Politecnico di Milano

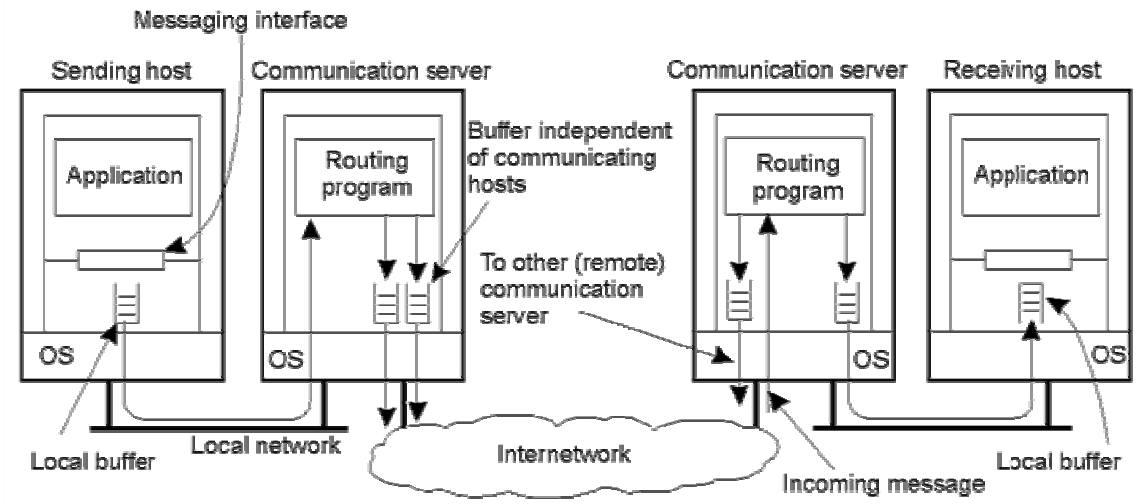
Introduction to MOM

- RPC/RMI foster a synchronous model
  - Natural programming abstraction, but:
    - Supports only point-to-point interaction
    - Synchronous communication is expensive
    - Intrinsically tight coupling between caller and callee, leads to "rigid" architectures
- Asynchronous communication models:
  - Often centered around the notion of message/event
    - Or persistent communication
  - Often supporting multi-point interaction
  - Brings more decoupling among components

Politecnico di Milano

Reference model

- The most straightforward form of asynchronous communication is message passing
  - Typically directly mapped on/provided by the underlying network OS functionality (e.g., socket, datagrams)
- In the case of MOM, the messaging infrastructure is often provided at the application level, by several "communication servers"
  - Through what is nowadays called an *overlay network*



Politecnico di Milano

MOM: Types of communication

- Synchronous vs. asynchronous
  - Synchronous: the sender is blocked until the recipient has stored (or received, or processed) the message
  - Asynchronous: the sender continues immediately after sending the message
- Transient vs. persistent
  - Transient: sender and receiver must both be running for the message to be delivered
  - Persistent: the message is stored in the communication system until it can be delivered
- Several alternatives (and combinations) are provided in practice
- Many are used also for implementing synchronous models (e.g., RPC)

Politecnico di Milano

Different MOM flavors

- Message oriented communication comes in two flavors: Message queuing and publish-subscribe systems
- Several common characteristics
  - Both are "message oriented"
  - Both offer a strong decoupling among components
  - Both are often based on a set of "servers" to route messages from sender to recipients and/or to support persistency
- But also a lot of differences...

