LogicBlaze

Building reliable and high performance messaging applications with POJOs and Apache ActiveMQ

James Strachan

Personal Introduction

- James Strachan
  - Apache Member & Committer on many open source projects at
    - Apache, Codehaus, OpenSymphony, SourceForge, Tigris
  - Co-founder of a few Open Source Projects
    - Apache ActiveMQ, Apache ServiceMix, Groovy, Apache Geronimo, Apache Ode, dom4j, Jaxen etc.
  - Chief Architect, LogicBlaze
    - An Enterprise Open Source company
    - Provide training, mentoring, support for open source SOA

Agenda

- Messaging 101
- Overview of Apache ActiveMQ
- Advanced features of Apache ActiveMQ
- Developing JMS applications with POJOs
- Conclusions

Messaging 101

- Messaging is…
  - Loosely coupled exchange of messages between producers and consumers
    - So producers and consumers know nothing of each other
    - they only know about destinations (queues and topics)
  - the ideal approach to building high performance distributed systems
    - Web services done right is really just a form of messaging using pointy brackets
  - Can be persistent or non-persistent
    - May add timeouts & priorities to messages

Topics (publish & subscribe)

- One message goes to 0-to-many consumers based on the current subscribers
  - Think like mailing lists or discussion forums
  - The producer is decoupled from the consumers; it doesn’t need to know who all the consumers are
- Ideal for publishing business events
  - Distributed observer pattern
  - Allows one part of your system to notify anyone else who may be interested in an event

Queues (load balancing)

- Messages are load balanced across many consumers
  - Each message goes to exactly one consumer
  - Consumers compete for messages
- Its easy to browse and monitor queues
  - Monitor the performance of your application, find the hotspots, alert if queues are too full
- If a consumer crashes during the processing of a message it is automatically redelivered to another consumer
- So queues implement <u>reliable load balancing with optional persistence</u>

Queues - ideal for grid style

- Ideal for building reliable grid applications
  - Fire requests at a cluster of servers
  - Let the JMS provider provide load balancing, redelivery and optional persistence
  - If the queue starts filling up, just boot up new servers
  - Easily monitor the performance and status of the system
    - View the throughput rates on each queue and queue size
  - Deals easily with parts of your system being taken down for maintenance

Agenda

- What is Apache ActiveMQ?

Apache ActiveMQ

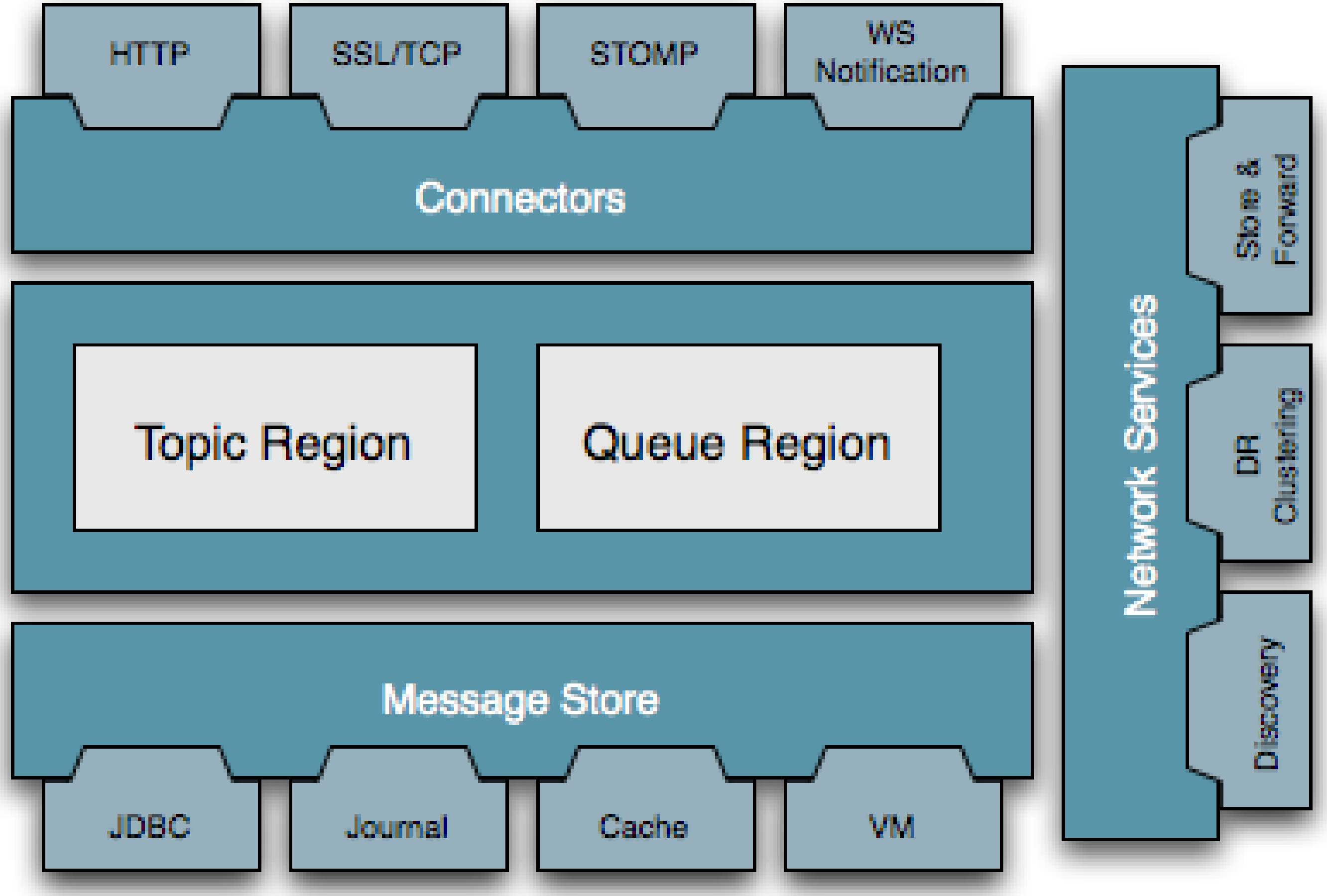
- Message fabric:
  - Message broker and clients for many languages and platforms
- Standards-based:
  - JMS 1.1, J2EE 1.4, JCA 1.5 and XA
- Integrated to:
  - JSE, Geronimo, Spring, Tomcat, JBoss and any J2EE 1.4 container (e.g., WebLogic/WebSphere)
- Transport Methods Supported:
  - TCP, SSL, HTTP, OpenWire, Stomp, UDP, multi-cast
- Capabilities include:
  - Queues, Topics, Durable and Non-durable Messaging with Plug-able Security and Persistence

