Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

SISTEMAS Y SERVICIOS DISTRIBUIDOS

LAB MANUAL 2: PROGRAMMING OF DISTRIBUTED SYSTEMS (1)

INTERPROCESS COMMUNICATIONS AND SOCKETS

Teacher:
Esteban Egea López

INDICE

1 Goals.

- Introducing interprocess communication and the *Sockets* interface.
- Using input and output libraries Java to create data streams.
- Learning the implementation of Java Sockets.
- Programming simple clients and servers.

2 Interprocess communication.

The (OS) operating systems isolate the resources devoted to the different processes that keep running. If a process wants to share information with another one you must use a certain communication mechanism. There are many mechanisms for communication between processes. The simplest is the use of shared files. In general, the OS provides various mechanisms, as signals, pipes, shared memory areas or *sockets.* There are higher - level abstractions, generically called *message passing systems,* which make use of multiple OS facilities. As we shall see later, Java RMI can be considered a message passing system.

3 Sockets.

The most commonly used generic mechanism for interprocess communication is the *Socket Interface.* This interface allows message passing between processes on the same machine or on different machines. In essence, the socket interface is an abstraction of the TCP/IP protocol. That is to say, it is the interface that allows to use the OS primitives for TCP/UDP/IP communications. Actually, the socket interface allows the use of other protocols as well.

In other words, a socket is an abstraction through which an application can send and receive data, just as an open file allows an application to read and write data in a storage system. A socket allows an application to connect to the network and communicate with other applications connected. The information written on a socket by a process in a machine can be read by another process on a different computer and vice versa.

There are different types of sockets:

- Datagram sockets, known as sockets not connection-oriented, using the UDP protocol.
- Stream sockets, connection-oriented sockets, which use TCP or SCTP.
- Raw sockets, available at level teams usually network. They allow direct access to lower level layers.
- There are also Unix Domain Sockets (UDS), which are an efficient alternative for communication between processes on the same machine.

From a practical standpoint, the sockets are a programming library (API). Therefore, we can find different implementations. The most common are called *Berkeley Sockets* [1] [2] [3], provided by most SO.

