Developer's guide to MQTT Brokers

Solution design and implementation implications

Brian Innes

Table of contents

1. Home

MQTT Brokers

Choosing a broker for an MQTT based project can be difficult, as there are many choices available and a number of features which can impact the design and implementation of a solution to fit a specific broker feature set.

This project aims to look at some features which can impact application design and implementation. Be aware, there are also many operational considerations which need to be taken into account when selecting a broker.

This project will only use Open Source brokers to allow developers to further explore the brokers and features without incurring costs, but the findings can be applied to commercial offerings.

Some of the features to be explored are:

- Scalability for resilience and load (cluster / bridge strategy)
- Client scalability (options with and without Shared subscriptions)
- Bridge to other brokers (same and different broker types)

1.1 Brokers to be considered

The following brokers will be used throughout this project, as they represent a selection of brokers with different combination of capabilities.

- Mosquitto
- EMQ X
- RabbitMQ

1.2 Test environment

Although the eventual target environment is a Kubernetes based system, the initial testing will be done using Docker.

Docker images used in test:

Broker	image	Architectures available	Clustered	Shared Subscriptions
Mosquitto	eclipse-mosquitto:1.6.12	linux/386, linux/amd64, linux/arm/v6, linux/arm64/v8, linux/ppc64le, linux/s390x	No	Yes
EMQ X	emqx/emqx:4.2.0	linux/386, linux/amd64, linux/arm/v7, linux/arm64/v8, linux/s390x	Yes	Yes
rabbitMQ	rabbitmq:3.8.9	linux/386, linux/amd64, linux/arm/v7, linux/arm64/v8, linux/ppc64le, linux/s390x	Yes	No

The test was done on an Intel i9 CPU, so the linux/amd64 architecture container images were used, but the table above shows the available architectures available on Docker Hub.

2. Deploying the brokers

The scripts in this repository should allow you to setup the 3 brokers, so you can experiment and test the various brokers options and configurations.

2.1 Prerequisites

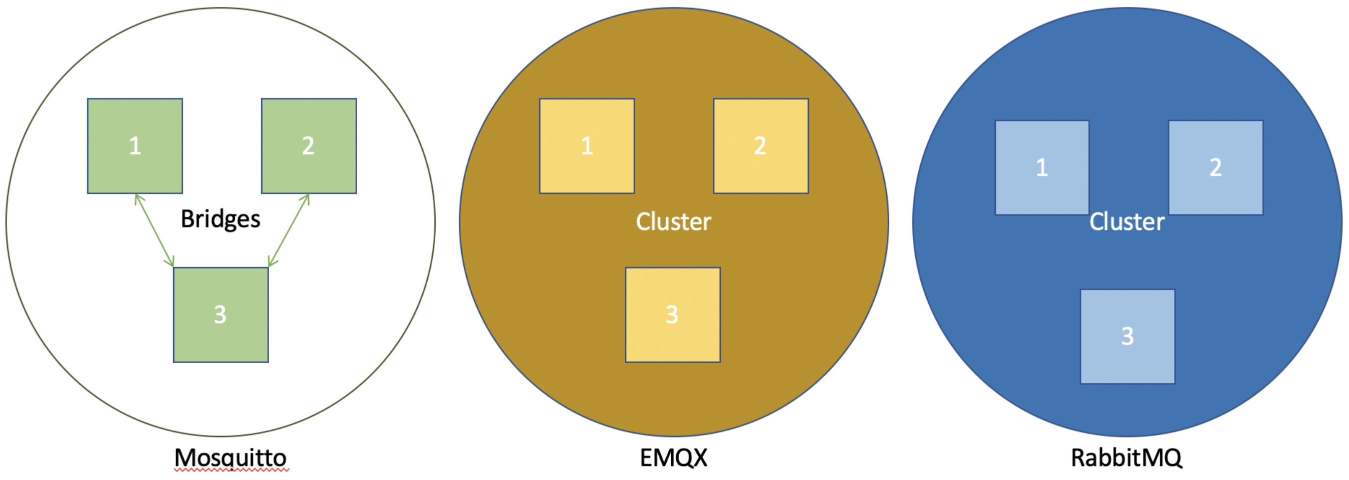
The scripts provided rely on Docker, so you need to have an up to date version of Docker installed on your system.

