JoramMQ, a distributed MQTT broker for the Internet of Things

White paper and performance evaluation

v1.2

September 2014

mqtt.jorammq.com

www.scalagent.com

1 Overview

Message Queue Telemetry Transport (MQTT) is an open Machine-to-Machine (M2M) protocol, that has been invented in 1999, and that is in the process of undergoing standardisation at OASIS[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).

MQTT allows to collect data from remote devices and more specifically from devices at the edges of the network, and push these data into systems in a data centre, for example a Big Data processing system based on Apache Hadoop.

MQTT also allows to publish data and alerts to systems like smartphones, tablets and laptops, enabling users to easily and efficiently monitor the data.

Figure 1 represents the architecture of an IoT application, collecting data from many devices, processing and storing these data, and notifying the final users with alerts and reports. The collected data can be directly published in real-time to the final users. Commands are triggered by the final users, archived in the data store and transmitted to the devices.

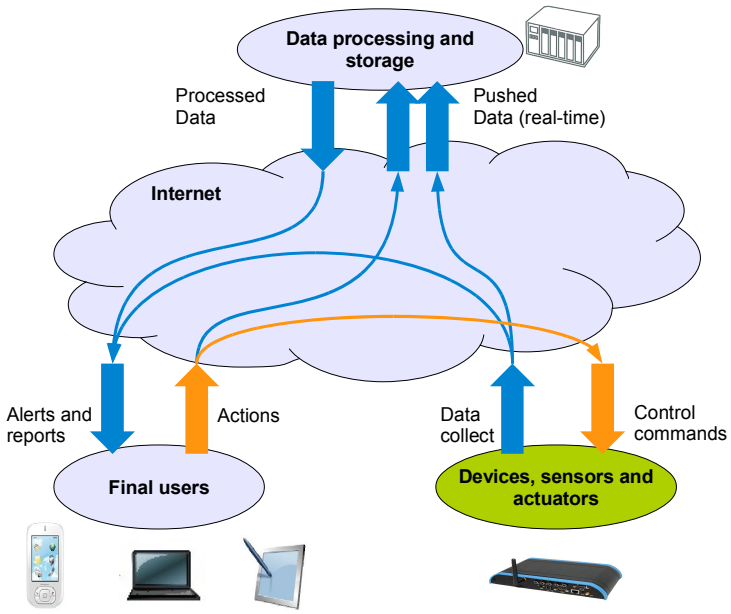

Figure 1: IoT application

In the IoT application represented above, every device, data processing system and monitoring interface (e.g. smartphone) is potentially an MQTT client that produces and consumes telemetry data. Control commands are also sent as MQTT messages like any other data types. Bi-directional messaging is a requirement addressed by MQTT to uniformly handle both telemetry and commands.

1 https://www.oasis-open.org/news/announcements/60-day-public-review-for-mqtt-version-3-1-1-cos01-ends-september-4th

The MQTT protocol relies on a message broker according to the hub and spoke model of Message Oriented Middleware (MOM). As shown by figure 2, every MQTT client, data processing application or device, producer or consumer, needs to connect to a central broker before communicating with other MQTT clients. The broker accepts published messages and delivers them to the interested consumers according to a Publish/Subscribe interaction pattern.

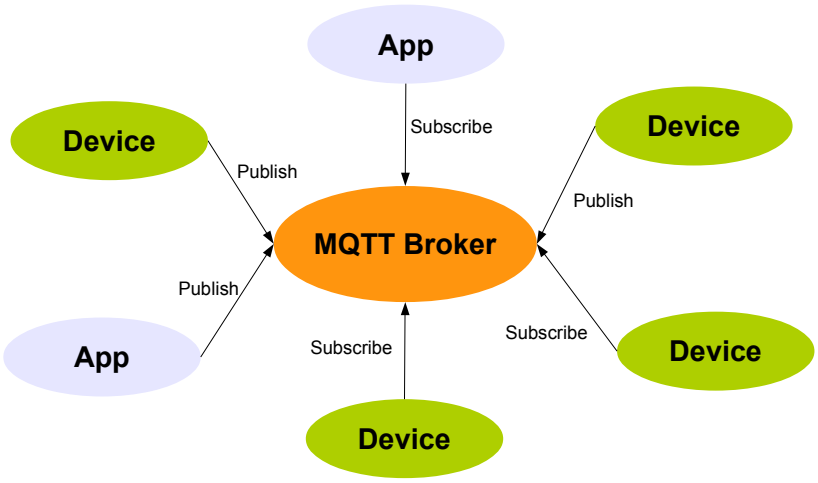

Figure 2: Hub and spoke communication

The broker model has advantages and drawbacks. This document gives some reasons why the broker model is particularly well-suited for IoT applications and how JoramMQ[2] provides the capabilities of MQTT while addressing the issues raised by the broker model.

In particular JoramMQ makes the broker scale with the number of connected devices by distributing JoramMQ servers on many hosts located in different networks close to the devices.

2 The JoramMQ offering by ScalAgent is built on top of the JORAM open-source product, a Message Oriented Middleware (MOM) that provides the JMS API. JoramMQ also provides the AMQP protocol (v0.9 and v1.0) and the MQTT protocol (v3.1).

Figure 3 illustrates the capability of JoramMQ to distribute the MQTT broker close to the devices, concentrate the telemetry data, and scale with the number of devices. JoramMQ servers topology is not limited to a hierarchy, i.e. a tree-like topology. However in the context of IoT, JoramMQ servers are mapped to the hierarchical structure of a system made of subsystems and devices. Applications can be connected at any level of the hierarchy. Servers are connected to each other through an efficient and reliable protocol that transmits messages in batches without increasing the transmission latency[3].

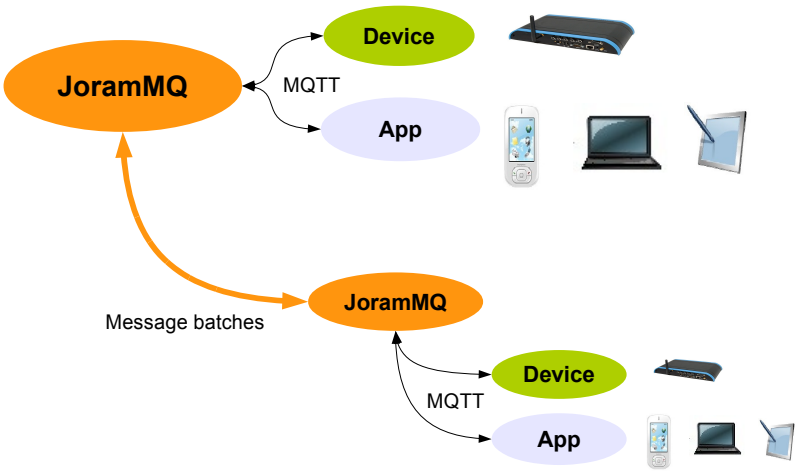

Figure 3: Distributed broker

This document is organised as follows:

Section 2 gives the advantages and drawbacks of the broker model regarding the Publish/Subscribe interaction pattern;

Section 3 explains how the message broker provided by JoramMQ scales with the number of MQTT clients;

Section 4 presents an MQTT performance test bench.

Sections 5, 6, and 7 describe different test scenarios and give JoramMQ performance results.

3 Instead of waiting for a predefined number of messages to be sent or a predefined time interval to elapse, JoramMQ gathers all the messages that are ready to be sent at a given time in one batch and pushes the batch to the network.

2 Publish/Suscribe and the broker model

MQTT is based on a Publish/Subscribe interaction pattern. A message is published once on a given topic, i.e. subject of interest, and every consumer that registered to this topic receives a copy of the message.

Figure 4 illustrates the Publish/Subscribe pattern with a publisher pushing a message containing the text “hello” to a topic having 3 subscribers. Every subscriber receives a copy of the message.

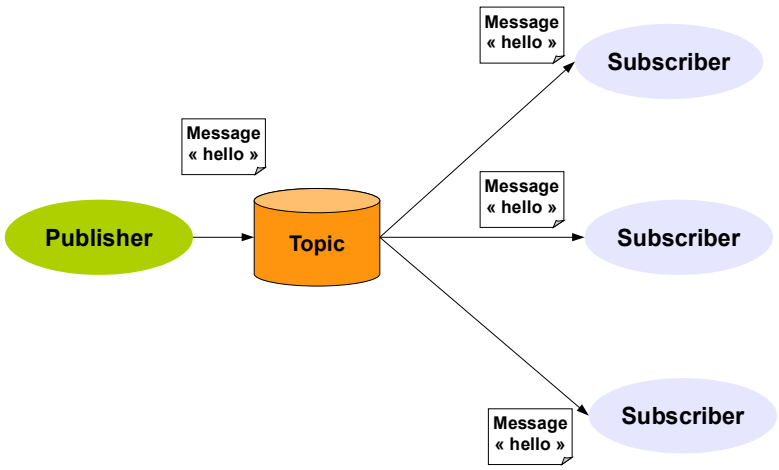

Figure 4: Publish/Subscribe interaction pattern

MQTT clients interact with a central broker, also called server, either by subscribing to topics, or publishing messages to topics. The broker matches every published message to the subscriptions. If there is no match, then the message is discarded. If there is one or more matches, then the message is delivered to each matching subscriber.

The broker model is often opposed to a brokerless approach, also called peer-to-peer architecture, where every application directly communicates with each other. It is interesting to compare both approaches especially regarding the Publish/Subscribe interaction pattern.

2.1 Benefits of the broker model

The broker model has the following advantages over the brokerless approach.

Space decoupling

Applications using Publish/Subscribe do not need to know the location of other applications. The only address an application needs to know is the network address of the broker. The broker then routes the messages to the right applications based on the data semantic, e.g. the MQTT topic, rather than on physical topology, e.g. the IP address.

Space decoupling is a real need for dynamic system environments where the physical address of a device or an application may be unknown, unreachable, or changing. This is particularly useful for pushing the telemetry data up to the consumers whose addresses are not known by the devices. This is also useful for sending commands to a target device, as the physical address of the device is usually not reachable outside the network of the device.

Another benefit of space decoupling is to enable many consumers to dynamically subscribe to some data without affecting the producers of these data. An IoT application needs to make available high volumes of data in ways that may not have been originally anticipated. Decoupling the knowledge of the meaningful information (e.g. the telemetry parameters and the commands) from the knowledge of the physical topology is a simple and efficient way to monitor and control many devices located in many different networks.

Time decoupling

Publishers and subscribers do not have to be timely coupled. The publisher application can push messages to the broker and terminate. The messages will be available for the subscriber application any time later.

The unreliability of the network edges, the distributed nature of IoT applications and the heterogeneity of the connected devices and applications (e.g. slow consumers) make time decoupling a fundamental and mandatory property of the communication in an IoT application.

Reliability

The broker model brings communication reliability to the client applications. Message delivery can be guaranteed without coupling the publisher application to the subscriber application by persisting the messages to disk[4]. IoT applications need a simple and robust solution like the broker model in order to guarantee that no data is lost[5].

4 JoramMQ optimizes the reliable delivery performance thanks to efficient persistence mechanisms and message batching.

5 An MQTT broker provides two reliable QoS levels called “at least once” and “exactly once”.

2.2 Drawbacks

The broker model also has the following drawbacks compared to the brokerless approach.

Higher message transmission latency

As presented in section 2.1, a broker brings time decoupling between a message producer and a consumer. Therefore, a broker increases the transmission latency of a message, especially if message persistence is required.

However, time-decoupling should not be sacrificed in the attempt to reduce latency well beyond the point that it matters in an IoT application.

JoramMQ has been designed to reduce the message transmission latency as much as possible in timely decoupled interactions.

Higher network bandwidth consumed

The broker model requires a “two hops” communication, from publisher to broker and broker to consumer, leading to a bigger amount of network bandwidth than with the brokerless approach.

This drawback is not fully correct in the Publish/Subscribe interaction pattern, because data are published once and consumed many times. So the “two hops” overhead is strongly reduced by the number of times the data are consumed. If the published data are consumed N times then the broker approach produces N+1 messages, whereas the brokerless approach produces N messages. The gain provided by the brokerless approach is lowered by the number of times the data are consumed.

Moreover the most constrained communication link is the connection to a device because of the low bandwidth and also the constrained capabilities of the device. A message broker allows to address these constraints and optimize the communication with the device.

Multicast is often presented as a solution to reduce the bandwidth usage. However all the subscribers generally do not listen to the same data. Therefore, even with a multicast approach, a broker (or gateway) is needed at the end to match the publications to the subscriptions and deliver the data to the matching subscribers, potentially located in a different network.

Centralised architecture

As all the messages of a system are passed through the central broker, it becomes the bottleneck of the whole system.

In order to avoid this issue, JoramMQ distributes the broker in multiple servers, potentially deployed in different local networks. The next section of this document explains how a distributed MQTT broker can be deployed with JoramMQ.

3 JoramMQ distributed broker

3.1 Overview

A typical IoT application where the MQTT protocol could be used is the monitoring and control of a system containing many devices collecting telemetry parameters from sensors and transmitting commands to actuators. Such a system is usually represented as a hierarchy of sub-systems, the highest level being the system and the lowest level being the sensors (represented by parameters) and the actuators (represented by commands).

Figure 5 gives an example of topic hierarchy.

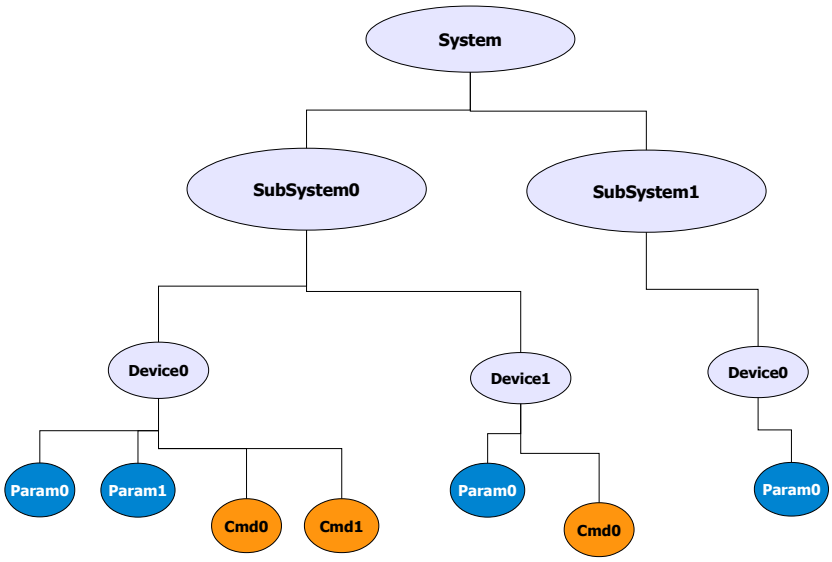

Figure 5: Topic hierarchy example

The published data can be easily mapped to an MQTT topic structure. For example, the name of the topic used by *Device0* in *SubSystem0* to publish the telemetry parameter *Param0* would be:

System/SubSystem0/Device0/Param0

Commands are also mapped to MQTT topics. For example, the command *Cmd0* provided by *Device0* in *SubSystem0* is mapped to the following topic:

System/SubSystem0/Device0/Cmd0

An MQTT topic is just a name, hierarchically structured. A topic has no particular physical location in the network. However, the subscribers need to create subscription contexts somewhere and published messages need to be routed to these contexts.

JoramMQ manages the subscription contexts and the published messages routing by combining three broker topologies:

1. centralised broker
2. clustered broker
3. distributed broker

3.2 Centralised broker

A centralised broker is made of a single server. Every MQTT client needs to connect to this server.

A subscription context is the root of a local subscription tree containing one node per subscription topic. The subscription tree is dynamically created according to the MQTT subscriptions. When a client subscribes to a given topic, the broker dynamically creates a subscription node for this topic.

A subscription context can be explicitly created for the top level topic, called “System” in figure 5. However this creation is not mandatory as a default subscription context is created in every JoramMQ server.

Figure 6 represents a subscription context that is dynamically populated with different subscription nodes according to the MQTT subscriptions.

Two kinds of subscription nodes are distinguished. The root node (blue background) is statically created with the subscription context. The other nodes (green background) are dynamically created as the subscriptions are made by the MQTT clients.

Published messages whose topic name starts with “System” are delivered to the subscription context “System”. Then the references of the subscribers are obtained by applying the MQTT topic matching rules at each level of the subscription tree.

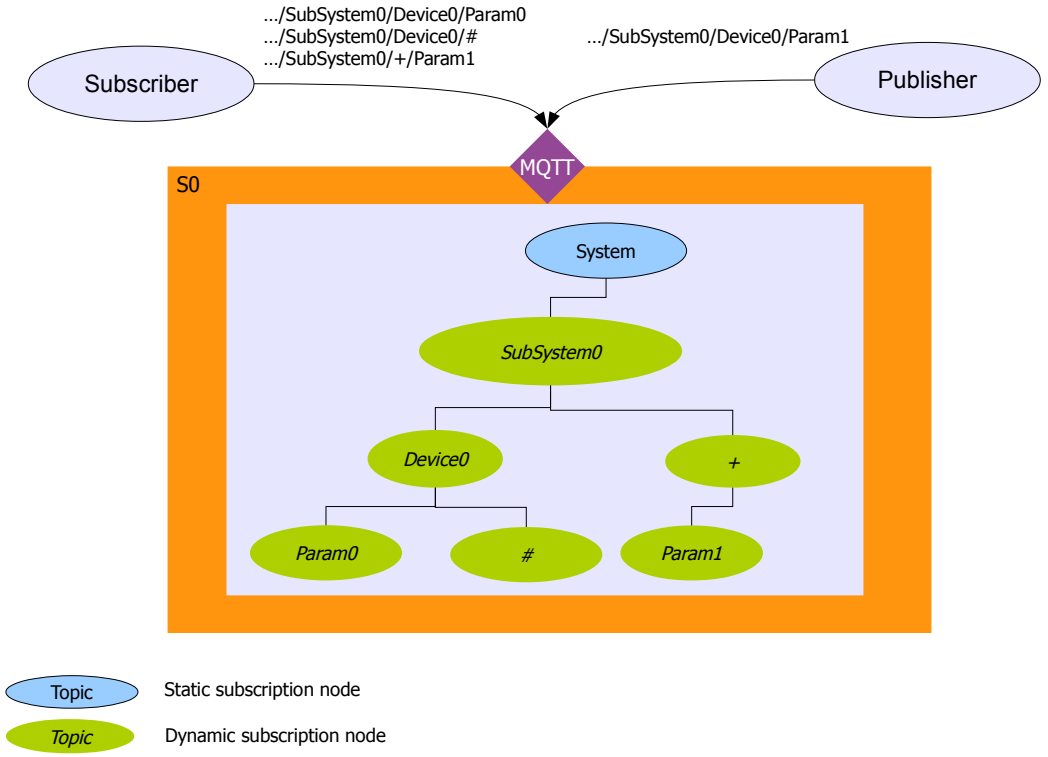

Figure 6: Centralised broker

3.3 Clustered broker

A clustered broker is made of a group of servers. Each server of the cluster knows the subscriptions made in this server and in the other servers of the cluster. Every subscription is replicated in all the servers of the cluster. A published message is forwarded[6] to every server that owns subscriptions to the topic of the message. The goal is to balance the subscription load in terms of the broker CPU and the network bandwidth used by the subscribers.

Every MQTT client needs to connect to one server of the cluster. The MQTT client sessions are not replicated. Therefore, a given MQTT client, once connected, cannot switch from a server to another, except if the MQTT session is configured to be cleaned after a disconnect.

Figure 7 illustrates how the subscription nodes are replicated among a cluster of two servers S0 and S1 according to the MQTT subscriptions. A subscription context is created in every server at the topic level that needs to be clustered, for example “System”. Then the subscription nodes are dynamically created by the MQTT subscription commands. The subscribers can connect to any server of the cluster.

