# Enterprise Message API C++ Edition 3.9.2.L1

OPEN SOURCE PERFORMANCE TOOLS GUIDE

Document Version: 3.9.2.L1
Date of issue: December 2025
Document ID: EMAC392L1PETOO.250

# Contents

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 About this Manual TEST

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

Authors and contributors include Enterprise Message API architects and developers who encountered and resolved many of issues you might face. 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. It describes how features of the API send and receive data with high throughput and low latency. This information applies both when the API connects directly to itself as well as when using intermediaries, such as LSEG Real-Time Distribution System components like that LSEG Real-Time Advanced Distribution Hub and LSEG Real-Time Advanced Distribution Server.

## 1.2 Audience

This document is written to help programmers take advantage of Enterprise Message API features and 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 Enterprise Message 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 C programming language in a networked environment. It is assumed that the reader has read the *Message API C Edition Developer's Guide* to have a basic familiarity with the API Transport and the interaction models of OMM Consumers, OMM Interactive Providers, and OMM Non-Interactive Providers.

## 1.3 Programming Language

The Enterprise Message API C edition is written to the C language. All code samples in this document and all example applications provided with the product are written in C.

## 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 |
| Enterprise Message API (EMA) | The Enterprise Message API (EMA) is an ease of use, open source, Open Message Model API. EMA is designed to provide clients rapid development of applications, minimizing lines of code and providing a broad range of flexibility. It provides flexible configuration with default values to simplify use and deployment. EMA is written on top of the Enterprise Transport API (ETA) utilizing the Value Added Reactor and Watchlist features of ETA. |

**Table 1: Acronyms and Abbreviations**

| ACRONYM | DEFINITION |
|---|---|
| Enterprise Transport API (ETA) | Enterprise Transport API is a high performance, low latency, foundation of the LSEG Real-Time SDK. It consists of transport, buffer management, compression, fragmentation and packing over each transport and encoders and decoders that implement the Open Message Model. Applications written to this layer achieve the highest throughput, lowest latency, low memory utilization, and low CPU utilization using a binary Wire Format when publishing or consuming content to/from LSEG Real-Time Distribution Systems. |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | Hypertext Transfer Protocol (Secure) |
| NIC | Network Interface Card |
| OMM | Open Message Model |
| OS | Operating System |
| RAM | Random Access Memory |
| RDM | Domain Model |
| RFA | Robust Foundation API |
| RSSL | Source Sink Library |
| LSEG Real-Time Distribution System | LSEG Real-Time Distribution System is LSEG's financial market data distribution platform. It consists of the LSEG Real-Time Advanced Distribution Server and LSEG Real-Time Advanced Distribution Hub. Applications written to the LSEG Real-Time SDK can connect to this distribution system. |
| Reactor | The Reactor is a low-level, open-source, easy-to-use layer above the Enterprise Transport API. It offers heartbeat management, connection and item recovery, and many other features to help simplify application code for users. |
| RWF | Wire Format |

**Table 1: Acronyms and Abbreviations (Continued)**

## 1.5 References

- *Enterprise Message API C++ Edition Developers Guide*
- *Enterprise Message API C++ Edition LSEG Domain Model Usage Guide*
- *Enterprise Message API C++ Edition Configuration Guide*
- *Enterprise Transport API C Edition Value Added Components Developers Guide*
- The LSEG Developer Community

## 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

- C structures, methods, in-line code snippets, and types are shown in `orange, Lucida Console` 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 (one or more lines of code) are show in Lucida Console font against an orange background. Comments in the code are in green font. For example:

```
/* decode contents into the filter list structure */
if ((retVal = rsslDecodeFilterList(&decIter, &filterList)) >= RSSL_RET_SUCCESS)
{
    /* create single filter entry and reuse while decoding each entry */
    RsslFilterEntry filterEntry = RSSL_INIT_FILTER_ENTRY;
```

### 1.7.2 Diagrams

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

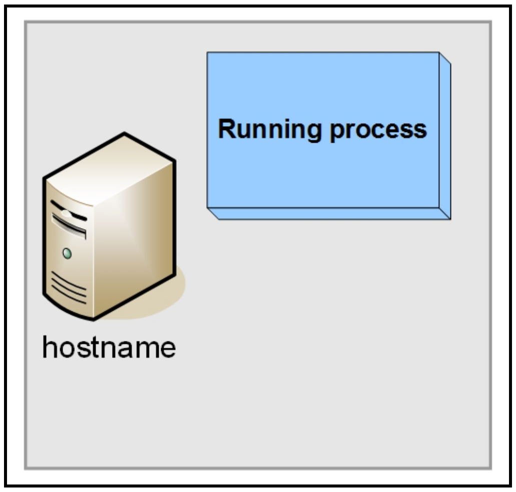


**Figure 1. Running Performance Example and Host Notation**

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

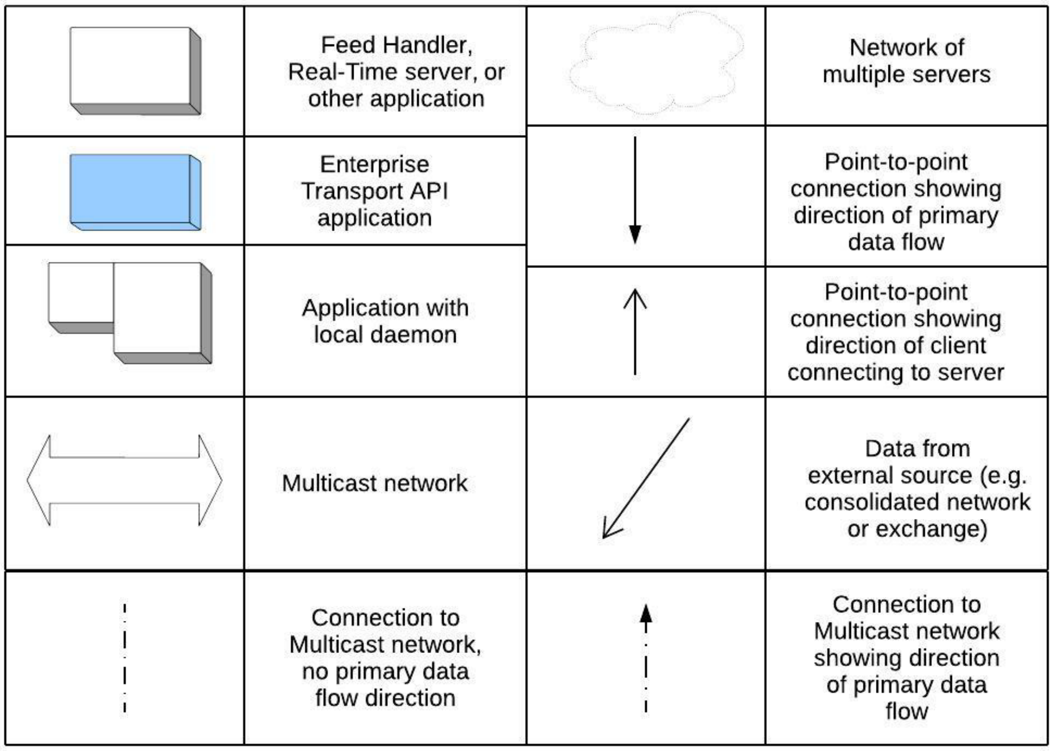

Feed Handler, Real-Time server, or other application
Network of multiple servers
Enterprise Transport API application
Point-to-point connection showing direction of primary data flow
Application with local daemon
Point-to-point connection showing direction of client connecting to server
Multicast network
Data from external source (e.g. consolidated network or exchange)
Connection to Multicast network, no primary data flow direction
Connection to Multicast network showing direction of primary data flow

**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 Enterprise Message API Performance Tool Suite

The Enterprise Message API C++-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 Message API provides a transport-only performance example which you can use to measure the performance of the Enterprise Message 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 Message API and a consumer from the Robust Foundation API).[1]

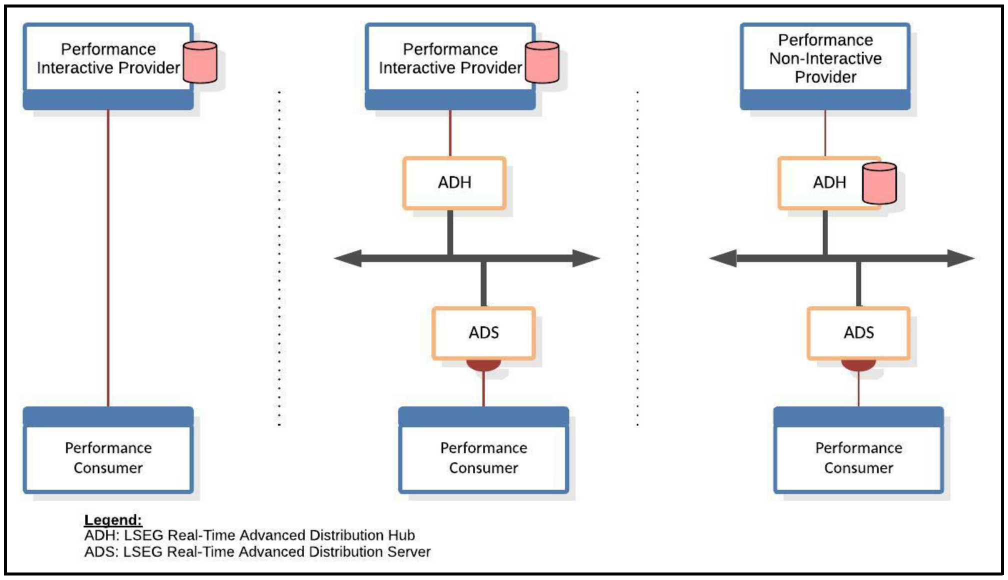

Performance Interactive Provider
Performance Interactive Provider
Performance Non-Interactive Provider
ADH
ADH
ADS
ADS
Performance Consumer
Performance Consumer
Performance Consumer
Legend:
ADH: LSEG Real-Time Advanced Distribution Hub
ADS: LSEG Real-Time Advanced Distribution Server

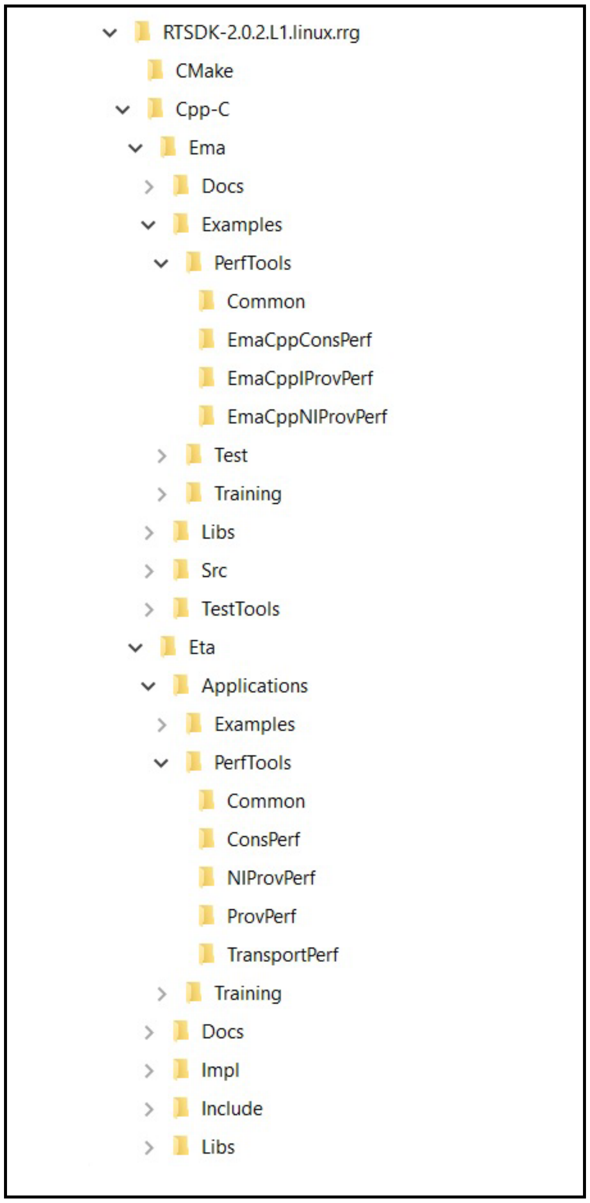
**Figure 3. Three Connection Options for the Open Message Model-based Performance Tools**

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.

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



```
v RTSDK-2.0.2.L1.linux.rrg
    CMake
  v Cpp-C
    v Ema
      > Docs
      v Examples
        v PerfTools
            Common
            EmaCppConsPerf
            EmaCppIProvPerf
            EmaCppNIProvPerf
        > Test
        > Training
      > Libs
      > Src
      > TestTools
    v Eta
      v Applications
        > Examples
        v PerfTools
            Common
            ConsPerf
            NIProvPerf
            ProvPerf
            TransportPerf
        > Training
      > Docs
      > Impl
      > Include
      > Libs
```

**Figure 4. Directory Structure of the Performance Tools**

Each example is distributed in its own directory.

**Libxml2**, an open source XML-parsing library, is also distributed along with the tools. Each example project is configured to build **Libxml2** as a dependent library.

The **PerfTools/common** directory includes two **.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 OMM messages.

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 **readme** files and comments included in source.

### 2.3.1 Building

On Linux, use `make` or `gmake` with the included **makefile** to build the examples.

