Communication Mechanisms

PID_00287534

Xavier Vilajosana Guillén
Leandro Navarro Moldes

Recommended minimum reading time: 4 hours

Universitat Oberta de Catalunya

Xavier Vilajosana Guillén

Computer Engineer from the UPC and with a PhD in Computer Science from the UOC. Lecturer in Computer Science, Multimedia and Telecommunications at the UOC, he is responsible for the Computer Networks Area of the Bachelor's degrees in Computer Engineering, Telecommunications Engineering and Multimedia Engineering. His interests include sensor networks, scalable routing protocols and distributed network resource management.

Leandro Navarro Moldes

PhD in Telecommunications specialized in distributed applications on the Internet. Professor at the Polytechnic University of Catalonia (UPC).

The assignment and creation of this UOC Learning Resource have been coordinated by the lecturer: Joan Manuel Marquès Puig

First edition: February 2022

Av. Tibidabo, 39-43, 08035 Barcelona
Authorship: Xavier Vilajosana Guillén, Leandro Navarro Moldes
Production: FUOC

Contents

Introduction

Nowadays, we cannot imagine the existence of an application that does not use a communication mechanism, either because the application has been developed as a set of threads or processes that communicate with each other, or because the application is networked and distributed, where its components interact in an orderly way, such as, for example, in a client-server model. We use communication primitives on a daily basis: for consulting web applications, exchanging information, playing content on our mobile devices, or for synchronizing the microcontrollers of our household domestic appliances.

This module aims to describe the techniques and the communication paradigms for applications or processes that are applied in the operation of most of today's software. Therefore, we present the communication mechanisms between processes, objects and applications that have come about as a result of the need of distributed, parallel, or concurrent processes, objects and applications to intercommunicate.

In the module, we will see the fundamental paradigms of application communication, where we highlight message passing and remote invocation as the greatest exponents. Then, the module presents the most widely used communication mechanisms: sockets, which are an implementation of the message passing paradigm, and the remote invocation of objects (and remote procedures), which consist of running code remotely. The module also introduces data encoding techniques, which can be either binary or textual, and poses the problem of communication between non-homogeneous participants, which requires agreements about the representation of information. Finally, communication mechanisms such as RMI, XML-RPC and SOAP, widely used today, are introduced.

Objectives

With the study of this module, you will achieve the following objectives:

1. Understanding the different communication paradigms between processes, applications or objects.

2. Understanding the transmission's problem of data information and the different ways of encoding the information. Distinguishing binary from textual encoding.

3. Understanding the operation of communication mechanisms with sockets.

4. Understanding the remote invocation of processes and objects, focusing on the main protocols that set it up as well as the leading characteristics of web services.

1. Communication paradigms

As soon as there is the possibility of executing processes in a parallel, concurrent or distributed way, a need to establish communication between these processes arises. Communication can be carried out by using different methodologies and techniques that are known as communication paradigms.

Communication between processes can be synchronous or asynchronous:

- **Synchronous communication** (or synchronic): is the exchange of information between processes in real time. This communication requires simultaneity of the processes.

  Example of synchronous communication

  Telephony is an example of **synchronous communication** because the sender and the receiver coincide in temporal space in terms of transmitting information, that is, talking on the phone.

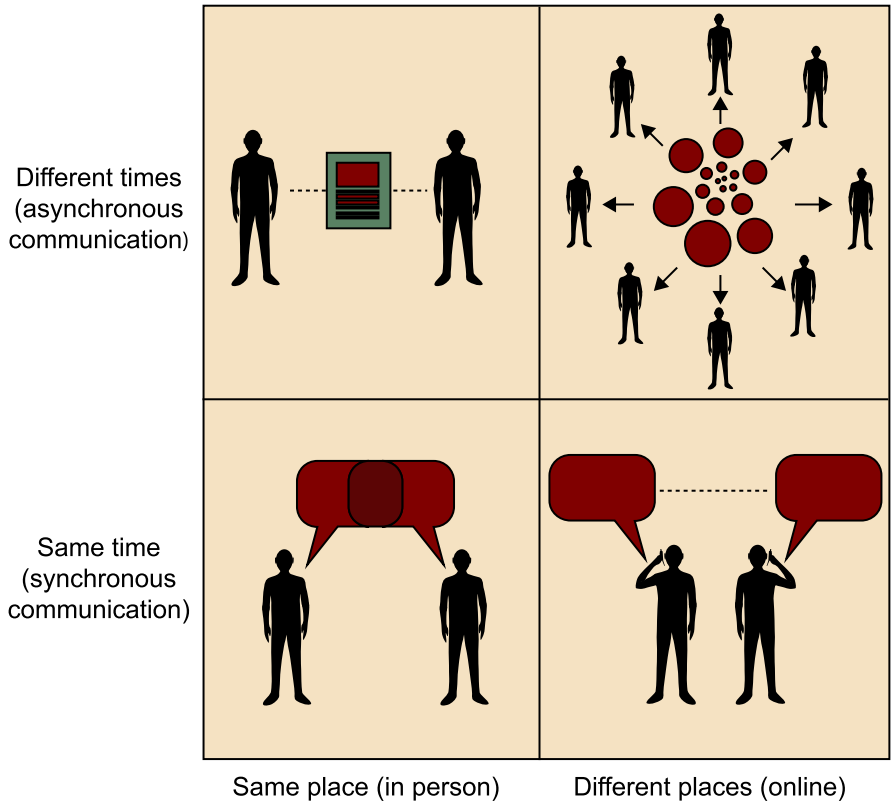
- **Asynchronous communication** (or asynchronic): is the communication that is established between two or more processes in a deferred manner in time, that is, when there is no temporal coincidence or simultaneity.

  Example of asynchronous communication

  The postal letter is a very old example of **asynchronous communication**. The sender and receiver do not coincide in the temporal space in terms of transmitting the information, that is, sending the letter. Current examples of asynchronous communication are e-mail and forums.

The reliability of the transmission must be taken into account in any communication mechanism, that is, the mechanism must assure us any information that is sent will arrive. If it does not arrive, the mechanism will notify us about it. A communication mechanism must also ensure the forwarding in order of the information being sent and define the topology of the communication process, that is, whether messages are sent one by one (unicast or peer-to-peer), one to many (multicasting or broadcasting) or many to one (client-server).

Figure 1



Communication paradigms share the same objective but their scope of application is diverse, due to its direct dependency on the application, the hardware, the interconnection between processes and the characteristics of the network. There are paradigms that are only suitable for establishing synchronous connections, while others only allow asynchronous connections. There are also paradigms that can be used both synchronously and asynchronously.

Next, we will see the most relevant communication paradigms, which include most of the forms of communication between processes, programs, or objects, whether on the same machine or on remote machines. It should be noted that the paradigms that are presented do not always represent the same scenario nor are they suitable for any situation. Some of them are more suitable for low-level process communication or even for scenarios characterized by many constraints, such as communication between microcontrollers. Others are better suited to structure communication in complex, high-level programs. Let' s see them.

1.1. Message passing

Message passing is a communication paradigm used in object-oriented programming, concurrent and parallel computing, and distributed computing. In this model, processes or objects can send and receive messages from the other processes, and even synchronize while waiting for a message.

In this paradigm, developers organize programs like a collection of tasks with private local variables, with the ability to send and receive data between tasks through message exchange. It is characterized by having a distributed and known address space that allows communication between tasks or objects.

Messages can be sent over the network or use shared memory, if available. Communications between two tasks are two-sided, where both participants have to call or invoke an operation (send, by the sender, and receive, by the receiver). We can call these communications *cooperative operations*, as they must be carried out by each process: the process that has the data and the process that wants to access the data. There may also be one-sided type communications in some implementations, if it is a single process that calls the operation and places all the required parameters, and the synchronization is done implicitly.

Conceptually, message passing is the most explicit communication paradigm. The sender generates a message that is sent through a communication channel using a specific primitive to perform this task. The receiver, using a primitive, is able to read the message from the channel. This paradigm perfectly supports both synchronous and asynchronous communication, which, in any case, would only have the memory limitation of the channel.

As advantages, message passing allows a link with the existing hardware, as it matches well with architectures that have a series of processors connected by a communication network (either an internal interconnection or a wired network). In terms of functionality, it includes a greater available expression for concurrent algorithms, since it provides an uncommonly explicit control in communication between processes.

In contrast, the main problem with message passing is the responsibility that the model places on the programmer, who has to explicitly implement the data distribution scheme, the communications between tasks, and their synchronization. In these cases, the programmer's responsibility is to avoid data dependencies, avoid deadlock and communications race conditions, as well as implement fault tolerance mechanisms for the applications.

Deadlock

A **deadlock** (also known as **Interlock**) is a situation where two or more actions wait for each other, unable to continue until the others end, so none of them manages to end.

In the message passing paradigm, it is the programmer who directly controls the flow of operations and data. The most commonly used means to implement this programming model is a library, which implements the API of primitives commonly used in message passing environments.

These message passing APIs typically include:

- Point-to-point message passing primitives. From the typical send and receive operations, to specialized variations of these.

- Synchronization primitives. The most common is the barrier, which implements a lock on all parallel application tasks (or on a part of them).

Barrier primitives

Barrier primitives are points of synchronization or locking of multiple tasks in a program. The barrier can also be the synchronization point of various distributed programs or processes.

- Collective communication primitives. Several tasks participate in an exchange of messages between them.

- Static (and/or dynamic) creation of task groups within the application, to restrict the scope of application of some primitives. It allows the conceptual separation of some interactions from the others within the application.

The specific implementation of a message passing mechanism has to consider:

- Whether the sending will be reliable.

- Whether the sending will guarantee the order of the information transmitted.

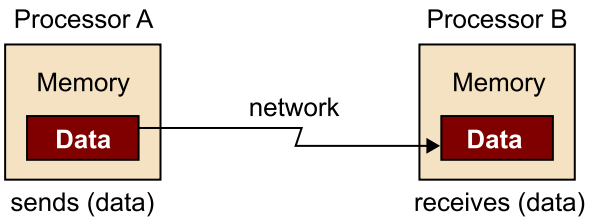
- Whether messages are sent one by one (unicast or peer-to-peer), one to many (multicasting or broadcasting) or many to one (client-server).

- Whether the communication is synchronous or asynchronous.

There are many implementations of this paradigm, of which we can highlight sockets, SOAP, CORBA, REST or D-Bus, among others.

Figure 2

Basic message communication



1.2. Shared memory

In the shared memory paradigm, memory is considered to be shared by the different processes. This memory can be accessed by multiple programs with the intention of sharing objects or variables and thus reduce redundancy.

In this model, programmers see the programs as a collection of processes that access local variables and a set of shared variables. Each process accesses the shared data through an asynchronous read or write operations. Therefore, since more than one process can perform the operations of accessing the same

shared data at the same time, mechanisms - such as semaphore or blocking mechanisms - must be implemented to solve any mutual exclusion problems that may arise.

