JMS
Java Messaging System

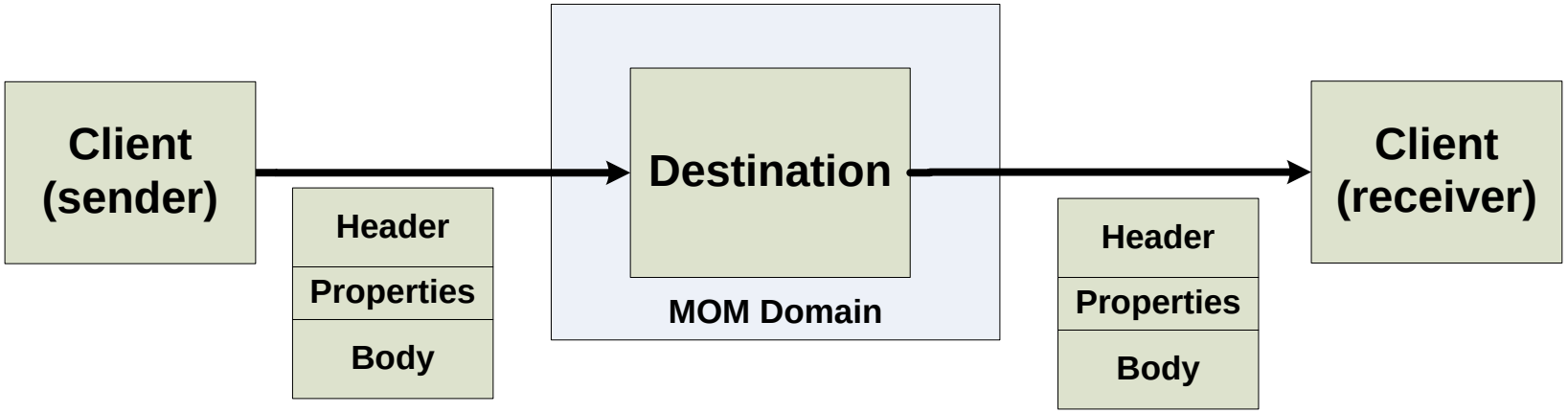
Message-Oriented Middleware (MOM)

- MOM provides a common, reliable way for creation, sending, receiving and reading messages in a distributed system
- Typically used in large enterprise systems
  - Banking
  - Telecom
  - etc

MOM Destinations

- MOM clients send their messages to a ***destination***

-The ***destination*** delivers it to the appropriate receiver



(This figure is not from the JMS tutorial)

MOM features

- Asynchronous communication
- Guaranteed message delivery
- Transaction support
- One time, in-order delivery
- Message routing services

MOM advantages

- Leads to loosely coupled components
  - Robust to change
  - Location indepence
  - Time indepence
- Simplicity
- Scalability
- Configurable Quality of Service (QoS)

Java Message Service (JMS)

- JMS is a J2EE API to access MOM products from Java
- ”JMS is analogous to JDBC”
  - JDBC is a vendor-independent access to many different rational databases
  - JMS is vendor-independent access to different messaging systems
- JMS provide the common MOM features
  - MOM products usually has plenty of proprietary features

JMS implementations

- Some MOM products that supports JMS:
  - ActiveMQ Appache
  - IBM MQSeries
  - BEA WebLogic JMS service
  - Sun’s iPlanet Message Queue
  - Progress SonicMQ
  - FioranoMQ
  - JBossMQ
  - etc

Java RMI vs. Messaging

- The differences between (Remote) Procedure Calls and message passing
  - (R)PC is a special case where the receiver takes preemptive priority over the sender (caller), i.e. The caller waits until the call has returned
- Another view: implement asynchronous messages by 2 RPC calls, one in each direction (so called ’call back’), often less efficient

Loosely coupled: 2 meanings

- The sender and the receiver need to know only what message format and what destination to use.
  - In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI)
- Loosely coupled regarding parallel and concurrent behaviour is best obtained by asynchronous message passing
- By separating message sent, and method invoked, RMI-like systems will become more loosely coupled in the first meaning, not in the second.

