Unit 8. Communication

Concurrency and Distributed Systems

Teaching Unit Objectives

- To Understand the mechanisms of communication in a distributed system: *message-based* communication.
  - To distinguish the main features of message-based communication mechanisms.
  - To Learn the operation of a typical communication mechanism
    - ROI (Remote Object Invocation) Communication Mechanism
    - Java RMI (Remote Method Invocation) Communication Mechanism
    - RESTful Web Services Communication Mechanism
    - Java Message Service (JMS) Communication Mechanism

Content

- Features of communication mechanisms
- Remote Object Invocation (ROI)
- Web Services
- Message Oriented Middlewares

Content

- Features of communication mechanisms
  - Usage
  - Structure and content of messages
  - Addressing
  - Synchrony
  - Persistence
- Remote Object Invocation (ROI)
- Web Services
- Message Oriented Middlewares

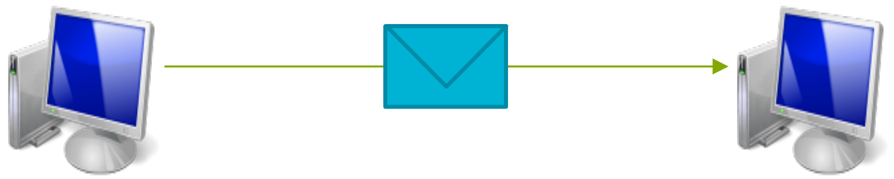
Features of communication mechanisms

- <u>Communication mechanisms</u>
  - They allow communication between processes running on different computers
  - Offered by the communication middlewares
  - Features:
    - Usage
    - Structure and content of messages
    - Addressing
    - Synchrony
    - Persistence

Features of communication mechanisms

- **Usage:**

A. Using basic communication primitives
  - **Send** and **Receive** operations
  - Examples: Sockets, message queues



Send (message) Receive(message)

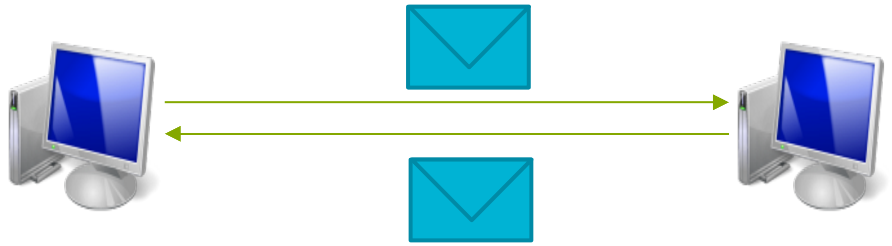
Features of communication mechanisms

- **Usage:**

B. Through constructions of the programming language

- Higher level of abstraction
- Sending and receiving messages is transparent to programmer
- Examples : Remote Procedure Call (RPC), Remote Object Invocation (ROI)

```
O1.method1 (arg0, arg1)          O1
                                 method1(arg0, arg1) {
                                     …
                                 }
```



Features of communication mechanisms

- **Message structure**

A. Non structured, only content
  - Free-form content
  - Examples: sockets

B. Structured in header + content
  - The header is a set of fields generally extensible
  - Free-form content
  - Examples: message queues

C. Structured transparent to programmer
  - Determined by the communication middleware
  - Examples: RPC/ROI

Features of communication mechanisms

- **Message Content**

A. Bytes
  - **Advantages**: efficient and compact
  - **Disadvantages**: difficult to process, binary representation is not equal among architectures and programming languages

B. Text
  - **Advantages**: independent of architecture and programming language
  - **Disadvantages**: less efficient
  - A language with high availability of libraries for its processing is normally used. Examples:
    - Extensible Markup Language (XML)
    - JavaScript Object Notation (JSON)

Features of communication mechanisms

- **Addressing**

A. Direct addressing:

- The sender computer sends messages **directly** to the receiver computer.
- Examples: sockets, web services, RPC/ROI

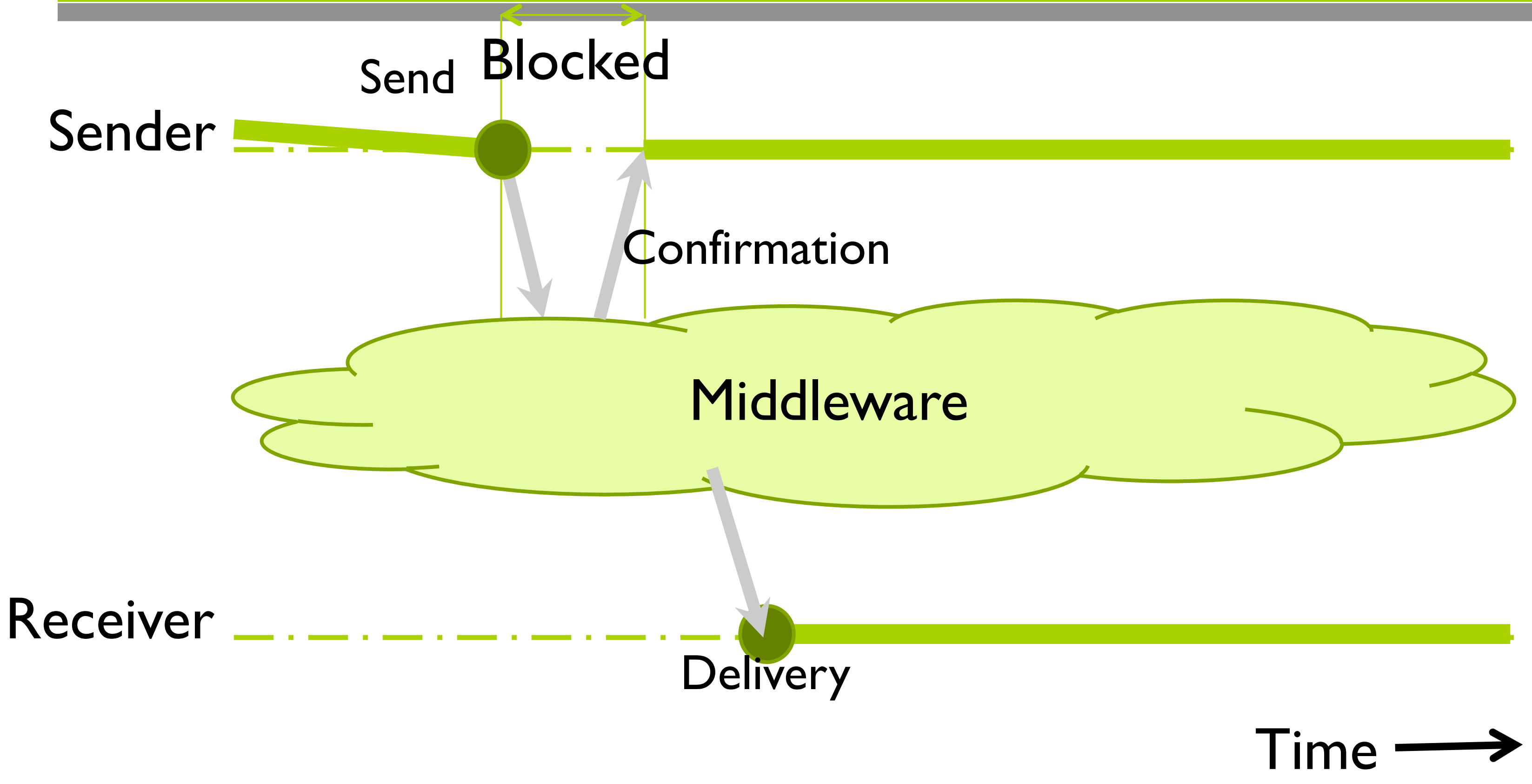
B. Indirect addressing:

- The sender computer sends messages to an **intermediate** (broker), who is in charge of sending them to the receiver computer.
- Examples: message queues

Features of communication mechanisms

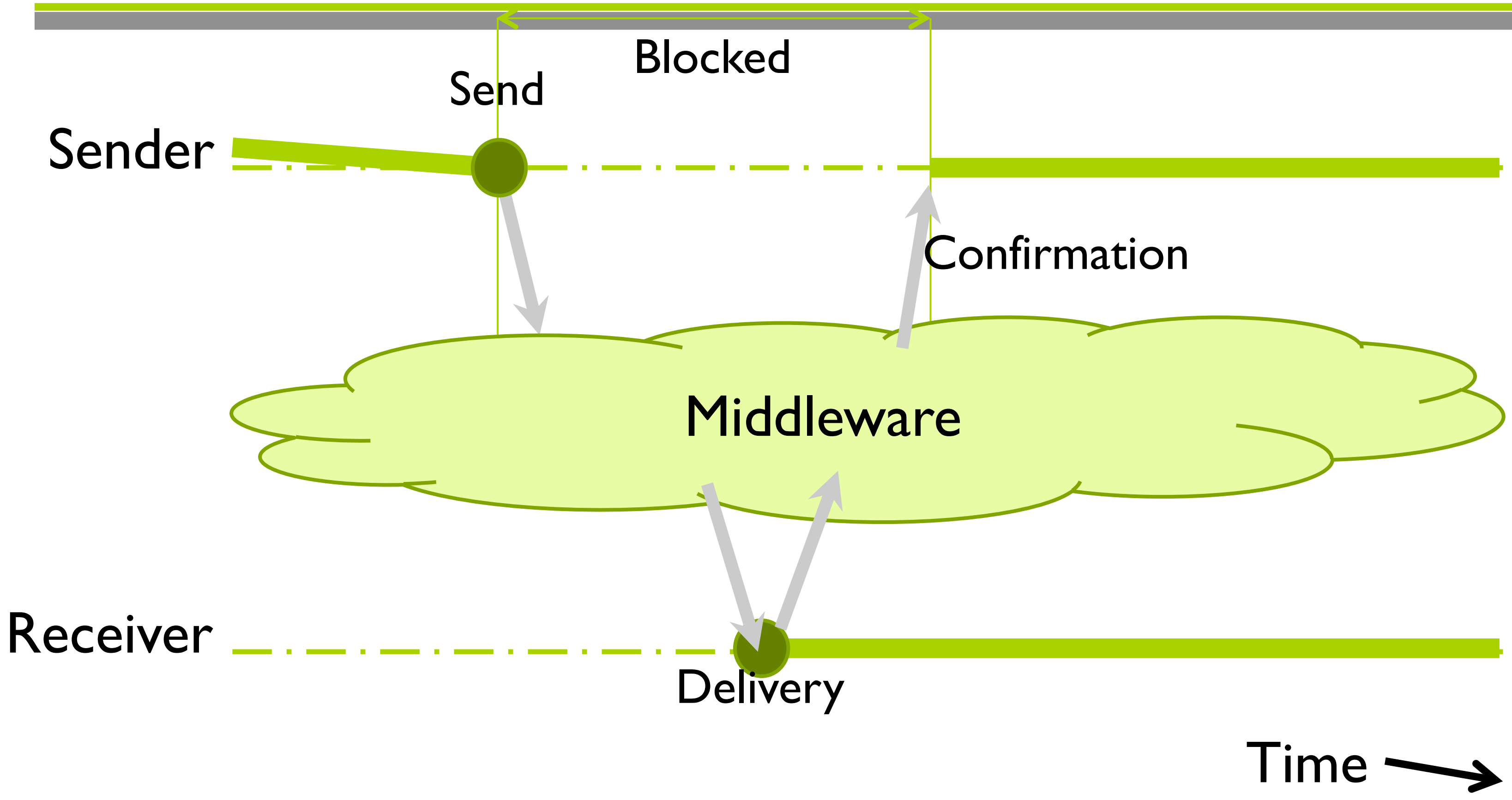
- **Synchrony**
  - A. Asynchronous communication: The middleware responds to the sender with the confirmation, after storing the message in its buffers
  - B. Synchronous communication (delivery): The middleware responds when the receiver has confirmed the successful delivery of the message
  - C. Synchronous communication (answer): The middleware responds when the receiver has notified to have processed the message

Aynchronous communication



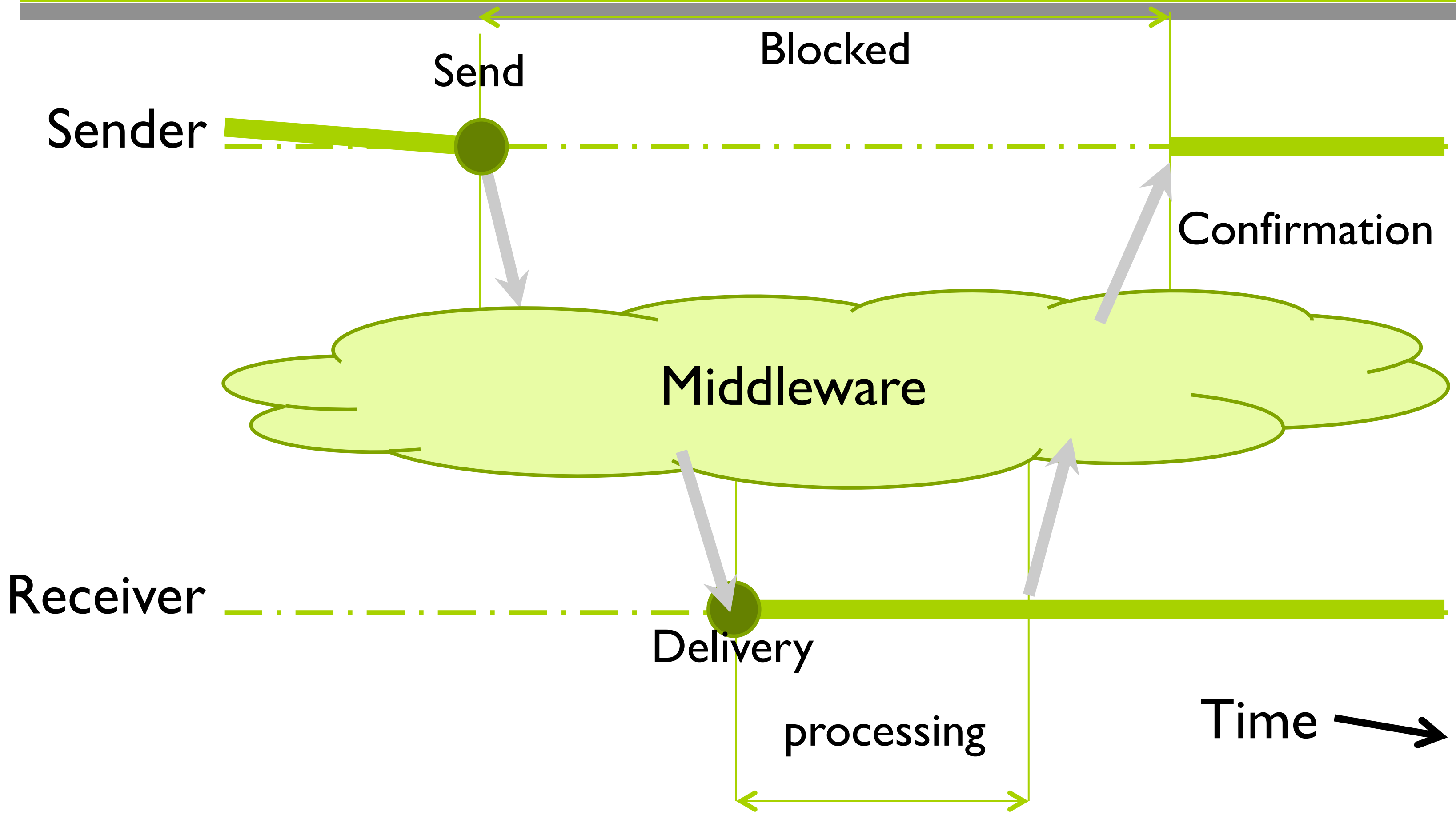
- The middleware responds to the sender with the confirmation, after storing the message in its buffers
- Examples: sockets UDP, message queues

Synchronous Communication (delivery)



- The middleware responds when the receiver has confirmed the successful delivery of the message
- Examples: TCP sockets , REST web services

Synchronous Communication (answer)