On Windows, open the appropriate project in Visual Studio.

### 2.3.2 Running

On Linux, once built, change to the target directory. The **makefile** will have linked the necessary support files (**350k.xml**, **MsgData.xml**, **RDMFieldDictionary**, and **enumtype.def**) to the target directory.

On Windows, after the build process finishes, to run the tool, copy the above support files to the target directory.

## 2.4 What Is 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 Message API applications and LSEG Real-Time Distribution System components provide higher throughput and lower latency than Robust Foundation API-based applications. In general, LSEG recommends that you use a Enterprise Message API C performance tool to drive and calculate the performance of other non-Enterprise Message API C-based performance tools. For example, if you want to test the performance of the consumer, use the Enterprise Message API C interactive or non-interactive provider to drive the publishing rather than a 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 **Consumer** 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 & 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.

The CPU and Memory Usage calculations represent the process as a whole, and are not normalized to the number of CPU cores nor running threads. All threads in each tool contribute to the overall execution time; it is possible for the reported CPU usage to be greater than 100% if the application runs multiple busy threads.

#### 2.4.4.1 CPU Usage Calculation

CPU Usage is calculated by periodically querying the OS for applications' "busy" and "overall" times. The difference from the previous value is used to calculate an average CPU usage for each interval and presented as a percentage:

$$\frac{\text{User-space Execution Time + System-space Execution Time}}{\text{Running Time}}$$

#### 2.4.4.2 Memory Usage Calculation

Memory Usage is calculated using the "Resident Set Size," as provided by the respective operating system (OS). This measures the memory (in RAM) in use by the application.

## 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 **EmaCppConsPerf** 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, **EmaCppConsPerf** 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 **EmaCppConsPerf** 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:

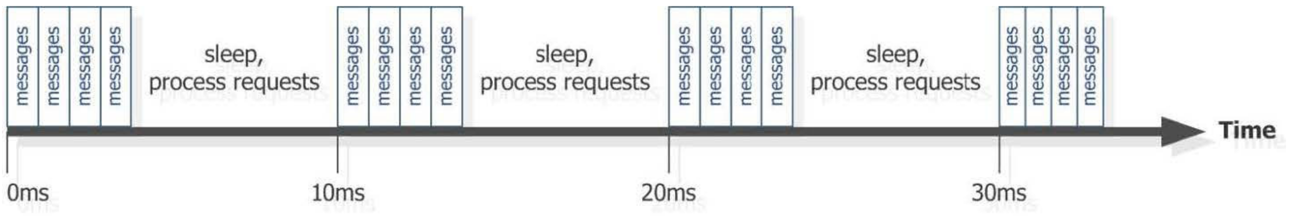


**Figure 5. Time Slicing Algorithm**

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

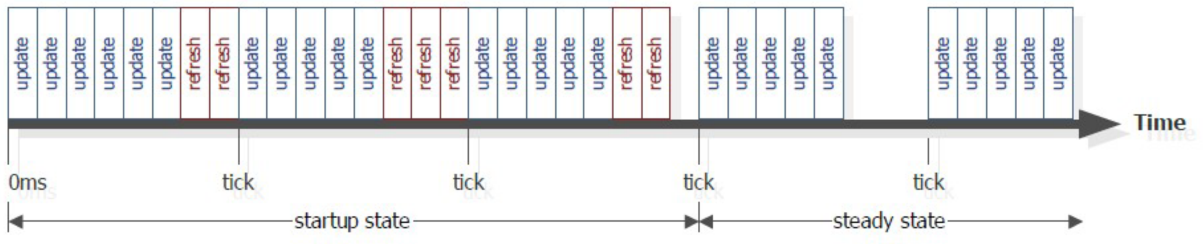


**Figure 6. Refresh Publishing Algorithm**

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.

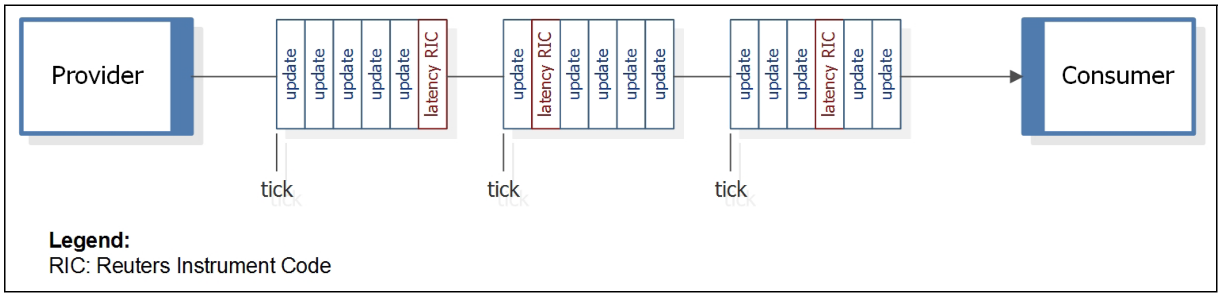


**Figure 7. Latency Instrument Codes within a Tick**

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 **EmaCppConsPerf** generates both sending and receiving timestamps.

**NOTE:** Open Message Model performance tool timestamp information contains the number of microseconds since an epoch.[a]

a. Windows uses `QueryPerformanceCounter()`, Linux uses `clock_gettime()` with the monotonic clock.

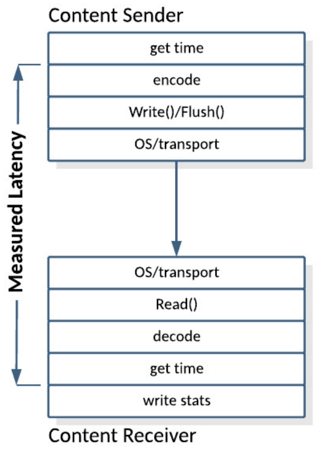


**Figure 8. Timing Diagram for Latency Measurements**

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.

# 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 **EmaCppConsPerf** implements an OMM consumer using the Enterprise Message API C++ Edition. It connects to a provider (such as **EmaCppIProvPerf** 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. **EmaCppConsPerf** uses Enterprise Message API to complete its start-up tasks. For more information, refer to the *Enterprise Message API Developers Guide*.

## 4.2 Threading and Scaling

The Enterprise Message 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 Message API.

Configure **EmaCppConsPerf** for multiple threads using the **`-threads`** command-line option. When multiple threads are configured, each thread opens its own connection to the provider. **EmaCppConsPerf** 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. Additionally, **EmaCppConsPerf** configures the Enterprise Message API to create an internal thread to dispatch received messages. You can set **EmaCppConsPerf** to not run the second thread inside the Enterprise Message API using **`-useUserDispatch`** command line option.

### 4.2.1 Consumer Lifecycle

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

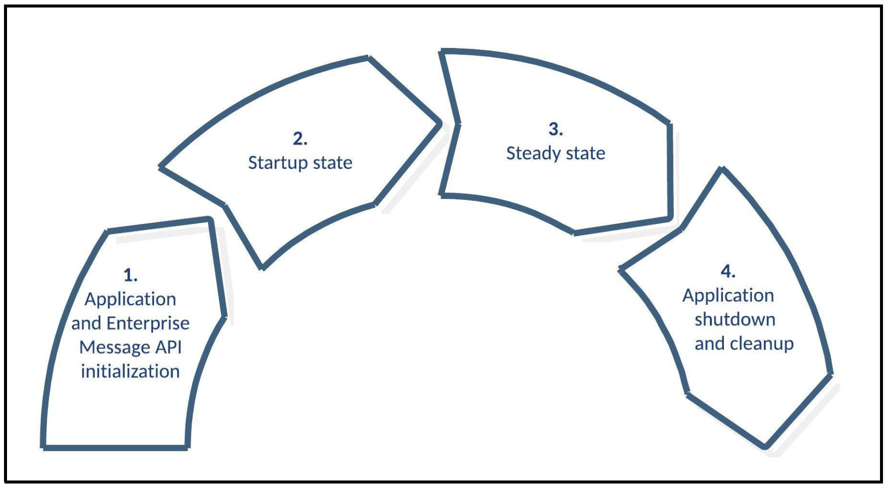

1.
Application and Enterprise Message API initialization
2.
Startup state
3.
Steady state
4.
Application shutdown and cleanup

**Figure 9. EmaCppConsPerf Lifecycle**

1. Application and Enterprise Message API Initialization.

   **EmaCppConsPerf** loads its configuration, initializes the Enterprise Message API, loads its item list using the specified file, and starts the thread(s) which connect to the provider to perform the test. The Enterprise Message API is configured by using **EmaConfig.xml**

   - 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 leverages an Enterprise Message API consumer to provide 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.
     - Processes 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.
   - Processes 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 Message API, any remaining resources, and exits.

### 4.2.2 Application Flow Diagram

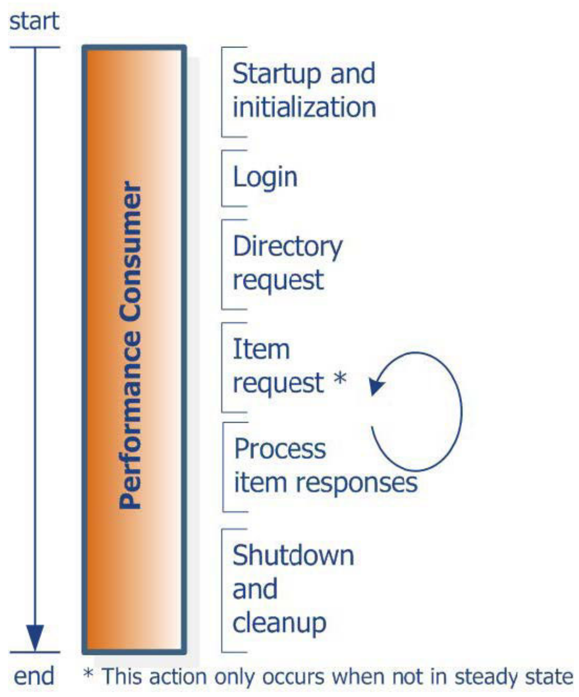


**Figure 10. EmaCppConsPerf Application Flow**

## 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 **EmaCppConsPerf** 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 **EmaCppConsPerf** 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, **EmaCppConsPerf** adds latency information to random post messages. When the posted content returns on the stream, **EmaCppConsPerf** 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 `rsslGetBuffer()`.
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 EmaCppConsPerf Configuration Options

**EmaCppConsPerf** uses **EmaConfig.xml** configuration file to setup an EMA consumer and set of command line options to configure specific behavior of performance tool.

**EmaConfig.xml** must have Consumer section in the Consumer group and appropriate Channel section in Channel group for correct configuration of the EMA consumer. For details on how to setup Consumer and Channel sections, refer to the *Enterprise Message API Configuration Guide*. For examples of configuration, refer to Section 4.5.1.

**EmaCppConsPerf** uses the command line option **`-consumerName`** to specify name of Consumer section.

The following table describes available configuration options.

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -apiThreads | -1 | Specifies the CPU core(s) to which the internal EMA API thread(s) will be bound to dispatch received messages. The parameter is used when the application configures EMA to work in API dispatch mode (see **`-useUserDispatch`**). **EmaCppConsPerf** does not bind the internal EMA API thread(s) to specific CPU core(s) by default. For details on the internal EMA API thread, refer to the *Enterprise Message API Developers Guide, 2.4 Product Architecture*.<br>Specifies the CPU physical mapping in format P:X C:Y T:Z, logical core ID (number), or no binding (-1).<br>Specifies the physical mapping which binds the thread to the specified physical processor, core, and thread (P:X C:Y T:Z). This syntax specifies a physical CPU to bind to. P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one.<br>-1 means no bind.<br>For example, when specified as "1,3", the internal EMA API threads will be bound to logical CPU cores 1 and 3.<br>The number of threads set by the parameter **`-threads`** should be matched.<br>For details on **`rsslBindThread`**, refer to the *Transport API C++ Edition Developers Guide*. |
| -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. |
| -consumerName | Perf_Consumer_ | Specifies the name of consumer in XML configuration file (**EmaConfig.xml**).Configures the name of the Consumer component in the configuration file (**EmaConfig.xml**) that will be used to configure the connection. |
| -delaySteadyStateCalc | 0 | Configures the time duration (in milliseconds), the consumer needs to wait to calculate the latency after receiving the last expected image. |
| -genericMsgLatencyRate | 0 | Controls the number of generic messages sent per second that contain latency information. This must be less than or equal 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 (see **`-tickRate`**). |
| -itemCount | 100000 | Sets the total number of items requested by the consumer. |

