IBM WebSphere MQ Telemetry

Performance Evaluations

V1.0

August 2010

Oliver Fenton

WebSphere MQ Performance
IBM UK Laboratories
Hursley Park
Winchester
Hampshire
SO21 2JN

First Edition, August 2010.

This edition applies to *WebSphere MQ V7.0* (and to all subsequent releases and modifications until otherwise indicated in new editions).

Notices

Preface

Target audience

This SupportPac is designed for people who:

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

The reader should have a general awareness of the supported operating systems, WebSphere MQ and of WebSphere MQ Telemetry in order to make best use of this SupportPac.

Contents of this SupportPac

This SupportPac includes:

- Release highlights performance charts.
- Performance measurements with figures and tables to present the performance capabilities of telemetry and multi-connection scenarios.
- Interpretation of the results and implications on designing or sizing of MQTT applications using telemetry.
- Local queue manager, client channel, and distributed queuing configurations.

Feedback on this SupportPac

We welcome constructive 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?

Please direct any comments of this nature to WMQPG@uk.ibm.com.

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

Contents

Tables

Figures

Overview

"The MQTT protocol enables a publish/subscribe messaging model in an extremely lightweight way. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium." (http://mqtt.org/)

Telemetry is an extension of WebSphere MQ that manages messages using the MQTT V3 protocol. It has been designed with connection scalability in mind and can cope with thousands of simultaneously connected devices. The MQTT protocol defines three distinct qualities of service which can now benefit from WebSphere MQ's reliability and robustness.

The three Qualities of Service (from here on referred to as QoS) as defined by the MQTT V3 protocol are:

- QoS level 0 At most once delivery. This is the fastest method of messaging using the protocol, but is trading assured delivery for performance.
- QoS level 1 At least once delivery. Ensures a message is received by the receiver at least once (i.e. duplicates are allowed).
- QoS level 2 Exactly once delivery. This is the best level of service achievable using the MQTT protocol. Using this method with WebSphere MQ Telemetry will ensure a message will always reach its destination exactly once.

Covered in this Report

This report describes and evaluates six key scenarios, and covers two of telemetry's supported platforms.

- Platforms
  - Red Hat Enterprise Linux V5.5 64-bit
  - Windows Server 2008

- Telemetry round-trip messaging multi-publisher, single-subscriber.
  - at MQTT QoS 0.
  - at MQTT QoS 1.
  - at MQTT QoS 2.

- Telemetry round-trip messaging multi-publisher, multi-subscriber.
  - at MQTT QoS 0.
  - at MQTT QoS 1.
  - at MQTT QoS 2.

This document does not cover the following telemetry features:

- SSL support.
- Performance measurements for connecting and subscribing clients.
- Interfacing with WebSphere MQ message topics in the following ways:
  - MQTT clients publishing to WebSphere MQ message topics for subscription by a non-MQTT subscriber and
  - WebSphere MQ non-MQTT publishers publishing to MQTT subscribed clients.

Test Scenarios

The core of this document describes two configurations used to demonstrate the performance of the telemetry server:

- Multi-publisher, single-subscriber
- Multi-publisher, multi-subscriber

For each scenario, message rates are recorded with up to 100,000 MQTT clients and at the three varying qualities of service supported by the MQTT V3 protocol- 0, 1 & 2.

Unless otherwise specified, the standard message sized used for all the measurements in this report is 256 bytes. This size includes the message content and is sent in addition to MQTT protocol headers as defined in the specification.

The following pages describe the two configurations.

Multi-Publisher, Single-Subscriber Scenario

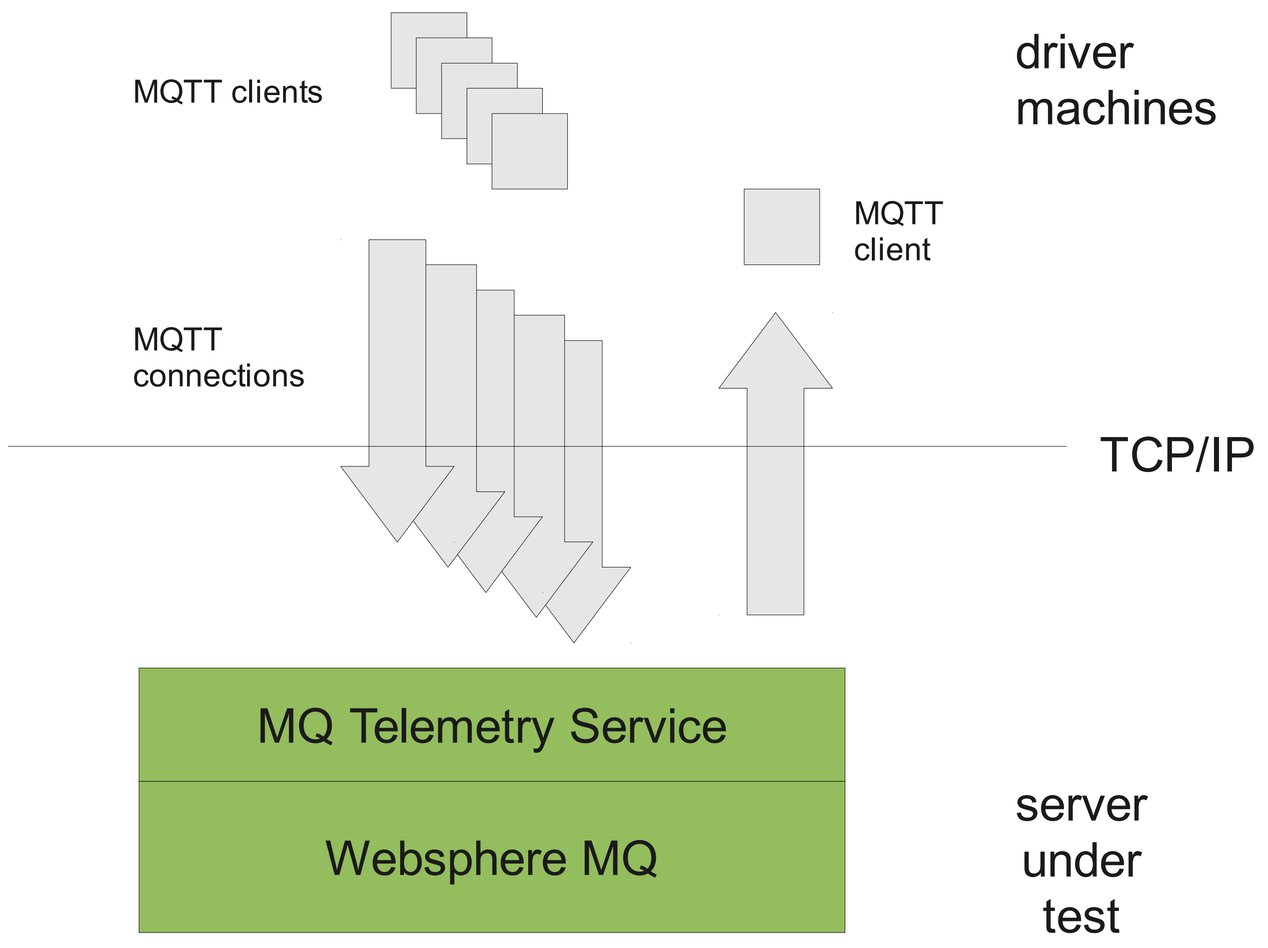

Figure 1: multi-publisher, single-subscriber setup

In this scenario, messages are published to the telemetry server 100 at a time from 100 randomly chosen MQTT clients. They publish to a single topic (called TOPIC1).

