Benchmark of

MQTT servers

ActiveMQ 5.10.0

Apollo 1.7

JoramMQ 1.1.3 (based on Joram 5.9.1)

Mosquitto 1.3.5

RabbitMQ 3.4.2

version 1.1

January 2015

mqtt.jorammq.com

www.scalagent.com

1 Introduction

Message Queue Telemetry Transport (MQTT) is an open Machine-to-Machine (M2M) protocol, that has been invented in 1999, and that has become an OASIS standard[1]. MQTT is a lightweight event and message oriented protocol allowing devices to asynchronously and efficiently communicate across constrained networks to remote systems. MQTT is now becoming one of the standard protocols for the Internet of Things (IoT).

The MQTT protocol relies on a messaging server following the hub and spoke model of Message Oriented Middleware (MOM). As shown in Figure 1, every MQTT client, data processing application or device, producer or consumer, needs to connect to a central server before communicating with other MQTT clients. The server accepts published messages and delivers them to the interested consumers according to a Publish/Subscribe interaction pattern.

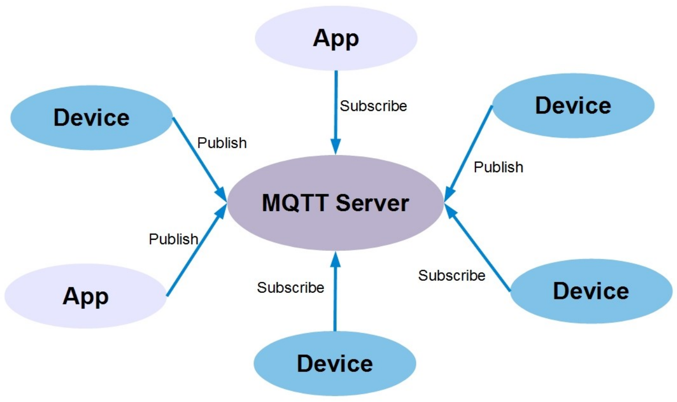

Figure 1: hub and spoke model

The goal of the benchmark is to evaluate the scalability of an MQTT server with the number of clients. The term « scalability » refers to the ability of the server to handle a growing number of clients. This benchmark does not add more nodes or resources (CPU, memory) to make the server scale, either horizontally or vertically. The server is launched on a single node with fixed resources.

The test bench allows to check that in a given context (QoS level, message throughput per client, message payload size), the server scales with the number of clients.

We, at ScalAgent, aimed at designing an objective benchmark of the different MQTT servers. The evaluation of the servers is made according to a simple and realistic test scenario, and the comparison is presented in terms of basic measurements (CPU, latency, message rates) with a detailed description of the conditions in which they have been obtained and how the different servers have been configured.

1 https://www.oasis-open.org/news/pr/foundational-iot-messaging-protocol-mqtt-becomes-international-oasis-standard

The following servers have been evaluated (alphabetical order):

- ActiveMQ 5.10.0
- Apollo 1.7
- JoramMQ 1.1.3 (based on Joram 5.9.1)
- Mosquitto 1.3.5
- RabbitMQ 3.4.2

The configuration of every server is the default one with few modifications detailed in section 5.

2 Test scenario

The test scenario is called “multi-publisher”. It simulates a large number of devices, for example smart meters, publishing telemetry data connected with a command centre. The devices are the publishers and the command centre is the subscriber.

Devices are structured as a 3-level tree. The top level represents the system, for example an electrical distribution circuit with a single power substation. The level below is called “subsystem”. In the metering use case, a subsystem would be the access point in the neighbourhood that enables the meters to reach Internet and publish data to the electricity company. An example of access point is an antenna mounted on a utility pole. The bottom level is the device, e.g. a smart meter.

The 3-level tree is mapped to an MQTT topic hierarchy. A fourth topic level is added below the devices to represent the telemetry parameters, for example the power consumption (kWh).

An MQTT topic is just a hierarchical name, for example:

“System/Subsystem/Device/Parameter”

Figure 2 represents the different levels of the topic hierarchy applied to the metering use case.