**Table 2: EmaCppConsPerf Configuration Options**

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -itemFile | 350k.xml | Configures the name of the item list file. |
| -latencyFile | None | Sets the name of the log file in which **EmaCppConsPerf** logs the latency retrieved from individual latency updates, generic messages, and posts. If a name is not specified, logging is disabled. |
| -mainThread | -1 | Specifies the CPU core to bind the main application thread that controls working threads, collects and prints statistics. By default, **EmaCppConsPerf** does not bind the main thread to specific CPU core.<br>Specifies the CPU physical mapping in format P:X C:Y T:Z, logical core ID (number), or no binding (-1).<br>Specifies the physical mapping which binds the thread to the specified physical processor, core, and thread (P:X C:Y T:Z). This syntax specifies a physical CPU to bind to. P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one. -1 means no bind.<br>For example, when specified "3", the main thread will be bound to logical CPU core 3.<br>For details on **`rsslBindThread`**, refer to the *Transport API C++ Edition Developers Guide*. |
| -msgFile | MsgData.xml | Configures the name of the file used by the consumer to determine the makeup of message payloads. For more details on input file information, refer to Section 8.1. |
| -noDisplayStats | (no argument) | Turns off printing statistics to the screen. |
| -postingLatencyRate | 0 | Controls the number of posts sent per second that contain latency information. This must be less than or equal to 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`**). |
| -requestRate | 35000 | Sets the number of item requests sent (per second). |
| -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. |
| -spTLSv1.2 | (no argument) | Specifies that TLSv1.2 can be used for an OpenSSL-based encrypted connection. |
| -spTLSv1.3 | (no argument) | Specifies that TLSv1.3 can be used for an OpenSSL-based encrypted connection. |
| -statsFile | ConsStats | Configures the base name that the consumer uses when writing its test statistics. |

**Table 2: EmaCppConsPerf Configuration Options (Continued)**

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -steadyStateTime | 300 | Configures how long (in seconds) the consumer continues to run the test after receiving the last expected image.<br>**`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 | The value of "**`-threads`**" serves two purposes: It defines the number of parallel threads/connections to be created AND where to bind each thread. For example, if an application wants to horizontally scale and open two connections, there should be two values for **`-threads`**: 0,1 OR P:0 C:0 T:0, P:0 C:0 T:1 OR -1, -1. The number of values specified indicates the number of threads to start (comma separated). The thread binding may be formatted or specified as follows: physical binding (P:X C:Y T:Z) OR logical binding (X) or -1 (no binding).<br>Physical binding: specify physical CPU to bind to: P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Logical Binding: For example, "1,3" creates two threads to establish connections with the provider, respectively bound to logical CPU cores 1 and 3.<br>In conjunction with **`-threads`**, one may specify **`-apiThreads`**, **`-workerThreads`**.<br>For details on **`rsslBindThread`**, refer to the *Transport API C++ Edition Developers Guide*. |
| -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. |
| -uname | None | Sets the user name for the login request. When unspecified, the system login name is used. |
| -useServiceId | 0 | Configures the consumer for adding the field "serviceId" in the Request message. For details on Request message, refer to *Enterprise Message API LSEG Domain Model Usage Guide*. |
| -useUserDispatch | 0 | Configures how **EmaCppConsPerf** and the Enterprise Message API dispatch received messages. When you select 0 (API dispatch model) then **EmaCppConsPerf** configures the Enterprise Message API to create an additional internal thread to dispatch received messages. When you select 1 (user dispatch model) the Enterprise Message API does not run a second thread and the **EmaCppConsPerf** is responsible for dispatching all received messages. By default, **EmaCppConsPerf** uses the API dispatch model.<br>For details on how an Enterprise Message API application dispatches received messages, refer to the *Enterprise Message API Developers Guide.* |
| -websocket | None | Configures the consumer for using websocket connection with specified protocol: "rssl.json.v2" or "rssl.rwf". |

**Table 2: EmaCppConsPerf Configuration Options (Continued)**

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -workerThreads | None | Specifies the CPU core(s) to which the Reactor worker thread(s) will be bound. **`EmaCppConsPerf`** does not bind the Reactor worker thread(s) to specific CPU core(s) by default. For details on Value Added Components, refer to the *Transport API Value Added Components Developers Guide*.<br>Specifies the CPU physical mapping in format P:X C:Y T:Z, logical core ID (number), or no binding (-1).<br>Specifies the physical mapping which binds the thread to the specified physical processor, core, and thread (P:X C:Y T:Z). This syntax specifies a physical CPU to bind to. P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one.<br>-1 means no bind.<br>For example, “1,3” specifies two Reactor worker threads, respectively bound to logical CPU cores 1 and 3.<br>The number of threads set by the parameter **`-threads`** should be matched.<br>For details on **`rsslBindThread`**, refer to the *Transport API C++ Edition Developers Guide*. |
| -writeStatsInterval | 5 | Configures the frequency (in seconds) at which statistics are printed to the screen and statistics file. |

**Table 2: EmaCppConsPerf Configuration Options (Continued)**

## 4.5 Input

**EmaCppConsPerf** 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 8, Input File Details.

### 4.5.1 EmaConfig.xml Examples

**EmaConfig.xml** must have a Consumer section in the Consumer group and appropriate Channel section in Channel group for correct configuration of the Enterprise Message API consumer.

For details on how to setup Consumer and Channel sections, refer to the *Enterprise Message API Configuration Guide*.

#### 4.5.1.1 Consumer Section

When creating a consumer section, you must include the **Name** and **Channel** fields. For details on **Name** and **Channel**, refer to the *Enterprise Message API Configuration Guide*.

```
<Consumer>
    <Name value="Perf_Consumer_1"/>
    <Channel value="Perf_Channel_1"/>
    <Logger value="Logger_1"/>
    <Dictionary value="Dictionary_1"/>
    <MaxDispatchCountApiThread value="6500"/>
    <MaxDispatchCountUserThread value="6500"/>
</Consumer>
```

**Example 1: Consumer Section Example**

#### 4.5.1.2 Channel Section

When creating a channel section, you must include the **Name** and **ChannelType** fields. For details on **Name** and **ChannelType,** refer to the *Enterprise Message API Configuration Guide*.

- To connect to the provider for TCP and WebSocket connections, you must specify **Host** and **Port** fields.
- To connect to the provider for encrypted connection, you must specify **Host**, **Port**, and **OpenSSLCAStore** fields.

```
<Channel>
    <Name value="Perf_Channel_1"/>
    <ChannelType value="ChannelType::RSSL_SOCKET"/>
    <CompressionType value="CompressionType::None"/>
    <GuaranteedOutputBuffers value="5000"/>
    <NumInputBuffers value="2048"/>
    <ConnectionPingTimeout value="30000"/>
    <TcpNodelay value="1"/>
    <DirectWrite value="0"/>
    <Host value="localhost"/>
    <Port value="14002"/>
</Channel>
```

**Example 2: Channel Section Example of TCP Connection Type**

```
<Channel>
    <Name value="Perf_Channel_Encr_1"/>
    <ChannelType value="ChannelType::RSSL_ENCRYPTED"/>
    <EncryptedProtocolType value="EncryptedProtocolType::RSSL_SOCKET"/>
    <CompressionType value="CompressionType::None"/>
    <GuaranteedOutputBuffers value="5000"/>
    <NumInputBuffers value="2048"/>
    <ConnectionPingTimeout value="30000"/>
    <TcpNodelay value="1"/>
    <Host value="localhost"/>
    <Port value="14002"/>
    <OpenSSLCAStore value="./RootCA.crt"/>
</Channel>
```

**Example 3: Channel Section Example of Encrypted Connection Type**

## 4.6 Output

**EmaCppConsPerf** 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 9, Output File Details.

### 4.6.1 EmaCppConsPerf Summary File Sample

```
--- TEST INPUTS ---

                Steady State Time: 300
          Delay Steady State Time: 0
                          Service: DIRECT_FEED
                  useUserDispatch: 0
                       mainThread: -1
                      Thread List: -1
                   ApiThread List: -1
                         Username: (use system login name)
                       Item Count: 100000
                Common Item Count: 0
```

```
                     Request Rate: 35000
                Request Snapshots: No
                     Posting Rate: 0
             Latency Posting Rate: 0
                 Generic Msg Rate: 0
         Latency Generic Msg Rate: 0
                        Item File: 350k.xml
                        Data File: MsgData.xml
                     Summary File: ConsSummary.out
                       Stats File: ConsStats
                 Latency Log File: latencyLog
                        Tick Rate: 1000

--- OVERALL SUMMARY ---

Startup State Statistics:
  Sampling duration (sec): 4.278
  Latency avg (usec): 139.2
  Latency std dev (usec): 53.5
  Latency max (usec): 229.0
  Latency min (usec): 58.0
  Avg update rate: 257
Steady State Statistics:
  Sampling duration (sec): 41.638
  Latency avg (usec): 81.8
  Latency std dev (usec): 26.3
  Latency max (usec): 380.0
  Latency min (usec): 42.0
  Avg update rate: 46737
Overall Statistics:
  Sampling duration (sec): 45.916
  Latency avg (usec): 81.8
  Latency std dev (usec): 26.4
  Latency max (usec): 380.0
  Latency min (usec): 42.0
  No GenMsg latency information was received.
  CPU/Memory samples: 9
  CPU Usage max (%): 97.49
  CPU Usage min (%): 29.76
  CPU Usage avg (%): 40.66
  Memory Usage max (MB): 338.33
  Memory Usage min (MB): 338.21
  Memory Usage avg (MB): 338.31
Test Statistics:
  Requests sent: 100000
  Refreshes received: 100000
  Updates received: 1947134
  Image retrieval time (sec): 4.278
  Avg image rate: 23377
  Avg update rate: 42409
```

Code Example 4: EmaCppConsPerf Summary File Sample

## 4.6.2 EmaCppConsPerf Statistics File Sample

```
UTC, Latency updates, Latency avg (usec), Latency std dev (usec), Latency max (usec), Latency min (usec),
    Images, Update rate (msg/sec), Posting Latency updates, Posting Latency avg (usec), Posting Latency
    std dev (usec), Posting Latency max (usec), Posting Latency min (usec), GenMsgs sent, GenMsgs
    received, GenMsg Latencies sent, GenMsg latencies received, GenMsg Latency avg (usec), GenMsg
    Latency std dev (usec), GenMsg Latency max (usec), GenMsg Latency min (usec), CPU usage (%), Memory
    (MB)
2021-06-02 13:07:27, 189, 81.5, 35.6, 230.0, 51.0, 100000, 3777, 0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0.0,
        0.0, 0.0, 0.0, 97.49, 338.21
2021-06-02 13:07:32, 2399, 81.7, 26.3, 258.0, 52.0, 0, 48034, 0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0.0, 0.0,
        0.0, 0.0, 33.32, 338.30
2021-06-02 13:07:37, 2362, 81.2, 25.0, 224.0, 42.0, 0, 47239, 0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0.0, 0.0,
        0.0, 0.0, 31.83, 338.30
2021-06-02 13:07:42, 2378, 80.5, 24.4, 255.0, 52.0, 0, 47566, 0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0.0, 0.0,
        0.0, 0.0, 34.31, 338.30
2021-06-02 13:07:47, 2372, 81.9, 25.5, 252.0, 51.0, 0, 47425, 0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0.0, 0.0,
        0.0, 0.0, 30.63, 338.30
2021-06-02 13:07:52, 2347, 82.2, 26.1, 292.0, 51.0, 0, 46970, 0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0.0, 0.0,
        0.0, 0.0, 34.64, 338.33
2021-06-02 13:07:58, 2387, 81.7, 26.8, 380.0, 52.0, 0, 47702, 0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0.0, 0.0,
        0.0, 0.0, 29.76, 338.33
2021-06-02 13:08:03, 2319, 84.3, 29.3, 263.0, 51.0, 0, 46387, 0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0.0, 0.0,
        0.0, 0.0, 36.86, 338.33
2021-06-02 13:08:08, 2366, 81.1, 26.3, 373.0, 52.0, 0, 47336, 0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 0.0, 0.0,
        0.0, 0.0, 37.11, 338.33
```

Code Example 5: EmaCppConsPerf Statistics File Sample

## 4.6.3 EmaCppConsPerf Latency File Sample

```
Message type, Send time, Receive time, Latency (usec)
Upd, 18873718238, 18873718467, 229
Upd, 18873722250, 18873722385, 135
Upd, 18873736516, 18873736631, 115
Upd, 18873748001, 18873748170, 169
Upd, 18873774757, 18873774824, 67
Upd, 18873816382, 18873816497, 115
Upd, 18873901823, 18873902021, 198
Upd, 18874110183, 18874110345, 162
Upd, 18874571156, 18874571334, 178
Upd, 18875361684, 18875361789, 105
Upd, 18876606790, 18876606848, 58
Upd, 18877995128, 18877995186, 58
```

```
Upd, 18877995646, 18877995699, 53
Upd, 18877998936, 18877998991, 55
Upd, 18878001029, 18878001099, 70
```

**Code Example 6: EmaCppConsPerf Latency File Sample**

### 4.6.4 EmaCppConsPerf Console Output Sample

```
005: Images: 100000, Posts:      0, UpdRate:     3777, CPU:  97.49%, Mem: 338.21MB
    Latency(usec): Avg:  81.5 StdDev:  35.6 Max:   230 Min:    51, Msgs: 189
    - Image retrieval time for 100000 images: 4.278s (23377 images/s)
010: Images:      0, Posts:      0, UpdRate:    48034, CPU:  33.32%, Mem: 338.30MB
    Latency(usec): Avg:  81.7 StdDev:  26.3 Max:   258 Min:    52, Msgs: 2399
015: Images:      0, Posts:      0, UpdRate:    47239, CPU:  31.83%, Mem: 338.30MB
    Latency(usec): Avg:  81.2 StdDev:  25.0 Max:   224 Min:    42, Msgs: 2362
020: Images:      0, Posts:      0, UpdRate:    47566, CPU:  34.31%, Mem: 338.30MB
    Latency(usec): Avg:  80.5 StdDev:  24.4 Max:   255 Min:    52, Msgs: 2378
025: Images:      0, Posts:      0, UpdRate:    47425, CPU:  30.63%, Mem: 338.30MB
    Latency(usec): Avg:  81.9 StdDev:  25.5 Max:   252 Min:    51, Msgs: 2372
030: Images:      0, Posts:      0, UpdRate:    46970, CPU:  34.64%, Mem: 338.33MB
Latency(usec): Avg:  82.2 StdDev:  26.1 Max:   292 Min:    51, Msgs: 2347
035: Images:      0, Posts:      0, UpdRate:    47702, CPU:  29.76%, Mem: 338.33MB
Latency(usec): Avg:  81.7 StdDev:  26.8 Max:   380 Min:    52, Msgs: 2387
```

**Code Example 7: EmaCppConsPerf 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.

**EmaCppIProvPerf** implements an OMM interactive provider using the Enterprise Message 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 as well as generic messages.