A single subscribing MQTT client retrieves all of the messages using its MQTT callback listener.

At start-up, 500 messages are initially published. Following the 500 message initialization, for every 100 messages the subscribing client receives, another 100 new messages are published by a random selection of publishers from the set of connected clients. This ensures the telemetry's transmit queue depth remains low (under 500) while ensuring the server always has work to do.

Multi-Publisher, Multi-Subscriber Scenario

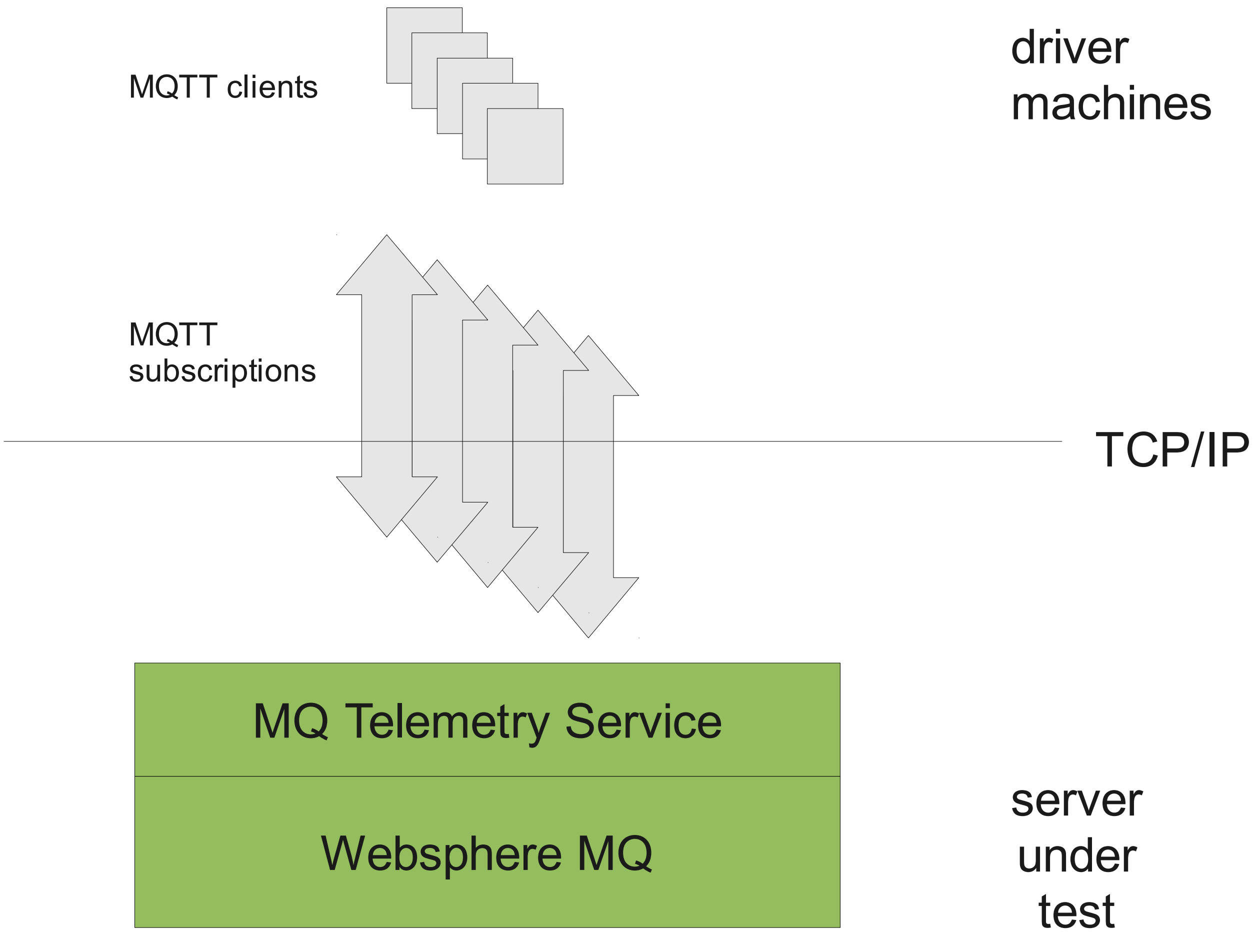

Figure 2: multi-publisher, multi-subscriber setup

Each client connects and subscribes to individual topics (one per client) before running the measurements. Each message is sent to a topic which has a single subscriber. The subscriber receives the message using its callback listener. As an example, with 50,000 clients, there are 50,000 topics, each with a single subscriber.

In this scenario, messages are published to the telemetry server 100 at a time from 100 randomly chosen MQTT clients. They publish to a topic (of the format rsmbX/mqY) which has a single subscriber.

Each message will only be received by a single subscriber using its MQTT callback listener.

At start-up, 500 messages are initially published. For every 100 messages the subscribing clients receive (summed together), another 100 new messages are published by a random selection of publishers from the set of connected clients. Using this technique, the telemetry's transmit queue depth remains low (under 500) while ensuring the server always has work to do.

Data Collection

Message rates are measured by the client application and are a measure of the time taken for a set of messages to be published to the telemetry server, processed by the telemetry server and finally pushed to the subscriber. The rate measured is the number of completed cycles per second starting from when the publisher first publishes the message and finishing when the subscriber asynchronously receives the message. In most cases, the measurements were taken after processing 500,000 messages.

The CPU percentages have been calculated from vmstat data. Readings of idle time were measured and subtracted from 100. This is equivalent to summing user time, system time and wait time.

Native memory data was collected using vmstat on Linux and a proprietary version on Windows which uses the same hooks as the Reliability and Performance Monitor.

Java heap usage was collected using the -verbose:gc JVM option. More information on this option can be found later in this report or in the IBM Java Diagnostics Guide.

