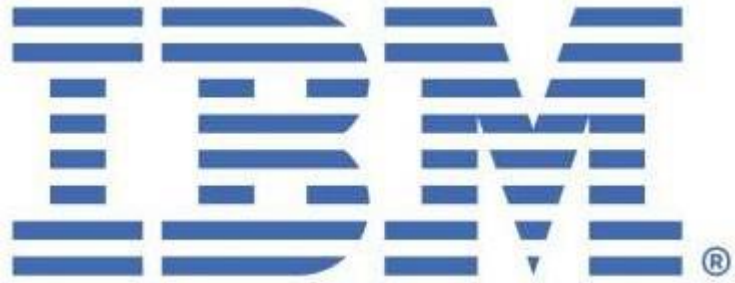

IBM MQ Telemetry Performance Report (x86-64 platform)

Version 1.0 - March 2026

Aneesha Yerrathota
IBM MQ Performance
IBM Bangalore
Laboratories
India

Notices

Please take Note!

Before using this report, please be sure to read the paragraphs on “disclaimers”, “warranty and liability exclusion”, “errors and omissions”, and the other general information paragraphs in the "Notices" section below.

First Edition, March 2026.

This edition applies to *IBM MQ V9.4.5* (and to all subsequent releases and modifications until otherwise indicated in new editions).

DISCLAIMERS

The performance data contained in this report was measured in a controlled environment. Results obtained in other environments may vary significantly.

You should not assume that the information contained in this report has been submitted to any formal testing by IBM.

Any use of this information and implementation of any of the techniques are the responsibility of the licensed user. Much depends on the ability of the licensed user to evaluate the data and to project the results into their own operational environment.

WARRANTY AND LIABILITY EXCLUSION

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

In Germany and Austria, notwithstanding the above exclusions, IBM's warranty and liability are governed only by the respective terms applicable for Germany and Austria in the corresponding IBM program license agreement(s).

ERRORS AND OMISSIONS

The information set forth in this report could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; any such change will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time and without notice.

INTENDED AUDIENCE

This report is intended for architects, systems programmers, analysts and programmers wanting to understand the performance characteristics of IBM MQ V9.4. The information is not intended as the specification of any programming interface that is provided by IBM MQ. It is assumed that the reader is familiar with the concepts and operation of IBM MQ V9.4.

LOCAL AVAILABILITY

References in this report to IBM products or programs do not imply that IBM intends to make these available in all countries in which IBM operates. Consult your local IBM representative for information on the products and services currently available in your area.

ALTERNATIVE PRODUCTS AND SERVICES

Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

USE OF INFORMATION PROVIDED BY YOU

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

TRADEMARKS AND SERVICE MARKS

The following terms used in this publication are trademarks of their respective companies in the United States, other countries or both:

- **IBM Corporation: IBM**
- **Oracle Corporation: Java**

Other company, product, and service names may be trademarks or service marks of others.

EXPORT REGULATIONS

You agree to comply with all applicable export and import laws and regulations.

Preface

This report presents the results of performance evaluations using the Eclipse Paho MQTT Client v3.1.1 with IBM MQ Telemetry 9.4.5 for Linux on x86_64 and is intended to assist with programming and capacity planning.

Target audience

The report is designed for people who:

- Will be designing and implementing solutions using IBM MQ Telemetry.
- Want to understand the performance limits of IBM MQ Telemetry.
- Want to understand what actions may be taken to tune IBM MQ Telemetry.

The reader should have a general awareness of the Linux operating system and of IBM MQ, Java programming language, and of IBM MQ Telemetry to make the best use of this report.

Whilst operating system, and MQ tuning details are given in this report (specific to the workloads presented), a more general consideration of tuning and best practices is available in a separate paper here:

https://ibm-messaging.github.io/mqperf/MQ_Performance_Best_Practices_v1.0.1.pdf

Feedback

We welcome feedback on this report.

- Does it provide the sort of information you want?
- Do you feel something important is missing?
- Is there too much technical detail, or not enough?
- Could the material be presented in a more useful manner?

Specific queries about performance problems on your IBM MQ system should be directed to your local IBM Representative or Support Centre.

Please direct any feedback on this report to aneesha.yerrathota@ibm.com.

Contents

Table of Contents

FIGURES

1 Introduction

IBM MQ Telemetry enables IBM MQ to connect with a wide range of telemetry devices, such as sensors, actuators, mobile phones, smart meters, medical devices, vehicles, and remote systems, using the MQTT protocol.

MQTT is a lightweight messaging protocol designed for communication between devices and message servers, especially in environments with limited device resources or unreliable networks. It is well suited to low-memory devices and networks with low bandwidth, high latency, or instability, and is widely used across industries such as energy, utilities, and retail.

The IBM MQ Extended Reach (MQXR) service, commonly known as the MQ Telemetry service, is a TCP/IP listener installed as an IBM MQ service. It starts and stops with the queue manager and appears as SYSTEM.MQXR.SERVICE when system objects are displayed. Its properties can be viewed and updated in the same way as any other IBM MQ service.

This report provides a performance overview of IBM MQ V9.4.5 Continuous delivery (CD) of MQTT Extended Reach (XR) on Linux on x86-64, focusing on large-scale deployments with up to 100,000 concurrent XR client connections using Eclipse Paho MQTT Client v3.1.1.

As with all performance sensitive tests, you should run your own tests where possible, to simulate your production environment and circumstances you are catering for.

For detailed information on configuring IBM MQ Telemetry and setting up MQTT clients, refer to the IBM MQ documentation: https://www.ibm.com/docs/en/ibm-mq/9.4.x?topic=overview-mq-telemetry

A sample MQTT configuration and setup guide for performance testing with MQTTPerfHarness is available at: https://github.com/ot4i/perf-harness/tree/main/samples/mqtt

2 Performance Workloads

This section describes the MQTT XR workload patterns used to evaluate IBM MQ performance on Linux on x86-64. The tests cover fan-in and fan-out scenarios at scale, with up to 100,000 XR client connections across different topic configurations and MQTT QoS levels.

In this report, an MQI client refers to an application that connects to IBM MQ using the native MQ Interface (MQI), as opposed to connecting through the MQTT protocol.

The scenarios covered in this report are:

MQ Telemetry Fan-In: 100,000 XR publishers connected to 1 topic with 1 MQI subscriber.
MQ Telemetry Fan-Out: 1 MQI publisher publishing 100,000 topics with 100,000 XR subscribers.
MQ Telemetry Multi-Topic Fan-In: 100,000 XR publishers connected across 5 topics with 5 MQI subscribers.
MQ Telemetry 50k Topic Fan-Out: 5 MQI publishers publishing 50,000 topics with 100,000 XR subscribers.
Each scenario is measured using the three MQTT quality of service (QoS) levels defined by MQTT Version 3:
QoS 0 – At most once delivery. This provides the highest performance but does not guarantee delivery.
QoS 1 – At least once delivery. This ensures messages are delivered at least once, though duplicates may occur.
QoS 2 – Exactly once delivery. This provides the highest level of delivery assurance, ensuring that each message is delivered once and only once.

The message size used in all tests is 256 bytes.

The graphs in this report show the number of messages per second processed by the connected publishers and subscribers. In fan-out scenarios, the reported rate reflects the total message delivery across all subscribers. The tables summarize the peak throughput achieved for each scenario, together with the corresponding client count and CPU utilization at peak throughput.

See Appendix C for terminology definitions used throughout this report

2.1 The Fan-In Workload

The fan-in workload measures the ability of IBM MQ XR to accept messages from many MQTT publishers and deliver them to IBM MQ subscribers. In this model, many XR clients publish to one or more MQTT topics, and the published messages are consumed by MQI subscriber applications connected through the native IBM MQ interface.

In the single-topic fan-in scenario, up to 100,000 publisher XR clients connect to the queue manager and publish to a single topic, with one MQI subscriber receiving the messages. In the multi-topic fan-in scenario, up to 100,000 publisher XR clients are distributed across five topics, with five MQI subscribers per topic. These scenarios are measured using MQTT QoS levels 0, 1, and 2.

This workload primarily stresses XR connection scale, publish processing, topic resolution, and delivery from MQTT publishers into IBM MQ subscribers. It is intended to represent environments in which a large population of telemetry XR clients send data into a smaller number of backend consumer applications.

