Computer Networks (*Redes de Computadores - RCOMP*) – 2024/2025

Laboratory Class 09 (PL09 – 3 hours)

Berkeley Sockets. Introduction to used environments: C/UNIX and JAVA.

Simple UDP clients and servers.

1. Programming languages and networking environments to be used

In this course, two programming languages will be used to develop network applications: **C** and **Java**

When developing network applications, the way they communicate with each other should not be dependent on the programming language, neither on the underling operating system.

This goal is achieved by defining with no ambiguities the contents of each communication, meaning the **application protocol**. To highlight the importance of establishing an unambiguous application protocol, during these classes, applications will be developed both in C and Java and run on different operating systems. Even so, because the same application protocol is used on both sides, they should communicate with each other without any problems.

In addition to timing issues (synchronization), one key factor to ensure the success of communication through an application protocol is specifying accurately the formats for exchanged data, and that must be implementation independent. For instance, when exchanging an integer number between two applications, sending the memory bytes where the integer is stored in the local system, is unacceptable. The way data is stored, depends on the operating system and platform, thus, sending data as is stored in the source node will most likely lead to misinterpretation on the destination node.

One of the simplest solutions for abstract data transfer is representing data as human legible text, this is because that is a concept that exists, and it’s supported on every system.

1.1. The DEI private networks

The *Departamento de Engenharia Informática* maintains an extensive infrastructure of networks with the purpose of supporting learning, investigation, and operational activities. Most of those networks are private, meaning they use private IP addresses, both IPv4 and IPv6.

As students should now be perfectly aware, private networks are not accessible from public networks (the internet), however, thanks to the NAT techniques, public networks may be accessible from private networks.

Important: even though **EDUROAM** is itself a private network, it is external to DEI, thus in what concerns interactions with the DEI private networks, it must be viewed as a public network.

1.2. The DEI laboratories network and available Linux servers

One of such DEI private networks is the laboratories network (LABS), it supports both IPv4: **10.8.0.0/16** and IPv6: **fd1e:2bae:c6fd:1008::/64**.

A node (e.g., workstation, laptop) is connected to the DEI laboratories network (LABS) in either of the two following conditions:

- It is physically connected by a cable to a network outlet at a DEI laboratory.
- It is connected to the DEI students’ VPN service (**deinet.dei.isep.ipp.pt**), and that may be accomplished from anywhere, as far as there is an internet connection available.

Additional information about the use of DEI VPN services, and other network services provided by DEI, is available, in Portuguese and English, at:

https://rede.dei.isep.ipp.pt

1.3. Available Linux servers

Among several other services, DEI maintains six Linux servers available to users, they are accessible through the SSH (Secure Shell) application protocol. A suitable SSH client must be used (e.g., Putty), once logged in, users have a command line terminal environment where they can edit, compile, and run C and Java applications.

SSH access to these servers is allowed from ISEP networks, but not from everywhere on the internet, outside ISEP you may yet access these servers by first connecting to a DEI VPN service.

Even though the SSH access is provided through a public network and public DNS names, each of these servers is also connected to the DEI laboratories private network (LABS). Table 1 presents the public DNS names for accessing those servers through SSH, and the corresponding IPv4 and IPv6 addresses those servers have at the LABS network.

Table 1 - Available DEI SSH servers

Public DNS name for SSH access	IPv4 address (DEI LABS)	IPv6 address (DEI LABS)
ssh1.dei.isep.ipp.pt (vsrv24.dei.isep.ipp.pt)	10.8.0.80	fd1e:2bae:c6fd:1008::80
ssh2.dei.isep.ipp.pt (vsrv25.dei.isep.ipp.pt)	10.8.0.81	fd1e:2bae:c6fd:1008::81
ssh3.dei.isep.ipp.pt (vsrv26.dei.isep.ipp.pt)	10.8.0.82	fd1e:2bae:c6fd:1008::82
ssh4.dei.isep.ipp.pt (vsrv27.dei.isep.ipp.pt)	10.8.0.83	fd1e:2bae:c6fd:1008::83
ssh5.dei.isep.ipp.pt (vsrv28.dei.isep.ipp.pt)	10.8.0.84	fd1e:2bae:c6fd:1008::84
ssh6.dei.isep.ipp.pt (vsrv29.dei.isep.ipp.pt)	10.8.0.85	fd1e:2bae:c6fd:1008::85

- For laboratory classes' practical exercises, these servers will be used when developing and testing network applications.

- When incorporating in the exercises network applications running in students' workstations, it is assumed those workstations are connected to the DEI LABS private network, either physically or through the **deinet.dei.isep.ipp.pt** VPN service.

- Mind in this scenario all applications will be running at the DEI LABS private, being a private network means servers running there will not be reachable from public networks (the internet), nevertheless, clients running at private networks can reach servers running at public networks.

When testing network applications using these available Linux servers, students should enrol different Linux servers, enforcing the real use of network communications (if both applications are running at the same server there will be no real network communication).

