Sockets

Daniel Hagimont

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

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

1

The first part of this lecture is devoted to sockets.

What are sockets

- Interface for programming network communication
- Allow building client/server applications
  - Applications where a client program can make invocations to server programs with messages (requests) rather than shared data (memory or files)
  - Example: a web browser and a web server
- Not only client/server applications
  - Example: a streaming applications (VOD)

2

Sockets are a programming interface (API) for implementing message exchanges between processes which may run on different machines.

Such message exchanges are often used to implement distributed applications following the client-server model.

In this model, a server is a program running one machine, which provides a service to some client programs running on other machines. A client may invoke this sevice by sending a message (called request) to the server program. Upon reception of a request, the server executes the treatments which correspond to the service, then it sends a message (called response) back to the client. The client is suspended after the emission of the request until reception of the response.

Notice that the request/response may include parameters/results.

A very popular exmple is the communication between a web browser (client) and a web server (server).

However, message exchanges can be used to implement other types of application, e.g. streaming applications like Video On Demand.

Two modes connected/not connected

- Connected mode (TCP)
  - Communication problems are handled automatically
  - Simple primitives for emission and reception
  - Costly connection management procedure
  - Stream of bytes: no message limits
- Not connected mode (UDP)
  - Light weight: less resource consumption
  - More efficient
  - Allow broadcast/multicast
  - All communication problems (packet loss) have to be handled by the application

3

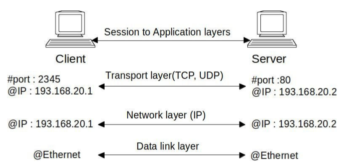
Communication between processes can be performed following 2 modes :

- the connected mode corresponds to the use of the TCP communication protocol. We can create a connection between the client process and the server process. The connection is bi-directional (both the client and the server can send data on the connection). The communication mode is a stream of byte, i.e. there's no message limit. Communication problems (reemission of lost packets, blocking in case of buffer saturation) are automatically handled by TCP. The establishment of the connection is costly.

- the non connected mode corresponds to the use of the UDP communication protocol. There's no connection establishment anymore, nor handling of communication problems. It reduces resource consumption. A message of any size can be sent (it is split into IP packets, and reassembled on reception). There's no guarantee regarding message reception (it has to be handled by the application). Notice that UDP allows sending messages in multicast or broadcast (in general on a local network).

Sockets

- Network access interface
- Developed in Unix BSD
- @IP, #port, protocol (TCP, UDP, ...)



4

Sockets were initially developed in Unix BSD (Berkeley Software Distribution). They provide access to the network.

At the bottom layer (data link), machines (or rather network cards) are identified by a MAC address (e.g. an Ethernet address).

At the middle level (network), machines are identified by an IP address. ARP is the protocol which allows translating an IP address into a MAC address on a local network.

At the top level (transport), a process on one machine is identified by a couple @IP / #port, e.g. a web server is accessible on port 80 (default port) on a machine.

The socket API

- Socket creation: socket(family, type, protocol)
- Opening the dialog:
  - Client: bind(..), connect(...)
  - Server: bind(..), listen(...), accept(...)
- Data transfer:
  - Connected mode: read(...), write(...), send(...), recv(...)
  - Non-connected mode: sendto(...), recvfrom(...), sendmsg(...), recvmsg(...)
- Closing the dialog:
  - close(...), shutdown(...)

5

The socket API includes a set of functions in a programming language (initially C) for managing communication between processes.

A socket is a file descriptor, similar to the file descriptors used to access files, except that writing or reading on such a descriptor sends or receives data to/from a remote process.

The socket API includes function for :

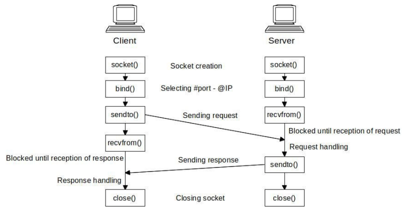
- creating a socket

- opening the dialog, i.e. initializing the socket (in connected or non-connected mode)

- transfering data (in connected or non-connected mode)

- closing the dialog

Client/Server in non-connected mode



6

We describe the schema of a request/response interaction between a client and a server. Here we consider its implementation with non connected sockets (UDP).

- both the client and the server create a socket with the socket() function which returns a file descriptor (fd, an index in the file descriptor table of the process). This fd is a parameter of all the following function calls.

- both the client and server call the bind() function which associates the socket with a local port of the machine (given as parameter). This port is the port used to receive messages (by the client or the server).

Generally, on the server side, this port in known in advance and given as parameter to bind(). The client knows this server port and communicates with the server identified with the IP address of the server and this server port. If the port is already used, bind() returns an error.

Generally, on the client side, the port given to bind() is 0, which means that bind() has to allocate a free port. This port is only used to receive responses.

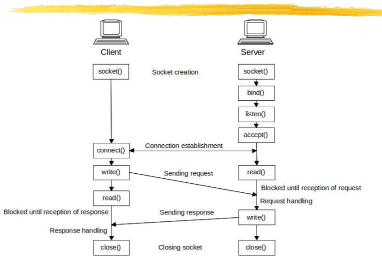
- the client can call the sendto() function to send a message, giving as parameter the IP address and port of the target server process.

- the server can call the recvfrom() function to wait for a message. This function blocks until reception of a message. Upon reception, the message (request) is handled.

- the server can send a response with sendto(). The IP address and port of the client process (which sent the request) can be found in the request message.

- the client waits for the response using the recvfrom() function. Upon reception, the client can handle the response.

Client/Server in connected mode



Here we consider a request/response interaction with connected sockets (TCP).

- both the client and the server create a socket with the socket() function which returns a file descriptor (fd). This fd is a parameter of all the following function calls.

- on the server side

  - bind() allows to associate the socket with a local port.
    This port is generally known (e.g. port 80 for a web server)
  - listen() allows to specify that the socket will be used to receive
    connection requests and how many connection requests can be pending
  - accept() blocks until reception of a connection request from a client.
    Upon reception of a connection request, accept() returns **a new socket**
    (a new fd) which is used by the server to send/receive on the
    established connection.

- on the client side

  - connect() allows to send a connection request to the server,
    giving as parameter the IP address and port of the target server process.
    connect() includes a call to bind() (this is hidden).
    After returning from connect(), the connection is established and
    the socket is used to send/receive.

What is important is the difference between the client and the server.

The client creates a socket, calls connect() and then use the socket to send/receive messages on that connection.

The server creates a socket, calls bind() and accept() and obtains a **NEW** socket for that connection with the client. The server may accept other connections with other clients and will obtain a different socket for each connection/client.

On a TCP connection, data may be sent/received with write/read functions on sockets (the same functions used to write/read data to/from a file).

socket() function

- int socket(int family, int type, int protocol)
- family
  - AF_INET: for Internet communications
  - AF_UNIX: for local communications
- type or mode
  - SOCK_STREAM: connected mode (TCP)
  - SOCK_DGRAM: non-connected mode (UDP)
  - SOCK_RAW: direct access to low layers (IP)
- protocol :
  - Protocol to use (different implementations can be installed)
  - 0 by default (standard)

8

We review the socket API in C.

socket() is the function which allows creating a socket.

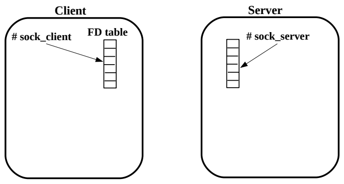
AF_UNIX is used for local (to a machine) communications, while AF_INET is used for remote communications.

The type should be SOCK_STREAM for connected communication (TCP) and SOCK_DGRAM for non connected communication (UDP). Sockets can also be used in RAW mode (direct access to the IP level).

The protocol to be used should be 0 for default protocols (TCP, UDP), but could be different if other protocols are installed.

Notice that socket() returns an integer which is a file descriptor.

After call to socket()



9

This is a representation of the states of the client and server processes after a call to socket() on both sides.

On both sides, an entry in the file descriptor table was allocated for the socket.

bind() function

- **int bind(int sock_desc, struct sockaddr *my_@, int lg_@)**
- **sock_desc: socket descriptor returned by socket()**
- **my_@: IP address and # port (local) that should be used**
- **Example (client or server):**

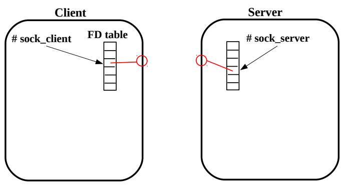
```
int sd;
struct sockaddr_in my_address; // @IP, #port, mode

sd = socket(AF_INET, SOCK_STREAM, 0);
my_address.sin_family = AF_INET;
my_address.sin_port = 0; // let system choose a port
my_address.sin_addr.s_addr = INADDR_ANY;
                              // any network interface

bind(sd, (struct sockaddr *)&my_address, sizeof(my_address));
```

10

The bind() function is invoked on both sides. It creates the association between a socket and a local port.

- sock_desc is the fd of the socket
- my_@ is a structure which describes initializations of the socket
  - sin_port = 0 means that bind() should allocate a free port
  - s_addr = INADDR_ANY means bind() can use any network interface (in case there are several network interfaces (cards))
- lg_@ is the size of the previous structure as it may differ depending on the OS

After call to bind()



We can already exchange messages in non-connected mode

11

This is a representation of the states of the client and server processes after a call to bind() on both sides.

On both sides, an socket in the file descriptor table is bound to a local port.

connect() function

- **int connect(int sock_desc, struct sockaddr * @_server, int lg_@)**
- **sock_desc: socket descriptor returned by socket()**
- **@_server: IP address and # port of the remote server**
- **Example of client:**

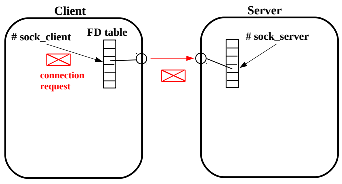
```
int sd;
struct sockaddr_in server; // @IP, #port, mode
struct hostent remote_host; // name et @IP

sd = socket(AF_INET, SOCK_STREAM, 0);
server.sin_family = AF_INET;
server.sin_port = htons(13);
remote_host = gethostbyname(“www.enseeiht.fr”); // DNS loookup
bcopy(remote_host->h_addr, (char *)&server.sin_addr,
      remote_host->hlength); // copy the address
connect(sd, (struct sockaddr *)&server, sizeof(server));
```

12

The connect() function is invoked on the client side. It sends a connection request to a remote server.

- sock_desc is the fd of the socket
- @_server is a structure which describes the remote server (@IP and port)
  - sin_port = the remote server port.
    htons (host to network) is a function which converts the port number (13) from a host representation to a network representation. This comes from the fact that an integer may have different representations on different hardware (little indian, big indian)
  - sin_addr = the @IP of the remote server
    gethostbyname() allows to obtain from DNS the IP from the machine name
    The IP address is a structure which has to be copied into the sin_addr structure.
- lg_@ is the size of the previous structure as it may differ depending on the OS

After call to connect()



This is a representation of the states of the client and server processes after a call to connect() on the client side.

A connection request has been sent from the client to the server.

listen() function

- **int listen(int sock_desc, int nbr)**
- **sock_desc: socket descriptor returned by socket()**
- **nbr: maximum number of pending connections**
- **Example of server:**

```
int sd;
struct sockaddr_in server; // @IP, #port, mode

sd = socket(AF_INET, SOCK_STREAM, 0);
server.sin_family = AF_INET;
server.sin_port = 0; // let system choose a port
server.sin_addr.s_addr = INADDR_ANY;
                           // any network interface
bind(sd, (struct sockaddr *)&server, sizeof(server));
listen(sd, 5);
```

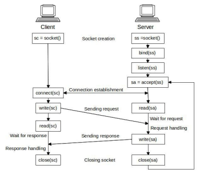

The listen() function is invoked on the server side to say that the socket will be used to receive connection requests and how many connection requests can be pending.

- sock_desc is the fd of the socket

- nbr is the number of tolerated pending connection requests (in a waiting queue). If the waiting queue is full, the connection from the client is rejected.

accept() function

- **int accept(int sock_desc, struct sockaddr *client, int lg_@)**
- **sock_desc: socket descriptor receiving connection requests**
- **client: identity of the client which requested the connection**
- **accept returns the socket descriptor associated with the accepted connection**

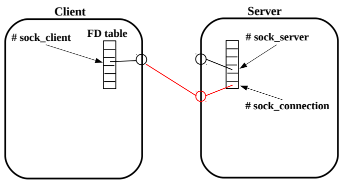


The accept() function is invoked on the server side. It blocks waiting for incoming connection requests. When a connection request is received, the blocked process is resumed and the function returns a new socket : the socket used to communicate with the client through the connection.