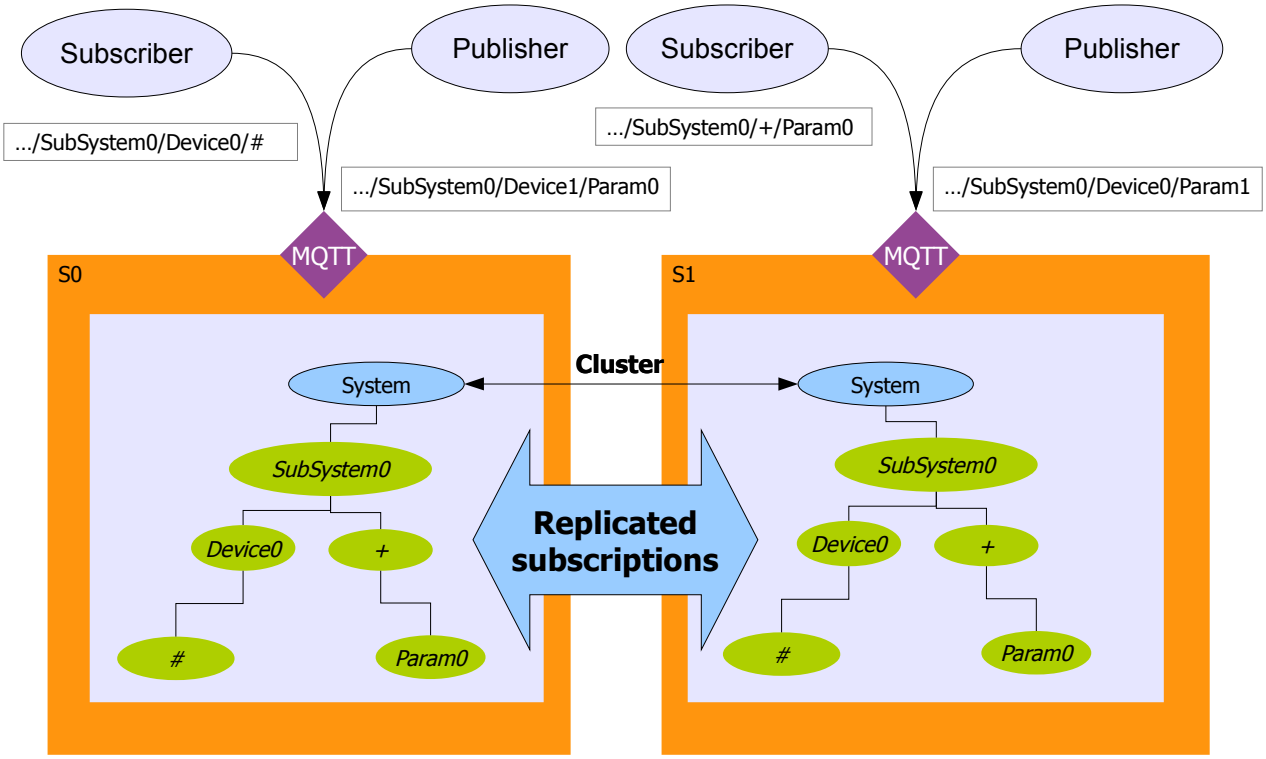

Figure 7: Clustered broker

6 The QoS level of the forwarded message is applied when the message is transmitted from one server to another server of the cluster. In case of server failure, no message published at QoS level 1 or 2 can be lost and the “exactly once” property of QoS 2 is ensured.

3.4 Distributed broker

The goal of a distributed broker is to scale with the number of MQTT clients using sub-topics in the topic hierarchy, especially if these clients are distributed, like for example devices at the edges of the network. Each device could be represented by a dedicated topic. The device would publish data to this topic and subscribe to commands coming from this topic.

A distributed broker is made of a hierarchy of subscription contexts distributed in several servers. In this subscription context hierarchy, each context has a unique parent (except the root) and several children (except the leaves of the tree). Published messages are transmitted upward and downward the tree of subscription contexts:

- upward as a child context systematically forwards the published messages to its parent;
- downward as a parent context transmits the published messages to the related children contexts.

The upward publishing is typically used by devices to push data up to the data processing and storage systems. In this way, the server used by the data processing is not overwhelmed by all the connections of the devices. The server used by a device gathers the messages and efficiently transmits them as batches in a single server-to-server connection.

The downward publishing can be used for example by final users to send commands to a device connected to a remote server.

Figure 8 shows how the subscription context “System/SubSystem0” can be located in a remote server (S1) close to the devices (e.g. same local network), allowing these devices to publish data and subscribe to commands. The remote subscription context acts as a concentrator gathering the MQTT messages published by the local publishers and pushing them to the higher level contexts. The MQTT messages are efficiently transmitted between servers thanks to a message batching protocol that does not increase the transmission latency.

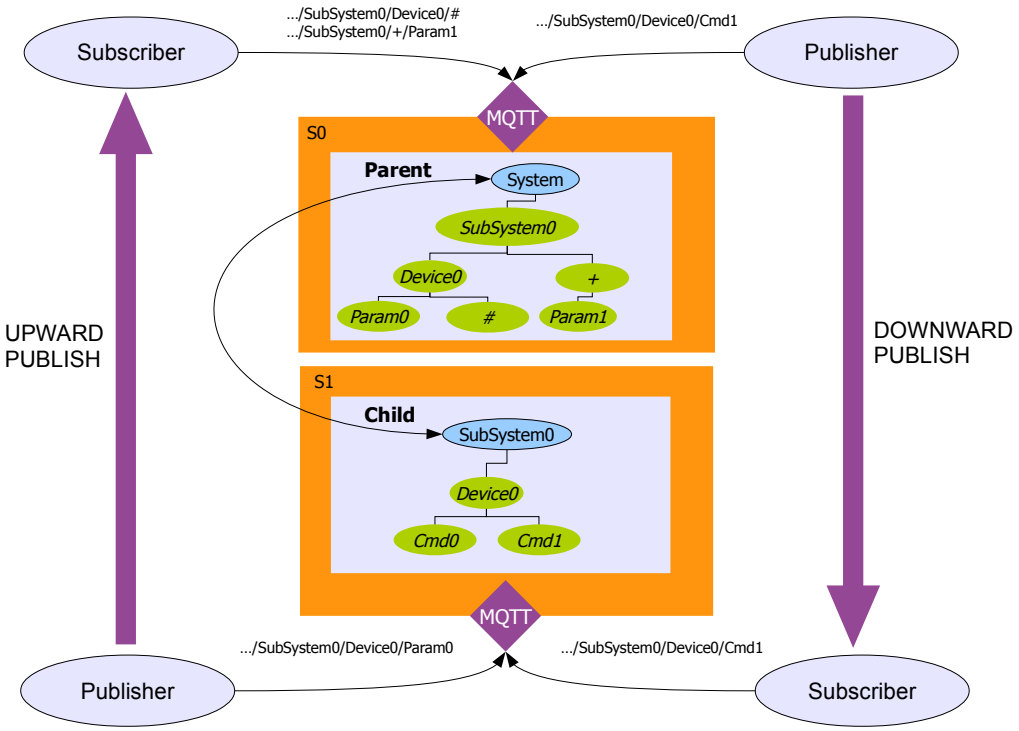

Figure 8: Distributed broker

3.5 Clustered and distributed broker

As explained in section 3.4, a distributed broker topology is structured as a hierarchy of servers, which may lead to a bottleneck at the level of the root server if it is not able to handle the message traffic. In this case, the distributed broker topology should be mixed with the clustered broker topology.

Figure 9 illustrates a mixed topology including two clustered servers S0 and S1, and two distributed servers S2 and S3 deployed close to multiple devices to monitor and control.

At the bottom, S2 and S3 aim at scaling with the number of devices and also at improving the connectivity at the edges of the network. The messages going through the root topic “System” are statically load-balanced across servers S0 and S1. Subsystem0 is connected to S0 and Subsystem1 is connected to S1.

At the top, the applications can connect either to S0 or S1, and subscribe or publish messages to the topic hierarchy below “System”. If an application, e.g. a data store, subscribes to the whole message flow going through the topic “System” then the servers S0 and S1 have to be able to deliver such a message throughput.

If the message throughput is too high, then the root topic “System” should be partitioned in several topics created in different servers. It should be noticed that the delivery of constant streaming of high volumes of data to a single client is not a normal use case for MQTT. Another delivery mechanism such as JMS (Java Message Service) or the protocol AMQP should be used instead. JoramMQ allows clients to connect to the broker by using different protocols such as AMQP, MQTT, and JMS/Joram. Such heterogeneous clients can then interoperate thanks to the common interaction pattern “Publish/Subscribe” provided by each protocol.

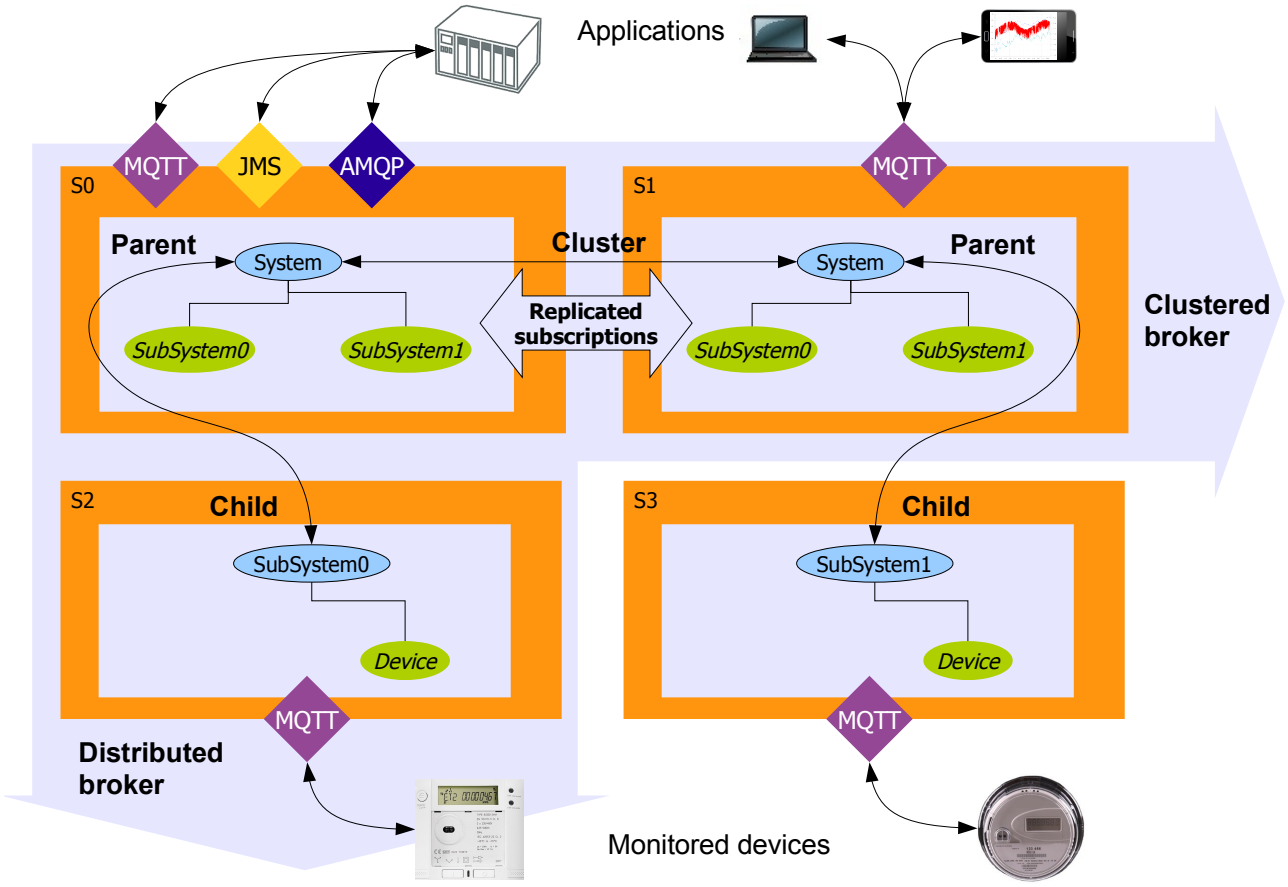

Figure 9: Clustered and distributed broker

4 MQTT performance test bench

4.1 Overview

The goal of the test bench is to evaluate the scalability of an MQTT broker with the number of clients, either publishers or subscribers. The test bench allows to check that in a given context (QoS level, message throughput per client), the broker scales with the number of clients.

Three test scenarios are provided by this test bench. They are described in terms of the number of publishers and subscribers (few or multiple), the number of topics per subscription and the number of subscribers per topic.

Scenario name	Publishers	Subscribers	Topics per subscription	Subscribers per topic
Multi-publisher	Multiple	Few	All	1
Multi-subscriber	Few	Multiple	1	1
Multi-subscription	Few	Multiple	Multiple	Multiple

The message payload size is fixed and set to 64 bytes.

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

QoS levels 1 and 2 are only tested with durable subscriptions, i.e. the clean session flag is false.

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

JoramMQ provides two ways to deliver messages at QoS 0. The first way consists in directly delivering the messages without queuing them before. In case of overflow, messages are dropped. The second way requires to queue messages at QoS 0. In case of overflow, messages are swapped to disk. The first way (not queued) is more efficient than the second one (queued). The first way is tested by the scenarios multi-publisher and multi-subscriber. The second way is tested by the multi-subscription scenario.

QoS level 1 requires that a Publish message is acknowledged after it has been physically persisted to storage. So the message needs to be persisted and a sync to disk needs to be performed before the acknowledgement can be returned to the publisher. QoS level 2 has the same requirement regarding acknowledgements PUBREC, PUBREL and PUBCOMP.

The following broker topologies are tested: centralised broker, clustered broker, distributed broker.

The results of the tests are the throughput of the messages that are delivered to the subscribers, called the “delivered throughput”[7], and the CPU consumed by the MQTT broker.

7 As the tests always reach a steady state, the “delivered throughput” is the throughput of the messages that go through the broker, i.e. that are pushed on the publisher side and delivered on the subscriber side. The delivered throughput is equal to the throughput of the messages that are pushed in the broker.

Four machines are used with the same configuration listed in the table below.

Java version	7
OS	CentOS Linux 6.0
Processor	Intel Core 2 Duo CPU E8400 3.00GHz
RAM	4 GB
Disk	SATA 7200 RPM

4.2 Centralised broker testing

The test environment used with a centralised broker is represented by figure 10. The broker is launched as a single server on a single machine. Clients are launched in several processes producing and consuming messages through a dedicated topic hierarchy. Topic hierarchies are not shared between processes. Each process produces and consumes messages using its own topic hierarchy. The scalability of the broker with the number of clients is tested by incrementally increasing the number of processes.

Three machines are required by the centralised broker testing: one for the broker and two for the clients.

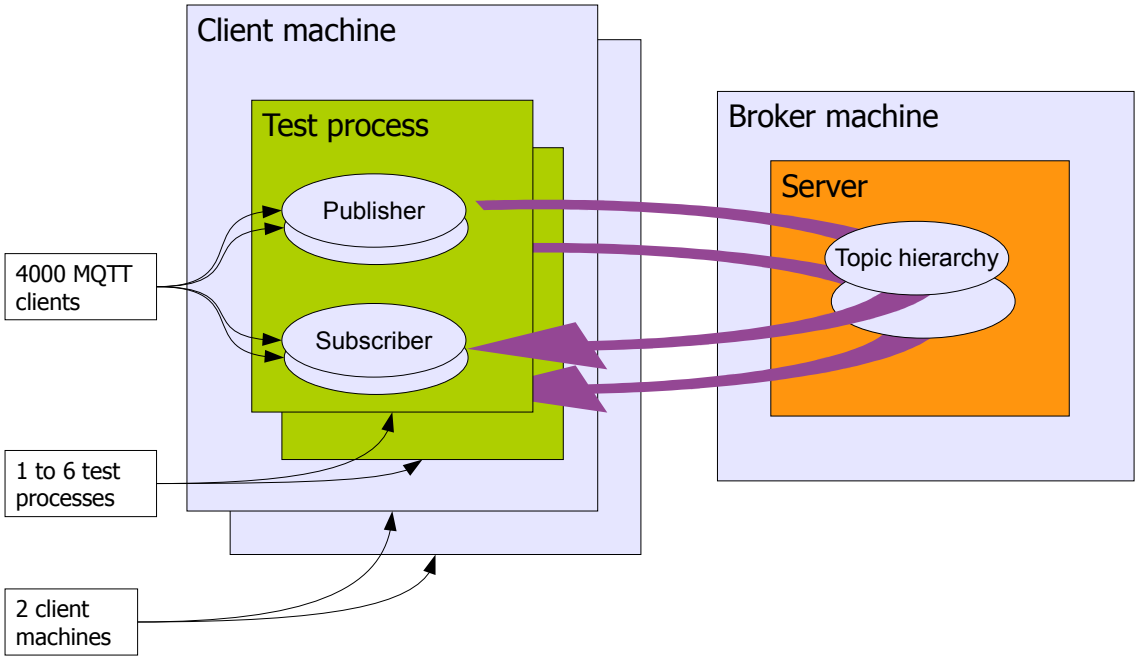

Figure 10: Centralised broker testing

In the multi-publisher and multi-subscriber scenarios, clients are added in batches of 4000 running in a new process (JVM). A limit of 6 processes per client machine, each process starting 4000 clients, allows to reach a maximum of 48.000 clients. The minimum number of clients is obtained with two client machines, each running one process. Therefore the minimum number of clients is 8000.

On the broker side, the maximum number of clients whose connections can be accepted depends on several limits:

- the maximum number of file descriptors, as one file descriptor is created per socket; this size can be easily increased at the OS level;
- the size of the TCP receiving and sending buffers;
- the amount of memory required by the broker to handle an MQTT client; JoramMQ does not need much memory to handle a client so this is not a limiting factor for the tests.

A centralised broker can accept more than 48.000 connections. However the global message throughput that a centralised broker can accept and deliver is limited. It is approximately constant with the number of connections. As a consequence, the message throughput per connection decreases with the number of connections. In the context of the tests, in order to reach interesting message throughputs per connection, the maximum number of connections with a centralised broker is limited to 48.000. The tests with clustered and distributed brokers are limited to 96.000 connections (see sections 4.3 and 4.4).

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 broker scales with the number of clients, each client producing or consuming at a fixed rate.

QoS levels 0 and 1 are tested at a nominal rate equal to 0,1 message per second (msg/s) per client, either publisher or subscriber. This rate makes sense for IoT applications where data are produced (e.g. telemetry) or consumed (e.g. commands) at a low frequency which may even be lower than 0,1 msg/s.

At QoS levels 0 and 1 two higher rates are tested depending on the scenario:

- Multi-publisher: 1 msg/s at QoS 0 and 0,25 msg/s at QoS 1
- Multi-subscriber: 0,6 msg/s at QoS 0 and 0,15 msg/s at QoS 1

The multi-subscriber scenario requires smaller rates than the multi-publisher scenario because delivering messages to multiple subscribers is more costly than accepting messages from multiple publishers.

At QoS 2, the nominal rate is above the maximum rate allowed with 48.000 clients. A smaller rate is chosen equal to 0,07 msg/s.

In the multi-publisher scenario, all the published message flows converge to the same endpoint on the consumer side, e.g. a data centre that receives all the collected telemetry data. In the multi-subscriber scenario, all the message flows are initiated from a unique endpoint on the producer side, e.g. a control centre that initiates commands.

4.3 Clustered broker testing

A cluster of two servers S0 and S1 is deployed and a replicated subscription context is created for each topic hierarchy.

The message rates are the same as with the centralised broker.

The number of MQTT clients is doubled. Clients are added in batches of 8000 running in a new process (JVM). This is twice the size of the batches used in the centralised broker testing (4000). The number of test processes goes from 1 to 6. Therefore, with two client machines, the maximum number of clients is 96.000.

Clients are assigned to a server in a round-robin way.

Four machines are required by the clustered broker testing: two for the broker and two for the clients.

Figure 11 represents how the clients are load-balanced across the clustered servers S0 and S1. The different client machines, test processes and topic hierarchies are not represented but they are the same as in figure 10 in section 4.2.

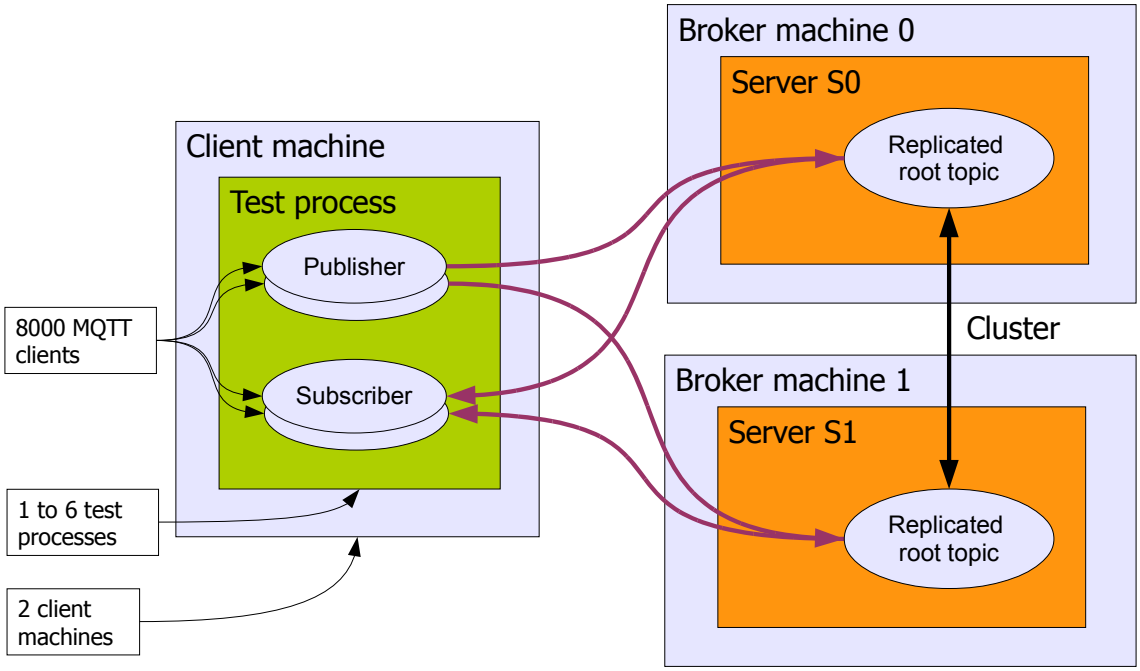

Figure 11: Clustered broker testing

4.4 Distributed broker testing

The distributed broker topology requires to distribute the subscription contexts, also called “topics” and “subtopics” hereafter. Two servers are deployed: S0 and S1. The root topic is created either on S0 or S1. Subtopics are distributed across S0 and S1.

