**Lab3**

*Last modified*: 26/10/2022 10:20

Use UDP to implement the problems from [Lab1](https://www.cs.ubbcluj.ro/~dadi/compnet/labs/lab1) and [Lab2](https://www.cs.ubbcluj.ro/~dadi/compnet/labs/lab2) and the following problems. Think of a way to write concurrent UDP servers. Use examples bellow as guidance.

[Example �UDP](https://www.cs.ubbcluj.ro/~dadi/compnet/labs/lab3/udp-example1-c.html)

[Example � UDP Broadcast](https://www.cs.ubbcluj.ro/~dadi/compnet/labs/lab3/udp-broadcast.html)

[Example SELECT� Multiplexed TCP Client-Server Chat application](https://www.cs.ubbcluj.ro/~dadi/compnet/labs/lab3/multiplexed_chat.html)

**Note:** Run the server and the client on different hosts. For this you need to know the IP address of the machine where the server runs (ifconfig/ipconfig)

**Note:**Learn to use Wireshark in order to inspect network traffic.

1.   The client sends periodical PING datagrams with a random content to a <server> and <port> specified in command line. The server returns back (echoes) the same packets (content). The client checks the content of the received packets to match what was sent and computes the round trip time and displays it to the user � for each sent packet.

2.   Implement the [Python concurrent example from lab2](https://www.cs.ubbcluj.ro/~dadi/compnet/labs/lab2/example-python-numbergguess-concurrent-threads.html) to work with UDP. You no longer need threads to be able to serve multiple clients � so the implementation should be much shorter.

3.   Implement the Chat server example (see the link bellow) using UDP and TCP �only this time each client should contact the server just for registration. All communication happens directly between the peers (clients) without passing trough the server. Each client maintains an endpoint (TCP/UDP) with the server just for learning the arrival/departure of other clients. You create a mesh architecture where all clients connect directly between each others.

4.      Write an UDP broadcast application that serves as client and server at the same time. The application is started with the network broadcast address (<NBCAST>) as argument in the command line.

1.       Upon launching the application listens on UDP port 7777.

2.    Every 3 seconds the application sends a UDP broadcast message to NBCAST port 7777 with the format: TIMEQUERY\0 (string)

3.    Whenever the application receives a TIMEQUERY demand it answers to the source IP:port with a string message: TIME HH:MM:SS\0 (current time) using unicast.

4.    Every 10 seconds the application sends a UDP broadcast message to NBCAST port 7777 with the format:� DATEQUERY\0 (string)

5.    Whenever the application receives a DATEQUERY demand it answers to the source IP:port with a string message: DATE DD:MM:YYYY\0 (current date) using unicast.

6.    The application will keep a list of peers (that answer to broadcast � IP:portno) and update the information anytime a unicast message is received upon a broadcast.

7.    When an entry in a list does not have any answer for 3 consecutive broadcasts it will be removed from the list.

8.    The list will be displayed (ip,date, time) on the screen upon each update (using a screen positioning api like conio or by erasing the screen before each update).

9.    Every malformed request/response received will be counted and displayed at the end of a screen update. You will have a list of malformed messages displayed with their source IP address. The list should be limited in size and implemented as a queue. Recent messages are added to the head and old messages are moving towards the tail.

**Note:**Suggestion: Implement the application on Windows, or run it on your laptop in order to be able (all of you simultaneously) to listen on port 7777. Your application should strictly follow the protocol and be able to interact with all applications written by your colleagues.

**Note:**On Windows in order to have timer like events (periodical events handled) use **timeSetEvent** or a similar function (the newer  **[CreateTimerQueueTimer](https://msdn.microsoft.com/en-us/library/ms682485(v=vs.85))**) and set a different callback function for each type of event.

**Note:** Sending broadcast UDP requires a *setsockopt(sock,SOL\_SOCKET,SO\_BROADCAST,&broadcast,sizeof(broadcast)* as in example.

Receiving broadcast usually doesn�t require any additional effort compared to a normal UDP application. If not able to receive broadcast on windows try to *setsockopt*�on the receiving socket as well.

**Note (Malformed traffic):**To generate malformed traffic one could use the **nc** (network cat) command on a linux like system as it follows:

*nc -4 �u <dstip> <dstport>*

�and type in anything until CTRD+D is pressed ! Anything typed in will be sent to the the remote IP and port using the specified protocol (u=udp)

or

*<command> | nc -4 �u 172.30.5.16 7777*

**Example of displayed results:**

172.30.0.4:7777���������������������������������� TIME 16:51:43�� DATE 08:04:2014

172.30.118.185:7777�������������������������� TIME 16:51:47�� DATE 08:04:2014

172.30.116.220:7777�������������������������� TIME 16:51:44�� DATE 08:04:2014

172.30.112.50:7777���������������������������� TIME 16:51:42�� DATE 08:04:2014

Malformed Data:

172.30.118.185:7777 - HELLO

172.30.118.185:7777 - HELLO

172.30.118.185:7777 - HELLO

172.30.118.185:7777 - HELLO

172.30.118.185:7777 - HELLO

172.30.116.220:7777 - asdasdDATE 08:04:201

172.30.116.220:7777 - asdasdTIME 16:30:50

172.30.116.220:7777 - asdasdTIME 16:30:52

172.30.116.220:7777 - asdasdTIME 16:30:56

172.30.116.220:7777 - asdasdTIME 16:30:59

�

UDP � Example � sending point to point messages

UDP allows sending individual packets over the IP protocol (just process demultiplication is done) to different applications running on different machines. Unlike TCP it does not support connections and streams. Each sendto operation

needs to be paired with a recvfrom.