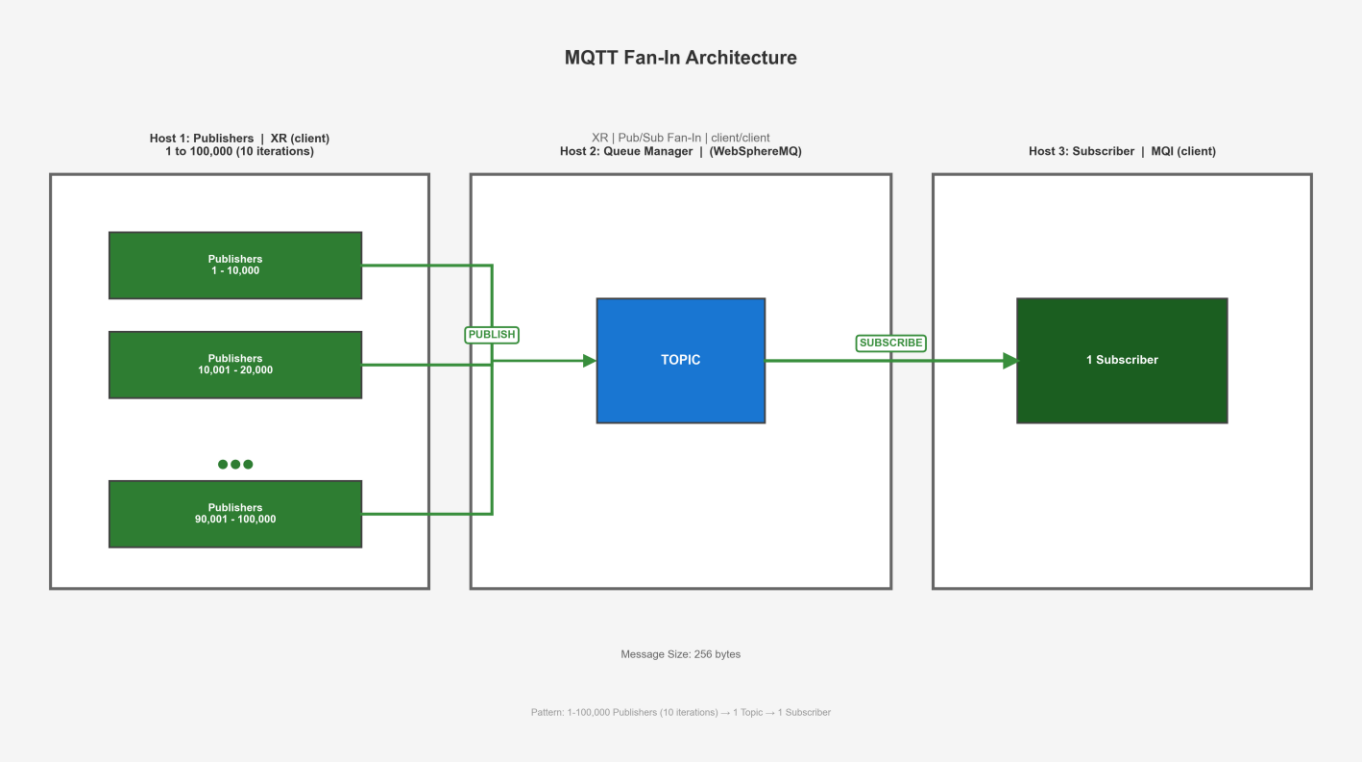

FIGURE 1 – XR FAN-IN TOPOLOGY

Figure 1 illustrates the topology used for the MQTT XR Fan-In workload. Many publisher XR clients connect to the queue manager using the MQTT protocol and publish messages to a topic hosted by the IBM MQ Telemetry (XR) service.

The published messages are routed through the queue manager and delivered to an MQI consumer application, which retrieves the messages using the native IBM MQ interface.

2.2 The Fan-Out Workload

The fan-out workload measures the ability of IBM MQ XR to distribute messages from a small number of publishing applications to a large population of MQTT subscribers. In this model, messages are published using the native IBM MQ interface and delivered to XR clients subscribed through the XR telemetry service.

In the first scenario, a single MQI publisher application sends messages to 100,000 topics hosted by the queue manager. Up to 100,000 subscriber XR clients connect using the MQTT protocol, with each XR client subscribed to a single topic. This scenario evaluates the scalability of the telemetry service when handling a very large number of topics and subscriber connections.

In the second scenario, five MQI publisher applications publish messages across 50,000 topics, with up to 100,000 subscriber XR clients distributed across those topics. This configuration introduces additional publishing concurrency while maintaining a large subscriber population. Reducing the number of topics allows the test to focus on the effects of multiple concurrent publishers and the fan-out delivery of messages to many XR subscribers.

These scenarios are measured using MQTT QoS levels 0, 1, and 2. This workload primarily stresses topic distribution, subscriber scaling, and the ability of IBM MQ Telemetry to efficiently deliver messages from MQ publishers to a large population of XR clients.

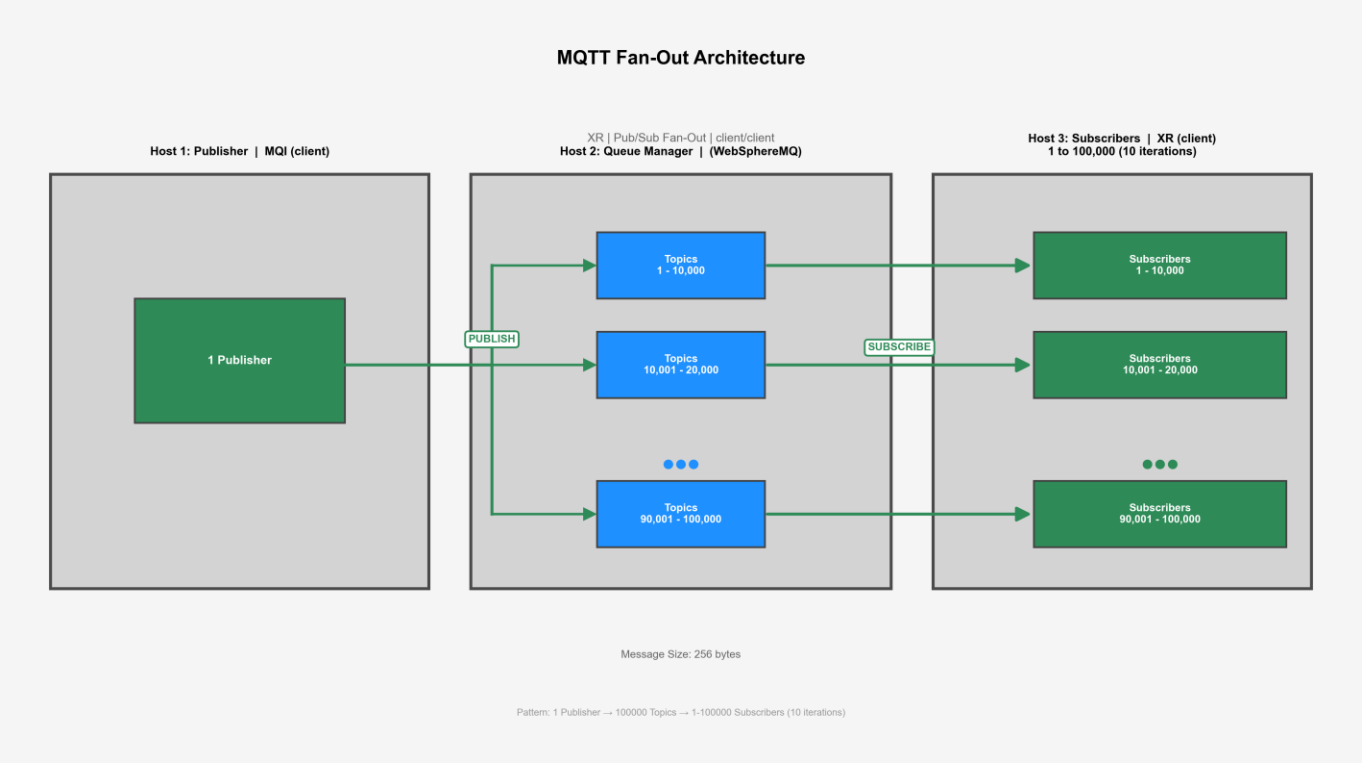

FIGURE 2 – XR FAN-OUT TOPOLOGY

Figure 2 illustrates the topology used for the MQTT XR Fan-Out workload. In this scenario, an MQI publisher application sends messages to topics hosted by the IBM MQ Telemetry (XR) service. Subscriber XR clients connect using the MQTT protocol and receive messages from their subscribed topics.

2.3 Test Calibration

Note that all tests in this report are rated. This means that a producer rate is set, which takes the overall throughput close to the limit for the test.

For the fan-in scenarios, the limiting factor is typically the rate at which the MQI subscriber application can consume messages delivered from the MQTT topic. Once the queue associated with the subscription begins to build up, performance may degrade.

For the fan-out scenarios, the limiting factors are the rate at which messages can be published by the MQI applications and the ability of the telemetry service to distribute those messages to many MQTT subscribers.

Many of the tests do not come close to exhausting CPU on the server machine. In this case, the workload could be scaled further by increasing the number of topics, publishers, or queue managers. The main purpose of the measurements below is to demonstrate that the product scales to many client connections without issue.

3 Test Results

3.1 Multi-Publisher, Few Subscribers

See section 2.1 for an overview of these scenarios.