Depending on the test scenario, the root topic either gathers the messages received from the subtopics (e.g. telemetry data) or routes messages to the subtopics (e.g. command messages).

In a real deployment, the distributed servers S0 and S1 would be deployed close to the remote MQTT clients (e.g. devices). Servers S0 and S1 would act as data concentrators. The root topic would be located in a third server.

Figure 12 represents how multiple publishers are load-balanced across S0 and S1. The multi-publisher scenario is presented in section 5.

In the same way, multiple subscribers can also be connected to S0 and S1. The multi-subscriber scenario is presented in section 6.

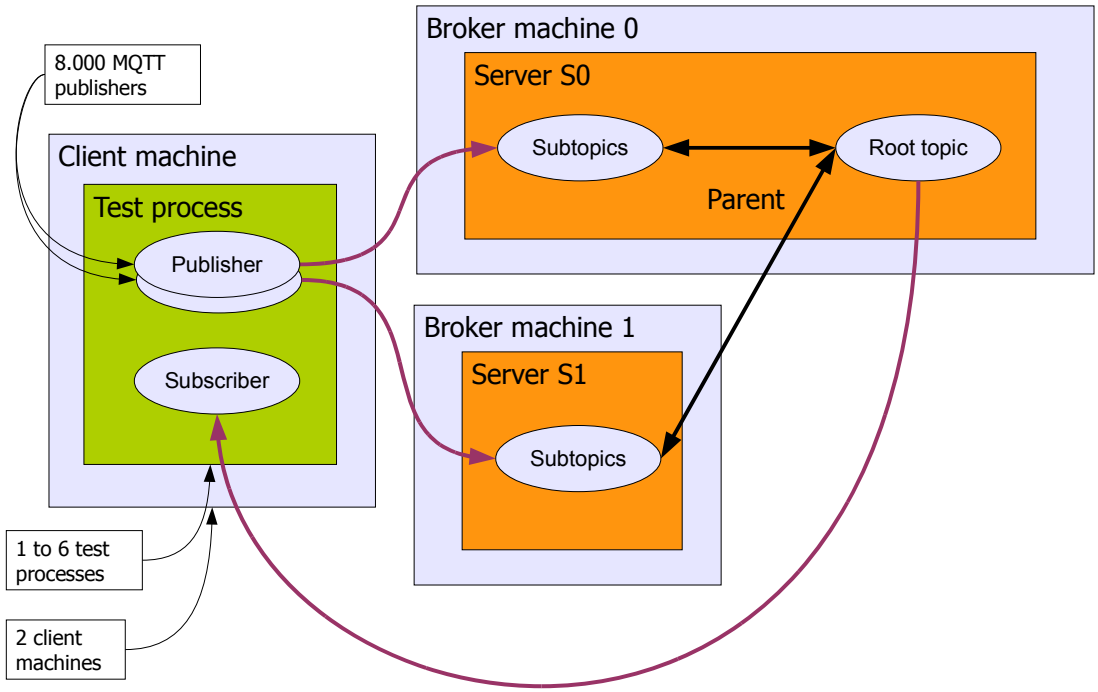

Figure 12: Distributed broker testing, multi-publisher scenario

5 Multi-publisher scenario

5.1 Overview

This scenario simulates a large number of devices, for example smart meters, publishing telemetry data to a central system. The devices are the publishers and the central system is the subscriber[8].

Devices are structured as a 3-level tree. The top level represents the system, for example an electrical distribution circuit. 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.

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

The test scenario defines a fixed size topic partition made of:

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

Therefore, in the metering use case, a topic partition represents an electrical distribution circuit with 4000 meters, each meter publishing 10 telemetry parameters.

8 The delivery of constant streaming of high volumes of data to a single client is not a normal use case for MQTT. Another delivery mechanism such as JMS or AMQP should be used. However the test avoids the dependence on some other mechanisms or protocols. Moreover, messages received by the subscribers are immediately dropped so there is no need for a load balancing mechanism. Only the cost of the message delivery itself could be balanced across several clients. JoramMQ could deliver the messages using the JMS 2.0 feature called “shared subscription” allowing to load-balance a subscription across several connections.

5.2 Centralised broker

Figure 13 shows how the multi-publisher scenario is tested with a centralised broker. There is only one subscriber per topic partition. This subscriber listens to all the topics of the partition by subscribing to “System/#”.

One publisher is created for every topic “Device”. Each publisher sends messages to the topics “Parameter” below the topic “Device”. In every partition, 4000 publishers send messages to 40.000 topics “Parameter” at a steady rate.

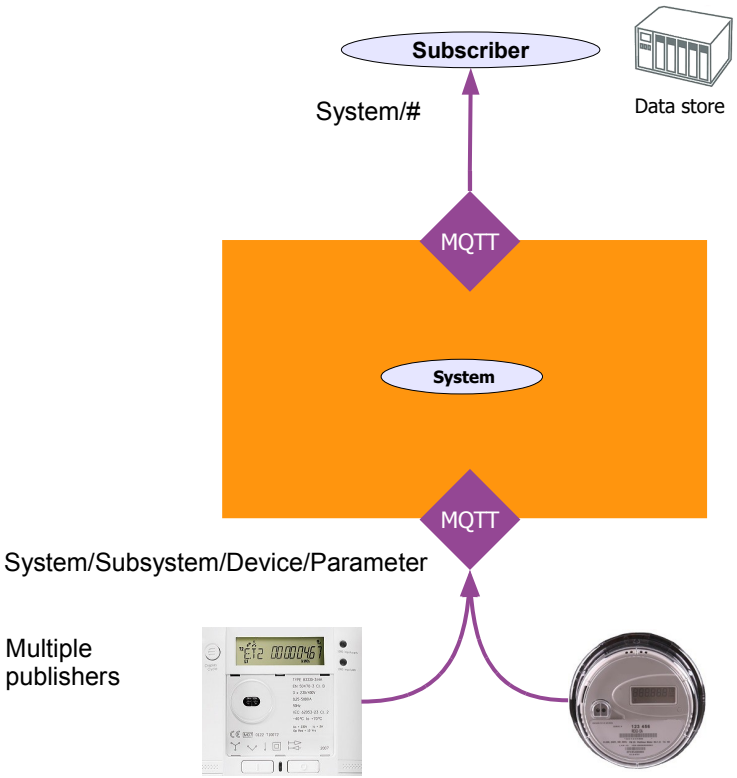

Figure 13: Multi-publisher, centralised broker

QoS 0

Partition configuration			
Produced message rate (msg/s)	**Publisher count**	**Throughput per publisher (msg/s)**	**Expected delivered throughput (msg/s)**
400	4000	0,1	400
4000	4000	1	4000

0,1 msg/s per publisher

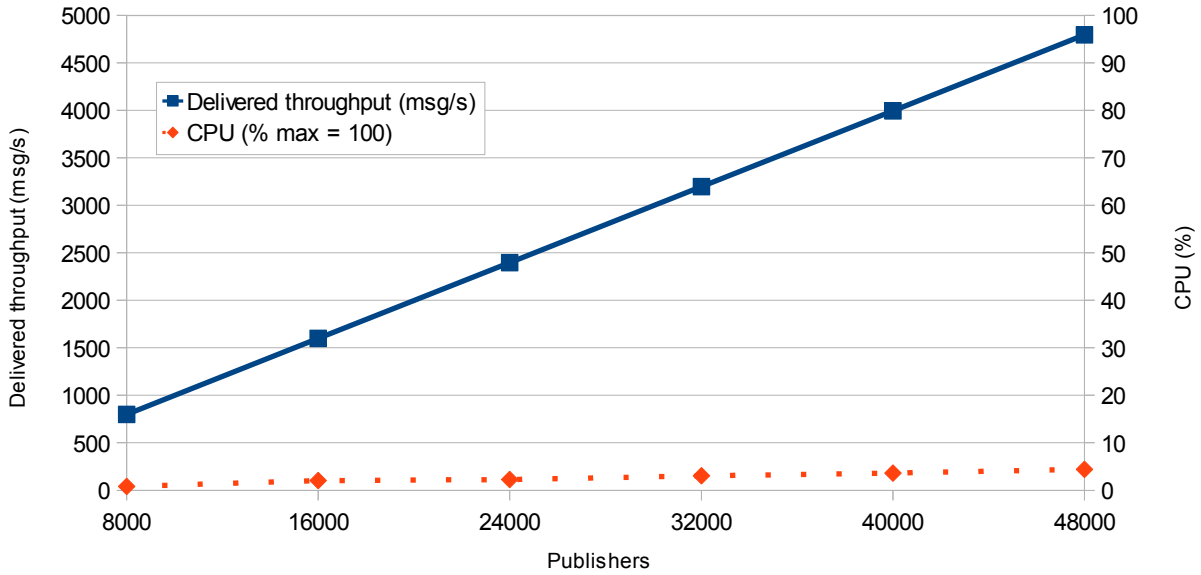

Figure 14: QoS 0 multi-publisher, centralised broker, 0,1 msg/s per publisher

1 msg/s per publisher

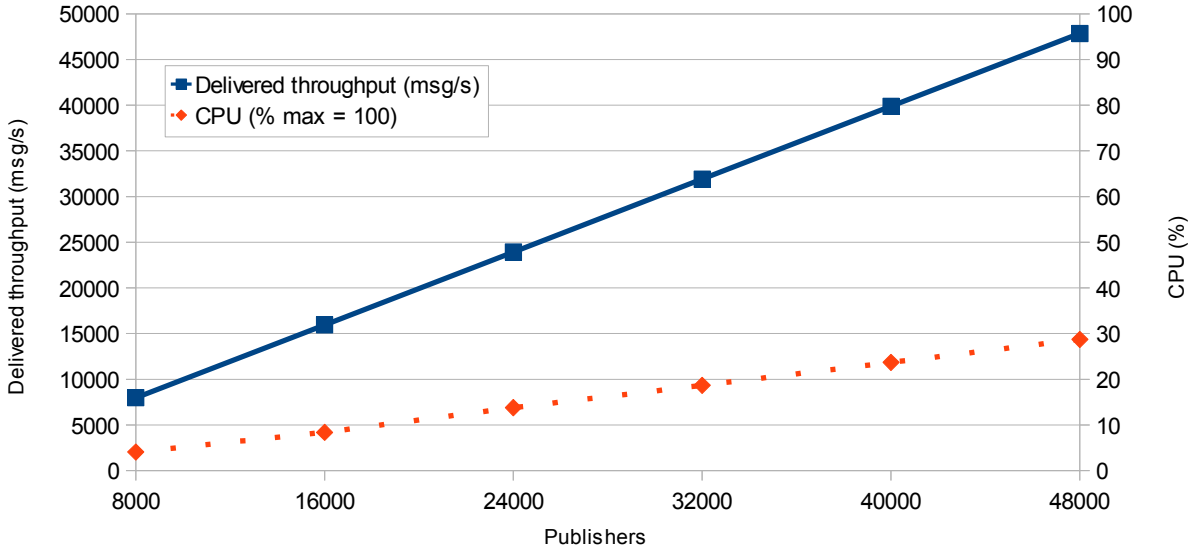

Figure 15: QoS 0, multi-publisher, centralised broker, 1 msg/s per publisher

QoS 1 and clean session False

Partition configuration			
Produced message rate (msg/s)	**Publisher count**	**Throughput per publisher (msg/s)**	**Expected delivered throughput (msg/s)**
400	4000	0,1	400
1000	4000	0,25	1000

0,1 msg/s per publisher

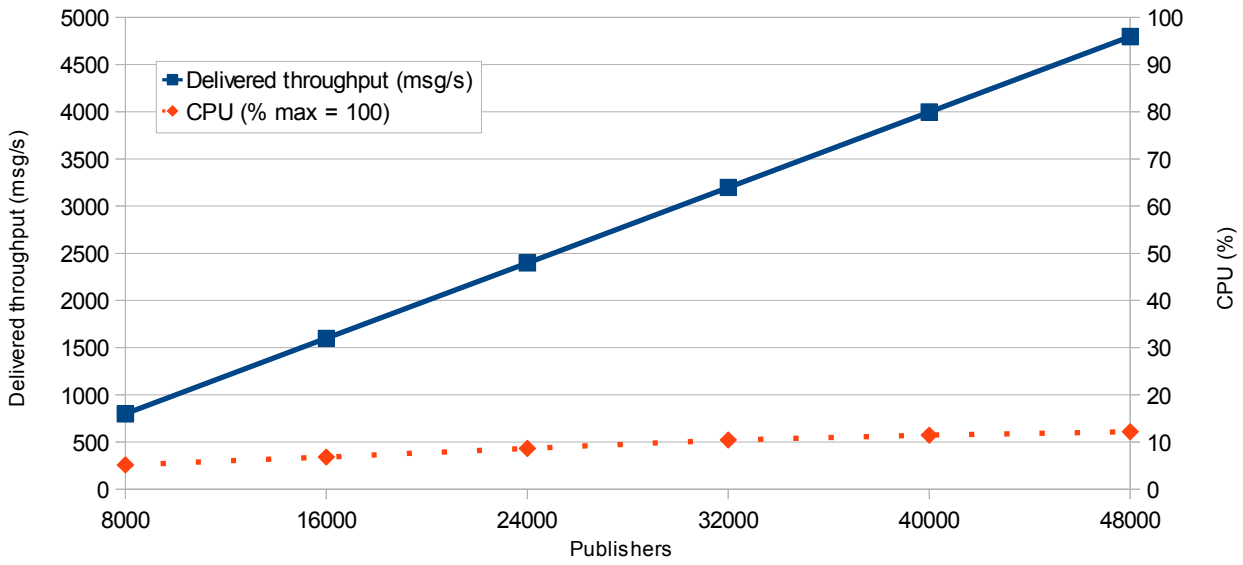

Figure 16: QoS 1 multi-publisher, centralised broker, 0,1 msg/s per publisher

0,25 msg/s per publisher

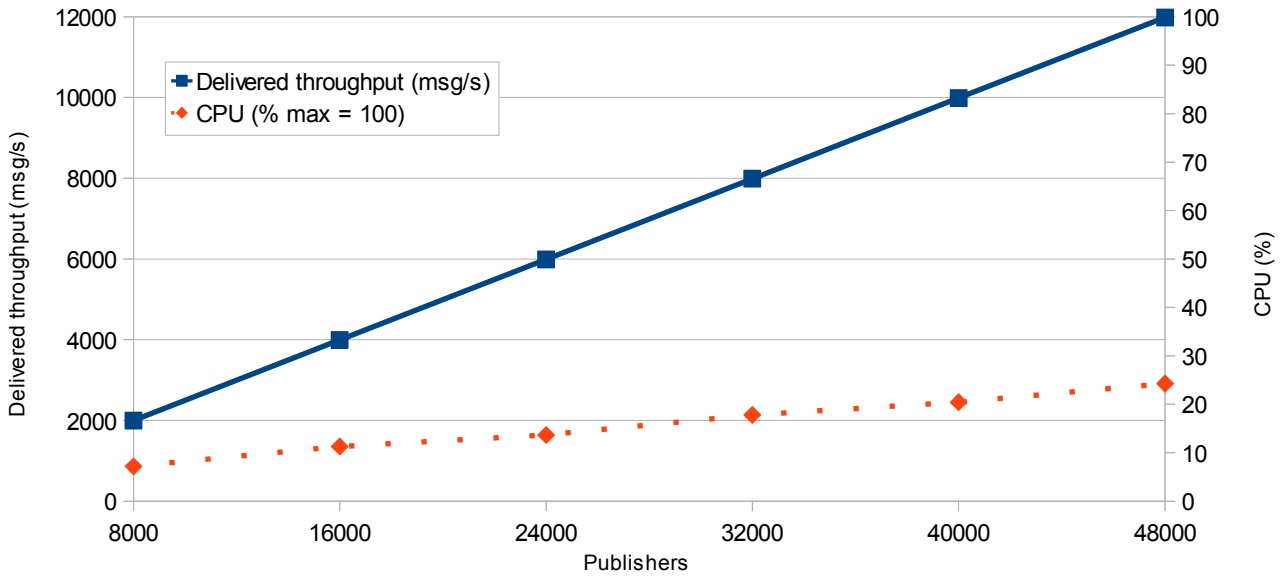

Figure 17: QoS 1, multi-publisher, centralised broker, 0,25 msg/s per publisher

QoS 2 and clean session False

Partition configuration			
Message rate (msg/s)	**Publisher count**	**Throughput per publisher (msg/s)**	**Expected delivered throughput (msg/s)**
280	4000	0,07	280

0,07 msg/s per publisher

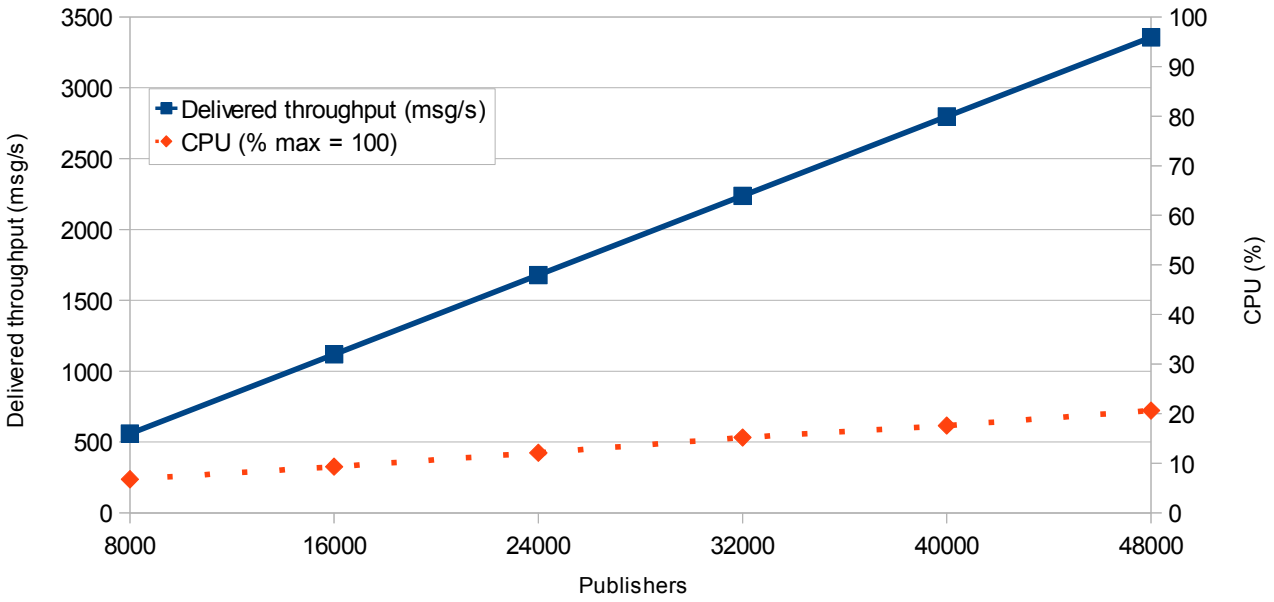

Figure 18: QoS 2 multi-publisher, centralised broker, 0,07 msg/s per publisher

5.3 Clustered broker

The multi-publisher scenario is tested with a clustered broker composed of two servers S0 and S1 as illustrated by figure 19. Publishers (8000 per partition) and subscribers (one per partition) are equally load-balanced across the two servers S0 and S1. Figure 19 only shows one subscriber connected to S0.

The subscriptions are replicated on both servers S0 and S1. If a publisher connected to S1 sends a message to a topic having a subscriber connected to S0, then the message is forwarded to S0.

