Computer Networks *(Redes de Computadores - RCOMP)* – 2025/2026

Laboratory Class 10 (PL10 – 2.5 hours)

UDP clients with timeout. UDP clients using broadcast. Implementing TCP clients and servers (C and Java).

1. Setting a timeout for UDP server's reply

In the previous lesson, UDP client and server applications were developed and tested, however, UDP is unreliable and that must be taken into account. When sending a UDP datagram, there is no guaranteed feedback under delivery point of view. Success in sending a UDP datagram doesn't mean anything about delivery, just that it was sent.

Under the client-server model (Figure 1):

- When idle, a UDP server-application is blocked at a receiving operation. Then, when a request arrives, it wakes up, processes the request, and finally sends back a reply.

- The UDP client application is most often under a user's direct control, so it will send a request when the user wishes. Next, the client application will be blocked at a receiving operation waiting for a server's reply. Finally, after receiving the reply, it presents the reply to the end user.

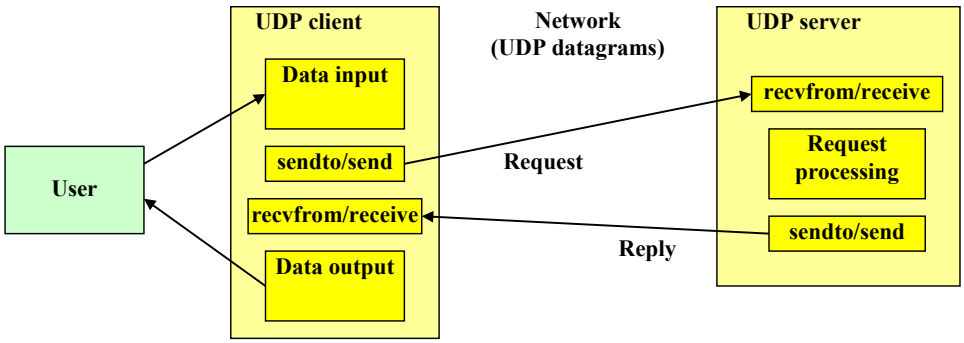

Figure 1 - UDP client/server interactions

However, a sent UDP datagram may be lost without any notification being given to the sender. Therefore, in UDP dialogs between clients and servers, either the request or the reply may never reach the destination and the sender will never know about that.

Under the UDP server-application's point of view, that is not as issue. It is not directly dependent on any delivery. If a request is lost, the server does not even know it has ever existed. If a reply is lost, that´s no concern for the server either, once it sends the reply its mission is finished.

On the other hand, on the UDP client application side, things get nasty. After sending the request, the client application becomes totally dependent on the arrival of a reply. What will happen to a UDP client when either the request is lost, or the reply is lost is that it gets blocked forever on a receiving operation, waiting for a server's reply that will never happen.

1.1. Test previous UDP client with delivery fail

To exhibit this issue, we can simply use one of the previous lesson's UDP client with no server application running on the other side (Figure 2).

Open one SSH session on **ssh3**.

Run the client using an unreachable server IP address:

./udp_cli 10.100.1.1

There is no reply to the first request and thus the client application gets blocked forever, it never returns from the **recvfrom()** function call.

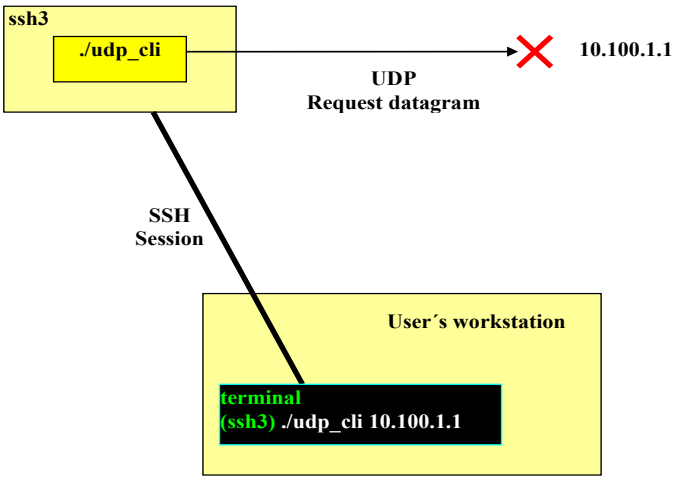

Figure 2 - UDP client sending a request with no server

1.2. Setting a read timeout

Solving this issue involves the UDP client only, the server keeps happily receiving requests (the ones that reach it) and sending replies (not concerned if they reach the client).

