

# Enterprise Message API

## C# Edition

### 3.1.0.L1

OPEN SOURCE PERFORMANCE TOOLS GUIDE

© **Refinitiv 2023**. All rights reserved.

Republication or redistribution of Refinitiv content, including by framing or similar means, is prohibited without the prior written consent of Refinitiv. 'Refinitiv' and the Refinitiv logo are registered trademarks and trademarks of Refinitiv.

Any software, including but not limited to: the code, screen, structure, sequence, and organization thereof, and its documentation are protected by national copyright laws and international treaty provisions. This manual is subject to U.S. and other national export regulations.

Refinitiv, by publishing this document, does not guarantee that any information contained herein is and will remain accurate or that use of the information will ensure correct and faultless operation of the relevant service or equipment. Refinitiv, its agents, and its employees, shall not be held liable to or through any user for any loss or damage whatsoever resulting from reliance on the information contained herein.

# Contents

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	About this Manual .....	1
1.2	Audience .....	1
1.3	Programming Language.....	1
1.4	Acronyms and Abbreviations .....	2
1.5	References .....	2
1.6	Documentation Feedback .....	2
1.7	Document Conventions.....	3
1.7.1	<i>Typographic</i> .....	3
1.7.2	<i>Diagrams</i> .....	3
<b>2</b>	<b>Open Source Performance Tool Suite Overview.....</b>	<b>5</b>
2.1	Overview .....	5
2.2	The Enterprise Message API Performance Tool Suite .....	5
2.3	Package Contents .....	6
2.3.1	<i>XML Files</i> .....	6
2.3.2	<i>Building and Running</i> .....	6
2.4	What Gets Measured and Reported? .....	7
2.4.1	<i>Latency</i> .....	7
2.4.2	<i>Throughput and Payload</i> .....	7
2.4.3	<i>Image Retrieval Time</i> .....	7
2.4.4	<i>CPU and Memory Usage</i> .....	7
2.5	Recorded Results and Output.....	8
2.5.1	<i>Summary File</i> .....	8
2.5.2	<i>Statistics File</i> .....	8
2.5.3	<i>Latency File</i> .....	8
<b>3</b>	<b>Latency Measurement Details.....</b>	<b>9</b>
3.1	Time-slicing .....	9
3.2	Latency.....	10
<b>4</b>	<b>ConsPerf .....</b>	<b>11</b>
4.1	Overview .....	11
4.2	Threading and Scaling .....	11
4.2.1	<i>Consumer Lifecycle</i> .....	11
4.2.2	<i>Diagram</i> .....	12
4.3	Latency Measurement.....	13
4.3.1	<i>Consumer Latency</i> .....	13
4.3.2	<i>Posting Latency</i> .....	13
4.4	ConsPerf Configuration Options .....	13
4.5	Input .....	15
4.5.1	<i>EmaConfig.xml Examples</i> .....	15
4.6	Output .....	16
4.6.1	<i>ConsPerf Summary File Sample</i> .....	16
4.6.2	<i>ConsPerf Statistics File Sample</i> .....	18
4.6.3	<i>ConsPerf Latency File Sample</i> .....	18
4.6.4	<i>ConsPerf Console Output Sample</i> .....	19
<b>5</b>	<b>Performance Measurement Scenarios .....</b>	<b>20</b>
5.1	Interactive Provider to Consumer, Through Refinitiv Real-Time Distribution System .....	20

5.2	Interactive Provider to Consumer, Direct Connect .....	21
5.3	Non-Interactive Provider to Consumer, Through Refinitiv Real-Time Distribution System .....	22
5.4	Consumer Posting on the Refinitiv Real-Time Distribution System .....	23
<b>6</b>	<b>Input File Details .....</b>	<b>24</b>
6.1	Message Content File and Format .....	24
6.1.1	<i>Encoding Fields</i> .....	24
6.1.2	<i>Sample Update Message</i> .....	25
6.2	Item List File .....	25
6.2.1	<i>Item Attributes</i> .....	26
6.2.2	<i>Sample Item List File</i> .....	26
<b>7</b>	<b>Output File Details .....</b>	<b>27</b>
7.1	Overview .....	27
7.2	Output Files and Their Descriptions .....	27
7.3	Latency File .....	28
7.4	File Import .....	28
<b>8</b>	<b>Performance Best Practices .....</b>	<b>30</b>
8.1	Overview .....	30
8.2	Transport Best Practices .....	30
8.2.1	<i>Reading Data</i> .....	30
8.2.2	<i>Writing Data</i> .....	30
8.2.3	<i>High-water Mark</i> .....	30
8.2.4	<i>Direct Write</i> .....	30
8.2.5	<i>Nagle's Algorithm</i> .....	31
8.2.6	<i>System Send and Receive Buffers</i> .....	31
8.2.7	<i>Enterprise Message API Buffering</i> .....	31
8.2.8	<i>Compression</i> .....	32
<b>Appendix A</b>	<b>Troubleshooting .....</b>	<b>33</b>
A.1	Can't Connect .....	33
A.2	Not Achieving Steady State .....	33
A.3	Consumer Tops Out but Not at 100% CPU .....	34
A.4	Initial Latencies Are High .....	34
A.5	Latency Values Are Very High .....	34

# List of Figures

Figure 1.	Running Performance Example and Host Notation .....	3
Figure 2.	Network Diagram Notation .....	4
Figure 3.	Three Connection Options for the Open Message Model-based Performance Tools .....	5
Figure 4.	Directory Structure of the Performance Tools.....	6
Figure 5.	Time Slicing Algorithm .....	9
Figure 6.	Refresh Publishing Algorithm.....	9
Figure 7.	Latency Reuters Instrument Codes within a Tick.....	10
Figure 8.	Timing Diagram for Latency Measurements .....	10
Figure 9.	EMA_ConsPerf Lifecycle .....	11
Figure 10.	EMA_ConsPerf Application Flow .....	13
Figure 11.	Interactive Provider to Consumer on Refinitiv Real-Time Distribution System .....	20
Figure 12.	Interactive Provider to Consumer, Direct Connect.....	21
Figure 13.	EmajNIProvPerf to Consumer on the Refinitiv Real-Time Distribution System .....	22
Figure 14.	Consumer Posting to Refinitiv Real-Time Distribution System .....	23
Figure 15.	Sample Excel Graph from ConsStats1.csv.....	28
Figure 16.	Sample Excel Graph of Latencies Over a 15-second Steady State Interval from ConsLatency1.csv .....	29
Figure 17.	Refinitiv Real-Time Advanced Distribution Server distribution.cnf .....	34

# List of Tables

Table 1:	Acronyms and Abbreviations .....	2
Table 2:	EMA_ConsPerf Configuration Options.....	13
Table 3:	Item Attributes .....	26
Table 4:	Performance Suite Applications and Associated Configuration Files .....	27