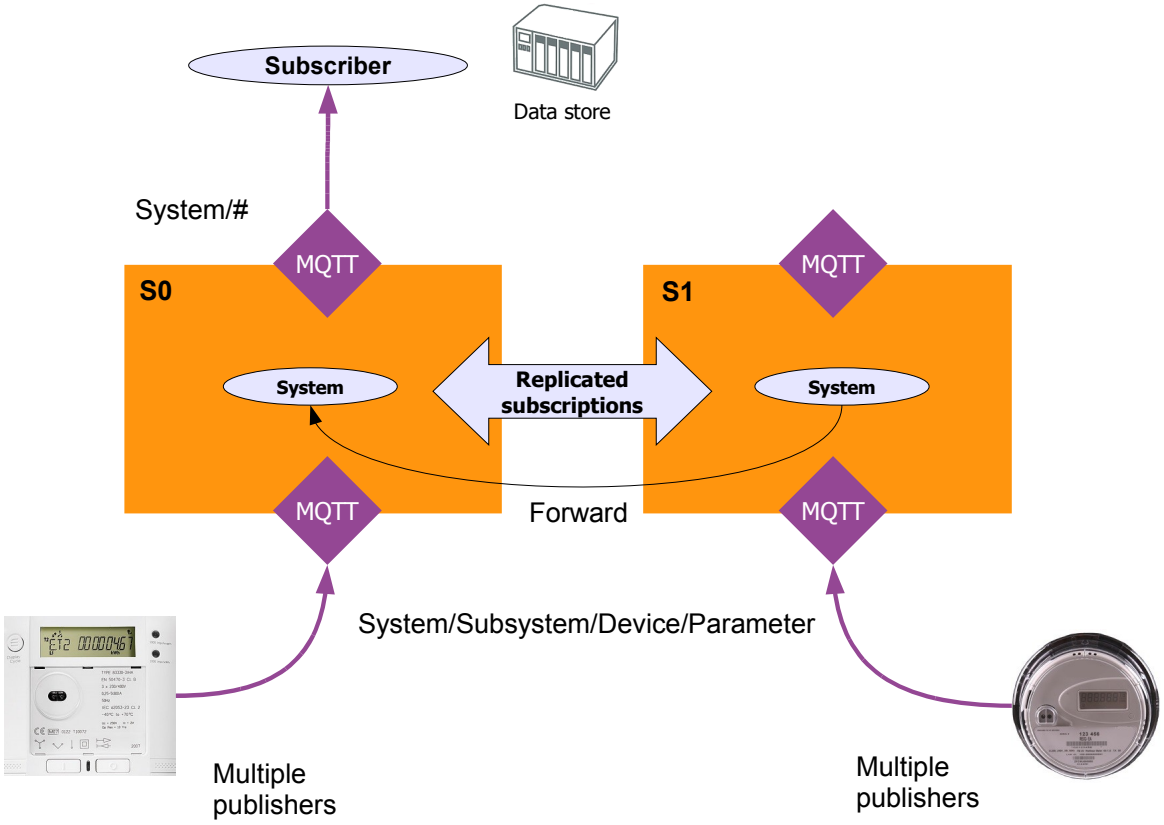

Figure 19: Multi-publisher, clustered broker

QoS 0

Partition configuration			
Produced message rate (msg/s)	**Publisher count**	**Throughput per publisher (msg/s)**	**Expected delivered throughput (msg/s)**
800	8000	0,1	800
8000	8000	1	8000

0,1 msg/s per publisher

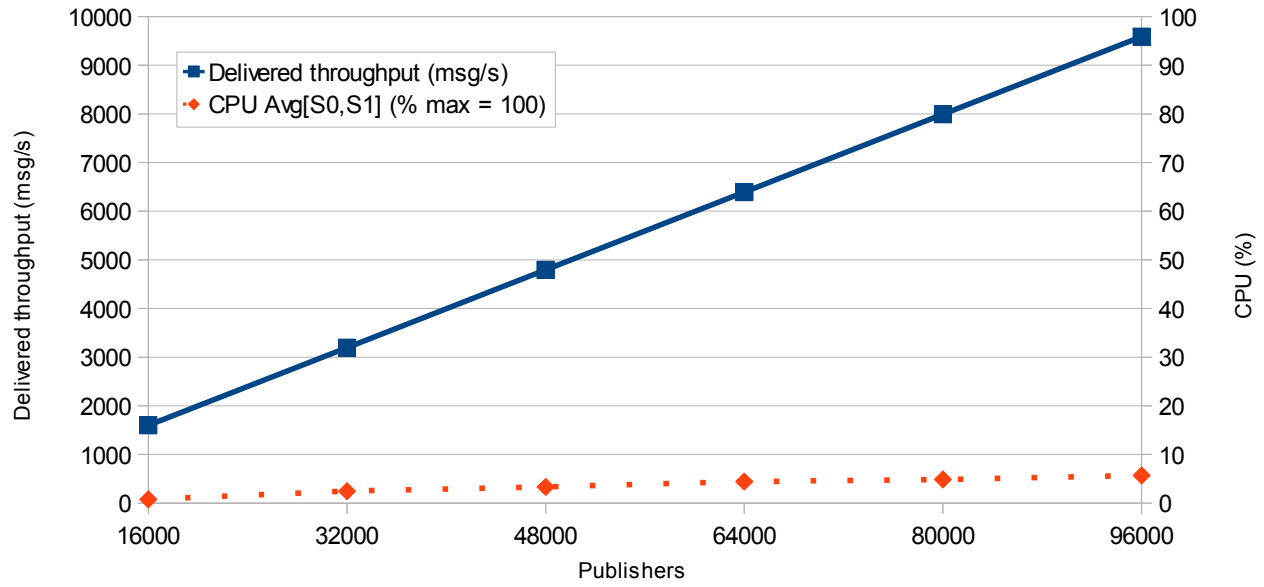

Figure 20: QoS 0, multi-publisher, clustered broker, 0,1 msg/s per publisher

1 msg/s per publisher

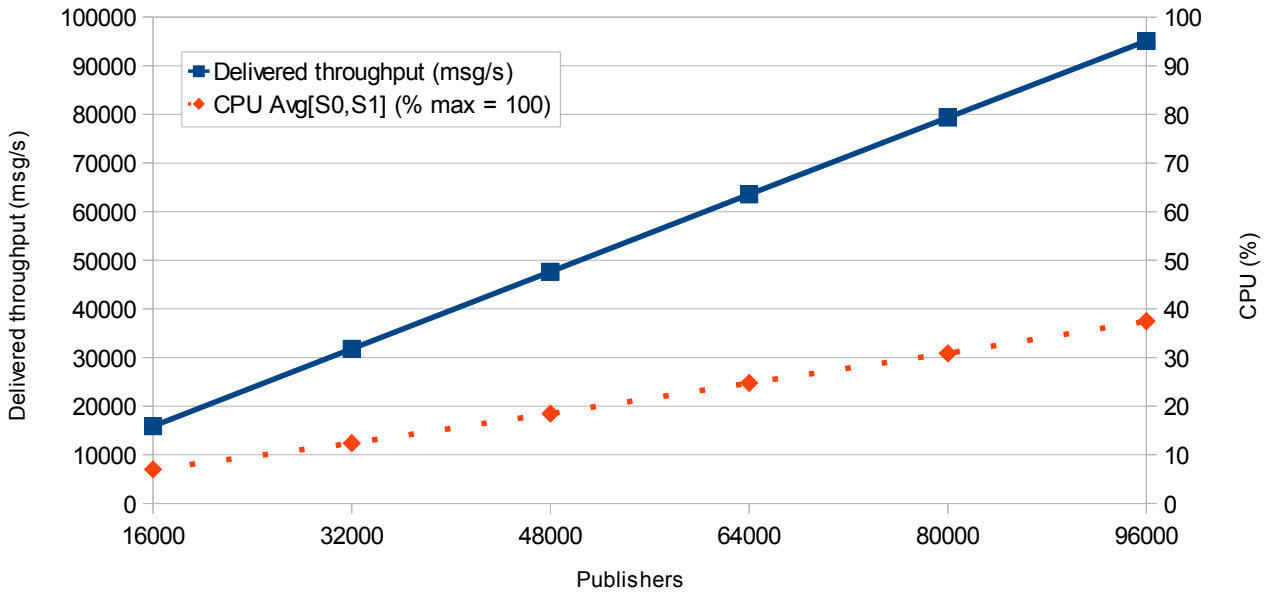

Figure 21: QoS 0, multi-publisher, clustered broker, 1 msg/s per publisher

QoS 1 and clean session False

Partition configuration			
Produced message rate (msg/s)	**Publisher count**	**Throughput per publisher (msg/s)**	**Expected delivered throughput (msg/s)**
800	8000	0,1	800
2000	8000	0,25	2000

0,1 msg/s per publisher

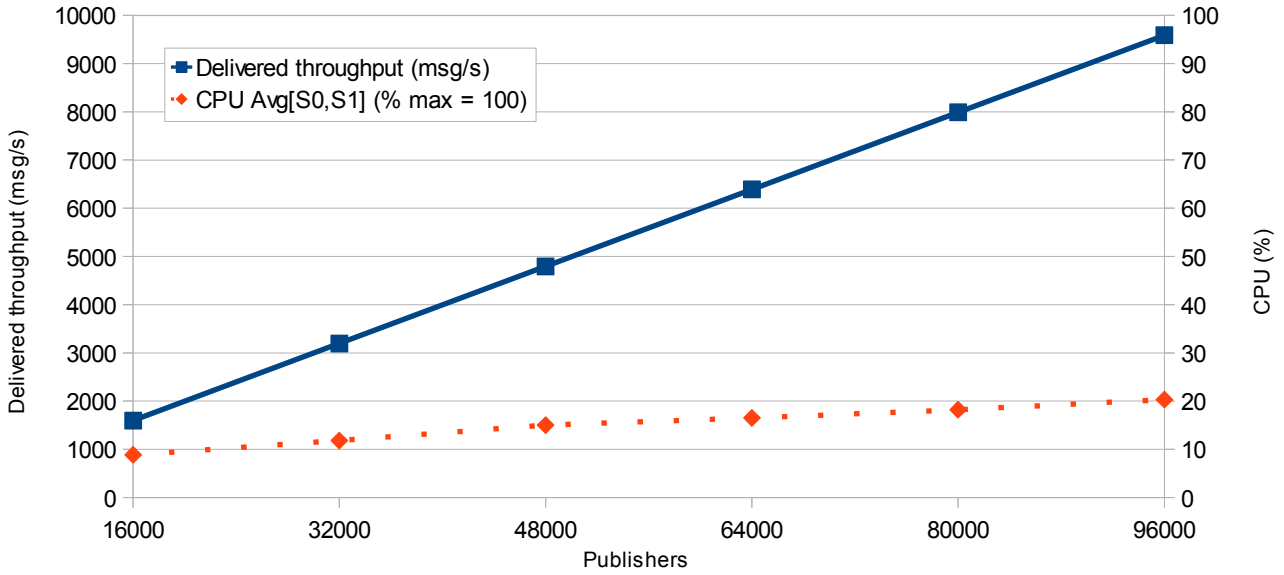

Figure 22: QoS 1, multi-publisher, clustered broker, 0,1 msg/s per partition

0,25 msg/s per publisher

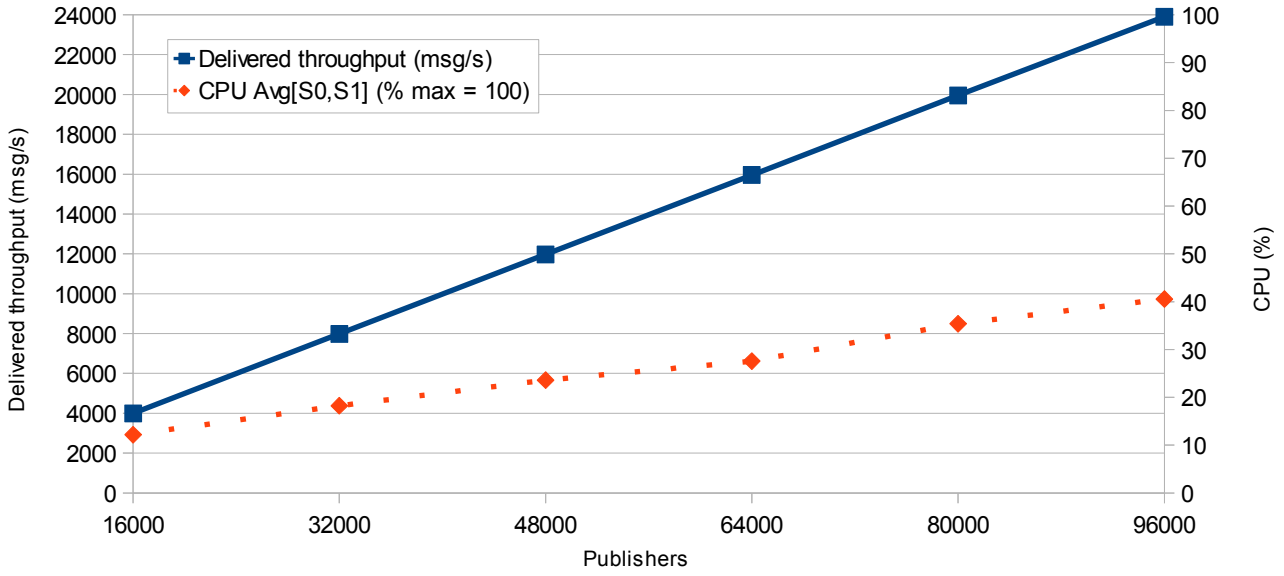

Figure 23: QoS 1, multi-publisher, clustered broker, 0,25 msg/s per partition

QoS 2 and clean session False

Partition configuration			
Produced message rate (msg/s)	**Publisher count**	**Throughput per publisher (msg/s)**	**Expected delivered throughput (msg/s)**
560	8000	0,07	560

0,07 msg/s per publisher

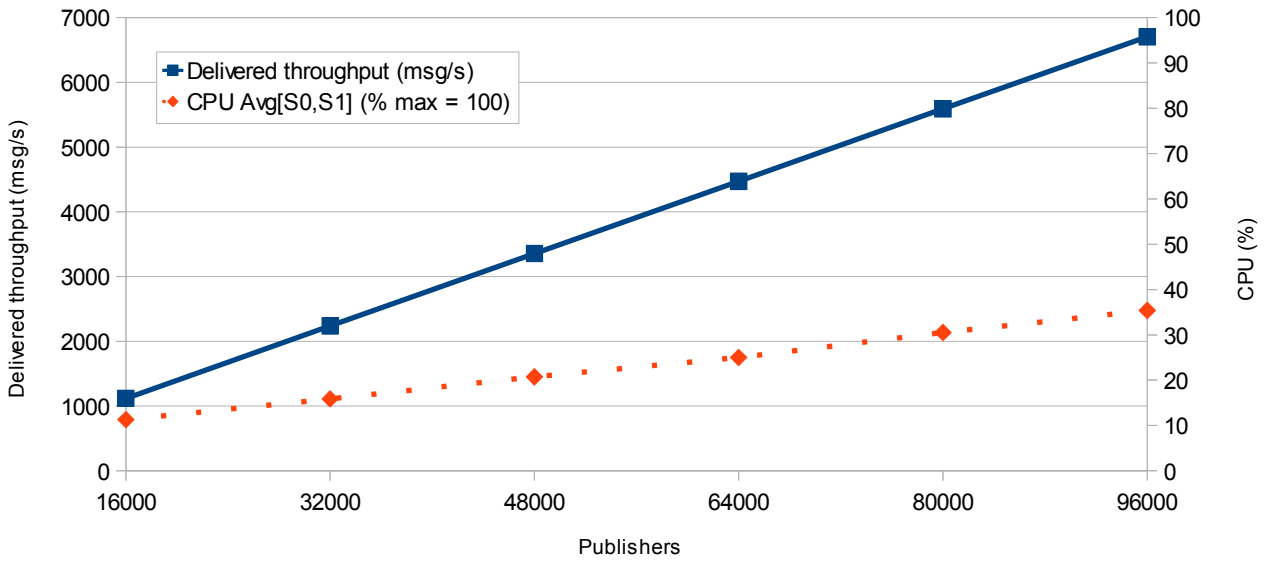

Figure 24: QoS 2, multi-publisher, clustered broker, 0,07 msg/s per publisher

5.4 Distributed broker

The multi-publisher scenario is tested with a distributed broker composed of two servers S0 and S1 as illustrated by figure 25. Publishers (8000 per partition) and subscribers (one per partition) are equally load-balanced across the two servers S0 and S1. Figure 25 only shows one root topic “System” and one subscriber connected to S0.

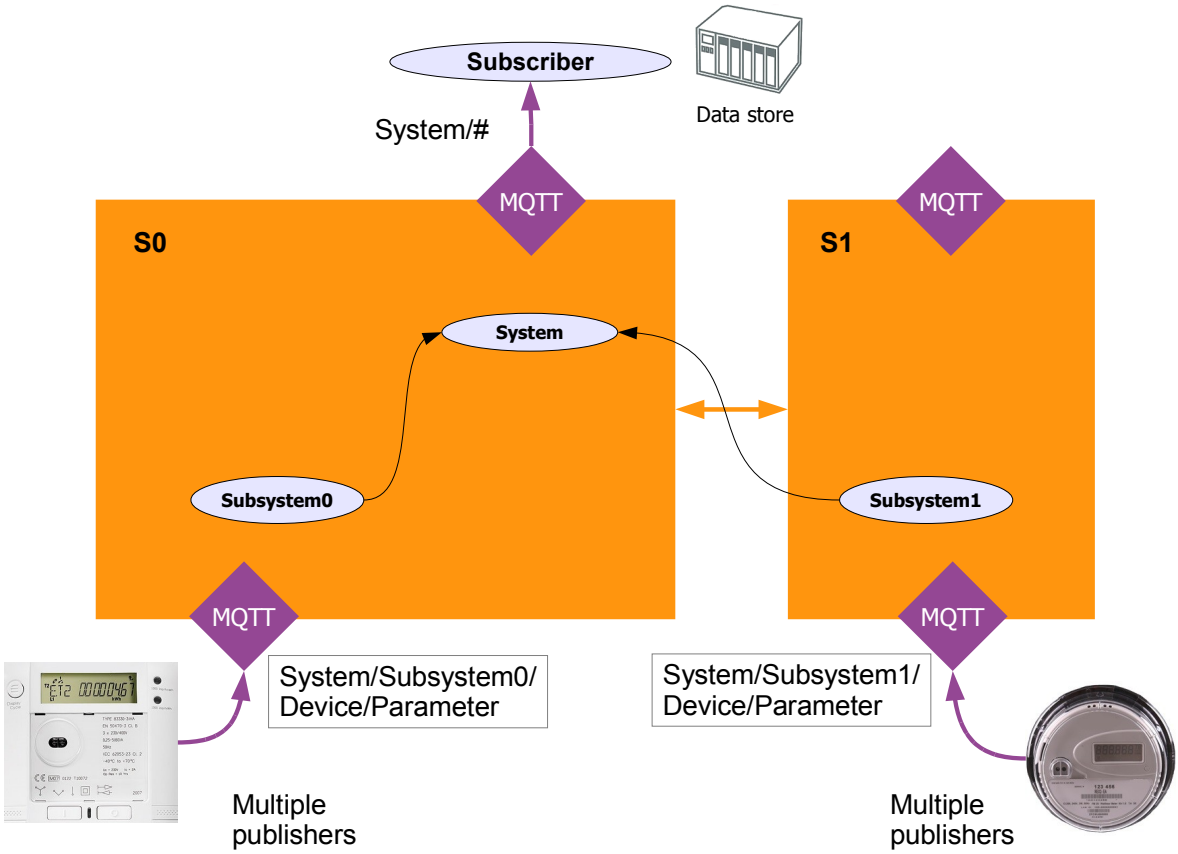

Figure 25: Multi-publisher, distributed broker

QoS 0

Partition configuration			
Produced message rate (msg/s)	**Publisher count**	**Throughput per publisher (msg/s)**	**Expected delivered throughput (msg/s)**
800	8000	0,1	800
8000	8000	1	8000