The strategy for the client is to avoid blocking forever, thus, it will establish a maximum time for the arrival of a reply, usually called a **timeout**, in this case, it is the server response timeout.

Both Java and C languages allow the setting of a timeout for socket operations, in Java by using the **setSoTimeout(int milliseconds)** method of the Socket class, in C by using the **setsockopt()** function, with the **SO_RCVTIMEO** option for receive timeout.

Setting a socket's timeout has the same effect in C and Java sockets: reading/receiving operations that would block until there is something to read/receive, will afterwards block only for up to the **timeout** value.

If the timeout expires without successfully reading/receiving data, the method or function will unblock with an error, in the case of Java, a **SocketTimeoutException** exception is raised, in the case of C the used function returns -1.

1.2.1. UDP client with server reply-timeout in Java language (UdpCliTo.java)

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

class UdpCliTo {
static InetAddress IPdestino;

private static int TIMEOUT=3;

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

 if(args.length!=1) {
        System.out.println("Server IPv4/IPv6 address or DNS name is required as argument");
        System.exit(1); }
```

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

 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
 DatagramSocket sock = new DatagramSocket();
 sock.setSoTimeout(1000*TIMEOUT); // set the socket timeout

 DatagramPacket udpPacket = new DatagramPacket(data, data.length, IPdestino, 9999);

 while(true) {
        System.out.print("Request 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);
        try {
                sock.receive(udpPacket);
                frase = new String( udpPacket.getData(), 0, udpPacket.getLength());
                System.out.println("Received reply: " + frase);
        } catch(SocketTimeoutException ex)
                {System.out.println("No reply from server");}
        }
 sock.close();
 }
}
```

1.2.2. UDP client with server reply-timeout in C language (udp_cli_to.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"

// server reply timeout in seconds
#define TIMEOUT 3

// 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;
       struct timeval to;

       if(argc!=2) {
              puts("Server IPv4/IPv6 address or DNS name is required as argument");
              exit(1);
              }

       bzero((char *)&req,sizeof(req));
       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;
        memcpy(&serverAddr,list->ai_addr,serverAddrLen);
        freeaddrinfo(list);

        bzero((char *)&req,sizeof(req));
        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);

        to.tv_sec = TIMEOUT;
        to.tv_usec = 0;
        setsockopt (sock,SOL_SOCKET,SO_RCVTIMEO,(char *)&to, sizeof(to));