ActiveMQ

Why Choose? ActiveMQ

- Fast
  - Fastest open source JMS provider by some margin and close, or better than, the proprietary alternatives
- Highly scalable
  - Clustering, peer-to-peer and federated networks support
  - Distributed Destinations
- Lots of open source clients
  - Clients for Java, C, C++, C#, ruby, python, perl, php, pike
- Easy to Use
  - Minimal Configuration (Dynamic Destination Creation)
- Open Source
  - Apache project & Geronimo JMS provider
  - backed by full commercial support from LogicBlaze
- Lots of advanced features
  - Covered in the next few slides :-)

Architecture



Architecture

- Pluggable features and strategies
  - Transports
    - TCP, SSL, HTTP, UDP, multicast
    - BIO, NIO, AIO
  - Pluggable Wire formats
    - OpenWire, XML, Stomp
  - Security
    - JAAS/JAAC plugin

Persistence

- JDBC provider
  - Defaults to embedded Apache Derby
  - Can use any RDMBS:
    - MySQL, PostGresql, Oracle, Sybase, DB2, Informix etc
- High performance journal
  - For short term persistence
  - Checkpoints to long term storage (JDBC) at regular intervals
- Kaha provider
  - Complete persistence option without using JDBC which is 2x-4x faster
  - Similar to Berkeley DB in some ways
    - Uses the journal for high performance persistence
    - Updates a separate index file asynchronously

Master / Slave (HA)

- A logical broker is created from a pair of physical brokers
  - a Master and Slave
- All messages and acknowledgements replicated to both physical brokers
  - provides redundancy on hardware failure
  - Auto-failover from master to slave
- Avoids loss of messages even if you have a catastrophic hardware failure or loose a data centre
  - Works on commodity hardware

Networks (store & forward)

- Federated network of brokers
  - Brokers have ‘network connectors’ between each other
  - Store forward or demand based forwarding
  - Can use fixed network routing tables or discovery
  - Clients can either piggy back the discovery mechanism or use fixed host lists etc
- Persistence model
  - Each broker owns a message
  - Can store and forward to other brokers
    - So the message ‘moves’ from broker to broker

Configuration

- Use Java code to configure JMS client/broker
- URI configuration mechanism
  - **vm://localhost?broker.persistent=false**
  - **tcp://locahost:61616?jms.useAsyncSend=true**
- JNDI provider (via properties file)
  - Allows queues/topics to be configured in JNDI
  - Dynamic queue/topic lookup.
    - E.g. client looks up in JNDI to ‘dynamicQueues/FOO.BAR’ will auto-create the queue dynamically in JNDI
- Spring based XML configuration file & factory beans
  - Easily integrate your own POJOs/extensions into client or broker
- Extended Spring XML configuration file using XBean to make XML more concise & easily validated

