Middleware and Web Services

Lecture 5: Messaging Systems

doc. Ing. Tomáš Vitvar, Ph.D.
tomas@vitvar.com • @TomasVitvar • http://vitvar.com

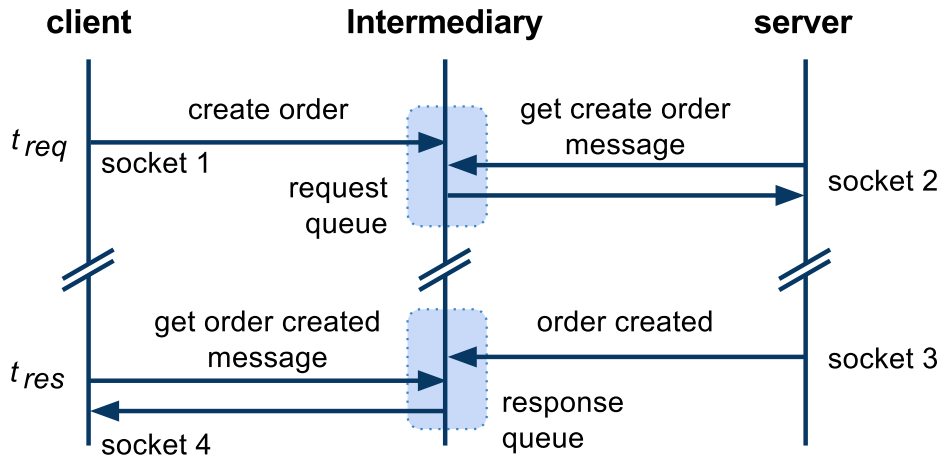
Czech Technical University in Prague
Faculty of Information Technologies • Software and Web Engineering • http://vitvar.com/courses/mdw

Modified: Thu May 18 2017, 10:31:52
Humla v0.3

Overview

- Messaging Systems
  - *Point-to-Point*
  - *Error Handling*
  - *Publish/Subscribe*
- Store and Forward

Recall: Asynchronous via Intermediary



- Intermediary
  - *A component that decouples a client-server communication*
  - *It increases reliability and performance*
    - → *The server may not be available when a client sends a request*
    - → *There can be multiple servers that can handle the request*

Messaging Systems

- Messaging Systems
  - *Also "Messaging Middleware" or "Message-Oriented Middleware" (MOM)*
  - *Two roles: a message consumer and a message producer*
  - *Asynchronous communication*
  - *"anonymity" between producers and consumers*
    → no matter "who", "where", "when" produced a message
  - *Ensures reliability and scalability*
- Loose coupling of applications
  - *A producer does not need to know about a consumer*
    → Messaging systems decouple a producer and a consumer
- Two types (Messaging Domains)
  - *Point-to-Point (message queue — MQ)*
  - *Publish/Subscribe (event-based)*

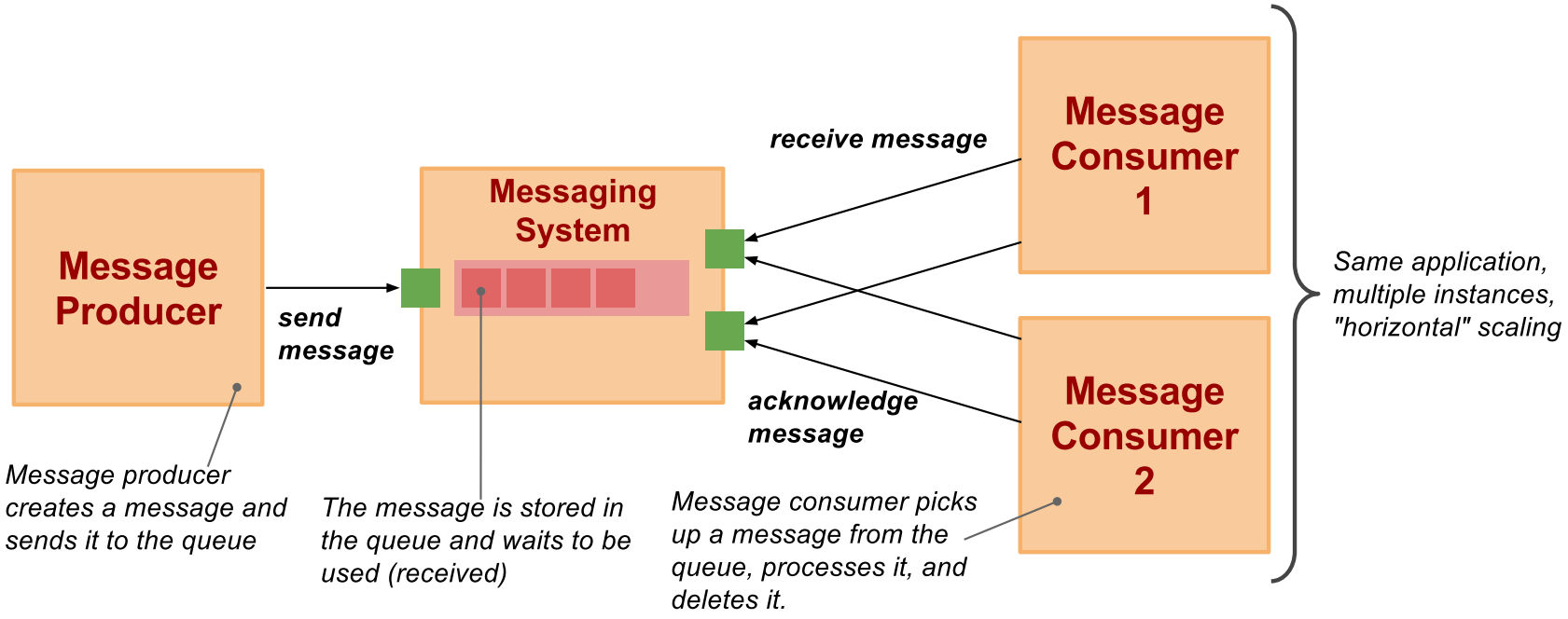
Java Message Service

- JMS – Java Message Service
  - *Java API for Message-Oriented Middleware*
  - *Java programs to create, send, receive and read messages*
  - *Standardized in JSR 914*
- Terminology
  - ***JMS Provider*** *– implementation of JMS system, part of application server*
  - ***JMS Client*** *– application that sends or receives JMS messages*
    - → ***JMS producer/publisher*** *– creates and sends messages*
    - → ***JMS consumer/subscriber*** *– receives and processes the messages*
  - ***JMS Message*** *– an object with data (payload) and properties*
  - ***JMS Queue*** *– storage that contains sent messages that are to be received; messages are processed only once*
  - ***JMS Topic*** *– storage that distributes messages to multiple subscribes*

