Distributed Systems

Messaging and JMS
Example scenario

- **Scenario:**
  - Store inventory is low
  - This impacts multiple departments

- **Inventory**
  - Sends a message to the factory when the inventory level for a product goes below a certain level so that the factory can provide more products.

- **Factory**
  - Sends a message to the parts components to provide more parts for it to assemble.

- **Parts**
  - Sends messages to their own inventory and order components to update their inventories and to order new parts from suppliers.

- **Factory and Parts**
  - Sends messages to the accounting component to update their budget numbers.

- **Sales**
  - Publishes updated catalog items to its sales force.

Source: Java EE 7 Tutorial
https://docs.oracle.com/javaee/7/tutorial/jms-concepts001.htm
Example scenario

• What communication style should we use between components?
  – RMI?
  – Shared database?
  – HTTP?

• What if:
  – Each of these software components are in different regions of the world?
  – The components are managed by different IT departments on different schedules?
  – Some components are legacy systems built on different software platforms/languages?
Example scenario

• What would be desirable in terms of dealing with:
  – Legacy systems?
  – Multiple platforms?
  – Some systems might be down with others are up?
  – Account for each department managing their own computing capacity by adding and removing resources to fit their needs?

• What we need is a way to have the system be:
  – Decoupled in space
  – Decoupled in time

• Indirect messaging provides a paradigm for meeting these needs.
Decoupled in space

Using Indirect Messaging...

• Sender does not need to know the identify of the receiver(s) and visa-versa

• Therefore, as long as a standard message format is defined, each department can:
  – modify their component
  – replace their component
  – replicate their component

• Good for handling legacy systems
  – E.g. Inventory does not need to write code to the Factory legacy API, only send agreed-upon formatted messages.
Decoupled in Time

Using Indirect Messaging...

- A component need not even be running
- The messaging system can store messages until they are successfully delivered
- Reliable delivery is insured
- E.g.
  - Parts Order might only run once a day.
  - If Sales is not currently receiving messages, it will eventually
  - If Accounting is running at capacity, messages are queued until it is ready for new ones
Two Messaging Modes

- **Point to Point**
  - Inventory to Factory
  - Inventory to Sales
  - Factory to Accounting

- **Publish / Subscribe**
  - Parts to Parts Inventory and Parts Order
*coupled

"Decoupled"
also known as:
"uncoupled"
"loosely coupled"
Messaging allows you to

- Abstract away a service provider interface (i.e. a software component API).
  - Providers can be replaced easily.
  - Providers implemented in multiple languages can interact easily.
- Similarly, abstract away a legacy system’s interface.
  - Allow multiple legacy systems to interact, and interact with clients.
- Allow for redundancy of service providers.
  - For scaling: add additional service providers
  - For fault tolerance: add alternative service providers
- Allow for service providers to be off-line
  - Messages are saved in a queue until the service provider is available again.
- Allow for multiple stages of processing to progress asynchronously.
In the non-IT world

• What are examples of indirect messaging in the physical (non-IT) world?
  – How is it decoupled in space?
  – How is it decoupled in time?
Some example scenarios

- Asynchronous communication
- Event-driven systems
- Multiple potential consumers
  - I.e. one of a set could handle the message
- Multiple interested parties
  - I.e. multiple systems want the message
Asynchronous Communication

• Enterprise chat and twitter type applications

• Systems reporting information to one or more other interested systems
Event-Driven Problem

• Something notable happens in your organization
  – User action on a web site
  – Transaction arriving via the net
  – Sensor threshold being met

• It requires a number of services to be done
  – From simple (e.g. logging) to whole business processes

• These need to be performed, some serially, some in parallel

• But they don’t need a response
Decoupled/Multiple Consumers

• You can decouple the producer of events from their consumers
• You can have multiple consumers available in each stage.
  – E.g. An electronic transaction arrives, any of our 20 transaction servers can service it
• Therefore as a developer, put it on a queue, and “The next available operator will be with your shortly.”
Multiple Interested Parties

• Similarly, an event might be of interest to multiple parties.
• E.g. The sale of an item might interest
  – Accounting to update books
  – Inventory to update stock
  – Purchasing to reorder stock
  – Sales floor to restock shelves
Message Oriented Middleware