For instance, if the purpose is testing two network applications A and B which communicate with each other, then they should be run at different servers, for instance, running application A at ssh2 and running application B at ssh3 as shown in Figure 1.

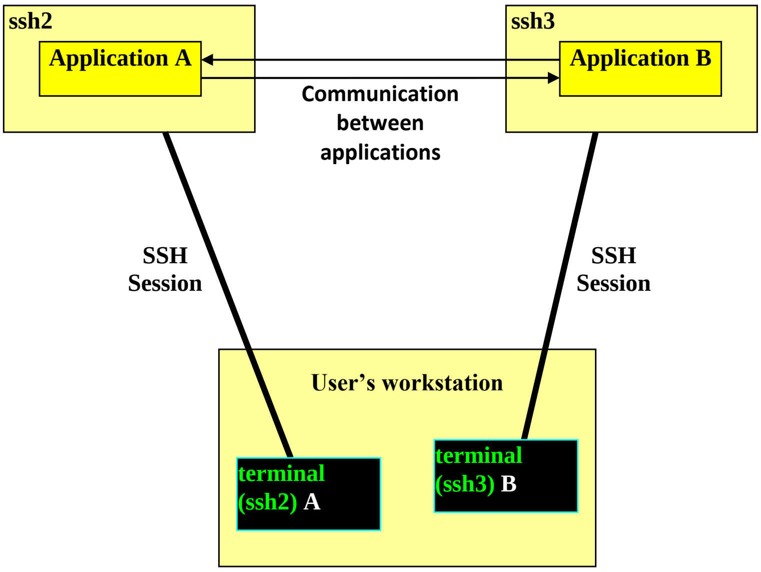

Figure 1 - Two network applications, running in different SSH servers

Please, take a good look at the above image to fully understand the overall scenario when using DEI provided SSH servers.

Example programs to be used in laboratory classes are available in the following repository:

```
https://github.com/asc-isep-ipp-pt/PROGS-RCOMP
```

(Bear in mind this repository may be updated until corresponding laboratory classes actually take place)

1.4. DEI Virtual Servers Cloud

In addition to the DEI supplied SSH Linux servers (Table 1), there is a complementary environment that may be used in DEI to run network applications.

The **DEI Virtual Servers Cloud** infrastructure allows DEI users to create and administrate servers.

Having the administrator role over the created servers has some advantages, for instance users may install any software package they need, also, they are allowed to use TCP and UDP local port numbers below 1024. Notice that in DEI provided SSH Linux servers, mentioned before, users don´t have administrative privileges.

This experimental environment operates over a cluster of servers, for now it has a single access frontend:

https://vs-ctl.dei.isep.ipp.pt/

Even though the frontend is publicly accessible, the created virtual servers run in a specific DEI private network, the VNET1, also supporting both IPv4: **10.9.0.0/16** and IPv6: **fd1e:2bae:c6fd:1009::/64**.

IPv4 and IPv6 routing between the DEI private networks is guaranteed, therefore, network applications running at the LABS networks will talk directly with network applications running at the VNET1 network.

At the **DEI Virtual Servers Cloud,** you will find an extensive variety of templates from where to create your server, for this course practical activities the suggestion goes to template number **57**:

Apache and others on Debian 11 (Bullseye)

This template has several software packages already installed, including the C and java compilers.

1.5. Compiling and running C applications (Linux)

- The source code can be created by using a plain text editor like **vi** or **nano**, therefore, by running a command like:

```
vi SOURCE-FILE.c          or     nano SOURCE-FILE.c
```

- The source file (SOURCE-FILE.c) may then be compiled using **gcc** (*GNU Compiler Collection*):

```
gcc SOURCE-FILE.c –o EXECUTABLE-FILE
```

If the **–o** option is not used, then an **a.out** named executable file will be created.

- To run the application, simply call it from the command line:

```
./EXECUTABLE-FILE
```

- In the formerly mentioned repository, each folder has a **Makefile**, thus, the **make** command should be used to compile all applications present in the folder.

1.6. Compiling and running Java applications (Linux and Windows)

- One major advantage of Java language is that compiled applications run over a platform known as Java Virtual Machine (JVM), this guarantees a high degree of abstraction from the underlying operating system. One of the most widely used JVM implementations is JRE (*Java Runtime Environment*) from ORACLE.

- JRE is available on the DEI Linux servers, and most likely already installed in the student personal workstation, thus applications develop in Java language may be used in any of these environments.

- Again, source code can be created by using a plain text editor like **vi** or **nano**, running command like:

```
vi SOURCE-FILE.java       or     nano SOURCE-FILE.java
```

- The source file (**SOURCE-FILE.java**) can then be compiled using the Java compiler:

```
javac SOURCE-FILE.java
```

A file named **CLASS-NAME.class** will be created for each class declared in source file **SOURCE-FILE.java**.

- To run the application, simply call the JRE:

```
java CLASS-NAME
```

Where **CLASS-NAME** is the name of the class implementing the **main** method.