- sock_desc is the fd of the socket used to receive connection requests

- client is a structure which is updated with the identity (@IP, port) of the client who requested the connection.

After call to accept()



This is a representation of the states of the client and server processes after a connection has been accepted by the server.

In the server, a new socket (#sock_connection) was allocated and allows the server to communicate with the client through the connection.

Message emission/reception functions

```
• int write(int sock_desc, char *buff, int lg_buff);
• int read(int sock_desc, char *buff, int lg_buff);
• int send(int sock_desc, char *buff, int lg_buff, int flag);
• int recv(int sock_desc, char *buff, int lg_buff, int flag);
• int sendto(int sock_desc, char *buff, int lg_buff, int flag,
            struct sockaddr *to, int lg_to);
• int recvfrom(int sock_desc, char *buff, int lg_buff, int flag,
            struct sockaddr *from, int lg_from);

• flag : options to control transmission parameters
        (consult man)
```

17

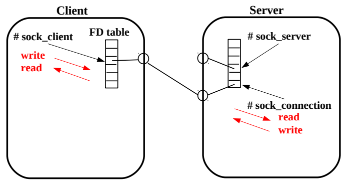
Many functions are available for sending and receiving messages (the list here is not exhaustive).

The first four only take a socket and buffer as parameters, so they are used for the connection mode.

The last two take a sockaddr structure, allowing to specify the address (IP and port) we are sending to or to know the address of the sender we are receiving from. So they are used for the non connected mode.

Many functions have flags for controlling their behavior.

Communication



18

This figure illustrates communication on a TCP connection.

Both the client and the server can use read/write functions on the sockets associated with the connection. The connection is bi-directional.

A concurrent server

- After fork() the child inherits the father's descriptors
- Example of server:

```
int sd, nsd;
...
sd = socket(AF_INET, SOCK_STREAM, 0);
...
bind(sd, (struct sockaddr *)&server, sizeof(server));
listen(sd, 5);
while (!end) {
     nsd = accept(sd, ...);
     if (fork() == 0) {
            close(sd); // the child doesn't need the father's socket

            /* here we handle the connection with the client */

            close(nsd); // close the connection with the client
            exit(0); // death of the child
     }
     close(nsd); // the father doesn't need the socket of the connction
}
```

19

This is a typical example of concurrent server. The server is concurrent as a child process is created for each accepted connection.

The server creates a socket, binds it to a local port, and calls listen().

It then loops and waits for incoming connections (accept()). For each received connection, accept() returns a new socket (nsd). For this new connection, the server creates a process (fork()). The child process handles data received on this connection. The father process loops and waits for another connection.

Programming Socket in Java

- package **java.net**
  - **InetAddress**
  - **Socket**
  - **ServerSocket**
  - **DatagramSocket / DatagramPacket**

20

We now study the socket API in the Java environment.

Sockets in Java are provided by the java.net package.

The main classes are :

- InetAddress

- Socket and ServerSocket for TCP

- DatagramSocket and DatagramPacket for UDP

Using `InetAddress` (1)

```
import java.net.*;
public class Enseeiht1 {

  public static void main (String[] args) {
    try {
      InetAddress address =
        InetAddress.getByName("www.enseeiht.fr");
      System.out.println(address);
    } catch (UnknownHostException e) {
      System.out.println("cannot find www.enseeiht.fr");
    }
  }
}
```

21

InetAddress allows invoking the DNS, translating with getByName() a machine name into an IP address. It returns an InetAddress instance which includes the IP address.

Using `InetAddress` (2)

```
import java.net.*;
public class Enseeiht2 {

  public static void main (String[] args) {
    try {
      InetAddress a = InetAddress.getLocalHost();
      System.out.println(a.getHostName() + " / " +
                         a.getHostAddress());
    } catch (UnknownHostException e) {
      System.out.println("No access to my address");
    }
  }
}
```

22

InetAddress also allows to obtain the InetAddress of the local host. The returned InetAddress instance includes the machine name and its IP address.

Client socket and TCP connexion

```
try {
   Socket s = new Socket("www.enseeiht.fr",80);
   …
} catch (UnknownHostException u) {
   System.out.println("Unknown host");
} catch (IOException e) {
   System.out.println("IO exception");
}
```

23

With TCP, a client can create a TCP connection with a target server (here www.enseeiht.fr), by creating an instance of the Socket class.

This operation corresponds to the calls in C of :

- socket()
- connect()

Reading/writing on a TCP connection

```
try {
   Socket s = new Socket ("www.enseeiht.fr",80);
   InputStream is = s.getInputStream();
   …
   OutputStream os = s.getOutputStream();
   …
} catch (Exception e) {
   System.err.println(e);
}
```

24

From this socket instance which is connected with the server, we can obtain 2 objects :

- an InputStream object which allows to read bytes
- an OutputStream object which allows to write bytes

With these objects, the client can send or receive data.

Server socket TCP connection

```
try {
    ServerSocket server = new ServerSocket(port);
    Socket s = server.accept();
    OutputStream os = s.getOutputStream();
    InputStream is = s.getIntputStream();
    …
} catch (IOException e) {
    System.err.println(e);
}
```


On the server side, the server can create a ServerSocket instance, giving a local port number as parameter. A ServerSocket instance is a socket for receiving connections. Therefore, this instanciation corresponds to the calls in C of :

- socket()
- bind()
- listen()

Then, the call of accept() on this instance blocks waiting for incoming connections. The process is resumed on connection reception, and accept() returns a Socket instance, which is the communication socket of the connection with the client. Like for the client side, we can obtain from this socket InputStream and OutputStream objects which provide communication methods.

Few words about classes for managing streams

- Suffix: type of stream
  - Stream of bytes (InputStream/OutputStream)
  - Stream of characters (Reader/Writer)
- Prefix: source or destination
  - ByteArray, File, Object …
  - Buffered, LineNumber, …

- https://www.developer.com/java/data/understanding-byte-streams-and-character-streams-in-java.html

InputStream and OutputStream are basic communication classes. They only allow to read and write bytes. They can be combined with many more elaborated classes.

The names of theses classes are composed of a prefix and a suffix.

The suffix indicates the type of the stream

- suffix = InputStream or OutputStream for streams of bytes
- suffix = Reader or Writer for a streams of characters (unicode representation)

The prefix indicates the source or destination of the stream

- examples are File or Object

For instance :

- a FileInputStream allows reading bytes from a file
- a FileWriter allows writing characters to a file

Few words about classes for managing streams

	Streams for reading	Streams for writing
Character streams	BufferedReader CharArrayReader FileReader InputStreamReader LineNumberReader PipedReader PushbackReader StringReader	BufferedWriter CharArrayWriter FileWriter OutputStreamWriter PipedWriter StringWriter
Byte streams	BufferedInputStream ByteArrayInputStream DataInputStream FileInputStream ObjectInputStream PipedInputStream PushbackInputStream SequenceInputStream	BufferedOutputStream ByteArrayOutputStream DataOuputStream FileOutputStream ObjetOutputStream PipedOutputStream PrintStream

Here is a table of the different classes.

Few words about classes for managing streams

```
BufferedReader br = new BufferedReader(
    new InputStreamReader(socket.getInputStream()));
String s = br.readLine();
```

- InputStreamReader: converts a byte stream into a character stream
- BufferedReader: implements buffering

```
PrintWriter pred = new PrintWriter(
    new BufferedWriter(
        new OutputStreamWriter(
            socket.getOutputStream())));
```

- PrintWriter: formatted printing

These classes can be combined (piped or chained) to obtain the desirable behavior.

In the first example, we obtain an InputStream from a socket. From this InputStream, we create a InputStreamReader which allows reading characters (Reader) from an InputStream (so it converts a stream of byte into a stream of characters). From this object, we create a BufferedReader, which provides buffering features like reading lines of characters.

In the second example, we obtain an OutputStream from a socket. From this OutputStream, we create a OutputStreamWriter which allows writing characters (Writer) to an OutputStream (so it converts a stream of characters into a stream of bytes). From this object, we create a BufferedWriter, which provides buffering features, and then a PrintWriter which provides formated printing (like println()).

Reading on a UDP socket

```
try {
  int p = 9999;
  byte[] t = new byte[10];
  DatagramSocket s = new DatagramSocket(p);
  DatagramPacket d = new DatagramPacket(t,t.length);
  s.receive(d);
  String str = new String(d.getData(), 0, d.getLength());
  System.out.println(d.getAddress()+"/"+d.getPort()+"/"+str);
  …
}
catch (Exception e) {
  System.err.println(e);
}
```

sender

A rapid look at programming UDP communication in Java.

On the receiving side, we create a DatagramSocket giving a local port number. It corresponds to the calls in C of : socket() and bind().

Then we can create a DatagramPacket giving an array of bytes.

Then

- receive() reads on the UDP socket and stores the data in the DatagramPacket
- getData() returns a byte array from the DatagramPacket (here we could have used the t variable)
- getLength() returns the size of the data actually received in the buffer
- getAddress() and getPort() return the address of the sender (IP and port), allowing to send a response.

Writing on a UDP socket

```
try {
   int p = 8888; // for receiving a response
   byte[] t = new byte[10];
   FileInputStream f = new FileInputStream("data.txt");
   int r = f.read(t);
   DatagramSocket s = new DatagramSocket(p);
   DatagramPacket d = new DatagramPacket(t, r,
     InetAddress.getByName("thor.enseeiht.fr"), 9999);
   s.send(d);
   …
} catch (Exception e) {
   System.err.println(e);
}
```

destination

On the sending side, we read in a byte array some data from a file.

We create a DatagramSocket giving a local port number.

Then we can create a DatagramPacket giving the array of bytes containing the data to send (r is the size of the data we read from the file), and also giving the destination (an InetAddress for the remote machine and a port from that machine). We send the packet with send().

A full example: TCP + serialization + threads

Passing an object (by value) with serialization

The object to be passed:

```
public class Person implements Serializable {
    String firstname;
    String lastname;
    int age ;
    public Person(String firstname, String lastname, int age) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.age = age;
    }
    public String toString() {
        return this.firstname+" "+this.lastname+" "+this.age;
    }
}
```


Here is an example of client server communication with TCP, with the creation of a thread in the server on connection reception, and with an object passed with serialization.

Serialization is a Java mechanism which allows an instance to be copied between remote hosts (e.g. from a client to a server). The instance is translated into a byte array on the source machine and the instance is reconstructed on the destination machine. Serialization applies recursively, meaning that instances referenced (by a field) from one serialized instance are also serialized (so we can serialize a graph of objects). To enable serialization, a class must implements the Serializable interface. Notice that a serializable class should not include references to non serializable objects (e.g. a system resource like Thread or Socket).

Here we describe a serializable class (Person) that we will use to demonstrate the transfer (copy) of an instance on a TCP connection.

A full example: TCP + serialization + threads

The client

```
public class Client {
   public static void main (String[] str) {
      try {
         Socket csock = new Socket("localhost",9999);
         ObjectOutputStream oos = new ObjectOutputStream (
                                    csock.getOutputStream());
         oos.writeObject(new Person("Dan","Hagi",53));
         csock.close();
      } catch (Exception e) {
               System.out.println("An error has occurred ...");
      }
   }
}
```


Here is the client side of the TCP example.

The main() method :

- creates a Socket which connects to a server located at localhost/9999
- from the OutputStream of the socket, it creates an ObjectOutputStream, which allows writing objects to an OutputStream. This ObjectOutputStream (oos) serializes objects and sends the data on the connection.
- writes a Person instance on oos. The instance is then serialized.
- finally closes the socket

A full example: TCP + serialization + threads

The server

```
public class Server {
   public static void main (String[] str) {
      try {
         ServerSocket ss;
         int port = 9999;
         ss = new ServerSocket(port);
         System.out.println("Server ready ...");
         while (true) {
            Slave sl = new Slave(ss.accept());
            sl.start();
         }
      } catch (Exception e) {
         System.out.println("An error has occurred ...");
      }
   }
}
```

33

Here is the server side of the TCP example.