# 1 Introduction

## 1.1 About this Manual

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

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

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

## 1.2 Audience

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

## 1.3 Programming Language

Enterprise Message API C# 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	Refinitiv Real-Time Advanced Data Hub
ADS	Refinitiv Real-Time Advanced Distribution Server
API	Application Programming Interface
CPU	Central Processing Unit
DMM	Domain Message Model
EMA	Enterprise Message API
RTSDK	Refinitiv Real-Time Software Developer Kit
ETA	Enterprise Transport API
OMM	Open Message Model
RDM	Refinitiv Domain Model
RFA	Robust Foundation API

**Table 1: Acronyms and Abbreviations**

## 1.5 References

- *Enterprise Transport API C# Edition Developers Guide*
- *Enterprise Transport API C# Edition Refinitiv Domain Model Usage Guide*
- *Enterprise Transport API C# Edition Value Added Components Developers Guide*
- *Enterprise Message API C# Edition Developers Guide*
- *Enterprise Message API C# Edition Domain Model Usage Guide*
- *Enterprise Message API C# Edition Configuration Guide*

## 1.6 Documentation Feedback

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

- Send us your comments via email at [ProductDocumentation@refinitiv.com](mailto:ProductDocumentation@refinitiv.com).
- Add your comments to the PDF using Adobe's **Comment** feature. After adding your comments, submit the entire PDF to Refinitiv by clicking **Send File** in the **File** menu. Use the [ProductDocumentation@refinitiv.com](mailto:ProductDocumentation@refinitiv.com) address.



## 1.7 Document Conventions

### 1.7.1 Typographic

This document uses the following types of conventions:

- Methods, in-line code snippets, and types are shown in **Courier New** font.
- Parameters, filenames, tools, utilities, and directories are shown in **Bold** font.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.





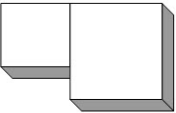





### 1.7.2 Diagrams

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



Figure 1. Running Performance Example and Host Notation

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

	Feed Handler, Refinitiv Real-Time server, or other application		Network of multiple servers
	Enterprise Message 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 Refinitiv Real-Time APIs. The tool suite covers the various Open Message Model-based API products and allows Refinitiv's internal and external clients to compare latency and throughput trade-offs of the various APIs and their differing functionality sets.

Refinitiv Real-Time Distribution System also offers the tools **testclient** and **testserver** for performance testing, focusing on throughput, latency, and capacity of Refinitiv 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.

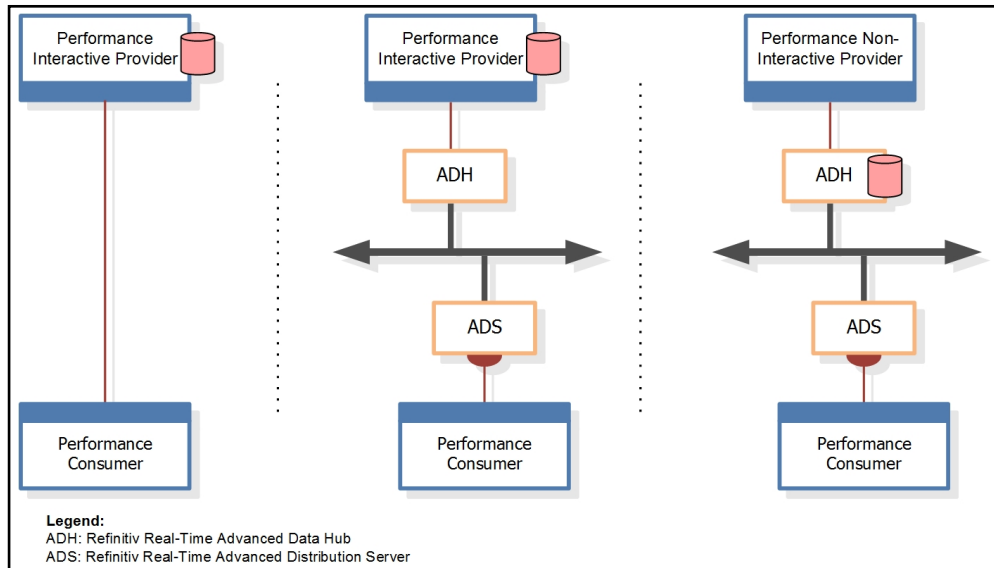
All tools in the suite are provided as buildable open-source and demonstrate best practice and coding for performance with their respective APIs. Future releases of API products will expand on these tests to include other areas of functionality (e.g., batch requesting, etc.). Clients can run these tools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

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

### 2.2 The Enterprise 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 Refinitiv 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 Refinitiv 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).<sup>1</sup>



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

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

In a typical Open Message Model configuration, latency through the system is measured either one-way from a provider to consumer, or round-trip from a consumer, through the system, and back.<sup>2</sup> 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.

## 2.3 Package Contents

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

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

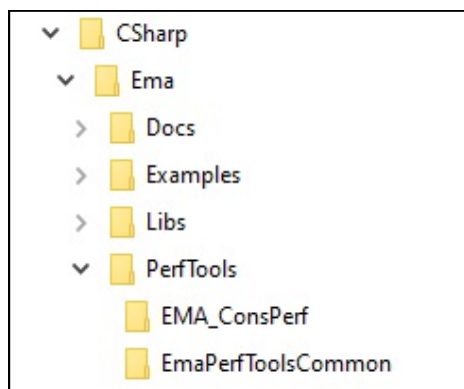


Figure 4. Directory Structure of the Performance Tools

### 2.3.1 XML Files

The **EmaPerfToolsCommon\_NET6.0** directory includes the following XML files:

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

### 2.3.2 Building and Running

In order to build and run the Consumer performance application, you need .NET 6.0 to be installed on your machine. The solution file **RTSDK\_NET6.0.sln** is located in the **CSharp** directory. In order to build it, execute the following command from **CSharp** directory:

```
dotnet build --configuration <Debug | Release> RTSDK_NET6.0.sln
```

For best performance, choose the Release configuration.

Performance tools use the XML files described in Section 2.3.1 as well as the dictionary files **enumtype.def** and **RDMFieldDictionary**. The latter files are located in the **CSharp** directory. Before running the applications, copy them to the output directories manually or by means of a post-build event script.

Run the applications from the respective output directories using the **dotnet** command.

Alternatively, open the solution in Microsoft Visual Studio 2022, build it via the menu Build -> Build Solution. Select **EMA\_ConsPerf**

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.