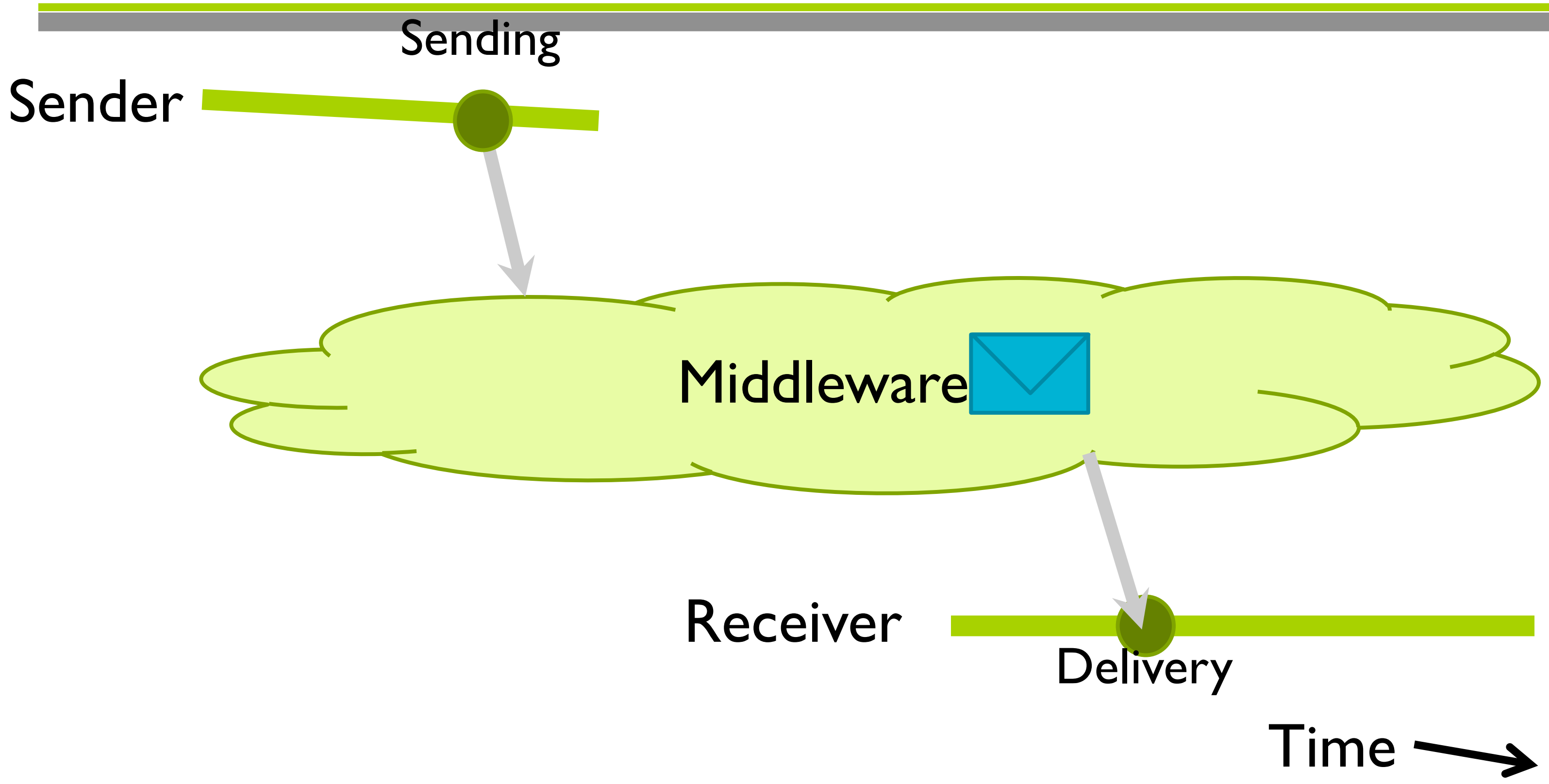


- The middleware responds to the sender when the receiver has notified to have processed the message
- Examples: RPC,ROI, SOAP web services

Features of communication mechanisms

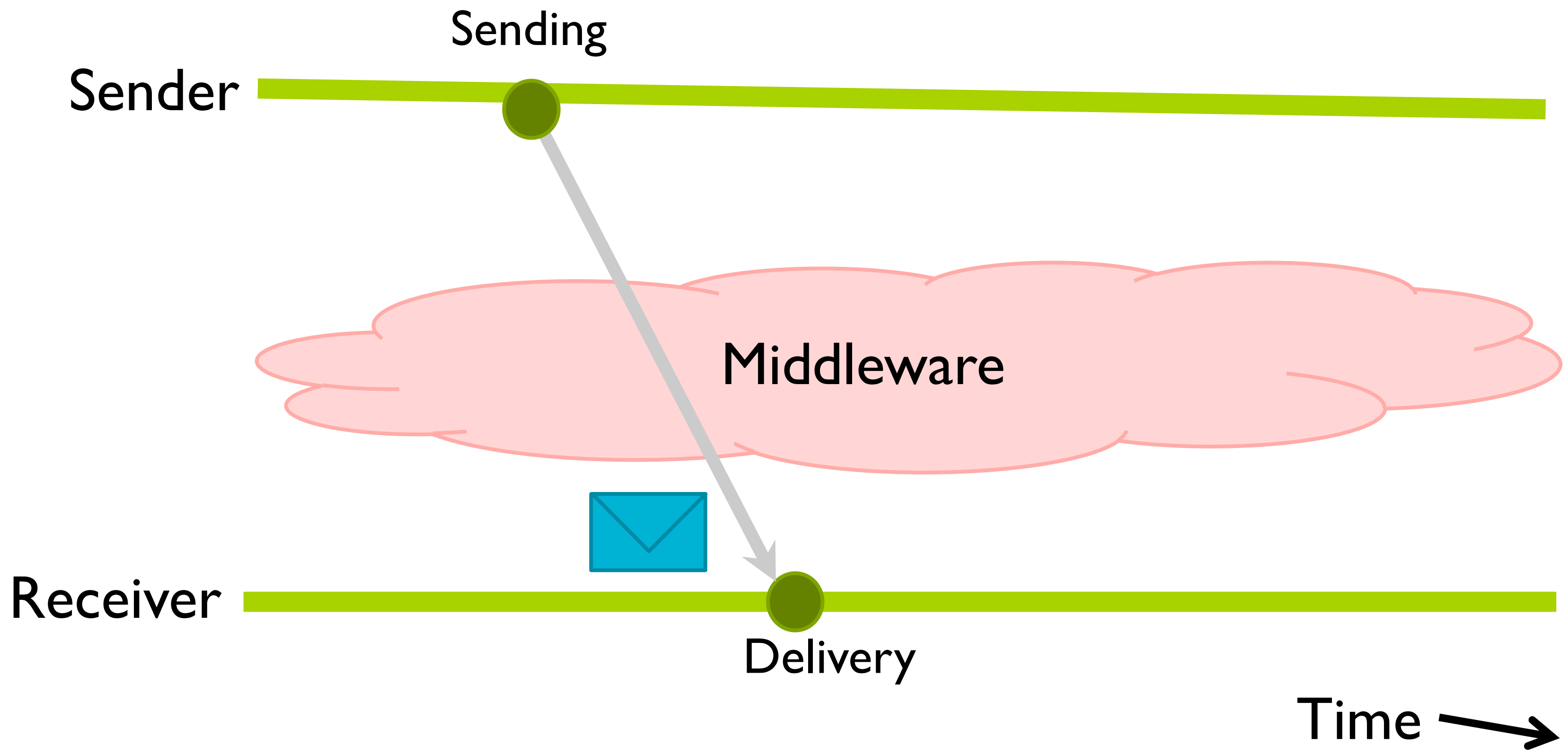
- **Persistence**
  - A. Persistent communication: The middleware can store delivery pending messages
  - B. Non-persistent communication: The middleware is not able to keep the messages to be transmitted

Persistent communication



- The middleware can store delivery pending messages
- Receiver does not need to be running when sender sends the message
- Sender can finish its executing before the message is delivered
- Examples: message queues

Non-persistent communication



- The middleware is not able to keep the messages to be transmitted
- Sender and receiver must be active to let messages be transmitted
- Examples: sockets, RPC/ROI, web services

Features of communication mechanisms

- These previous features are normally useful to categorize and understand the different communication mechanisms
  - However, reality is complex and diverse and there are cases of communication mechanisms in which a precise characterization is not possible.

Content

Remote Object Concept

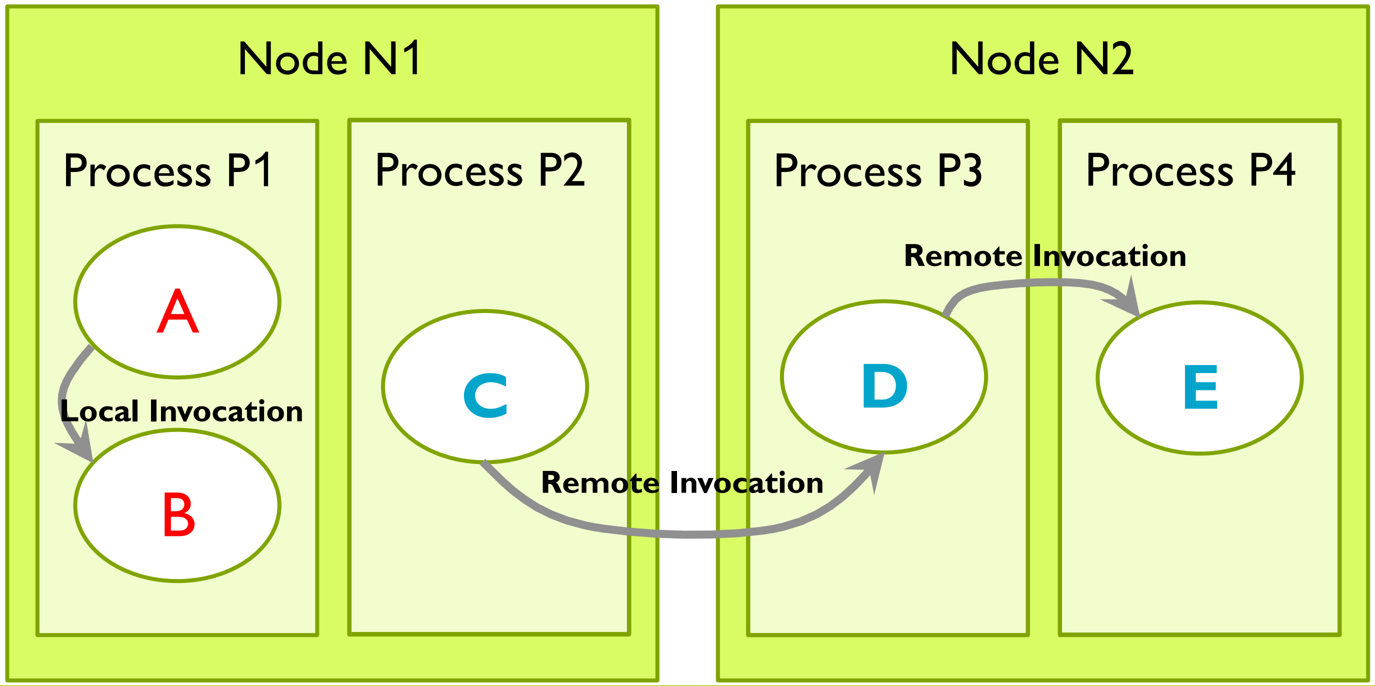
- **Remote object**: an object that can receive invocations from other address spaces.
  - Any remote object is instantiated in a server and can answer to invocations from both local or remote clients
- Applications are organized as a dynamic collection of objects that might be in different nodes
  - Both applications and objects can invoke methods of other objects in a remote way.
  - We assume here that any object is totally hold in a single node
    - Although there are distributed object models that allow dividing objects in parts that are hold in different nodes (i.e. fragmented objects)

Advantages of the Distributed Object Model

- To make profit of expressivity, abstraction capacity and flexibility of the **OO paradigm**
- **Location** transparency:
  - The invocation syntax of the methods of an object does not depend on the address space where the object is located.
  - You can locate objects taking into account different criteria: access locality, administrative restrictions, security, ….
- You can **reuse** legacy applications "encapsulating" them into objects (by using the "Wrapper" design pattern)
- **Scalability**: objects can be distributed through the network, taking into account the current demand.

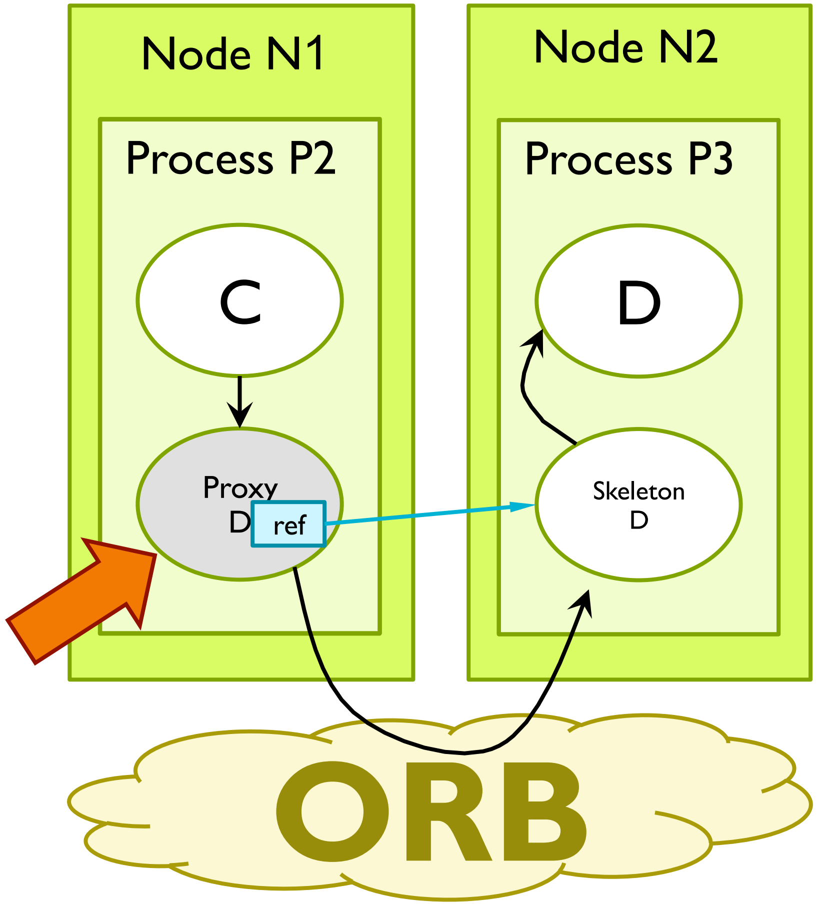
Types of invocations

- **Local invocation**: both invoker and invoked objects are located in the same process (e.g. **A and B objects**)
- **Remote invocation (*ROI – Remote Object Invocation*)**: invoker and invoked objects are in different processes, which can be inside the same node (e.g. **D and E** objects) or in different nodes (e.g. **C and D**)



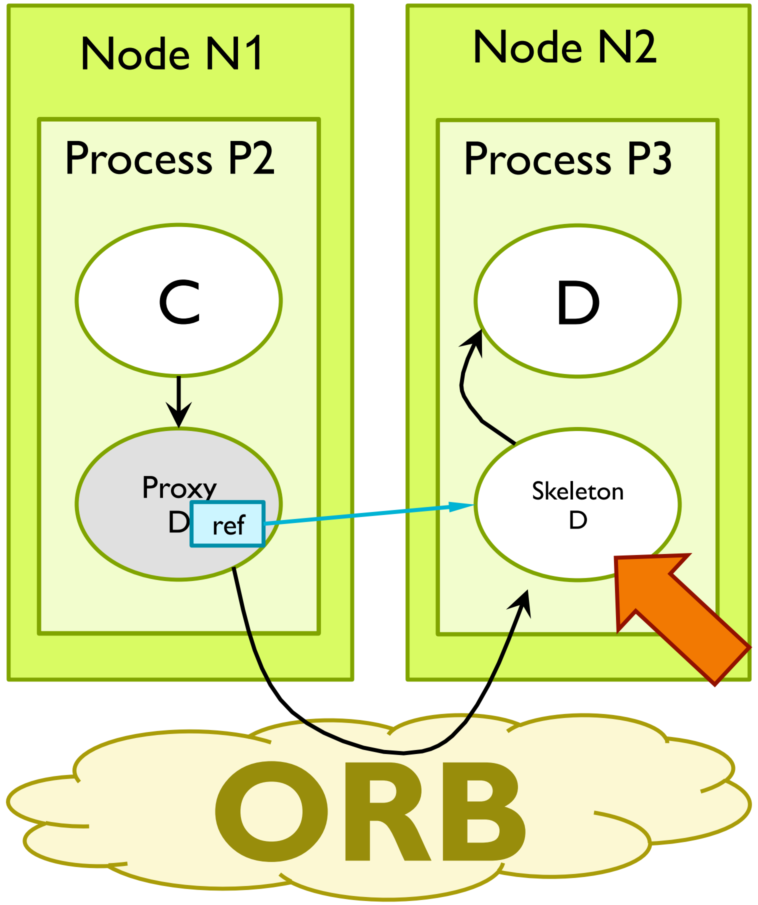
Elements involved in an ROI

- **Proxy**
  - Provides the **same interface** as the remote object
  - Contains a **reference** to the remote object
    - Provides access to the remote object and its interface
  - It is created at runtime when the remote object is accessed for the first time



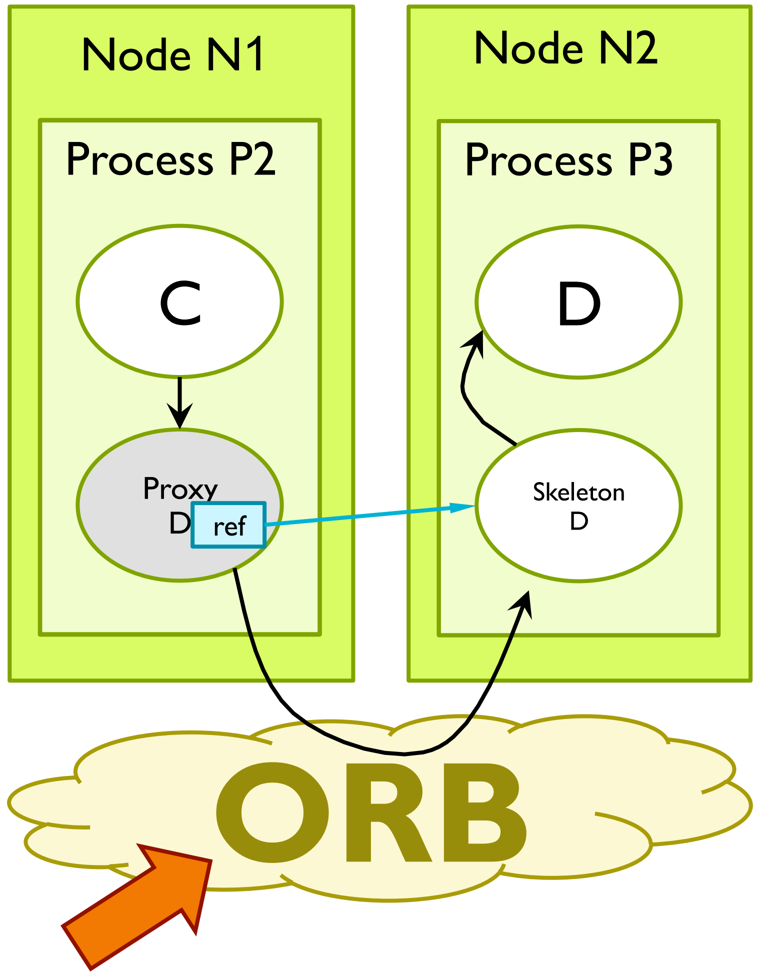
Elements involved in an ROI

- **Skeleton**
  - **Receives requests** from clients
  - Performs the **actual calls** to the remote object methods
  - It is created at run-time when the remote object is created