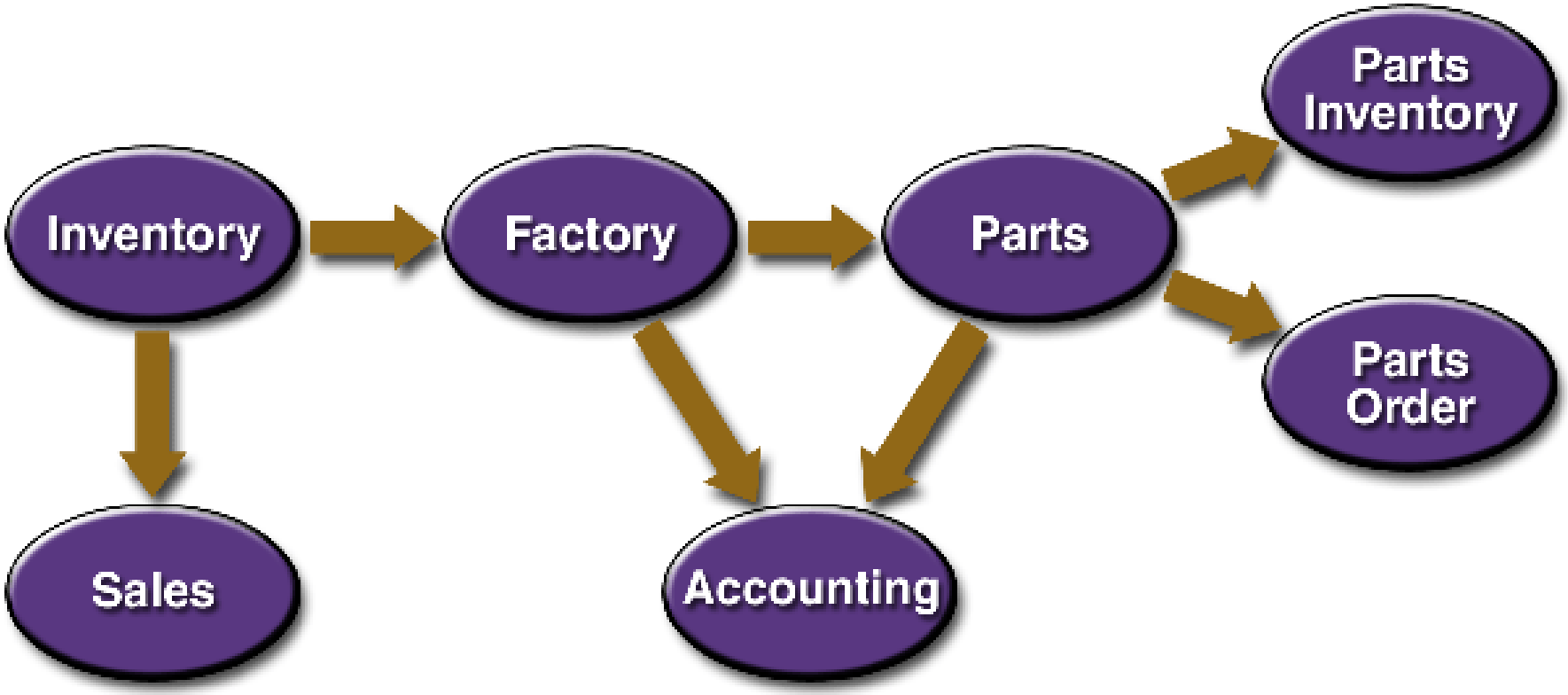
Not addressed by the JMS API

- Transport protocol for messaging
- Message metadata repository
- Load balancing criteria and Fault tolerant mechanisms
- Standardized error notification (apart from Exceptions)
- Administration API
- Security API (dig. signatures, key distr.)

