Architecture of Enterprise Applications 8
Messaging

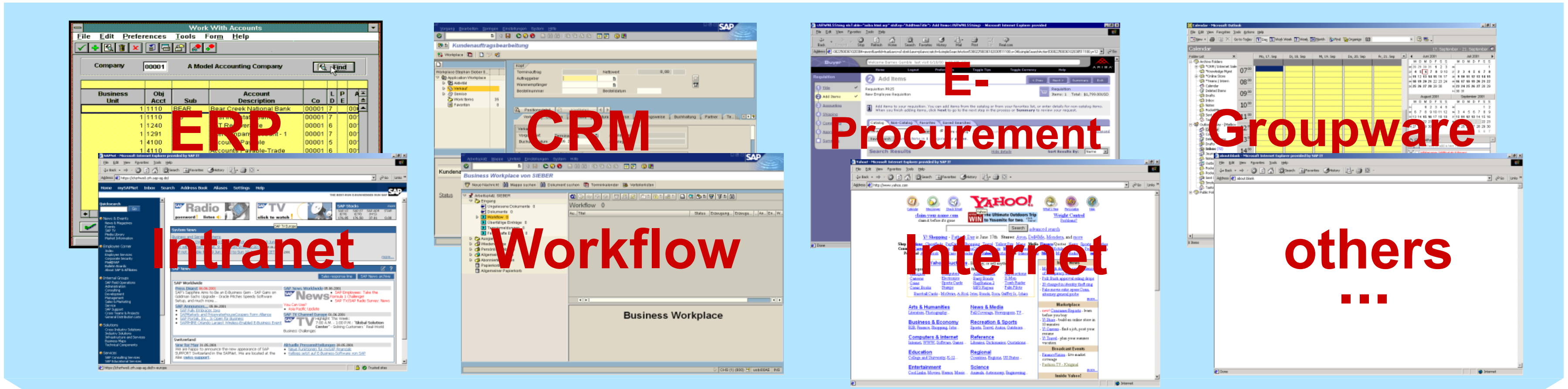
Haopeng Chen

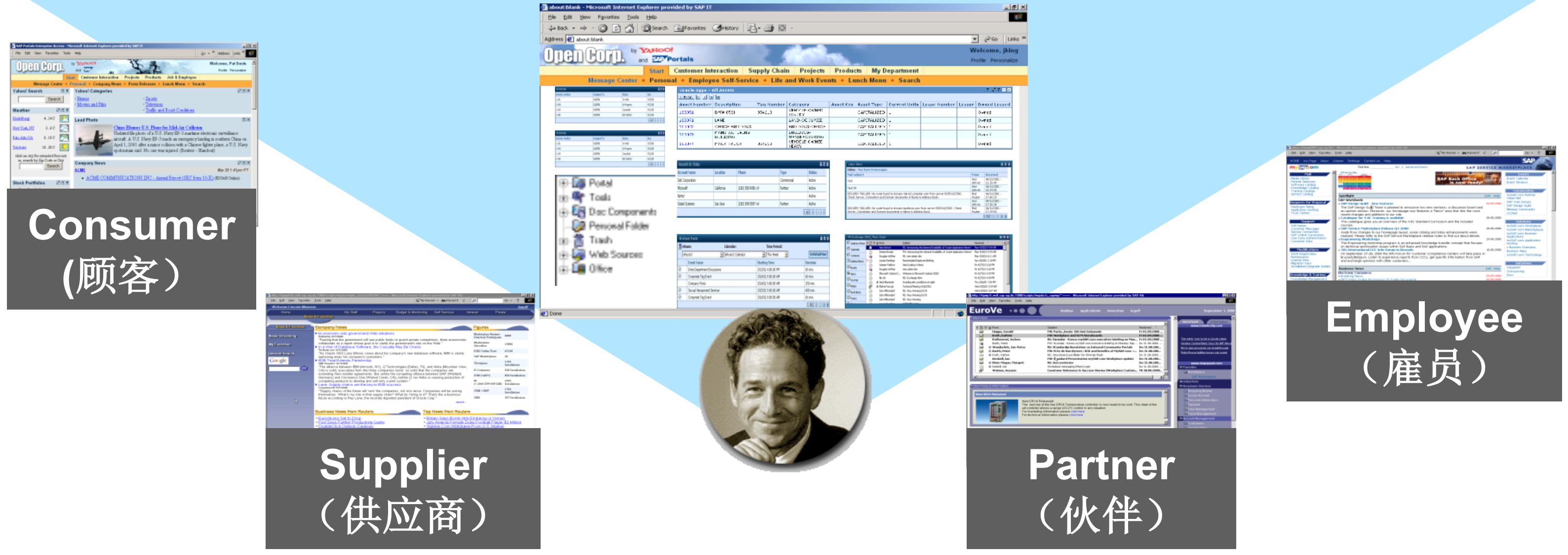
RE*liable,* ***IN****telligent and* ***S****calable Systems Group (****REINS****)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents

