Introduction to JMS & Apache ActiveMQ

The web meeting will begin shortly

Dial-in Information:

Participant Code: 90448865
US Toll free: (1) 877 375 2160
US Toll: (1) 973 935 2036
United Kingdom: 08082348621
Germany: 08001821592
Netherlands: 08000226187
Russian Federation: 81080025511012
Sweden: 020797609
Hong Kong: 800901136

Introduction to JMS & Apache ActiveMQ

Aaron Mulder
Chariot Solutions

Agenda

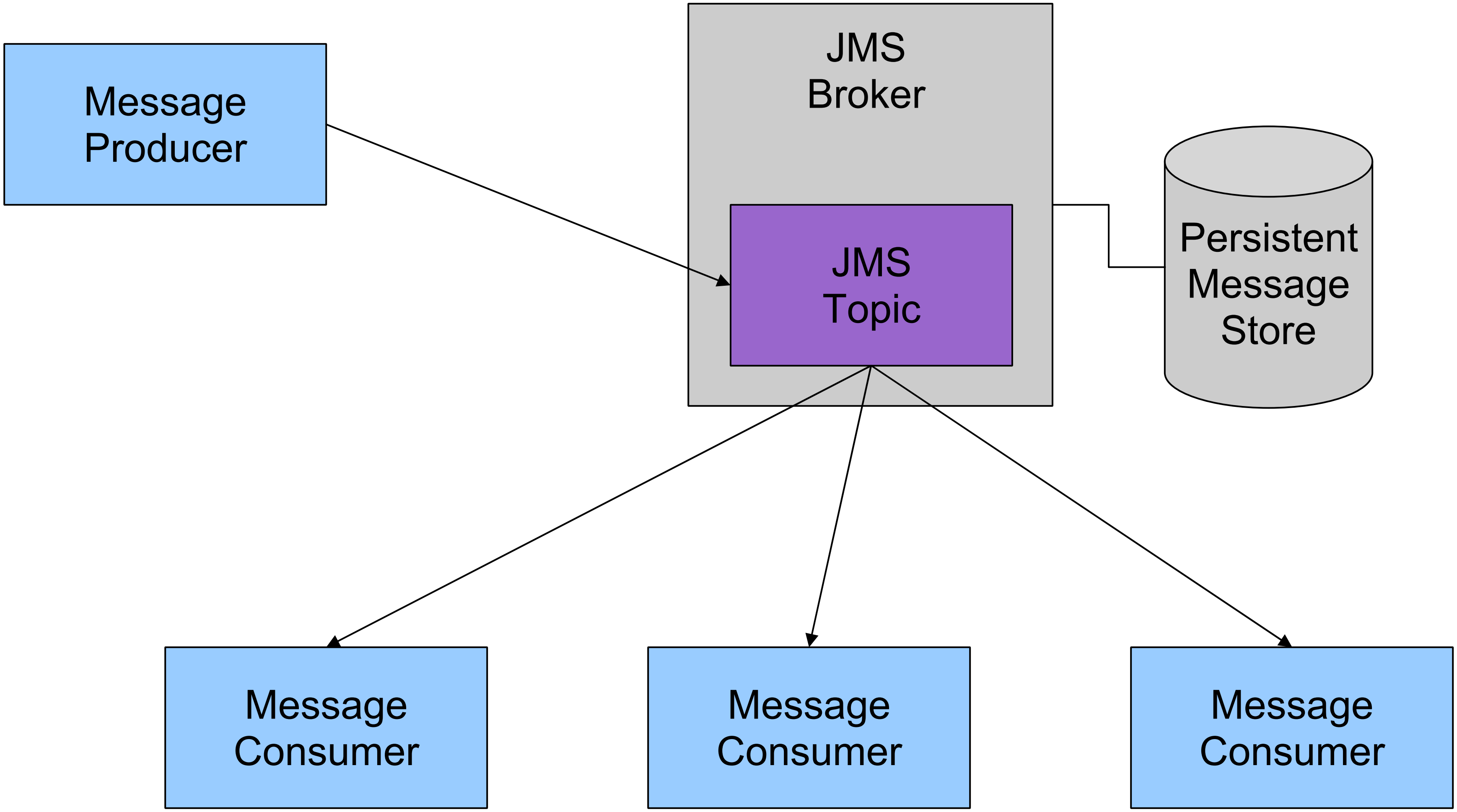
- Introduction to Messaging
- JMS Basics & Sample Code
- Advanced JMS Concepts
- Introduction to ActiveMQ
- ActiveMQ Features & Configuration
- ActiveMQ in Production
- Friends of ActiveMQ

Messaging

Why Messaging?