**EmaCppIProvPerf** uses **EmaConfig.xml** file to configure the Enterprise Message API.

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 Message API that incorporates the Value Add Reactor component from the Transport API to complete these tasks. For more information, refer to the *Enterprise Message API Developers Guide*.

## 5.2 Threading and Scaling

The Enterprise Message 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 Message API.

**EmaCppIProvPerf** always creates at least one working thread. You can configure **EmaCppIProvPerf** 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. Note that one consumer establishes one connection.

The application working thread leverages an Enterprise Message API provider that is configured as an interactive provider. The provider is implemented by the VA Reactor and runs the internal, VA Reactor logic. Additionally, **EmaCppIProvPerf** configures the Enterprise Message API to create a second, internal thread to dispatch received messages. You can configure **EmaCppIProvPerf** to not run the second thread inside the Enterprise Message API by using **`-useUserDispatch`** command-line option.

The main application thread monitors the other application level's threads, collects and reports statistics from them.

## 5.3 Provider Lifecycle

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

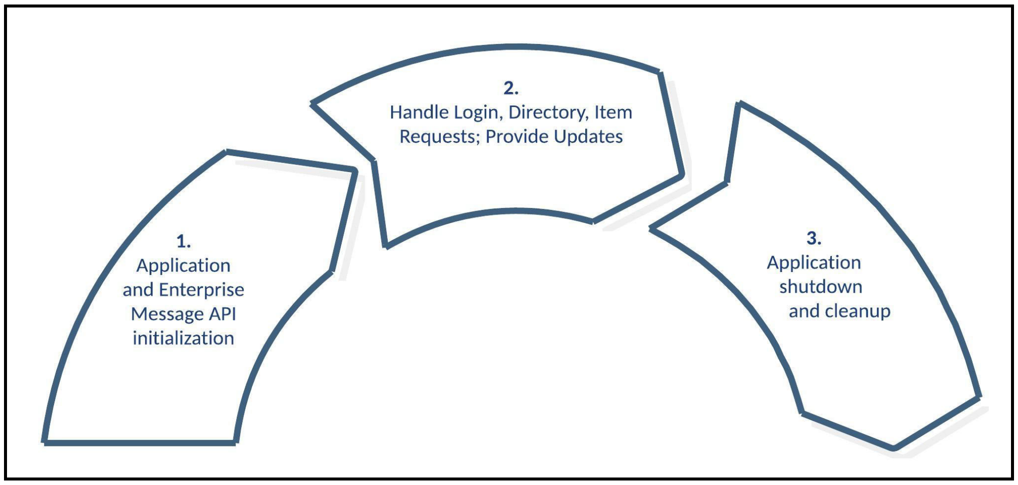

1.
Application and Enterprise Message API initialization
2.
Handle Login, Directory, Item Requests; Provide Updates
3.
Application shutdown and cleanup

**Figure 11. EmaCppIProvPerf Application Flow**

1. Application and Enterprise Message API Initialization.

   **EmaCppIProvPerf** loads its configuration, initializes the Enterprise Message API, loads its sample message data using specified files, and starts one or more threads (as configured) to provide data to consumers. The Enterprise Message API is configured by using **EmaConfig.xml**.

   The main thread periodically collects and writes statistics from the connection thread(s) until the test is over.

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

   - Send a burst of updates for items currently open on existing connections.
   - Send a burst of generic messages (if configured to do so).
   - Send reflected post 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 Message API and remaining resources, and exits.

   **EmaCppIProvPerf** 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 **EmaCppIProvPerf** application.

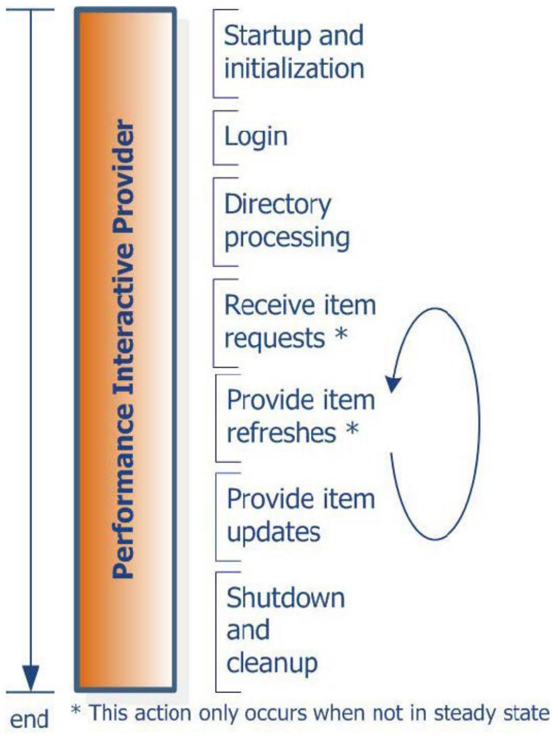


**Figure 12. EmaCppIProvPerf Application Flow**

## 5.4 Latency Measurement

**EmaCppIProvPerf** 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**) for Update message, field TIM_TRK_2 (**3903**) for Post message, and field TIM_TRK_3 (**3904**) for Generic message. Latency is measured after **ConsPerf** completes decoding.

### ▶ Interactive Provider Latency Measurement Sequence:

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

## 5.5 EmaCppIProvPerf Configuration Options

**EmaCppIProvPerf** uses **EmaConfig.xml** configuration file to setup an EMA interactive provider and set of command line options to configure specific behavior of performance tool.

**EmaConfig.xml** must have IProvider section in the Provider group and appropriate Server section in Server group for correct configuration of the EMA interactive provider. For details on how to setup IProvider and Server sections, refer to the *Enterprise Message API Configuration Guide*. For examples of configuration, refer to Section 5.6.1.

**EmaCppIProvPerf** uses the command line option -`providerName` to specify name of Provider section.

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -apiThreads | -1 | Specifies the CPU core(s) to bind the internal EMA API thread(s) to dispatch received messages. The parameter is used when the application configures EMA to work in API dispatch mode (see `-useUserDispatch`). **EmaCppIProvPerf** does not bind the internal EMA API thread(s) to specific CPU core(s) by default. For details on the internal EMA API thread, refer to the *Enterprise Message API Developers Guide, 2.4 Product Architecture*.<br>Specifies the CPU physical mapping in format P:X C:Y T:Z, logical core ID (number), or no binding (-1).<br>Specifies the physical mapping which binds the thread to the specified physical processor, core, and thread (P:X C:Y T:Z). This syntax specifies a physical CPU to bind to. P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one. -1 means no bind.<br>For example, when specified as "1,3", the internal EMA API threads will be bound to logical CPU cores 1 and 3.<br>The number of threads set by the parameter `-threads` should be matched.<br>For details on `rsslBindThread`, refer to the *Transport API C++ Edition Developers Guide*. |
| -genericMsgLatencyRate | 0 | Sets the number of generic messages sent (per second) that contain latency data. This number must be less than or equal to the total generic message rate (see `-genericMsgRate`). When you set the value to“all” then latency data is added to each generic message. |

**Table 3: EmaCppIProvPerf Configuration Options**

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -genericMsgRate | 0 | Sets the number of generic messages sent per second. This number cannot be less than the tick rate unless it is zero(see **`-tickRate`**). |
| -latencyFile | None | Specifies the name of the log file in which **EmaCppIProvPerf** logs the latency retrieved from individual latency updates, generic messages.<br>If a name is not specified, logging is disabled. |
| -latencyUpdateRate | 10 | Sets the number of updates sent per second containing latency information. This number must be less than or equal to the total update rate (see **`-updateRate`**).<br>**NOTE:** When you set the value “all” then latency data is added to each update message. |
| -mainThread | -1 | Specifies the CPU core to bind the main application thread that controls working threads, collects and prints statistics. By default, **EmaCppIProvPerf** does not bind the main thread to specific CPU core.<br>Specifies the CPU physical mapping in format P:X C:Y T:Z, logical core ID (number), or no binding (-1).<br>Specifies the physical mapping which binds the thread to the specified physical processor, core, and thread (P:X C:Y T:Z). This syntax specifies a physical CPU to bind to. P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one. -1 means no bind.<br>For example, when specified "3", the main thread will be bound to logical CPU core 3.<br>For details on **`rsslBindThread`**, refer to the *Transport API C++ Edition Developers Guide.* |
| -maxPackCount | 1 | Specifies maximum number of messages packed in a buffer (when count is greater than 1, packing is enabled). |
| -measureEncode | (no argument) | Configures **EmaCppIProvPerf** to measure encoding time of messages. By default, the measurement is not produced. |
| -measureDecode | (no argument) | Configures **EmaCppIProvPerf** to measure decoding time of messages. By default, the measurement is not produced. |
| -msgFile | MsgData.xml | Specifies the file that the provider uses to determine message content. For more details on input file information, refer to Section 8.1. |
| -nanoTime | (no argument) | Specifies the nanosecond precision for latency information instead of microsecond. |
| -noDisplayStats | (no argument) | Turns off printing statistics to the screen. |
| -packBufSize | 6000 | Sets size of buffer to use when message packing is enabled, i.e. **`maxPackCount`** > 1. |
| -providerName | Perf_Provider_ | Specifies the name of provider in XML configuration file (**EmaConfig.xml**). |
| -preEnc | (no argument) | Specifies pre-encoding for updates and generic messages. All the template messages (see **MsgData.xml**) will be encoded before real sending. It decreases the time required for message preparation. By default, **EmaCppIProvPerf** encodes messages each time.<br>**NOTE:** Whenever a latency data is required, the messages that contain latency are encoded (see -**`genericMsgLatencyRate`**, **`-latencyUpdateRate`**). |

**Table 3: EmaCppIProvPerf Configuration Options (Continued)**

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -runTime | 360 | Sets the length of time **EmaCppIProvPerf** runs (in seconds). |
| -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. |
| -threads | None | The value of "**`-threads`**" serves two purposes: it defines the number of parallel threads/connections to be created AND where to bind each thread. For example, if an application wants to horizontally scale and open two connections, there should be two values for **`-threads`**: 0,1 OR P:0 C:0 T:0, P:0 C:0 T:1 OR -1, -1. The number of values specified indicates the number of threads to start (comma separated). The thread binding may be formatted or specified as follows: physical binding (P:X C:Y T:Z) OR logical binding (X) or -1 (no binding).<br>Physical binding: specify physical CPU to bind to: P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one. -1 means no bind.<br>For example, when specified "1,3", it creates two threads to publish items respectively, and they are bound to CPU cores 1 and 3.<br>In conjunction with -threads, one may specify **`-apiThreads`** and **`-workerThreads`**.<br>For details on **`rsslBindThread`**, refer to the *Transport API Developers Guide*. |
| -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. |
| -updateRate | 100000 | Configures the number of updates sent per second, per connection.<br>**NOTE:** This cannot be less than the tick rate, unless it is zero (see -**`tickRate`**). |
| -useUserDispatch | 0 | Configures how **EmaCppIProvPerf** and Enterprise Message API dispatch received messages. When you select 0 (API dispatch model) then **EmaCppIProvPerf** configures the Enterprise Message API to create an additional internal thread to dispatch received messages. When you select 1 (user dispatch model) the Enterprise Message API does not run a second thread and the **EmaCppIProvPerf** is responsible for dispatching all received messages. By default, **EmaCppIProvPerf** uses the API dispatch model.<br>For details on how an Enterprise Message API application dispatches received messages, refer to the *Enterprise Message API Developers Guide*. |

**Table 3: EmaCppIProvPerf Configuration Options (Continued)**

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -workerThreads | <none> | Sets the list of CPUs bound to Reactor worker threads.By default, **EmaCpplProvPerf** does not bind the Reactor worker thread(s) to specific CPU core(s). For details on Value Added Components, refer to the *Transport API Value Added Components Developers Guide*.<br>Specifies the CPU physical mapping in format P:X C:Y T:Z, logical core ID (number), or no binding (-1).<br>Specifies the physical mapping which binds the thread to the specified physical processor, core, and thread (P:X C:Y T:Z). This syntax specifies a physical CPU to bind to. P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one. -1 means no bind.<br>For example, “1,3” specifies two Reactor worker threads, respectively bound to logical CPU cores 1 and 3.<br>The number of threads set by the **`-threads`** parameter should be matched.<br>For details on **`rsslBindThread`**, refer to the *Transport API Developers Guide*. |
| -writeStatsInterval | 5 | Sets how often statistics are printed to the screen and statistics file (in seconds). |

**Table 3: EmaCpplProvPerf Configuration Options (Continued)**

## 5.6 Input Files

**EmaCppConsPerf** 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 8, Input File Details.

### 5.6.1 EmaConfig.xml Examples

**EmaConfig.xml** must have IProvider section in the Provider group and appropriate Server section in Server group for correct configuration of the Enterprise Message API interactive provider.