Measurements

Linux Multi-Publisher, Single-Subscriber Scenario

Connections	QoS 0	QoS 1	QoS 2
1000	2780	2004	1645
10000	2569	1844	1527
20000	2430	1702	1369
30000	2287	1587	1328
40000	2262	1547	1246
50000	2199	1474	1163
60000	2126	1416	1151
70000	2008	1395	1093
80000	2004	1373	1047
90000	1900	1300	987
100000	1855	1204	938

QoS 0 CPU	QoS 1 CPU	QoS 2 CPU
68	66	66
71	69	70
75	74	75
78	75	76
80	77	79
81	78	80
82	80	82
84	81	84
86	82	86
88	84	88
89	86	89

Table 1: Linux multi-publisher, single-subscriber - message rates (msgs/sec), CPU (%)

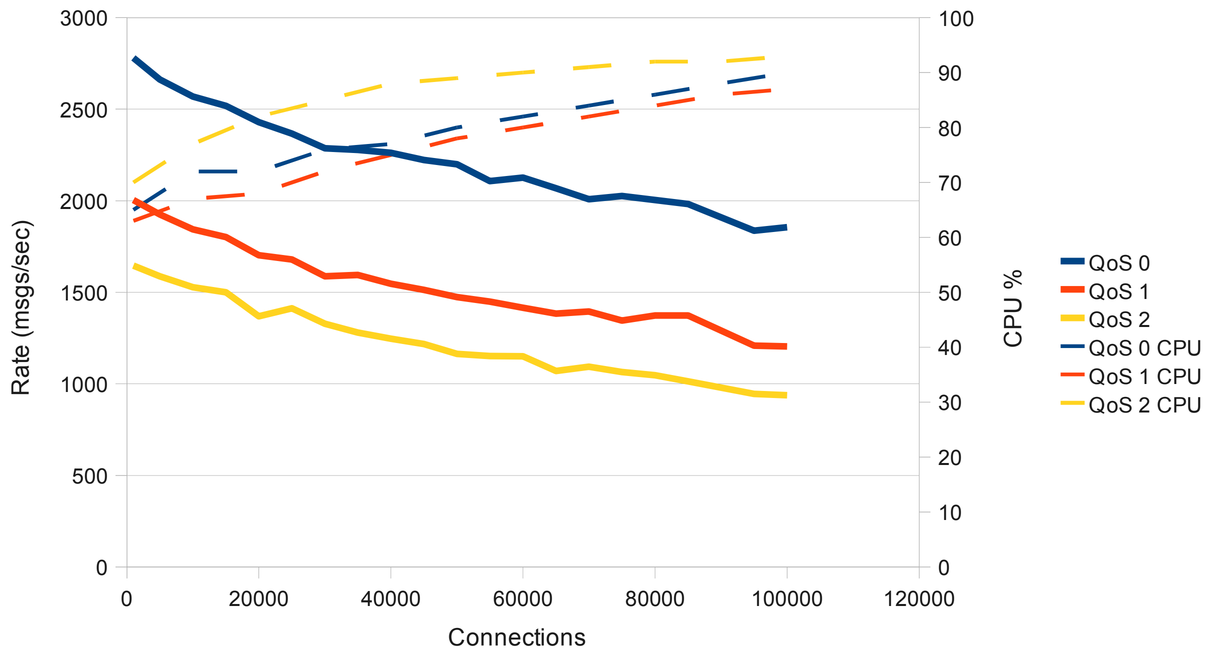

Figure 3: Linux multi-publisher, single-subscriber graph

With 1000 connections, telemetry can serve messages at nearly 3000 msgs/sec. As the number of connections increases, the number of messages through the server decreases steadily and the CPU usage approaches 100%. At 100,000 connections, the server can still process nearly 1000 msgs/sec at QoS 2.

Linux Multi-Publisher, Multi-Subscriber Scenario

Connections	QoS 0	QoS 1	QoS 2	QoS 0 CPU	QoS 1 CPU	QoS 2 CPU
1000	1877	1235	1088	71	67	70
10000	1435	967	949	85	81	81
20000	1361	1008	1117	88	83	86
30000	1313	1145	1146	89	84	85
40000	1256	1299	1076	90	80	87
50000	1270	1329	1023	89	81	89
60000	1291	1389	1065	90	82	90
70000	1232	1254	1025	90	85	91
80000	1216	1086	1018	90	84	91
90000	1237	1070	958	91	87	91
100000	1127	973	834	92	88	91

Table 2: Linux multi-publisher, multi-subscriber - message rates (msgs/sec), CPU (%)

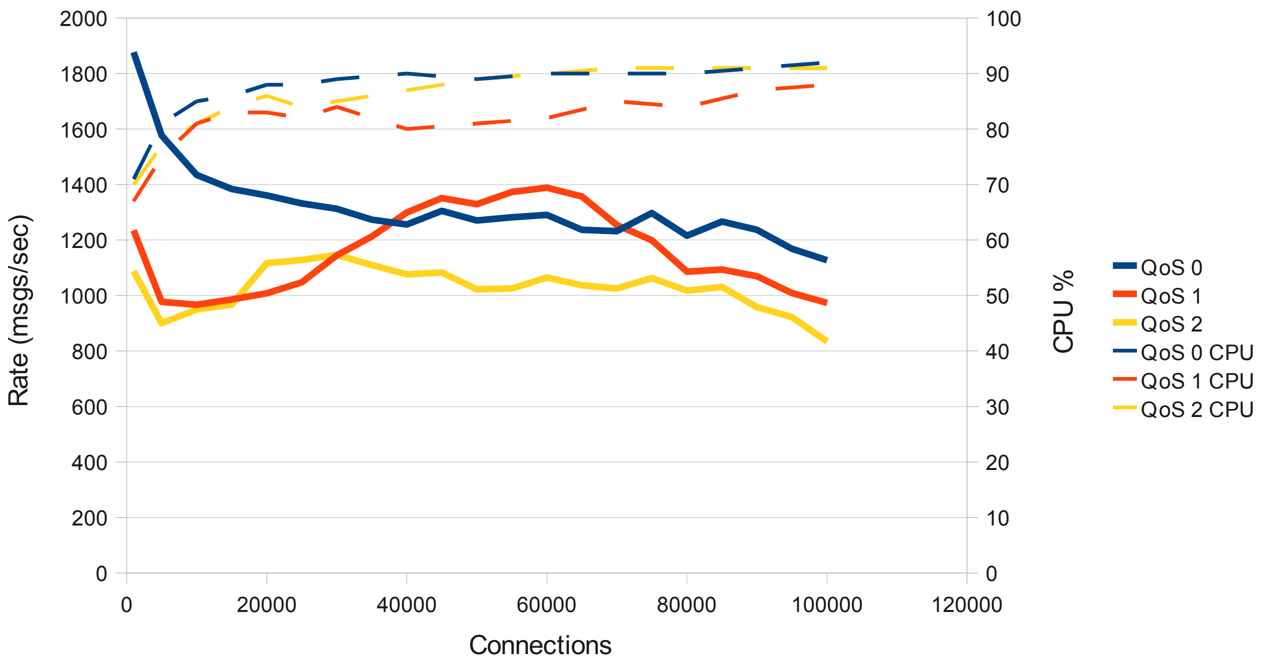

Figure 4: Linux multi-publisher, multi-subscriber graph

This scenario overlaps the multi-publisher, single-subscriber scenario, while demonstrating the performance change with up to 100,000 subscribers. Once again the peak rate was measured with the smallest number of connections, and the rate, even with the maximum number of subscribers is still near 1000msgs/sec. There is a dip in message rates between 0 and 50,000 connections before the rate starts to steadily decrease as in the multi-publisher, single-subscriber scenario.