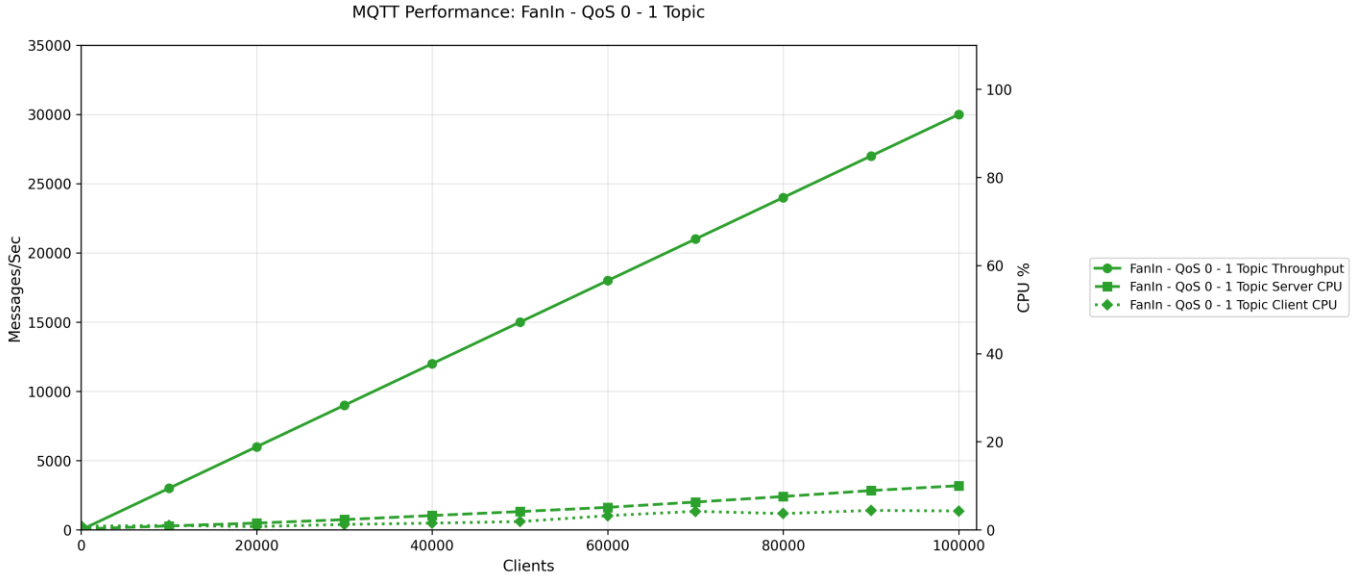

FIGURE 3 – XR FAN-IN QOS 0 WORKLOAD

XR publishers were started from a single client host and connected to the queue manager over a 100 Gb network link. Each client for JVM hosted 2000 publisher threads, with additional JVMs started as the number of publishers increased.

The publish rate for each publisher was 0.3 messages/sec so for 100,000 publishers there were potentially 100,000 * 0.3 msgs/sec through the system = 30,000 msgs/sec. Publish operations were performed in batches, with each publisher committing 1000 messages per batch. The MQI subscriber consumed messages in batches of 500 to reduce commit overhead during the measurements.

In this scenario, up to 100,000 XR publishers were connected to the queue manager and published messages to a single topic. A single MQI subscriber application consumed the messages delivered from the topic.

The workload achieved ~30,000 msgs/sec, which follows the theoretical maximum rate for this configuration. For QoS0, the test scaled cleanly as the number of publisher connections increased, demonstrating a near-linear increase in the overall messaging rate of the queue manager.

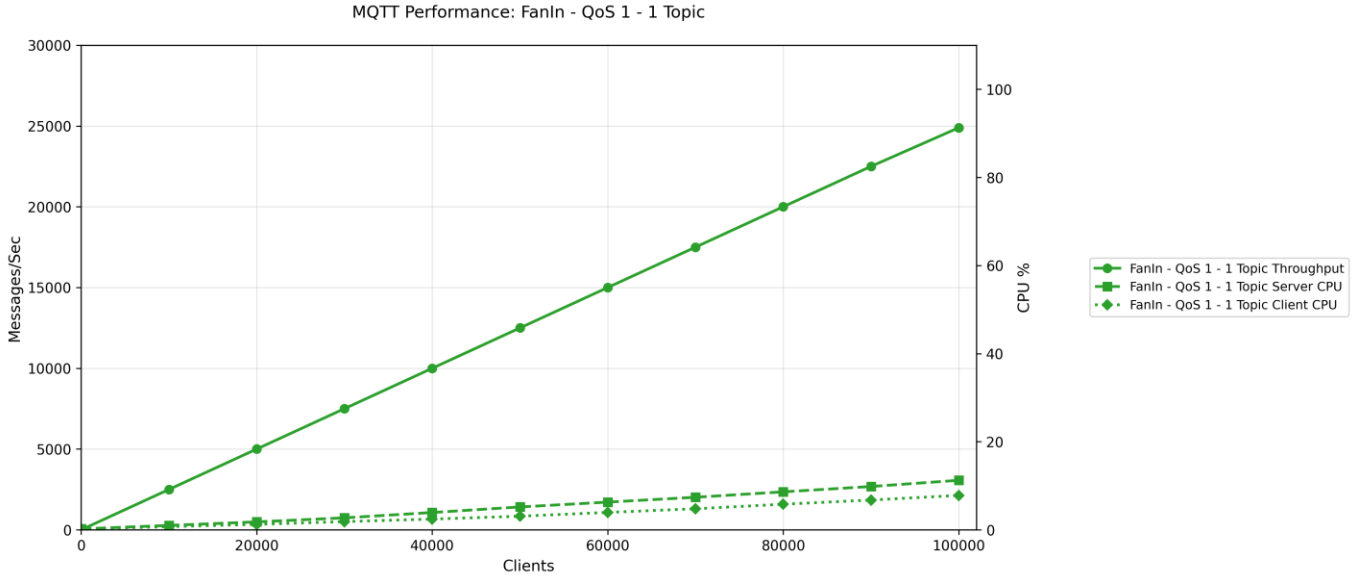

FIGURE 4 – XR FAN-IN QOS 1 WORKLOAD

The QoS1 test was setup identically to the QoS0 test, but the publish rate for each publisher was 0.25 msgs/sec so for every 100,000 publishers there were potentially 100,000 * 0.25 msgs/sec through the system = 25,000msgs/sec.

The workload achieved 24,892.13 msgs/sec, which closely follows the theoretical maximum rate for this configuration. The QoS1 measurements scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

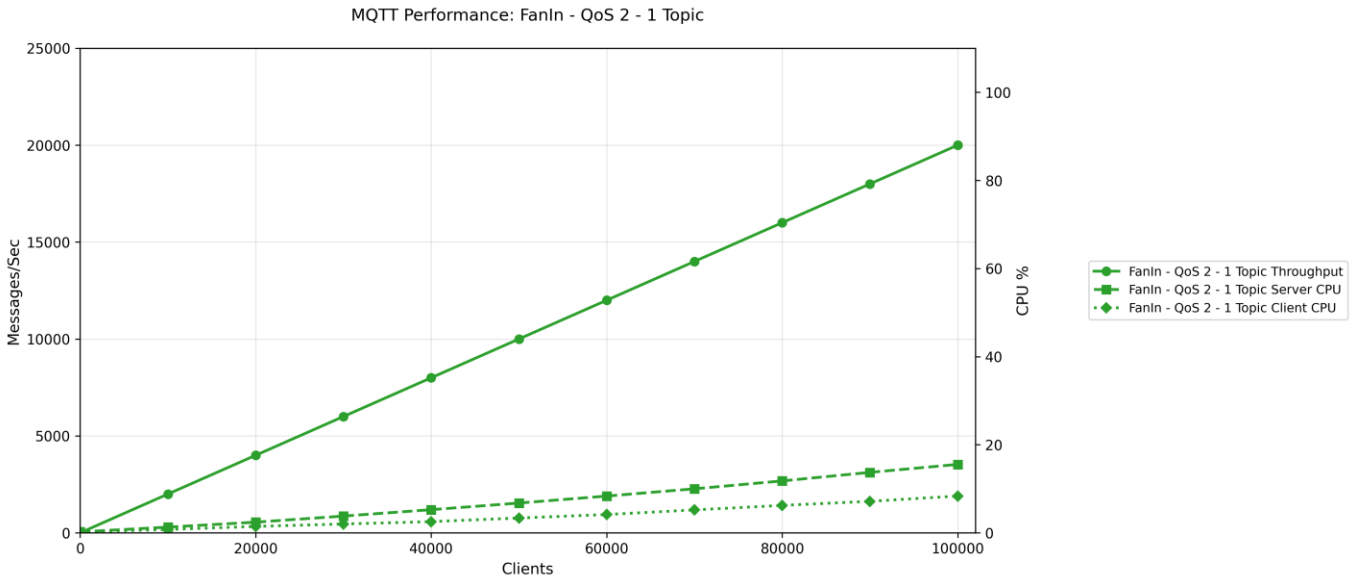

FIGURE 5 – XR FAN-IN QOS 2 WORKLOAD

The QoS2 test was setup identically to the QoS0 test, but the publish rate for each publisher was 0.2 msgs/sec so for every 100,000 publishers there were potentially 100,000 * 0.2 msgs/sec through the system = 20,000msgs/sec.