Challenge

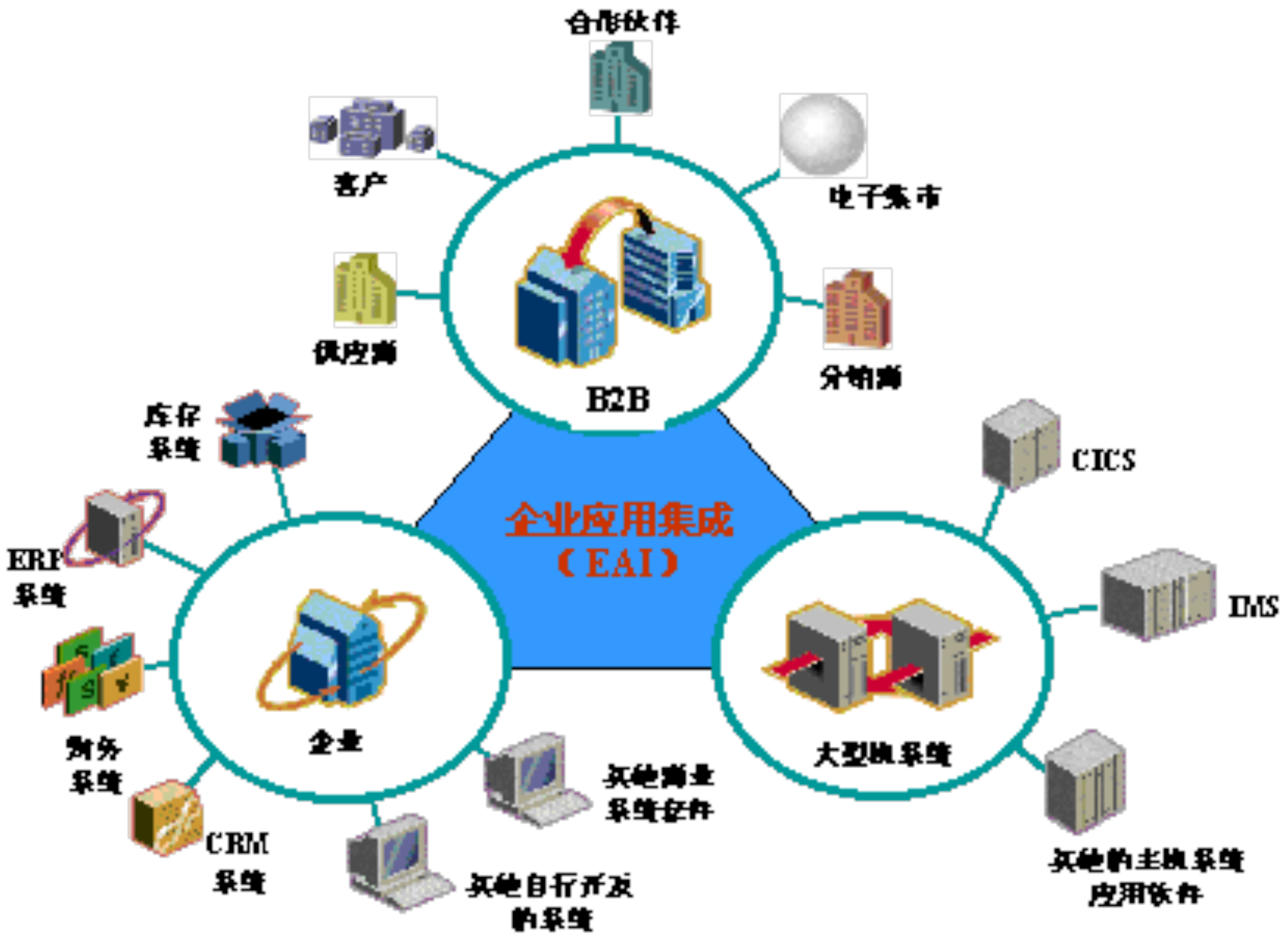
REliable, INtelligent & Scalable Systems





