU:   0.00%, Mem: 2191.73MB
030: UpdRate:   100040, CPU:   0.00%, Mem: 2191.73MB
035: UpdRate:   100020, CPU:   0.00%, Mem: 2191.73MB
040: UpdRate:   100040, CPU:   0.00%, Mem: 2191.73MB
045: UpdRate:   100020, CPU:   0.00%, Mem: 2191.73MB
```

Code Example 10: NIProvPerf Console Output Sample

7 TransportPerf Performance Tool

7.1 Overview

TransportPerf measures the performance of the various API transport layers. **TransportPerf** does not use Open Message Model messages, instead sending opaque content with minimal encoding and decoding. To enforce the proper ordering of data, **TransportPerf** embeds a sequence number in each buffer.

TransportPerf can act as a server or client. A typical use case is to start **TransportPerf** as a server and then start **TransportPerf** as a client to connect to the server.

7.2 Threading and Scaling

The Enterprise Transport API allows calls from multiple threads, such that applications can scale their work across multiple cores (via multi-threading) to handle multiple connections. To support multi-threading, each application must enable global locking when calling `Transport.initialize()` to ensure that shared resources in the Enterprise Transport API are protected.

You can configure **TransportPerf** for multiple threads by using the `-threads` option. The result depends on whether the application is run as a server or client.

- When running as a server, each thread is used to balance incoming connections (similar to **ProvPerf**).
- When running as a client, each thread creates its own connection (similar to **ConsPerf**).

The main thread monitors the other threads and collects and reports their statistics.

7.3 TransportPerf Life Cycle

The lifecycle of **TransportPerf** is divided into the following phases:

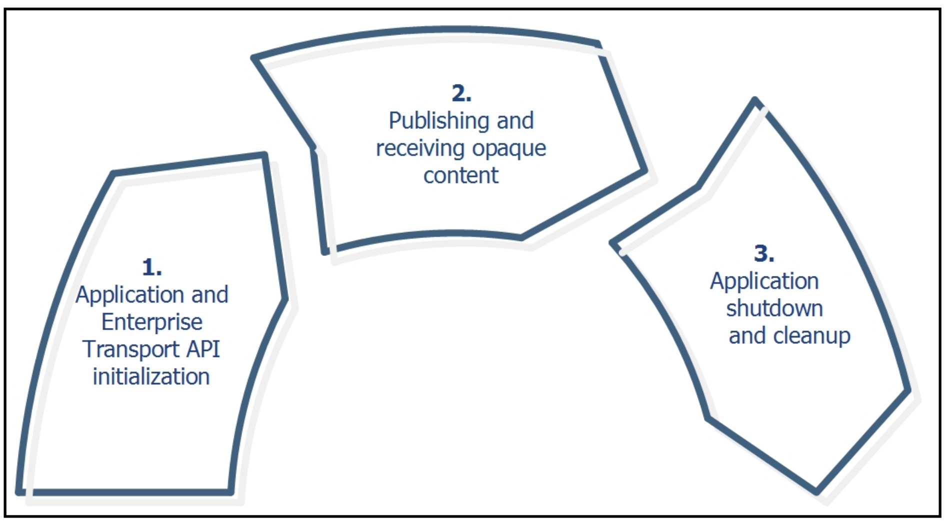

Figure 17. TransportPerf Lifecycle 1

1. Application and Enterprise Transport API initialization.

   TransportPerf loads its configuration, initializes the Enterprise Transport API, and starts the transport thread(s) that read and write data. The main thread now:

   - If acting as server, accepts client connections and passes them to one of the transport threads.
   - If acting as client, connects out to server.
   - Periodically collects and writes statistics from the transport thread(s) for the remainder of the test.

2. Publishing and receiving opaque content.

   The transport thread performs the following actions until its run time expires:

   - Adds new connections passed from the main thread.
   - Sends bursts of messages on existing connections, as specified by command line arguments.
   - Uses remaining time segments to read from the transport and process incoming data.

3. Application shutdown and cleanup.

   The transport thread stops. The main thread collects any remaining data from the transport threads, cleans them up, and writes the final summary statistics. Finally, the main thread uninitializes the Enterprise Transport API, cleans up remaining resources, and exits.

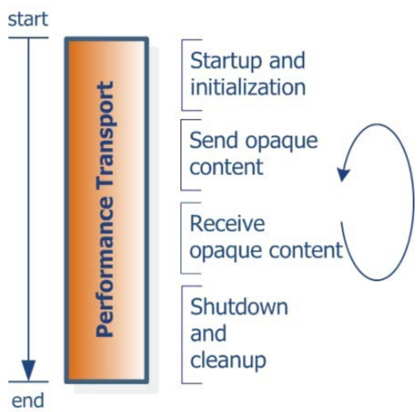

Figure 18. TransportPerf Application Flow

7.4 Message Payload

TransportPerf writes a message of the configured size obeying the following rules:

- The message starts with an 8-byte sequence number, which is checked by the receiver of the message.
- If the message contains a timestamp, the stamp is added after the sequence number as an 8-byte integer.
- The remainder of the message is set to zeros.

7.5 Latency Measurement

TransportPerf writes a timestamp (with nanosecond granularity) as part of the message payload Section 7.4. To determine latency, the receiving **TransportPerf** reads the timestamp and compares it to the current time.

▶ Transport Performance Latency Measurement Sequence:

1. Get the current time (**t1**).
2. Obtain an output buffer using `Channel.getBuffer()`.
3. Encode the message, including time **t1**.
4. Pass the message to the API, which passes it to underlying transport.

   To determine latency, the consuming application reads the timestamp from the payload and compares it against the current time.

7.6 TransportPerf Configuration Options

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-appType	server	Configures the type of application that **TransportPerf** simulates (available settings include server or client).
-busyRead	None	Configures the application to continually read rather than use notifications.
-compressionLevel	5	When **TransportPerf** is configured to compress messages, `-compressionLevel` sets the level of compression.
-compressionType	None	Sets the type of compression **TransportPerf** uses when compressing messages. By default, **TransportPerf** does not compress its messages. Options include: • <none> • zlib • lz4
-connType	socket	Specifies the connection type that the application uses. Supported options are `socket`, `http`, `encrypted`, `reliableMCast`, and `shmem`.
-directWrite	False	Sets whether the **WriteFlags.DIRECT_SOCKET_WRITE** flag is used to call `rsslWrite`. Using this flag causes the write to attempt to bypass the Enterprise Transport API output queue, reducing latency but decreasing throughput capability.
-h	localhost	When using TCP socket or shared memory connection types, specifies the hostname to which the client application connects.
-highWaterMark	0	Configures the Enterprise Transport API's "High-water Mark," the amount of data (in bytes) that the Enterprise Transport API queues before flushing it to the network. Adjusting this might provide a trade-off of throughput vs. latency when writing large bursts of data. If set to 0, a default of 6144 is used.
-if	None	Configures **interfaceName** (a **ConnectOptions**/**BindOptions** parameter), which configures the NIC that the provider uses for its server. On computers connected to multiple networks, you can use this parameter to force the provider to use a specific network.

Table 5: TransportPerf Configuration Options

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-keyfile	""	Server private keystore for Java encryption.
-keypasswd	""	Keystore password.
-krbfile	""	Proxy Kerberos configuration file.
-latencyFile	None	Configures **TransportPerf** to log the latency of data from updates and posts, and specifies the name of the file in which **TransportPerf** stores the results.
-latencyMsgRate	10	Sets the number of messages that **TransportPerf** sends per second to measure latency. This cannot be larger than **`-msgRate`**.
-maxFragmentSize	6144	Configures **maxFragmentSize** (a **BindOptions** parameter), which controls the size (in bytes) of the buffers in the Enterprise Transport API's buffer pool.
-mcastStats	(no argument)	Enables **TransportPerf** to print multicast statistics. To print multicast statistics, **TransportPerf** must also enable channel locks in the Enterprise Transport API.
-msgRate	100000	Sets the number of messages that **TransportPerf** sends per second. **`-msgRate`** cannot be less than **`-latencyMsgRate`** as latency messages are a type of message and hence included in this number. For example: using **`-latencyMsgRate 1000 -msgRate 1000`** sends 1000 messages per second with each message carrying latency data.
-msgSize	76	Configures the size (in bytes) of messages sent by **TransportPerf**.
-noDisplayStats	(no argument)	Turns off printing statistics to the screen.
-outputBufs	5000	Configures **guaranteedOutputBuffers** (a **ConnectOptions**/**BindOptions** parameter), which sets the minimum guaranteed number of output buffers created for each channel. **TIP:** Set this value large enough so that the provider does not run out of buffers while writing but low enough so as not to waste memory and slow down performance.
-p	14002	When using TCP socket or shared memory connection types, specifies the port number over which the consumer connects. This option applies to both types of applications (server or client).
-pack	No packing	Sets the number of messages to pack in each buffer using **`Channel.packBuffer()`**.
-pdomain	""	Proxy domain.
-ph	""	Proxy host.
-pl	""	Specifies a comma- or whitespace-delimited list of WebSocket sub-protocols in order of preference for requesting a protocol or listing only supported protocols respectively for client and server **`appType`** (e.g., **`rssl.rwf, rssl.json.v2, tr_json2`**). **NOTE:** LSEG plans to deprecate **`tr_json2`** and replace it with **`rssl.json.v2`**.
-plogin	""	Login ID on proxy server.
-pp	""	Proxy number for proxy server.
-ppasswd	""	Password on proxy server.

Table 5: TransportPerf Configuration Options (Continued)

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-proxy	false	When set uses proxy server.
-prr	""	Port roam range.
-ra	None	When using the multicast connection type, configures the multicast receive address.
-recvBufSize	None	Configures the system receive buffer size. By default, the OS setting is used.
-reflectMsgs	False	Sets **TransportPerf** to reflect back all received messages.
-rp	None	When using the multicast connection type, configures the multicast receive port.
-runTime	300	Sets how long **TransportPerf** runs, in seconds.
-sa	None	When using the multicast connection type, configures the multicast send address.
-sendBufSize	None	Configures the system send buffer size. By default, the OS setting is used.
-sp	None	When using the multicast connection type, configures the multicast send port.
-statsFile	TransportStats	Specifies the base name used in writing the application's test statistics.
-summaryFile	TransportSummary.out	Specifies the name of the file to which **TransportPerf** writes the application's test summary.
-tcpDelay	False	Configures **tcp_nodelay** (a **ConnectOptions**/**BindOptions** parameter) which sets whether the underlying connection uses Nagle's algorithm; a method for more efficiently using network packet headers by batching data. By default, **TransportPerf** disables Nagle's algorithm due to the increased latency from buffering. **`-tcpDelay`** enables it, which can raise throughput capability.
-threads	None	Sets the number of threads that the application creates and the CPU core to which each thread binds.
-tickRate	1000	Sets the number of cycles (i.e., ticks) per second executed by the **TransportPerf** main loop. Adjusting the tick rate changes the size of request/post bursts: a higher tick rate results in smaller individual bursts and smoother traffic.
-u	None	When using the multicast connection type, configures the unicast port.
-writeStatsInterval	5	Configures how often (in seconds) **TransportPerf** prints stats to screen and to the statistics file.

Table 5: TransportPerf Configuration Options (Continued)

7.7 Input

TransportPerf does not require input files.

7.8 Output

During the test, **TransportPerf** records the following statistics:

- Message rate (sent and received)
- Data rate (sent and received)
- Latency statistics
- CPU and memory usage

NOTE: The data sent rate is taken from **`Channel.write()`**'s **bytesWritten** parameter. This value includes any internal headers used in transport, packing, and/or compression. The data received rate is based on message lengths returned from **`Channel.read()`**, which does not indicate whether data was compressed and does not include any transport header overhead. As a result, rates may differ between the sender and receiver.

For more detailed output file information, refer to Section 10.

7.8.1 TransportPerf Summary File Sample