The main() method :

- creates a ServerSocket bound to local port 9999
- then it loops on connection reception
  - accept() blocks and when resumed by a connection reception, it returns a Socket instance.
  - it creates a Slave instance (giving it a reference to the Socket instance) Slave is a class which implements a thread (explained next slide).
  - the thread is started

A full example: TCP + serialization + threads

The slave

```
public class Slave extends Thread {
   Socket ssock;
   public Slave(Socket s) {
      this.ssock = s;
   }
   public void run() {
      try {
         ObjectInputStream ois = new ObjectInputStream(
                                    ssock.getInputStream());
         Person v = (Person)ois.readObject();
         System.out.println("Received person: "+ v.toString());
         ssock.close();
      } catch (Exception e) {
         System.out.println("An error has occurred ...");
      }
   }
}
```

34

A way to program a thread is to implement a class which inherits from the Thread class. This class MUST implement the run() method which is invoked when the thread starts. NB : a thread is started with the start() method, not the run() method.

Here, the Slave class :

- inherits from Thread
- has a contructor to receive the socket it has to deal with
- implements a run() method which
  - creates an ObjectInputstream instance (ois) for reading objects from the stream of the socket. This ObjectInputstream instance reads data from the stream of the socket and deserializes the received objects.
  - reads an object on ois (the instance is deserialized) and casts it to Person (it is supposed to be a Person)

Conclusion

- Programming with sockets
  - Quite simple
  - Allow fine-grained control over exchanges messages
  - Basic, can be verbose and error prone
- Higher level paradigms
  - Remote procedure/method invocation
  - Message oriented middleware / persistent messages
  - ….

Many tutorials about socket programming on the Web …

Example : https://www.tutorialspoint.com/java/java_networking.htm

35

In conclusion, programming with sockets is more or less simple (complex with C, simple in Java). Its allows to do everything regarding distribution.

But for complex applications, even in Java, it may be really error prone.

This is why higher level programming paradigms were proposed.

In the next lectures, we will study some of them (remote invocation and message middleware).

Notice that many tutorials are available on the net for socket programming.

Client-server model

Daniel Hagimont

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

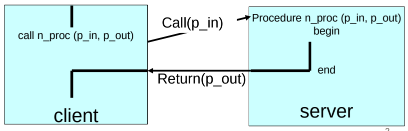
Daniel.Hagimont@enseeiht.fr
http://hagimont.perso.enseeiht.fr

1

This lecture is about the client server model. It reviews the concepts and illustrates them with its instanciation in the Java environment, with Remote Method Invocation (RMI).

Client-server model based on message passing

- Two exchanged messages (at least)
  - The first message corresponds to the request. It includes the parameters of the request.
  - The second message corresponds to the response. It includes the result parameters from the response.



2

Client-server interactions can be implemented with message passing (using sockets).

You then have at least 2 messages exchanged for such an interaction.

The first message corresponds to the request, including parameters, and the second message corresponds to the response, including result parameters.

The client's execution is suspended after sending of the request, until reception of the response.

We can observe that such an interaction looks like a procedure call, except that the caller (client) and the callee (server) are located on different machines.

Remote Procedure Call (RPC) Principles

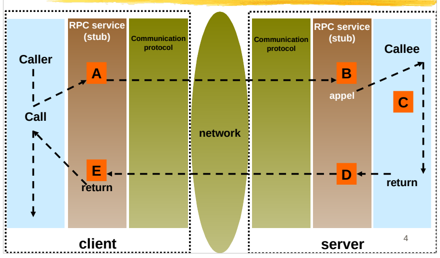
- Generating most of the code
  - Emission and reception of messages
  - Detection and re-emission of lost messages
- Objectives: the developer should be able to program the application without the burden to deal with messages

3

We call RPC (Remote Procedure Call) a tool which simplifies the development of applications relying on such client-server interactions, by generating the code which implements message exchanges (requests and responses). The idea is that all this code can be generated from a description of the interface of the procedure (which can be called on the server from a remote client).

The objective is to allow the developer to program and test his application as if it was centralized (executed on one machine) without the burden to deal with message exchanges. The code enabling the application to be distributed can be generated and the code of the application is kept simple.

RPC [Birrel & Nelson 84] Implementation principle



4

This is the general principle of RPC tools.

In blue, you have 2 code segments, the caller in the client which invokes (call) a service, and the callee in the server which provides the service.

In a centralized environment, the call would be a simple procedure call between the caller and the callee.

The principle of a RPC tool is to generate 2 code segments (in brown) called the client stub (left) and the server stub (right).

The client stub represents the service on the client machine and gives the illusion that the service is local (can be invoked locally with a simple procedure call). The client stub implements the same procedure as the server in order to give this illusion. A call to the procedure in the client stub creates and sends a request message to the server. The server stub receives and transforms request messages into local procedure calls on the server machine.

RPC (point A) Implementation principle

- On the caller side
  - The client makes a procedural call to the client stub
    - The parameters of the procedure are passed to the stub
  - At point A
    - The stub collects the parameters and assembles a message including the parameters (parameter marshalling)
    - An identifier is generated for the RPC call and included in the message
    - A watchdog timer is initialized
    - Problem: how to obtain the address of the server (a naming service registers procedures/servers)
    - The stub transmits the message to the transport protocol for emission on the network

5

On the caller side, the client performs a procedure call (invoking the service) as if the service was local to the client machine. Notice that the client stub implements the same procedure as the server, but the implementation of that procedure is different.

At point A, the client stub is called and receives the parameters from the procedure call. It assembles a request message which includes these parameters (this step is called parameter marshalling). An identifier for this RPC call is generated and included in the request message. This identifier allows to detect on the server side the reception of 2 requests for the same call (if the message is supposed to be lost and re-emitted).

A watchdog timer is initialized. It wakes up after a given time. If we don't receive a response before the wakeup, we consider that the request was lost and the request is re-emitted.

One problem here is to get the address (IP/port) of the server process for sending requests. Generally a naming service allows to register available procedures and their addresses.

The stub can then send the request message with the communication protocol (generally UDP as the data to be transmitted is not large).

The client is then suspended, waiting for the response message.

RPC (points B et C)
Implementation principle

- On the callee side
  - The transport protocol delivers the message to the RPC service (server stub)
  - At point B
    - The server stub disassembles the parameters (parameter unmarshalling)
    - The RPC identifier is registered
  - The call is then transmitted to the remote procedure which is executed (point C)
  - The return from the procedure returns back to the server stub which receives the result parameters (point D)

6

On the callee side, the communication protocol delivers the request message to the server stub. At point B, the server stub disassembles the parameters of the call (this step is called parameter unmarshalling). The RPC identifier is registered to detect redundant requests for the same call.

The call is then reproduced, i.e. the procedure to be called in the callee is actually called (point C). This is a normal procedure call. On return, the procedure returns back (point D) to the server stub (with some result parameters).

RPC (point D)
Implementation principle

- On the callee side
  - At point D
    - The result parameters are assembled in a message
    - Another watchdog timer is initialized
    - The server stub transmits the message to the transport protocol for emission on the network

7

At point D, the server stub assembles the result parameters in a response message.

Another watchdog timer is initialized. It wakes up after a given time. If we don't receive an acknowledgment from the client (that the response was received) before the wakeup, we consider that the response was lost and the response is re-emitted.

The server stub can then send the response with the communication protocol.

RPC (point E)
Implementation principle

- On the caller side
  - The transport protocol delivers the response message to the RPC service (client stub)
  - At point E
    - The client stub disassembles the result parameters (parameter unmarshalling)
    - The watchdog timer created at point A is disabled
    - An acknowledgment message with the RPC identifier is sent to the server stub (the watchdog timer created at point D can be disabled)
    - The result parameters are transmitted to the caller with a procedure return

8

On the caller side, the communication protocol delivers the response message to the client stub.

At point E, the client stub disassembles the result parameters (parameter unmarshalling).

The watchdog timer created at point A can be disabled.

An acknowledgment message with the RPC identifier is sent to the server stub (the watchdog timer created at point D can be disabled).

The result parameters are transmitted to the caller with a procedure return.

RPC
Role of stubs

<u>Client stub</u>

- It is the procedure which interfaces with the client
  - Receives the call locally
  - Transforms it into a remote call with a sent message
  - Receives results in a message
  - Returns results with a normal procedure return

<u>Server stub</u>

- It is the procedure on the server node
  - Receives the call as a message
  - Performs the procedure call on the server node
  - Receives the results of the call locally
  - Transmits the results remotely as a message

9

We summarize here the role of the client stub and the server stub.

RPC
Message loss

- On the client side
  - If the watchdog expires
    - Re-emission of the message (with the same RPC identifier)
    - Abandon after N attempts
- On the server side
  - If the watchdog expires
  - Or if we receive a message with a known RPC identifier
    - Re-emission of the response message
    - Abandon after N attempts
- On the client side
  - If we receive a message with a known RPC identifier
    - Re-emission of the acknowledgment message

We provide here a global view of the handling of message loss.

On the client side, we created a watchdog before sending the request. If this watchdog expires, we can suppose that the request was lost and we re-send the request with the same RPC identifier (we re-initialize the watchdog before sending). We abandon after N attempts, assuming that the network is down.

On the server side, we create a watchdog before sending the response. As previously, we re-send the response if the watchdog expires. Another case on the server side is when we receive a request with a known RPC identifier (requests are logged). This means that we already received this request and the procedure was called and the response sent, but the response was lost. Then we re-send the response. As previously, we abandon after N attempts.

Finally, on the client side, if we receive a response with a known RPC identifier (response are logged), i.e. a response that we already received, it means that the acknowledgment sent to the server was lost and we re-send it.

RPC
Problems

- Failure handling
  - Network or server congestion
    - The response arrives too late (critical systems)
  - The client crashes during the request handling on the server
  - The server crashes during the handling of the request
  - Failure of the communication system
  - What guarantees ?
- Security problems
  - Client authentication
  - Server authentication
  - Privacy of exchanges
- Performance
- Designation
- Practical aspects
  - Adaptation to heterogeneity conditions (protocols, languages, hardware)

Many other problems can be handled by RPC systems.

The handling of failures covers many types of failure :

- dealing with network or server congestion. Messages may be re-emitted, but redundant messages must be managed. In a real-time system, the execution time of a procedure is specified and the procedure should return an error if the deadline is not respected.

- dealing with the crash of the client or the server during the handling of the request, or the failure of the communication system. The system should provide guarantees (e.g. transactional behavior).

A RPC tool may also integrate security features, like authentication and encryption of exchanges.

Many other aspects were also considered :

- performance of RPC, especially the optimization when the client and server processes are on the same machine, or on the same LAN.

- designation : different designation scheme can be provided, for identifying the target (process) of call.

- heterogeneity : a lot of work was done to enable heterogeneity (of languages, OS …) between the caller and callee (see CORBA).

RPC
IDL : interface specification

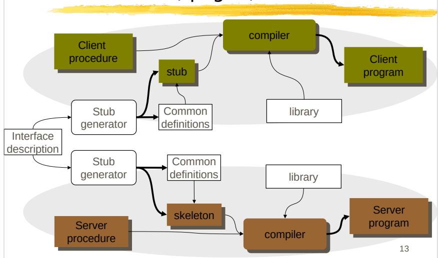
- Use of an interface description language (IDL)
  - Specification which is common to the client and the server
  - Definition of parameter types et natures (IN, OUT, IN-OUT)
- Use of the IDL description to generate:
  - The client stub (also called proxy or stub)
  - The server stub (also called skeleton)

Generally, a RPC tool generates stubs from the specification of the interface of the procedure which can be called remotely.

An IDL (Interface Description Language) is a simple language for describing the interface of a procedure which can be called through a RPC system. It simply allows describing the signature of the procedure, including the type of the parameters (data structures).

Such a specification allows to generate the client stub (sometimes called proxy or simply stub) and the server stub (often called skeleton).

RPC
Functional mode (rpcgen)



rpcgen is one of the first RPC tools which was available in a Unix/C environment.

From the interface description (expressed with the IDL), a stub generator generates both the sub and skeleton.

On the client side, the client procedure (caller) is compiled with the stub in order to obtain an executable binary (client program).

On the server side, the server procedure (callee) is compiled with the skeleton in order to obtain a executable binary (server program).

These 2 binaries can be installed on different machines and executed.

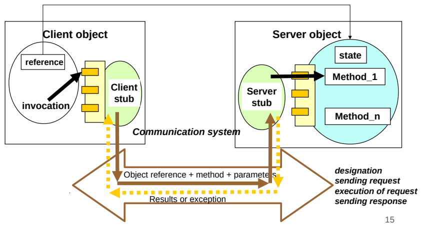
Java Remote Method Invocation
RMI

- An object based RPC integrated within Java
- Interaction between objects located in different address spaces (*Java Virtual Machines* - JVM) on remote machines
- Easy to use: a remote object is invoked as if it was local

Java RMI (Remote Method Invocation) is an example of implementation of a RPC tool integrated in a language environment (here Java).