- Again, in the repository each folder has a **Makefile**, the **make** command can be used to compile all applications present in the folder.

2. UDP clients and servers

UDP clients and servers are application that use UDP datagrams to communicate with each other, using the **client-server model**. As represented in Figure 2, the server application receives a UDP datagram transporting the request, then processes the request's content and sends back another UDP datagram containing the response (the processing outcome).

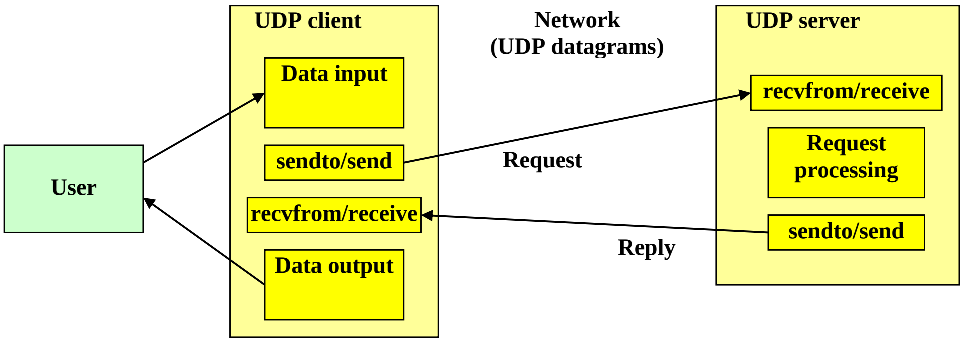

Figure 2 - UDP client/server interactions

Typically, user interaction takes place at the client application side, to start the user is normally required to provide the server's node address to where the client will be sending the requests. Then data to be processed is prompted to the user and sent inside a UDP datagram to the provided server's address. The client application must then wait for a UDP datagram arrival containing the response and usually presents the content to the user.

2.1. Implementing an example of UDP client and UDP server

Create a pair of applications: a UDP client and a UDP server with the following features:

The client application:

1 - Receives a server's IP address or DNS name as the first argument in the command line.

2 - Reads a text line on the console (*string*), if the text content is "exit" then the client application exits, otherwise, sends the string's content (ASCII text) inside a UDP datagram (request) to the server, to a fixed port number (9999 in the provided example).

3 - Receives (waits for) a UDP datagram (response) containing a string and prints the string's content at the terminal.

4 – Repeats from step 2

The server application:

1 - Receives a UDP datagram (request) in a fixed port number (9999 in the provided example), containing a string. The client source IP address and port number should be printed at the terminal.

2 - Mirrors the string.

3 - Send back to the client a UDP datagram (response) containing the mirrored string.

4 – Repeats from step 1.

Remarks: Both IPv4 and IPv6 should be supported. To avoid conflicts, given that several students may use the same DEI Linux server to run the server application, each should use a different port number, as suggested by the laboratory class teacher.

2.1.1. UDP client in C language (udp_cli.c)

```
#include <strings.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUF_SIZE 300
#define SERVER_PORT "9999"

// read a string from stdin protecting buffer overflow
#define GETS(B,S) {fgets(B,S-2,stdin);B[strlen(B)-1]=0;}

int main(int argc, char **argv) {
       struct sockaddr_storage serverAddr;
       int sock, res, err;
       unsigned int serverAddrLen;
       char linha[BUF_SIZE];
       struct addrinfo  req, *list;

       if(argc!=2) {
              puts("Server IPv4/IPv6 address or DNS name is required as argument");
              exit(1);
              }
       bzero((char *)&req,sizeof(req));
       // let getaddrinfo set the family depending on the supplied server address
       req.ai_family = AF_UNSPEC;
       req.ai_socktype = SOCK_DGRAM;
       err=getaddrinfo(argv[1], SERVER_PORT , &req, &list);
       if(err) {
              printf("Failed to get server address, error: %s\n",gai_strerror(err)); exit(1); }
       serverAddrLen=list->ai_addrlen;
       // store the server address for later use when sending requests
       memcpy(&serverAddr,list->ai_addr,serverAddrLen);  freeaddrinfo(list);

       bzero((char *)&req,sizeof(req));
       // for the local address, request the same family as determined for the server address
       req.ai_family = serverAddr.ss_family;
       req.ai_socktype = SOCK_DGRAM;
       req.ai_flags = AI_PASSIVE;                 // local address
       err=getaddrinfo(NULL, "0" , &req, &list);  // port 0 = auto assign
       if(err) {
              printf("Failed to get local address, error: %s\n",gai_strerror(err)); exit(1); }

       sock=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
       if(sock==-1) {
              perror("Failed to open socket"); freeaddrinfo(list); exit(1);}
       if(bind(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
              perror("Failed to bind socket");close(sock);freeaddrinfo(list);exit(1);}

       freeaddrinfo(list);

       while(1) {
              printf("Request sentence to send (\"exit\" to quit): ");
               GETS(linha,BUF_SIZE);
              if(!strcmp(linha,"exit")) break;
              sendto(sock,linha,strlen(linha),0,(struct sockaddr *)&serverAddr,serverAddrLen);
              res=recvfrom(sock,linha,BUF_SIZE,0,(struct sockaddr *)&serverAddr,&serverAddrLen);
               linha[res]=0; /* NULL terminate the string */
              printf("Received reply: %s\n",linha);
              }
       close(sock); exit(0);
       }
```

- The client application starts by analysing the server's address to where it is supposed to send the requests. This is most relevant because depending on the type of address, the appropriate corresponding local socket must be created. The getaddrinfo() function analyses the provided server's address, with the given arguments: SOCK_DGRAM, of any family (**AF_UNSPEC**), for the server address, and the port number where the server will be receiving.

