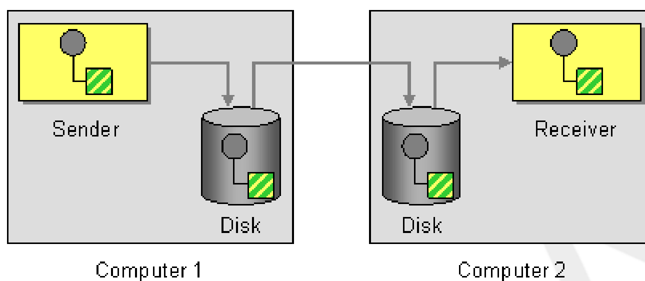
Guaranteed Delivery

Overview

Guaranteed delivery means that once a message has been pushed into a message channel, the messaging system guarantees that the message will reach its destination, even if parts of the application should fail. In general, messaging systems implement the guaranteed delivery pattern by writing messages to persistent storage before attempting to deliver them to their destination.

Figure 12. Guaranteed Delivery Pattern



Components that support guaranteed delivery

The following Mediation Router components support the guaranteed delivery pattern:

- JMS on page 44
- ActiveMQ on page 45
- ActiveMQ Journal on page 47

JMS

In JMS, the `deliveryPersistent` query option indicates whether persistent storage of messages is enabled or not. But it is normally unnecessary to set this option, because the default behavior is to enable persistent delivery. To configure all the details of guaranteed delivery, it is necessary to set configuration options on the JMS provider. These details vary, depending on what JMS provider you are using. For example, MQSeries, TibCo, BEA, Sonic, and so on, all provide various qualities of service to support guaranteed delivery.

DRAFT

See JMS in the *Component Reference* for more details.

ActiveMQ

In ActiveMQ, messaged persistence is enabled by default. From version 5 onwards, ActiveMQ uses the AMQ message store as the default persistence mechanism. There are several different approaches you can take to enabling message persistence in ActiveMQ.

The simplest option (different from the figure shown above) is to enable persistence in a central broker and then connect to the broker using a reliable protocol. After the message has been sent to the central broker, delivery to consumers is guaranteed. For example, in the Mediation Router configuration file, `META-INF/spring/camel-context.xml`, you could configure the ActiveMQ component to connect to the central broker using the OpenWire/TCP protocol as follows:

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.compon
ent.ActiveMQComponent">
    <property name="brokerURL" value="tcp://somehost:61616"/>

  </bean>
  ...
</beans>
```

If you prefer to implement an architecture where messages are stored locally before being sent to a remote endpoint (similar to the figure shown above), you can do this by instantiating an embedded broker in your Mediation Router application. A simple way to achieve this is to use the ActiveMQ Peer-to-Peer protocol, which implicitly creates an embedded broker in order to communicate with other peer endpoints. For example, in the `camel-context.xml` configuration file, you could configure the ActiveMQ component to connect to all of the peers in group, `GroupA`, as follows:

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.compon
ent.ActiveMQComponent">
    <property name="brokerURL" value="peer://GroupA/broker1"/>

  </bean>
  ...
</beans>
```

Where `broker1` is the broker name of the embedded broker (other peers in the group should use different broker names). One limiting feature of the Peer-to-Peer protocol is that it relies on IP multicast to locate the other peers in its group. This makes it unsuitable for use in wide area networks (and even some local area networks do not have IP multicast enabled).

A more flexible way to create an embedded broker in the ActiveMQ component is to exploit ActiveMQ's VM protocol, which connects to an embedded broker instance. If a broker of the required name does not already exist, the VM protocol automatically creates one. You can use this mechanism to create an embedded broker with custom configuration. For example:

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.compon
ent.ActiveMQComponent">
    <property name="brokerURL" value="vm://broker1?brokerCon
fig=xbean:activemq.xml"/>
  </bean>
  ...
</beans>
```

Where `activemq.xml` is an ActiveMQ file, which configures the embedded broker instance. Within the ActiveMQ configuration file, you can choose to enable one of the following persistence mechanisms:

- *AMQ persistence—(the default)* a fast and reliable message store that is native to ActiveMQ. For details, see amqPersistenceAdapter [http://activemq.apache.org/maven/activemq-core/xsddoc/http___activemq.org_config_1.0/element/amqPersistenceAdapter.html#] and AMQ Message Store [http://activemq.apache.org/amq-message-store.html].

- *JDBC persistence*—uses JDBC to store messages in any JDBC-compatible database. For details, see jdbcPersistenceAdapter [http://activemq.apache.org/maven/activemq-core/xsddoc/http___activemq.org_config_1.0/element/jdbcPersistenceAdapter.html#] and ActiveMQ Persistence [http://activemq.apache.org/persistence.html].

- *Journal persistence*—a fast persistence mechanism that stores messages in a rolling log file. For details, see journalPersistenceAdapter [http://activemq.apache.org/maven/activemq-core/xsddoc/http___activemq.org_config_1.0/element/journalPersistenceAdapter.html#] and ActiveMQ Persistence [http://activemq.apache.org/persistence.html].

- *Kaha persistence*—a persistence mechanism developed specially for ActiveMQ. For details, see kahaPersistenceAdapter

[http://activemq.apache.org/maven/activemq-core/xsddoc/http___activemq.org_config_1.0/element/kahaPersistenceAdapter.html#] and ActiveMQ Persistence [http://activemq.apache.org/persistence.html].

See ActiveMQ in the *Component Reference* for more details.

ActiveMQ Journal

The ActiveMQ Journal component is optimized for the special use case where multiple, concurrent producers write messages to queues, but there is only one active consumer. Messages are stored in rolling log files and concurrent writes are aggregated in order to boost efficiency.

See ActiveMQ Journal in the *Component Reference* for more details.

DRAFT