Redes de Computadores (RCOMP) – 2017/2018 Laboratory Class Script PL13	
• Network applications development over Berkeley Sockets. • Introduction to used environments: C/UNIX and JAVA. • UDP clients and servers. • Project 2 start - objectives and guidelines	

1. Network applications will be developed in two programming languages: C and Java

When developing network applications it's mandatory that the way they communicate with each other is not dependent on the programming language or the underling operating system.

This goal is achieved by defining with no ambiguities the contents of each communication, i.e. 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. Even so, 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 settling accurately and implementation independent the data formats.

For instance, when transmitting an integer between two applications, sending the memory bytes where the integer is stored is wrong. The way data is stored, depends on the operating system and platform, thus, sending data as it's stored in the source node will very likely lead to misinterpretation on the destination node.

One of the simplest solution for abstract data transfer is representing data as legible text.

1.1. Laboratories' network and available Linux servers for students

- A single network is shared by all DEI laboratories, it's a private network supporting both IPv4: 10.8.0.0/16 and IPv6: fd1e:2bae:c6fd:1008::/64.

- A node is connected to the laboratories' network in one of these conditions:

  a) It's connected by a cable to a network outlet in a DEI laboratory.

  b) It's connected to the DEI student's VPN service (deinet.dei.isep.ipp.pt).

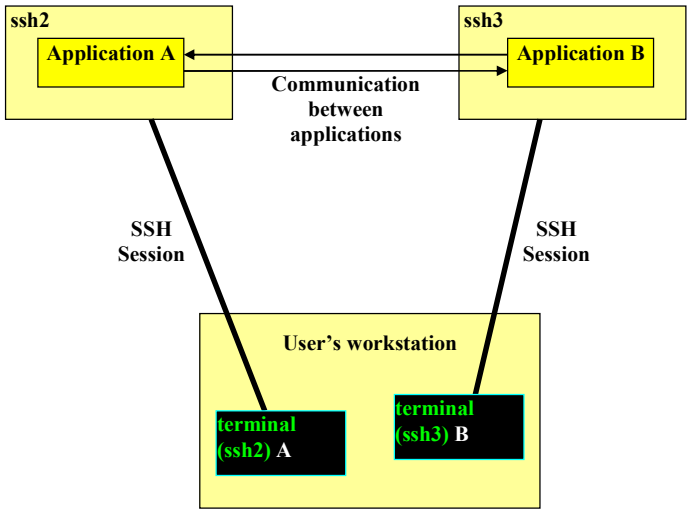
- Six Linux servers are available to users through SSH (Secure Shell). Once logged in, users have a command line terminal environment where they can write, compile and run C and Java applications. Although SSH access is provided through a public network, these servers are also connected to the DEI laboratories' private network.

DNS name for public Access	IPv4 address (LABS)	IPv6 address (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

- Students should use these servers to develop and test the network applications in laboratory classes, nevertheless, personal workstations can also be used for this purpose. **Remember though a network application running on a public network will not be able to reach a network application running on a private network.**

- 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 on the same server there will be no real network communication.

- For instance, with the purpose of testing two network applications A and B which communicate with each other, they should be run on different servers, for instance, run application A in ssh2 and run application B in ssh3:



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 matching classes actually take place)

1.2. Compiling and running C applications (Linux)

- Source code can be created by using a simple text editor like **vi** or **nano**, running command like:

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

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

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

If no **-o** option is used, an **a.out** executable file will be created.

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

```
./EXECUTABLE-FILE
```

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

1.3. 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 Windows workstation (certainly, it is also available in students personal workstations), thus applications develop in Java language may be used in any of these environments.

- Source code can be created by using a simple 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 **CLASS-NAME.class** file will be created for each class declared in source file **SOURCE-FILE.java**.

- To run the application call the JRE:

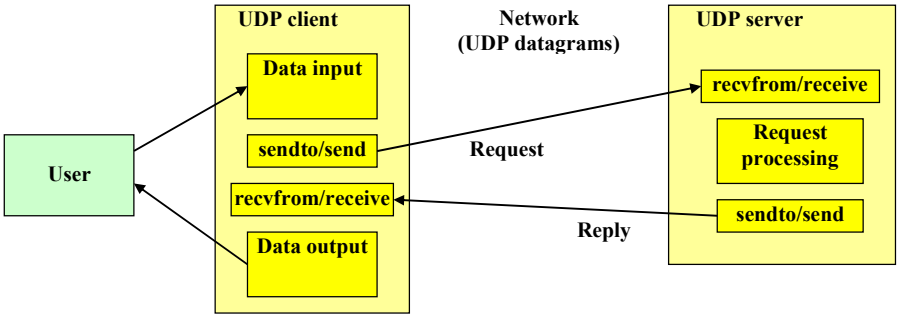
java CLASS-NAME

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 uses UDP datagrams to communicate with each other using the client-server model:



The server application receives a UDP datagram transporting the request, then processes the request content and sends back another UDP datagram containing the reply (the result of processing).