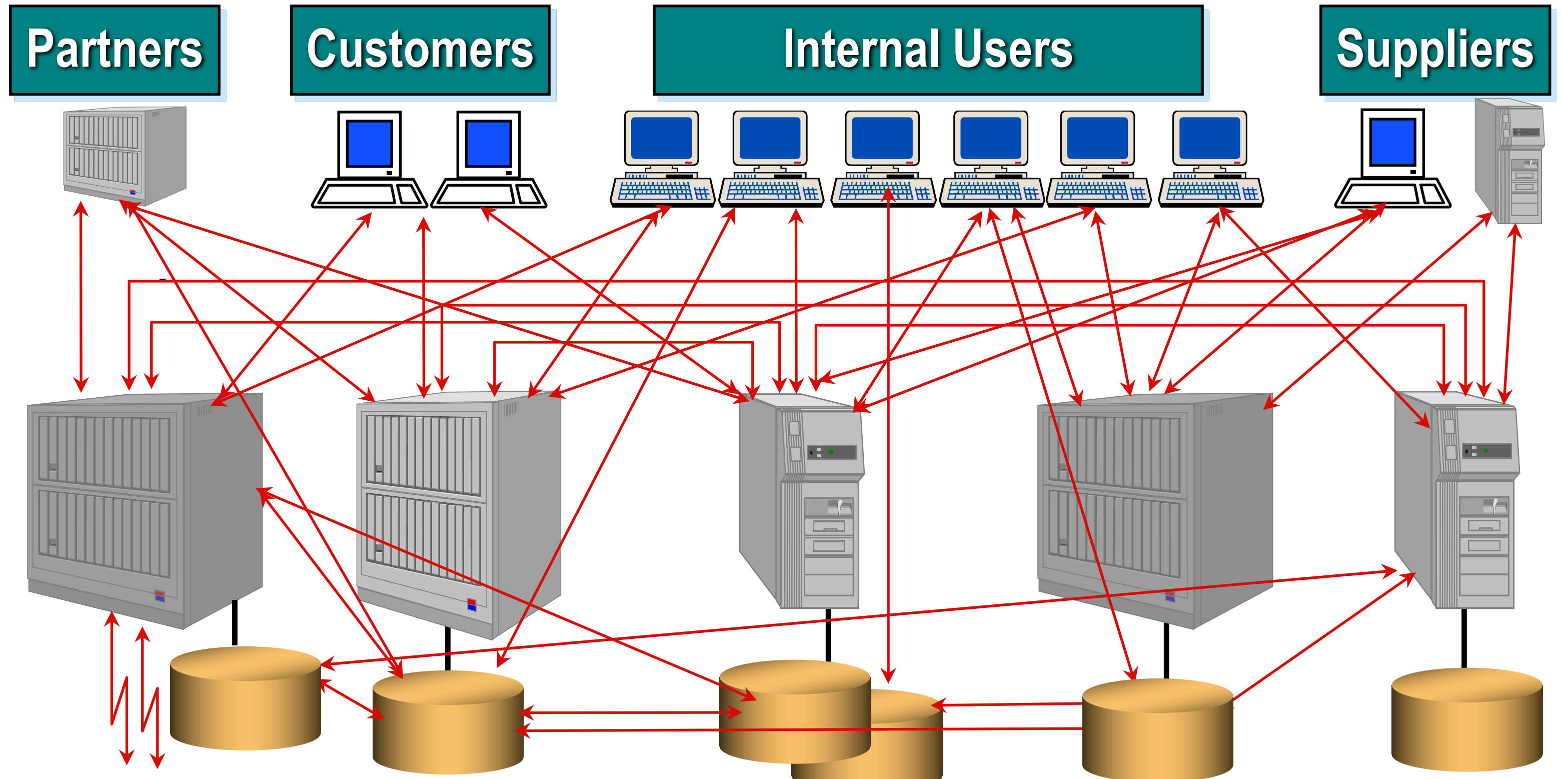
Enterprise Application Integration

REliable, INtelligent & Scalable Systems



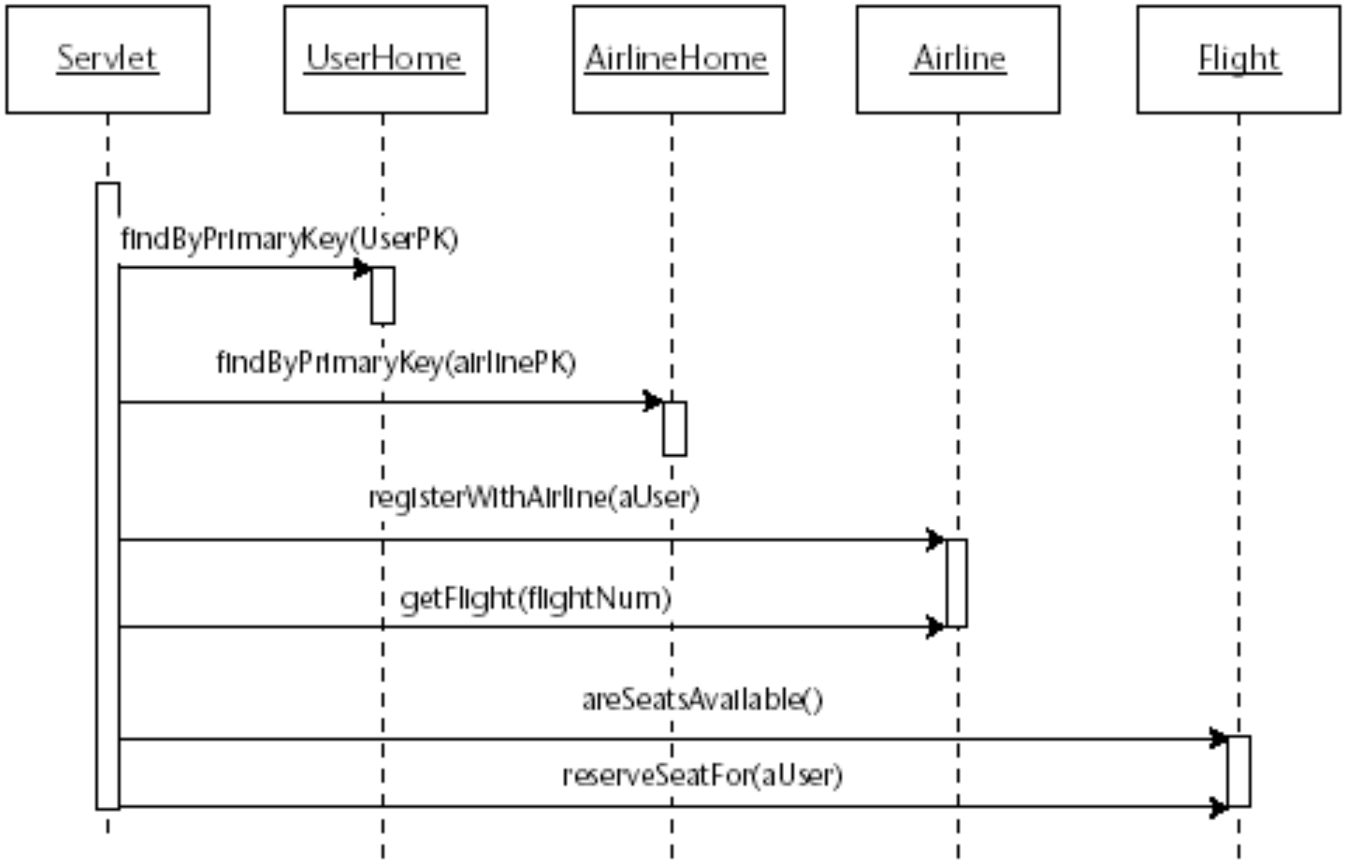
Nightmare of EAI

REliable, INtelligent & Scalable Systems



Quick response by asyn.

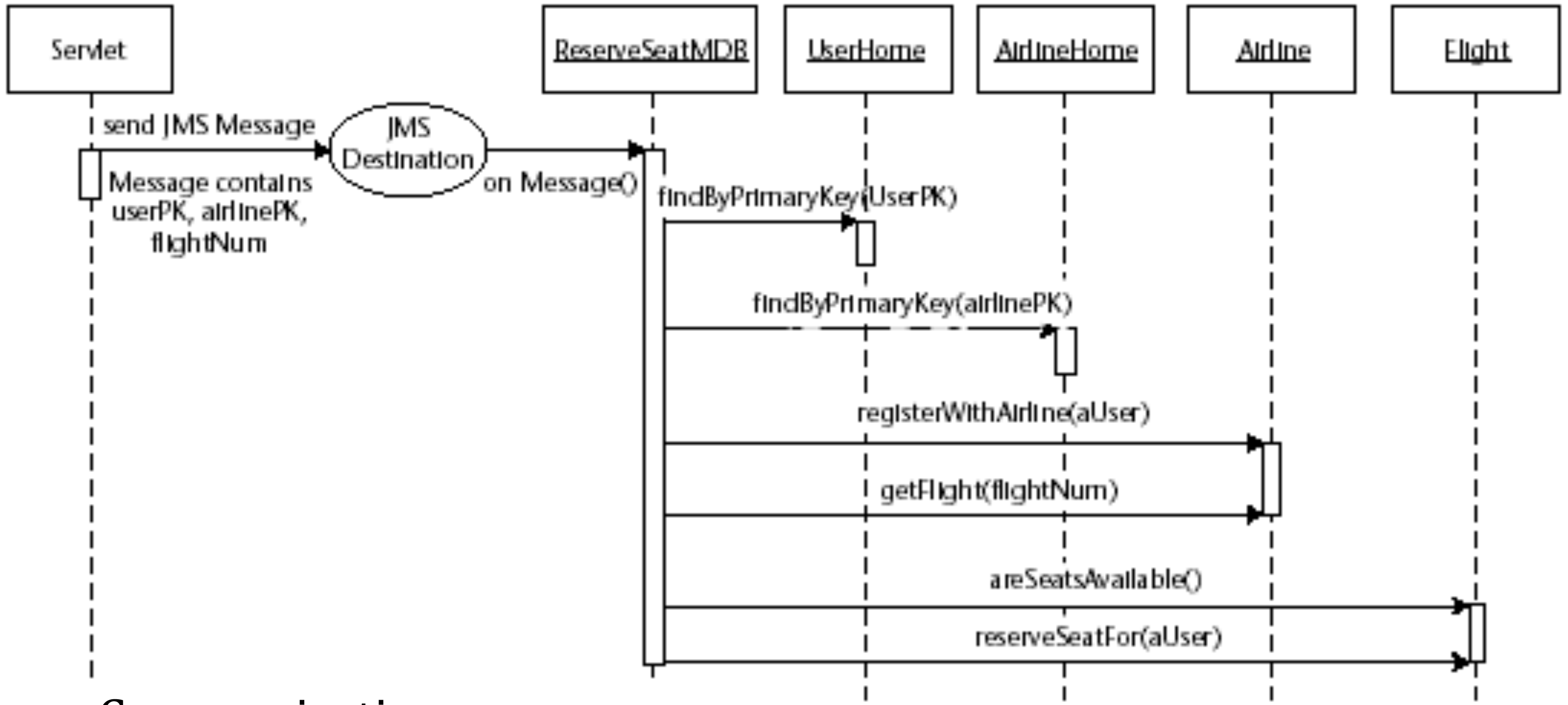
REliable, INtelligent & Scalable Systems



- Unacceptable response time
- No fault tolerance

Quick response by asyn.