        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);
               if(res>0) {
                      linha[res]=0; /* NULL terminate the string */
                      printf("Received reply: %s\n",linha);
                      }
               else
                       printf("No reply from server\n");
               }
        close(sock);   exit(0); }
```

1.2.3. Testing the new UDP client applications

Test the new UDP clients with the previous lesson's UDP servers.

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

Place **udp_srv** running in **ssh1**, then start **udp_cli_to** in **ssh3** (Figure 3).

./udp_cli_to 10.8.0.80

or

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

Test the new client when the server application is running, and when it's not (use CTLR+C to stop the

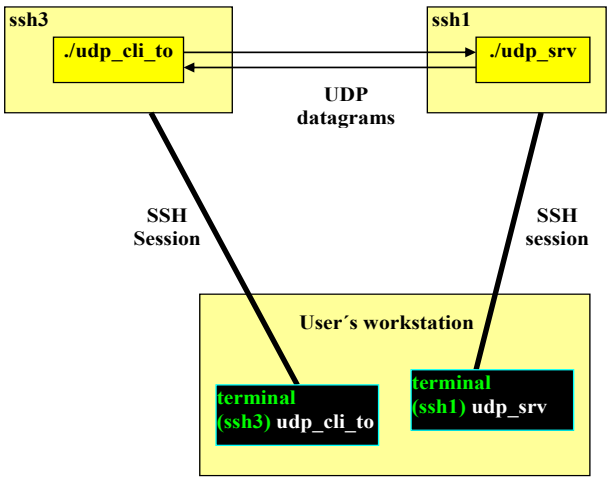

Figure 3 - UDP client and server on two SSH servers

server application). Now the client never gets blocked.

The same layout to test the Java version (Figure 4), on **ssh3**:

java UdpCliTo 10.8.0.80

or

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

Again, test the new client when the server application is running and when it's not (use CTLR+C to stop the server application). And again, now the client never gets blocked.

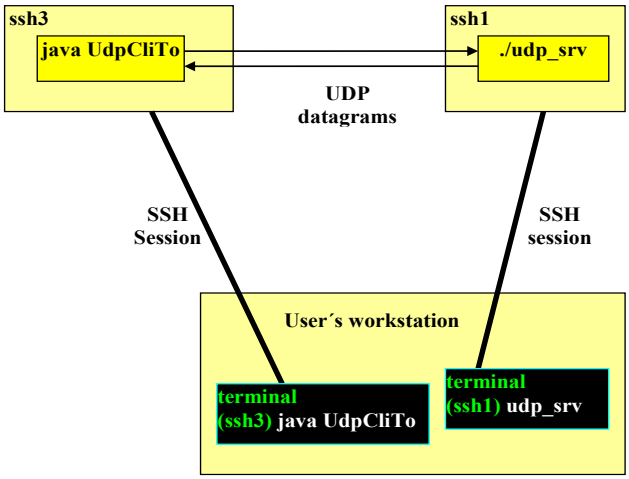

Figure 4 - UDP client/Java and server/C on two SSH servers

2. Using broadcast

UDP has significant disadvantages over TCP, most notably the total lack of reliability. Also, UDP is connectionless, this means data is sent in pieces, each transported by individual UDP datagrams, this may present a challenge when large volumes of data are to be transferred because UDP datagrams payload size should never be over 512 bytes).

Despite this, UDP has some advantages as well. First, it's very simple and thus with less overhead, UDP can be used to achieve a higher performance than TCP. This is true only as far as error rates are very low, for instance in a LAN, otherwise TCP is a better solution.

One feature available in UDP, and not in TCP, is sending to a broadcast or multicast address. These addresses represent sets of nodes, when a packet is sent to one of these addresses, all nodes belonging to that set will receive a packet's copy. Connection oriented protocols like TCP can't use this, they are designed for communications between two applications through a connection.

IGMP in IPv4 and ICMPv6 in IPv6 are used to manage multicast groups, namely, adding nodes to a multicast group and removing nodes from a multicast group. In IPv4 (not in IPv6) there´s a special multicast address known as the broadcast address; it represents all nodes belonging to an IPv4 network.

The main use of sending to a broadcast or multicast address is locating nodes in a network, if a UDP client application sends the first request to a broadcast address, whatever the server address is, as far as it's on same IPv4 network it will be reached, after receiving the first reply the UDP client application then known the server's address and next requests don't need to be sent to the broadcast address anymore.

Although, as you know, each IPv4 network has its own broadcast address, that should not be hardcoded into applications as it would only work on that specific IPv4 network. Instead, the generic IPv4 broadcast address should be used instead: **255.255.255.255**.

2.1. Enabling broadcast

The use of broadcast addresses with UDP sockets is quite straight, it is just a matter of setting the UDP datagram's destination address to a broadcast address. However, both in C and Java, sending to broadcast addresses is not allowed/enabled by default, prior to start sending, broadcast must be enabled.

In Java, the **DatagramSocket** method **setBroadcast(boolean on)** enables or disables broadcast, in C the **setsockopt()** function with the **SO_BROADCAST** option achieves the same goal.

2.2. Using broadcast in UDP clients

We will now use broadcast in our UDP client applications to locate the server application, instead of requiring the user to specify the server's address we will locate it by sending the first request to the broadcast address.

Once a reply to the first request is received, then we then know the server's address, so there is no point in keep sending to the broadcast address.

2.2.1. UDP broadcast client in C language (udp_cli_bcast.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"
#define BCAST_ADDRESS "255.255.255.255"

// 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, val, res, err;
	unsigned int serverAddrLen;
	char linha[BUF_SIZE];
	struct addrinfo req, *list;


	bzero((char *)&req,sizeof(req));
	// there's no broadcast address in IPv6, so we request an IPv4 address
	req.ai_family = AF_INET;
	req.ai_socktype = SOCK_DGRAM;
	err=getaddrinfo(BCAST_ADDRESS, SERVER_PORT , &req, &list);
	if(err) {
		printf("Failed to get broadcast address: %s\n",gai_strerror(err)); exit(1); }
	serverAddrLen=list->ai_addrlen;
	memcpy(&serverAddr,list->ai_addr,serverAddrLen);  // store the broadcast address for later
	freeaddrinfo(list);

	bzero((char *)&req,sizeof(req));
	req.ai_family = AF_INET;
	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);}

	// activate broadcast permission
	val=1; setsockopt(sock,SOL_SOCKET, SO_BROADCAST, &val, sizeof(val));

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

