Sistemas Distribuídos

Interprocess Communication and Synchronization

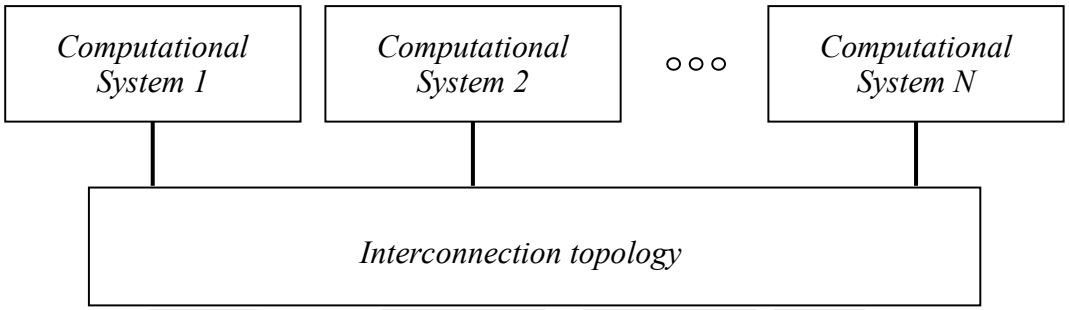
Message Passing

António Rui Borges

Summary

- *Communication system*
- *Programming interface*
  - *TCP protocol*
  - *UDP protocol*
- *Transformation of a concurrent solution into a distributed message passing solution*
  - *Transformation principles*
  - *Message structure*
  - *Client architecture*
  - *Server architecture*
- *Suggested reading*

Communication system - 1



Parameters affecting the performance of a communication system are

- *latency* – delay that occurs after the execution of a *send* operation and the beginning of data reception (it can be envisaged as the transfer of an empty message)
- *data transfer rate* – speed of data transmission between the sender and the receiver
- *bandwidth* – system *throughput* (volume of message traffic per unit of time).

$$message\ transmission\ time = latency + length / data\ transfer\ rate$$

Communication system - 2

When considering *real time multimedia* applications, there is another relevant property which plays an important role, *quality of service*. It describes the system capability to meet *deadline* constraints imposed by the transmission and the processing of data flows in continuum. In order for these operations to occur in a satisfactory manner, one requires an upper limit for latency and a lower limit for bandwidth of the associated data channels.

Present day communication systems are quite reliable. Failures are usually related to errors in the software at the sender or the receiver side than to network errors. Thus, it is common practice to transfer to the applications the responsibility of dealing with the detection and the correction of the remaining errors, a procedure that is known as the *end-to-end argument*.

Communication system - 3

From the application programmer point of view, the communication system must be viewed in an integrated and abstract manner masking the underlying complexity of the diverse physical networks it encompasses.

Thus, network software is arranged in a hierarchy of layers. Each layer presenting an operational interface to the layers above it that describes the properties of the communication system at this level in a logical fashion. A layer is represented by a software module present in every computer system attached to the network.

Each module appears to communicate directly with the corresponding module in another computer system, but in reality data is not transmitted directly between the modules at each level. Each layer of the network software communicates instead by local procedure calls with the levels above and below it.