Overview

- Messaging Systems
  - *Point-to-Point*
  - *Error Handling*
  - *Publish/Subscribe*
- Store and Forward

Conceptual Architecture



- *"1 : 1" relationship between a producer and a consumer*
  *→ **one message must be processed by one consumer***
- ***no time-dependency** between message producer and consumer*
  → consumer does not need to exist when producer sends a message
- *Message exists in the queue until it is used by a consumer*
- *message consumers take as many messages as they are able to serve*

Basic Types of Queues

- Queues in client-server architecture (request-response)
- Input Queue
  - *a client places a message to the queue*
  - *a server reads the message and process it*
- Output Queue
  - *a server places output message (response) to the queue*
  - *a client reads the message*
- Error Queue
  - *a server reads the message from the input queue*
  - *when processing of the message fails, the server places the message to the error queue*
  - *there can be several attempts to process the message before it is placed to the error queue*

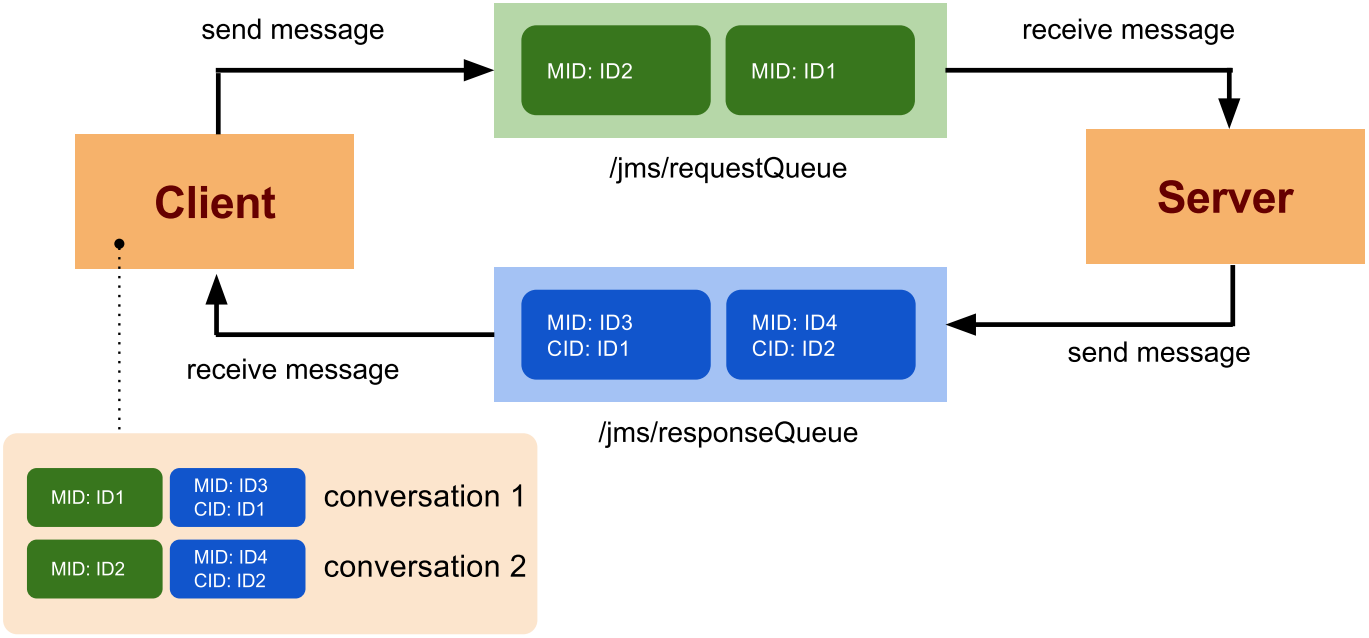
JMS Queues

- JMS Provider implmented by Weblogic
- Configuration
  1. *Create a **JMS server**, targeted to a managed server*
     - *In a cluster, every managed server has its own JMS server*
     - *JMS server has a persistent store where it stores messages in queues (persistent store can be file-based or JDBC-based)*
  2. *Create a queue, specify a JNDI name for the queue*
     - *a **queue** targeted to a single JMS server*
     - *a **distributed queue** targeted to the cluster (all JMS servers)*
  3. *Create a **connection factory** (optional), specify a JNDI name for the connection factory*
     - *A JMS client uses the connection factory to create a connection with the JMS server*
- Run a JMS client
  - *a JMS producer – sends a message to the queue*
  - *a JMS consumer – receives a message from the queue*