Agenda

- Advanced features of Apache ActiveMQ

Advanced features of ActiveMQ

- Wildcards
  - Subscribe to different hierarchies in your destination namespaces
    - Products.Books.Computing.EIP
    - Products.Books.Computing.*
    - Products.>
- Selectors
  - Provide content based filtering on messages using SQL 92 syntax
    - Customer = ‘gold’ and product in (1, 2, 3) and JMSPriority > 5
  - Supports XPath on the message body for XML messages

- Durable Topics kinda suck
  - Only one thread allowed to subscribe to a logical subscripition
- Virtual Destinations to the rescue!
  - Publish to a topic
    - VirtualTopics.Foo
  - Subscribe to a queue which logically represents a durable topic subscription
    - Consumer.MyConsumerName.VirtualTopics.Foo
    - Now can have many consumers load balancing
    - Can browse the subscription like any other queue

Advanced features of ActiveMQ

- Easy unit testing of JMS
  - Really fast startup
  - No separate process required
  - No persistence so no need to clear out queues before/after tests

```
ConnectionFactory factory =
    new ActiveMQConnectionFactory(“vm://localhost?broker.persistent=false”);
```

Advanced features of ActiveMQ

- Embedded brokers
  - Run the message broker inside the JVM of a producer or consumer
    - Can reduce a network hop
- Supports peer based and federated networks
  - Peer transport
    - JMS clients auto discover each other and use embedded brokers

- Automatic reconnection
  - If a broker dies or a network error prevents connection, the client can
    - automatically reconnect to another available broker
    - redeliver any in-flight messages
  - Customizable back-off algorithm etc
    - Pluggable discovery mechanism or
    - Fixed list of hosts/URLs to use
  - Clients can use failover: URL to connect to one of a number of brokers

failover:tcp://host1:port,tcp://host2:port,http://host3:port

Advanced features of ActiveMQ

- Queue ordering
  - Order is maintained for messages on a queue for a single consumer
  - With any JMS provider if you have multiple consumers on a queue you loose ordering
- One solution is Exclusive Queues
  - Basically one consumer owns the queue until it dies, then another consumer takes over
  - Useful if you want to deploy a homogeneous cluster of consumers where queue ordering is important and you want to auto-failover if a node dies
  - Though there is no parallelisation with this approach
    - Just 1 thread is processing all of the messages

Advanced features of ActiveMQ

- Message Groups (sticky load balancing & grid partitioning)
  - Maintain order for messages within a Message Group while providing load balancing across a cluster of consumers for different Message Groups
    - A Message Group is defined by setting the JMSXGroupID String header
    - E.g. set JMSXGroupID to the user ID or product code
      - All messages for that user/product will maintain their order and be processed usually by the same consumer
      - Different users/products will be processed in parallel on different consumers
    - Implements sticky load balancing for JMS
      - JMSXGroupID is a little like jsessionid in the web world
  - Efficiently partition your messages across many consumers
    - Reliable and supports failover
      - each value of JMSXGroupID is owned by one consumer
      - If the consumer dies, another one takes over and messages are redelivered
    - Preserves ordering but allows increased parallelisation and load balancing
    - Reduces concurrency and increases cache efficiency
      - Only 1 thread in your network will be processing a single JMSXGroupID value at once

Multi-client Connectivity

- REST
  - Use regular HTTP operations
- Ajax Messaging
  - very efficient Comet server using ActiveMQ and Jetty
- Stomp
  - Current: Ruby, Python, Perl, PHP, .Net/C#, C, C++
    - Note that the Ruby, Python, Perl and PHP clients are pure - no C library required
- OpenWire
  - works for Java, C, C++ and C# today (all pure language clients)
- ServiceMix
  - Multi-protocol bridge (e.g. MQSeries/RV to ActiveMQ bridge etc.)
  - Supported clients/transports: email, jabber, FTP, WebDAV, Samba, HTTP, foreign JMS
  - WS-Notification & WS-ReliableMessaging (via ServiceMix)
  - any JBI/JCA binding (e.g. CORBA/CICS/SAP)