For details on how to setup IProvider and Server sections, refer to the *Enterprise Message API Configuration Guide.*

#### 5.6.1.1 IProvider Section

When creating an IProvider section, you must include the **Name** and **Server** fields. For details on **Name** and **Server** fields, refer to the *Enterprise Message API Configuration Guide*.

```
<IProvider>
    <Name value="Perf_Provider"/>
    <Server value="Perf_Server_1"/>
    <Directory value="Directory_2"/>
    <Logger value="Logger_1"/>
    <ItemCountHint value="10000"/>
    <ServiceCountHint value="10000" />
    <CatchUnhandledException value="0" />
    <MaxDispatchCountApiThread value="500" />
    <MaxDispatchCountUserThread value="500" />
    <RefreshFirstRequired value="1" />
</IProvider>
```

**Example 8: IProvider Section Example**

#### 5.6.1.2 The Server Section

When creating an Server section, you must include the **Name** and **ServerType** fields. For details on **Name** and **ServerType** fields, refer to the *Enterprise Message API Configuration Guide*.

- You can use **WsProtocols** parameter when **ServerType** is set to *RSSL_WEBSOCKET*.
- You can use the following parameters when **ServerType** is set to *RSSL_ENCRYPTED*: **ServerCert** and **ServerPrivateKey**.

```
<Server>
    <Name value="Perf_Server_1"/>
    <ServerType value="ServerType::RSSL_SOCKET"/>
    <CompressionType value="CompressionType::None"/>
    <GuaranteedOutputBuffers value="50000"/>
    <ConnectionPingTimeout value="30000"/>
    <TcpNodelay value="1"/>
    <Port value="14002"/>
    <HighWaterMark value="6144"/>
    <InterfaceName value=""/>
    <DirectWrite value="0"/>
    <MaxFragmentSize value="6144"/>
    <NumInputBuffers value="10000"/>
    <SysRecvBufSize value="65535"/>
    <SysSendBufSize value="65535"/>
</Server>
```

**Example 9: Server Section Example of TCP Connection Type**

```
<Server>
    <Name value="Perf_Server_Websock_1"/>
    <ServerType value="ServerType::RSSL_WEBSOCKET"/>
    <CompressionType value="CompressionType::None"/>
    <GuaranteedOutputBuffers value="50000"/>
    <ConnectionPingTimeout value="30000"/>
    <TcpNodelay value="1"/>
    <Port value="14002"/>
    <MaxFragmentSize value="6144"/>
    <WsProtocols value="rssl.json.v2, rssl.rwf, tr_json2"/>
</Server>
```

**Example 10: Server Section Example for using WebSocket Protocol**

```
<Server>
    <Name value="Perf_Server_Encr_1"/>
    <ServerType value="ServerType::RSSL_ENCRYPTED"/>
    <CompressionType value="CompressionType::None"/>
    <GuaranteedOutputBuffers value="50000"/>
    <ConnectionPingTimeout value="30000"/>
    <TcpNodelay value="1"/>
    <ServerCert value="./cert/localhost.crt"/>
    <ServerPrivateKey value="./cert/localhost.key"/>
    <CipherSuite value=""/>
    <WsProtocols value="rssl.json.v2, rssl.rwf, tr_json2"/>
</Server>
```

**Example 11: Server Section Example of Encrypted Connection**

## 5.7 Output

**EmaCppIProvPerf** records statistics during a test such as:

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

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

### 5.7.1 EmaCppIProvPerf Summary File Sample

```
--- TEST INPUTS ---
                Run Time: 360
           Provider Name: Perf_Provider_1
         useUserDispatch: No
        mainThread CpuId: -1
             Thread List: -1
            Summary File: IProvSummary.out
        Latency Log File: (none)
    Write Stats Interval: 5
           Display Stats: Yes
               Tick Rate: 1000
             Update Rate: 1000
     Latency Update Rate: 10
        Generic Msg Rate: 0
Latency Generic Msg Rate: 0
      Refresh Burst Size: 10
               Data File: MsgData.xml
     Pre-Encoded Updates: No
         Nanosecond Time: No
          Measure Encode: No

--- OVERALL SUMMARY ---
Overall Statistics:
  No GenMsg latency information was received.
  Image requests received: 100000
  Updates sent: 52733
  CPU/Memory samples: 10
  CPU Usage max (%): 163.90
  CPU Usage min (%): 99.81
  CPU Usage avg (%): 113.08
  Memory Usage max (MB): 546.78
  Memory Usage min (MB): 546.68
  Memory Usage avg (MB): 546.77
```

**Code Example 12: EmaCppIProvPerf Summary File Sample**

### 5.7.2 EmaCppIProvPerf Statistics File Sample

```
UTC, Requests received, Images sent, Updates sent, Posts reflected, GenMsgs sent, GenMsgs received,
  GenMSg Latencies sent, GenMsg Latencies received, GenMsg Latency avg (usec), GenMsg Latency std dev
  (usec), GenMsg Latency max (usec), GenMsg Latency min (usec), CPU usage (%), Memory (MB)
2021-06-04 18:23:47, 100000, 97066, 11, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 125.53, 546.68
2021-06-04 18:23:52, 0, 2934, 9189, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 102.13, 546.78
2021-06-04 18:23:57, 0, 0, 5004, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 100.23, 546.78
2021-06-04 18:24:02, 0, 0, 5002, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 102.15, 546.78
2021-06-04 18:24:07, 0, 0, 4993, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 99.83, 546.78
2021-06-04 18:24:12, 0, 0, 5004, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 99.92, 546.78
2021-06-04 18:24:17, 0, 0, 4994, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 99.81, 546.78
2021-06-04 18:24:22, 0, 0, 5001, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 112.18, 546.78
2021-06-04 18:24:27, 0, 0, 5005, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 163.90, 546.78
2021-06-04 18:24:32, 0, 0, 4994, 0, 0, 0, 0, 0, 0.0, 0.0, 0.0, 0.0, 125.14, 546.78
```

**Code Example 13: EmaCppIProvPerf Statistics File Sample**

### 5.7.3 EmaCppIProvPerf Console Output Sample

```
005: UpdRate:        2, CPU: 125.53%, Mem: 546.68MB
  - Received 100000 item requests (total: 100000), sent 97066 images (total: 97066)
010: UpdRate:     1837, CPU: 102.13%, Mem: 546.78MB
  - Received 0 item requests (total: 100000), sent 2934 images (total: 100000)
015: UpdRate:     1000, CPU: 100.23%, Mem: 546.78MB
020: UpdRate:     1000, CPU: 102.15%, Mem: 546.78MB
025: UpdRate:      998, CPU:  99.83%, Mem: 546.78MB
030: UpdRate:     1000, CPU:  99.92%, Mem: 546.78MB
035: UpdRate:      998, CPU:  99.81%, Mem: 546.78MB
040: UpdRate:     1000, CPU: 112.18%, Mem: 546.78MB
045: UpdRate:     1001, CPU: 163.90%, Mem: 546.78MB
050: UpdRate:      998, CPU: 125.14%, Mem: 546.78MB
```

**Code Example 14: EmaCppIProvPerf 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.

**EmaCppNIProvPerf** 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.

**EmaCppNIProvPerf** uses **EmaConfig.xml** file to configure the Enterprise Message API.

When connecting, the non-interactive provider performs some administrative tasks, like processing system logins and publishing a directory refresh. The **EmaCppNIProvPerf** uses Enterprise Message API that incorporates the Value Add Reactor component from the Transport API to complete these tasks. For more information, refer to the *Enterprise Message API Developers Guide*.

## 6.2 Threading and Scaling

The Enterprise Message API is designed to allow sending data from multiple threads. So, the applications can scale their work across multiple cores by creating multiple threads to handle multiple connections through the Enterprise Message API.

You can configure **EmaCppNIProvPerf** 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 **`-itemCount`** 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

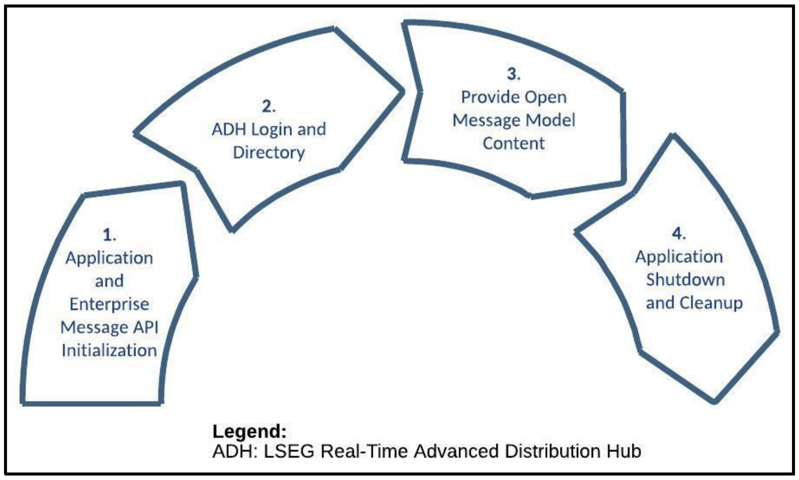

1. Application and Enterprise Message API Initialization
2. ADH Login and Directory
3. Provide Open Message Model Content
4. Application Shutdown and Cleanup
Legend:
ADH: LSEG Real-Time Advanced Distribution Hub

**Figure 13. EmaCppNIProvPerf Lifecycle**

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

1. Application and Enterprise Message API Initialization.

   In this phase **EmaCppNIProvPerf**:

   - Loads its configuration.

- Initializes the Enterprise Message API.
- Loads its 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 by using the Enterprise Message API. Once the connection succeeds, the test begins and any subsequent disconnection ends the test.

2. LSEG Real-Time Advanced Distribution Hub Login and Directory.
3. The connection thread configures the Enterprise Message API to perform login operation, publish its service,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 Message API and any remaining resources and then exits.

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

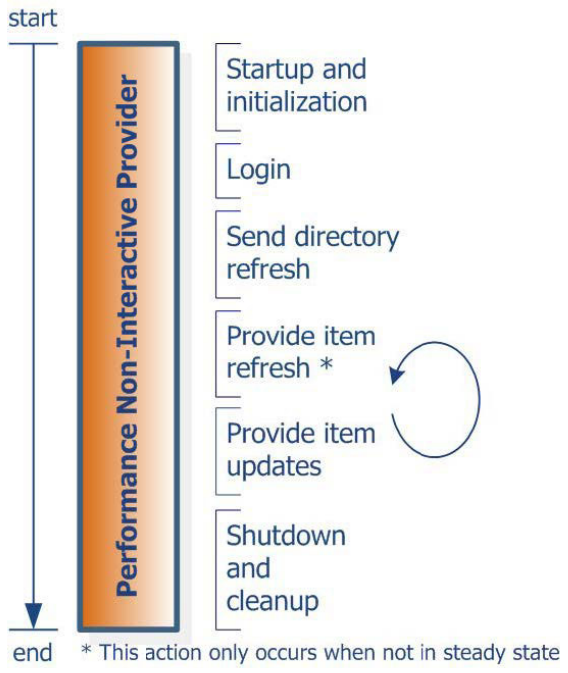

start
Performance Non-Interactive Provider
Startup and initialization
Login
Send directory refresh
Provide item refresh *
Provide item updates
Shutdown and cleanup
end
* This action only occurs when not in steady state

**Figure 14. EmaCppNIProvPerf Application Flow**

## 6.4 Latency Measurement

**EmaCppNIProvPerf** 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 a Consumer Performance tool decodes the message and payload.

### ▶ Non-Interactive Provider Latency Measurement Sequence:

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

## 6.5 EmaCppNIProvPerf Configuration Options

**EmaCppNIProvPerf** uses **EmaConfig.xml** configuration file to setup an Enterprise Message API Non-Interactive provider and set of command line options to configure specific behavior of performance tool.

**EmaConfig.xml** must have NiProvider section in the NiProviderGroup group and appropriate Channel section in Channel group for correct configuration of the EMA Non-Interactive provider. For details on how to setup NiProvider and Channel sections, refer to the *Enterprise Message API Configuration Guide*. For examples of configuration, refer to Section 6.6.1.