Elements involved in an ROI

- **Object Request Broker (ORB)**
  - Main component of an object-oriented middleware
  - It is in charge of:
    - **Indentifying and locating** the objects
    - **Carrying out the remote invocations** from proxies to skeletons
    - **Managing the object life-cycle** (creation, registry, activation and deletion)

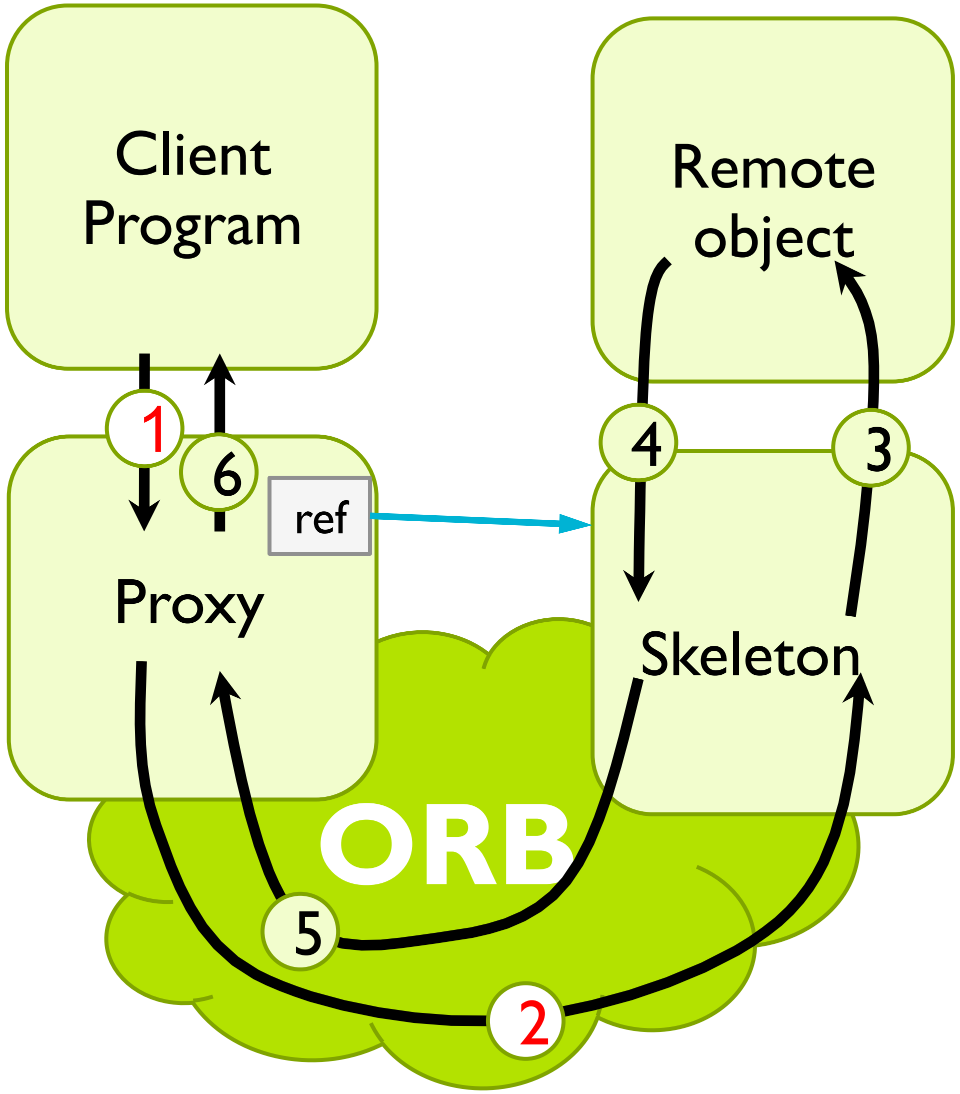


Steps of a Remote Object Invocation

Steps 1 and 2

1. The client process invokes the method on the local proxy related to the remote object

2. The proxy *marshalls* the arguments and, using the object reference, calls the ORB. The ORB manages the invocation, making the message arrive to the skeleton.

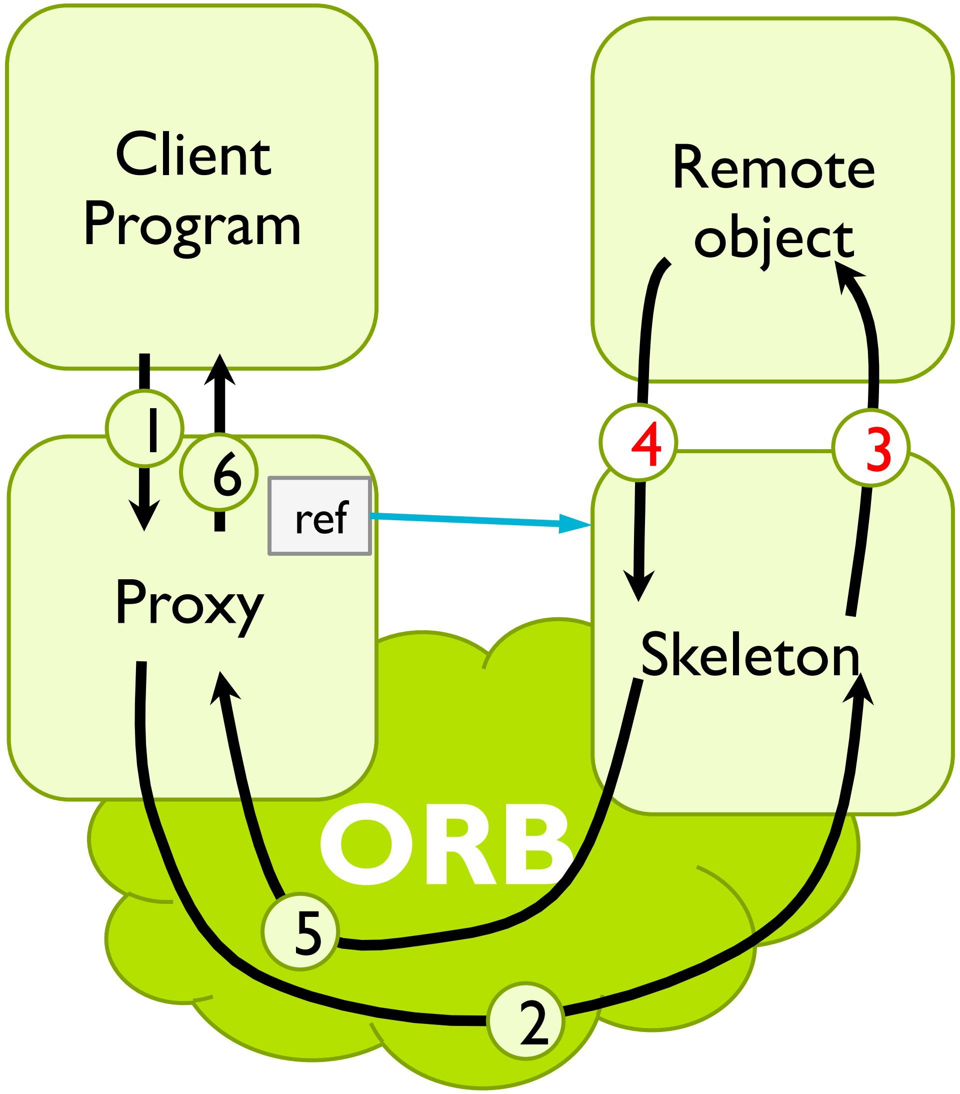


Steps of a Remote Object Invocation

Steps 3 and 4

3. The skeleton *unmarshalls* the arguments, sends the invocation to the requested method, and waits until the method completes.

4. The invoked method completes and the skeleton is unlocked

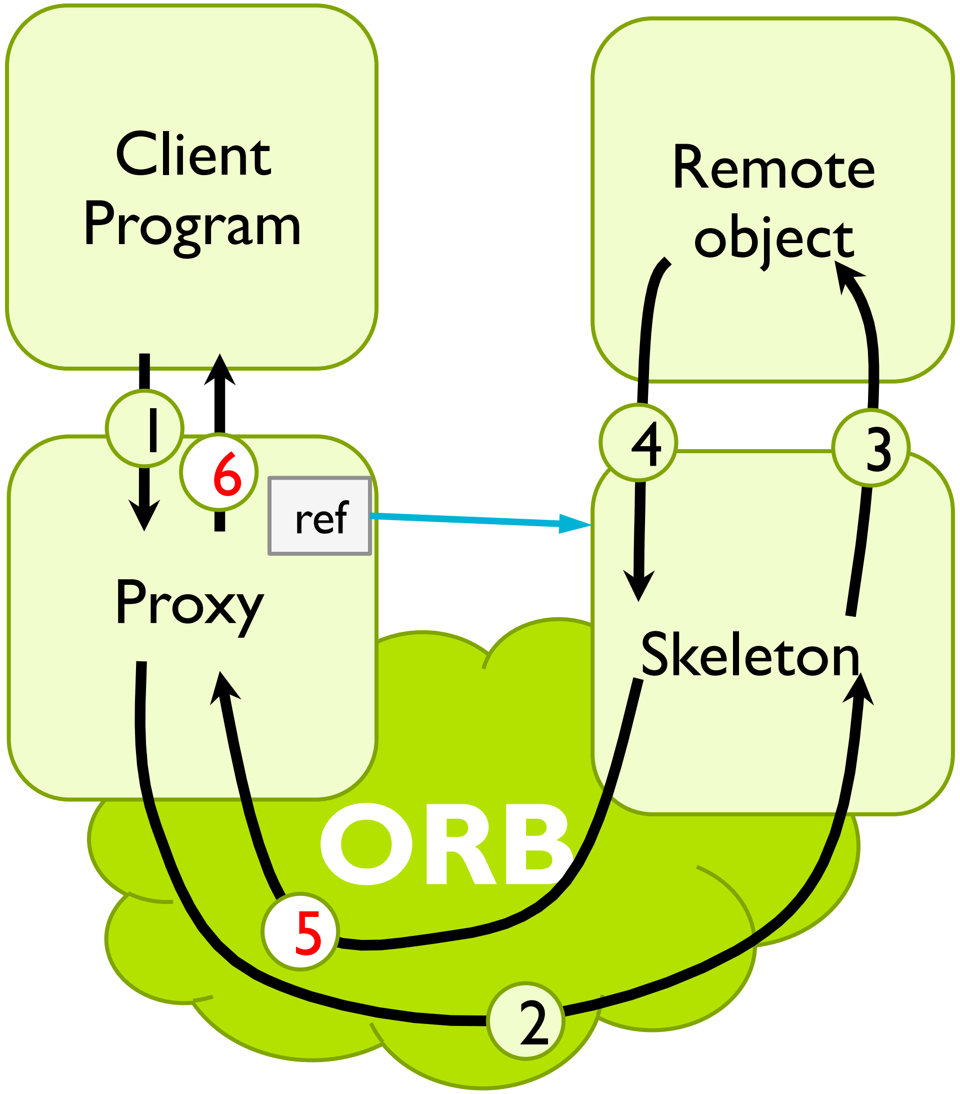


Steps of a Remote Object Invocation

Steps 5 and 6

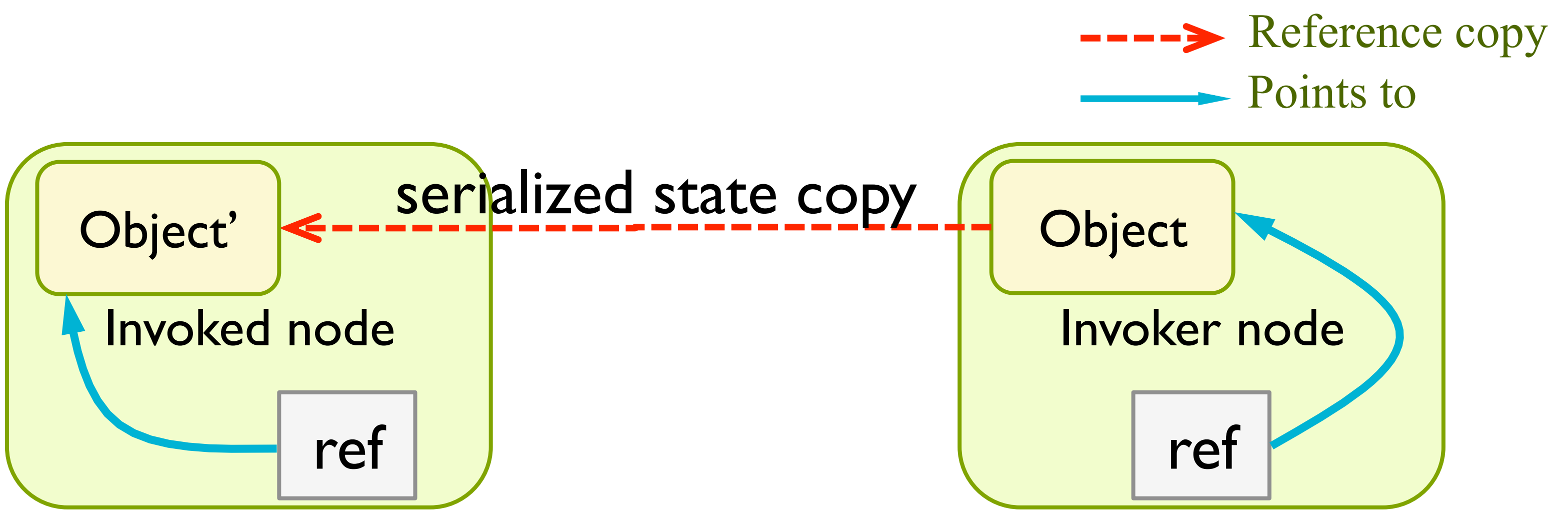
5. The skeleton marshalls the results and calls the ORB who makes the message arrive to the proxy

6. The proxy *unmarshalls* the results and returns them to the client process.



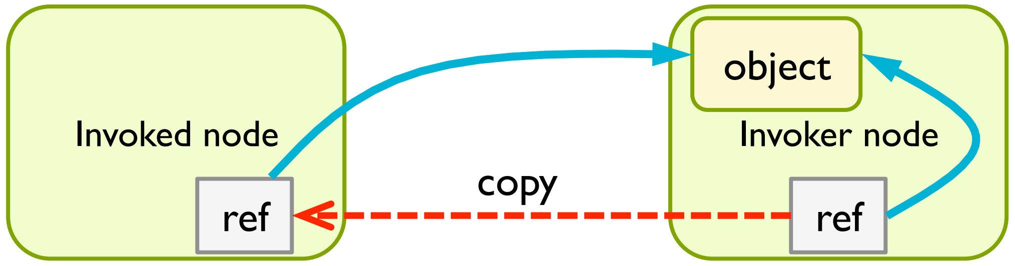
Passing objects as arguments in invocations

- **Passing by value:**
  - The state of the "original object" is packaged, by means of a process called **serialization**
  - The serialized object is transmitted to the destination node, where it is used to create a new copy of the original object
  - Both objects (original and copy) evolve separately

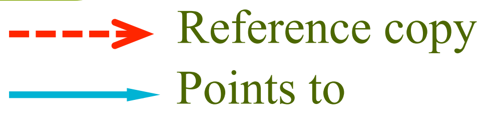


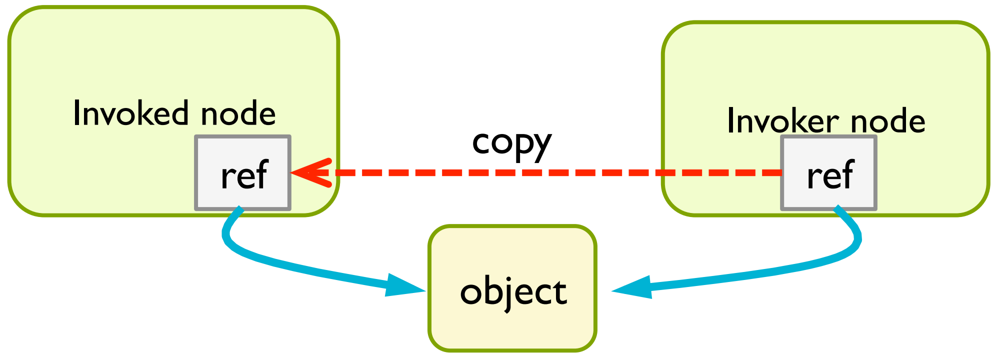
Passing objects as arguments in invocations

- **Passing by reference:**
  - In order to pass an object by reference it's enough to copy the reference from the invoker to the invoked node
  - It does not matter that the object belongs to the invoker node...



