# PYTHON MESSAGING: ACTIVEMQ AND RABBITMQ

Published on 20 October 2009 under ActiveMQ, Messaging, Python, RabbitMQ

I found myself recently with a task of testing various Python messaging clients. In this article I'll present my findings regarding performance and scalability on two Python clients connecting to ActiveMQ and RabbitMQ message brokers.

For ActiveMQ Python client, I used pyactivemq library version 0.1.0. It's basically a Python wrapper for ActiveMQ-CPP library, allowing Python clients to communicate with the broker using both OpenWire (ActiveMQ specific binary protocol) and Stomp (Simple text-oriented protocol) protocols. Since it uses ActiveMQ-CPP, installation requires special attention and some of those details you can find in a related blog post.

As a RabbitMQ client I used py-amqplib version 0.6, since that's currently the most natural solution if you don't want to use frameworks like Twisted.

I used the latest broker versions, which means ActiveMQ 5.3.0 and RabbitMQ 1.7. Also, all numbers shown below are for tests executed on an "average linux desktop box" running Ubuntu 9.0.4.

Now let's go testing.

## Performance

The performance test I used is really simple: One producer, one consumer and one queue. Both producer and consumer tries to do their best in terms of performances and we're not going to use any transactions. I used small text messages, with text `Example message _num_`. The full source code of tests can be found at http://github.com/dejanb/pymsg/. For every client (and broker) there are two Python scripts:

- test_receive_async.py – which starts an asynchronous consumer in a thread and samples consuming rates every 10 seconds
- test_send.py – which starts a producer in a thread and samples producing rates every 10 seconds

Both of these scripts try to get 100 rate samples, which means tests last for about 15 minutes.

### Persistent messages

ActiveMQ 5.3.0 comes with the number of example configuration files that helps you configure it for different usage scenarios. One of those configuration files is called `conf/activemq-throughput.xml` and it is the one that should be used for high performance scenarios.

To start a broker with this configuration file, just execute:

```
bin/activemq xbean:conf/activemq-throughput.xml
```

Now you can run the consumer and producer with

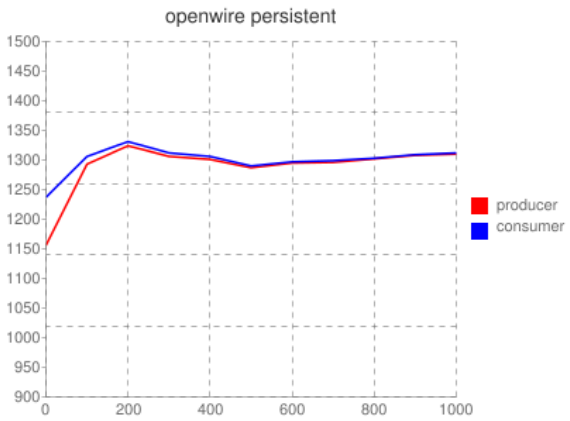
```
python test_receive_async.py
```

and

```
python test_send.py
```

in separate console windows.

When ran like this, the client will use OpenWire protocol. The following diagram shows the result (the full output can be found here).




As you can see, after the initial spike, both producer and consumer have consistent rates of around 1300 msg/sec.

To repeat the same test, now using Stomp wire protocol, we'll execute the scripts with one additional parameter, `stomp` (of course, a clean broker start is needed)

```
python test_receive_async.py stomp
```