```
--- TEST INPUTS ---

               Runtime: 180 sec
       Connection Type: socket
              Hostname: localhost
                  Port: 14002
              App Type: client
          Thread Count: 1
             Busy Read: No
              Msg Size: 76
              Msg Rate: 100000
      Latency Msg Rate: 10
        Output Buffers: 5000
     Max Fragment Size: 6144
      Send Buffer Size: 0(use default)
      Recv Buffer Size: 0(use default)
       High Water Mark: 0(use default)
      Compression Type: none(0)
     Compression Level: 5
        Interface Name: (use default)
           Tcp_NoDelay: Yes
             Tick Rate: 1000
     Use Direct Writes: No
      Latency Log File: (none)
          Summary File: TransportSummary_24806.out
            Stats File: TransportStats
  Write Stats Interval: 5
         Display Stats: Yes
               Packing: No

--- OVERALL SUMMARY ---

Statistics:
  Latency avg (usec): 19.896
  Latency std dev (usec): 160.616
  Latency max (usec): 5304.720
  Latency min (usec): 9.052
  Sampling duration(sec): 179.98
  Msgs Sent: 17998200
```

```
  Msgs Received: 17999300
  Data Sent (MB): 1355.99
  Data Received (MB): 1304.58
  Avg. Msg Sent Rate: 100000
  Avg. Msg Recv Rate: 100006
  Avg. Data Sent Rate (MB): 7.53
  Avg. Data Recv Rate (MB): 7.25
  CPU/Memory samples: 36
  CPU Usage max (%): 0.18
  CPU Usage min (%): 0.00
  CPU Usage avg (%): 0.00
  Memory Usage max (MB): 31.20
  Memory Usage min (MB): 21.93
  Memory Usage avg (MB): 28.90
  Process ID: 24806
```

Code Example 11: TransportPerf Summary File Sample

7.8.2 TransportPerf Statistics File Sample

```
UTC, Msgs sent, Bytes sent, Msgs received, Bytes received, Latency msgs received, Latency avg (usec),
        Latency std dev (usec), Latency max (usec), Latency min (usec), CPU usage (%), Memory (MB)
2013-05-14 06:35:59, 498000, 39342000, 499100, 37931600, 50, 206.135, 954.202, 5304.720, 9.902, 0.18,
        21.93
2013-05-14 06:36:04, 500000, 39500000, 500000, 38000000, 50, 14.107, 2.554, 19.491, 9.227, 0.00, 21.99
2013-05-14 06:36:09, 500000, 39500000, 500000, 38000000, 50, 14.836, 2.667, 19.105, 10.180, 0.00, 21.99
2013-05-14 06:36:14, 500000, 39500000, 500000, 38000000, 50, 14.720, 2.710, 19.046, 9.721, 0.00, 21.99
2013-05-14 06:36:19, 500000, 39500000, 500000, 38000000, 50, 14.434, 2.293, 19.497, 10.722, 0.00, 21.99
2013-05-14 06:36:24, 500000, 39500000, 500000, 38000000, 50, 14.133, 2.401, 19.122, 9.683, 0.00, 21.99
2013-05-14 06:36:29, 500000, 39500000, 500000, 38000000, 50, 14.878, 2.527, 19.159, 9.773, 0.00, 21.99
2013-05-14 06:36:34, 500000, 39500000, 500000, 38000000, 50, 14.874, 2.719, 20.240, 10.386, 0.00, 23.96
2013-05-14 06:36:39, 500000, 39500000, 500000, 38000000, 50, 14.460, 2.303, 19.820, 9.975, 0.00, 23.96
```

Code Example 12: TransportPerf Statistics File Sample

7.8.3 TransportPerf Console Output Sample

```
030:
  Sent: MsgRate:   100000, DataRate:   7.534MBps
  Recv: MsgRate:   100000, DataRate:   7.248MBps
  Latency (usec): Avg:  14.133 StdDev:   2.401 Max:  19.122 Min:   9.683, Msgs: 50
  CPU:   0.00% Mem:    21.99MB
035:
  Sent: MsgRate:   100000, DataRate:   7.534MBps
  Recv: MsgRate:   100000, DataRate:   7.248MBps
  Latency (usec): Avg:  14.878 StdDev:   2.527 Max:  19.159 Min:   9.773, Msgs: 50
  CPU:   0.00% Mem:    21.99MB
```