The workload achieved ~20,000 msgs/sec, which follows the theoretical maximum rate for this configuration. The QoS2 measurements scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

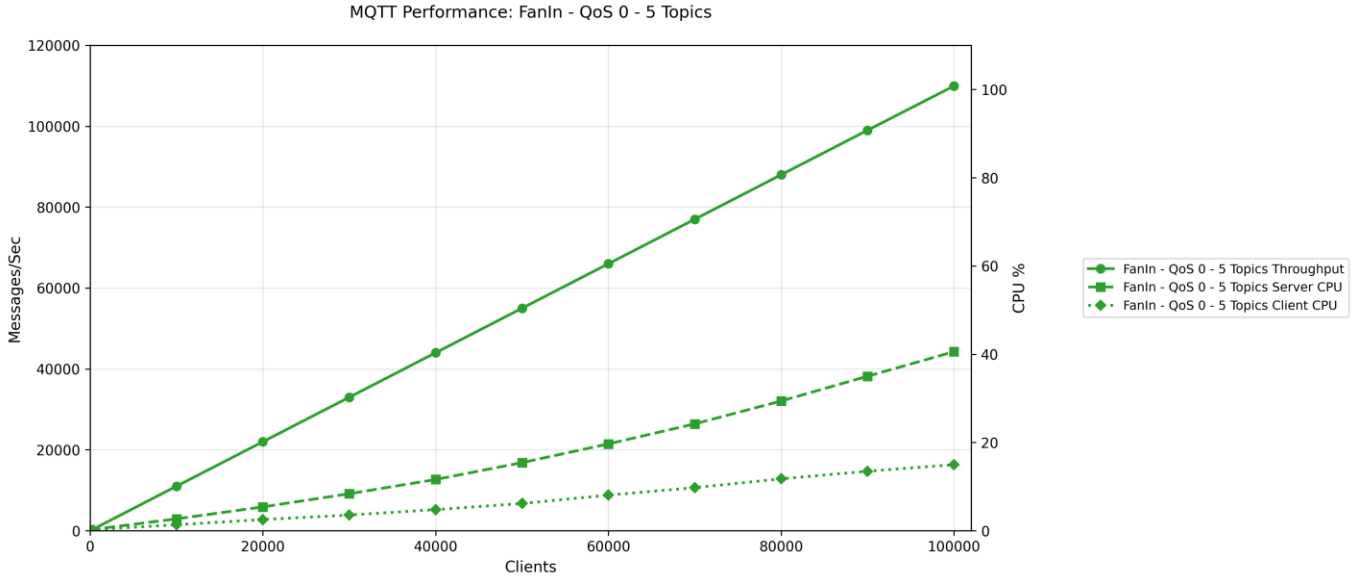

FIGURE 6 – XR FAN-IN QOS 0 MULTI-TOPIC WORKLOAD

The multi-topic fan-in test was configured similarly to the single-topic fan-in workload, but the publishers were distributed across five topics. Each topic was associated with a separate MQI subscriber application which consumed the messages delivered from the topic.

XR publishers were started from a single client host and connected to the queue manager

over a 100 Gb network link. Each client for JVM hosted 2000 publisher threads, with additional JVMs started as the number of publishers increased. For the multi-topic tests, publisher batching was increased to 5000 messages while the subscriber batch size remained at 500.

The publish rate for each publisher in this multi-topic QoS0 test was 1.1 msgs/sec, so for 100,000 publishers there were potentially 100,000 * 1.1 msgs/sec through the system = 110,000 msgs/sec.

The workload achieved 109,946.48 msgs/sec, which closely follows the theoretical maximum rate for this configuration. With the workload distributed across multiple topics and subscribers, the queue manager was able to process messages concurrently across the topic subscriptions. The results show that the queue manager was able to efficiently process the workload with 100,000 publisher connections distributed across five topics.

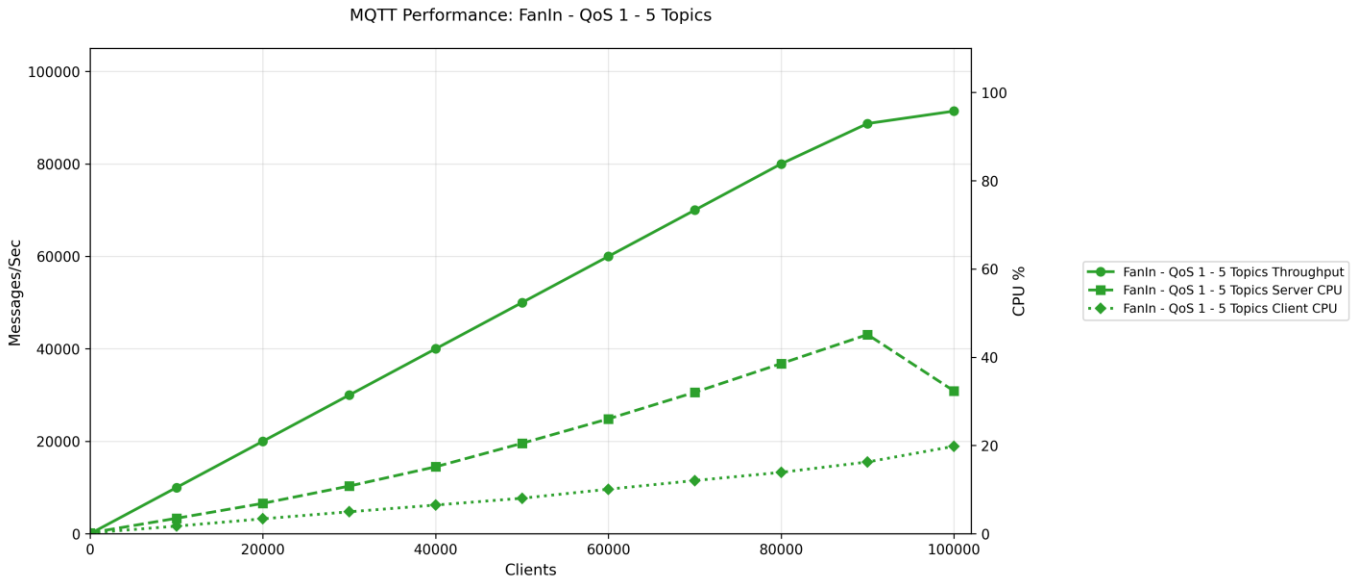

FIGURE 7 – XR FAN-IN QOS 1 MULTI-TOPIC WORKLOAD

The QoS1 test was setup identically to the QoS0 test but the publish rate for each publisher was 1 msgs/sec so for every 100,000 publishers there were potentially 100,000 * 1 msgs/sec through the system = 100,000 msgs/sec, with 100,000 publishers connected to the queue manager, the system achieved a peak messaging rate of approximately 91409.095 msgs/sec across the five topics.

The QoS1 measurements scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

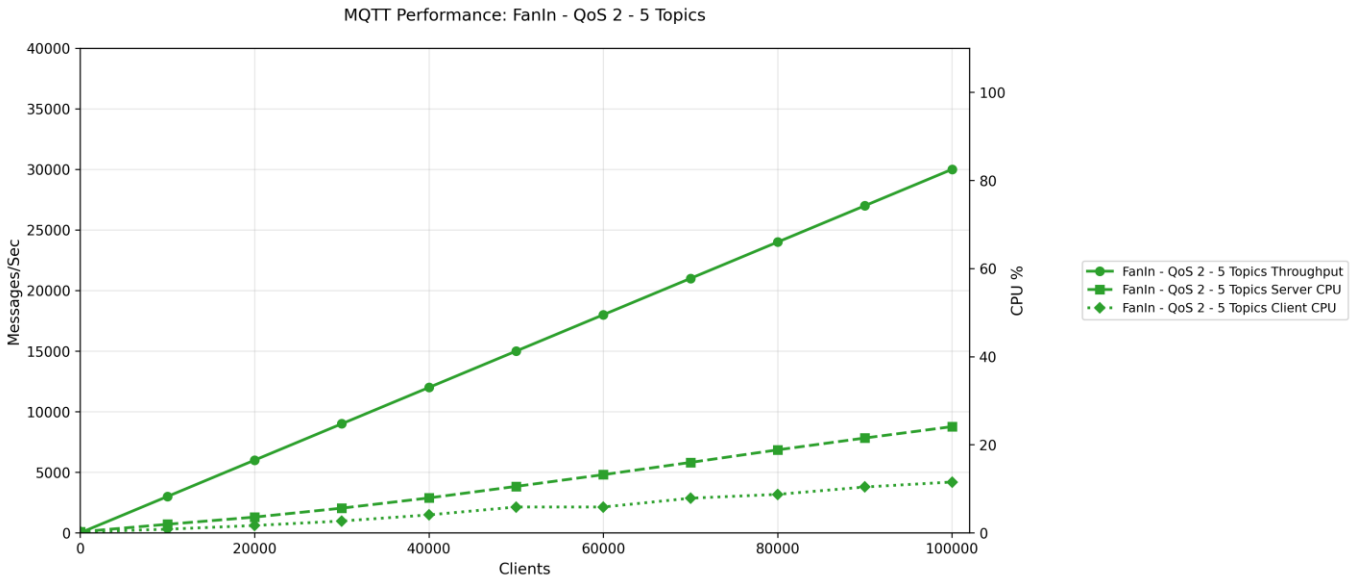

FIGURE 8 – XR FAN-IN QOS 2 MULTI-TOPIC WORKLOAD

The QoS2 test was setup identically to the QoS0 test, but the publish rate for each publisher was 0.30 msgs/sec so for every 100,000 publishers there were potentially 100,000 * 0.30 msgs/sec through the system = 30,000 msgs/sec.

The workload achieved 30,000 msgs/sec, which follows the theoretical maximum rate for this configuration. The QoS2 measurements scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

3.2 Few-Publishers, Multi- Subscriber

See section 2.1 for an overview of these scenarios.