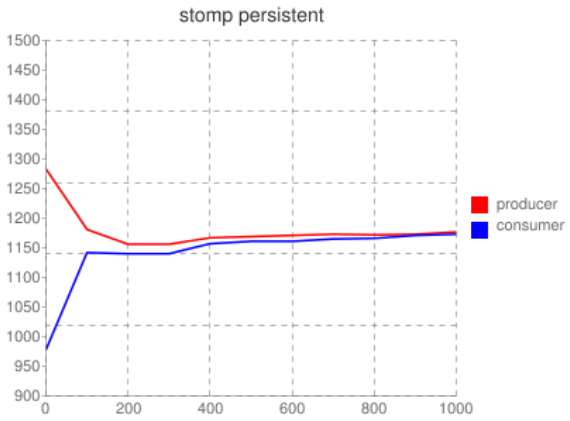
and

```
python test_send.py stomp
```

**Dejan Bosanac**

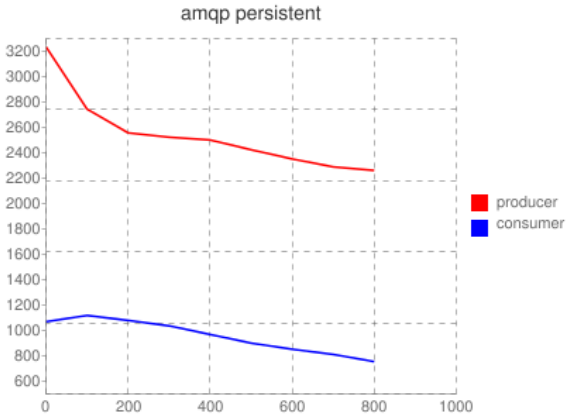
Senior Software Engineer at Red Hat

The results for pyactivemq client using Stomp protocols can be found here and the diagram visualizing those data is shown below.




Again, after the initial glitches, producer and consumer settles on a bit more than 1150 msg/sec. This is an interesting result that shows that with persistent messages the limit is not in the network protocol, but the ability of the broker to successfully persist messages.

Now let's try do the same with RabbitMQ and py-amqplib client. The output with the result samples can be found here and they are visualized on the following diagram.




On the first look you can notice the high message producing rates, starting with more than 3000 msg/sec and slowly decreasing toward 2000 messages per second. On the other hand the consumer is considerably slower at rates around 1000 msg/sec. This situation causes more and more messages being stored in the broker, when it finally crashes after 12-13 minutes. I tried setting Memory-based flow control, but without much success.

The high value on a producer side have one more important drawback regarding reliability. The AMQP protocol send operation is strictly asynchronous, meaning that producer does not have any confirmation from the broker that the message was actually queued. So what I observed is that the number of messages sent shown by the producer and messages stored in the broker are not in sync. You can check this by starting the producer that sends a large number of messages for some time and check the number of messages in the broker. You'll notice that messages arrive to the broker, "long" after the producer has finished its job. So if broker crashes in the meantime, all messages producer thought it sent are lost. I didn't test this with transactions, so I'm not sure if the behavior is somewhat different in this case. My guess it that it is and that is one of the things that I'd like to test further.

Also, RabbitMQ allows you to set the "returned listener" on the channel which can be used by the broker to return messages that were not routed (if the `mandatory` parameter is used during publishing) or could be immediately delivered (if the `immediate` parameter is used during publishing). py-amqplib library is currently missing this functionality, which can be another reliability issue for concern.

So generally, it seems like RabbitMQ have a very high producing rate (sacrificing reliability), while consuming rates are pretty standard and a bit lower than those seen by ActiveMQ consumers.

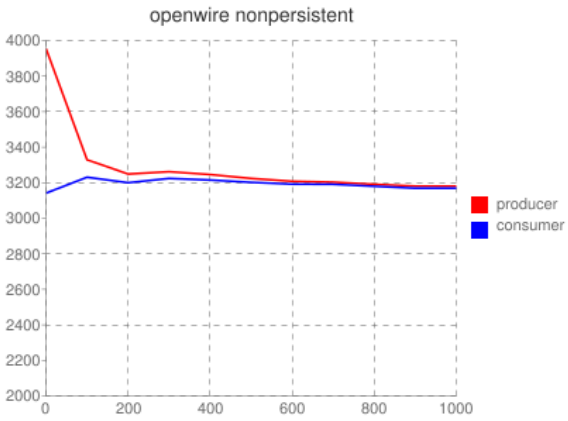
**Non-persistent messages**

Now let's try to do all that again with non-persistent messages. To modify pyactivemq producer to send non-persistent messages, we have to uncomment the following line

```
#self.producer.deliveryMode = DeliveryMode.NON_PERSISTENT
```

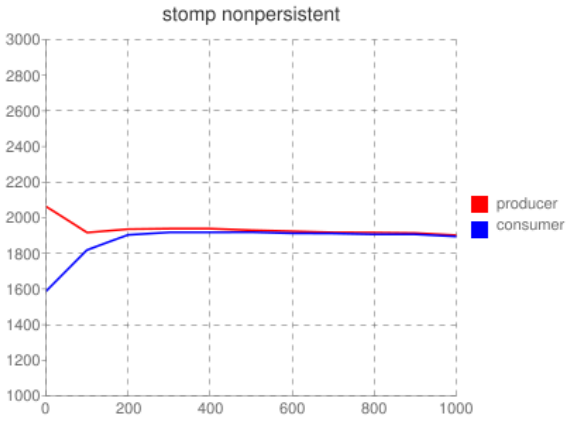
in perftest.py library.

