Message-Oriented Middleware

Java Message Service

Professeur Olivier Gruber

Université Joseph Fourier

Projet SARDES (INRIA et IMAG-LSR)

Olivier.Gruber@inria.fr

Java Message Service

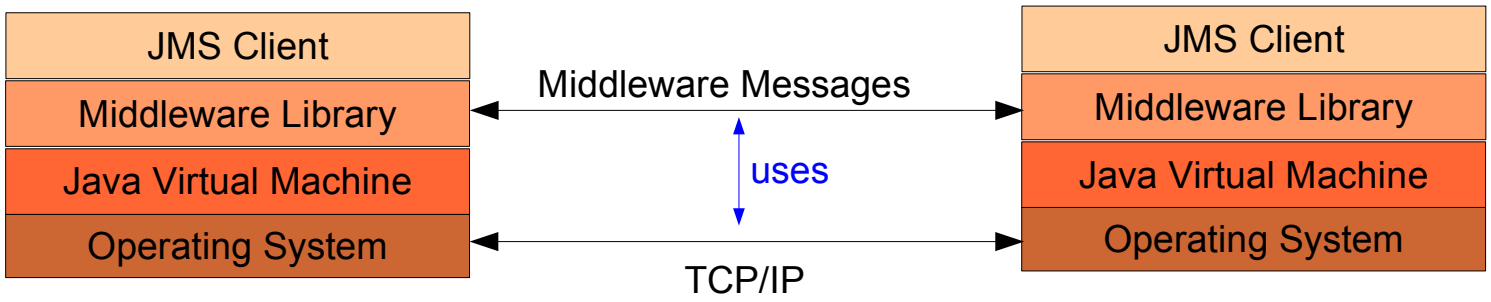
- Message-Oriented Middleware Specification
  - From SUN Corp.
  - APIs specified in Java, implementation is not specified
- Standardization Effort
  - Support two programming models
    - Point-to-point model
    - Publish-subscribe model
  - Could be used to
    - Integrate existing systems such as IBM WebShpere MQ or TIB/Rendez-Vous
    - Promote new implementations and designs (like JORAM)

JMS Providers

- Open Source
  - JORAM (ObjectWeb)
    - http://www.objectweb.org/joram
  - OpenJMS
    - http://openjms.exolb.org/
  - JDBMS
    - http://www.jdbms.org/
  - XmlBlaster
    - http://www.xmlBlaster.org/
- Products
  - BEA Systems Inc
  - GemStone
  - IBM
  - Oracle Corporation
  - Sun Microsystem
  - Allaire Corporation – JRun Server
  - Saga Software Inc.
  - SoftWired Inc.
  - SwiftMQ
  - Venue Software Corp.
  - Sunopsis
  - Orion
  - Nirvana
  - Fiorano/EMS Lite

JMS Architecture

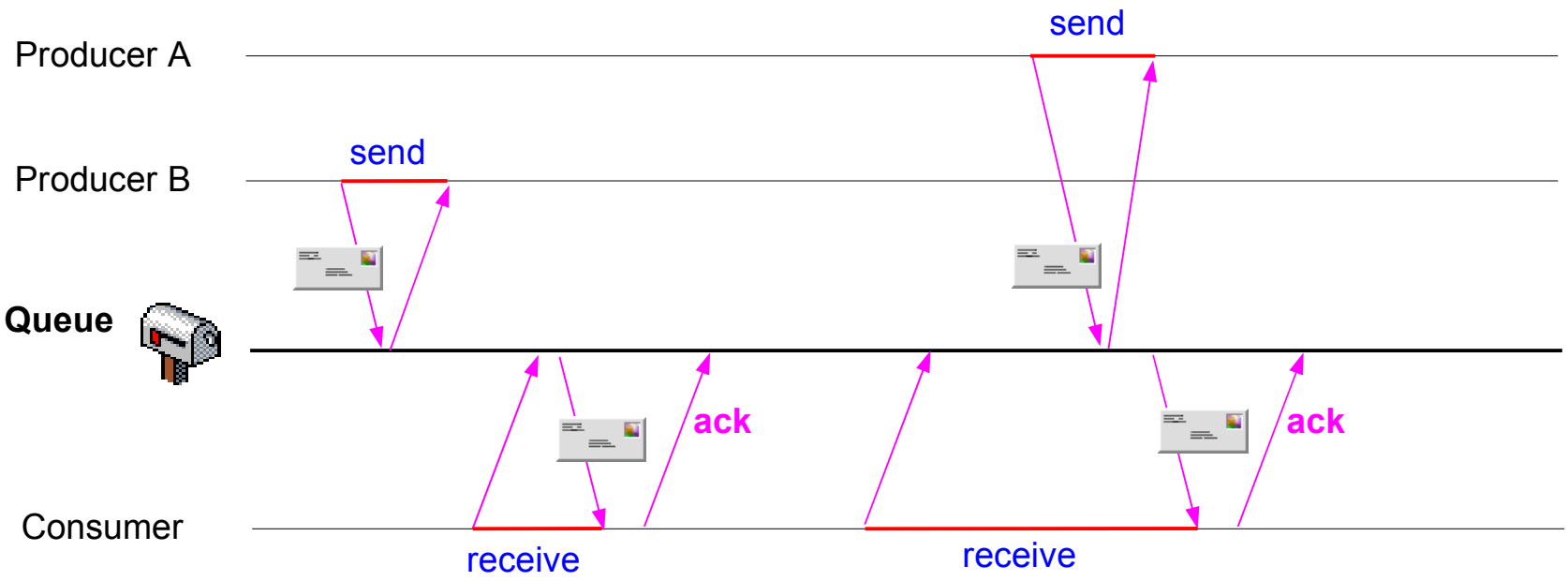
- JMS Clients
  - Java applications
    - Uses a Java library to
      - Connect to the middleware
      - To produces or consumes messages
    - Messages flow from producers to consumers
      - Can mix both point-to-point and publish-subscribe paradigms
    - Messages can be of different types
      - Such as structured text (XML), binary formats, serialized Java objects, etc.
      - Often *tunneled* over TCP/IP by the middleware