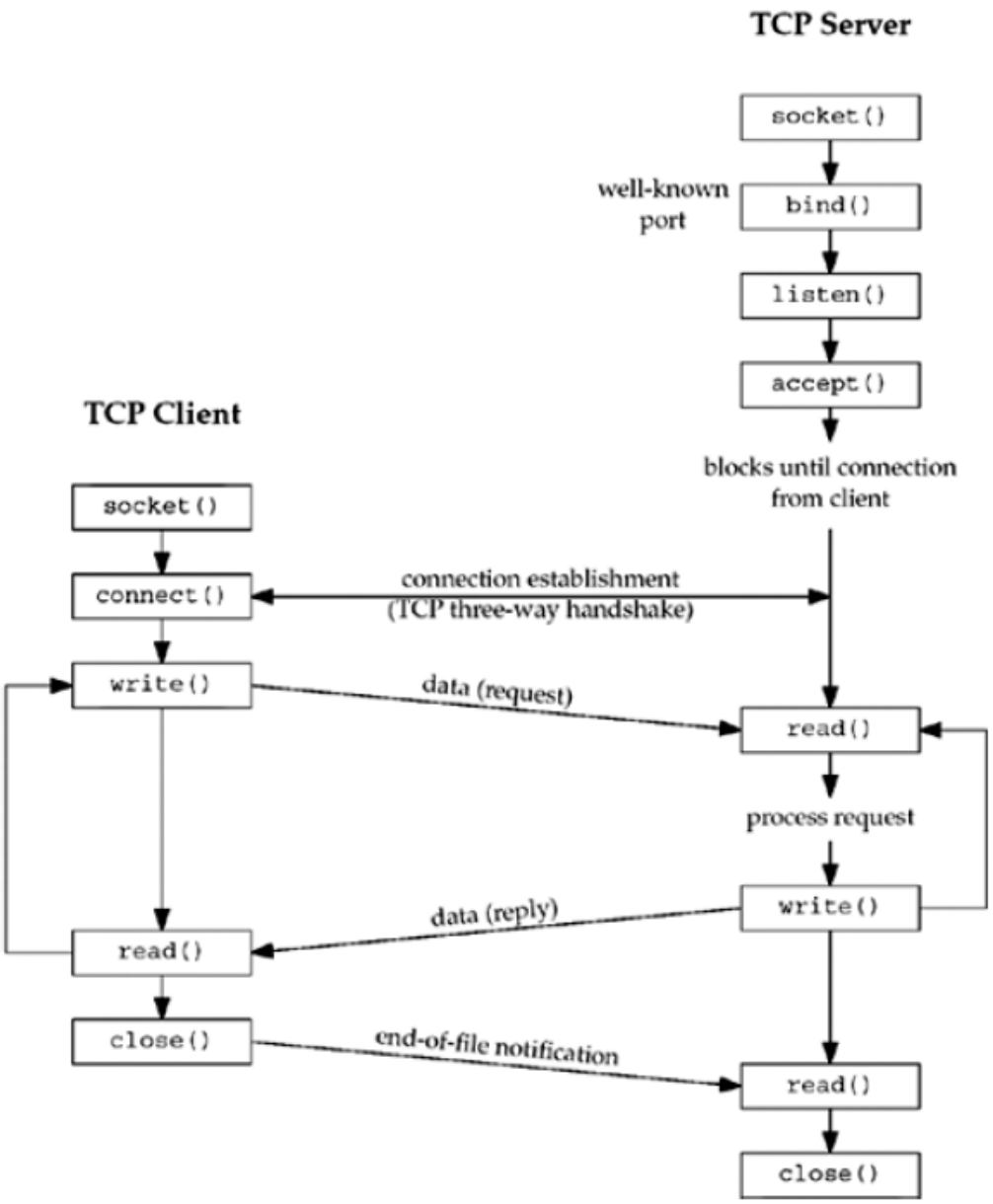


Figure.1 generic flow chart of communication with Sockets for TCP.

The most important functions of Berkeley Sockets are as the following ones:

- socket () creates a new socket of a particular type. The OS assigns resources and returns a descriptor.
- bind (). Is used (usually by the server) to bind with a port between 0 and 65535. The ports below 1024 are reserved and can only be used by the superuser.
- listen(). The server uses it to set the TCP connection to LISTEN state. Accepts an extra parameter indicating the size of the connection queue, or *backlog.* Connection requests to a specific port are queued until they are accepted. If the queue is full, the connection is rejected.
- accept (). The server invokes it after *listen ()* to accept connection requests. That is, if there is any request in the connection queue, it is accepted at this moment. When accepted, a new socket is generated. The original is used to keep listening to requests and manage the queue,

while the new contains the TCP connection itself and is used to send and receive data. By default it is a blocking call, i.e if there are no pending connections, the calling process is suspended until a new connection request arrives. The sockets can be configured in non-blocking mode, and then it returns an error.

- connect (). Used by the client. A free local port is assigned and it attempts to establish a new TCP connection.

Depending on the type of sockets in use (ie protocol), the flowchart calls will be different. The following timing diagram shows the relationship of the invocations of the various functions and the current exchange of TCP packets, in addition to the states through which it passes the protocol:

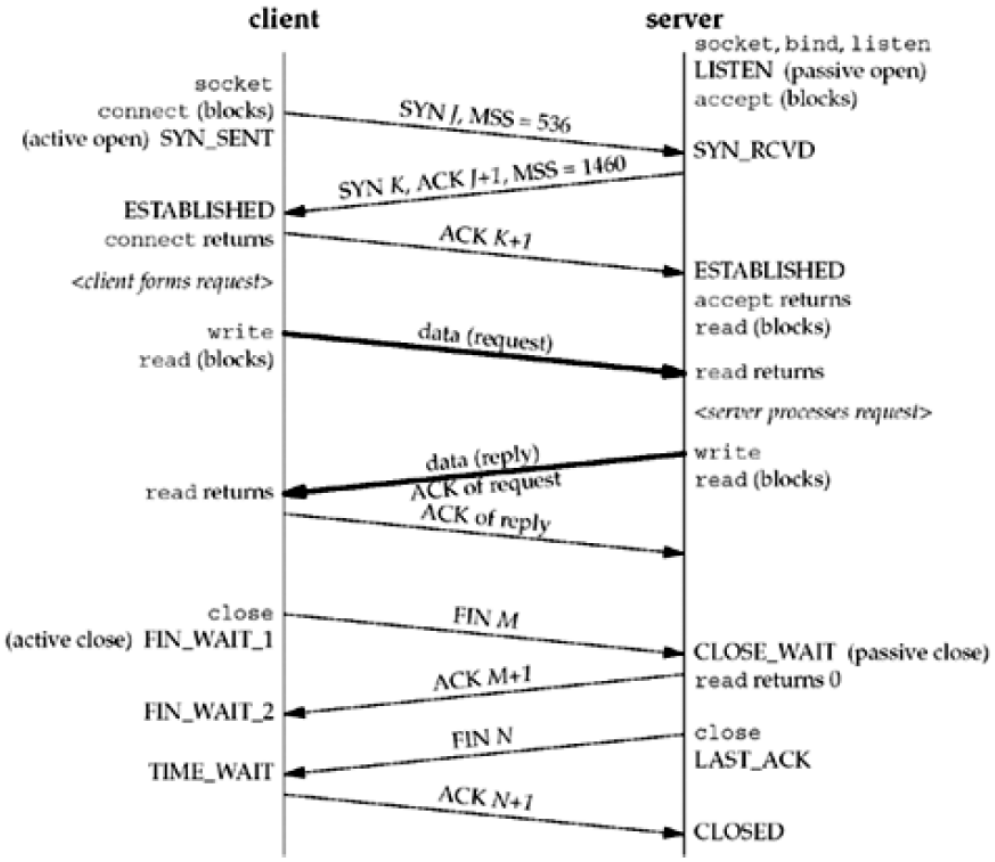


Figure 2 Packet Exchange on a TCP connection

4 Sockets in Java

It is essential that you get used to using the Java documentation during the labs.

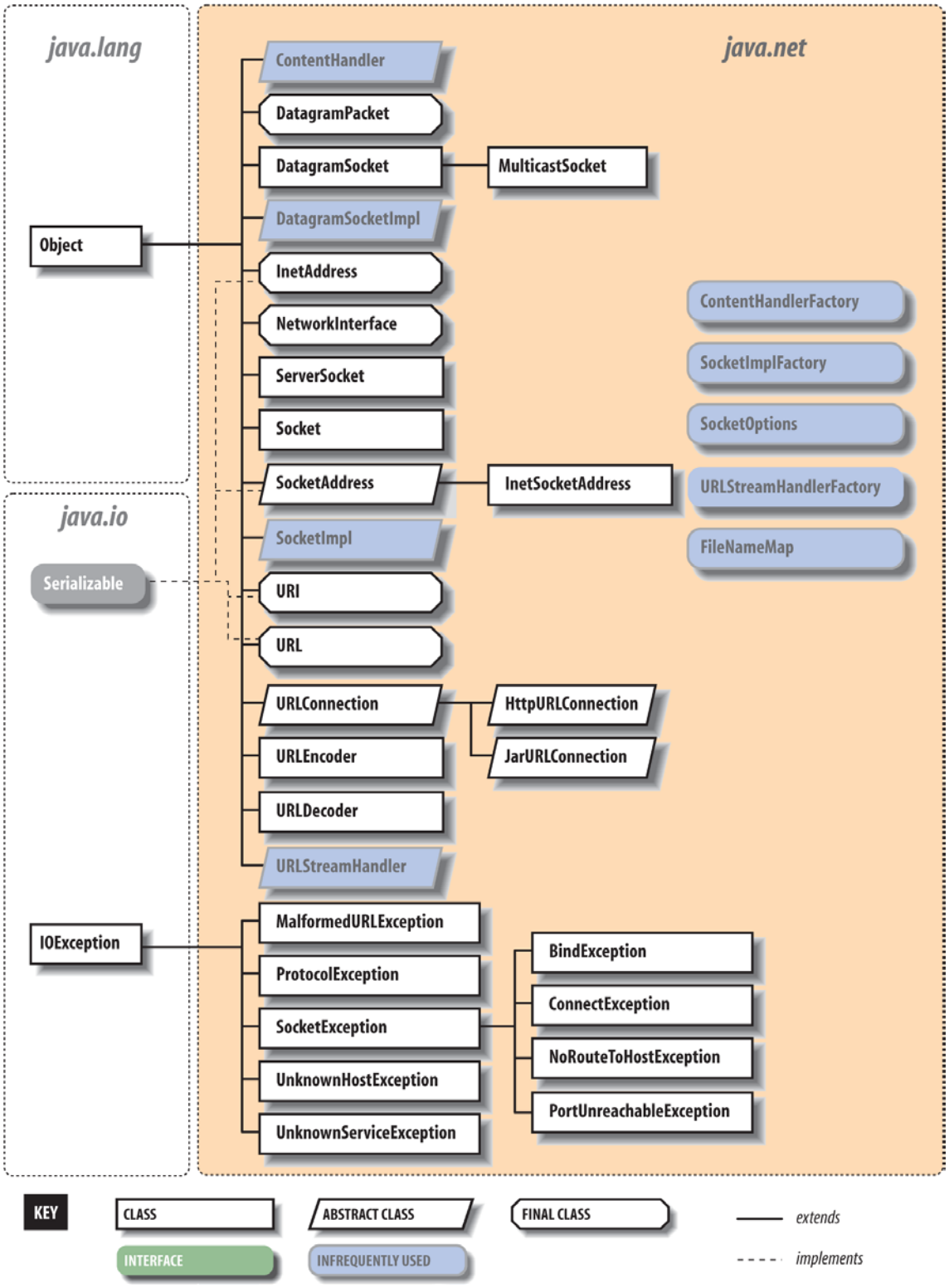
Java provides its own implementation of the *socket* interface in the java.net package. It is a simplified and object-oriented interface. The classes provide network communication support independent of the underlying system. That is, internally they use the appropriate OS calls.