It allows the invocation of methods on instances located on remote machines (in a remote JVM). Such a remote method invocation is programmed as if the target object was local to the current JVM.

Java RMI
Principle

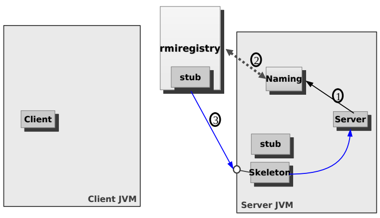


The general principle of Java RMI is illustrated in this figure.

A client object in one JVM (left) includes a reference to a server object (remote) in another JVM (right).

This reference is actually a reference to a local stub object (client stub). This stub transforms a method call into a request message (which includes an object reference to identify the object in the server JVM, an method identifier and the parameters of the call). This request message is received by a skeleton object (server stub) which performs the actual method call on the server object.

Java RMI
Server side



We describe the general functioning the RMI before describing its programming model.

We assume that a Server class has been programmed following the RMI programming model.

On the server side, when the Server class is instantiated, stub and skeleton objects are instantiated The skeleton object is associated with a local port of the machine for receiving requests.

In order to make the Server object accessible from clients, it must be registered in a naming service called rmiregistry (the rmiregistry runs in another JVM). This registration is possible thanks to the Naming class which provides a bind method (which registers the association between a name ("foo") and the Server object).

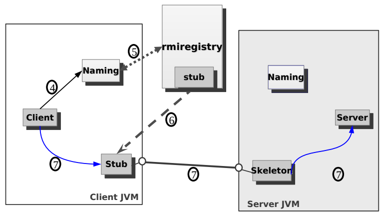
This registration in the rmiregistry makes a copy of the stub in the rmiregistry (and registers its association with "foo"). The rmiregistry is ready to deliver copies of the stub to clients.

Java RMI
Server side

- 0 – At object creation time, a *stub* and a *skeleton* (with a communication port) are created on the server
- 1 – The server registers its instance with a naming service (*rmiregistry*) using the *Naming* class (*bind* method)
- 2 – The naming service (*rmiregistry*) registers the *stub*
- 3 – The naming service is ready to give the *stub* to clients

We recall here the main step of creation of a Server object and registration in the rmiregistry.

Java RMI
Client side



On the client side, the client can fetch a reference to the Server object from the rmiregistry. This is possible thanks to the Naming class which provides a lookup method (which queries the object registered with a name ("foo")).

The query on the rmiregistry returns a copy of the stub (associated with "foo"). This stub implements the same interface as the Server object. It can be used by the client to invoke a method. The stub creates and sends a request message to the skeleton which performs the actual call on the Server object.

Java RMI
Client side

- 4 – The client makes a call to the naming service (*rmiregistry*) using the *Naming* class to obtain a copy of the stub of the server object (*lookup* method)
- 5 – The naming service delivers a copy of the *stub*
- 6 - The *stub* is installed in the client and its Java reference is returned to the client
- 7 – The client performs a remote invocation by calling a method on the *stub*

We recall here the main step of querying the rmiregistry and performing a method invocation.

Java RMI
Utilization

- Coding
  - Writing the server interface
  - Writing the server class which implements the interface
  - Writing the client which invokes the remote server object
- Compiling
  - Compiling Java sources (javac)
  - Generation of *stubs* et *skeletons* (rmic)
    - *(not required anymore, dynamic generation)*
- Execution
  - Launching the naming service (*rmiregistry*)
  - Launching the server
  - Launching the client

Here are the main steps for using RMI.

Regarding coding :

- you must define the Java interface of the Server. This interface is used both by the Server and the Client.

- the Server class implements the previous interface. The Server is instantiated and the instance is registered in the rmiregistry.

- the Client can declare a variable whose type is the previous interface. The Client obtains a copy of the stub from the rmiregistry. The stub implements the interface. The Client can call a method on this stub.

Regarding compiling :

- the application is compiled with javac as usually

- the stub and skeleton classes can be generated with rmic (a stub generator). This is not necessary anymore on recent versions of Java, the stubs being generated dynamically when needed.

Regarding execution :

- you have to launch the rmiregistry

- then you can launch the server and then the client

Java RMI
Programming

- Programming a remote interface
  - public interface
  - interface: extends java.rmi.Remote
  - methods: throws java.rmi.RemoteException
  - serializable parameters: implements Serializable
  - references parameters: implements Remote
- Programming a remote class
  - implements the previous interface
  - extends java.rmi.server.UnicastRemoteObject
  - same rules for methods

Programming RMI applications comes with programming constraints.

For the interface of the Server :

- the interface must be public

- the interface must implement the Remote interface

- all the methods must throw RemoteException

- parameters of remote methods can be of built-in type (int, char), or a Java reference. In this last case, their type must be an interface which is either Serializable or Remote (this is detailed later).

For the Server class :

- it must implement the previous interface

- it must extend the UnicastRemoteObject class

- same rules for methods (as in the previous interface)

Java RMI
Example: interface

file Hello.java

```
public interface Hello extends java.rmi.Remote {
  public void sayHello()
          throws java.rmi.RemoteException;
}
```

Description of the interface

We review a very simple example.

Here is the definition of the interface.

Interface Hello implements Remote and throws RemoteException.

Java RMI
Example: server

file HelloImpl.java

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject
                              implements Hello {
    String message;

   // Constructor implementation
   public HelloImpl(String msg) throws java.rmi.RemoteException {
         message = msg;
   }
  // Implementation of the remote method
   public void sayHello() throws java.rmi.RemoteException {
          System.out.println(message);
   }

      ...
```

Implementation of the server class

Here is the code of the server class.

Class HelloImpl extends UnicastRemoteObject and implements interface Hello.

Your constructors must throw RemoteException.

The remote method sayHello() throws RemoteException.

Java RMI
Example: server

file HelloImpl.java

```
     ...

   public static void main(String args[]) {
         try {
                  // Create an instance of the server object
                  Hello obj = new HelloImpl("hello");
                  // Register the object with the naming service
                  Naming.rebind("//my_machine/my_server", obj);
                  System.out.println("HelloImpl " + " bound in registry");
         } catch (Exception exc) {... }
   }
}
```

Implementation of the server class

NOTICE : in this example, the naming service (rmiregistry) must have been launched before execution of the server

The rest of the code of the server.

The main method creates an instance of the server class (HelloImpl) and registers it in the rmiregistry, thanks to the Naming class.

The URL passed in the rebind() method is //<machine-name>:<port>/<name>

- machine-name is the name of the machine which runs the rmiregistry
- port is the port used by the rmiregistry (the default port is 1099)
- name is the name identifying the registered object in the rmiregistry

In its implementation in Java, the rmiregistry has to be colocated (on the same machine) with the JVM which runs the server object. A work around is to implement another rmiregistry (allowing remote registrations).

Notice that after the registration, this is the end of the main method and the JVM would exit. This is not the case, since when we instantiated the server object, a skeleton was instantiated with creation of a communication socket and of a thread waiting for incoming requests. Because of that thread, the JVM does not exit.

Here, we assume that the rmiregistry was launched (rmiregistry is an executable) on the same machine as the server object, with the command :

rmiregistry <port> (default is 1099)

Java RMI
running the rmiregistry within the server JVM

file HelloImpl.java

```
public static void main(String args[]) {
   int port;    String URL;

   try {
     Integer I = new Integer(args[0]); port = I.intValue();
   } catch (Exception ex) {
     System.out.println(" Please enter: java HelloImpl <port>"); return;
   }

   try {
     // Launching the naming service – rmiregistry – within the JVM
     Registry registry = LocateRegistry.createRegistry(port);

     // Create an instance of the server object
     Hello obj = new HelloImpl();

     // compute the URL of the server
     URL = "//"+InetAddress.getLocalHost().getHostName()+":"+
                              port+"/my_server";
     Naming.rebind(URL, obj);
   } catch (Exception exc) { ...}
}
```


In this other version, we launch a rmiregistry in the same JVM as the one hosting the server object.

The createRegistry method launches a rmiregistry within the local JVM on the specified port.

The interest of doing so is that when you start the application, a rmiregistry is automatically launched and when you kill the JVM, the rmiregistry is also killed. This is very convenient when debugging.

Java RMI
Example: client

file HelloClient.java

```
import java.rmi.*;

public class HelloClient {
  public static void main(String args[]) {
    try {
      // get the stub of the server object from the rmiregistry
      Hello obj = (Hello) Naming.lookup("//my_machine/my_server");
      // Invocation of a method on the remote object
      obj.sayHello();
    } catch (Exception exc) { … }
  }
}
```

Implementation of the client class

26

Here is the code on the client side.

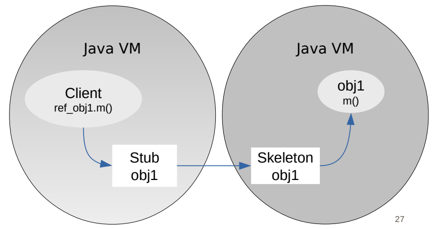
It first requests a reference to the target object from the rmiregistry, using the lookup method from the Naming class (the used URL is the same as before.

Notice that here the client can be executing on a different machine.

The rmiregistry returns a stub instance. This stub instance implements the same interface as the server object (here Hello). So we can cast the obtained reference with the Hello interface.

Then, invoking a method on the remote object is programmed as if the object was local.

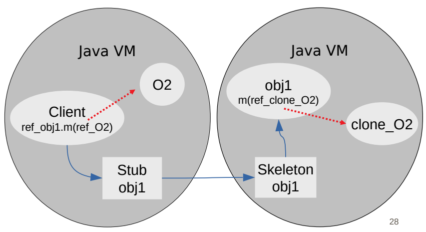
Java RMI
Principle of remote method invocation



27

To summarize the functioning of Java RMI, a client which obtained (from the rmiregistry) a remote reference (ref_obj1) to a remote object (obj1) has actually a reference to a local stub object (Stub obj1). The client can invoke a method m() on the remote object. It will invoke this method on the stub, which will send the request message. This message is received by the skeleton (Skeleton obj1) which performs the actual invocation on the server object.

Java RMI
Serializable object parameter passing



28

Parameters passed in a remote method can be of built-in types (int char …). Then the parameters are simply copied (transferred) in the remote server.

If a parameter is a Java reference to an object, the type of the parameter in the method signature must be an interface. Then, there are 2 possibilities:

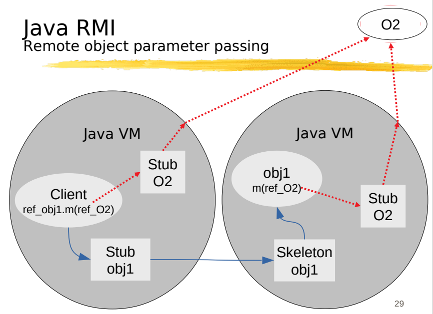
- Serializable. If the interface is serializable (inherits from Serializable), then the passed object is copied to the server (the object is cloned).

- Remote. If the interface is Remote (inherits from Remote), then the remote reference (i.e. the stub) is passed to the server, meaning that the stub is copied in the server. Therefore, the passed object becomes accessible remotely in the server.

- if the interface is neither Serializable nor Remote, this is an error (it should not compile).

This figure illustrates the Serializable case. The client passes as parameter a reference to object O2 which is local in the client. Then, O2 is copied to the server and the invoked method (m) receives a reference to a clone of object O2 in the server.

Java RMI
Remote object parameter passing



29

This figure illustrates the Remote case. The client passes as parameter a reference to object O2 which is remote (in another JVM). It means that the reference to O2 in the client is a local reference to a sub of O2. Then, the stub of O2 is copied to the invoked server and the invoked method (m) receives a reference to a copy of stub of O2 in the server. Therefore, m() receives a remote reference to O2.

Java RMI
Compiling

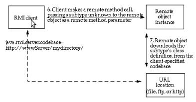
- Compiling the interface, the server and the client
  - javac Hello.java HelloImpl.java HelloClient.java
- Generation of stubs (*not needed anymore*)
  - rmic HelloImpl
    - *skeleton* in HelloImpl_Skel.class
    - *stub* in HelloImpl_Stub.class

To execute the application, you must first compile the interface and the server and client classes.

As previously mentioned, generating stubs and skeletons is not necessary anymore, but you can still do it.

Java RMI
Deployment

- Launching the naming service
  - rmiregistry &
- launching the server
  - java HelloImpl
  - java -Djava.rmi.server.codebase=http://my_machine/...
    - URL of a web server from which the client JVM will be able to download missing classes
    - Example: serialization
- Launching the client
  - java HelloClient



Here we explicitly launch the rmiregistry in a shell.

Then, we can launch the server and then the client.

One tricky issue is the availability of classes. Assume the client invokes a method m(Data d) on the server, Data being an interface which is Serializable. Both the client and the server know the interface Data (it was necessary to use the method m and to compile the code). Then the client may invoke m passing an instance of class ClientData (which implements Data). But the server which receives a copy of the object does not have the ClientData class (and different clients may have different implementations of the Data interface).

More generally, a JVM may transfer copies of objects (with Serialization) to other JVMs. How can the first JVM make these classes available to other JVMs. The solution is to specify, when launching a JVM, a web site from which classes can be downloaded. When classes are missing for using a serialized object, the classes are downloaded and installed dynamically.

java -Djava.rmi.server.codebase =URL <a class>

When launching a JVM this way, we specify that if serialized instances are given to other JVMs, the missing classes can be found on the web site defined by URL.

Java RMI: conclusion

- Very good example of RPC
  - Easy to use
  - Well integrated within Java
  - Java reference parameter passing: serialization or remote reference
  - Deployment: dynamic loading of serializable classes
  - Designation with URL

Many tutorials about RMI programming on the Web …

Example : https://www.tutorialspoint.com/java_rmi/java_rmi_application.htm

To conclude this lecture, Java RMI is a very example of RPC integrated in a Java.

Many tutorials about Java RMI can be found on the net.

Web Services

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 web services.

Motivations

- Motivations
  - Coarse-grained application integration
  - Unit of integration: the "service" (interface + contract)
- Constraints
  - Applications developed independently, without anticipation of any integration
  - Heterogeneous applications (models, platforms, languages)
- Consequences
  - No definition of a common model
  - Elementary common basis
    - For communication protocols (messages)
    - For the description of services (interface)
  - Base choice: XML (because of its adaptability)

The example of RPC tool we have seen, Java RMI, is restricted to interactions within Java applications, allowing remote invocations of Java objects.

With Web services, the motivation is to provide a RPC facility for the interaction (and integration) of coarse-grained applications (that we call services). A service is supposed to be much bigger than a simple Java object.

Web Services (WS)

- Conceptual contribution
  - No new fundamental concept …
  - … so, what for ?
- Concrete contribution
  - Practically address the heterogeneity problem
  - Large-scale (world wide) integration of application
  - Heavy implication of main IT actors

No new conceptual concepts here, but justified by

- it addresses the problem of heterogeneity. Providers and consumers may be of different organizations and use different languages, OS …

- it was pushed by the main IT actors

Basic form of WS : XML-RPC

Description in XML of a remote procedure call
Parameter types are specified in an XML schema

