Message Oriented Middleware

Daniel Hagimont

IRIT/ENSEEIHT
2 rue Charles Camichel - BP 7122
31071 TOULOUSE CEDEX 7

Daniel.Hagimont@enseeiht.fr
http://hagimont.perso.enseeiht.fr

This lecture is about messaging services that are provided in the context of Message Oriented Middleware (MOM).

Message based model Introduction

- Client-server model
  - Synchronous calls
  - Appropriate for tightly coupled components
  - Explicit designation of the destination
  - Connection 1-1
- Message model
  - Asynchronous communication
  - Anonymous designation (e.g.: announcement on a newsgroup)
  - Connection 1-N

For the moment, we can consider that the message model consists in programming distributed applications with simple message exchanges.

The message model has fundamentally different properties compared to the client-server model.

The client-server model :

- relies on synchronous calls (with a request and a response, the client being suspended waiting for the response)

- is well suited for tightly coupled components, i.e. the caller depends on the service provided by the callee

- there's an explicit designation of the callee by the caller

- it's a one to one connection

In opposition, the message model :

- relies on asynchronous communications (the sender does not wait for a response)

- there can be a anonymous designation (when you send a message to anybody who may be interested like an announcement on a newsgroup)

- it's can be a one to many connection

Message based model Introduction

- Application example
  - Supervision of equipments in a network
  - E.g. average load on a set of servers
- Client-server solution
  - Periodic invocation
- Message based solution
  - Each equipment notifies state changes
  - Administrators subscribe notifications

We give here an example of application where the message model is better suited. Let's consider the supervision of equipments in a cluster (e.g. the load of the cluster's machines).

A client-server based solution would require a central server performing periodic invocations of all the servers in the cluster.

A message based solution would see each server notify the central server whenever the load changes.

Message based services
... used everyday

- Electronic forums (News)
  - Pull technologies
  - consummers can subscribe to a forum
  - producers can publish information in a forum
  - consummers can login and read the content anytime
- Electronic mail
  - Push technologies
  - mailing lists (multicast - publish/subscribe)
  - consummers can subscribe a mailing list
  - producers can send emails to a mailing list
  - Consummers receive emails without having to perform any specific action

- Asynchronous
- Anonymous
- 1-N

The message model is already used for many applications in daily use.

For instance, electronic forums (news) are relying on the message model. Producers publish (send) information on a forum. Consumers subscribe to a forum and read (pull) the information published on the forums they subscribed.

Another example is electronic mail with mailing lists. A producer can send email to a mailing list and consumers can subscribe mailing lists and the messages sent (push) to these mailing lists are received by those consumers.

In both examples, communication is asynchronous, anonymous and may involve several receivers.

Message based middleware Principles

- Message Passing (communication with messages)
- Message Queuing (communication with persistent message queues)
- Publish/Subscribe (communication with subscriptions)
- Events (communication with callbacks)

Message based middleware were designed to provide developers with a system support for managing messages and programming distributed applications which exchange messages, with the properties presented previously (asynchronous, anonymous, 1-N).

In this context, we distinguish 3 kinds of such messaging service :

- Message passing

- Message queuing

- Publish/subscribe

And one additional service commonly found which is event programming.

Message based middleware
Message passing

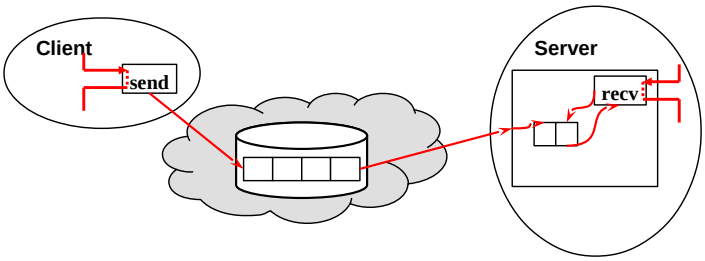
- Communication with message
  - In a classical environment: sockets
  - In a parallel programming environment: PVM, MPI
  - Other environments: ports (e.g. Mach)

Message passing is the simplest service which consists in allowing to send asynchronous messages.

In a classical environment, message passing relies on the socket interface, but it can have other forms, e.g. in an environment devoted to parallel applications (PVM and MPI are parallel environments providing message passing interfaces). Other systems may provides message passing with an interface different from socket (e.g. ports in Mach).