- Asynchronous operation
  - A client can schedule work to be done and return immediately
  - A client can be notified in an event-driven fashion when something has happened
- Loose Coupling
  - Systems connected via messaging need not be aware of each other, or use the same technologies
  - Key messages can easily be routed to many systems
  - Message broker is buffer in case of downtime

Systems Using Messaging



Why Messaging, con't

- Fire and Forget
  - Message Broker can guarantee messages are recorded and delivered even vs. crashes
- Parallelize Work
  - Messages in a single stream can be handled in parallel by many client threads or systems
- Throttle Work
  - A large amount of work can be split across a small number of threads or systems to throttle effort (e.g. due to licensing restrictions)

Why Not Messaging?

- Many commercial messaging products or app server messaging features have steep license costs
- Many developers are not comfortable writing messaging code
- There is no single prescription for a messaging architecture
  - The messaging configuration depends heavily on the specific business needs and requirements

JMS

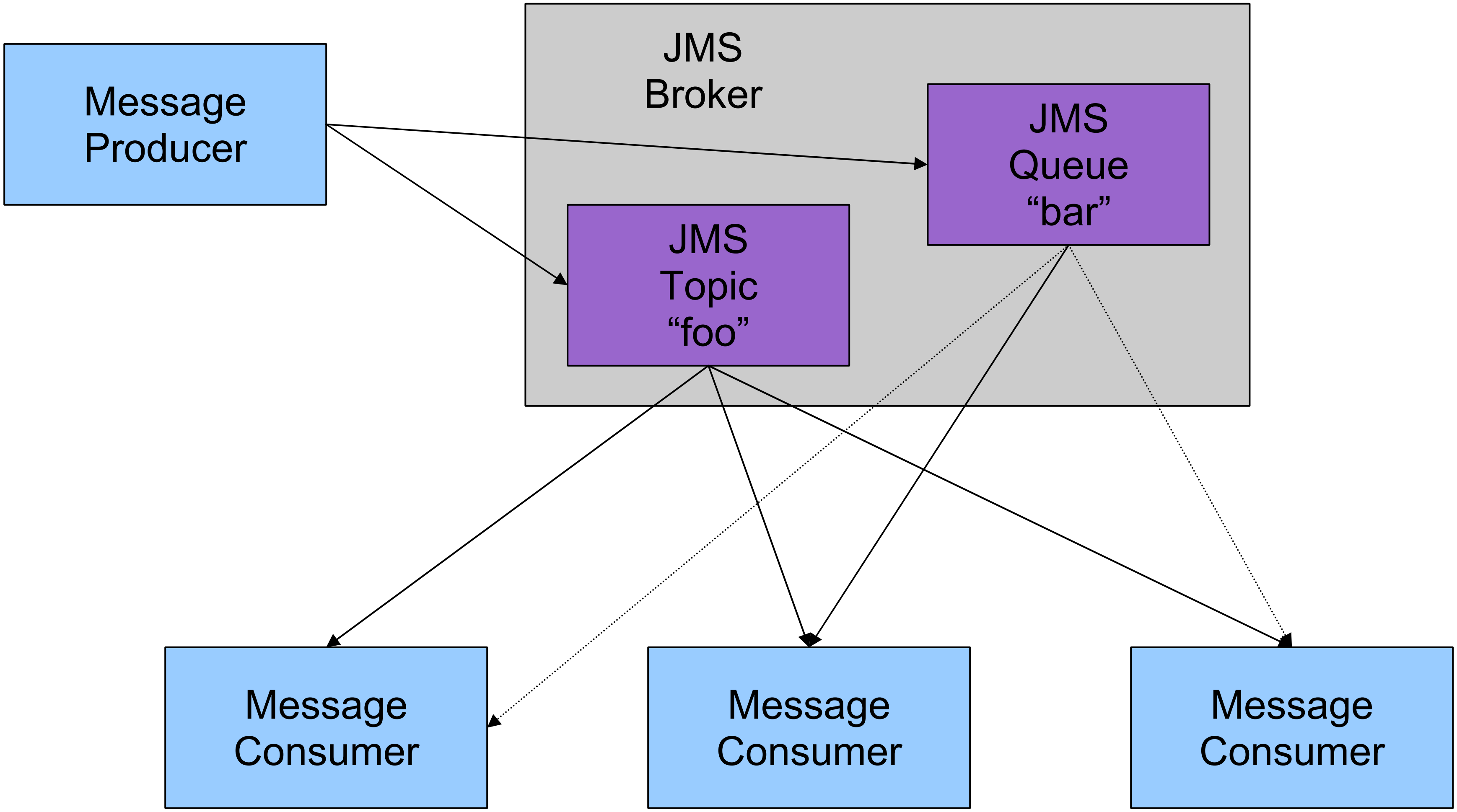
About JMS

- The Java API for messaging
- Included in Java EE, available standalone
- Key concepts include the message broker, message producers, message consumers, JMS topics vs. queues, and various message formats and processing options
- Includes a plain Java API (a little complex); simplified in Java EE, Spring, etc.
  - Those platforms simplify the process of establishing a connection to the message broker

JMS Destinations

- Each message is sent to a particular named “destination” (a topic or queue)
- Each message to a queue is processed by exactly one consumer
  - Typically, one of many connected consumers
- Each message to a topic is processed by *all* the consumers connected to that topic
  - May even be saved for consumers that happen to be disconnected at the moment

JMS Flow



Message Producer
JMS Broker