JMS Message

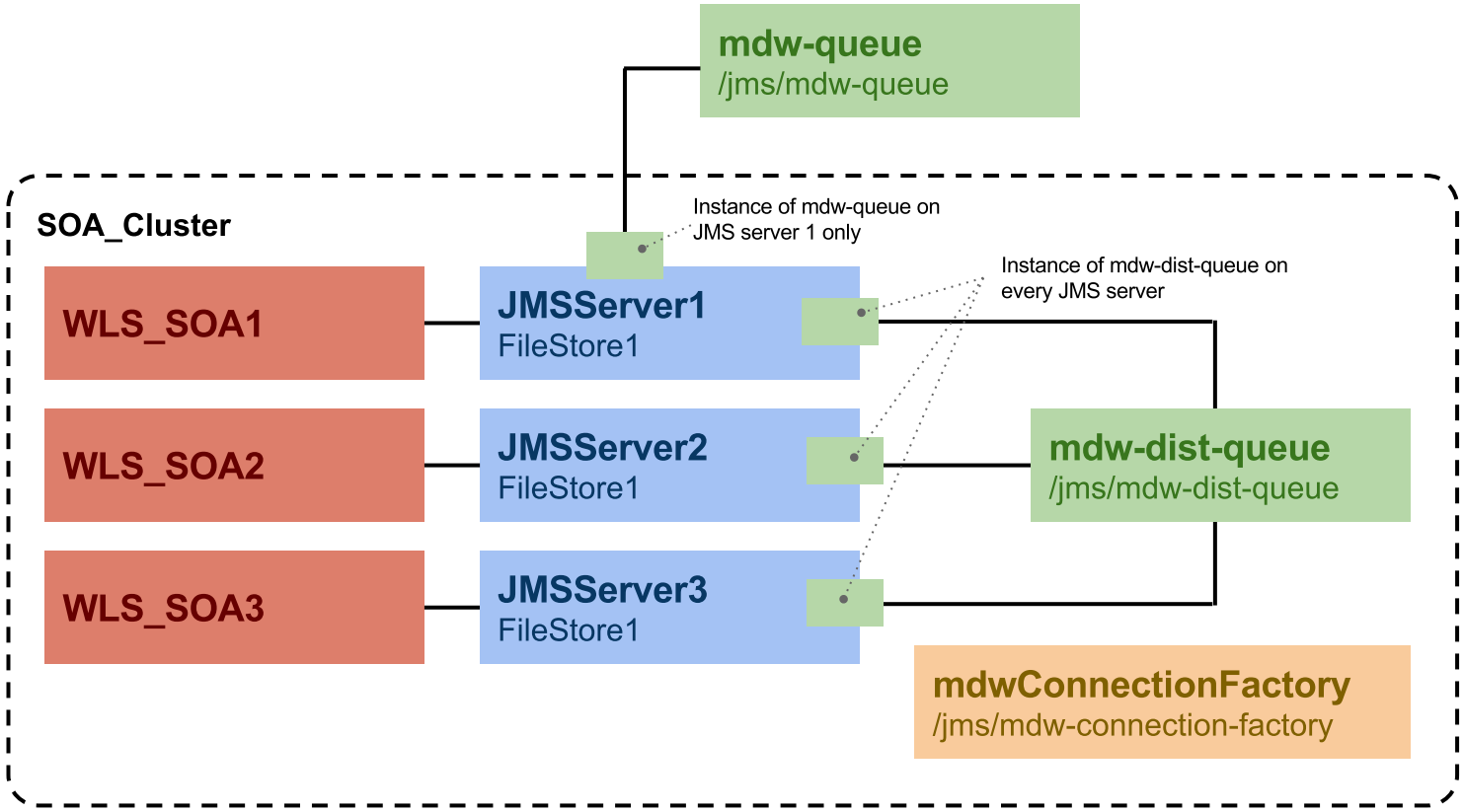
- JMS Message Header
  - *Priority*
    - → *priority that will be used to consume the message*
    - → *normal priority – 1-4, high priority 5-9*
  - *Delivery mode*
    - → *persistent – message stored in a storage during send operation*
    - → *non-persistent – JMS server only stores the message in memory*
  - *Time to live*
    - → *The time the message stays in the queue, the message is removed after it elapses*
    - → *The JMS consumer must consume the message before the time elapses*
  - *Message ID*
    - → *ID of the message set by the client (in request-response communication)*
  - *Reply To*
    - → *A response queue set by the client*
  - *Correlation ID*
    - → *ID of the message set by the receiver for response*
- Payload (data)
  - *text, map message (key-values), byte message, object message (serializable java object)*

Conversation in Request-Response

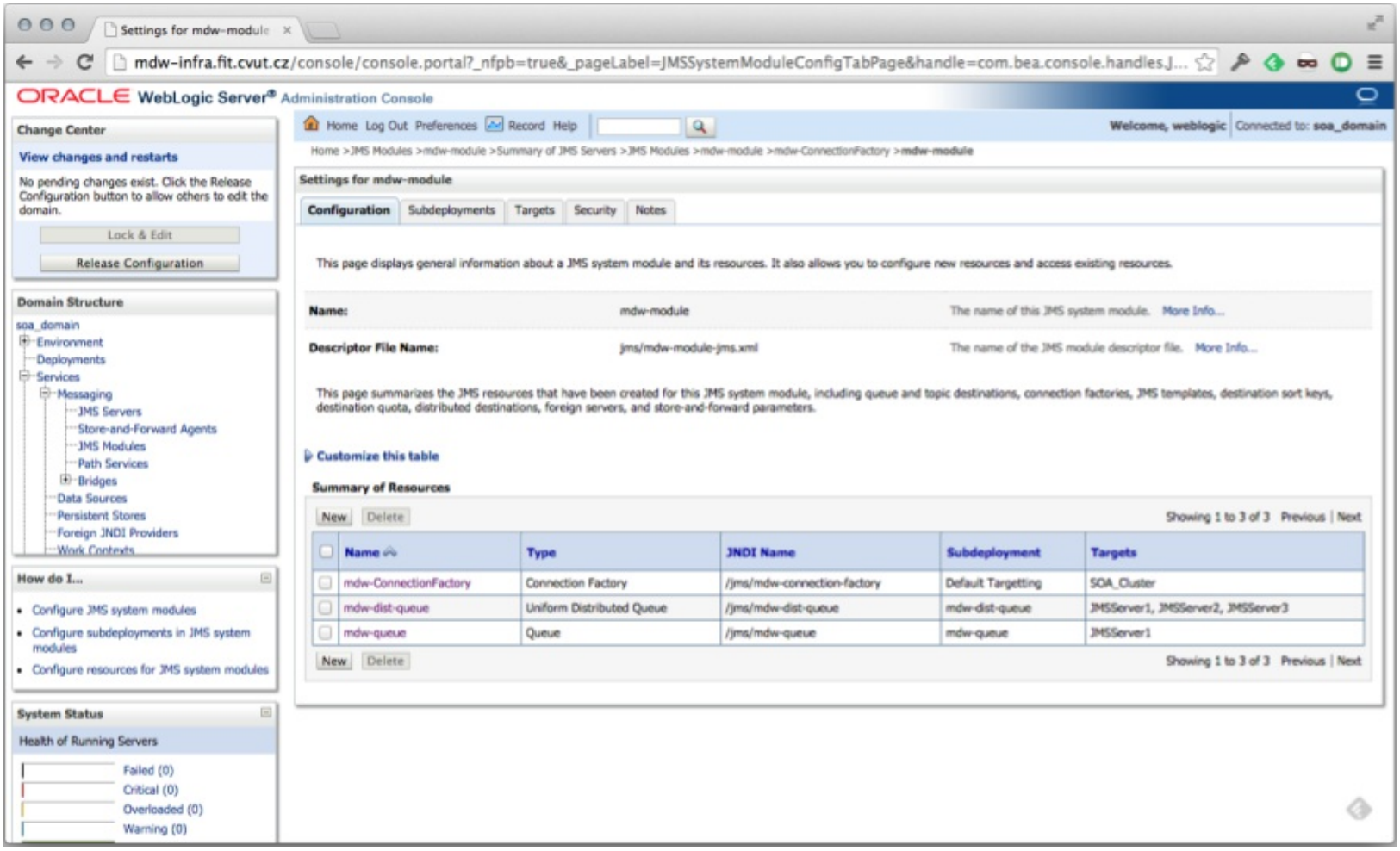


- Steps
  1. *Client sends a message to the request queue with some message ID1*
  2. *Server receives it, process it and sends a response to the response queue*
     - *The response message has its own message ID3 and a correlation ID that has a value of message ID1*
  3. *Client receives the response message and correlates with the request message*