Message based middleware
Message Queuing

- Queue of messages
  - persistent messages (reliability)
- Independence between the emitter and the receiver
  - The receiver is not necessarily active
    - => increased asynchronism
  - Several receivers (anonymous)



Message Queuing is the first advanced service which may be provided by a MOM.

The basic difference with message passing is that message queuing provides message persistence.

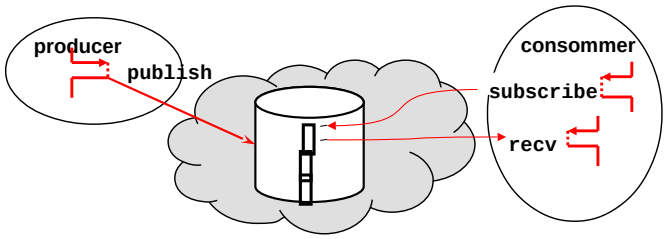
A queue may be allocated and used by clients (producers) or servers (consumers). The queue is managed in the network, meaning that it is not managed in clients neither servers. It is instead managed on machines managed in the middle, i.e. by the message middleware.

Messages are persistent in the sense that we don't require the producer and the consumer to be active at the same time for sending a message (which is the case for message passing). The client may send a message in the queue while the server is inactive (the machine is down). The message will be read by the server at a later time, and may be the client will be inactive at that time.

Another aspect of independence is the fact that a queue may be shared by several producers and consumers. It already provides a sort of anonymous designation.

Message based middleware Publish/Subscribe

- Anonymous designation
  - The receiver subscribes to a topic
    - Subject-based
    - Content-based
  - The producer sends a message to a topic
- Communication 1-N
  - Several receivers may subscribe



The second advanced service is the publish/subscribe (pub/sub) service.

A receiver may subscribe to a topic.

There are generally 2 types of pub/sub system:

- subject-based : topics are predefined subjects (i.e. subjects have to be created by an administrator)

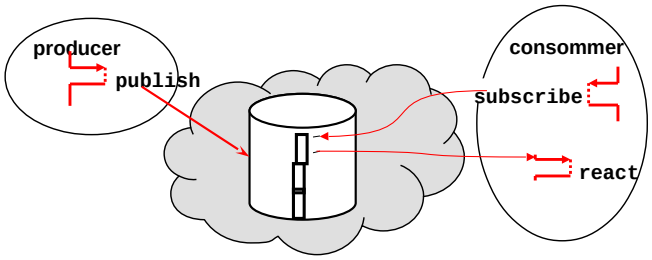
- content-based : topics are filters on the content of messages (e.g. I want to receive messages which include ….)

A producer sends a message to a topic, i.e. either to a given subject or simply with a content. All the receivers who subscribed to the subject, or requested a content which fits with the sent message, will receive a copy of the message.

Here the pub/sub communication service allows message persistence, anonymous designation and multiple receivers.

Message based middleware Events

- Basic concepts: events, reactions (handling associated with event reception)
- Attachement: association between an event type and a reaction



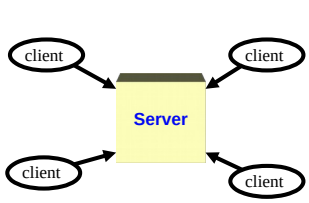
- Exists for all forms of messaging (Message Passing, Message Queuing, Publish/Subscribe)

In order not to have to periodically consult message queues (associated with message queuing or pub/sub) message based middleware often introduces support for event programming.

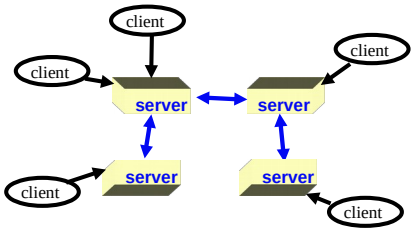
It mainly allows the association between an event (reception of a message) and a reaction (handling program).

Such a facility is available for all forms of communication (message passing, queuing or pub/sub).

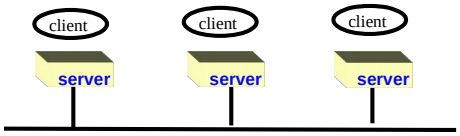
Message based middleware Implementations



Centralized server
(Hub & spoke)



Distributed servers
(Snow flake)



Software bus

Different implementation strategies may be used.

