RabbitMQ Performance and Scalability Analysis

by Brett Jones, Scott Luxenberg, David McGrath, Paul Trampert, and Jonathon Weldon
CS 4284: Systems and Networking Capstone Faculty Advisor: Dr. Ali Butt

Background:

RabbitMQ is a variant of the Advanced Message Queuing Protocol (AMQP). The RabbitMQ server is written in a language called Erlang and is built on the Open Telecom platform, which is an application server written in Erlang. Work on RabbitMQ development started in the summer of 2006 leading to its first public release, under MPL, in 2007. RabbitMQ is used as a broker to control the sending and receiving of messages. It can be used in two ways, one with a queue-like system, and the other in a publisher-subscribe message system using an exchange. By creating a group of publishers and subscribers that can access the messaging nodes, it can create a network of information that can span from small to large scale in a local area, or over large geographical distance. The distribution of information sent from the publishers to the hub to be distributed to the necessary subscribers allows for applications to run while relying on data from other locations, wherever they may be. This allows RabbitMQ to be useful in designing architectures for small localized systems to large, geographically dispersed interactive systems.

Motivation:

We teamed up with Rackspace US, with the help of Gabe Westmaas. We were given access to six servers with RabbitMQ installed upon them, three in Dallas and three in Chicago. Our goal was to analyze the scalability of the RabbitMQ architecture and to determine if it is feasible to use as the communication backbone for distributed systems. Through testing we could determine the possible uses and limitations of using the RabbitMQ architecture. Once finished, the results would be important to future projects as it would open up more options to developers for the underlying structure of applications.

Process:

We tested by varying the number of subscribers and publishers in a node, which was accomplished by clustering nodes into groups and seeing how performance increased or decreased with the combinations of node. Another parameter that we controlled for scalability was the number of messages sent in a test. For us, we used either 4K messages a set or 16K messages a set.

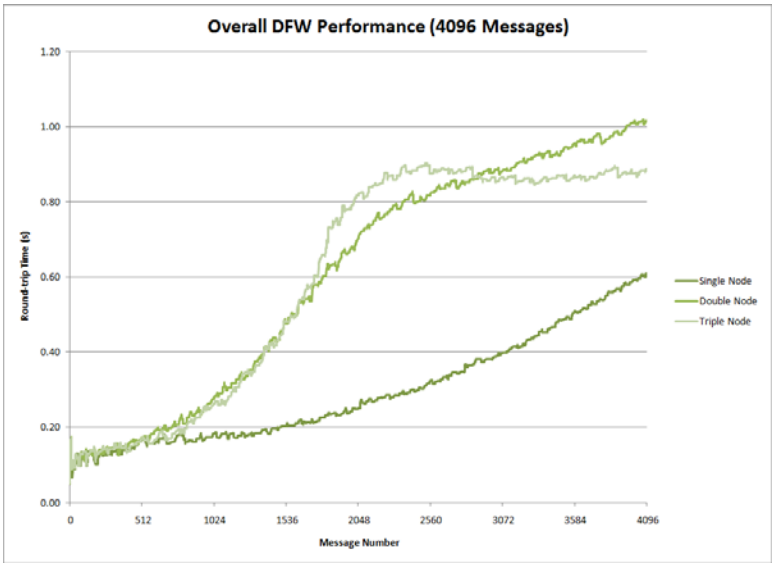
By testing the throughput in each individual area, we could determine the potential bottlenecks in high traffic situations and whether or not the architecture scaled appropriately. The site describing RabbitMQ claimed that it would scale linearly, but there were not any tests available that seemed to back up the claim. Therefore, we tested loads applied to the server with a single publisher and subscriber, multiple publishers and a single subscriber, single publisher and multiple subscribers, and with a multiple of each. By testing each of these individually, we would not only narrow down the main causes for slowing speed at higher loads, but also be able to supply information on similarly implemented systems with the same set up as one of these tests.