So the java.net package provides, among others, the following classes:

- Socket: Implements the TCP connection on the client side.
- ServerSocket: Implements the TCP connection on the server side waiting for the connection of customers.

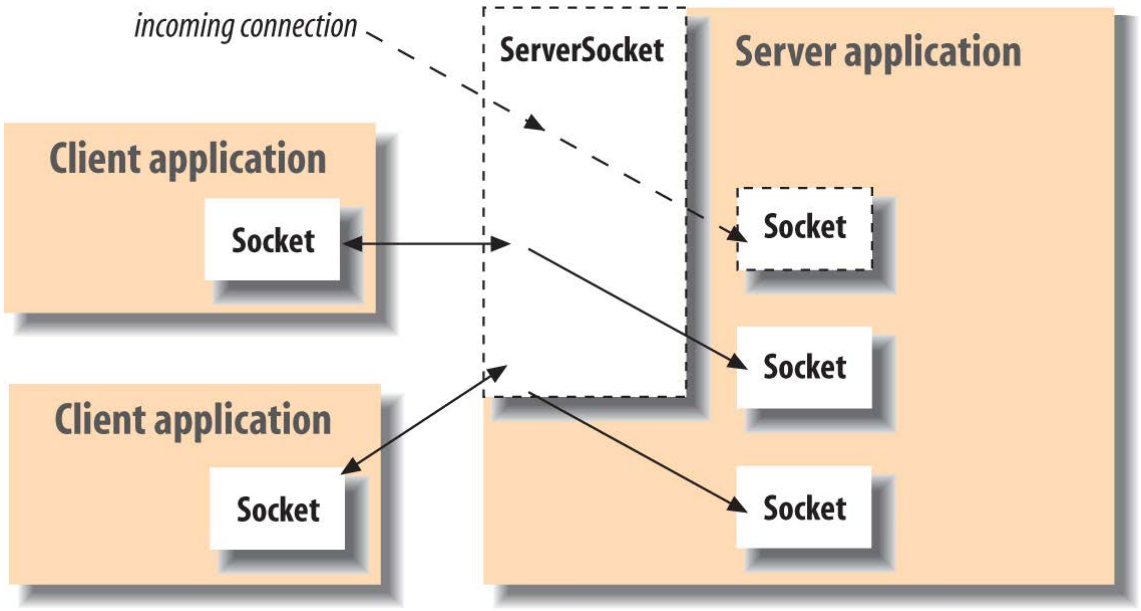
- DatagramSocket: implements both the server and the client when UDP is used
- DatagramPacket: Implement a "datagram packet", which is used for connectionless packet services, such as UDP.
- InetAddress: Responsible for implementing the IP address.