Client Application

MOM (Queues)

Client Application

Client Application

Client Application
Message Oriented Middleware

• Some include:
  – Apache ActiveMQ
  – Oracle Weblogic
  – Oracle AQ
  – Jboss HornetQ
  – IBM WebSphere Application Server
  – IBM WebSphere MQ
Indirect Messaging Protocols

• Most have proprietary message protocols and formats
• Two open standards:
  – Extensible Messaging and Presence Protocol (XMPP)
    • Older, based on Jabber, has grown out of instant messaging
    • XML based message formatting
    • http://xmpp.org
  – Advanced Message Queuing Protocol (AMQP)
    • Newer, grew out of banking, gaining broad support
    • Defines the format of messages
    • Allows for interoperability for systems who can produce and consume these formatted messages.
    • http://www.amqp.org
  – Both have wide (though different) adoption
Java Message Service (JMS)

- An API for performing indirect messaging.
- It is an abstraction API like JNDI and JDBC.
- Interacts with some Message Oriented Middleware (MOM)
- JMS is a client-facing interface, meant to abstract way the particulars of any MOM.
- In theory, you should have portability of systems written with JMS such that they can work with any MOM.
- API is javax.jms
Compatibility

• JMS systems can interact with non-JMS systems
  – Such as another system that uses the native messaging system's API

• The JMS connector is responsible for matching message formats.

• Two vendor options
  – Translate JMS into native mode
    • Can interoperate with non-JMS clients
  – Encapsulate JMS within native mode
    • Only JMS clients can communicate
Our Class MOM 😊

- We’ll be using the MOM built into Glassfish
JMS !≠ MOM

• JMS is not a MOM
• For example:
  – IBM WebSphere MQ is a MOM
  – You can program applications in any language (e.g. C++, .NET) to interact with WebSphere
  – You can use "WebSphere MQ classes for Java" to:
    • Connect as a WebSphere MQ client
    • Directly access the full WebSphere MQ API
• Alternatively
  – WebSphere MQ supports a JMS interface
  – You can use JMS to interact with WebSphere MQ
  – This system written using the JMS API could be moved to use another MOM (e.g. GlassFish).
Two modes

Message Communications Modes

• **Point to point** (In JMS called Queues)
  – Decoupled
  – Guaranteed delivery to one and only one message handler
  – Might be multiple such handlers that could take a request

• **Publish Subscribe** (In JMS called Topics)
  – Decoupled
  – Similar to a bulletin board or newsgroup
    • Potentially one to many
  – One publishes to a topic and zero to many subscribe to it
  – Guarantees may be configured.
    • If no consumer currently cares about this event it expires
    • Or messages can be held until consumers are available
Main Components

• Messaging clients
  – produce and consume messages

• Message destinations
  – Queues/Topics to send and receive messages

• A JMS-Compatible MOM
Queues / Topics

• Queues and Topics are *administratively managed resources*
  – They are not part of the client's software
  – They are created and destroyed by the MOM, not by the client
    • You log into the MOM administrative interface and create or destroy them.
    • Or you define them in a deployment descriptor or an annotation.

• Glassfish admin
  – You create and destroy Queues and Topics in the administrative interface.
JMS API

Connection Factory
  └── Connection
    └── Message Producer
        ├── Session
        │   └── Message
        │       └── Destination
        └── Message Consumer
            └── Destination

creates
  └── creates
    └── creates
          └── receives from
            └── sends to
Creating a ConnectionFactory and Destinations

These are created in the GlassFish Admin Console

Destination == Queue or Topic
Creating Queues & Topics

- Start Glassfish
- View Admin Console
- Under Resources -> JMS Resources
- Add Connection Factory
- Add Destination

(Detailed directions will be in a lab.)
JMS API

@Resource(lookup = "jms/myConnectionFactory")
private ConnectionFactory cf;

@Resource(lookup = "jms/myQueue")
private Queue q;

Destination resources may be Queues or Topics.
Connection con = cf.createConnection();

A live connection to the MOM.

You typically only need one Connection to the MOM.

Must be ‘started’ before receiving messages.
(Automatic in Java 7, but good to be safe.)
Session session = con.createSession();

A single, serialized flow of messages between the client and the MOM. This flow may be transacted.