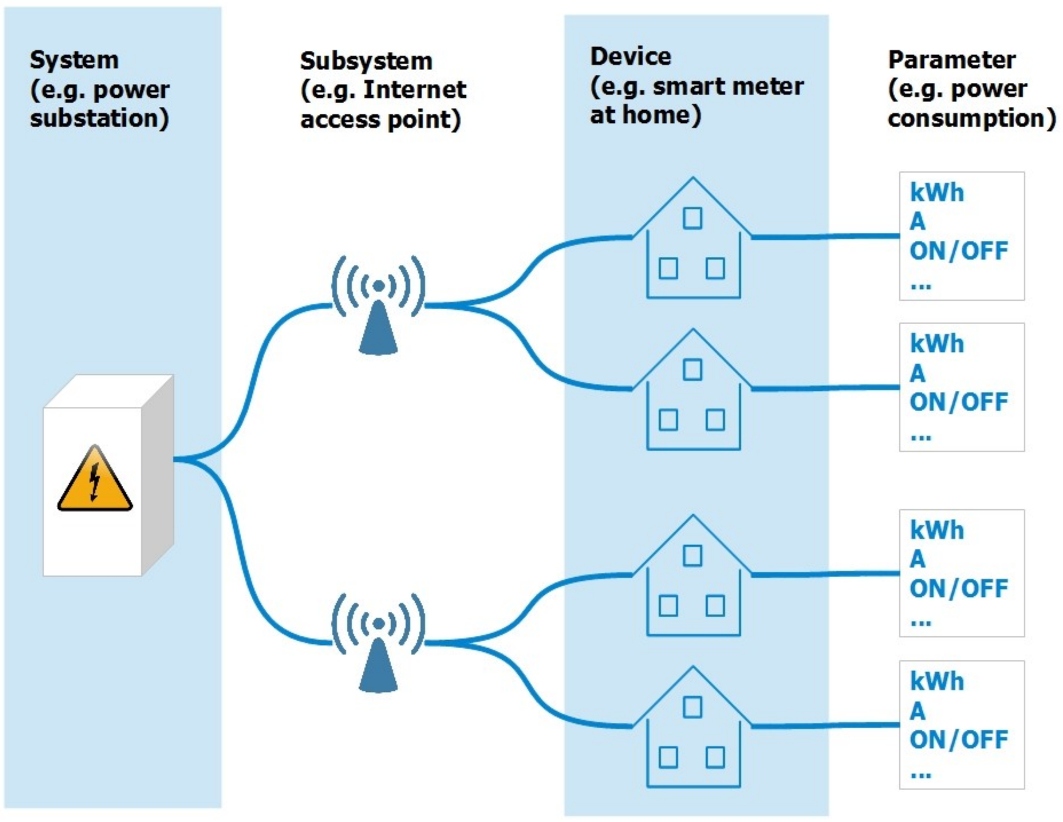

Figure 2: MQTT topic hierarchy for the metering use case

The “multi-publisher” scenario described above is executed in the test environment represented in Figure 3. The MQTT server is launched as a single instance on a single machine[2].

Each device is represented by an MQTT client, called “publisher” that creates an MQTT connection[3] and publishes messages to its topics, i.e. the topics representing the telemetry parameters of the device.

The topics are organized in partitions in order to have independent test instances producing and consuming messages through a dedicated topic hierarchy. In this way the load can be increased by simply adding new test instances, executed in several processes, and on several machines. Each test instance should not be directly affected by the other test instances. They should all run independently.

The command centre is represented by several MQTT clients, called “subscribers”. One subscriber is created for every topic partition. A connection is opened and a subscription is made to the root topic of the partition, i.e. the topic representing the “System” level as explained above.

Test instances are launched on two machines, each instance handling 1000 publishers[4] and 1 subscriber[5]. The ratio 1/1000 happens to be the most convenient to evaluate the performance of the various servers. The total number of parameter topics per partition is 10.000.

