Introducing

Apache ActiveMQ

— Nandakumar Purohit

Apache ActiveMQ

What will you learn?

- Connecting options to Brokers
- Creating Simple JMS Applications
- Dividing up with Queues
- Event processing with Topics
- Selecting Messages
- JMS Request/Response Pattern
- Scheduling Message Delivery

Apache ActiveMQ

Connecting options to Brokers

Option Name	Default Value	Description
`alwaysSessionAsync`	TRUE	When `true` a separate thread is used for dispatching messages for each Session in the Connection. A separate thread is always used when there's more than one session, or the session isn't in `Session.AUTO_ACKNOWLEDGE` or `Session.DUPS_OK_ACKNOWLEDGE` mode.
`alwaysSyncSend`	FALSE	When `true` a `MessageProducer` will always use Sync sends when sending a Message even if it is not required for the Delivery Mode.
`auditDepth`	2048	The size of the message window that will be audited for duplicates and out of order messages.
`auditMaximumProducerNumber`	64	Maximum number of producers that will be audited.
`checkForDuplicates`	TRUE	When `true` the consumer will check for duplicate messages and properly handle the message to make sure that it is not processed twice inadvertently.
`clientID`	null	Sets the JMS clientID to use for the connection.
`closeTimeout`	15000	Sets the timeout, in milliseconds, before a close is considered complete. Normally a `close()` on a connection waits for confirmation from the broker. This allows the close operation to timeout preventing the client from hanging when no broker is available.

Apache ActiveMQ

Connecting options to Brokers

Option Name	Default Value	Description
`consumerExpiryCheckEnabled`	TRUE	Controls whether message expiration checking is done in each `MessageConsumer` prior to dispatching a message. Disabling this check can lead to consumption of expired messages. (since 5.11).
`copyMessageOnSend`	TRUE	Should a JMS message be copied to a new JMS Message object as part of the `send()` method in JMS. This is enabled by default to be compliant with the JMS specification. For a performance boost set to `false` if you do not mutate JMS messages after they are sent.
`disableTimeStampsByDefault`	FALSE	Sets whether or not timestamps on messages should be disabled or not. For a small performance boost set to `false`.
`dispatchAsync`	FALSE	Should the broker dispatch messages asynchronously to the consumer?
`nestedMapAndListEnabled`	TRUE	Controls whether Structured Message Properties and MapMessages are supported so that Message properties and `MapMessage` entries can contain nested Map and List objects. Available from version 4.1.
`objectMessageSerializationDefered`	FALSE	When an object is set on an `ObjectMessage` the JMS spec requires the object be serialized by that set method. When `true` the object will not be serialized. The object may subsequently be serialized if the message needs to be sent over a socket or stored to disk.

Apache ActiveMQ

Connecting options to Brokers

Option Name	Default Value	Description
`optimizeAcknowledge`	FALSE	Enables an optimized acknowledgement mode where messages are acknowledged in batches rather than individually. Alternatively, you could use `Session.DUPS_OK_ACKNOWLEDGE` acknowledgement mode for the consumers which can often be faster. **WARNING**: enabling this issue could cause some issues with auto-acknowledgement on reconnection.
`optimizeAcknowledgeTimeOut`	300	If > 0, specifies the max time, in milliseconds, between batch acknowledgements when `optimizeAcknowledge` is enabled. (since 5.6).
`optimizedAckScheduledAckInterval`	0	If > 0, specifies a time interval upon which all the outstanding ACKs are delivered when optimized acknowledge is used so that a long running consumer that doesn't receive any more messages will eventually ACK the last few un-ACK'ed messages (since 5.7).
`optimizedMessageDispatch`	TRUE	If `true` a larger prefetch limit is used - only applicable for durable topic subscribers.
`useAsyncSend`	FALSE	Forces the use of Async Sends which adds a massive performance boost; but means that the `send()` method will return immediately whether the message has been sent or not which could lead to message loss.
`useCompression`	FALSE	Enables the use of compression on the message's body.