```
<methodCall>
   <methodName>meteo.temperature</methodName>
   <params>
      <param>
         <value><int>31130</int></value>
      </param>
   </params>
</methodCall>
```

Description in XML of parameter retuns

```
<methodResponse>
   <params>
      <param>
         <value><int>25</int></value>
      </param>
   </params>
</methodResponse>
```

Interest : independence with respect to platforms and communication protocols

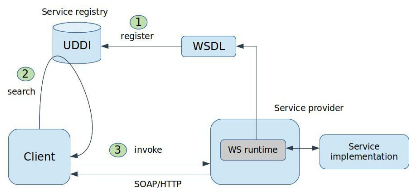
XML-RPC was a precursor of what are web services now.

XML-RPC was a RPC protocol relying of XML for the representation of requests and HTTP for the transport of requests.

The idea was to be independent from execution platforms or languages and to rely on widely recognized and adopted formats.

XML-RPC was a precursor and evolved into SOAP, the protocol used in web services.

Architecture of WS



This figure illustrates the architecture of web services (WS).

A service provider may implement a service in any language and/or platform, as soon as a runtime for WS exists in his environment.

The runtime is a composed of

- stub and skeleton generators

- a WSDL generator

- a web server for making services available on the internet

Then, on the server side, the service implementation is linked with the web server, in order to be able to receive requests through the HTTP communication protocol. A skeleton is generated and is a web application in the web server. A WSDL description (Web Service Description Language) of the service is generated and published, i.e. made available to potential clients.

The WS architecture specifies that a service registry (a naming service) should be used for the publication and discovery of WSDL descriptions. However, UDDI was not actually used. Generally the WSDL of a WS can be published on a Web server as any document.

On the client side, the WSDL description can be copied and used to generate a stub in the environment of the client. Notice that the environment of the client is not mandatorily the same as the one of the server. Then the client can implement an application which is able to invoke the WS by calling the stub.

The stub communicates with the skeleton with the SOAP/HTTP protocol which is a standard.

HTTP and SOAP are standards from the W3C.

SOAP describes the syntax of request and response messages which are transported with HTTP.

Elements of WS

- Description of a service
  - WSDL : Web Services Description Language
  - Standard notation for the description of a service interface
- Access to a service
  - SOAP : Simple Object Access Protocol
  - Internet protocol allowing communication between Web Services
- Registry of services
  - UDDI : Universal Description, Discovery and Integration
  - Protocol for registration and discovery of services

Therefore the main elements of WS are :

- the description of the service in WSDL. Generally, from an implementation of a service (e.g. a procedure), tools are provided to generate the WSDL description of the service, which is published for clients. The clients can used this WSDL description to generate stubs so that calls to the service can be programmed easily.

- access protocols which are SOAP (for the content of messages) and HTTP (for the transport). All the WS runtimes (in any environment) comply with these standards.

- registries of service (UDDI) which are not really used.

Tools

- From a program, we can generate a WS skeleton
  - Example: from a Java program, we generate
    - A servlet which receives SOAP/HTTP requests and reproduces the invocation on an instance of the class
    - A WDSL file which describes the WS interface
- The generated WSDL file can be given to clients
- From WSDL file, we can generate a WS stub
  - Example: from a WSDL file, we generate Java classes which can be used to invoke the remote service
- Programming is simplified
- Such tools are available in different langage environments

To illustrate this, we give an example of use in the Java environment.

In the Java environment, a WS tool is used to generate from a program (with an exported interface) a skeleton as a servlet. A servlet is a Java program which runs in a web server. This servlet/skeleton received SOAP/HTTP requests and reproduces the invocation on an instance of the class. The WSDL specification of the WS is also generated.

The WSDL file is published and imported by the client.

From the WSDL specification, the client can generate stubs which make it easier to program WS invocations.

In the following slides, we give an example with Apache Axis.

Example: programming a Web Services

- Eclipse JEE
- Apache Axis
- Creation of a Web Service
  - From a Java class
  - In the Tomcat runtime
  - Generation of the WSDL file
- Creation of a client application
  - Generation of stubs from a WSDL file
  - Programming of the client

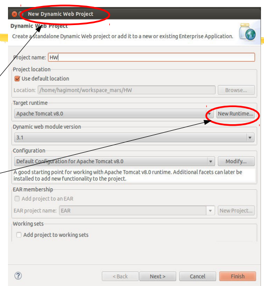
We use Eclipse JEE and Apache Axis which is available in Eclipse JEE.

Apache Axis is used to generate from a Java class a servlet which is installed in the Tomcat engine (the web server). It also generates the WSDL description which describes the interface of the WS.

On the client side, the WSDL description is used to generate stubs which are used to invoke the WS in a client program.

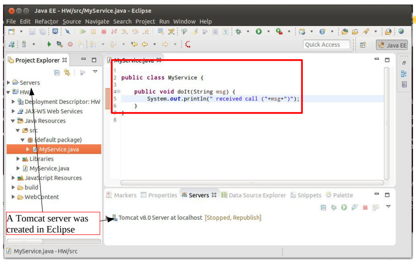
Create a Dynamic Web Project

- Eclipse JEE
- Open JEE perspective
- Create a Dynamic Web Project
- Add your Tomcat runtime



In Eclipse, we create a dynamic web project (a project allowing the develop servlets) and add the Tomcat runtime.

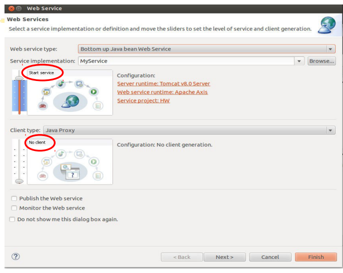
Create a Class



In this project, we create a class.

Notice that a Tomcat server is running in Eclipse.

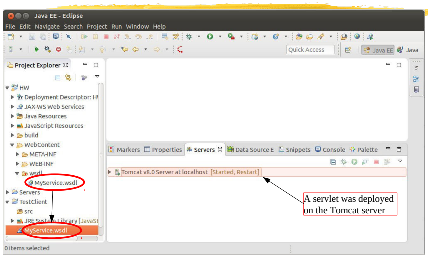
From source file : Web Service → create Web Service



11

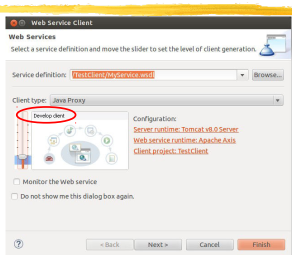
From the source file of the class, we can generate (right click) a WS from this file.

Copy the generated WSDL file in a new Java project



Then, we create a new Java project and copy the WSDL description in the new project.

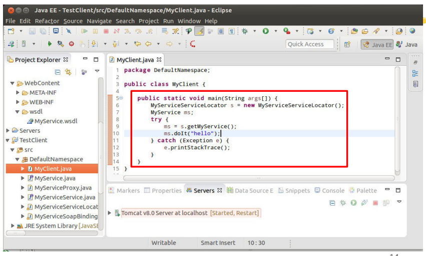
From the WSDL file Web Service → Generate Client (Develop Client)



13

In the new project, from the WSDL file, we generate (right click) the stubs (develop Client).

Program a client

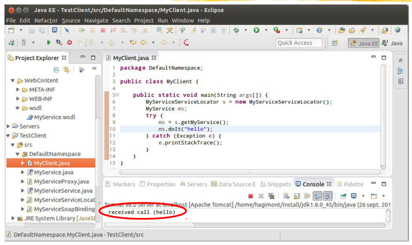


14

In the new project, we can program an application which makes an invocation of the WS.

The procedure to follow to invoke the WS depends on the tool used (here Apache Axis).

Run



We can then run the client program which invokes the WS.

Generated WSDL

```
<wsdl:definitions targetNamespace="http://DefaultNamespace"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://DefaultNamespace"
xmlns:intf="http://DefaultNamespace" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://DefaultNamespace"
xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="doIt">
    <complexType>
     <sequence>
      <element name="msg" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="doItResponse">
    <complexType/>
   </element>
  </schema>
 </wsdl:types>
```

16

We can have a look at the WSDL description.

We can see that the WSDL syntax is not very simple. Therefore such WSDL descriptions are not written by the user, but generally generated by the tool on the server side and imported by the client.

Generated WSDL

```
<wsdl:message name="doItResponse">
   <wsdl:part element="impl:doItResponse" name="parameters">
   </wsdl:part>
 </wsdl:message>
 <wsdl:message name="doItRequest">
   <wsdl:part element="impl:doIt" name="parameters">
   </wsdl:part>
 </wsdl:message>
 <wsdl:portType name="MyService">
   <wsdl:operation name="doIt">
     <wsdl:input message="impl:doItRequest" name="doItRequest">
    </wsdl:input>
     <wsdl:output message="impl:doItResponse" name="doItResponse">
    </wsdl:output>
   </wsdl:operation>
 </wsdl:portType>
```

17

Very verbose !

Generated WSDL

```xml
<wsdl:binding name="MyServiceSoapBinding" type="impl:MyService">
    <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="doIt">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="doItRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="doItResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="MyServiceService">
    <wsdl:port binding="impl:MyServiceSoapBinding" name="MyService">
      <wsdlsoap:address location="http://localhost:8080/HW/services/MyService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```


Very very verbose !