The simplest one is a centralized server remotely used by all clients. This is appropriate for testing, but not for real use as it represents a single-point-of-failure.

Another organization is an interconnection of distributed servers. The interconnection generally depends on the geographic and administrative distribution of clients. The server may implement routing of messages according to the subscriptions from clients.

The last organization is the software bus where all servers know each others. This is generally a strategy used on local (small scale) networks.

Java Message Service

- JMS: Java API defining a uniform interface for accessing messaging systems
  - IBM (WebSphere MQ), Oracle (WebLogic)
  - Apache ActiveMQ, RabbitMQ

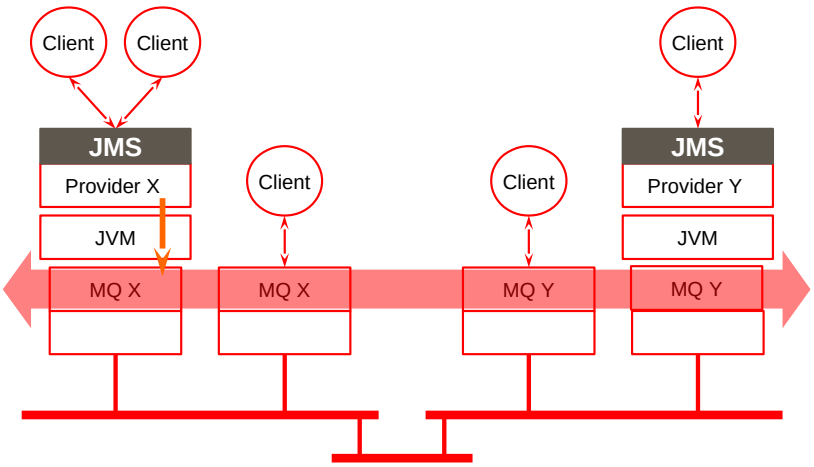
  - Message Queue
  - Publish/Subscribe
  - Event

With the popularity of MOMs, and the development of Java, was proposed a common specification of an API for using a MOM from Java. It should be the same API for all messaging systems (from different providers).

This is JMS for Java Message Service. JMS defines a set of Java interfaces which allows a client to access a messaging system. JMS tries to minimize the concepts to learn and manipulate to use a messaging system, while preserving the diversity of all the existing MOMs.

JMS defines interfaces for managing message Queues and Publish/Subscribe.

JMS: an interface (portability, not Interoperability)



Interoperability : AMQP (Advanced Message Queuing Protocol)

12

It is important to note that JMS is an interface. Since it is implemented by many MOM providers, it implies that if you implement your applications with JMS, it will run on many MOMs (from different providers). So JMS addressed the issue of the portability of applications.

However, JMS does not bring interoperability. The messages emitted by provider X may have a different format from those emitted by provider Y.