Apache ActiveMQ

Connecting options to Brokers

Option Name	Default Value	Description
`useRetroactiveConsumer`	FALSE	Sets whether or not retroactive consumers are enabled. Retroactive consumers allow non-durable topic subscribers to receive old messages that were published before the non-durable subscriber started.
`warnAboutUnstartedConnectionTimeout`	500	The timeout, in milliseconds, from the time of connection creation to when a warning is generated if the connection is not properly started via Connection.start() and a message is received by a consumer. It is a very common gotcha to forget to start the connection and then wonder why no messages are delivered so this option makes the default case to create a warning if the user forgets. To disable the warning just set the value to < `0`.

Example

```
tcp://localhost:61616?
jms.prefetchPolicy.all=100&jms.redeliveryPolicy.maximumRedeliveries=5
```

Apache ActiveMQ

Transport Configuration Options

Transport Type	Description
`The AUTO Transport`	Starting with 5.13.0 ActiveMQ has support for automatic wire protocol detection over TCP, SSL, NIO, and NIO SSL. OpenWire, STOMP, AMQP, and MQTT are supported. For details see the AUTO Transport Referen
`The VM Transport`	• The VM transport allows clients to connect to each other inside the VM without the overhead of the network communication. • The first client to use the VM connection will **boot an embedded broker.** • Subsequent connections will attach to the same broker. • Once all VM connections to the broker have been closed, the embedded broker will **automatically shutdown.**
`The AMQP Transport`	• Starting with 5.6.0 ActiveMQ also supports MQTT. • Its a light weight publish/subscribe messaging transport
`The MQTT Transport`	• Starting with 5.6.0 ActiveMQ also supports MQTT • Its a light weight publish/subscribe messaging transport
`The TCP Transport`	The TCP transport allows clients to connect a remote ActiveMQ using a a TCP socket
`The NIO Transport`	Same as the TCP transport, except that the New I/O (NIO) package is used, which may provide better performance

Apache ActiveMQ

Transport Configuration Options

Transport Type	Description
`The Failover Transport`	• The Failover transport layers reconnect logic on top of any of the other transports. • Its configuration syntax allows you to specify any number of composite URIs. • The Failover transport randomly chooses one of the composite URIs and attempts to establish a connection to it. • If it does not succeed or if it subsequently fails, a new connection is established to one of the other URIs in the list.

Apache ActiveMQ

Simple Application

- In this recipe we are going to create a very simple JMS application that places a message on a queue in ActiveMQ and consumes that message from the queue immediately after. This example demonstrates many of the JMS APIs that we will be using in the future recipes.

Apache ActiveMQ

Simple Application - How It Works

This example defines three methods: `before()`, `run()`, and `after()`, which we call in sequence. Each of these methods breaks out the basic steps you must do in any JMS application:

In the `before()` method we set up our connection to the ActiveMQ Broker and create the basic JMS boilerplate objects that we'll use in the `run()` method later.

- Create a new JMS `ConnectionFactory` using the concrete type `ActiveMQConnectionFactory`, which is provided in the ActiveMQ client library. The `ConnectionFactory` is supplied the URI of our running broker, namely, `tcp://localhost:61616`. Every `ActiveMQConnectionFactory` needs a URI telling where and how to connect to the broker.
- Create a new JMS `Connection` object by invoking the `ConnectionFactory` object's `createConnection` method. If the connection to the broker cannot be created we will get an exception at this point.
- We can now start our new `Connection`. A JMS `Connection` object won't deliver any messages to your `MessageConsumer` until you start it so it's very important to remember to call `connection.start()`.
- Create a new JMS `Session` object by calling the `Connection` object's `createSession()` method.
  Since `Session` requires a message acknowledgement mode, we specify that in the `createSession()` method. We choose auto acknowledgement mode so we don't have to worry about manually acknowledging messages in this simple application.
- Finally we create a `Destination` object that our application's `producer` and `consumer` will use to send and receive our simple message. In our example we create a queue but it would work exactly the same if we had used a Topic domain for this simple example.