  - … or to a third node



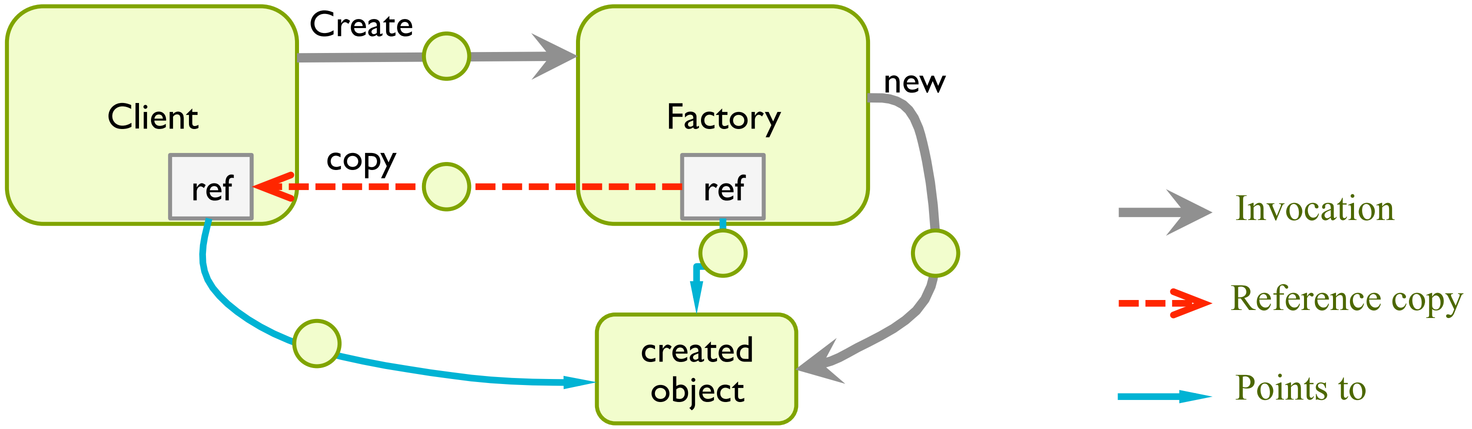


Creation of objects

- In ROI, object creation (and its registry in the ORB) can be done by two different ways:
  - By initiative of the client
    - The client requests a factory to create the object.
    - A **factory** is a server object that creates objects of a specific type
  - By initiative of the server
    - A server process creates the object and registers it in the ORB

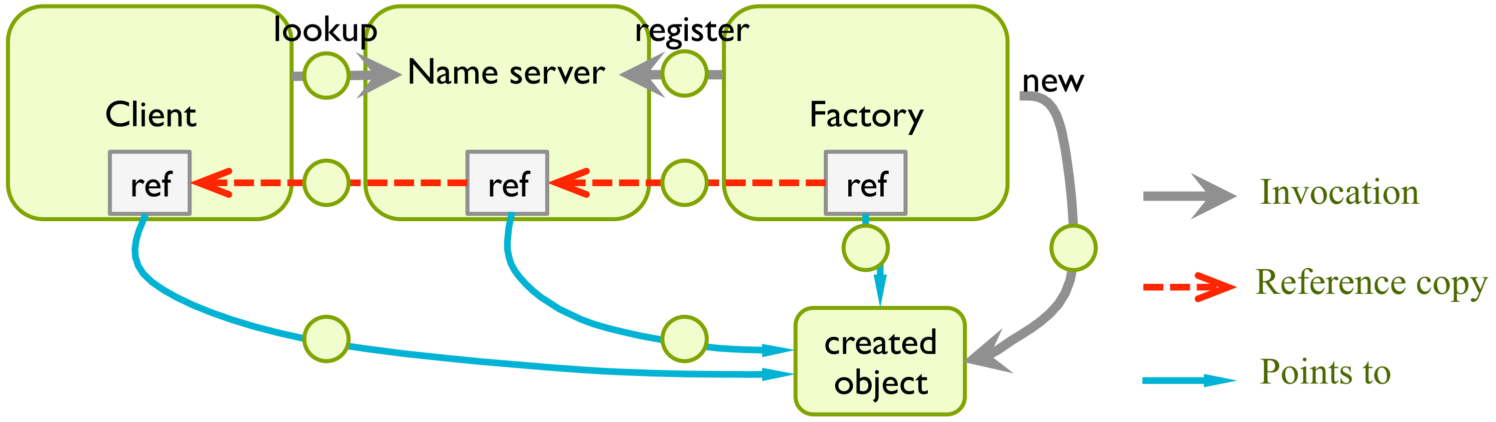
Creation of objects: by initiative of the client

1. The client requests a ***factory*** (a server that creates objects of a specific type) to create the object
   We assume that the client has a proxy to invoke this factory. And there is a method that will return a reference to the object to be created
2. The factory creates the requested object and registers it in the ORB. As a result, it obtains a reference to the object and a skeleton.
3. The factory returns a copy of this reference to the client



Creation of objects: by initiative of the server

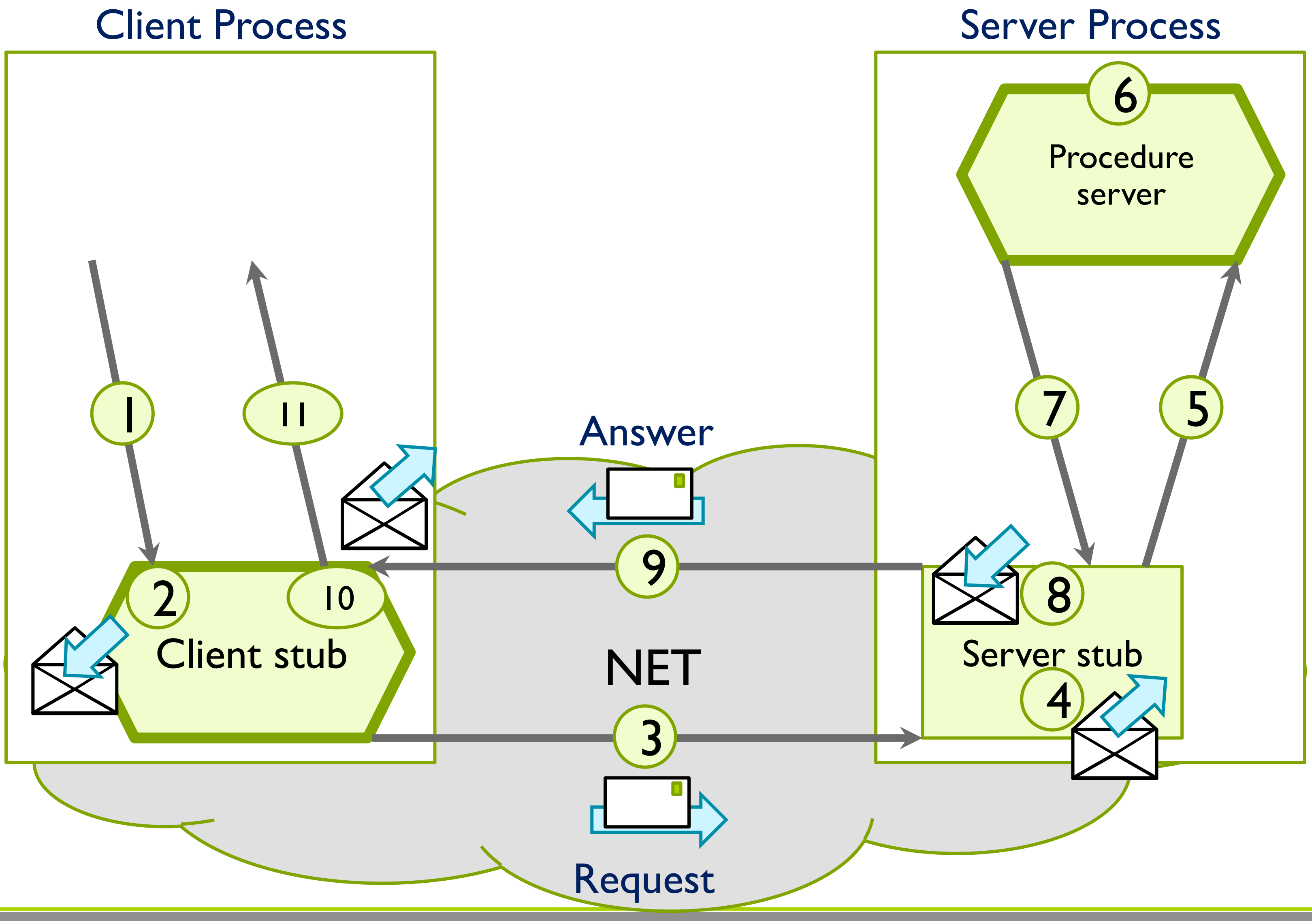
1. A process, which will become the server, creates the object and registers it in the ORB, who creates its first reference
2. The server process uses the reference to register it in a **name server**, giving a specific text string as name.
3. Any other process that knows the name used for registering the object can contact the name server and obtain a reference of this object.



Remote Procedure Call (RPC)

- RPC is the precursor of ROI
- It has the same goals: call a "remote procedure or function" similarly as being a local call.
- It doesn't use objects, but procedures.
- The remote node offers a catalogue of procedures.
- These procedures can be called in a transparent way (as it were local) from the client nodes.
  - This mechanism is implemented by the ***client stub*** (equivalent to a *proxy* of ROI) and the ***server stub*** (equivalent to a *skelenton* of ROI)

Remote Procedure Call (RPC)



Remote Procedure Call (RPC)

- Steps:
  1. Invocation to local procedure
  2. Marshalling of input parameters into a message
  3. Send the message to server and wait for the answer
  4. Unmarshalling of the message and extract the input parameters
  5. Call the procedure
  6. Execute the procedure
  7. Control return to the server *stub*
  8. Marshalling of output parameters and result into a message
  9. Send the answer message
  10. Unmarshalling the message and extract the output parameters and result
  11. Return the control to the code that invoked the local procedure

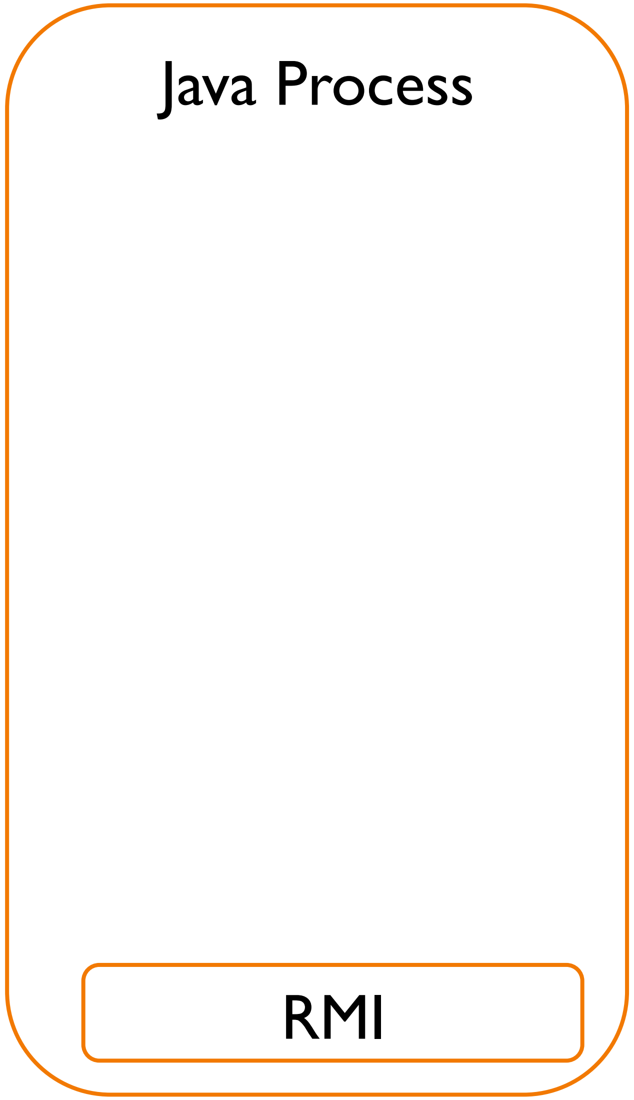
Content

What is Java RMI?

- **Java RMI** (*Remote Method Invocation*) is an **object-oriented** communication **middleware** that provides a solution for a specific OO language (Java) which gives support to portability
  - It is not multi-language, but it is **multi-platform**
  - It allows **invoking** Java object methods of another JVM, and **passing Java objects** as arguments when invoking these methods.

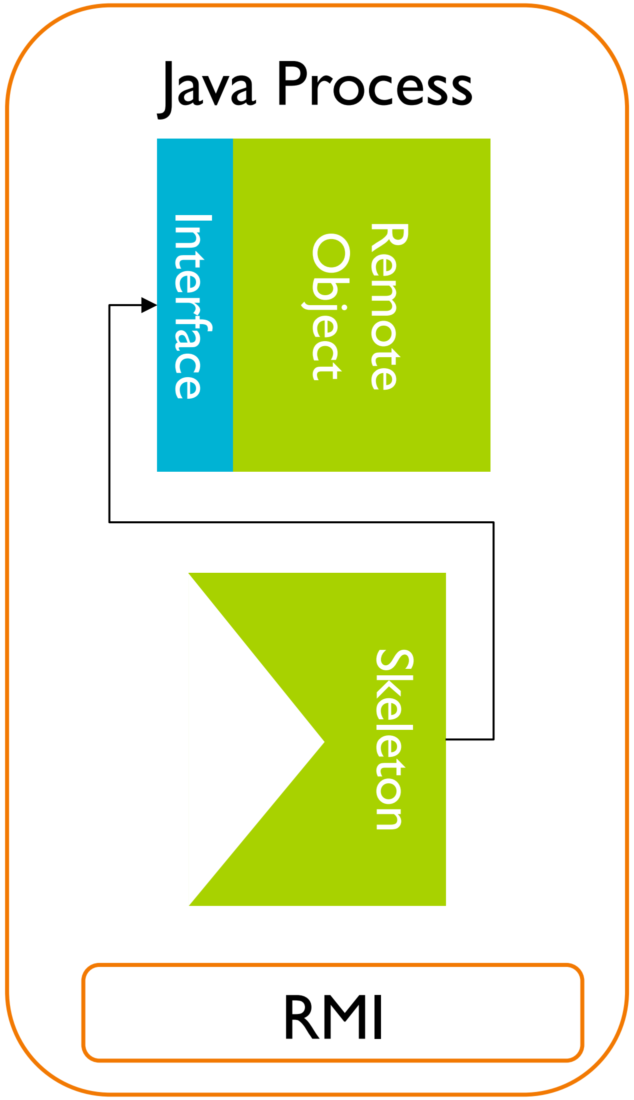
What is Java RMI?

- The RMI component is automatically incorporated into a Java process when its API is used
  - It listens to requests that arrive in a TCP port

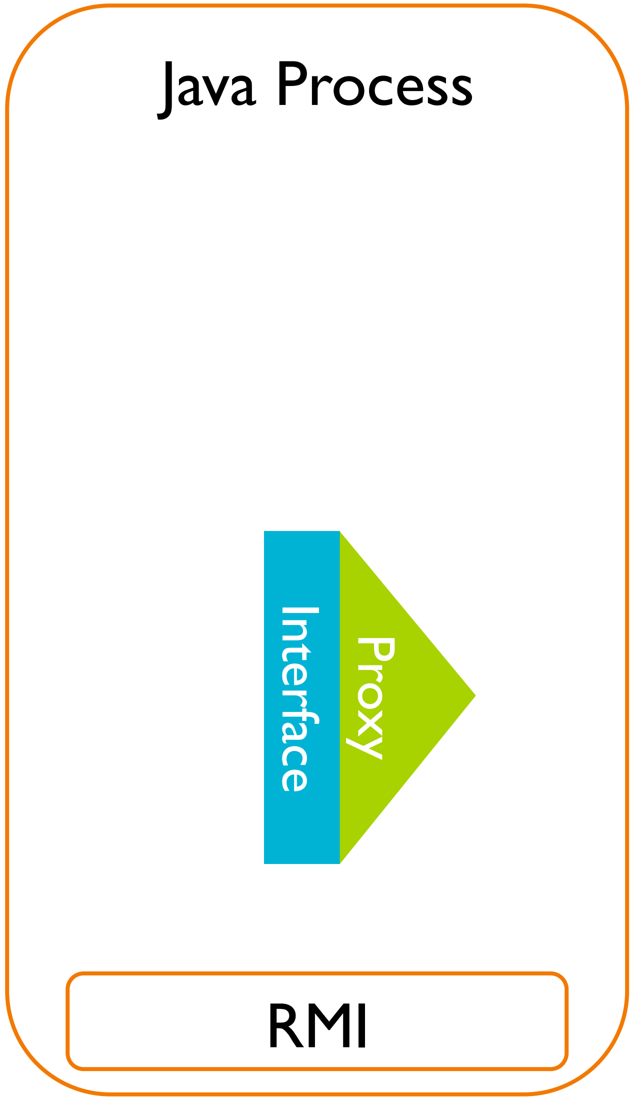


What is Java RMI?