JMS Queue "bar"
JMS Topic "foo"
Message Consumer
Message Consumer
Message Consumer

JMS Messages

- Composed of headers and a body
- The headers are name/value pairs, and a consumer may filter on header values
  - Or could just use separate topics/queues instead
  - Some standard headers, also app-defined properties
- The body is different for different types of messages, but most common is the text message with e.g. text, SOAP, XML, YaML, etc.
  - Also MapMessage, StreamMessage, BytesMessage, ObjectMessage

Message Delivery

- Every message must be acknowledged or it will be redelivered
  - Consumer may be configured to acknowledge automatically or manually (normally auto)
  - Consumer may also use transactions, which auto-acknowledge on commit
- One standard header is JMSReplyTo, so you can name a destination that a reply message should be sent to
  - But up to consumer to send that reply

Synchronous vs. Asynchronous

- Producers are effectively synchronous – the send method returns when the broker accepts the message for delivery
- Consumers may be synchronous or asynchronous:
  - Sync: poll the broker for a new message, maybe with a timeout. Control returns when message available.
  - Async: consumer provides a listener, and the listener is invoked when a new message is available
    - But normally only one invocation at a time for a given listener

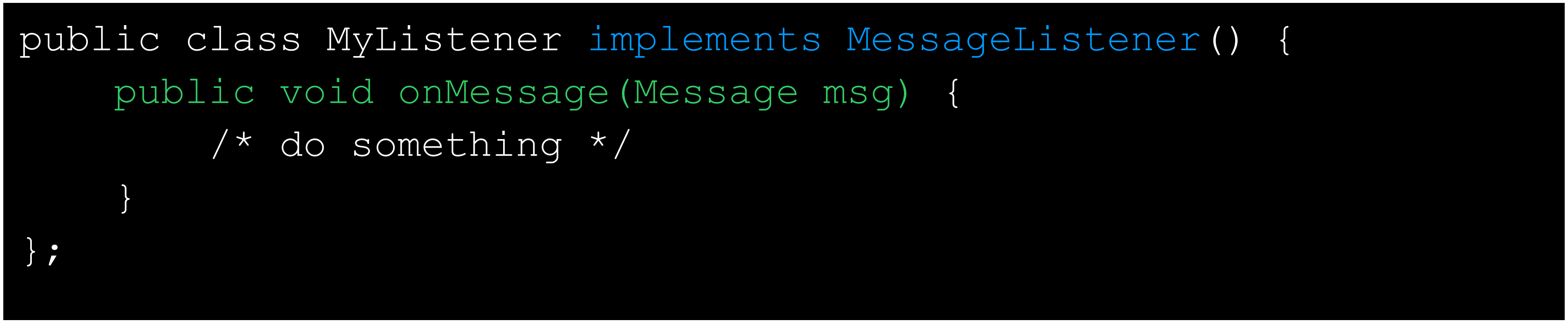
JMS Producer Example

```
// Create the reusable JMS objects
String url = “tcp://localhost:61616”
ConnectionFactory factory = new ActiveMQConnectionFactory(url);
Connection connection = factory.createConnection();
Session session = connection.createSession(false,
                                    Session.AUTO_ACKNOWLEDGE);
Topic topic = session.createTopic(“TestTopic”);
MessageProducer producer = session.createProducer(topic);

// Create and send the message
TextMessage msg = session.createTextMessage();
msg.setText(“Hello JMS World”);
producer.send(msg);

// clean up (not shown here)↰
```