0,1 msg/s per publisher

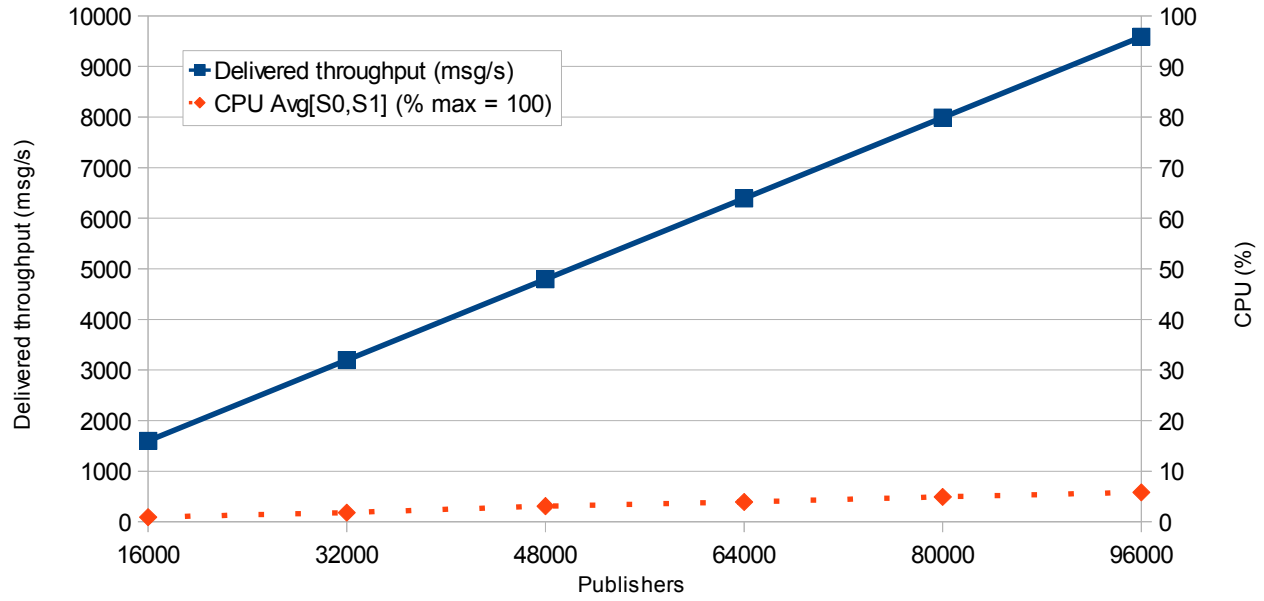

Figure 26: QoS 0, multi-publisher, distributed broker, 0,1 msg/s per publisher

1 msg/s per publisher

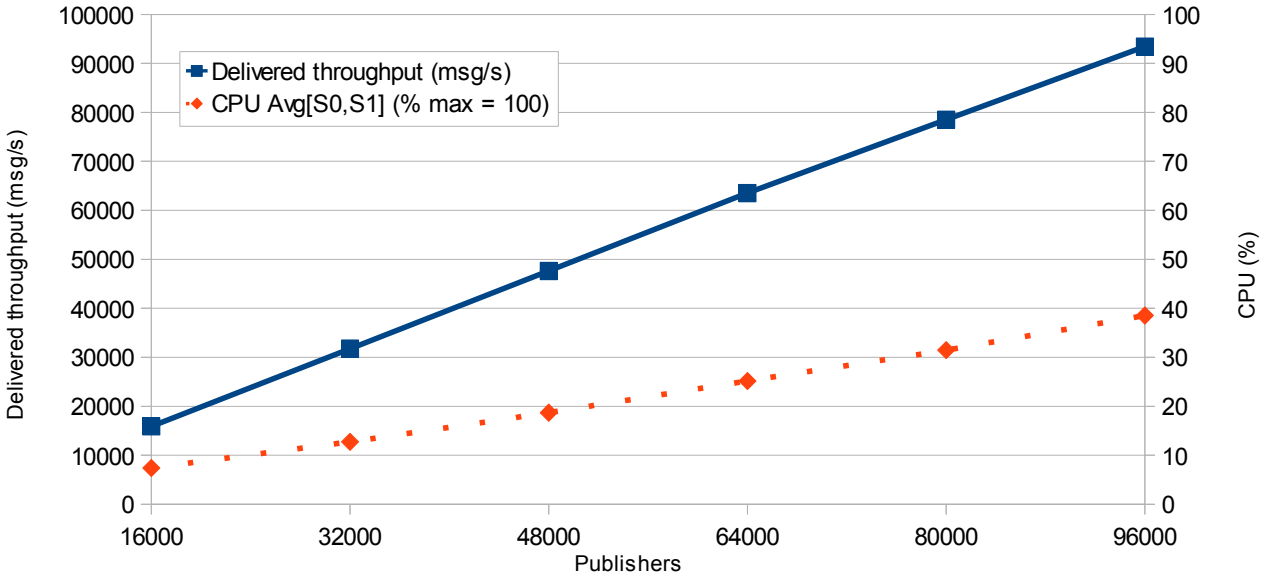

Figure 27: QoS 0, multi-publisher, distributed broker, 1 msg/s per publisher

QoS 1 and clean session False

Partition configuration			
Produced message rate (msg/s)	**Publisher count**	**Throughput per publisher (msg/s)**	**Expected delivered throughput (msg/s)**
800	8000	0,1	800
2000	8000	0,25	2000

0,1 msg/s per publisher

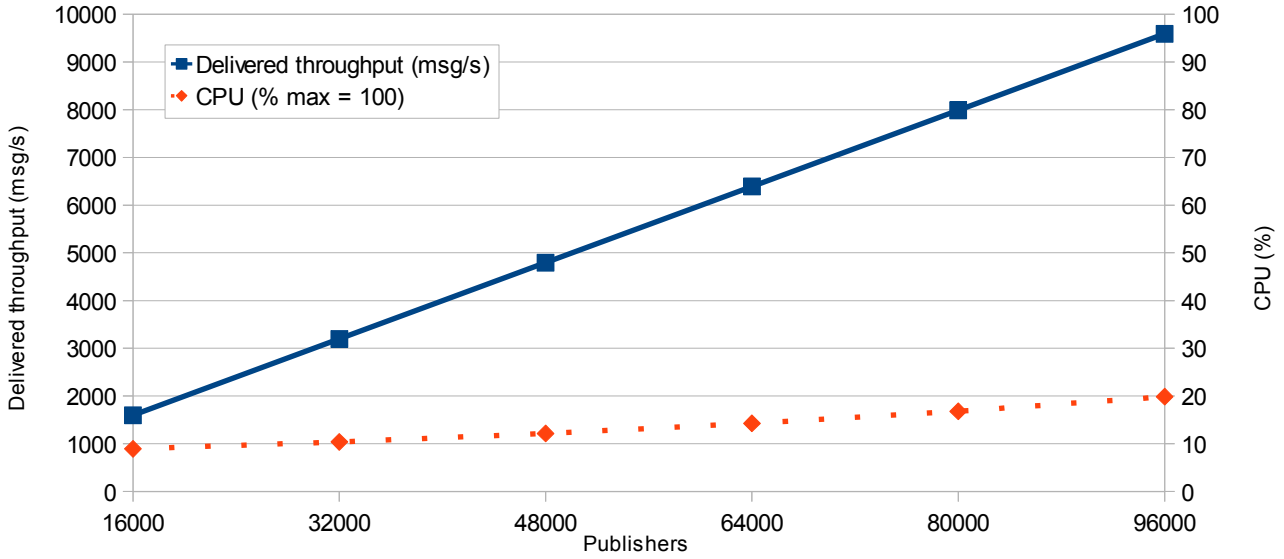

Figure 28: QoS 1, multi-publisher, distributed broker, 0,1 msg/s per publisher

0,25 msg/s per publisher

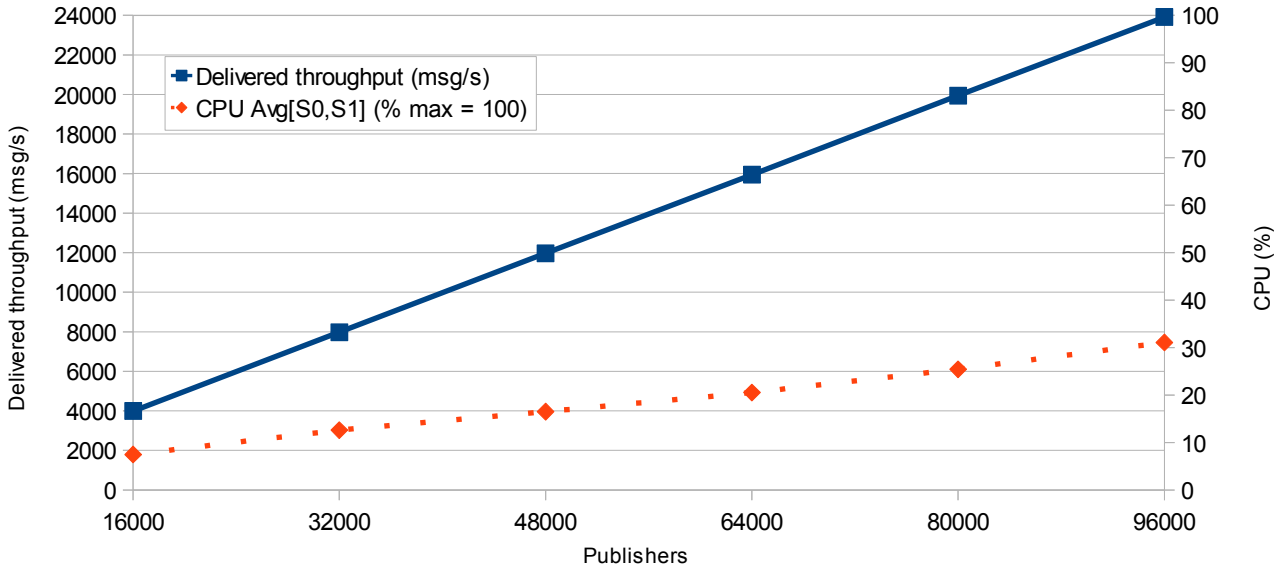

Figure 29: QoS 1, multi-publisher, distributed broker, 0,25 msg/s per publisher

QoS 2 and clean session False

Partition configuration			
Produced message rate (msg/s)	**Publisher count**	**Throughput per publisher (msg/s)**	**Expected delivered throughput (msg/s)**
560	8000	0,07	560

0,07 msg/s per publisher

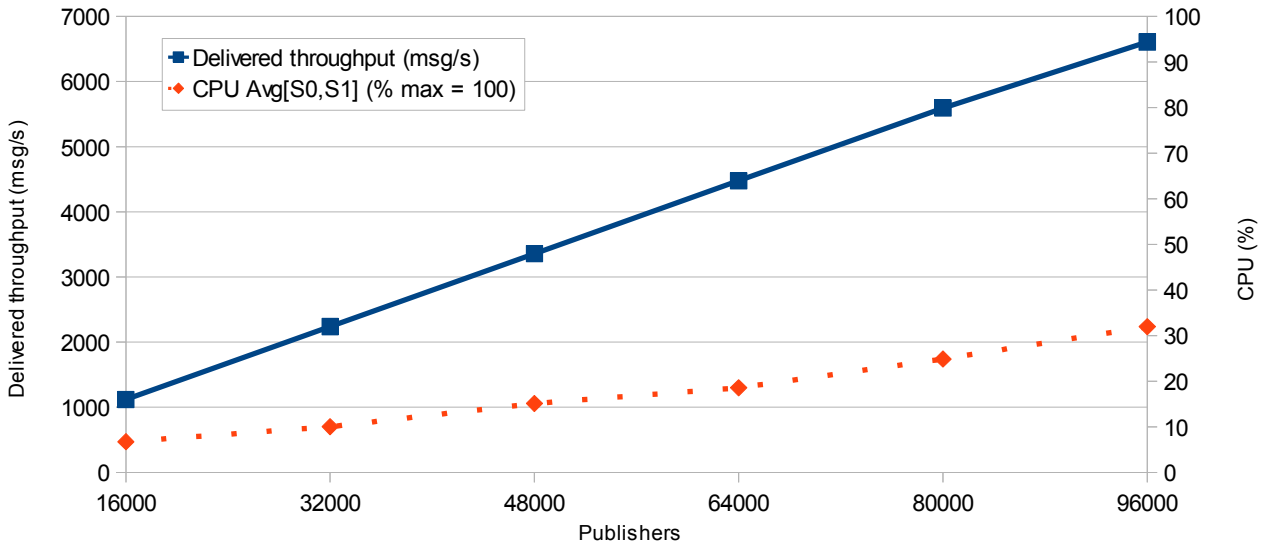

Figure 30: QoS 2, multi-publisher, distributed broker, 0,07 msg/s per publisher

5.5 Conclusion

The scenario called “multi-publisher” simulates a large number of devices, for example smart meters, publishing telemetry data to a central system. The devices are the publishers and the central system is the subscriber. Using a single server and constrained physical resources (2 cores, 4 GB RAM and SATA disk) JoramMQ can scale to 48.000 publishers, each publisher producing:

- 1 message per second at QoS 0
- 0,25 message per second at QoS 1
- 0,07 message per second at QoS 2

With a clustered broker topology including two servers, JoramMQ scales to twice the number of publishers, i.e. 96.000, producing messages at the same rates as above.

The distributed broker topology approximately gives the same results as the clustered topology.

6 Multi-subscriber scenario

6.1 Overview

This scenario simulates a large number of devices, for example smart meters, controlled by a central system. The meters are the subscribers and the central system is the publisher.

Smart meters provide two way communications, allowing commands to be sent towards the devices, for example a remote ON/OFF switch, or a request/response operation to retrieve the value of a parameter. This test scenario only simulates the command (the request), not the potential response. The command response would be notified as a telemetry parameter, for example a status update indicating that the switch is ON.

Topics are partitioned in several hierarchies in the same way as in the multi-publisher scenario described in section 5. A partition represents a circuit of 4000 devices providing a control interface.

The test scenario defines a fixed size topic partition made of:

- 1 root topic “System”
- 40 topics “Subsystem”
- 100 topics “Device” per subsystem
- 1 topic “Command” per device

6.2 Centralised broker

Figure 31 shows how the multi-subscriber scenario is tested with a centralised broker. There is only one publisher per topic partition. This publisher sends messages to all the topics of the partition at a steady rate.

One subscriber is created for every topic “Device”. Each subscriber receives messages from the topic “Command” below the topic “Device”. In every partition, 4000 subscribers receive messages from 4000 topics “Command” at a steady rate.

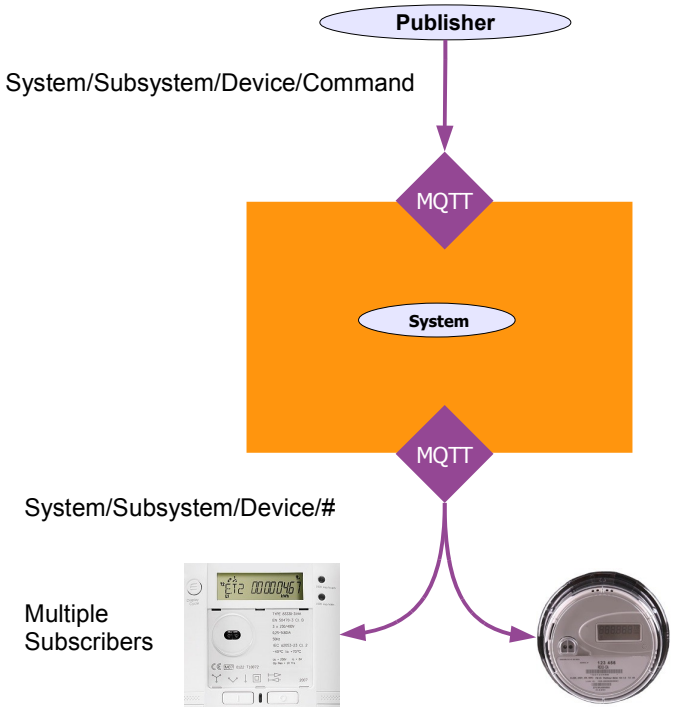

Figure 31: Multi-subscriber, centralised broker

QoS 0

Partition configuration			
Message rate (msg/s)	**Command topic count**	**Throughput per topic (msg/s)**	**Expected delivered throughput per subscriber (msg/s)**
400	4000	0,1	0,1
2400	4000	0,6	0,6

0,1 msg/s per subscriber

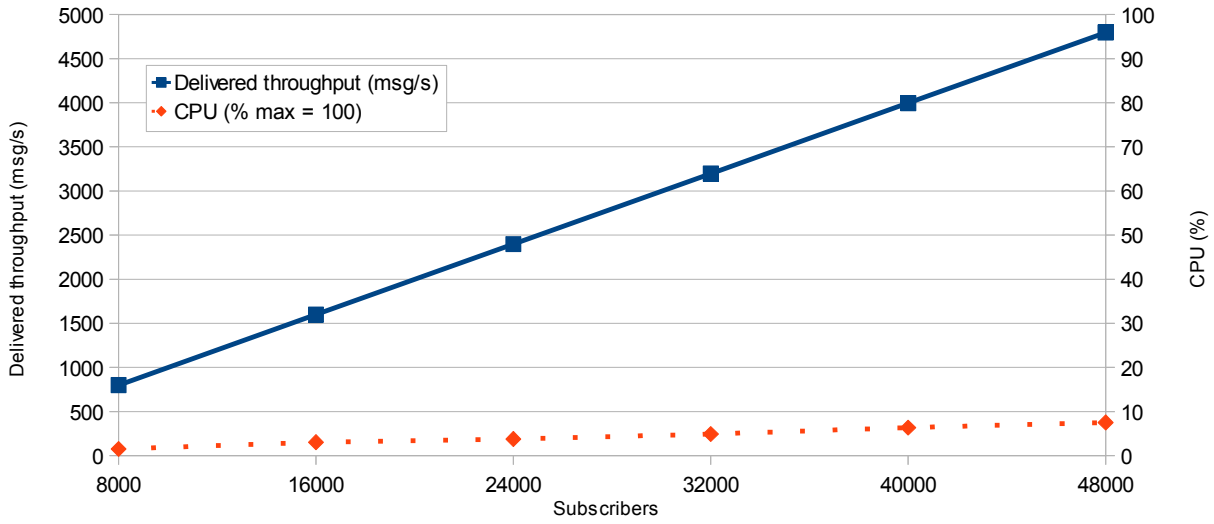

Figure 32: QoS 0, multi-subscriber, centralised broker, 0,1 msg/s per subscriber

0,6 msg/s per subscriber

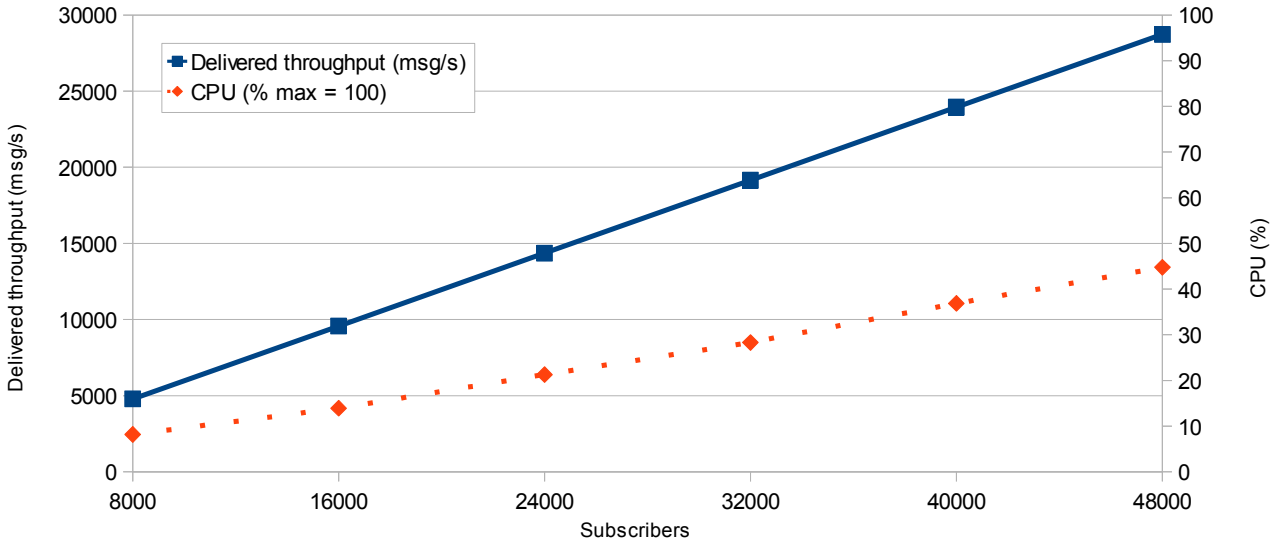

Figure 33: QoS 0, multi-subscriber, centralised broker, 0,6 msg/s per subscriber

QoS 1 and clean session False

Partition configuration			
Message rate (msg/s)	**Command topic count**	**Throughput per topic (msg/s)**	**Expected delivered throughput per subscriber (msg/s)**
400	4000	0,1	0,1
600	4000	0,15	0,15

0,1 msg/s per subscriber

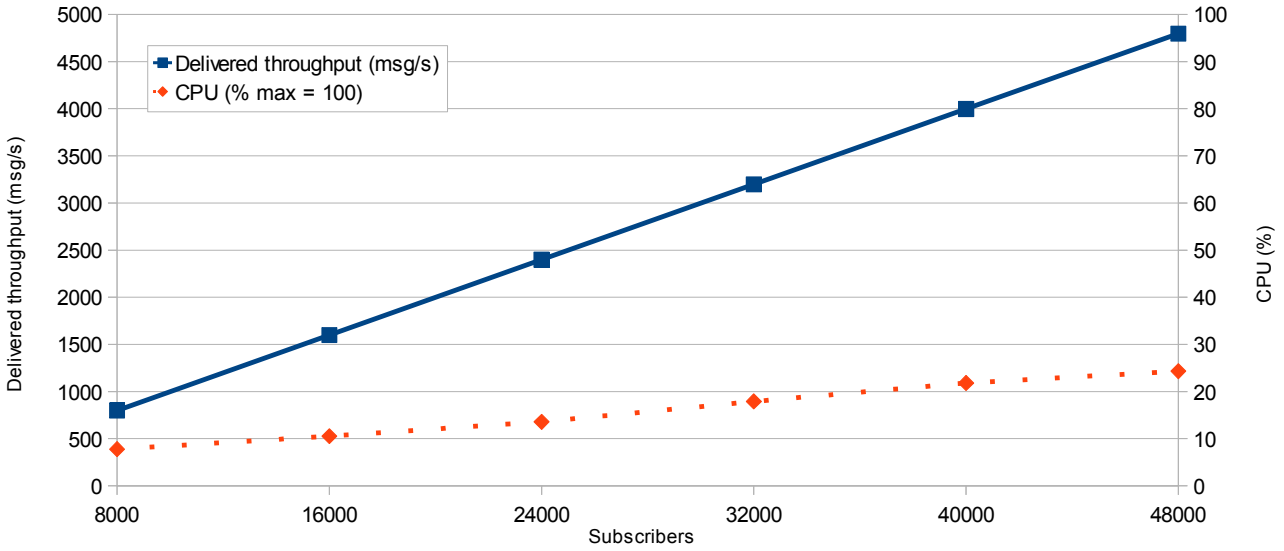

Figure 34: QoS 1, multi-subscriber, centralised broker, 0,1 msg/s per subscriber

0,15 msg/s per subscriber

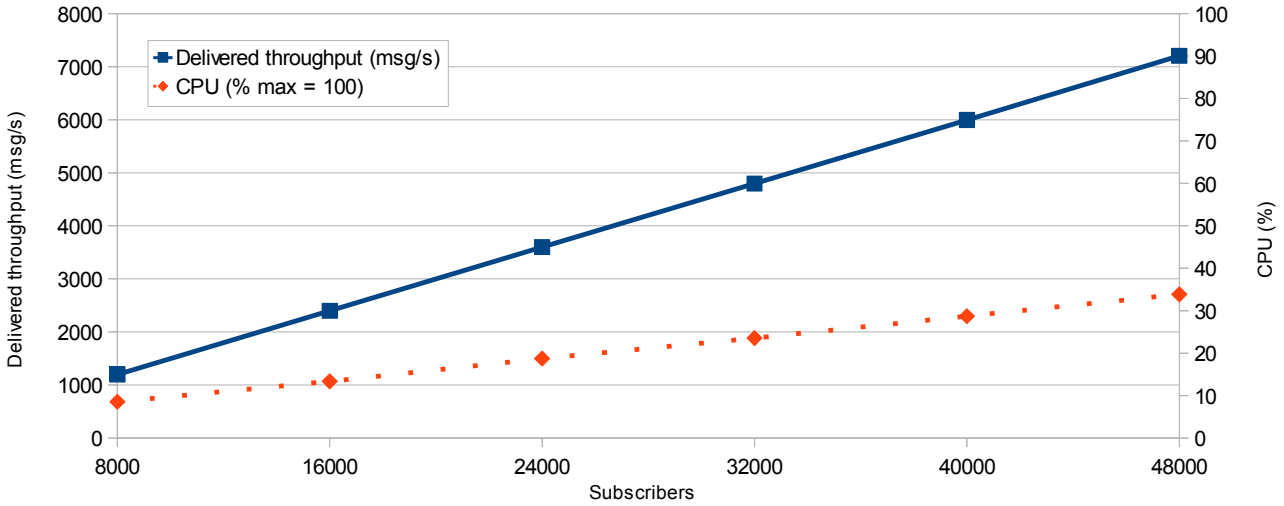

Figure 35: QoS 1, multi-subscriber, centralised broker, 0,15 msg/s per subscriber

