Sockets

Daniel Hagimont

IRIT/ENSEEIHT
2 rue Charles Camichel - BP 7122
31071 TOULOUSE CEDEX 7

Daniel.Hagimont@enseeiht.fr
http://hagimont.perso.enseeiht.fr

The first part of this lecture is devoted to sockets.

What are sockets

- Interface for programming network communication
- Allow building client/server applications
  - Applications where a client program can make invocations to server programs with messages (requests) rather than shared data (memory or files)
  - Example: a web browser and a web server
- Not only client/server applications
  - Example: a streaming applications (VOD)

Sockets are a programming interface (API) for implementing message exchanges between processes which may run on different machines.

Such message exchanges are often used to implement distributed applications following the client-server model.

In this model, a server is a program running one machine, which provides a service to some client programs running on other machines. A client may invoke this sevice by sending a message (called request) to the server program. Upon reception of a request, the server executes the treatments which correspond to the service, then it sends a message (called response) back to the client. The client is suspended after the emission of the request until reception of the response.

Notice that the request/response may include parameters/results.

A very popular exmple is the communication between a web browser (client) and a web server (server).

However, message exchanges can be used to implement other types of application, e.g. streaming applications like Video On Demand.

Two modes
connected/not connected

- Connected mode (TCP)
  - Communication problems are handled automatically
  - Simple primitives for emission and reception
  - Costly connection management procedure
  - Stream of bytes: no message limits
- Not connected mode (UDP)
  - Light weight: less resource consumption
  - More efficient
  - Allow broadcast/multicast
  - All communication problems (packet loss) have to be handled by the application
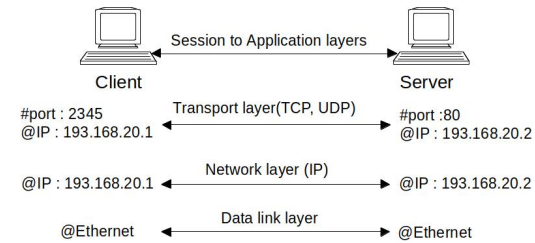
3

Communication between processes can be performed following 2 modes :

- the connected mode corresponds to the use of the TCP communication protocol. We can create a connection between the client process and the server process. The connection is bi-directional (both the client and the server can send data on the connection). The communication mode is a stream of byte, i.e. there's no message limit. Communication problems (reemission of lost packets, blocking in case of buffer saturation) are automatically handled by TCP. The establishment of the connection is costly.

- the non connected mode corresponds to the use of the UDP communication protocol. There's no connection establishment anymore, nor handling of communication problems. It reduces resource consumption. A message of any size can be sent (it is split into IP packets, and reassembled on reception). There's no guarantee regarding message reception (it has to be handled by the application). Notice that UDP allows sending messages in multicast or broadcast (in general on a local network).

Sockets

- Network access interface
- Developed in Unix BSD
- @IP, #port, protocol (TCP, UDP, ...)

Session to Application layers

Client — Server

#port : 2345 — Transport layer(TCP, UDP) — #port :80
@IP : 193.168.20.1 — @IP : 193.168.20.2

@IP : 193.168.20.1 — Network layer (IP) — @IP : 193.168.20.2

@Ethernet — Data link layer — @Ethernet

4

Sockets were initially developed in Unix BSD (Berkeley Software Distribution). They provide access to the network.

At the bottom layer (data link), machines (or rather network cards) are identified by a MAC address (e.g. an Ethernet address).

At the middle level (network), machines are identified by an IP address. ARP is the protocol which allows translating an IP address into a MAC address on a local network.

At the top level (transport), a process on one machine is identified by a couple @IP / #port, e.g. a web server is accessible on port 80 (default port) on a machine.

The socket API

- Socket creation: socket(family, type, protocol)
- Opening the dialog:
  - Client: bind(..), connect(...)
  - Server: bind(..), listen(...), accept(...)
- Data transfer:
  - Connected mode: read(...), write(...), send(...), recv(...)
  - Non-connected mode: sendto(...), recvfrom(...), sendmsg(...), recvmsg(...)
- Closing the dialog:
  - close(...), shutdown(...)