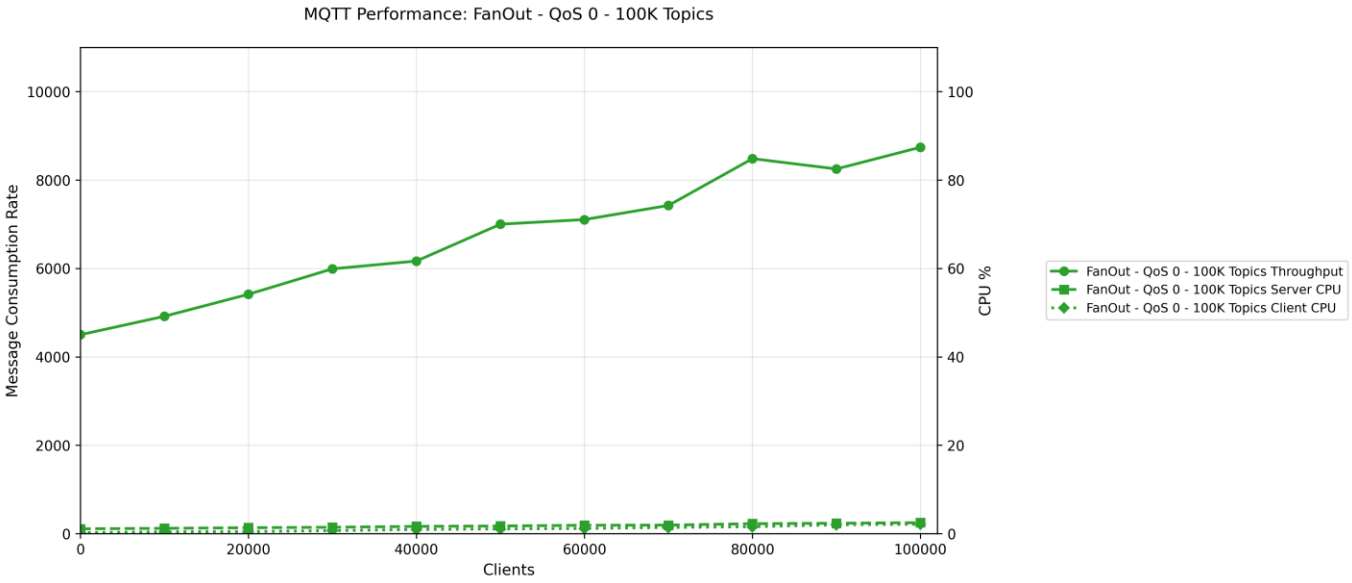

FIGURE 9 – XR FAN-OUT QOS 0 WORKLOAD

The QoS0 fan-out test was configured with 1 publisher, 100,000 topics, and 100,000 XR

subscribers, with one subscriber connected to each topic.

In the fan-out workload, a separate topic was created for each XR subscriber, resulting in 100,000 topics for 100,000 connected subscribers. This one-to-one mapping between topics and subscribers avoids the message of multiplication that would occur if many subscribers were attached to the same topic. The configuration therefore allows the test to focus on the scalability of topic management and message distribution across a very large number of topics and client connections.

Subscriber XR clients were started from a single client host. Each client for JVM hosted 2000 subscriber threads, with additional JVMs started as the number of subscribers increased.

Messages were published by the MQI publisher application at a configured rate of 900 msgs/sec. The peak messaging rate observed in this test was approximately 8740 msgs/sec across the subscriber population.

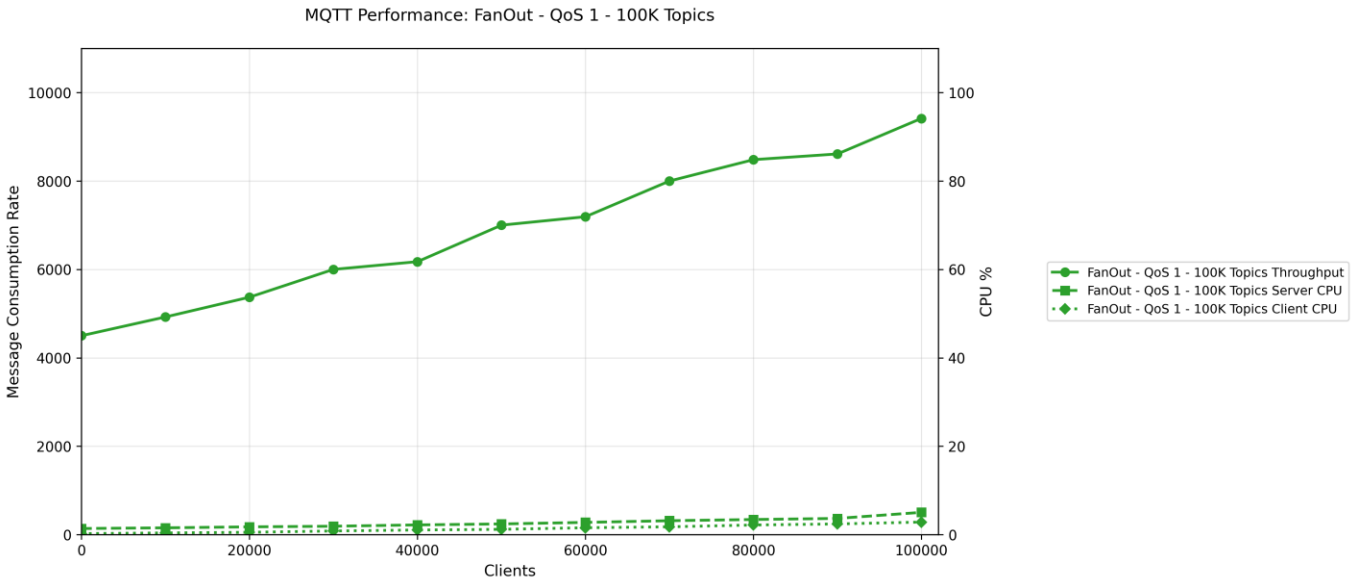

FIGURE 10 – XR FAN-OUT QOS 1 WORKLOAD

The QoS1 test was setup identically to the QoS0 test. Messages were published by the MQI publisher application at a configured rate of 900 msgs/sec. The peak rate observed in this test was approximately 9415 msgs/sec across the subscriber population.

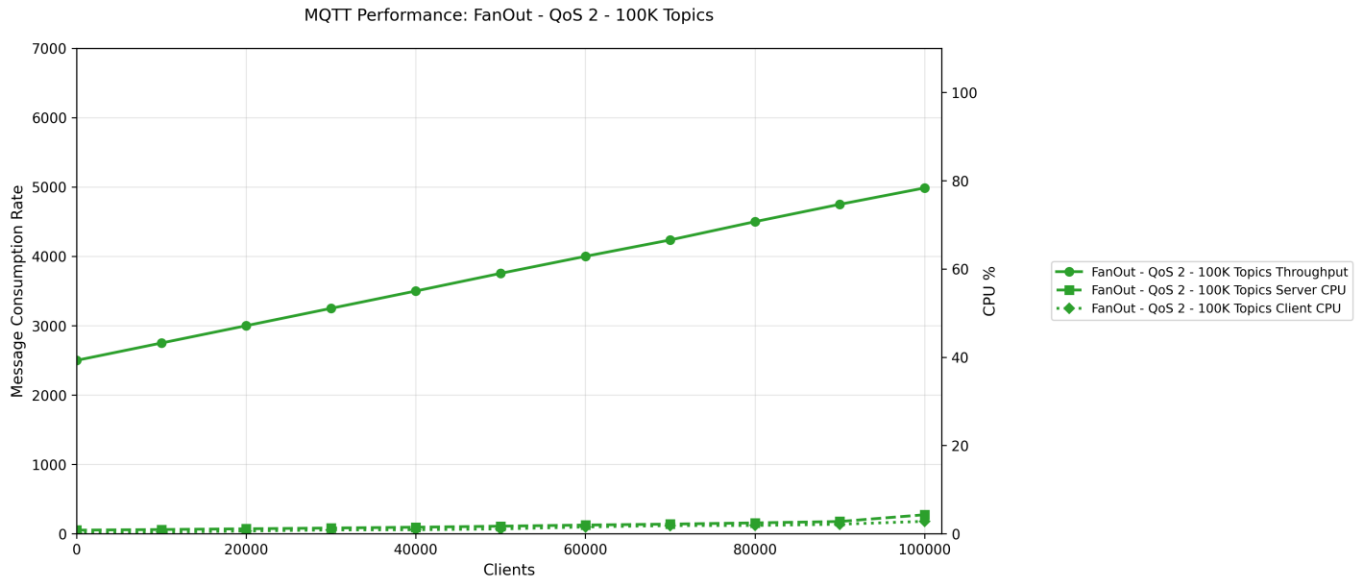

FIGURE 11 – XR FAN-OUT QOS 2 WORKLOAD

The QoS2 test was setup identically to the QoS0 test. Messages were published by the MQI publisher application at a configured rate of 500 msgs/sec. The peak rate observed in this test was approximately 4986 msgs/sec across the subscriber population.