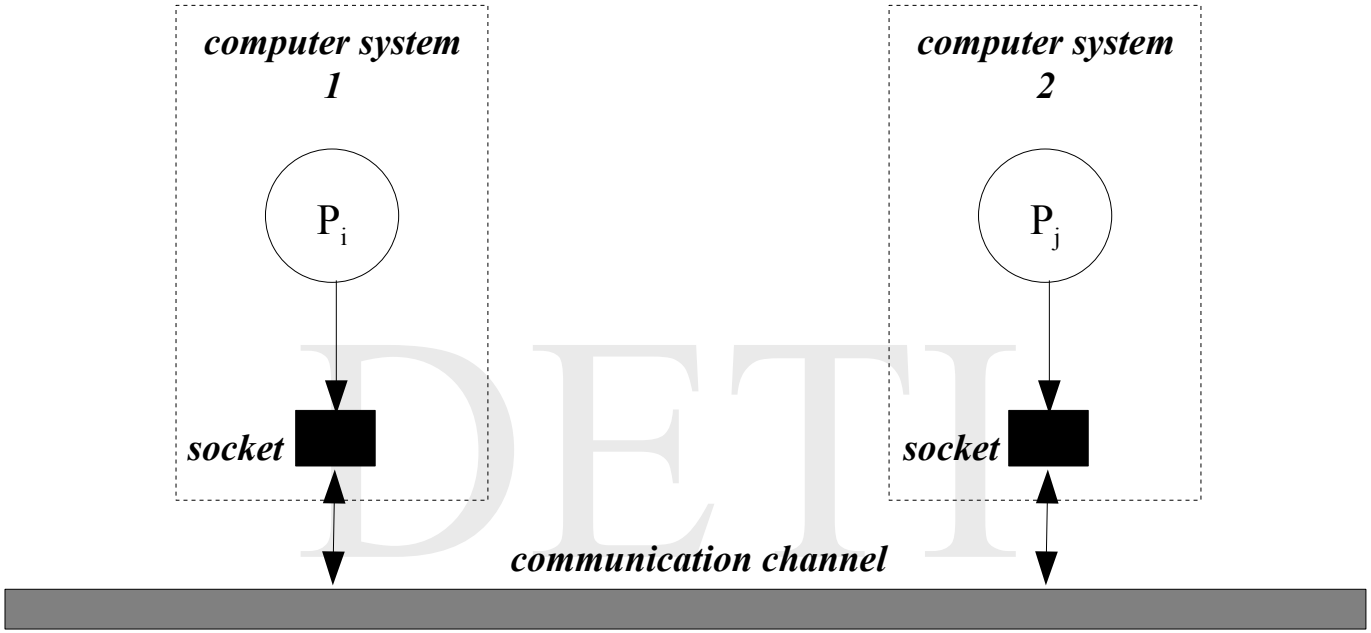
On the *sending side*, each layer, but the topmost, accepts data in a well-defined format from the layer above and applies a transformation procedure to encapsulate it in another well-defined format before passing it to the layer below. *On the receiving side*, the converse transformations are applied to data as it traverses the layers in the reverse direction.

Communication system - 4

OSI model

Application	protocols designed to meet the communication requirements of specific applications
Presentation	protocols to enable data transmission in a network specific representation which is hardware independent
Session	protocols for error detection and automatic recovery
Transport	protocols to enable message addressing to communication ports associated with processes
Network	protocols to enable the transfer of data packets between the network nodes
Data link (logical)	protocols to enable data transmission between network nodes connected by the same physical link
Physical	specification of the circuits and signals that drive a particular physical link

Programming interface – 1



Middleware presents to the application programmer a device, called *end point of communication* or *socket*, to enable message exchange among processes which do not share an addressing space.

Sockets are characterized by the *IP address* of the computer system and a *port* which defines within the computer system the end point of a specific communication channel.

Programming interface – 2

There are two main protocols for message exchange

- *TCP* – it is a *connection-oriented protocol*, meaning that a virtual communication channel must be established between the end points before any data exchange may take place
  it allows *bidirectional communication* because, once the channel has been established, a stream of data may flow from each end point
  it is *asymmetric*, since it was specifically designed for the client-server model, it assumes a different role for each end point
- *UDP* – it is a *connectionless* protocol, meaning that no virtual communication is required for data exchange to take place
  it only allows unidirectional communication because it assumes the transmission of a single message from one of the end points to the other
  it is *symmetric* because no different role is assigned to either of the end points.

TCP protocol – 1

TCP protocol requires two types of sockets

- *listening socket* – instantiated by the *server* and where it is listening for a connection request from a *client*
- *communication socket* – instantiated by the *client* when it needs a data exchange with the *server*, and by the *server* itself when it establishes a virtual communication channel with the *client* .