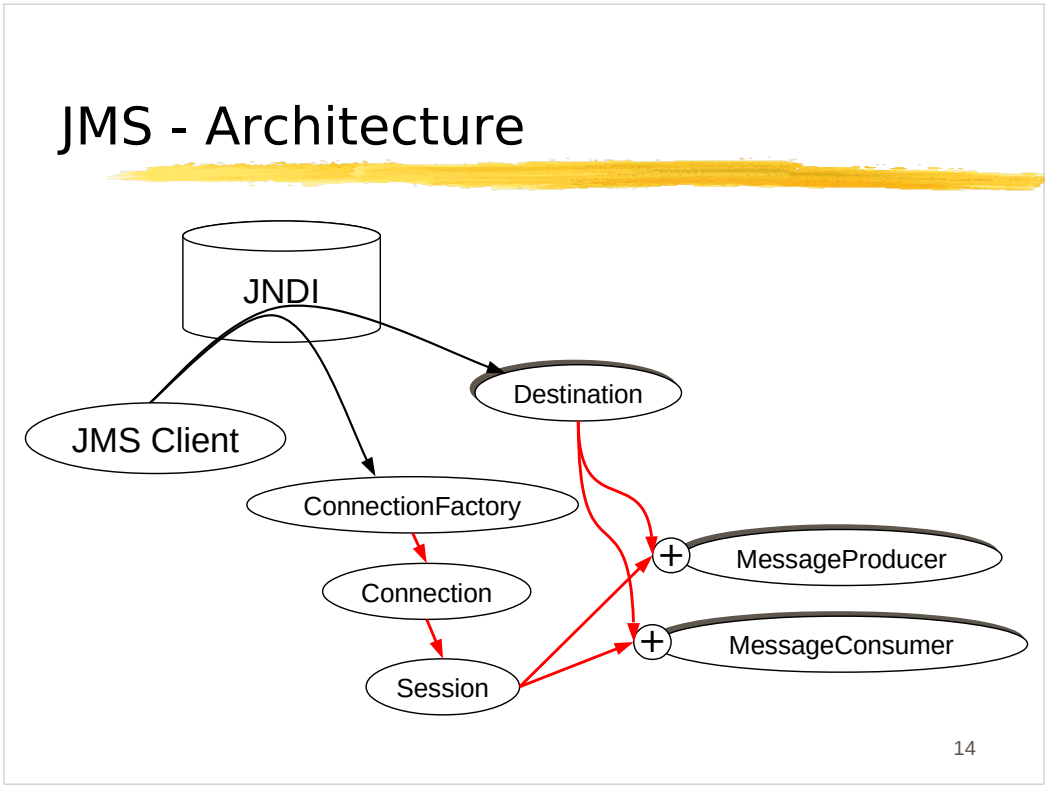
Portability was brought to MOMs with the standardization of AMQP which defines format of exchanged data at the network level.

JMS interface

- *ConnectionFactory*: factory to create a connection with a JMS server
- *Connection*: an active connection with a JMS server
- *Destination*: a location (source or destination)
- *Session*: a single-thread context for emitting or receiving
- *MessageProducer*: an object for emitting in a session
- *MessageConsummer*: an object for receiving in a session

- Implementations of these interface are specific to providers …

JMS may appear complex, but it is rather systematic, and also it had to satisfy all the providers (if the designers wanted all the providers to implement it).



This figure illustrates how these interfaces can be used.

JNDI is the interface of a naming service (such as rmiregistry which is an instance of such a naming service). We assume a JNDI service is available.

A JMS client can obtain from the JNDI service a reference to a ConnectionFactory, which allows to create a Connection (with the JMS server) and then to create a session in this JMS server.

The JMS client can also obtain from the JNDI service a reference to a Destination (an abstract type which can actually refer to a Queue or a Topic).

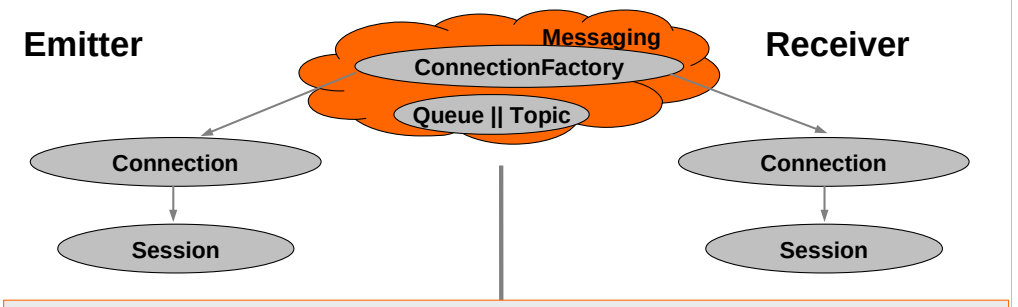
From a session and a destination, we can create a MessageProducer and a MessageConsumer allowing to emit and receive messages.