SOAP request(with TCP/IP Monitor)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance">

    <soapenv:Body>
        <doIt xmlns="http://DefaultNamespace">
            <msg>hello</msg>
        </doIt>
    </soapenv:Body>

</soapenv:Envelope>
```


We can have a look at the SOAP request. This is simply a standardized format for exchanged messages.

SOAP response

```xml
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <soapenv:Body>
        <doItResponse xmlns="http://DefaultNamespace"/>
    </soapenv:Body>

</soapenv:Envelope>
```


Here is the SOAP response.

REST Web Services

- A simplified version, not a standard, rather a style
- Use of HTTP methods
  - GET : get data from the server
  - POST : create data in the server
  - PUT : update data in the server
  - DELETE : delete data in the server
- Invoked service in the URL : <URL>/service
- Parameter passing
  - HTTP parameters
  - XML or JSON
- Many development environments
  - Examples : resteasy, jersey

SOAP/WSDL based WS were very popular few years ago. They are now obsolete.

An evolution of WS is REST WS. This is a simplified version which is very popular now. Notice the REST WS is not a standard, but rather a recommendation or a style of implementation.

It relies on HTTP requests (GET, POST, PUT, DELETE), but mainly GET and POST are used. GET is used when you want to read (only) data from the WS while POST is used when you want to modify something in the WS.

The service that you call is encoded in the URL : <url>/service

Parameter passing can be based on HTTP parameters, XML or JSON (in the body of requests or responses).

The description of a REST WS is simply a document describing the services that you may call and the passed parameters (names, formats).

Example of existing REST WS

Currency API Request

The base URL for our currency API is

`https://www.amdoren.com/api/currency.php`

Request Parameters

Parameter	Description
api_key	Your assigned API key. This parameter is required.
from	The currency you would like to convert from. This parameter is required.
to	The currency you would like to convert to. This parameter is required.
amount	The amount to convert from. This parameter is optional. Default is a value of 1.

Example:
To get the latest exchange rate in EUR for 1 USD:

`https://www.amdoren.com/api/currency.php?api_key=IBZzdLmM2yCYaXjgTZ6x&from=USD&to=EUR`

Currency API Response

Element	Description
error	Error code. Value greater than zero indicates an error. See list below.
error_message	Short decription of the error. See list below.
amount	The exchange rate or amount converted.

Example:
JSON data returned from our currency API request:

{ "error" : 0, "error_message" : "-", "amount" : 0.90168 }

- www.amdoren.com
- Currency converter

22

Here is an example of description of a REST WS. This is for a currency converter.

It says that you have one service available :

https://www.amdoren.com/api/currency.php

It lists the parameters that may be passed in the HTTP GET request. A example is given.

It then describes the response which is a JSON. A example is given.

Example with Resteasy (server)

- WS class

```
@Path("/")
public class Facade {

    static Hashtable<String, Person> ht = new Hashtable<String, Person>();

    @POST
    @Path("/addperson")
    @Consumes({ "application/json" })
    public Response addPerson(Person p) {
         ht.put(p.getId(), p);
         return Response.status(201).entity("person added").build();
    }

    @GET
    @Path("/getperson")
    @Produces({ "application/json" })
    public Person getPerson(@QueryParam("id") String id) {
         return ht.get(id);
    }

    @GET
    @Path("/listpersons")
    @Produces({ "application/json" })
    public Collection<Person> listPersons() {
         return ht.values();
    }

}
```

Receives a JSON
Deserialized into a Java object
void method

Returns an object
Serialized into a JSON
Receives an id parameter

Person is a simple POJO

23

As for SOAP/WS, many tools were implemented to help developers.

Here, we present Resteasy (Jersey is also a very popular one you may look at).

On the server side, you can use annotations in a Java program to say :

- each method is associated with a path in the URL used to access the WS

- @Path : specifies the element of the path associated with the class or the method. Here method addPerson() is associated with path /addperson

- @POST or @GET : specifies which HTTP method is used. Notice that GET returns an object (data) while POST returns an HTTP code (and a message).

- @Consumes : specifies that we receive a JSON object which is deserialized into a Java object.

- @Produces : specifies that we return a Java object which is serialized into a JSON object.

- @QueryParam : the getPerson() method has an "id" parameter. The QueryParam annotation associates this parameter with an "id" HTTP parameter.

Example with Resteasy (server)

- Add the RestEasy jars in Tomcat
- In eclipse
  - Create a Dynamic Web Project
  - Add RestEasy jars in the buildpath
  - Create a package
  - Implement the WS class (Facade + Person)
  - Add a class RestApp

```
public class RestApp extends Application {
        private Set<Object> singletons = new HashSet<Object>();
        public RestApp() {
                singletons.add(new Facade());
        }
        Public Set<Object> getSingletons() {
                return singletons;
        }
}
```

24

To run this example :

- add the Resteasy jars in Tomcat and Eclipse

- create a dynamic web project (a servlet project)

Example with Resteasy (server)

- Add a web.xml descriptor in the WebContent/WEB-INF folder

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" version="3.1">
  <display-name>essai-server</display-name>
  <servlet>
    <servlet-name>resteasy-servlet</servlet-name>
    <servlet-class>
            org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
        </servlet-class>
    <init-param>
      <param-name>javax.ws.rs.Application</param-name>
      <param-value>pack.RestApp</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>resteasy-servlet</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

- Export the war in Tomcat

25

Add the descriptor and export a war

Publish the WS

- Just write a documentation which says that
  - The WS is available at http://localhost:8080/
  - Method addperson with POST receives a person JSON :

```json
{
   "firstname":"Alain",
   "lastname":"Tchana",
   "phone":"0102030405",
   "email":"alain.tchana@enseeiht.fr"
}
```

  - Method getperson with GET receives an id and returns a person
  - Method listperson returns a JSON including a set of persons
- A user may use any tool (not only RestEasy)

Publication of a REST WS is simply a document describing the interface.

Example with Resteasy (client)

- From a documentation of REST WS we can write the interface

```java
@Path("/")
public interface FacadeInterface {

    @POST
    @Path("/addperson")
    @Consumes({ "application/json" })
    public Response addPerson(Person p);

    @GET
    @Path("/getperson")
    @Produces({ "application/json" })
    public Person getPerson(@QueryParam("id") String id);

    @GET
    @Path("/listpersons")
    @Produces({ "application/json" })
    public Collection<Person> listPersons();

}
```


On the client side, from the documentation, a user can write a Java interface with Resteasy annotations. Of course, it's very similar to what we wrote on the server side, but we could do it for a WS we don't know (we only have the documentation).

Example with Resteasy (client)

- And write a class which invokes the WS

```java
public class Client {

   public static void main(String args[]) {

       final String path = "http://localhost:8080/rs-server-person/rest";

       ResteasyClient client = new ResteasyClientBuilder().build();
       ResteasyWebTarget target = client.target(UriBuilder.fromPath(path));
       FacadeInterface proxy = target.proxy(FacadeInterface.class);

       Response resp;
       resp = proxy.addPerson(new Person("007","James Bond"));
       System.out.println("HTTP code: " + resp.getStatus()
                                 +"  message: "+resp.readEntity(String.class));
       resp.close();
       resp = proxy.addPerson(new Person("006","Dan Hagi"));
       System.out.println("HTTP code: " + resp.getStatus()
                                 +"  message: "+resp.readEntity(String.class));
       resp.close();

       Collection<Person> l = proxy.listPersons();
       for (Person p : l) System.out.println("list Person: "+p.getId()+"/"+p.getName());

       Person p = proxy.getPerson("006");
       System.out.println("get Person: "+p.getId()+"/"+p.getName());
   }

}
```


The previous annotated interface (FacadeInterface) makes it easy to invoke the service. We can build a proxy object of type FacadeInterface.

This proxy allows programming service invocations simply as method calls.

Example with Resteasy (client)

- In eclipse
  - Create a Java Project
  - Add RestEasy jars in the buildpath
  - Implement the Java bean that correspond to the JSON
    - Automatic generation with https://www.site24x7.com/tools/json-to-java.html
  - Implement the interface and the client class (FacadeInterface + Client)
  - Run

This is the procedure to run the client.

Example of existing REST WS

Currency API Request

The base URL for our currency API is

```
https://www.amdoren.com/api/currency.php
```

Request Parameters

Parameter	Description
api_key	Your assigned API key. This parameter is required.
from	The currency you would like to convert from. This parameter is required.
to	The currency you would like to convert to. This parameter is required.
amount	The amount to convert from. This parameter is optional. Default is a value of 1.

Example:
To get the latest exchange rate in EUR for 1 USD:

```
https://www.amdoren.com/api/currency.php?api_key=IBZzdLmM2yCYaXjgTZ6x&from=USD&to=EUR
```

Currency API Response

Element	Description
error	Error code. Value greater than zero indicates an error. See list below.
error_message	Short decription of the error. See list below.
amount	The exchange rate or amount converted.

Example:
JSON data returned from our currency API request:

{ "error" : 0, "error_message" : "-", "amount" : 0.90168 }

- www.amdoren.com
- Currency converter

Here is again the currency converter example we have seen previously.

We don't have the code of the server, but want to use Resteasy to develop a client.

Example of existing REST WS

Interface

```java
@Path("/")
public interface ServiceInterface {

    @GET
    @Path("/currency.php")
    @Produces({ "application/json" })
    public Result convert(@QueryParam("api_key") String key, @QueryParam("from") String from,
                         @QueryParam("to")String to);

}
```

Java bean (from JSON)

```java
public class Result {

    String error;
    String error_message;
    String amount;

    // getters/setters
```


As said in the documentation, the conversion method takes 3 HTTP parameters (api_key, from, to, the last is optional) and it returns a JSON.

The 3 HTTP parameters are associated with Java parameters (with @QueryParam) and a Java bean is created for the JSON.

Example of existing REST WS

Client

```java
public class Client {

    public static void main(String args[]) {

        final String path = "https://www.amdoren.com/api";

        ResteasyClient client = new ResteasyClientBuilder().build();
        ResteasyWebTarget target = client.target(UriBuilder.fromPath(path));
        ServiceInterface proxy = target.proxy(ServiceInterface.class);

        Result r = proxy.convert("9xwjRjxTtnzuGKH7LcWC5Vengr52F3", "EUR", "AMD");