- An object is invokable remotely
  - if you implement an interface that extends the Remote interface
- For each of these remote objects, RMI dynamically creates an object (not accessible by the user) called a skeleton



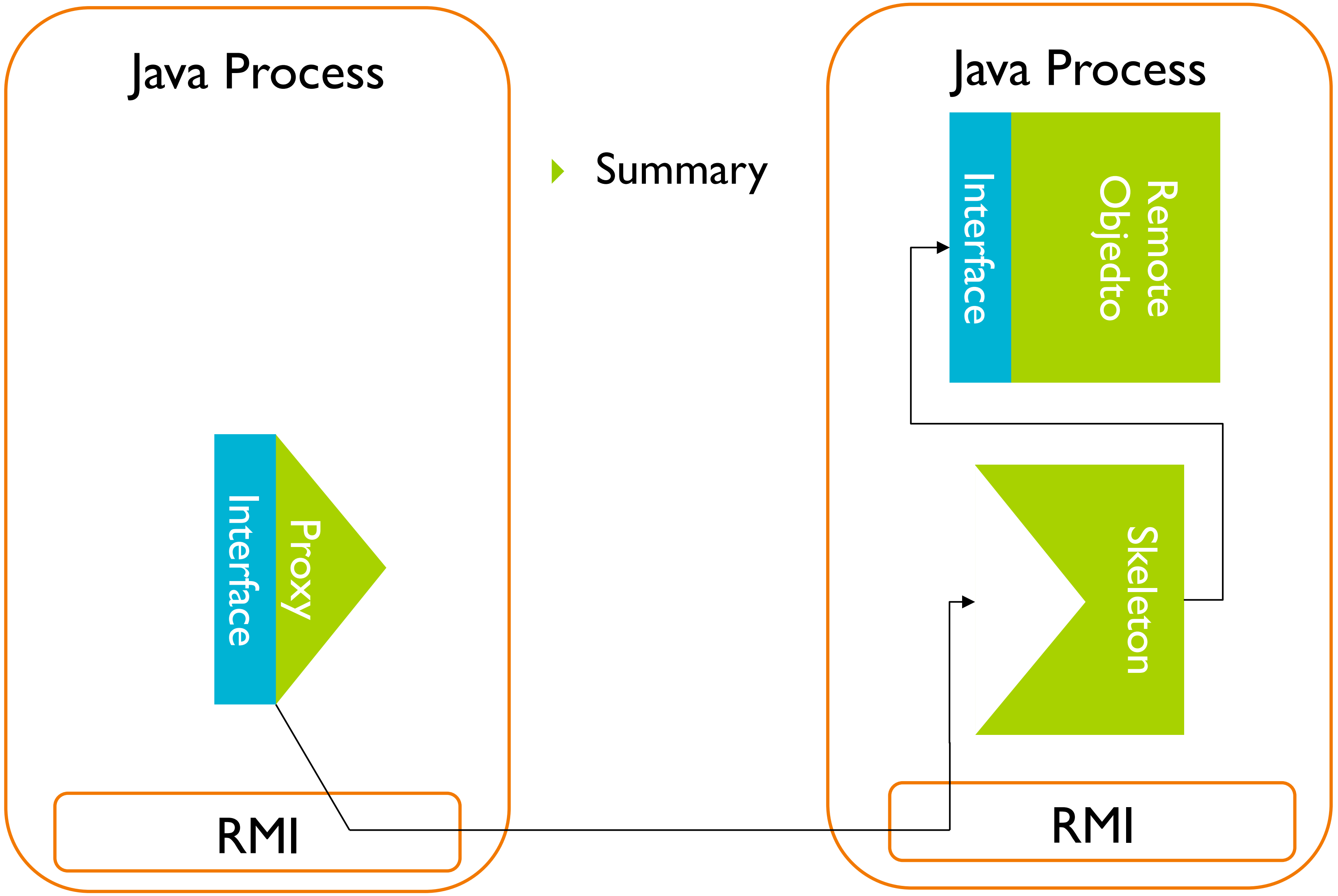
What is Java RMI?



- To invoke a remote object from another process, a Proxy object is used
  - Its interface is identical to that one of the remote object
- It contains a reference to the remote object
  - IP address + port + which object
  - Allows locating the skeleton of the remote object

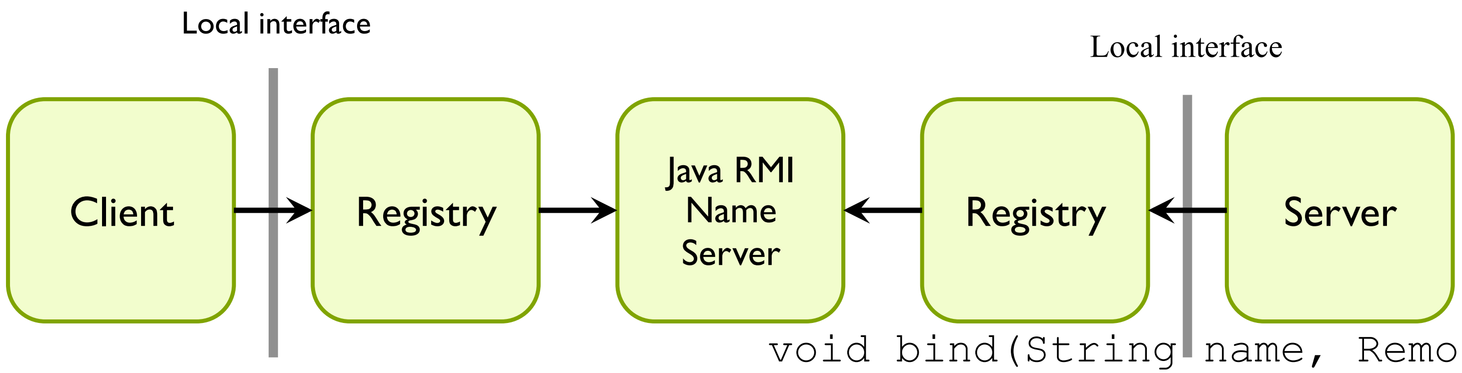
What is Java RMI?

- Summary



Java RMI Name Server

- The name server stores, for each object:
  - **symbolic name + reference**
- The name server can be hold in any node. It can be accessed by both the client and the server using a local interface named **Registry**
- In Java Oracle distributions, the name server is launched using the **rmiregistry** order.



```
Remote lookup(String name)
String[ ] list()
```

```
void bind(String name, Remote obj)
void rebind(String name, Remote obj)
void unbind(String name)
```

Developing a Java RMI application→ NOT RMI Application

```
public interface Hello {
  String greetings();
}

class ImplHello implements Hello {
  ImplHello() {..} // constructor
  public String greetings() {return ”Hello Everyone!";}
}

...
Hello h = new ImplHello();
System.out.println(h.greetings());
```

Developing a Java RMI application→ Rules for programming remote objects in Java

- **Remote object Interface**
  - The interface of the remote object must extend **`java.rmi.Remote`**
  - Methods of this interface must indicate that they can generate the **`RemoteException`** exception
  - From the interface definition, the Java compiler generates proxies and skeletons

```
public interface Hello extends Remote {
  String greetings() throws RemoteException;
}
```

Developing a Java RMI application→ Rules for programming remote objects in Java

Remote Object Class

- The class of the remote object must:
  - **implement** the remote interface
  - **extend** **`java.rmi.server.UnicastRemoteObject`**
- This allows registering the objects in the Java ORB

```
class ImplHello extends UnicastRemoteObject implements Hello {
  ImplHello() throws RemoteException {..} // constructor
  public String greetings() throws RemoteException {
    return ”Hello Everyone";
  }
}
```

Developing a Java RMI application→ Server and Client applications

// SERVER

```
  ...
Registry reg = LocateRegistry.getRegistry(host, port);
reg.rebind("objectHello", new ImplHello());
System.out.println(”Hello Server prepared ");
```

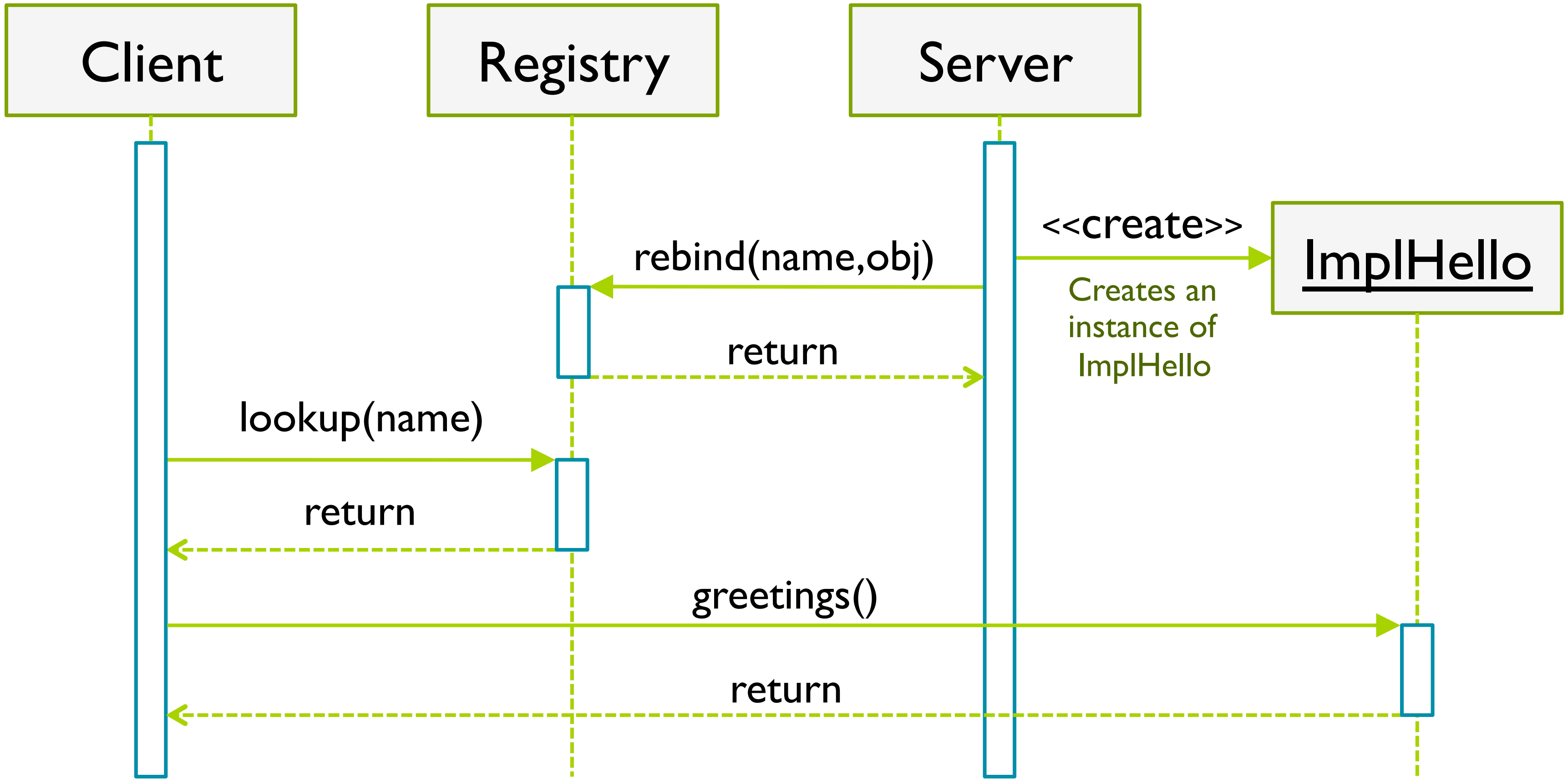
// CLIENT

```
...
Registry reg = LocateRegistry.getRegistry(host, port);
Hello h = (Hello) reg.lookup("objectHello");
System.out.println(h.greetings());
```

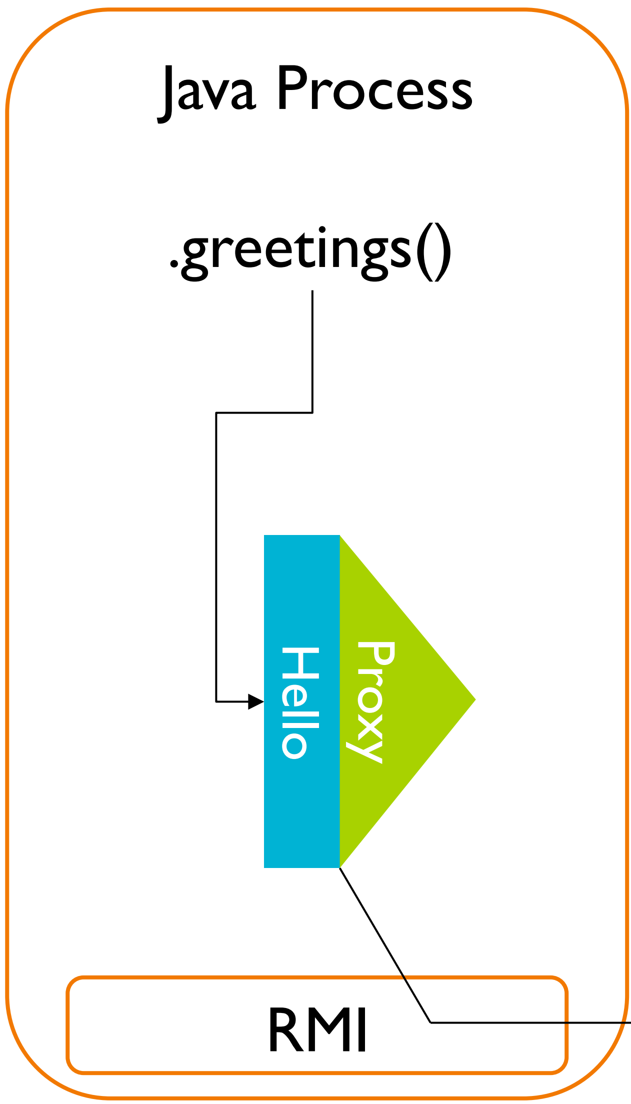
// The Registry address (host, port) has been provided as argument

Developing a Java RMI application

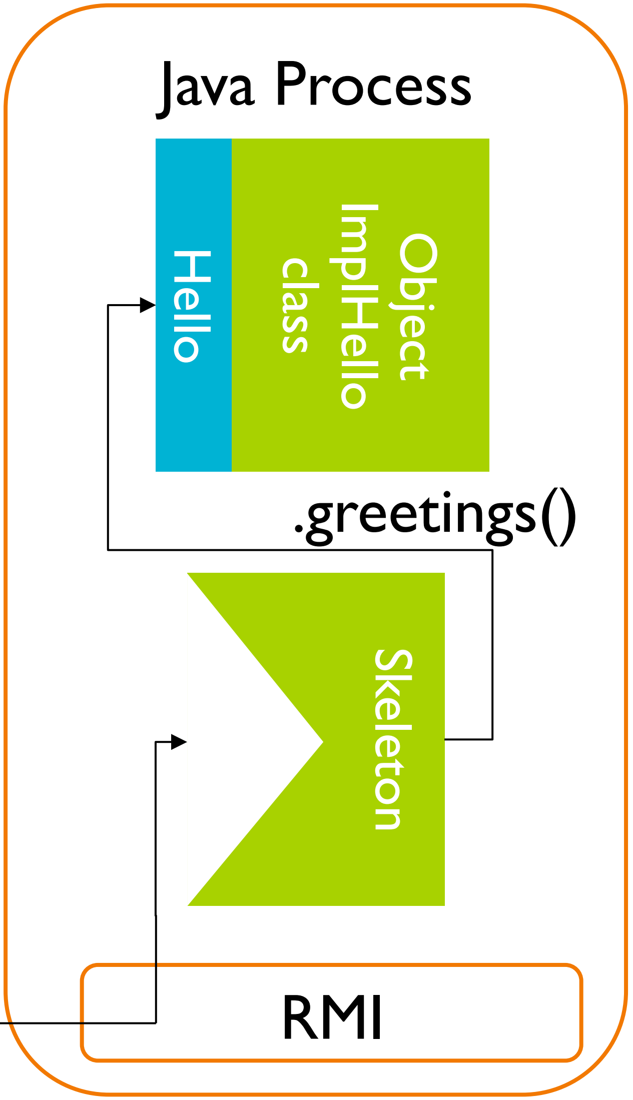
- **Steps in the execution of this example:**



Developing a Java RMI application



- Remote object that implements the Hello interface



Passing objects as arguments in Java RMI

- When invoking a method, we can pass objects as arguments
  - If the object passed as argument implements the **Remote** interface
    - It is passed by reference
    - The object is shared by previous and new references
  - If the object passed as argument **DOES NOT** implement the **Remote** interface
    - It is **serialized** and passed by value
    - An object in the final virtual machine is created totally independent from the original one

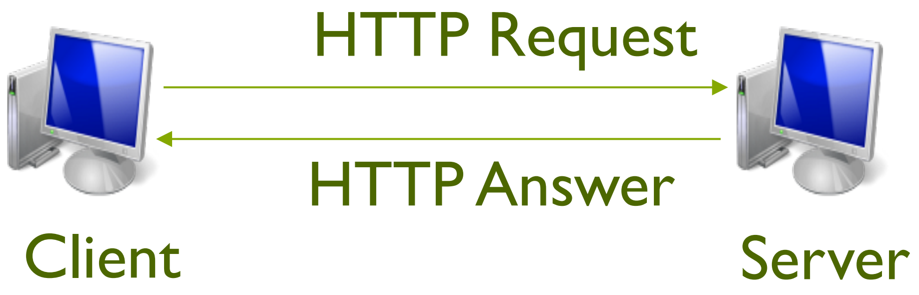
Java RMI communication Features

- Usage:
  - Calls to remote methods make transparent to programmer the usage of basic primitives for communication
- Structure and content of messages
  - Determined by the Java compiler, transparent to programmer
- Addressing
  - Direct to the computer where the remote object is
- Synchrony
  - Synchronous in answer. It is blocked till the remote object finishes
- Persistence
  - Non-persistent. The remote object must be active.

