RabbitMQ Performance Analysis

GHITA Bogdan, SCPD
TOMA Claudiu-Mihai, SCPD

0. Purpose

The purpose of this project is testing the performance of the open source message queueing system, RabbitMQ[1]. We will start by defining the metrics that need to be evaluated, followed by the constraints we need to satisfy in order to obtain accurate results, then design a generic architecture suitable for our problem. Next we will build an automated testing system which allows the user to easily define test cases and visualize results.

1. Metrics

The main metrics that reflect the performance of RabbitMQ are the message publish rate and the message delivery rate. Other queueing metrics are the message get, acknowledge and redeliver rates. Other performance indicators are CPU (idle, user, system, iowait) and memory (free, cached, buffered) usage, followed by disk I/O (reads & writes per unit time, I/O wait percentages), free disk space, Network throughput (bytes received, bytes sent) vs. maximum network link throughput and System load average (/proc/loadavg).

Our goal is to monitor these parameters while running different test cases, then plot the values for each metric and build interactive charts.

2. Constraints

Generally, a message queueing system implies at least 3 types of entities: the queue, the producers - which publish messages to the queue - and the consumers - which consume messages from it. These entities usually run on different systems and are not usually in the same network either.

Since our goal is to monitor the RabbitMQ system, the target for the tests will be the queue. Thus, we need to make sure that the bottleneck will be on the RabbitMQ nodes and not on other parts of the system (consumers, producers, network bandwidth).

Another constraint is that the system resources monitoring process is accurate and reflects only the utilization of resources consumed by the three entities describe above. For this purpose we need to isolate the RabbitMQ system the producers and the consumers from each other and from other entities.

3. Architecture

The architecture will consist of one or more RabbitMQ nodes that will be either form a cluster[2] or run independently. The consumers and producers will be RabbitMQ clients written in Python using the Pika[3] library.

In order to satisfy the isolation constraint we will use Docker[4]. Each RabbitMQ node will run in a separate container and the producers and consumers will run in a common container. This ensures that the monitoring process will be accurate as we can easily inspect resource utilisation of each container.

Docker also helps us address the other constraint: placing the bottleneck on the RabbitMQ system. We can achieve this by limiting the resources of the RabbitMQ and giving much more resources to the other containers. Another trick that helps in lowering the load on the producers is using pregenerate messages loaded in memory before testing starts. This way, the producers only send bytes stored in memory and the consumers only receive them.

Furthermore, in order to ensure that the bottleneck is not on the network, we need every fast network transmission. We can achieve this by all containers on the same physical machine. This way, the network transfers will be done in memory, achieving high speed.

4. Testing system

The system is split in 4 entities: test subject, monitoring agent, results visualization, and test automation. The test subject consists of the RabbitMQ, producer and consumer containers. The monitoring agent is a Python scripts that periodically gathers the target metrics. The RabbitMQ specific metrics will be obtained using the RabbitMQ Management Plugin, through the HTTP API[5] and the system metrics will be gathered using the Docker Python API[6].

In order to store and visualize the results we deployed the ELK Stack[7]. The monitoring agent sends the results to Logstash through the HTTP API, where they get parsed and inserted into Elasticsearch. From Kibana the results can be visualized either as raw data or by building charts.

Tests are defined in JSON files and have the following format:

```
{
"begin_delay_s": 15,
"run_time_s": 180,
"end_delay_s": 15,
"exchange": "performance-analysis",
"queue": "performance-analysis.exp-rand",
"distribution": "exp-rand",
"file": "data/small/1K.out"
}
```

The automation script written in Python makes sure the ELK container is running, starts the monitoring agent and then proceeds to running each the test cases one at the time. It saves the start and end time of each test case and generates the Kibana URL with the time interval in order to visualize the results independently.

5. Test cases

The metrics that can be varied in order to generate test cases are: number of messages per second, the size of the messages, the number of producers and consumers, the number of RabbitMQ nodes, the resource limits for the RabbitMQ container.

In order to vary these parameters we chose to use 4 common distributions: Gaussian Distribution, Poisson Distribution, Geometric Distribution and Exponential Distribution. For each test case random numbers are sampled from one of these distribution and can be scaled and used in determining the values for the metrics. For each distribution there is a separate queue in RabbitMQ.

In addition to random sampling, we chose to build a special test case which uses an exponential sequence to test the limits of the system.

6. Results

We defined 5 test cases, one for each distribution and one for the exponential sequence. Every test lasts 2 minutes, uses one RabbitMQ node, one producer and one consumer. The producers upload messages of size 1K. The RabbitMQ container is limited to one CPU and 512MB RAM. The results are the following:

a) Gaussian, Poisson, Geometric, Exponential:

RabbitMQ handles well the load and manages to deliver messages at the rates that they are published (between 800 and 3500 messages per second). The system resources utilization is acceptable. The RabbitMQ container uses around 50% of the memory and 40% CPU.

e) Exponential Series:

We used powers of 2 to generate messages. Each second the producer publishes 2^i messages, with i starting from 0 and increasing each second. The RabbitMQ system manages to deliver messages at a maximum rate of 8000 message per second and reaches the 100% CPU utilisation limit and 60% memory usage.

7. Conclusion

RabbitMQ is a highly scalable message broker managing to deliver messages a high rates even when it has very limited resources. It dose not crash or lose any message as it reaches the hardware limits, making it a very reliable system.

8. References

[1]https://www.rabbitmq.com/
[2]https://www.rabbitmq.com/clustering.html
[3]https://pika.readthedocs.io
[4]https://www.docker.com/
[5]https://www.rabbitmq.com/management.html
[6]https://docker-py.readthedocs.io
[7]https://www.elastic.co/elk-stack