```
040:
  Sent: MsgRate:   100000, DataRate:   7.534MBps
  Recv: MsgRate:   100000, DataRate:   7.248MBps
  Latency (usec): Avg:  14.874 StdDev:   2.719 Max:  20.240 Min:  10.386, Msgs: 50
  CPU:   0.00% Mem:    23.96MB
045:
  Sent: MsgRate:   100000, DataRate:   7.534MBps
  Recv: MsgRate:   100000, DataRate:   7.248MBps
  Latency (usec): Avg:  14.460 StdDev:   2.303 Max:  19.820 Min:   9.975, Msgs: 50
  CPU:   0.00% Mem:    23.96MB
050:
  Sent: MsgRate:   100000, DataRate:   7.534MBps
  Recv: MsgRate:   100000, DataRate:   7.248MBps
  Latency (usec): Avg:  14.213 StdDev:   2.291 Max:  19.355 Min:  10.093, Msgs: 50
  CPU:   0.00% Mem:    29.23MB
```

Code Example 13: TransportPerf Console Output Sample

8 Performance Measurement Scenarios

8.1 Interactive Provider to Consumer, Through LSEG Real-Time Distribution System

You can measure interactive providers by connecting the following components, as described below and shown in the following picture:

- Connect **ConsPerf** to an LSEG Real-Time Advanced Distribution Server.
- Connect the LSEG Real-Time Advanced Distribution Server to an LSEG Real-Time Advanced Distribution Hub. You can do so using the RRCP backbone.
- Connect the LSEG Real-Time Advanced Distribution Hub with an instance of **ProvPerf** or **EMAC ProvPerf**.

You can perform this test with caching enabled or disabled in the LSEG Real-Time Advanced Distribution Hub or LSEG Real-Time Advanced Distribution Server, as **ProvPerf** acts as the cache of record in this scenario.

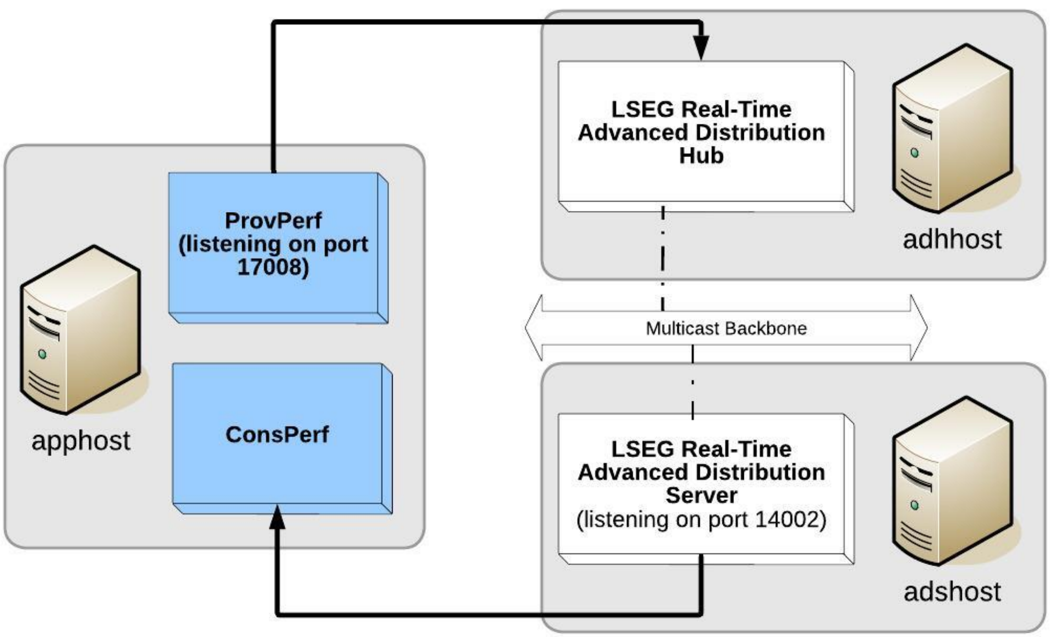

Figure 19. Interactive Provider to Consumer on LSEG Real-Time Distribution System

To run a basic performance measurement using a setup like the one pictured in Figure 5, run **ProvPerf** and **ConsPerf** with the following command-line options. These options assume TEST_FEED is the service being used and 17008 is the port number. Modify the example values as necessary.

```
java com.refinitiv.eta.perftools.provperf.ProvPerf -p 17008 -serviceName TEST_FEED

java com.refinitiv.eta.perftools.consperf.ConsPerf -h adshost -p 14002 -serviceName TEST_FEED
```

8.2 Interactive Provider to Consumer, Direct Connect

You can measure the interactive providers of data by connecting **ConsPerf** directly to **ProvPerf**.

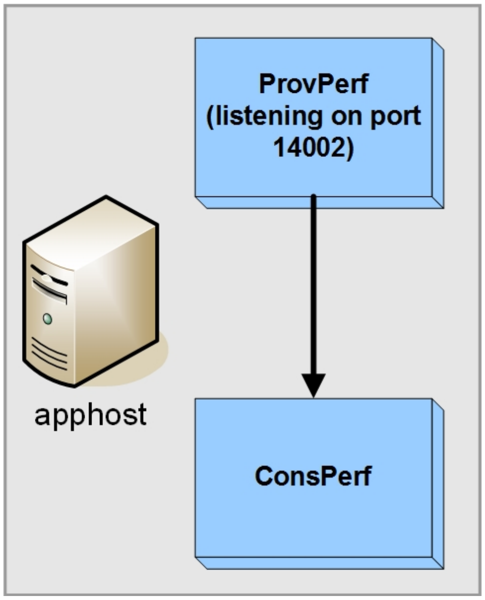

Figure 20. Interactive Provider to Consumer, Direct Connect

▶ To run a basic performance measurement:

Using their default configuration options, you can run this test without any additional command-line options. Simply run the provider and consumer applications as follows:

```
java com.refinitiv.eta.perftools.provperf.ProvPerf

java com.refinitiv.eta.perftools.consperf.ConsPerf
```

8.3 Non-Interactive Provider to Consumer, Through LSEG Real-Time Distribution System

You can measure non-interactive providers on LSEG Real-Time Distribution System by connecting the following components, as described below and displayed in the following picture:

- Connect **ConsPerf** to an LSEG Real-Time Advanced Distribution Server.
- Connect the LSEG Real-Time Advanced Distribution Server with an LSEG Real-Time Advanced Distribution Hub. You can do so by using the RRCP backbone.
- Connect **NIProvPerf** to the LSEG Real-Time Advanced Distribution Hub. Ensure that the LSEG Real-Time Advanced Distribution Hub has caching enabled, because it acts as the cache of record in this scenario.

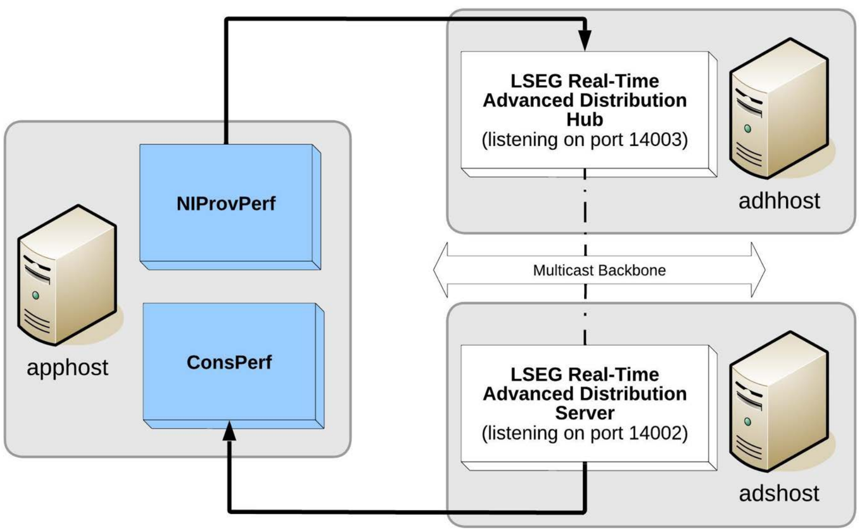

Figure 21. NIProvPerf to Consumer on the LSEG Real-Time Distribution System

ConsPerf may receive a Closed status if it requests an item not yet provided by **NIProvPerf** to the LSEG Real-Time Advanced Distribution Hub cache. To ensure the test completes successfully, you must do either one of the following:

1. Preload the LSEG Real-Time Advanced Distribution Hub cache. **NIProvPerf** must have provided refreshes for all of its items to the LSEG Real-Time Advanced Distribution Hub before **ConsPerf** connects to the LSEG Real-Time Advanced Distribution Server.
2. Configure the LSEG Real-Time Advanced Distribution Hub to provide temporary refreshes in place of the uncached items. **ConsPerf** knows to allow these images, and does not count them towards the image retrieval time, due to their Suspect data state.

For more details on this configuration, refer to the *LSEG Real-Time Advanced Distribution Hub Software Installation Manual*.

▶ To run a basic performance measurement:

To run a basic performance measurement using a setup like the one pictured in this section, run **NIProvPerf** and **ConsPerf** with the following command-line options. These options assume the provided service is TEST_FEED. Modify the example values as necessary.