Politecnico di Milano

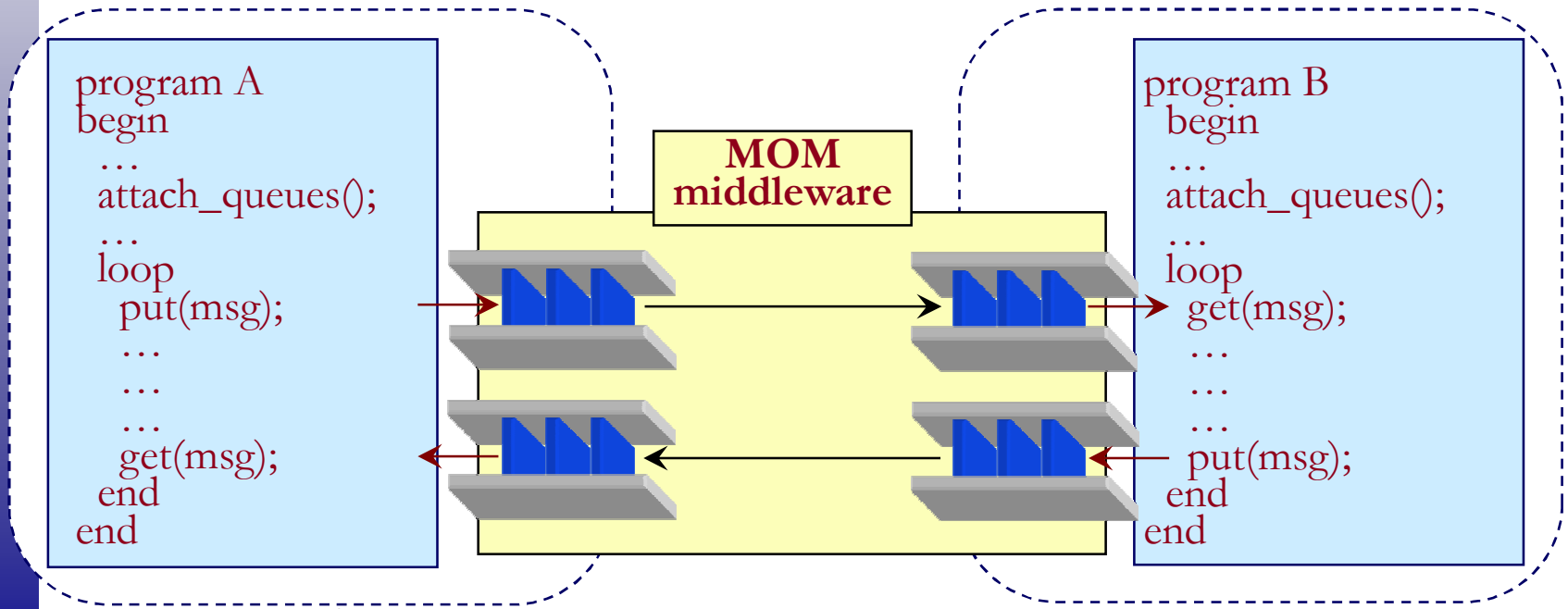
Message queuing

- Point-to-point persistent asynchronous communication
  - Typically guarantee only eventual insertion of the message in the recipient queue (no guarantee about the recipient's behavior)
  - Communication is decoupled in time and space
  - Can be regarded as a generalization of the e-mail
- Intrinsically peer-to-peer architecture
- Each component holds an input queue and an output queue
- Many commercial systems:
  - IBM MQSeries (now WebSphere MQ), DECmessageQ, Microsoft Message Queues (MSMQ), Tivoli, Java Message Service (JMS), …

Politecnico di Milano

Queuing: Communication primitives

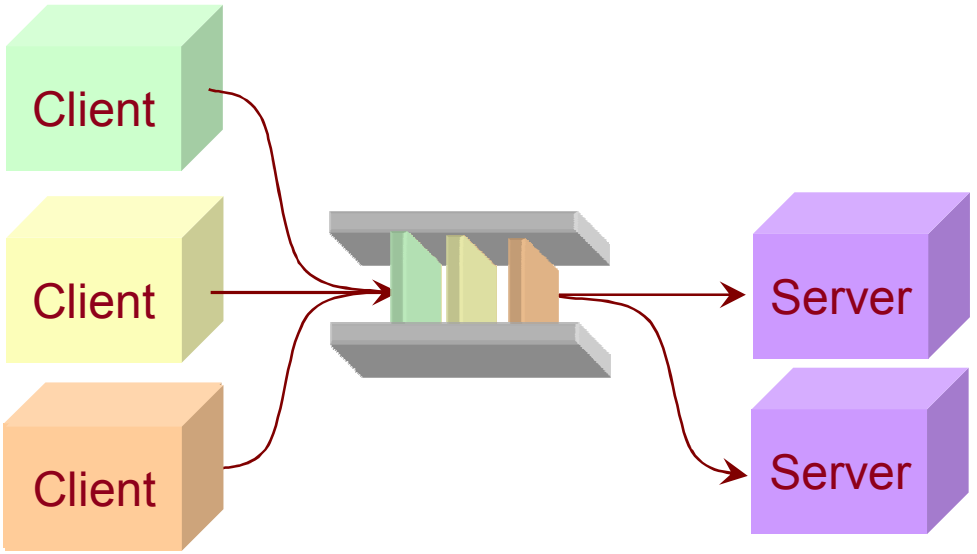
Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue



Politecnico di Milano

Client-Server with queues

- Clients send requests to the server's queue
- The server asynchronously fetches requests, processes them, and returns results in the clients' queues
  - Thanks to persistency and asynchronicity, clients need not remain connected
  - Queue sharing simplifies load balancing



Politecnico di Milano

Message queuing: Architectural Issues

- Queues are identified by symbolic names
  - Need for a lookup service, possibly distributed
  - Often pre-deployed static topology
- Queues are manipulated by queue managers
  - Local and/or remote, acting as relays (a.k.a. applicative routers)
- Relays often organized in an overlay network
  - Messages are routed by using application-level criteria, and by relying on a partial knowledge of the network
  - Improves fault tolerance
  - Provides applications with multi-point without IP-level multicast
- Message brokers provide application-level gateways supporting message conversion
  - Useful when integrating sub-systems

Politecnico di Milano

Publish-subscribe

- Application components can *publish* asynchronous *event notifications*, and/or declare their interest in event classes by issuing a *subscription*
  - extremely simple API: only two primitives (`publish`, `subscribe`)
  - event notifications are simply messages
- Subscriptions are collected by an *event dispatcher* component, responsible for routing events to all matching subscribers
  - Can be centralized or distributed
- Communication is transiently asynchronous, implicit, multipoint
- High degree of decoupling among components
  - Easy to add and remove components
  - Appropriate for dynamic environments

Politecnico di Milano

Subscription Language

- The expressiveness of the subscription language allows one to distinguish between:
  - *Subject-based* (or topic-based)
    - The set of subjects is determined a priori
    - Analogous to multicast
    - e.g., subscribe to all events about "Distributed Systems"
  - *Content-based*
    - Subscriptions contain expressions (event filters) that allow clients to filter events based on their content
    - The set of filters is determined by client subscriptions
    - A single event may match multiple subscriptions
    - e.g., subscribe to all events about a "Distributed System" class with date greater than 16.11.2004 and held in classroom D04
- The two can be combined

Politecnico di Milano

JMS: An introduction

- The Java Message Service (JMS) defines an API to allow for creation, sending, receiving and reading messages belonging to an enterprise system
  - Strives for portability among different platforms
  - Tries to integrate seamlessly with the Java language
- First specification appeared in 1998
- JMS core concepts:
  - *JMS provider:* The entity that implements the JMS API for a messaging product
  - *JMS client:* The applicative software that uses the services of a JMS provider through the JMS API
  - *JMS domains:* Either point-to-point (queue based) or publish-subscribe (topic based)

Politecnico di Milano

JMS domains

- The point-to-point domain:
  - Each message is addressed to a specific *queue*
  - Client extract messages from the queue(s) established to hold their messages
- The publish-subscribe domain is centered on a specific content-hierarchy (*topics*)
  - The system takes care of distributing the messages belonging to a specific part of the content hierarchy to all clients interested in that content
- JMS provides a set of domain-independent interfaces allowing communication in both domains under a common API
  - Usually called "common interfaces"

Politecnico di Milano

JMS Architecture

- *JMS Clients:* Java programs sending and/or receiving messages
- *Non-JMS Clients:* Clients using the native API of the messaging system
- *Messages:* Data containers defined by the application to exchange information
- *JMS Provider:* The messaging product implementing the JMS API together with (possible) administrative and quality-control functionality
- *Administered Objects:* Preconfigured JMS objects created by an administrator and used by the clients