Olivier.Gruber@inria.fr

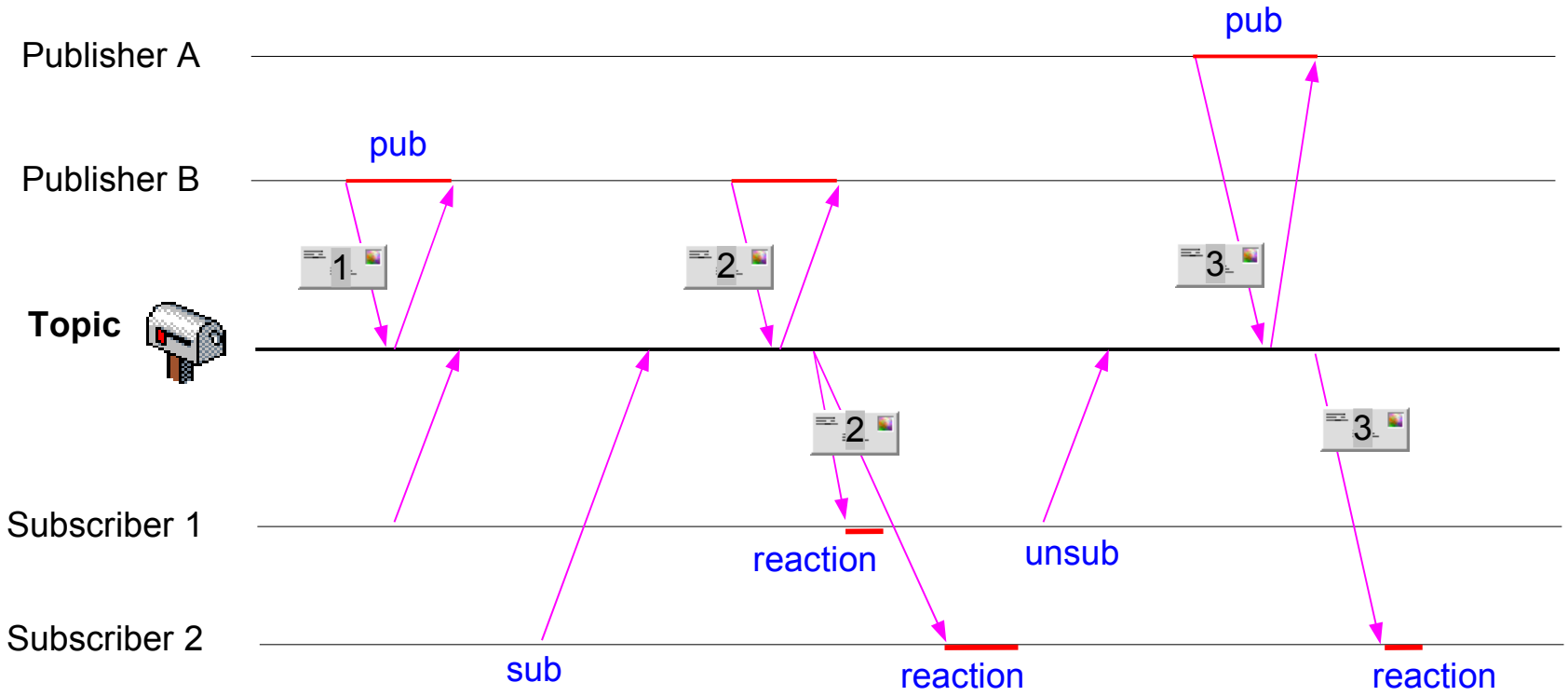
Messaging Domains

- Point-to-Point Messaging Domain
  - Each message has only one consumer
  - A sender and a receiver of a message have no timing dependencies
  - The receiver acknowledges the successful processing of a message



Messaging Domains

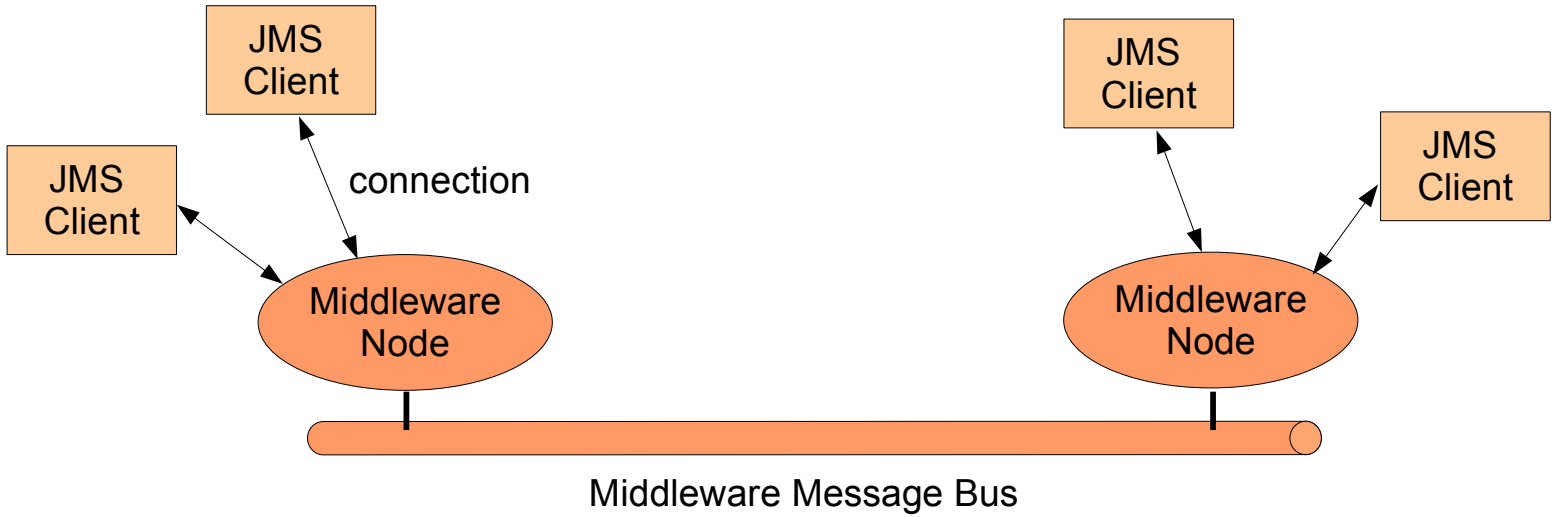
- Publish-Subscribe Messaging Domain
  - Each message may have multiple consumers
  - Publishers and subscribers have a timing dependency
    - Messages to a topic are **only** delivered to subscribed clients



Olivier.Gruber@inria.fr

JMS Architecture

- JMS Provider
  - A messaging system that implements the JMS interfaces
    - For e.g. J2EE 1.3 includes a JMS provider
  - Provides also administrative and control features
    - Create the typology of middleware nodes
    - Creates JMS objects such as message queues or topics
    - Offer the ability to set options and adjust the quality of service



Olivier.Gruber@inria.fr

JMS – Messages

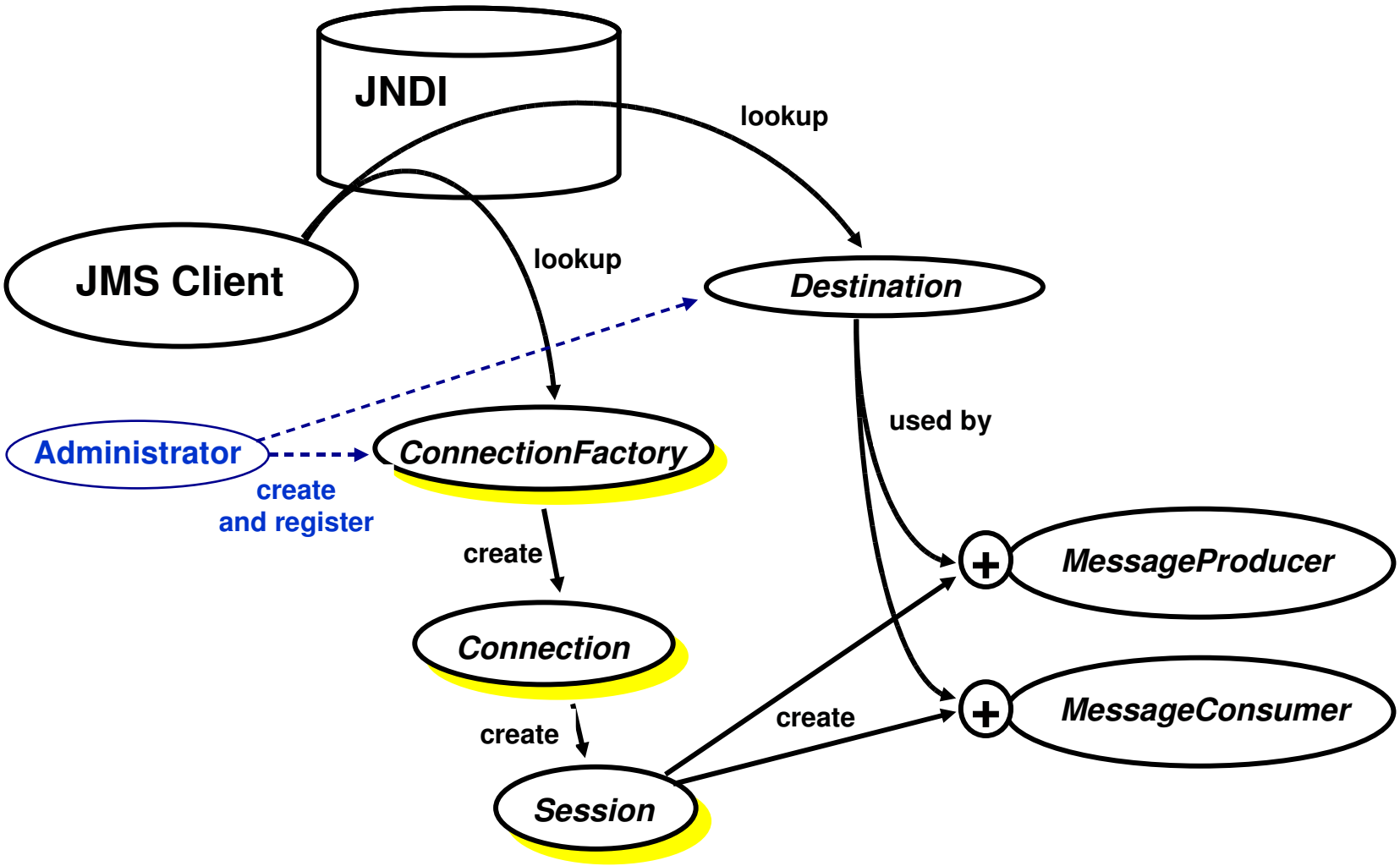
- Message Format
  - Header
  - Properties (optional)
  - Body (optional)
- Message Header
  - JMSMessageID
    - Unique identity of the message
  - JMSDestination
    - Destination of the message
  - JMSExpiration
    - Expiration date for the message
  - JMSTimestamp
    - Time at which the message was handed off to be sent
    - Optional