The scripts provided have been tested on up to date versions of Linux (using bash shell), MacOS and Windows 10 (using Command prompt)

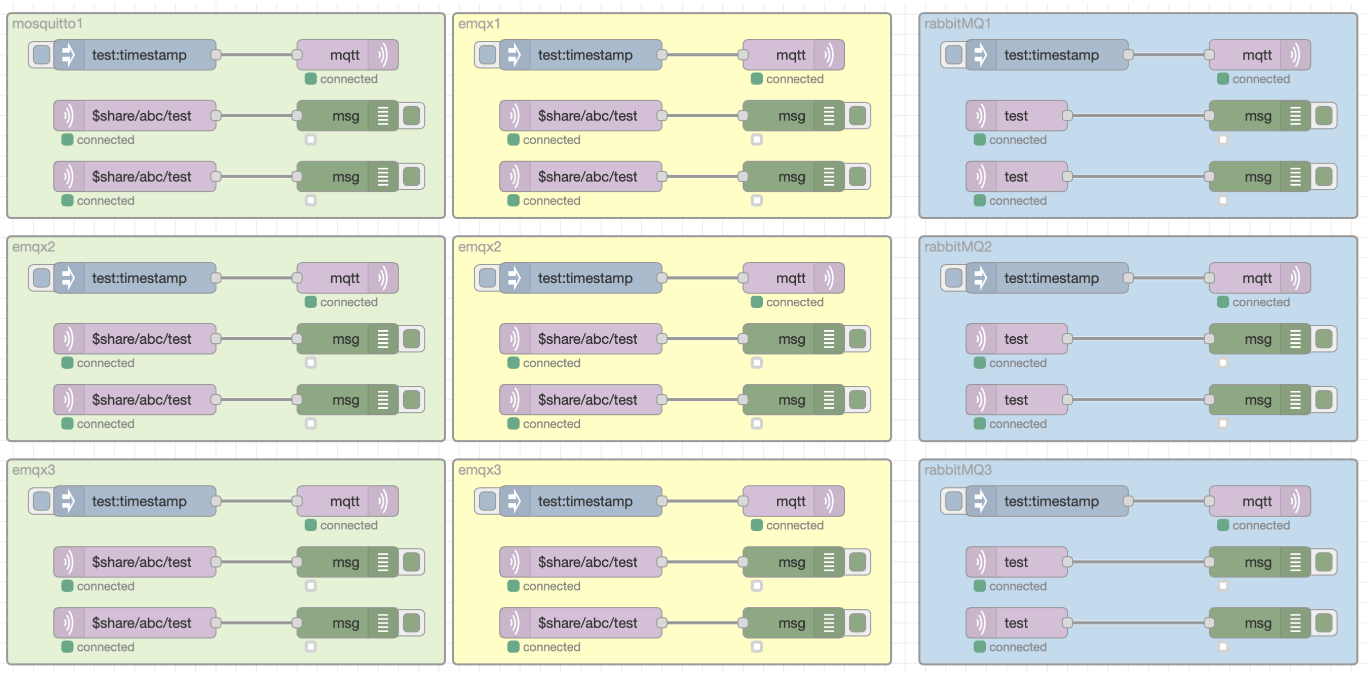
This project assumes you know the basics of MQTT, it is not an MQTT tutorial.

2.2 Running the scripts

The deploy script will setup a Docker network then deploy 3 instances of each of the brokers included in the test. For EMQX and Rabbit MQ the brokers will be configured as a cluster. Mosquitto has 2 bridges configured to transfer messages between brokers 1 and 3 and 2 and 3. All topics can cross the bridges in both directions.