Politecnico di Milano

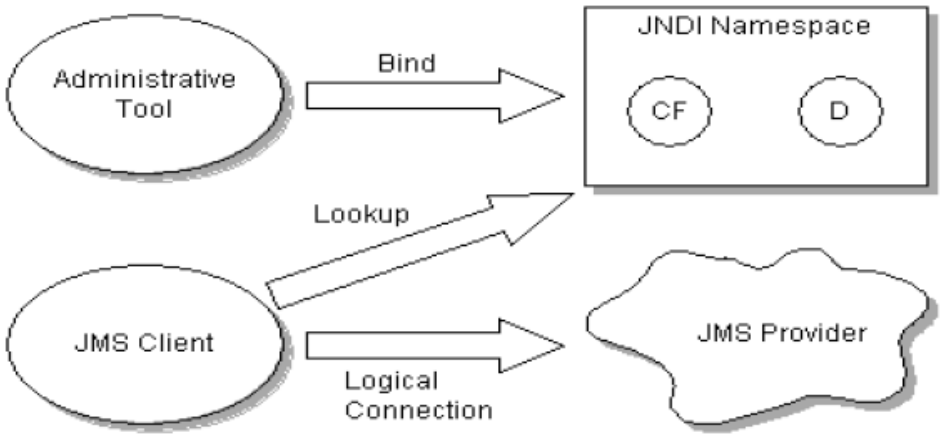
JMS Administration

- To allow for portability, the details regarding a specific JMS Provider must be hidden
- This is done by defining JMS administered objects
  - Created and customized by administrators
  - Later accessed via portable JMS interfaces by clients
- Two types of administered objects:
  - *ConnectionFactory:* Used by a client to establish a connection to the JMS provider
  - *Destination:* Used by a client to specify the destination of messages it will send

Politecnico di Milano

JMS Lookup

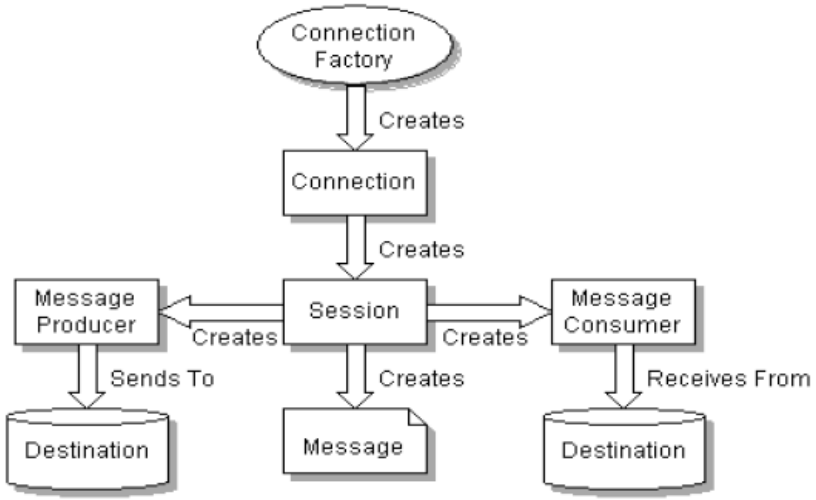
- Administered objects are placed in a JNDI namespace under well-known identifiers
- To establish a connection to a JMS provider, clients retrieves a `ConnectionFactory` from the JNDI namespace



Politecnico di Milano

JMS Object Relationships

- A `Connection` represents a link between the client and a JMS Provider
- A `Destination` encapsulates the identity of a message destination
- A `Session` is a single-threaded context for sending and receiving messages
- Message producers and consumers send and receive messages in a `Session`, respectively



Politecnico di Milano

Developing JMS applications

- A JMS client will typically:
  - Use JNDI to find a `ConnectionFactory` object
  - Use JNDI to find one or more `Destination` object(s)
    - Predefined identifiers can also be used to instantiate new `Destinations`
  - Use the `ConnectionFactory` to establish a `Connection` with the JMS Provider
  - Use the `Connection` to create one or more `Session`(s)
  - Combine `Session`(s) and `Destination`(s) to create the needed `MessageConsumer` and `MessageProducer`
  - Tell the `Connection` to start delivery of messages