REliable, INtelligent & Scalable Systems



- Asyn. Communication
- Elimination of single failure point

- Weak typed auguments
- No result value
- No exception to be thrown

What Is Messaging?

- Messaging is a method of communication between software components or applications.
  - A messaging system is a peer-to-peer facility:
  - A messaging client can send messages to, and receive messages from, any other client.
  - Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

- Messaging enables distributed communication that is loosely coupled.

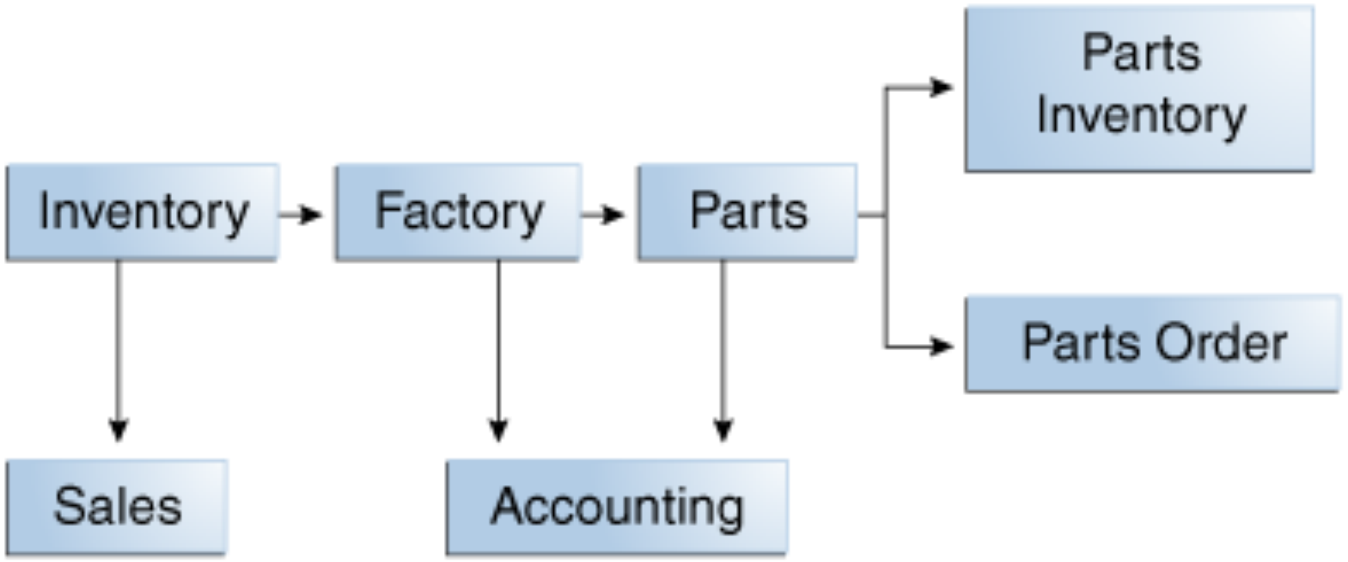
What Is the JMS API?

- The Java Message Service is a Java API that allows applications to create, send, receive, and read messages.

- JMS enables communication that is not only loosely coupled but also:
  - **Asynchronous**: A receiving client does not have to receive messages at the same time the sending client sends them. The sending client can send them and go on to other tasks; the receiving client can receive them much later.
  - **Reliable**: A messaging provider that implements the JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

When Can We Use Messaging?

REliable, INtelligent & Scalable Systems



JMS API Work with the Java EE Platform

- The JMS API in the Java EE platform has the following features.
  - Application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message. Application clients can in addition set a message listener that allows JMS messages to be delivered to it asynchronously by being notified when a message is available.
  - Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages in the EJB container. An application server typically pools message-driven beans to implement concurrent processing of messages.
  - Message send and receive operations can participate in Java Transaction API (JTA) transactions, which allow JMS operations and database accesses to take place within a single transaction.

Message

- Applications produce or consume messages

- The format of message is elastic, including three parts:
  - A header
  - Properties (optional)
  - A body (optional)

header

- Includes some pre-defined fields
  - JMSDestination (S)
  - JMSDeliveryMode (S)
  - JMSMessageID (S)
  - JMSTimestamp (S)
  - JMSCorrelationID (C)
  - JMSReplyTo (C)
  - JMSRedelivered (P)
  - JMSType (C)
  - JMSExpiration (S)
  - JMSPriority (S)
- Clients can not extend the fields

Properties

REliable, INtelligent & Scalable Systems

- Includes some pre-defined fields
  - JMSXUserID (S)
  - JMSXAppID (S)
  - JMSXDeliveryCount (S)
  - JMSXGroupID (C)
  - JMSXGroupSeq (C)
  - JMSXProducerTXID (S)
  - JMSXConsumerTXID (S)
  - JMSXRcvTimestamp (S)
  - JMSXState (P)

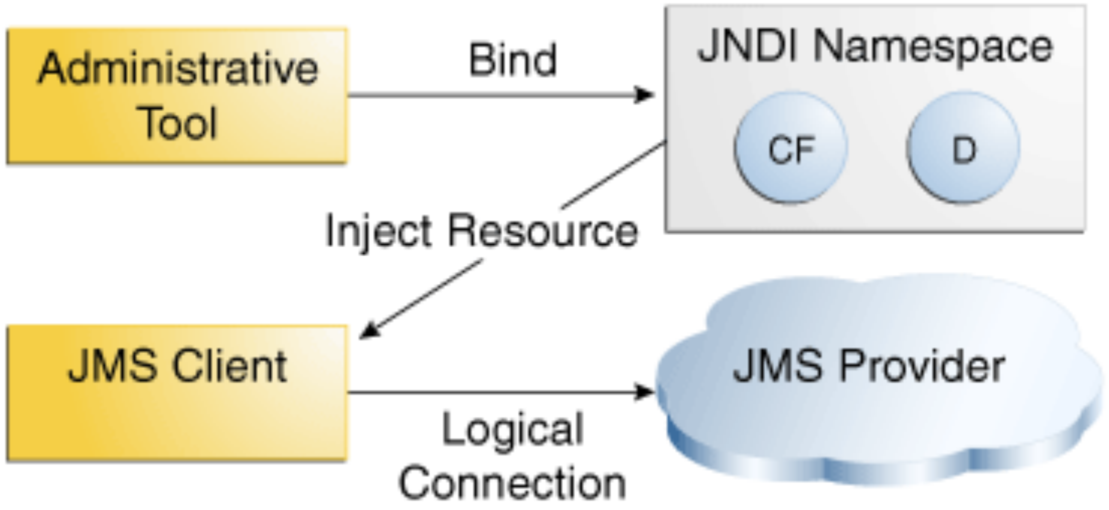
- Clients can extend the fields
  - Property name: follows the rule of naming selectors
  - Property value: boolean, byte, short, int, long, float, double, String