- To be able to free the dynamic memory allocated by getaddrinfo() for the linked list, and because later the server address structure will be required, the server address structure is copied from the linked list to serverAddr.

- Now the appropriate local address can be obtained by calling getaddrinfo() again, this time, the specific family, as determined by the getaddrinfo() previous call (for the server address) is requested. Because it's a local address, the flag AI_PASSIVE must be used, and the host address for getaddrinfo() may be NULL. Being a client, it does not need a fixed port number, so "0" is used instructing bind to use any available (not in use) local port number.

- The data created by getaddrinfo() can now be used to open a suitable socket and bind it to the appropriate local address.

- Now everything is ready for communications to take place, the client application reads a text line from the console to a buffer and then sends a UDP datagram carrying the string to the server. Afterwards the client waits for a response UDP datagram (the client application blocks here).

- When (and if) a response UDP datagram arrives, recvfrom() unblocks, stores the datagram payload in the buffer, and returns the number of bytes in the received payload. For the buffer to be directly printed in C it must be null terminated, so the zero value is placed in the buffer position corresponding to the number of bytes received.

2.1.2. UDP client in Java language (*UdpCli.java*)

```
import java.io.*;
import java.net.*;

class UdpCli {
      static InetAddress serverIP;

      public static void main(String args[]) throws Exception {
            byte[] data = new byte[300];
            String frase;

            if(args.length!=1) {
                  System.out.println("Server IP address/DNS name is required as argument");
                  System.exit(1);
                  }

            try { serverIP = InetAddress.getByName(args[0]); }
            catch(UnknownHostException ex) {
                  System.out.println("Invalid server address supplied: "  + args[0]);
                  System.exit(1);
                  }

            DatagramSocket sock = new DatagramSocket();
            DatagramPacket udpPacket = new DatagramPacket(data, data.length, serverIP, 9999);

            BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

            while(true) {
                  System.out.print("Sentence to send (\"exit\" to quit): ");
                  frase = in.readLine();
                  if(frase.compareTo("exit")==0) break;
                  udpPacket.setData(frase.getBytes());
                  udpPacket.setLength(frase.length());
                  sock.send(udpPacket);
                  udpPacket.setData(data);
                  udpPacket.setLength(data.length);
                  sock.receive(udpPacket);
                  frase = new String( udpPacket.getData(), 0, udpPacket.getLength());
                  System.out.println("Received reply: " + frase);
                  }
```

```
		sock.close();
		}
	}
```

- The getByName() method of the InetAddress class is used to transform the argument string representing the server (IP address or DNS name) into an InetAddress object holding the server’s IP address.

- A DatagramSocket class object is created, the used constructor takes no arguments, so any local free port number will be assigned (this is equivalent to binding to port zero in C language).

- A DatagramPacket object is created, the used constructor receives the content of the datagram (payload), the payload’s number of bytes, the destination IP address, and the destination port number. This DatagramPacket object will be used for both sending the request and receiving the reply.

- A line of text is read from the console to a string, if the content is “exit”, then the loop is finished, and the application exists after closing the socket.

- Otherwise, the string’s content is set as payload for the datagram and the datagram can then be sent by calling the socket’s send() method.

- The DatagramPacket object is then prepared for receiving the server reply, the buffer and maximum datagram size are set. Next the response datagram may be received by calling the socket’s receive() method. **This is a blocking operation**.

- When (and if) a response UDP datagram arrives, the receive() method unblocks filling data in the DatagramPacket object and buffer.

- For the purpose of printing at the console, a string is created from the received datagram’s payload.