        System.out.println("convert: "+r.getError()+"/"+r.getError_message()
                                      +"/"+r.getAmount()) ;
    }
}
```


And here is an example of client which invokes the service.

Interesting links

- Registry of services
  - https://www.programmableweb.com/category/all/apis
  - https://github.com/toddmotto/public-apis/blob/master/README.md
- Generation of POJO from JSON
  - https://www.site24x7.com/tools/json-to-java.html

Here are interesting links :

- sites where you can find interesting services

- a site which allows generating Java beans (POJO) from JSON

Conclusion

- Web Services: a RPC over HTTP
- Interesting for heterogeneity as there are tools in all environments
- Recently
  - SOAP WS less used
  - REST + XML/JSON more popular

To conclude, Web services aim at implementing a RPC service on top of HTTP and relying on standard formats (XML, JSON).

One of the main interest is the independence between the server (the service provider) and the client (the service consumer). They can be from different organizations and use different tools, OS, or languages.

The recent evolution is an obsolescence of SOAP and an increased popularity of REST and JSON.

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

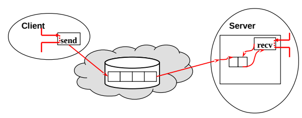
- Communication with message
  - In a classical environment: sockets
  - In a parallel programming environment: PVM, MPI
  - Other environments: ports (e.g. Mach)

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.

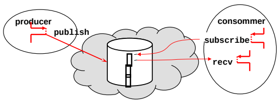
A queue may be allocated and used by clients (producers) or servers (consumers). The queue is managed in the network, meaning that it is not managed in clients neither servers. It is instead managed on machines managed in the middle, i.e. by the message middleware.

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)

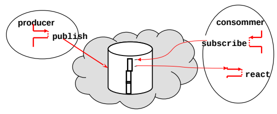
- content-based : topics are filters on the content of messages (e.g. I want to receive messages which include ….)

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



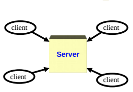
- Exists for all forms of messaging (Message Passing, Message Queuing, Publish/Subscribe)

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.

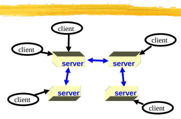
It mainly allows the association between an event (reception of a message) and a reaction (handling program).

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

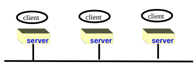
Message based middleware
Implementations



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

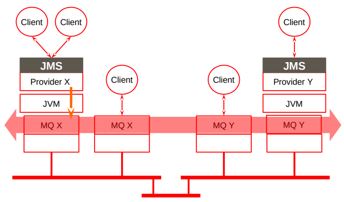
  - Message Queue
  - Publish/Subscribe
  - Event

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

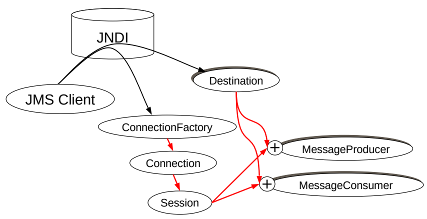
- *ConnectionFactory*: factory to create a connection with a JMS server
- *Connection*: an active connection with a JMS server
- *Destination*: a location (source or destination)
- *Session*: a single-thread context for emitting or receiving
- *MessageProducer*: an object for emitting in a session
-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

- Implementations of these interface are specific to providers …

13

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).

JMS - Architecture



14

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).

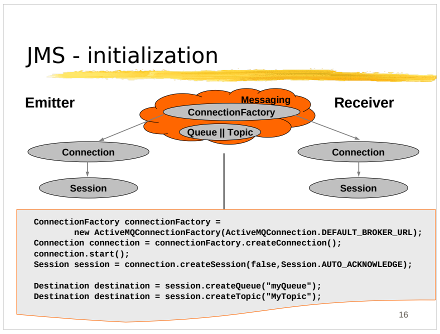
From a session and a destination, we can create a MessageProducer and a MessageConsumer allowing to emit and receive messages.

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*

15

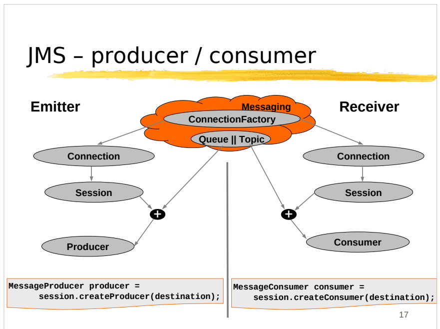
The interfaces described previously are abstract and are specialized according to the use of message queuing (Point-To-Point) or Publish/Subscribe



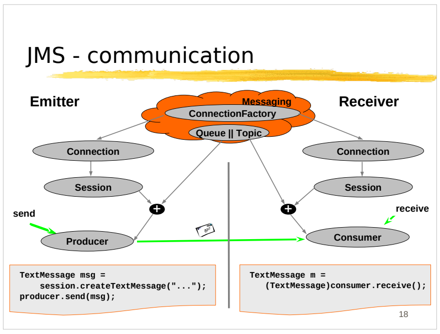
Here is the code which is common to the emitter and receiver for initializing the connection with the JMS server and obtaining a destination (one of the 2 lines should be chosen, queue or topic ...).

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.

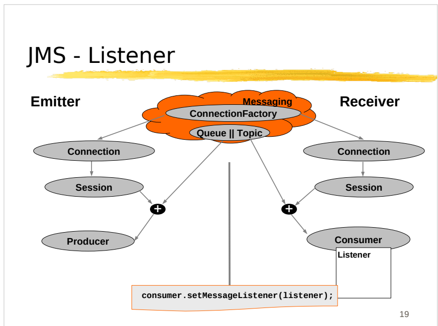


Here, with a session and a destination, we create a producer (left) and a consumer (right).



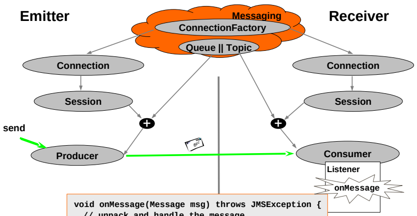
On the left, we can send a message (here a TextMessage) with a producer.

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.

JMS - Listener




```
void onMessage(Message msg) throws JMSException {
  // unpack and handle the message
}
```


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();
```

24

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

25

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.

Enterprise Service Bus

Daniel Hagimont

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

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

1

This lecture is about Enterprise Service Bus.

Integration - requirements

- Software bricks (applications)
  - Coarse-grain
  - Distributed
  - Different technologies (protocols, systems, API …)
  - After development
- Collaboration/Integration
  - Distributed communication
  - Adaptation of interfaces and data
  - Complex collaboration schemes (not only client-server)

2

The objective behind ESB is to provide support for the integration of different applications in a computing infrastructure.

Imagine the different types of software used in an organization. Let's consider the case of administration at N7. There are many software specialized for managing staff, managing students, managing accounting (budgets) … All these software are generally heterogeneous, i.e. there isn't a unique software covering every aspect, provided by a unique company.

The problem is to integrate these software into a consistent information system.

So, the requirement is to manage applications which are coarse-grain bricks, running on different machines (distributed). All these bricks are using different technologies. And the integration happens after development, generally at installation time.

ESB addresses the need for a way to allow collaboration between these software bricks, but it is unanticipated, so it has to allow adaptation of interfaces and data, and also different interaction schemes.

Problem statement

- For so many years, CIO are confronted with the problem of
  - Integrating heterogeneous applications
  - Building complex software architectures
  - Maintaining them
- With applications which were not anticipated to work together

3

Such an integration (after development, of existing applications) has been a difficult (hardly addressed) challenge for many years.

Integration vs interoperability

- Definition : interoperability is the capacity for a system to exchange information and services in a heterogeneous technological and organisational environment (IEEE, 1990)
- L'interoperability can be ensured by
  - The developer (CORBA, RPC, RMI)
  - The integrator (applications already exist)

4

We need to depict more precisely the difference between integration and interoperability.

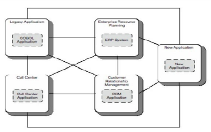
A definition (IEEE) of interoperability is given here. We see that the interoperability problem can be both addressed at development time or later at integration time.

We can observe that interoperability is a general property. It can be obtain at development time by sharing common tools between the developed applications.

Integration targets interoperability between applications which were developed independently.

Point to point integration

- Adhoc technologies
- *The accidental architecture*
- Spaghetti effect



5

A first solution is point to point integration.

Each pair of applications which have to collaborate is interconnected with adhoc technologies, i.e. implementing a specific connector, with a programming language or scripting language. An interconnection is a way to exchange events and data between applications.

This solution leads to what we call the accidental architecture (unanticipated) or also the spaghetti effect.

The obtained architecture is complex and almost impossible to debug.

ETL (Extract, Transform, Load)

- The most popular solution
- Export of data, adaptation and injection in other applications
- In batch mode (generally at night)
- Problem of update latency

6

A widely used solution is called ETL for Extract Transform Load.

The principle is that each application exports its data (useful for others). Then, these data can be adapted and imported by other applications.

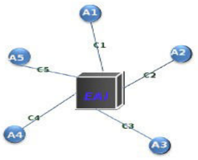
This is generally done in batch, periodically (generally at night).

The main problem with this approach is the latency of updates.

This latency can be a real problem, for instance in the management of a stock, we can sell products that are not available or not be able to sell an in-stock product.

EAI (Enterprise Application Integration)

- As a multi-socket
  - One connector per application
  - The EAI route messages between applications



7

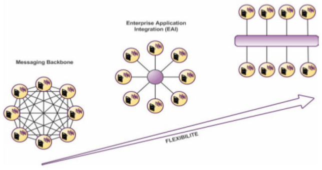
The analysis of the previous proposals led to the design of a type of middleware called EAI (Enterprise Application Integration). This is a kind of hub (like an electrical multi-socket) which allows connecting applications. Each application is connected to the EAI with a connector which may be developed for that specific application. The connector allows sending events and data to the EAI which routes them between applications, following a defined policy.

Therefore, compared to the spaghetti architecture, it provides an organized architecture and a systematic way of integrating applications.

Overall, the EAI provides a means to develop connectors and a server where the interconnection pattern is defined.

ESB (Enterprise Service Bus)

- A decentralized EAI
- Rely on standards (XML, WS, JMS ...)



8

EAI are generally centralized.

ESB is an evolution where the middleware is decentralized and where standards are used for data representation (e.g. XML) and communication (e.g. web services and JMS).

What makes an ESB

- A bus (MOM)
- Data (often XML)
- Adaptators/connectors (WS, ...)
- A control flow (routing)

- Objective : foster interconnection

9

So, what makes an ESB is :

- a message oriented middleware (MOM) generally implementing JMS for communication

- data representation with standards (often XML)

- connectors/adaptators for interfacing with existing applications

- means for controlling routing of data

The overall objective (as for EAI) is to provide a structured way to implement application interconnection.

ESB : products

- Proprietary
  - BEA Aqualogic (bought by Oracle)
  - IBM WebSphere Enterprise Service Bus
  - Sonic ESB from Progress Software
  - Cape Clear (spinoff from IONA)
  - Mule
- OpenSource
  - Apache ServiceMix
  - Jboss ESB
  - OW2 Petals (Toulouse!)

10

There are many providers of such technologies.

Notice that the Apache foundation has it own implementation.

In the labwork associated with this lecture, we will use Mule (for its simplicity).

Mule

- Mule is a Java-based enterprise service bus (ESB) and integration platform that allows developers to quickly and easily connect applications to exchange data following the service-oriented architecture (SOA) methodology. Mule enables easy integration of existing systems, regardless of the different technologies that the applications use, including JMS, Web Services, JDBC, HTTP, and more.

11

Here is the definition of mule by MuleSoft.

What Mule ESB does

- Decouples business logic from integration
- Location transparency
- Transport protocol conversion
- Message transformation
- Message routing
- Message enhancement
- Reliability (transactions)
- Security
- Scalability

12

Mule has the following characteristics:

- decouples business logic : don't need to code integration behavior within applications

- location transparency : an application does not need to know the producer of data it may receive

- transport protocol conversion : applications using different protocols can be interconnected

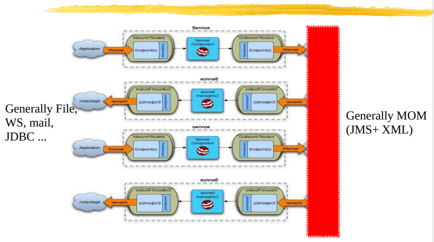
- message transformation : messages exchanged between applications using different data format can be adapted

- message routing : an update in one application can be routed to other applications

- message enhancement : as for message transformation

Mule also addresses reliability, security and scalability.

Overall view



13

Basically, Mule allows building connectors (horizontal lines) by assembling (reusing) components.

Mule could be used to build a connector between each pair of application, but it would lead to a spaghetti architecture.

Instead, the philosophy of ESB is that each application should be connected with a connector to a communication bus, generally a MOM (relying on JMS and XML standards).

On the left, each connector is connected with an application with a protocol that the application uses (e.g. File, Web service, mail, JDBC …).

This is a means to have a clear and adaptable architecture.

Mule concepts

- Endpoints
  - Channel for sending or receiving data
- Scopes
  - Processing blocks : polling, synchronizing, grouping …
- Components
  - Custom logic
- Transformers
  - Data conversion
- Filters
  - Filtering messages in flows
- Flow controls
  - Routing messages in different branches of the flow
- Error handlers

14

As said before, Mule relies on a set of components, allowing to build connectors.

Here are the types of components that Mule provides:

Endpoints. They are the contact points (or interfaces) with applications. They implement a protocol which is used by applications to export/import data.

Scopes. They implement a processing (in the sense of scheduling) behavior in a connector. They can be used for polling periodically a state change, waiting for an event (synchronizing), etc.

Components. They are used to implement a custom component.

Transformers. They are components used to implement data conversion.

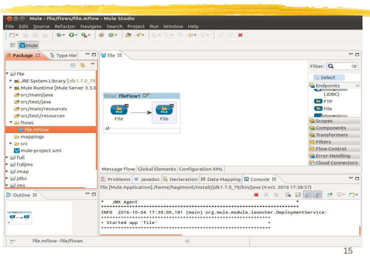
Filters. They are used to filter messages propagated in the connector.

Flow controls. They are used to create branches in the flow of messages within a connector, for creating different routes for messages or replicating messages.

Error handlers. They are used for defining error handlers.

In the following, not all these concepts are presented, the goal being to introduce the main concepts.

Mule Studio

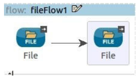


Mule comes with a graphical development environment called Mule studio. It allows to draw a component architecture which implements a connector. Components can be selected from a palette (on the right).

Here, the "Message Flow" tab corresponds to the graphical panel. We can select the "Configuration XML" tab which shows the XML description of the architecture. In the documentation, Mule says that the description of the architecture should be made in XML and that the graphical tools is provided to help the design.