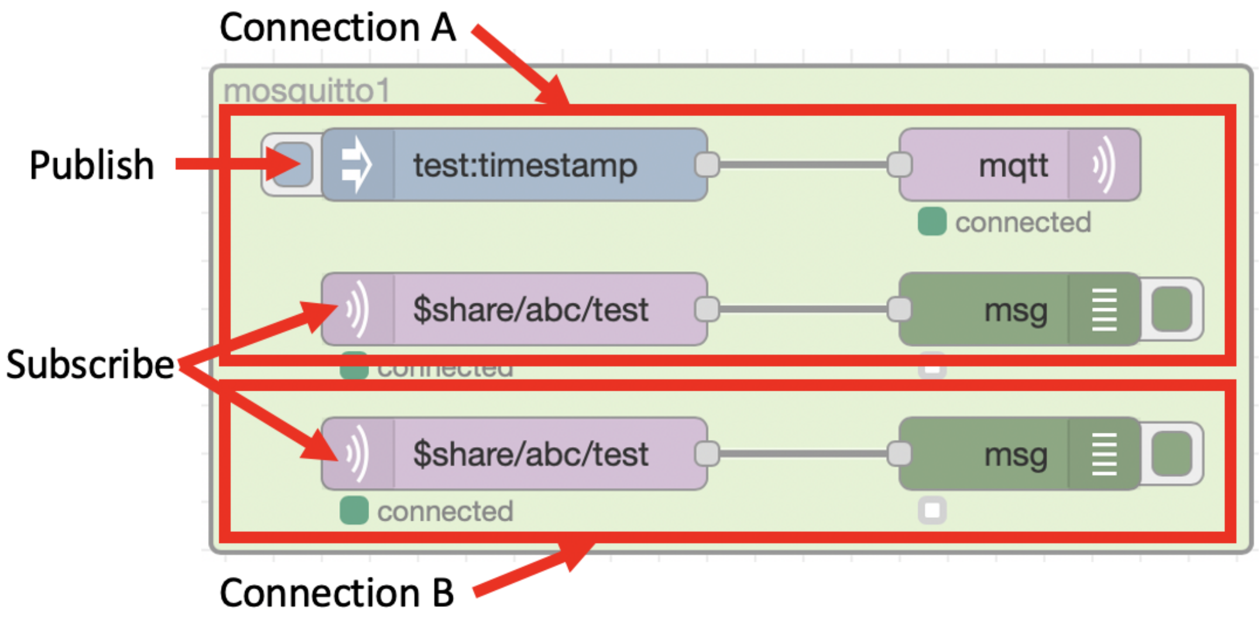


A Node-RED instance is then deployed with a test flow added. This Node-RED flow allows you to test each of the broker configurations, as there are 2 connections made to each of the 3 instances of broker. Where the broker supports shared subscriptions, the Node-RED node will create a shared subscription.



Each shaded box represents 1 broker, and each of the MQTT-in nodes has their own connection to the broker, with a subscription to the **test** topic

The Inject nodes publishes a message to the **test** topic on the connected broker and the debug nodes allow you to see which MQTT clients received that message.



2.2.1 Running the test

To run the test simply run **deploy.sh** (on Linux or MacOS) in a terminal window or **deploy.bat** (on Windows) in a command window.

This will deploy the brokers and Node-RED as Docker containers.

2.2.2 Accessing Node-RED

To access the Node-RED user interface you can launch a browser on the same system running the Docker containers and browse to http://localhost:1880

2.2.3 Testing each broker

You can use the button on the inject nodes to publish a message to a specific instance of a broker, then use the debug node status or output in the debug panel to see which client connection(s) received the message.

You should see that each broker produces different results, with the different capabilities and configurations of the brokers under test.

2.2.4 Tidying up

Once you have completed the testing you can run the cleanup.sh or cleanup.bat script to delete all the containers and the Docker network configuration.

Warning

If you made any changes to the Node-RED flow, then running the cleanup script will delete the container and you will loose the modified Node-RED flow

2.3 Configuration files

The brokers and Node-RED flow are configured and initialised using files from the **files** folder in this project. At deploy time a new directory is created called **data**, and content from the **files** folder is copied to the data folder. Content from the **data** folder is mapped into the running docker containers to make configuration available to the brokers and Node-RED.

You will see the EMQX and RabbitMQ brokers share a common configuration across all brokers, with environment variables or command line options being used to pass in parameters unique to each broker, such as the broker name.

The Mosquitto brokers each have a unique configuration files, as the bridges configuration is within the configuration files.

The Node-RED flow is fully populated, including passwords, from the config.json, flows.json and flows_cred.json files. If you modify the Node-RED flow and want to save the changes, then you should save the files from the **data** directory before running the *cleanup* script. In the **data** folder the config.json file has a leading *full stop* character, so is actually called **.config.json** and may be hidden in your file explorer application.

3. Clustered MQTT Brokers

Clustering is a mechanism to spread a broker over multiple systems. There are a number of MQTT brokers available that support clustering.

With the introduction of cloud infrastructure deployments models are changing. In more traditional hosting environments, some infrastructure components were vertically scaled using large server machines, redundant or specialist hardware, but with cloud there is a move to standardised, commodity hardware with horizontal scalability used to deliver increased load and resilience of applications and services.