JMS – Message Types

- Message Types
  - Message types define the type of the message body
  - JMS defines standard types
    - Message: no body
    - TextMessage: a Java String
    - MapMessage: a set of name/value pairs (values must be Java primitive types)
    - BytesMessage: a stream of uninterpreted bytes
    - StreamMessage: a stream of Java primitive types
    - ObjectMessage: a serialized Java object
  - Domain-specific types are possible
    - Help interpret the bodies of the standard types in a domain-specific way
- Header Field: JMSType
  - Specifies the type of the message
    - Some JMS providers require that the type be specified
    - *For portability reasons*, developers should always set this field (even if not used)

Olivier.Gruber@inria.fr

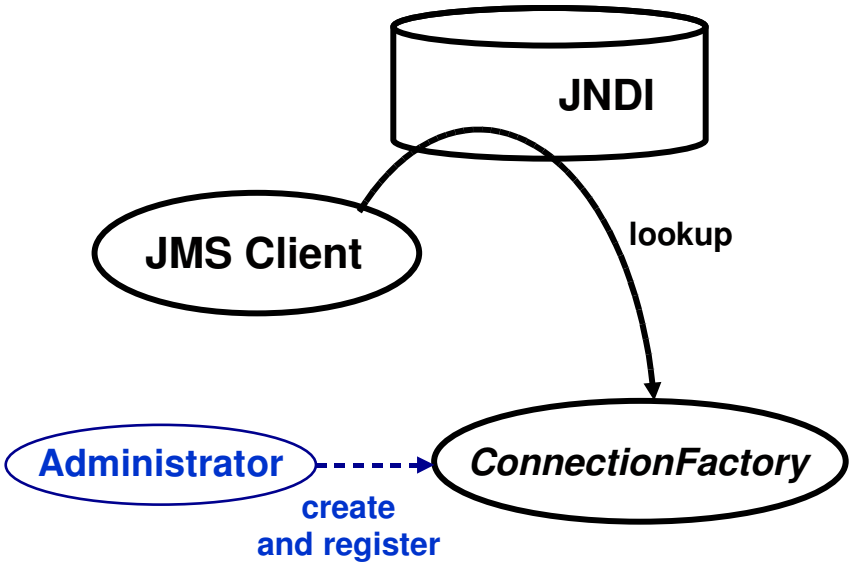
JMS Model Overview



Olivier.Gruber@inria.fr

JMS – Connections

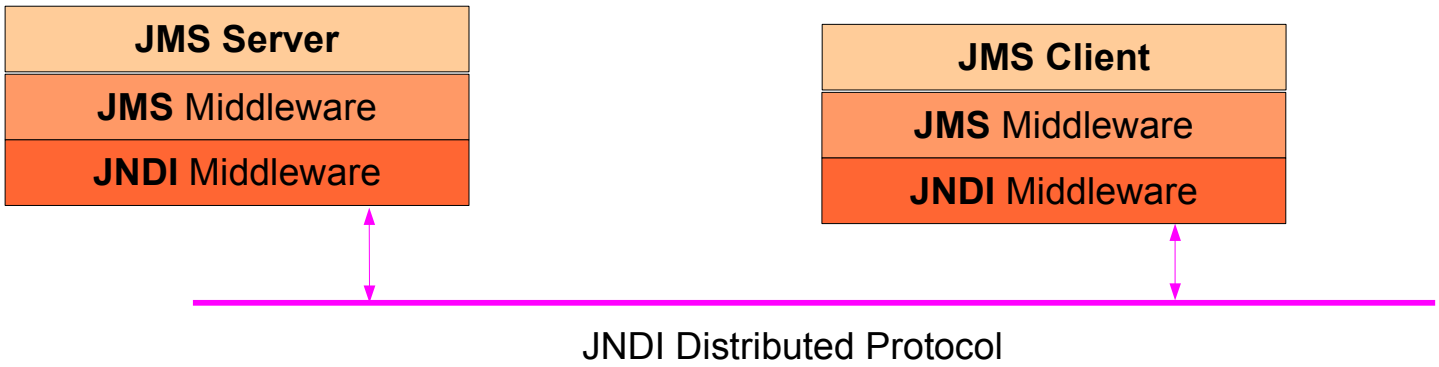
- Connection Factory
  - Encapsulates a connection configuration set by an administrator
    - For example, a factory to create TCP/IP connections
  - Each factory is either an instance of
    - QueueConnectionFactory or TopicConnectionFactory
    - Each factory is named in JNDI



JMS – Connections

- Connection Factory

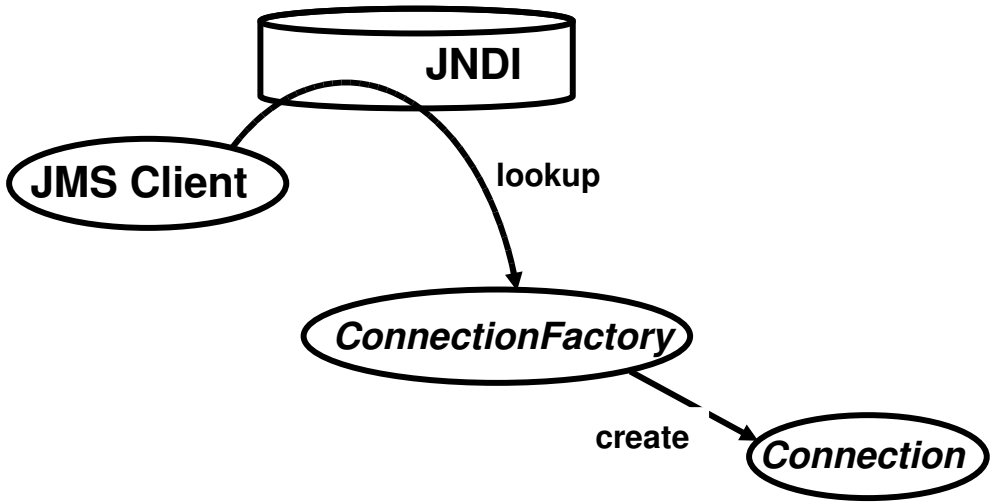
```
InitialContext ictxt;
QueueConnectionFactory qcf;
TopicConnectionFactory tcf;
try {
  ictxt = new InitialContext();

  qcf = (QueueConnectionFactory) ictxt.lookup("QueueConnectionFactory");

  tcf = (TopicConnectionFactory) ictxt.lookup("TopicConnectionFactory");

} catch (NamingException ex) {
  // handler...
}
```



Olivier.Gruber@inria.fr

JMS – Connections

- Connection Objects
  - Represents a connection between a JMS client and a JMS provider
    - Created with user credentials if authentication is required
    - Indirect factory (via sessions) for message consumers and producers
  - Lifecycle
    - Users may start, pause and stop a connection
      - It starts, pauses or stops message delivery to consumers
      - No effects on message producers (a stopped connection can produce messages)