5

The socket API includes a set of functions in a programming language (initially C) for managing communication between processes.
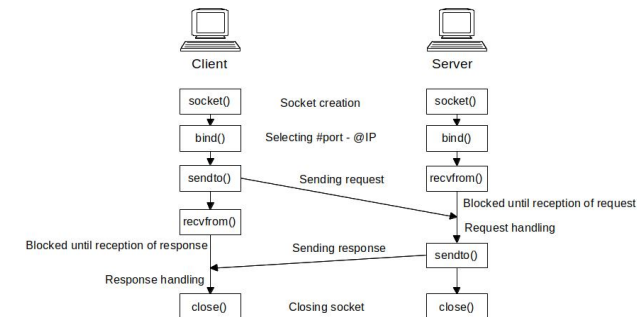
A socket is a file descriptor, similar to the file descriptors used to access files, except that writing or reading on such a descriptor sends or receives data to/from a remote process.

The socket API includes function for :

- creating a socket

- opening the dialog, i.e. initializing the socket (in connected or non-connected mode)

- transfering data (in connected or non-connected mode)

- closing the dialog

Client/Server in non-connected mode

Client		Server
socket()	Socket creation	socket()
bind()	Selecting #port - @IP	bind()
sendto()	Sending request	recvfrom()
recvfrom()		Blocked until reception of request
		Request handling
Blocked until reception of response	Sending response	sendto()
Response handling		
close()	Closing socket	close()

6

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.

Client/Server in connected mode
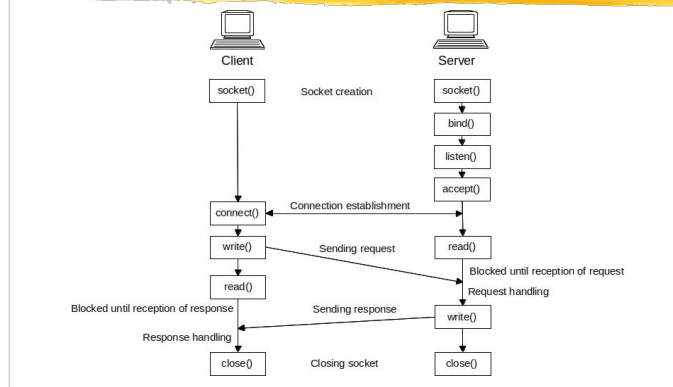
Here we consider a request/response interaction with connected sockets (TCP).

- 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).

socket() function

- int socket(int family, int type, int protocol)
- family
  - AF_INET: for Internet communications
  - AF_UNIX: for local communications
- type or mode
  - SOCK_STREAM: connected mode (TCP)
  - SOCK_DGRAM: non-connected mode (UDP)
  - SOCK_RAW: direct access to low layers (IP)
- protocol :
  - Protocol to use (different implementations can be installed)
  - 0 by default (standard)

8

We review the socket API in C.

socket() is the function which allows creating a socket.

AF_UNIX is used for local (to a machine) communications, while AF_INET is used for remote communications.

The type should be SOCK_STREAM for connected communication (TCP) and SOCK_DGRAM for non connected communication (UDP). Sockets can also be used in RAW mode (direct access to the IP level).

The protocol to be used should be 0 for default protocols (TCP, UDP), but could be different if other protocols are installed.

Notice that socket() returns an integer which is a file descriptor.

After call to socket()

Client

Server

sock_client   **FD table**

sock_server
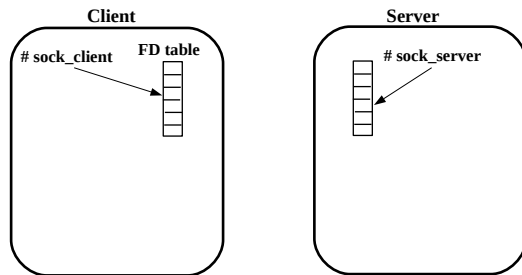
This is a representation of the states of the client and server processes after a call to socket() on both sides.

On both sides, an entry in the file descriptor table was allocated for the socket.

bind() function

- **int bind(int sock_desc, struct sockaddr *my_@, int lg_@)**
- **sock_desc: socket descriptor returned by socket()**
- **my_@: IP address and # port (local) that should be used**
- **Example (client or server):**

```
int sd;
struct sockaddr_in my_address; // @IP, #port, mode

sd = socket(AF_INET, SOCK_STREAM, 0);
my_address.sin_family = AF_INET;
my_address.sin_port = 0; // let system choose a port
my_address.sin_addr.s_addr = INADDR_ANY;
                          // any network interface

bind(sd, (struct sockaddr *)&my_address, sizeof(my_address));
```

The bind() function is invoked on both sides. It creates the association between a socket and a local port.