TCP protocol – 2

Client side

```
instantiateComSocket ();
connectToServer (serverPublicAdd);
openInputStream ();
openOutputStream ();
writeRequest ();
readReply ();
closeOutpoutStream();
closeInputStream();
closeComSocket ();
```

Server side

Thread base

```
instantiateListenSocket (serverPublicAdd);

while (true)
{ comSocket =
     listenToClientConnectionReq (serverPublicAdd);
  instantiateServiceProxyAgent (comSocket)
  startServiceProxyAgent ();
}
```

Service Proxy Agent

```
openInputStream ();
openOutputStream ();
readRequest ();
localExecution ();
writeReply ();
closeOutpoutStream();
closeInputStream();
closeComSocket ();
```

UDP protocol – 1

UDP protocol requires a single type of socket to transmit a message, called a *datagram packet*, from the source to the destination point

- *receiving socket* – instantiated by the *receiver* at a specific port for packet reception from different sources
- *sending socket* – instantiated by the *sender* for packet transmission to different destinations.

UDP protocol – 2

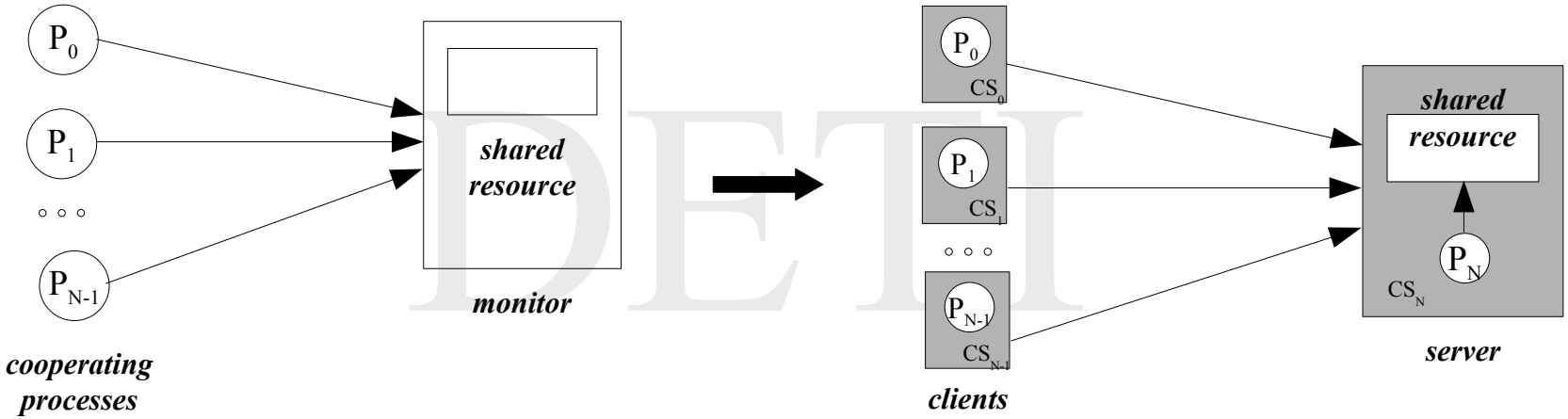
Source side

```
instantiateSendSocket ();
convMesgToByteArray
            (msg, byteArray);
instantiateDataPacket
     (byteArray, DestPublicAdd);
send (dataPacket);
```

Destination side

```
instantiateRecSocket
                (DestPublicAdd);
receive (dataPacket);
convByteArrayToMesg
                 (byteArray, msg);
```

Transformation principles – 1



Transformation principles – 2

A key feature of the transformation is that it must be carried out with minimal changes of the concurrent code, namely, the whole interaction mechanism among the different entities must be kept as it was previously set.

Since both the cooperating processes and the shared resource reside in different computer systems, there is no sharing of addressing space which entails that

- method invocation upon the shared resource must be carried out through message exchange: one message for the call and another for the call return
- besides the method parameters and the return value, messages should include those attributes of the caller process which are relevant for the execution of the method, or that are changed by its execution
- all message parameters are to be passed by value.