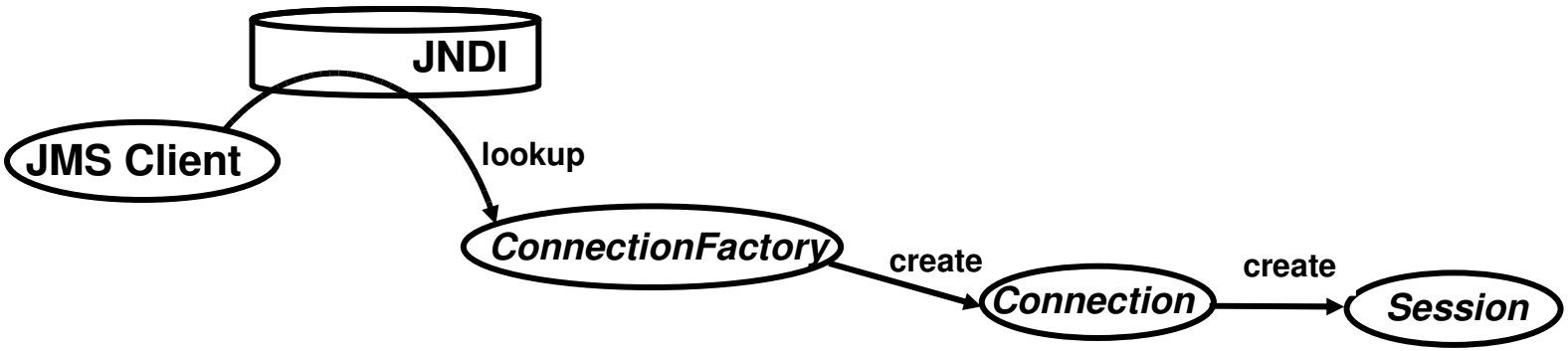
JMS – Connections

- Connection Objects

```
InitialContext ictxt;
QueueConnectionFactory qcf;
TopicConnectionFactory tcf;
QueueConnection queueConnection;
TopicConnection topicConnection;

try {
  ictxt = new InitialContext();

  qcf = (QueueConnectionFactory) ictxt.lookup("QueueConnectionFactory");
  queueConnection = qcf.createQueueConnection() ;

  tcf = (TopicConnectionFactory) ictxt.lookup("TopicConnectionFactory");
  topicConnection = tcf.createTopicConnection() ;

} catch (NamingException ex) {
  // handler...
} catch (JMSException ex) {
  // handler...
}
```

JMS - Sessions

- Session Objects
  - A session is a factory
    - For message producers and consumers
      - Instances of the *MessageConsumer* and *MessageProducer* class
      - Either for queues or topics
    - For tempory queues and topics
      - Although their scope is the connection, not the session
  - Each session object is a single-threaded context
    - It is used to produce and consume messages
    - Defines a serial order for consumed and produced messages
    - Provides a transactional context for sending and receiving messages



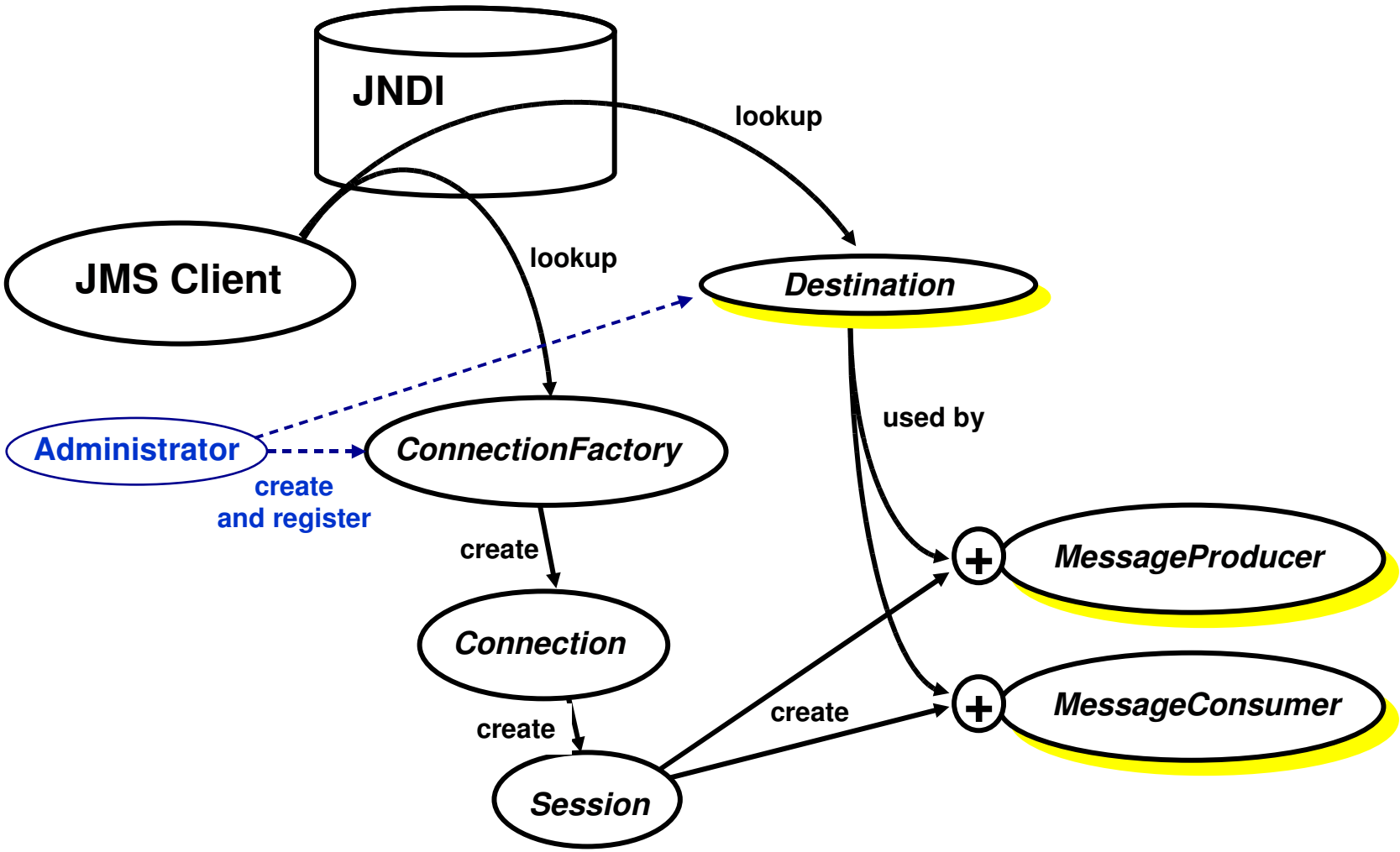
Olivier.Gruber@inria.fr

JMS - Sessions

- Session Objects

```
QueueSession queueSession;
TopicSession topicSession;

try {
  ictxt = new InitialContext();

  qcf = (QueueConnectionFactory) ictxt.lookup("QueueConnectionFactory");
  queueConnection = qcf.createQueueConnection() ;

  queueSession = queueConnection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);


  tcf = (TopicConnectionFactory) ictxt.lookup("TopicConnectionFactory");
  topicConnection = tcf.createTopicConnection() ;

  topicSession = topicConnection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);

} catch (NamingException ex) {
  // handler...
} catch (JMSException ex) {
  // handler...
}
```

JMS Model Overview (again...)



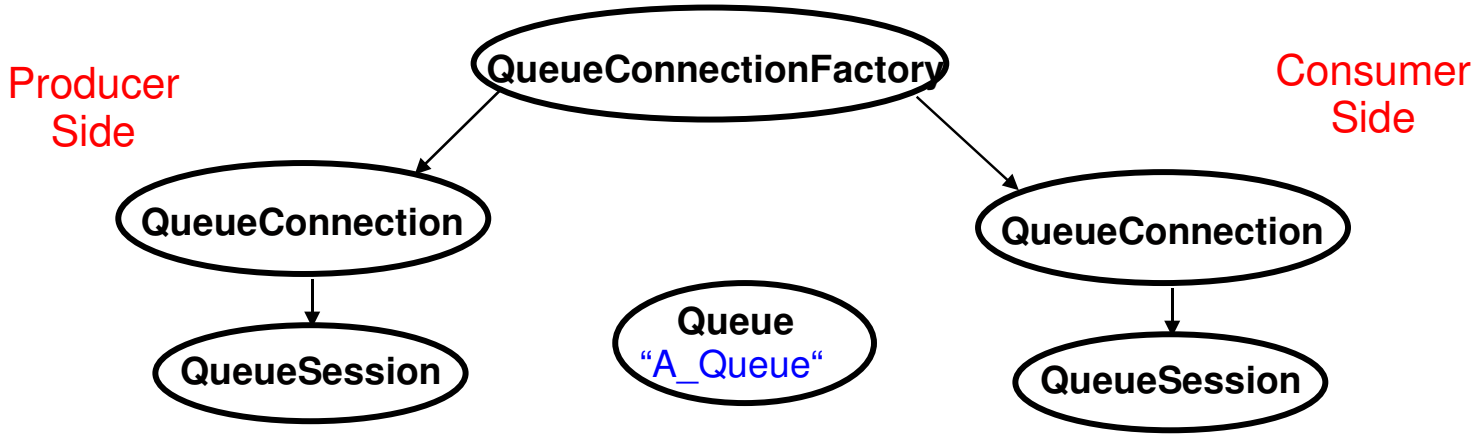
Olivier.Gruber@inria.fr

JMS – Destinations

- Destination Objects
  - Usually created and named administratively in JNDI
    - Can be created programmatically (through *Session* objects)
  - The target of produced messages or the source of consumed messages
    - Either a queue or a topic
- Producer Objects
  - Created by Session objects
  - Used to send messages to a destination
- Consumer Objects
  - Also created by session objects
  - Used to receive messages from a destination
    - Either push or pull model