```
java com.refinitiv.eta.perftools.niprovperf.NIProvPerf -h adhhost -p 14003 -serviceName TEST_FEED

java com.refinitiv.eta.perftools.consperf.ConsPerf -h adshost -p 14002 -serviceName TEST_FEED
```

8.4 Consumer Posting on the LSEG Real-Time Distribution System

To measure posting performance on the LSEG Real-Time Distribution System, connect the following components, as described below and displayed in the following picture:

- Connect **ConsPerf** to an LSEG Real-Time Advanced Distribution Server.
- Connect the LSEG Real-Time Advanced Distribution Server to an LSEG Real-Time Advanced Distribution Hub. You can do so using the RRCP backbone.
- Connect **NIProvPerf** to the LSEG Real-Time Advanced Distribution Hub. The LSEG Real-Time Advanced Distribution Hub must have caching enabled, because it acts as the cache of record in this scenario.

As the posted messages return from the LSEG Real-Time Distribution System, the consumer can distinguish them via the presence of their **PostUserInfo**. When configured to do so, **ConsPerf** embeds timestamps in some of its posts which it uses to measure round-trip latency.

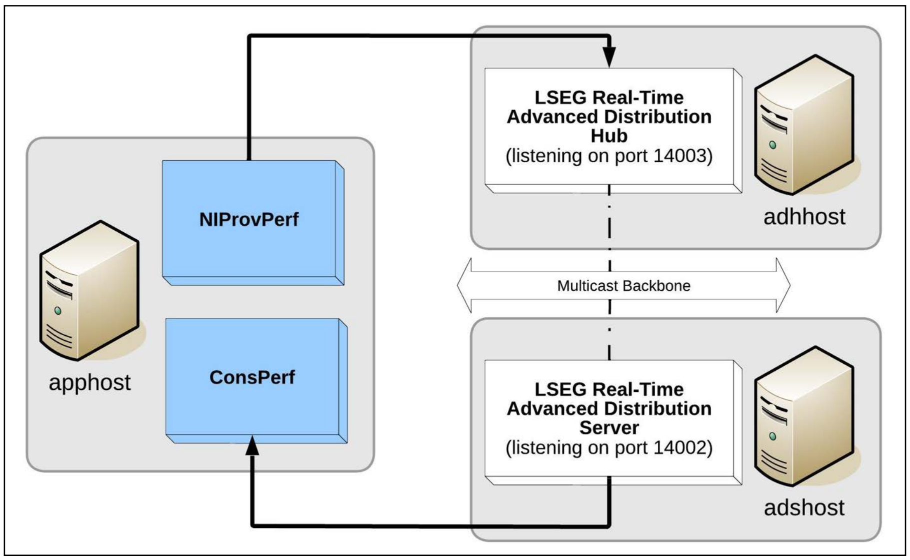

Figure 22. Consumer Posting to LSEG Real-Time Distribution System

In this case, **NIProvPerf** provides service and items. Update traffic is optional. If you want to test posting without updates, configure **NIProvPerf** by specifying **`-updateRate 0 -latencyUpdateRate 0`** in the command line.

Additionally, if you want only posting traffic, you do not need to run a provider application. You can configure the LSEG Real-Time Distribution System to provide the necessary service information and refresh content. For more details on this configuration, refer to the *LSEG Real-Time Advanced Distribution Hub Software Installation Manual*.

1. Configure **Perf_NIP_Channel_1**, change <Host value="*adhhost*"/>, <Port value="*14003*"/>.
2. Configure **Perf_Channel_1**, change <Host value="*adshost*"/>, <Port value="*14002*">.
3. Configure **Perf_Directory_1**, change <Service><Name value="*TEST_FEED*"/>.

To run a basic performance measurement using a setup like the one pictured in Figure 22, run **NIProvPerf** and **ConsPerf** with the following command-line options. These options assume TEST_FEED is the service being provided. Modify the example values as necessary.

```
java com.refinitiv.eta.perftools.niprovperf.NIProvPerf -h adhhost -p 14003 -serviceName TEST_FEED
java com.refinitiv.eta.perftools.consperf.ConsPerf -h adshost -p 14002 -serviceName TEST_FEED
```

8.5 Transport Performance, Direct Connect with TCP

You can measure the performance of the transport[1] in a direct connect scenario. This test can determine the throughput and latency you can achieve using different transport types supported in the Enterprise Transport API. This scenario shows the TCP Socket transport type.

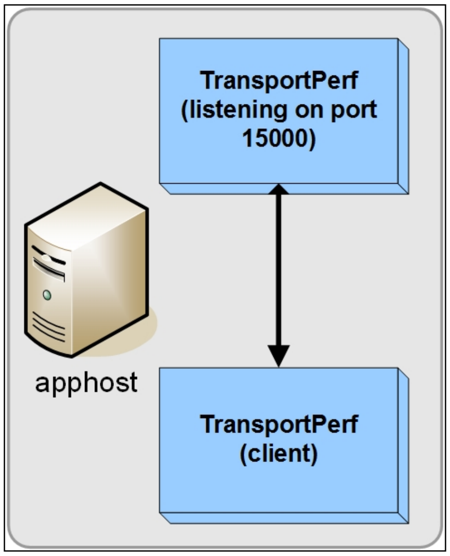

Figure 23. Transport Performance, TCP Direct Connect

▶ To run a basic performance measurement:

To run a basic performance measurement using the setup pictured in Figure 23, run two instances of **TransportPerf** as follows:

```
TransportPerf -p 15000 -runTime 900 -connType socket -msgRate 100000 -latencyMsgRate 1000
        -tickRate 1000 -appType server

TransportPerf -p 15000 -connType socket -runTime 60 -latencyMsgRate 0 -msgRate 0 -tickRate 1000
        -appType client
```

1. Other than timestamp and sequence number encoding and decoding, this application does not perform other content operations or inspections.

8.6 Transport Performance, Direct Connect with TCP, Reflection

To measure latency across hosts, **TransportPerf** supports reflecting messages. When reflecting is enabled, instead of **TransportPerf** producing its own messages, each read message is written back as-is to the same connection. Reflecting is enabled by the **`-reflectMsgs`** command-line option and may be done by servers or clients.

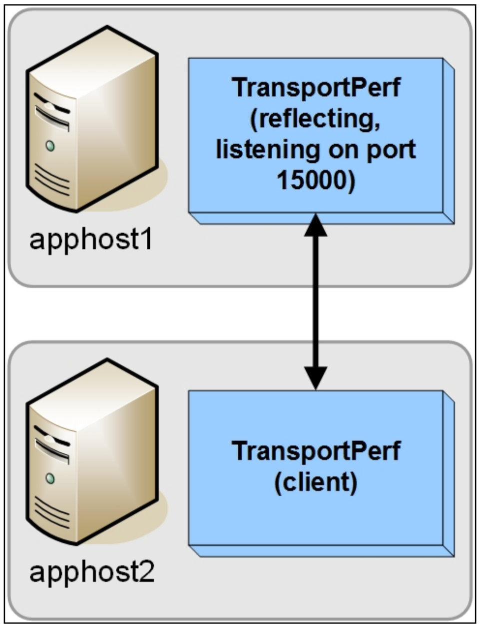

Figure 24. Transport Performance, TCP Direct Connect with Reflection

▶ To run a Basic Performance Measurement:

To run a basic performance measurement using a setup like the one pictured in Figure 24, run two instances of **TransportPerf** with the following command-line options:

```
TransportPerf -p 15000 -runTime 900 -connType socket -tickRate 1000 -appType server -reflectMsgs