Interfaces PTP et P/S

	Point-To-Point	Publish/Subscribe
ConnectionFactory	*QueueConnectionFactory*	*TopicConnectionFactory*
Connection	*QueueConnection*	*TopicConnection*
Destination	*Queue*	*Topic*
Session	*QueueSession*	*TopicSession*
MessageProducer	*QueueSender*	*TopicPublisher*
MessageConsumer	*QueueReceiver*	*TopicSubscriber*

The interfaces described previously are abstract and are specialized according to the use of message queuing (Point-To-Point) or Publish/Subscribe

JMS - initialization



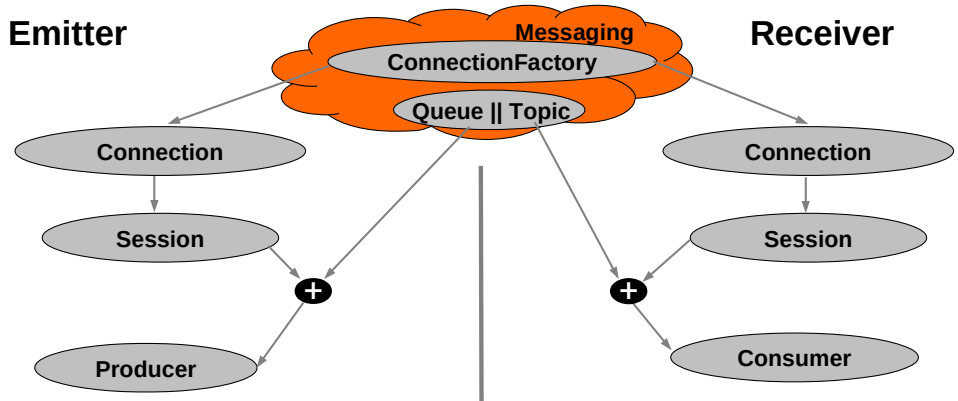

```
ConnectionFactory connectionFactory =
        new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT_BROKER_URL);
Connection connection = connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

Destination destination = session.createQueue("myQueue");
Destination destination = session.createTopic("MyTopic");
```


Here is the code which is common to the emitter and receiver for initializing the connection with the JMS server and obtaining a destination (one of the 2 lines should be chosen, queue or topic ...).

Notice that with ActiveMQ (this is not JMS standard, but specific to ActiveMQ), createQueue() and createTopic() take a URL as parameter, so the same URL used by 2 clients implies the same destination. These ActiveMQ methods correspond to the query of JNDI.

In ActiveMQ, destinations are instantiated at first use.

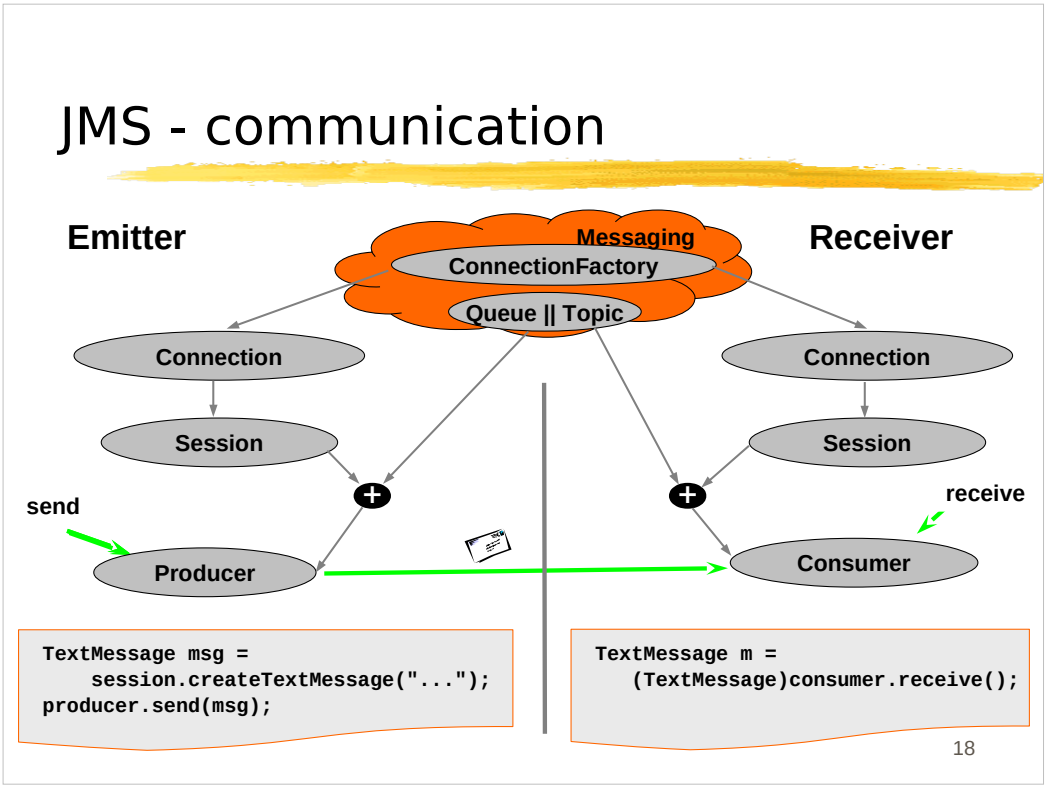
JMS – producer / consumer




```
MessageProducer producer =
      session.createProducer(destination);
```

