Lab 1 : Java Sockets

1. Goals

In this lab you will work with a low-level mechanism for distributed communication. You will discover that Java sockets do not provide:

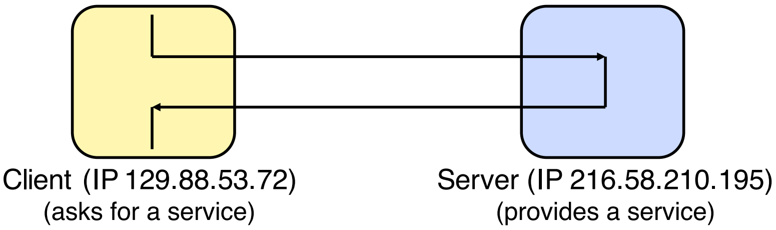
- location transparency
- naming transparency
- programming support for complex data exchange
- programming support for application-level protocols
- failure transparency

2. Documentation

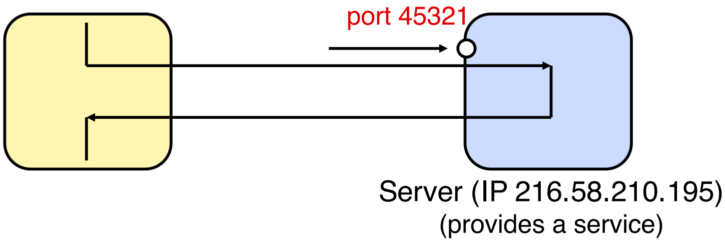
[1] https://docs.oracle.com/javase/tutorial/networking/sockets/
[2] https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html
[3] https://docs.oracle.com/javase/7/docs/api/

3. Introduction to sockets

At the network level, a client and a server communicate using the network protocols such as TCP or UDP. The client and the server are designated by the IP addresses of their machines. The IP address corresponding to a symbolic name (e.g. http://www.google.fr) can be obtained using the DNS (Domain Naming Service).

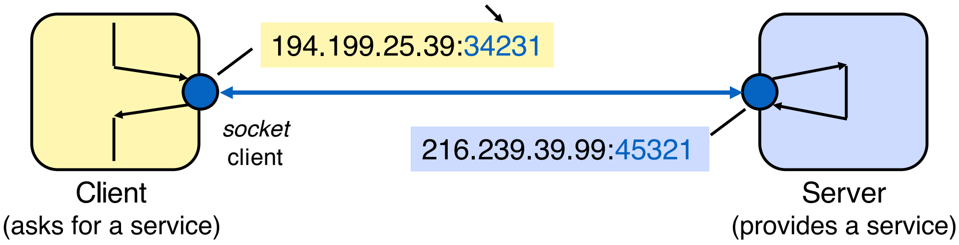


Usually, on one physical machine, acting as a server, there are multiple services running. Examples are ssh, ftp, mail, web servers, etc. To differentiate them during a distributed communication, we use ports that designate the corresponding server process. The port numbers are encoded using 16 bits and the numbers form 0 to 1023 are reserved for special services.



In many cases, for simplicity, when we talk about servers, we mean the software services that are provided by some process on a given physical machine.

Sockets provide an easy-to-use interface for communicating through TCP and UDP. They designate a communication point (at the sender or the receiver side) composed by an IP address and a port number.



If the TCP protocol is used, a connection is first established between the client and the server and all subsequent messages are exchanged using this connection. In Java, the classes to use are `ServerSocket` and `Socket`.

If the UDP protocol is used, no connection is established and the messages are exchanged one by one. The Java class to use is `DatagrammSocket`.

In connected mode (TCP mode), the sequence to establish the connection between the client and the server is the following:

1. SERVER : launch the server. The server has a server socket waiting for client connections.
2. CLIENT : launch the client. The client tries to connect to the server.
3. SERVER : the server receives the connection request and accepts it. It creates a dedicated socket to communicate with the client.
4. CLIENT : if the connection succeeded, the client can communicate with the server. The communication is done via reading/writing to/from the socket.

5. Basic example : the Echo server

In this example, a server receives a string from a client and sends it back.

The server code is given in `EchoServer.java.`

- The server creates the listening server socket using the port number given in the command line and found in the `main args` (lines 12-13).
- It waits for a client connection and when there is a request, creates the socket to communicate with the client (line 14).
- it initializes the structures to read from and write to the client socket
- while there is no problem (like the client disconnecting for example, the server reads what the client has sent and sends it back (lines 21-22)

If you do not know the syntax of the try statement with ressources, it just means that the ressources that have been allocated (e.g the PrinterWriter out) are guaranteed to be freed (closed) at the end of the execution (see [2]).

```
1. public class EchoServer {
2.     public static void main(String[] args) throws IOException {
3.
4.         if (args.length != 1) {
5.             System.err.println("Usage: java EchoServer <port number>");
6.             System.exit(1);
7.         }
8.
9.         int portNumber = Integer.parseInt(args[0]);
10.
11.        try (
12.            ServerSocket serverSocket =
13.                new ServerSocket(Integer.parseInt(args[0]));
14.            Socket clientSocket = serverSocket.accept();
15.            PrintWriter out =
16.                new PrintWriter(clientSocket.getOutputStream(), true);
17.            BufferedReader in = new BufferedReader(
18.                new InputStreamReader(clientSocket.getInputStream()));
19.        ) {
20.            String inputLine;
21.            while ((inputLine = in.readLine()) != null) {
22.                out.println(inputLine);
23.            }
24.        } catch (IOException e) {
25.            System.out.println("Exception caught when trying to listen on port "
26.                + portNumber + " or listening for a connection");
27.            System.out.println(e.getMessage());
28.        }
29.    }
30.}
```