The following diagram shows the classes of the java.net package



For TCP, Java distinguishes two types of classes: one for the client and one for the server. A server program creates a specific type of socket that is used to listen to client requests (ServerSocket). If a connection is requested, the program creates a new Socket, used to exchange data with the client using input and output streams.

These relationships are shown in the following diagram:



The use of Java Sockets interface is as follows. The server will create a ServerSocket and starts listening, ready to accept connection requests. That is, incoming connections requests are queued until accepted by the server. If there is no queued connection request when *accept()* is called, the server goes to sleep until a new connection request is available. The client, in turn, will create a Socket to establish a connection. When the connection is established, the *accept* () function at the server will return an instance of Socket with the connection between client and server. On the server we have:

```
ServerSocket listener = new ServerSocket (5050);

while (true) {

Socket con= listener.accept(); //Goes to sleep if no connection is
pending

// Attends request ...

con.close ();

}
```

On the client:

```
Socket con = new Socket ("ait.upct.es", 5050);
```

5 Exercises

Use Eclipse to develop the following programs.

You will develop a client program and simple server that will be refined in the next sessions. Start with a server that prints on screen the information sent by the client. To do this, perform the following exercises:

a. **Program a server that prints on the screen the string that is sent by the client. Make sure that the sockets always are closed, using try / catch blocks and finally.**

   To do this, create two new classes: SimpleServer, with the runServer (int port) method, which will implement the server, and the StartServer class, which will have a main () method that will start the server.

   - You have to implement the common loop shown on page 8 above.
   - Print both the ServerSocket and the Socket to the screen when you have accepted a connection.
   - The rest of the code in the loop basically consists of reading / writing the socket data using the appropriate Java IO classes (Practice 1). What classes from the Java IO hierarchy will you use? Why?
   - Does the Socket class provide any method to get a Reader or Writer? How can you convert an InputStream to a Reader?
   - To ensure that sockets are always closed you have to enter a try / finally block for the Socket and another one for the ServerSocket.

b. **Make a client program that sends a string to the server and close the connection**. Use **PrintWriter to write the data.**

   To do this, create two new classes: SimpleClient, with the runClient (int port) method, which will implement the client, and the StartClient class, which will have a main () method that will start the client.

   - What address will the client connect to?
   - Once the connection to the server is established you will have to use the corresponding Java IO classes again to read / write from the Socket.

c. **Modify the server to show on screen the lines of text that are sent by the client until you find the string "exit" in them.**

   - To do this, wrap the Socket reading in an infinite loop. Check whether the string you read is null or "quit" to exit the loop.

d. **Modify the client so that the user can manually enter phrases and send them to the server until the string "exit" is entered.**

   - Use, within an infinite loop, reading from standard input.

e. **Connect with a client to your server program and, before leaving, run another instance of his client and try to connect to the same server program. What is the result? Use wireshark to investigate the state of the new connection.**

f. **Use telnet instead of your client to connect to the server program.**

   On Linux just use the telnet command. On Windows, here are several methods to enable it: https://social.technet.microsoft.com/wiki/contents/articles/38433.windows-10-enabling-telnet-client.aspx

   - According to what you see, what exactly does the telnet program do?

6 References

1. http://beej.us/guide/bgnet/output/html/multipage/index.html
2. http://en.wikipedia.org/wiki/Berkeley_sockets

7 Bibliography

1. Bruce Eckel, Thinking in Java, 4th Edition, Prentice Hall, 2006

(Previous versions are available for free in electronic format
http://www.mindviewinc.com/Books/downloads.html)

1. David Reilly, Java Network Programming and Distributed Computing, Addison-Wesley, 2002.
2. Stevens, Unix Network Programming, Addison Wesley.