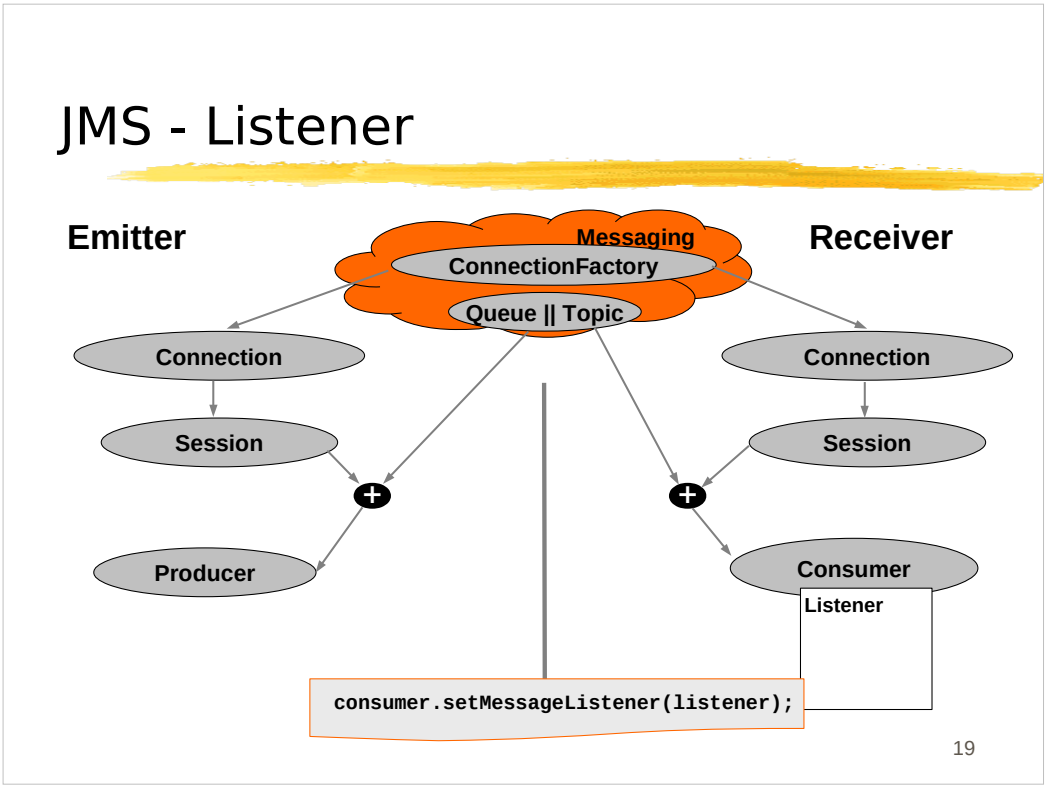
```
MessageConsumer consumer =
    session.createConsumer(destination);
```


Here, with a session and a destination, we create a producer (left) and a consumer (right).

