Apache ActiveMQ: Introduction and Experience

Andy Huntwork
Brian Long

Agenda

- What is ActiveMQ?
- What is Messaging?
- What is JMS?
- How do you use it?
- How can you configure it?
- How does it perform?
- How can you monitor it?
- Experience and Recommendations
- Resources

What is ActiveMQ?

- "Apache ActiveMQ is a fast open source JMS 1.1 Message Fabric which supports clustering, peer networks, discovery, TCP, SSL, multicast, persistence, XA and integrates seamlessly into J2EE 1.4 containers, light weight containers and any Java Virtual Machine together with having a host of Cross Language Clients. "

  -- http://www.activemq.org

What is Messaging?

- Most generally, loosely-coupled communication between two or more entities.
- *Generally* connotes:
  - Middleware solution
  - Use in the context of distributed computing
  - Ability to operate in a heterogeneous environment
  - Coordination by a broker or agent
  - Messages can be stored, routed, transformed
  - Availability and delivery guarantees
  - Asynchronous communication

What is JMS?

- Message oriented middleware for sending **asynchronous** messages between two or more clients.
- JSR 914.
- Elements : providers, clients, producers, consumers, messages, queues, and topics.
- Supports point-to-point (queue) and publish/subscribe models (topics).

What is JMS? (Consumer/Receiver)

```
…
public void receive() throws Exception {
   ConnectionFactory factory = . . . ;
   Connection connection = factory.createConnection();
   Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
   // listen on our 'java' queue
   Destination destination = session.createQueue(receiveQueueName);
   MessageConsumer consumer = session.createConsumer(destination);
   consumer.setMessageListener(new MessageListener() {
      public void onMessage(Message message) {
         try {
            System.out.println("Got message '" + ((TextMessage)message).getText()+"'");
      } catch(JMSException e) {
            System.out.println("exception " + e);
         }
      }
   });
   connection.start();
   System.out.println("Listening for 15s");
   Thread.sleep(15000);
   session.close();
   Connection.close();
}
…
```

What is JMS? (Sender/Producer)

```
…
public void send() throws Exception {
   ConnectionFactory factory = . . .;
   Connection connection = factory.createConnection();
   Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

   Queue queue = . . .;
   MessageProducer producer = session.createProducer(queue);

   // create our message
   Message message = session.createTextMessage("Greetings from JMS!");

   // construct a reply-to destination - our java queue - on the message. this is optional!
   Destination destination = session.createQueue(receiveQueueName);
   message.setJMSReplyTo(destination);

   producer.send(queue,message);

   session.close();
   connection.close();
}
…
```

What is ActiveMQ?

- Broker based
  - Producers deliver messages to a broker
  - Consumers receive messages from a broker
- Supports high availability through the use of redundant brokers
- Provides the JMS implementation for Apache Geronimo

Transport Options

- ActiveMQ supports multiple transport configurations.
  - Transport URIs:
    - in-process/VM – for communicating with in-process broker
    - IP multicast – needs reliability layer added.
    - TCP, UDP, HTTP, SSL, and more…
  - Layered URIs:
    - Reliable/Failover – reconnect, resend, reorder
    - List/Fanout – adds reconnect + replication logic.
    - Discovery – adds reliability + use of URI discovery agent.

Transport URIs

- Basic Transport URIs:

```
TCP:
      tcp://localhost:61616,tcp://remotehost:61616

VM/In-Process:
      vm://broker1?marshal=false&broker.persistent=false
```

- Layered Transport URIs:

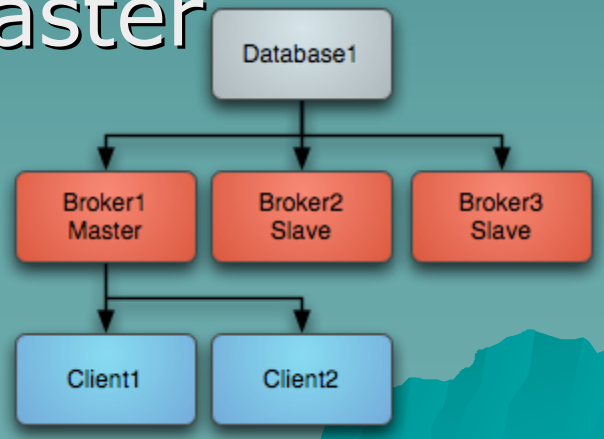
```
TCP w/ Failover:
      failover:(tcp://localhost:61616,tcp://remotehost:61616)?initialReconnectDelay=100

Multicast w/ Discovery:
      discovery:(multicast://default)?initialReconnectDelay=100
```