In the **shared memory model**, the programmer sees the application as a collection of tasks that are normally assigned asynchronously to execution threads. The execution threads have access to the shared memory space with the above-mentioned control mechanisms.

Shared memory is a very fast way to communicate processes (as opposed to message passing, as in the case of sockets). However, it is less powerful than communication with message passing primitives, since it is essential to be able to share memory and, therefore, the processes have to be on the same machine.

There are, however, approximations to distributed shared memory systems. These systems allow sharing an amount of memory remotely. A constraint on these systems is that they must be part of a cluster, since a very fast network interconnection is required for memory sharing to make sense. Shared memory is frequently organized into memory pages that are remotely addressable and accessible. Another way to organize memory is with the named tuple space, where the minimum unit of information sharing is defined by a minimum size, called tuple.

Clusters

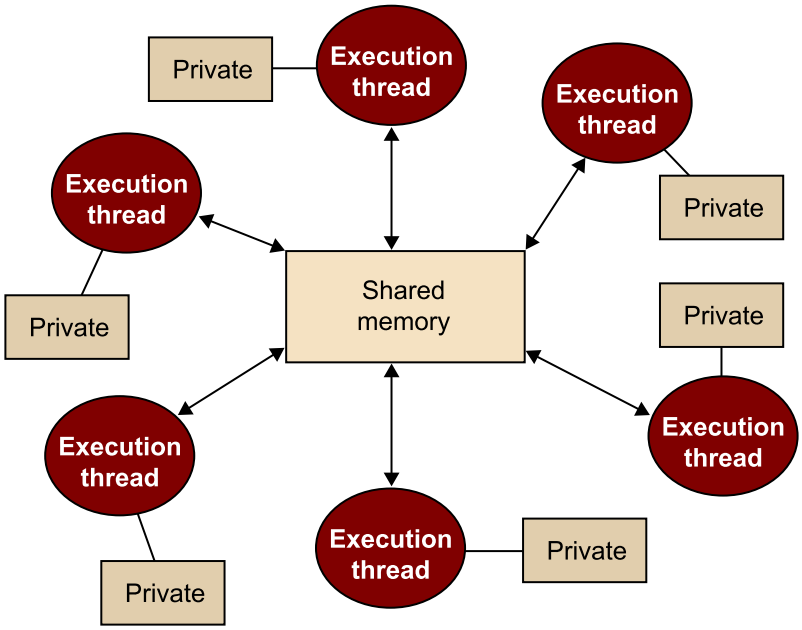
A cluster is a structure composed of collections of computers with similar characteristics interconnected through a local area network. The computers make use of the same operating system and middleware which is responsible for abstracting and virtualizing the different computers of the system, so that it gives the user the vision of a single operating system. Clusters are systems dedicated to supercomputing. The operating system of a cluster is standard and therefore, it is the middleware that provides the libraries that allow parallel computing.

Implementing distributed shared memory systems requires a model of data consistency achieved by a protocol to maintain data consistency.

Some typical examples

Typical examples of shared memory systems include the Linux 2.6 kernel, which offers / dev/shm as a mount point for shared memory, or OpenMP implementations.

Figure 3



1.3. Remote invocation

Remote invocation or remote execution of procedures (RPC[1]), together with message passing, is one of the most frequently used techniques when developing distributed applications.

(1)RPC stands for *remote procedure call*.

Remote invocation is a process or object-oriented[2] communication paradigm consisting of the possibility of executing a subroutine or method in another address space (usually another computer accessed through the net).

(2)In the case of running remote object methods, the paradigm is called *remote method invocation* (RMI).

The particularity of RPC methods is that the application programmer is totally unaware that the method execution is being done remotely. Nor does the programmer need to encode all the details of this remote interaction. Remote invocation is somehow analogous to the message passing technique. The sender calls a primitive on the receiver, and then the receiver responds to the sender, with the result. The difference lies in the fact that remote invocation does not require the receiver to call any method in order to receive the request and normally the process of transmission of the information is atomic.

Recommended reading

It is interesting to read the article "Implementing remote procedure calls" by Birrell, Nelson, etc., describing the first embodiment of a transparent remote invocation system. The efficiency comparisons must be highlighted.

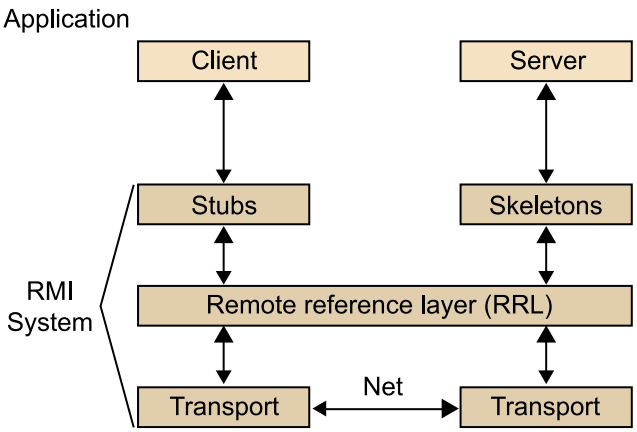
An RPC is initiated by a client, who sends a request to a remote server in order to perform a specific procedure or method. In this call, the client sends the parameters to the server (a process known as *marshaling,* which we will see later). The server computes the response and sends it to the client, who is blocked, waiting for the response. Upon receipt, the client's application continues its process. There are many implementations of the RPC paradigm, usually incompatible with each other.

An important difference between RPC and local calls among objects or processes lies in the fact that remote calls may fail due to network problems. When there is a fault, the sender has to manage it, taking the following aspects into account:

- The method was not executed remotely because the call failed before invoking the method.
- The execution of the method failed.
- The response with results of the method execution failed.

In this way, and to avoid complex handling of exceptions, we seek idempotent remote methods; that is, if they are executed more than once, the result is the same.

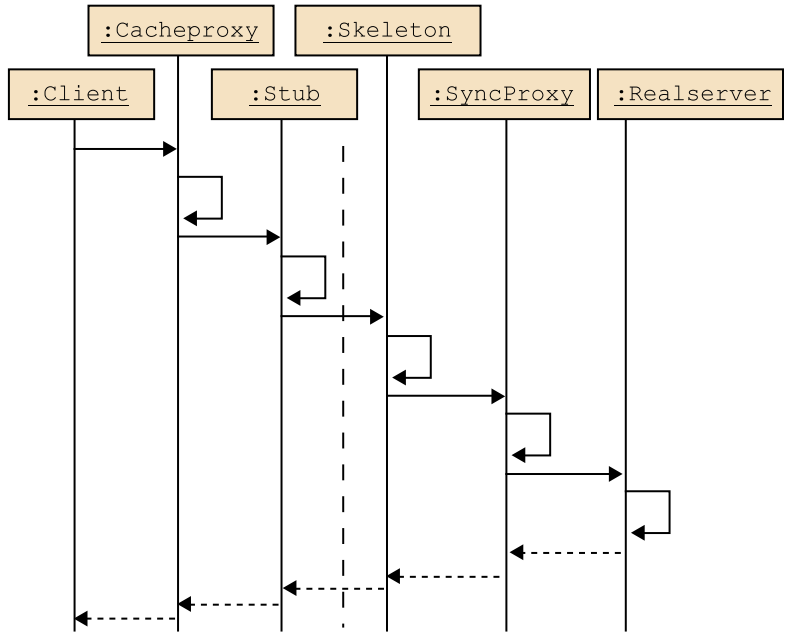
Figure 4



Remote invocation usually uses an architecture of components consisting of the following elements:

- The process or client object.
- The remote method interface: it is local to the client, and contains the signature of the methods offered by the remote object. This component is usually called a stub.
- The process or remote object.
- The remote method interface and implementation, contacted by the stub, and then, already locally, calls the remote object method. It is usually called a skeleton.

Figure 5

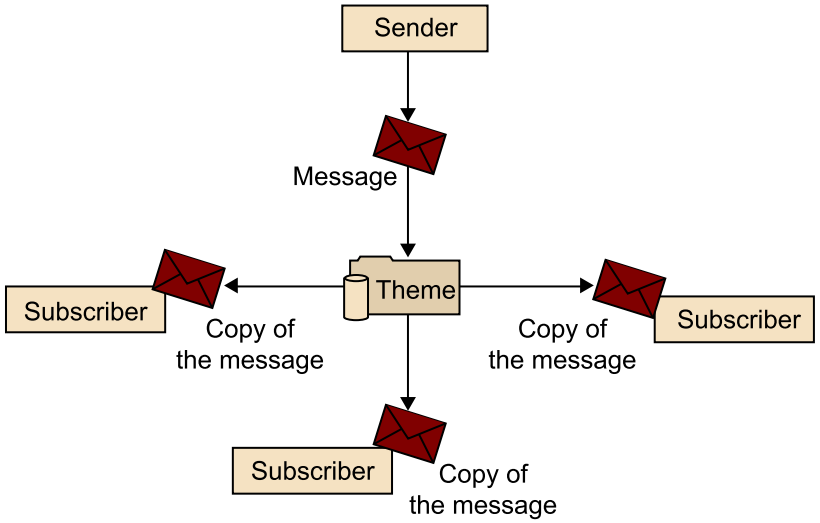


1.4. Publish subscribe

It is an asynchronous communication paradigm. The publishers do not send their messages to specific recipients. Instead, messages are published in different categories, without knowing who the subscribers are.

Subscribers express interest in one or more categories and only receive messages from those categories. This paradigm allows a decoupling of publishers and receivers, providing greater system scalability and more changing or dynamic network topologies.

Figure 6



To behave as described in the publish-subscribe paradigm, the processes act with the following roles:

a) Producer of information: application that has the information to be disseminated. The producer publishes this information without having to know who is interested in receiving it. It sends the information through channels.

b) Consumer of information: application interested in receiving information. The consumer subscribes to channels that disseminate information that it is interested in. The consumer receives this information through these channels.

c) Broker: it is between the producer and the consumer of the information. It receives information from producers and subscription requests from consumers. It is also responsible for forwarding published information to recipients subscribed to the channel. This broker may be distributed. In this case, different brokers must organize themselves to provide the channels.

d) Channel: these are the (logical) connectors between producers and consumers of information. The channel determines several of the properties of the distributed information and the supported functionalities: type of information; data format; possibilities for users to customize the channel (for example, content selection or modes of operation); whether content expires or is persistent; strategy to be followed for making updates; whether data are delivered only once (at the time of occurrence, such as on television) or whether, instead, we guarantee content reception, regardless of when it was generated; mode of operation (if supported by off-line mode of operation); payment (payment policy used: pay for viewing, for time, for content ...).