Windows Multi-Publisher, Single-Subscriber Scenario

For the Windows runs, measurements were taken with up to 50,000 connections. Above this will require more than 2GB of memory which is the standard on a 32-bit Windows machine, running a 32-bit JVM.

Connections	QoS 0	QoS 1	QoS 2
1000	3200	2800	2000
10000	2820	2423	1601
20000	2353	1774	1267
30000	2177	1559	999
40000	1964	1332	826
50000	1793	1248	798

QoS 0 CPU	QoS 1 CPU	QoS 2 CPU
68	66	66
71	69	70
75	74	75
78	75	76
80	77	79
81	78	80

Table 3: Windows multi-publisher, single-subscriber - message rates (msgs/sec), CPU (%)

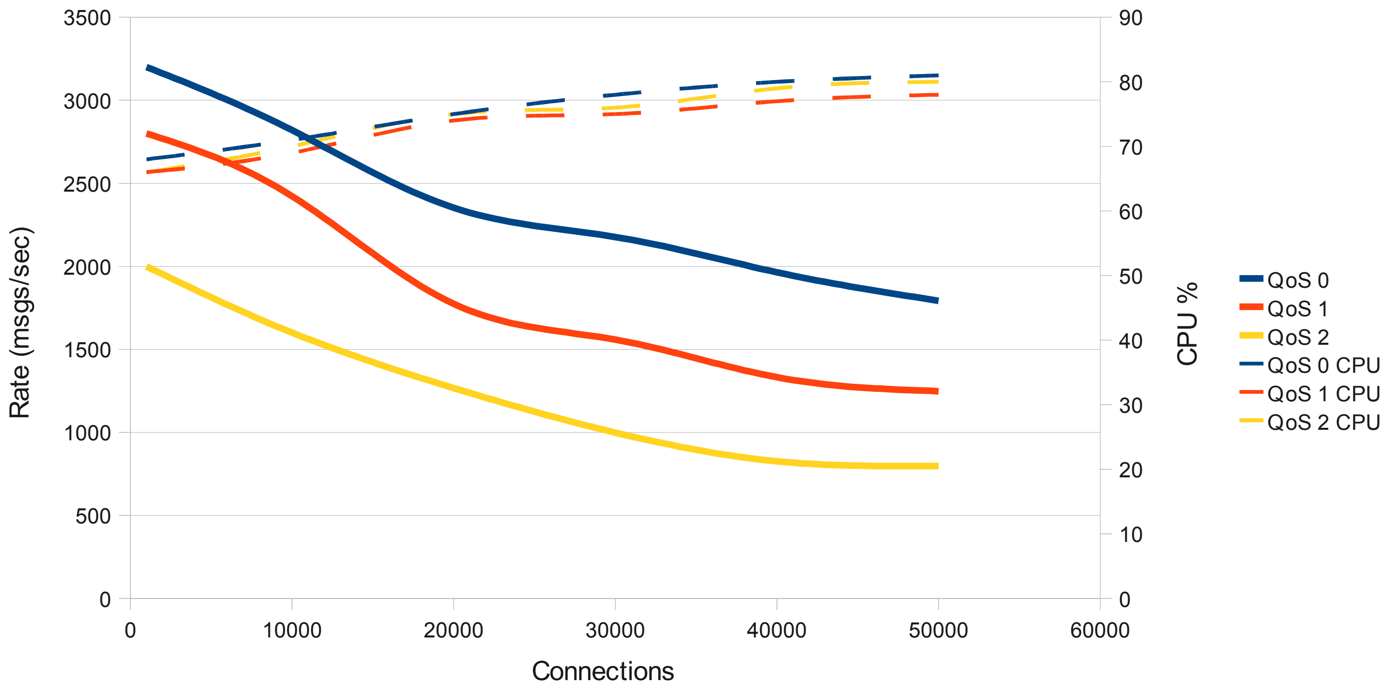

Figure 5: Windows multi-publisher, single-subscriber graph

The maximum message rate peaks with the lowest number of connections and decreases steadily as the number of connections is increased.

Windows Multi-Publisher, Multi-Subscriber Scenario

Connections	QoS 0	QoS 1	QoS 2		QoS 0 CPU	QoS 1 CPU	QoS 2 CPU
1000	3157	2035	1819		68	66	66
10000	2179	2288	1592		71	69	70
20000	1849	1856	1302		75	74	75
30000	1609	1608	1058		78	75	76
40000	1671	1209	920		80	77	79
50000	1825	1220	684		81	78	80

Table 4: Windows multi-publisher, multi-subscriber - message rates (msgs/sec), CPU (%)