A clustered broker behaves like a single broker, with published messages being delivered to all clients with an appropriate subscription, no matter which cluster member they are connected to.

Clustered brokers allow a broker to be deployed to cloud infrastructure and some clustered brokers have built in features to support deployment and configuration within a Kubernetes environment.

One key advantage of using a clustered broker is that it removes the single point of failure from a solution. If a cluster member fails then the other members of the cluster still continue working and client that were connected to the failed broker are able to immediately reconnect to one of the working cluster members.

Most clustered brokers have intelligent routing, where all messages do not need to be sent to all cluster members. Only cluster members with a client subscription matching a message topic will be sent the message to forward to subscribed clients.

In the test setup both the EMQX and RabbitMQ brokers support clustering.

When looking at different clustered brokers there are differences with the internal routing of messages and how the failure of a cluster member is handled, so you do need to understand the functional and non-functional requirements you need from a broker.

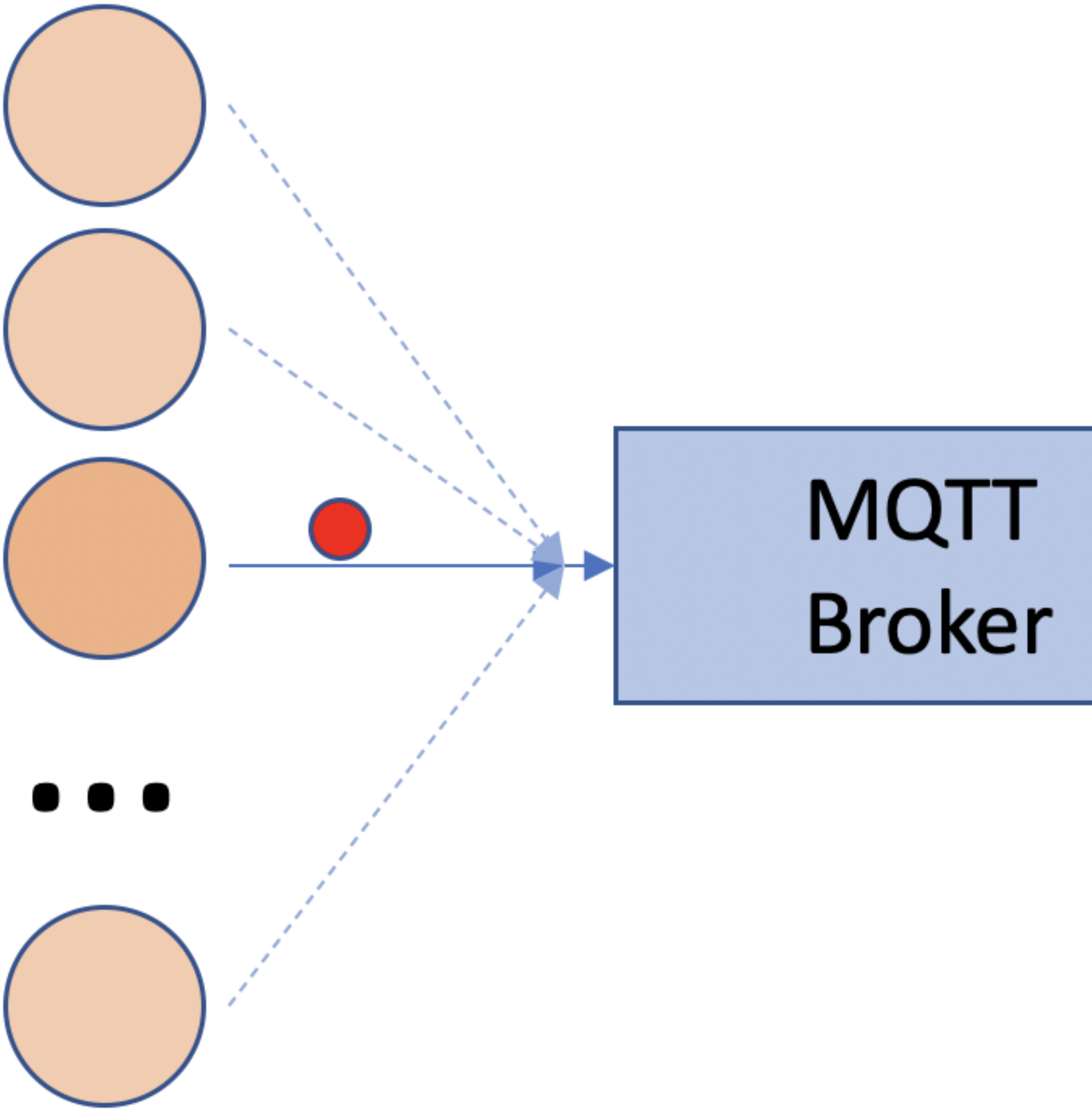
4. Shared Subscriptions

Version 5 of the MQTT specification introduced shared subscriptions. A a number of brokers added their own custom shared subscription capability before v5 of the MQTT standard was released, so you may come across alternate shared subscription mechanisms.

The MQTT v5 standard shared subscription works by pre-pending **$share/\<group>/** to the topic.

Subscribing to the topic **$share/abc/test** means the client has subscribed to receive messages published to topic **test**, but in a shared subscription group named **abc**.

All clients using the same subscription group and topic will belong to the same shared subscription, which results in only 1 client in the group receiving each published message to the subscribed topic.


MQTT
Broker

Standard Subscription
All clients get all messages