When Can You Use the JMS API?

- Some arguments, not all are relevant for RT engineering
- The provider wants the components not to depend on information about other components' interfaces, so that components can be easily replaced.
  - This is just a rewriting of the first loosely coupled definition from previous slide
- The provider wants the application to run whether or not all components are up and running simultaneously.
- The application business model allows a component to send information to another and to continue to operate without receiving an immediate response.

Ex. Application



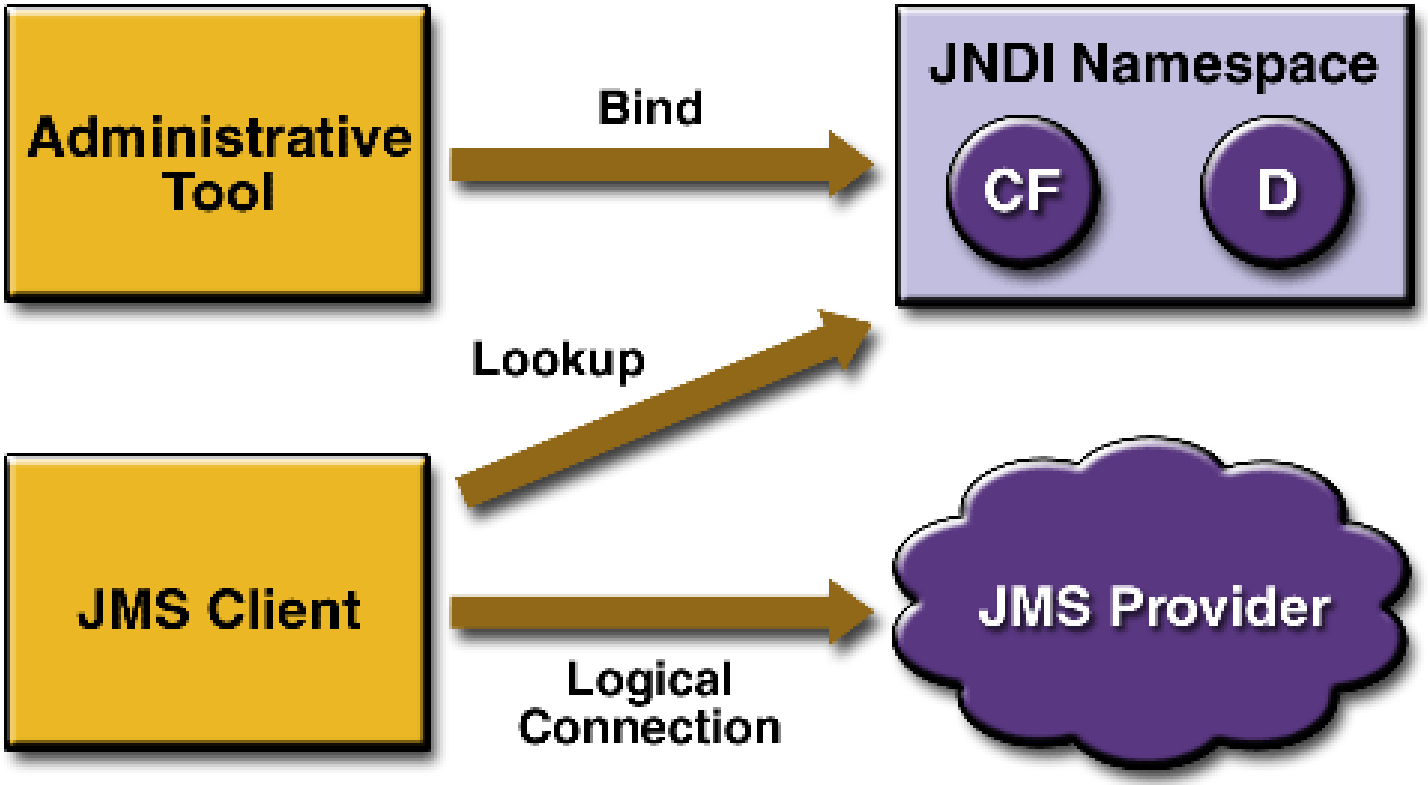
Basic JMS API Concepts

- A *JMS provider* is a messaging system that implements JMS and provides administrative and control features.
- *JMS clients* are the programs or components, written in the Java, that produce and consume messages.
- *Messages* are the objects that communicate information between JMS clients.
- *Administered objects* are preconfigured JMS objects created by an administrator for the use of clients.