JMS Async Consumer Example

```
// Create the reusable JMS objects
String url = “tcp://localhost:61616”
ConnectionFactory factory = new ActiveMQConnectionFactory(url);
Connection connection = factory.createConnection();
Session session = connection.createSession(false,
                                    Session.AUTO_ACKNOWLEDGE);
Topic topic = session.createTopic(“TestTopic”);
MessageConsumer consumer = session.createConsumer(topic);

// Listen for arriving messages
MessageListener listener = new MessageListener() {
    public void onMessage(Message msg) { /* do something */ }
};
consumer.setMessageListener(listener);
connection.start();
```

Spring/Java EE Async Consumer



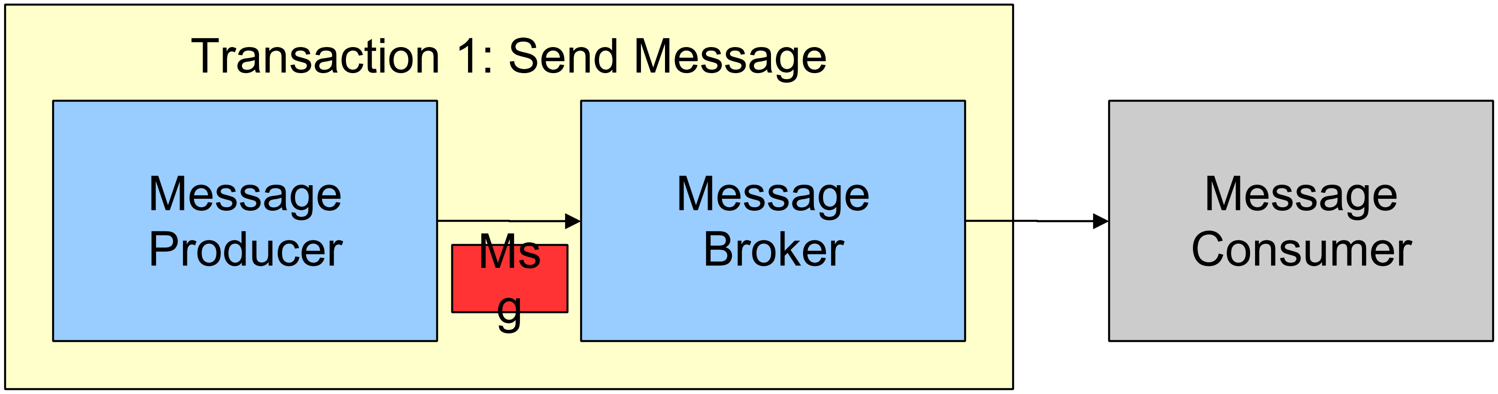
```
public class MyListener implements MessageListener() {
    public void onMessage(Message msg) {
        /* do something */
    }
};
```

- That's it!
- The configuration with regard to a specific broker, destination, etc. is handled in the Spring or Java EE configuration (not that it's necessarily pretty)
- The container manages threading, connections and sessions, etc.

Transactions

- May be local or XA (e.g. spanning JMS & a DB)
- May be used by both producers and consumers
- A transaction only encompasses the exchange between the producer and broker or consumer and broker; it's not end-to-end
- If producer rolls back, message isn't sent
- If consumer rolls back, message is redelivered
- For end-to-end “business transactions” maybe use JMSReplyTo or BPEL or something?