TransportPerf. -h apphost1 -p 15000 -connType socket -runTime 60 -tickRate 1000 -appType client
        -msgRate 100000 -latencyMsgRate 1000
```

8.7 Transport Performance, Direct Connect with Multicast

Because of the nature of multicast, both applications are configured as clients. Also, this example uses separate 'send' and 'receive' networks.

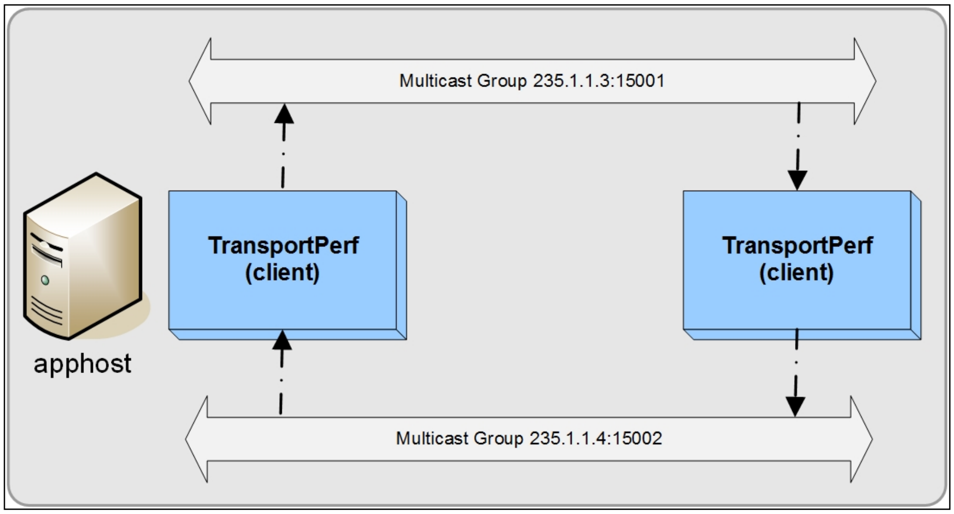

Figure 25. Transport Performance, Multicast Direct Connect

▶ To run a basic performance measurement:

To run a basic performance measurement using a setup like the one pictured in Figure 25, run two instances of **TransportPerf** as follows:

```
TransportPerf -runTime 90 -connType reliableMCast -u 12345 -sp 15001 -sa 235.1.1.3 -rp 15002
        -ra 235.1.1.4 -latencyMsgRate 1000 -msgRate 10000 -tickRate 1000 -appType client

TransportPerf -u 14006 -sp 15002 -sa 235.1.1.4 -rp 15001 -ra 235.1.1.3 -connType reliableMCast
        -runTime 60 -appType client -msgRate 0 -latencyMsgRate 0 -tickRate 1000
```

8.8 Transport Performance, Direct Connect with Shared Memory

The following example uses a small **maxFragmentSize** to reduce the size of the shared memory segment and uses the **`-threads`** option. The provider and consumer each use one thread: the provider thread is bound to core 0, and the consumer thread is bound to core 1. Such a setup presumes the system has at least two cores.

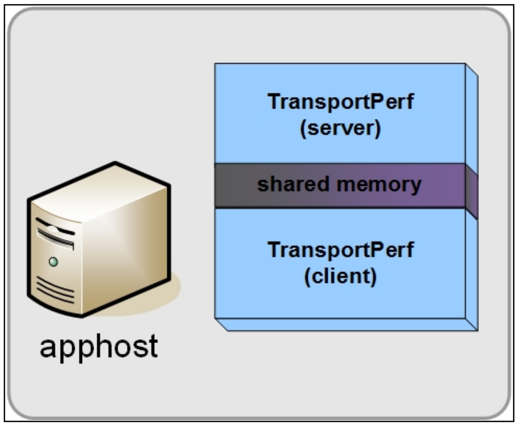

Figure 26. TransportPerf, Shared Memory Direct Connect

▶ To run a basic performance measurement:

To run a basic performance measurement using a setup like the one pictured in Figure 26, run two instances of **TransportPerf** as follows:

```
TransportPerf -p 15000 -runTime 90 -connType shmem -msgRate 10000 -latencyMsgRate 1000
        -tickRate 1000 -appType server -outputBufs 9000 -maxFragmentSize 256 -threads 0

TransportPerf -p 15000 -connType shmem -runTime 60 -latencyMsgRate 0 -msgRate 0 -tickRate 1000
        -appType client -threads 1
```

9 Input File Details

9.1 Message Content File and Format

The message data XML file (**MsgData.xml**) provided with the Performance Suite describes sample data for the refreshes, updates, and posts encoded by the tools. You can customize **MsgData.xml** to suit desired test scenarios.

The XML file must contain data for:

- One refresh message.
- At least one update message.
- At least one post message, if posting from **ConsPerf**.
- At least one generic message, if configured for exchanging generic messages.

Refresh data provides the image for each item provided by **ProvPerf** or **NIProvPerf**. When providing updates, provider tools encode update messages in a round-robin manner for each item. Likewise, when posting, the **ConsPerf** encodes posts in a round-robin fashion for each item.

9.1.1 Encoding Fields

Performance tools can encode in their fields any of the primitive types supported by the Enterprise Transport API.

For types such as **`Real.value(String)`** and **`Date.value(String)`**, the input follows the conversion format of the Enterprise Transport API's string conversion function (e.g. **`Real.value(String)`**, **`Date.value(String)`**,).

Each field must have the correct type for its ID according to the dictionary loaded by the tool. Fields are validated by the message data parser.

9.1.2 Sample Update Message

```
<updateMsg>
    <dataBody>
        <fieldList entryCount="13" entryCount="23">
            <fieldEntry fieldId="22" dataType="RSSL_DT_REAL" data="2848.560000"/>
            <fieldEntry fieldId="25" dataType="RSSL_DT_REAL" data="2849.610000"/>
            <fieldEntry fieldId="30" dataType="RSSL_DT_REAL" data="1"/>
            <fieldEntry fieldId="31" dataType="RSSL_DT_REAL" data="1"/>
            <fieldEntry fieldId="6579" dataType="RSSL_DT_RMTES_STRING" data="R"/>
            <fieldEntry fieldId="6580" dataType="RSSL_DT_RMTES_STRING" data="R"/>
            <fieldEntry fieldId="114" dataType="RSSL_DT_REAL" data="13.340000"/>
            <fieldEntry fieldId="1000" dataType="RSSL_DT_RMTES_STRING" data=" "/>
            <fieldEntry fieldId="8937" dataType="RSSL_DT_ENUM" data="0"/>
            <fieldEntry fieldId="211" dataType="RSSL_DT_REAL" data="31701"/>
            <fieldEntry fieldId="118" dataType="RSSL_DT_ENUM" data="0"/>
            <fieldEntry fieldId="3264" dataType="RSSL_DT_ENUM" data="0"/>
            <fieldEntry fieldId="3887" dataType="RSSL_DT_REAL" data="39100330"/>
            <fieldEntry fieldId="8935" dataType="RSSL_DT_ENUM" data="1"/>
            <fieldEntry fieldId="1501" dataType="RSSL_DT_RMTES_STRING" data=" "/>
            <fieldEntry fieldId="12783" dataType="RSSL_DT_ENUM" data="4"/>
            <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="57132000"/>
            <fieldEntry fieldId="1025" dataType="RSSL_DT_TIME" data="15:52:12:000:000:000"/>
            <fieldEntry fieldId="5" dataType="RSSL_DT_TIME" data="15:52:00:000:000:000"/>
            <fieldEntry fieldId="8406" dataType="RSSL_DT_RMTES_STRING" data=" "/>
            <fieldEntry fieldId="1041" dataType="RSSL_DT_RMTES_STRING" data=" "/>
            <fieldEntry fieldId="203" dataType="RSSL_DT_REAL" data="2848.560000"/>
            <fieldEntry fieldId="14238" dataType="RSSL_DT_TIME" data="15:52:12:000:000:000"/>
        </fieldList>
    </dataBody>
</updateMsg>
```

Code Example 14: Sample Update Message

9.2 Item List File

The Item List File configures the full list of items as requested by **ConsPerf** . Each entry specifies the item’s name and how it is requested. The file must contain enough entries to satisfy the number of items needed by the respective tool.

The sample file **350k.xml** contains 350,000 items, some of which allow posting.

9.2.1 Item Attributes

ATTRIBUTE NAME	DEFAULT	DESCRIPTION
domain	(none, required)	Specifies the domain from which the item is requested. This must be set to **MarketPrice**.
genMsg	“false”	If set to true, generic messages are sent for this item (if generic messages are enabled).
name	(none, required)	Specifies the name used in the MsgKey when requesting the item.
post	“false”	If set to **true**, **ConsPerf** sends posts to this item (if posting is enabled).
snapshot	“false”	If set to **true**, **ConsPerf** requests this item as a snapshot (i.e., without setting the **STREAMING** flag on the request).

Table 6: Item Attributes

9.2.2 Sample Item List File