- sock_desc is the fd of the socket

- my_@ is a structure which describes initializations of the socket

    - sin_port = 0 means that bind() should allocate a free port

    - s_addr = INADDR_ANY means bind() can use any network interface

                (in case there are several network interfaces (cards))

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

After call to bind()

Client

sock_client   FD table

Server

sock_server
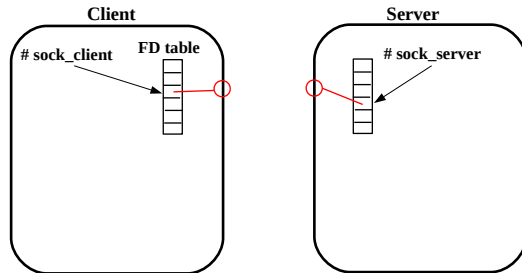
We can already exchange messages in non-connected mode

This is a representation of the states of the client and server processes after a call to bind() on both sides.

On both sides, an socket in the file descriptor table is bound to a local port.

---

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));
```

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

After call to connect()

Client

 # sock_client　　**FD table**

connection request

Server

 # sock_server
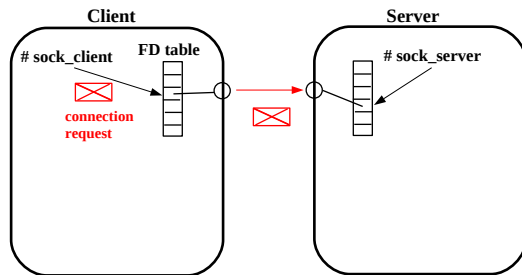
This is a representation of the states of the client and server processes after a call to connect() on the client side.

A connection request has been sent from the client to the server.

---

listen() function

- **int listen(int sock_desc, int nbr)**
- **sock_desc: socket descriptor returned by socket()**
- **nbr: maximum number of pending connections**
- **Example of server:**

```
int sd;
struct sockaddr_in server; // @IP, #port, mode

sd = socket(AF_INET, SOCK_STREAM, 0);
server.sin_family = AF_INET;
server.sin_port = 0; // let system choose a port
server.sin_addr.s_addr = INADDR_ANY;
                        // any network interface
bind(sd, (struct sockaddr *)&server, sizeof(server));
listen(sd, 5);
```
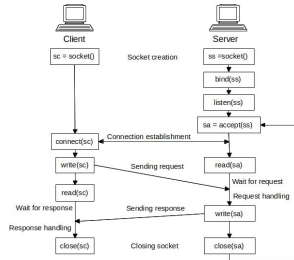
The listen() function is invoked on the server side to say that the socket will be used to receive connection requests and how many connection requests can be pending.

- sock_desc is the fd of the socket

- nbr is the number of tolerated pending connection requests (in a waiting queue). If the waiting queue is full, the connection from the client is rejected.

accept() function

- **int accept(int sock_desc, struct sockaddr *client, int lg_@)**
- **sock_desc: socket descriptor receiving connection requests**
- **client: identity of the client which requested the connection**
- **accept returns the socket descriptor associated with the accepted connection**

The accept() function is invoked on the server side. It blocks waiting for incoming connection requests. When a connection request is received, the blocked process is resumed and the function returns a new socket : the socket used to communicate with the client through the connection.

- sock_desc is the fd of the socket used to receive connection requests

- client is a structure which is updated with the identity (@IP, port) of the client who requested the connection.

After call to accept()
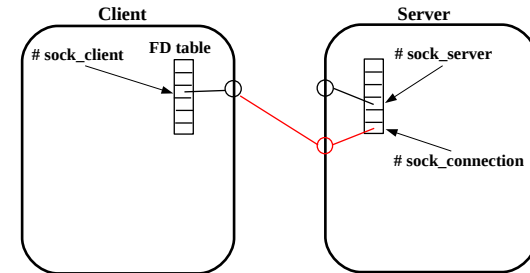
This is a representation of the states of the client and server processes after a connection has been accepted by the server.

In the server, a new socket (#sock_connection) was allocated and allows the server to communicate with the client through the connection.

Message emission/reception functions

- `int write(int sock_desc, char *buff, int lg_buff);`
- `int read(int sock_desc, char *buff, int lg_buff);`
- `int send(int sock_desc, char *buff, int lg_buff, int flag);`
- `int recv(int sock_desc, char *buff, int lg_buff, int flag);`
- `int sendto(int sock_desc, char *buff, int lg_buff, int flag, struct sockaddr *to, int lg_to);`
- `int recvfrom(int sock_desc, char *buff, int lg_buff, int flag, struct sockaddr *from, int lg_from);`

- `flag : options to control transmission parameters (consult man)`

17

Communication

18

Many functions are available for sending and receiving messages (the list here is not exhaustive).

The first four only take a socket and buffer as parameters, so they are used for the connection mode.

The last two take a sockaddr structure, allowing to specify the address (IP and port) we are sending to or to know the address of the sender we are receiving from. So they are used for the non connected mode.

Many functions have flags for controlling their behavior.
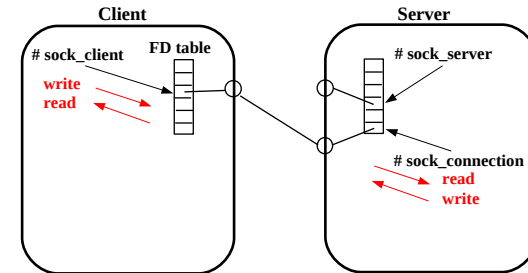
This figure illustrates communication on a TCP connection.

Both the client and the server can use read/write functions on the sockets associated with the connection. The connection is bi-directional.

A concurrent server

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