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

2.1. Implementing an example of UDP client and server

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

The client application:

1 - Receives a server 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 else sends its content (ASCII text) inside a UDP datagram (request) to the server, to a fixed port number (9999 in the provided sample).

3 - Receives a UDP datagram (reply) containing a string and prints the string content on the console.

4 – Repeats from step 2

The server application:

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

2 - Mirrors the string

3 - Send back to the client a UDP datagram (reply) 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 Linux server to run the server application, each should use a different port number 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's 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 the server will be receiving on.

- 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 determined by 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 free port number.

- Data created by getaddrinfo() can now be used to open the appropriate 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 characters to the server. Afterwards the client waits for a response UDP datagram (the client application blocks here).

- When (and if) a reply UDP datagram arrives, recvfrom() unblocks puts the datagram payload in the buffer and returns the number of bytes in the payload. In order 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.

2.1.1. 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 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 a reply datagram can be received by calling the socket receive() method. **This is a blocking operation**.

- When (and if) a reply UDP datagram arrives, 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 payload.

2.1.2. 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 sample).

- 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()**.

2.1.3. 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 constructor used 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 constructor used defines only a buffer and the buffer size. The IP address and port number are 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 replies. 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 reply datagram. The reply is sent by calling the socket's send() method.

3. Sample 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 the application is listening for requests on. In client applications, the remote server port to where the client is sending requests to.

Two server applications using the same port number can't 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 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 udp_cli is expecting the server IP address on the command line.

3.2. Compile 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 command line use ***java UdpCli*** and ***java UdpSrv***, again, don't forget ***UdpCli*** main() method is expecting the server IP address on 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.

3.3.1. C/Linux – C/Linux

Open two SSH sessions, one in **ssh1** and another in **ssh3**.

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

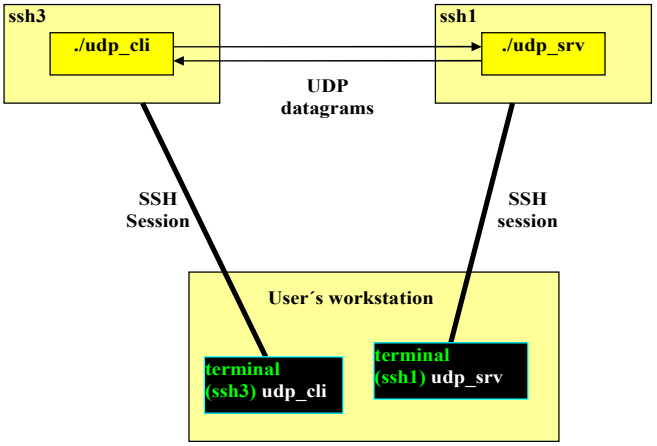
The **udp_cli** application requires the server's IP address on the command line, so use:

./udp_cli 10.8.0.80

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

Also, test using IPv6:

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



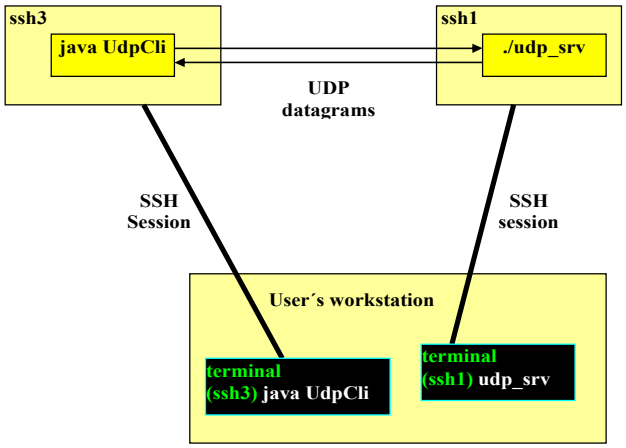
3.3.2. JAVA/Linux – C/Linux

Run test with exactly the same disposition, but now using the Java version of the client application. On **ssh3**:

java UdpCli 10.8.0.80

, and for IPv6:

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



3.3.3. JAVA/Other – C/Linux

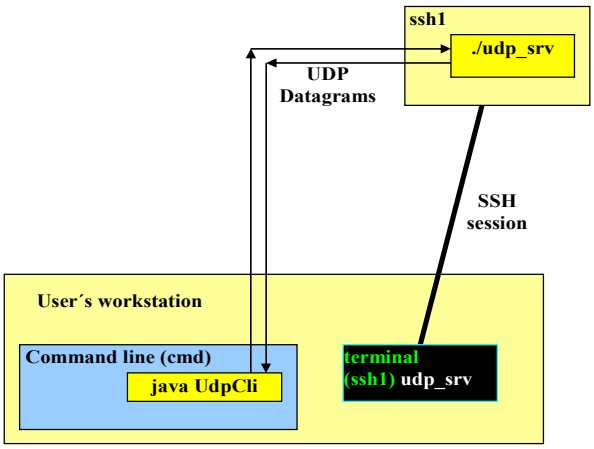
Keep the server application running in **ssh1**.
Compile the Java client application on your personal workstation, the run it:

java UdpCli 10.8.0.80

, and for IPv6:

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

WARNING: for this layout to work, ssh1 must be reachable from the user's personal workstation, thus it must be either cable connected to the laboratory network or connected to the DEI VPN service. Also, not all VPN services support IPv6.



3.3.4. C/Linux – JAVA/Linux

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

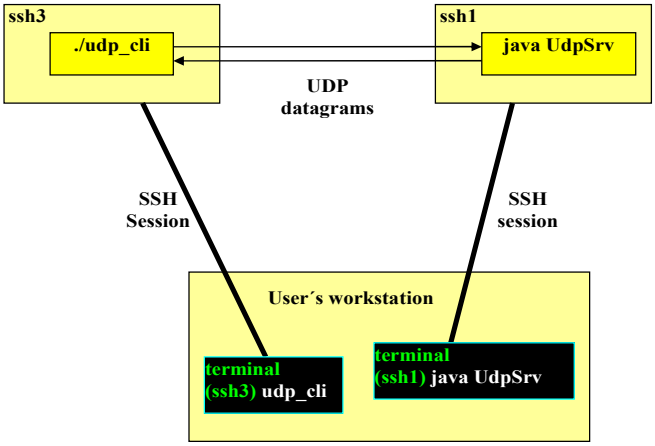
In **ssh3** run:

./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



3.3.5. JAVA/Linux – JAVA/Linux

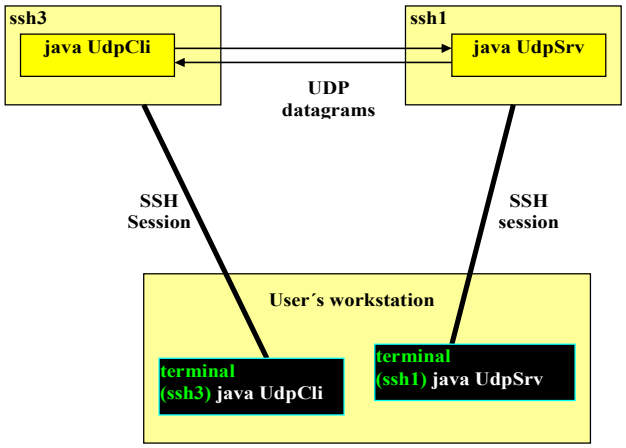
The same as before, but now use Java versions for both the client and the server.

In **ssh3** run:

java UdpCli 10.8.0.80

, and for IPv6:

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



3.3.6. JAVA/Other – JAVA/Linux

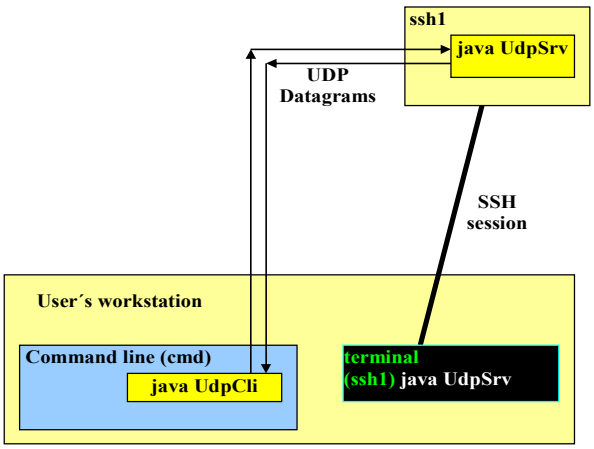
The same as 3.3.3., but now with the Java version of the server in **ssh1**. Compile the Java client application on your personal workstation, and then run it:

java UdpCli 10.8.0.80

, and for IPv6:

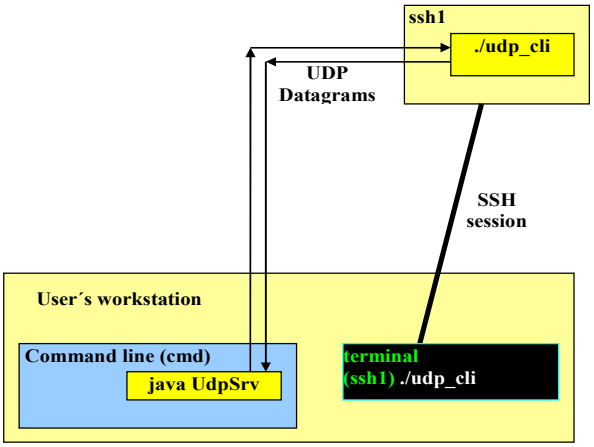
java UdpCli fd1e:2bae:c6fd:1008::80

WARNING: for this layout to work ssh1 must be reachable from the user's personal workstation, thus it must be either cable connected to the laboratory network or connected to a DEI VPN service.



3.3.7. Others – UdpSrv on the user's workstation

WARNING: workstations usually have a local firewall enabled, standard workstation firewall setups block all incoming traffic. 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 to your port number.



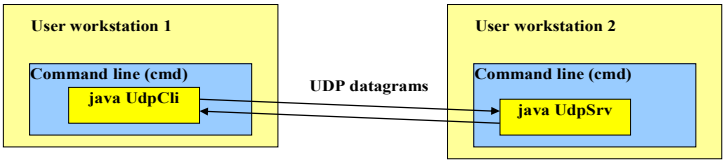
Moreover, and again, the server application must be reachable from the node you are running the client application, so the workstation should be connected to the DEI private laboratories network.

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



4. Project 2 start - objectives and guidelines

4.1. Project's proposal analysis by students

4.2. Teacher's briefing - comments and clarifications

- The main application (the server) manages walls.
- The server provides a web interface (HTTP) allowing the following use cases:
    - View text messages in a named wall. As new messages are added or removed the displayed messages must be automatically updated.
    - Add a text message to a named wall, if the wall doesn't exist is created.
    - Remove a text message from a named wall.
    - Remove a wall (and all messages within).
- The web interface is loaded only once, no HTML form submissions are to be used. The server should provide web services to the browser. The browser by using the JavaScript object **XMLHttpRequest** will act as a web services consumer (AJAX).
- Regarding HTTP, web services and AJAX, as soon as they are available at Moodle, **Lecture 09**, **TP10** and **PL21** should be studied in advance. They provide support and examples on these subjects.
- The server application is also able to receive text messages through UDP, thus it's also a UDP server. Each UDP request contains the wall name and the text message, the exact format definition of UDP requests and replies must be established (the application protocol).
- Beyond the server, a second application must be developed to send a UDP request to the server application. This will be a simple UDP client and will implement a single use case: Add a text message to a named wall.
- Concurrency issues will be present, namely mutual exclusion on accessing the text messages and walls. In Java language, most concurrency issues can be handled through the **synchronized** declaration. A support document on this subject is available at Moodle.
- Either C or Java programming language can be used, no other languages are allowed. Of course, for AJAX, JavaScript will also be used.
- Upcoming PL classes will address most implementation techniques required for this project.

4.3. First steps

This project's completion requires the use of several technologies and programming techniques yet to be addressed in lectures and classes. As mentioned before, **Lecture 09 (HTTP and web services)**, **TP10** and **PL21** must be studied in advance.

For now, teams should embrace the following steps:

- Create the project's git repository (Bitbucket) and make it accessible to the team members and the PL teacher. This may require a previous decision on the programming language to be used.

- Study the problem's domain model and draft versions of the sequence diagrams for required use cases.

- Establish the UDP application protocol, including UDP requests and replies contents specification.