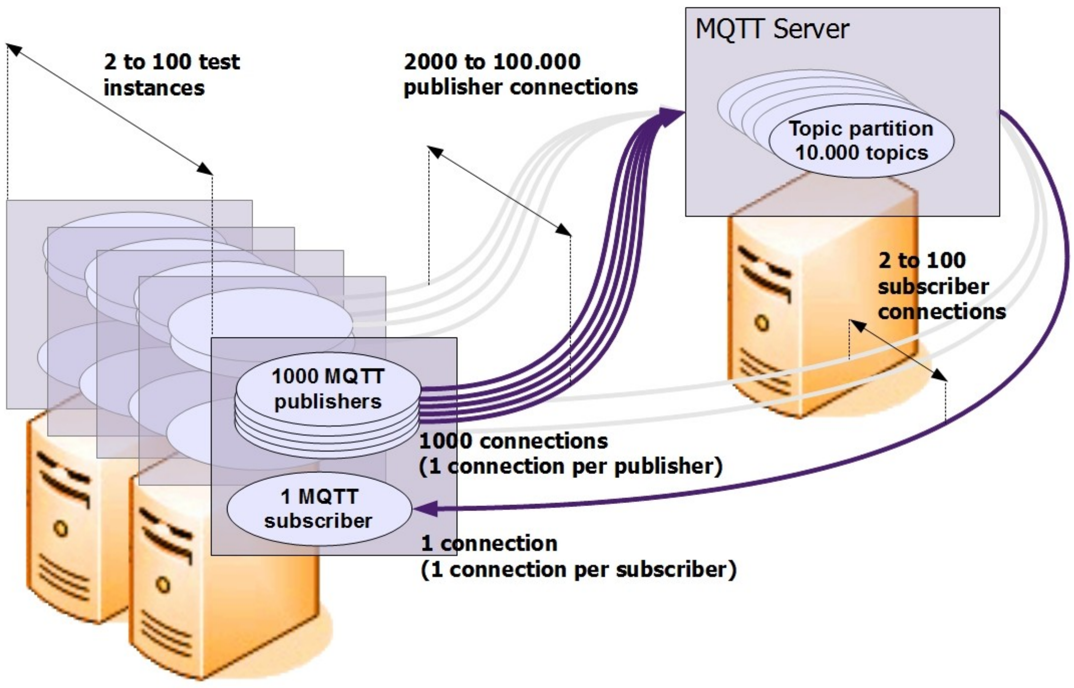
MQTT Server
2 to 100 test instances
2000 to 100.000 publisher connections
Topic partition 10.000 topics
2 to 100 subscriber connections
1000 MQTT publishers
1000 connections (1 connection per publisher)
1 MQTT subscriber
1 connection (1 connection per subscriber)

Figure 3: test environment

2 For scalability tests launching the MQTT server on several nodes see the document at: http://www.scalagent.com/IMG/pdf/JoramMQ_MQTT_white_paper-v1-2.pdf

3 One MQTT connection is created for every publisher, i.e. connections are not shared between publishers.

4 The test environment does not allocate one thread per publisher in order to minimize the CPU load. Samples of messages (100) are asynchronously published by a single thread through a given number of publishers (1000) in a round-robin way at a fixed rate (one sample every 100 ms).

5 Subscribers are activated by threads managed by the MQTT client library.

The goal of the benchmark is to evaluate the impact of the number of publishers on every MQTT server, in terms of the delivered throughput (message rate on the subscriber side), the CPU usage of the server, and the time required to transmit a message from a publisher to a subscriber, i.e. the message transmission latency. There should be no limitation caused by the clients (affecting each other) or by the network.

One topic partition is made of:

- 1 root topic “System”
- 10 topics “Subsystem”
- 100 topics “Device” per subsystem
- 10 topics “Parameter” per device

In the metering use case, a topic partition represents an electrical distribution circuit with 1000 meters, each meter publishing 10 telemetry parameters.

The scalability of the server with the number of clients is evaluated by incrementally increasing the number of test instances. The scalability test starts with a minimum of 2000 publishers and 2 subscribers, and tries to reach a maximum of 100.000 publishers and 100 subscribers.

The publishers send messages at a steady rate. The tests do not try to reach the maximum message throughput. The goal is to show how the server scales with the number of publishers, each publishing at a fixed rate.

The message rate is set to 1 message per second per publisher. In this way the number of publishers (and connections) also gives the global message throughput, for example 100.000 publishers produce 100.000 messages per second through 100.000 MQTT connections.

The message payload size is fixed and set to 64 bytes. This payload size is small but makes sense in the metering use case as every message transmits a single telemetry parameter. Such a small size has been chosen in order not to overwhelm the server memory, limited to 3 GB, either directly (memory allocation for every message) or indirectly (TCP buffer size). However, a future version of this benchmark will test several payload sizes.