Transaction Diagram



Transaction 1: Send Message

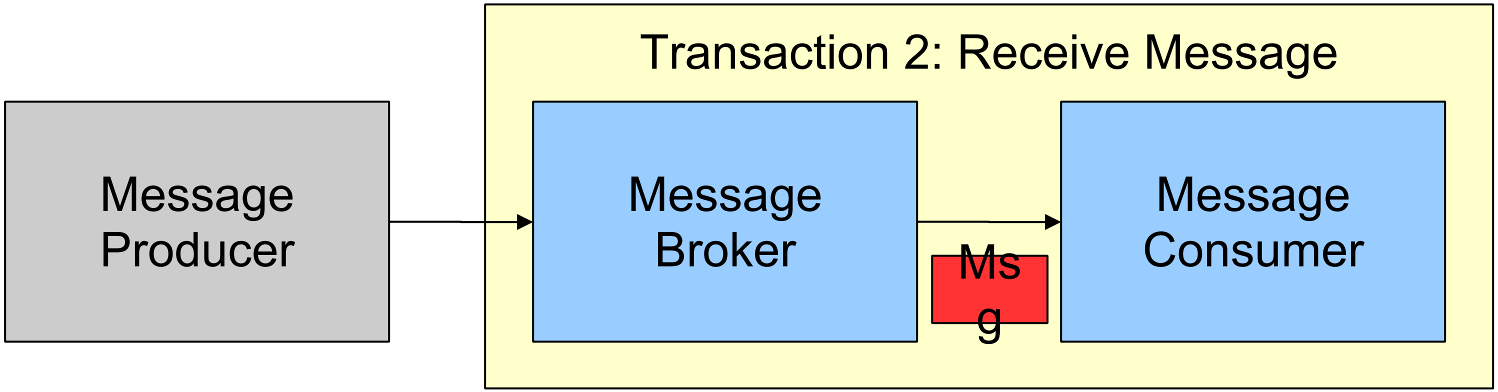
Message Producer

Msg

Message Broker

Message Consumer

In between: Broker has message, not yet delivered



Transaction 2: Receive Message

Message Producer

Message Broker

Msg

Message Consumer

Still More Options

- Messages may be persistent (saved to e.g. disk in case of broker crash)
  - If not, messages would be lost if the broker crashes or is restarted
- Topic subscribers may be durable (if they disconnect and reconnect, they'll get the messages sent while they were offline)
  - If not, a consumer on a topic will miss any messages sent while the consumer was disconnected
  - But also no risk of receiving really old messages

JMS Summary

- Many concepts and options, covering many typical messaging scenarios
- Already seeing some critical options for a messaging architecture
  - Message type & custom properties
  - Topics or queues?
  - Durable? Persistent?
  - Acknowledge modes or transactions?

ActiveMQ Features

About ActiveMQ

- An open-source message broker (compare to JBossMQ, or many commercial products)
  - See http://activemq.apache.org/
- Generally stable and high-performance
- Can be run standalone, or inside another process, app server, or Java EE application
- Supports everything JMS requires, plus various extensions
- Integrates well into other products

ActiveMQ Messaging Extensions

- Virtual Destinations (load-balancing and failover for topics)
- Retroactive Subscriptions (subscriber can receive some number of previous messages on connect)
- Exclusive Consumers & Message Groups (load-balancing and failover while preserving message ordering)
- Mirrored queues (monitor queue messages)
- And many more...

ActiveMQ Client Connectivity

- Dictated by the wire protocol a client uses to talk to the broker
- Generally there are two protocol options – OpenWire (binary) and Stomp (text)
  - OpenWire is the default and has the most history and best support (including SSL) – for Java, .NET, etc.
  - Stomp is easiest to develop for and therefore has the most cross-language support (Perl, Python, Ruby, ...)
- Also a variety of other special-purpose protocols (Jabber, adapters for REST/AJAX, etc.)

ActiveMQ Persistence Options

- Different strategies available for storing persistent messages
  - to local files, database, etc.
  - or both – stored to local files and then periodically batch undelivered messages to the DB...
- Default implementation changed between ActiveMQ 4.x and 5.x
- May still customize the persistence engine based on specific performance requirements

ActiveMQ Security and Management

- OpenWire protocol can use SSL for encryption
- Broker can use authentication (e.g. username/password required to connect)
  - Uses JAAS to identify the back-end user data store (properties files, DB, LDAP, etc.)
- JMX management enabled by default
  - Use a tool like JConsole to monitor queues, etc.
- Web Console available as well

ActiveMQ Testing

- ActiveMQ can easily run in an embedded, non-persistent, in-VM only mode for unit tests
- Also easily to run ActiveMQ via beans in a Spring context, if you're testing with Spring
- ActiveMQ includes a simple JNDI provider if you want to test Java EE code that relies on JNDI lookups to access JMS resources
- Can use tools like JMeter to load test the broker