Message structure – 1

A message is transmitted through a communication channel and, at the lowest level, may be seen as an array of bytes. Since the client and the server processes are separate programs, it crucial for the receiver to know how to interpret this array of bytes and how to build from this data the values of the message parameters.

Thus, a message contents must include not only the values of the parameters, but also their type and how they are structured. The operation of building a message with these characteristics is called *information marshaling*, and the opposite operation of retrieving the values of the parameters from the array of bytes as *information unmarshaling*.

In Java, marshaling and unmarshaling of information is hidden from the programmer. It is only required that the message data type be defined as implementing the `Serializable` interface.

Message structure – 2

```
import java.io.Serializable;

public class Message implements Serializable
{
    private static final long serialVersionUID = <long literal>;

    /* definition of the message parameters */

    /* message instantiation */

    /* public methods for getting the values of the message parameters */
}
```


- if a message parameter is of a reference data type, it must also implement the `Serializable` interface
- this rule should be applied in a recursive way so that what remains are parameters of primitive data types

Message structure – 3

```
import java.io.Serializable;

public class Record implements Serializable
{
    private static final long serialVersionUID = 20140404L;

    public int nEmp;
    public String nome;

    public Record (int nEmp, String nome)
    {   this.nEmp = nEmp;
        this.nome = nome;
    }
}
```


serial representation of `new Record (105, "Ana Francisca")`

```
ACED0005737200065265636F726400000000013351740200024900046E456D704C00046E6F6D6574
00124C6A6176612F6C616E672F537472696E673B787000000006974000D416E61204672616E636973
6361
```

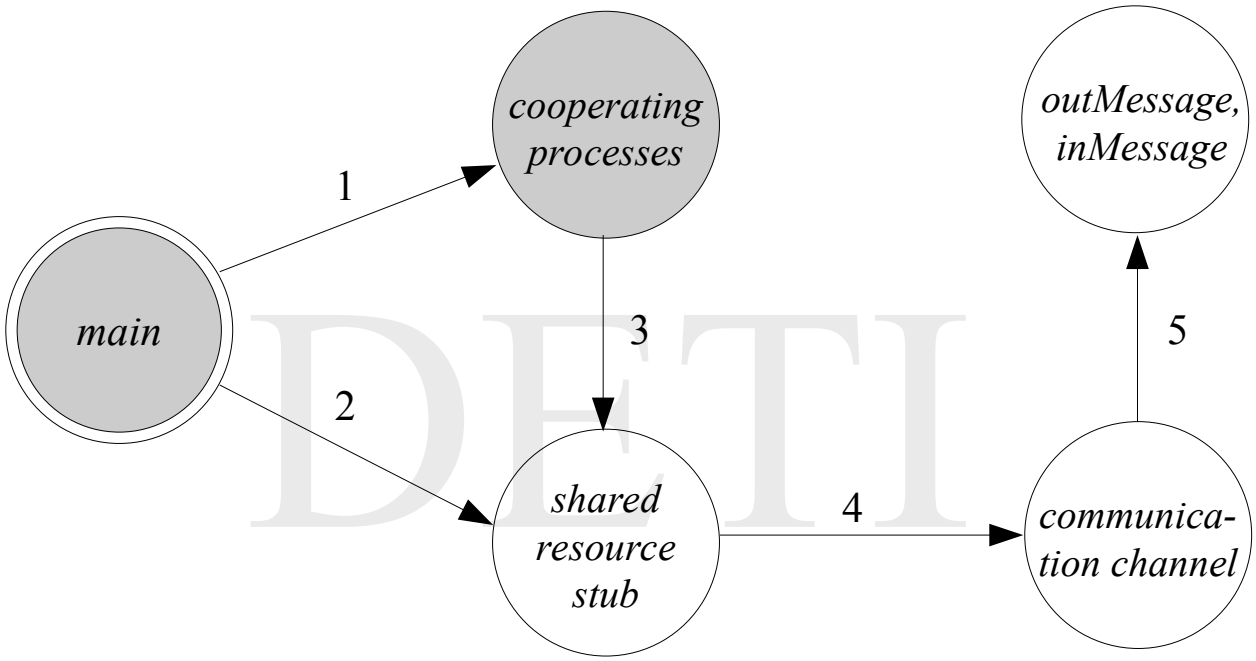
run `ShowMain` in the package `showSerialization` to see the interpretation of the array of bytes above