QoS is tested by assigning the same QoS level to publishers and subscribers.

MQTT provides three QoS levels:

- QoS level 0, also called “at most once”, which means that messages may be delivered once or not at all.
- QoS level 1, also called “at least once”, ensures that the messages arrive at the receiver at least once.
- QoS level 2, also called “exactly once” because messages are neither lost or duplicated.

Only QoS levels 0 and 1 have been tested. QoS 2 will be tested in a next version of this benchmark.

At QoS 0, MQTT publishers send messages and forget about them. No acknowledgement is returned by the server and the message is not persistently stored.

At QoS 1, MQTT publishers expect an acknowledgement after having sent a message. Moreover the server should make the message persistent before delivering it and before the acknowledgement is returned to the publisher. The message has to be really written to disk and not only flushed to the OS cache[6].

QoS level 1 is only tested with durable subscriptions, i.e. subscriptions that enqueue messages in a persistent way. MQTT uses a flag called “Clean Session” to specify a durable subscription. If Clean Session is false, then the subscription is durable.

The tests check that there is no message loss, even at QoS level 0.

The 3 machines used for the benchmark have the same configuration listed in the table hereafter. The CPU and memory resources are limited to only two cores and 4 GB RAM. The network bandwidth is ensured by a Gigabit switch. The network should not be a limiting factor for the tests.

Java version	7u51 x64
OS	CentOS 6.0
Processor	Intel Core 2 Duo CPU E8400 3.00GHz
RAM	4 GB
Disk	SATA 7200 RPM
Network	Gigabit switch

6 A primitive like 'fsync' has to be called.

The results of the tests are the throughput of the messages that are delivered to the subscribers[7], the CPU consumed by the MQTT server, and the message transmission latency, i.e. the latency caused by the network and the MQTT server.

A test is executed as a sequence of 3 steps. During the first step, every thread launches the publishers and the subscribers. Connections are opened. A subscription is made to a rendezvous topic in order to synchronize the start of all the publishers.

The second step starts after a time delay in order to allow the first step to be completed by every thread. Each thread publishes a message to the rendezvous topic. In this way, as the total number of threads is known, each thread can handle a countdown and start publishing messages when the countdown reaches zero. Messages are asynchronously published in samples of 100 messages every 100 ms, each message being sent by a different publisher. The message production lasts 5 minutes, which appeared to be long enough to reach a steady state[8]. The second step ends when all the messages have been received by the subscribers. Therefore, the test also checks that there is no message lost by the server. The delivered throughput, CPU usage and message transmission latency are measured after a warmup period equal to 1 minute. The CPU is periodically measured with the Linux command 'top' and an average is made at the end of the test. The latency is measured for every published message. A timestamp is added to the payload of every message. As the clock used by the publishers and the subscribers is the same (same machine), the time required to transmit a message from a publisher to a subscriber can be obtained. The latency result shown by the graphics is the average of the latencies of all the messages published after the warmup period.

The third step starts after a time delay allowing not to affect the measures done during the second step. During this step, subscribers unsubscribe and connections are closed.

The MQTT server is started only once for all the tests.

Every server is assigned the same amount of available memory. Java servers (ActiveMQ, Apollo and JoramMQ) are assigned a maximum amount of memory equal to 3 GB. RabbitMQ automatically calculates a high watermark equal to 3 GB. Mosquitto does not define any limit.

The effect of the memory limit has not been studied by this benchmark.

The clients are written in Java. They do not affect the performance results. The CPU level on the client side does not exceed 70%. This CPU limit is obtained with 50.000 publishers launched on a single machine, with 5 processes (JVM process) and 10 threads per process.

7 The tests should always reach a steady state, otherwise the results are not valid. Therefore, the delivered throughput is equal to the throughput of the messages that are pushed to the server.

8 Tests launching the maximum number of publishers supported by an MQTT server, and lasting 50 minutes, have shown that the steady state (i.e. steady delivered throughput, CPU usage and latency) is reached very quickly in less than a minute. Some mechanisms like Java's Garbage Collection have a CPU usage that fluctuates over time with a long period, e.g. 30 minutes. However no perturbation has been observed during the long test lasting 50 minutes. At worst, the tests could be considered as giving an optimistic view of the different MQTT servers.