```c
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
}
```

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.

Programming Socket in Java

- package **java.net**
  - ➢ **InetAddress**
  - ➢ **Socket**
  - ➢ **ServerSocket**
  - ➢ **DatagramSocket / DatagramPacket**

We now study the socket API in the Java environment.

Sockets in Java are provided by the java.net package.

The main classes are :

- InetAddress

- Socket and ServerSocket for TCP

- DatagramSocket and DatagramPacket for UDP

Using **InetAddress** (1)

```java
import java.net.*;
public class Enseeiht1 {

  public static void main (String[] args) {
    try {
      InetAddress address =
        InetAddress.getByName("www.enseeiht.fr");
      System.out.println(address);
    } catch (UnknownHostException e) {
      System.out.println("cannot find www.enseeiht.fr");
    }
  }
}
```

21

InetAddress allows invoking the DNS, translating with getByName() a machine name into an IP address. It returns an InetAddress instance which includes the IP address.

Using **InetAddress** (2)

```java
import java.net.*;
public class Enseeiht2 {

  public static void main (String[] args) {
    try {
      InetAddress a = InetAddress.getLocalHost();
      System.out.println(a.getHostName() + " / " +
                         a.getHostAddress());
    } catch (UnknownHostException e) {
      System.out.println("No access to my address");
    }
  }
}
```

22

InetAddress also allows to obtain the InetAddress of the local host. The returned InetAddress instance includes the machine name and its IP address.

Client socket and TCP connexion

```
try {
    Socket s = new Socket("www.enseeiht.fr",80);
    …
} catch (UnknownHostException u) {
    System.out.println("Unknown host");
} catch (IOException e) {
    System.out.println("IO exception");
}
```

Reading/writing on a TCP connection

```
try {
    Socket s = new Socket ("www.enseeiht.fr",80);
    InputStream is = s.getInputStream();
    …
    OutputStream os = s.getOutputStream();
    …
} catch (Exception e) {
    System.err.println(e);
}
```

With TCP, a client can create a TCP connection with a target server (here www.enseeiht.fr), by creating an instance of the Socket class.

This operation corresponds to the calls in C of :

- socket()

- connect()

From this socket instance which is connected with the server, we can obtain 2 objects :

- an InputStream object which allows to read bytes

- an OutputStream object which allows to write bytes

With these objects, the client can send or receive data.

Server socket TCP connection

```java
try {
    ServerSocket server = new ServerSocket(port);
    Socket s = server.accept();
    OutputStream os = s.getOutputStream();
    InputStream is = s.getIntputStream();
    …
} catch (IOException e) {
    System.err.println(e);
}
```

Few words about classes for managing streams

- Suffix: type of stream
  - Stream of bytes (InputStream/OutputStream)
  - Stream of characters (Reader/Writer)
- Prefix: source or destination
  - ByteArray, File, Object …
  - Buffered, LineNumber, …

- https://www.developer.com/java/data/understanding-byte-streams-and-character-streams-in-java.html

On the server side, the server can create a ServerSocket instance, giving a local port number as parameter. A ServerSocket instance is a socket for receiving connections. Therefore, this instanciation corresponds to the calls in C of :

- socket()

- bind()

- listen()

Then, the call of accept() on this instance blocks waiting for incoming connections. The process is resumed on connection reception, and accept() returns a Socket instance, which is the communication socket of the connection with the client. Like for the client side, we can obtain from this socket InputStream and OutputStream objects which provide communication methods.

InputStream and OutputStream are basic communication classes. They only allow to read and write bytes. They can be combined with many more elaborated classes.

The names of theses classes are composed of a prefix and a suffix.

The suffix indicates the type of the stream

- suffix = InputStream or OutputStream for streams of bytes

- suffix = Reader or Writer for a streams of characters (unicode representation)

The prefix indicates the source or destination of the stream

- examples are File or Object

For instance :

- a FileInputStream allows reading bytes from a file

- a FileWriter allows writing characters to a file

Few words about classes for managing streams

	Streams for reading	Streams for writing
Character streams	BufferedReader CharArrayReader FileReader InputStreamReader LineNumberReader PipedReader PushbackReader StringReader	BufferedWriter CharArrayWriter FileWriter OutputStreamWriter PipedWriter StringWriter
Byte streams	BufferedInputStream ByteArrayInputStream DataInputStream FileInputStream ObjectInputStream PipedInputStream PushbackInputStream SequenceInputStream	BufferedOutputStream ByteArrayOutputStream DataOuputStream FileOutputStream ObjetOutputStream PipedOutputStream PrintStream

Here is a table of the different classes.

Few words about classes for managing streams

