Java Message Service (JMS)

CS 595

Aysu Betin-Can

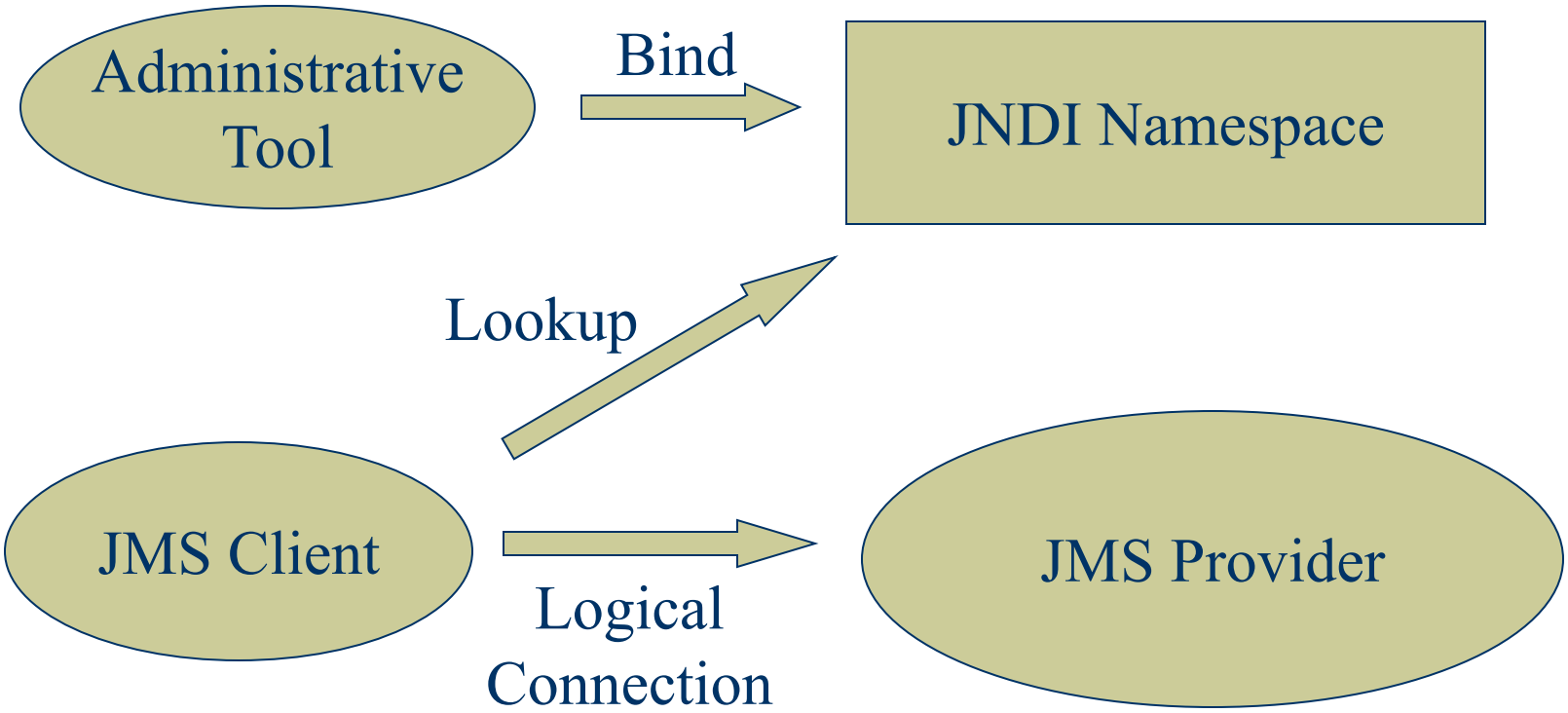
What is JMS?

- A **specification** that describes a common way for Java programs to create, send, receive and read distributed enterprise messages
- *loosely coupled* communication
- *Asynchronous* messaging
- *Reliable* delivery
  - A message is guaranteed to be delivered once and only once.
- Outside the specification
  - Security services
  - Management services

A JMS Application

- JMS Clients
  - Java programs that send/receive messages
- Messages
- Administered Objects
  - preconfigured JMS objects created by an admin for the use of clients
  - ConnectionFactory, Destination (queue or topic)
- JMS Provider
  - messaging system that implements JMS and administrative functionality