As we have seen, the publish-subscribe paradigm allows an asynchronous distribution of information. Next, we show some situations and some aspects where this paradigm may be an appropriate alternative:

a) Location: for processes it is a problem to know where the information is that interests them. There is a decoupling of the process and the data. The process subscribes to some channels and then the information provider assumes the active role of getting their information to the interested parties.

b) Focus: because the process explicitly states its preferences, it is easy to provide information focused on its interests.

c) Current events: data can be distributed as they become available. The information provider may invalidate obsolete data.

d) Tailoring: the provider can also decide which information the recipient sees and does not see.

e) Reduced traffic: as the system disseminates information to those who are interested in receiving it, traffic in the net is greatly reduced.

The publish-subscribe paradigm is designed to provide three types of services:

1) Coordinate processes.

2) Play content.

3) Inform people.

Some of the fields that implement applications with the publish-subscribe paradigm are:

a) Newsgroups and mailing lists. Usenet messages and mailing lists can be thought of as somewhat primitive publish-subscribe systems. For example, Usenet messages distribute articles throughout Internet. A message server subscribes to other message servers and receives messages from its subscription groups. When a new article is generated in a group, the server generating it ensures its distribution to other servers. Nowadays, Usenet is in disuse, even though it still retains large amounts of information.

Usenet

Usenet is a historic communications system between computer networks, created in 1979, before the Internet we know today. It is a network formed by newsgroups, including discussions on different issues, musical content, films, television series and digital documents.

Its operation is very similar to peer-to-peer networks because users share non-profit content, without a single company managing it or controlling downloads.

Like peer-to-peer networks, it also works by using an open and decentralized protocol.

Currently, it is a little-used alternative in comparison to existing web options, or compared to the most popular peer-to-peer systems. However, it contains a significant amount of available information and millions of shared files.

b) Stock exchange and news. Systems reporting on stock market shares evolution, or news agencies, are other examples of publish-subscribe systems. In these systems, users specify interests and the system has to ensure users have the most updated information as possible at all times.

c) Traffic information systems. As in stock market and news applications, information must be sent mostly in real time. Information is also distributed via computers or mobile devices.

d) Software distribution. Many systems require software to be updated frequently: for example, the update manager of some operating systems. Using a publish-subscribe application ensures the system is in continuous operation, and updated in the latest version without security problems for updates.

e) Alarm, monitoring, surveillance and control services.

The publish-subscribe paradigm is usually implemented following an event-driven programming paradigm that facilitates the existence of publishers and subscribers.

1.5. Event-driven programming model

The publish-subscribe paradigm requires programming models to facilitate asynchrony and decoupling of components. Thus, one of the most widely used programming paradigms to implement this communication model is *event-driven programming*.

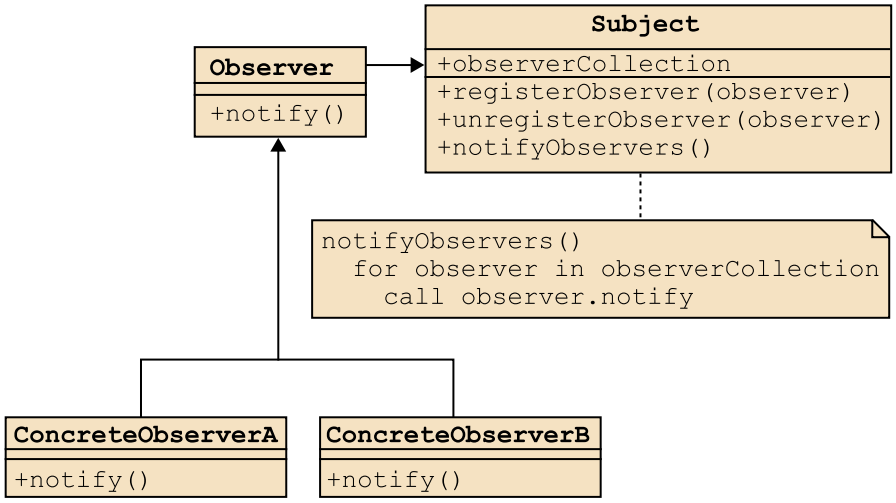
> In event-driven programming, events occurring in the system determine both program structure and execution, either defined by users or caused by the program itself.

In sequential (or structured) programming, the programmer defines the program flow. In event-driven programming, the user directs the execution flow. For each action of the user, an event will be generated, and the program will respond.

An event-driven program typically implements a state machine, with state transitions being driven by events. The most commonly used technique for the implementation of event-driven communication is the "observer" design pattern. In this pattern, a process or "observer" object receives notifications from its subjects. The processes that generate events know the observers, and notify them of the event as soon as it happens.

This programming paradigm is used to implement the publish-subscribe paradigm, and also in the development of graphical user interfaces, embedded systems programming and standard instant messaging protocols, like XMPP.

Figure 7



In the UML class diagram represented in figure 7, we can see that the subject knows a list of observers that register at a given moment. From then on, they can receive a notification from the subject. Each observer implements the receipt of the notification at their convenience.

2. Communication mechanisms

There is a wide range of communication mechanisms that implement the paradigms described in the previous section. In this section, we will present the mechanisms with a broader use.

When two processes communicate, they exchange information. We will see that this exchange has some requirements.

2.1. Data encoding

Any remote invocation procedure needs to transfer data over the net as a sequence of bytes.

This exchange is possible if there is an agreement about data representation and interpretation between both parties. This agreement must be coherent regarding the following aspects:

a)Mechanism of encoding (and decoding) data arguments and results: a process for transforming information, expressed in the form of data structures, into a sequence of bytes. This process is called *conversion*, *serialization* or *marshaling*.

b)Interpretation of data: it depends on the work environment and, in turn, is conditioned by the computer architecture, the operating system and the programming language. It becomes complicated when communicating machines have different characteristics.

These aspects form a protocol in terms of transport, connecting two network endpoints, with the call mechanism (RPC), rather than the "reliable pipe" model offered by TCP.

Marshaling

The serialization and deserialization mechanism is called *marshaling* and *unmarshaling* in honour of General Marshall, a soldier who organized the Normandy landings in the Second World War: troops and military personnel were arranged with great order on the Dover beach. A careful and orderly mechanism of serial transport by ship ended up reproducing the same organization on the Normandy beach. A tank would be a data structure in our applications, a ship would be an IP packet, a soldier would be an integer, etc.

2.1.1. The problem of data encoding

To encode data, a representation format must be selected or agreed upon for the data that will be used as arguments and results of the remote operations call. This includes defining what type of data will be used: character, integer, real, etc., endianness (the order in which the bytes are sent), repertoire or set of characters to be used, and features of our data specification language. For example, whether data types that use pointers can be defined; how to make data types correspond and construct complex data types with the chosen format, and with data types of common programming languages.

External representation of data is the convention for representing data structures during transport through the network: to convert or marshal arguments and results.

The most common data types correspond to the primitive data types of the most popular processors and programming languages, such as C or C++.

- Base types: integer, real, character, enumerated.
- Flat types: aggregations, arrays, vectors (padding).
- Complex types: with pointers, for example, tree (to be "flattened", etc.).

There are several possible strategies for converting the data represented in each computer to be transferred, from internal formats to a serialized format. The aim is to reduce complexity and cost of data conversion, as follows:

a) Transfer in sender's internal format.

b) Transform the data to send them in the internal format of the receiver. In both cases the problem is complex, as one of the extremes does not have to do anything, while the other should know how to represent the data for any possible representation (architecture). It does not seem like a viable solution.

c) Send in an intermediate canonical form (for example, how big-endian IP headers are represented). Each computer only has to know how to convert between its internal format and the canonical format.

d) Do not do any conversion, if the two computers are similar. In the previous case, two machines with the same internal architecture may make two unnecessary conversions for the data to travel in the canonical format when they could have communicated without making any conversion. If the format of the computer at the other end is known, the transfer can be optimized, saving unnecessary conversions.

e) In the format of the sender, which includes a format indication, and the receiver converts. In this way, the sender has neither to know the receiver nor has to carry out any conversion. The job is for the receiver, who has to apply the necessary transformation for each possible format. In some cases, data formats and sizes are fixed, and all the receiver has to do is change the byte order in some data. This latter case is known as **endianness**, and can be of two types:

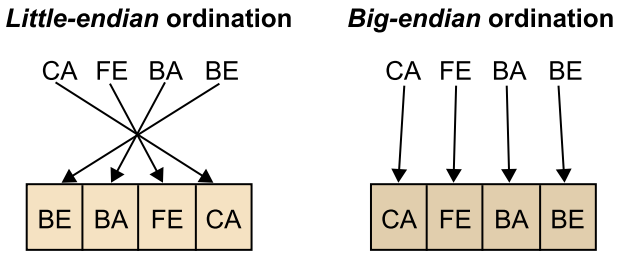
- ***Little-endian***: the least significant byte is saved in the smallest memory address, which will be the address of the data.

- ***Big-endian***: the most significant byte is saved in the smallest memory address, which will be the address of the data.

Little-endian formats

The processors of Intel 80X86, Pentium, Alpha, Windows operating systems, OSF/1 and various file formats are little-endian.

Figure 8



Big-endian formats

The processors of Motorola 680x0, Hewlett-Packard's PA-RISC and Sun's SuperSPARC are big-endian. The Silicon Graphics MIPS and IBM/Motorola's PowerPC Processors can work as little-endian and big-endian. Apple's operating system is also big-endian, and so is the Java virtual machine. Internet is also big-endian: in TCP and IP, the most significant byte is transmitted first.

Binary encoding consists of encoding data with the minimum possible number of bits. This reduces the amount of information transmitted.

Examples of binary coding

Some examples of protocols using this type of encoding are: DNS, LDAP, SNMP, NFS.

In binary encoding, data is not readable once transformed. They require a reverse process to reconstruct the initial message. Binary encoding enables more efficient communication.

Text encoding consists of transforming each of the data to be transmitted to a textual format, that is, an ASCII alphabet character.

In this encoding, structured data is also transformed into text. Conventions or protocols are needed to transform them. The data, once encoded, maintains readability to a certain extent, to the detriment of the amount of space they occupy.

Examples of text encoding

Text encoding is used by protocols like HTTP, SMTP, POP or IMAP.

In both codifications, the problem of **encoding the information structure** arises:

- Element bounding: how to mark the end or beginning of an element in a sequence.

- Type of information (and interpretation): how to express which datum represents each component of a collection, especially if we omit any when sending them or if their order is variable.

- Structuring (belonging to a set): how to indicate that a certain datum belongs to a group.

Data delimitation

Any encoding system must determine a way to separate the parts that compose a data sequence.