Hands-on with ActiveMQ

Versions & Packaging

- ActiveMQ 4.1.2 is the latest “stable” release
- v5.0.0 is available but clearly a “pre-SP1” product
- v5.1.0 in the process of being released right now, and will be the go-to release
- 5.x includes the Web Console (which is a bit DIY under 4.x)
- Both versions also available through Maven 2 (e.g. for embedded usage)

Configuration

- The standalone ActiveMQ broker is driven off an XML configuration file
  - Can also easily configure an embedded ActiveMQ broker to read that configuration file (e.g. in Spring)
- Also includes a Java EE ResourceAdapter to plug into any application server
  - Can also use the standard XML config file, or just individual config properties on the RA
- Or can create and configure a broker entirely programmatically

ActiveMQ Configuration Example

```
<beans ...>
  <broker xmlns="http://activemq.org/config/1.0"
brokerName="MyBroker" dataDirectory="${activemq.base}/data">
    <transportConnectors>
       <transportConnector name="openwire"
                           uri="tcp://localhost:60010" />
       <transportConnector name="stomp"
                           uri="stomp://localhost:60020"/>
    </transportConnectors>
    <networkConnectors>
      <networkConnector name="Broker1ToBroker2"
                        uri="static://(tcp://localhost:60011)"
                        failover="true" />
    </networkConnectors>
  </broker>
</beans>
```

Broker vs. Client Configuration

- The broker configuration need not list specific destinations (they're created on demand)
- Individual messages or destinations can be configured by the clients code (e.g. topic or queue, persistent/durable or not, retroactive subscriber, etc.)
- Higher-level settings made in the broker config (memory and persistence settings, subscription recovery policy, virtual/mirrored destinations and names, etc.)

ActiveMQ in Production

Clustering

- Two clustering strategies:
  - Master/Slave(s) – best reliability, no improved scalability
  - Network of Brokers – best scalability, better availability, somewhat improved reliability
- Network of Brokers is best if you can live with the side effects
  - Messages may be delivered twice or substantially delayed (also out of order) in a failure scenario
  - Messages may be lost if a broker dies for good

Monitoring

- ActiveMQ can hook into a JVM or (with a little work) application server JMX provider
- Standard JMX tools can inspect the broker state, queue state, etc.
- The Web Console is handy, but not as useful to integrate into a larger monitoring environment
- Can use a JMX-to-SNMP bridge if needed

Security

- Most often, a “system” will authenticate to ActiveMQ, rather than a “user”
  - Easy to set up properties files with security accounts for a handful of systems
- But if user-level authentication is desirable, can use a slightly more complex JAAS configuration to refer to an existing user data store
  - Just make sure the client environment can access the user's username and password to use when connecting to ActiveMQ

Performance Considerations

- Real-world performance depends on:
  - topics vs. queues
  - number of producers vs. consumers
  - message size (smaller faster)
  - clustering strategy (master/slave slower)
  - persistence
  - message selectors
  - durable subscriptions
  - security options
  - transactions
  - VM (e.g. Sun vs. IBM vs. JRockit)
  - consumer speed and connectivity
  - hardware

Performance Tuning

- Best advice is to benchmark using the configuration and messaging patterns that will be used in production
  - Using JMeter, custom test scripts, etc.
  - Include representative messages, processing overhead, selectors, etc.
- Use durability/persistence only when needed
- Don't use unnecessarily large messages
- Use separate destinations vs. complex selectors where possible

Client Performance Tuning

- Slow consumers cause a slower broker
  - Consider Message Groups, Virtual Destinations, etc. to parallelize consumers
- Try batching many send operations in one transaction
- Make sure to pool or reuse the right JMS objects (Connection, Session, MessageProducer, MessageConsumer, maybe the JMSMessage) to avoid connection overhead

Friends of ActiveMQ

Friends of ActiveMQ

- ActiveMQ integrates into numerous environments (app server and so on)
- Apache Camel implements the standard Enterprise Integration Patterns, using ActiveMQ as the transport
- Apache ServiceMix is an ESB that uses ActiveMQ as the transport
- Apache Axis and Apache CXF are SOAP stacks that can uses ActiveMQ as the transport

Q&A

Thank You for Attending

Find out more at:

- http://springsource.com
- http://chariotsolutions.com/
- http://www.apache.org/

Join the discussion at:

- http://activemq.apache.org/

Commercial Support:

sales@springsource.com