**application** from the list of startup projects, and run it.

## 2.4 What Gets Measured and Reported?

### 2.4.1 Latency

Each performance tool embeds timestamp information in its messages' payloads. The tool uses these timestamps to determine the overall time taken to send and process a message and its payload through the API and, where applicable, the Refinitiv 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 Refinitiv Real-Time Distribution System components provide higher throughput and lower latency than Robust Foundation API-based applications. In general, Refinitiv 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 tool measures the overall time taken to receive a full set of images for items requested through the system. This time is measured from the start of the first request to the reception of the final expected image.

### 2.4.4 CPU and Memory Usage

Performance tools record a periodic sampling of CPU and Memory usage. This allows for consistent monitoring of resource use and can be used to determine the impact of various features and application modifications. Performance Tools measure CPU usage and memory consumption using data provided by **System.Diagnostics.Process** class. The **Process.TotalProcessorTime.TotalMilliseconds** property is used to access the total time the process used the CPU, and **Process.PeakWorkingSet64** property is used to get the peak memory consumption.

- CPU usage calculation: the percentage of CPU used is calculated as **cpuUsage / (N \* timePassed)**, where N is the number of cores utilized by the application process, **cpuUsage** is the total processor time in milliseconds between two consecutive CPU usage statistics refreshes, and **timePassed** is the time between two consecutive CPU usage statistics refreshes.
- Memory Usage Calculation: memory usage is obtained via the **PeakWorkingSet64** property of the **Process** class, which provides peak memory usage in bytes.

## 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 **EMA\_ConsPerf** includes the average latency, update rate, and CPU/memory usage for the application's run time.

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

### 2.5.2 Statistics File

Each tool periodically records statistics relevant to that tool. For example, **EMA\_ConsPerf** records:

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

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

Each tool can configure statistics recording via the following options:

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

### 2.5.3 Latency File

You can configure **EMA\_ConsPerf** 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.

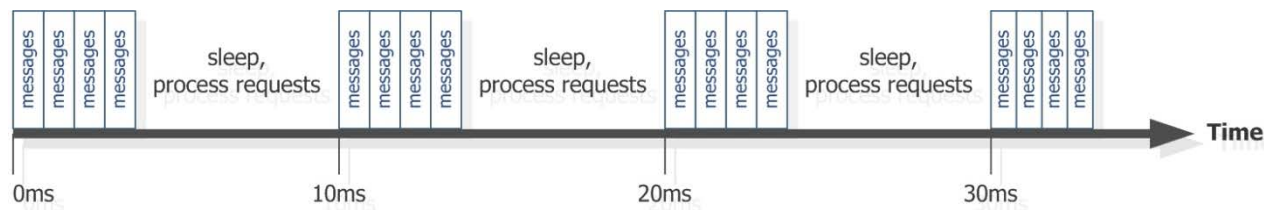
For example, an application might observe the following loop:

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.

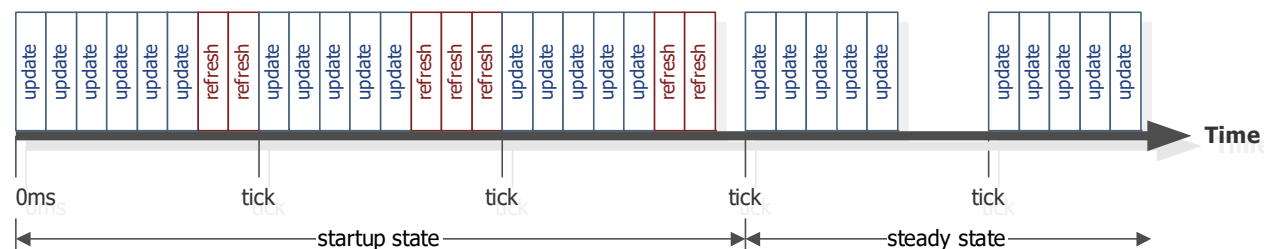
**NOTE:** `-tickRate` does not affect the Round Trip Time feature.

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 **EmajIProvPerf** or **EmajNIProvPerf** sends an update burst, the remaining time is used to send outstanding refreshes:



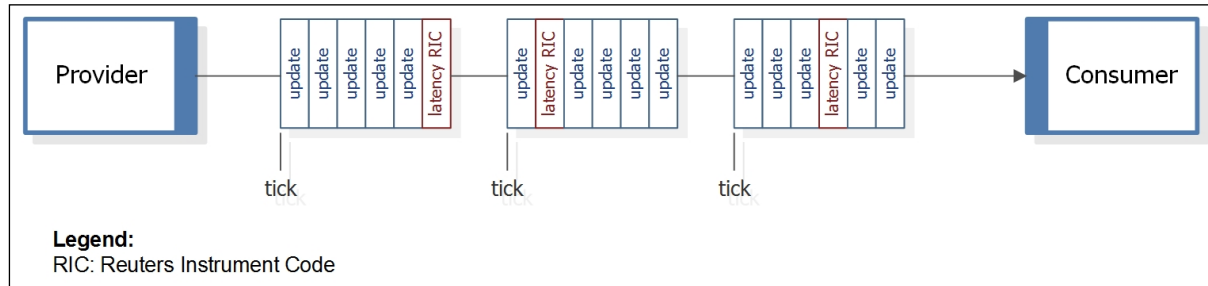
**Figure 6. Refresh Publishing Algorithm**

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.

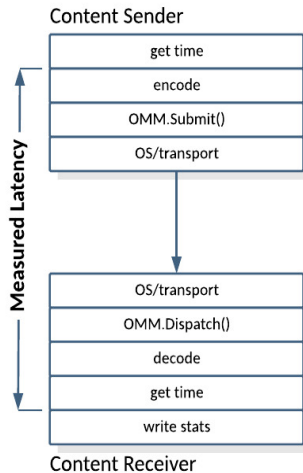


**Figure 7. Latency Reuters Instrument Codes within a Tick**

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 **EMA\_ConsPerf** generates both sending and receiving timestamps.

**NOTE:** Open Message Model performance tool timestamp information contains the number of microseconds since an epoch.<sup>a</sup>

a. Enterprise Message API C# uses **Stopwatch.GetTimestamp() / (Stopwatch.Frequency / 1000.0)** for microseconds.