Body

REliable, INtelligent & Scalable Systems

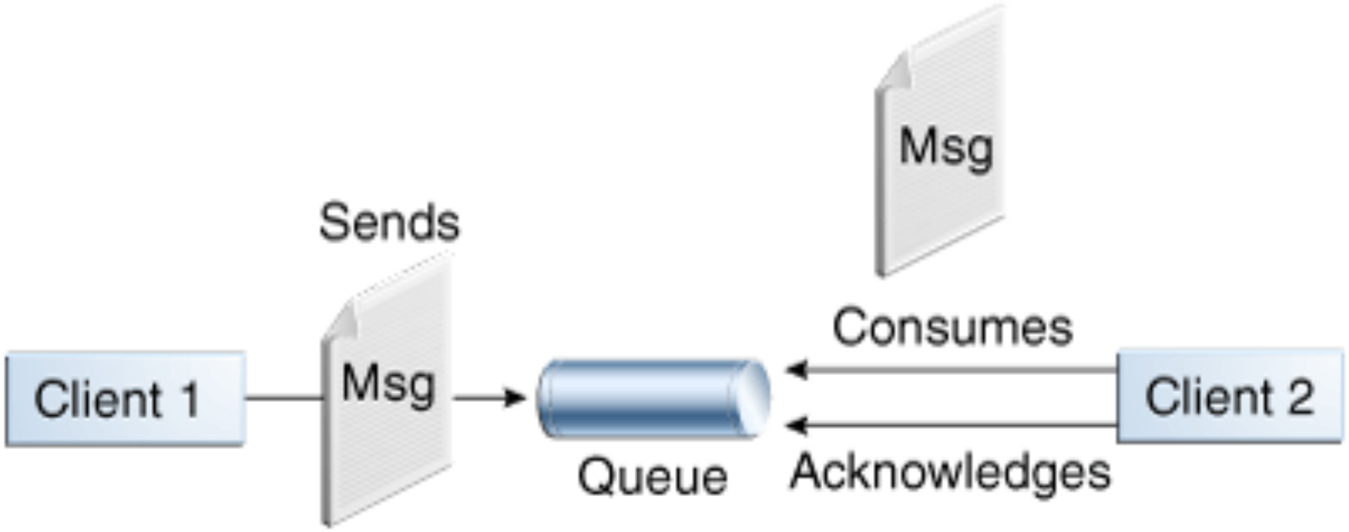
Message Type	Body Contains
TextMessage	A java.lang.String object (for example, the contents of an XML file).
MapMessage	A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

JMS API Architecture



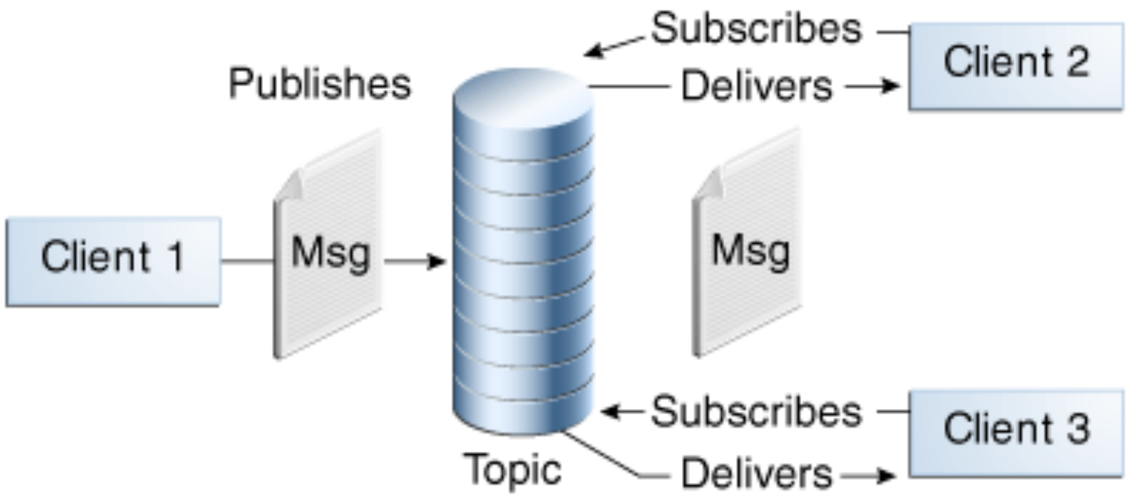
Messaging Styles

- A **point-to-point** (PTP) product or application is built on the concept of message **queues**, senders, and receivers.
  - Each message has only one consumer.
  - The receiver can fetch the message whether or not it was running when the client sent the message.