4.1 Why use shared subscriptions?

Shared subscriptions are really useful when an applications needs to be scaled using horizontal scaling, where additional instances of an application are run and clients are load balanced between the instances. Horizontal scaling is the standard mechanism used to scale workloads in cloud computing. Applications can be scaled to cope with load and/or to ensure resilience.

Without shared subscriptions all instances of an app receive a copy of every message. For some applications this is desirable, but for many applications this can result in duplicate work being done, but also duplicate data being created, which is undesirable and left to the application design or implementation to work around.

5. Bridging Brokers

MQTT broker bridging is another mechanism available to distribute loads over multiple brokers.

A bridge is a broker to broker connection with a set of rules that specify what messages are sent across the bridge. The rules can specify if messages matching a rule can only flow in 1 direction, or if they can flow in both directions across the bridge. This allows messages published to one broker to be delivered to clients connected to a different broker.

Care must be taken when using bridges to ensure that the bridges do not form loops, where a message could be perpetually sent between brokers in a never ending loop.

Many clustered brokers also support MQTT bridges. However, when brokers not supporting clustering scale using bridges alone then individual brokers are single points of failure.

In the test configuration broker 1 and 2 have bridges to broker 3, allowing all messages to pass both ways across the bridge. A bridge will ensure a message sent across the bridge in one direction does not get sent back to the original broker, but this doesn't apply when crossing multiple bridges. But a message arriving on a broker across the bridge will also be forwarded across any other bridge configured on the receiving broker.

If Broker 3 fails then messages published to Broker 1 will no longer be received on Broker 2

5.1 Loops

If another bridge was introduced between brokers 1 and 2 then:

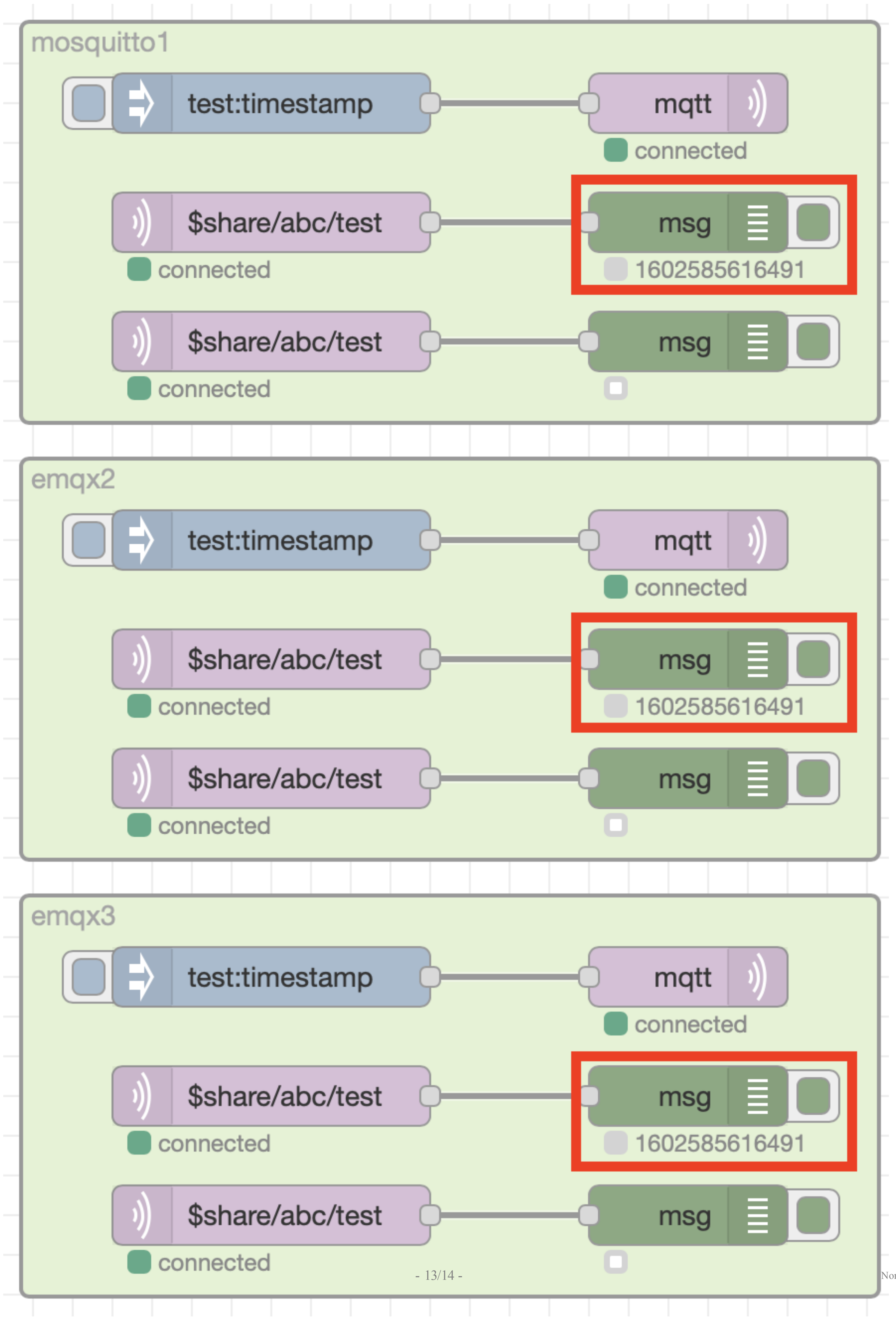
- a message published to broker 1 would be sent to brokers 2 and 3
- the message arriving at broker 2 wouldn't be sent to back across the bridge to broker 1, but it would be sent across the bridge to broker 3
- the message arriving at broker 3 wouldn't be sent back across the bridge to broker 1, but it would be sent across the bridge to broker 2

At this point brokers 2 and 3 each have 2 copies of the message originally published to broker 1. Now with the message on broker 2, which arrived via broker 3, now on it's second hop (Broker 1 -> Broker 3 -> Broker 2), the message wouldn't be sent back to broker 3, but would be sent to broker 1 - and so the loop continues. Similarly the message sent to broker 3 from broker 2 would also then be forwarded to Broker 1. Each message would end up on this perpetual loops causing a message storm, which will overwhelm all the brokers.

5.2 Shared Subscriptions and Bridges

Shared subscriptions do not work across bridges, so clients connecting to a broker with the same shared subscription will only share messages with other clients belonging to the same broker.

To see this look at the test deployment. When a single message is published then 1 client on each broker receives the message, as the message is sent across the 2 bridges so all brokers receive the message. They then forward it to 1 client in the shared subscription group on each broker, so as there are 3 brokers, 3 clients receive the message.


mosquitto1
test:timestamp
mqtt
connected
$share/abc/test
connected
msg
1602585616491
$share/abc/test
connected
msg
emqx2
test:timestamp
mqtt
connected
$share/abc/test
connected
msg
1602585616491
$share/abc/test
connected
msg
emqx3
test:timestamp
mqtt
connected
$share/abc/test
connected
msg
1602585616491
$share/abc/test
connected
msg

compare that with the EMQX broker, which is clustered rather than using bridges, where the shared subscriptions are cluster wide, so when a message is published to the broker then only a single client with the shared subscription will receive the message.

Using shared subscriptions in conjunction with bridges does add some additional constraints, where all clients belonging to a shared subscription need to connect to the same broker.