**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 ConsPerf

### 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 **EMA\_ConsPerf** implements an Open Message Model consumer using the Enterprise Message API C# Edition. It connects to a provider (such as **EmajlProvPerf** or Refinitiv 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. **EMA\_ConsPerf** 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

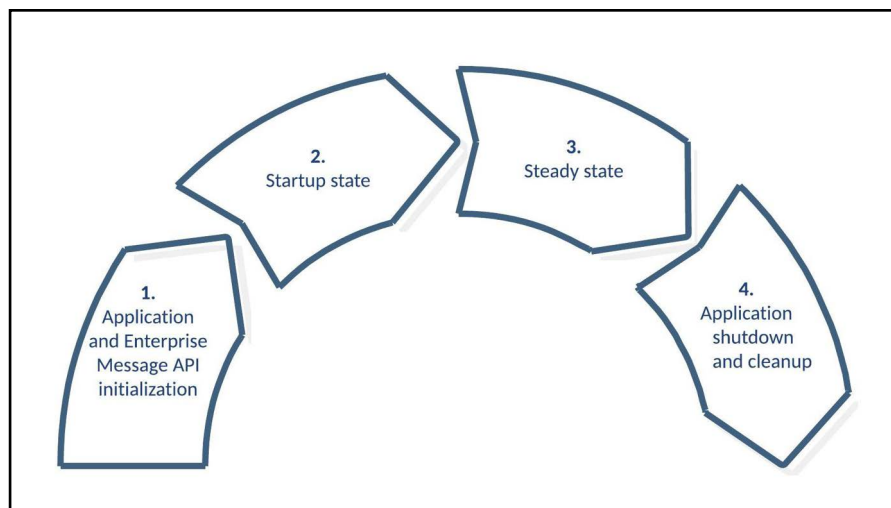
The Enterprise Message API is designed to allow calls from multiple threads, such that applications can scale their work across multiple cores. Applications can leverage this feature by creating multiple threads to handle multiple connections through the Enterprise Message API.

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

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

#### 4.2.1 Consumer Lifecycle

The lifecycle of **EMA\_ConsPerf** is divided into the following sections:



**Figure 9. EMA\_ConsPerf Lifecycle**

1. Application and Enterprise Message API Initialization.

**EMA\_ConsPerf** 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 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:
- 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.
  - 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:

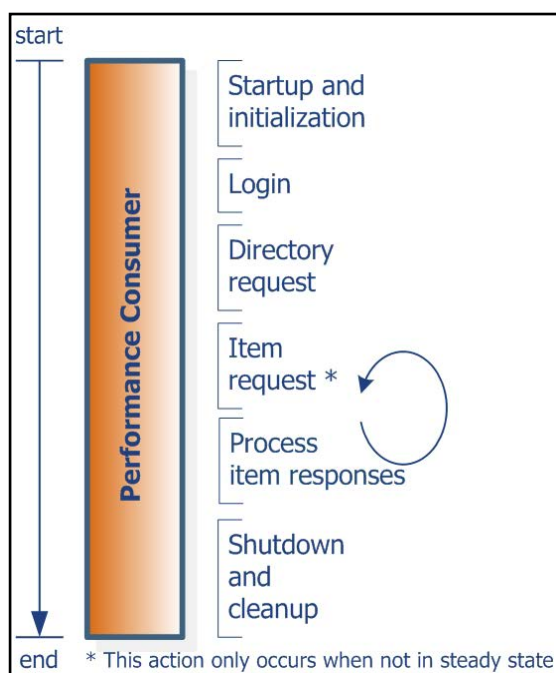
- If configured for posting, the thread sends a burst of post messages.
- updates from the provider.
- If configured to do so, sends a burst of generic messages.

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 Diagram



**Figure 10. EMA\_ConsPerf Application Flow**

## 4.3 Latency Measurement

Provider applications encode the timestamp as part of their message payload. The initial timestamp is taken at the start of encoding, and added as field TIM\_TRK\_1 (3902) in Update messages and TIM\_TRK\_3 (3904) in Generic messages. When this field is detected, the **EMA\_ConsPerf** gets the current time and computes the difference to measure latency.

When configured to do so via appropriate command line parameters, the Consumer application will encode timestamps as part of Generic messages payload. The timestamp is taken at the start of encoding and stored in the field TIM\_TRK\_3 (3904). The Performance Provider application can detect this field and calculate the latency by subtracting the received value from the current timestamp.

### 4.3.1 Consumer Latency

#### ► Consumer Latency Measurement Sequence:

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

### 4.3.2 Posting Latency

## 4.4 ConsPerf Configuration Options

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-commonItemCount	0	If multiple consumer threads are created (see <b>-threads</b> ), 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	<none>	Specifies the name of consumer in XML configuration file ( <b>EmaConfig.xml</b> ).
-delaySteadyStateCalc	0	Time the Consumer will wait before starting to calculate the latency. Since the Consumer has to handle pending update messages that were accumulated due to the processing of the refresh messages, there can be latency spikes for some period after reaching the steady state. Configuring this parameter appropriately can help to avoid the distortion of latency statistics due to these spikes.
-downcastDecoding	False	Turns on the EMA data load downcast feature during decoding response payload.
-genericMsgLatencyRate	0	Controls the number of generic messages sent that contain latency information. This must be less than to the total generic message rate (see <b>-genericMsgRate</b> ).
-genericMsgRate	0	Controls the number of generic messages sent per second. This cannot be less than the tick rate, unless it is zero.
-itemCount	100000	Sets the total number of items requested by the consumer.
-itemFile	350k.xml	Configures the name of the item list file.

**Table 2: EMA\_ConsPerf Configuration Options**

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-latencyFile	<none>	Sets the name of the log file in which <b>EMA_ConsPerf</b> logs the latency retrieved from individual latency updates, generic messages, and posts. If a name is not specified, logging is disabled.
-msgFile	MsgData.xml	Configures the name of the file used by the consumer to determine the makeup of message payloads.
-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 the total post message rate (see <b>-postingRate</b> ).
-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 <b>-tickRate</b> ).
-requestRate	13500	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	False	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 <b>-steadyStateTime</b> to 0 in that the requests are specifically made without the "STREAMING" <b>RequestMsg</b> flag.
-statsFile	ConsStats	Configures the base name that the consumer uses when writing its test statistics.
-steadyStateTime	300	Configures how long (in seconds) the consumer continues to run the test after receiving the last expected image.  <b>steadyStateTime</b> has a second function: after beginning the test, if the consumer does not receive all expected images within this segment of time, the consumer times out. In this case, it exits and indicates that it did not reach steady state.
-summaryFile	ConsSummary.out	Configures the name of the file to which the consumer writes its test summary.
-threads	<none>	Sets the number of threads the consumer starts. Each specified thread starts its own connection to the configured provider.
-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.
-useUserDispatch	False	Configures how <b>EMA_ConsPerf</b> and the Enterprise Message API dispatch receives messages. When you select <i>false</i> (API dispatch model), then <b>EMA_ConsPerf</b> configures the Enterprise Message API to create an additional internal thread to dispatch received messages. When you select <i>true</i> (user dispatch model) the Enterprise Message API does not run a second thread and the <b>EMA_ConsPerf</b> is responsible for dispatching all received messages. By default, <b>EMA_ConsPerf</b> uses the API dispatch model.  For details on how an Enterprise Message API application dispatches received messages, refer to the <i>Enterprise Message API Developers Guide</i> .