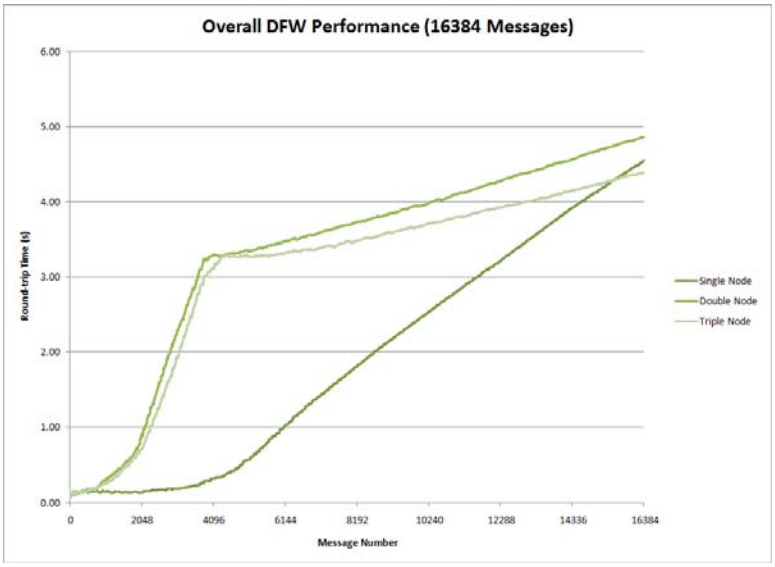
Single Publisher Single Subscriber:

Our first and most basic test involved using a single publisher and a single subscriber on a content feed served by a RabbitMQ cluster with a varying number of servers. Each message sent over the feed was

a kilobyte in size and contained a message number (to ensure ordering) and the system time when the message was sent; the rest of the message was zeroed out. Since the system time was used, we found that we had to run the publisher and subscriber from the same physical machine in order to get meaningful results; using separate machines introduced clock skew, which, at the precision of nanoseconds, is a significant issue. In our initial tests, we discovered that the machine we were using was not powerful enough to handle both the publisher and subscriber, as we noticed scheduler issues affecting the results. We decided to switch to using the Virginia Tech rlogin cluster, as the machines in the cluster were sufficiently powerful to run both the publisher and subscriber with no scheduling issues. With this resolved, we tested the Dallas/Fort Worth (DFW) servers in one-node, two-node, and three-node clusters using data sets of 4K (4096) and 16K (16384) messages. The subscriber code, after receiving all messages, printed out the message number and transit time (to nanosecond precision) to a text file; these files were imported into a spreadsheet and charted by message number. Each test was run three times and averaged, with the averages plotted together.

The 4K message set test showed a significant performance hit with clustering compared to no clustering, as shown in Figure 1. Until roughly 512 messages, the performances are relatively close, but beyond that, the round-trip times for messages sent through the clustered messengers increases much faster than for the single-node messenger. However, this increase levels off for the clustered messengers after roughly 2048 messages are processed, whereas the performance of the single-node messenger continues to degrade. This effect is due to the synchronization cost of distributing the messages between the servers in the messenger cluster; while the messages are still coming in, the server receiving the messages has to distribute them to the other servers in the cluster, and the cluster as a whole has to decide which server will send which message to the subscribers. Once the publisher stops publishing messages, the messenger cluster can push out the messages to the subscribers as fast as the network capacity will allow, which is why the graphs flatten out for the clustered messengers. The single-node messenger does not have the synchronization cost, but it also does not have the luxury of other servers to help send messages to the subscribers, so the performance graph neither spikes nor flattens out.