Olivier.Gruber@inria.fr

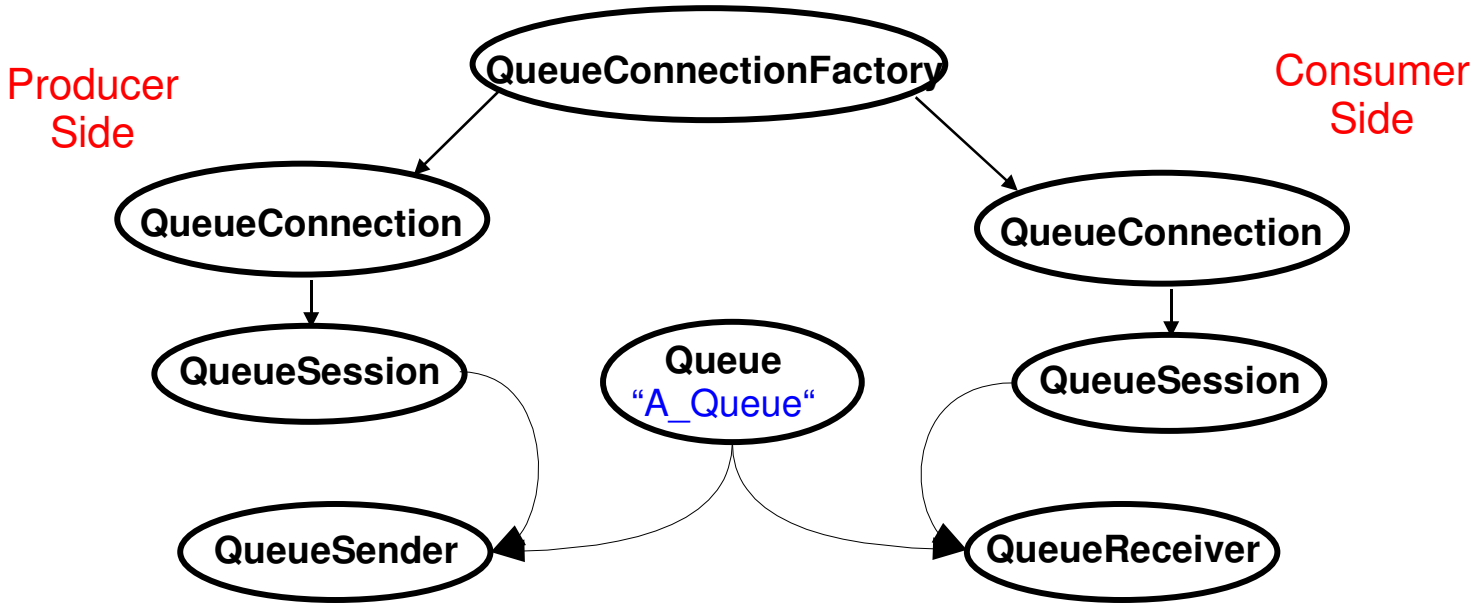
JMS – Queue Destination




```
InitialContext ictxt;
QueueConnectionFactory qcf;
QueueConnection queueConnection;
QueueSession queueSession;
try {
  ictxt = new InitialContext();

  qcf = (QueueConnectionFactory) ictxt.lookup("QueueConnectionFactory");
  queueConnection = qcf.createQueueConnection() ;
  queueSession = queueConnection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

} catch (NamingException ex) {
  // handler...
} catch (JMSException ex) {
  // handler...
}
```

Olivier.Gruber@inria.fr

JMS – Queue Destination

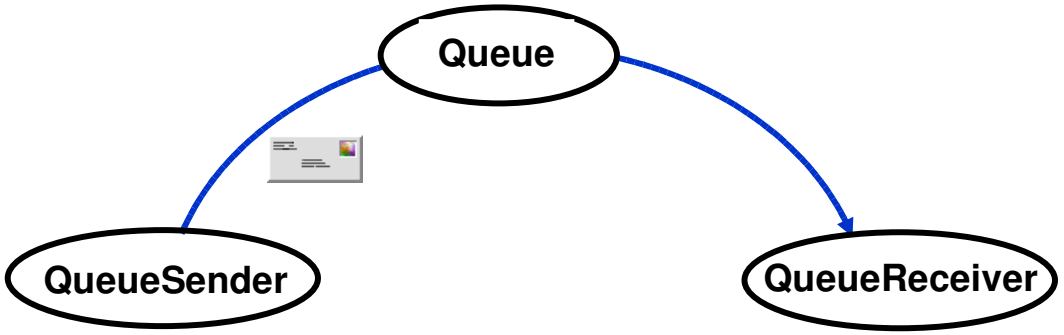



```
Queue queue;
queue = (Queue)ictxt.lookup("A_Queue") ;

QueueSender sender;
sender = queueSession.createSender(queue);
```

```
Queue queue;
queue = (Queue)ictxt.lookup("A_Queue") ;

MessageConsumer consumer;
consumer = queueSession.createConsumer(queue);

QueueReceiver receiver;
receiver=(QueueReceiver)consumer;
```

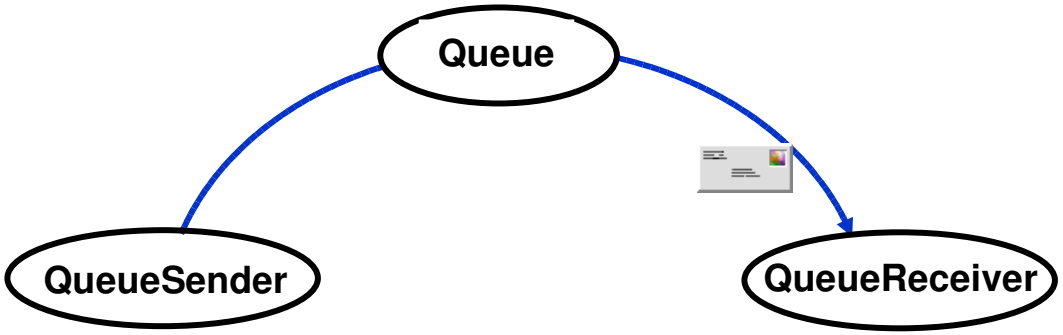
Olivier.Gruber@inria.fr

JMS – Queue Example (pull)




```
QueueSender sender;
sender = queueSession.createSender(queue);

byte bytes[] = new byte[64];
BytesMessage bytesMessage;
bytesMessage = queueSession.createBytesMessage();
bytesMessage.writeBytes(bytes);
sender.send(bytesMessage);

TextMessage textMessage;
textMessage = queueSession.createTextMessage();
textMessage.setText("Hello World");
sender.send(textMessage);
```

JMS – Queue Example (pull)




```
QueueReceiver receiver;
receiver=(QueueReceiver)consumer;

Message message = consumer.receive();
if (message instanceof TextMessage) {
  TextMessage textMessage;
  textMessage = (TextMessage) message;
  System.out.println(textMessage.getText());

} else if (message instanceof BytesMessage) {
  BytesMessage bytesMessage;
  bytesMessage = (BytesMessage)message;
  System.out.println(bytesMessage.readUTF());
}
```

Olivier.Gruber@inria.fr

JMS – Push Consumer

```
public class PushConsumer {

  public static void main(String[] args) throws Exception {

    InitialContext ictx;
    ictx = new InitialContext();
    Destination queue = (Destination) ictx.lookup("queue");
    ConnectionFactory cf = (ConnectionFactory) ictx.lookup("qcf");
    ictx.close();

    Connection cnx = cf.createConnection("user", "password");
    Session sess = cnx.createSession(false, Session.AUTO_ACKNOWLEDGE);
    MessageConsumer recv = sess.createConsumer(queue);

    recv.setMessageListener(new MessageListener() {
      public void onMessage(Message msg) {
        try {
          if (msg instanceof TextMessage) {
            System.out.println(((TextMessage) msg).getText());
          } else if (msg instanceof ObjectMessage) {
            System.out.println(((ObjectMessage) msg).getObject());
          }
        } catch (JMSException jE) {
          System.err.println("Exception in listener: " + jE);
        }
      }
    });
    cnx.start();
  }
}
```