```
        perror("Bind failed");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);
        }
```

- We know the addresses used are IPv4, there is no broadcast in IPv6, so we request an IPv4 socket (AF_INET). The server will receive the request in IPv4; thus it will reply using IPv4.

- When the server reply is received, the reply source address (server's address) is stored in **serverAddr** and thus it will be used as destination address for the next request in the next loop.

2.2.2. UDP broadcast client in Java language (UdpCliBcast.java)

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

class UdpCliBcast {
        static InetAddress targetIP;

        public static void main(String args[]) throws Exception {
                byte[] data = new byte[300];
                String frase;
                targetIP=InetAddress.getByName("255.255.255.255");

                DatagramSocket sock = new DatagramSocket();
                sock.setBroadcast(true);
                DatagramPacket udpPacket = new DatagramPacket(data, data.length, targetIP, 9999);

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

                while(true) {
                        System.out.print("Request 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();
                }
        }
```

- When the server reply is received, the reply source address (server address and port number) is stored in the datagram, and thus, will be used as destination address for the next request. **Only the first request is sent to the broadcast address**.

2.3. Testing

For testing, start one of the previous lesson's UDP servers in one of the SSH servers.

Now, run the new UDP broadcast client in another SSH server, or in your personal workstation, as far as it is connected to the laboratories network it should work as well:

./udp_cli_bcast or **java UdpCliBcast**

As you can see, no server address is provided to the client application, nevertheless it's able to get a reply from the server application by sending the first request to the broadcast address.

2.4. Testing again

Now, **start more than one UDP server application** (of course at different SSH servers), and **while more than one server application is running**, start and test the broadcast UDP client application again:

./udp_cli_bcast or **java UdpCliBcast**

Send successive different request strings using the client application, and check the replies you get, what's happening?

Something we have not foreseen…

The problem is, when sending a request to a broadcast address several replies may arrive (one for each server on the network). However, our client application is reading one single reply for each sent request, so additional received replies are kept in the client's buffer.

After sending the second request, the client application reads a reply, but that's not the reply to the second request, instead, is a reply to the first request (kept in the buffer).

When using broadcast, this is a typical issue whenever there are several servers, and it must be addressed properly by the client application. Yet this issue may arise or not, depending on the number of servers in the network. Moreover, the number of replies we get for a request sent to the broadcast address is not predictable, it's the number of servers running on the network.

2.5. Solving the issue (multiple replies to a broadcast request)

Solving this issue is left for the student as extra classes work.

Tips about feasible alternative solutions:

- **Use a receive timeout**: after sending the first request do not read just one single reply, keep receiving all available replies until there are no more, and the timeout expires. Though setting the correct timeout to receive all replies may be tricky.

- **Check the replies source addresses**: when the first reply is received, store the source address, on next requests ignore replies coming from other addresses.

- **Send a special first request**: the first request is not user entered, instead it is a special application-defined string, and by doing so, on next requests (user entered) all replies containing the special string can be ignored. Don´t forget the special string will be mirrored by the server. Also, it must be ensured that, by great bad luck, the user will not type the exact same string, as defined by the application for the initial request.

3. TCP (Transmission Control Protocol)

TCP is a reliable connection-oriented transport protocol with automatic error correction. It establishes a dedicated communication channel (TCP connection) between a pair of applications. Through the TCP connection, data is sent in a continuous byte stream, preserving the data sequence, and guaranteeing data delivery.

Because is connection-oriented, prior to data transactions, a connection between two applications must be established. One of the two enrolled applications must take the initiative of emitting a connection request to the counterpart. In a client-server architecture, that role is therefore taken by the client.

Once the TCP connection is established between the two applications, data can be sent and receive simply by writing and reading bytes, however, these operations must be synchronised at the byte level.

This is rather different from UDP, where synchronisation is at datagram level. In UDP, a datagram sending operation in one application must match a datagram receiving operation in the counterpart, however, the application receiving a datagram is not required to specify how many bytes it will be receiving, just that is receiving a datagram, the number of bytes transported by the datagram will be known after receiving.

With TCP, a byte level synchronisation is required, this means the writing (sending) of N bytes in one application, must be matched by the reading (receiving) of exactly the same N bytes in the counterpart application.

If the number of bytes we try to read is less than those written, some will be unread and will appear in the next reading operation. If the number of bytes we try to read is greater than those written, then, the reading operation will block waiting for the missing bytes.

In TCP, synchronisation is a key feature to be settled by the **application protocol**, for the required byte level synchronisation on TCP connections, three approaches can be used:

a) Use a pre-agreed fixed number of bytes in each transaction, this way the reader always knows how many bytes it should read.

b) Before sending the data itself, send information about the number of bytes the data is made of. The reader starts by getting the data length, then it knows how many data bytes it should read next. This solution is used in HTTP protocol, where the message header has a *Content-Length* field indicating the number of bytes in the message's body.

c) Use a specific pre-agreed byte (or bytes sequence) as an end-of-data marker. Thus, the receiver must then read one byte at a time, and check if it's the end-of-data marker. This solution is also used in HTTP, the CR+LF sequence is used to mark the end of each header's line, also, the CR+LF+CR+LF sequence (an empty line) is used to mark the header's end.