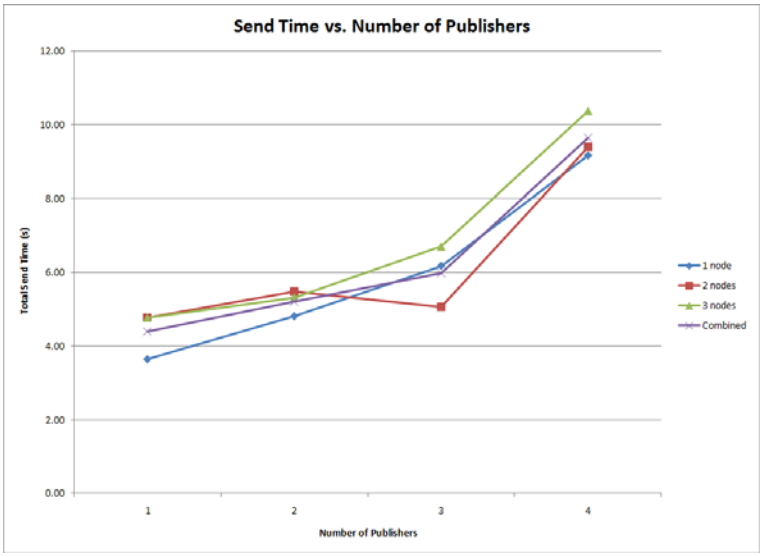


The 16K message set test shows a similar story to the 4K message set test. The performance of the clustered messengers takes a significant performance hit from the synchronization of the messages, but once the publisher ceases sending messages to the messenger, the graph becomes much flatter. The performance of the single-node messenger, as before, initially retains quick turnaround times, but gradually deteriorates. A notable difference here is that the single-node messenger eventually produces slower turnarounds than the cluster of three nodes.

Overall, this test suggests that performance may be an issue, as clustering will be necessary for fault tolerance. The publisher/messenger/subscriber setup provides a single point of failure with the messengers, so using a single server as a messenger would not be sufficient. Fortunately, this test suggests that a larger cluster provides greater performance than smaller clusters, as the three-node cluster outperformed the two-node cluster, so while the synchronization inherent in clustering presents a performance hit, a large cluster seems to help mitigate this performance hit.

Multiple Publishers Single Subscriber:

We also tested multiple publishers. To do this, we created a one publisher and varied the number of publishers. Each publisher would wait until a set time and begin transmitting its data set. The publishers' data sets were something substantial, 16,000 1 kilobyte messages. We also observed the effects of clustering RabbitMQ nodes, varying the number of nodes from 1 to 3. Each test was run in three trials to ensure accuracy. Publishers were run from the Virginia Tech Computer Science Linux cluster (rlogin cluster).



As expected, the time for publishes to send data increased linearly as more publishers competed for access to the RabbitMQ node. Surprisingly, as more RabbitMQ nodes were added in a cluster, no extra send time seemed apparent. This is likely because the overhead of maintaining synchronization among the nodes of the cluster does not require resources sufficient to slow processing of incoming messages.

It would be interesting to explore this test on a larger scale. That could mean sending more messages from more publishers, and with more nodes in the cluster. It would also be useful to study any difference in behavior if we were to run the publishers from within the same data center as the RabbitMQ cluster.

Single Publisher Multiple Subscribers:

In order to test subscriber throughput, a single publisher was first used to send an increasing message volume to the messaging node. The subscriber would then start up and access the queue while timing how long it would take the subscriber to fully receive the messages. Since the receiving of each message in the client code occurs in O(1) time, the receiving of N messages obviously yielded O(N) results. As N was increased for each test by a factor of 2, the time taken to fully receive the messages increased linearly.

In order to test how clustering affects throughput, multiple subscribers would need to have been run simultaneously to receive messages. Having more than one machine attempting to send messages

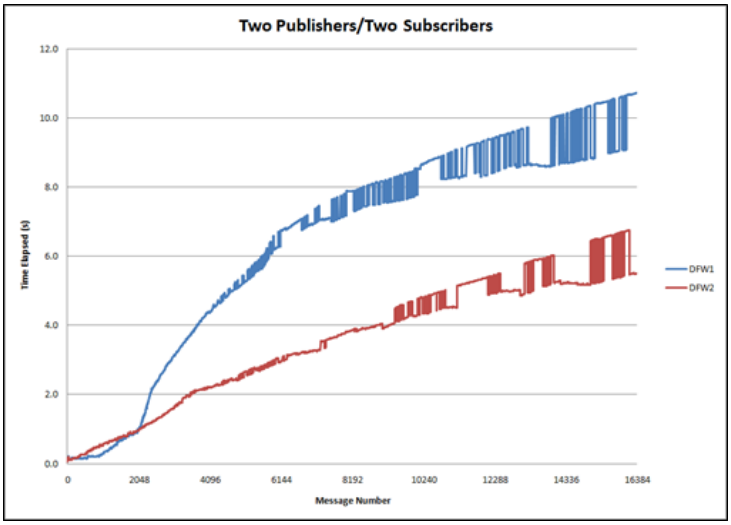
to a single subscriber would do nothing for improving performance, and would instead only slow it down as it would have to synchronize the messages between two or more nodes. However, if there are multiple subscribers, then the messages need to be duplicated across a number of subscribers. Having just one messaging node doing this can cause a delay that, as the message load increases, could surpass the initial delay from synchronizing across clustered servers. Once this initial delay is over, the timing increase as the message volume rises, would be a slower rate than with just one messaging node.