2.1.3. UDP server in C language (*udp_srv.c*)

```
#include <strings.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUF_SIZE 300
#define SERVER_PORT "9999"

int main(void) {
	struct sockaddr_storage client;
	int err, sock, res, i;
	unsigned int adl;
	char linha[BUF_SIZE], linha1[BUF_SIZE];
	char cliIPtext[BUF_SIZE], cliPortText[BUF_SIZE];
	struct addrinfo  req, *list;

	bzero((char *)&req,sizeof(req));
	// request a IPv6 local address will allow both IPv4 and IPv6 clients to use it
	req.ai_family = AF_INET6;
	req.ai_socktype = SOCK_DGRAM;
	req.ai_flags = AI_PASSIVE;   // local address

	err=getaddrinfo(NULL, SERVER_PORT , &req, &list);

	if(err) {
		printf("Failed to get local address, error: %s\n",gai_strerror(err)); exit(1); }

	sock=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
	if(sock==-1) {
		perror("Failed to open socket"); freeaddrinfo(list); exit(1);}

	if(bind(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
```

```
                perror("Bind failed");close(sock);freeaddrinfo(list);exit(1);}

        freeaddrinfo(list);

        puts("Listening for UDP requests (IPv6/IPv4). Use CTRL+C to terminate the server");

        adl=sizeof(client);
        while(1) {
                 res=recvfrom(sock,linha,BUF_SIZE,0,(struct sockaddr *)&client,&adl);
                if(!getnameinfo((struct sockaddr *)&client,adl,
                        cliIPtext,BUF_SIZE,cliPortText,BUF_SIZE,NI_NUMERICHOST|NI_NUMERICSERV))
                        printf("Request from node %s, port number %s\n", cliIPtext, cliPortText);
                else puts("Got request, but failed to get client address");
                for(i=0;i<res;i++) linha1[res-1-i]=linha[i]; // create a mirror of the text line
                sendto(sock,linha1,res,0,(struct sockaddr *)&client,adl);
                }
        close(sock);
        exit(0);
        }
```

- The server application requests to the getaddrinfo() function an IPv6 local address (AF_INET6), this will allow UDP clients using either UDP over IPv4 or UDP over IPv6, the only catch is that IPv4 client addresses will be handled as IPv4-Mapped IPv6 addresses.

- To the getaddrinfo() function, a NULL host is passed because this is a local address and the fixed port number is enforced (9999 in the example).

- Data provided by getaddrinfo() is then used to create the socket and bind it to the local address and port number.

- The server then starts a never-ending loop for receiving requests and sending corresponding replies.

- The server's main loop starts by calling recvfrom() to receive a UDP datagram transporting the request, if there's no request to be received the process will be blocked until one arrives. The client address (IP address and port number) is then stored in **client** structure of sockaddr_storage type, **the sockaddr_storage structure size must be defined in last argument (integer pointer) prior to calling recvfrom()**.

- The **getnameinfo()** is used to obtain strings representing the source IP address and source port number stored by **recvfrom()** in **client** structure. Please remember that being an IPv6 socket, IPv4 client addresses will appear as IPv4-Mapped.

- A mirror of the received string is then created and sent to the client's address (IP and port), as stored in the **client** structure by **recvfrom()**.

2.1.4. UDP server in Java language (*UdpSrv.java*)

```
import java.io.*;
import java.net.*;

class UdpSrv {
        static DatagramSocket sock;
        public static void main(String args[]) throws Exception {
                byte[] data = new byte[300];
                byte[] data1 = new byte[300];
                int i, len;

                try { sock = new DatagramSocket(9999); }
                catch(BindException ex) {
                        System.out.println("Bind to local port failed");
                        System.exit(1);
                        }

                DatagramPacket udpPacket= new DatagramPacket(data, data.length);
                System.out.println("Listening for UDP requests (IPv6/IPv4). CTRL+C to terminate");
                while(true) {
                        udpPacket.setData(data);
                         udpPacket.setLength(data.length);
                sock.receive(udpPacket);
                len=udpPacket.getLength();
                System.out.println("Request from: " +
                       udpPacket.getAddress().getHostAddress() + " port: " +
                       udpPacket.getPort());
                for(i=0;i<len;i++) data1[len-1-i]=data[i];
                udpPacket.setData(data1);
                udpPacket.setLength(len);
                sock.send(udpPacket);
               }
        }
}
```

- A DatagramSocket class object is created, the used constructor receives an integer fixed local port number to bind the socket to. This, of course may, raise an exception if that UDP port number is already being used on the local host.

- A new DatagramPacket object is created, the used constructor defines only a buffer and the buffer size. The IP address and port number will be set when a datagram is received (source IP address and source port number).

- The server then starts a never-ending loop for receiving requests and sending corresponding responses. The receive() method is called to receive the datagram (client request), if no datagram has been received, the thread will block until one arrives.

- After receiving the request, the source IP address and source port number are stored in the DatagramPacket object, so there is no need to change them when sending a reply because the same DatagramPacket is used for that purpose. Source IP address and source port number are printed at the server's console.

- A mirrored version of the string carried by the request UDP datagram is created and defined as the payload of the response datagram. The response is sent by calling the socket's send() method.

3. Example applications building and testing

Before testing, the only change required in the source code is on port numbers. On server applications, the local port number where the application is listening for requests on. In client applications, the remote server port to where the client is sending requests.

Of course, two server applications using the same port number cannot run on the same node. Because all students are going to use the same set of hosts (the mentioned SSH servers), the class teacher will settle a different port number for each student or team.

3.1. Compile the C applications in Linux

```
gcc udp_cli.c –o udp_cli
gcc udp_srv.c –o udp_srv
```

Executable files udp_cli and udp_srv can then be called on the command line by running *./udp_cli* and *./udp_srv*, don´t forget the udp_cli is expecting the server's IP address on the command line.