Politecnico di Milano

Example: The point-to-point domain

```
import javax.jms.*; import javax.naming.*;

public class Sender {
  static Context ictx = null;
  public static void main(String[] args) throws
     Exception {
    ictx = new InitialContext();
    Queue queue = (Queue) ictx.lookup("queue");
    QueueConnectionFactory qcf =
     (QueueConnectionFactory) ictx.lookup("qcf");
    ictx.close();

    QueueConnection qc = qcf.createQueueConnection();
    QueueSession qs = qc.createQueueSession(false,
                      Session.AUTO_ACKNOWLEDGE);
    QueueSender qsend = qs.createSender(queue);
    TextMessage msg = qs.createTextMessage();

    int i;
    for (i = 0; i < 10; i++) {
      msg.setText("Test number " + i);
      qsend.send(msg);
    }

    qc.close();
  }
}
```

```
import javax.jms.*; import javax.naming.*;

public class Receiver {
  static Context ictx = null;
  public static void main(String[] args) throws
     Exception {
    ictx = new InitialContext();
    Queue queue = (Queue) ictx.lookup("queue");
    QueueConnectionFactory qcf =
     (QueueConnectionFactory) ictx.lookup("qcf");
    ictx.close();

    QueueConnection qc = qcf.createQueueConnection();
    QueueSession qs = qc.createQueueSession(false,
                      Session.AUTO_ACKNOWLEDGE);
    QueueReceiver qrec = qs.createReceiver(queue);
    TextMessage msg;

    qc.start();

    int i;
    for (i = 0; i < 10; i++) {
      msg = (TextMessage) qrec.receive();
      System.out.println("Msg received: " +
        msg.getText());
    }

    qc.close();
  }
}
```

Politecnico di Milano

Example: The pub-sub domain

```
import javax.jms.*; import javax.naming.*;

public class Publisher {
  static Context ictx = null;
  public static void main(String[] args) throws Exception {
    ictx = new InitialContext();
    Topic topic = (Topic) ictx.lookup("topic");
    TopicConnectionFactory tcf = (TopicConnectionFactory)
     ictx.lookup("tcf");
    ictx.close();

    TopicConnection tc = tcf.createTopicConnection();
    TopicSession ts = tc.createTopicSession(true,
                      Session.AUTO_ACKNOWLEDGE);
    TopicPublisher tpub = ts.createPublisher(topic);
    TextMessage msg = ts.createTextMessage();

    int i;
    for (i = 0; i < 10; i++) {
      msg.setText("Test number " + i);
      tpub.publish(msg);
    }

    ts.commit();
    tc.close();
  }
}
```

```
import javax.jms.*; import javax.naming.*;

public class Subscriber {
  static Context ictx = null;
  public static void main(String[] args) throws Exception {
    ictx = new InitialContext();
    Topic topic = (Topic) ictx.lookup("topic");
    TopicConnectionFactory tcf = (TopicConnectionFactory)
     ictx.lookup("tcf");
    ictx.close();

    TopicConnection tc = tcf.createTopicConnection();
    TopicSession ts = tc.createTopicSession(true,
                      Session.AUTO_ACKNOWLEDGE);
    TopicSubscriber tsub = ts.createSubscriber(topic);

    tsub.setMessageListener(new MsgListener());
    tc.start();
    System.in.read();
    tc.close();
  }
}
class MsgListener implements MessageListener {
  String id;
  public MsgListener() {id = "";}
  public MsgListener(String id) {this.id = id;
  }
  public void onMessage(Message msg) {
    TextMessage tmsg = (TextMessage) msg;
    try {
      System.out.println(id+": "+tmsg.getText());
    } catch (JMSException jE) {
      jE.printStackTrace();
    }
  }
}
```

Politecnico di Milano

Example: The common interfaces

```
import javax.jms.*; import javax.naming.*;

public class Producer {
  static Context ictx = null;
  public static void main(String[] args) throws
     Exception {
    ictx = new InitialContext();
    Queue queue = (Queue) ictx.lookup("queue");
    Topic topic = (Topic) ictx.lookup("topic");
    ConnectionFactory cf =
      (ConnectionFactory) ictx.lookup("cf");
    ictx.close();

    Connection cnx = cf.createConnection();
    Session sess = cnx.createSession(false,
                   Session.AUTO_ACKNOWLEDGE);
    MessageProducer prod = sess.createProducer(null);
    TextMessage msg = sess.createTextMessage();

    int i;
    for (i = 0; i < 10; i++) {
      msg.setText("Test number " + i);
      prod.send(queue, msg);
      prod.send(topic, msg);
    }

    cnx.close();
  }
}
```

```
import javax.jms.*; import javax.naming.*;

public class Subscriber {
  static Context ictx = null;
  public static void main(String[] args) throws
     Exception {
    ictx = new InitialContext();
    Queue queue = (Queue) ictx.lookup("queue");
    Topic topic = (Topic) ictx.lookup("topic");
    ConnectionFactory cf =
      (ConnectionFactory) ictx.lookup("cf");
    ictx.close();

    Connection cnx = cf.createConnection();
    Session sess = cnx.createSession(false,
                   Session.AUTO_ACKNOWLEDGE);
    MessageConsumer recv = sess.createConsumer(queue);
    MessageConsumer subs = sess.createConsumer(topic);

    recv.setMessageListener(new MsgListener("Queue"));
    subs.setMessageListener(new MsgListener("Topic"));

    cnx.start();
    System.in.read();
    cnx.close();
  }
}
```