QoS 2 and clean session False

Partition configuration			
Message rate (msg/s)	**Command topic count**	**Throughput per topic (msg/s)**	**Expected delivered throughput per subscriber (msg/s)**
280	4000	0,07	0,07

0,07 msg/s per subscriber

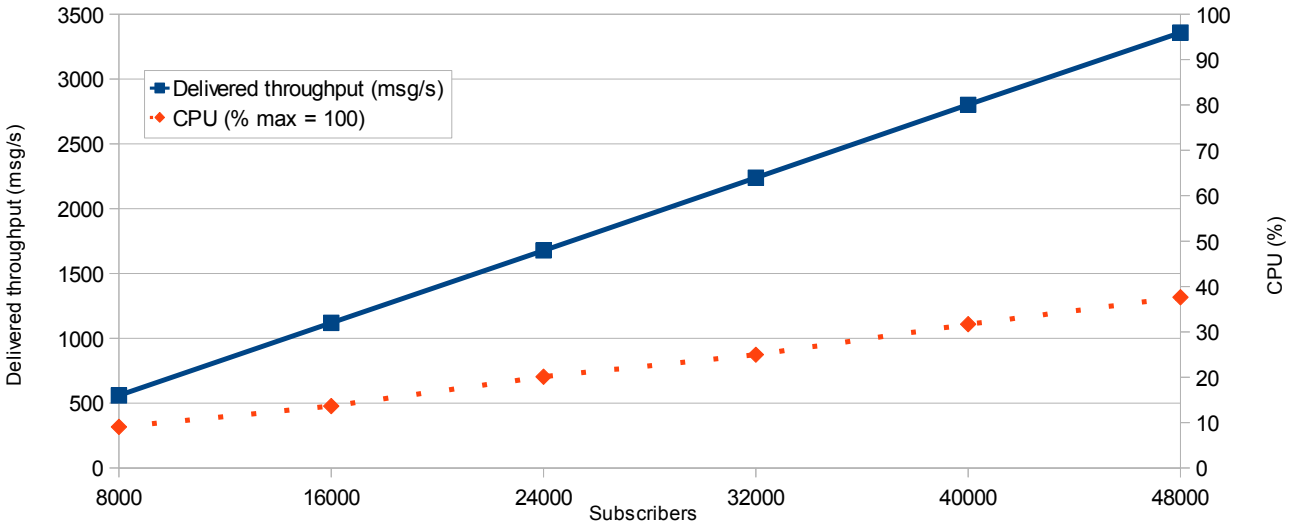

Figure 36: QoS 2, multi-subscriber, centralised broker, 0,07 msg/s per subscriber

6.3 Clustered broker

The multi-subscriber scenario is tested with a clustered broker composed of two servers S0 and S1 as illustrated by figure 37. Subscribers (8000 per partition) and publishers (one per partition) are equally load-balanced across the two servers S0 and S1. Figure 37 only shows one publisher connected to S0.

The subscriptions are replicated on both servers S0 and S1. If a publisher connected to S0 sends a message to a topic having a subscriber connected to S1, then the message is forwarded to S1.

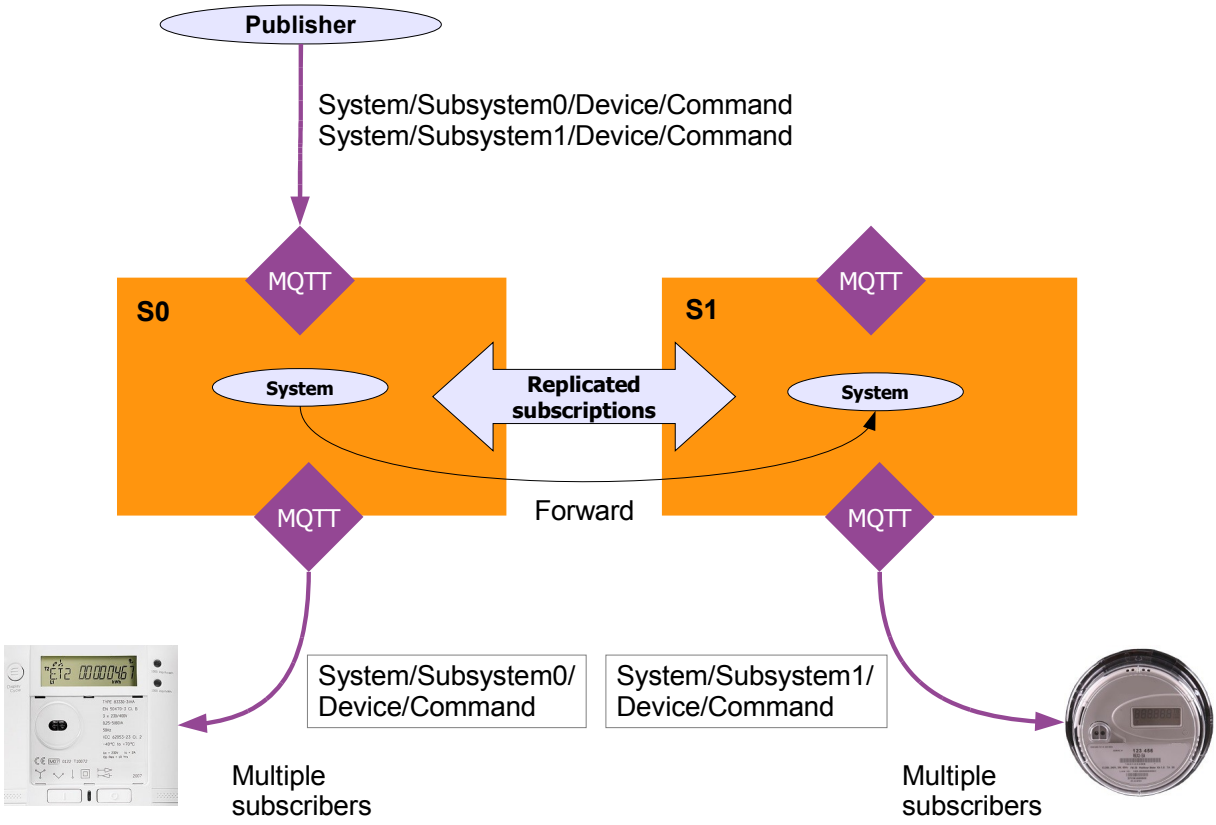

Figure 37: Multi-subscriber, clustered broker

QoS 0

Partition configuration			
Message rate (msg/s)	**Command topic count**	**Throughput per topic (msg/s)**	**Expected delivered throughput per subscriber (msg/s)**
800	8000	0,1	0,1
4800	8000	0,6	0,6

0,1 msg/s per subscriber

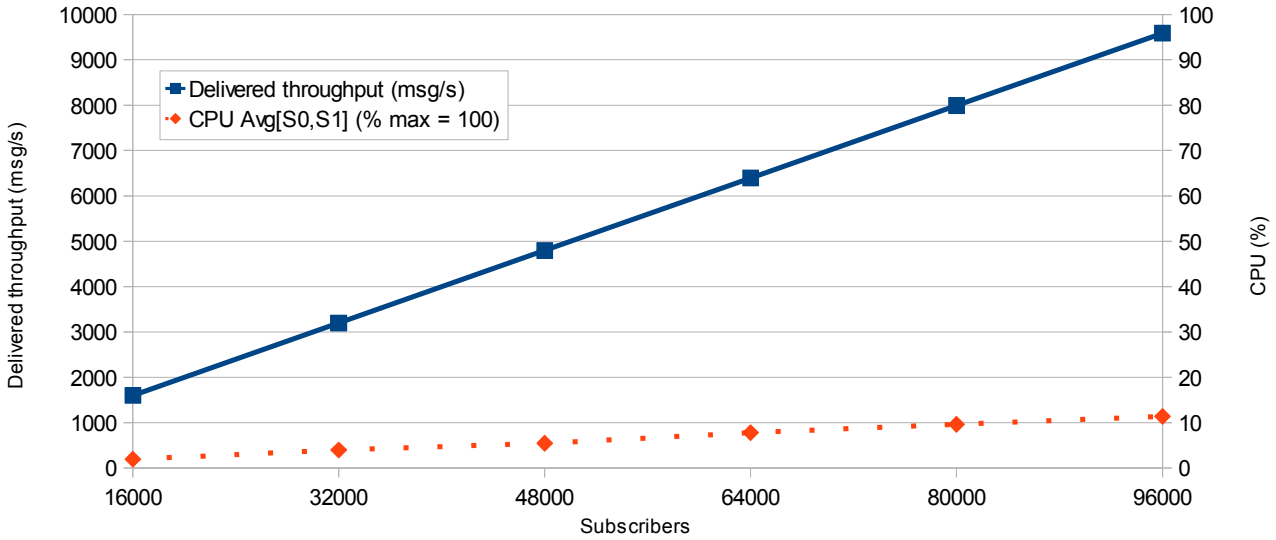

Figure 38: QoS 0, multi-subscriber, clustered broker, 0,1 msg/s per subscriber

0,6 msg/s per subscriber

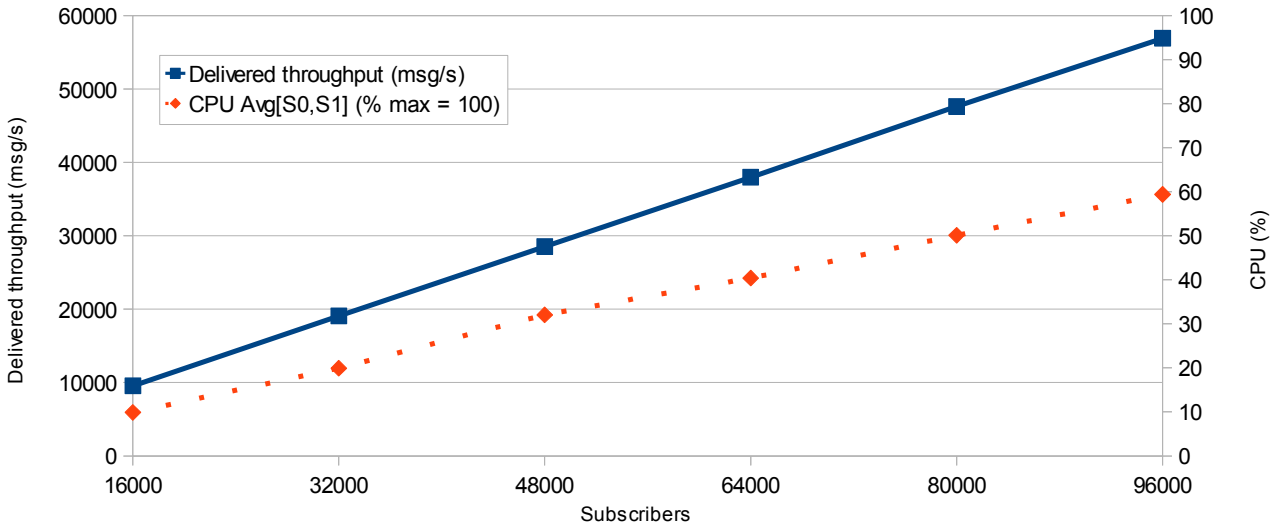

Figure 39: QoS 0, multi-subscriber, clustered broker, 0,6 msg/s per subscriber

QoS 1 and clean session False

Partition configuration			
Message rate (msg/s)	**Command topic count**	**Throughput per topic (msg/s)**	**Expected delivered throughput per subscriber (msg/s)**
800	8000	0,1	0,1
1200	8000	0,15	0,15

0,1 msg/s per subscriber

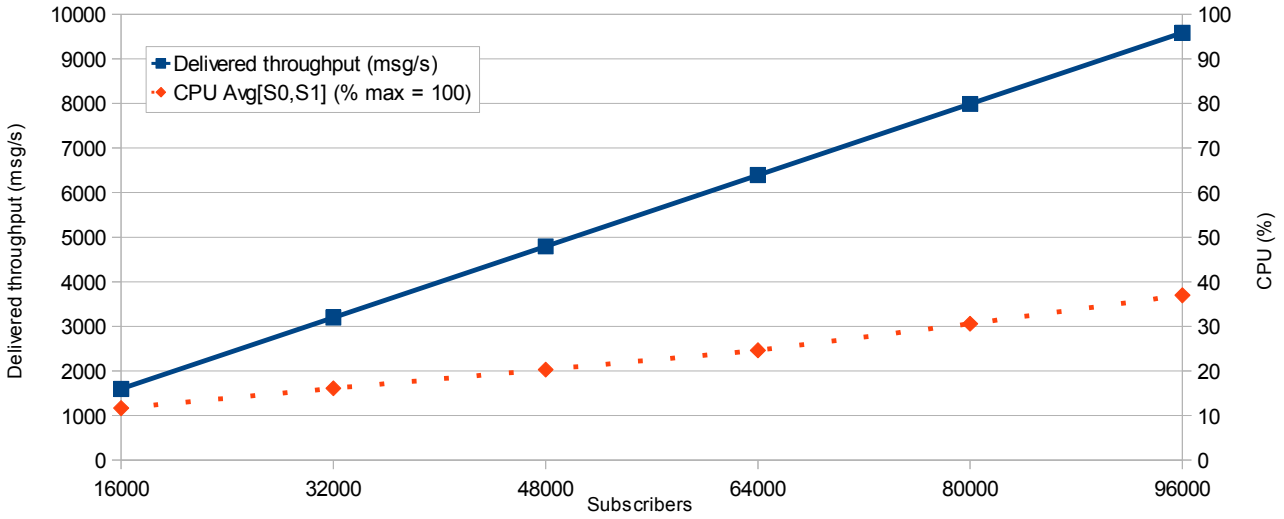

Figure 40: QoS 1, multi-subscriber, clustered broker, 0,1 msg/s per subscriber

0,15 msg/s per subscriber

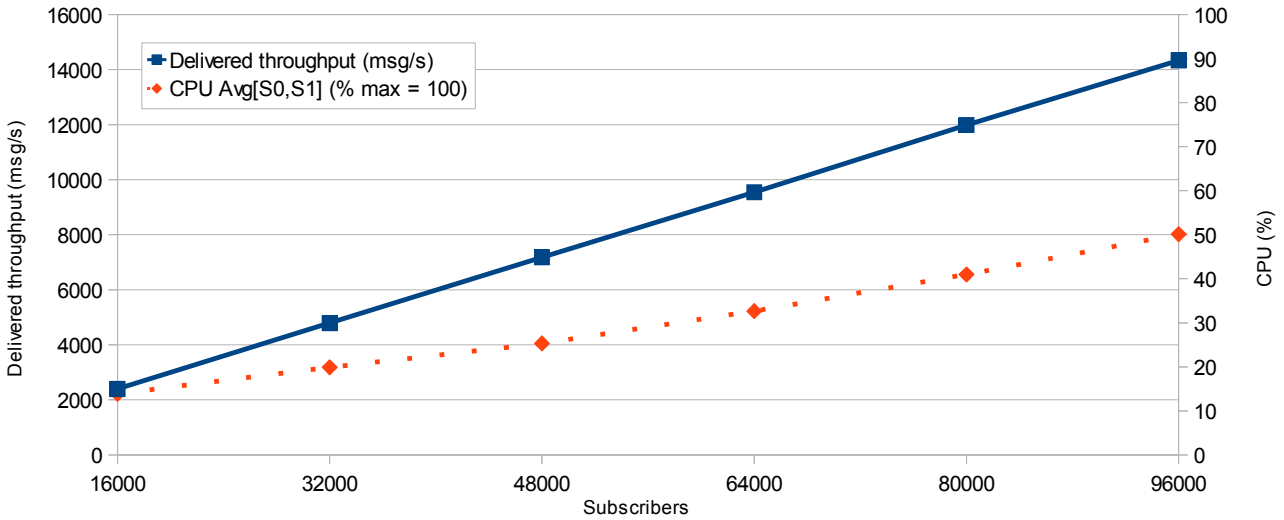

Figure 41: QoS 1, multi-subscriber, clustered broker, 0,15 msg/s per subscriber

QoS 2 and clean session False

Partition configuration			
Message rate (msg/s)	**Command topic count**	**Throughput per topic (msg/s)**	**Expected delivered throughput per subscriber (msg/s)**
320	8000	0,04	0,04

0,04 msg/s per subscriber

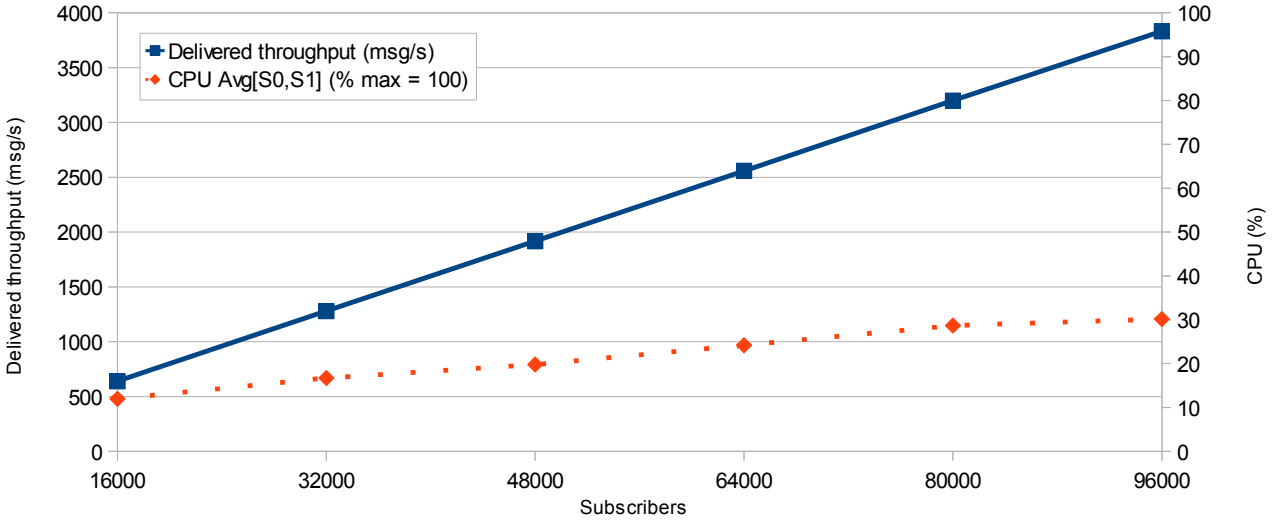

Figure 42: QoS 2, multi-subscriber, clustered broker, 0,04 msg/s per subscriber

6.4 Distributed broker

The multi-subscriber scenario is tested with a distributed broker composed of two servers S0 and S1 as illustrated by figure 43. Subscribers (8000 per partition) and publishers (one per partition) are equally load-balanced across the two servers S0 and S1. Figure 43 only shows one root topic “System” and one publisher connected to S0.