This last alternative is easy to implement if data is made of a limited set of possible byte values, like with ASCII text. If data bytes are allowed to have any value, additional processing will be required, namely, any mark value occurring in data will have to be masked on the sender and unmasked on the receiver to avoid being wrongly interpreted.

4. Using TCP with Berkeley Sockets – C Language

4.1. TCP connection establishment

As mentioned before, to establish a TCP connection two applications must assume different roles, therefore each will use a different function to perform its specific role.

int connect(int socket, struct sockaddr *address, int address_len);

connect() – is called by the application wishing to create the TCP connection (the TCP client), for successful completion, this must match an ***accept()*** function call on the counterpart. The connect() function receives as argument a pointer to a caller-defined structure with the server's IP address and port number, to where the connection request will be sent. If an error occurs it returns -1, otherwise the TCP connection is established, and the socket is then connected to the counterpart.

int accept(int socket, struct sockaddr *address, int *addrlen);

accept() – is called by the application wishing to receive a TCP connection request (the TCP server), this is a **blocking function**, if when called, there are no pending connection requests it will wait until one arrives. If **address** is not NULL it will be used to store the counterpart's IP address and port number, if so the value in **addrlen** must be defined by the caller otherwise it can also be NULL.

When the accept() function unblocks with no error (on error it returns -1), the TCP connection is established and **a new socket is returned**. The new socket, returned by accept(), is connected to the counterpart. The original socket, used in the accept() function call, is kept open and available for receiving other TCP connection requests from other clients.

4.2. Sending and receiving data (reading and writing)

Once the TCP connection is established, sending and receiving data can be accomplished by using the standard ***write()*** and ***read()*** functions over the connected socket (Figure 5). Data is sent and received in a continuous byte stream, synchronization is required when establishing the connection (connect/accept), and afterwards when transferring data (write/read).

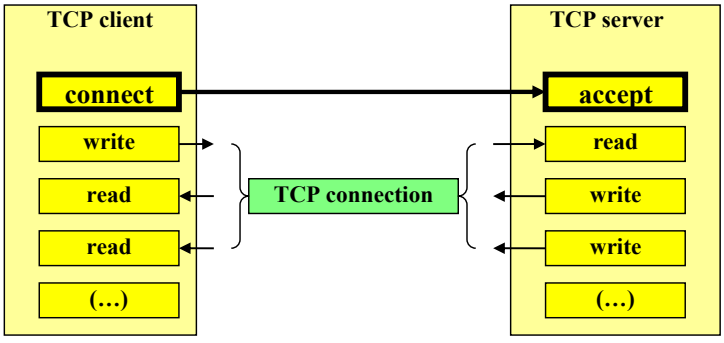

Figure 5 - Connect/accept and write/read synchronization in TCP (C language)

4.3. Multi-process TCP servers

A TCP server job is complex because it must be always available to accept new incoming connection requests, on one hand, and at the same time it must read incoming requests from already connected clients, in every already connected socket. Each time the **accept()** function returns with success, there is one more connected socket for the server application to handle with. One way to solve this issue is by creating a parallel sub-task (e.g., process or thread) for each socket.

In the following typical implementation layout, the parent process keeps calling the **accept()** function. For each established connection, it creates a child process with the purpose of handling the connected client requests on the new socket.

```
for(;;) {
      newSock=accept(sock, …);
      if(!fork()) {            // child process
            close(sock);
            /* process all client requests on newSock */
            close(newSock);
            exit(0);           // child exits
            }
      close(newSock);          // parent process
      }
```

Using this type of solution has some advantages, each client has an independent process dedicated to it, thus interferences between sessions of different clients are unlike, on the flip side, if some type of interaction between clients' sessions is required, then IPC (Inter Process Communication) will have to be used.

4.4. Port numbers and pending requests queue

As usual for clients, a TCP client doesn't need a fixed local port number, in fact, for TCP clients, binding is optional. If when the **connect()** function is called the socket is not bound, it will be bound to one local free port number.

Again, as usual for servers, the TCP server must use a fixed local port number so that clients know to where they should send the connection request.

Additionally, after binding to a fixed local port number, the TCP server must also set the size of the pending connections requests queue (pending stands for not yet accepted), the maximum possible size is defined by ***SOMAXCONN***.