Apache ActiveMQ

Simple Application - How It Works

The code for the `run()` is divided into two parts, one of which performs the send of our simple message and then our attempts to receive it. Let's break down the send portion first:

- Using the `Session` we created earlier, we first create a `MessageProducer`; this is the object that will allow us to send our simple message.
- Create a new `TextMessage` object using the `Session` object and assign it a payload at the same time.
- Send the message using the `MessageProducer` object's `send()` method. The `MessageProducer` object knows where to send the message since we passed in a `Destination` object when we created it.
- Finally, we close down the `MessageProducer` object since we won't use it again in this application.

Receiver

- First we create a `MessageConsumer` object using `Session` just as we did for the `MessageProducer` previously
- We use the `MessageConsumer` object blocking the `receive()` call to wait for the broker to route the message we sent previously to our producer
- Once we receive the message we need to cast it to the correct type since the `receive()` method returns a base message reference
- We print out a little message showing the payload of the `TextMessage` we sent previously

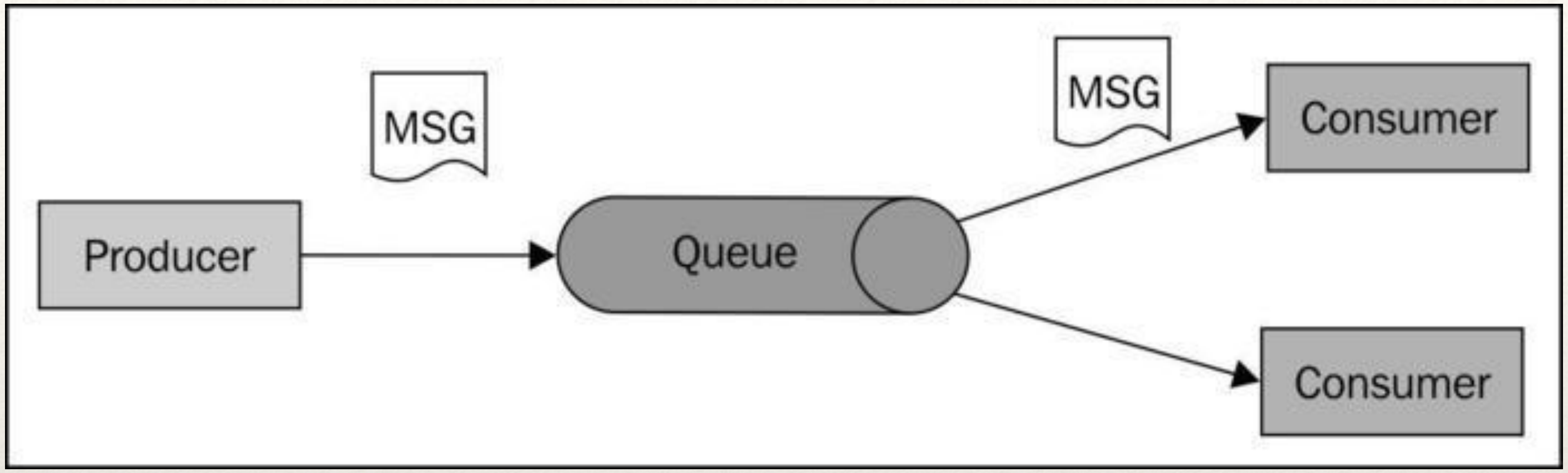
Apache ActiveMQ

Dividing up work with Queues

- In this recipe we look at how the JMS queue can be used to divide up work with a simple job queue example.

Apache ActiveMQ

Dividing up work with Queues - How It Works



- When a message is sent to a queue and subsequently dispatched to a consumer it's sent only to that consumer and no other. This type of message dispatching is often referred to as **"once and only once dispatch".**
- **What would happen if our producer sent a message and there were no consumers to receive it?** In that case the queue would hold onto the message until a consumer arrived to dequeue it, or if it has a set expiration it will be removed once that time has elapsed.