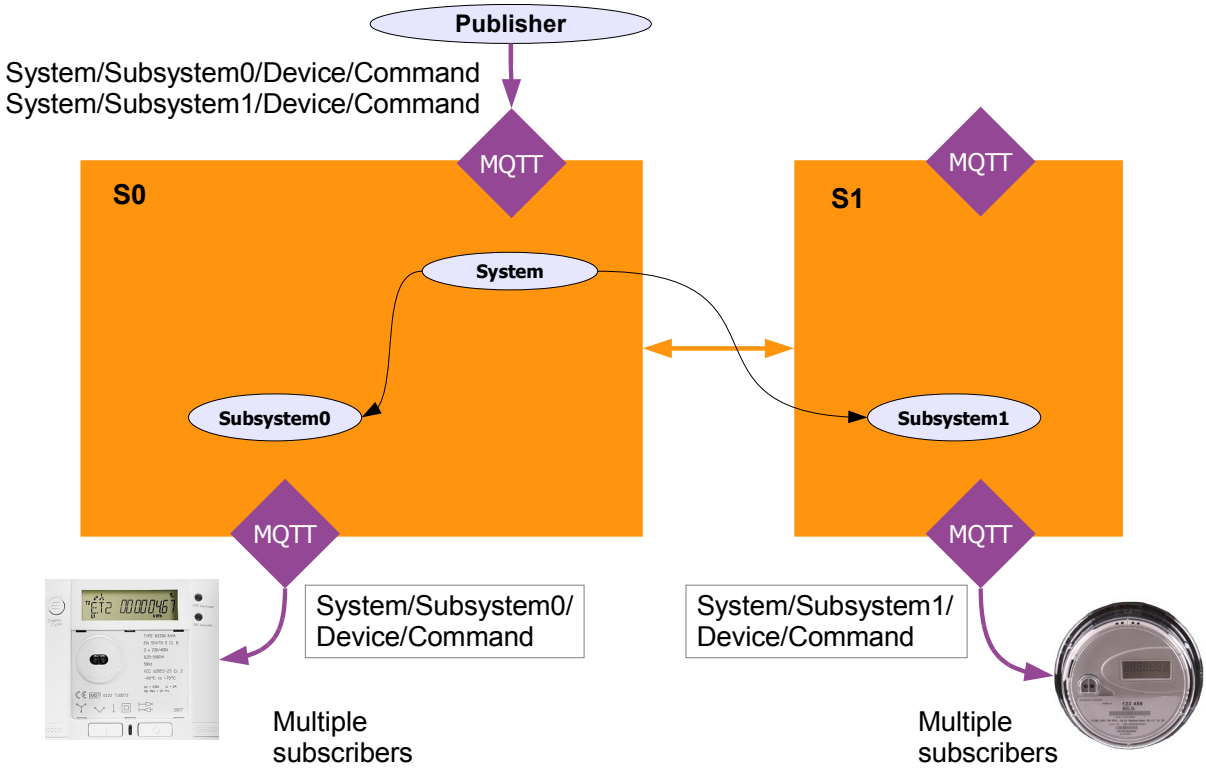

Figure 43: Multi-subscriber, distributed broker

QoS 0

Partition configuration			
Message rate (msg/s)	**Command topic count**	**Throughput per topic (msg/s)**	**Expected delivered throughput per subscriber (msg/s)**
800	8000	0,1	0,1
4800	8000	0,6	0,1

0,1 msg/s per subscriber

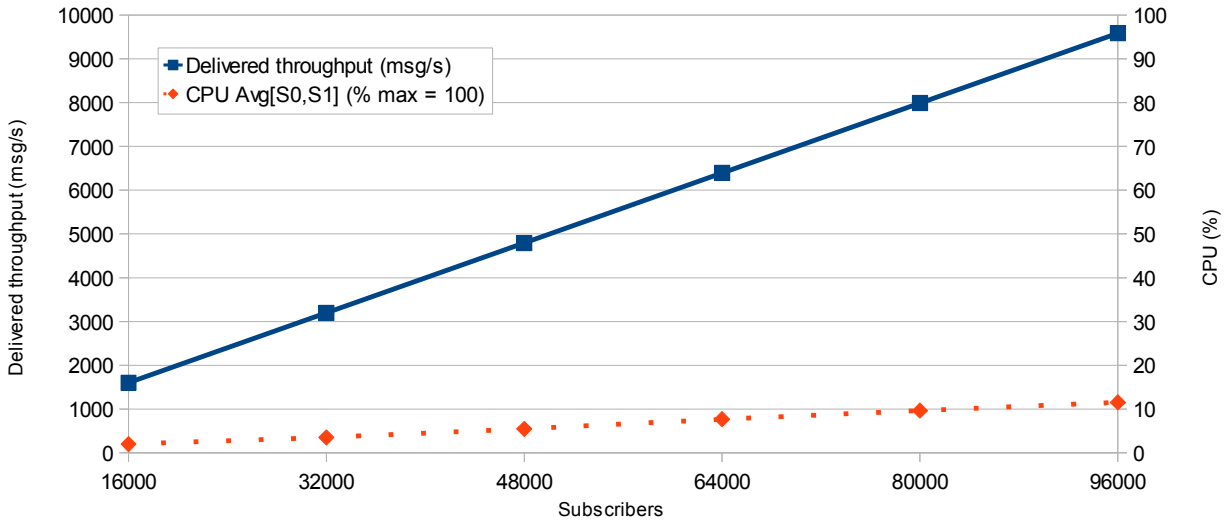

Figure 44: QoS 0, multi-subscriber, distributed broker, 0,1 msg/s per subscriber

0,6 msg/s per subscriber

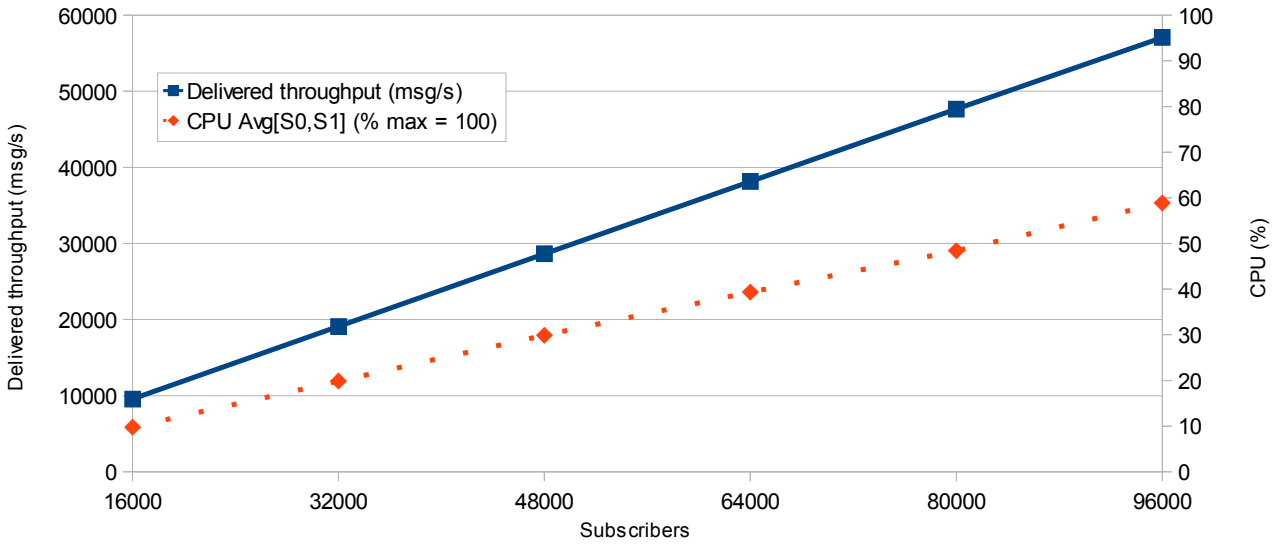

Figure 45: QoS 0, multi-subscriber, distributed broker, 0,6 msg/s per subscriber

QoS 1 and clean session False

Partition configuration			
Message rate (msg/s)	**Command topic count**	**Throughput per topic (msg/s)**	**Expected delivered throughput per subscriber (msg/s)**
800	8000	0,1	0,1
1200	8000	0,15	0,15

0,1 msg/s per subscriber

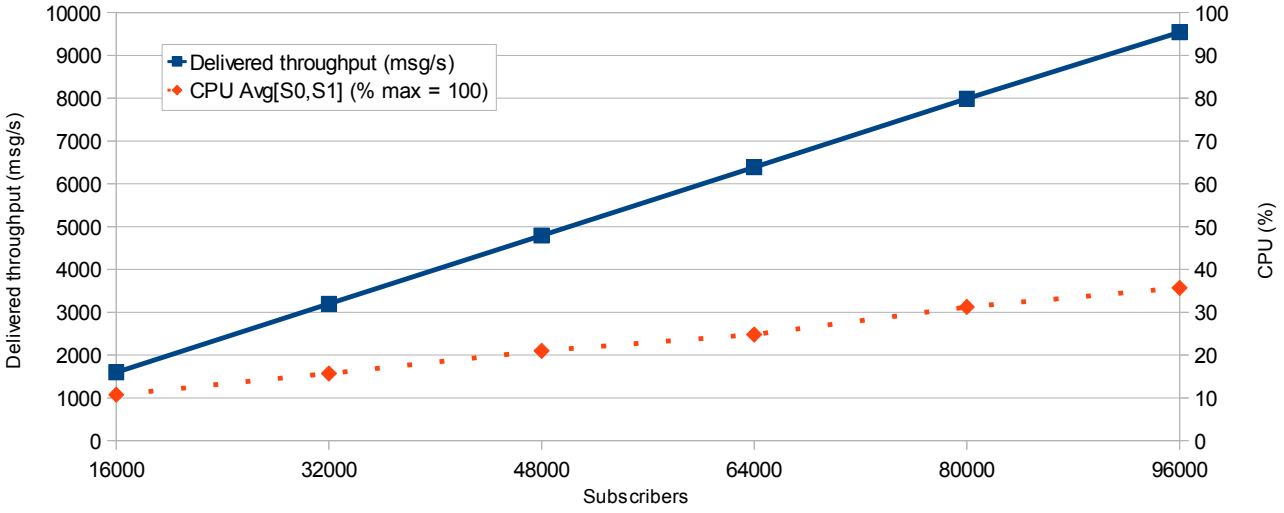

Figure 46: QoS 1, multi-subscriber, distributed broker, 0,1 msg/s per subscriber

0,15 msg/s per subscriber

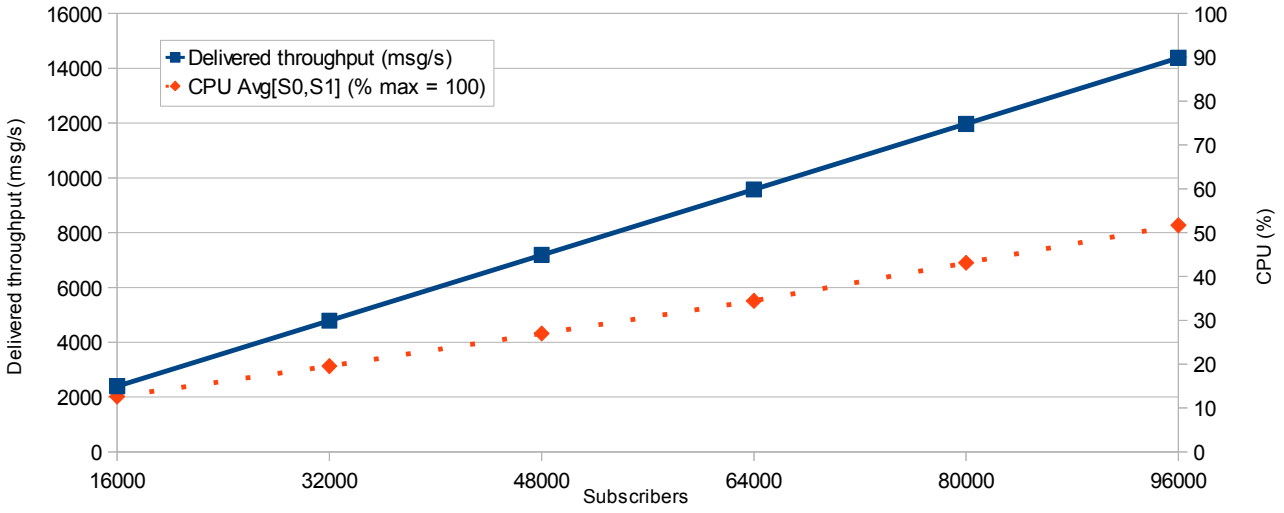

Figure 47: QoS 1, multi-subscriber, distributed broker, 0,15 msg/s per subscriber

QoS 2 and clean session False

Partition configuration			
Message rate (msg/s)	**Command topic count**	**Throughput per topic (msg/s)**	**Expected delivered throughput per subscriber (msg/s)**
320	8000	0,04	0,04

0,04 msg/s per subscriber

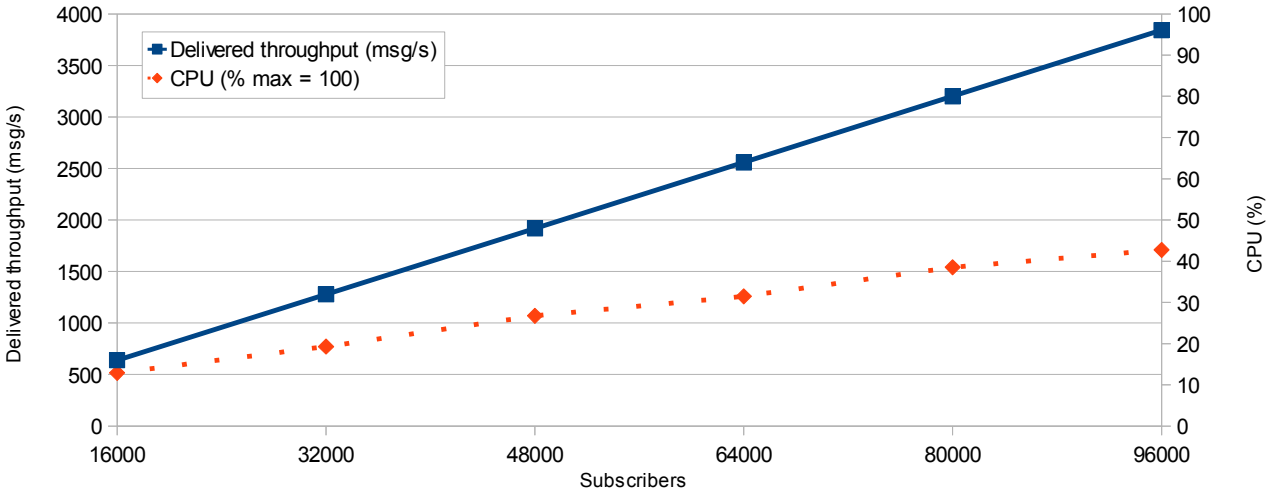

Figure 48: QoS 2, multi-subscriber, distributed broker, 0,04 msg/s per subscriber

6.5 Conclusion

The scenario called “multi-subscriber” simulates a large number of devices, for example smart meters, controlled by a central system. The devices are the subscribers and the central system is the publisher. Using a single server and constrained physical resources (2 cores, 4 GB RAM and SATA disk) JoramMQ can scale to 48.000 subscribers, each subscriber consuming:

- 0,6 message per second at QoS 0
- 0,15 message per second at QoS 1
- 0,07 message per second at QoS 2

With a clustered broker topology using two servers, JoramMQ scales to twice the number of subscribers, i.e. 96.000, consuming at the same rate as with the centralised broker, except at QoS 2 where the rate has to be lowered to 0,04 msg/s.

The distributed broker topology approximately gives the same results as the clustered topology.

7 Multi-subscription scenario

7.1 Overview

The scenario called “multi-subscription” simulates a large number of client applications, each subscribed to many MQTT topics. The multi-subscription scenario is different than the multi-subscriber scenario. In the multi-subscriber scenario (see section 6), each subscriber represents a device that has registered to only one command. In the multi-subscription scenario, each subscriber represents an application that has registered to many telemetry parameters.

This situation could happen for example in a payload operations centre publishing telemetry parameters for a given space mission to remote observers such as scientists remotely located. The observers would be MQTT subscribers registered to topics representing the telemetry parameters.

The multi-subscription scenario launches 1200 subscribers in batches of 100. There are less subscribers than in the multi-subscriber scenario but the delivered message throughput per subscriber is higher.

Two tests are done. The first one checks that the broker scales with the number of subscribers. The second one adds a slow consumer and checks if the broker is affected or not by this consumer. JoramMQ handles a slow consumer according to the QoS level specified by the message delivery:

- at QoS 1 and 2, messages are queued and swapped[9] to disk in case of overflow;
- at QoS 0, if the messages are queued, then they are swapped to disk in case of overflow;
- at QoS 0, if the messages are not queued, then the messages that cannot be delivered because of the overflow are dropped; in this particular case, the test accepts that some messages are lost.

Only the centralised broker is tested.

9 A message is swapped only once, even if several copies of the message are delivered to different subscriptions. If a subscription is durable then the message has already been persisted and therefore does not need to be swapped. It is just removed from the memory.

7.2 Centralised broker

In the same way as in the multi-subscriber scenario, topics are partitioned in several hierarchies. In the payload operations centre use case, a partition represents the telemetry parameters published by a spacecraft for a given mission.

Each partition is structured as follows:

- 1 root topic
- 10 subsystem topics
- 100 device topics per subsystem
- 10 parameter topics per device

There are one publisher and 100 subscribers per partition.

The publisher sends messages to the 10.000 parameter topics at a steady rate.

Each subscriber listens to the parameters of 100 device topics using the wildcard '#'. As there are 10 parameters per device, each subscriber listens to 1000 topics “Parameter”.

Each device topic has 10 subscribers. So every published message is copied and delivered 10 times.

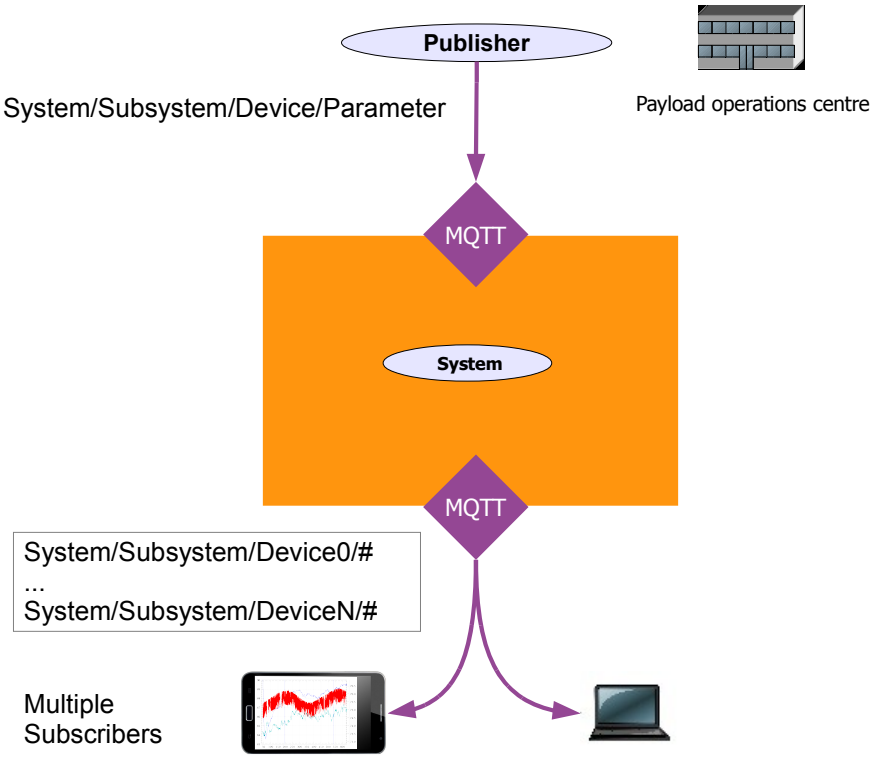

Figure 49: Multi-subscription, centralised broker

QoS 0, not queued

Partition configuration				
Message rate (msg/s)	**Topic count**	**Throughput per topic (msg/s)**	**Topics per subscriber**	**Expected delivered throughput per subscriber (msg/s)**
1500	10.000	0,15	1000	150