3 Benchmark results with QoS level 0 (no acknowledgement, no persistence)

3.1 server CPU usage

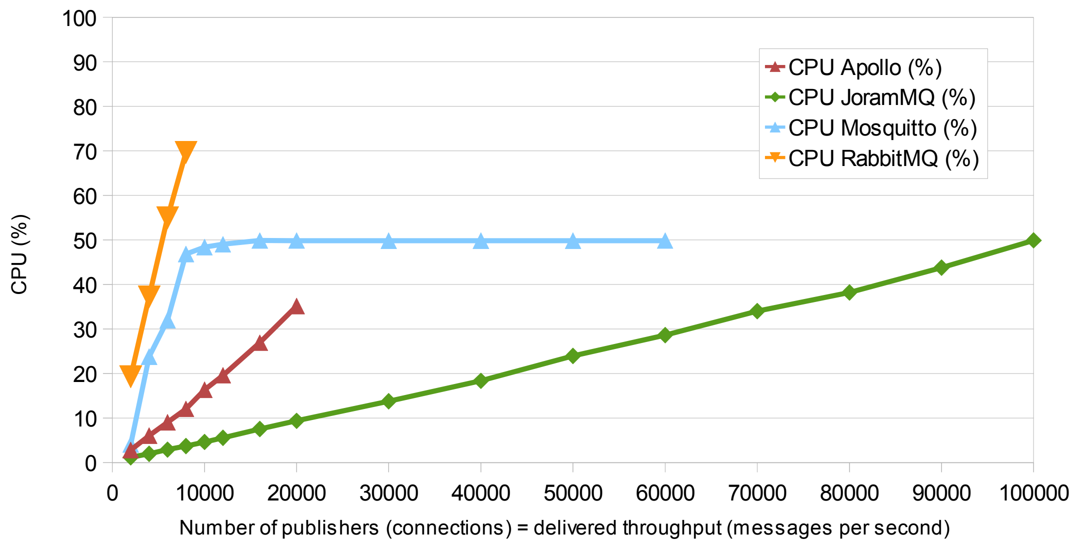

Figure 4: server CPU usage

3.2 Message transmission latency

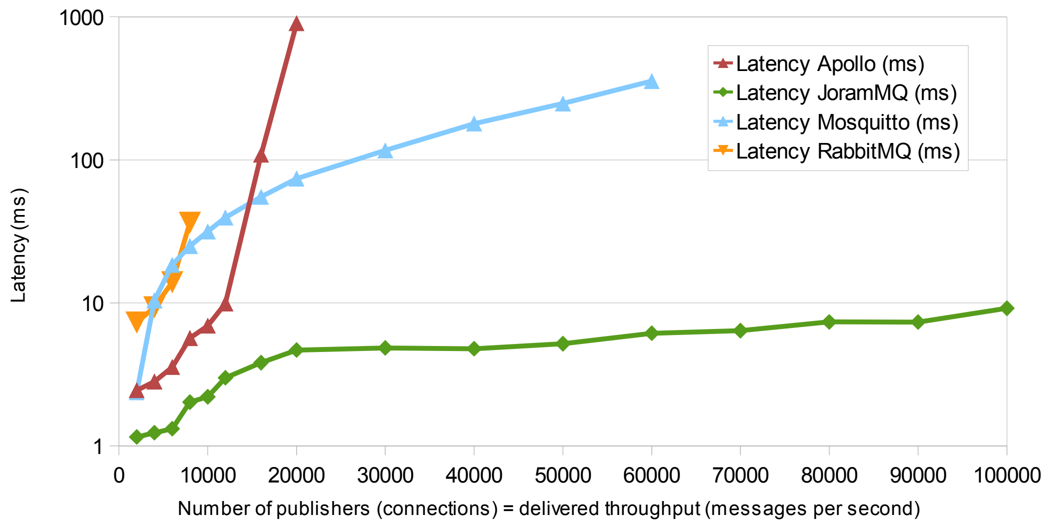

Figure 5: message transmission latency

3.3 Discussion

ActiveMQ has been configured to use the “mqtt+nio” connector (named after the “nio” Java library) that should bring more scalability than the basic “mqtt” connector. However, despite the use of this connector, ActiveMQ cannot handle the minimum load: 2000 publishers and 2000 messages per second. Therefore, ActiveMQ does not appear in the results.