The actual testing of multiple subscribers is incomplete. In order to obtain a successful time comparison, the subscribers would have to start receiving at the same time, otherwise if they are started while the publisher is still sending messages, it could cause publisher delay to be factored in. Also, if the subscribers are not started at the same time, some messages could be lost to some subscribers as the first one receives them and the messaging node does not see any other subscribers at the time. This test can hopefully be completed successfully in the future.

Multiple Publishers Multiple Subscribers:

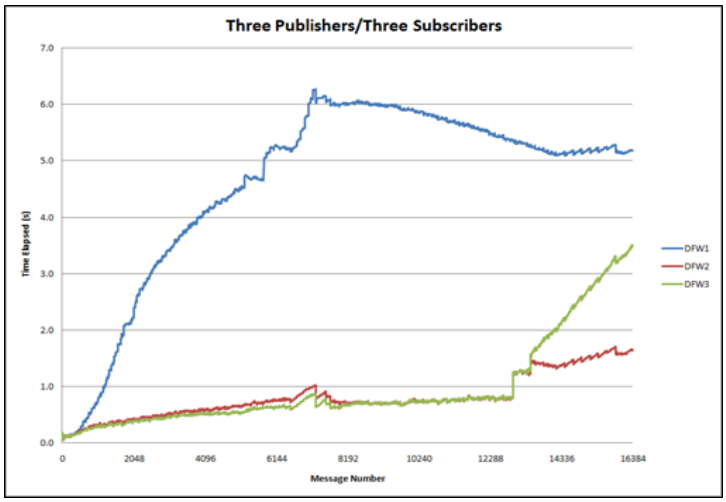
The last test scheme, relative to the sense of testing publishers and subscribers, was to vary the number of subscribers and publishers equally. In order to test this, the testing scheme had to be shifted away from the queue style messenger to the exchange type messenger. The reasoning behind this shift is that in the queue system, only one server can receive messages at a time, but with a exchange system, then all servers can read the messages sent from one server at the same time. With this change made, then the multiple server multiple publisher test was able to be run with numbers that made more sense. (When this test had first been run on a queue style implementation, round trip time was apparently taking upwards of 80 seconds, which did not seem right at all). As in the previous tests, the message scheme used to test was that of messages with size of 1K, containing the message number and the time sent. The test was run off the rlogin cluster to gain better accuracy with a more powerful machine than the earlier server tests.

The test was run with three settings, with a pairing of one subscriber to one publisher, two publishers and subscribers, then finally three subscribers and publishers. The trick to this was that one subscriber could not send to its respective publisher, rather it would send to one of the other publishers, and its respective publisher would get its data from a different subscriber. Though eventually all the data would propagate out to the other subscribers, it helped for timing purposes to send out the data to a different subscriber first to test worst case latency. The size of the message sets were all 16K, since the 4K data sets had shown to be relatively inaccurate as they did not stress the system as much. All three tests were run on the DFW servers, adding a server each time to add the additional publishers and subscribers. To keep a constant, DFW1 was always the head node/disk node, since having one node remain the constant head made more sense for reading data, as the graphs later show. As the 1 subscriber 1 publisher test returned results nearly identical to that of the original single publisher, single subscriber test, even with the exchange vs queue difference, only the 2 publisher/subscriber and 3 publisher/subscriber results will be discussed.