Example Queues Configuration in Weblogic



Weblogic Configuration



JMS Producer Example (1)

```
1  public class JMSProducer {
2
3      // connection factory, connection, session, sender, message
4      private QueueConnectionFactory qconFactory;
5      private QueueConnection qcon;
6      private QueueSession qsession;
7      private QueueSender qsender;
8      private Queue queue;
9      private TextMessage msg;
10
11     // creates a connection to the WLS using a JNDI context
12     public void init(Context ctx, String queueName) throws NamingException, JMSException {
13
14         // creates connection factory based on JNDI and a connection, creates a session
15         qconFactory = (QueueConnectionFactory) ctx.lookup(Config.JMS_FACTORY);
16         qcon = qconFactory.createQueueConnection();
17         qsession = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
18
19         // lookups the queue using the JNDI context
20         queue = (Queue) ctx.lookup(queueName);
21
22         // create sender and message
23         qsender = qsession.createSender(queue);
24         msg = qsession.createTextMessage();
25     }
26
27     // close sender, connection and the session
28     public void close() throws JMSException {
29         qsender.close(); qsession.close();
30         qcon.close();
31     }
```

JMS Producer Example (2)

```
32      // sends the message to the queue
33      public void send(String queueName, String message) throws Exception {
34          // create a JNDI context to lookup JNDI objects (connection factory and queue)
35          Hashtable env = new Hashtable();
36          env.put(Context.INITIAL_CONTEXT_FACTORY, Config.JNDI_FACTORY);
37          env.put(Context.PROVIDER_URL, Config.PROVIDER_URL);
38
39          InitialContext ic = new InitialContext(env);
40          init(ic, queueName);
41
42          // send the message and close
43          try {
44              msg.setText(message);
45              qsender.send(msg, DeliveryMode.PERSISTENT, 8, 0);
46              System.out.println("The message was sent to the destination " +
47                      qsender.getDestination().toString());
48          } finally {
49              close();
50          }
51      }
52
53      public static void main(String[] args) throws Exception {
54          // JNDI name of the queue and a text message
55          String msg = args[0];
56          String queueName = args[1];
57
58          // create the producer object and send the message
59          JMSProducer producer = new JMSProducer();
60          producer.send(msg, queueName);
61      }
62  }
```

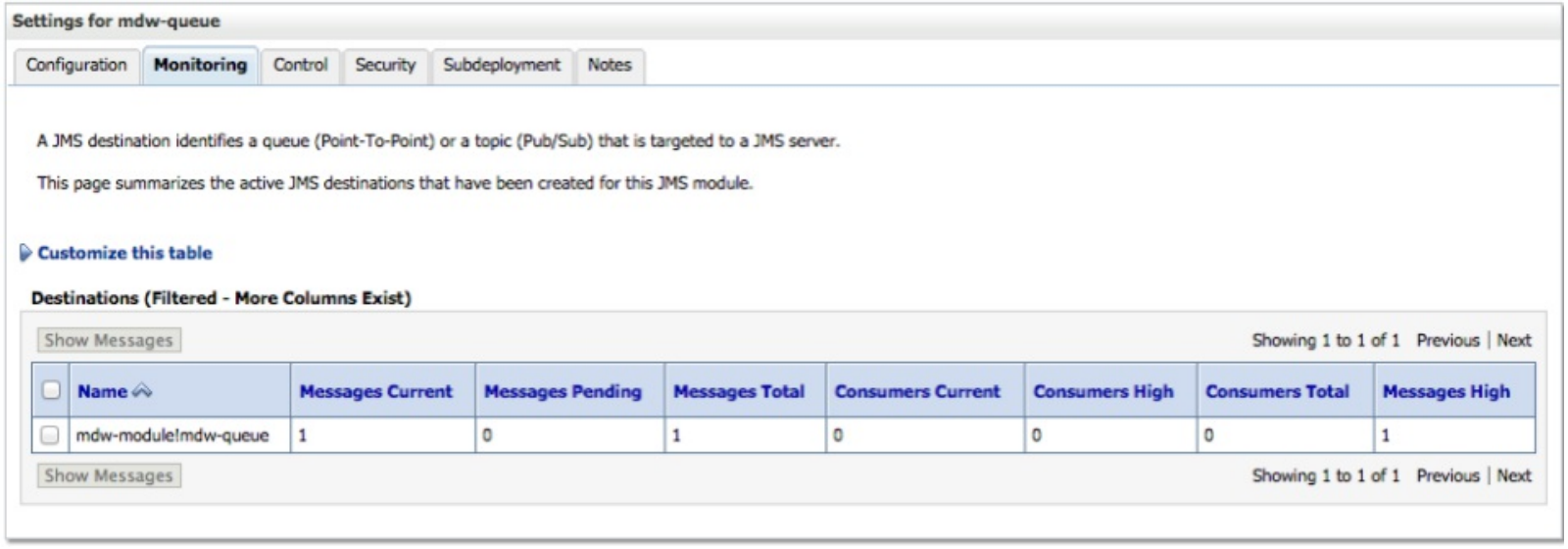
JMS Producer Example (3)

- Send a message to the queue
  - *You need to create a VPN connection to the environment*
    → *JMS Producer connects to the cluster*
    (or one of the managed servers in the cluster)
  - *Arguments: (1) JNDI name of the queue, and (2) text message*
    → `/jms/mdw-queue` *or* `/jms/mdw-dist-queue`

```
1  $ ./jmsproducer.sh /jms/mdw-queue message_from_mdw_lecture
2  $ The message was sent to the destination mdw-module!mdw-queue
```

Monitor the JMS Queue

- How many messages
  - ***Messages current***
    - → *number of messages in the queue waiting to be processed (**backlog**)*
  - ***Messages pending***
    - → *number of messages being processed (either being sent by a producer or being received by a consumer). Such messages have not been committed (acknowledged)*



