Redes de Computadores (RCOMP) – 2017/2018

Laboratory Class Script PL14

- Network applications development over Berkeley Sockets.
- UDP clients with timeout.
- UDP clients using broadcast.

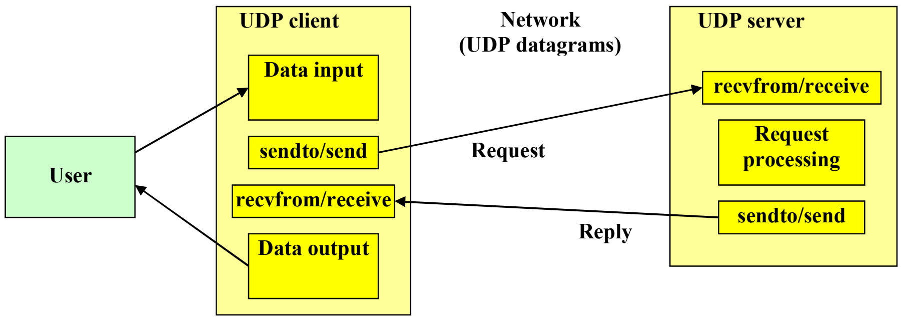
1. Setting a timeout for UDP server's reply

- 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's 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.

- When idle, a UDP server-application is blocked on 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 will be blocked on a receiving operation waiting for a server's reply. Finally, after receiving the reply, it presents the reply to the end user.



- A sent UDP datagram may be lost without any notification to the sender. Therefore, within UDP dialogs between clients and servers both the request and 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's not a problem. It's not directly dependent on any delivery. If a request is lost, the server doesn't even know it ever existed. If a reply is lost, that´s no concern for the server either, once it sends the reply its mission is finished.

- Conversely, on the UDP client-application's side, things get complicated. After sending the request, the client-application becomes totally dependent on a reply arrival.

- What will happen to a UDP client when either the request or the reply are lost is that it will get blocked forever on a receiving operation waiting for a server's reply that will never arrive.

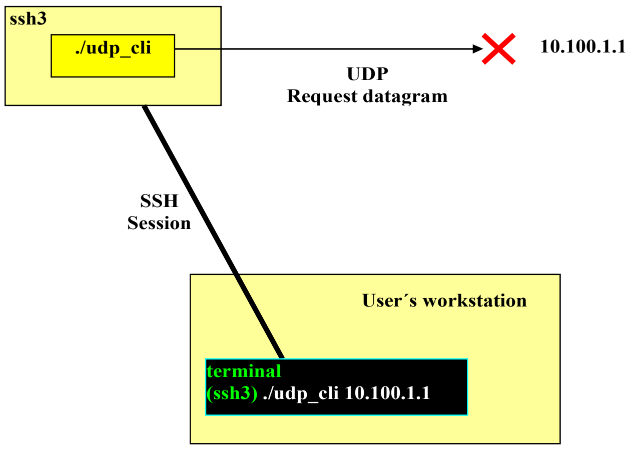
1.1. Test previous UDP client with delivery fail

To exhibit this problem simply use one of the previous lesson's UDP clients with no server application running.

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.

1.2. Setting a read timeout

Solving this issue may involve the UDP client only, then 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 avoiding blocking forever, it will establish a maximum time for reply arrival, usually called a timeout, in this case, the server's reply timeout.

Both Java and C languages allow setting a timeout for socket operations, in Java by using the **setSoTimeout(int milliseconds)** 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's 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 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;
```

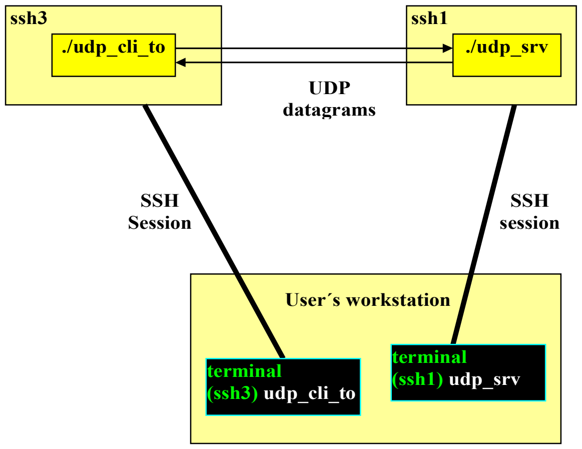
```
        req.ai_socktype = SOCK_DGRAM;
        err=getaddrinfo(argv[1], SERVER_PORT , &req, &list);
        if(err) {
               printf("Failed to get server address, error: %s\n",gai_strerror(err)); exit(1);
               }
        serverAddrLen=list->ai_addrlen;
        memcpy(&serverAddr,list->ai_addr,serverAddrLen);
        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 other in **ssh3**.

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



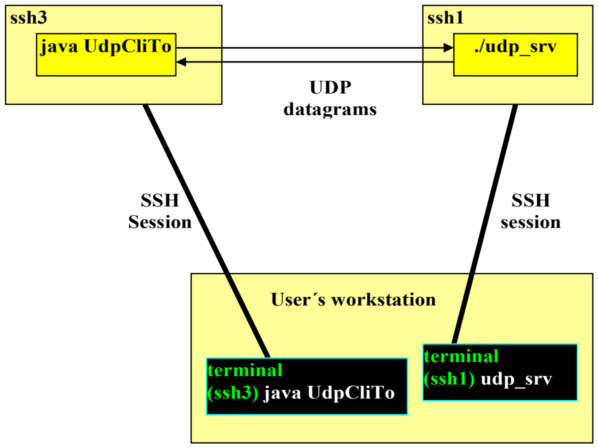
```
./udp_cli_to 10.8.0.80
```

or

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

Test the new client when the server application is running and when is not (use CTLR+C to stop the server application). Now the client never gets blocked.

The same layout to test the Java version, 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 is not (use CTLR+C to stop the server application). And again, now the client never gets blocked.

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 shouldn't 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.

Moreover, one feature available in UDP and not in TCP is sending to a broadcast or multicast address. These addresses represent sets of nodes, if 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 is on same IPv4 network it will be reached, after receiving the 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 shouldn't be hard coded into applications as it would only work on one specific IPv4 network. Instead, the generic IPv4 broadcast address should be used: **255.255.255.255**.

2.1. Enabling broadcast

The use of broadcast addresses with UDP sockets is quite straight, it's 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 enabled by default, prior to start sending broadcast has to 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 we then know the server's address, so there's 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, even because 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 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 it will be used as destination address for the next request. **Only the first request is sent to the broadcast address.**

2.3. Testing

For the purpose of testing start one of the previous lesson 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:

./udp_cli_bcast or **java UdpCliBcast**

As you can see no server address is provided to the client application, nevertheless it is 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 in 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, what is 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 request sent, so other received replies are kept in the buffer.

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

When using broadcast this is a typical issue and must be handled properly by the client application.

Solving this issue is a job for the students in the remaining class time.

Tips about possible alternative solutions:

- **Use receive timeout:** after sending the first request don't receive just a single reply, keep receiving all available replies until there are no more and the timeout expires.

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

- **Send a flag on the first request:** the first request is not user entered, instead is a special application-defined string, 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.