3.2. Compile the Java applications

```
javac UdpCli.java
javac UdpSrv.java
```

Java runnable class files ***UdpCli.class*** and ***UdpSrv.class*** are created (the class file names match the names of the classes declared in source code). To run the main() method of these classes, at the command line use

java UdpCli and ***java UdpSrv***, again, don't forget ***UdpCli*** main() method is expecting the server's IP address or DNS name at the command line.

3.3. Testing applications

Several client/server scenarios can be tested, while performing the following tests always pay attention to the server terminal console for feedback about the client's IP address and port number.

Depending on nodes where applications are run, several possible scenarios exist, they are explored next.

3.3.1. C/Linux – C/Linux

As presented in Figure 3, open two SSH sessions, one in **ssh1** and another in **ssh3**.

Place **udp_srv** running in **ssh1**, then start **udp_cli** in **ssh3**.

The **udp_cli** application requires the server's IP address on the command line, so use:

```
./udp_cli 10.8.0.80
```

Test the applications by entering strings at the client's console and seeing responses being received.

Also, test using IPv6 instead of IPv4:

```
./udp_cli fd1e:2bae:c6fd:1008::80
```

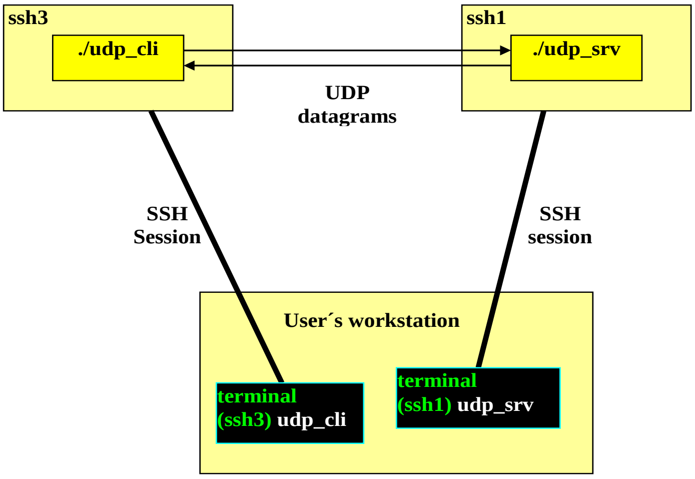

Figure 3 - UDP client/C and server/C running in two SSH servers

3.3.2. JAVA/Linux – C/Linux

Run the test with exactly the same outline as before but now using the Java version of the client application (Figure 4).

On **ssh3** run:

```
java UdpCli 10.8.0.80
```

, and for IPv6:

```
java UdpCli fd1e:2bae:c6fd:1008::80
```

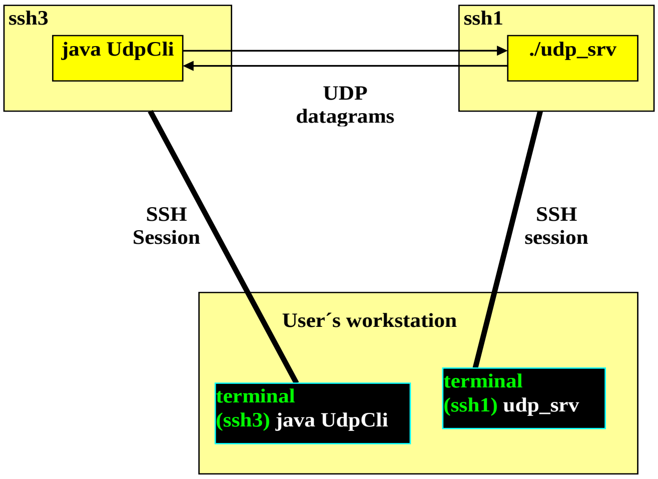

Figure 4 - UDP client/Java and server/C running in two SSH servers

3.3.3. JAVA/Other – C/Linux

Keep the server application running in **ssh1**. Compile the Java client application on your personal workstation (Figure 5), to run it use:

```
java UdpCli 10.8.0.80
```

, and for IPv6:

```
java UdpCli fd1e:2bae:c6fd:1008::80
```

WARNINGS: for this layout to work, ssh1 must be reachable from the user's workstation. Meaning the user's workstation must be connected to a DEI network or a DEI VPN service. Also bear in mind, not all VPN services support IPv6.