JMS Consumer Example (1)

```java
public class JMSConsumer implements MessageListener {

    // connection factory
    private QueueConnectionFactory qconFactory;

    // connection to a queue
    private QueueConnection qcon;

    // session within a connection
    private QueueSession qsession;

    // queue receiver that receives a message to the queue
    private QueueReceiver qreceiver;

    // queue where the message will be sent to
    private Queue queue;

    // callback when the message exist in the queue
    public void onMessage(Message msg) {
        try {
            String msgText;
            if (msg instanceof TextMessage) {
                msgText = ((TextMessage) msg).getText();
            } else {
                msgText = msg.toString();
            }
            System.out.println("Message Received: " + msgText);
        } catch (JMSException jmse) {
            System.err.println("An exception occurred: " + jmse.getMessage());
        }
    }
```

JMS Consumer Example (2)

```
32  // create a connection to the WLS using a JNDI context
33  public void init(Context ctx, String queueName)
34          throws NamingException, JMSException {
35
36      qconFactory = (QueueConnectionFactory) ctx.lookup(Config.JMS_FACTORY);
37      qcon = qconFactory.createQueueConnection();
38      qsession = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
39      queue = (Queue) ctx.lookup(queueName);
40
41      qreceiver = qsession.createReceiver(queue);
42      qreceiver.setMessageListener(this);
43
44      qcon.start();
45  }
46
47  // start receiving messages from the queue
48  public void receive(String queueName) throws Exception {
49      Hashtable env = new Hashtable();
50      env.put(Context.INITIAL_CONTEXT_FACTORY, Config.JNDI_FACTORY);
51      env.put(Context.PROVIDER_URL, Config.PROVIDER_URL);
52
53      InitialContext ic = new InitialContext(env);
54      init(ic, queueName);
55
56      System.out.println("Connected to " + queue.toString() + ", receiving messages...");
57
58      // loop until ctrl+c
59      while (true) {
60          this.wait();
61      }
62  }
```

JMS Consumer Example (3)

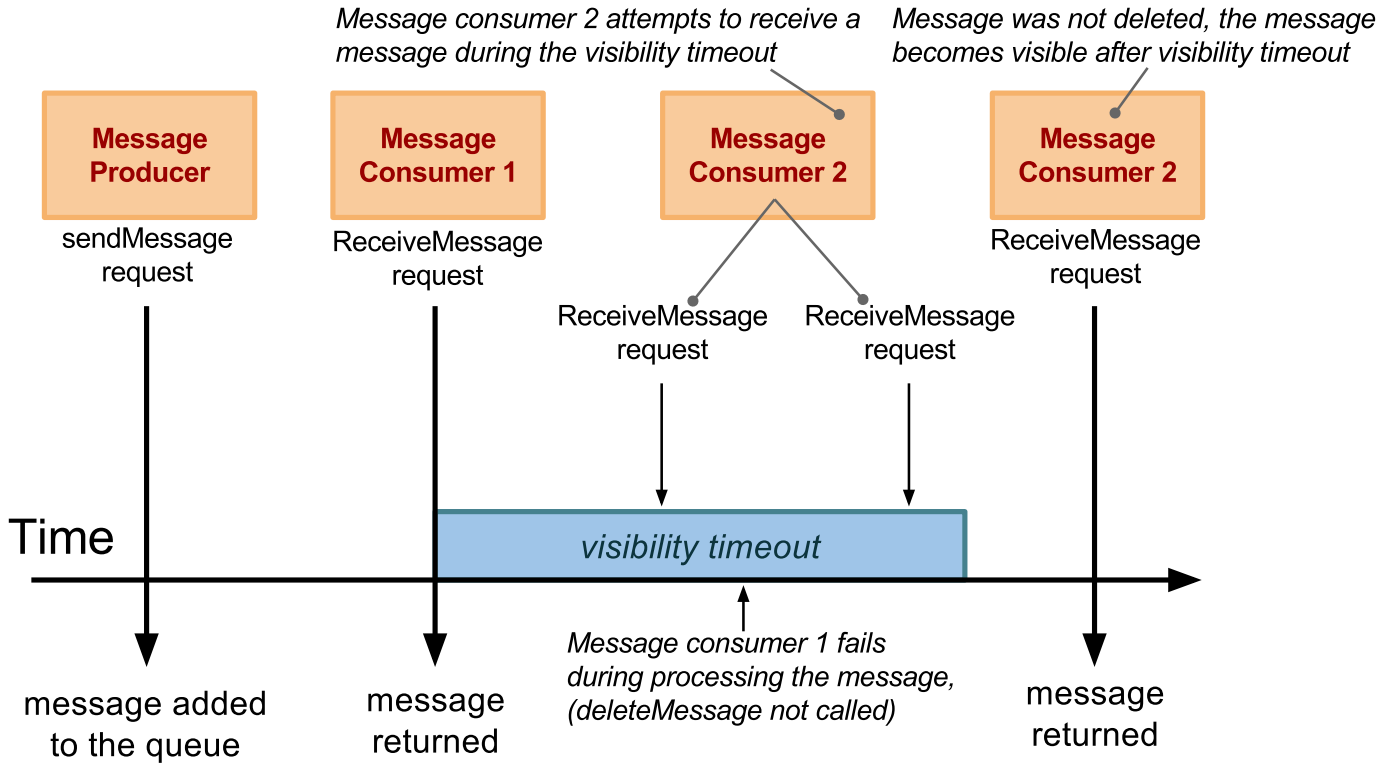
- Receive a message from the queue
  - *You need to create a VPN connection to the environment*
    → *JMS Consumer connects to the cluster*
    (or one of the managed servers in the cluster)
  - *Arguments: (1) JNDI name of the queue*
    → `/jms/mdw-queue` *or* `/jms/mdw-dist-queue`

```
1  $ ./jmsconsumer.sh /jms/mdw-queue
2  $ Connected to mdw-module!mdw-queue, receiving messages...
3    Message Received: message-from-mdw-lecture1
4    Message Received: message-from-mdw-lecture2
```