Apache ActiveMQ

Dividing up work with Queues - How It Works

The properties of the JMS queue make it an ideal means of processing jobs; let's review those properties again in the context of a job processing application:

- Messages placed on a queue are persisted until they are consumed by `MessageConsumer`, jobs aren't lost if no active consumer is online.
- Queued messages can have a **time to live** (**TTL**) set, which causes them to time out and be removed from the queue. This is good if a queued job is time sensitive.
- Queued messages are **delivered in the order** they were placed on the queue if a single consumer is active, this is good if the jobs are order dependent.
- When more than one consumer is reading from the queue, the messages on the queue are **spread out among all active consumers** to allow for load balancing of your queued up jobs.

Apache ActiveMQ

Dividing up work with Queues - How It Works

The Job Producer

```java
for (int i = 0; i < 1000; ++i) {
  TextMessage message = session.createTextMessage("Job number: " + i);
  message.setIntProperty("JobID", i);
  producer.send(message);
  System.out.println("Producer sent Job("+i+")");
}
```

The Job Consumer

```java
public void run() throws Exception {
  MessageConsumer consumer = session.createConsumer(destination);
  consumer.setMessageListener(new JobQListener());
  TimeUnit.MINUTES.sleep(5);
  connection.stop();
  consumer.close();
}


public void onMessage(Message message) {
  try {
    int jobId = message.getIntProperty("JobID");
    System.out.println("Worker processing job: " + jobId);
    TimeUnit.MILLISECONDS.sleep(jobDelay.nextInt(100));
  } catch (Exception e) {
        System.out.println("Worker caught an Exception");
    }
}
```

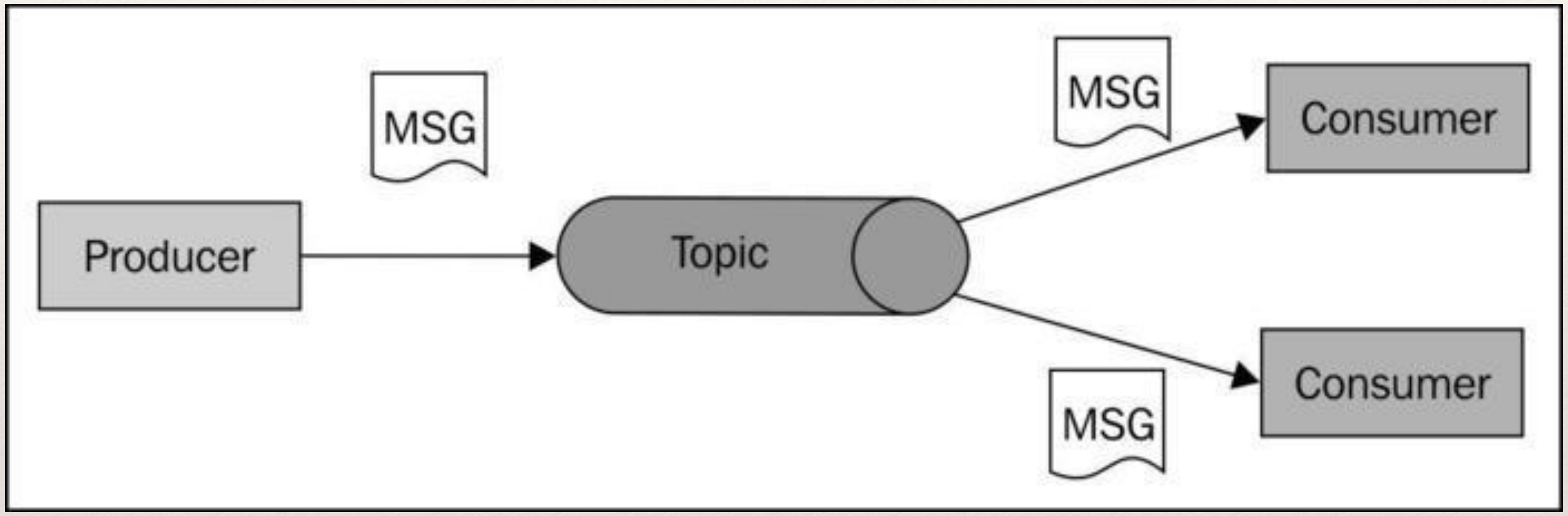
Apache ActiveMQ

Event processing with Topics

- In this recipe we will explore how to use JMS topics as event channels by looking at a sample stock price update service and a simple consumer of those events.