150 msg/s per subscriber

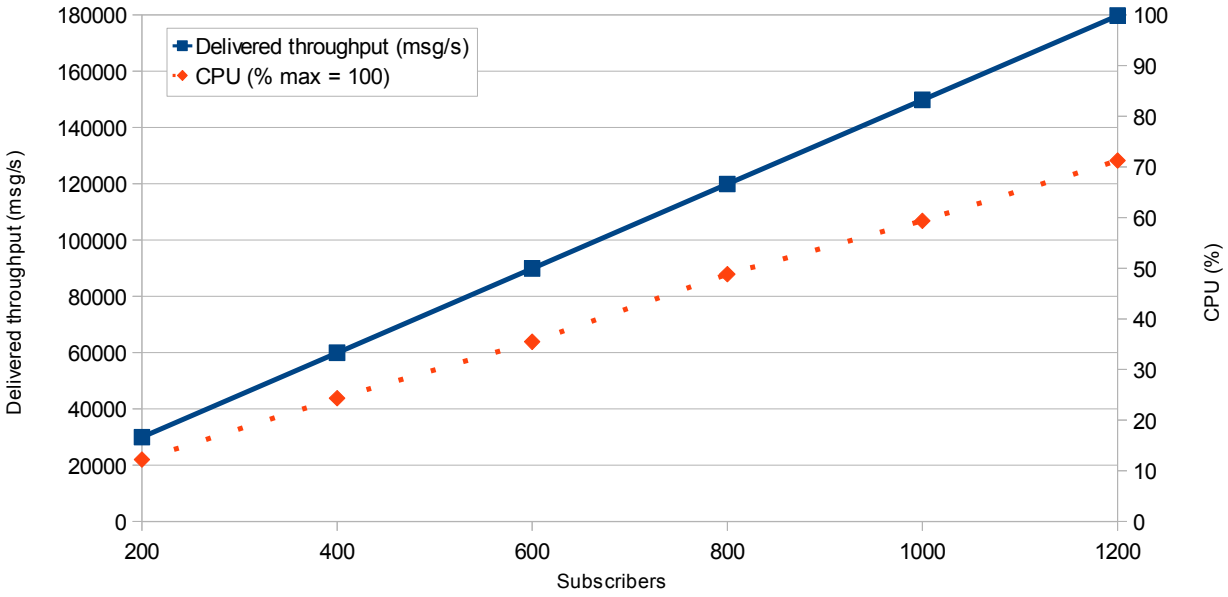

Figure 50: QoS 0 (not queued), multi-topic subscription, 150 msg/s per subscriber

150 msg/s per subscriber, 1 slow consumer

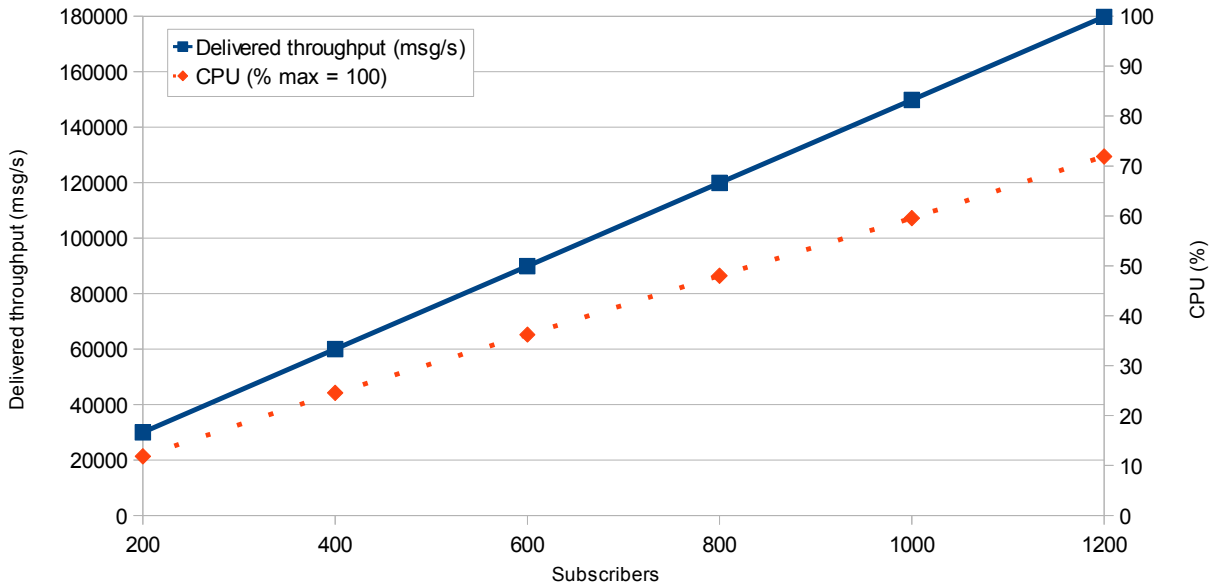

Figure 51: QoS 0 (not queued), multi-topic subscription, 150 msg/s per subscriber, 1 slow consumer

QoS 0, queued

Partition configuration				
Message rate (msg/s)	**Topic count**	**Throughput per topic (msg/s)**	**Topics per subscriber**	**Expected delivered throughput per subscriber (msg/s)**
1500	10.000	0,15	1000	150

150 msg/s per subscriber

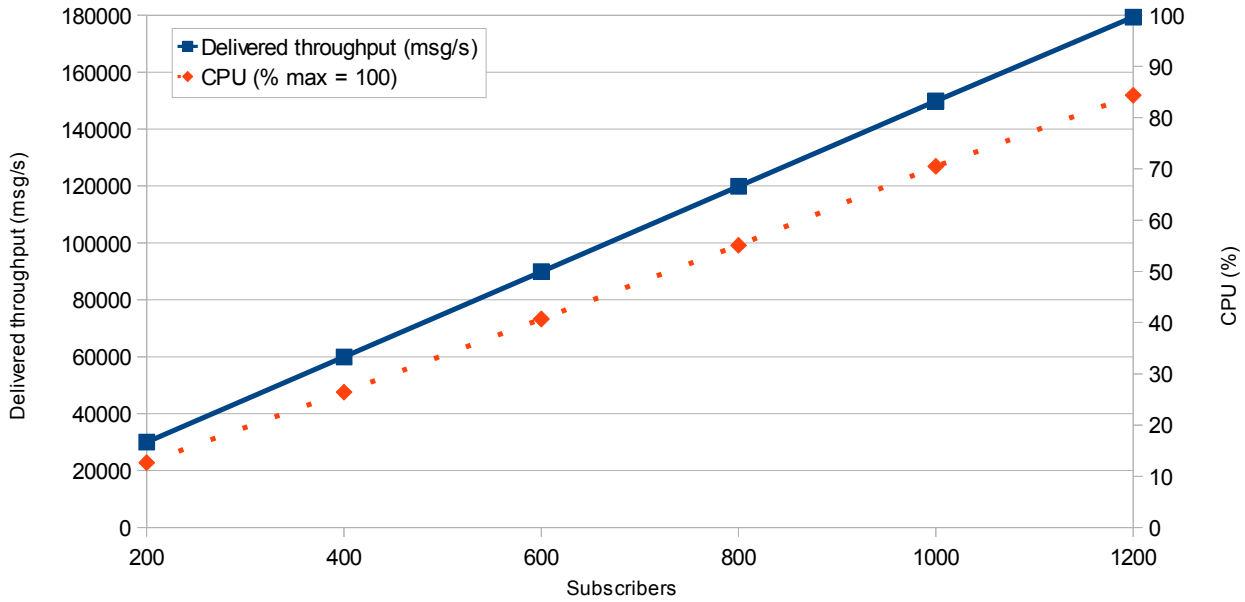

Figure 52: QoS 0 (queued), multi-topic subscription, 150 msg/s per subscriber

150 msg/s per subscriber, 1 slow consumer

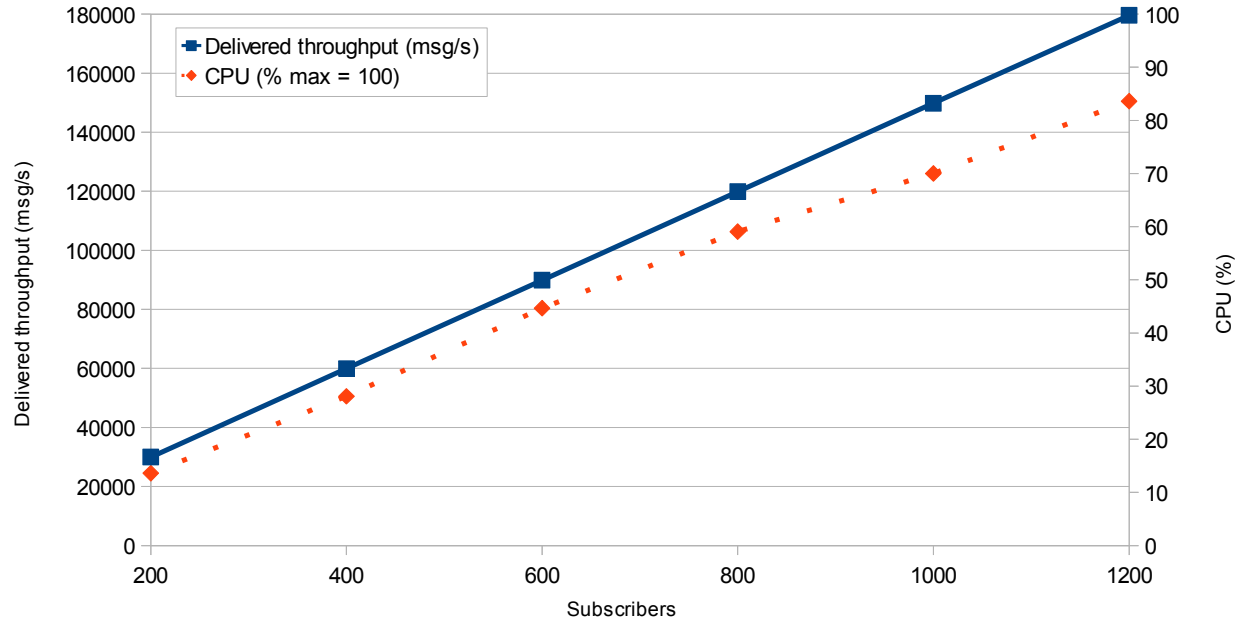

Figure 53: QoS 0 (queued), multi-topic subscription, 150 msg/s per subscriber, 1 slow consumer

QoS 1 and clean session False

Partition configuration				
Message rate (msg/s)	**Topic count**	**Throughput per topic (msg/s)**	**Topics per subscriber**	**Expected delivered throughput per subscriber (msg/s)**
400	10.000	0,04	1000	40

40 msg/s per subscriber

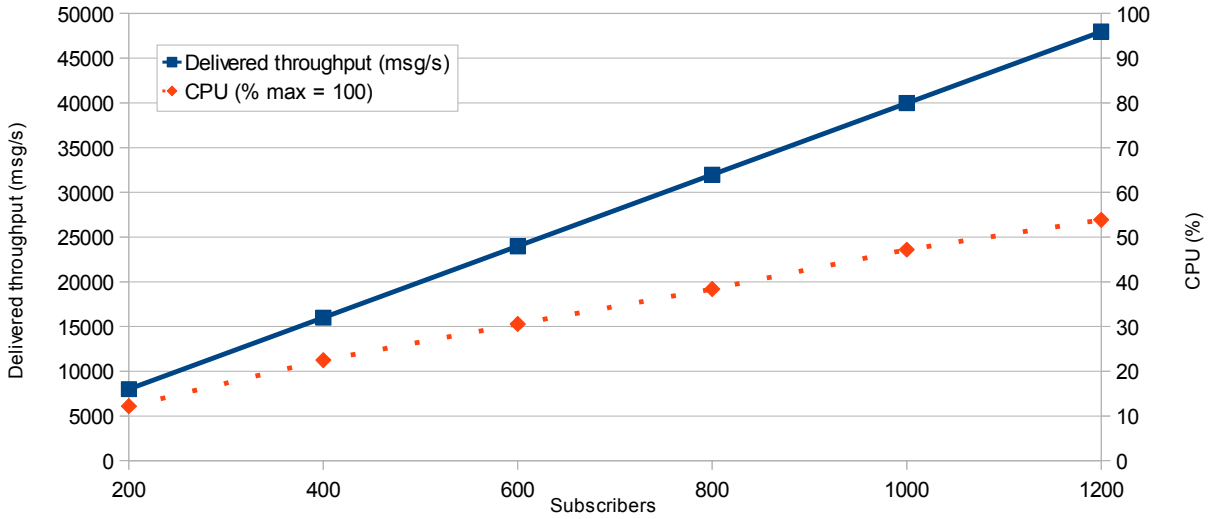

Figure 54: QoS 1, multi-topic subscription, 40 msg/s per subscriber

40 msg/s per subscriber, 1 slow consumer

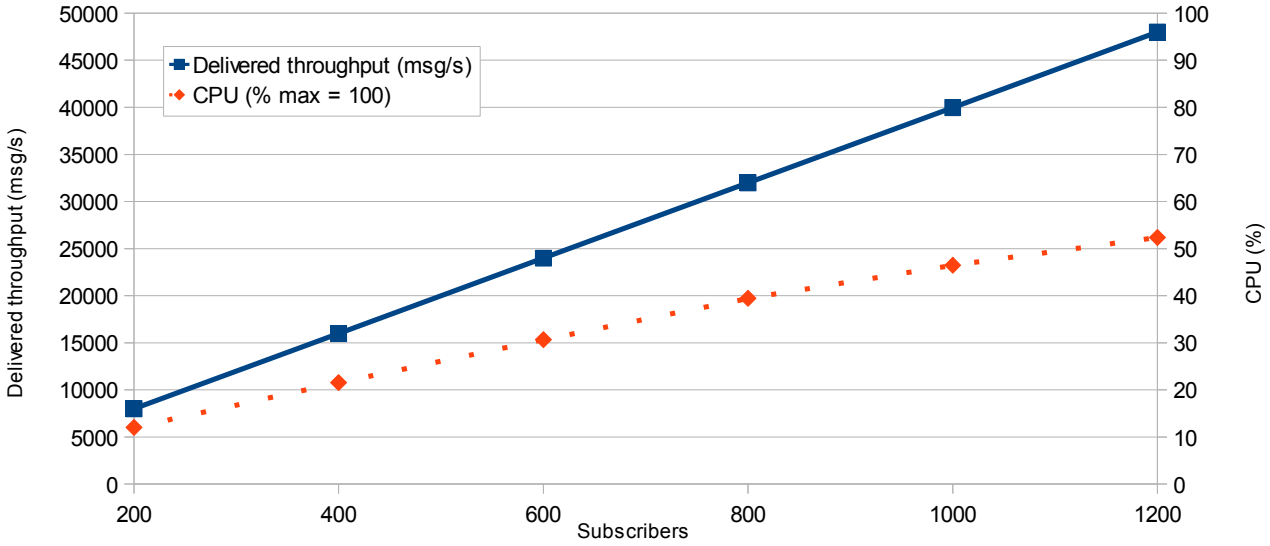

Figure 55: QoS 1, multi-topic subscription, 40 msg/s per subscriber, 1 slow consumer

QoS 2 and clean session False

Partition configuration				
Message rate (msg/s)	**Topic count**	**Throughput per topic (msg/s)**	**Topics per subscriber**	**Expected delivered throughput per subscriber (msg/s)**
100	10.000	0,01	1000	10

10 msg/s per subscriber

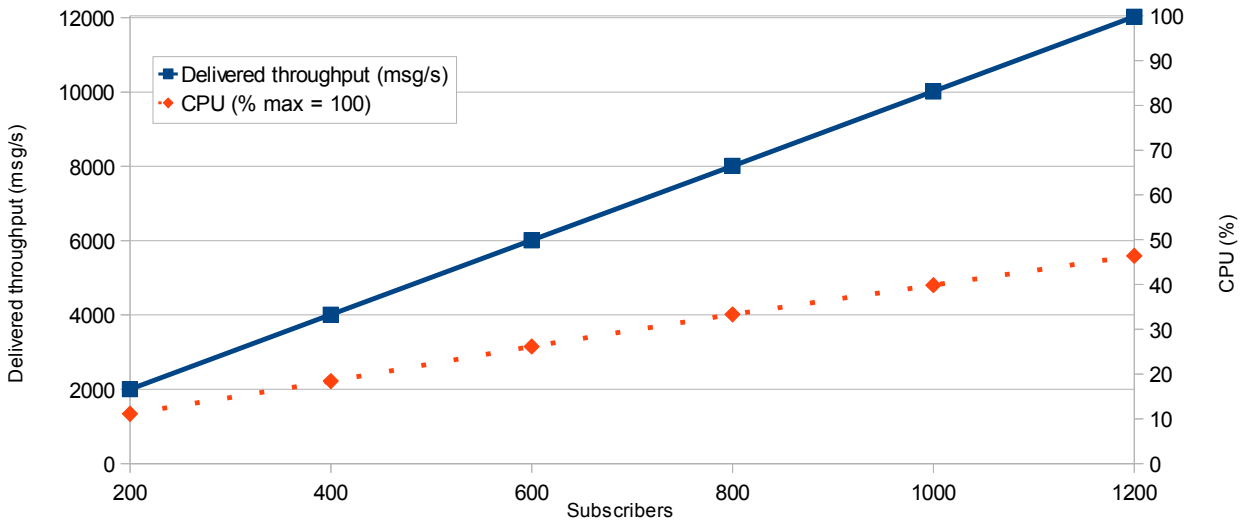

Figure 56: QoS 2, multi-topic subscription, 10 msg/s per subscriber

10 msg/s per subscriber, 1 slow consumer

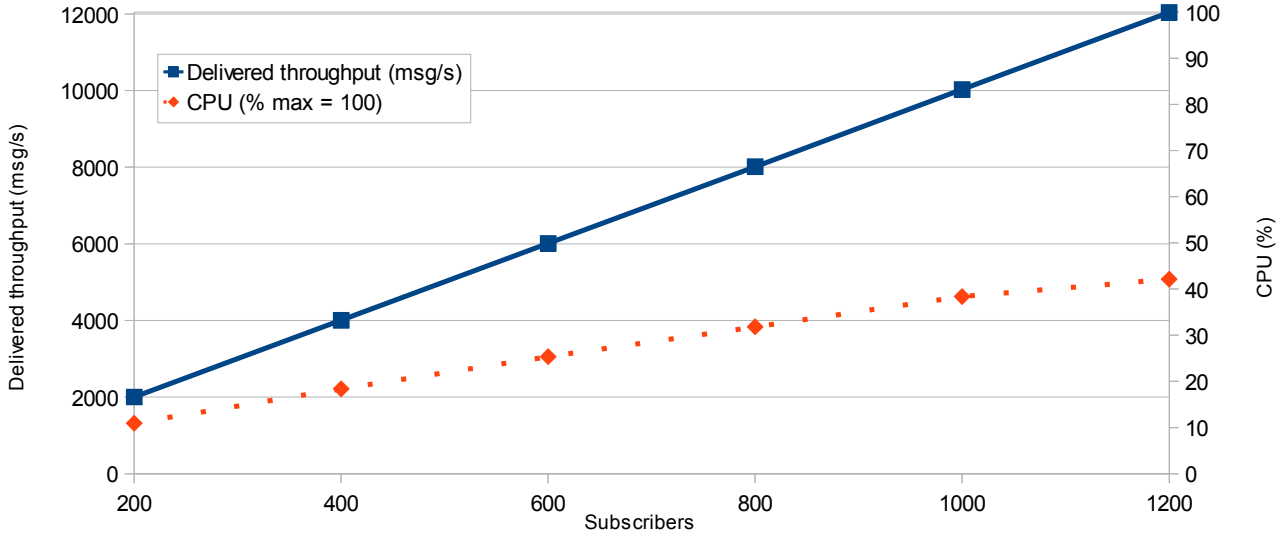

Figure 57: QoS 2, multi-topic subscription, 10 msg/s per subscriber, 1 slow consumer

7.3 Conclusion

The scenario called “multi-subscription” simulates a large number of client applications, each subscribed to many MQTT topics.

The tests show that the broker scales to 1200 client applications while delivering:

- 180.000 messages per second at QoS 0, not queued
- 120.000 messages per second at QoS 0, queued
- 48.000 messages per second at QoS 1
- 12.000 messages per second at QoS 2

The tests also show that adding a slow consumer does not affect the broker performance. The same throughputs are reached with approximately the same CPU level.

8 Conclusion

Using a single server and constrained physical resources (2 cores, 4 GB RAM and SATA disk) JoramMQ is able to handle 48.000 MQTT clients, each client producing or consuming messages at different rates depending on the QoS levels[10].

The nominal message rate equal to 0,1 message per second per MQTT client has been chosen as a typical message rate for IoT applications, e.g. a smart metering use case where clients are devices producing telemetry data (sensors) and consuming control commands (actuators).

At QoS levels 0 and 1, in the context of the tests, the nominal rate is very low and can be increased depending on the scenario. For example, at QoS 0, the multi-publisher scenario allows to reach 1 message per second per publisher, which is ten times the nominal rate. The maximum rate has not been reached, except at QoS level 2.

The same message rates have been handled with twice more clients[11], i.e. 96.000 thanks to the clustered and distributed topologies provided by JoramMQ.

The largest message throughputs that have been reached in these tests are:

- 96.000 messages per second at QoS 0
- 24.000 messages per second at QoS 1
- 6.700 messages per second at QoS 2

Finally, a different use case has been tested where MQTT clients are applications subscribed to real-time telemetry data. Each subscriber has registered to multiple topics. The number of subscribers is limited to 1200. In this situation, JoramMQ can deliver the following message throughputs to all the subscribers:

- 180.000 messages per second at QoS 0
- 48.000 messages per second at QoS 1
- 12.000 messages per second at QoS 2

10 QoS levels 1 and 2 have been tested by ensuring that messages are persisted and that a sync to disk is performed before acknowledgements are returned (PUBACK, PUBREC, PUBREL and PUBCOMP).

11 Except the multi-subscriber scenario at QoS level 2 which required to lower the rate by 40%, from 0,07 to 0,04 message per second.