**EmaCppNIProvPerf** uses the command line option -`providerName` to specify name of NiProvider section**.**

**EmaCppNIProvPerf** 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. |
| -apiThreads | -1 | Specifies the CPU core(s) to bind the internal EMA API thread(s) to dispatch received messages. The parameter is used when the application configures EMA to work in API dispatch mode (see `-useUserDispatch`). By default, **EmaCppNIProvPerf** does not bind the internal EMA API thread(s) to specific CPU core(s). For details on the internal EMA API thread, refer to the *Enterprise Message API Developers Guide, 2.4 Product Architecture*.<br>Specifies the CPU physical mapping in format P:X C:Y T:Z, logical core ID (number), or no binding (-1).<br>Specifies the physical mapping which binds the thread to the specified physical processor, core, and thread (P:X C:Y T:Z). This syntax specifies a physical CPU to bind to. P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one. -1 means no bind.<br>For example, when specified as "1,3", the internal EMA API threads will be bound to logical CPU cores 1 and 3.<br>The number of threads set by the parameter `-threads` should be matched.<br>For details on `rsslBindThread`, refer to the *Transport API C++ Edition Developers Guide*. |

**Table 4: EmaCppNIProvPerf Configuration Options**

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -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. |
| -key | | Specifies the file containing the server private key for encryption. |
| -latencyUpdateRate | 10 | Sets the number of updates with latency information sent per second.<br>**NOTE:**<br>• This number must be less than or equal to the total update rate (see -**`updateRate`**).<br>• When you set the value “all” then the latency data is added to each update message. |
| -mainThread | -1 | Specifies the CPU core to bind the main application thread that controls working threads, collects and prints statistics. By default, **EmaCppNIProvPerf** does not bind the main thread to specific CPU core.<br>Specifies the CPU physical mapping in format P:X C:Y T:Z, logical core ID (number), or no binding (-1).<br>Specifies the physical mapping which binds the thread to the specified physical processor, core, and thread (P:X C:Y T:Z). This syntax specifies a physical CPU to bind to. P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one. -1 means no bind.<br>For example, when specified "3", the main thread will be bound to logical CPU core 3.<br>For details on **`rsslBindThread`**, refer to the *Transport API C++ Edition Developers Guide*. |
| -maxPackCount | 1 | Specifies maximum number of messages packed in a buffer. When count is > 1, packing is enabled. |
| -measureDecode | (no argument) | Configures **EmaCppNIProvPerf** to measure decoding time of messages. By default, the measurement is not produced. |
| -measureEncode | (no argument) | Configures **EmaCppNIProvPerf** to measure encoding time of messages. By default, the measurement is not produced. |
| -msgFile | MsgData.xml | Specifies the file that determines the provider’s message content. For more details on input file information, refer to Section 8.1. |
| -nanoTime | (no argument) | Specifies nanosecond precision for latency information instead of microsecond. |
| -noDisplayStats | (no argument) | Turns off printing statistics to the screen. |
| -packBufSize | 6000 | If message packing is enabled (i.e. **`maxPackCount`** > 1), sets the size of buffer to use. |

**Table 4: EmaCppNIProvPerf Configuration Options (Continued)**

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -preEnc | (no argument) | Specifies pre-encoding for update messages. All the template message (see **MsgData.xml**) will be encoded before real sending. It decreases the time required for message preparation. By default, **EmaCppNIProvPerf** encodes messages each time.<br>**NOTE:** When a latency data is required then the messages that contain latency is encoded each time (see -`latencyUpdateRate`). |
| -providerName | Perf_NIProvider_ | Specifies the name of provider in XML configuration file (**EmaConfig.xml**). |
| -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.<br>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 for which **EmaCppNIProvPerf** runs, in seconds. |
| -serviceId | 1 | Specifies the provider's service ID. |
| -serviceName | DIRECT_FEED | Specifies the provider's service name. |
| -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. |
| -threads | | The value of "`-threads`" serves two purposes: It defines the number of parallel threads/connections to be created AND where to bind each thread. For example, if an application wants to horizontally scale and open two connections, there should be two values for `-threads`: 0,1 OR P:0 C:0 T:0, P:0 C:0 T:1 OR -1, -1. The number of values specified indicates the number of threads to start (comma separated). The thread binding may be formatted or specified as follows: physical binding (P:X C:Y T:Z) OR logical binding (X) or -1 (no binding).<br>Physical binding: specify physical CPU to bind to: P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one. -1 means no bind.<br>For example, when specified as "1,3", it creates two threads to publish items respectively, and they are bound to CPU cores 1 and 3.<br>In conjunction with `-threads`, one may specify `-apiThreads` and `-workerThreads`. For details on `rsslBindThread`, refer to the *Transport API C++ Edition Developers Guide*. |
| -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. |
| -updateRate | 100000 | Sets the total number of updates sent per second, per connection.<br>**NOTE:** This cannot be less than the tick rate, unless it is **0** (see -`tickRate`). |
| -useServiceId | (no argument) | Turns on the usage of the service ID. See the `-serviceId` option. |

**Table 4: EmaCppNIProvPerf Configuration Options (Continued)**

| COMMAND-LINE OPTION | DEFAULT | DESCRIPTION |
|---|---|---|
| -useUserDispatch | 0 | Configures how **EmaCppNIProvPerf** and Enterprise Message API dispatch receive messages. When you select 0 (API dispatch model), then **EmaCppNIProvPerf** configures the Enterprise Message API to create an additional internal thread to dispatch received messages. When you select 1 (user dispatch model), the Enterprise Message API does not run a second thread and the **EmaCppNIProvPerf** is responsible for dispatching all received messages. By default, **EmaCppNIProvPerf** uses the API dispatch model.<br>For details on how an Enterprise Message API application dispatches received messages, refer to the *Enterprise Message API Developers Guide.* |
| -workerThreads | None | Specifies the CPU core(s) to bind the Reactor worker thread(s). By default, **EmaCppNIProvPerf** does not bind the Reactor worker thread(s) to specific CPU core(s). For details on Value Added Components, refer to the *Transport API Value Added Components Developers Guide*.<br>Specifies the CPU physical mapping in format P:X C:Y T:Z, logical core ID (number), or no binding (-1).<br>Specifies the physical mapping which binds the thread to the specified physical processor, core, and thread (P:X C:Y T:Z). This syntax specifies a physical CPU to bind to. P refers to processor, C refers to core, and T refers to thread. If T is not specified (or T:#), the thread will be bound to all threads on the specified processor. If C is not specified (or C:#), the thread will be bound to all cores and threads on that processor.<br>Specifying only one number causes a logical core ID to be bound instead of a physical one. -1 means no bind.<br>For example, when specified as "1,3", the Reactor worker threads will be bound to logical CPU cores 1 and 3.<br>The number of threads set by the parameter **`-threads`** should be matched.<br>For details on **`rsslBindThread`**, refer to the *Transport API C++ Edition Developers Guide*. |
| -writeStatsInterval | 5 | Sets how often statistics are printed to the screen and statistics file (in seconds). |

**Table 4: EmaCppNIProvPerf Configuration Options (Continued)**

## 6.6 Input Files

**EmaCppNIProvPerf** requires the following files:

- An XML configuration file for initializing Enterprise Message API. The package includes a default file (**EmaConfig.xml)** with this information.
- An XML file that describes **EmaCppNIProvPerf** 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 **EmaCppNIProvPerf** should publish. By default, the package includes the file, **350k.xml**.

For more detailed input file information, refer to Chapter 8, Input File Details8, Input File Details.

### 6.6.1 EmaConfig.xml Examples

**EmaConfig.xml** must have NiProvider section in the **NiProviderGroup** group and appropriate Channel section in the **Channel** group for correct configuration the Enterprise Message API Non-Interactive provider. For details on how to setup NiProvider and Channel sections, refer to the *Enterprise Message API Configuration Guide*.

**NOTE:** All type of connections require appropriate configuration of ADH. For more information on configuration, refer to the *ADS or ADH Software Installation Manuals*.

#### 6.6.1.1 NiProvider Section

When creating a NiProvider section, you must include the **Name** and **Channel** fields. For details on **Name** and **Channel**, refer to the *Enterprise Message API Configuration Guide.*

```
<NiProvider>
    <Name value="Perf_NIProvider"/>
    <Channel value="Perf_NIP_Channel_1"/>
    <Directory value="Perf_Directory_1"/>
    <Logger value="Logger_1"/>
    <XmlTraceToStdout value="0"/>
</NiProvider>
```

**Example 15: NiProvider Section Example**

#### 6.6.1.2 Channel Section

When creating a channel section, you must include the **Name** and **ChannelType** fields. For details on **Name** and **ChannelType,** refer to the *Enterprise Message API Configuration Guide.*

You must specify **Host** and **Port** fields to connect to ADH for TCP connection.

You must specify **RecvAddress, RecvPort, SendAddress, SendPort, UnicastPort,** and **InterfaceName** fields to connect to ADH for reliable multi-cast connection.

You must specify **Host**, **Port**, and **OpenSSLCAStore** fields to connect to ADH for encrypted connection.

```
<Channel>
    <Name value="Perf_NIP_Channel_1"/>
    <ChannelType value="ChannelType::RSSL_SOCKET"/>
    <GuaranteedOutputBuffers value="100000"/>
    <ConnectionPingTimeout value="30000"/>
    <TcpNodelay value="0"/>
    <Host value="adh_ip_address"/>
    <Port value="14003"/>
</Channel>
```

**Example 16: Channel Section Example of the TCP Connection Type**

```
<Channel>
    <Name value="Perf_NIP_Channel_Mcast_1"/>
    <ChannelType value="ChannelType::RSSL_RELIABLE_MCAST"/>
    <RecvAddress value="mcast_recv_ip_address"/>
    <RecvPort value="mcast_recv_port"/>
    <SendAddress value="mcast_send_ip_address"/>
    <SendPort value="mcast_send_port"/>
    <UnicastPort value="50000"/>
    <InterfaceName value="ip_address"/>
</Channel>
```

**Example 17: Server Section Example of the Reliable Multi-cast Connection**

```
<Channel>
    <Name value="Perf_NIP_Channel_Encr_1"/>
    <ChannelType value="ChannelType::RSSL_ENCRYPTED"/>
    <EncryptedProtocolType value="EncryptedProtocolType::RSSL_SOCKET"/>
    <GuaranteedOutputBuffers value="100000"/>
    <ConnectionPingTimeout value="30000"/>
    <TcpNodelay value="0"/>
    <Host value="adh_ap_address"/>
    <Port value="adh_rsslServerPort"/>
    <OpenSSLCAStore value="./myCA.pem"/>
</Channel>
```

**Example 18: Server Section Example of the Encrypted Connection**

## 6.7 Output

**EmaCppNIProvPerf** 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.

### 6.7.1 EmaCppNIProvPerf Summary File Sample

```
--- TEST INPUTS ---
              Run Time: 360
         Provider Name: Perf_NIProvider
       useUserDispatch: No
      mainThread CpuId: -1
           Thread List: -1
          Summary File: NIProvSummary.out
            Stats File: NIProvStats
  Write Stats Interval: 5
         Display Stats: Yes
            Item Count: 10000
             Tick Rate: 1000
           Update Rate: 100000
   Latency Update Rate: 10
    Refresh Burst Size: 10
             Item File: 350k.xml
             Data File: MsgData.xml
          Service Name: NI_PUB
   Pre-Encoded Updates: No
       Nanosecond Time: No
        Measure Encode: No

--- OVERALL SUMMARY ---
Overall Statistics:
  Images sent: 10000
  Updates sent: 5323442
  CPU/Memory samples: 10
  CPU Usage max (%): 67.68
  CPU Usage min (%): 60.09
  CPU Usage avg (%): 62.81
  Memory Usage max (MB): 555.56
  Memory Usage min (MB): 472.52
  Memory Usage avg (MB): 514.23
```

**Code Example 19: EmaCppNIProvPerf Summary File Sample**

### 6.7.2 EmaCppNIProvPerf Statistics File Sample

```
UTC, Images sent, Updates sent, CPU usage (%), Memory (MB)
2021-06-08 15:11:50, 10000, 465383, 63.14, 472.52
2021-06-08 15:11:55, 0, 499995, 64.26, 482.06
2021-06-08 15:12:00, 0, 500022, 62.47, 491.62
2021-06-08 15:12:05, 0, 500000, 67.68, 500.39
2021-06-08 15:12:10, 0, 499948, 61.46, 509.67
2021-06-08 15:12:15, 0, 500052, 60.09, 518.95
2021-06-08 15:12:20, 0, 499984, 61.43, 527.97
```

```
2021-06-08 15:12:25, 0, 499991, 61.01, 537.25
2021-06-08 15:12:30, 0, 500025, 62.72, 546.28
2021-06-08 15:12:35, 0, 500000, 63.88, 555.56
```

**Code Example 20: EmaCppNIProvPerf Statistics File Sample**

### 6.7.3 EmaCppNIProvPerf Console Output Sample

```
005: UpdRate:    93076, CPU:  63.14%, Mem: 472.52MB
    - Sent 10000 images (total: 10000)
010: UpdRate:    99999, CPU:  64.26%, Mem: 482.06MB
015: UpdRate:   100004, CPU:  62.47%, Mem: 491.62MB
020: UpdRate:   100000, CPU:  67.68%, Mem: 500.39MB
025: UpdRate:    99989, CPU:  61.46%, Mem: 509.67MB
030: UpdRate:   100010, CPU:  60.09%, Mem: 518.95MB
035: UpdRate:    99996, CPU:  61.43%, Mem: 527.97MB
040: UpdRate:    99998, CPU:  61.01%, Mem: 537.25MB
045: UpdRate:   100005, CPU:  62.72%, Mem: 546.28MB
050: UpdRate:   100000, CPU:  63.88%, Mem: 555.56MB
```

**Code Example 21: EmaCppNIProvPerf Console Output Sample**

# 7 Performance Measurement Scenarios

## 7.1 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 **EmaCppConsPerf** 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 **EmaCppIProvPerf** 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.