```
<itemList>
    <item domain="MarketPrice" name="RDT1" post="true" genMsg="true" />
    <item domain="MarketPrice" name="RDT2" post="true" />
    <item domain="MarketPrice" name="RDT3" post="true" />
    <item domain="MarketPrice" name="RDT4" post="true" />
    <item domain="MarketPrice" name="RDT5" post="true" />
    <item domain="MarketPrice" name="RDT6" post="true" />
    <item domain="MarketPrice" name="RDT7" post="true" />
    <item domain="MarketPrice" name="RDT8" />
    <item domain="MarketPrice" name="RDT9" />
    <item domain="MarketPrice" name="RDT10" />
    <item domain="MarketPrice" name="RDT11" />
    <item domain="MarketPrice" name="RDT12" />
    <item domain="MarketPrice" name="RDT13" />
    <item domain="MarketPrice" name="RDT14" />
    <item domain="MarketPrice" name="RDT15" />
    <item domain="MarketPrice" name="RDT16" />
    <item domain="MarketPrice" name="RDT17" />
    <item domain="MarketPrice" name="RDT18" />
</itemList>
```

Code Example 15: Sample Item List File

10 Output File Details

10.1 Overview

Applications in the Performance Suite send similar output to the console and to files. Each application can configure its output using the configuration parameters:

- **writeStatsInterval** (1 to *n*): The interval (in seconds) at which timed statistics are written to files.
- **noDisplayStats**: Disables statistics output to the console.

Providers and consumers output different statistics but in a similar fashion. Each application can be configured to output a summary file, a statistics file, and in the case of the consumer, a latency file comprised of individual latencies for each received latency item.

10.2 Output Files and Their Descriptions

You can configure the names of output files, though applications append the client number to their stats and latency files. So for example, a horizontal scaling test with two consumer threads produces two statistics files: **ConsStats1.csv** and **ConsStats2.csv**.

Default output filenames (and the associated parameters you use to generate the files) are as follows:

PARAMETER	DEFAULT	DESCRIPTION
-latencyFile	(none)	Specifies the filename of the latency file produced.
-statsFile	*ToolType*Statsclient.csv[a]	Specifies the filename of the statistics file produced.
-summaryFile	*ToolType*Summary.txt	Specifies the filename of the summary file produced.

Table 7: Performance Suite Applications and Associated Configuration Files

a. Where *ToolType* is either **Cons**, **IProv**, **NIProv**, or **Transport**.

10.3 Latency File

The latency file is a comma-separated value file containing individual latencies, in microseconds, for timestamps received during the test. It is only created by **ConsPerf**.

NOTE: Due to the potentially large amount of output in scenarios that use a high latency message rate, this file is not produced by default.

The interval in seconds that statistics are written to the file is controlled by the **writeStatsInterval** configuration parameter, which defaults to **5**.