Overall, while still suffering performance hits that result in longer processing times than that during the single publisher, single subscriber tests, the overall time seems to be less than that of the multiple publishers, single subscriber. As was discussed earlier, one node (DFW1) was always kept as a disk node, serving as a backup in case of any issues to the other nodes. However, this meant that the performance on DFW1 would be worse than the performance on the other nodes, as show in Figures 5 and 6. In the two node system, an oddity developed in the receiving order of the graph, which can be seen by the oscillating movements of total time travelled during the test. We were led to believe that this occurred because the exchange might have been confusing which server should be receiving the messages first, as when it was the proper server, the bottom oscillated time should be read, but when it was read from the wrong server, then the top oscillated time should be read. However, even factoring this in, the 2 Publisher/Subscriber test seems to compare fairly well to all versions of the multiple publishers/single subscriber test, as even at the worst time it is about equal to the worst time of one of the tests in that scheme. The not head node (DFW2) actually performed a lot better overall, and timed better than any multiple publisher/subscriber test.

However, the marked improvement appears in the three subscriber/three publisher set. While the head node has a lot more latency, it is less than in the two node system, and after peaking (when the work is presumably done), the overall time decreases by the end. However, the other node show marked improvement, the overall running time being cut in at least half. This is showing that while having the performance drops seen in clustering, having a 1:1 ratio for publishers and subscribers helps offset that cost as more nodes are added in, so having 1 publisher and 1 subscriber for each node in the cluster can improve performance to offset the original drop caused by adding nodes. Based on these predictions, it is possible by allocating more disk nodes with respect to the RAM nodes added, the hits on the disk node could possibly lessen as well. For optimal performance, then it appears best to have a publisher and subscriber for head node in the cluster.



Remote System Monitor:

For a demonstration of how RabbitMQ might be used in a distributed system, we developed a remote system monitoring service on top of the communication framework. This system consisted of a broadcast program on each of the monitored nodes, the RabbitMQ server, and a receiver program. The broadcast program repeatedly sends a message to a RabbitMQ exchange called “rsysmon” at user defined intervals. These messages contain the machines name, a time stamp of when the message was sent, the number of active processes, the total number of processes, and the amount of free memory as according to the system files in /var/proc. The RabbitMQ server then distributes the messages to any machines listening to the rsysmon exchange. The receiver program listens to the incoming messages and maintains a hash table of the currently active machines. If one of the machines in the hash table is not heard from in 10 seconds, the machine is assumed dead and is removed from the table of active nodes. The hash table of active nodes is printed out to an html file on an Apache web server roughly every second.

Theoretically, receiver can monitor as many nodes as desired. However, due to time constraints, we were not able to test this theory. Due to the linear nature of the message reception by the receiver program, there is probably a bottleneck in which too many monitored nodes cause the receiver to

become backed up, and its data output significantly delayed. This might be an area of interest for further research in this project.

Results / Conclusions:

Based on the results of our testing, we believe that RabbitMQ could potentially be used as a communication backbone for a distributed system. The message queuing and exchange systems proved to be reliable in our tests, and upon research into RabbitMQ's API, we found that there are methods to asynchronously confirm reception of messages.

In terms of performance, however, the results were inconclusive. Message passing speeds seemed to generally decrease with the addition of multiple nodes to the RabbitMQ cluster up to a certain number of messages, at which point the cluster began to perform faster than a single node. However, we believe that the primary intent of developers implementing the RabbitMQ clustering capabilities was to increase reliability, which means that the initial decrease in performance of a RabbitMQ cluster vs. a single server would probably be due to data replication across nodes.

Future Recommendations

Through the course of this research, we discovered some tests that we considered worth running, but due to time constraints were unable to include in this project. The RabbitMQ version we used in this project was version 1.8.0, the latest in the Advanced Packaging Tool repository at the time of installation. However, the latest version of RabbitMQ is 2.4.1; we think it would be worthwhile to test the latest version of RabbitMQ to see if any differences in performance arise. Also, in our tests, we used fixed sets of data sent as rapidly as possible until all elements were sent; while this gave us some insights into how RabbitMQ works, it is not a very accurate representation of the load a RabbitMQ messenger cluster is likely to experience. As such, further testing should attempt to model that load more accurately. Finally, we were only able to test up to three-node clusters in a single location; again, while this gave us insights into the scalability of RabbitMQ, a cluster of three servers is not a large scale. Further testing should scale the size of the messenger cluster to much larger numbers and more locations to better discern how larger clusters affects the performance of RabbitMQ. These suggestions should serve as a sufficient starting place for any future research on this topic.

Special thanks to Gabe Westmaas, Rackspace, US Inc., and Virginia Tech for their resources and generous support of this project.