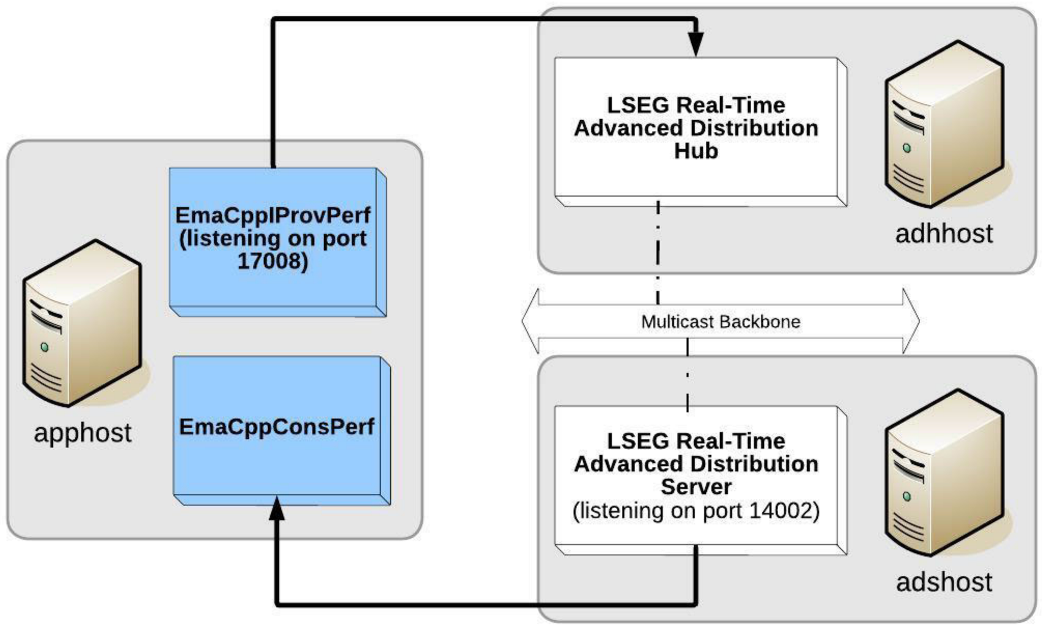

LSEG Real-Time Advanced Distribution Hub
adhhost
EmaCppIProvPerf (listening on port 17008)
apphost
EmaCppConsPerf
Multicast Backbone
LSEG Real-Time Advanced Distribution Server (listening on port 14002)
adshost

**Figure 15. Interactive Provider to Consumer on LSEG Real-Time Distribution System**

▶ To run a basic performance measurement:

1. Configure **Perf_Server_1**, change <Port value="*17008*" />.
2. Configure **Perf_Channel_1**, change <Host value="*adshost*" />, <Port value="*14002*"/>.
3. Configure **Directory_2**, change <Service><Name value="*TEST_FEED*"/>.
4. Run **ProvPerf** and **EmaCppConsPerf** 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.

```
EmaCppIProvPerf -providerName Perf_Provider
EmaCppConsPerf -serviceName TEST_FEED -consumerName Perf_Consumer_1
```

## 7.2 Interactive Provider to Consumer, Direct Connect

You can measure the interactive providers of data by connecting **EmaCppConsPerf** directly to **EmaCppIProvPerf**.

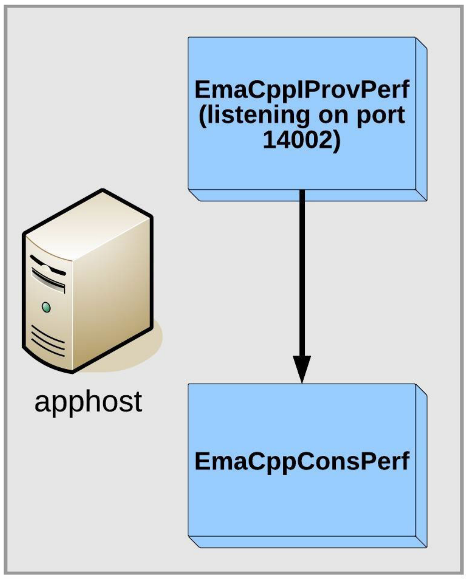


**Figure 16. Interactive Provider to Consumer, Direct Connect**

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:

```
EmaCppIProvPerf -providerName Perf_Provider
EmaCppConsPerf -serviceName DIRECT_FEED -consumerName Perf_Consumer_1
```

## 7.3 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 **EmaCppConsPerf** 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 **EmaCppNIProvPerf** 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.

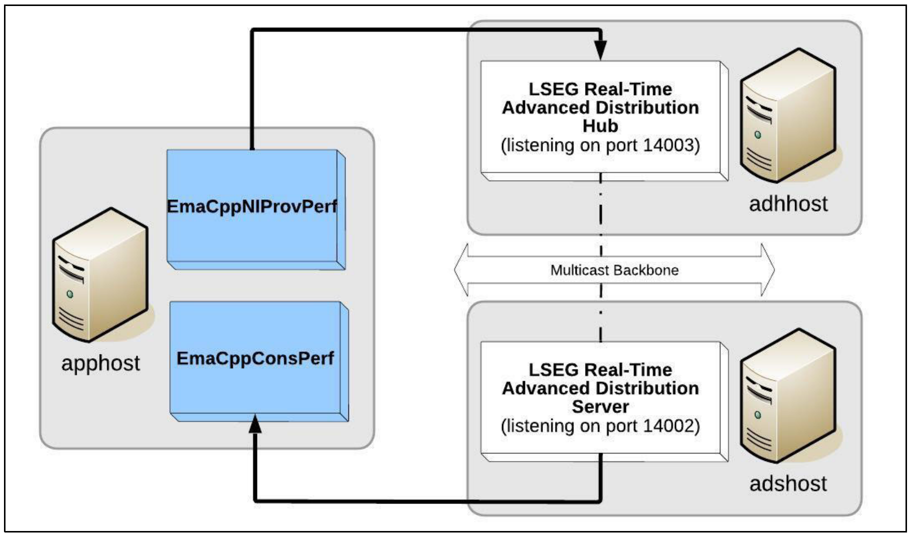

LSEG Real-Time Advanced Distribution Hub
(listening on port 14003)
adhhost
EmaCppNIProvPerf
Multicast Backbone
apphost
EmaCppConsPerf
LSEG Real-Time Advanced Distribution Server
(listening on port 14002)
adshost

**Figure 17. EmaCppNIProvPerf to Consumer on the LSEG Real-Time Distribution System**

**EmaCppConsPerf** may receive a Closed status if it requests an item not yet provided by **EmaCppNIProvPerf** 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. **EmaCppNIProvPerf** must have provided refreshes for all of its items to the LSEG Real-Time Advanced Distribution Hub before **EmaCppConsPerf** 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. **EmaCppConsPerf** 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:

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*"/>.
4. Run **EmaCppNIProvPerf** and **EmaCppConsPerf** with the following command-line options. These options assume the provided service is TEST_FEED. Modify the example values as necessary.

```
EmaCppNIProvPerf -serviceName TEST_FEED -providerName Perf_NIProvider_1
```

```
EmaCppConsPerf -serviceName TEST_FEED
-consumerName Perf_Consumer_1
```

## 7.4 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 **EMACppConsPerf** 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 **EmaCppNIProvPerf** 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 **RsslPostUserInfo**. When configured to do so, **EMACppConsPerf** embeds timestamps in some of its posts which it uses to measure round-trip latency.

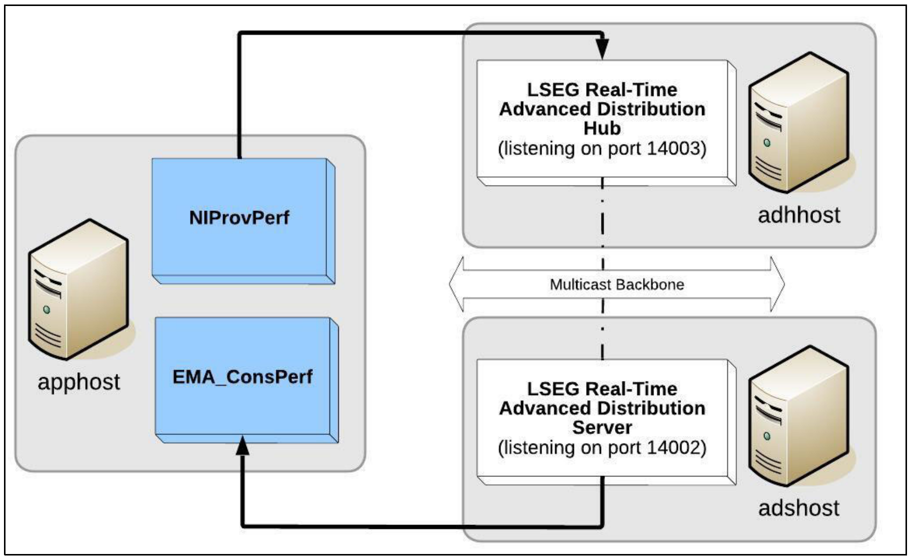

LSEG Real-Time Advanced Distribution Hub (listening on port 14003)
adhhost
NIProvPerf
Multicast Backbone
apphost
EMA_ConsPerf
LSEG Real-Time Advanced Distribution Server (listening on port 14002)
adshost

**Figure 18. Consumer Posting to LSEG Real-Time Distribution System**

Update traffic is optional. If you want to test posting without updates, configure **EmaCppNIProvPerf** 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*.

### ▶ To run a basic performance measurement:

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*"/>.

4. Run **EmaCppNIProvPerf** and **EmaCppConsPerf** with the following command-line options. These options assume TEST_FEED is the service being provided. Modify the example values as necessary.

```
EmaCppNIProvPerf -h adhhost -p 14003 -serviceName TEST_FEED
EmaCppConsPerf -h adshost -p 14002 -serviceName TEST_FEED -postingRate 10000 -postingLatencyRate 10
```

# 8 Input File Details

## 8.1 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 **EmaCppConsPerf**.
- At least one generic message, if configured for exchanging generic messages.

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

### 8.1.1 Encoding Fields

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

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.

### 8.1.2 Sample Update Message

```
<updateMsg>
    <dataBody>
        <fieldList 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 22: Sample Update Message**

### 8.1.3 Sample MarketByOrder Data

Performance tools also support MarketByOrder data, however it is currently experimental. To allow tools to provide MarketByOrder data, you can add the following data to **MsgData.xml**:

```
<!-- MarketByOrder -->
<marketByOrderMsgList>

<!-- ORDER_SIDE enumerations: BID is 1, ASK is 2 -->

<refreshMsg>
    <dataBody>
        <map>
            <fieldSetDefs>
                <fieldSetDef setId="0">
                    <fieldSetDefEntry fieldId="3427" dataType="RSSL_DT_REAL" />
                    <fieldSetDefEntry fieldId="3429" dataType="RSSL_DT_REAL" />
                    <fieldSetDefEntry fieldId="3855" dataType="RSSL_DT_UINT_2" />
                    <fieldSetDefEntry fieldId="212" dataType="RSSL_DT_RMTES_STRING" />
                    <fieldSetDefEntry fieldId="3428" dataType="RSSL_DT_ENUM" />
                </fieldSetDef>
            </fieldSetDefs>
            <mapEntry action="RSSL_MPEA_ADD_ENTRY" key="100" >
                <fieldList setId="0">
                    <fieldEntry fieldId="3427" dataType="RSSL_DT_REAL" data="360287970189639.67"/>
                    <fieldEntry fieldId="3429" dataType="RSSL_DT_REAL" data="36028797018963967"/>
                    <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="0"/>
                    <fieldEntry fieldId="212" dataType="RSSL_DT_RMTES_STRING" data="MarketMaker1"/>
                    <fieldEntry fieldId="3428" dataType="RSSL_DT_ENUM" data="1"/>
                </fieldList>
            </mapEntry>
            <mapEntry action="RSSL_MPEA_ADD_ENTRY" key="101" >
                <fieldList setId="0">
                    <fieldEntry fieldId="3427" dataType="RSSL_DT_REAL" data="360287970189638.67"/>
                    <fieldEntry fieldId="3429" dataType="RSSL_DT_REAL" data="36028797018963467"/>
                    <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="500"/>
                    <fieldEntry fieldId="212" dataType="RSSL_DT_RMTES_STRING" data="MarketMaker2"/>
                    <fieldEntry fieldId="3428" dataType="RSSL_DT_ENUM" data="1"/>
                </fieldList>
            </mapEntry>
            <mapEntry action="RSSL_MPEA_ADD_ENTRY" key="102" >
                <fieldList setId="0">
                    <fieldEntry fieldId="3427" dataType="RSSL_DT_REAL" data="360287970189637.67"/>
                    <fieldEntry fieldId="3429" dataType="RSSL_DT_REAL" data="36028797018962967"/>
                    <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="1000"/>
                    <fieldEntry fieldId="212" dataType="RSSL_DT_RMTES_STRING" data="MarketMaker3"/>
                    <fieldEntry fieldId="3428" dataType="RSSL_DT_ENUM" data="2"/>
                </fieldList>
            </mapEntry>
            <mapEntry action="RSSL_MPEA_ADD_ENTRY" key="103" >
```

```
                <fieldList setId="0">
                    <fieldEntry fieldId="3427" dataType="RSSL_DT_REAL" data="360287970189636.67"/>
                    <fieldEntry fieldId="3429" dataType="RSSL_DT_REAL" data="36028797018962467"/>
                    <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="1500"/>
                    <fieldEntry fieldId="212" dataType="RSSL_DT_RMTES_STRING" data="MarketMaker4"/>
                    <fieldEntry fieldId="3428" dataType="RSSL_DT_ENUM" data="2"/>
                </fieldList>
            </mapEntry>
            <mapEntry action="RSSL_MPEA_ADD_ENTRY" key="104" >
                <fieldList setId="0">
                    <fieldEntry fieldId="3427" dataType="RSSL_DT_REAL" data="360287970189635.67"/>
                    <fieldEntry fieldId="3429" dataType="RSSL_DT_REAL" data="36028797018961967"/>
                    <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="2000"/>
                    <fieldEntry fieldId="212" dataType="RSSL_DT_RMTES_STRING" data="MarketMaker5"/>
                    <fieldEntry fieldId="3428" dataType="RSSL_DT_ENUM" data="2"/>
                </fieldList>
            </mapEntry>
        </map>
    </dataBody>
</refreshMsg>