Overview

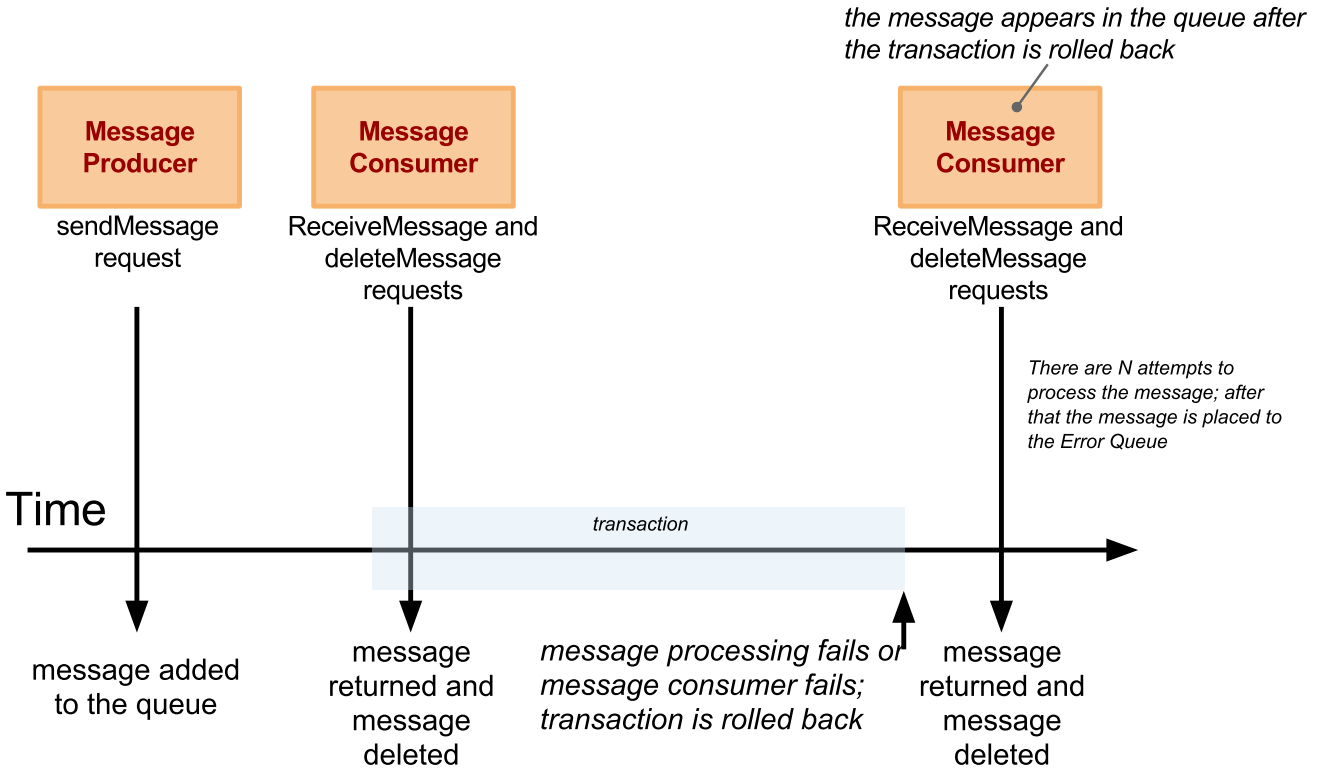
- Messaging Systems
  - *Point-to-Point*
  - *Error Handling*
  - *Publish/Subscribe*
- Store and Forward

Error Handling Using Timeout



- *message consumers or message processing may fail*
- ***visibility timeout** – time during which the message exist in the queue, and need to be deleted by the consumer (~ 30 seconds)*
- *Example technology: Amazon Simple Queue Service (SQS)*

Error Handling Using Transactions

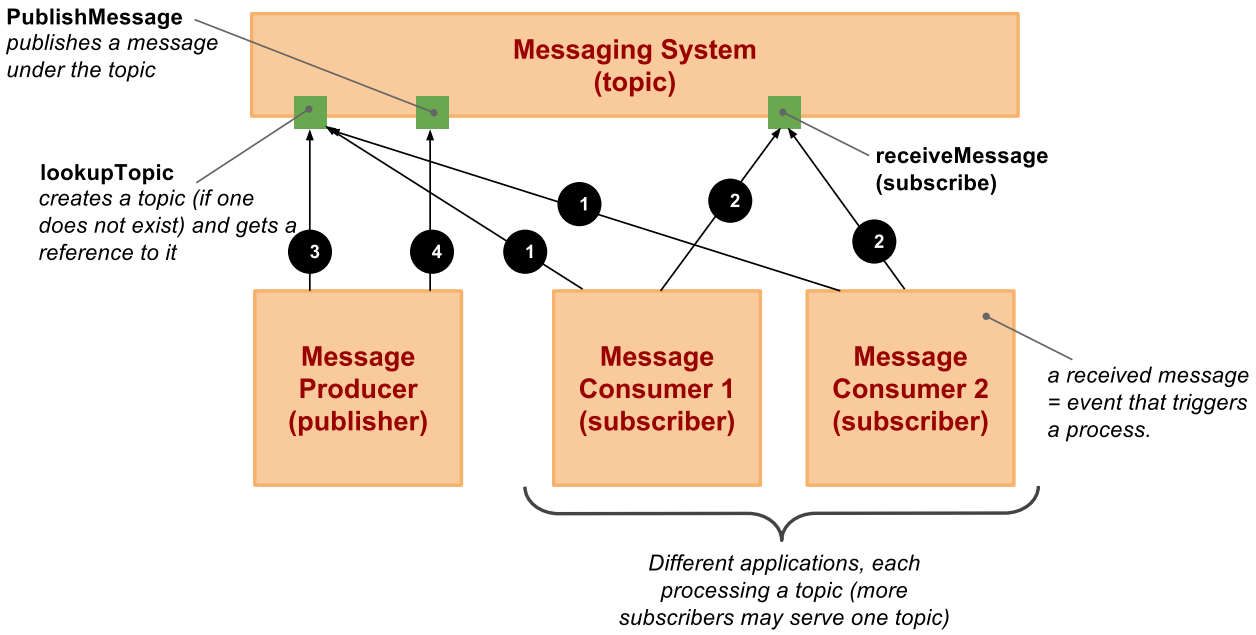


- ***transaction*** *– message consumer opens a transaction and deletes the message; when the processing is succesfull the transaction is committed otherwise it is rolled back and the message appears in the queue again*
- *Example technology: JMS, Weblogic server*

Overview

- Messaging Systems
  - *Point-to-Point*
  - *Error Handling*
  - *Publish/Subscribe*
- Store and Forward

Publish/Subscribe System



- *occurrence of a message = event that triggers one or more processes*
- *a "1 : N" relationship between producer and consumer*
  → ***one message can be processed by many different subscribers***
- ***time-dependency*** *between publisher and subscriber*
  → subscriber must first subscribe to a topic and then publisher can publish a message under that topic
- *a message is deleted when all its subscribers consume it*