RabbitMQ cannot handle more than 8000 publishers producing 8000 messages per second. The limiting factor seems to be the CPU usage.

Apollo consumes too much memory and cannot handle more than 20.000 publishers.

Mosquitto succeeds in handling 60.000 publishers despite a high level of CPU usage probably caused by the way connections are handled. The high transmission latency suggests that messages are buffered before being processed. Mosquitto is single threaded which explains the maximum CPU at 50% of the dual-core. The steady CPU level stuck at 50% hides the fact that the server progressively reaches the maximum number of publishers. The expected throughput is not obtained above 60.000 publishers.

JoramMQ manages to handle 100.000 publishers producing 100.000 messages per second, with a low CPU usage (50%) and a low latency (10 ms). A future version of this benchmark will evaluate the performance of JoramMQ above 100.000 publishers.

4 Benchmark results with QoS level 1 and durable subscriptions (publish acknowledgement, message persistence)

4.1 server CPU usage

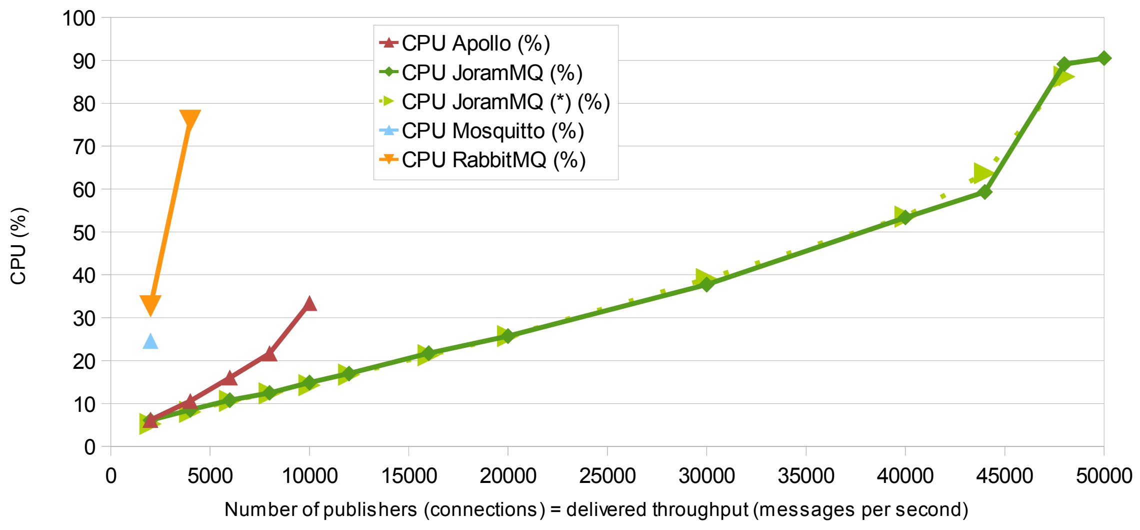

Figure 6: server CPU usage

4.2 Message transmission latency

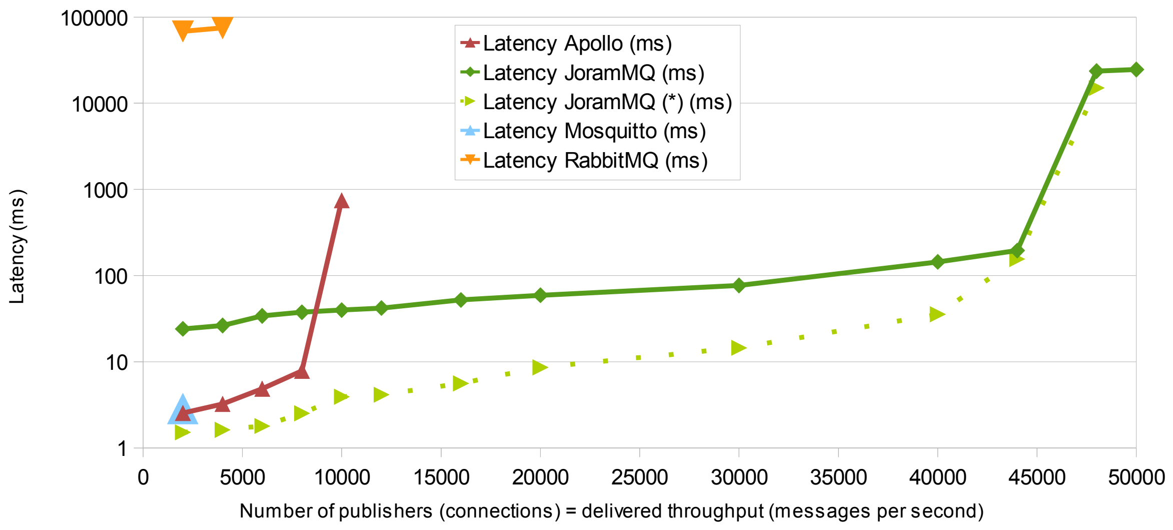

Figure 7: message transmission latency

(*) Special configuration of JoramMQ (dotted line), optimized in the same way as Apollo's default configuration (see 4.3).

4.3 Discussion

Mosquitto could not be configured to save every message before acknowledgement to the publisher and delivery to the subscriber (see section 5.4 for more details). The unique result shown above (at 2000 publishers) has been obtained by configuring Mosquitto to trigger a save every second. Above 4000 publishers, Mosquitto cannot keep up with the incoming throughput.

RabbitMQ hardly passes the test below 4000 publishers. The steady state is not reached. The delivering rate is lesser than the incoming rate as shown by RabbitMQ administration console. The maximum memory allowed is used (the high watermark is 3 GB). Messages are accumulated in the MQTT server which explains the very high latency.

By default, Apollo uses an optimization allowing to deliver a message before it is actually saved in the data store, so before it is effectively enqueued[9]. This optimized behaviour explains why the latency is almost the same as at QoS level 0. However this optimization is not fully compliant with the MQTT specification[10].