Topologies

- ActiveMQ supports multiple broker configurations
  - Single broker
  - Active broker with passive failover broker
  - Fully connected network of brokers
  - Peer To Peer

Master/Slave

- All brokers try to acquire a lock on the database
- First one wins
- Lock released on broker death
- Acquired by the next master



Network of Brokers

- Composed of multiple active brokers
- Producers and consumers connect to any broker
- Messages are relayed from broker to broker
- Trivial to configure
  - Multicast discovery; or
  - Fixed list of brokers; or
  - Pluggable discovery mechanism

Running the Broker

- Configure the brokers
  - Write a Spring XML config file.
  - Really it's Xbeans translating to Spring XML
  - Basic configuration is about 10 lines.
- Run the broker application
  - Config can come from the file system or classpath through xbean: URI.
- Examples…

Minimal Broker Configuration And Invocation

- Basic config runs with JMX, journaled JDBC for persistence, on multicast with the default transport connector.

```
<beans xmlns="http://activemq.org/config/1.0">
  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>
  <broker useJmx="true">
    <managementContext>
       <managementContext connectorPort="1099" jmxDomainName="org.apache.activemq"/>
    </managementContext>
    <persistenceAdapter>
       <journaledJDBC journalLogFiles="5" dataDirectory="${activemq.home}/activemq-data"/>
    </persistenceAdapter>
   <transportConnectors>
       <transportConnector name="default" uri="tcp://localhost:61616"/>
    </transportConnectors>
    <networkConnectors>
       <networkConnector name="default" uri="multicast://default"/>
    </networkConnectors>
  </broker>
</beans>
```

- Run the broker:

```
C:\proj\incubator-activemq-4.0.2\bin>activemq xbean:file:/proj/activemq/minimal-broker-config.xml
```

A Closer Look At Broker Config

```
<beans xmlns="http://activemq.org/config/1.0">
  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>
  <broker useJmx="true"> <!- enable JMX in the broker -->
    <managementContext>
      <!-- define the connection port for JMX and the name of the broker's MBean -->
      <managementContext connectorPort="1099" jmxDomainName="org.apache.activemq"/>
    </managementContext>
    <persistenceAdapter>
      <!-- ${activemq.home} is dereferenced via PropertyPlaceholderConfigurer -->
      <journaledJDBC journalLogFiles="5" dataDirectory="${activemq.home}/activemq-data"/>
    </persistenceAdapter>
   <transportConnectors>
      <!-- our clients will connect to use at these endpoints -->
      <transportConnector name="default" uri="tcp://localhost:61616" discoveryUri="multicast://default"/>
    </transportConnectors>
    <networkConnectors>
      <!-- specify a mechanism for connecting to other brokers -->
      <networkConnector name="default" uri="multicast://default"/>
    </networkConnectors>
  </broker>
</beans>
```

- These beans and attributes are only the tip of the iceberg…
- (see http://activemq.apache.org/xbean-xml-reference-41.html)

How do you use it from Java?

- Add 8+ jars to your classpath.
- Create a connection:

```
new ActiveMQConnectionFactory ("tcp://localhost:61616")
```

- Create a destination

```
new ActiveMQTopic("topic://my.topic.name")
```

- Now send and receive using the JMS API.

- Next, some examples, which show basic queue production/consumption and Java/Ruby interop through the broker!

How do you use it from Java? (Main)

```
import javax.jms.*;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQQueue;

public class QueueSenderReceiver {

   String broker = "tcp://localhost:61616"; // should match URI of our transportConnector
   String receiveQueueName = "java"; // for listening to messages sent to us.
   String sendQueueName = "ruby";    // for sending messages to our ruby counterpart.

   public void receive() throws Exception {
        // … we'll fill in these details on the next slides
   }

   public void send() throws Exception {
        // … we'll fill in these details on the next slides
   }

   public static void main(String args[]) throws Exception {
      QueueSenderReceiver qsr = new QueueSenderReceiver();
      qsr.send();
      qsr.receive();
   }
}
```