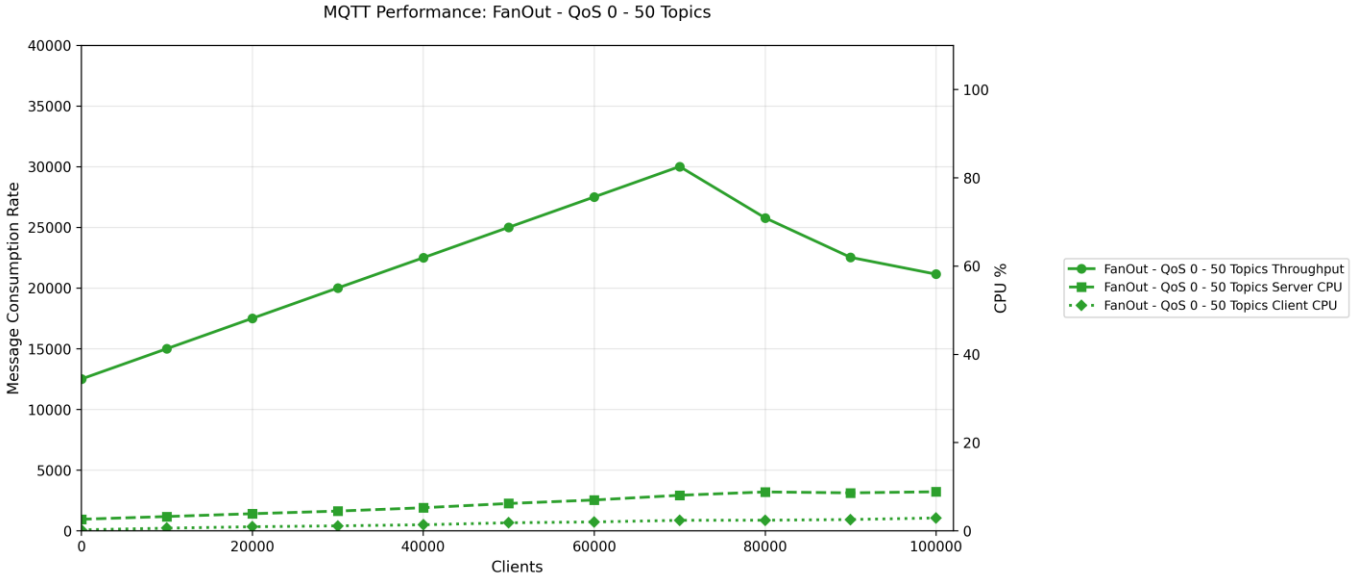

FIGURE 12 – XR FAN-OUT QOS 0 50K TOPIC WORKLOAD

In this fan-out scenario, the workload was modified to introduce additional publishing concurrency. Five MQI publisher applications were used to publish messages across 50,000 topics, with 100,000 XR subscribers distributed across those topics.

Reducing the number of topics while increasing the number of publishers allows the test to focus on the ability of the queue manager to process messages from multiple concurrent publishing applications and distribute them efficiently to a large population of MQTT

subscribers. This configuration therefore stresses both publisher's concurrency and message distribution across a large set of topics and subscribers.

Messages were published by the MQI publisher application at a configured rate of 2500 msgs/sec. The peak messaging rate observed in this test was approximately 30,008 msgs/sec across the subscriber population.

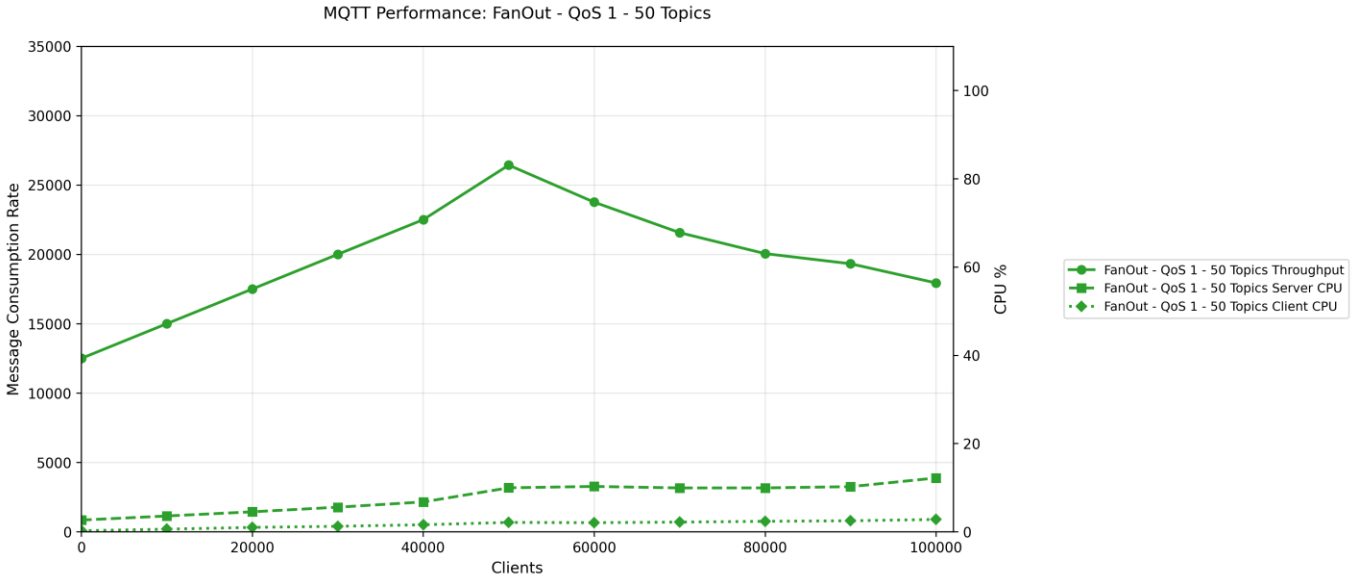

FIGURE 13 – XR FAN-OUT QOS 1 50K TOPIC WORKLOAD

The QoS1 test was setup identically to the QoS0 test. Messages were published by the MQI publisher application at a configured rate of 2500 msgs/sec. The peak rate observed in this test was approximately 26,430 msgs/sec across the subscriber population.

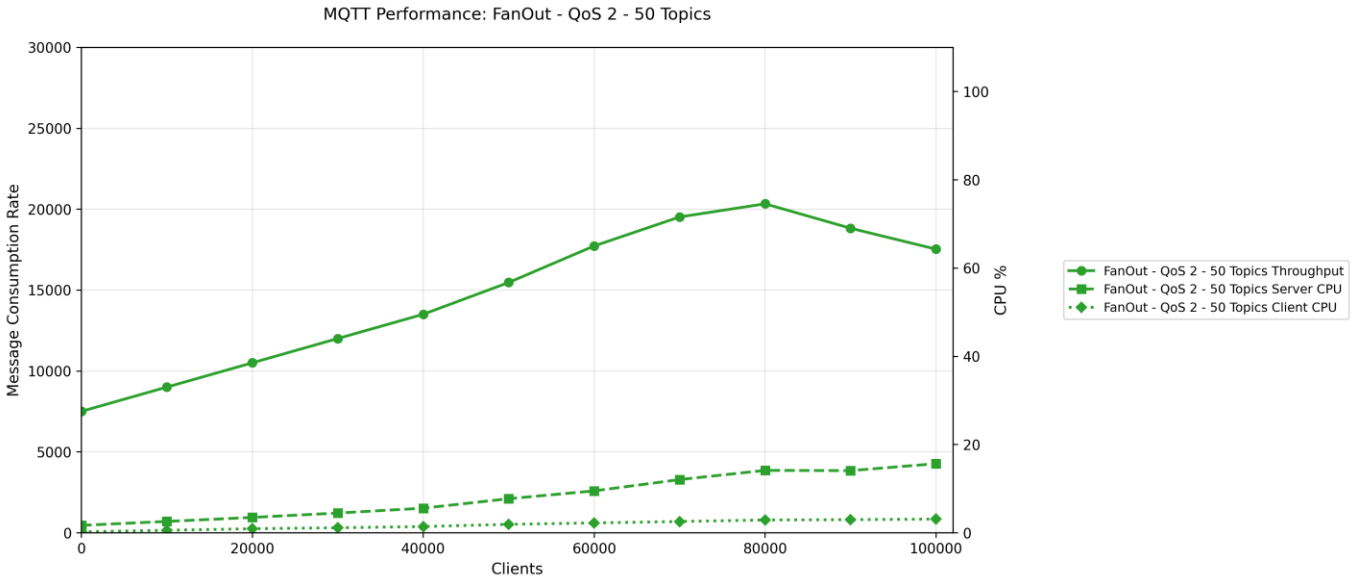

FIGURE 14 – XR FAN-OUT QOS 2 50K TOPIC WORKLOAD

The QoS2 test was setup identically to the QoS0 test. Messages were published by the MQI publisher application at a configured rate of 1500 msgs/sec. The peak rate observed in this test was approximately 20326 msgs/sec across the subscriber population.

4 Tuning Parameters and Considerations

This section describes the queue manager tuning parameters used for the measurements presented in this report and provides general tuning considerations. Some of these recommendations may not apply directly to the telemetry workloads used in this report for tests but are left here for your information.

4.1 Tuning the queue manager

The following tuning actions were applied to the queue manager used for these measurements.

• LogBufferPages = 4096

Size of memory used to build log I/O records.

• LogFilePages = 16384

Size of each log disk file extent.

• LogPrimaryFiles = 16

Number of primary log extents in the log cycle.

• LogWriteIntegrity = TripleWrite

Ensures the integrity of persistent log records.

• MQIBindType = FASTPATH

Allows channels to run as an extension of the queue manager address space to reduce context switching overhead.

• SHARECNV = 1

Disables channel sharing across multiple conversations to avoid contention when large numbers of telemetry clients are connected.

• DefaultQBufferSize = 50000000

Allocates memory to hold non-persistent messages in memory before spilling to the file system.

• DefaultPQBufferSize = 50000000

Allocates memory to buffer persistent messages.

4.2 Tuning the heap size for Java

During operation, current garbage collectors (GC) will normally interrupt the execution of all other threads in a JVM to some extent. The level of interruption depends on the amount and the type of work the GC is doing.

This is largely dependent on how the memory is being used by the application and the GC settings currently in operation. JMS has characteristics such that fixed memory requirements are low, but transient memory requirements can be high, depending on message size and application design. Without tuning, or with incorrect tuning, the automatic garbage collection policies of Java can adversely affect messaging performance. The most common GC settings are:

–Xms Minimum heap size.
–Xmx Maximum heap size.
–verbose:gc Display garbage collection events.