Apache ActiveMQ

Event processing with Topics - How It Works



Apache ActiveMQ

Event processing with Topics

Durable Subscription

While it's true that the default behavior of a JMS topic is to not retain any message that is sent to it when there are no active consumers, there are cases where you might want to have a **consumer go offline and, when they come back online, receive any messages that were sent while they were offline.**

The JMS specification provides for this by defining a type of topic subscription known as a **durable subscription.**

To create a durable subscription your code must call the `createDurableSubscriber()` method of a `Session` instance as well as give your connection a unique name that your application must reuse each time it runs. Let's take a look at the changes we would need to make to our stock price consumer to make it a durable subscriber:

```
public void before() throws Exception {
        …
        connection.setClientID(“DurableConsumer3");


public void run() throws Exception {
        //MessageConsumer consumer = session.createConsumer(destination);
       MessageConsumer consumer = session.createDurableSubscriber((Topic)destination,
"DurableConsumer3");
```

Apache ActiveMQ

Selecting Messages

- Building on what we learned in the preceding recipe, we will modify our stock price consumer to use JMS selectors to filter the events it receives from the update service.

Apache ActiveMQ

Selecting Messages

What is a JMS selector?

A selector is a way of attaching a filter to a given subscription, also known as **content-based routing**. In JMS, the selector uses message properties and headers compared against Boolean expressions to filter messages. The Boolean expressions are defined using SQL92 syntax.

Element	Valid values
Identifiers	Property or header field reference (such as JMSCorrelationID, price, and date). The following values are not possible: `NULL`, `TRUE`, `FALSE`, `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, `IS`.
Operators	`AND`, `OR`, `LIKE`, `BETWEEN`, `=`, `<>`, `<`, `>`, `<=`, `>=`, `IS NULL`, `IS NOT NULL`.
Literals	The two Boolean literals, `TRUE` and `FALSE`. Exact number literals that have no decimal point; for example, +20 and -256, 42. Approximate number literals. These literals can use scientific notation or decimals; for example, -21.4E4, 5E2 and +34.4928.

Apache ActiveMQ

Selecting Messages

We can add a variable named `selector` to our consumer application and initialized it by reading a system property named `QuoteSel`, which defaults to `symbol = 'GOOG'`.

We then pass in our set selector value to the `Session` object's `createConsumer()` method.

```
selector = System.getProperty("QuoteSel", "symbol = 'GOOG'");

selector = System.getProperty("QuoteSel", "symbol IN (‘GOOG','AAPL')");

selector = System.getProperty("QuoteSel", "symbol = 'GOOG' AND price < 500");
```