First example

```
<?xml version="1.0" encoding="UTF-8"?>

<mule xmlns="http://www.mulesoft.org/schema/mule/core"
xmlns:file="http://www.mulesoft.org/schema/mule/file"
xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
xmlns:spring="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="CE-3.3.0" xsi:schemaLocation="
     http://www.mulesoft.org/schema/mule/file
     http://www.mulesoft.org/schema/mule/file/current/mule-file.xsd
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-current.xsd
     http://www.mulesoft.org/schema/mule/core
     http://www.mulesoft.org/schema/mule/core/current/mule.xsd ">

     <flow name="fileFlow1" doc:name="fileFlow1">
          <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
          <file:outbound-endpoint path="/tmp/out" responseTimeout="10000" doc:name="File"/>
     </flow>
</mule>
```



Here is the XML description of a simple example.

It defines a connector which links 2 File endpoints.

A File endpoint detects the creation (or modification) of a file. There are several possible parameters, e.g. whether the file should be deleted in the source.

In this example, the creation of a source file is detected in the source, the content of the file is transmitted as a message and the file is stored in the target.

In the XML description, each time a component is used, an XMLschema is added (like an include) in the header.

Notice that this is a simple connector, but a connector used in a ESB architecture should have at least one endpoint which is JMS (like in the overall view of ESB).

Endpoints

- Examples
  - <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
  - <jms:outbound-endpoint queue="MyQueue" connector-ref="Active_MQ" doc:name="JMS"/>
- Endpoints
  - Inbound and outbound
    - Ajax, JDBC, FTP, File, HTTP, JMS, RMI, SSL, TCP, UDP, VM ...
  - Inbound only
    - IMAP, POP3, Servlet, Twitter ...
  - Outbound only
    - SMTP ...
- New endpoints can be developed

There are many endpoints available. Some can be used for free and some have to be purchased.

We have seen the File endpoint on the previous slide. Another example is JMS which connects with a JMS MOM.

Endpoints can be inbound or outbound. And some endpoints can be both.

New endpoints can be developed in order to connect with application specific protocols.

Transformers

- Default transformers (associated with endpoint)
  - jmsmessage-to-object-transformer
- Custom transformers
  - Override default transformers
  - object-to-xml, xml-to-object, json-to-object ...
- New transformers can be developed

```
public class MyTransformer extends AbstractTransformer {
        public Object doTransform(Object src, String encoding) throws TransformerException {
        }
}
```


Transformers are components associated with endpoints, which adapt the data format of the content of messages.

Some endpoints have default transformers, but their transformers can be redefined. For instance, a JMS inbound endpoint (<jms:inbound-endpoint>) has a default transformer jmsmessage-to-object-transformer, so that a JMS message is implicitly received as a Java object.

Other available transformers are called custom transformers and can be chained after/before endpoints.

And finally, users may program their own transformers, but generally they don't have to.

Components

- Customize the message flows

```
public class Filter implements Callable {
        public Object onCall(MuleEventContext eventContext) throws Exception {
                person p = (person)eventContext.getMessage().getPayload();
                return null;
        }
}
```

19

Components in Mule are simply components programmed in Java. They can adapt the messages which flow in a connector.

In this example, we know that the payload of the message is a Person Java object and we can adapt it or even remove the message (return null).

Transformer-component example

Custom component

```
<flow name="xmlFlow1" doc:name="xmlFlow1">
        <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
        <mulexml:xml-to-object-transformer doc:name="XML to Object"/>
        <component class="Filter" doc:name="Java"/>
        <mulexml:object-to-xml-transformer doc:name="Object to XML"/>
        <file:outbound-endpoint path="/tmp/out" responseTimeout="10000" doc:name="File"/>
</flow>
```

Use xstream (xml to Java bean conversion)

20

Here is an example where we use transformers and a component.

A File endpoint allows detecting the creation of a file in the /tmp/in directory of the local machine. The content of the file is transformed by a XML-to-Object transformer. This transformer uses Xstream which transforms the XML textual representation (the content of the file) into a Java bean object (by default each field in the XML corresponds to a field in the Java bean). Then a Java component is used to adapt this Java bean. The adapted Java bean is then passed to an Object-to-XML transformer which converts the Java bean into an XML textual document. This XML document is then stored in a file in the /tmp/out directory.

Flow controls

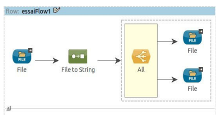
- All
  - Sends messages to all routes
- Choice
  - Routes messages based on expressions
- First successful
  - Sends a message to a list of routes until one is processed successfully
- Round robin
  - Send a message to the next route in the circular list of route
- ...

21

Flow control components allow routing messages following different paths. Here are some examples of such components. In all examples, it's a one-to-N hub, with different routing policies.

Flow control example

```
<flow name="essaiFlow1" doc:name="essaiFlow1">
    <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
    <file:file-to-string-transformer doc:name="File to String"/>
    <all doc:name="All">
        <processor-chain>
            <file:outbound-endpoint path="/tmp/out1" responseTimeout="10000" doc:name="File"/>
        </processor-chain>
        <processor-chain>
            <file:outbound-endpoint path="/tmp/out2" responseTimeout="10000" doc:name="File"/>
        </processor-chain>
    </all>
</flow>
```



22

Here is an example with a All flow control component.

A File endpoint allows detecting the creation of a file in the /tmp/in directory of the local machine. The File-to-String transformer produces a String (this is necessary to have at least one processing component). Thanks to the All flow control component, the message is replicated and routed towards two destination File endpoints.

Global elements

- Global elements have to be declared to configure some mule elements
  - JMS connector

```
<jms:activemq-connector name="Active_MQ" specification="1.1" username="admin"
password="admin" brokerURL="tcp://localhost:61616" validateConnections="true"
doc:name="Active MQ"/>
```

  - IMAP connector

```
<imaps:connector name="IMAP" validateConnections="true" checkFrequency="1000"
doc:name="IMAP">
    <imaps:tls-client path="*" storePassword="*"/>
    <imaps:tls-trust-store path="*" storePassword="*"/>
</imaps:connector>
```

23

Some definitions are declared in Global Elements (another tab in Mule Studio). They mainly define connectors with external servers, such as a JMS server, a IMAP server or a database server.

Here are two examples :

- a JMS connector describing the connection with an Apache ActiveMQ server

- an IMAP connector describing the connection with an IMAP server

Global elements

- Global elements have to be declared to configure some mule elements
  - Data source (with a bean)

```
<spring:beans>
    <spring:bean id="dataSource" name="dataSource"
        class="org.enhydra.jdbc.standard.StandardDataSource"
        destroy-method="shutdown">
            <spring:property name="driverName" value="org.hsqldb.jdbcDriver"/>
            <spring:property name="url" value="jdbc:hsqldb:hsql://localhost/xdb"/>
            <spring:property name="user" value="sa"/>
    </spring:bean>
</spring:beans>
```

  - Database connection

```
<jdbc:connector name="Database__JDBC_" dataSource-ref="dataSource"
validateConnections="true" queryTimeout="-1" pollingFrequency="0"
doc:name="Database (JDBC)"/>
```

24

Here are some other global elements :

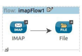
- a Data source, i.e. the address of a database accessible with JDBC

- the database connector which uses this data source

IMAP / SMTP examples

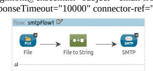
- IMAP

```
<flow name="imapFlow1" doc:name="imapFlow1">
    <imaps:inbound-endpoint host="imap.gmail.com" port="993" user="tpdhlogin"
    password="tpdhpasswd" responseTimeout="10000" connector-ref="IMAP" doc:name="IMAP"/>
    <file:outbound-endpoint path="/tmp/out" responseTimeout="10000" doc:name="File"/>
</flow>
```



- SMTP

```
<flow name="smtpFlow1" doc:name="smtpFlow1">
    <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
    <file:file-to-string-transformer doc:name="File to String"/>
    <smtp:outbound-endpoint host="mail.enseeiht.fr" port="587" user="hagimont"
        to="hagimont@enseeiht.fr" from="hagimont@enseeiht.fr" subject="email from Mule"
        replyTo="tpdhlogin@gmail.com" responseTimeout="10000" connector-ref="SMTP"
        doc:name="SMTP"/>
</flow>
```



25

The first example receives emails from an IMAP server (gmail.com) using the account tpdhlogin/tpdhpasswd. Each email is then stored in a file in the /tmp/out directory.

The second example detects the creation of a file in the /tmp/in directory, transforms the file content into a String which is sent in an email (to me) using the SMTP server from N7.

JDBC example

```
<flow name="jdbcFlow1" doc:name="jdbcFlow1">
    <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
    <mulexml:xml-to-object-transformer doc:name="XML to Object"/>
    <jdbc:outbound-endpoint exchange-pattern="one-way" queryKey="insertion"
        queryTimeout="-1" connector-ref="Database__JDBC_" doc:name="Database (JDBC)">
        <jdbc:query key="insertion" value="insert into accounts values
                    (#[payload.nom], #[payload.prenom], #[payload.email]);"/>
    </jdbc:outbound-endpoint>
</flow>
```

26

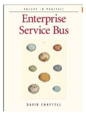
This last example detects the creation of a file in the /tmp/in directory, transforms the content of the file (supposedly an XML document) into a Java bean object which is passed to a Database component. This database component :

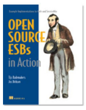
- references a database connector (Database__JDBC_) which includes the address of the database

- includes the definition of a query to execute. The query inserts a new account in the database. Notice that the fields of the received Java bean object can be accessed in the request with #[payload.field]

Bibliography

- General
  - Enterprise Service Bus: Theory in Practice (O'Reilly)
- Technical
  - Open-Source ESBs in Action (Manning)





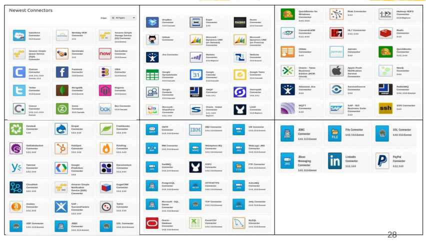
27

If you are interested by ESB

- the first book is more about the philosophy of ESB

- the second is more tutorial oriented with both references to Mule and Apache ServiceMix.

Connector market place

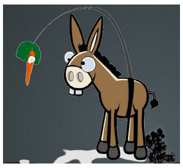


28

ESB is very popular in the industry as it responds to a very practical problem.

Mule is provided for free with a set of basic components. More sophisticated components can be purchased from a marketplace.

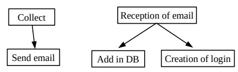
The end



29

Labwork - ESB - Mule

- Scenario : registration of students
  - Data collection (Collect application)
    - Allows to export data to XML format
  - Reception by email and validation by a secretary
    - Validation by replying to the email
  - Integration in a list managed in the web server
    - Addition in a database
  - Creation of a login for the student
    - Using a web service



30

For the labwork, we will use Mule to implement a scenario. The registration of students in a school involves different software:

- a data collection application which allows to enter in a graphical interface the name, email, login, password of a student. The data are exported to an XML file.

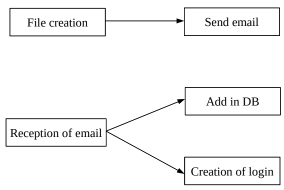
- Email application. A registration is sent by email to a secretary who validates the registration. The secretary validates the registration by replying to the email (@validation).

- On reception of the validation (@validation) :

  - the registration is added in a database

  - a login is created for the new student (with a web service)

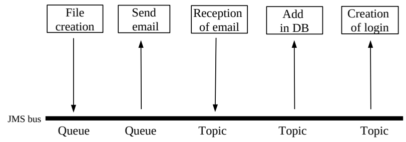
First integration (spaghetti)



31

This is a first possible integration scheme. Since applications are directly interconnected with connectors, this is typically a spaghetti architecture.

Deuxième intégration (ESB)



32

An ESB integration relies on a JMS bus. Each application is connected to that bus.

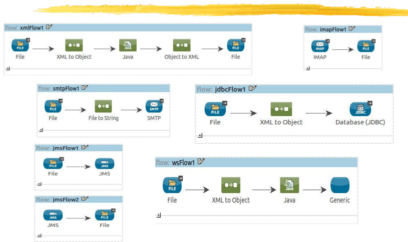
- The detection of a new file exported by the Collect application sends the file content on a JMS queue.

- A message received on that queue triggers the emission of a email to the secretary. The secretary validates the registration by replying to the email.

- The reception of the validation email generates a message (with the registration characteristics) on a topic. Two connectors subscribe to that topic:

  - A message reception on the topic adds the registration into the database
  - A message reception on the topic creates the login for the student.

Different examples



33

This is a list of the little examples that are given to you for the labwork. From these little examples, you can implement the registration scenario described above.