How do you use it from Java? (Receiver/Consumer)

```
…
public void receive() throws Exception {
   ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(broker);
   Connection connection = factory.createConnection();
   Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
   // listen on our 'java' queue
   Destination destination = session.createQueue(receiveQueueName);
   MessageConsumer consumer = session.createConsumer(destination);
   consumer.setMessageListener(new MessageListener() {
      public void onMessage(Message message) {
         try {
            System.out.println("Got message '" + ((TextMessage)message).getText()+"'");
      } catch(JMSException e) {
            System.out.println("exception " + e);
         }
      }
   });
   connection.start();
   System.out.println("Listening for 15s");
   Thread.sleep(15000);
   session.close();
   Connection.close();
}
…
```

How do you use it from Java? (Sender/Producer)

```
…
public void send() throws Exception {
   ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(broker);
   Connection connection = factory.createConnection();
   Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

   // send to the ruby queue
   Queue queue = new ActiveMQQueue(sendQueueName);
   MessageProducer producer = session.createProducer(queue);

   // create our message
   Message message = session.createTextMessage("Greetings from Java!");

   // construct a reply-to destination - our java queue - on the message. this is optional!
   Destination destination = session.createQueue(receiveQueueName);
   message.setJMSReplyTo(destination);

   producer.send(queue,message);

   session.close();
   connection.close();
}
…
```

How do you use it from Ruby?

- Make sure STOMP transport is enabled:

```
<beans xmlns="http://activemq.org/config/1.0">
  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>
  <broker useJmx="true">
    <managementContext>
      <managementContext connectorPort="1099" jmxDomainName="org.apache.activemq"/>
    </managementContext>
    <persistenceAdapter>
      <journaledJDBC journalLogFiles="5" dataDirectory="${activemq.home}/activemq-data"/>
    </persistenceAdapter>
   <transportConnectors>
      <transportConnector name="default" uri="tcp://localhost:61616"
discoveryUri="multicast://default"/>
      <transportConnector name="stomp" uri="stomp://localhost:61613"/>
    </transportConnectors>
    <networkConnectors>
      <networkConnector name="default" uri="multicast://default"/>
    </networkConnectors>
  </broker>
</beans>
```

Ruby Send/Subscribe Client

```
#!/usr/bin/env ruby -w

require 'stomp'

client = Stomp::Client.open "", "", "localhost", 61613 # should match URI of our transportConnector

# STOMP requires you to prefix queue names with '/queue/' - actual queue name is just 'java'
# Also not that we can bake a reply-to queue name into our msg header.
client.send("/queue/java", 'Greetings from Ruby!', { "persistent" => true, 'reply-to' => '/queue/ruby' })

# Now subscribe to and listen for messages on our 'ruby' queue
client.subscribe("/queue/ruby", { "persistent" => true }) do |msg|
  puts "Got Message: '#{msg.body}' on #{msg.headers['destination']}, reply-to = #{msg.headers['reply-
to']}"
end

sleep 5

client.close
```

We had to hack up stomp.rb to not send up content-type or content-length headers in the message. Not sure why, but this prevented JMS from dequeueing the message.

How do you use it from C++?

- A burning question on the minds of many Java programmers, no doubt.
- Amazon.com wrote and contributed the OpenWire C++ client to the Apache Software Foundation.
- Find it here: http://activemq.apache.org/openwire-cpp-client.html