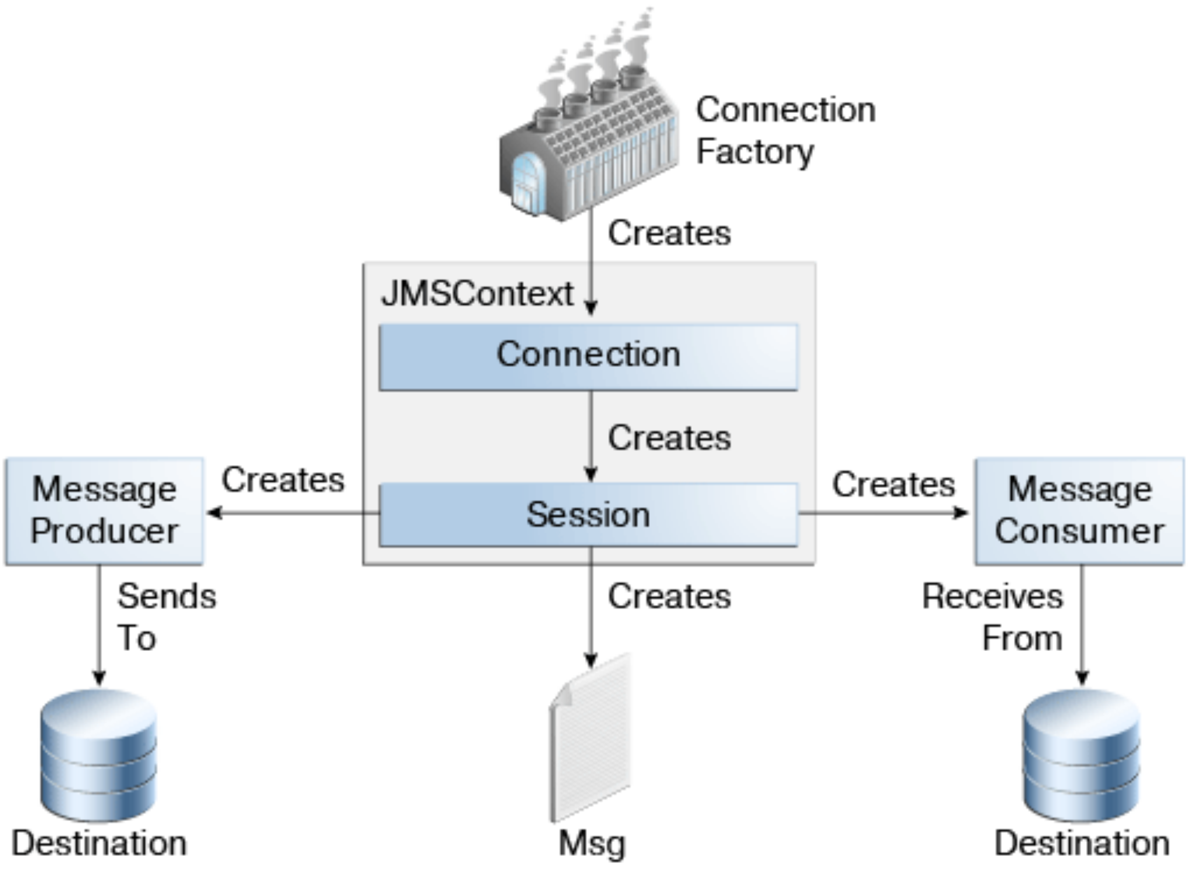


Messaging Styles

- In a **publish/subscribe** (pub/sub) product or application, clients address messages to a **topic**, which functions somewhat like a bulletin board. Publishers and subscribers can dynamically publish or subscribe to the topic.
  - Each message can have multiple consumers.
  - A client that subscribes to a topic can consume only messages sent *after* the client has created a subscription, and the consumer must continue to be active in order for it to consume messages.

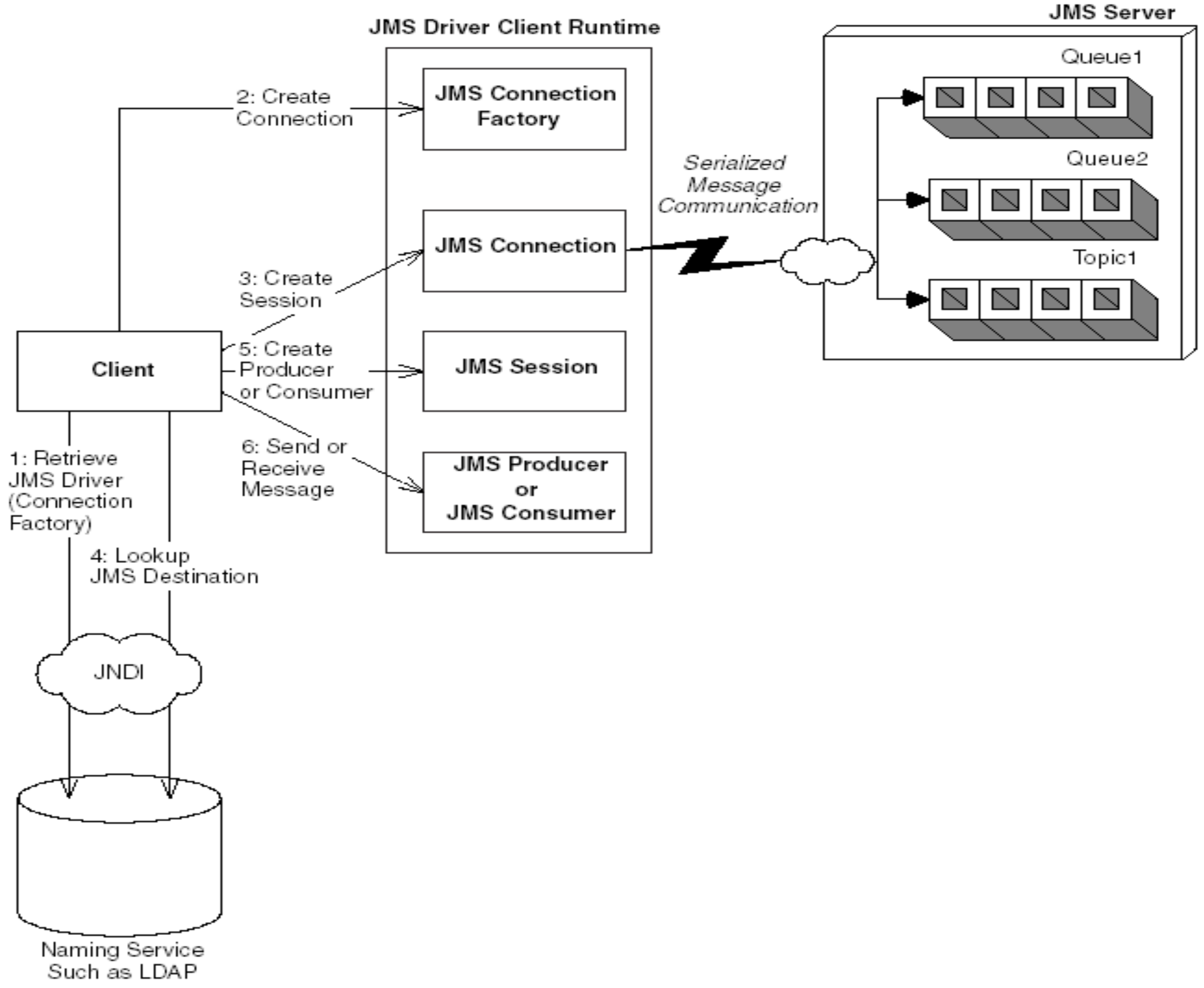


The JMS API Programming Model



JMS programming model

REliable, INtelligent & Scalable Systems



JMS Driver Client Runtime
JMS Server
Queue1
Queue2
Topic1
2: Create Connection
JMS Connection Factory
Serialized Message Communication
JMS Connection
3: Create Session
Client
5: Create Producer or Consumer
JMS Session
6: Send or Receive Message
JMS Producer or JMS Consumer
1: Retrieve JMS Driver (Connection Factory)
4: Lookup JMS Destination
JNDI
Naming Service Such as LDAP

JMS programming model

- Get a reference to `ConnectionFactory`
- A **connection factory** is the object a client uses to create a connection to a provider.

```
import javax.naming.*;
import javax.jms.*;
QueueConnectionFactory queueConnectionFactory;
Context messaging = new InitialContext();
queueConnectionFactory = (QueueConnectionFactory)
      messaging.lookup("QueueConnectionFactory");
```

- or

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
```

JMS programming model

- A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes.
  - In the PTP messaging style, destinations are called queues.
  - In the pub/sub messaging style, destinations are called topics.