Publish/Subscribe API

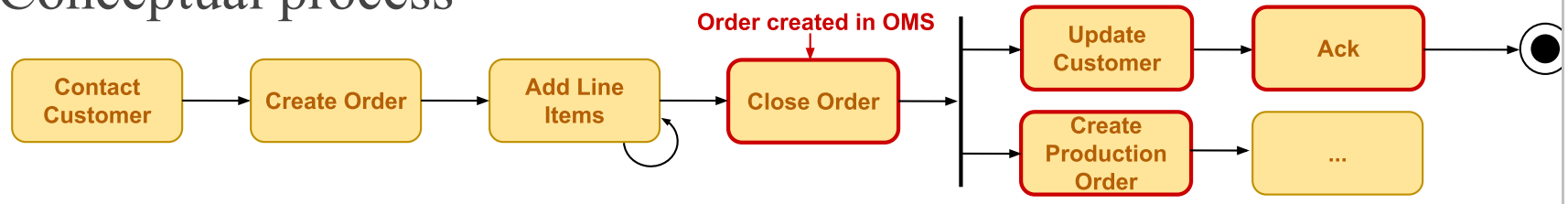
- `lookupTopic`
  - *lookups or creates a topic*
  - *called by the subscriber first and then by the publisher*
- `receiveMessage`
  - *request to receive (read) a message under the topic*
  - *called by the subscriber*
  - *Implementation specific:*
    - → *synchronous – blocking, with timeout*
    - → *asynchronous – through event listener*
- `publishMessage`
  - *publishes a message under the topic*
  - *called by the publisher*

Event-driven Communication

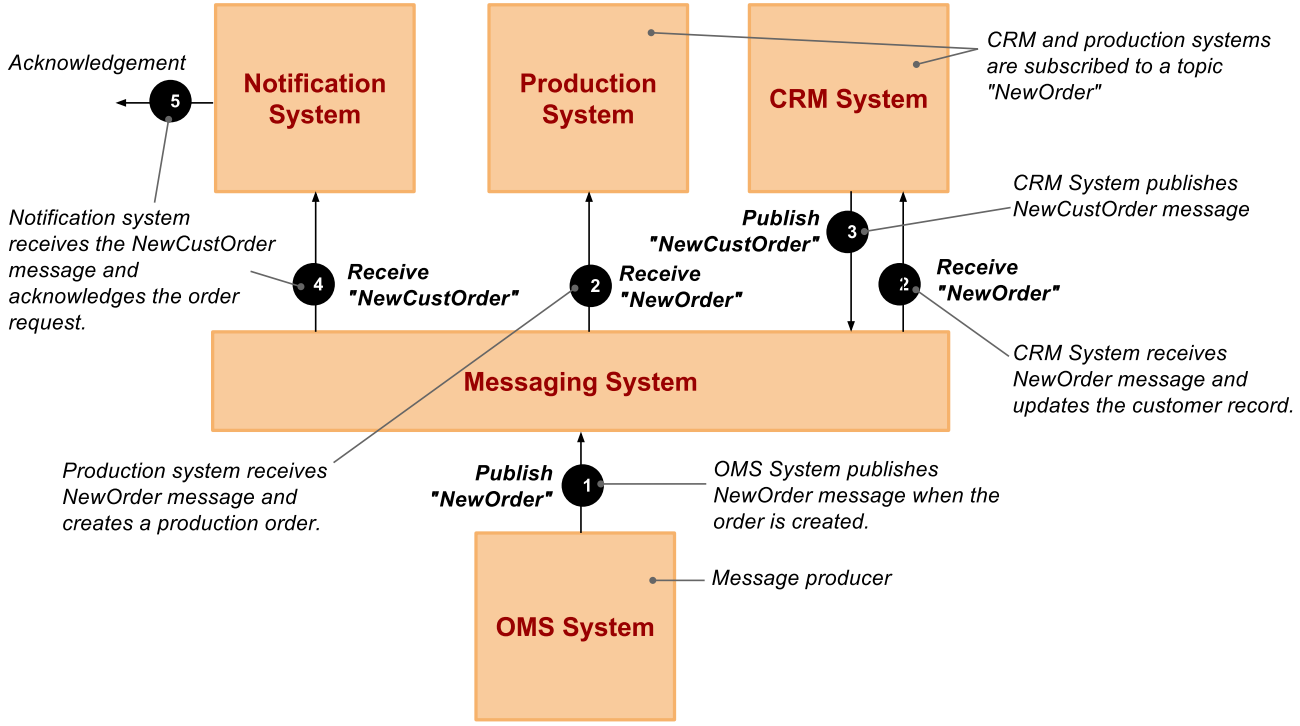
- Event
  - *Occurrence of a message with certain topic*
- Event-driven Process
  - *events trigger actions*
  - *one event may trigger more actions*
  - *loose coupling – not all actions need to be known at design time*