A code example for a TCP server socket setup is:

```
bind(sock, …);
listen(sock,SOMAXCONN);
newSock=accept(sock, …);
```

5. Using TCP with Berkeley Sockets – Java Language

5.1. TCP connection establishment

In Java, there's a specific class for TCP connections requests reception: the ***ServerSocket*** class. One of the constructors receives the local port number where connection requests will be received:

public ServerSocket(int port) throws IOException

Of course, this class will be used by the TCP server application.

The TCP client takes the initiative of sending a connection request to the server by instantiating the **Socket** class, one of its constructors receives an IP address and a port number:

public Socket(InetAddress address, int port) throws IOException

The connection establishment will be successful if on the specified IP address there's a TCP server application using the specified local port number and calls the *accept()* method of the *ServerSocket* class. As with C language, the *accept()* method is blocking and waits for the next connection request. On success, the *accept()* method return a new socket connected to the client, in this case it will be a Socket class object.

5.2. Sending and receiving data (reading and writing)

Like in C language, after the connection has been established sending and receiving data can be accomplished using the *write()* and *read()* methods, respectively. In the case of Java, not directly over the connected socket, but over the connected socket's ***OutputStream*** and ***InputStream***, respectively.

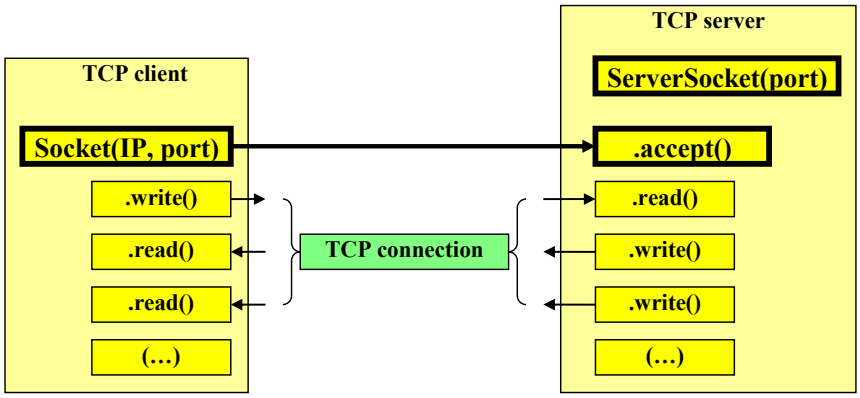

Figure 6 - Connect/accept and write/read synchronization in TCP (Java language)

5.3. Multi-thread TCP servers

A TCP server in Java has the same complex job to perform as in C language, again, parallel sub-tasks can be used, and for Java we will use threads.

For each new accepted connection, a new thread will be started to handle it:

```
ServerSocket sock = ServerSocket(PORT);
Socket cliSock;
while(true) {
    cliSock=sock.accept();
    new Thread(new tcp_client_thread(cliSock)).start();
    }
```

6. Implementing a simple TCP client and server

Create a TCP client and a TCP server with the following features and **application protocol specification**:

- The server IP address (IPv4, IPv6 or DNS name) is provided to the client as the first argument at the command line.

- Once connected, the client sends a list of integer numbers terminated with the zero value.

- The server accepts TCP connections from clients. For logging, each new connection and disconnection should be presented at the server console, showing the client IP address and port number.

- The server calculates the sum of the sent integers and sends back the result.

- When the client wants to exit it should send an empty list (started by the zero value).

- Each integer is sent as a sequence of 4 bytes in order of increasing significance, i.e., first the LSB (Least Significant Byte) and last the MSB (Most Significant Byte).

So, the sequence of bytes A, B, C, D represents the number given by:

NUMBER = A + 256xB + 256x256xC + 256x256x256xD

For instance:

The number 10 is sent as the sequence of bytes: 10, 0, 0, 0

The number 300 is sent as the sequence of bytes: 44, 1, 0, 0

This might look an odd way of sending an integer number. The problem is, directly sending integer numbers as they are stored in the local host's memory is not an option because they can be stored differently in the source host and destination host, for instance when a Java client is sending to a C server.

Of course, one other often used option to send data in an implementation independent representation, is by sending the humanly readable representation of data. This application protocol could specify that each integer is sent in the form of its text representation.

This is a good option because all programming APIs have function to parse most data types from textual representations into local storing and also functions to produce textual representations from local storing.