```
Queue queue;
queue = (Queue)messaging.lookup(“theQueue");
Topic topic;
topic = (Topic)messaging.lookup(“theTopic");
```

- Or

```
@Resource(lookup = "jms/MyQueue")
private static Queue queue;
@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```

JMS programming model

REliable, INtelligent & Scalable Systems

- Get a Connection and a Session

- A **connection** encapsulates a virtual connection with a JMS provider.
- A **session** is a single-threaded context for producing and consuming messages.
  - You normally create a session (as well as a connection) by creating a JMSContext object

```
JMSContext context = connectionFactory.createContext();
```

JMS programming model

- Create a Producer or a Consumer

```
try (JMSContext context = connectionFactory.createContext();)
{
 JMSProducer producer = context.createProducer();
 ...
context.createProducer().send(dest, message);
```

- Or

```
try (JMSContext context = connectionFactory.createContext();)
{
 JMSConsumer consumer = context.createConsumer(dest);
 ...
Message m = consumer.receive();
Message m = consumer.receive(1000);
```

JMS programming model

REliable, INtelligent & Scalable Systems

- Create a message

```
TextMessage message = context.createTextMessage();
message.setText(msg_text);
// msg_text is a String
context.createProducer().send(message);


Message m = consumer.receive();
if (m instanceof TextMessage) {
 String message = m.getBody(String.class);
 System.out.println("Reading message: " + message);
} else {
 // Handle error or process another message type
}
```

JMS programming model

- JMS Message Listener
- A message listener is an object that acts as an asynchronous event handler for messages.

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);


Listener:
      void onMessage(Message inMessage)
```

JMS programming model

- Selection of messages
- Producer:

```
String data;
TextMessage message;
message = session.createTextMessage();
message.setText(data);
message.setStringProperty(“Selector”, “Technology”);
```

JMS programming model

- Selection of messages
- Consumer:

```
String selector;
selector = new String(“(Selector = ‘Technology’)”);

JMSConsumer consumer = context.createConsumer(dest, selector);
```

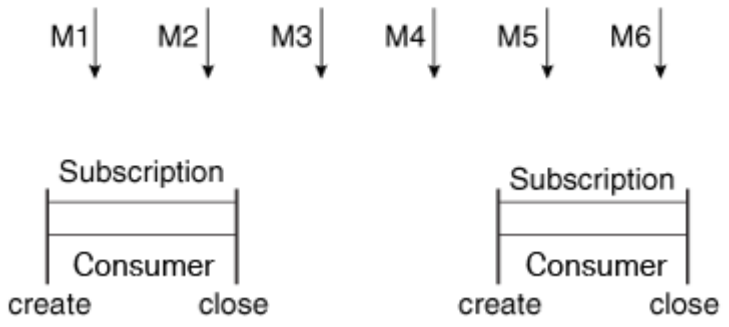
JMS programming model

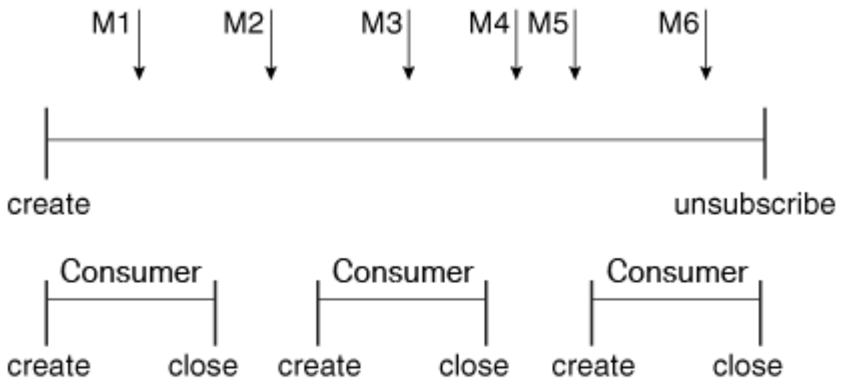
- Durable subscription

```
String subName = "MySub";
JMSConsumer consumer =
      context.createDurableConsumer(myTopic, subName);

consumer.close();
context.unsubscribe(subName);

JMSConsumer consumer =
   context.createSharedDurableConsumer(topic, "MakeItLast");
```

JMS programming model

REliable, INtelligent & Scalable Systems





JMS programming model

REliable, INtelligent & Scalable Systems

- JMS Browser
  - allows you to browse the messages in the queue and display the header values for each message.