JMS Destinations (Administrated objects)

- ***Queue*** (Point-to-point)
  - Messages placed in the queue are processed by a single client
  - But multiple clients can listen (message goes to first available)
- ***Topic*** (Publish/Subscribe)
  - All clients subscribing to a Topic get a copy of the message each.

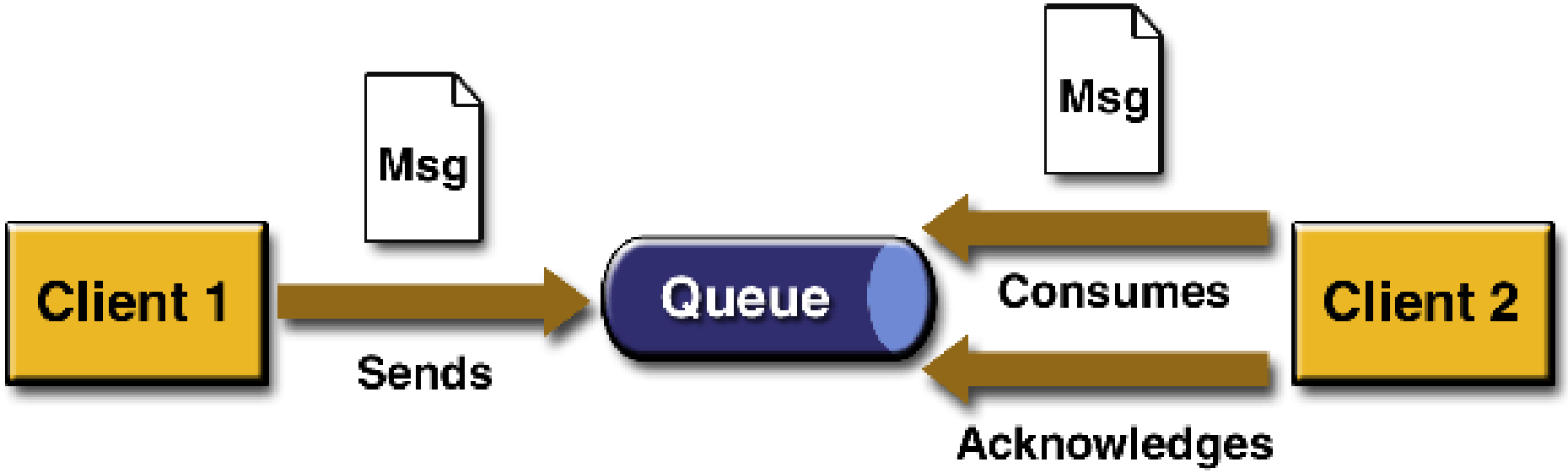
JMS API Architecture



2 approaches to messaging:

- *point-to-point*
- *publish/subscribe*
- The JMS Specification provides a separate domain for each approach and defines compliance for each domain.
- A standalone JMS provider may implement one or both domains (approach types).