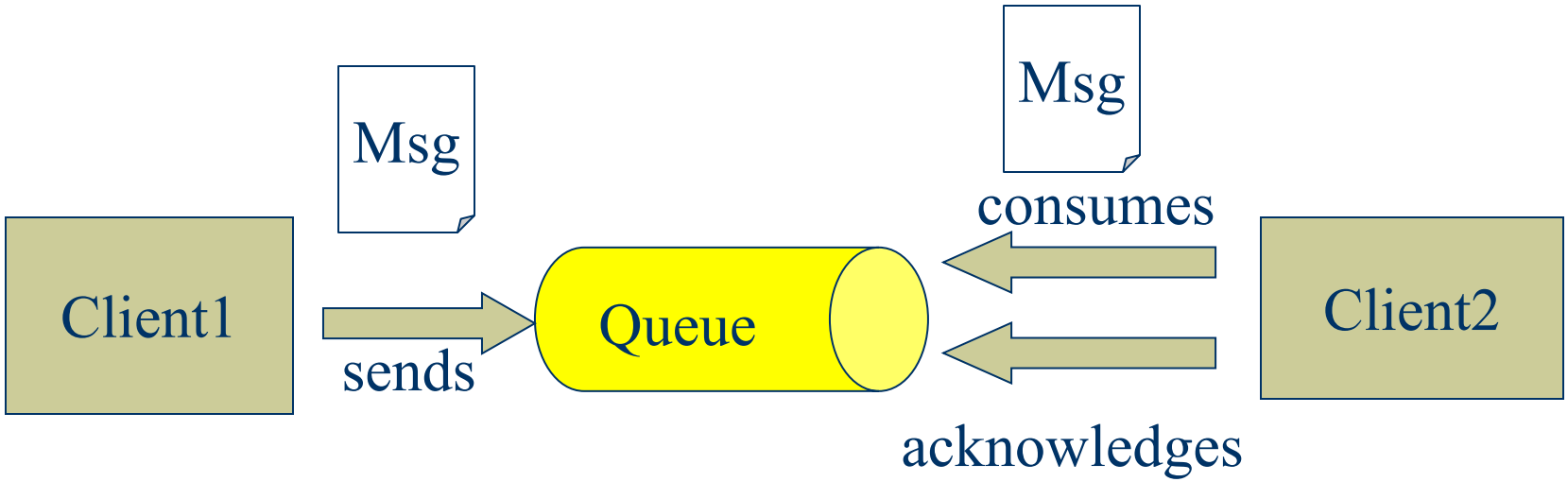
JMS Administration



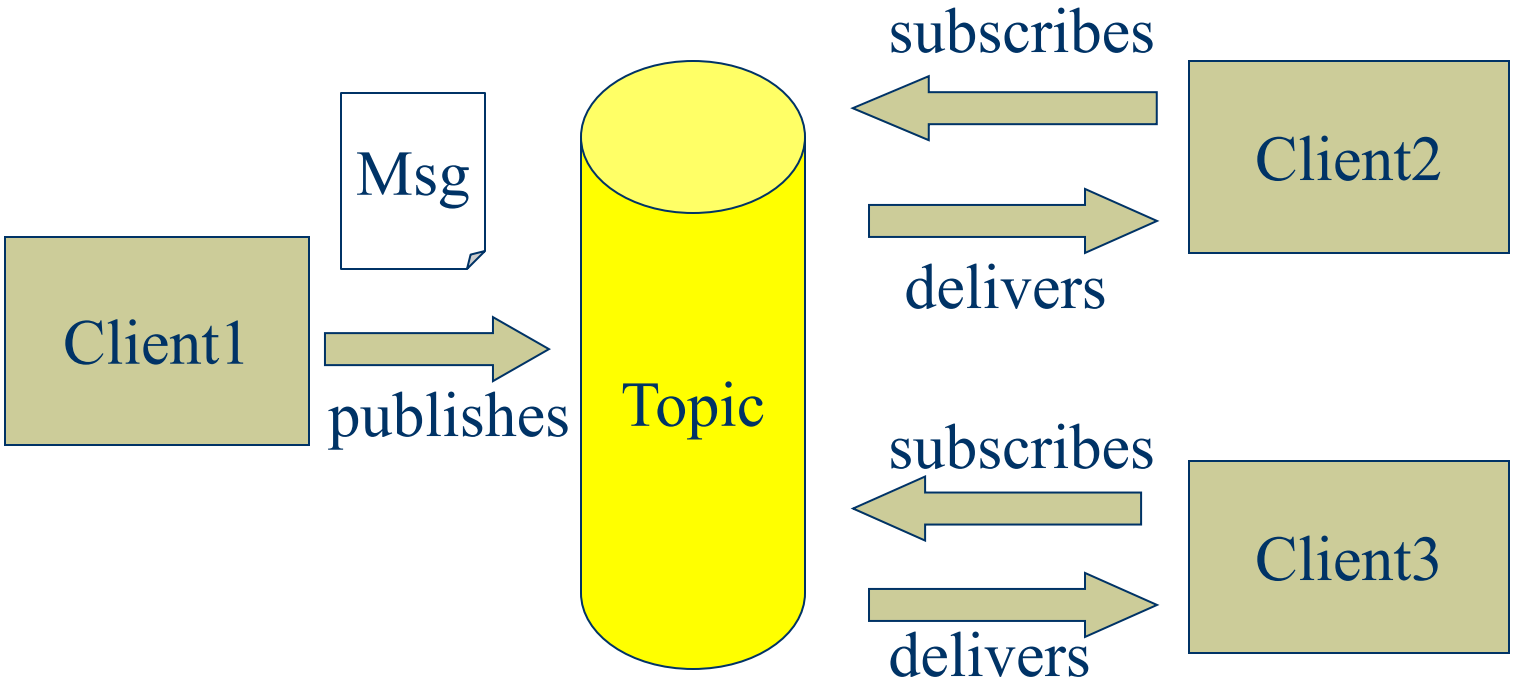
JMS Messaging Domains

- Point-to-Point (PTP)
  - built around the concept of message queues
  - each message has only one consumer
- Publish-Subscribe systems
  - uses a “topic” to send and receive messages
  - each message has multiple consumers

Point-to-Point Messaging