Client architecture – 1

In the concurrent solution, the `main` thread instantiated both the cooperating processes and the shared resource. Now, it can not any more instantiate the shared resource since it belongs to a different addressing space. However, to keep most of the code unchanged, a remote reference to the shared resource is to be instantiated. This remote reference, usually called the *stub* of the shared resource, has as instantiation parameters the internet address of the server where the shared resource is located and the number of the listening port. All other values used formerly for its instantiation must now be passed through the invocation of a new method on the stub.

It is the stub responsibility to convert all method invocations on the shared resource into an exchange of messages with the server where the shared resource is located.

Client architecture – 2



1 – instantiate, start, join

2 – instantiate, possible parameter communication for initialization, shutdown

3 – already defined methods

4 – instantiate, open, close, writeObject, readObject

5 – instantiate, get field values

Client architecture – 3

Data type which defines the main thread

It remains mostly unchanged. The modifications are the following

- the stub of the shared resource is instantiated instead of the shared resource itself
- if other values were formerly required for the instantiation of the shared resource, those values are now passed through the invocation of a new method on the stub
- if the server shutdown is required after the end of operations, the invocation of a new method on the stub is required.

Data type which defines the cooperating processes

It remains mostly unchanged. The modifications are the following

- a reference to the stub of the shared resource is passed upon instantiation instead of a reference to the shared resource itself.

Client architecture – 4

Data type which defines the stub of the shared resource

It is new and must be created. It is rather regular and, for the operations that are invoked on it, the following steps must be defined

- a communication channel to server is opened (instantiated)
- an outgoing message is instantiated based on the method identification, its parameters and the values of those attributes of the caller process which are relevant for the execution of the method
- the outgoing message (service request) is sent
- an incoming message (reply) is received and is checked for correctness
- those attributes of the caller process that were affected by the execution of the method are to be updated
- the communication channel is closed
- the method returns.

Client architecture – 5

Data type which defines the communication channel

It new and must be created. Its key feature is to encapsulate the operations carried out on sockets. The `ClientCom` data type, which is provided in the examples, can be used as it is or be modified according to specific requirements.

Data type which defines the message

It new and must be created. There may be a single data type that encompasses all the cases or multiple data types suitable for different situations.