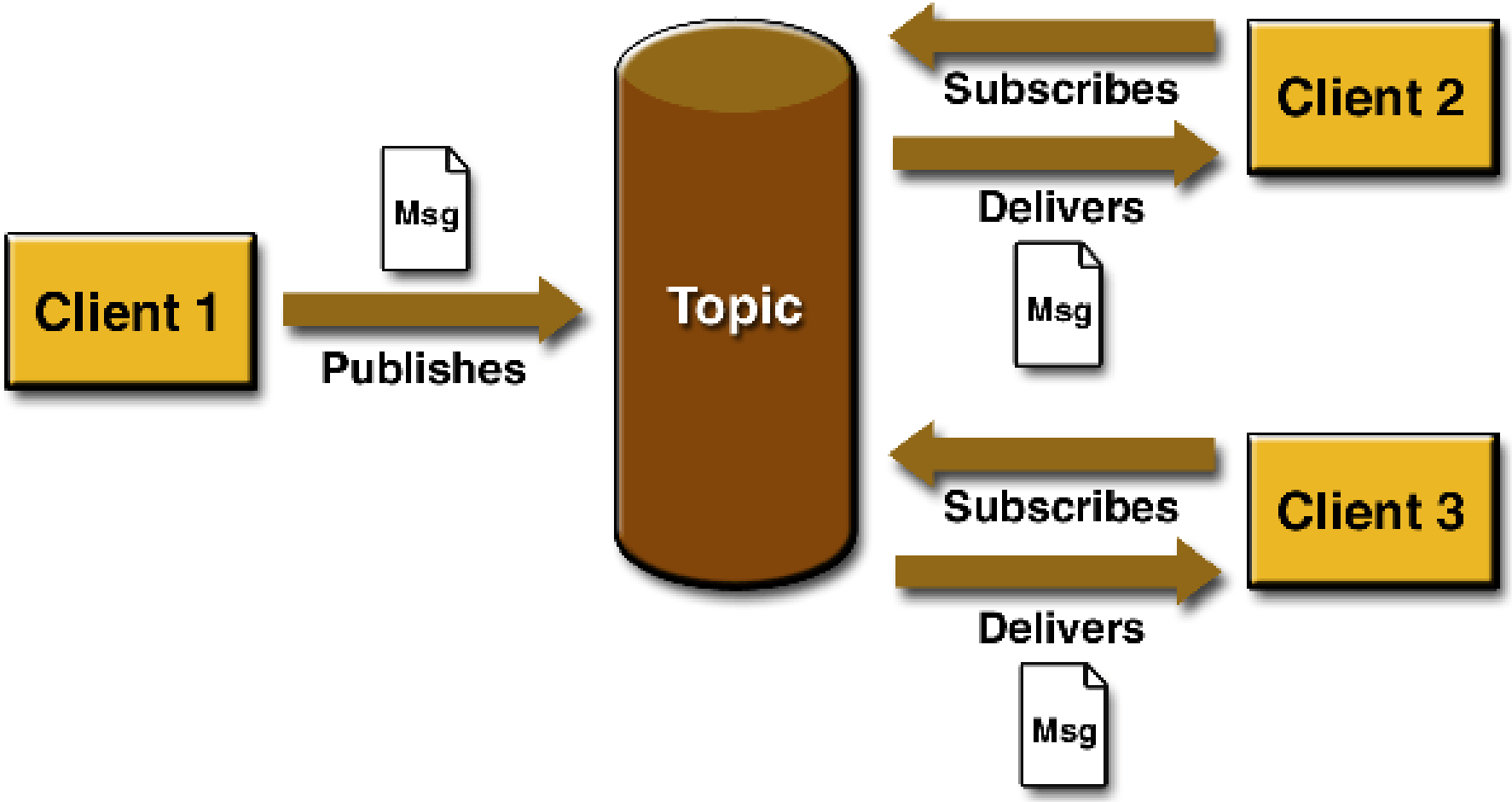
Point-to-point messaging



Point-to-point messaging

- Each message has only one consumer.
- A sender and a receiver of a message have no timing dependencies.
- The receiver can fetch the message whether or not it was running when the client sent the message.
- The receiver acknowledges the successful processing of a message.