```
BufferedReader br = new BufferedReader(
    new InputStreamReader(socket.getInputStream()));
String s = br.readLine();
```

- InputStreamReader: converts a byte stream into a character stream
- BufferedReader: implements buffering

```
PrintWriter pred = new PrintWriter(
    new BufferedWriter(
        new OutputStreamWriter(
            socket.getOutputStream())));
```

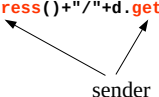- PrintWriter: formatted printing

These classes can be combined (piped or chained) to obtain the desirable behavior.

In the first example, we obtain an InputStream from a socket. From this InputStream, we create a InputStreamReader which allows reading characters (Reader) from an InputStream (so it converts a stream of byte into a stream of characters). From this object, we create a BufferedReader, which provides buffering features like reading lines of characters.

In the second example, we obtain an OutputStream from a socket. From this OutputStream, we create a OutputStreamWriter which allows writing characters (Writer) to an OutputStream (so it converts a stream of characters into a stream of bytes). From this object, we create a BufferedWriter, which provides buffering features, and then a PrintWriter which provides formated printing (like println()).

Reading on a UDP socket

```java
try {
    int p = 9999;
    byte[] t = new byte[10];
    DatagramSocket s = new DatagramSocket(p);
    DatagramPacket d = new DatagramPacket(t,t.length);
    s.receive(d);
    String str = new String(d.getData(), 0, d.getLength());
    System.out.println(d.getAddress()+"/"+d.getPort()+"/"+str);
    …
}
catch (Exception e) {
    System.err.println(e);
}
```

sender

Writing on a UDP socket

```java
try {
    int p = 8888; // for receiving a response
    byte[] t = new byte[10];
    FileInputStream f = new FileInputStream("data.txt");
    int r = f.read(t);
    DatagramSocket s = new DatagramSocket(p);
    DatagramPacket d = new DatagramPacket(t, r,
        InetAddress.getByName("thor.enseeiht.fr"), 9999);
    s.send(d);
    …
} catch (Exception e) {
    System.err.println(e);
}
```

destination

A rapid look at programming UDP communication in Java.

On the receiving side, we create a DatagramSocket giving a local port number. It corresponds to the calls in C of : socket() and bind().
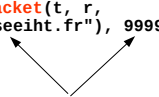
Then we can create a DatagramPacket giving an array of bytes.

Then

- receive() reads on the UDP socket and stores the data in the DatagramPacket

- getData() returns a byte array from the DatagramPacket (here we could have used the t variable)

- getLength() returns the size of the data actually received in the buffer

- getAddress() and getPort() return the address of the sender (IP and port), allowing to send a response.

On the sending side, we read in a byte array some data from a file.

We create a DatagramSocket giving a local port number.

Then we can create a DatagramPacket giving the array of bytes containing the data to send (r is the size of the data we read from the file), and also giving the destination (an InetAddress for the remote machine and a port from that machine). We send the packet with send().

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

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 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.

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

```java
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

A full example:
TCP + serialization + threads

The slave

```java
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

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 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)

Conclusion

- Programming with sockets
  - Quite simple
  - Allow fine-grained control over exchanges messages
  - Basic, can be verbose and error prone
- Higher level paradigms
  - Remote procedure/method invocation
  - Message oriented middleware / persistent messages
  - ....

Many tutorials about socket programming on the Web …
Example : https://www.tutorialspoint.com/java/java_networking.htm

35

In conclusion, programming with sockets is more or less simple (complex with C, simple in Java). Its allows to do everything regarding distribution.

But for complex applications, even in Java, it may be really error prone.

This is why higher level programming paradigms were proposed.

In the next lectures, we will study some of them (remote invocation and message middleware).

Notice that many tutorials are available on the net for socket programming.