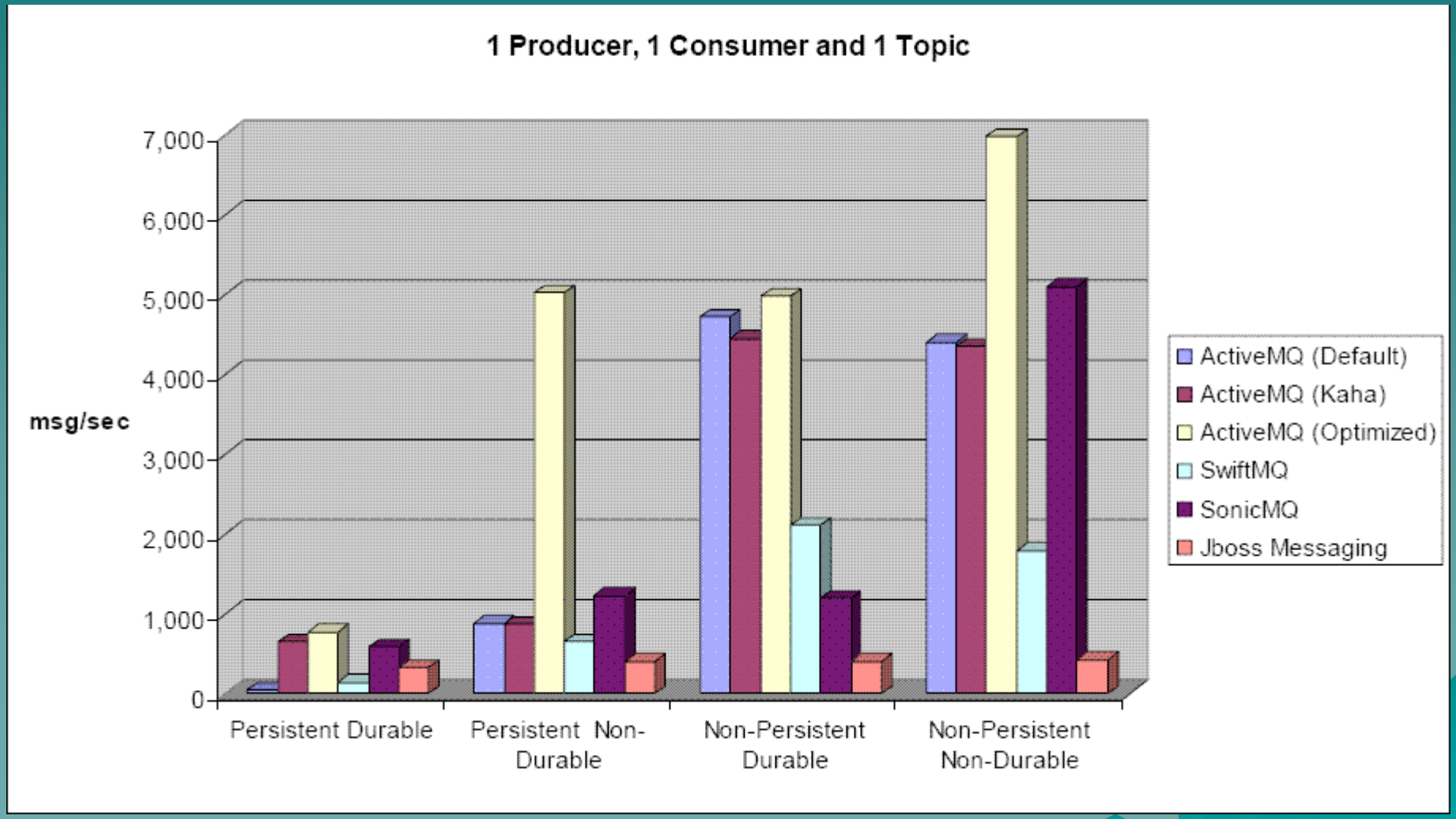
How Does It Perform? (Scenarios)

- Default
  - JDBC + journal message persistence
- Optimized (Async Send)
  - No guaranteed message persistence
  - JMS Spec violation
- Kaha
  - Log-structured file
  - Optimized for short-lived messages

```
<persistenceAdapter>
    <kahaPersistenceAdapter dir="file:${activemq.base}/kaha-data" maxDataFileLength="33554432"/>
</persistenceAdapter>
```

- Durable
  - Durable subscriber doesn't lose messages while disconnected

How does it perform?