Event–driven Process Example

- Conceptual process



- Event-driven process implementation



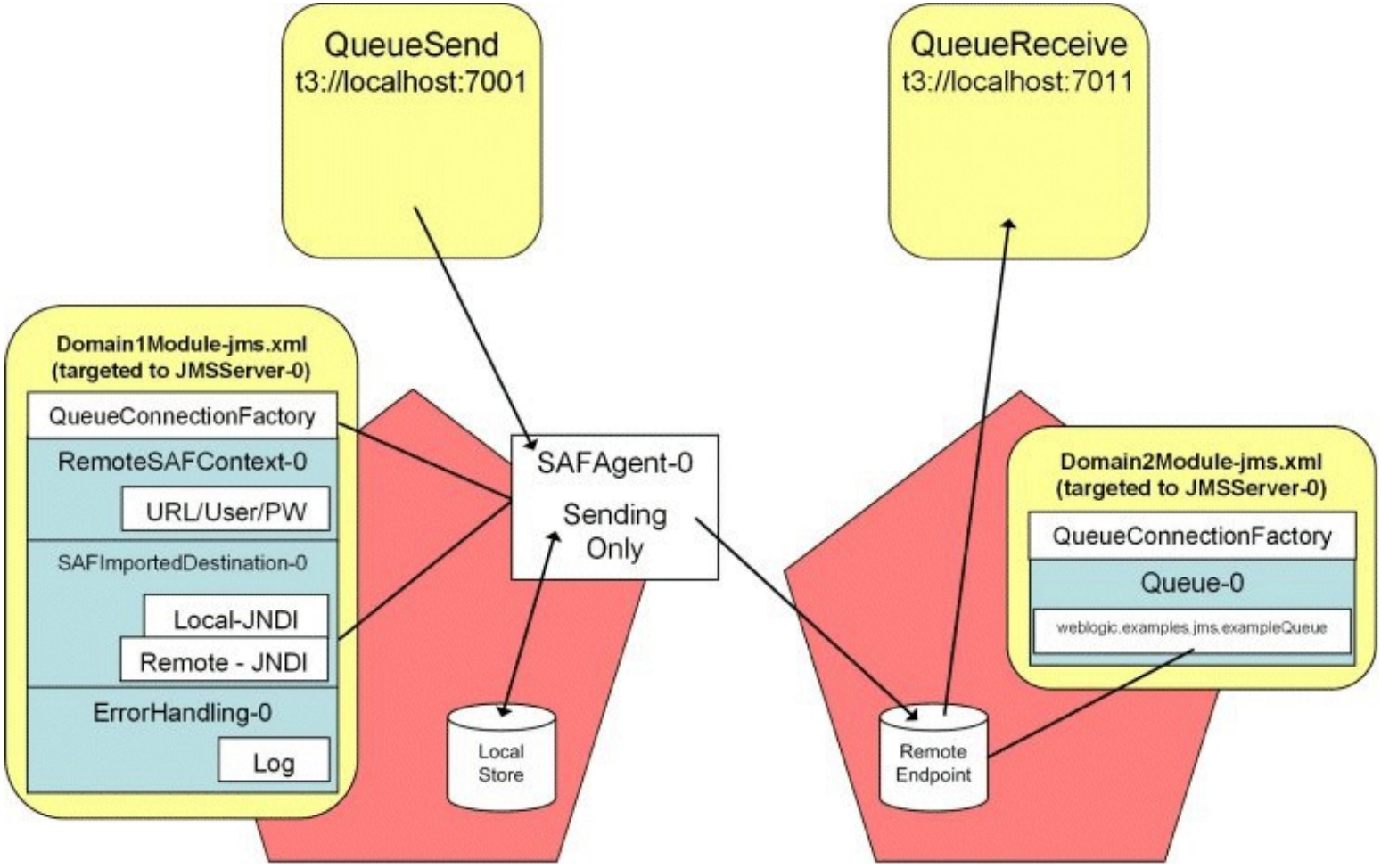
Overview

- Messaging Systems
- Store and Forward

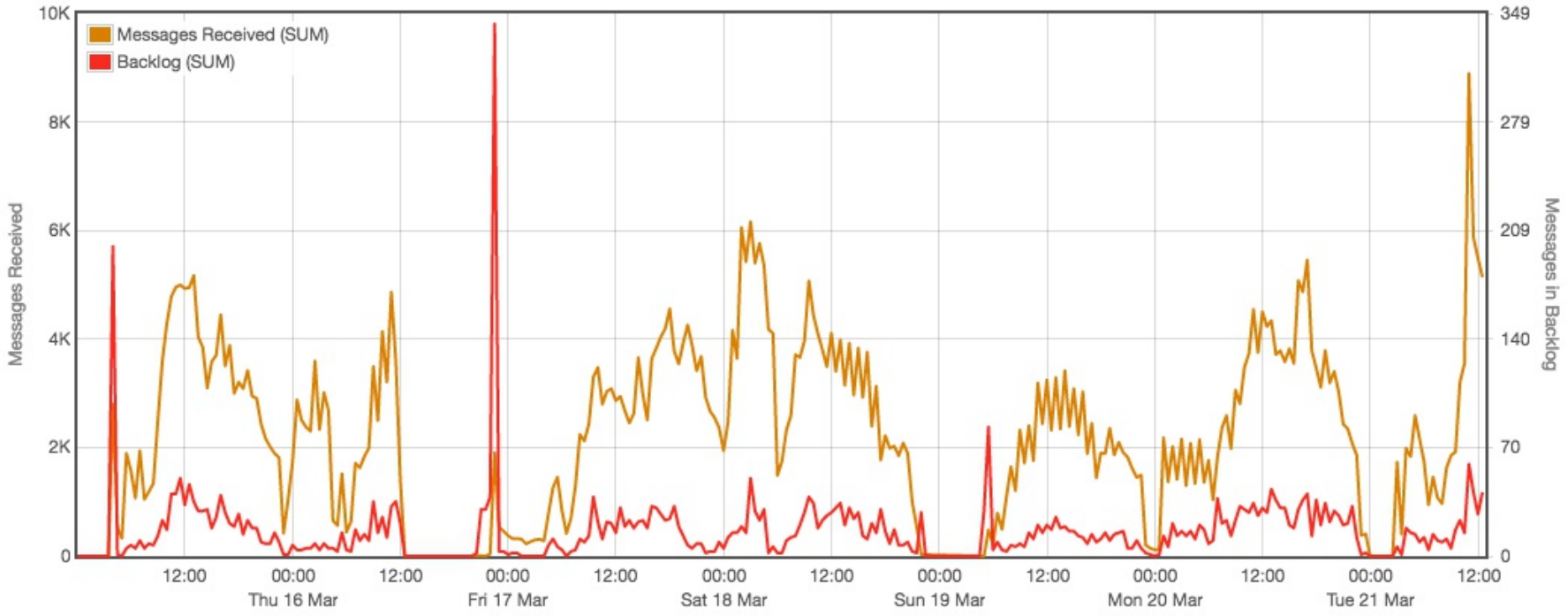
Store and Forward Agents

- Motivation
  - *Several environments (integration middleware, OSM, CRM)*
  - *Sending messages across environments*
  - *A need to "decouple" environments*
  - *Destination (queue) might not be always available*
    - → *Destination environment is down or busy*
- SAF = Store and Forward
  - *Agent – a component deployed to an application server instance*
    - → *has a queue and its own storage*
    - → *has configured a remote destination*
- Scenario
  1. *JMS producer sends a message to the agent's queue*
  2. *Agent forwards the message to the remote destination*
  3. *When the remote destination is not available, the agent keeps the message in its queue and retries to send the message after some time*
  4. *As a result of the unavailability, there can be a **backlog** of messages in the agent's queue*

Example SAF Agent Architecture



Example SAF Agent Backlog



- Messages received
  - *number of messages received to the SAF agent queue in every hour*
- Backlog
  - *number of messages waiting to be sent across to the destination queue*
  - *Destination system cannot catch up with number of messages being sent across*