You can have multiple Sessions on a Connection. E.g. A Session for each Queue or Topic being read or written to.
JMS API

The producer writes to a Queue or Topic.

The consumer receives from a Queue or Topic.

Alternatively, can be done by implementing MessageListener and the onMessage() method.

MessageProducer writer = session.createProducer(q);
JMS API

Connection Factory
creates
Connection
creates
Message Producer
Session
creates
Message Consumer
creates
Message
receives from
Destination
sends to
Destination

Subtypes provide implementations for different types of content.

TextMessage msg = session.createTextMessage();
JMS Message Types

• Stream
  – Sequential stream of Java primitive data types.

• Map
  – Set of name-value pairs
    • Names are String objects
    • Values are Java primitives (including String)
  – Can be accessed sequentially or randomly by name.

• Text
  – Message is a String object
    • Plain-text message
    • XML messages

• Object
  – Serialized Java object
    • Simple if in Java-only environment

• Bytes
  – Stream of un-interpreted bytes.
  – To encode a message body to match an existing message format
Message Driven Bean

- **Enterprise JavaBean (EJB)**
  - server-side architecture for creating and managing modular components of enterprise applications
  - we don't cover these in DS

- **Message Driven Bean (MDB)**
  - components that are executed asynchronously by messages coming available in a Queue or Topic.
Message Writing & Reading

• Message Production
  – Only 1 way: Synchronous
    • MessageProducer writer...
    • writer.send(message);

• Message Consumption
  – Two ways:
    • Asynchronous:
      – Register as a listener on a Queue or Topic
    • Synchronous:
      – Read and block until message available (or timeout)
Message Consumption

• Asynchronous
  – A client registers a *listener* with the messaging system to be called when a message is available
  – Uses Message Driven Bean
Message Consumption

- Synchronous
  - Uses a MessageConsumer
  - A client blocks and waits for a message to arrive
  - Block forever, or wait only for a *timeout* period
MessageConsumer

// Lookup the ConnectionFactory using resource injection and assign to cf
@Resource(lookup = "jms/myConnectionFactory")
private ConnectionFactory cf;

// lookup the Queue using resource injection and assign to q
@Resource(lookup = "jms/myQueueThree")
private Queue q;

// ... then later

// With the ConnectionFactory, establish a Connection, and then a Session on that Connection
Connection con = cf.createConnection();
Session session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
con.start(); // Be sure to start to connection!

MessageConsumer reader = session.createConsumer(q);
TextMessage tm = null;
while ((tm = (TextMessage) reader.receive(1000)) != null) {
    // Do something with tm.getText()
}

// Close the connection
con.close();

This is covered in video 4 of JMS lab
Consuming Messages

• Could a servlet be a queue listener?
Can you have multiple clients?

What would the Queues be? What if the consumer was to send back a response? What problems would arise?
Temporary Queues

// With a Context, ConnectionFactory, Connection, and QueueSession (named qsession)

// MAKE SURE TO START THE CONNECTION!!
con.start();

// On the QueueSession, create a TemporaryQueue
Queue replyQueue = qsession.createTemporaryQueue();

// With a Message (msg) of any type, set the replyTo to the Temporary Queue
msg.setJMSReplyTo(replyQueue);

// Publish the message to a Topic (or send to a Queue).
// The receiver can reply via the replyQueue
writer.publish(msg);

// Create a MessageConsumer
MessageConsumer reader = session.createConsumer(replyQueue);

// Receive from the Queue
MapMessage receivemsg = (MapMessage) reader.receive(1000); // timeout in 1 second
Temporary Queues

// On the receiving side, for example in a Message Driven Bean

// Get the Temporary Queue from the Message “tm”
Queue replyQueue = (Queue) tm.getJMSReplyTo();

// Create a MessageProducer to write to the queue
MessageProducer writer = session.createProducer(replyQueue);

// With new message “msg”, send it to the queue, back to the source of the Message “tm”
writer.send(msg);
Multiple Subscribers

• What would happen if you had multiple Topic subscribers?
  – How is it similar or different than having multiple Message Driven Beans listening to a Queue?