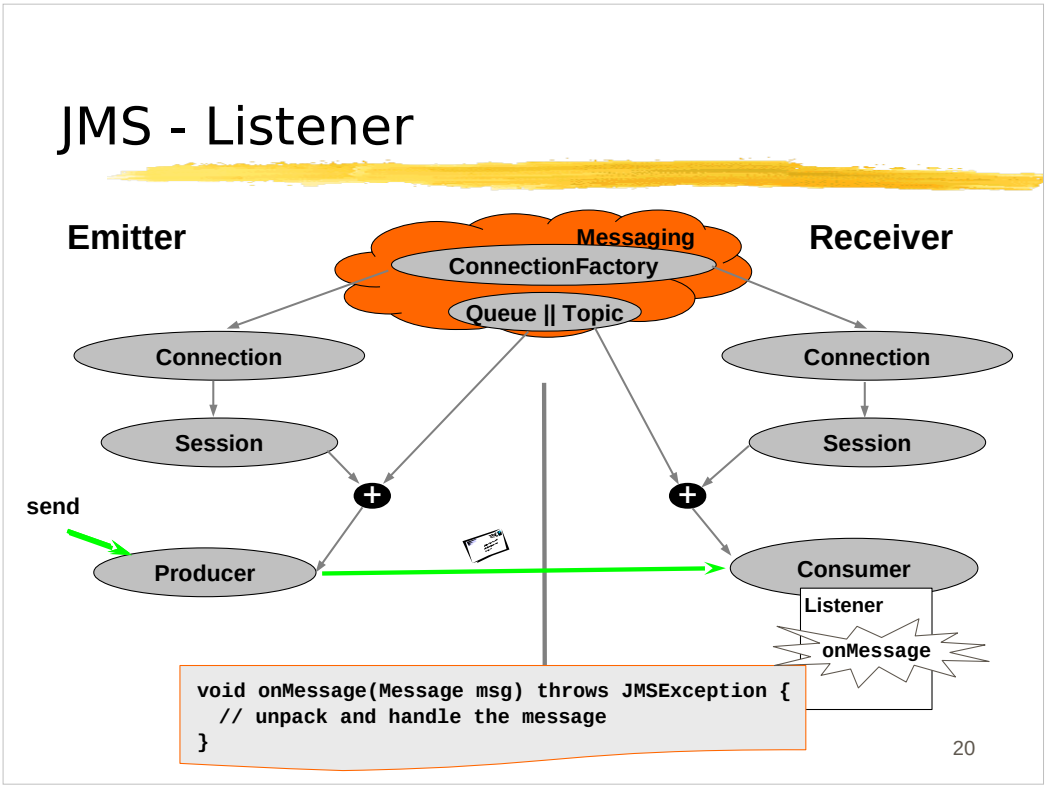


On the left, we can send a message (here a TextMessage) with a producer.

On the right, we can receive a message (here a TextMessage) with a consumer.



On the consumer side, we can associate a reaction to a message reception event.



The registered listener is an instance of a class which implements the onMessage() reaction method.

JMS – messages

- TextMessage (a character string)

```
String data;
TextMessage message = session.createTextMessage();
message.setText(data);
```

```
String data;
data = message.getText();
```

- BytesMessage (bytes array)

```
byte[] data;
BytesMessage message = session.createByteMessage();
message.writeBytes(data);
```

```
byte[] data;
int length;
length = message.readBytes(data);
```


In JMS, messages are types. We can allocate :

- TextMessage (like String)
- BytesMessage (like byte[]).

JMS – messages

- MapMessage (sequence of key-value pair)
  - A value is a primitive type

```
MapMessage message = session.createMapMessage();

message.setString("Name", "…");
message.setDouble("Value", doubleValue);
message.setLong("Time", longValue);
```

```
String name = message.getString("Name");
double value = message.getDouble("Value");
long time = message.getLong("Time");
```

JMS – messages

- StreamMessage (sequence of values)
  - A value is a primitive type
  - Reading should respect the sequence order to writing

```
StreamMessage message = session.createStreamMessage();

message.writeString("…");
message.writeDouble(doubleValue);
message.writeLong(longValue);
```

```
String name = message.readString();
double value = message.readDouble();
long time = message.readLong();
```

JMS – messages

- ObjectMessage (serialized objects)

```
ObjectMessage message = session.createObjectMessage();

message.setObject(obj);
```

```
obj = message.getObject();
```

Conclusions

- Communication with messages
  - Simple programming model
  - Many extensions, variants …
    - Message software bus, actors models, multi-agent systems …
  - Widely used for interconnecting tools, existing, developed independently
- However… it is only apparently simple
  - Propagation and report of errors
  - Development tools

Even if the message model may seem to be very simple and primitive, many extensions and variants exist.

MOMs are widely used for interconnecting tools, integrating tools that were developed independently.

Notice that simplicity is only apparent, as asynchronism makes it difficult to debug or to have deterministic behaviors.