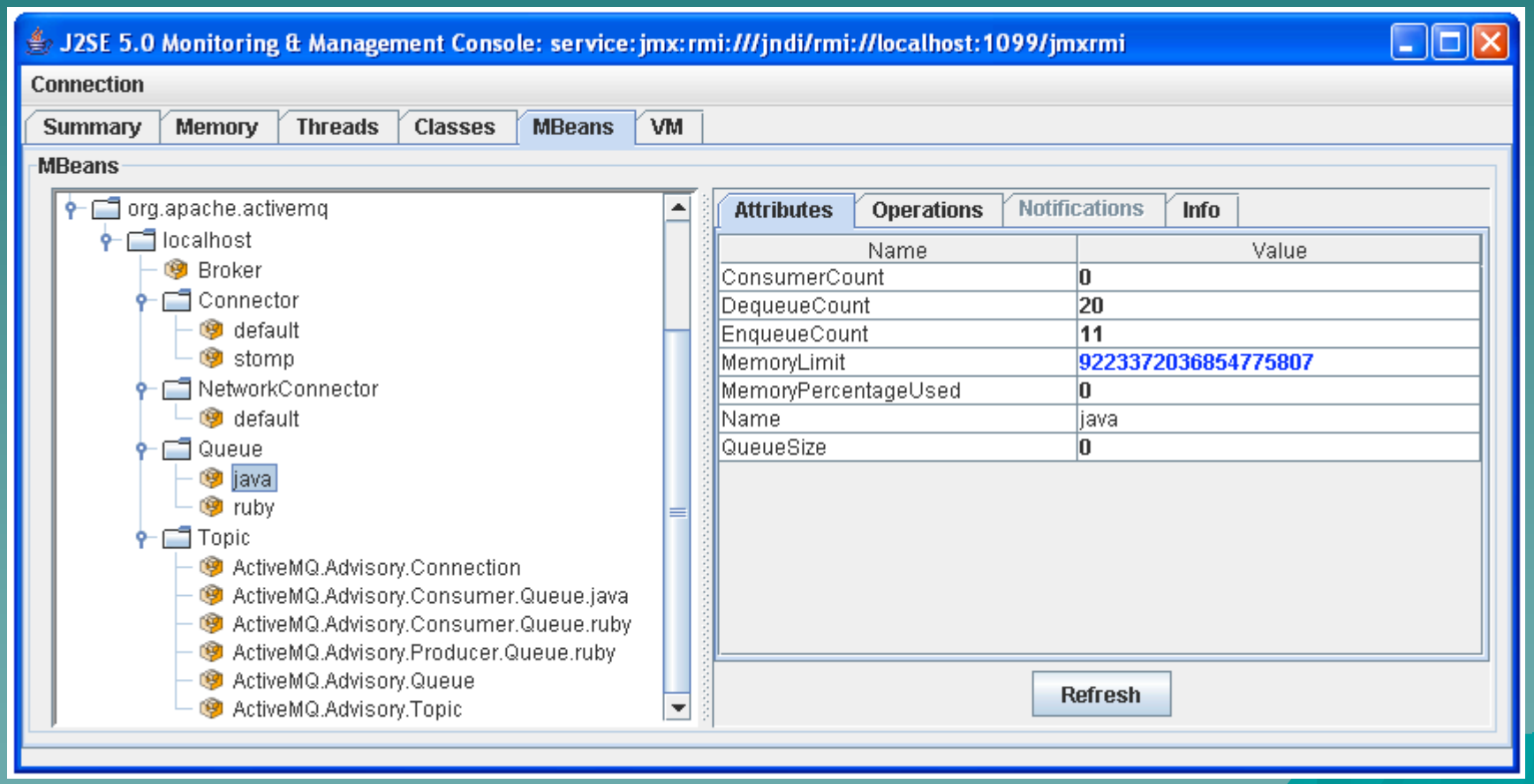
How does is perform?

Broker	Persistent Durable	Persistent Non-Durable	Non-Persistent Durable	Non-Persistent Non-Durable
ActiveMQ (Default)	44	878	4,713	4,404
ActiveMQ (Kaha)	656	866	4,443	4,346
ActiveMQ (Optimized)	769	5,026	4,979	6,979
SwiftMQ	138	655	2,114	1,791
SonicMQ	582	1,229	1,194	5,095
JBoss Messaging	330	398	398	423