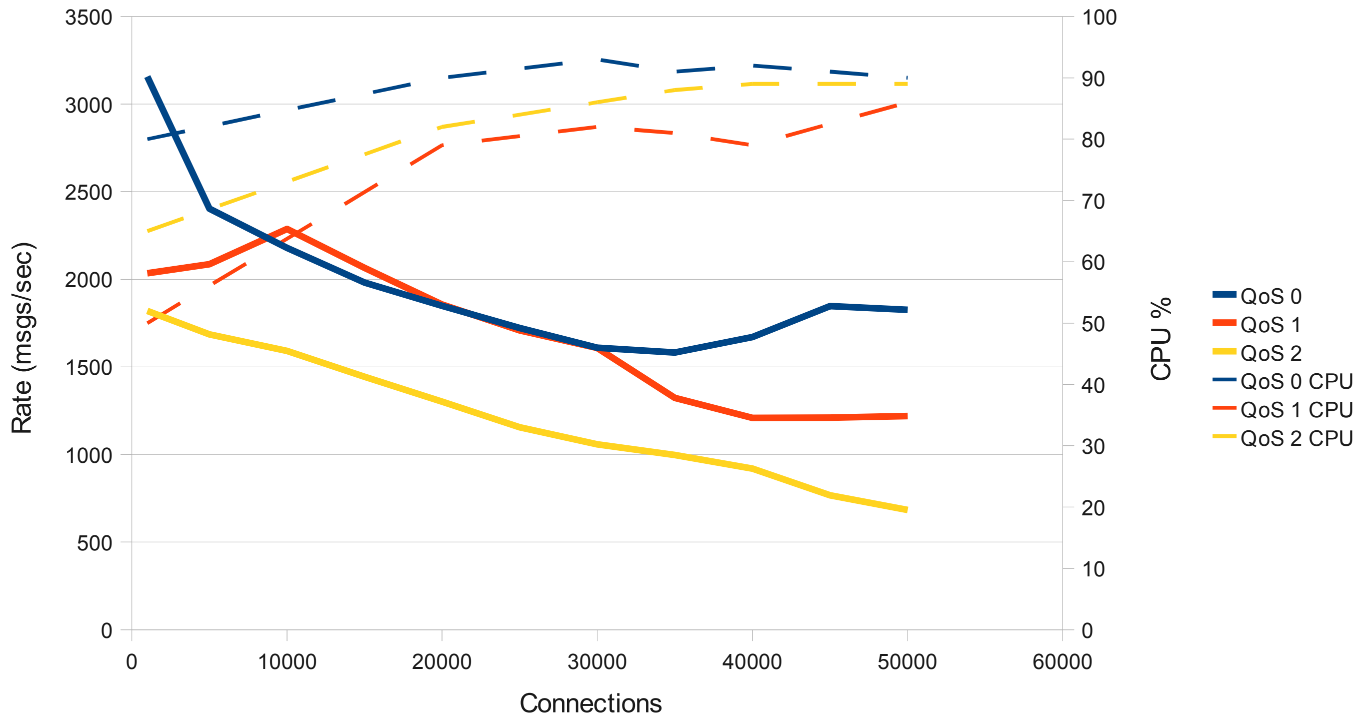

Figure 6: Windows multi-publisher, multi-subscriber graph

As in the Linux runs, there is a slight dip in message rates as the number of connections is increased. QoS 1 messaging rates fall in-line with QoS 0 between 10,000 and 30,000 connections,

Extending the Tests

Varying Message Sizes

The MQTT V3 protocol enables a publish / subscribe messaging model in an extremely lightweight way. It is useful for sending small messages from a wide variety of devices. The telemetry server has been designed for rapid delivery of small messages over this protocol. As such, this document is focussed on small messages. All measurements taken in other sections of this report use a message size of 256 bytes.

This section demonstrates the effects of alternate message lengths on the telemetry server. Messages are randomly generated character strings of the lengths 32 bytes, 256 bytes, 2048 bytes, 16,384 bytes and 65536 bytes.

These tests have been carried out on Linux at QoS 2 and for the multi-publish, multi-subscribe scenario only.

Connections	**32 bytes**	**256 bytes**	**2048 bytes**	**16384 bytes**	**65536 bytes**
1000	1089	1088	974	649	262
10000	953	949	826	542	226
20000	1206	1117	941	520	221
30000	1181	1146	1016	498	222
40000	1112	1076	974	516	220
50000	1064	1023	886	501	200

Table 5: effect of varying message size - message rates (msgs/sec)

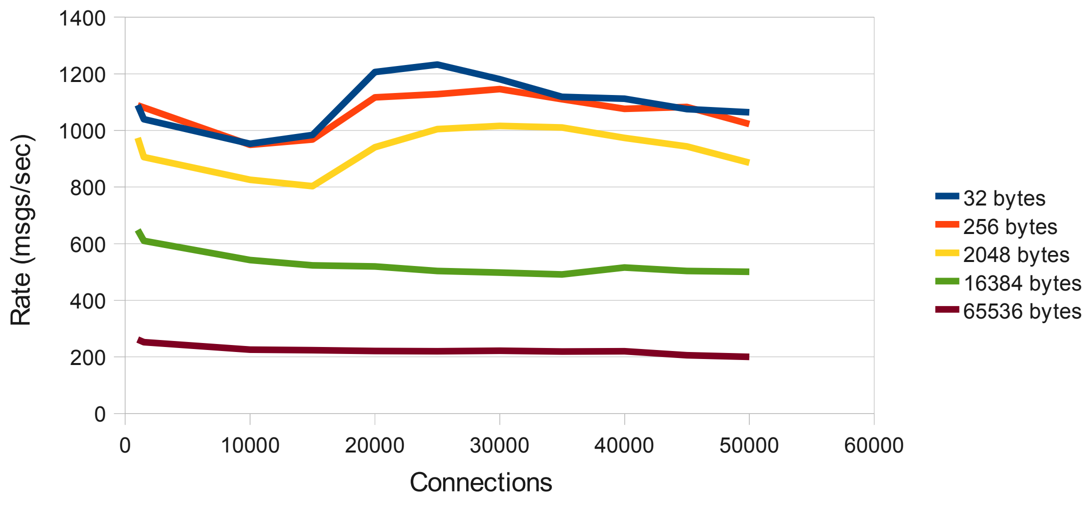

Figure 7: effect of varying message size

Capacity Limits

The performance measurements for maximum throughput rates, as recorded above are constrained primarily by CPU. When considering an telemetry solution, both Java and native memory requirements should be taken into account. This section lists the requirements as shown on the server machines during our measurements.

Native Memory

Recorded is the amount of native memory required per connection, subscription and during messaging. Only peak figures are recorded. Note that native memory requirements for N-connections will include telemetry requirements along with requirements for WebSphere MQ, the JVM, its heap, and any non-MQ background processes.

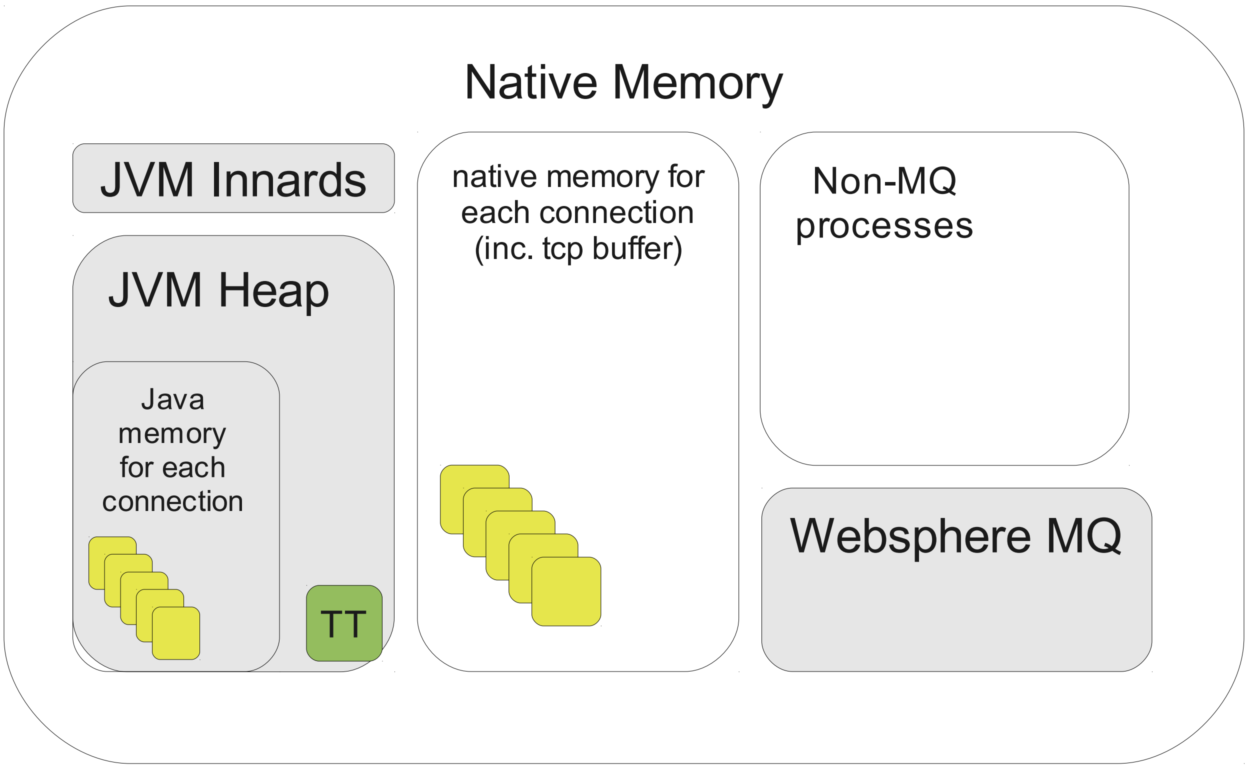
Native Memory
JVM Innards
JVM Heap
Java memory for each connection
TT
native memory for each connection (inc. tcp buffer)
Non-MQ processes
Websphere MQ

Figure 8: simplistic view of native memory usage in a telemetry system (Note: 'TT' represents the telemetry service)

Native memory has been measured using vmstat's free memory counter.