As an example, the following line fixes the heap size at 512MB and enables verbose garbage collection.

java –Xms512M –Xmx512M –verbose:gc

For the performance tests described in this paper, the Java runtime environment used was:

Java 21.0.9 LTS
IBM Semeru Runtime Open Edition
Eclipse OpenJ9 VM

The heap size for the MQXR service (`SYSTEM.MQXR.SERVICE`) was fixed at 4 GB using the following parameters in the `STARTARG options:`

-J "-Xms4096M" -J "-Xmx4096M"

For configuration details on running large-scale MQTT performance tests (up to 100,000 concurrent clients), see our: MQTTPerfHarness Tutorial
https://github.com/ot4i/perf-harness/blob/main/samples/mqtt/mqttperfharness_tutorial.md

Recommendations

- Use –verbose:gc to monitor the frequency of your application's garbage collection under different loads and adjust the minimum and maximum heap sizes accordingly.
- A garbage collection interval of less than one second is detrimental to performance. A sensible minimum GC interval is 1-2 seconds, but consideration should also be given to the GC pause time.
- If sufficient physical memory is available, configure –Xms and –Xmx to the same value to prevent heap resizing during runtime.
- JVM implementations provide multiple garbage collection policies. The OpenJ9 runtime used in these tests automatically selects an appropriate default policy, but GC behaviour should still be monitored when running workloads with different

message sizes or client concurrency levels.

4.3 JVM Warmup

JVMs employ sophisticated Just-In-Time (JIT) compilers to optimize the executable code. These JITs can continue to recompile selected java methods for many minutes or even hours after the jvm have initialized. Full performance may not be achieved until this is completed, and indeed the cost of compilation can slow down performance in the early stages of execution. In most cases, the default JIT settings will give best overall performance, but in situations where a faster startup is desirable, the JIT activity can be reduced at the expense of absolute performance.

4.4 Adjusting Linux cgroup Limits for Large Client Counts

When running large-scale MQTT workloads, such as 100,000 publishers or subscribers on a single host, the operating system may limit the number of processes or threads that can be created by a user. Modern Linux distributions enforce these limits using control groups (cgroups).

Even when traditional limits such as ulimit are configured with high values, the systemd cgroup configuration may still restrict the maximum number of tasks (threads or processes) that a user can create. If this limit is reached, additional MQTT clients may fail to start or connect.

In such cases, the cgroup limit for the user's slice must be increased. The current task limit for the user slice can be inspected using the following command:

```
systemctl show user-30000.slice | grep TasksMax
```

If large-scale workloads require more tasks than the current limit allows, the limit can be temporarily increased using:

```
systemctl set-property user-30000.slice TasksMax=1048576
```

4.5 Shared Conversations

IBM MQ channels support shared conversations, which allow multiple client connections to share a single TCP socket. This feature reduces the number of sockets required when multiple client threads operate within the same process. The maximum number of conversations that can share a channel instance is controlled by the SHARECNV attribute on the channel definition.

For workloads where clients produce or consume low message rates, shared conversations

can improve efficiency by reducing network resource usage. However, in high-throughput workloads where individual clients produce or consume large numbers of messages per second, shared conversations can become a bottleneck because multiple conversations compete for the same TCP socket.

For performance-sensitive workloads, it is recommended to configure channels with: SHARECNV (1). This ensures that each client's connection uses a dedicated TCP socket, avoiding contention caused by multiple conversations sharing the same connection.

4.6 Message Quality of Service

IBM MQ provides different message delivery guarantees that allow applications to balance reliability and performance. In general, stronger delivery guarantees require additional processing and therefore reduce the maximum achievable throughput.

The telemetry workloads in this report use the three MQTT Quality of Service (QoS) levels defined by the MQTT protocol. These QoS levels determine how messages are delivered and acknowledged, and map to different performance characteristics:

- QoS 0 – Messages are delivered with at-most-once semantics and do not require acknowledgements. This mode provides the highest performance.
- QoS 1 – Messages are delivered with at-least-once semantics and require acknowledgement by the receiver.
- QoS 2 – Messages are delivered with exactly-once semantics and involve additional MQTT protocol exchanges to guarantee message delivery.

MQTT QoS levels are often used in conjunction with transactional processing. For applications requiring strong delivery guarantees (QoS 2), transactional processing is typically used to ensure atomicity and consistency. For high-throughput scenarios where occasional message loss is acceptable (QoS 0), non-transactional processing provides better performance.

How IBM MQ Handles MQTT QoS Levels:

- QoS 0 messages are typically not written to the queue manager recovery log, providing higher performance. However, they may be lost if the queue manager stops unexpectedly.

- QoS 1 and QoS 2 messages are written to the recovery log and can be recovered following queue manager failures. This provides stronger delivery guarantees at the cost of additional processing overhead.

- In the case of a controlled shutdown and restart, QoS 0 messages can optionally be preserved by configuring the queue with NPMCLASS=HIGH.

Applications must select the appropriate QoS level based on their reliability requirements, as stronger delivery guarantees generally result in lower maximum messaging throughput. The

performance measurements in this report demonstrate the throughput impact of each QoS level under large-scale workloads.

5 Identifying Limits During Large-Scale Testing

5.1 Too Many Publishes in Progress (32202)

During some fan-in workloads using MQTT QoS1 and QoS2 messaging, the following exception was observed when running very large numbers of publishing XR clients.

Error Code: 32202 – Too many publishes in progress

```
t_logs]$ Publisher105: Uncaught exception.
:>> Too many publishes in progress (32202)
:>>         at org.eclipse.paho.client.mqttv3.internal.ClientState.send(ClientState.java:527)
:>>         at org.eclipse.paho.client.mqttv3.internal.ClientComms.internalSend(ClientComms.java:163)
:>>         at org.eclipse.paho.client.mqttv3.internal.ClientComms.sendNoWait(ClientComms.java:195)
:>>         at org.eclipse.paho.client.mqttv3.MqttTopic.publish(MqttTopic.java:128)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.Publisher.oneIteration(Publisher.java:88)
:>>         at com.ibm.uk.hursley.perfharness.WorkerThread.pace(WorkerThread.java:407)
:>>         at com.ibm.uk.hursley.perfharness.WorkerThread.pace(WorkerThread.java:507)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.MqttWorkerThread.run(MqttWorkerThread.java:188)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.Publisher.run(Publisher.java:78)
```

This Exception occurs when the MQTT client exceeds the maximum number of in-flight (unacknowledged) to publish operations. In MQTT QoS1 and QoS2 messaging, each published message requires an acknowledgement (PUBACK) from the server before it is removed from the in-flight window.

In the test configuration used for this workload, a very large number of publishers XR clients were generating messages concurrently. When publishers transmitted messages faster than acknowledgements could be processed by the broker, the client-side in-flight message window became full, and the client returned the above exception.

This behaviour is expected in high-throughput QoS1 and QoS2 workloads where client publish rates exceed the acknowledgement processing rate.

The Eclipse Paho MQTT client enforces a default limit of 10 in-flight messages (maxInflight). The maximum number of in-flight messages can be increased in the client configuration using the setMaxInflight() method, for example:

MqttConnectOptions options = new MqttConnectOptions();
options.setMaxInflight(1000).

To address this condition during testing, the publisher workload was distributed across multiple JVM processes and the configured publish rate was reduced to ensure that the rate of message transmission did not exceed the acknowledgement processing capacity of the system.