JoramMQ does not provide this optimization by default in order to be strictly compliant with the MQTT specification. However, for a fair comparison of the latency between JoramMQ and Apollo, the same optimization has been applied to JoramMQ and presented with a dotted line. The results show that when the optimization is enabled, JoramMQ's latency is lower than Apollo's latency.

Apollo cannot handle more than 10.000 publishers. Apollo's limiting factor seems to be linked to an excessive memory usage. The maximum allowed memory is almost entirely used with 12.000 publishers and the CPU becomes very unsteady, often reaching 100%. An OutOfMemoryError is sometimes raised by the server.

JoramMQ succeeds with 44.000 publishers. At 48.000 publishers, the CPU usage is 90% and JoramMQ cannot handle the incoming message throughput equal to 48.000 messages per second.

9 See the attribute 'flush_delay' of the element 'leveldb_store' in Apollo configuration.

10 If the server is stopped or if a system crash occurs during the delivery of a message, then the message redelivery depends on the client that published the message. As the publish acknowledgement has not been returned by the server, the client should re-publish the message. However the message may not be re-published, and if it is, then it may be redelivered with a different MQTT Packet Identifier. Moreover the Packet Identifier that has been used for the initial delivery (before the server has stopped) may be re-used for another message. The following statements of MQTT 3.1.1 are not fulfilled: MQTT-2.3.1-2, MQTT-2.3.1-3, MQTT-2.3.1-4, MQTT-4.4.0-1.

5 Configuration of the servers

The default configuration provided by each server has been used except for the following aspects, as long as they can be configured:

- the maximum amount of memory allowed
- the size of the TCP socket buffers (sending and receiving)
- the persistence of the messages published with QoS 1 and the ability to deliver a persistent message before it has been stored
- the maximum number of in-flight messages, i.e. QoS 1 messages that can be in the process of being transmitted simultaneously

The maximum amount of memory used by the servers written in Java (ActiveMQ, Apollo and JoramMQ) has been set to 3 GB. RabbitMQ automatically deduces the same limit from the available RAM (indicator called “high watermark”). Mosquitto does not define such a limit.