Java Heap Memory

The Java memory data shown is the amount required for objects in the Java heap. This can be set using the -Xmx JVM setting (see Tuning Considerations). When allocating Java heap memory it is advisable to allocate more than the required amount to prevent regular garbage collections and leading to poor performance. For example, a system requiring 600MB of Java heap should be pre-allocated around 1024MB (using -Xmx1024m).

Java heap memory was measured using the -verbose:gc JVM command line option.

Sockets & File Descriptors

Each incoming connection to telemetry requires an open socket. On Linux each socket uses a file descriptor so the telemetry process must be able to open sufficient file descriptors to manage this requirement. A description of how to increase the default file descriptor limit on Linux can be found in Tuning Considerations.

Memory Requirements Per Connection

	JVM Heap Memory per connection (KB)	Native memory per connection (KB)
Connected	6.00	18.00
Connected & Subscribed @ QoS 2	6.86	25.00

Table 6: Linux memory usage per connection

For every QoS 2 subscriber, an additional 6.86KB should be added to the Java heap requirements and 25.00KB to the native memory requirements.

	JVM Heap Memory per connection (KB)	Native memory per connection (KB)
Connected	4.00	12.00
Connected & Subscribed @ QoS 2	4.60	15.00

Table 7: Windows memory usage per connection

The Windows measurements were taken using a 32-bit JVM. This explains why the storage numbers for the JVM are different on Windows verses Linux.

Memory Requirements for N Connections

This table shows the peak memory usage during a multi-publisher, multi-subscriber test after a run at QoS 2 using 256-byte messages. The data shown with 0 connections show the base-memory usage before any tests are run.

Connections	Peak JVM Heap Memory (MB)	Peak Native Memory Used (MB)	Open File Descriptors
0	5	746	99
1000	13	910	1099
10000	74	1084	10099
20000	142	1357	20099
30000	210	1613	30099
40000	293	1769	40099
50000	366	1999	50099
60000	421	2264	60099
70000	520	2468	70099
80000	552	2580	80099
90000	627	2977	90099
100000	690	2977	100099

Table 8: Linux maximum memory requirements for N connections

Connections	**Peak JVM Heap Memory (MB)**	**Peak Native Memory Used (MB)**
0	4	1320
1000	9	1340
10000	50	1444
20000	95	1521
30000	140	1669
40000	187	1768
50000	234	1825

Table 9: Windows maximum memory requirements for N connections

Tuning Considerations

To optimise message throughput using Websphere MQ Telemetry there are several WebSphere MQ, Java and operating system tuning parameters that can be used to improve performance. Each of the following settings should be considered and used appropriately for the desired setup.

Performance reports with tuning information for WebSphere MQ v7.0 on all supported operating systems can be found via. the IBM SupportPac web page.

WebSphere MQ Considerations

MAXHANDS is a queue manager parameter that sets the maximum number of open handles that any one connection can have at the same time. To enable connections of thousands of telemetry clients, this number should be set appropriately to cope. Default is 256.

e.g. ALTER QMGR MAXHANDS(999999999)

MQTT messages using the telemetry server are placed on the WebSphere MQ queue **'SYSTEM.MQTT.TRANSMIT.QUEUE'**. Messages may be placed on the queue for each subscriber to receive a message so, for ten subscribers receiving a single message, up to ten 'transmit' messages will be placed on the transmit queue. Ensure the transmit queue size is sufficiently large to cope with the expected message rates. See the **MAXDEPTH** parameter on the queue description.

A message queue uses an in-memory buffer to store its queued messages. When the buffer becomes full messages start being written to disk. To avoid writing to disk, consider using the **DefaultPQBufferSize** (persistent messages) and **DefaultPBufferSize** (non-persistent messages) parameters to alter the buffer sizes and improve performance. On 32-bit queue managers (like Windows), the defaults are 64kb / 128kb for persistent / non-persistent messages. On 64-bit queue managers, the default buffer sizes are 128kb and 256kb respectively.

These parameters must be set before starting the queue manager and will apply to all queues defined in that invocation of the queue manager.

On Linux, set the TuningParameters stanza in the queue manager's qm.ini file. e.g.

```
TuningParameters:
   DefaultQBufferSize=1048576
   DefaultPQBufferSize=1048576
```

On Windows, the TuningParameters stanza must be set in the registry. Use either MQ Explorer or the amqmdain command to set these buffers. See the WebSphere MQ Infocenter for more information.

The Telemetry Server and Java Considerations

The telemetry server is a Java application that extends WebSphere MQ. A Java Runtime Environment (JRE) is supplied with the Service and may need to be optimized for a large number of connections. To manage large numbers of connections, it is advisable to increase the initial Java heap size beyond the default settings. **Xmx** sets the maximum allowable heap size for the JRE. **Xms** sets the initial heap size for the JRE. For 50,000 connections, this document advises setting Xmx and Xms to 1024MB.

Set Xmx and Xms using the java.properties file in the mqxr configuration directory e.g.

```
# Heap sizing options - uncomment the following lines to set the heap to 1G
-Xmx1024m
-Xms1024m
```

Some notes on Xmx and Xms:

- If Xms is set lower than Xmx, the Java Heap (from the perspective of the JRE) can dynamically change in size between Xms and Xmx. This optimizes memory use and will optimise performance under constant load but will result in a degradation of performance on occasions when the heap size changes. This is due to extended garbage collection times on these occasions.
- The size of Java heap must all be backed by physical memory. Each time a garbage collection occurs, the whole heap must be loaded into physical memory. If there is not enough physical memory available, swapping of the heap will cause a large performance degradation.

Operating System Considerations

Linux

Each open socket requires one file descriptor on Linux so the telemetry service requires a file descriptor for each client connection. In addition, the service requires around 100 additional file descriptors for loading jars and configuration files. Ensure the file descriptor limit for a process exceeds the required number.

To set the file descriptor limit, edit /etc/security/limits.conf and append the following:

```
@mqm soft nofile 120000
@mqm hard nofile 120000
```

Windows

To exceed 2000 connections on Windows, apply IBM Java APAR IZ78262.