Content

Web Services

- Software system designed to profice **computer-computer interactions** using the network *[W3C Web Services Glossary]*
- Usually based on a **client-server** (request-response) architecture implemented on the **HTTP** protocol
- Web pages are not requested, but consultations and actions



Web Services

- There are many variants. The most representative are:
  - Web services based on SOAP and WSDL
    - Generally known simply as "web services"
    - SOAP (Simple Object Access Protocol): XML specification of the information that is exchanged
    - WSDL (Web Services Description Language): XML specification that describes the functionality offered by a web service
  - RESTful web services
    - Simpler and more flexible alternative, which has made them very popular nowadays, largely replacing "classic" web services
    - JSON (JavaScript Object Notation) is the most common format for the exchange of information
    - It does not require any functionality description language. However, they start to be used, being OAS (OpenAPI Specification) the most used currently.

Web services

- Alternatives to use web services
  - Construction and direct processing of the content of HTTP messages.
  - Automatic generation of code provided by frameworks for development and deployment of web services, both for the client side and the server side, based on the definition of the service
    - Given the request-response mechanism and automatic code generation, web services could be considered as a form of RPC

Content

RESTful Web Services

- They are one of the **most important technologies** for developing web applications
- They are **available** in the vast majority of programming languages and development *frameworks*
- Much of its success is due to its **ease of use**
- It is **not** a strict standard
- It is focused on **resources**
- They are services **without state**

- REST + Web Services = **RESTful Web Services**

How to reference resources in REST

- URIs for refering to resorces
- Examples:
  - http://administracion.upv.es/alumno
    - Set of all students of this university
  - http://administracion.upv.es/alumno/123456789F
    - A student whose identifier is 123456789F

Resource representation in REST

- Free resource representation
  - Most common ones: XML and JSON

XML

```xml
<Person>
  <ID>123456789F</ID>
  <Name>Agustín Espinosa</Name>
  <Email>agessa@alumno.upv.es</Email>
</Person>
```

JSON

```json
{
  "ID": "123456789F",
  "Name": "Agustín Espinosa",
  "Email": "agessa@alumno.upv.es"
}
```

Operations in REST

- HTTP methods to tell the server the type of operator to be done

Method	Operation in the server
GET	Read a resource
POST	Create a resource
PUT	Modify a resource
DELETE	Delete a resource
…	…

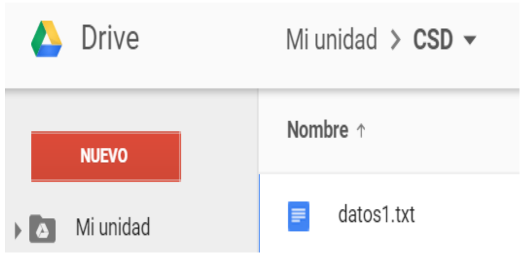
Status codes in REST

- The response indicates how the service call ended by using HTTP status codes
- Examples:

HTTP Code	Typical meaning
200	Everything ok
201	Resource has been created
400	Wrong request
404	Resource not found
500	Failure in the service provider

Parameters in REST

- The URIs can include parameters for:
  - Making queries
    - http://administracion.upv.es/alumno?apellido=Espinosa
  - Paging the answers
    - http://administracion.upv.es/alumno?pagina=4&tampagina=50
  - Providing authentication information
    - http://administracion.upv.es/alumno?key=01234567-89ab-cdef-0123-456789abcdef
  - …



Example: Google Drive Web API

HTTP Request

```
GET https://www.googleapis.com/drive/v2/files?q=title
+%3D+'datos1.txt'&
fields=items(id%2Ctitle)&key={YOUR_API_KEY}
```

HTTP Answer

```
200 OK
{
 "items": [
  {
   "id":
"1Buza8URDmLc4vbb2EP_mXkktRmRtelVMPaOjkSWtzq4",
   "title": "datos1.txt"
  }
 ]
}
```

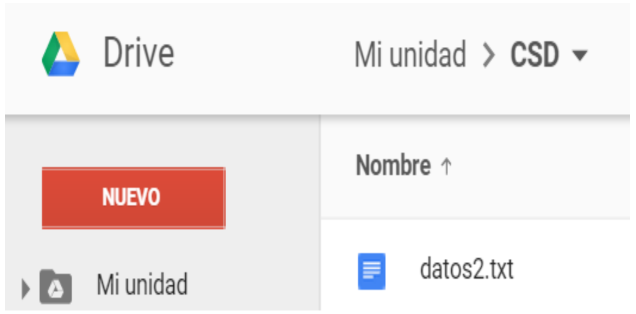
HTTP Request

```
PUT https://www.googleapis.com/drive/v2/files/
1Buza8URDmLc4vbb2EP_mXkktRmRtelVMPaOjkSWtzq4?
fields=title&key={YOUR_API_KEY}

{
 "title": "datos2.txt"
}
```

HTTP Answer

```
200 OK
 {
 "title": "datos2.txt"
}
```



Features of REST communication

- REST is a communication mechanism difficult to be categorized, depending on the point of view used for analysis:
  - **Approach 1:** from its internal mechanism point of view, its features are related to a communication based on the HTTP protocol, and thus, it uses TCP sockets.
  - **Approach 2:** from its usage point of view, it is basically a request-answer mechanism, similar to RPC and ROI in many aspects.

Features of REST communication

- Usage:
  - Approach 1: Basic primitives of sending and receiving (sockets API)
  - Approach 2: hight-level API provided by the service provider
- Structure and content of messages
  - Approach 1: Content codified with HTTP, XML, JSON
  - Approach 2: Content hidden by the API provided by the service provider
- Addressing
  - Direct by means of requests to the computer supporting the service
- Synchrony
  - Approach 1: Synchronous in delivery
  - Approach 2: Synchronous in answer
- Persistence
  - Non-persistent

Content

Message Oriented Middlewares (MOMs)

- Middlewares that offer **message-based** communication.
- Senders send messages not directly to receivers, but to an intermediate element usually called **queue**.
  - Once the message has been deposited in the queue, the sender continues with its execution, without waiting for the recipient to collect the message.
  - The receiver collects the message at any time (immediately when it is deposited or later)
  - The communication is therefore **asynchronous**.

Message Oriented Middlewares (MOMs)

- Most of these systems are based on a communication **broker**:
  - A server process
  - Fully manages the queues:
    - Creation and deletion of queues
    - It handles senders' deliveries and deposits the messages in the indicated queue
    - It handles receivers' requests to collect messages from the indicated queue
    - It keeps pending delivery messages in the queues, although senders and receivers are not running, not even the broker itself, thus offering persistence.
- Notable exception: ZeroMQ
  - It does not require the existence of the broker

Message Oriented Middlewares (MOMs)

- Main advantages
  - They allow highly decoupled components of the distributed system.
  - They can offer a high degree of availability through active replicas of brokers
  - It is possible to achieve a high degree of security by being centralized systems
- Main drawbacks
  - Lower performance, by introducing the broker as an intermediate element in communications
  - Difficult scalability, again due to the existence of the broker
  - Lack of standardization. There are open protocols but many of the most used implementations are proprietary.

Content

Java Message Service 2.0 (JMS)

- Java Message Service (JMS) is a **Java API** that enables applications to send and receive messages
- It offers a **loosely coupled communication**
  - The sender sends messages to a **destiny** and the receiver receives messages from this destiny
  - Sender and receiver do not need to know each other, they just need to agree on the format of message content.

Java Message Service 2.0 (JMS)

- Interesting to use JMS when:
  - You do not want that components of an application have to know the interfaces of other components
  - It is not necessary that all components are running simultaneously
  - The components, after sending a message, do not need to receive an immediate response to continue operating normally

Java Message Service 2.0 (JMS) → Components

- **JMS Components**
  - JMS Providers
    - Messaging system that implements the JMS interfaces and provides administrative and control tools
  - JMS Clients
    - Program or component written in Java that produces or consumes messages
  - Messages
    - Objects that communicate information between JMS clients
  - Administered objects
    - Preconfigured objects created by an administrator for the use of clients.
    - Two kinds:
      - □ Connection factories
      - □ Destinations: queues and *topics*

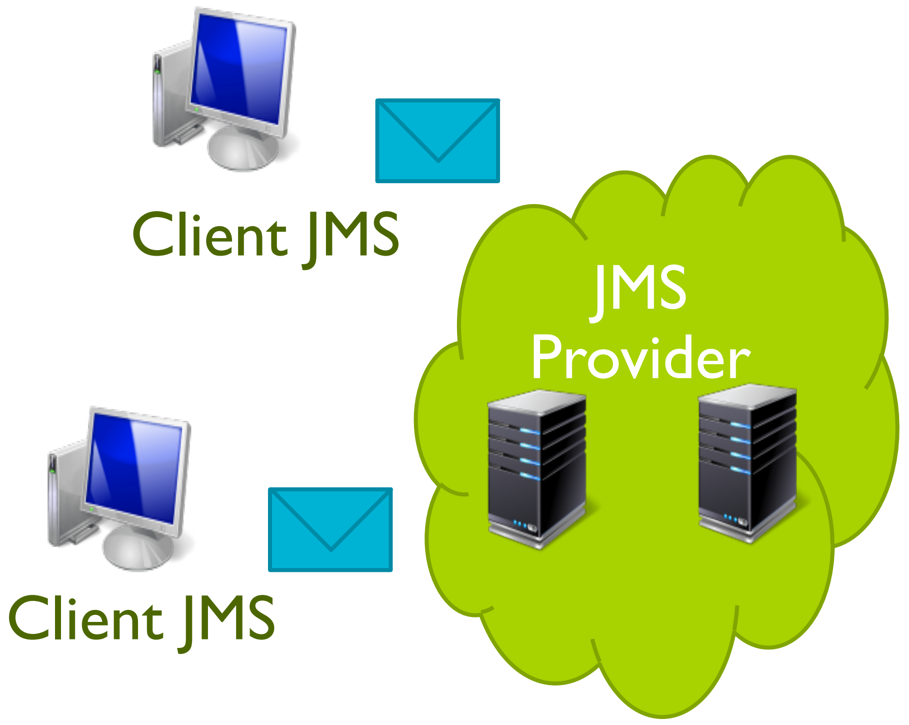
Java Message Service 2.0 (JMS) → Components

- JMS Provider
  - It is a messaging system that implements the JMS interfaces and provides administrative and control tools
  - It is included in the servers of the Java Enterprise Edition (JEE) platform:
    - GlassFish, Oracle WebLogic Server, IBM WebSphere Application Server, Apache Geronimo, etc.
  - Also available as independent servers (only JMS)
    - Apache ActiveMQ, JBoss Messaging, SwiftMQ, Open Message Queue, etc.

Java Message Service 2.0 (JMS) → Components

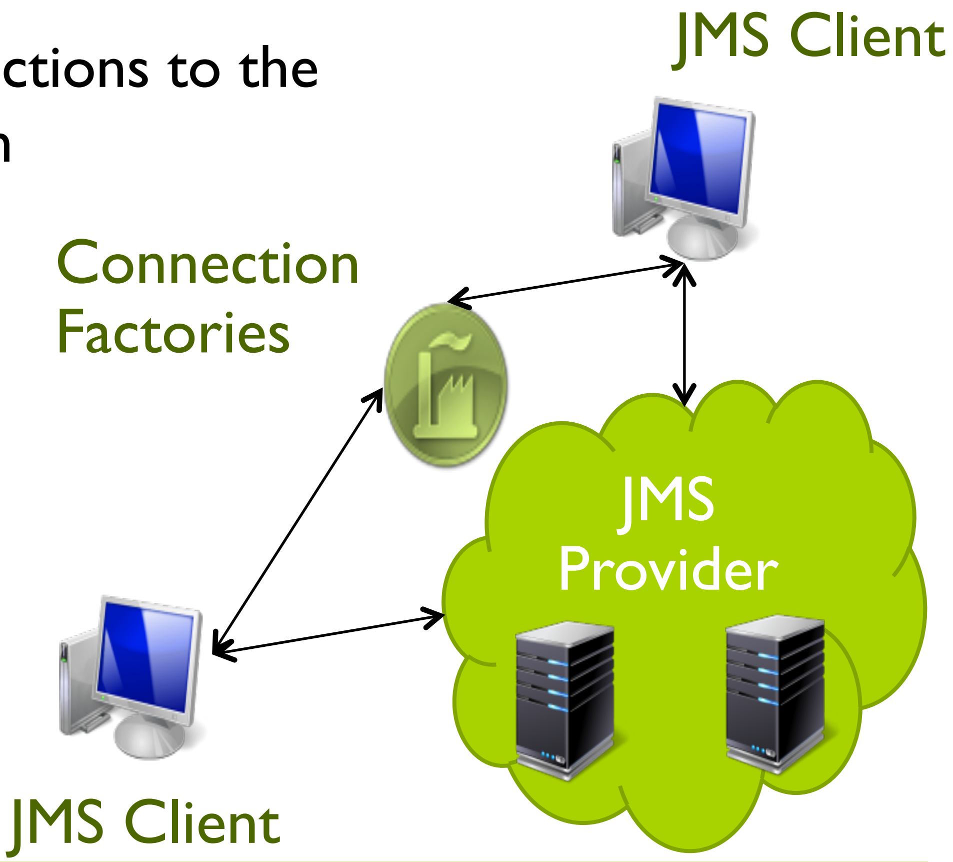
- <u>Message Structure:</u>

Header
Predefined fields defined by JMS (identifier, sending timestamp, etc.)
Properties
Header extension. Fields defined by the program
Body
Types: empty, text, bytes, serialized object, etc.



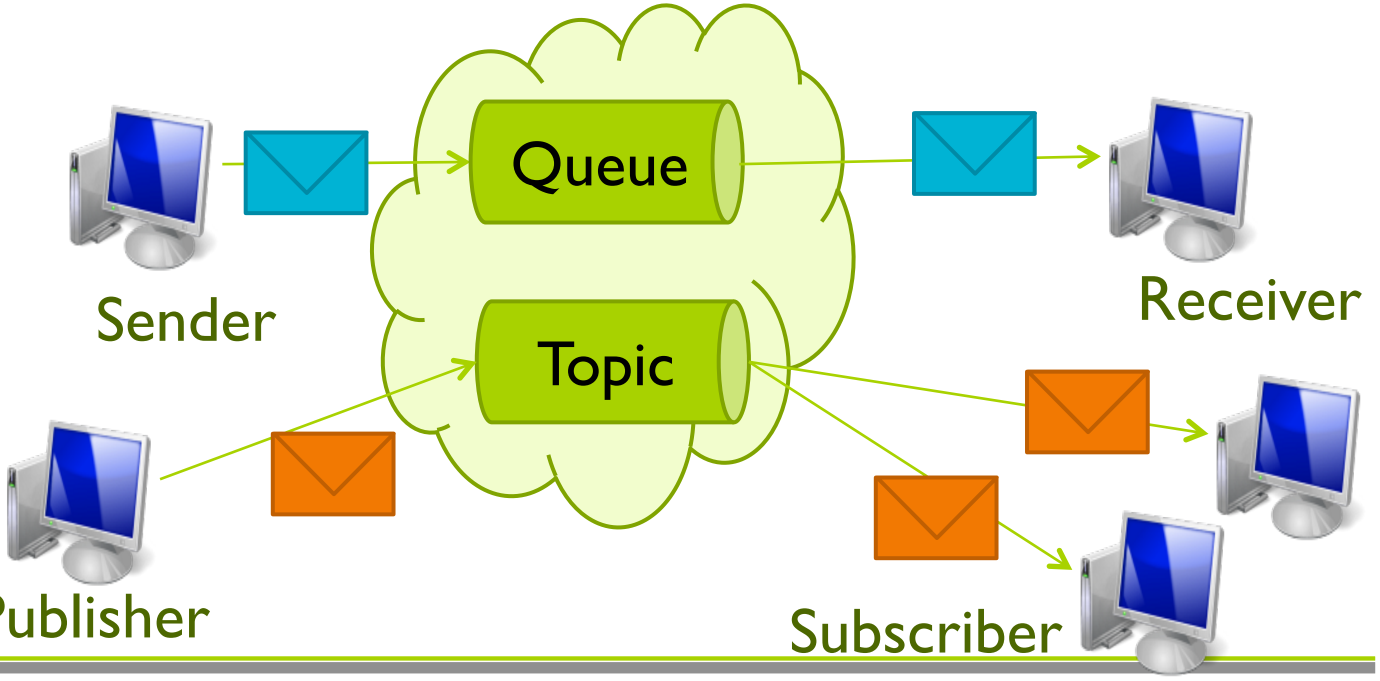
Java Message Service 2.0 (JMS) → Components

- **Connection Factories**
  - Created by the adminitrative tools of the JMS provider
  - They are used to create connections to the clients of the messaging system



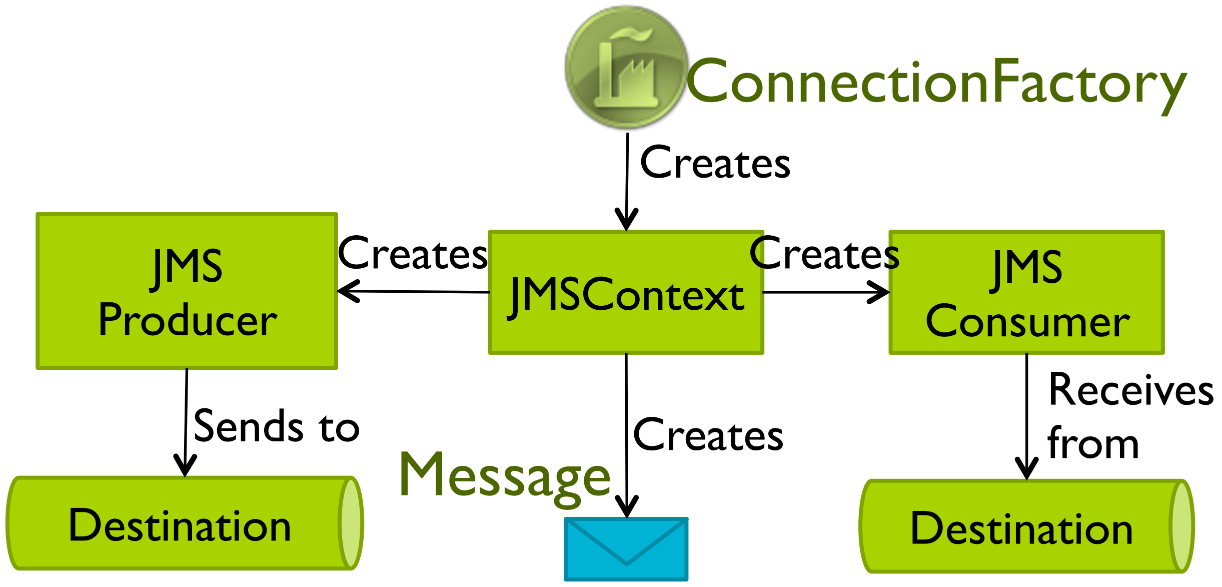
Java Message Service 2.0 (JMS) → Components