Table 2: EMA\_ConsPerf Configuration Options (Continued)

COMMAND-LINE OPTION	DEFAULT	DESCRIPTION
-useServiceId	False	Configures the consumer for adding the field "serviceId" in the Request message. For details on Request message, refer to <i>Enterprise Message API Refinitiv Domain Model Usage Guide</i> .
-uname	<none>	Configures the user name for the login request. When unspecified, the system login name is used.
-writeStatsInterval	5	Configures the frequency (in seconds) at which statistics are printed to the screen and statistics file.

**Table 2: EMA\_ConsPerf Configuration Options (Continued)**

## 4.5 Input

EMA\_ConsPerf requires the following files:

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

For more details on input file information, refer to Chapter 6, 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"/>
  <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, you must specify **Host** and **Port** 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"/>
<Host value="localhost"/>
<Port value="14002"/>
</Channel>

```

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

## 4.6 Output

EMA\_ConsPerf records statistics during a test such as:

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

For more details on output file information, refer to Chapter 7, Output File Details.

### 4.6.1 ConsPerf Summary File Sample

```

--- TEST INPUTS ---

    Steady State Time: 90
Delay Steady State Time: 0 msec
    Service: DIRECT_FEED
    UseServiceId: No
    Thread Count: 1
    Username: (use system login name)
    Item Count: 100000
    Common Item Count: 0
    Request Rate: 5000
    Request Snapshots: No
    Posting Rate: 0
    Latency Posting Rate: 0
    Generic Msg Rate: 0
Generic Msg Latency Rate: 0
    Item File: 350k.xml
    Data File: MsgData.xml

```

```

M    Summary File: example-summary.log
      Stats File: stats_file_example.log
      Latency Log File: latency-example.log
      Tick Rate: 1000
      Prime JVM: No
      DowncastDecoding: False
      OperationModel Usage: API_DISPATCH

--- OVERALL SUMMARY ---

Startup State Statistics:
  Sampling duration (sec): 19.202
  Latency avg (usec): 1850.1
  Latency std dev (usec): 3088.5
  Latency max (usec): 31027.0
  Latency min (usec): 82.0
  Avg update rate: 100076

Steady State Statistics:
  Sampling duration (sec): 90.419
  Latency avg (usec): 1139.3
  Latency std dev (usec): 597.8
  Latency max (usec): 3078.0
  Latency min (usec): 114.0
  Avg update rate: 101094

Overall Statistics:
  Sampling duration (sec): 109.621
  Latency avg (usec): 1270.1
  Latency std dev (usec): 1454.6
  Latency max (usec): 31027.0
  Latency min (usec): 82.0
  No GenMsg latency information was received.
  CPU/Memory samples: 25
  CPU Usage max (%): 168.94
  CPU Usage min (%): 0.00
  CPU Usage avg (%): 30.70
  Memory Usage max (MB): 3531.31
  Memory Usage min (MB): 3304.13
  Memory Usage avg (MB): 3480.88

Test Statistics:
  Requests sent: 100000
  Refreshes received: 100000
  Updates received: 11059036
  Image retrieval time (sec): 19.202
  Avg image rate: 5208
  Avg update rate: 100916

```

### Code Example 3: ConsPerf Summary File Sample

### 4.6.2 ConsPerf Statistics File Sample

```
UTC, Latency updates, Latency avg (usec), Latency std dev (usec), Latency max (usec), Latency min (usec),
Images, Update rate, Posting Latency updates, Posting Latency avg (usec),
Posting Latency std dev (usec), Posting Latency max (usec), Posting Latency min (usec),
GenMsgs sent, GenMsg Latencies sent, GenMsgs received, GenMsg Latencies received,
GenMsg Latency avg (usec), GenMsg Latency std dev (usec), GenMsg Latency max (usec),
GenMsg Latency min (usec), CPU usage (%), Memory(MB)
7/23/2021 11:36,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4.36,3317.54
7/23/2021 11:36,51,2975.3,5800.6,31027,137,30056,99750,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,168.94,3377.99
7/23/2021 11:36,52,1480.5,744.4,2969,82,30369,99988,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,118.56,3446.14
7/23/2021 11:37,48,1532.9,706.4,3441,523,30273,100001,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,119.58,3510.69
7/23/2021 11:37,53,1371,678.6,2721,143,9302,99998,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,118.7,3531.31
7/23/2021 11:37,48,1138.4,592.9,2347,155,0,100250,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,111.87,3530.68
```

**Code Example 4: ConsPerf Statistics File Sample**

### 4.6.3 ConsPerf Latency File Sample

```
Message type, Send Time, Receive Time, Latency (usec)
Upd, 418804488530, 418804510273, 21743
Upd, 418804506639, 418804537666, 31027
Upd, 418804606015, 418804611456, 5441
Upd, 418804642580, 418804646533, 3953
Upd, 418804745927, 418804768476, 22549
Upd, 418804860607, 418804866335, 5728
Upd, 418805190939, 418805191244, 305
Upd, 418805330612, 418805332488, 1876
Upd, 418805430248, 418805430908, 660
Upd, 418805430828, 418805431667, 839
Upd, 418805476205, 418805476928, 723
Upd, 418805492069, 418805493319, 1250
Upd, 418805809690, 418805810589, 899
Upd, 418805831271, 418805832032, 761
Upd, 418805892999, 418805893611, 612
```

**Code Example 5: ConsPerf Latency File Sample**

### 4.6.4 ConsPerf Console Output Sample

```
005: Images: 100000, UpdRate: 66793, CPU: 0.00%, Mem: 2191.73MB
Latency(usec): Avg:23798.1 StdDev:49182.1 Max:178689.0 Min: 98.0, Msgs: 33
- Image retrieval time for 100000 images: 1.222s (81813 images/s)
010: Images: 0, UpdRate: 99866, CPU: 0.00%, Mem: 2191.73MB
Latency(usec): Avg: 138.1 StdDev: 25.7 Max: 203.0 Min: 85.0, Msgs: 52
015: Images: 0, UpdRate: 99900, CPU: 0.00%, Mem: 2191.73MB
```