Olivier.Gruber@inria.fr

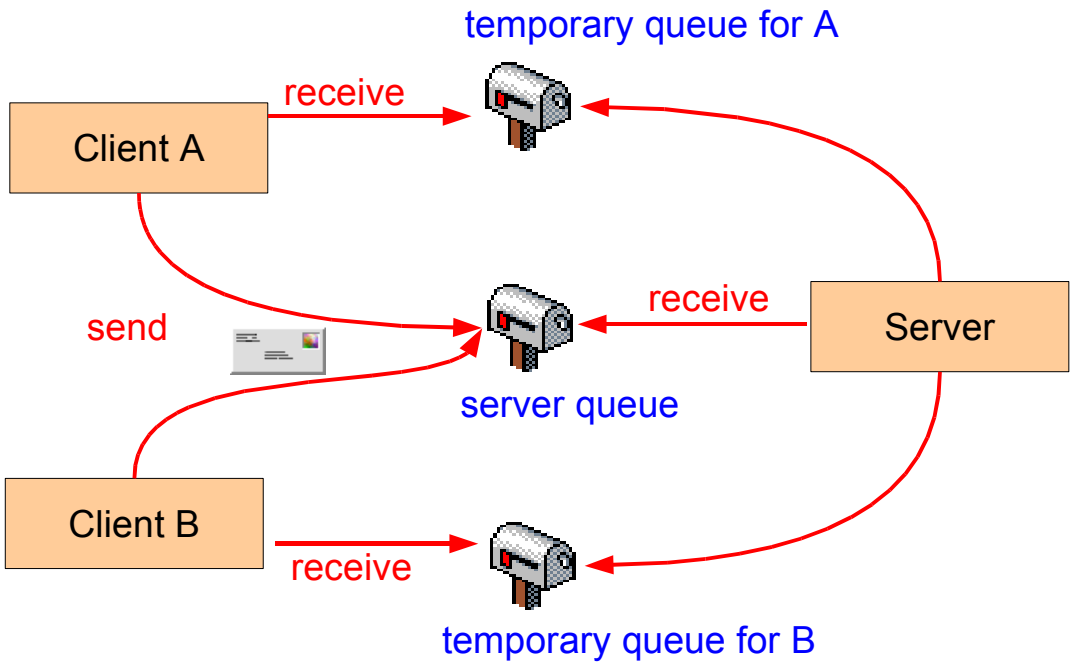
JMS – Queue Filtering

- Message Producers
  - Describe the messages they produce
  - Use the message header fields
- Message Consumers
  - Express a filter as a subset of SQL-92
    - Field identifiers, literal values
    - Logical connectors (OR, AND, NOT)
    - Comparison connectors (<, >, ==, etc.)
    - Some other operators such as IN, IS [NOT] NULL, LIKE, BETWEEN, etc.
  - Example

```
String selector = "JMSType='car' AND (color='blue' OR color='red')
                        AND price IS NOT NULL";

QueueReceiver receiver = session.createReceiver(queue,selector);
```

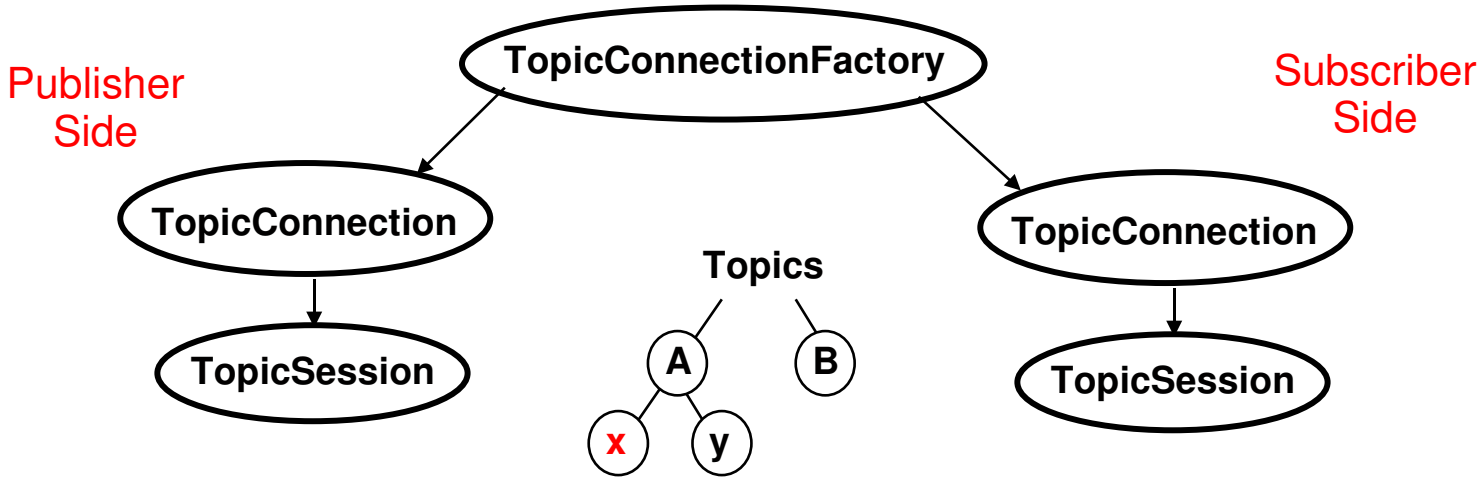
JMS – Temporary Queue Example

- Client-Server Paradigm
  - A typical pattern using normal message queues and temporary queues
    - When producing a message for a destination
      - Set the *JMSReplyTo destination* to a local temporary destination
    - When receiving a message
      - Reply by sending the response to the *JMSReplyTo destination*



Olivier.Gruber@inria.fr

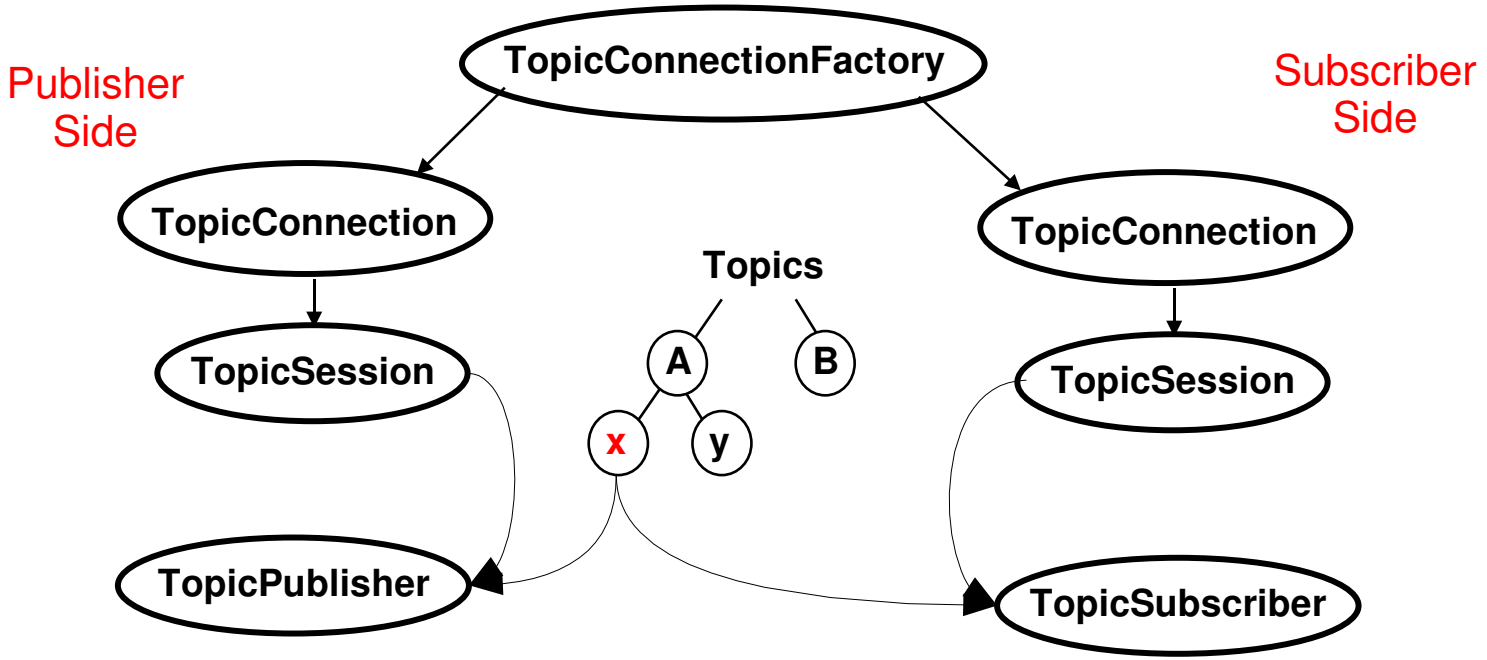
JMS – Topic Destination




```
InitialContext ictxt;
TopicConnectionFactory tcf;
TopicConnection topicConnection;
TopicSession topicSession;
try {
  ictxt = new InitialContext();

  tcf = (TopicConnectionFactory) ictxt.lookup("TopicConnectionFactory");
  topicConnection = tcf.createTopicConnection();
  topicSession = topicConnection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
} catch (NamingException ex) {
  // handler...
} catch (JMSException ex) {
  // handler...
}
```