Some common options for data delimitation in Internet protocols are the following:

- "**Fixed length**": compact but inflexible. It is only recommended for data of consistently fixed length. In data of variable length, either a lot of space is wasted, or there is a risk of not being able to express some data, and there would be no way to fix it by omitting the length. It is an agreement between participants in a communication: to make adjustments, everyone would have to agree on the change.

- "**By length**": it is an improvement on the previous delimitation, but the length has to be known from the beginning. This can be inconvenient for the information producer. It may require generating all the data to be able to calculate the length and send this first datum. By contrast, the recipient will know exactly how much space the data will occupy and can then read the indicated bytes without difficulty.

- "**By length to chunks**": used when the data's total length is ignored from the beginning. Chunks of data can be sent as they are produced. This allows starting to issue the data earlier, and the producer can save memory dedicated to keep data ready for submission.

- "**Delimited**": instead of calculating the length, a symbol is reserved to separate data. If the delimiter appears between the data, a way to "ignore it" must be defined.

2.2. Data encoding formats

There are many data formats for transport, known as wire formats. These formats have been specified and are widely used. They allow overcoming diversity of languages, operating systems and machines. The most relevant formats are the following:

a) Text:

- XML: used in XML-RPC, SOAP.
- JSON: used in AJAX applications.

b) Binary:

- External data representation (XDR): used in RPC (v2; ONC-RPC).
- Abstract syntax notation (ASN.1): used in several protocols, such as, for example X.509, SNMP.
- Network data representation (NDR): used in DCOM, DCE-RPC.
- Common data representation (CDR): used in CORBA (GIOP, IIOP).
- Java remote messaging protocol (JRMP): used for communicating Java virtual machines (Java-RMI).

2.2.1. Text encoding with XML

The XML[3] language is, according to the Web Consortium, the universal format for structured documents and data on the web. That is why it has become a massively used standard for encoding interoperable data on Internet.

(3)XML stands for *extensible markup language*.

XML is used by various protocols to encode (marshal) exchanged data. For example, WEBDAV. It is an extension of the HTTP protocol to treat a web server as if it were an online file server. Another example is SOAP, a mechanism for remote invocation of operations.

However, XML has some constraints:

a) It is a text encoding format, and binary data must be sent encoded in Base64 format, or in addition to the XML document (using a link, as HTML does for images).

b) It may require a lot of text, although it can then be compressed, using a general compressor, like Gzip or Compress, or with a specific XML compressor, like the one studied by the work-group on binary characterization in XML.

Example of text encoding with XML

In the following example, you can see an XML document transported in HTTP protocol, to call an operation of a web service in XML-RPC protocol.

```
<?xml version="1.0"?>
<methodCall>
    <methodName>search</methodName>
    <params>
       <param>
```

```
            <value>
            <struct>
            <member> <name>name</name><value>Juan</value></member>
            <member> <name>age</name><value><i4>42</i4></value></member>
            </struct>
            </value>
        </param>
    </params>
</methodCall>
</methodCall>
```

As you can see, it is not a compact representation, but very informative and easy to understand. In XML-RPC, parameter types are specified as XML tags, like i4 for integer types, seen in the above example. By contrast, other more complex invocation mechanisms, like SOAP, take more advantage of XML possibilities.

More specifically, one of the points of XML documents that is worth highlighting is the validation or restriction of documents based on established contracts. In general, we can say a document complying with XML syntax is well formed. If the document also meets a set of additional constraints specified by a certain contract, it can be considered valid.

The grammatical **characteristics of the contract language** (elements of a language and their combinations) can be expressed in a section of the document, or in a separate document, in two ways:

1) DTD, or document type definition. It was the only format in existence at the beginning of XML. DTD syntax is not XML and lacks data types to restrict values.

2) XML Schema. It is a more recent format, based on XML, with data types and the ability to express more constraints. It is preferable to DTD, and we will focus on it from now on.

XML Schema is a very powerful tool. It allows defining data types and imposing a series of constraints. Thus, XML Schema allows the following constraints:

- Control over data: *length* (number of characters in string, binary), *maxlength* (maximum string length, binary), *lexical representation* (possible representations, for example, DD-MM-YYYY), *enumeration*, *maxInclusive* (maximum possible), *maxExclusive* (maximum exclusive), *minInclusive* (minimum possible), *minExclusive* (minimum exclusive), *precision* (number of digits), *scale* (digits with decimal part), *encoding* (binary).

- Control over refinement: inheritance and extension mechanisms.

- Control over extensibility: open, can be refined, closed.

These characteristics of XML and its XML Schema make it ideal as an interoperable data encoding language. In accordance with this, many contracts have been created based on XML schemas, such as SensorML, MathML, BioML, XMPP or SOAP, among others.

XML Schema data types

XML Schema supports the following data types: *string* (character sequence, ISO 10646, unicode), *Boolean*, *real*, *decimal* (*real*, *no Exp*), *integer* [–*T*,0,*T*], *non-negative integer* [0,*T*], *positive integer* [1,*T*], *non-positive integer* [–*T*,0], *negative integer* [–*T*,–1], *date-Time* (iso8601), *date* (*dateTime*), *time* (*dateTime*), *timePeriod* (*dateTime*), *binary* (data+codif. hex / base64), *uri* (rfc2396), *language* (sp, ca, en, rfc1766), in addition to allowing user-defined types.

2.2.2. Text encoding with JSON

JSON[4] encoding has acquired some relevance as a lightweight format for data exchange in AJAX environments, especially since the Web's evolution to Web 2.0. JSON is a subset of Javascript's literal object notation, but is not restricted to this language. It is used by many programming environments.

(4)JSON is an acronym for *Javascript object notation.*.

One of the advantages of JSON over XML as a data interchange format is that writing a JSON semantic parser is much simpler. In Javascript, JSON can be trivially parsed using the *eval* procedure. That is why it has become popular in remote invocation web environments, like AJAX. It has also been used as a REST[5] service encoding language and in invocation mechanisms, such as JSON-RPC.

(5)REST is an acronym for *representational state transfer.*

However, JSON does not specify data types or allow constraints, as XML and its XML Schema do. It is simply a textual format for grouping and structuring content. Even so, due to its simplicity and speed, it is used by Google or Yahoo servers, where the data flow volume between client and server is very large.

Example of text encoding with JSON

```
--> {"method": "postMessage", "params": ["Hello all!"], "id": 99}

<-- {"result": 1, "error": null, "id": 99}]]]
```

2.2.3. Binary encoding with XDR

XDR[6] is a data presentation protocol. It allows the transfer of data between machines with different architectures and operating systems. It supports all types of C language, except for the pointer to function. It defines the intermediate canonical form that it uses for transmitting data.

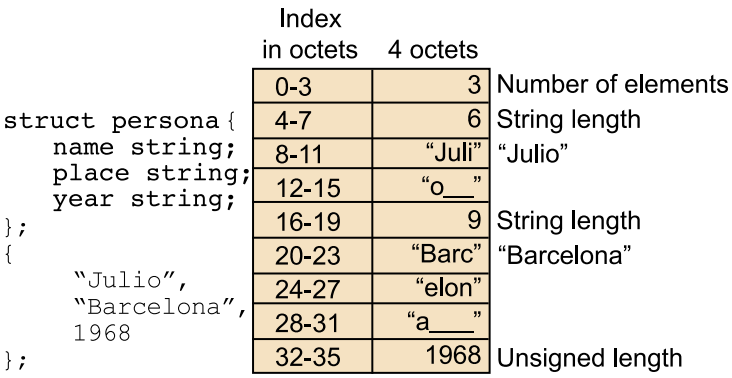
(6)XDR stands for *external data representation.*

The NFS distributed file system uses XDR as a data description language for data exchange. It is used with ONC RPC remote procedure calls.

XDR standard

XDR standard is defined in RFC 4506 (obsolete RFC 1014 and RFC 1832).

Figure 9



2.2.4. Binary encoding with ASN.1

Abstract syntax notation one or ASN.1[7] is a metalanguage for data representation, regardless of hardware, operating system, and its forms of internal data representation. It is used, among others, by the SNMP protocol. It is characterized for allowing compact representation of very long objects.

(7)ASN.1 is the acronym for *abstract syntax notation one*.

Characteristics:

- Format: {tag, length, value}.
- It uses 8-bit, multibyte type tags.
- Length in bytes of value: if < 127 occupies 1 byte.
- Value: for example, integer complement to 2, big-endian.

Encoding rules:

- BER: *basic* ...; not very efficient, a lot of redundancy, extension support.
- DER: *distinguished* ...; there are no options (for security); defined encoding length.
- CER: *canonical* ...; there are no options (for security); undefined encoding length.
- PER: *packet* ...; very compact, not very extensible.
- LWER: *light weight*; almost internal structure, fast encoding/decoding.

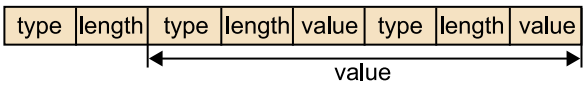
Length field:

If *length*: 0-127 bytes → 0, *length*, *value*

If *length* > 127 → 1, *value*, *value* (includes *length*)

Compound type:

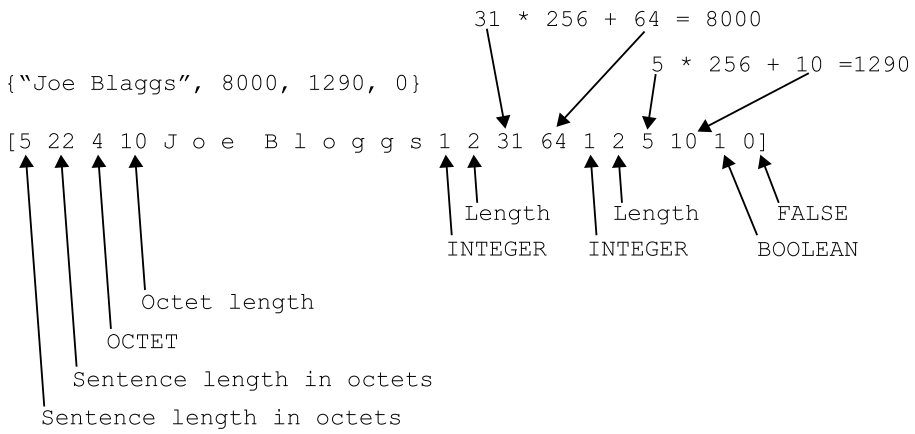
Figure 10



It has a lower performance than XDR because it is more complex, the alignment of values in words is worse, and (un)marshaling is somewhat more complex (for example, to deal with the *length* field).

Figure 11

```
struct employee {    employee :: SEQUENCE {
CHAR   NAME[32];     Name           OCTET STRING, --32 characters
int    salary;       Salary         INTEGER,
int    entryDate;    EntryDate      INTEGER,
int    sex;          Sex            BOOLEAN
};                   }
```



Finally, we present a summary table of some encoding metalanguages.