Now we can rerun our tests. For OpenWire protocol results now quite different




showing that the whole throughput of non-persistent messages through the broker is around 3000 msg/sec.

Similarly, the results for Stomp are visualized in the diagram below.



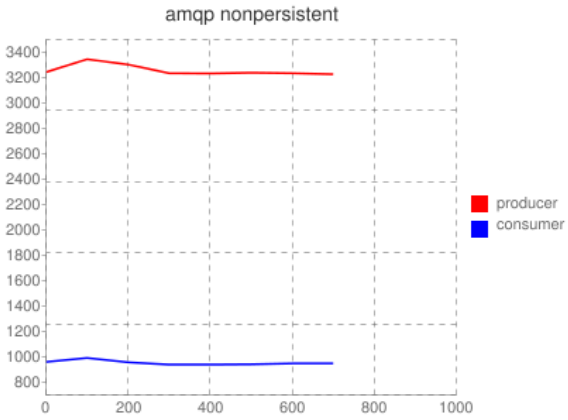

This shows that at around 2000 msg/sec the limitations of Stomp text-oriented protocol kicks in and limit the throughput.

To modify RabbitMQ client to send non-persistent messages, we have to comment the following line

```
msg.properties["delivery_mode"] = 2
```

in the PerfProducer class.

Results now look like this




which is pretty much the same as in the persistent case. This is probably due to the nature of queuing and persisting messages in the broker discussed above. The only difference is that now the broker keep the rates at steady numbers of around 3200 msg/sec for producer and 950 for consumer.

### Scalability

I also tried a simple scalability test against two brokers: try to send a message to as many queues as you can. The test source can be found in the appropriate test_scale.py file for both py-amqplib and pyactivemq libraries.

When you run appropriate

```
python test_scale.py
```

against RabbitMQ you'll get that the message was sent to the around 32000 queues before it crashes, which is very good result.

ActiveMQ, again, comes with the predefined configuration file which should be used if you want to scale your broker (`conf/activemq-scalability.xml`). As it's stated in the configuration file, there a couple of minor changes needed to be made to the startup script in order to achieve maximum scalability. We'll use these

```
ACTIVEMQ_OPTS="-Xmx1024M -Dorg.apache.activemq.UseDedicatedTaskRunner=false"
```

properties in order to give broker a bit more memory and turn of dedicated task runner.

When started with the configuration like this, the

```
python test_scale.py
```

shows that message was sent to impressive 64000 queues before it crashed.

We can try the scalability of Stomp protocol as well. In order to do that, we should add

```
<transportConnector name="stomp+nio" uri="stomp+nio://0.0.0.0:61613"/>
```

to the list of available connectors.

Now we can start the test again

```
python test_scale.py stomp
```

and we'll get the result very similar to the one of the default OpenWire connector.

### Conclusion

To conclude, both ActiveMQ and RabbitMQ are decent brokers that will serve their purpose well in normal conditions, but put to their extremes in terms of throughput, scalability and reliablilty, ActiveMQ currently outperforms RabbitMQ for messaging usage in Python. All tests performed here are done with honest intentions of providing realistic results. If you spot any problems or have any ideas what more can be tested (transactions, etc.), please let me know.



### 24 Responses to "Python messaging: ActiveMQ and RabbitMQ"

Benjamin W. Smith
20 October 2009

Great post, thanks for all the data!

I'd be interested in seeing how my pure Python STOMP client library stacks up http://bitbucket.org/benjaminws/python-stomp/ Think I might run some tests!

Reply

Dejan Bosanac
20 October 2009

Benjamin, that would be great. Looking forward to your results.

Reply

Michael Carter
20 October 2009

You should benchmark morbidQ as well to see how the pure-python message queue stacks up.

Reply

Kevin Nuckolls
21 October 2009

Did you ever find a way to confirm that a message had been published successfully to the mq?

Reply

Joachim Schipper
21 October 2009

Benchmarks are interesting, but *very* hard to get right. I'm afraid that I have some doubts about your results. (Note: I don't know much about either message queue, and I'm going by your graphs only.)