5.2 OutOfMemoryError When Creating Publisher Threads

During some large-scale publisher workloads, an OutOfMemoryError was observed on the client driver machine when attempting to create very large numbers of publisher/consumer threads within a single host.

```
t_logs]$ :>> JVMDUMP010I Java dump written to /tmp/perf2client/javacore.20260114.122159.562717.0008.txt
:>> Exception in thread "Publisher6" java.lang.OutOfMemoryError: Failed to create a thread: retVal -1073741830, errno 11
:>>         at java.base/java.lang.Thread.startImpl(Native Method)
:>>         at java.base/java.lang.Thread.start0(Unknown Source)
:>>         at java.base/java.lang.Thread.start(Unknown Source)
:>>         at org.eclipse.paho.client.mqttv3.internal.ClientComms$DisconnectBG.start(ClientComms.java:766)
:>>         at org.eclipse.paho.client.mqttv3.internal.ClientComms.disconnect(ClientComms.java:525)
:>>         at org.eclipse.paho.client.mqttv3.MqttAsyncClient.disconnect(MqttAsyncClient.java:818)
:>>         at org.eclipse.paho.client.mqttv3.MqttAsyncClient.disconnect(MqttAsyncClient.java:779)
:>>         at org.eclipse.paho.client.mqttv3.MqttAsyncClient.disconnect(MqttAsyncClient.java:788)
:>>         at org.eclipse.paho.client.mqttv3.MqttClient.disconnect(MqttClient.java:347)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.MqttWorkerThread.destroyMQTTResources(MqttWorkerThread.java:126)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.Publisher.destroyMQTTResources(Publisher.java:72)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.MqttWorkerThread.run(MqttWorkerThread.java:219)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.Publisher.run(Publisher.java:78)
:>> JVMDUMP032I JVM requested Snap dump using '/tmp/perf2client/Snap.20260114.122200.562717.0012.trc' in response to an event
:>> JVMDUMP010I Snap dump written to {nothing to snap}
:>> JVMDUMP056I Processed dump event "systhrow", detail "java/lang/OutOfMemoryError" at 2026/01/14 12:22:03 (2.917 seconds).
:>> Exception in thread "Publisher17" java.lang.OutOfMemoryError: Failed to create a thread: retVal -1073741830, errno 11
:>>         at java.base/java.lang.Thread.startImpl(Native Method)
:>>         at java.base/java.lang.Thread.start0(Unknown Source)
:>>         at java.base/java.lang.Thread.start(Unknown Source)
:>>         at org.eclipse.paho.client.mqttv3.internal.ClientComms$DisconnectBG.start(ClientComms.java:766)
:>>         at org.eclipse.paho.client.mqttv3.internal.ClientComms.disconnect(ClientComms.java:525)
:>>         at org.eclipse.paho.client.mqttv3.MqttAsyncClient.disconnect(MqttAsyncClient.java:818)
:>>         at org.eclipse.paho.client.mqttv3.MqttAsyncClient.disconnect(MqttAsyncClient.java:779)
:>>         at org.eclipse.paho.client.mqttv3.MqttAsyncClient.disconnect(MqttAsyncClient.java:788)
:>>         at org.eclipse.paho.client.mqttv3.MqttClient.disconnect(MqttClient.java:347)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.MqttWorkerThread.destroyMQTTResources(MqttWorkerThread.java:126)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.Publisher.destroyMQTTResources(Publisher.java:72)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.MqttWorkerThread.run(MqttWorkerThread.java:219)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.Publisher.run(Publisher.java:78)
:>> JVMDUMP056I Processed dump event "systhrow", detail "java/lang/OutOfMemoryError" at 2026/01/14 12:22:03 (3.750 seconds).
:>> Exception in thread "Publisher5" java.lang.OutOfMemoryError: Failed to create a thread: retVal -1073741830, errno 11
:>>         at java.base/java.lang.Thread.startImpl(Native Method)
:>>         at java.base/java.lang.Thread.start0(Unknown Source)
:>>         at java.base/java.lang.Thread.start(Unknown Source)
:>>         at org.eclipse.paho.client.mqttv3.internal.ClientComms$DisconnectBG.start(ClientComms.java:766)
:>>         at org.eclipse.paho.client.mqttv3.internal.ClientComms.disconnect(ClientComms.java:525)
:>>         at org.eclipse.paho.client.mqttv3.MqttAsyncClient.disconnect(MqttAsyncClient.java:818)
:>>         at org.eclipse.paho.client.mqttv3.MqttAsyncClient.disconnect(MqttAsyncClient.java:779)
:>>         at org.eclipse.paho.client.mqttv3.MqttAsyncClient.disconnect(MqttAsyncClient.java:788)
:>>         at org.eclipse.paho.client.mqttv3.MqttClient.disconnect(MqttClient.java:347)
:>>         at com.ibm.uk.hursley.perfharness.mqtt.MqttWorkerThread.destroyMQTTResources(MqttWorkerThread.java:126)
```

This exception occurs when the operating system or JVM cannot allocate additional native threads. In these tests each MQTT publisher or subscriber client runs as a separate thread, and large-scale workloads involving tens of thousands of clients can exceed the thread or memory limits available to a single user.

The issue was addressed by distributing the client workload across multiple JVM processes and increasing the cgroup resource limits on the client driver's machine to allow a larger number of threads to be created.

systemctl set-property user-30000.slice TasksMax=1048576

6 Machine and Test Configurations.

The MQTT clients used to drive the tests in this report were located on dedicated Linux x86-

64 driver machines. The client workloads were executed using multiple JVM processes, with the number of JVMs varying depending on the scale of each test scenario.

The client driver machines and the queue manager server were connected using 100 Gb LAN links to ensure that network bandwidth did not become a limiting factor during the measurements.

Client workloads were executed using multiple JVM processes, with the number of JVMs varying depending on the scale of each test scenario.

Client Driver Machines & Server Machine

SR630 V2 (ThinkSystem) Type 7Z71:
CPU
2 × Intel® Xeon® Gold 6346 processors @ 3.10 GHz
16 cores per socket (32 physical cores, 64 logical CPUs)
Memory
256 GB RAM
Network
100 Gb Ethernet LAN
Operating System
Red Hat Enterprise Linux 9.7 (x86-64)
MQ Recovery Log and Queue Storage
2 × NVMe SSD devices (~2.9TB each) configured as a RAID0 array (5.8TB total).

The driver machines were responsible for generating the MQTT publisher and subscriber workloads used in the fan-in and fan-out scenarios described in this report.

Due to differences in hardware configurations and workload characteristics, the results presented in this report should not be compared directly with measurements from other environments.

7 Conclusion

This report presented performance measurements for MQTT workloads using IBM MQ Telemetry on x86-64 platforms. The tests evaluated both fan-in and fan-out messaging patterns across large numbers of MQTT clients, topics, and quality of service levels.

The results demonstrate that IBM MQ Telemetry can efficiently support very large numbers of concurrent MQTT clients while maintaining stable message throughput. Fan-in workloads showed the ability of the queue manager to process large volumes of publishing clients, while fan-out workloads demonstrated scalable message distribution across large topic sets and subscriber populations.

The measurements also illustrate the impact of MQTT quality of service levels on throughput, with higher delivery guarantees introducing processing overhead.

Overall, the results confirm that IBM MQ provides a scalable platform for MQTT-based messaging workloads and can support large-scale telemetry environments involving tens of thousands of clients and topics.

Appendix A: Software (all hosts)

Red Hat Enterprise Linux release 9.7 (Plow)

MQ-CPH MQI test driver (see Appendix B:)

IBM MQ V9.4.5

Eclipse Paho MQTT Client v3.1.1

A.1 Parameters Set for Measurements in This Report

The tuning detailed below was set specifically for the tests being run for this performance report but in general follow the best practices.

A.1.1 Operating System

A good starting point is to run the IBM supplied program mqconfig. The following Linux parameters were set for measurements in this report.

/etc/sysctl.conf

```
fs.file-max = 19557658
net.ipv4.ip_local_port_range = 1024 65535
net.core.rmem_max = 2147483647
net.core.wmem_max = 2147483647
net.ipv4.tcp_rmem = 4096 87380 2147483647
net.ipv4.tcp_wmem = 4096 65536 2147483647
vm.max_map_count = 1966080
kernel.pid_max = 655360
kernel.msgmnb = 131072
kernel.msgmax = 131072
kernel.msgmni = 32768
kernel.shmmni = 8192
kernel.shmall = 18446744073692774399
kernel.shmmax = 18446744073692774399
kernel.sched_latency_ns = 2000000
kernel.sched_min_granularity_ns = 1000000
kernel.sched_wakeup_granularity_ns = 400000
vm.overcommit_memory = 0
```

/etc/security/limits.d/mqm.conf

```
@mqm soft nofile 1048576
@mqm hard nofile 1048576
@mqm soft nproc  1048576
@mqm hard nproc  1048576
```

A.1.2 IBM MQ

The following parameters are added or modified in the qm.ini files for the tests run in section 0 of this report:

```
Channels:
   ChlauthEarlyAdopt=Yes
   ChlauthIgnoreUserCase=No
   MQIBindType=FASTPATH
   MaxActiveChannels=5000
   MaxChannels=5000
Log:
   LogBufferPages=4096
   LogFilePages=16384
   LogPrimaryFiles=16
   LogSecondaryFiles=2
   LogType=CIRCULAR
   LogWriteIntegrity=TripleWrite
SSL:
   AllowTLSV13=Yes
   MinimumRSAKeySize=2048
Service:
   EntryPoints=14
   Name=AuthorizationService
ServiceComponent:
   ComponentDataSize=0
   Module=amqzfu
   Name=MQSeries.UNIX.auth.service
   Service=AuthorizationService
TCP:
   ClntRcvBuffSize=0
   ClntSndBuffSize=0
   RcvBuffSize=0
   RcvRcvBuffSize=0
   RcvSndBuffSize=0
   SecureCommsOnly=NO
   SndBuffSize=0
   SvrRcvBuffSize=0
   SvrSndBuffSize=0
TuningParameters:
   DefaultPQBufferSize= 50000000
   DefaultQBufferSize= 50000000
```

Appendix B: Resources

MQ Performance GitHub Site
https://ibm-messaging.github.io/mqperf/

IBM MQ Test Harnesses Launch Page (includes links to containerized versions of MQ-CPH & JMSPerfHarness): Test Harnesses.

MQ-CPH (The IBM MQ Performance Harness for MQI in C)
https://github.com/ibm-messaging/mq-cph

PerfHarness (The IBM MQ Performance Harness for Java, including MQTTPerfHarness) https://github.com/ot4i/perf-harness

Tutorial: _mqttperfharness_tutorial.md

Appendix C: Glossary and Terminology

IBM MQ Telemetry (Preferred term throughout document)

- The complete telemetry feature set within IBM MQ that enables MQTT protocol support

MQXR (MQ Extended Reach)

- Technical name for the IBM MQ telemetry service
- Appears in system objects as SYSTEM.MQXR.SERVICE
- Use when referring to the specific service component

MQTT (Message Queuing Telemetry Transport)

- The lightweight messaging protocol (ISO standard)
- Use when discussing the protocol itself

XR Client

- A client application connecting via the MQTT protocol through the MQXR service
- Use consistently instead of "MQTT client" or "telemetry client"

MQI Client

- A client application using the native IBM MQ Interface
- Distinguished from XR clients