TCP/IP programming

RES 841

November 2013

Introduction

The goal of this lab is to discover the classical programming interfaces in the TCP/IP world. The API, called *Socket*, allows programs to communicate, similarly to classical inter-process communication (pipes, queues, etc.). It is similar for most of the programming languages and offers an interface over the transport layer. It proposes two communication modes:

- Transmission of simple messages, without any guarantee on the delivery using UDP. UDP only offers the messages transfer service, we talk about *non-connected* or *Datagram* mode.
- Bi-directional and reliable exchange of messages using the TCP protocol. We talk about *connected* mode.

1 Lab description

In this lab you will use the socket API detailed below to write two programs ($Client$ and $Server$) that realize the following actions:

1. $Client$ will query `www.google.com` and get the home page in HTML code.
2. $Client$ will use the school's SMTP server (`smtp.enst.fr`) to send the HTML code to your e-mail address.
3. $Client$ will transmit the web page to $Server$, that will be listening on port 12345 and that will display, without interpreting, the page code.

The lab intentionally contains few details so that you think about how to build these programs and so that you are confronted to the classical difficulties. To understand the exchanges to implement, you can refer to the previous lab, but also examine what a real browser does and refer to the protocols documentations (RFC etc.).

The lab requires an Internet connection and no administrator access. You will then realize it on real machines, and **not** within the virtualized environment.

To use the socket API, you will include some libraries. If you use C language, you will need to include `<sys/socket.h>` and `<netdb.h>`. You could also need `<stdio.h>`, `<string.h>` or `<unistd.h>`. In Java, you will import `java.net.*` and `java.io.*`.

2 UDP Sockets

With UDP, as reliable delivery of messages is not guaranteed, a receiver starts to listen for incoming transmissions on an applicative port. To realize this operation, a process starts to create a *socket* object, describing the communication parameters (address type, port number), then it *registers* this socket at the operating system level. This registration is generally called `bind`. The system will redirect the traffic incoming on the specified port to the application. Below are two examples of usage of this API, in C and Java.

```
// Socket object creation
//  Parameter 1 : communication domain (PF_INET = IPv4 Internet)
//  Parameter 2 :  socket type (SOCK_DGRAM = UDP)
//  Parameter 3 : protocol to use (if multiple choices)
int s = socket(PF_INET, SOCK_DGRAM, 0);

// Create a structure that stores the address and port number on which
   the system will listen.
struct sockaddr_in myAddress;

  // Address type ; AF_INET means IP address
myAddress.sin_family=AF_INET;

  // Receiver address to listen on -- Any interface
myAddress.sin_addr.s_addr = htonl(INADDR_ANY);

  // Port on which the system should listen
myAddress.sin_port=htons(80);

// OS registration
//  Param. 1 :  socket objet
//  Param. 2 & 3 : address and address size
bind(s, (struct sockaddr *)&myAddress, sizeof(myAddress));

// Structure that describes the emitter address
struct sockaddr_in senderAddress;
socklen_t length = sizeof(senderAddress);

// Effective reception (blocking) of a message
//  Param 1 : socket
//  Param 2 & 3 : reception buffer and buffer size
//  Param 4 : flags (unused here)
//  Params 5 & 6 : sources address and address size
char message[255];
int nbCars;
nbCars = recvfrom (s, message, 255, 0, (struct sockaddr *)&senderAddress
   , &length);

// Display, usage, etc. of the received data
```

Listing 1: UDP Socket in C: receiver side

```
// Socket object creation
DatagramSocket s=new DatagramSocket(null);

// Create an object that stores the address and port number on which the
    system will listen.
//  Param 1 : IP address on which the system shall listen (null for all)
//  Param 2 : port on which the system will listen
InetSocketAddress myAddress = new InetSocketAddress((InetAddress)null,
   80);

// OS registration
s.bind(myAddress);

// Effective reception (blocking) of a message
byte[] message = new byte[0x100];
DatagramPacket packet=new DatagramPacket(message, message.length);
s.receive(packet);

// Acquisition of the emitter address
InetSocketAddress senderAddress=(InetSocketAddress)packet.
   getSocketAddress();


// Socket closing
s.close();
```

Listing 2: UDP Socket in Java: receiver side

On the other size of the network, an emitter realizes the corresponding operation, creating a similar *Socket* object, specifying the IP address and the destination port. Then the emitter effectively sends its data. Note the utilization of the `gethostbyname` function, or of the `InetAddress.getByName` method that allows to query the DNS. The DNS answer is returned in a complex structure in which the relevant data needs to be extracted.

```
// Socket object creation
//  Parameter 1 : communication domain (PF_INET = IPv4 Internet)
//  Parameter 2 :  socket type (SOCK_DGRAM = UDP)
//  Parameter 3 : protocol to use (if multiple choices)
int s = socket(PF_INET, SOCK_DGRAM, 0);

// DNS query to get the destination IP address
struct hostent *destination = gethostbyname("www.enst.fr");
in_addr_t destIPAddr = *((in_addr_t *)(destination->h_addr));

// Structure that describes the receiver address and port number
struct sockaddr_in destAddress;

  //  Address type ; AF_INET means IP address
destAddress.sin_family=AF_INET;

  // Receiver address
destAddress.sin_addr.s_addr=destIPAddr;
```

```
  // Receiver port
destAddress.sin_port=htons(80);

// Preparation of the message to send
char message[] = "Hello World\n";

// Effective emission of the message
//  Param. 1 : socket identifier
//  Param 2 & 3 : message & message length
//  Param 4 : flags to specify a few parameters
//  Params 5 & 6 : destination address and structure size
sendto(s, message ,strlen(message), 0, (struct sockaddr *)&destAddress,
    sizeof(destAddress));
```