The size of the TCP socket buffers has been reduced to 1024 Bytes in order to avoid an excessive memory allocation due to the large number of publisher connections (up to 100.000). If the server allows to create several MQTT connectors with different listening ports and socket buffer sizes, then two connectors are created, one for the multiple publishers (2000 to 100.000), and another one for the few subscribers (2 to 100).

The persistence of the messages published with QoS 1 needs to be enabled. ActiveMQ, Apollo, JoramMQ and RabbitMQ are configured to persist QoS level 1 messages with a sync to disk by default.

Mosquitto disables persistence by default and does not allow the sync to disk. Mosquitto does not use 'fsync', but 'fclose' which only flushes the user-space buffers. As a consequence some messages can be lost in case of a system crash.

Apollo enables by default an optimization allowing to deliver a persistent message before it has been stored. JoramMQ also provides this optimization but not by default as it breaks some of the MQTT specification statements (see 4.3).

The maximum number of in-flight messages should not restrain the message delivery to the subscribers.

5.1 ActiveMQ

Two MQTT connectors are specified in 'activemq.xml' as follows:

```
<transportConnector name="mqtt+nio" uri="mqtt+nio://0.0.0.0:1883?
maximumConnections=100000&amp;wireFormat.maxFrameSize=104857600&amp;socke
tBufferSize=1024"/>

<transportConnector name="mqtt" uri="mqtt://0.0.0.0:1884?
maximumConnections=100&amp;wireFormat.maxFrameSize=104857600&amp;socketBu
fferSize=32768"/>
```

5.2 Apollo

Two MQTT connectors are specified in 'apollo.xml' as follows:

```
  <connector id="tcp-mqtt1" bind="tcp://0.0.0.0:1883" protocol="mqtt"
    receive_buffer_size="1024" send_buffer_size="1024"
    receive_buffer_auto_tune="false" send_buffer_auto_tune="false"/>

  <connector id="tcp-mqtt2" bind="tcp://0.0.0.0:1884" protocol="mqtt"
    receive_buffer_size="32768" send_buffer_size="32768"
    receive_buffer_auto_tune="false" send_buffer_auto_tune="false"/>
```

5.3 JoramMQ

Two MQTT connectors are specified in 'jorammq.xml' as follows:

```
tcp://0.0.0.0:1883?
prod.win=1024&amp;cons.win=2048576&amp;mx.inflight=32768&amp;qos0.win=167
77216&amp;snd.buf=1024&amp;rcv.buf=1024&amp;q0=false

tcp://0.0.0.0:1884?
prod.win=2048576&amp;cons.win=2048576&amp;mx.inflight=32768&amp;qos0.win=
16777216&amp;snd.buf=32768&amp;rcv.buf=32768&amp;q0=false
```

5.4 Mosquitto

Mosquitto can provide several connectors but the socket buffer size cannot be configured. So only one connector has been used, the default one listening to port 1883.

By default, Mosquitto does not enable persistence. So the parameter 'persistence' has been explicitly enabled when testing QoS level 1.

The parameter 'autosave_on_changes' should allow to save a message as soon as it has arrived. But enabling 'autosave_on_changes' does not trigger any write call to disk during the tests (as shown by the Linux command 'iotop'). So 'autosave_on_changes' has been left to its default value 'false', which saves the in-memory database to disk by treating 'autosave_interval' as a time in seconds.

The parameter 'autosave_interval' is set to 1 second in order to regularly save the in-memory database to disk.

Mosquitto should not drop any QoS 1 message so 'max_queued_messages' is set to 0. And the message delivery rate should not be constrained by a limit of the number of in-flight messages so 'max_inflight_messages' is set to 0.

The file 'mosquitto.conf' is modified as follows:

```
persistence true
autosave_on_changes false
autosave_interval 1
max_queued_messages 0
max_inflight_messages 0
```

5.5 RabbitMQ

RabbitMQ can provide several connectors but with the same configuration. So only one has been specified in 'rabbitmq.config' as follows.

```
       {tcp_listeners,      [1883]},
       {tcp_listen_options, [binary,
         {packet,    raw},
         {reuseaddr, true},
         {backlog,   128},
         {sndbuf,    1024},
         {recbuf,    1024},
         {nodelay,   true}]}]}
```