Name	Binary	Legible
ASN.1	Yes (BER, CER, DER, PER, ECN)	Yes (XER, via ECN)
BSON	Yes	No
Comma-separated values (CSV)	No	Yes
JSON	No, but see BSON	Yes
Netstrings	Yes	Yes
OGDL	Yes (Binary Schema 1.0)	Yes
Property list	Yes	Yes
Protocol buffers	Yes	Partial
S-expressions	No	Yes
Structured data exchange Formats	Yes	No

Name	Binary	Legible
Thrift	Yes	Partial
External Data Representation	Yes	No
XML	Partial (binary XML)	Yes
YAML	No	Yes

2.3. Remote invocation mechanisms

As we have seen, remote invocation mechanism, or RPC[8], is an extension of the function call mechanism. It allows remote codes to be called as if they were local. The client process calls a function that seems to be the client code (client stub), but only collects and sends network input data to another server code (server stub) which calls the requested function. The results of the call follow the reverse path.

(8)RPC stands for *remote procedure call*.

It would be ideal if the separation was not noticed (we would call it a *transparent mechanism*, in the sense that it is not perceived). But it is not easy, because a network like Internet has a more complex behaviour than a PC's internal bus. Packets are lost, disordered and duplicated, and sometimes the network fails.

To make the call "transparent" (invisible), new mechanisms for transporting it through the net must be introduced. This task is performed by new components, called stubs.

Stubs are the intermediate code responsible for managing remote connections between client and server, and establishing encoding protocols for passing parameters and remote invocation results. Thus, stubs are access and localization transparency architects in the middleware of remote invocation procedures.

To generate these stubs, the role of remote interfaces must be understood.

Locally, programming languages organize programs into modules which communicate with each other through calls to procedures. To control interaction between modules, each module's **interface** defines the signature of the procedures or functions that can be called.

In remote invocation, it is essential to define an interface of the procedures or services that will be called remotely. Based on the information of this interface's signatures and parameters, the tools for generating stubs can create the intermediate code needed for making these remote calls transparent to the programmer.

The calling mechanisms of ONC-RPC and RMI follow this calling model, operating in a homogeneous environment: the same programming language and, in the case of Java-RMI, the same machine (the Java virtual machine). This "separation" between the process machine requesting a service, and the one that carries it out, enables introducing variants to allow interaction between heterogeneous systems, in the following aspects:

- Architecture (format: size and organization of data).
- Programming language (affects the way of calling and moving arguments).
- Running environment (operating system).

Support for heterogeneity causes a greater complexity of the environment. More parameters can be "controlled", but the programmer must deal with them: the call is not transparent, but the programmer has to write "a lot" (decide several aspects) in order to make any remote call.

Thus, in remote invocation mechanisms that support heterogeneity, it is common to use an interface definition language (IDL).

This IDL is independent of the programming language, and stubs generators can be created from it, for different programming languages.

The fundamental advantage of supporting heterogeneity is that programs written in different programming languages, running on different operating systems, will be able to communicate. The Corba and SOAP calling mechanisms use this call model in a heterogeneous environment.

In a broad sense, the HTTP protocol used in the Web is also a RPC: it is a request/response protocol. All browsers call the same operations on servers: *doc = get(uri)*, *put (uri, doc)*, *post (uri, text)*, and editing operations that use XML-encoded data: *res = propfind (uri, args)*, *res = proppatch(uri, args)*, etc. A generalization of these mechanisms for RPC is what is adopted in, for example, XML-RPC or SOAP.

2.3.1. Naming Systems

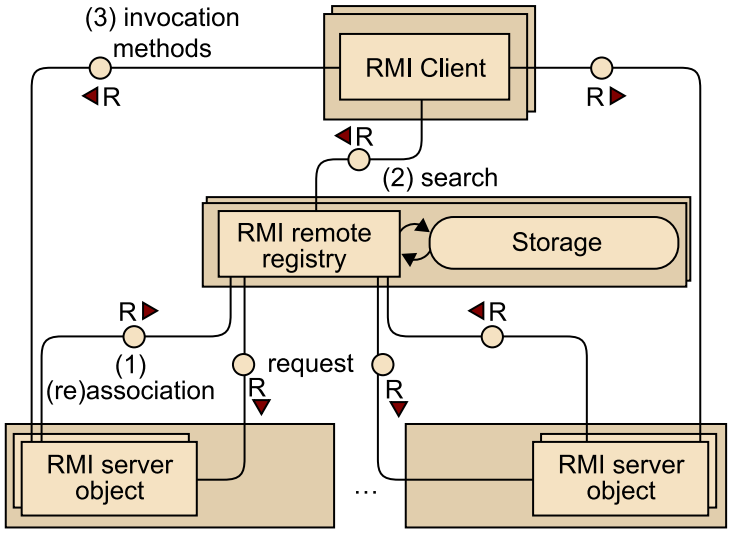
Another aspect to be taken into account is how to find a distributed service that runs on another machine in the network. The quickest way is to know the IP address and the port where the service runs. However, this goes against

the principle of access transparency, essential in any type of middleware. If the remote service changed its location on the network, all the clients would have to change the address.

To achieve this transparency, *naming systems* are used.

A **naming system** stores a list of associations between names (symbolic identifiers) and values (web addresses). For example, DNS stores associations between domain names and associated IPs, in ASCII.

Figure 12. How RMI Registry works



(1) The server registers the remote object.
(2) The client searches for the remote object in the registry.
(3) The client calls the method in the remote object.

In this way, we no longer have to know the IP of the website we want to request and, in addition, access transparency is achieved. That is, the server IP can be changed transparently to clients. In the case of remote procedure invocation, naming systems store the service's symbolic identifier (*echoService*), its IP and associated ports. Clients use the symbolic identifier to get the service's real address and they connect to it. This is the case of *RMIRegistry* in Java RMI or the CORBA Naming Service.

Finally, it should be noted that there are more sophisticated naming systems, such as UDDI or LDAP. They enable sophisticated searches for information associated with the service. However, in the context of web services, it is very common to locate services directly, using the URL.

2.3.2. Sockets

In 1983, a series of calls were developed within the Unix kernel scope, to facilitate the design of applications for intercommunicating on the Network. These calls, along with some auxiliary functions from UNIX's standard library, became the so-called socket programming interface.

A **socket** is an access point to communication services in the transport area. Every socket has an associated address that identifies it. By knowing it, we can establish communication with a socket so that it acts as the end of a two-way channel.

Sockets represent the implementation of the message passing paradigm that we have seen above. Sockets are important because most communication mechanisms end up using them as the final mechanism for data transmission. By this we mean that, for example, implementation of web services with SOAP, or remote invocation with RMI, ends up using a socket to transmit the information, even though this is transparent to the user of the mechanism.

The socket library is characterized for its simplicity and it offers the basic functionalities to create the socket, and read and write through it. There are several implementations of this library, among others, the Berkeley Sockets, which are offered by the Linux kernel. WinSock is another implementation used by Microsoft operating systems which is also based on the Berkeley Sockets.

Examples of methods offered by the Berkeley Sockets library

Some examples of methods offered by this library are:

- *socket*(): creates a socket of a certain type.
- *bind*(): used on the server to associate the socket with an address and a port.
- *listen*(): used on the server to put the socket into LISTEN mode.
- *connect*(): used on the client to connect to a server address and port. At the end of the call, the connection is established.
- *accept*(): used on the server to accept a connection.
- *send*() and *recv*(), or *write*() and *read*(), or *recvfrom*() and *sendto*(): are used to receive and read data from a socket.
- *close*(): the connection is closed (if it is TCP). Resources are released.
- *gethostbyname*() and *gethostbyaddr*(): are used to resolve names and addresses.
- *select*(): used for selecting sockets prepared for read/write, or that have errors.
- *poll*(): used for checking the socket's state.

Figure 13 illustrates the lifecycle of a TCP or connection-oriented socket. Once the socket has been created on the server and is bound to an address and a port, it waits for requests from the client. When the client calls the socket, the server accepts the request and communication is established.

Figure 13. Lifecycle of a TCP socket

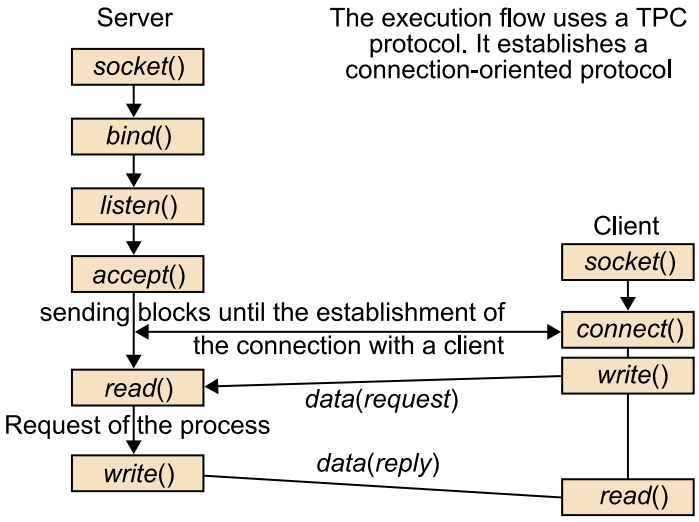
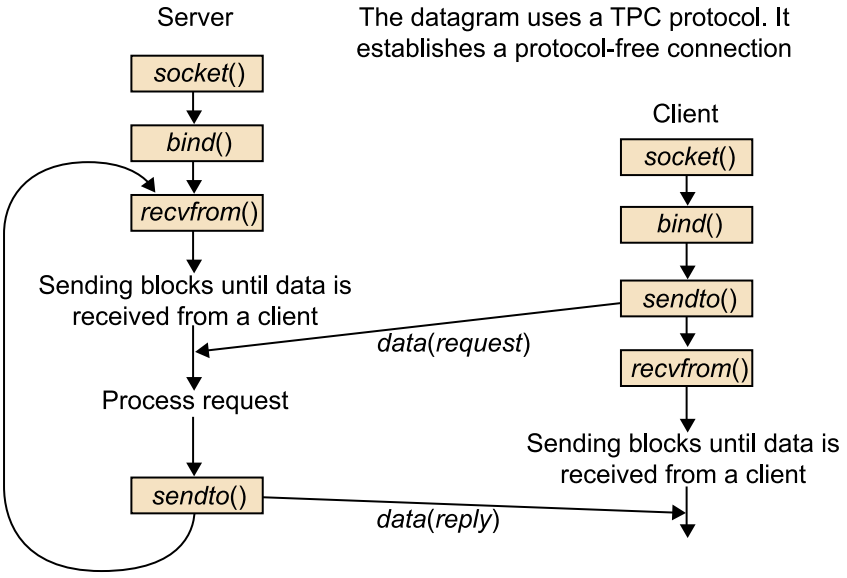


Figure 14 shows the flow diagram of the lifecycle of a connectionless UDP socket. A server socket is created, bound to an address and a port, and waits until it receives data from a client.