```
Message type, Send time, Receive time, Latency (usec)
Upd, 353725032296, 353725032521, 225
Upd, 353725045319, 353725045569, 250
Upd, 353725092300, 353725092521, 221
Pst, 353724892323, 353724894740, 2417
Pst, 353724925257, 353724926441, 1184
Pst, 353725105324, 353725106762, 1438
Upd, 353725359645, 353725359859, 214
Upd, 353725610354, 353725610619, 265
```

Code Example 16: Sample ConsLatency.csv Showing Update and Post latencies during a Test Run

10.4 File Import

You can import output **.csv** files into data analysis software. For example, you can use Microsoft Excel and Microsoft Access to import and quickly analyze your test results. Shown below are graphs created in Excel after importing a statistics **.csv** file for a test run. Note that these are sample graphs and do not imply the real performance results of the tool suite.

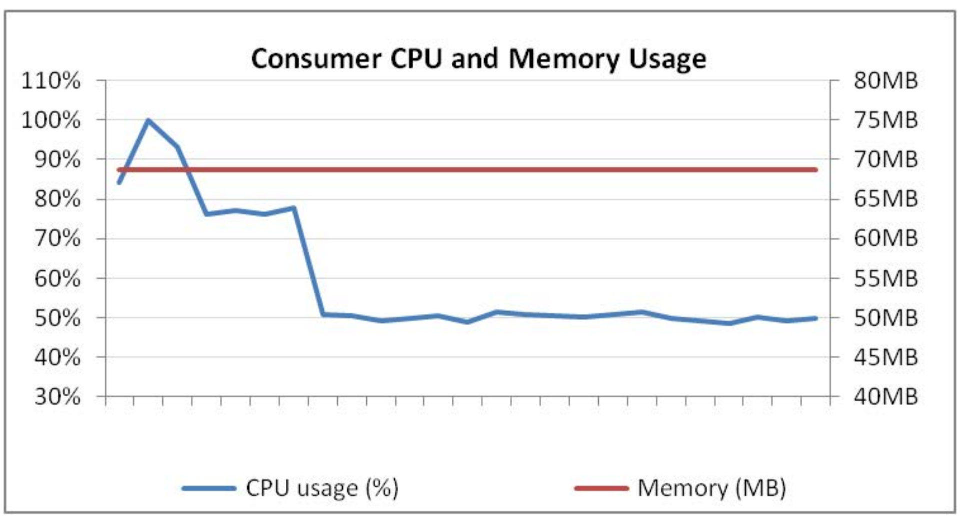

Figure 27. Sample Excel Graph from ConsStats1.csv

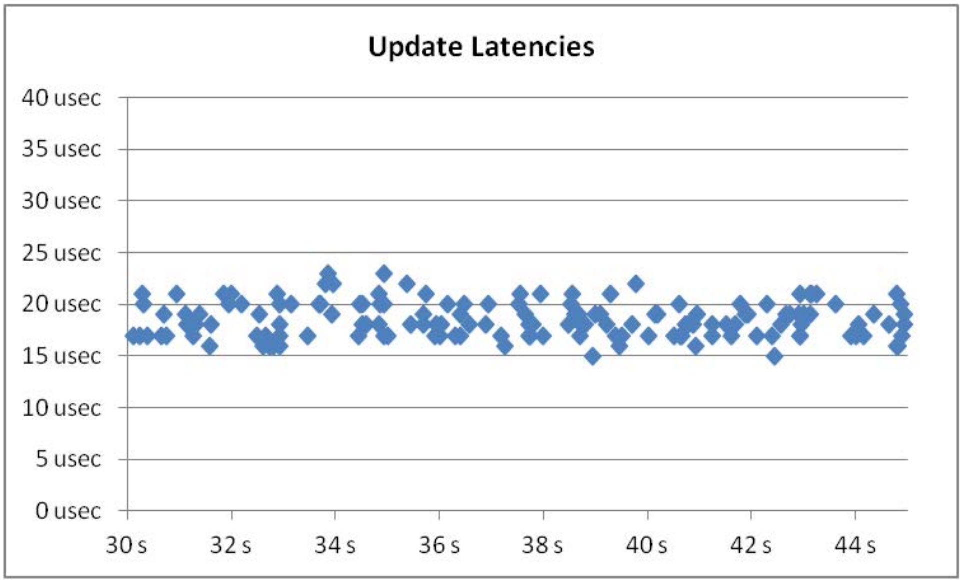

Figure 28. Sample Excel Graph of Latencies Over a 15-second Steady State Interval from ConsLatency1.csv

11 Performance Best Practices

11.1 Overview

The Performance Test Tools Suite leverages a number of features of the Enterprise Transport API to achieve high throughput and low latency when sending and receiving messages. This section briefly describes test tool features, the features' benefits, and how the tools use them. For more details on each feature, refer to the Enterprise Transport API Configuration Guide.

11.2 Transport Best Practices

11.2.1 rsslRead

When calling `Channel.read()`, the **ReadArgs.readRetVal** parameter indicates whether more content is available for immediate processing. Because this content might have already been read from the underlying transport, an I/O notifier might not inform an application that this content is available. When the value is greater than **TransportReturnCodes.SUCCESS** (**0**), the application should call `Channel.read()` again without waiting for further notification. Even if no message was returned, `Channel.read()` can return a value greater than **TransportReturnCodes.SUCCESS**.

All applications follow a reading model that leverages the Enterprise Transport API's queuing and notification mechanisms to read quickly without making unnecessary function calls:

1. Wait for notification on the descriptors associated with a channel using its preferred I/O notifier. The application will only call `Channel.read()` on channels that indicate the presence of readable data.
2. Keep calling `Channel.read()` as long as the return value is greater than **TransportReturnCodes.SUCCESS**. Such return values indicate that there is still data to process in the Enterprise Transport API's input buffer.

11.2.2 rsslWrite, rsslFlush

To make efficient use of underlying transport method calls, the **Channel.write() Submit** method passes messages to an outbound queue of the specified priority, rather than immediately writing the message to the network. **Channel.write()** indicates that there is queued content by returning a value greater than **TransportReturnCodes.SUCCESS**.

The network write occurs if:

- The application calls **Channel.flush()** Enterprise Transport API internally calls **Write()** on the Channel instance in the Transport layer
- The **WriteFlags.DIRECT_SOCKET_WRITE** flag is passed into the **Channel.write()** method.
- The amount of queued data reaches a configurable highwater mark, i.e., using the **-highWaterMark** option**HighWaterMark** configuration parameter in Channel or Server group causes **Channel.write()** **Submit** to pass queued content to the underlying transport.

You can use a simple algorithm to write large amounts of content efficiently while still maintaining low latency:

- Write all currently desired content via **Channel.write()**, relying on the highwater mark to periodically flush.
- When there is no more content to write, call **Channel.flush()** to flush any remaining data. After all data is written to the network, **Channel.flush()** returns **TransportReturnCodes.SUCCESS**.

NOTE: A positive value returned from `Channel.write()` means there is queued content. It is not passed to the underlying transport unless the application performs one of the previously mentioned actions.

Enterprise Transport API Performance Tools observe the following model when writing message bursts:

1. Write the entire burst of messages using **Channel.write()**.
2. After the burst finishes, check the return value from the last call to **Channel.write()**. If the value is greater than **TransportReturnCodes.SUCCESS**, data remains in the Enterprise Transport API's output queue. Use the channel's selector in conjunction with an I/O notifier to notify the application when it can flush remaining data.
3. Call **Channel.flush()** on channels indicated by the notifier. Continue to invoke the notifier and **Channel.flush()** until all data is successfully provided to the underlying transport.

Each application has a command-line option (**-highWaterMark**) for adjusting the high-water mark.

11.2.3 Packing

To efficiently use buffer space, reduce the number of writes to the transport,[1] and improve throughput, applications can use the `Channel.packBuffer()` method to pack multiple messages into a single buffer. Each call to `Channel.packBuffer()` adds another message to the existing buffer. Though packing messages can help increase throughput, overall effects will vary depending on the type of transport you use. In general:

- Slower transports benefit more than faster transports
- More saturated transports benefit more than less-saturated ones.

Because packing adds latency, it is important that you weigh the trade-offs when deciding to employ packing. For example, if 15 messages are packed into a single buffer, the first message will sit in the buffer longer before being written than the last message added to the buffer. An application can reduce this latency by sending the packed buffer before it is full, often through the use of a timing mechanism that indicates a latency threshold has been reached. In general, the more messages packed into a buffer, the higher the latency penalty.

ProvPerf, and **NIProvPerf**, and **TransportPerf** perform the following steps when configured to pack buffers:

1. Get a packable buffer of a specified size, using `Channel.getBuffer()`.
2. Encode messages into the buffer, calling `Channel.packBuffer()` after each message is encoded.

After the configured number of messages are packed or when the buffer is full, `Channel.write()` is called to pass the packed buffer to the Enterprise Transport API.

1. Reducing underlying transport header overhead.

11.2.4 High-water Mark

Higher throughput is usually achieved by making a small number of large writes to the transport instead of doing a large number of small writes. For example, writing one 6000-byte buffer is generally more efficient than writing 1000 six-byte buffers. To achieve higher efficiencies, the Enterprise Transport API employs the concept of a high-water mark. When the application calls **Channel.write() Submit**, the Enterprise Transport API does not always immediately pass the buffer to the transport; instead, the Enterprise Transport API passes data to the transport after the size of its buffer reaches the high-water mark.[2] Note that the high-water mark applies to written data and not to the number of bytes requested in the call to **Channel.getBuffer()**.

For example, assume a high-water mark of 6144 bytes. If an application gets a buffercreates a message, encodes 500 bytes of content, and passes this to **Channel.write() Submit**, the high-water mark will be triggered after thirteen buffers. At that point, the Enterprise Transport API's output queue will contain thirteen buffers, each with approximately 500 bytes that it can pass to the underlying transport, instead of passing one at a time.

You can configure each individual connection's high-water mark. The application can also invoke data being passed to the underlying transport by using the **Channel.flush()** routine or passing the **WriteFlags.DIRECT_SOCKET_WRITE** flag to **Channel.write()**.

Note the throughput and latency implications. Balance the use of the high-water mark and **Channel.flush()**flush by Enterprise Transport API accordingly:

- In high-throughput situations, it is better to make large writes to achieve higher efficiencies (i.e., in this case use the high-water mark).
- In low-throughput situations, data might linger in Enterprise Transport API queues for longer periods and thus incur latency (i.e., in this case, use **Channel.flush()**).

11.2.5 Direct Socket Write

On the Enterprise Transport API level, the **Channel.write()** method can be instructed to attempt to pass data directly to the underlying transport by specifying the **WriteFlags.DIRECT_SOCKET_WRITE**flag when calling **Channel.write()**. This flag causes the Enterprise Transport API to check its current outbound queue depths:

- If the queues are empty, the Enterprise Transport API passes data directly to the underlying transport, bypassing all queuing logic and delays.
- If the queues are not empty, the Enterprise Transport API adds data to the appropriate queue, with queued content being passed to the underlying transport in the appropriate order.[3]

Enterprise Message API allows to configure the Direct Write option for the Transport layer. Using this option can reduce latency,[4] as the message might not get queued. However this option also reduces throughput and increases CPU usage due to more frequent socket writes. You can offset the loss in throughput by packing buffers, though doing so can increase packing latency.

11.2.6 Nagle's Algorithm

For TCP socket connection types, you can set the underlying transport to use Nagle's Algorithm to combine small content fragments into larger network frames. While this algorithm reduces transport overhead (optimizing bandwidth usage), it also increases latency, especially when sending small messages at lower data rates.

To minimize latency, the Performance Tools use **tcp_nodelay** (an **ConnectOptions** parameter), which disables Nagle's Algorithm. However, you can reenable this feature using the **-tcpDelay** option. For further details, refer to each tool's list of available options.

---

2. As previously noted, the application can invoke data being passed to the transport through the use of `Channel.flush()` or the **WriteFlags.DIRECT_SOCKET_WRITE** flag.
3. As determined by the various priorities with which the content was written and the flush strategy you configure.
4. As long as the underlying transport can accept the content.

11.2.7 System Send and Receive Buffers

For TCP socket connections, the OS uses system send and receive buffers for exchanging content. When the Enterprise Transport API flushes data to the underlying transport, it passes through these system buffers. During times of high throughput, the application might provide data faster than the underlying transport can send it. If this happens, the system buffers can fill up, and as a result, the underlying transport refuses to accept data. In this case, the transport accepts new data only after some of its buffered content is sent and acknowledged.

If the user instructs the Enterprise Transport API to pass queued data to the underlying transport but the OS cannot accept additional content at the time, **Channel.write()** or **Channel.flush()** will return a positive return value. A positive value indicates that content has been queued in the Enterprise Transport API and the application Enterprise Transport API should flush it at a subsequent time. However, this state is not considered a failure condition, and the Enterprise Transport API still has the data in its buffers. In this situation, the OP_WRITE selection key write file descriptor of the connection can be added to the channel's selector, which notifies the application when it can pass additional content to the OS.

You can configure the system's send and receive buffer sizes in the OS, as detailed in OS-specific documentation. Additionally, the Enterprise Transport API allows users to configure this via the **ConnectOptions**, **BindOptions**, and to dynamically increase or decrease buffer size during runtime using the **Channel.ioctl()** call.

11.2.8 Enterprise Transport API Buffering

The Enterprise Transport API uses various optimization techniques for efficient input and output of content, many revolving around pre-allocated buffers which minimize memory creation and destruction. Pre-allocated buffers queue outbound data as well as read large byte-streams from underlying transports.

When a connection is established, the maximum size buffer is negotiated, allowing the Enterprise Transport API to create input and output buffers that work well with respect to that connection. Because input and output strategies have different challenges, these pre-allocated buffer pools are handled differently depending on whether they are input or output buffers.

11.2.8.1 Input Buffering

The Enterprise Transport API input buffer is created as one large continuous block of memory, controlled by **numInputBuffers** (configured via **ConnectOptions** or **BindOptions**). The number of bytes created in the input buffer is determined by the configured value multiplied by the negotiated **maxFragmentSize**. Having one large block of memory allows **IChannel.Read() Channel.read()** to get as many bytes from a single call to the underlying transport as possible. When the input buffer holds data, the Enterprise Transport API determines message boundaries and returns a single message to the user. As the application makes subsequent **IChannel.Read()Channel.read()** calls, additional messages are dispatched from the input buffer. After fully processing the input buffer, the Enterprise Transport API goes back to the underlying transport to again fill the input buffer.

The intent is to have the Enterprise Transport API read only when needed and to read as much as possible. The amount of data the Enterprise Transport API actually reads from the network depends on the number of input buffers and the amount of data that the OS has available at that time.

11.2.8.2 Output Buffering

Output buffering is handled differently from input buffering. Because each buffer can be written as a different priority, a continuous block of memory will not work. The Enterprise Transport API creates the configured number of buffers, treating each buffer as a separate entity. Such a division allows the use of multiple buffers simultaneously, as well as allowing buffers to co-exist in different priority-based output queues.

You should configure the number of output buffers according to the application's expected output load. The **guaranteedOutputBuffer** setting controls the number of output buffers available exclusively to that channel, where all of these buffers are created up-front. When the channel runs on a server application, you can also configure the **maxOutputBuffers**, allowing the application to use buffers from the server's shared buffer pool. The shared pool is grown on-demand, but allows for connections under heavy load to temporarily grab a buffer for use. When returned to the shared pool, another channel can then use this same buffer. This enables the server to balance the memory usage of pre-allocated guaranteed output buffers with the traffic spike tolerance of a dynamic shared buffer pool.

Increasing the number of output buffers can improve performance when sending high volumes. An application should be aware of trade-offs of using too much memory and thus potentially slowing the process. If the receiving process cannot keep up with the send rate, a condition can develop for the sender where all output buffers are in use, waiting to be transmitted. The number of output buffers can be dynamically grown using **Channel.ioctl()** or, more commonly, the connection is terminated as a result.

11.2.8.3 Fragmentation

The negotiated maximum buffer size is the maximum size that the application will send in a single buffer. In cases where an application uses **Channel.getBuffer()** to request a buffer with a size larger than the maximum, the requested size will be returned to the user. When the content passes to **Channel.write()**, the Enterprise Transport API fragments the content on behalf of the application, breaking apart larger content into individual buffers whose individual sizes do not exceed the agreed upon maximum. On the receiving side, the Enterprise Transport API reassembles the fragments back into a single buffer containing all relevant content.

This transport level fragmentation incurs multiple copies and potential memory allocations. To avoid such overhead, applications should ensure that the maximum buffer size (**BindOptions.MaxFragmentSize**) is large enough for commonly sent messages to fit into a single buffer.

11.2.9 Compression

The Enterprise Transport API supports the use of data compression. Generally, compressing data reduces the amount of data passed to the underlying transport. But compression has some drawbacks to consider:

- Compression requires additional processing.[5]
- Compression copies data: as the user-provided buffer is read by the compression algorithm, output data is compressed into a different buffer. As a result, compression will generally require more buffers from the Enterprise Transport API's buffer pool.

TransportPerf exposes the option to enable compression using **-compressionType** and **-compressionLevel**.

11.3 Encoder and Decoder Best Practice: Single-Pass Encoding

Enterprise Transport API Performance Tools encodes data so as to minimize copying. Thus, the application encoding process begins by starting with the top-level container and working down in a linear fashion.

For example, when encoding a Market Price message, the message header is encoded (**`Msg.encodeInit()`**), followed by the field list payload (**`FieldList.encodeInit()`** … **`FieldList.encodeComplete()`**). After the payload is encoded, message encoding is completed (**`Msg.encodeComplete()`**).

Encoding the field list prior to the message header would require it to be encoded into a temporary buffer which would then be passed to the message encoder (**`Msg.encode()`**). This approach would incur multiple buffer allocations and copies to complete encoding.

11.4 Other Practices: JVM Priming

The JVM performs just-in-time compilation and optimization of executing bytecode. This takes processing time and affects latency values during the start-up state of performance applications. JVM priming reduces the impact of this start-up overhead and can be enabled by using the **`-primeJVM`** option.

JVM priming is accomplished by sending a snapshot request for all items before sending the actual streaming requests for the items. Latency measurements are only taken for updates so refreshes from the snapshot requests are used to prime the JVM. This results in lower latency values in the start-up state.

5. Overhead will vary based on the type of compression used and the level of compression applied.

Appendix A Troubleshooting

A.1 Can’t Connect

There are many reasons why a consumer or provider might not be able to connect. Several common ones are listed below:

- Check the consumer’s and provider’s **serviceName** parameters. These must match. The consumer will wait until the service is available and accepting requests.
- Check the LSEG Real-Time Advanced Distribution Hub (**adhmon**) and LSEG Real-Time Advanced Distribution Server (**adsmon**) to see whether the desired service is up.
- Check the LSEG Real-Time Advanced Distribution Hub’s configuration to make sure that the provider’s host is listed in the **hostList** configuration setting.
- Check that the provider is listening on the correct TCP Port.
- Check that the consumer is connecting to the correct **hostName** and TCP Port.
- In direct-connect mode, start the provider first, then start the consumer. Starting the consumer first results in a connection timeout, which creates a (by default) 15 second delay until the client retries the connection attempt.
- When connecting through LSEG Real-Time Distribution System, check that the desired service is up on both the LSEG Real-Time Advanced Distribution Hub and LSEG Real-Time Advanced Distribution Server before starting the consumer (or wait the appropriate amount of time.) Starting the consumer too quickly results in a connection retry after (by default) 15 seconds.

A.2 Not Achieving Steady State

There are several reasons why a consumer might not reach a steady state:

- The **steadyStateTime** value may be too small. When publishing in latency mode or at high update rates, providers will take longer to process image requests. For example, if **steadyStateTime** is set to **30s** but the provider can publish only 2,500 images per second, the consumer times out before it receives its 100,000 images.
- The provider might be overloaded. If the provider is publishing at or near 100% CPU for its configured update rate, it will be either unable or barely able to service incoming image requests, which causes images to trickle back to the consumer.
- The consumer might be overloaded. At higher update rates, allocating a higher min and max heap space on the JVM can help alleviate issues.
- If using a non-interactive provider application, the provider and consumer watchlists might not match, resulting in the consumer application requesting items that never appear in the LSEG Real-Time Advanced Distribution Hub cache.

A.3 Consumer Tops Out but Not at 100% CPU

In some cases, when connecting to LSEG Real-Time Distribution System, the consumer appears to be overloaded even though no thread is using the maximum CPU. Such a situation might be a symptom of a bottleneck on the LSEG Real-Time Advanced Distribution Server, which can be resolved by increasing the size of the **guaranteedOutputBuffers** and **maxOutputBuffers** to 5,000 in **distribution.cnf**:

```
[...]
*ads*maxOutputBuffers : 5000
*ads*guaranteedOutputBuffers : 5000
[...]
```

Figure 29. LSEG Real-Time Advanced Distribution Server distribution.cnf

While this may increase the overall throughput, it can also increase message latency.

A.4 Initial Latencies Are High

- Initial latencies during startup and immediately following the transition to steady state might be high. At high update rates, the system processes its entire overhead for updates plus all refresh traffic, resulting in an increased workload and higher latency. It can take several seconds for the system to “settle” following the transition to steady state. Increasing the provider’s output buffers might help.
- Some older systems may not have a reliable timer that the tools can use to measure latencies. Try using a newer system and/or newer OS.The JVM performs just-in-time compilation and optimization of executing bytecode. This takes processing time and affects latency values during the start-up state of performance applications. JVM priming is used to reduce the impact of this start-up overhead. This feature can be enabled by using the **`-primeJVM`** option.
-

A.5 Latency Values Are Very High

- Run the applications on the same machine.
- Use a reliable clock to gather timestamp information.
- Perform appropriate system-wide tuning.
-
- Consider packing messages into the same buffer. It is possible that the connection type cannot sustain the data rate when sent as individual messages.

Document ID: ETAJ390L2PETOO.250
Date of issue: May 2025