OpenWire

- Designed for high performance
  - Binary & small and fast as possible
  - Can go beyond Stomp/JMS in features
  - Backwards compatible with any 4.x client
    - So can upgrade broker and not worry about clients
- Code generate marshalling code & test suite
  - Use Java Command POJOs with annotations
  - Guarrentees all clients work at the marshalling level
- ActiveMQ 4.0 has full OpenWire support for
  - Java, C, C++, C#
  - Both C++ and C# have their own language versions of the JMS API (CMS and NMS respectively) which are easy to implement on any transport (e.g. Stomp and OpenWire)

Stomp overview

- Designed so anyone can write a client really easily in any language
- Supports core JMS/MOM features
  - Connection, security
  - Sending, Receiving, Subscribe & Unsubscribe
  - Concurrent subscriptions, requests and response
  - Message acknowledgements (or auto-ack)
  - Transactions
  - JMS headers (correlation, replyto, message-id, message group etc)
  - Different QoS
    - queues, topics, temporary queue, topics
    - durable/non-durable)
- Optimized for ease of client (so makes a few performance trade offs)
  - E.g. text based, like HTTP
    - A little bigger than need be
    - A little slower than could be to parse
  - Default is text based rather than content-size/binary based

Stomp: connecting

These examples show using telnet as a Stomp client…

Sends…

```
    CONNECT
    login:<username>
    passcode:<passcode>

    ^@
```

Receives

```
    CONNECTED
    session:<session-id>

    ^@
```

Stomp: sending messages

Send a message to a queue or topic

```
SEND
destination:/topic/foo.bar

hello world!
^@
```

Stomp: sending with headers

Send using correlation, request/response, headers

```
SEND
destination:/queue/orders.books
reply-to:/temporaryQueue/James
correlation-id:4324234
amazonSecurityToken:abc234
amazonCustomerRating:gold

<order id=“123” customer=“jstrachan”>
 <book isin=“1234”/>
 <book isin=“456”/>
</order>
^@
```

Stomp: subscribing

Subscribe first

```
SUBSCRIBE
destination:/queue/orders.books

^@
```

Then later you’ll receive inbound messages

```
MESSAGE
destination:/queue/orders.books
Message-id:abc123
reply-to:/temporaryQueue/james134

This is the message
^@
```

Agenda

- Developing JMS applications with POJOs

Using JMS efficiently

- Pool expensive resources like Connections, Sessions, MessageProducers, MessageConsumers
  - They are relatively expensive to create/destroy as each create/destroy requires a blocking request/response with the broker
  - These objects are designed to be super efficient after they are created, so create them on startup and pool
- Be very careful with Spring's JmsTemplate
  - Each operation creates/destroys a Connection, Session, MessageProducer/MessageConsumer
  - Normally very inefficient unless you use a pooling wrapper over your JMS provider
    - E.g. PoolingConnectionFactory in ActiveMQ
  - Only good for sending

Using JMS efficiently

- Synchronous v Asynchronous trade-off
  - Asynchronous is much faster; though sometimes for reliability you must block until a message is on disk
- JMS transactions boost performance when performing multiple operations
  - E.g. consume a message and send 1..N messages in a transaction
  - Only pay the *sync cost* once, on the commit/rollback rather than on each send() or acknowledge()
  - With XA you can use batching to reduce the *sync cost*
- ActiveMQ Tuning Guide
  - **http://devzone.logicblaze.com/site/how-to-tune-activemq.html**

Using JMS efficiently

- Consuming messages in servers efficiently is harder than you might think
  - Each JMS Session uses 1 thread to process messages for its MesageConsumers
    - This is usually fine for GUIs
  - In servers you typically want a large thread pool and pool of Connections/Sessions/MessageConsumers to process inbound messages efficiently
  - You often want a pool of MessageListener objects too
- Message Driven Beans are one solution
  - They are EJBs and use JCA to do the JMS resource & thread pooling together with transactions & exception handling

POJO Remoting

- Write POJOs for your business logic
  - Keep your POJOs free of any middleware code
- Inject the middleware at deployment time
  - E.g. Spring Remoting, SCA, JAX-WS
  - Clean separation of concerns between middleware and business logic
- Middleware is increasingly becoming invisible!
  - Allows you to change the middleware easily as your requirements and topology change

Lingo

- http://lingo.codehaus.org/
- POJO Remoting for JMS using Spring
  - Hides the JMS code from your application
    - uses dynamic proxies/cglib to make client proxies
- Supports various *message exchange patterns*
  - Synchronous request/response
  - Asynchronous one ways & subscriptions
  - Asynchronous request/response
- Works well with JCA and Jencks

Lingo - Java code

```
public interface ExampleService {

   Foo regularRPC(String name);
   void anotherRPC() throws Exception;

   @OneWay
   void someOneWayMethod(String name, int
   age);

   @OneWay
   void watchPrices(String stock, MyCallback
   listener);
}
```