```
QueueBrowser browser = context.createBrowser(queue);
```

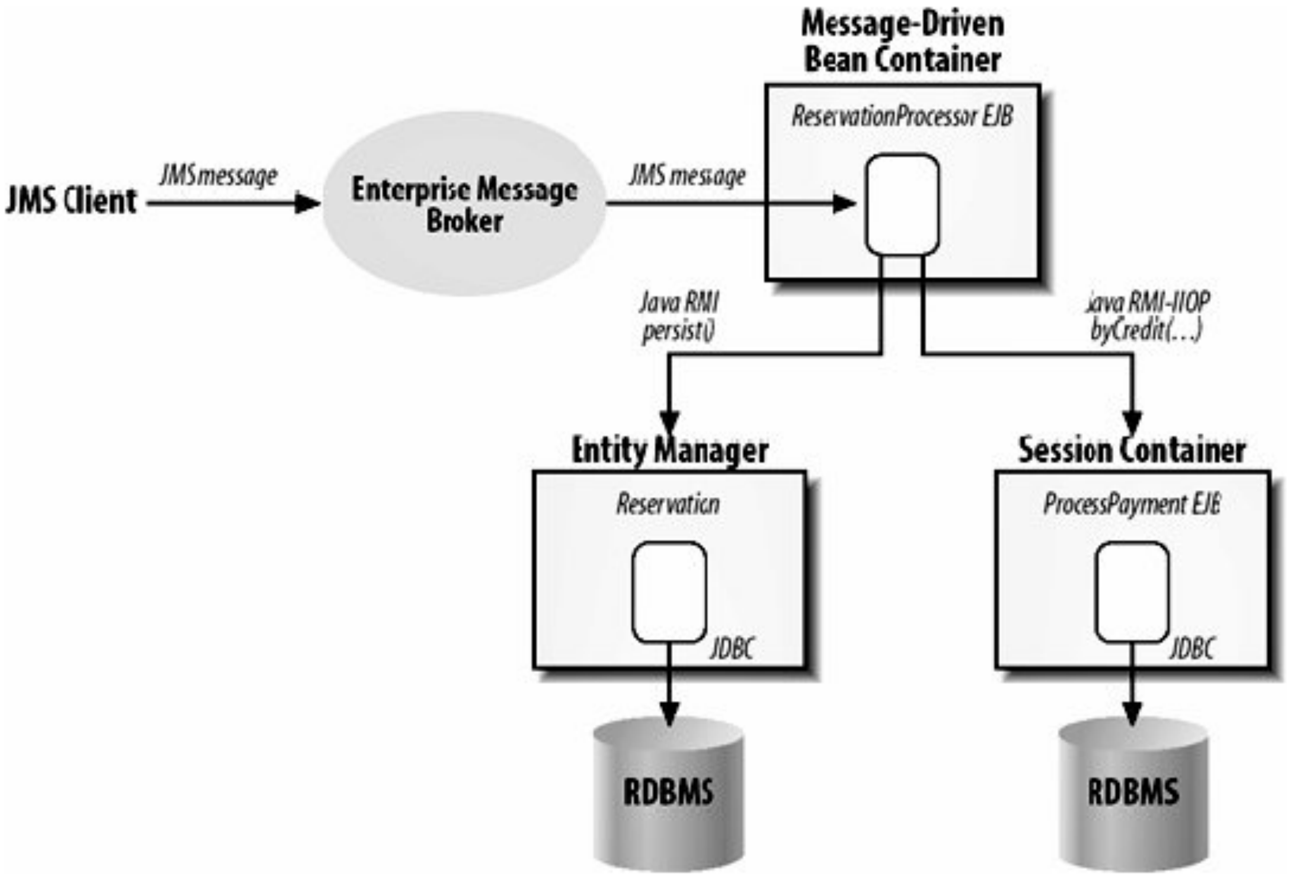
JMS programming model

- JMS Exception Handling
  - The root class for all checked exceptions in the JMS API is `JMSException`

Message driven bean

- A message-driven bean is a complete enterprise bean
  - it does not have a remote or local business interface.
  - These interfaces are absent because the message-driven bean responds only to asynchronous messages.

Message driven bean



Message driven bean

- A message-driven bean class has the following requirements:
  - It must be annotated with the @MessageDriven annotation if it does not use a deployment descriptor.
  - The class must be defined as public, but not as abstract or final.
  - It must contain a public constructor with no arguments.

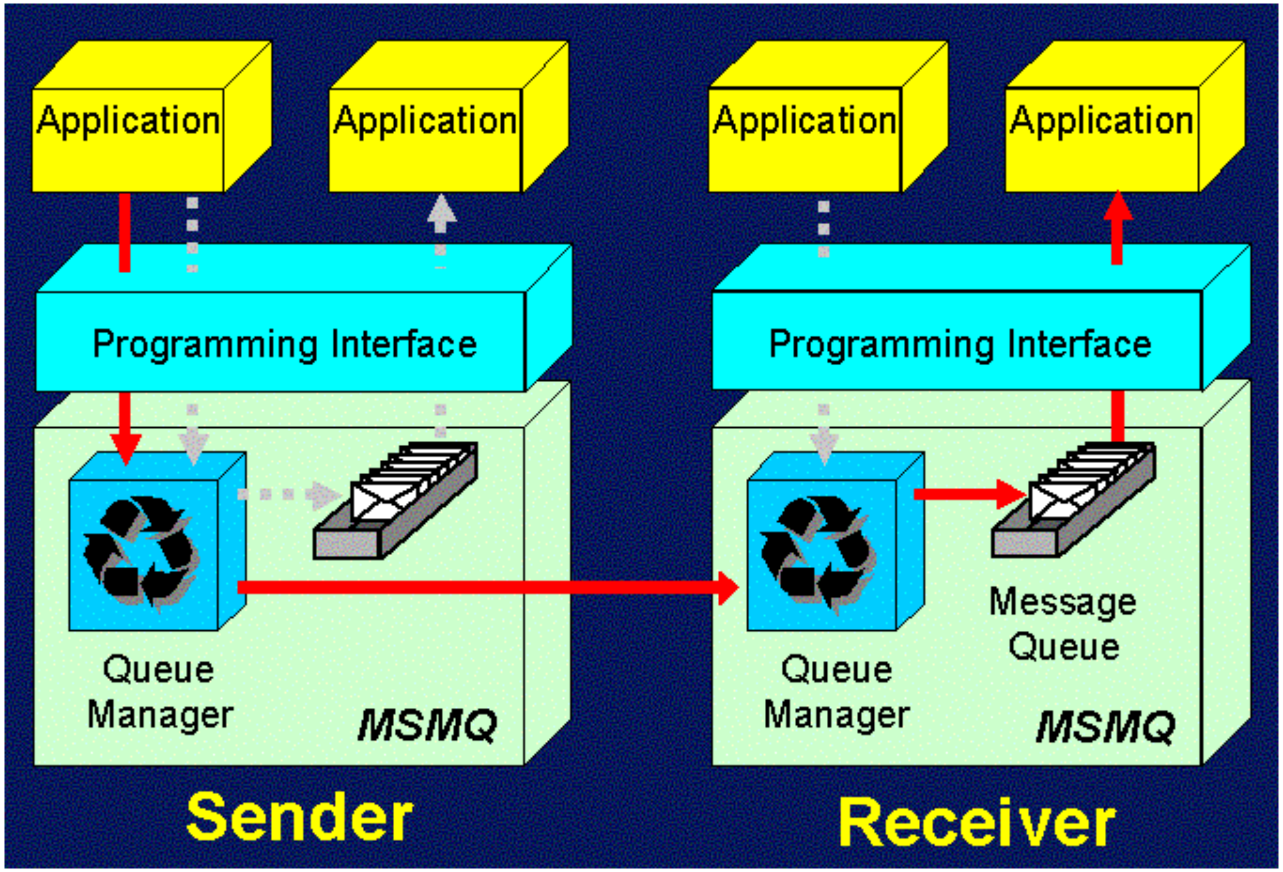
- It is recommended, but not required, that
  - a message-driven bean class implement the message listener interface for the message type it supports.
  - A bean that supports the JMS API implements the `javax.jms.MessageListener` interface, which means that it must provide an `onMessage` method with the following signature:
  - `void onMessage(Message inMessage)`

An example

REliable, INtelligent & Scalable Systems

```
@MessageDriven(activationConfig = {
 @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "jms/MyQueue"),
 @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue")
})
public class SimpleMessageBean implements MessageListener {
 @Resource private MessageDrivenContext mdc;
 static final Logger logger = Logger.getLogger("SimpleMessageBean");

 public SimpleMessageBean() { }

 @Override
 public void onMessage(Message inMessage) {
   try {
           if (inMessage instanceof TextMessage) {
              logger.log(Level.INFO,
              "MESSAGE BEAN: Message received: {0}",
              inMessage.getBody(String.class));
           } else {
              logger.log(Level.WARNING,
              "Message of wrong type: {0}",
              inMessage.getClass().getName());
           }
   } catch (JMSException e) {
           logger.log(Level.SEVERE, "SimpleMessageBean.onMessage: JMSException: {0}", e.toString());
           mdc.setRollbackOnly();
   }
  }
}
```

Messaging in .NET



Project

- To modify a function with messaging, such as order processing.
  - To implement this function as a MDB

References

REliable, INtelligent & Scalable Systems

- The Java EE 7 Tutorial
  - http://docs.oracle.com/javaee/7/tutorial/doc/javaeetutorial7.pdf

Thank You!