Olivier.Gruber@inria.fr

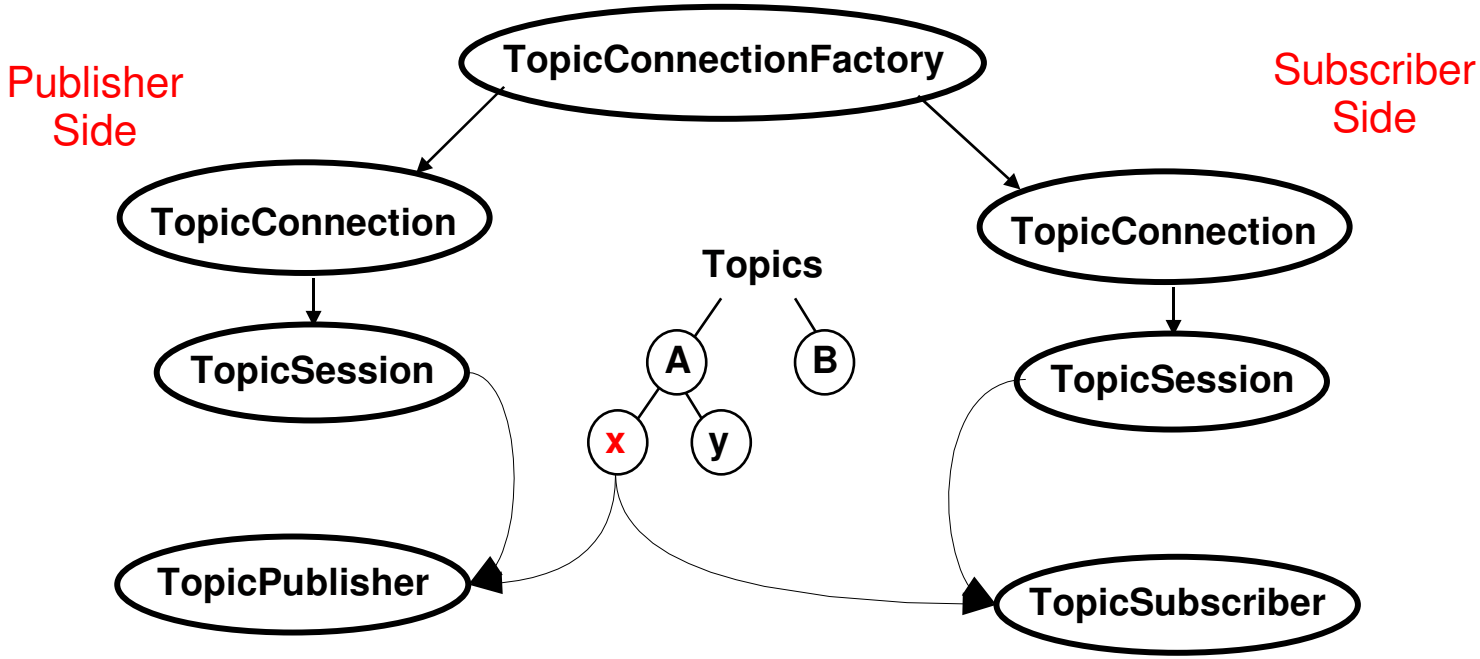
JMS – Topic Destination




```
Topic topic = (Topic) ictxt.lookup("/A/x");

TopicPublisher publisher;
publisher = topicSession.createPublisher(topic);
```

Olivier.Gruber@inria.fr

JMS – Topic Destination

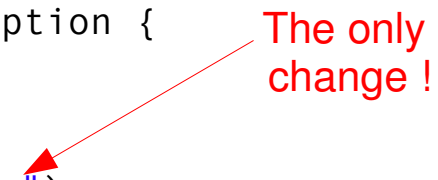



```
Topic topic = (Topic) ictxt.lookup("/A/x");

MessageConsumer consumer;
consumer = topicSession.createConsumer(topic);

TopicSubscriber subscriber;
subscriber=(TopicSubscriber)consumer;
```

Olivier.Gruber@inria.fr

JMS – Push Consumer

```
public class PushConsumer {

  public static void main(String[] args) throws Exception {

    InitialContext ictx;
    ictx = new InitialContext();
    Destination queue = (Destination) ictx.lookup("topic");
    ConnectionFactory cf = (ConnectionFactory) ictx.lookup("qcf");
    ictx.close();

    Connection cnx = cf.createConnection("user", "password");
    Session sess = cnx.createSession(false, Session.AUTO_ACKNOWLEDGE);
    MessageConsumer recv = sess.createConsumer(queue);

    recv.setMessageListener(new MessageListener() {
      public void onMessage(Message msg) {
        try {
          if (msg instanceof TextMessage) {
            System.out.println(((TextMessage) msg).getText());
          } else if (msg instanceof ObjectMessage) {
            System.out.println(((ObjectMessage) msg).getObject());
          }
        } catch (JMSException jE) {
          System.err.println("Exception in listener: " + jE);
        }
      }
    });
    cnx.start();
  }
}
```



Olivier.Gruber@inria.fr

JMS – Delivery

- Message Delivery on Connections
  - A connection must be started before any message may be received
    - Important for both synchronous or asynchronous delivery
    - Synchronous Delivery
      - Developers use explicit calls to receive
        - Such calls may be blocking or not
      - Possible on both queues and topics
    - Asynchronous Delivery
      - Developers register listeners for messages
        - JMS defines the MessageListener interface with the *onMessage* method
      - Possible on both queues and topics
  - Message delivery may be paused
    - This is done by calling Session.stop() method
      - Blocks current and future calls to receive
      - Stops all further asynchronous delivery
    - Session.stop() method will only return once delivery has effectively paused
      - So expect onMessage callbacks on asynchronous sessions

Olivier.Gruber@inria.fr

JMS – Threading Model

- Single-Threaded Model
  - Sessions are designed for serial use by one thread of control
    - Each session must be used synchronously or (exclusive-or) asynchronously
  - Synchronous paradigm
    - Typical usage pattern is the event loop
      - The control thread (**your thread**) waits on a message (receive call)
      - Once awaken, that control thread may use different message producers
      - No two threads may call receive concurrently on the same session
      - Do not register message listeners
    - Start the session's connection before calling receive
  - Asynchronous sessions
    - Only register listeners on stopped a connection
      - Register one or more listeners and then call Connection.start()
    - The control thread (**not your thread**) is dedicated to those listeners
      - Sessions serialize asynchronous message deliveries
      - Per session, there is no parallelism between onMessage methods

Olivier.Gruber@inria.fr

JMS – Threading Model

- Multi-Threaded Model
  - Use multiple sessions per connection for this
  - **Multi-threaded Publish-Subscribe**
    - Different sessions on the same connection may each have a TopicSubcriber for the same topic
    - Messages are multicasted to all listeners on the session control threads
    - If one consumer lags behind, it does not slow down the others
  - **Multi-threaded Point-to-Point**
    - No defined semantics in JMS for multiple QueueReceiver for the same queue
    - JMS Providers are allowed to support this
    - This is provider-dependent and therefore non portable