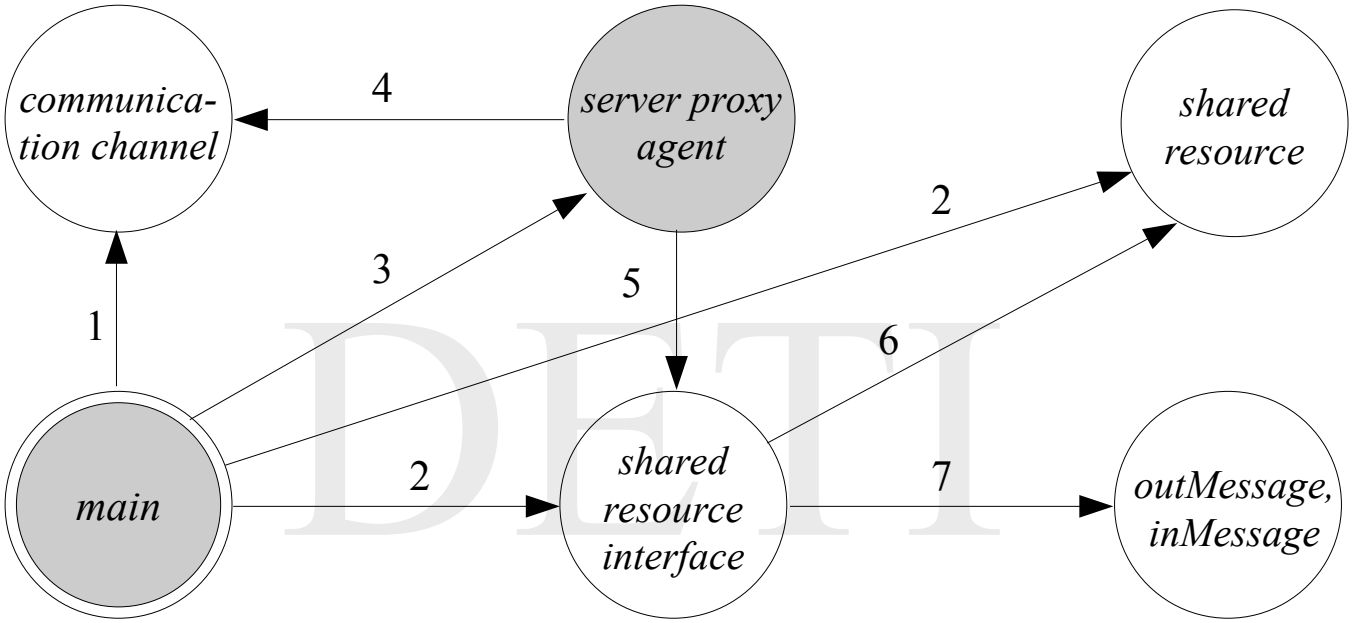
Server architecture – 1

The shared resource is a passive entity. So, in order to have it operational, the main thread, or base thread, of the server must instantiate it as well as a communication channel listening on the public address for service requests.

When a service request comes, the base thread has to instantiate and start a *service proxy agent* thread to deal with the request and returns next to the listening activity to take care of possible new incoming service requests (*server replication variant*).

The service proxy agent receives the incoming message, decodes it and sets itself as a clone of the client by incorporating the process attributes that are required, invokes the corresponding method on the shared resource, composes the outgoing message, sends it, closes the communication channel and terminates.

Server architecture – 2



1 – instantiate, start, end, accept

2 – instantiate

3 – instantiate, start

4 - readObject, writeObject, close

5 – processAndReply

6 – already defined methods

7 – instantiate, get field values

Server architecture – 3

Data type which defines the main thread

It is new and must be created. It is, however, almost invariant for all servers. The modifications are the following

- the public address for service requests depends on each server
- the shared resource and its interface which are instantiated, are specific of each server.

Data type which defines the service proxy agent thread

It is new and must be created. It is, however, almost invariant for all servers. The modifications are the following

- if one wants to have it run as a clone of the several classes of clients which access the server, it should implement the interfaces to each class related to the setting and the getting of the relevant attributes.

Server architecture – 4

Data type which defines the interface to the shared resource

It is new and must be created. Its organization, however, is invariant for all servers. The internal structure is the following

- there is only a public method, `processAndReply`, which decodes the incoming message (service request), processes it and generates the outgoing message (reply)
- the internal operation may be divided in two parts
  - incoming message validation with eventual incorporation of the client attributes
  - method invocation and outgoing message generation.

Data type which defines the shared resource

It remains mostly unchanged. The modifications are the following

- if there are specific references to the cooperating processes data types, they are to be changed to the service proxy agent data type.

Server architecture – 5

Data type which defines the communication channel

It new and must be created. Its key feature is to encapsulate the operations carried out on sockets. The `ServerCom` data type, which is provided in the examples, can be used as it is or be modified according to specific requirements.

Data type which defines the message

It is the same data type used in the client side.

Server architecture – 6

In general, in a given application, several servers are involved, some of them requesting services on others. Thus, one has a situation where some of the servers are simultaneously servers and clients.

This presents no conceptual difficulty. One has only to merge both functionalities producing a mixed architecture.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 4: *Interprocess communication*
    Sections 4.1 to 4.4
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 4: *Communication*
    Sections 4.1 to 4.3