Testing Active MQ broker - report

Wojciech Czech

August 9, 2007

Introduction

This document presents results of experiments on Active MQ massage broker. The aim of our tests was to evaluate capabilities of broker in different communication scenarios. The main subject of our interests is number of messages broker can forward per second. We also try to investigate how this measure changes with increasing number of parallel producers and consumers or message topics.

Experiments

The tested communication is asynchronous, without subscriber acknowledgments, receipts and persistent messages.

Software

The testing framework consists of several elements:

- AMQ message broker 5.0 running inside Tomcat 5.5.
- Simple STOMP publisher, written as shell script using `netcat` program to write STOMP messages directly to open socket of AMQ STOMP connector.
- Simple STOMP subscriber written with the use of `netcat` program, getting messages directly from STOMP connector.
- STOMP publisher written in Python which parses gFTP logs and sends messages created on this basis to specified topic.
- STOMP subscriber written in Python which get get messages from any topic.
- Message summarizer (Java, JMS API) which consumes messages from given topic and counts how many messages it received in given period (e.g., per second), then it publishes this information on given topic. This program allows us to measure performance of broker in different configurations. We can also use the variant of summarizer which performs a little bit more processing - it consumes gFTP logs and counts number of bytes forwarded by given gFTP server within specified period.
- Python message counter (similar to one written in Java)

Hardware

- `lxb6117`: two CPUs Intel Xeon 2.80GHz, 2GB RAM
- `lxb6118`: two CPUs Intel Xeon 2.80GHz, 2GB RAM
- *nemrod*: Intel Core Duo 2 1.83GHz, 2GB RAM
- `lxplus096`: two CPUs Intel Xeon 2.80GHz, 2GB RAM
- `lxplus097`: two CPUs Intel Xeon 2.80GHz, 2GB RAM
- `lxb6117` connected to `lxb6118` via Gigabit Ethernet

Results

Test 1a: single subscriber - single publisher

Configuration

- `lxb6117` - broker
- `lxb6118` - netcat producer, prepared set of 1048578 39B messages, sending time measured with the use of `time` program
- `nemrod` - netcat consumer
- message size: 39B

Results

- average CPU load (broker process): 101%
- average MEM usage (broker process): 19.6%
- messages per second: 18489.16 (producer)

Test 1b: single subscriber - single publisher, scaling with message size

Configuration

- `lxb6117` - broker
- `lxb6118` - netcat producer, prepared set of 3524578 1KB messages, sending time measured with the use of `time` program
- `nemrod` - netcat consumer
- message size: 1KB

Results

- average CPU load (broker process): 101%
- average MEM usage (broker process): 20.7%
- messages per second: 18108.19 (producer)

Test 1c: single subscriber - single publisher, scaling with message size

Configuration

- `lxb6117` - broker
- `lxb6118` - netcat producer, prepared set of 40357 10KB messages, sending time measured with the use of `time` program, netcat consumer
- message size: 10KB

Results

- average CPU load (broker process): 128%
- average MEM usage (broker process): 20.6%
- messages per second: 1841.10 (producer)

Test 2: client written in Java, consumer counts messages got per second

Configuration

- `lxb6117` - broker
- `lxb6118` - Java summarizer (counts number of messages caught per second), netcat producer, python consumer (gets messages published by summarizer)
- message size: 39B

Results

- average CPU load (broker process): 117%
- average MEM usage (broker process): 19.6%
- average CPU load (summarizer process): 63%
- average MEM usage (summarizer process): 1.4%
- summarizer gets approximately 9500 messages per second

Test 3: scaling with number of producers

Configuration

- `lxb6117` - broker
- `lxb6118` - Java summarizer, 10 netcat producers, python consumer (gets messages published by summarizer)
- message size: 39B

Results

- average CPU load (broker process): 165%

- average MEM usage (broker process): 19.6%
- average CPU load (summarizer process): 70%
- average MEM usage (summarizer process): 1.4%
- summarizer gets approximately 12000 messages per second