JMS – Message Ordering

- Partial Ordering
  - **Messages sent by a session to a destination must be received in the same order**
  - No order across destinations or sessions
    - No global order
      - Would be too costly, JMS providers would not accept such a constraint
      - A practical *consensus* position, typical of standard bodies...
    - No causal order
      - Even within the same application using different destinations or sessions
  - This partial order is important however
    - It defines the semantics of message acknowledgments and recovery

JMS Reliability

- Basic Reliability Mechanisms
  - Controlling message acknowledgments
  - Specifying message persistence
  - Setting message priority levels
  - Allowing messages to expire
- Advanced Reliability Mechanisms
  - Creating durable subscriptions
  - Using local transations
  - Using distributed transactions (if supported)

JMS – Basic Reliability

- Acknowledgement policy
  - Until a message has been acknowledged
  - It is not considered as successfully consumed
- Steps of a successful consumption
  - The client is delivered the message by the JMS middleware
  - The client processes the message
  - The message is acknowledged
    - Either by the JMS client code or by the JMS middleware
    - This acknowledgement policy is set on a session

JMS – Basic Reliability

- Basic Acknowledgement Policies
  - AUTO_ACKNOWLEDGE
    - Middleware automatically acknowledges messages
      - Upon successfully returned from a receive call
      - Upon the termination of an asynchronous delivery (the onMessage method)
  - CLIENT_ACKNOWLEDGE
    - Explicit acknowledgement needed from client code
      - The JMS client code explicitly calls *Session.acknowledge()*
      - Acknowledges all delivered messages from the last acknowledge message
    - Session recovery
      - The JMS client code **may explicitly call** *Session.recover()*
      - Recovers all delivered messages up to the last acknowledged message
  - DUPS_OK_ACKNOWLEDGE
    - Lazy background acknowledgement of messages
    - May result in duplicate delivery of messages
      - If the JMS provider fails
      - Redelivery is indicated through the *JMSRedelivered* message header

Olivier.Gruber@inria.fr

JMS – Basic Reliability

- QueueSession Termination
  - Received but unacknowledged messages are kept
  - They will be redelivered upon the next receive on that queue
  - Unless the message expires (see Time-To-Live)
- TopicSession Termination
  - Received but unacknowledged messages are kept only if there is at least one durable subscription
  - Otherwise received but unacknowledged messages are dropped

JMS – Basic Reliability

- Message Time-To-Live
  - Each message may have a Time-To-Live
    - Set on a per-producer basis through the MessageProducer.setTimeToLive(int)
    - Or on a per-message basis when the message is produced
  - JMS provider drops messages once their TTL has expired
- Message Priority
  - Each message may have a priority
    - Set on a per-producer basis through the MessageProducer.setPriority(int)
    - Or on a per-message basis when the message is produced
  - JMS provider delivers higher priority messages first

JMS – Basic Reliability

- Message Persistence
  - This is about produced messages (sent or published)
    - Determines whether messages may be lost if the JMS provider fails
  - **Persistent Mode**
    - The semantics is to deliver transient messages ***once and only once***
      - Messages are not lost in transit, even with JMS Provider failures
      - But messages must never be delivered more than once to a given client
  - **Transient Mode**
    - The semantics is to deliver transient messages ***at most once***
      - Messages may be lost in transit
      - But messages must never be delivered more than once to a given client
  - Specified on message producers
    - Through the use of the *MessageProducer.setDeliveryMode(int)*
      - Can also be specified on a per message basis
    - Both persistent and non-persistent messages can be delivered to the same destination

JMS – Reliability

- Basic Reliability Mechanisms
  - Controlling message acknowledgments
  - Specifying message persistence
  - Setting message priority levels
  - Allowing messages to expire
- **Advanced Reliability Mechanisms**
  - Creating durable subscriptions
    - Offers the reliability of queues to the pub-sub paradigm
  - Using local transations
    - Group message consumption and production in **local atomic** units of work
  - Using distributed transactions (if supported)
    - Group message consumption and production in **global atomic** units of work

JMS – Advanced Reliability

- Subscription Durability
  - This is about message consumption through subscriptions
  - **Non-Durable Subscriptions**
    - By default, subscriptions are non durable
      - Messages are delivered only to registered listeners
      - Once delivered, messages are dropped
    - When a non-durable consumer terminates
      - Messages awaiting delivery to that consumer will be dropped
  - **Durable Subscriptions**
    - Messages are either delivered or stored for future delivery
      - Durable subscriptions are created with a unique identifier that allows to reconnect to the same subscription at a later time
      - A session may have only one durable subscription for a given topic
    - Stronger than persistent semantics
      - For messages that clients acknowledge explicitly
      - Only durable subscriptions are able to recover unacknowledged messages

Olivier.Gruber@inria.fr

JMS – Advanced Reliability

- Durable Subscription

```
try {
  ictxt = new InitialContext();

  tcf = (TopicConnectionFactory) ictxt.lookup("TopicConnectionFactory");
  topicConnection = tcf.createTopicConnection() ;
  topicSession = topicConnection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE)

  Topic topic = (Topic) ictxt.lookup("/A/x");

  TopicSubscriber topicSubscriber;
  topicSubscriber = topicSession.createDurableSubscriber(topic,"MySub");

  topicConnection.start();

  // .... your code

  topicSubscriber.close(); // closes the subscription

  topicSession.unsubscribe("MySub");
  // frees the internal resources for the durable subscription
  // that would otherwise keep messages

} catch (NamingException ex) {
  // handler...
} catch (JMSException ex) {
  // handler...
}
```

Olivier.Gruber@inria.fr

JMS – Advanced Reliability

- Local Transactions
  - A session may be transacted or not (specified when created)
    - Follows a saga model
      - A transacted session supports a single sequence of transactions
      - Completion of the current transaction starts a new one
    - Each transaction is an ***atomic unit of work***
      - *Groups produced and consumed messages*
      - Upon commit --- *Session.commit()*
        - All consumed messages are acknowledged
        - All produced messages are sent (and made persistent if necessary)
      - Upon rollback --- *Session.rollback()*
        - All consumed messages are recovered and will be redelivered
        - All produced messages are dropped (destroyed and not sent)

Olivier.Gruber@inria.fr

JMS – Advanced Reliability

- Distributed Transactions
  - JMS does not require that JMS providers support distributed transactions
    - But if they do provide distributed transactions, they must comply to either
      - JTA *XARessource* (distributed transaction participant)
      - Be itself a JTA distributed transaction monitor
    - Extended core concepts
      - *XAConnectionFactory*
      - *XAConnection*
      - *XASession*
    - *Each transaction is an **atomic unit of work across sessions***
      - *Groups produced and consumed messages*
      - Global two-phase commit across sessions from different clients
  - Examples
    - A banking system that integrates through JMS several databases
    - A booking system like Expedia

Olivier.Gruber@inria.fr

JMS – Reliability Summary

Published as	Non-Durable Subscriber	Durable Subscriber	Queue
NON PERSISTENT	at most once (missed if inactive) (may be lost)	at most once (may be lost)	at most once (may be lost)
PERSISTENT	once and only once (missed if inactive)	once and only once	once and only once

Messages may also **expire (TTL)** before they are delivered (either in transit or waiting at clients). This applies to both both synchronous and asynchronous delivery

Olivier.Gruber@inria.fr

JMS – Resource Management

- Resource Allocation
  - JMS Providers may allocate substantial resources for
    - Connections and sessions as well as message producers or consumers
    - Both Java and non-Java resources
  - *The usual Java approach to this resource problem...*
    - Introduces a close method (like the dispose on SWT objects for instance)
    - Calling the close method ensures proper resource management
    - Relying on Java Garbage Collection may not be timely enough

JMS – Does and Do not

- Java Messaging Service
  - Standardize on the programming model and the API
  - Does not standardize administrative tasks
    - Creating and registering JMS objects
      - Queues, topics or connection factories
    - Distributed architecture of the middleware
      - On which network nodes are JMS objects?
      - Are those objects replicated, clustered?
    - Levels of security in message transport
      - Authentication and encryption
    - De facto, reduces portability of applications
      - It is often hard to totally separate the application from its administrative aspects
      - E.g. dynamic creation of JMS middleware nodes, queues or topics
  - Does not standardize interoperability
    - Nothing is said about how two JMS providers may interoperate
      - Sending and receiving messages across two JMS providers
      - No naming conventions for JMS objects (like using reverse-domain names)

Olivier.Gruber@inria.fr