Publish/ Subscribe



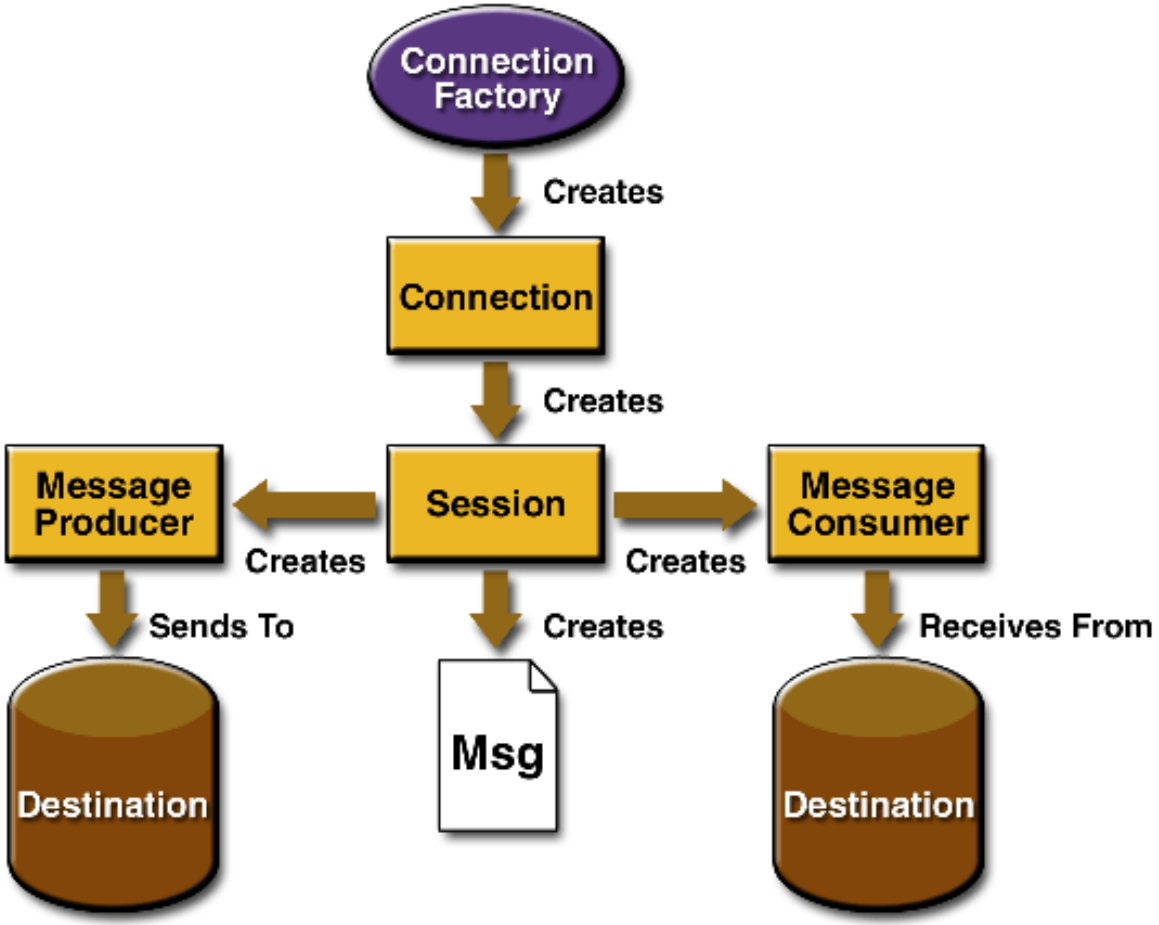
Publish/subscribe

- Each message may have multiple consumers.
- Publishers and subscribers have a timing dependency.
  - A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages. (this may be relaxed)

Message Consumption

- **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
- **Asynchronously.** A client can register a *message listener* with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage` method, which acts on the contents of the message.