Figure 14. Lifecycle of a UDP socket



Sockets are communication mechanisms offered by the transport layer. This layer is not responsible for providing security and protection mechanisms for information transmitted through a channel. Therefore, the upper layers must be responsible for protecting data transmitted through a channel. Among others, there are application-level protocols to enable secure communications,

like Transport Layer Security (TLS) or Secure Socket Layer (SSL). These provide cryptographic mechanisms to secure end-to-end connections in the transport layer.

2.3.3. RPC Protocols

Some of the most relevant RPC protocols will be presented in this subsection.

RMI

The Java remote method invocation is a distributed object model, specifically designed for the Java language. Therefore, it maintains the Java semantics for the local object model, thus facilitating the implementation and the use of distributed objects. In the Java distributed objects model, remote object methods can be called by objects in a different virtual machine (VM). Objects of this type are described by one or more remote interfaces that contain the definition of the methods of the object that is possible to call remotely.

Remote method invocation (**RMI**) is the action of calling a method of a remote object and has exactly the same call syntax as that of calling a local object.

The intended goals of supporting distributed objects in Java are:

- Providing remote invocation of objects in different VMs.
- Allowing calls to servers from applets
- Integrating distributed object models into the Java language in a natural way, preserving the Java object semantics as much as possible.
- Making it as simple as possible to write distributed applications.
- Preserving the security provided by the Java environment.
- Providing various semantics for remote object references (persistent, non-persistent, and “delayed activation”).

In general, remote invocation will involve the following steps:

A) In the server

1) Define the interface of the service that will be remotely accessible. To make an object remote in RMI, it must be declared that it implements the *Remote* interface. As remote invocation can fail, each interface method also declares the *java.rmi.RemoteException* exception, in the *throws* section.

2) Implement the service code complying with this interface, and generate the corresponding stubs. In RMI, the stubs compiler or generator is called rmic, and is responsible for generating the intermediate code. In current versions, it is no longer necessary to generate server stubs (skeletons) at compile time (in versions prior to 1.2 it is required). The RMI environment uses runtime reflection to make the calls.

3) Finally, the server object is instantiated and published or registered in a naming system to make it accessible to clients. In RMI, the naming system is called *RMIRegistry*, accessible via the API *java.rmi.Naming*.

B) In the client

1) Generate the client stubs from the remote service interface. This can be done at compile or runtime. In compilation, rmic generates the stubs (by default, in JRMP protocol) required for remote invocation. However, it is not necessary with dynamic proxies, which allow bypassing rmic. This results in dynamic stub generation at runtime.

2) The client can now locate the server using the naming system, and call operations remotely. In this phase, client stubs can also be obtained dynamically, by downloading them from a server when locating the service.

Finally, it should be noted that Java RMI allows passing by reference[9] or passing by value[10]. To call remote object methods, the machine where the remote object is located sends a stub, instead of a copy of the object (its proxy or broker acting in reference to the remote object). This is dynamically constructed and loaded during execution, when necessary.

To make an object travel, it must be serialized (*Java object serialization*). The data and implementation code have to be moved, and a copy is created in the remote location. This implies that any serializable Java class can be passed by parameter between remote RMI objects.

[9] **by reference**: "remote" object methods can be called from "far" (other virtual machines).

[10] **by value**: the object "migrates" completely through the network; it results in a local object and its methods are called locally.

Reflection in Java

Reflection is a program's ability to deduce an object's properties. These can be the list of methods, arguments, and types. It allows dynamically building a stub for any object at runtime.

Example of calling a remote method in a Java application

Here is an example of a small Java application that calls a remote method. The Java object *HelloImpl.java* contains the remote object implementation (the server):

```
public class HelloImpl extends UnicastRemoteObject implements Hello {
public HelloImpl() throws RemoteException {
   super();
}
public String sayHello() {
   return "Hello friends!";
}
```

```
public static void main(String args[]) {
if (System.getSecurityManager() == null) {
    // Create and install a security manager
    System.setSecurityManager(new RMISecurityManager());
}
try {
   HelloImpl obj = new HelloImpl();
   // Associate this instance with the name    "HelloServer"
   Naming.rebind("//myhost/HelloServer", obj);
   System.out.println("HelloServer associated with the registry");
} catch (Exception y) {}{
   System.out.println("Error" + e.getMessage());
}
}
}
```

A client locating the server object gets a dynamic stub in return, and calls the remote method *sayHello*():

```
try {
   obj = (Hello)Naming.lookup("/server/HelloServer");
   message = obj.sayHello();
} catch (Exception y) {
   System.out.println("Exception:" + e.getMessage());
}
```

In their API, RMI and Java offer objects to manage the security of the application, such as, for example, preventing an applet from accessing a disk, or controlling the ports it can open to establish connections. One of these objects is the SecurityManager, to which different security policies can be specified for an application.

It should be noted that similar mechanisms have subsequently appeared in other programming languages, like Remoting in Csharp (C#).

SUN-RPC

RFC 1831 describes the SUN-RPC that was designed for client-server communication by SUN's NFS network file system. It is also called ONC-RPC (*open network computing*) and is used by various SUN operating systems, as well as in NFS distributions. In this protocol, developers make use of RPC through UDP or TCP connections, according to their preferences. SUN-RPC uses XDR as its data encoding metalanguage, and a stub generator for C programming language, called rpcgen.

Further reading

R. Srinivasan (August, 1995). "RPC: Remote Procedure Call Protocol Specification Version 2". *Internet RFC 1831*.

The client program imports the appropriate service interface and calls remote procedures, such as *READ* or *WRITE*. The stubs generated by rpcgen are responsible for marshaling and unmarshalling information, and making the remote invocation process transparent. SUN-RPC does not have a naming system, instead links are made locally by a service, called a port mapper, running on each computer. Each port mapper maintains information about the local services running on it. When a client imports an interface, it must specify the server address, as well as the number identifying the program and version. These are used for discerning the service to be returned.

In the following code, we see how a client calls a read method of a remote server.

```
/* File: C.c - client of the FileReadWrite service. */
#include <stdio.h>
#include <rpc/rpc.h>
#include "FileReadWrite .h"
main(int argc, char ** argv)
{
   CLIENT *clientHandle;
   char *serverName = "coffee";
   readargs a;
   Data *data;
   /* creates the socket and gets a pointer to the service*/
   clientHandle= clnt_create(serverName, FILEREADWRITE, VERSION, "udp");
    if (clientHandle==NULL){
      clnt_pcreateerror(serverName); /* we cannot contact server */
      exit(1);
   }
   a.f = 10;
   a.position = 100;
   a.length = 1000;
   data = read_2(&a, clientHandle);/* we call the remote read */
   ...
   clnt_destroy(clientHandle);
   /* close the socket */
}
```

The server side implementer uses the header file created by rpcgen as an interface to define the methods that the server must offer.

The server code is thus a compendium of method implementations plus a main program and a set of routines for marshaling and unmarshalling, created by rpcgen.

```
/* File S.c - server of the FileReadWrite service */
#include <stdio.h>
#include <rpc/rpc.h>
#include"FileReadWrite.h"
void * write_2(writeargs *a)
{
   /* do the writing to the file */
}

Data * read_2(readargs * a)
{
   static Data result;
```

```
    /* must be static */
    result.buffer = ...
    /* w read the file */
    result.length = ...
    /* amount read from file */
    return &result;
}
```

SUN-RPC's main characteristics can be summarized as follows:

- It uses XDR as its encoding language.
- It does not guarantee “at maximum once” semantics.
- It works over UDP, enabling broadcast.
- It is based on selection: UDP selects the process, RPC, the procedure.
- Port mapper (port 111): program number→port.
- Messages limited by a maximum IP size (32 kB).
- Authentication on each request.
- Stateless server.
- If the relay reaches the server while the response travels to the client, it does not notice the duplicate; it does notice it “during the process”, and eliminates it. This short-term memory is not a problem on a local network.

XML-RPC or web call (HTTP)

The web’s transfer protocol, HTTP, was conceived as a request/response mechanism for exchanging messages and calling predefined methods (*GET*, *HEAD*, *PUT*, *POST*, *DELETE*, *PROPFIND*, *PROPPATCH*). With the emergence of the NCSA HTTPD server, a simple mechanism was defined (common gateway interface or CGI[11]) to call commands in the Unix filters' style; the web server calls a command with standard input and arguments provided in the HTTP request, and with standard output back to the HTTP response. These services receive content of a HTML form from a browser, encoded in a simple format, and return a HTML code for a browser to present. They are used for people using a browser to access services, but not for a process to call operations.

[11]CGI stands for *common gateway interface.*

However, despite the orientation of the HTTP protocol for obtaining and accessing remote documents, several remote invocation mechanisms have used it as a client-server communication protocol. This is primarily because it is an open and flexible protocol, widely used in Internet, but also due to security

reasons. Most organizations allow external HTTP transit, but limit unknown TCP or UDP communications with firewalls. For this reason, CORBA or RMI, working over TCP or UDP, can encounter communication problems between remote points of the Internet's network.

One of the first remote invocation mechanisms to adopt HTTP as a communication protocol is called XML-RPC. As the name suggests, this protocol chose XML as the data encoding language between client and server. The use of XML and HTTP makes XML-RPC an ideal protocol for intercommunicating heterogeneous services in different languages and platforms.

Call to an operation (in Python language):

```
-> x.search({'name':'juan','age':42})


<- {'id': 123456}
```

Calling the above operation triggers these object exchanges in HTTP:

Call:

```
<?xml version="1.0"?>
<methodCall>
   <methodName>search</methodName>
   <params>
      <param>
         <value>
            <struct>
               <member>
                  <name>name</name>
                  <value>juan</value>
               </member>
               <member>
                  <name>age</name>
                  <value><i4>42</i4><value>
               </member>
            </struct>
         </value>
      </param>
   </params>
</methodCall>
</html>
```

To build a web service, different components are necessary:

- A request/response protocol to communicate with a process on a server: HTTP.

- A format for representing the data exchanged in a call: XML can be the basis for building it, although others, such as JSON, can also be used.

- A programming interface for calling web application services. XML-RPC defines a set of types, limited by the fact that each language must map its own types. Also, the types are XML tags, and not predefined XML Schema types. It is a very simple approach, to enable simple implementation. As a consequence, XML-RPC libraries have been implemented in many existing programming languages.