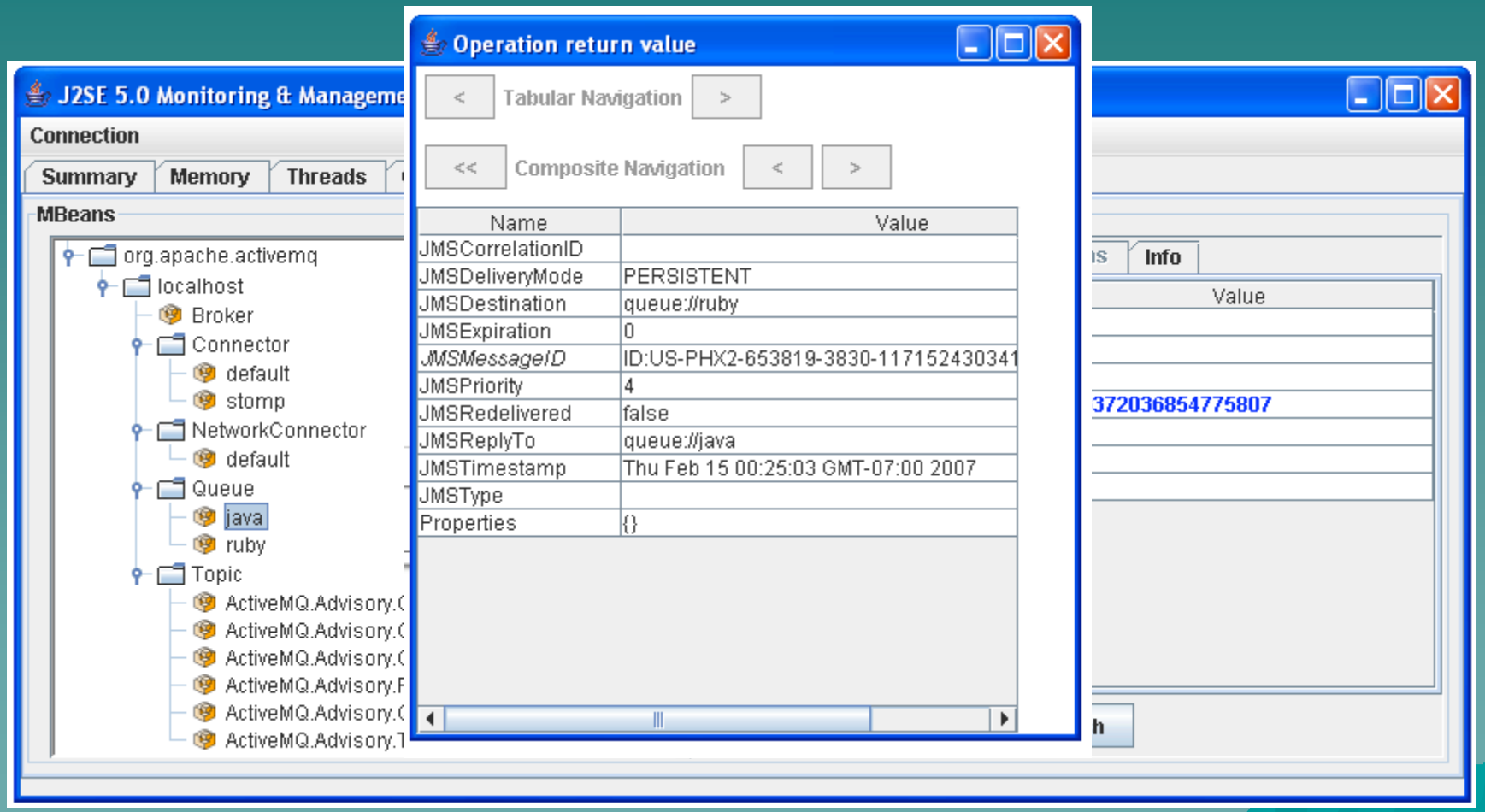
How do you monitor ActiveMQ?

- MBeans exposed through JMX
- Broker, Destination (Queue/Topic), Connector, Subscription, etc.
- MBea attributes Queue.ConsumerCount, Queue.De/EnqueueCount, Queue.QueueSize, etc, expose application-level data.
- MBean methods: Queue.browseAsTable, Queue.purge, Broker.start, Broker.stop, etc.
- browse/browseAsTable methods for access to message content and headers per Destination.

How do you monitor ActiveMQ? (JConsole/ JMX)