Summary

The total message throughput on Linux with up to 100,000 client connections subscribed is above 800 message per second using persistent, Quality of Service 2 messages using the MQTT protocol.

On Windows, the total message throughput with up to 50,000 client connections subscribed is above 650 messages pre second at Quality of Service 2 and can sustain higher rates at the lower QoS levels. Using messages around the 256 byte size, this is near the peak memory limit in a 2GB restrained system.

With small messages (less than 1024bytes), message rates are similar. Running with larger messages will see lower messaging rates.

To get the best performance and the largest number of connected devices into the MQ Telemetry service, we recommend tuning the following to your needs:

- Java heap size
- Number of open sockets
- MAXHANDS queue manager parameter
- DefaultQBufferSize for QoS 0
- DefaultPQBufferSize for QoS 1 & QoS 2

Appendix

Simulating Thousands of Clients Using Telemetry Daemon for Devices

To simulate thousands of clients connecting to the telemetry server endpoint, the supplied Telemetry Device Daemons were used. This section explains the setup used to connect thousands of devices to the endpoint.

Multi-Publisher, Single-Subscriber

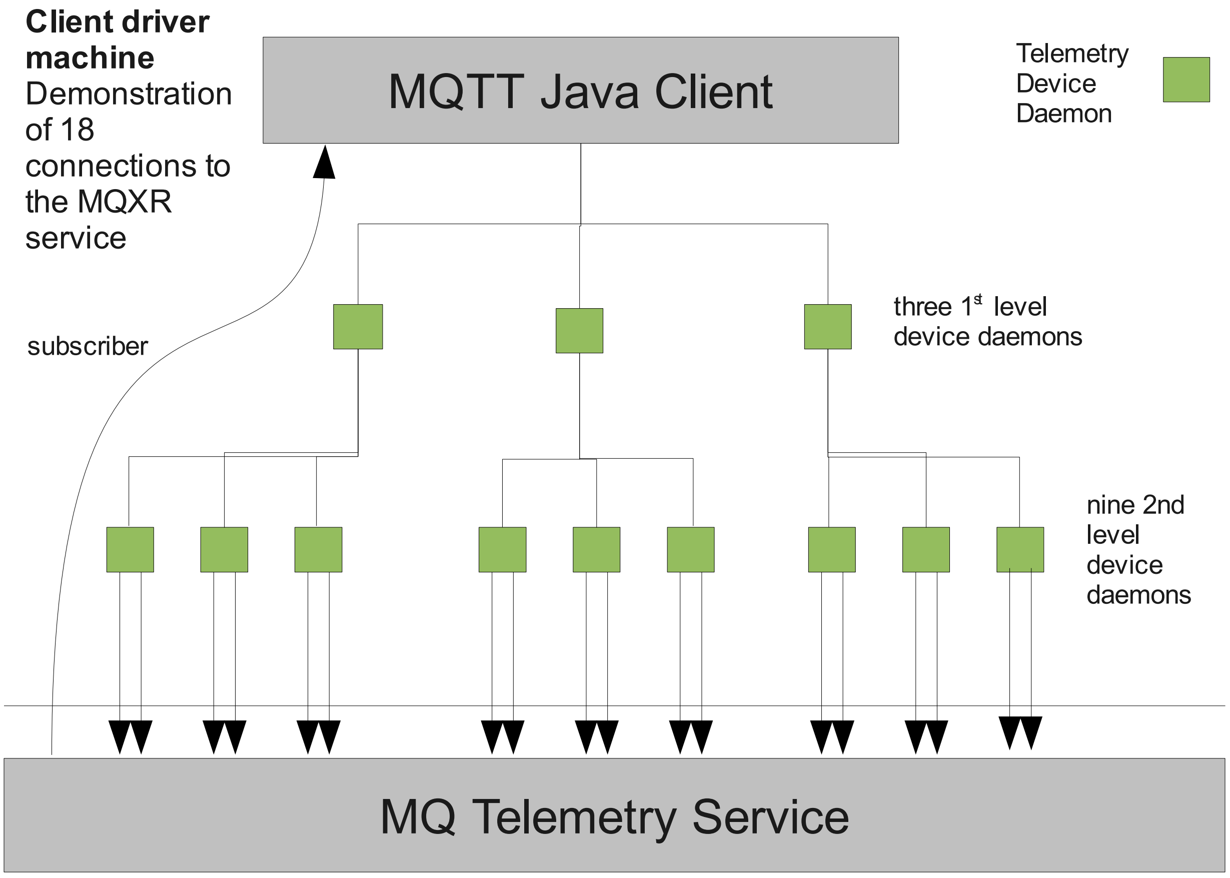
Client driver machine
Demonstration of 18 connections to the MQXR service
MQTT Java Client
Telemetry Device Daemon
subscriber
three 1st level device daemons
nine 2nd level device daemons
MQ Telemetry Service

Figure 9: client setup for multi-publisher, single-subscriber scenario

To drive multiple connections into the telemetry Server, the client setup uses a hierarchy of Telemetry Device Daemons (supplied with the product) to span connections from a single Java client into the telemetry Server.

In this scenario, the environment is setup as described:

- N 2nd-level Telemetry Device Daemons connect to the telemetry server using (for measurements in this report) 10 independent connections. N * 10 is the total number of connections to the server under test. For simplification, figure 1 shows nine 2nd-level device daemons (N) & two connections from each daemon (instead of 10). For this report, measurements show up to 100,000 total connections to the server in varying configurations.

- X 1st-level Telemetry Device Daemons connect to the N 2nd-level device daemons. Each device daemon is single-threaded so to prevent a bottleneck, the scenario uses multiple device daemons.

For this report, each 1st level Telemetry Device Daemon bridged to 10 2nd level Telemetry Device Daemons which, in turn, created and managed 10 connections to the telemetry server endpoint.

1st Level TDDs	**2nd Level TDDs**	**MQ Telemetry Connections (per 2nd Level TDD)**	**Connections under Test**
10	100	10	1000
100	1000	10	10000
200	2000	10	20000
300	3000	10	30000
400	4000	10	40000
500	5000	10	50000
600	6000	10	60000
700	7000	10	70000
800	8000	10	80000
900	9000	10	90000
1000	10000	10	100000

Table 10: hierarchy of Telemetry Device Daemons (TDDs) used for large numbers of server connections

Above 50,000 connections, 2nd-Level TDDs numbered 5000 and above were started on the second client machine (see Hardware Used for this Report) and the configuration of 1st-Level TDDs (all still on the first client machine) altered to manage this.