XML-RPC data types

<i4> or <int>
<boolean>
<string>
<double>
<dateTime.iso8601>
<struct>
<array>
<base64>

Response:

```
<?xml version='1.0'?>
<methodResponse>
   <params>
      <param>
         <value>
            <struct>
               <member>
                  <name>id</name>
                  <value><int>123456</int></value>
               </member>
               <member>
                  <name>name_complete</name>
                  <value>juan perez</value>
               </member>
            </struct>
         </value>
      </param>
   </params>
</methodResponse>
```

It should be noted that XML-RPC has been designed to be very light and very simple. Therefore, it is not comparable in functionalities to systems like CORBA or RMI. Thus, XML-RPC does not define remote interfaces, has no stubs compiler or generator, has no naming systems, has no exceptions, and does not offer pass by reference. It is designed for stateless services with a very simple interface. The standard server URL is used to locate the service. Furthermore, the client must know the service's parameters and types beforehand, in order to call it; all responsibility lies with the client. That is why it is not necessary to generate stubs.

SOAP

Calling distributed objects using RMI, CORBA, or DCOM technologies has a fundamental interoperability problem between heterogeneous systems. Consequently, complex gateways and bridges had to be used, to allow the translation of protocols and their communication. Thus, systems developed in Microsoft technologies (DCOM) found it difficult to communicate with CORBA-based systems. In addition, the use of owner ports and binary encoding also caused problems, due to firewalls and protected corporate networks. To solve these problems, the current trend is to rely on open and interoperable protocols, such as HTTP as a transport channel, and XML as an information encoding language. Along these lines, the Userland company began by defining the XML-RPC protocol as a simple remote invocation protocol, based on HTTP and XML. Later, Userland and Microsoft developed a more complete and complex remote invocation protocol, called SOAP.

SOAP stands for *simple object access protocol*, the specification of a protocol for the exchange of structured information on web services implementation. SOAP uses XML as the metalanguage for encoding data. It uses other protocols, like HTTP and RPC, for message negotiation and transmission. SOAP offers basic messaging services and functionalities to enable defining other web services at a higher level. In a way, it can be considered as the fundamental layer of the web services protocol stack.

1) SOAP messages

As we have seen, SOAP is a simple XML-based protocol which allows applications to exchange information over HTTP.

A SOAP message is an ordinary XML document. It contains the following elements:

- Envelope element, identifying the XML document as a SOAP message.
- Header element, containing header message information.
- Body element, containing call and response information.
- Fault element, containing error and status information.

All elements are declared in the Envelope's default namespace.

Skeleton of a SOAP message

```
<?xmlv version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Header>
```

```
...
</soap:Header>
<soap:Body>
...
   <soap:Fault>
      ...
   </soap:Fault>
</soap:Body>
</soap:Envelope>
```

a) The Envelope element

The Envelope element is the root of the SOAP message. It defines the document as a SOAP message. The *xmlns:soap namespace* always takes the value "http://www.w3.org/2001/12/soap-envelope", because it defines the Envelope as a SOAP Envelope. If another namespace was used, the application would generate an error, and discard the message.

The *encodingStyle* attribute is used to define the document's data types. The attribute can appear in any SOAP element and applies to the element itself and its descendants (children). A SOAP message does not have a type definition by default, but must always be linked to one. To do this we will use:

```
soap:encodingStyle="URI"
```

b) The Header element

The Header element is optional and contains application-specific information, like authentication, payment, etc.

If there is a Header element, it will be the first of the children (descendants) of the Envelope element.

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Header>
   <m:Trans xmlns:m="http://www.uoc.edu/transaction/" soap:mustUnderstand="1">
      234
   </m:Trans>
</soap:Header>
...
...
</soap:Envelope>
```

The example contains a Header with the *Trans* element, *mustUnderstand* attribute, with value 1, and a value, 234.

SOAP defines three attributes in the default namespace "http://www.w3.org/2001/12/soap-envelope". These attributes are *mustUnderstand*, *actor*, and *encodingStyle*.

The header attributes define how the message recipient should process it. The *mustUnderstand* attribute is used to tell the recipient if it must know how to process the element, or not. If the element *mustUnderstand* = "1", the recipient must understand the value it receives for this element. If it is not recognized, the recipient will fail, indicating that it has not been able to process the Header.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
   <m:Trans xmlns:m="http://www.uoc.edu/transaction/" soap:actor="http://www.uoc.edu/appml/">
      234
   </m:Trans>
</soap:Header>
...
...
</soap:Envelope>
```

The *actor* attribute is used to direct the Header element to a specific recipient.
The syntax is:

```
soap:actor="URI"
```

c) The Body element

Each of the children of the Body element must indicate their namespace.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body>
   <m:GetResults xmlns:m="http://www.uoc.edu/results">
      <m:Item>Networks</m:Item>
   </m:GetResults>
</soap:Body>

</soap:Envelope>
```

The example makes a request for the results of the UOC's *Networks* subject. Note that the *m:GetResults* and *item* elements are application-specific. The answer would look like:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body>
 <m:GetResultsResponse xmlns:m="http://www.uoc.edu/results">
 <m:Result>Notable</m:Result>
 </m:GetResultsResponse>
</soap:Body>

</soap:Envelope>
```

d) The Fault element

It is an optional element, used to indicate errors and status information. If this element is present, it will always be a descendant of the Body element. The Fault element will only appear once, as a child of the Body element.

The Fault element has the following sub-elements:

Sub-element	Description
<faultcode>	Code to identify the error.
<faultstring>	Explanation of the error.
<faultactor>	Information about who caused the error.
<detail>	Application-specific information.

2) SOAP Fault codes

The codes of the faultcode sub-element are:

Error	Description
VersionMismatch	An invalid namespace has been found in the Envelope element.
MustUnderstand	A child of the Header element with the attribute *mustUnderstand* with value 1 has not been understood.
Client	The message is malformed or the content is incorrect.
Server	There has been an error on the server, or the message could not be processed.

3) SOAP interfaces: Web Service Description Language (WSDL)

As in any remote invocation technology, a contract must be established to define the offered service. This contract will define the name of the methods, the parameter types, and the result. WSDL is an XML language that defines the name of the methods, and accepted parameter types. Also, there will be tools in each language to generate stubs from the contract (WSDL2Java, WSDL2C#, WSDL2Python...).

Unlike Java RMI, in SOAP, both clients and servers can be implemented in any programming language (with SOAP libraries, of course). WSDL is also a W3C standard.

Let's look at a WSDL example for the following Java class:

```
public class Calculator {
public int add(int y1, int y2) {
   return y1 + y2;
}
public int subtract(int y1, int y2) {
   return y1 - y2;
}
}
```

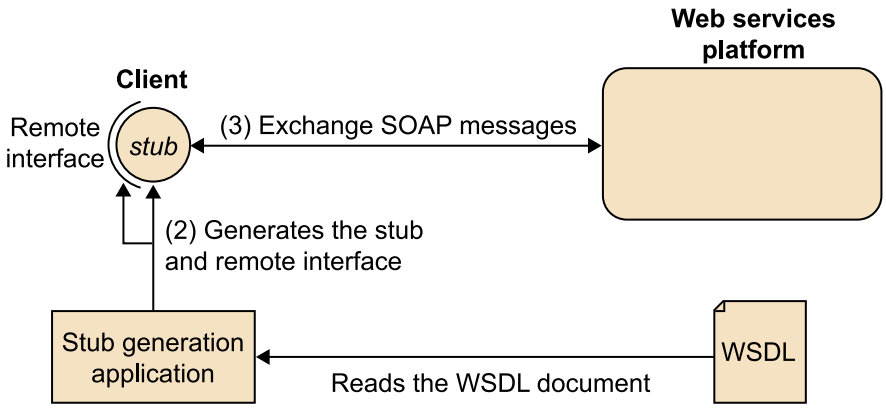
The WSDL would look like this:

```
<wsdl:definitions targetNamespace="http://localhost:8080/axis/Calculator.jws">
<!--
WSDL created by Apache Axis version: 1.2RC3
Built on Feb 28, 2005 (10:15:14 EST)
-->
<wsdl:message name="subtractRequest">
<wsdl:part name="i1" type="xsd:int"/>
<wsdl:part name="i2" type="xsd:int"/>
</wsdl:message>
<wsdl:message name="subtractResponse">
<wsdl:part name="subtractReturn" type="xsd:int"/>
</wsdl:message>
<wsdl:message name="addResponse">
<wsdl:part name="addReturn" type="xsd:int"/>
</wsdl:message>
<wsdl:message name="addRequest">
<wsdl:part name="i1" type="xsd:int"/>
<wsdl:part name="i2" type="xsd:int"/>
</wsdl:message>
<wsdl:portType name="Calculator">
<wsdl:operation name="add" parameterOrder="i1 i2">
```

```
<wsdl:input message="impl:addRequest" name="addRequest"/>
<wsdl:output message="impl:addResponse" name="addResponse"/>
</wsdl:operation>
<wsdl:operation name="subtract" parameterOrder="i1 i2">
<wsdl:input message="impl:subtractRequest" name="subtractRequest"/>
<wsdl:output message="impl:subtractResponse" name="subtractResponse"/>
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="CalculatorSoapBinding" type="impl:Calculator">
<wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="add">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="addRequest">
<wsdlsoap:body encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
namespace="http://DefaultNamespace" use="encoded"/>
</wsdl:input>
<wsdl:output name="addResponse">
<wsdlsoap:body encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
namespace="http://localhost:8080/axis/Calculator.jws" use="encoded"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="subtract">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="subtractRequest">
<wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://DefaultNamespace" use="encoded"/>
</wsdl:input>
<wsdl:output name="subtractResponse">
<wsdlsoap:body encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
namespace="http://localhost:8080/axis/Calculator.jws" use="encoded"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="CalculatorService">
<wsdl:port binding="impl:CalculatorSoapBinding" name="Calculator">
<wsdlsoap:address location="http://localhost:8080/axis/Calculator.jws"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

In this file, we find a definition of operations offered by the service (portType, operation), of the request and response messages exchanged (message) – including type of parameters (type) –, and communication protocols (bindings).

Figure 15



In general, it will not be necessary for programmers to know the WSDL language format. There are automated tools to generate the WSDL, from the code in each programming language. However, only the basic types are standardized in SOAP.

Thus, for example, the standard correspondence for Java is as follows:

```
xsd:base64Binary byte[]
xsd:boolean boolean
xsd:byte byte
xsd:dateTime java.util.Calendar
xsd:decimal java.math.BigDecimal
xsd:double double
xsd:float float
xsd:hexBinary byte[]
xsd:int int
xsd:integer java.math.BigInteger
xsd:long long
xsd:QName javax.xml.namespace.QName
xsd:short short
xsd:string java.lang.String
```

In figure 15, we can see that SOAP does use XML in its standard type encoding of XML Schema, like xsd:string, or xsd int, among others. Actually, other XML encoding schemas can be used.

What is described in this way follows the schema referenced with the URL http://schemas.xmlsoap.org/soap/encoding/.

The use of other complex types depends on each language and environment. Sometimes encoding has to be implemented (or some tool in the environment must be used). In general, most implementations accept arrays as a complex data structure. However, compound collections may not be recognized by other implementations of SOAP in different languages.

In terms of security, SOAP offers extensions for inclusion of certificates in each message. In addition, a SOAP message can be sent over a channel, secured with TLS or SSH, in the same way we do with HTTP connections.

2.3.4. Location of services

To call remote services from a client, they must first be located by means of some method. For this, it is possible to use direct locations to the service (including the server and port) or through a remote naming system. In CORBA, for example, direct locators are called IOR[12] and the naming system is *CORBA naming service*.

(12)IOR stands for *interoperable object reference*.

In RMI, we can use the *RMIRegistry* as a naming system for locating objects. To locate SOAP services, we can use its WSDL's URL as a direct locator, or use the service called UDDI[13].

(13)UDDI stands for *universal description, discovery and integration*.

However, due to the simplicity of direct URL localization, developers tend to avoid using the UDDI, in favour of the direct method. Along these lines, web services on Internet publish their WSDL's URL on web pages, so clients can call them.

There are also specialized websites, such as www.xmethods.com, with lists of public services, including the URL of each one of them. The use of UDDI is more associated with business models in which, similar to the yellow pages, clients can discover the business web services they need. Along these lines, Microsoft offers its Microsoft UDDI Business Registry as an access tool.

Links of interest

IBM and Microsoft also offer web services registry UDDI directories:

Microsoft© Server and Cloud Platform.

Thus, the discovery level allows a service consumer to obtain service descriptions. The two available mechanisms are universal description, discovery and integration (UDDI), and the web services inspection language (WS-inspection).

This allows closing the cycle of a web service:

- A service provider publishes its offers in a service registry.
- A service registry collects offers and can be examined by a service consumer.
- A service consumer searches for a certain service in a service registry and, when he/she finds it, binds with the selected service provider to call operations with SOAP.

In turn, more complex systems such as Microsoft's .NET, or the United Nations' ebXML, are built over these levels, to support integration services between companies.

2.3.5. REST

The acronym REST[14] names an HTTP and XML-based application communication protocol.

(14)REST stands for *representational state transfer*.

Restful

Systems that follow REST principles are often called RESTful.

Originally, REST referred to an architecture of applications in the net, but the term has been generalized to refer to any web interface using HTTP and XML for information exchange. It differs from SOAP or XML-RPC because REST does not use any additional abstractions outside of those offered by HTTP.

Therefore, REST encompasses two concepts: an application architecture, and a form of communication between applications, based on XML. In this subsection, we are particularly concerned with REST's second definition.

Application architecture

An architecture style is a set of constraints to be applied when creating something. A software architecture style is a set of constraints, describing the characteristics for guiding the implementation of a software system. REST is an architectural style, used for creating software. Using this software, clients (user agents) can make requests for services (endpoints). REST is a way to implement a client/server architecture style. In fact, REST explicitly relies on the client/server architecture style.

To understand REST, we have to bear in mind how the web works today, and more specifically, the HTTP:

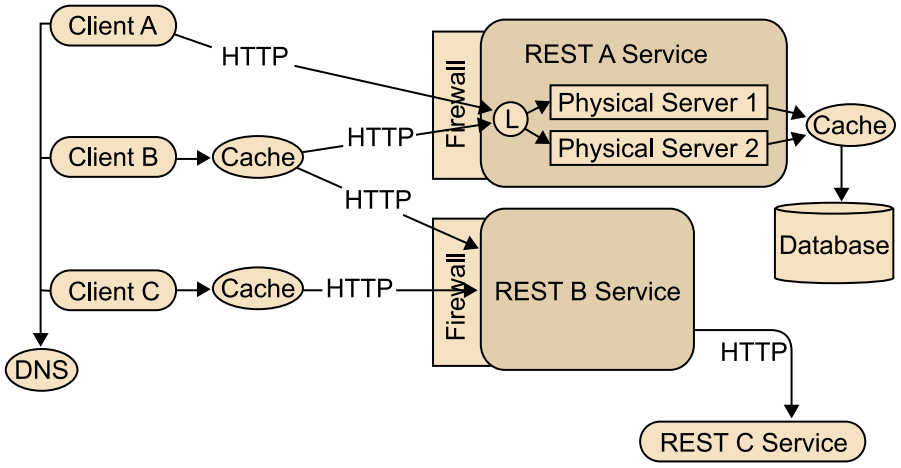
- On the web, user agents interact with resources, and resources are everything that can be named and represented. Each resource can be resolved through a single interface (URI).

- Interaction with resources (located through their unique URI) is done through a uniform interface, using standard HTTP "verbs" (*GET*, *POST*, *PUT* and *DELETE*). The declaration of resource type is also important in the interaction. It is designated by the *HTTP Content-Type* header (XHTML, XML, JPG, PNG, and JSON are some well-known media types.)

- Resources describe themselves. All the information needed to process a request for a resource is within the same request (allowing services to be stateless).

- Resources contain links to other resources (multimedia links).

Figure 17



A small example

Let's imagine that the UOC wants to offer a service for consulting results. This service could give us our result for the subject we requested.

When creating the RESTful service, we could answer the following questions:

1) What will the resources be?

2) Which URIs will be used to identify the resources?

3) Which HTTP "verbs" will we allow?

The resources in our REST service are the subjects, the students and the results. The XML representation of these three resources would be constructed in this way: *results*, *subjects* and *students*. (Representation of resources can be in another definition metalanguage. It does not have to be XML.)

Once the resources are established, their URIs can be defined. The definition only needs to be relative, since the absolute identification will be given by the server's URL, where the service will run. In our example, we could use /{student}, /{subject} and /{result} as the URI of each of the resources. Thus, /{Josep Armengol}/{Structure of computer networks (05.098)/{B} would correspond to the result B of the student Josep Armengol, for the subject *Structure of computer networks* (05.098).

Finally, to determine the operations that can be done on the resource, we only need to ensure it will be in read-only mode, as it is a query application for students; therefore, we will allow the *GET* operation.

Advantages of REST

REST is becoming a widely used mechanism. It offers a set of advantages, making it ideal for large-scale web applications. Its characteristics are as follows:

- **Cacheable**: as a REST request is still a stateless HTTP request, it is easily stored in cache memories in the network. This makes the mechanism favour system scalability. Consecutive requests only consult the server once to reduce the load on the system.

- **Scalable**: the fact that REST mechanisms are HTTP-based allows them to inherit HTTP properties. Therefore, REST is widely scalable. This feature is favoured by the fact that resources are self-descriptive and, consequently,

do not establish dependencies or requirements for the maintenance of a state.

- **Unalterable**: as REST uses HTTP verbs (*GET*, *POST*, *PUT* and *DELETE*), the protocol cannot be altered and thus, it maintains the simplicity of its use.

- **Interoperable**: resources are defined in a language independent of client and server machines' codes. Communication is carried out using a standard protocol, like HTTP. This allows different technologies and hardware architectures to use this communication mechanism, without the need to know endpoint particularities.

- **Simple**: REST offers four basic primitives, enabling interaction with resources described with URI. Any set of functionalities can be developed with these tools, and it turns REST into an extremely simple protocol.

Finally, some people may think that REST is a bit restrictive, if the systems to be built are very complex, since certain functionalities may seem difficult to implement with the primitives (*GET*, *POST*, *PUT* and *DELETE*). As always, this decision is left to the developers.

Summary

The module has presented the main communication paradigms. These include message passing, where two objects, processes or applications, use primitives for explicit bidirectional communication. Message passing is one of the most widely used communication paradigms and is implemented, for example, by sockets.

We have also seen that there are other paradigms, like shared memory. If the processes are on the same machine, they can share an address space that allows them to exchange information. In the case of distributed applications, there are implementations to allow memory sharing remotely. Tuple spaces is one of these.

Remote process invocation and remote method invocation are the names given to the paradigm of calling methods remotely. Processes or objects can call methods on other remote applications or objects as if they were the same machine. This paradigm requires mechanisms for encoding the information to be transmitted, because processes or objects may be implemented in different programming languages, or may be running on different hardware. As maximum exponents of this paradigm, we have seen RMI, XML-RPC, and SOAP.

The module has introduced us to the event-driven communication paradigm. In this paradigm, communication is completely asynchronous, and allows decoupling between the senders of events and their receivers.

We have also seen the publish-subscribe paradigm. This is widely used in object-oriented models, in development of user graphic environments, and in large-scale distributed applications.

Bibliography

Birman, K. P. (2005). *Reliable distributed systems: technologies, web services, and applications. Springer middleware for communications*. Qusay Mahmoud: Wiley 2004.

Burke, B. (2006). *Enterprise JavaBeans 3.0*. O'Reilly Media.

Cerami, E. (2002). *Web services essentials* (O'Reilly XML). O'Reilly Media.

Cornell, G.; Horsmann, C. (1996). *Core Java*. Prentice Hall.

Coulouris, G.; Dollimore, J.; Kindberg, T. (2005). *Distributed systems: concepts & design* (4th ed.). Addison-Wesley. [Spanish translation: *Sistemas distribuidos: conceptos y diseño* (2001, 3rd ed.). Pearson.]

Englander, R. (2002). *Java and SOAP*. O'Reilly Media.

Loosemore, S.; Stallman, R. M.; McGrath, R., and others (1992). *The GNU. C library reference manual*. Boston: Free Software Foundation.

Márquez García, F. M. (1996). *UNIX. Advanced programming*. Madrid: Ra-ma.

Oberg, R. (2001). *Mastering RMI: developing enterprise applications in Java and EJB*. John Wiley & Sons.

Orfali, R. (1998). *Client/server programming with Java and CORBA*. New York: Wiley & sons.

Pacheco, P. (2000). *Parallel programming with MPI*. Morgan Kaufmann.

Peterson, L.; Davie, B. (2003). *Computer networks: a system approach* (3rd ed.). EUA: Morgan-Kauffman.

Rifflet, J. M. (1992). *UNIX Communications*. Madrid: McGraw-Hill.

Sinha, P. (1997). *Distributed operating systems: concepts & design*. IEEE Press.

Sridharan, P. (1997). *Advanced Java networking*. Prentice Hall.

St. Laurent, S. (2001). *Programming web services with XML-RPC*. Reilly Media.

Stevens, W. R. (2005). *Advanced programming in the UNIX environment* (2nd ed.). Addison-Wesley Professional.

Tanembaum, A.; Steen, M. (2007). *Distributed systems: principles and paradigms, 2/E*. Prentice Hall.

Web pages:

RMI: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

W3C, Web Services Architecture: http://www.w3.org/TR/ws-arch/

JAX-WS: https://jax-ws.dev.java.net/

DCOM: http://www.microsoft.com/com/default.mspx