Listing 3: UDP Socket in C: sender side

```
// DNS query to get the destination IP address
InetAddress destination = InetAddress.getByName("www.enst.fr");

// Object to store the address
//  Param. 1 : IP address
//  Param. 2 : port number
InetSocketAddress destIPAddr=new InetSocketAddress(destination, 80);

// Creation of the socket object
DatagramSocket s=new DatagramSocket(null);

// Preparation of the message to send
String message = "Hello World\n";
byte[] payload = message.getBytes();
DatagramPacket packet = new DatagramPacket(payload, payload.length,
    destIPAddr);

// Effective emission of the message
s.send(packet);

// Connection closing
s.close();
```

Listing 4: UDP Socket in Java: sender side

3 TCP Sockets

With TCP, a communication includes similar steps. It adds a connection opening and closing phases. All the messages that belong to the connection are sent reliably, TCP managing retransmissions when a message is not received.

The receiver, similarly to UDP, has to listen on a port. However, you may note that unlike UDP, TCP can manage multiple connections on a single port. Indeed, TCP can create a new socket using the `accept` function, this socket freeing the initial port for new connections. TCP manages a queue of incoming connections and this queue size can be increased or decreased with a parameter passed to the `listen` function, called between the `bind` and `accept` calls in C. For Java, this method is not present, but it is automatically invoked when the socket is created. The socket type, finally, differs from UDP and TCP (`SOCK_STREAM` instead of `SOCK_DGRAM`).

```
int s = socket(PF_INET, SOCK_STREAM, 0);

struct sockaddr_in myAddress;
myAddress.sin_family=AF_INET;
myAddress.sin_addr.s_addr = htonl(INADDR_ANY);
myAddress.sin_port=htons(80);

// Socket registration
bind(s, (struct sockaddr *)&myAddress, sizeof(myAddress));

// Start listening (non blocking) to connection requests.
//  Param 1 : socket
//  Param 2 : maximum number of queued connections (before rejection)
listen(s, 5);

struct sockaddr_in senderAddress;
socklen_t length = sizeof(senderAddress);

// Creation of a new socket, dedicated to an individual peer
int sData = accept (s, (struct sockaddr *)&senderAddress, &length);

// Communication with send and receive

char msg[] = "Hello World\n";
send(sData, msg ,strlen(msg), 0);

char message[255];
int nbCars;
nbCars = recv (sData, message, 255, 0);


// Display, usage, etc. of the received data

// Close BOTH sockets
close (sData);
close(s);
```

Listing 5: TCP Socket in C: receiver side

```
ServerSocket s = new ServerSocket(80);

// Creation of a new socket, dedicated to the communication
Socket sData = s.accept();

InetSocketAddress senderAddress=(InetSocketAddress)sData.
   getRemoteSocketAddress();

// Communication with read and write
String message = "Hello World\n";
PrintWriter outBuf = new PrintWriter(new BufferedWriter(new
   OutputStreamWriter(sData.getOutputStream())), true);
outBuf.println(message);

BufferedReader inBuf = new BufferedReader(new InputStreamReader(sData.
   getInputStream()));
String inText = inBuf.readLine();

// Display, usage, etc. of the received data

// Close BOTH sockets
sData.close();
s.close();
```

Listing 6: TCP Socket in Java: receiver side

The emitter side is similar to UDP. You will node that the *socket* type is different of the UDP socket type (`SOCK_STREAM` instead of `SOCK_DGRAM`), and a `connect` function is called before starting the applicative dialog.

```
int s = socket(PF_INET, SOCK_STREAM, 0);

struct hostent *destination = gethostbyname("www.enst.fr");
in_addr_t destIPAddr = *((in_addr_t *)(destination->h_addr));

struct sockaddr_in destAddress;
destAddress.sin_family=AF_INET;
destAddress.sin_addr.s_addr=destIPAddr;
destAddress.sin_port=htons(80);

// Socket connection
connect(s, (struct sockaddr *)&destAddress, sizeof(destAddress));

// Communication with send and recv
// See examples at the receiver side

// Display, exploitation of the data

// Close the sockets
close(s);
```

Listing 7: TCP Socket in C: sender side

```
InetAddress destination = InetAddress.getByName("www.enst.fr");

InetSocketAddress destIPAddr=new InetSocketAddress(destination, 80);

// Socket creation
Socket s=new Socket();

// Socket connection
s.connect(destIPAddr);

// Communication with send and recv
// See examples at the receiver side

// Display, exploitation of the data

// Close the sockets
s.close();
```

Listing 8: TCP Socket in Java: sender side