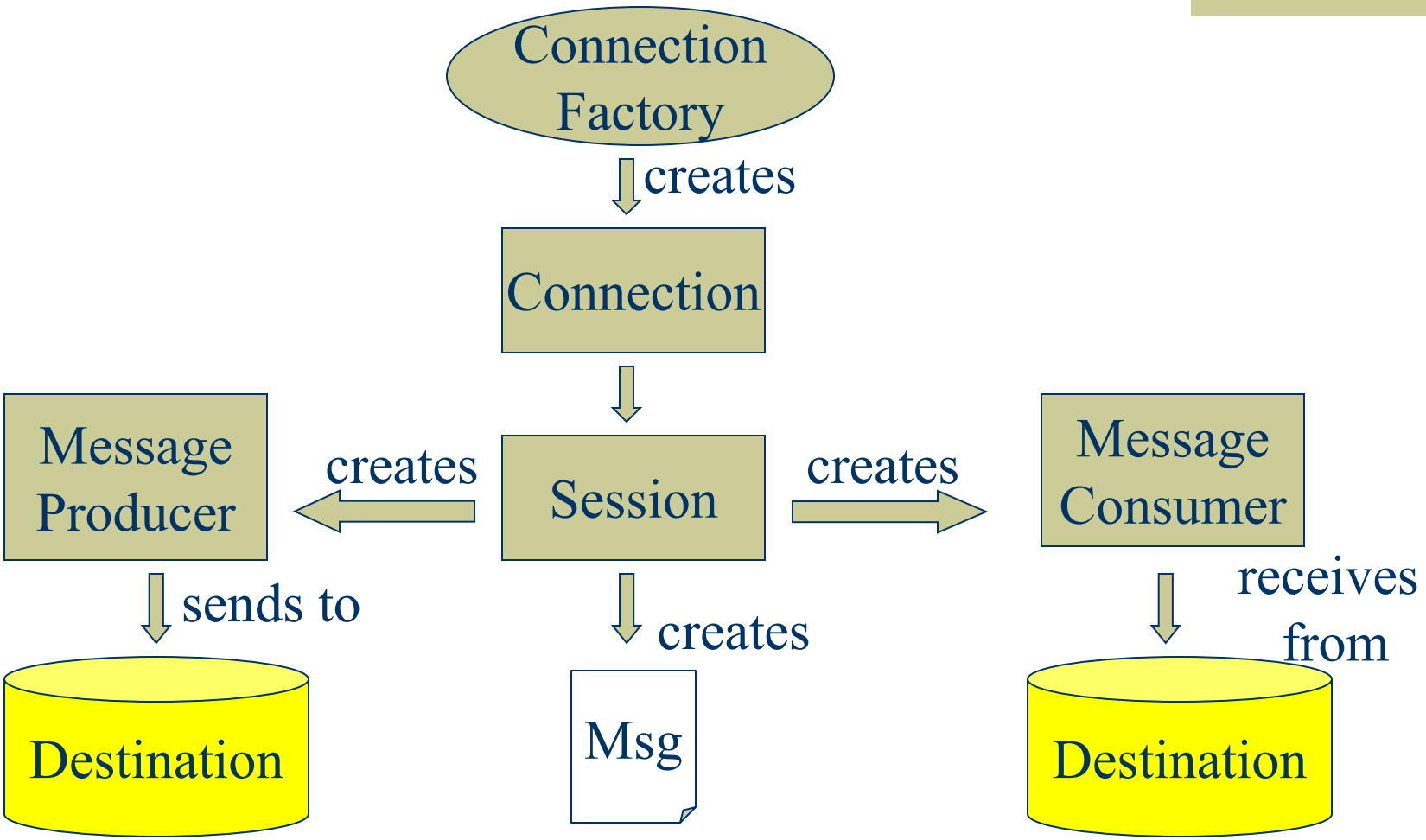
Publish/Subscribe Messaging



Message Consumptions

- Synchronously
  - A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method.
  - The receive method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit.
- Asynchronously
  - A client can register a *message listener* with a consumer.
  - Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage() method.

