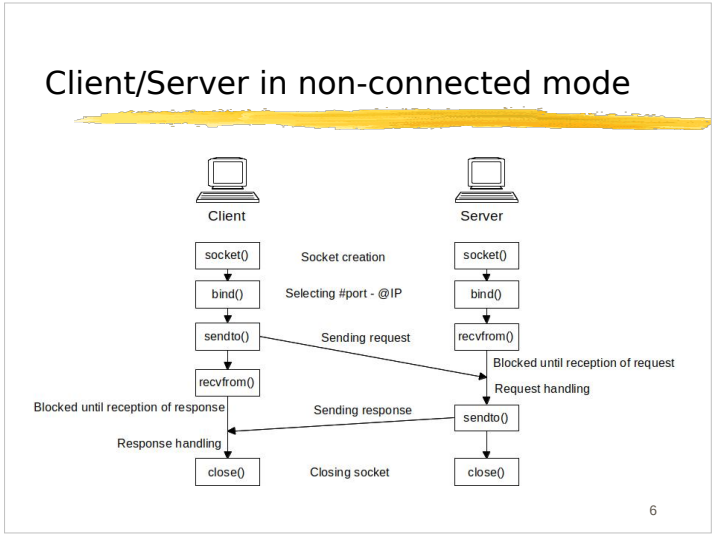


We describe the schema of a request/response interaction between a client and a server. Here we consider its implementation with non connected sockets (UDP).

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

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

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

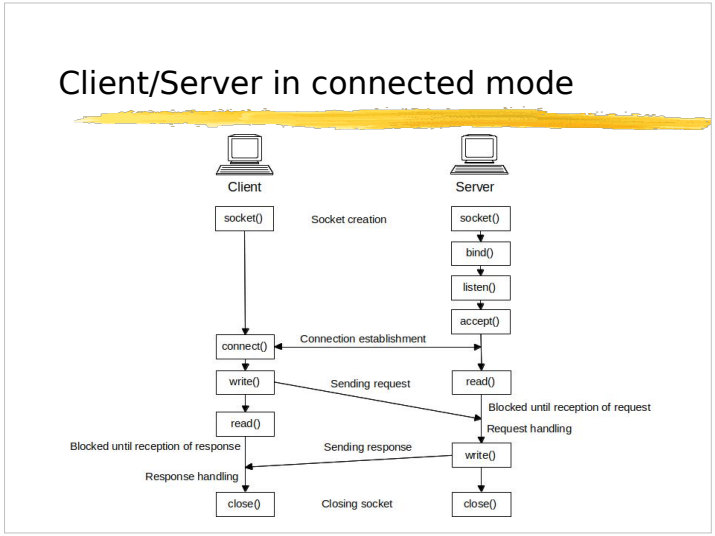
Generally, on the client side, the port given to bind() is 0, which means that bind() has to allocate a free port. This port is only used to receive responses.

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

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

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

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



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

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

- on the server side

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

- on the client side

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

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

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

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

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

connect() function

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

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

sd = socket(AF_INET, SOCK_STREAM, 0);
server.sin_family = AF_INET;
server.sin_port = htons(80);
remote_host = gethostbyname("www.enseeiht.fr"); // DNS loookup
bcopy(remote_host->h_addr, (char *)&server.sin_addr,
       remote_host->hlength); // copy the address
connect(sd, (struct sockaddr *)&server, sizeof(server));
```

12

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

- sock_desc is the fd of the socket

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

- lg_@ is the size of the previous structure as it may differ depending on the OS

A concurrent server

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

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

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

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

19

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

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

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

A full example: TCP + serialization + threads

Passing an object (by value) with serialization

The object to be passed:

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

31

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

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

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

A full example: TCP + serialization + threads

The client

```
public class Client {
   public static void main (String[] str) {
      try {
         Socket csock = new Socket("localhost",9999);
         ObjectOutputStream oos = new ObjectOutputStream (
                                        csock.getOutputStream());
         oos.writeObject(new Person("Dan","Hagi",55));
         csock.close();
      } catch (Exception e) {
                System.out.println("An error has occurred ...");
      }
   }
}
```

32

Here is the client side of the TCP example.

The main() method :

- creates a Socket which connects to a server located at localhost/9999

- from the OutputStream of the socket, it creates an ObjectOutputStream, which allows writing objects to an OutputStream. This ObjectOutputStream (oos) serializes objects and sends the data on the connection.

- writes a Person instance on oos. The instance is then serialized.

- finally closes the socket

A full example: TCP + serialization + threads

The server

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

33

Here is the server side of the TCP example.

The main() method :

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

A full example: TCP + serialization + threads

The slave

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

34

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

Here, the Slave class :

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