```
Latency(usec): Avg: 139.3 StdDev: 29.8 Max: 194.0 Min: 100.0, Msgs: 52
020: Images:      0, UpdRate:    99900, CPU: 100.00%, Mem: 2191.73MB
Latency(usec): Avg: 143.0 StdDev: 28.3 Max: 185.0 Min: 100.0, Msgs: 45
025: Images:      0, UpdRate:    99900, CPU:  0.00%, Mem: 2191.73MB
Latency(usec): Avg: 111.0 StdDev: 13.3 Max: 164.0 Min:  96.0, Msgs: 51
030: Images:      0, UpdRate:    99920, CPU:  0.00%, Mem: 2191.73MB
Latency(usec): Avg: 120.1 StdDev: 12.3 Max: 151.0 Min: 102.0, Msgs: 51
035: Images:      0, UpdRate:    99900, CPU:  0.00%, Mem: 2191.73MB
Latency(usec): Avg: 121.7 StdDev: 22.3 Max: 180.0 Min:  99.0, Msgs: 53
```

#### Code Example 6: ConsPerf Console Output Sample

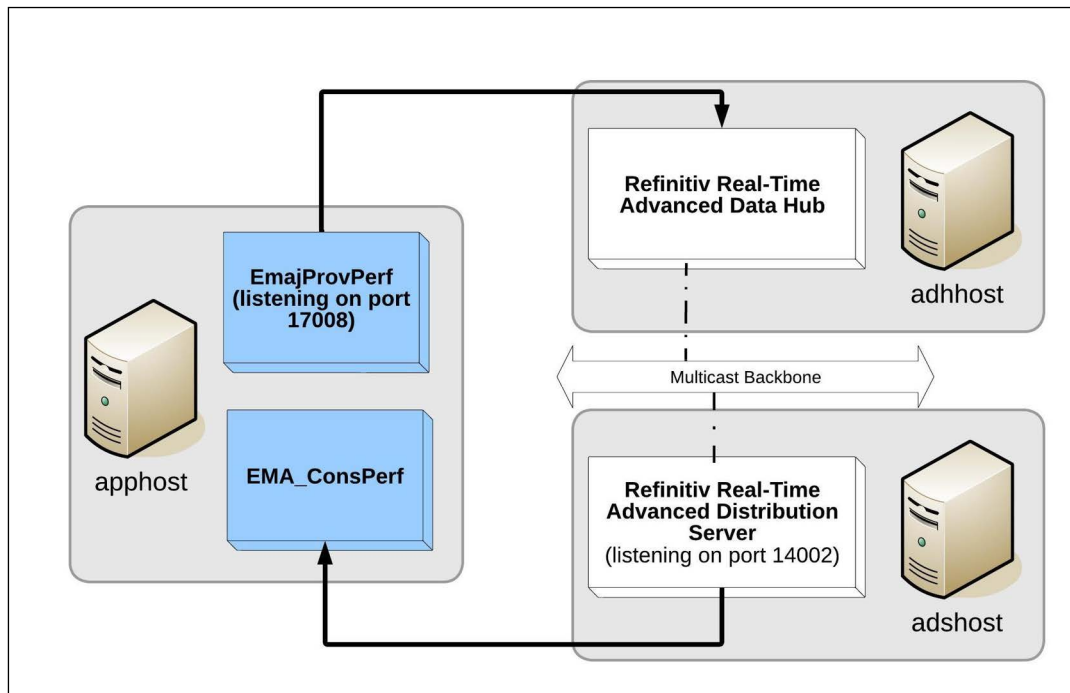
## 5 Performance Measurement Scenarios

### 5.1 Interactive Provider to Consumer, Through Refinitiv Real-Time Distribution System

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

- Connect **EMA\_ConsPerf** to a Refinitiv Real-Time Advanced Distribution Server.
- Connect the Refinitiv Real-Time Advanced Distribution Server to a Refinitiv Real-Time Advanced Data Hub. You can do so using the RRCP backbone.
- Connect the Refinitiv Real-Time Advanced Data Hub with an instance of **EmajlProvPerf** or **EMAC ProvPerf**

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



**Figure 11. Interactive Provider to Consumer on Refinitiv Real-Time Distribution System**

► To run a basic performance measurement:

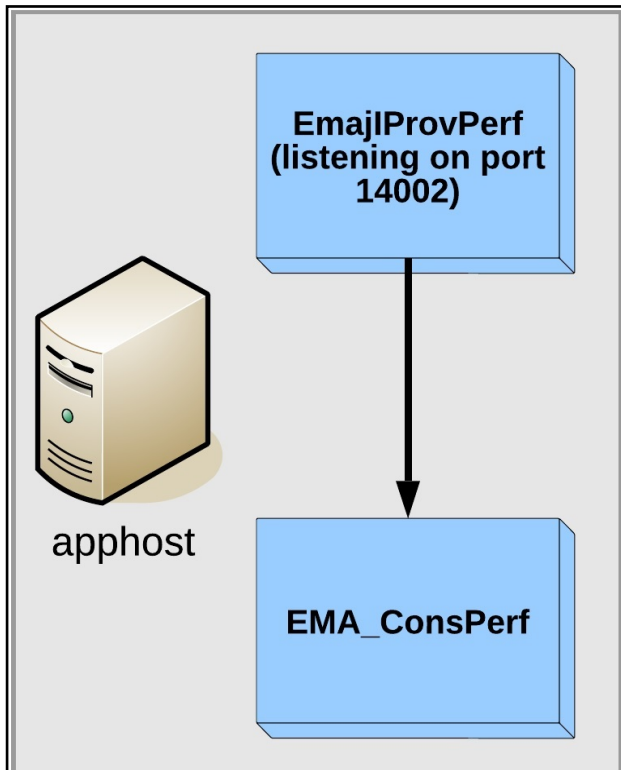
1. Configure **Perf\_Server\_1**, change `<Port value="17008" />`.
2. Configure **Perf\_Channel\_1**, change `<Host value="adshost" />`, `<Port value="14002"/>`.
3. Configure **Directory\_2**, change `<Service><Name value="TEST_FEED"/>`.

Run **ProvPerf** and **EMA\_ConsPerf** with the following command-line options. These options assume TEST\_FEED is the service being used and 17008 is the port number. Modify the example values as necessary.