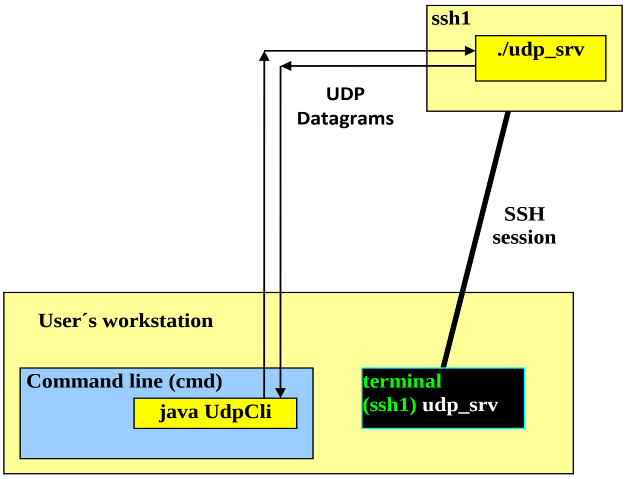

Figure 5 - UDP client/Java at the workstation, server/C at an SSH server

3.3.4. C/Linux – JAVA/Linux

Like in 3.3.1, but now use the Java version of the server application in **ssh1** (Figure 6).

In **ssh3** run:

```
./udp_cli 10.8.0.80
```

Test the applications by entering strings at the client and seeing the replies being received.

Also test using IPv6:

```
./udp_cli fd1e:2bae:c6fd:1008::80
```

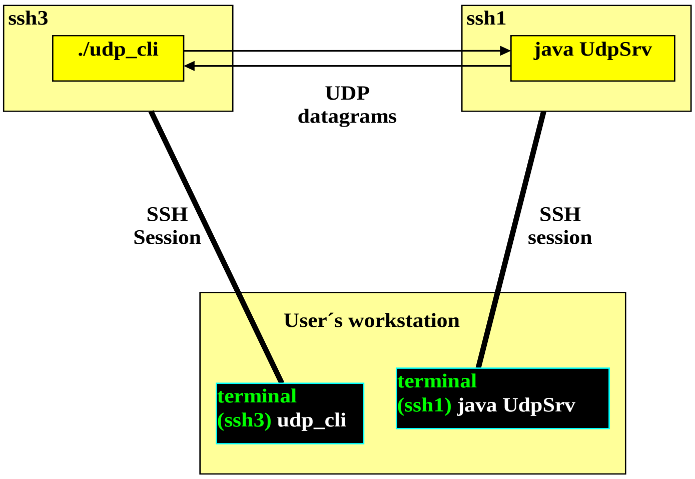

Figure 6 - UDP client/C and server/Java running in two SSH servers

3.3.5. JAVA/Linux – JAVA/Linux

The same as before but now use Java versions for both the client and the server (Figure 7).

In **ssh3** run:

java UdpCli 10.8.0.80

, and for IPv6:

java UdpCli fd1e:2bae:c6fd:1008::80

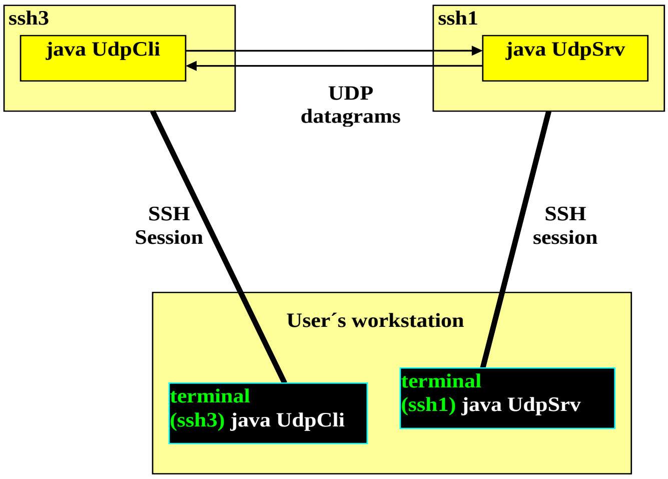

Figure 7 - UDP client/Java and server/Java running in two SSH servers

3.3.6. JAVA/Other – JAVA/Linux

The same as for 3.3.3, but now with the Java version of the server in **ssh1** (Figure 8). Compile the Java client application on your own personal workstation, and then run it:

java UdpCli 10.8.0.80

, and for IPv6:

java UdpCli fd1e:2bae:c6fd:1008::80

WARNINGS: for this layout to work, ssh1 must be reachable from the user's workstation. Meaning the user's workstation must be connected to a DEI network or a DEI VPN service. Also bear in mind, not all VPN services support IPv6.