Politecnico di Milano

More JMS concepts

Concurrency

- JMS restricts concurrent access to `Destination`, `ConnectionFactory` and `Connection`
- `Session`, `MessageProducer` and `MessageConsumer` are not designed for concurrent usage by multiple threads
  - `Session` supports transactions, for which it is often difficult to write multithreading support
    - If concurrent send and receive of messages is required, it is better to use multiple sessions
  - Asynchronous message consumers running concurrently would require complicated concurrency support
    - Also, either a `Session` uses synchronous receive, or asynchronous ones
      - It is erroneous to use both

Politecnico di Milano

More JMS concepts
Message ordering & acknowledgment

- FIFO between a given sender and a given receiver modulo priority
- No guarantee across destinations or across a destination's messages sent by different producers
- If a `Session` is transacted, message acks are automatically handled by commit
- If a `Session` is not transacted, you can either set
  - `DUPS_OK_ACKNOWLEDGE`: instructs the `Session` to lazily acknowledge messages; this minimizes overhead while possibly generating duplicates messages if the Provider fails
  - `AUTO_ACKNOWLEDGE`: the `Session` automatically acknowledges messages while avoiding duplicates at the cost of higher overhead
  - `CLIENT_ACKNOWLEDGE`: the JMS Client explicitly acknowledges messages by calling a message's `acknowledge()` method

Politecnico di Milano

More JMS concepts
Message delivery mode

- It can either be:
  - `NON_PERSISTENT`: does not require logging to stable storage while exposing minimum overhead, a JMS provider's failure can cause a message to be lost
  - `PERSISTENT`: instructs the JMS provider to take extra care to guarantee message delivery
- A JMS provider must deliver a `NON_PERSISTENT` message at most once
- On the contrary, a JMS provider must deliver a `PERSISTENT` message once and only once

Politecnico di Milano

More JMS concepts

JMS message model

- The header part contains information used by both clients and providers to identify and route messages
  - All messages support the same set of header fields
  - Several fields are present, among them:
    - `JMSMessageID` contains a values that uniquely identify a message
    - `JMSCorrelationID` contains an identifier that links this message to another one (it can be used to link a query to a reply)
    - `JMSReplyTo` contains a `Destination` supplied by a client where a reply to the message must be sent
    - `JMSTimestamp` contains the time a message has been handed off to a JMS provider to be sent
    - `JMSPriority` contains the message's priority, ten levels of priority can be defined, from 0 (lowest) to 9 (highest)
- Additional information can be possibly added using the property part
  - It can contain application-specific properties, standard properties defined by JMS (act as optional header fields), or provider-specific properties
  - Properties are `<string,value>` pairs, where `value` can be one of Java built-in types or `String`
  - When a message is received, its properties are in read-only mode
  - The method `getPropertyNames()` returns an `Enumeration` of all the property names
  - The main use of properties is to support customized message selection
- The body part specifies the actual message content

Politecnico di Milano

More JMS concepts
JMS message selectors

- Allow a JMS Client to customize the messages actually delivered, based on the application's interests
- A message selector is a predicate (a `String`) on a message's header and property values, whose syntax is derived from SQL's conditional expression syntax
  - The selector is passed as a parameter when the `Consumer` is created
  - The message is delivered when the selector evaluates to true
  - Type checking must be taken into account
- Example:

```
myMsg.setIntProperty("Property1",2);
myMsg.setStringProperty("Property2","Hi");
myMsg.setFloatProperty("Property3",3.4821);
myMsg.SetStringProperty("Property4","2");
```

Message Selector	Matches?
"Property1 = 2 AND Property3 < 4"	Yes
"Property3 NOT BETWEEN 2 AND 5"	No
"Property2 IN ('Ciao', 'Hallo', 'Hi')"	Yes
"Property5 IS NULL"	Yes
"Property4 > 1"	No