First, an innocuous error: the first "stomp persistent" graph clearly levels out at ~1150, not ~1500 msgs/sec (at least, I assume that's what you are measuring?). This is probably a typo.

More importantly, based on your benchmarks, these pieces of software have different (default?) behaviour under heavy load: ActiveMQ chokes and RabbitMQ falls over. In your test – sending messages quicker than the queue can handle for a long time – the first strategy is better; but if you have a load spike, RabbitMQ will allow you to offload everything into the queue quickly and get your submitting processes back to doing useful work.

Imagine that you are running a website which requires a message queue and which gets Slashdotted. If you are running ActiveMQ, the queue will handle this fine but you will see more and more frontend processes (mod_php/paste/mongrel/…) waiting on the queue. After a while, you run out of frontend processes and the site becomes unbearably slow.
If, on the other hand, you are running RabbitMQ, the frontend processes will quickly put data into the queue and your site will remain reachable. If the queue gets too deep, though, RabbitMQ crashes and the site becomes completely unreachable and/or the backend stops working (depending on how you handle such failures).

For short load spikes, you'd prefer RabbitMQ: sure, you have a large queue, but the queue can be processed once the load spike is over and your frontend remains responsive.

Finally, your benchmarks suggest that the maximum sustained throughput for either implementation is about ~1200 msgs/sec, which makes RabbitMQ look decidedly less bad.

(Also, very few websites should produce ~1200 msgs/sec!)

Reply

Dejan Bosanac
21 October 2009

Hi Michael, I didn't have resources to benchmark MorbidQ at this point, but it would be interesting to see the numbers.

Reply

Dejan Bosanac
21 October 2009

Hi Kevin, ActiveMQ uses synchronous sends by default which means that producer waits for the broker confirmation that the message was successfully queued.

Reply

Dejan Bosanac
21 October 2009

Hi Joachim,

fixed the typo regarding Stomp, thanks.

Regarding your notes on the benchmark results, I agree that it is hard to do them right for all use cases. That's why I stated what I'm trying to achieve with this small benchmark.

ActiveMQ doesn't choke under the load, it holds steady rates on both producing and consuming sides. The numbers for reliable persistent messaging are around 1300 msg/sec. The usecase you're describing is more the situation where you'd be using non-persistent messaging and that's where ActiveMQ (used with OpenWire) can handle 3000 msg/sec for a very long time (not in a spike).

Also, ActiveMQ can be tuned to handle load spikes (sacrificing reliability), using async sends and turning off producer flow control, but that's indeed the topic for some other discussion.

Reply

Rob Davies
21 October 2009

A couple of things worth pointing out about ActiveMQ – by default the contract it enforces is that when messages are sent persistently, it is written to store before a receipt is sent back to the publisher. So you know that the publish is successful. If you don't do this, the message could be lost anywhere on the wire or while awaiting to be persisted by the broker.

Btw – RabbitMQ must be persisting messages asynchronously in its broker – independent from receiving a message, else the publish rates would be very different between persistent and non-persistent delivery. Is that a bad thing ? – Not if you have applications that can either recover from message loss – or that don't require reliable delivery. In which case – why not use non-persistent messages?

ActiveMQ also has a unique cursoring mechanism for message storage – it doesn't hold either a copy of the message or a permanent reference to the stored message – because both of these actually restrict how big a message queue can get by the amount of memory the broker has available. The upper limit for queues is bound by how much disk space you have available. ActiveMQ will only block a producer when it reaches the store limit. There are memory limits available too – but these are used to limit how many persistent messages will be cursored into the broker to await dispatch.

For the non-persistent case, messages can be optionally off-lined to disk if a memory limit is reached.

Reply

Artur Siekielski
21 October 2009

I have also played with ActiveMQ and RabbitMQ with Python (I've used ActiveMQ from Python and Java in production environments). I haven't made benchmarks as performance is totally sufficient for my purposes. But I always have got problems with ActiveMQ. Sometimes consumers stopped to get messages or some other deadlock happened. I have rewritten mq code several times and never got fully stable situation. These situations were rare, eg. one per week or one per 1mln of messages, but it's not acceptable for production. There are a lot of stability issues in ActiveMQ's bugtracker. After this (and other) experiences I'm careful with Java threaded servers.