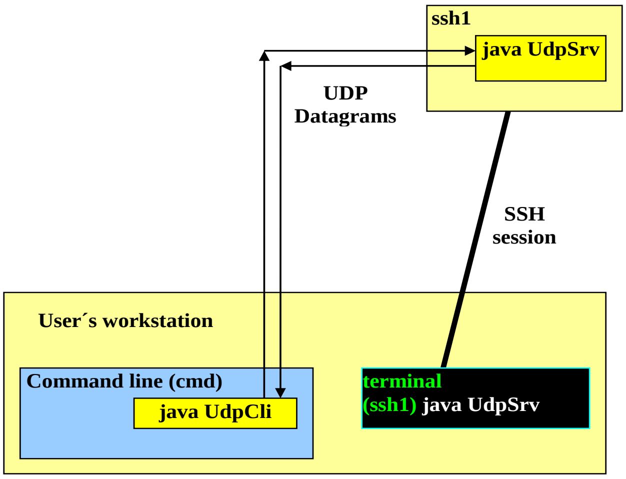

Figure 8 - UDP client/C at the workstation, server/Java at an SSH server

3.3.7. Others – UdpSrv on the user's workstation (Figure 9 and Figure 10)

WARNINGS: workstations usually have a local firewall enabled; standard firewall configurations block all incoming traffic by default. In the workstation running **UdpSrv**, you may have to either temporarily disable the firewall for the purpose of this experiment or create an incoming traffic rule allowing UDP traffic into your port number. Moreover, and again, the server application must be reachable from the node you are running the client application. So, either the server is running on a public network, or both the server and the client are on interconnected private networks.

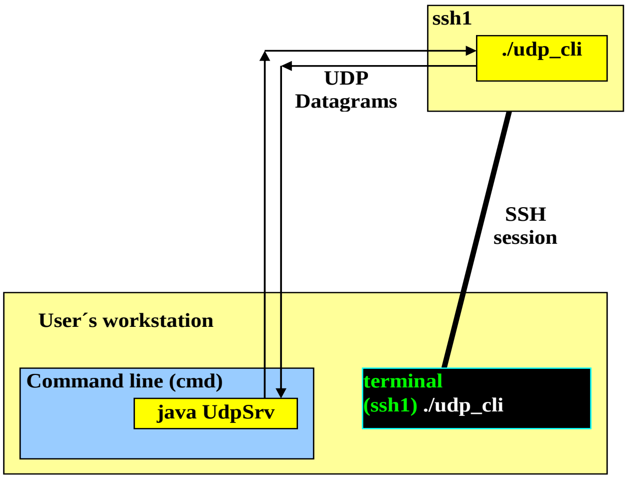

Figure 9 - UDP server/Java at the workstation and client/C at an SSH server

To run the client, you must first know the IP address of the node you are running the server on (in the Windows command line, the **ipconfig** command may be used to get that).
Then run:

./udp_cli SERVER-IP-ADDRESS

or

java UdpCli SERVER-IP-ADDRESS

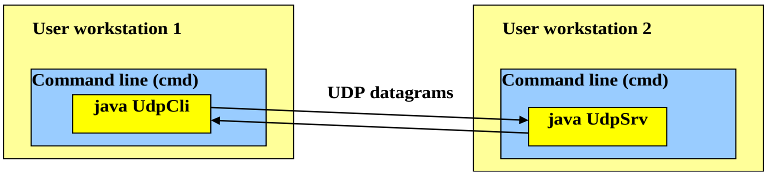

Figure 10 - UDP client/Java and server/Java at two workstations

3.3.8. Others – localhost only, a last resource solution (Figure 11)

If no other option is available (e.g., the workstation has no network connection), these applications can still be tested in the user's workstation, in this case the server address (and the client address as well) is the one assigned to the loopback interface:

IPv4:	**127.0.0.1**
IPv6:	**::1**

On most operating systems the **localhost** hostname is mapped to these addresses, so it may be used as well to specify the server's address.
Communications using the loopback interface are limited to applications running locally and are usually ignored by default firewall configurations.

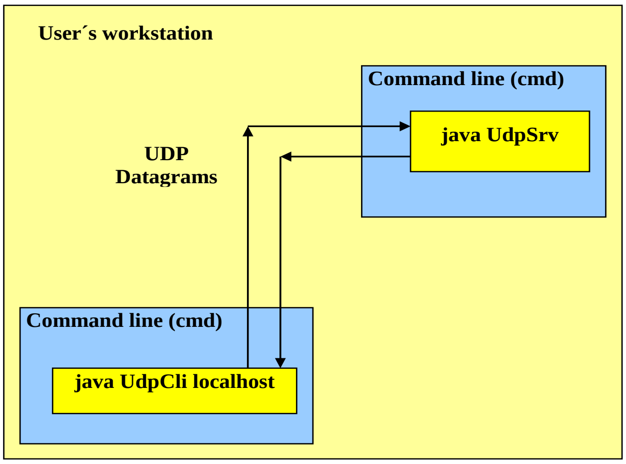

Figure 11 - UDP client and server running at the same workstation

Open two command line windows and run the server application in one of them:

java UdpSrv

In the other command line window start the client:

java UdpCli localhost

Instead of **localhost**, **127.0.0.1** or **::1** may be used to force the use of IPv4 or IPv6, respectively.

Additional remark about wireless networks:

One would expect that by having two laptops connected to the same wireless infrastructure (e.g., EDUROAM), with no VPN connected, then direct communication between them, would be warranted.

And yet that might not be the case; to avoid some types of misuse, network administrators have the habit of configuring wireless access-points to block traffic between wireless nodes associated to the same access-point.

This may ultimately lead to a rather peculiar outcome: neighbouring laptops are not able to communicate (as they are associated to the same access-point), and yet more apart laptops may be able to communicate (because they are associated to different access-points).