<updateMsg>
    <dataBody>
        <map>
            <mapEntry action="RSSL_MPEA_UPDATE_ENTRY" key="100" >
                <fieldList>
                    <fieldEntry fieldId="3427" dataType="RSSL_DT_REAL" data="360287970189638.67"/>
                    <fieldEntry fieldId="3429" dataType="RSSL_DT_REAL" data="36028797018963867"/>
                    <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="0"/>
                </fieldList>
            </mapEntry>
            <mapEntry action="RSSL_MPEA_UPDATE_ENTRY" key="101" >
                <fieldList>
                    <fieldEntry fieldId="3427" dataType="RSSL_DT_REAL" data="360287970189635.67"/>
                    <fieldEntry fieldId="3429" dataType="RSSL_DT_REAL" data="36028797018963567"/>
                    <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="500"/>
                </fieldList>
            </mapEntry>
            <mapEntry action="RSSL_MPEA_UPDATE_ENTRY" key="102" >
                <fieldList>
                    <fieldEntry fieldId="3427" dataType="RSSL_DT_REAL" data="360287970189632.67"/>
                    <fieldEntry fieldId="3429" dataType="RSSL_DT_REAL" data="36028797018963267"/>
                    <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="1000"/>
                </fieldList>
            </mapEntry>
            <mapEntry action="RSSL_MPEA_UPDATE_ENTRY" key="103" >
                <fieldList>
                    <fieldEntry fieldId="3427" dataType="RSSL_DT_REAL" data="360287970189629.67"/>
                    <fieldEntry fieldId="3429" dataType="RSSL_DT_REAL" data="36028797018962967"/>
                    <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="1500"/>
                </fieldList>
            </mapEntry>
            <mapEntry action="RSSL_MPEA_UPDATE_ENTRY" key="104" >
                <fieldList>
                    <fieldEntry fieldId="3427" dataType="RSSL_DT_REAL" data="360287970189626.67"/>
                    <fieldEntry fieldId="3429" dataType="RSSL_DT_REAL" data="36028797018962667"/>
                    <fieldEntry fieldId="3855" dataType="RSSL_DT_UINT" data="2000"/>
                </fieldList>
            </mapEntry>
        </map>
    </dataBody>
</updateMsg>

</marketByOrderMsgList>
```

**Code Example 23: Sample MarketByOrder Data**

## 8.2 Item List File

The Item List File configures the full list of items as requested by **EmaCppConsPerf** or published by **EmaCppNIProvPerf**. 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.

### 8.2.1 Item Attributes

| ATTRIBUTE NAME | DEFAULT | DESCRIPTION |
|---|---|---|
| domain | (none, required) | Specifies the domain from which the item is requested.<br>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**, **EmaCppConsPerf** sends posts to this item (if posting is enabled). |
| snapshot | "false" | If set to **true**, **EmaCppConsPerf** requests this item as a snapshot (i.e., without setting the **RSSL_RQF_STREAMING** flag on the request). |

**Table 5: Item Attributes**

### 8.2.2 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 24: Sample Item List File**

# 9 Output File Details

## 9.1 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.

## 9.2 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 6: Performance Suite Applications and Associated Configuration Files**

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

## 9.3 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 **EmaCppConsPerf**.

**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 25: Sample ConsLatency.csv Showing Update and Post latencies during a Test Run**

## 9.4 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.

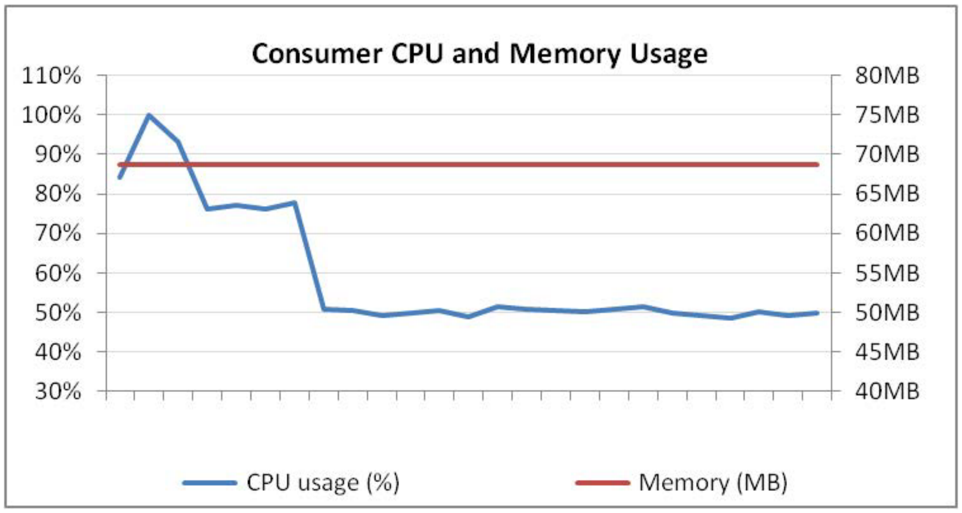


**Figure 19. Sample Excel Graph from ConsStats1.csv**

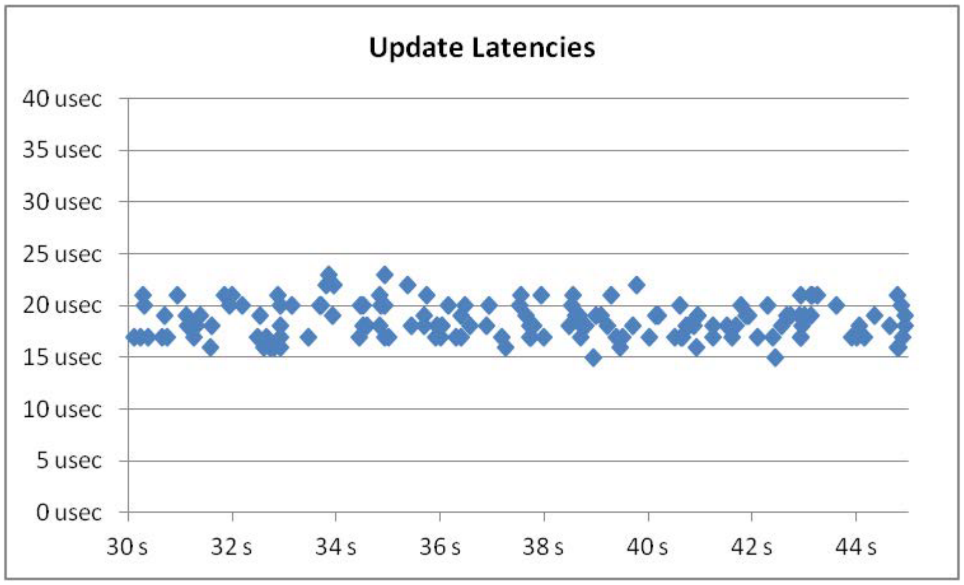


**Figure 20. Sample Excel Graph of Latencies Over a 15-second Steady State Interval from ConsLatency1.csv**

# 10 Performance Best Practices

## 10.1 Overview

The Performance Test Tools Suite leverages a number of features of the Enterprise Message 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 Message API Configuration Guide.

## 10.2 Enterprise Message API Best Practices

### 10.2.1 dispatch

The **`dispatch()`** method is used to read messages from network. The application can call this method by using the user thread or API dispatching thread depending on the operational model of OmmConsumer and OmmProvider. The application thread or API dispatching thread must always call the **`dispatch()`** method to receive messages via callback methods and process these messages. Application can configure the **MaxDispatchCountApiThread** and **MaxDispatchCountUserThread** parameters to specify maximum number of messages that can be dispatched in a single call of the **`dispatch()`** method before taking a break.

### 10.2.2 submit

To make efficient use of underlying transport function calls, the **submit** function passes messages to an outbound queue of the specified priority, rather than immediately writing the message to the network.

The network write occurs if:

- Enterprise Message API internally calls **rsslFlush** on the Channel instance in the Transport layer.
- **HighWaterMark** configuration parameter in Channel or Server group **submit** If **DirectWrite** is set, the API will flush each message upon submission without waiting to queue data. If set in conjunction with **highWaterMark**, **directWrite** takes precedence.

### 10.2.3 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 Message API employs the concept of a high-water mark. When the application calls **submit** , the Enterprise Message API does not always immediately pass the buffer to the transport; instead, the Enterprise Message API passes data to the transport after the size of its buffer reaches the high-water mark.

For example, assume a high-water mark of 6144 bytes. If an application, creates a message, encodes 500 bytes of content, and passes this to **submit** , the high-water mark will be triggered after thirteen buffers. At that point, the Enterprise Message 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.

Note the throughput and latency implications. Balance the use of the high-water mark and flush by Enterprise Message 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 Message API queues for longer periods and thus incur latency.

### 10.2.4 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 **TcpNodelay** configuration parameter, which disables Nagle's Algorithm.

### 10.2.5 System Send and Receive Buffers

For TCP socket connections, the OS uses system send and receive buffers for exchanging content. When the Enterprise Message 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 Message API to pass queued data to the underlying transport but the OS cannot accept additional content at the time, then the content must be queued in the Enterprise Message API and Enterprise Message API should flush it at a subsequent time. However, this state is not considered a failure condition, and the Enterprise Message 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, 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 Message API allows users to configure this via **SysSendBufSize** and **SysRecvBufSize** configuration parameters in Channel or Server group.

### 10.2.6 Additional Parameters to Consider

You can use additional parameters in Enterprise Message API to balance reading/writing to network. Following are considerations to achieve performance with Enterprise Message API:

- Buffers:
  - **NumInputBuffer** parameter: preset value to a predicted volume of content from network; this can be dynamically changed.
  - **GuaranteedOutputBuffers** parameter: pre-allocated number of buffers may be adjusted to account for typical message rates; this can be dynamically changed.
- **ItemCountHint** parameter: controls sizing of internal structures and should be set to expected watchlist size.
- **DispatchTimeoutApiThread** parameter: By default this is set to -1 for EMA internal thread to be notified every time a new message is received. Under high volumes, the recommendation is to leave the default to -1 as EMA internal thread is active after a specified timeout or whenever there is something to read.
- **MaxDispatchCountApiThread** or **MaxDispatchCountUserThread** parameter: choose to set this value to number of messages to read.
- **UserDispatch** parameter: "1" as a return value means there is something to read.
- Requesting views: reduce the amount of content received by request to only the FIDs used by application.
- Use update filtering to limit number of received updates based on the **updateType** parameter when requesting streaming content. For Login Request definition, possible **`updateType`** values per domain (e.g., "Market Price", "Market By Order", and so on), and configuration of **`UpdateTypeFilter`** and **`NegativeUpdateTypeFilter`**, refer to the *Enterprise Message API RDM Usage Guide*.

**NOTE:** For additional tuning such as thread binding, turning off services, system send/recv buffer size tuning, and other, refer to the *RTSDK Performance Test Results* document.

### 10.2.7 Enterprise Message API Buffering

The Enterprise Message 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 Message 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.

#### 10.2.7.1 Input Buffering

The Enterprise Message API input buffer is created as one large continuous block of memory, controlled by **NumInputBuffers** configuration parameter in Channel or Server group. 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 **dispatch** call to get as many bytes from a single call to the underlying transport as possible. When the input buffer holds data, the Enterprise Message API determines message boundaries and returns a single message to the user. As the application makes subsequent dispatch calls, additional messages are dispatched from the input buffer. After fully processing the input buffer, the Enterprise Message API goes back to the underlying transport to again fill the input buffer.

The intent is to have the Enterprise Message API read only when needed and to read as much as possible. The amount of data the Enterprise Message 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.

#### 10.2.7.2 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 Message 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** configuration parameter setting controls the number of output buffers available exclusively to that channel, where all of these buffers are created up-front.

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.

#### 10.2.7.3 Fragmentation

The negotiated maximum buffer size is the maximum size that the application will send in a single buffer. In cases where an application encodes a message larger than the maximum, the requested size will be returned to the user. When the content passes to **submit**, the Enterprise Message 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 Message 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 specified by the **MaxFragmentSize** configuration parameter is large enough for commonly sent messages to fit into a single buffer.

### 10.2.8 Compression

The Enterprise Message 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.[1]
- 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 Message API's buffer pool.

## 10.3 Encoder and Decoder Best Practices

### 10.3.1 Single-Pass Encoding

Enterprise Message API 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, followed by the field list payload. After the payload is encoded, message encoding is completed.

## 10.4 Other Practices: CPU Binding

Although the OS tries to balance its load intelligently across multi-core processors, you can improve performance by locking the threads of a process to specific cores and CPUs. This lessens the likelihood of switching a thread from one core to another, which impacts processing time and invalidates the cache that the thread has filled.

Each Enterprise Message API Performance Tool allows CPU binding through its respective **`-threads`** option. LSEG recommends that you test different bindings for each tool to see which works best on each system.

1. 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.
- 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 21. 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.

•

## 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: EMAC392L1PETOO.250
Date of issue: December 2025