JMS API Programming Model



JMS Client Example

- Setting up a connection and creating a session

```
InitialContext jndiContext=new InitialContext();
//look up for the connection factory
ConnectionFactory cf=jndiContext.lookup(connectionfactoryname);
//create a connection
Connection connection=cf.createConnection();
//create a session
Session session=connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
//create a destination object
Destination dest1=(Queue) jndiContext.lookup("/jms/myQueue"); //for PointToPoint
Destination dest2=(Topic)jndiContext.lookup("/jms/myTopic"); //for publish-subscribe
```

Producer Sample

- Setup connection and create a session
- Creating producer

  MessageProducer producer=session.createProducer(dest1);
- Send a message

  Message m=session.createTextMessage();

  m.setText("just another message");

  producer.send(m);
- Closing the connection

  connection.close();

Consumer Sample (Synchronous)

- Setup connection and create a session
- Creating consumer

  MessageConsumer consumer=session.createConsumer(dest1);
- Start receiving messages

  connection.start();

  Message m=consumer.receive();

Consumer Sample (Asynchronous)

- Setup the connection, create a session
- Create consumer
- Registering the listener
  - MessageListener listener=new myListener();
  - consumer.setMessageListener(listener);
- myListener should have onMessage()

```
public void onMessage(Message msg){
    //read the massage and do computation
}
```

Listener Example

```
public void onMessage(Message message) {
   TextMessage msg = null;
   try {
      if (message instanceof TextMessage) {
         msg = (TextMessage) message;
         System.out.println("Reading message: " + msg.getText());
      } else {
         System.out.println("Message of wrong type: " +
            message.getClass().getName());
      }
   } catch (JMSException e) {
      System.out.println("JMSException in onMessage(): " + e.toString());
   } catch (Throwable t) {
      System.out.println("Exception in onMessage():" + t.getMessage());
   }
}
```

JMS Messages

- Message Header
  - used for identifying and routing messages
  - contains vendor-specified values, but could also contain application-specific data
  - typically name/value pairs
- Message Properties (optional)
- Message Body(optional)
  - contains the data
  - five different message body types in the JMS specification

JMS Message Types

Message Type	Contains	Some Methods
TextMessage	String	getText,setText
MapMessage	set of name/value pairs	setString,setDouble,setLong,getDouble,getString
BytesMessage	stream of uninterpreted bytes	writeBytes,readBytes
StreamMessage	stream of primitive values	writeString,writeDouble, writeLong,readString
ObjectMessage	serialize object	setObject,getObject

More JMS Features

- Durable subscription
  - by default a subscriber gets only messages published on a topic while a subscriber is alive
  - durable subscription retains messages until a they are received by a subscriber or expire
- Request/Reply
  - by creating temporary queues and topics
    - Session.createTemporaryQueue()
  - producer=session.createProducer(msg.getJMSReplyTo());
    reply= session.createTextMessage("reply");
    reply.setJMSCorrelationID(msg.getJMSMessageID);
    producer.send(reply);

More JMS Features

- Transacted sessions
  - session=connection.createSession(true,0)
  - combination of queue and topic operation in one transaction is allowed
  - void onMessage(Message m) {
    try { Message m2=processOrder(m);
    publisher.publish(m2); session.commit();
    } catch(Exception e) { session.rollback(); }

More JMS Features

- Persistent/nonpersistent delivery
  - producer.setDeliveryMethod(DeliveryMode.NON_PERSISTENT);
  - producer.send(mesg, DeliveryMode.NON_PERSISTENT ,3,1000);
- Message selectors
  - SQL-like syntax for accessing header:

```
subscriber  = session.createSubscriber(topic, “priority > 6 AND type = ‘alert’ ”);
```

  - Point to point: selector determines single recipient
  - Pub-sub: acts as filter

JMS and JNDI

- JMS does not define a standard address syntax by which clients communicate with each other
- Instead JMS utilizes Java Naming & Directory Interface(JNDI).
- Using JNDI provides the following advantages:
  - It hides provider-specific details from JMS clients.
  - It abstracts JMS administrative information into Java objects that are easily organized and administrated from a common management console.
  - Since there will be JNDI providers for all popular naming services, this means JMS providers can deliver one implementation of administered objects that will run everywhere. Thereby eliminating deployment and configuration issues.