6.1. The TCP client in C language (*tcp_cli_sum.c*)

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

#define BUF_SIZE 30
#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) {
       int err, sock;
       unsigned long f, i, n, num;
       unsigned char bt;
       char linha[BUF_SIZE];
       struct addrinfo  req, *list;

       if(argc!=2) {
              puts("Server's 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_STREAM;
       err=getaddrinfo(argv[1], SERVER_PORT , &req, &list);
       if(err) {
              printf("Failed to get server 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(connect(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
```

```
                perror("Failed connect"); freeaddrinfo(list); close(sock); exit(1);}

        do {
                do {
                        printf("Enter a positive integer to SUM (zero to terminate): ");
                        GETS(linha,BUF_SIZE);
                        while(sscanf(linha,"%li",&num)!=1 || num<0) {
                                puts("Invalid number");
                                GETS(linha,BUF_SIZE);
                                }
                        n=num;
                        for(i=0;i<4;i++) {
                                bt=n%256; write(sock,&bt,1); n=n/256; }
                        }
                while(num);
                num=0; f=1; for(i=0;i<4;i++) {read(sock,&bt,1); num=num+bt*f; f=f*256;}
                printf("SUM RESULT=%lu\n",num);
                }
        while(num);
        close(sock);
        exit(0);
        }
```

To create the appropriate socket, the same tactic as with previous UDP clients is used: let getaddrinfo() determine the address family of the provided server address, and then, create the local socket accordingly.

Nevertheless, other strategies are possible to support both IPv4 and IPv6 addresses, one would be, using always an IPv6 socket. The issue when using an IPv6 socket is it can't handle an IPv4 server address. The solution would require determining if the server's address is IPv4, and in that case transform the IPv4 address into IPv4-Mapped (A.B.C.D -> ::ffff:A.B.C.D).

Back to the example code, the SOCK_STREAM (TCP) socket is then created using the values provided by **getaddrinfo()** and the TCP connection is established to the server node IP address and port number. If the connection establishment fails (for instance because the server is not running), then **connect()** returns -1.

For each user-entered integer number, the four bytes representing it are sent to the server, when the number sent is zero (end of the list), the server is supposed to send back a reply, also as an integer in the form of 4 bytes.

If the reply is zero, this means we have sent an empty list and want to exit, so we close the socket, and thus the TCP connection.

6.2. TCP client in Java language (*TcpCliSum.java*)

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

class TcpCliSum {
        static InetAddress serverIP;   static Socket sock;
        public static void main(String args[]) throws Exception {
                if(args.length!=1) {
                        System.out.println("Server IPv4/IPv6 address or DNS name is required");
                        System.exit(1); }
                try { serverIP = InetAddress.getByName(args[0]); }
                catch(UnknownHostException ex) {
                        System.out.println("Invalid server specified: " + args[0]);
                        System.exit(1); }
                try { sock = new Socket(serverIP, 9999); }
                catch(IOException ex) {
                        System.out.println("Failed to establish TCP connection");
                        System.exit(1); }
                BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
                DataOutputStream sOut = new DataOutputStream(sock.getOutputStream());
                DataInputStream sIn = new DataInputStream(sock.getInputStream());
```

```
            String frase; long f,i,n,num;
            do {
                  do {
                        num=-1;
                        while(num<0) {
                              System.out.print(
                                  "Enter a positive integer to SUM (zero to terminate): ");
                              frase = in.readLine();
                              try { num=Integer.parseInt(frase); }
                              catch(NumberFormatException ex) {num=-1;}
                              if(num<0) System.out.println("Invalid number");
                              }
                        n=num; for(i=0;i<4;i++) {sOut.write((byte)(n%256)); n=n/256; }
                        }
                  while(num!=0);
                  num=0; f=1;
                  for(i=0;i<4;i++) {num=num+f*sIn.read(); f=f*256;}
                  System.out.println("SUM RESULT = " + num);
                  }
            while(num!=0);
            sock.close();
            }
      }
```

It's basically similar to the C language implementation, the TCP connection is established by instantiating a Socket class object specifying to the constructor the server's IP address and port number.

For reading and writing through the established TCP connection, the connected socket's InputStream and OutputStream are required.