```
./gradlew runEMAPerfProvider --args="-providerName Perf_Provider_1"
> ./EMA_ConsPerf -serviceName DIRECT_FEED -consumerName Perf_Consumer_1
```

## 5.2 Interactive Provider to Consumer, Direct Connect

You can measure the interactive providers of data by connecting **EMA\_ConsPerf** directly to **EmajlProvPerf**.



**Figure 12. Interactive Provider to Consumer, Direct Connect**

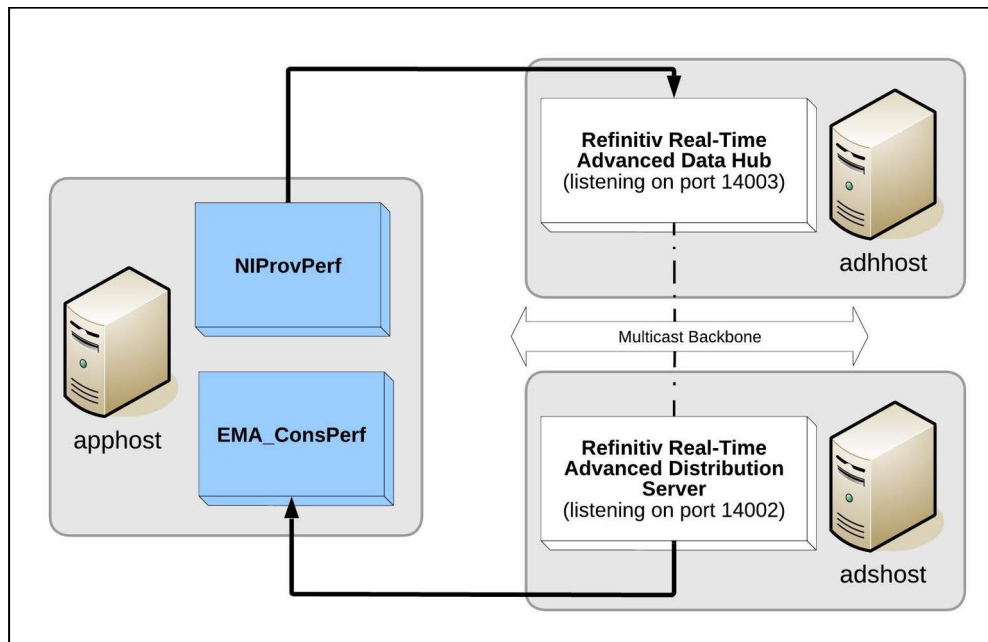
Using their default configuration options, you can run this test without any additional command-line options. Simply run the provider and consumer applications as follows:

```
./gradlew runEMAPerfProvider  
> ./EMA_ConsPerf
```

### 5.3 Non-Interactive Provider to Consumer, Through Refinitiv Real-Time Distribution System

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

- Connect **EMA\_ConsPerf** to a Refinitiv Real-Time Advanced Distribution Server.
- Connect the Refinitiv Real-Time Advanced Distribution Server with a Refinitiv Real-Time Advanced Data Hub. You can do so by using the RRCP backbone.
- Connect **EmajNIProvPerf** to the Refinitiv Real-Time Advanced Data Hub. Ensure that the Refinitiv Real-Time Advanced Data Hub has caching enabled, because it acts as the cache of record in this scenario.



**Figure 13. EmajNIProvPerf to Consumer on the Refinitiv Real-Time Distribution System**

**EMA\_ConsPerf** may receive a Closed status if it requests an item not yet provided by **EmajNIProvPerf** to the Refinitiv Real-Time Advanced Data Hub cache. To ensure the test completes successfully, you must do either one of the following:

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

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

► **To run a basic performance measurement:**

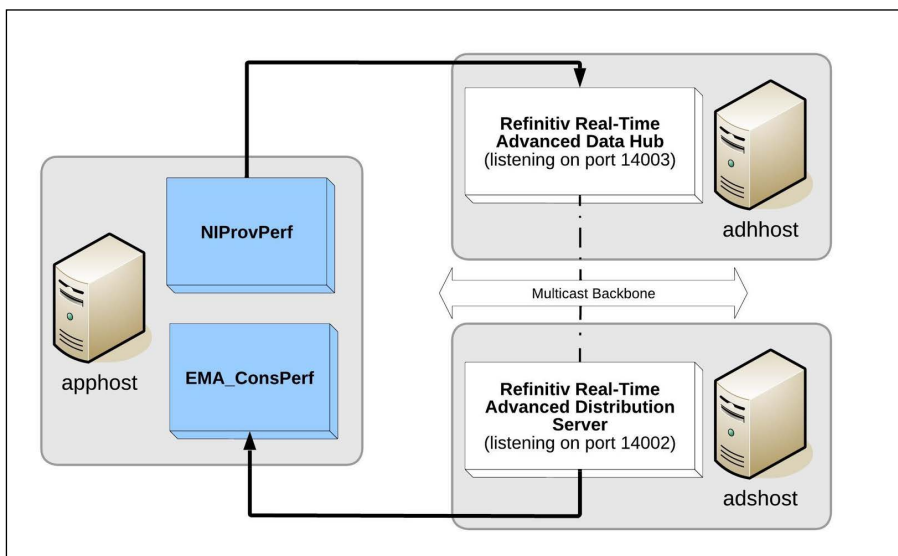
```
./gradlew runEMAPerfNIPProvider --args="-serviceName TEST_FEED -providerName Perf_NIPProvider_1"
> ./EMA_ConsPerf -serviceName TEST_FEED -consumerName Perf_Consumer_1
```

## 5.4 Consumer Posting on the Refinitiv Real-Time Distribution System

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

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

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



**Figure 14. Consumer Posting to Refinitiv Real-Time Distribution System**

Update traffic is optional. If you want to test posting without updates, configure 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 Refinitiv Real-Time Distribution System to provide the necessary service information and refresh content. For more details on this configuration, refer to the *Refinitiv Real-Time Advanced Data Hub Software Installation Manual*.

### C++ or Java NIProvPerf