The basic building blocks of a JMS application



A JMS message has three parts:

- A header
- Properties (optional)
- A body (optional)

JMS headers

Table 3.1: How JMS Message Header Field Values Are Set

Header Field	Set By
JMSDestination	send or publish method
JMSDeliveryMode	send or publish method
JMSExpiration	send or publish method
JMSPriority	send or publish method
JMSMessageID	send or publish method
JMSTimestamp	send or publish method
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	JMS provider

Message types and bodies

Table 3.2: JMS Message Types

Message Type	Body Contains
`TextMessage`	A `java.lang.String` object (for example, the contents of an Extensible Markup Language 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.

A Simple Point-to-Point Example

- Simple JMS clients : Simple standalone programs that run outside the server as class files.
- The clients demonstrate the basic tasks:
  - Creating a connection and a session
  - Creating message producers and consumers
  - Sending and receiving messages

The sending program, SimpleQueueSender.java:

1. Performs a Java Naming and Directory Interface (JNDI) API lookup of the QueueConnectionFactory and queue
2. Creates a connection and a session
3. Creates a QueueSender
4. Creates a TextMessage
5. Sends one or more messages to the queue
6. Sends a control message [end of stream]
7. Closes the connection in a finally block, automatically closing the session and QueueSender

The receiving program, SimpleQueueReceiver.java,:

1. Performs a JNDI API lookup of the QueueConnectionFactory and queue
2. Creates a connection and a session
3. Creates a QueueReceiver
4. Starts the connection, causing message delivery to begin
5. Receives the messages sent to the queue until the end-of-message-stream control message is received
6. Closes the connection in a finally block, automatically closing the session and QueueReceiver

Simple receive

- If you specify no arguments or an argument of 0, the method blocks indefinitely until a message arrives:
- Message m = queueReceiver.receive();
- Message m = queueReceiver.receive(0)
- But if you do not want your program to consume system resources unnecessarily, use a timed synchronous receive.

Timed synchronous receive

- If you do not want your program to consume system resources unnecessarily, do one of the following:
- Call the receive method with a timeout argument greater than 0:
  - Message m = queueReceiver.receive(1); // 1 ms
- Call the receiveNoWait method, which receives a message only if one is available:
  - Message m = queueReceiver.receiveNoWait();

Writing the Pub/Sub Client Programs

- Create connections and sessions
- Create message publisher
- Create message consumer (subscribers))
- Send /receive messages
- Close down

Publishing program, SimpleTopicPublisher.java:

1. Performs a JNDI API lookup of the TopicConnectionFactory and topic
2. Creates a connection and a session
3. Creates a TopicPublisher
4. Creates a TextMessage
5. Publishes one or more messages to the topic
6. Closes the connection, which automatically closes the session and Topic- Publisher

Receiving program, SimpleTopicSubscriber.java:

1. Performs a JNDI API lookup of the TopicConnectionFactory and topic
2. Creates a connection and a session
3. Creates a TopicSubscriber
4. Creates an instance of the TextListener class and registers it as the message listener for the TopicSubscriber
5. Starts the connection, (message delivery begins)
6. Listens for the messages published to the topic, stopping when the user enters the character q
7. Closes the connection, which automatically closes the session and TopicSubscriber

The message listener, TextListener.java:

1. When a message arrives, the onMessage method is called automatically.
2. The onMessage method converts the incoming message to a TextMessage and displays its content.

Creating Robust JMS applications

- The most reliable way to produce a message is to send a PERSISTENT message within a transaction. JMS messages are PERSISTENT by default.
- A *transaction* is a unit of work into which you can group a series of operations
- These bullets are mostly relevant for DB intense applications, not for RT like telephony
- Other types of robustness, such as built by timers are as relevant for some RT systems

Using Basic Reliability Mechanisms

- **Controlling message acknowledgment.** You can specify various levels of control over message acknowledgment.
- **Specifying message persistence.** You can specify that messages are persistent, meaning that they must not be lost in the event of a provider failure.
- **Setting message priority levels.**
- **Allowing messages to expire.** You can specify an expiration time for messages