6.3. TCP server in C language (*tcp_srv_sum.c*) – Unix Multi-Process

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.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 from;
      int err, newSock, sock;
      unsigned int adl;
      unsigned long i, f, n, num, sum;
      unsigned char bt;
      char cliIPtext[BUF_SIZE], cliPortText[BUF_SIZE];
      struct addrinfo  req, *list;

      bzero((char *)&req,sizeof(req));
      // requesting a IPv6 local address will allow both IPv4 and IPv6 clients to use it
      req.ai_family = AF_INET6;
      req.ai_socktype = SOCK_STREAM;
      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);
      listen(sock,SOMAXCONN);
      puts("Accepting TCP connections (IPv6/IPv4). Use CTRL+C to terminate the server");
```

```
	adl=sizeof(from);
	for(;;) {
		newSock=accept(sock,(struct sockaddr *)&from,&adl);
		if(!fork()) {
			close(sock);
			getnameinfo((struct sockaddr *)&from,adl,cliIPtext,BUF_SIZE,
				cliPortText,BUF_SIZE, NI_NUMERICHOST|NI_NUMERICSERV);
			printf("New connection from %s, port number %s\n", cliIPtext, cliPortText);
			do {
				sum=0;
				do {
					num=0;f=1;
					for(i=0;i<4;i++) {
						read(newSock,&bt,1); num=num+bt*f; f=256*f; }
						sum=sum+num;}
				while(num);
				n=sum;
				for(i=0;i<4;i++) {
					bt=n%256; write(newSock,&bt,1); n=n/256; }
				}
			while(sum);
			close(newSock);
			printf("Connection %s, port number %s closed\n", cliIPtext, cliPortText);
			exit(0);
			}
		close(newSock);
		}
	close(sock);
	}
```

Using an IPv6 socket will allow both IPv4 and IPv6, however, IPv4 addresses will be handled as IPv4-Mapped.

After opening the SOCK_STREAM (TCP) socket, it's bound to the local address (including the fixed port number), and the pending connections request queue size is defined (listen).

The main (infinite) loop calls **accept()**, when it unblocks, it returns a new socket connected to the client (**newSock**), **fork()** is then used to create a child process to handle the client's requests on **newSock**, meanwhile the parent process calls accept() again for additional clients.

6.4. TCP server in Java Language (*TcpSrvSum.java*) – Multi-Thread

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

class TcpSrvSum {
	static ServerSocket sock;

	public static void main(String args[]) throws Exception {
		Socket cliSock;
		try { sock = new ServerSocket(9999); }
		catch(IOException ex) {
			System.out.println("Failed to open server socket"); System.exit(1);
			}
		while(true) {
			cliSock=sock.accept();
			new Thread(new TcpSrvSumThread(cliSock)).start();
			}
		}
	}

class TcpSrvSumThread implements Runnable {
	private Socket s;
	private DataOutputStream sOut;
	private DataInputStream sIn;

	public TcpSrvSumThread(Socket cli_s) { s=cli_s;}
	public void run() {
```

```
            long f,i,num,sum;
            InetAddress clientIP;
            clientIP=s.getInetAddress();
            System.out.println("New client connection from " + clientIP.getHostAddress() +
                   ", port number " + s.getPort());
            try {
                   sOut = new DataOutputStream(s.getOutputStream());
                   sIn = new DataInputStream(s.getInputStream());
                   do {
                          sum=0;
                          do {
                                 num=0; f=1; for(i=0;i<4;i++) {num=num+f*sIn.read(); f=f*256;}
                                 sum=sum+num;
                                 }
                          while(num>0);
                          num=sum;
                          for(i=0;i<4;i++) {sOut.write((byte)(num%256)); num=num/256; }
                          }
                   while(sum>0);

                   System.out.println("Client " + clientIP.getHostAddress() +
                          ", port number: " + s.getPort() + " disconnected");
                   s.close();
                   }
            catch(IOException ex) { System.out.println("IOException"); }
            }
      }
```

Two classes are defined in the same source file, the application itself on class **TCPSrvSum** implementing the **main()** method, and the **TcpSrvSumThread** class defines a thread to be created for each connected client.

A **ServerSocket** class object is instantiated, the constructor receives the local port number where TCP connections are to be received. Then the **accept()** method is called in a loop, for each established connection a thread is created and started.

6.5. Applications testing

- After changing the port numbers in both the source files (client and server), place one server application running on a node.

- In another node, use the client application to connect to the server, test it by sending a zero-terminated sequence of integer numbers.

- Test the server application with several clients connected.

- Test both with C and Java versions.

Additional exercise - extra classes work.

Implement the required changes to one of the example TCP servers (C or Java) in order to add the following feature:

- The server application receives as command line arguments, a list of IP addresses.

- When a connection request is accepted, the server checks if the client's address is on the list, if so, proceeds normally, and otherwise closes the connection (refuses the unknown client).

- In either case, at the server console, a message should indicate if the client access was granted or not.

Note: in the case of the C language version of the server, one additional issue arises: an IPv6 socket is used, therefore IPv4 client addresses will appear as IPv4-Mapped format.