```
./gradlew runEMAPerfNIPProvider --args="-serviceName TEST_FEED -providerName Perf_NIPProvider_1"
> ./EMA_ConsPerf -consumerName Perf_Consumer_1 -postingRate 10000 -postingLatencyRate 10
```

## 6 Input File Details

### 6.1 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 **EMA\_ConsPerf**.
- At least one generic message, if configured for exchanging generic messages.

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

#### 6.1.1 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.

### 6.1.2 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 7: Sample Update Message**

## 6.2 Item List File

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

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

### 6.2.1 Item Attributes

ATTRIBUTE NAME	DEFAULT	DESCRIPTION
domain	(none, required)	Specifies the domain from which the item is requested. This must be set to <b>MarketPrice</b> .
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 <b>true</b> , <b>EMA_ConsPerf</b> sends posts to this item (if posting is enabled).
snapshot	"false"	If set to <b>true</b> , <b>EMA_ConsPerf</b> requests this item as a snapshot (i.e., without setting the <b>STREAMING</b> flag on the request).

**Table 3: Item Attributes**

### 6.2.2 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 8: Sample Item List File**



## 7 Output File Details

### 7.1 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.

### 7.2 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	<i>ToolTypeStatsclient.csv</i> <sup>a</sup>	Specifies the filename of the statistics file produced.
-summaryFile	<i>ToolTypeSummary.txt</i>	Specifies the filename of the summary file produced.

**Table 4: Performance Suite Applications and Associated Configuration Files**

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

## 7.3 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 **EMA\_ConsPerf**.

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

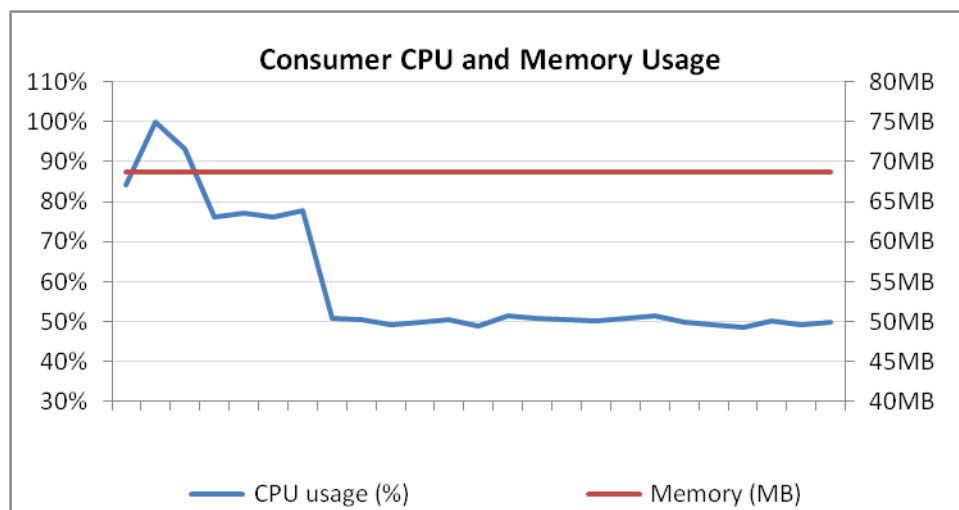
The interval in seconds that statistics are written to the file is controlled by the **writeStatsInterval** configuration parameter, which defaults to 5.

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

**Code Example 9: Sample ConsLatency.csv Showing Update and Post latencies during a Test Run**

## 7.4 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 15. Sample Excel Graph from ConsStats1.csv**

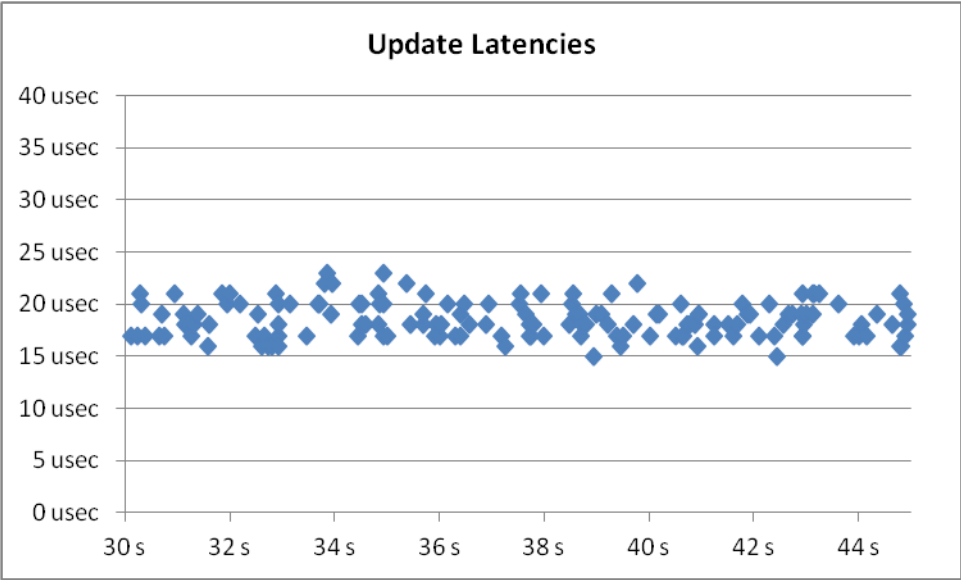


Figure 16. Sample Excel Graph of Latencies Over a 15-second Steady State Interval from ConsLatency1.csv

## 8 Performance Best Practices

### 8.1 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*.

### 8.2 Transport Best Practices

#### 8.2.1 Reading Data

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.

#### 8.2.2 Writing Data

To make efficient use of underlying transport method calls, the **Submit** method 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 **Write()** on the Channel instance in the Transport layer
- **HighWaterMark** configuration parameter in Channel or Server group causes **Submit** to pass queued content to the underlying transport.

#### 8.2.3 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.

#### 8.2.4 Direct Write

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

- If the queues are empty, the Enterprise Transport API passes data directly to the underlying transport, bypassing all queuing logic and delays.
- If the queues are not empty, the Enterprise Transport API adds data to the appropriate queue, with queued content being passed to the underlying transport in the appropriate order.<sup>1</sup>

Enterprise Message API allows to configure the Direct Write option for the Transport layer. Using this option can reduce latency,<sup>2</sup> as the message might not get queued. However this option also reduces throughput and increases CPU usage due to more frequent socket writes. You can offset the loss in throughput by packing buffers, though doing so can increase packing latency.

### 8.2.5 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, which disables Nagle's Algorithm.

### 8.2.6 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, content 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.

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

### 8.2.7 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.

#### 8.2.7.1 Input Buffering

The Enterprise Message API input buffer is created as one large continuous block of memory, controlled by The number of bytes created in the input buffer is determined by the configured value multiplied by the negotiated **maxFragmentSize**. Having one large block of memory allows **IChannel.Read()** 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 **IChannel.Read()** 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.

---

1. As determined by the various priorities with which the content was written and the flush strategy you configure.  
2. As long as the underlying transport can accept the content.

### 8.2.7.2 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** setting controls the number of output buffers available exclusively to that channel, where all of these buffers are created upfront. **maxOutputBuffers**

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.

### 8.2.7.3 Fragmentation

The negotiated maximum buffer size is the maximum size that the application will send in a single buffer. In cases where an application larger than the maximum, the requested size will be returned to the user. When the content passes to, 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 is large enough for commonly sent messages to fit into a single buffer.

## 8.2.8 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.<sup>3</sup>
- 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.

---

3. 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 Refinitiv Real-Time Advanced Data Hub (**adhmon**) and Refinitiv Real-Time Advanced Distribution Server (**adsmon**) to see whether the desired service is up.
- Check the Refinitiv Real-Time Advanced Data 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 Refinitiv Real-Time Distribution System, check that the desired service is up on both the Refinitiv Real-Time Advanced Data Hub and Refinitiv 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 Refinitiv Real-Time Advanced Data Hub cache.

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

In some cases, when connecting to Refinitiv 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 Refinitiv 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 17. Refinitiv 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.
-



© 2023 Refinitiv. All rights reserved.

Republication or redistribution of Refinitiv content, including by framing or similar means, is prohibited without the prior written consent of Refinitiv. 'Refinitiv' and the Refinitiv logo are registered trademarks and trademarks of Refinitiv.

Any third party names or marks are the trademarks or registered trademarks of the relevant third party.

Document ID: EMACSharp310PETOO.230

Date of issue: November 2023