Apache ActiveMQ

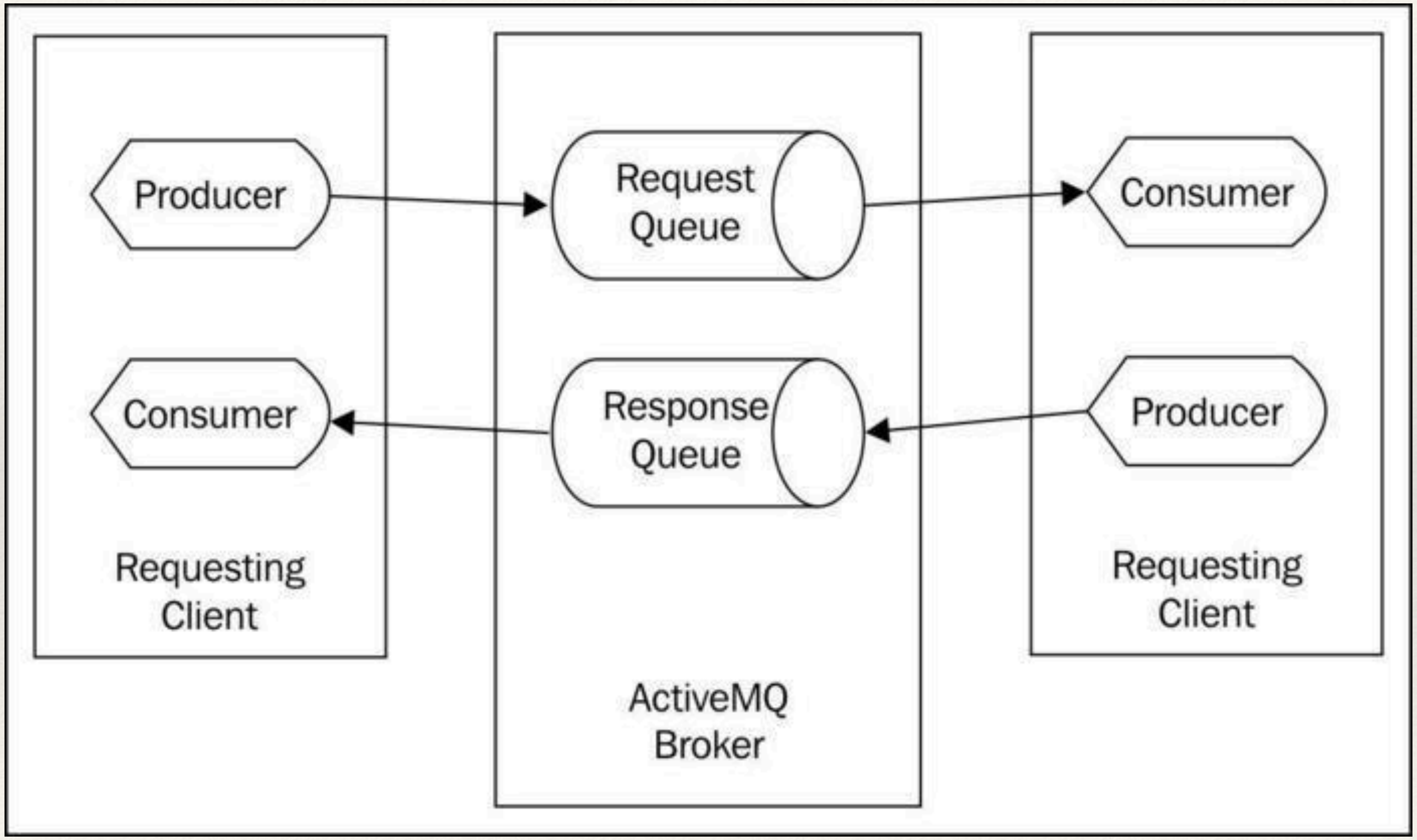
Using JMS Request/Response Pattern

- In this recipe we are going to take a look at a messaging pattern known as request/response messaging.

Apache ActiveMQ

Using JMS Request/Response Pattern

How It Works



Apache ActiveMQ

Using JMS Request/Response Pattern

How It Works

- Traditionally, this type of architecture has been implemented using **TCP client and server** applications that **operate synchronously** across a network connection.
- The biggest **issue is about scaling**. Because the TCP client and server are tightly coupled, it's difficult to add new clients to handle an increasing workload.
- Using messaging based architecture we can reduce or eliminate this scaling issue along with other issues of fault tolerance and so on.
- In the messaging paradigm, a requester sends a request message to a queue located on a remote broker and one or more responders can take this message, process it, and return a response message to the requester.

Apache ActiveMQ

Using JMS Request/Response Pattern

How It Works

The `JMSReplyTo` property for a JMS message was added just for this sort of messaging pattern.
The responder application doesn't have to know anything about preconfigured destinations for sending responses, it just needs to query the message for its `JMSReplyTo` address.
This is yet another example of the benefits of the loose coupling that comes from using the JMS API.

In our sample application, we create a JMS temporary queue and assign that to `JMSReplyTo` for every request message we send.

A JMS temporary destination works a lot like its non-temporary counterpart, however there are three key differences:

- The **lifetime** of a temporary destination is **tied to that of the connection object** that created it. Once the connection is closed, the temporary destination is destroyed.