Test 4: scaling with number of producers

Configuration

- `lxb6117` - broker
- `lxb6118` - Java summarizer, 100 netcat producers, python consumer (gets messages published by summarizer)
- message size: 39B

Results

- average CPU load (broker process): 169%
- average MEM usage (broker process): 20.7%
- average CPU load (summarizer process): 75%
- average MEM usage (summarizer process): 2.6%
- summarizer gets approximately 12000 messages per second

Test 5: producer written in Python

- `lxb6117` - broker
- `lxb6118` - Java summarizer, python producer, python consumer (gets messages published by summarizer)
- `nemrod` - python producer
- message size: 39B

Results

- average CPU load (broker process): 191%
- average MEM usage (broker process): 20.7%
- average CPU load (summarizer process): 67%
- average MEM usage (summarizer process): 1.4%
- summarizer gets approximately 12500 messages per second

Test 6: scaling with number of producers

- `lxb6117` - broker
- `lxb6118` - Java summarizer, 20 python producers, python consumer (gets messages published by summarizer)
- `nemrod` - 2 python producers
- message size: 39B

Results

- average CPU load (broker process): 141%
- average MEM usage (broker process): 20.7%
- average CPU load (summarizer process): 47%
- average MEM usage (summarizer process): 1.6%
- summarizer gets approximately 9000 - 11000 messages per second with short, periodic decreases to 3000 msg/s

Test 7: scaling with number of producers

- `lxb6117` - broker
- `lxb6118` - Java summarizer, 20 python producers, python consumer (gets messages published by summarizer)
- `nemrod` - 20 python producers
- message size: 39B

Results

- average CPU load (broker process): 115%
- average MEM usage (broker process): 20.7%
- average CPU load (summarizer process): 45%
- average MEM usage (summarizer process): 4%
- summarizer gets approximately 5000 - 11800 messages per second, strong fluctuations

Test 8: scaling with number of consumers

- `lxb6117` - broker
- `lxb6118` - netcat producer, 2 netcat consumers
- `nemrod` - 2 netcat consumers
- message size: 39B

Results

- average CPU load (broker process): 101%
- average MEM usage (broker process): 20.7%
- producer sends 18483.62 msgs/s (1048576 messages in 56.733s)

Test 9a: scaling with number of consumers

- `lxb6117` - broker
- `lxb6118` - Java summarizer, python consumer (gets messages published by summarizer)
- `lxplus96` - python consumer
- `lxplus97` - python consumer
- message size: 39B

Results

- average CPU load (broker process): 30%
- average MEM usage (broker process): 20.6%
- summarizer gets at the beginning approximately 6000 messages per second, after a while the efficiency decreases to approximately 400 msgs/s

Test 9b: scaling with number of consumers

- `lxb6117` - broker
- `lxb6118` - Java summarizer, python consumer (gets messages published by summarizer)
- `lxplus96` - Java summarizer
- `lxplus97` - Java summarizer
- message size: 39B

Results

- average CPU load (broker process): 130%
- average MEM usage (broker process): 20.2%
- each summarizer gets approximately 5000 messages per second (what gives 15000 msgs/sec) and the efficiency does not decrease (as in case of Python subscribers)

Test 10: scaling with number of topics

- `lxb6117` - broker
- `lxb6118` - Java summarizer, python consumer (gets messages published by summarizer)
- 4 different topics
- 4 pairs producer/consumer: (`lxb6118` - Java summarizer, `nemrod` - netcat), (`lxplus97` python producer, `lxplus96` python consumer), (`lxplus96` python producer, `lxplus97` python consumer), (`nemrod` python producer, `lxb6118` python consumer)
- message size: 39B

Results

- average CPU load (broker process): 60%
- average MEM usage (broker process): 20.7%
- summarizer gets at the beginning approximately 6000 messages per second, after a while the efficiency decreases to approximately 400 msgs/s

Durable subscriptions

Persistent messages can be provided by connecting AMQ broker to database engine. With the use of JDBC any database can be used, for instance MySQL, Oracle or Postgres. AMQ can use journaling to improve performance of data storing. The broker running at `lxb6117` uses MySQL database running at the same machine and high performance journal. If external database is not available, the embedded persistence mechanism can be exploited: Kaha or Derby database.