- **Destinations**
  - Created by the administrative tools of the JMS provider
  - Types
    - Queues: delivery to one single client
    - Topics: delivery to multiples clients



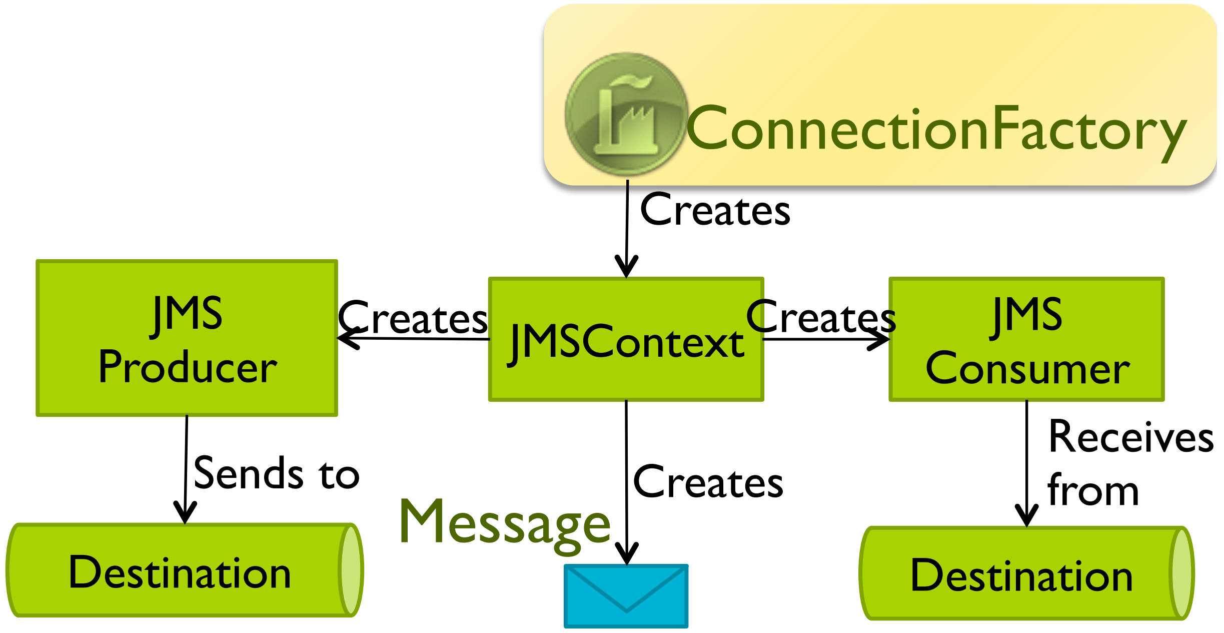
Java Message Service 2.0 (JMS) → Programming Model

- **Elements of the programming model**
  - Connection Factory Interface
  - JMSContext Interface
  - JMSProducer Interface
  - JMSConsumer Interface
  - Destination Interface
  - Message Interface



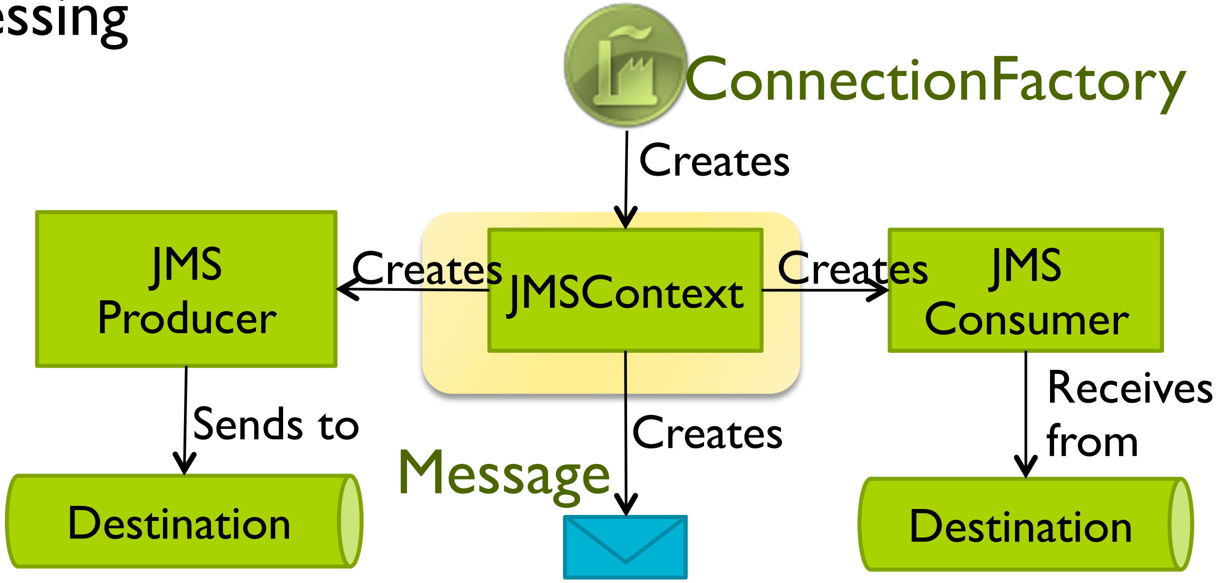
Java Message Service 2.0 (JMS) → Programming Model

- **<u>Connection Factory Interface</u>**
  - The objects that implement it:
    - Link the application with an administered object
    - Create connections with the JMS provider



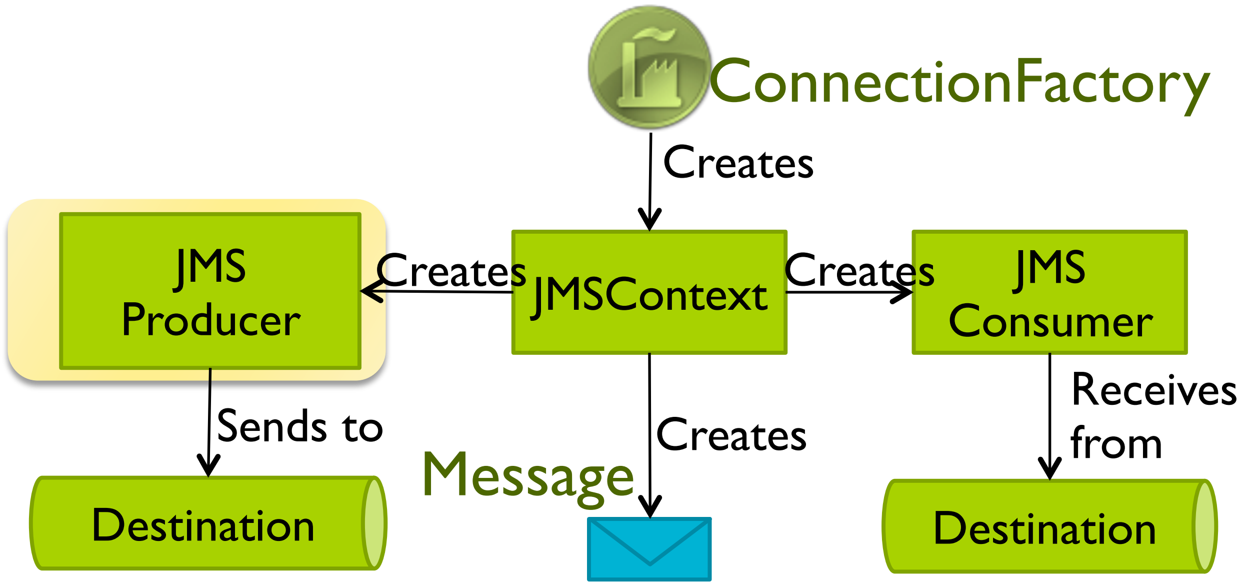
Java Message Service 2.0 (JMS) → Programming Model

- **JMSContext Interface**
  - The objects that implement it:
    - Maintain a connection with the JMS provider
    - They are usable by a single thread of execution
    - They process sendings and receivings sequentially
    - An application can use several JMSContext objects if it requires of concurrent processing



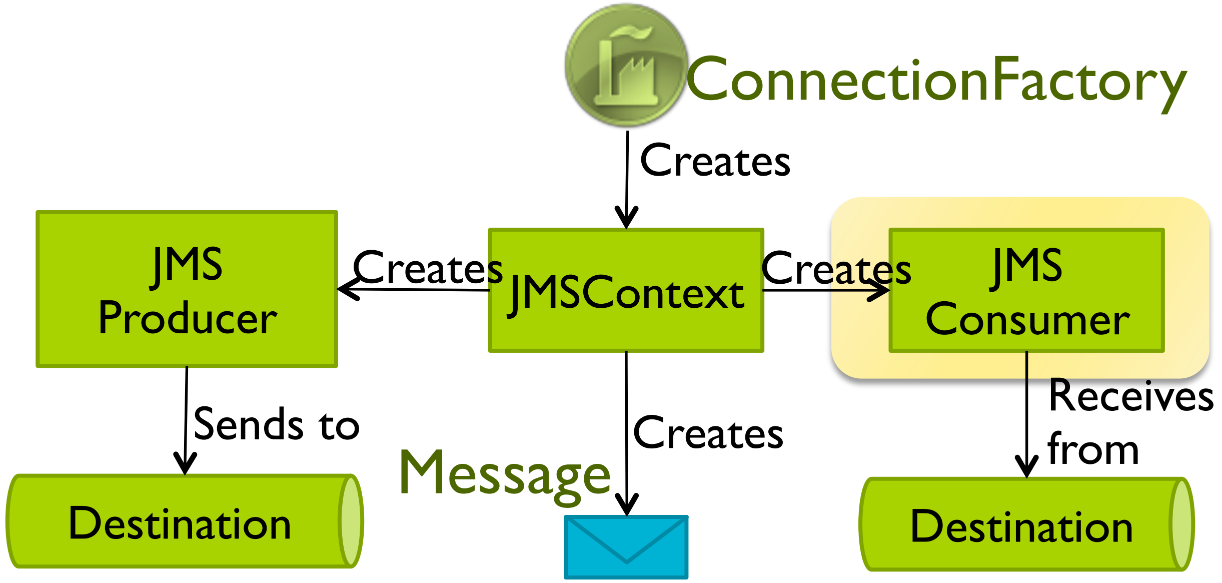
Java Message Service 2.0 (JMS) → Programming Model

- **JMSProducer Interface**
  - The objects that implement it:
    - They enable sending messages to queues and topics



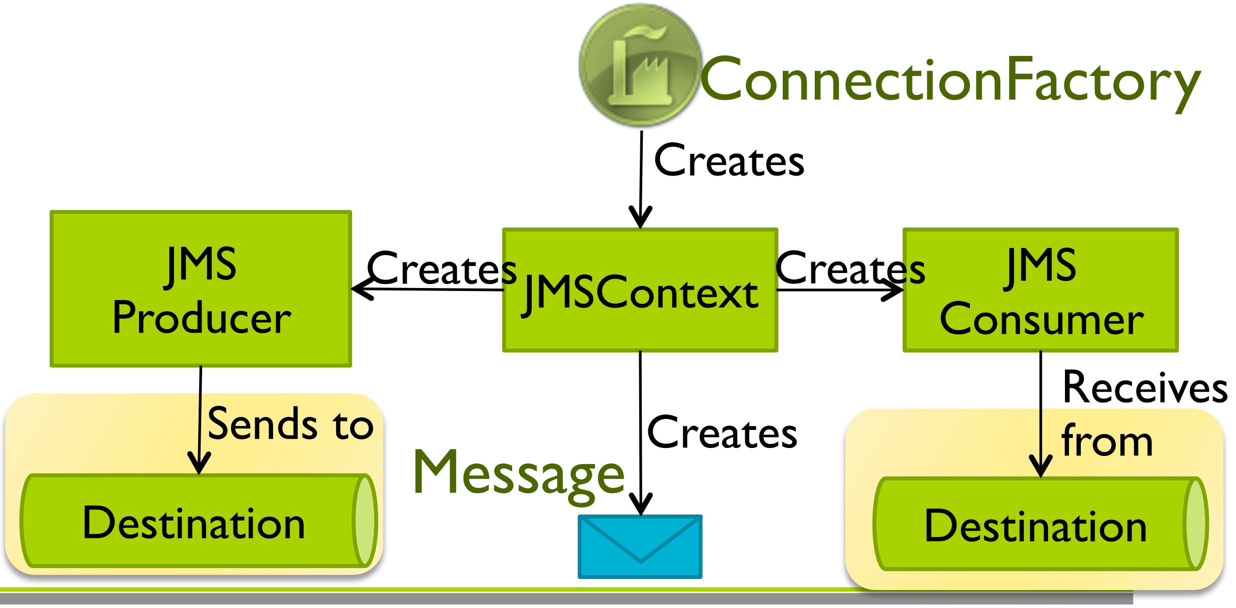
Java Message Service 2.0 (JMS) → Programming Model

- **JMSConsumer Interface**
  - The objects that implement it:
    - Enable receiving messages from queues and topics



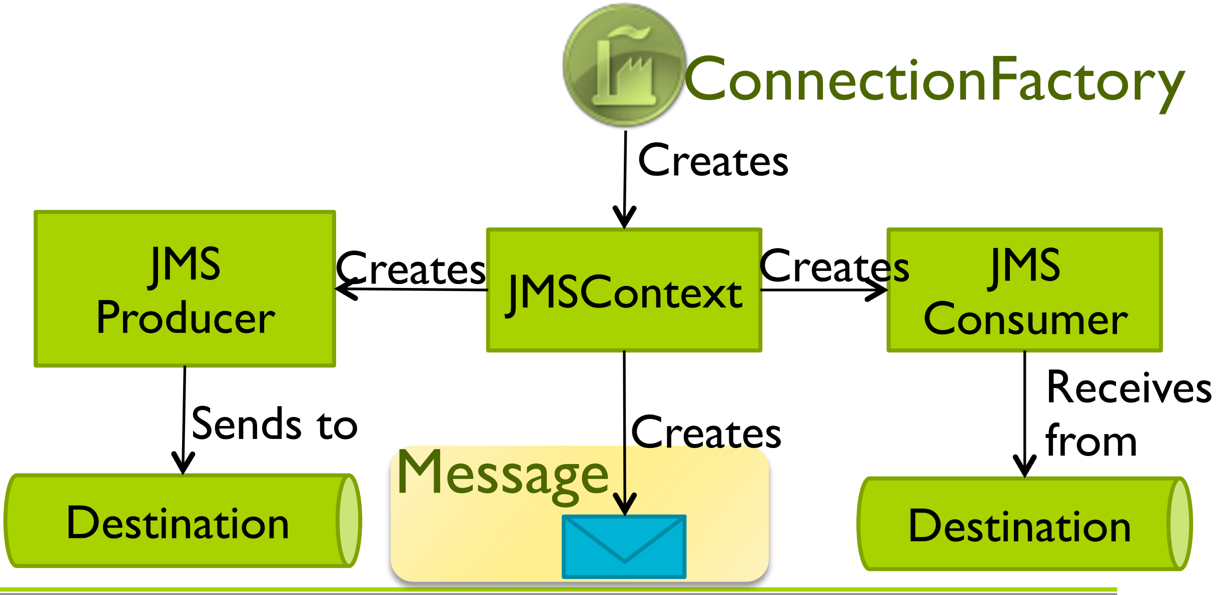
Java Message Service 2.0 (JMS) → Programming Model

- **Destination Interface**
  - The objects that implement it:
    - Link the application with an administered object
    - Encapsulate a specific JMS provider address
  - Subinterfaces:
    - Queue
    - Topic



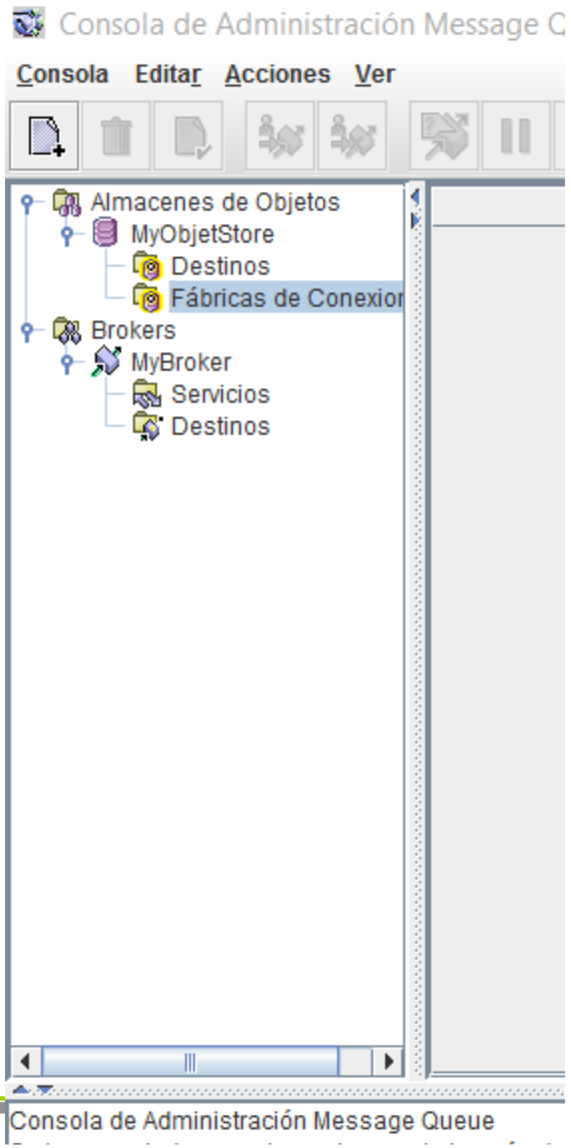
Java Message Service 2.0 (JMS) → Programming Model