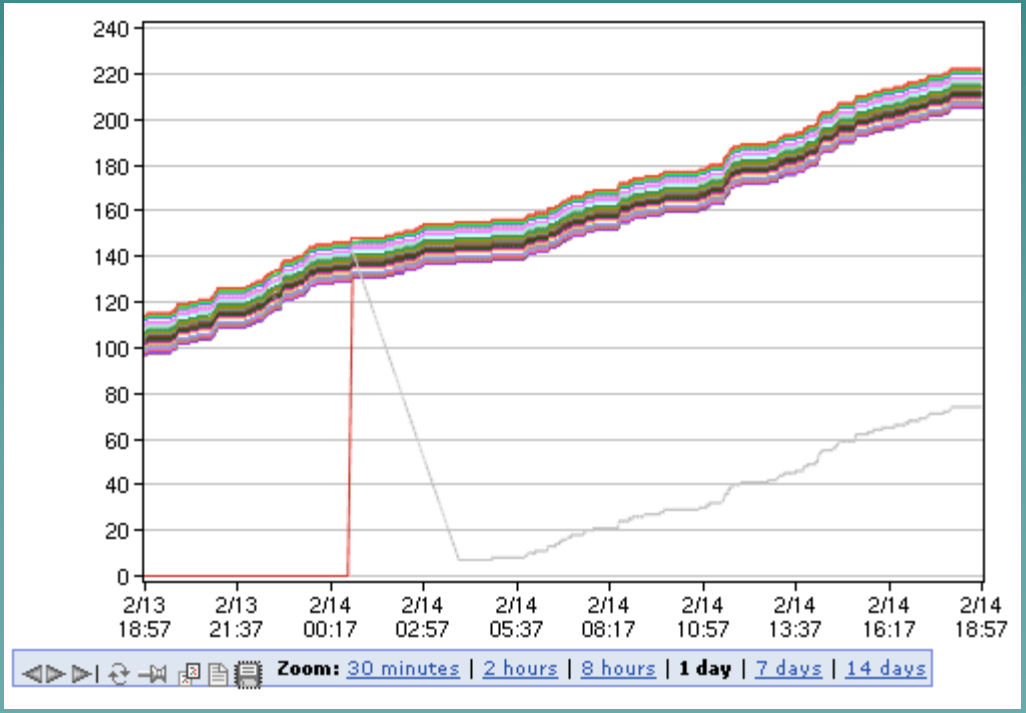


How do you monitor ActiveMQ? (JConsole/ JMX)



How do you monitor ActiveMQ? (JMX)

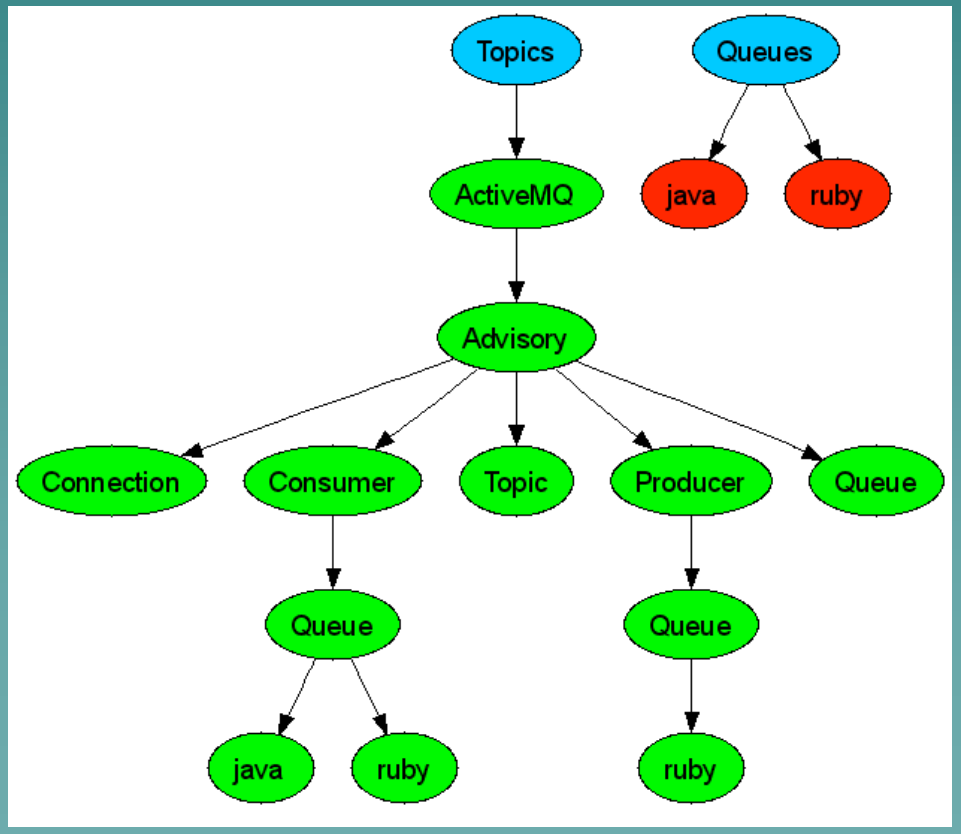
- We also emit the JMX metrics and can graph them within our existing metrics infrastructure:

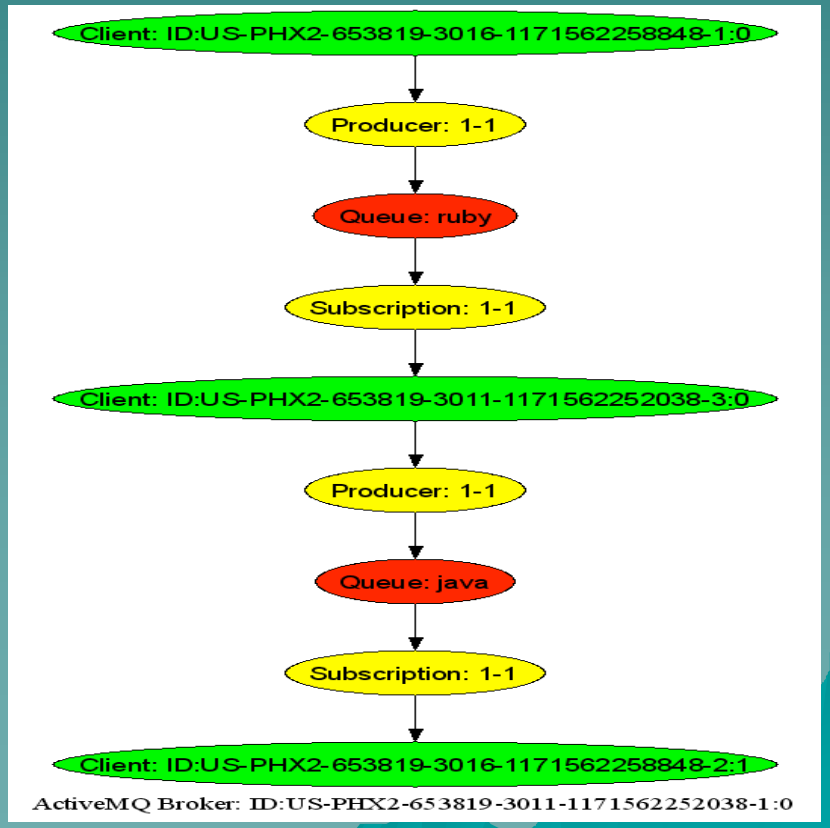


(Topic metric. Lower line indicates a host in our fleet that went down and later recovered.)

How do you monitor ActiveMQ? (DOT)

- DOT graphs of connections and broker network





Experience

- Our Configuration:
  - 1 message/sec to queues
  - 1 message/sec to topics
  - 10 different queues, mainly for coordination of asynchronous work
  - 5 different topics, mainly for stack-wide event notification
  - 20 simultaneous QueueReceivers
  - 30 simultaneous TopicSubscribers
  - MessageConsumers spread over 18 hosts
  - Single broker for Topics
  - Single broker for Queues

Experience

- It works correctly…
- …most of the time.

Experience – Observed Issues

- Prefetching
  - Consumers receive batches of messages, not just 1 at a time.
  - Trades throughput for latency
  - Naively tuned prefetch will leave some consumers starving for work
  - Turn down prefetching per Queue with ?consumer.prefetchSize=N (N > 0)
  - Important to tune your prefetchSize to the actual throughput of your message consumers
  - Controller per Destination – no way to describe an aggregate prefetch amount across all queues. Doesn't work well with a homogenous set of MessageConsumers who listen to all queues (our setup).

Experience – Observed Issues

- Message delivery stops when consumers are under heavy load.
- Rare OutOfMemoryError at startup
  - Related to massive onslaught of reconnections

Experience and Recommendations

- Version 4.0.2 is way more stable and reliable that 3.2.1
- Also better than 4.0
  - It's getting better!
- Broker is definitely not bulletproof.
  - 4.0 is 8 months old; 27000 lines of code
- Use it when message delay and occasional maintenance is acceptable and the features are compelling.
- Watch it. Quite a few people are getting paid to make it solid. It will end up solid.

Resources

- http://www.activemq.org/site/home.html
- Broker configuration reference: http://www.activemq.org/site/xml-reference.html
- Cross-language clients: http://www.activemq.org/site/cross-language-clients.html
- JMX Info: http://www.activemq.org/site/jmx.html