Durable topic subscriptions were tested with STOMP Python clients. To provide persistent messages the following conditions should be satisfied

- The SEND message (producer) contains header `persistent:true`,
- The CONNECT message (subscriber) contains not empty header `client-id`,
- The SUBSCRIBE message (subscriber) contains non empty header `activemq.subscriptionName` (`durable-subscriber-name` described on `http://stomp.codehaus.org/Stomp+JMS` does not work).

The headers `client-id` and `activemq.subscriptionName` form identifier of durable subscription, which allows for delivering messages published between disconnection and next connection of client.

Selectors

Filtering messages using SQL 92 syntax (similarly as in JMS 1.1) is provided via `selector` header in SUBSCRIBE STOMP message. Unfortunately, because STOMP is text based, the numbers in selectors do not work, only string-based comparison is possible.

High availability and Networks of Brokers

High availability and fault tolerance is provided by Master-Slave mechanisms. We can use three kinds of configurations

- Pure Master-Slave
  (`http://activemq.apache.org/masterslave.html`),
- Shared File System Master-Slave
  (`http://activemq.apache.org/shared-file-system-master-slave.html`),
- JDBC Master-Slave
  (`http://activemq.apache.org/jdbc-master-slave.html`).

The first configuration supports only one Slave with full replication of Master state. The Slave is connected to Master, being able to discover Master's failure. The client using special failover transport (Java) can dynamically switch broker without need to restart. The Pure Master-Slave mechanism was tested with the use of Java client. Unfortunately the Python STOMP library we use, does not provide dynamic failover for publishers.

The Shared File System Master-Slave uses exclusive locks is shared broker data directory to provide mutual exclusion. First broker which gets the lock becomes Master, other brokers have to wait. Similar concept is used in case of JDBC Master-Slave, when Master broker gets exclusive lock to database. The main disadvantage of JDBC Master-Slave mechanism is that high performace journaling cannot be used.

Networks of Brokers mechanism provides scalability by enabling distributed queues and topics. The network of AMQ brokers can be treated as IP network, where brokers are similar to routers. The publisher and subscriber can be connected to different brokers and the message will be forwarded and delivered accordingly. Networks of Brokers do not provide full fault tolerance, because the message located at given moment on broker can be lost if it fails. However, because of brokers redundancy we obtain high availability, with clients choosing one of available brokers to connect to or using failover transport (dynamic switching broker after disconnection). Network of Brokers connections can be configured similarly as routing tables. One can specify TTL, per-topic rules etc. Clients can connect to network by static list of brokers' URL's, via multicast discovery agent or with the use of Zeroconf discovery.

Problems

- STOMP Python subscribers

  The results of performance tests with Python subscribers are different from the others, e.g., Test 9a vs. Test 9b. It appears that Python subscriber is not able to take as many as 10000 messages per second. For instance, in Test9a at the beginning, we get about 6000 mgs/s (according to our expectations) but then we observe significant slowdown. Interestingly, neither broker machine nor subscriber machine has major CPU load. The throughput of network connecting broker and subscriber is also sufficient to handle 10000 messages per second. The main problem is that client has negative impact on broker and whole communication.

The most probable reason for this slowdown are TCP connection issues or broker waiting for acknowledgments. I tried to overcome this problem by setting `activemq.prefetchSize` header (which potentially accelerate consumer) but it did not change anything. Moreover, when we take into account results of Test 1a in which we obtained 18000 msgs/s without setting any additional headers, it seems that this is not a cause of the problem.

- Numbers in selectors - do not work
- Failover in Python subscribers

Summary

- AMQ broker provides fast message forwarding with approximate rate 15000 msg/s.
- Upgrade to version 5.0 did not change performance significantly.
- Message forwarding scales well with number of producers, subscribers or topics. Better scalability can be obtained with the use of networks of brokers.
- Durable subscriptions (STOMP) work fine, provided that required headers are added to STOMP messages.
- One can use subscriptions with filters, however when using STOMP only string-based comparison is available.