Multi-Publisher, Multi-Subscriber

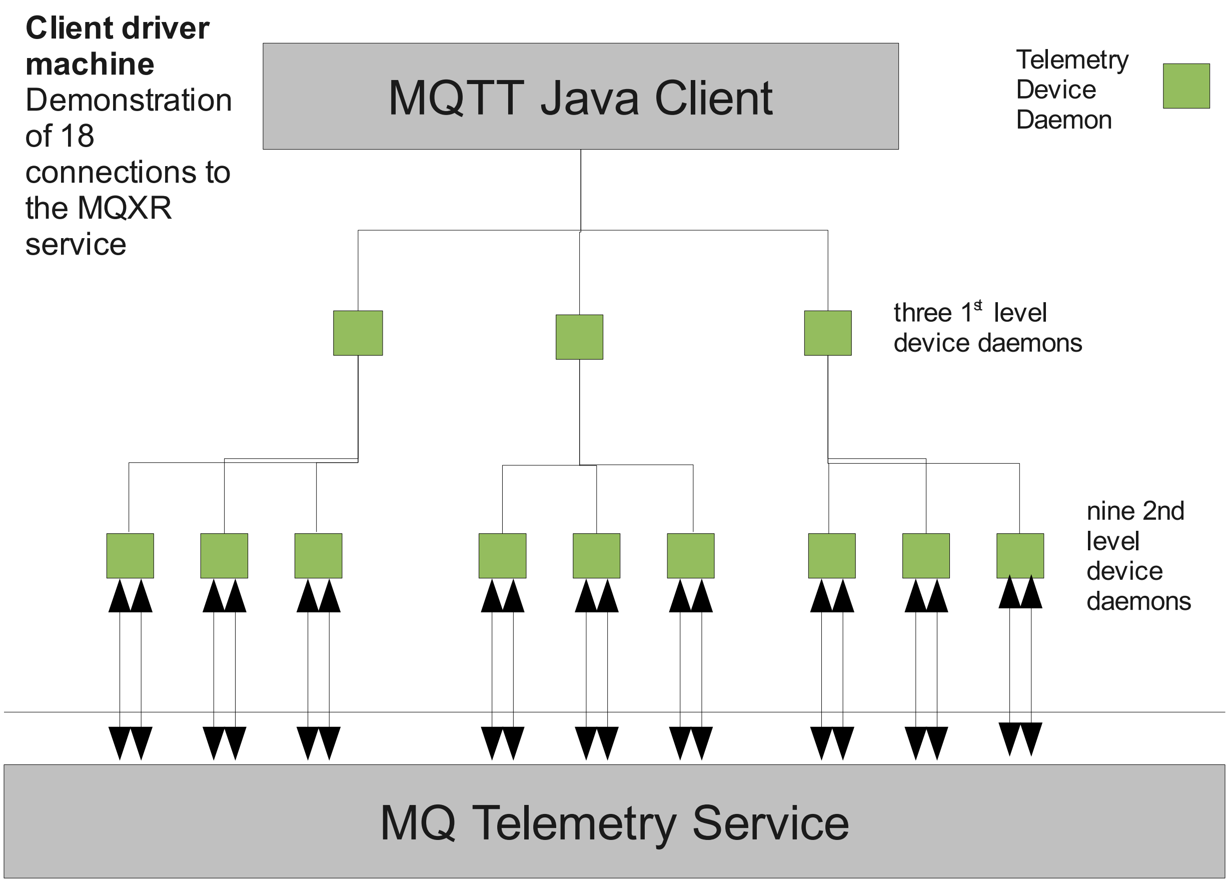
Client driver machine
Demonstration of 18 connections to the MQXR service
MQTT Java Client
Telemetry Device Daemon
three 1st level device daemons
nine 2nd level device daemons
MQ Telemetry Service

Figure 10: client setup for multi-publisher, multi-subscriber scenario

For the multi-publisher, multi-subscriber scenario, the Java Client publishes several messages to each of the '1st level device daemons'. Each of these forwards the message to each of its bridging (2nd-level) device daemons. Each of the '2nd level device daemons' has several connections established with the telemetry server which it forwards the published message to as if it were an independent MQTT publisher.

The setup for this scenario is very similar to the multi-publisher, single-subscriber scenario. The changes made are:

- removal of the separate single subscriber
- change in 1st and 2nd-Level TDD configuration to subscribe as well as connect to the MQ Telemetry Service. Messages are now filtered down the hierarchy as they are published, and pushed back up the hierarchy as the subscriber callbacks get invoked.
- Modification to the client application to receive messages from the 1st-Level TDD callbacks instead of the single separate subscriber.

Client Tuning

A standard network card is restricted to 65535 open outbound sockets on both Linux and Windows. When a request for a new socket arrives, the port number is chosen from the range of unused ephemeral ports. By default this range only allows several thousand of these ports to be used.

To extend this range on Linux, in /etc/sysctl.conf set

net.ipv4.ip_local_port_range = 8192 65535

To extend this range on Windows, update the registry key MaxUserPort:

```
Key: HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Tcpip\Parameters\MaxUserPort
Value: 65535
```

Hardware Used for this Report

Linux server machine (Red Hat 5.5 kernel 2.6.18-194.3.1.el5 x86_64)

Processor: 3.66GHz Intel Xeon

Architecture: 2 dual core CPU (4 way SMP)

Hyper-threading disabled

Memory (RAM): 4Gb

Disk: 2 Internal 16bit SCSI (90Gb each, 1 O/S, swap)

2 SAN disks on DS6000 (5Gb each, 1 queue, 1 log)

Network: 1Gbit Ethernet Adapter

Windows server machine (Windows Server 2008 64-bit, SP1)

Processor: 3.33GHz Intel Xeon

Architecture: 2 dual core CPU (4 way SMP)

Hyper-threading disabled

Memory (RAM): 4Gb

Disk: 2 Internal 16bit SCSI (90Gb each, 1 O/S, swap)

Network: 1Gbit Ethernet Adapter

Linux driver machine (Red Hat 5.5 kernel 2.6.18-164.9.1.el5 x86_64)

Processor: 3.66GHz Intel Xeon

Architecture: 2 dual core CPU (4 way SMP)

Hyper-threading disabled

Memory (RAM): 4Gb

Network: 1Gbit Ethernet Adapter

Linux driver machine (Red Hat 5.5 kernel 2.6.18-194.8.1.el5 x86_64)

Processor: 3.33GHz Intel Xeon

Architecture: 2 dual core CPU (4 way SMP)

Hyper-threading disabled

Memory (RAM): 4Gb

Network: 1Gbit Ethernet Adapter

SAN disk subsystem

MQ Log and Queues on SAN disks on DS6000. 5GB allocated for both Log and Queues on Linux.

The MQ SAN consists of a pair of 2026 model 432 (McDATA ES-4700) switches running at 4Gb/s with 32 ports each. They are connected together via two inter-switch links to form a single SAN fabric.

The MQ hosts attach via this SAN to a DS6800 disk array (1750 model 511) with one expansion drawer.

Each drawer (controller + expansion) contains 16 x 73Gb 15K fibre channel disk drives, so there are a total of 32 physical drives.

The 32 drives are configured as four RAID-5 arrays, each of which is 6+Parity+Spare (the number of spares is defined by the configuration of the DS6800).

The controller has an effective cache size of 2.6Gb plus 0.3Gb of NVS