The client code is given in `EchoClient.java.`

- The client tries to connect to the server using the IP address and the port number given in the command line (line 14)
- Then repeatedly read something from the standard input, sends it to the server, receives the answer and prints it to the screen (lines 25-27)

```
1. public class EchoClient {
2.     public static void main(String[] args) throws IOException {
3.
4.         if (args.length != 2) {
5.             System.err.println(
6.                 "Usage: java EchoClient <host name> <port number>");
7.             System.exit(1);
8.         }
9.
10.        String hostName = args[0];
11.        int portNumber = Integer.parseInt(args[1]);
12.
13.        try (
14.            Socket echoSocket = new Socket(hostName, portNumber);
15.            PrintWriter out =
16.                new PrintWriter(echoSocket.getOutputStream(), true);
17.            BufferedReader in =
18.                new BufferedReader(
19.                    new InputStreamReader(echoSocket.getInputStream()));
20.            BufferedReader stdIn =
21.                new BufferedReader(
22.                    new InputStreamReader(System.in))
```

```
        ) {
            String userInput;
            while ((userInput = stdIn.readLine()) != null) {
                out.println(userInput);
                System.out.println("echo: " + in.readLine());
            }
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " +
                hostName);
            System.exit(1);
        }
    }
}
```

Questions

a) Run the program.

b) What happens if the server stops?

c) What happens if the client stops?

6. Exercices

6.1 Calculator server

The goal of this exercice is to show you that sockets do not facilitate the programming of distributed communication protocols.

Implement a server which provides a calculator service with the following interface

```
interface Calculator_itf {
	public int plus(int, int);
	public int minus (int, int);
	public int divide(int, int);
	public int multiply(int, int);
}
```

You are required to:

- implement a client who prints a textual menu to choose from the available operations
- implement the server
- define a client-server protocol based on distinct messages for the arithmetic operation and the two arguments. I.e you are required to NOT implement a SOAP-like protocol where a String is sent to the server which parses it.
- test the application

6.2 Phone registry

*The goal of this exercice is to show you that sockets do not handle high-level objects. You need to take care that your objects are **Serializable** to send them over the network.*

Implement a server which provides a phone registry service with the following interface:

```
interface Registry_itf {
	public void add(Person p);
	public String getPhone(String name);
	public Iterable<Person> getAll();
	public Person search(String name);
}
```

You are required to:

- define the Person class
- implement a client who prints a textual menu to choose from the available operations
- implement the server
- define a client-server protocol for the interactions
- test the application

6.3 Point-to-point communication

The goal of this exercice is to show you that the communication features of sockets are low-level and are not transparent in terms of naming and location.

This exercice is possibly too long to finish during the lab, if you want to finish it, you will need to work during your free time.

Implement a client-server application in which:
- there is one central server
- clients connect to the server
- clients may ask the server who the other clients are
- a client A may send a message to another client B

Hints

- For a client to be able to send a message to another client, (the client A should be able to refer to the client B), you will need client identifiers. What kind?

- The client A should be able to send a message to the client B, how?

  - *Is it possible to use the server to do so?*
    I.e, is it possible to send the message to the server and « ask it » to forward to the client B?
    How the client B could receive the message?
  - *Maybe modify the client B for it to play also a server role?*
    Thus the client A could connect to it directly and communicate.
    Supposing that client B provides a server capacity (a server socket),
    how the client A could get the communication point?

6.3 Broadcast

Implement a client-server application in which:
- there is one central server
- clients connect to the server
- clients may ask the server who the other clients are
- a client A may send a message to all other clients

7. Complements : managing multiple clients

In the previous example, the server is capable of managing only one client. To manage multiple clients, the server should be multi-threaded and have one thread per client.

The previous program may be changed in the following way (the client does not change).

EchoServer.java

```
public class EchoServer {
    public static void main(String[] args) throws IOException {

        if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try (ServerSocket serverSocket = new ServerSocket(Integer.parseInt(args[0]));) {
             while (true) {
                    Socket clientSocket = serverSocket.accept();
                    EchoThread et = new EchoThread(clientSocket);
                    et.start();
             }
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

EchoThread.java

```
public class EchoThread extends Thread {
       private Socket clientSocket;

       public EchoThread(Socket s) {
             clientSocket = s;
       }

       public void run() {
             try (PrintWriter out =
                    new PrintWriter(clientSocket.getOutputStream(), true);
                  BufferedReader in = new BufferedReader(
                    new InputStreamReader(clientSocket.getInputStream()));) {

                    String inputLine;
                    while ((inputLine = in.readLine()) != null) {
                           out.println(inputLine);
                    }
             } catch (Exception e) {
                    System.out.println(
                           "Exception caught when trying to communicate with client ");
                    System.out.println(e.getMessage());            }
       }
}
```

Credits

This lab reuses materials from

- the official Java site
- lectures prepared by Renaud Lachaize