I haven't used RabbitMQ in production yet but it in tests I haven't encountered deadlock sitution yet, and Erlang OTP promises rock solid stability. The broker looks much lighter and has direct client library for Python.

Reply

Artur Siekielski
21 October 2009

Rob Davies, what you suggest that persistency in RabbitMQ is not supported, is not true. It depends on transaction mode you use. See this from mailinglist:

I should also point out that AMQP's basic.publish is fundamentally
asynchronous and thus limited in the guarantees it provides. You get
stronger guarantees when wrapping your publish requests in transactions

(see channel.tx{Select,Commit,Rollback}() in the Java API). The server guarantees that persistent messages have been written to disk by the time it returns tx.commit-ok, though obviously this carries a performance penalty.

Reply

Rob Davies
21 October 2009

Hi Artur,
I don't think I suggested anywhere that RabbitMQ didn't support persistence, just highlighting the difference between ActiveMQ and RabbitMQ for non-transacted messaging (which is what the benchmark did).

I hope you get the opportunity to try ActiveMQ 5.3 – its been through a lot of testing and production deployments (indirectly as the Fuse Message Broker) – and this release is a lot more stable because of that.

cheers,

Rob

Reply

alexis
23 October 2009

Hi everyone,

Alexis here from RabbitMQ.. A few quick comments..

First off thanks to Dejan for a great write up. This kind of work makes it much easier for customers to replicate tests, and try out other configurations for themselves.

One thing that really jumps out from Dejan's piece is that both RabbitMQ and ActiveMQ are quite straightforward to set up and do useful, comparable, predictable work with. This is a good win for open source message brokers.

However we do have a few points that should be considered by people looking at running these tests for themselves:

The broker set-up in RabbitMQ is not 'like for like' with ActiveMQ. Dejan – I realise that you are an ActiveMQ expert, and using RabbitMQ for the first time, but you have used a specific configuration for ActiveMQ and compared it with a default configuration for RabbitMQ. Unless you want to use a default configuration for ActiveMQ, it might be better to use a custom configuration for RabbitMQ as well – we would be happy to help.

For example, if you want to see better numbers with RabbitMQ you can set 'NoAck'. We just ran a quick test on a desktop and a python client consumed at 2,447 mps with acks turned off, vs, on the same platform, 1,708 mps with acks turned on.

The numbers, of course, provide no guide as to behaviour on other platforms. But this makes quite a big difference. For example if you want high performance you might be better off using a C client. Java and .NET are not bad either. It's pretty easy to get much higher message rates on those platforms, if that is what matters for your business.

As to Python specifically: there are quite a few Python clients for RabbitMQ. They all perform in different ways, and some have very different designs: eg txAMQP for Twisted. I think it is fair to say that none of them are optimised for speed. All the RabbitMQ clients are pure Python. In the case of ActiveMQ, Dejan used ActiveMQ-CPP. If this uses C++ as the name suggests, then it might be more useful to compare it with the RabbitMQ C client.

As a rule of thumb, single producer / single consumer configurations are rare in cases where performance matters. It would be interesting to see other tests, e.g. running multiple consumers per producer, with some kind of fanout.

Rob, the reason the RabbitMQ numbers for persistent and non-persistent are quite similar is that the RabbitMQ broker is not being overloaded because the clients being used are only able to produce messages at a certain rate in the tests. To get higher numbers for non-persistent messaging with RabbitMQ, you need faster clients or more appropriate test harnesses, or both. If you switch to C clients on reasonable hardware you should start to see a bigger difference between persistent and non-persistent cases.

By all means ignore all of the above if you don't care about performance.

If you have any questions or comments on the above, please list them in the comments here.

If you care about test cases that differ in any way from that described in the blog post, then please contact us or contact Dejan directly – or comment here.

Cheers,

alexis
RabbitMQ

Reply

Hiram Chirino
23 October 2009

Hi Alexis,

Doing apples to apples comparisons of middle-ware can be difficult. But given the problem space "Python Messaging", I think the test setup was pretty fair. I think your 1st, 2nd, and 5th points should be discussed further as they may make it a more apples to apples comparison.

Point 3, 4 totally not in the "Python Messaging" arena, so I think they be safely discarded for purposes of this benchmark. Point 6 seems to allude that the poor quality of the client implementation is what is limiting the test's rate. I think the client implementations are an important part of middle-ware evaluation. If you can suggest a better python client to use against RabbitMQ, please do so.