- Only a `MessageConsumer` object created from the connection that created the temporary destination can consume from that temporary destination.

- **Temporary destinations don't offer the message persistence** or reliability guarantees that normal destinations do.

Apache ActiveMQ

Using JMS Request/Response Pattern

How It Works

Did you notice that we also assign a **correlation ID** to each request?

Since there could be multiple responses for requests we've sent, we might want to match up each response to ensure all our work gets done.

For example, our application could have stored the correlation IDs in a map along with the job data and matched up the responses.

Also, our application could have checked on a timeout if any outstanding requests hadn't arrived and could have resubmitted the unfinished job, or logged a warning to the administrator.

The JMSCorrelationID field allows you to build this sort of book keeping into your request/response applications easily.

Apache ActiveMQ

Using JMS Request/Response Pattern

How It Works

As you can see, we create both a producer and consumer for our response example once again. The `MessageProducer` object we created was assigned a `null` destination; this is called an **anonymous producer**.

We can use the **anonymous producer** to send messages to **any destination**, which is great since we don't know at startup which destination we are going to publish our responses to. We'd rather not create a new `MessageProducer` object every time a message arrives since that would add more network traffic and load to our broker.

When a request message is received by the responder application, it queries the message for the `JMSReplyTo` destination to which it should send the response. Once the responder knows where to send its answer, it constructs the appropriate response message and assigns it the `JMSCorrelationID` method that the requester will use to identify the response prior to sending it back to the responder.

Apache ActiveMQ

Using JMS Request/Response Pattern

- In this recipe we will learn how to schedule message delivery in ActiveMQ.

Apache ActiveMQ

Using JMS Request/Response Pattern

- Schedule a type of wake-up-call message to be delivered after a 10 second delay
- Then have it delivered again every 5 seconds, 9 more times.
- Our sample application uses the predefined message header values in the `ScheduledMessage` utility class provided in the ActiveMQ Client JAR;
- The available header values for scheduled messages are shown in the following table:

AMQ_SCHEDULED_DELAY	The time in **milliseconds** that the broker will **hold the message** for before scheduling it for delivery.
AMQ_SCHEDULED_PERIOD	The time in milliseconds to **wait between successive message deliveries.**
AMQ_SHCEUDLED_REPEAT	The number of times the message will be **rescheduled for delivery** after the first scheduled delivery is completed.
AMQ_SCHEDULED_CRON	• Scheduled message delivery based on a standard CRON tab entry. • This value always takes priority over the delay and repeat period.

Apache ActiveMQ

Using JMS Request/Response Pattern

The CRON-based scheduler field can be used to do more complicated scheduling than can be done with the other fields.
Let's say you want to schedule a message for **delivery once an hour;** you can do so using code similar to the following:

```
MessageProducer producer = session.createProducer(destination);
TextMessage message = session.createTextMessage();
message.setStringProperty(ScheduledMessage.AMQ_SCHEDULED_CRON, "0 * * * *");
producer.send(message);
```

Apache ActiveMQ

Using JMS Request/Response Pattern

We can get even more elaborate by combining the CRON settings and the other settings for delay repeat and period. The CRON entry takes precedence over the other values.

For instance, if we have a message that should be delivered 10 times at the start of every hour and we wanted a 1 second delay between each message,

we could do so using the following code:

```
MessageProducer producer = session.createProducer(destination);
        TextMessage message = session.createTextMessage();
        message.setStringProperty(
            ScheduledMessage.AMQ_SCHEDULED_CRON, "0 * * * *");
        message.setLongProperty(
            ScheduledMessage.AMQ_SCHEDULED_DELAY, 1000);
        message.setLongProperty(
            ScheduledMessage.AMQ_SCHEDULED_PERIOD, 1000);
        message.setIntProperty(
            ScheduledMessage.AMQ_SCHEDULED_REPEAT, 9);
        producer.send(message);
```

Apache ActiveMQ

Use Case Demos

- Simple JMS Application
- Publish-Subscribe Application
- Message Selectors for Consumers
- Scheduling Message Delivery