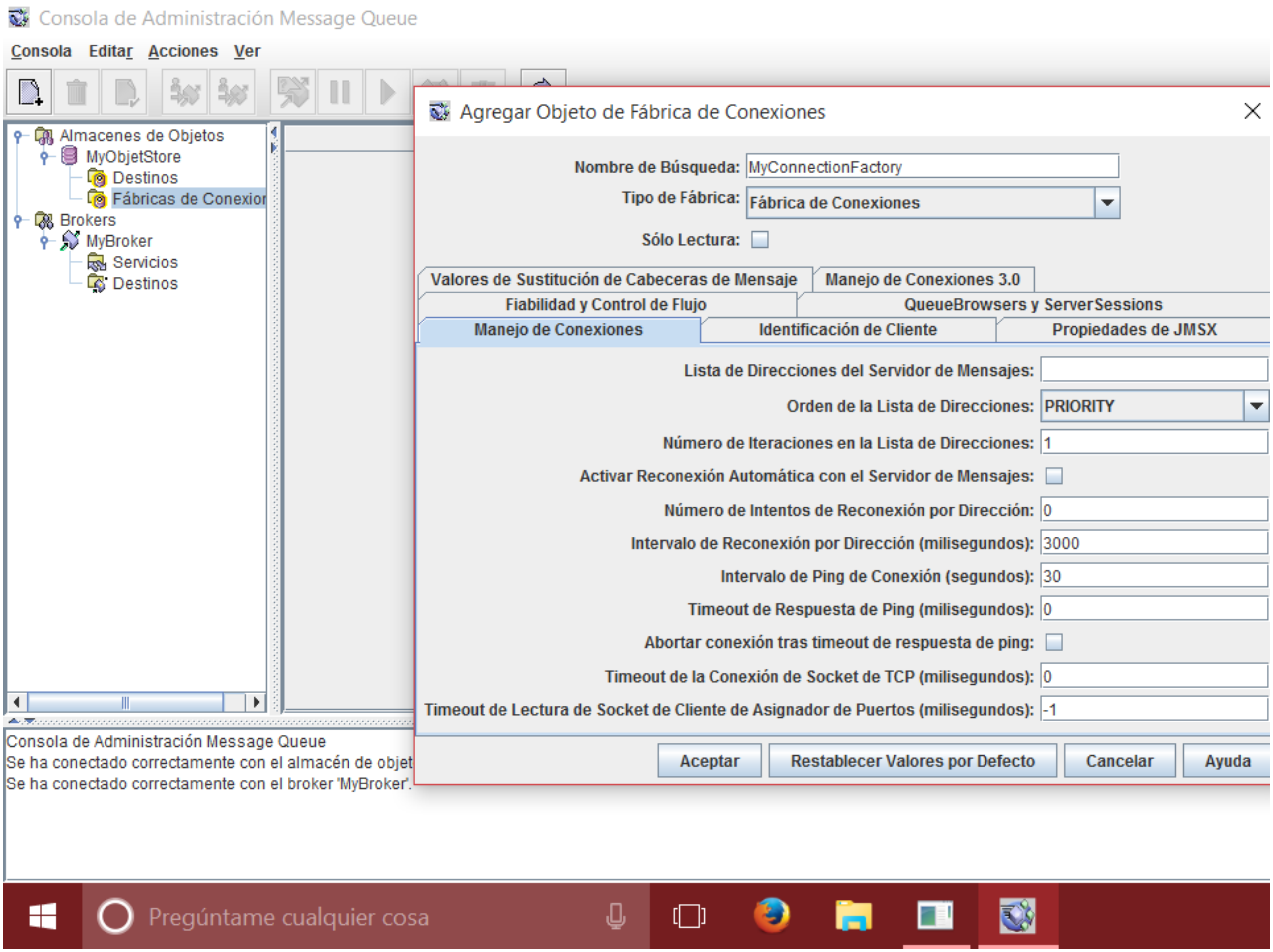
- **Message Interface**
  - The objects that implement it:
    - They are the messages that are sent and received in JMS
  - Subinterfaces:
    - TextMessage
    - BytesMessage
    - ObjectMessage
    - MapMessage
    - etc.



Java Message Service 2.0 (JMS) → Example

- Producer
  - Sends a text message to "MyQueue" queue and ends
- Consumer
  - Receives messages from "MyQueue" queue
  - Shows each message
  - Ends after 10 seconds without receiving any message
- JMS Provider
  - Open Message Queue

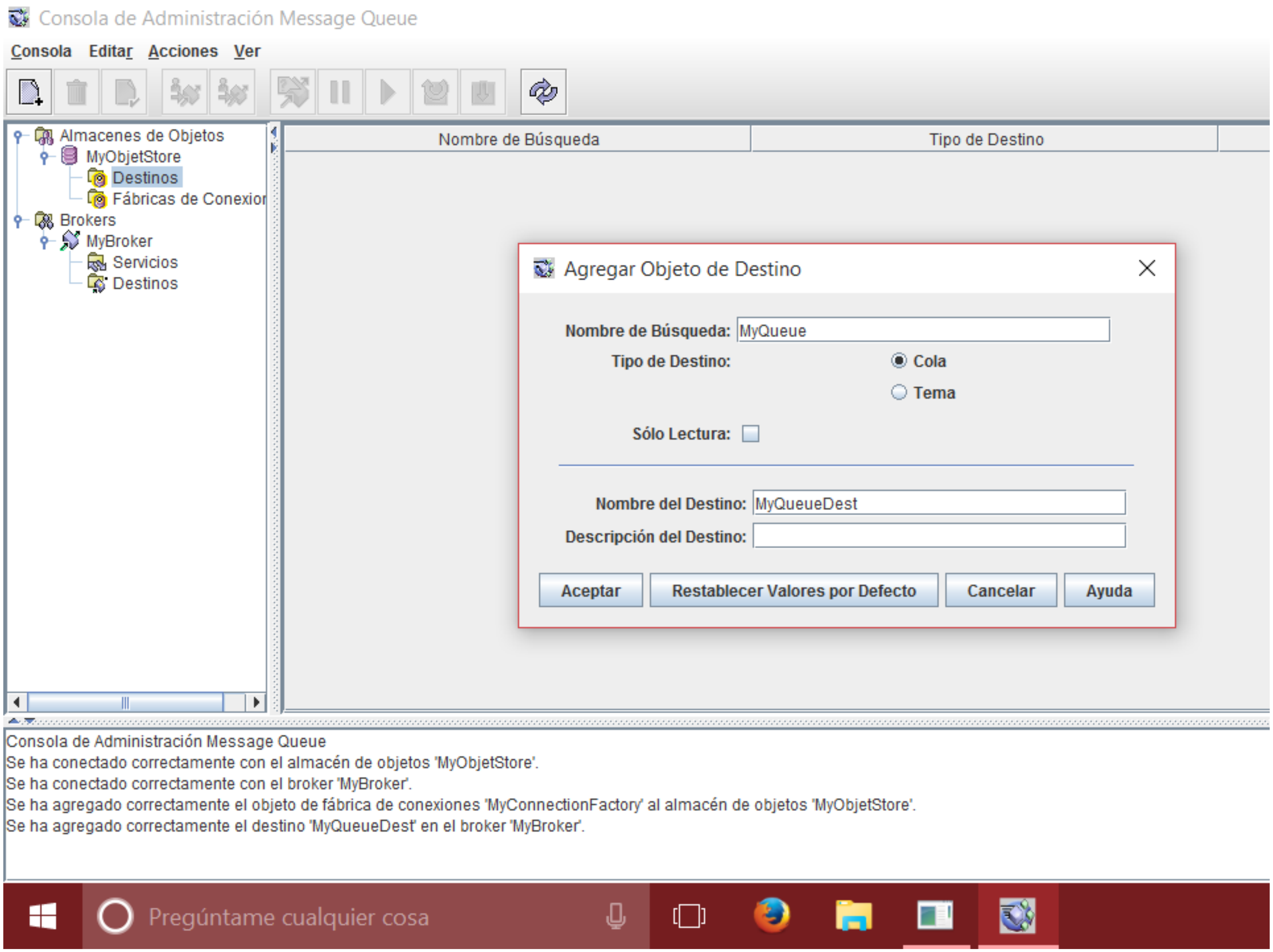



Consola de Administración Message Queue
Consola
Editar
Acciones
Ver
Almacenes de Objetos
MyObjetStore
Destinos
Fábricas de Conexion
Brokers
MyBroker
Servicios
Destinos
Agregar Objeto de Fábrica de Conexiones
Valores de Sustitución de Cabeceras de Mensaje
Manejo de Conexiones 3.0
Fiabilidad y Control de Flujo
QueueBrowsers y ServerSessions
Manejo de Conexiones
Identificación de Cliente
Propiedades de JMSX
Aceptar
Restablecer Valores por Defecto
Cancelar
Ayuda
Pregúntame cualquier cosa

Consola de Administración Message Queue
Se ha conectado correctamente con el almacén de objet
Se ha conectado correctamente con el broker 'MyBroker'.

Nombre de Búsqueda:	MyConnectionFactory
Tipo de Fábrica:	Fábrica de Conexiones
Sólo Lectura:	☐

Lista de Direcciones del Servidor de Mensajes:	
Orden de la Lista de Direcciones:	PRIORITY
Número de Iteraciones en la Lista de Direcciones:	1
Activar Reconexión Automática con el Servidor de Mensajes:	☐
Número de Intentos de Reconexión por Dirección:	0
Intervalo de Reconexión por Dirección (milisegundos):	3000
Intervalo de Ping de Conexión (segundos):	30
Timeout de Respuesta de Ping (milisegundos):	0
Abortar conexión tras timeout de respuesta de ping:	☐
Timeout de la Conexión de Socket de TCP (milisegundos):	0
Timeout de Lectura de Socket de Cliente de Asignador de Puertos (milisegundos):	-1


Consola de Administración Message Queue
Consola Editar Acciones Ver
Almacenes de Objetos
MyObjetStore
Destinos
Fábricas de Conexior
Brokers
MyBroker
Servicios
Destinos
Nombre de Búsqueda
Tipo de Destino
Agregar Objeto de Destino
Aceptar
Restablecer Valores por Defecto
Cancelar
Ayuda
Pregúntame cualquier cosa

Nombre de Búsqueda: MyQueue

Tipo de Destino: [✓ Cola] Tema

Sólo Lectura: ☐

Nombre del Destino: MyQueueDest

Descripción del Destino:

```
Consola de Administración Message Queue
Se ha conectado correctamente con el almacén de objetos 'MyObjetStore'.
Se ha conectado correctamente con el broker 'MyBroker'.
Se ha agregado correctamente el objeto de fábrica de conexiones 'MyConnectionFactory' al almacén de objetos 'MyObjetStore'.
Se ha agregado correctamente el destino 'MyQueueDest' en el broker 'MyBroker'.
```

Java Message Service 2.0 (JMS) → Example

- **Producer**

```
import javax.jms.*; import javax.naming.*; // JMS and JNDI Interfaces
public class Producer {
    public static void main(String[] args) {
        try {
           Context jndiCtx = new InitialContext(); // JNDI Initialization
           // Access to administered objects
           ConnectionFactory connectionFactory =
             (javax.jms.ConnectionFactory) jndiCtx.lookup("MyConnectionFactory");
           Queue queue = (javax.jms.Queue) jndiCtx.lookup("MyQueue");
            // Creates context from the connection factory
           JMSContext context = connectionFactory.createContext();
            // Creates message producer from the context
           JMSProducer producer = context.createProducer();
           // Creates message from the context
           TextMessage message = context.createTextMessage();
          // Builds the body of the message (text type)
           message.setText("This is a message");
            // Creates and links a property to the message
           message.setBooleanProperty("Important", true);
           // Sends the message to the queue through the message producer
          // Waits the JMS provider to store it, but doesn't wait the client to receive it
           producer.send(queue, message);

        } catch (JMSException | JMSRuntimeException | NamingException e) { ...
```

Java Message Service 2.0 (JMS) → Example

Consumer

```
import javax.jms.*; import javax.naming.*; // JMS and JNDI Interfaces
public class Consumer {
    public static void main(String[] args) {
        try {
           Context jndiCtx = new InitialContext();  // JNDI Initialization
           // Access to administered objects
           ConnectionFactory connectionFactory =
               (javax.jms.ConnectionFactory) jndiCtx.lookup("MyConnectionFactory");
           Queue queue = (javax.jms.Queue) jndiCtx.lookup("MyQueue");
            // Creates context from the connection factory
           JMSContext context = connectionFactory.createContext();
            // Creates message consumer from the context
           //  It is linked to a specific queue
           JMSConsumer consumer = context.createConsumer(queue);
            // Waits a message for 10 seconds maximum
           while (true) {
              Message m = consumer.receive(10000);
              if (m != null) {// Message received
                  System.out.println(m);
              } else {
                  break; // Maximum waiting achieved
              }
           }
        } catch (JMSRuntimeException | NamingException e) { ...
```

Java Message Service 2.0 (JMS) → Example

```
Text:    This is a message
Class:
com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID():      ID:
9-192.168.137.1(f9:0:96:1b:b8:68)-58313-1427645760159
getJMSTimestamp():      1427645760159
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSDeliveryTime():   0
getJMSPriority():       4
Properties:             {Important=true, JMSXDeliveryCount=1}
```

Body

Header

Properties

JMS communication features

- Usage
  - Basic primitives for sending and receiving
- Structure and content of messages
  - Header, properties, body (with different types
- Addressing
  - Indirect through JMS provider
- Synchrony
  - Asynchronous. The sender keeps on running when it delivers the message to the JMS provider
- Persistence
  - Persistent. Even if the JMS provider stops, because it stores the message in secondary storage.

Learning results of this Teaching Unit

- At the end of this unit, the student should be able to:
  - Characterize the message-based communication mechanisms, describing their most relevant features.
  - Characterize the mechanism of remote object invocation (ROI) and the usage of references to object.
  - Characterize the Java RMI mechanisms and describe the steps for developing a Java RMI application
  - Characterize the Web Services. Describe the main features of RESTful web services, its usage of resources and REST operations.
  - Characterize the message queues and the Java Message Service communication mechanism.

Bibliography

- Remote procedure call (RPC)
  - Bruce J. Nelson. **Remote Procedure Call**. Phd Thesis, Carnegie Mellon Univ., 1981.
  - Andrew D. Birrel y Bruce J. Nelson. **Implementing remote procedure calls**. ACM Trans. Comput. Syst., 2(1):39–59, febrero 1984.
- Remote Object invocation (ROI)
  - Marc Shapiro. **Structure and encapsulation in distributed systems: The proxy principle**. In 6th Intnl. Conf. on Distrib. Comput. Sys. (ICDCS), pp. 198–204, Cambridge, Massachusetts, EE.UU., mayo 1986.
- Java RMI
  - Oracle Corp. **Remote Method Invocation home**. November 2012.

    http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html
  - Oracle Corp. **The JavaTM tutorials: Rmi**. November 2012. *http://docs.oracle.com/javase/tutorial/rmi/index.html*

Bibliography

- RESTful Web Services

  - **Tutorial "Developing RESTful APIs with JAX-RS"**. Java Brains. Koushik Kothagal. *http://javabrains.koushik.org/courses/javaee_jaxrs*
  - **"Simplemente REST"** By Gabriel Fagúndez. TechMeetUp. 23rd November 2013. *https://www.youtube.com/watch?v=NXtMM7Wmn8M*
  - Rafael Navarro Marset. REST vs Web Services. **Modelado, Diseño e Implementación de Servicios Web 2006-07**. ELP-DSIC-UPV. *http://users.dsic.upv.es/~rnavarro/NewWeb/docs/RestVsWebServices.pdf*
  - The Java EE 6 Tutorial. Chapter 18 - **Introduction to Web Services.** Oracle. January 2013. *https://docs.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf*

Bibliography

- Java Message Service
  - **Introducción a Java Message Service**. Departamento de Ciencia de la Computación e Inteligencia Artificial. Universidad de Alicante.

    http://expertojava.ua.es/j2ee/publico/mens-2010-11/sesion01-apuntes.html

  - **Introducing the Java Message Service**. Willy Farrel. ibm.com/developerWorks *http://www.ibm.com/developerworks/java/tutorials/j-jms/j-jms-updated.html*

  - **Java Message Service Concepts**. Java Platform, Enterprise Edition: The Java EE Tutorial. Chapter 45. Oracle Java Documentation. *http://docs.oracle.com/javaee/7/tutorial/partmessaging.htm#GFIRP3*

  - **Package javax.jms** *http://docs.oracle.com/javaee/7/api/javax/jms/package-summary.htm*