Lingo - client side

```
<!-- client side proxy-->
<bean id="client"
    class="org.logicblaze.lingo.jms.JmsProxyFactoryBean">
  <property name="serviceInterface" value=”com.acme.ExampleService"/>
  <property name="connectionFactory" ref="jmsFactory"/>
  <property name="destination" ref="exampleDestination"/>
</bean>

<!-- JMS ConnectionFactory to use -->
<bean id="jmsFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value=”tcp://localhost:61616"/>
</bean>

<!-- JMS Destination to use -->
<bean id="exampleDestination"
    class="org.apache.activemq.command.ActiveMQQueue">
  <constructor-arg index="0"
    value="test.org.logicblaze.lingo.example"/>
</bean>
```

Lingo - server side

```
<!-- the server side -->
<bean id="server"
   class="org.logicblaze.lingo.jms.JmsServiceExporte
   r">
  <property name="service" ref="serverImpl"/>
  <property name="serviceInterface"
   value=”com.acme.ExampleService"/>
  <property name="connectionFactory"
   ref="jmsFactory"/>
  <property name="destination"
   ref="exampleDestination"/>
</bean>

<!-- implementation of the service -->
<bean id="serverImpl"
   class=”com.acme.ExampleServiceImpl"
   singleton="true"/>
```

Agenda

- Conclusions

Conclusions

- Apache ActiveMQ is a great way to build a high performance, reliable and scalable distributed system
  - Load balancing
  - Data partitioning
  - High Availability
  - Loose coupling
- POJO frameworks like Lingo and Jencks can help hide the JMS plumbing code and implement efficient pooling and concurrent processing of messages

Conclusions

- Useful links
  - Apache ActiveMQ
    - http://incubator.apache.org/activemq/
  - Apache ServiceMix (JBI based ESB)
    - http://incubator.apache.org/servicemix/
  - Jencks
    - http://jencks.codehaus.org/
  - Lingo
    - http://lingo.codehaus.org/
  - Support and Services
    - Http://logicblaze.com
- Questions and Answers

Jencks

- http://jencks.codehaus.org/
- Implements Message Driven POJOs
  - Local and XA Transaction handling
  - Pools connections, sessions, threads
  - Exception handling & retry etc
  - A full JCA container which reuses the JCA Resource Adapters
- Supports any API – JMS, JDBC, JCA CCI, JBI, JAX-RPC etc
- Supports outbound pooling
  - E.g. pooled JMS sending, JDBC
  - Works with local or XA transactions too
- Uses the JCA Resource Adapters for resources for optimal pooling, resource usage and performance
  - E.g. the ActiveMQ Resource Adapter offers sophisticated concurrency control, resource usage and batching of requests to boost performance when using XA

Jencks

```
<!-- JCA Container -->
 <bean id="jencks" class="org.jencks.JCAContainer">
   <property name="bootstrapContext">
     <bean
  class="org.jencks.factory.BootstrapContextFactoryBean">
       <property name="threadPoolSize" value="25"/>
     </bean>
   </property>

   <property name="resourceAdapter">
     <bean id="activeMQResourceAdapter"
  class="org.apache.activemq.ra.ActiveMQResourceAdapter">
       <property name="serverUrl"
  value="tcp://localhost:61616"/>
     </bean>
   </property>
 </bean>
```

Jencks

```
<!-- Message Driven POJO -->
<bean id="inboundConnectorA" class="org.jencks.JCAConnector">
  <property name="jcaContainer" ref="jencks" />

  <!-- subscription details -->
  <property name="activationSpec">
    <bean class="org.apache.activemq.ra.ActiveMQActivationSpec">
      <property name="destination" value=”Foo.Bar"/>
      <property name="destinationType" value="javax.jms.Topic"/>
    </bean>
  </property>

  <property name="ref" value="echoBean"/>
</bean>

<!-- The actual POJO for processing messages which implements
  MessageListener -->
<bean id="echoBean" class=”com.acme.MyMessageListener"
  singleton="true"/>
```

Jencks

- ActivationSpec allows you to configure
  - Destination & optional selector and durable topic subscriptions
- ActiveMQActivationSpec extensions
  - Concurrency per subscription
    - how many messages can be processed in parallel
  - Redelivery policy
    - If a message fails use exponential backoff?
    - How many redeliveries?
    - Depends on what the listener does on how you deal with it
      - is a particular operation prone to deadlocks?
  - Pre-fetch options
    - How aggressively should the Resource Adapter preload asynchronously messages to be processed
    - RAM versus performance tradeoffs