Regards,

Hiram

Reply

alexis
24 October 2009

Thanks Hiram,

Happy to discuss this further.

Meanwhile, I guess my main points can be summarised as follows (leaving aside the issue of broker configuration tuning for now).

You are trying to test two brokers that can both handle higher loads.
You are using a single producer, single consumer test case, where the client is the bottleneck, and which does not overload the brokers.
Therefore you are really *comparing the client throughputs*.
Both brokers have lots of Python type clients. AFAIK, none of them are optimised for raw speed. So I am not sure how much we learn from comparing these clients. Moreover I am less sure how useful the numbers are if you use different brokers to compare them. But if you are certain that it's the right thing to do, then you may want to compare clients like for like:

If the ActiveMQ-CPP client is a pure Python client then by all means compare it with a pure Python RabbitMQ client.
If the ActiveMQ-CPP client is hybrid Python/C++ then compare it with a hybrid client, or with a C/C++ client, whichever is more appropriate.
Consider also looking at Async style clients like Twisted (which is Python btw)
I think your work has a lot of merit – it would just be good to be clear about what is being tested, and to what end.

Cheers

alexis

Reply

Dejan Bosanac
26 October 2009

Hi Alexis,

thanks for your feedback. Just a few more comments and clarifications from my side.

Of course that no test can cover all possible use cases of a complex domain such as asynchronous messaging, but here I described a case I was interested in at the moment (not so rare IMHO) and thought I'd share results. It would be great to do more testing of other various aspects, such as transactions, clusters, different number of consumers/producers, various platforms such as C, Java, LAMP, .NET, etc. But all this require a lot of resources and definitely cannot be covered with one blog post. I hope I'll find some more time in the future to cover some of these topics.

As for this concrete test, I didn't use any specific ActiveMQ features for tuning performance of producers/consumers, such as "asynchronous send" or "optimized acknowledge". I just merely started a broker with different configuration files distributed with the broker.

Also, I was interested in Python messaging at the moment. The py-activemq client relies on ActiveMQ-CPP, but it is still a regular Python client and I think the best solution for connecting to ActiveMQ. That's why it was used in the test.

Again, thanks for your valuable input and I hope we'll have more of these constructive tests and discussions in the future to the benefit of our users.

Cheers,
Dejan

Reply

alexis
26 October 2009

Dejan,

Thanks, that is all very useful info. I am glad that ActiveMQ-CPP is pure Python. Forgive me for being confused by the 'CPP' part of its name 

I appreciate that you cannot test all possible use cases and have to start somewhere. My main point is that your test is really testing the clients not the brokers. Quite frankly I would be a more than a little surprised if RabbitMQ, ActiveMQ, Qpid and other 'general purpose' open source message brokers exhibited significant differences in throughput for a whole set of these sorts of cases.

alexis

Reply

Dejan Bosanac
26 October 2009

Alexis,

sorry if I wasn't clear, py-activemq relies on C++ lib underneath, but that doesn't mean it isn't a "regular" Python client, right?

I get your point regarding brokers vs. clients, but generally I was interested in what would user see in this particular use case.

Cheers
Dejan

Reply

alexis
27 October 2009

Dejan,

Oh, sorry, I did not read your previous comment with sufficient care and had misunderstood.

So, to recap:

ActiveMQ-CPP is a C++ library
py-activemq is Python but uses ActiveMQ-CPP underneath
If the use of ActiveMQ-CPP is to improve py-activemq client performance then I would not call py-activemq a ‘regular’ or ‘native Python’ client. Instead I would call it a ‘hybrid’ Python/C++ client whose purpose is to achieve higher performance.

I understand the use case you wanted to demonstrate.

alexis

Reply

Abbas
18 August 2012

Hi,

Has someone tried benchmarking persistent MQ on a server using SSD for message data storage?

Thanks,
Abbas

Reply

Dejan
18 August 2012

Hi Abbas, I haven’t seen any. You can take a look at Hiram’s benchmarks

http://hiramchirino.com/blog/2011/12/stomp-messaging-benchmarks-activemq-vs-apollo-vs-hornetq-vs-rabbitmq/

and execute them in your environment

Reply

## Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

SUBMIT