|  |  |
| --- | --- |
| **UDP client** | **UDP Server** |
| /\* UDP client in the internet domain \*/  #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <netdb.h>  #include <stdio.h>    void error(char \*);  int main(int argc, char \*argv[])  {  �� int sock, length, n;  �� struct sockaddr\_in server, from; // IP Addressing(ip, port, type) Stuff  �� struct hostent \*hp; // DNS stuff  �� char buffer[256];    �� if (argc != 3) { printf("Usage: %s <server\_name> <port>\n",argv[0]);  ������������������� exit(1);  �� }  �� sock= socket(AF\_INET, SOCK\_DGRAM, 0);  �� if (sock < 0) error("socket");    �� server.sin\_family = AF\_INET;  �� hp = gethostbyname(argv[1]);  �� if (hp==0) error("Unknown host");    �� bcopy((char \*)hp->h\_addr, (char \*)&server.sin\_addr, hp->h\_length);  �� server.sin\_port = htons(atoi(argv[2]));  �� length=sizeof(struct sockaddr\_in);  ��  �� printf("Please enter the message: ");  �� bzero(buffer,256);  �� fgets(buffer,255,stdin);  �� n=sendto(sock,buffer,strlen(buffer),0,&server,length);  �� if (n < 0) error("Sendto");  ��  �� n = recvfrom(sock,buffer,256,0,&from, &length);  �� if (n < 0) error("recvfrom");  �� write(1,"Got an ack: ",12);  �� write(1,buffer,n);  }    void error(char \*msg)  {  ��� perror(msg);  ��� exit(0);  } | /\* Creates a datagram server.� The port  �� number is passed as an argument.� This  �� server runs forever \*/    #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <netdb.h>  #include <stdio.h>    void error(char \*msg)  {  ��� perror(msg);  ��� exit(0);  }    int main(int argc, char \*argv[])  {  �� int sock, length, fromlen, n;  �� struct sockaddr\_in server;  �� struct sockaddr\_in from;  �� char buf[1024];    � �if (argc < 2) {  ����� fprintf(stderr, "ERROR, no port provided\n");  ����� exit(0);  �� }  �  �� sock=socket(AF\_INET, SOCK\_DGRAM, 0);  �� if (sock < 0) error("Opening socket");  �� length = sizeof(server);  �� bzero(&server,length);  �� server.sin\_family=AF\_INET;  �� server.sin\_addr.s\_addr=INADDR\_ANY;  �� server.sin\_port=htons(atoi(argv[1]));    �� if (bind(sock,(struct sockaddr \*)&server,length)<0)  ������ error("binding");    �� fromlen = sizeof(struct sockaddr\_in);  �� while (1) {  ������ n = recvfrom(sock,buf,1024,0,(struct sockaddr \*)&from,&fromlen);  ������ if (n < 0) error("recvfrom");  ������ write(1,"Received a datagram: ",21);  ������ write(1,buf,n);  ������ n = sendto(sock,"Got your message\n",17,  ����������������� 0,(struct sockaddr \*)&from,fromlen);  ������ if (n� < 0) error("sendto");  �� }  �} |

UDP � Broadcast

UDP broadcast is a technique that allows sending UDP datagram from a single source to all computers in a LAN. In order to send a UDP datagram addressed to all computers in the local area network it needs to be sent to a special address called the **Broadcast address.**The broadcast address for a LAN is either the highest address in the local subnetwork or the universal broadcast address: 255.255.255.255. In order to receive broadcast messages the receiver needs to enable **SO\_BROADCAST** option on the socket (see bellow)

|  |  |
| --- | --- |
| **UDP Broadcast Sender** | **UDP Receiver** |
| #include"winsock2.h"  #include<iostream>  #include<conio.h>  �  using namespace std;  �  #define MYPORT 9009��� // the port users will be connecting to  �  �  int main()  {  �� �WSADATA wsaData;  ��� WSAStartup(MAKEWORD(2,2), &wsaData);  �  ��� SOCKET sock;  ��� sock = socket(AF\_INET,SOCK\_DGRAM,0);  �  ��� char broadcast = '1';  �  ��� if(setsockopt(sock,SOL\_SOCKET,SO\_BROADCAST,&broadcast,sizeof(broadcast)) < 0)  ��� {  ������� cout<<"Error in setting Broadcast option";  ������� closesocket(sock);  ������� return 0;  ��� }  �  ��� struct sockaddr\_in Recv\_addr;��  ��� struct sockaddr\_in Sender\_addr;  �  ��� int len = sizeof(struct sockaddr\_in);  �  ��� char sendMSG[] ="Broadcast message from SLAVE TAG";  �  ��� char recvbuff[50] = "";  ��� int recvbufflen = 50;  �  ��� Recv\_addr.sin\_family������ = AF\_INET;��������  ��� Recv\_addr.sin\_port�������� = htons(MYPORT);���  //� Recv\_addr.sin\_addr.s\_addr� = INADDR\_BROADCAST; // this isq equiv to 255.255.255.255  // better use subnet broadcast (for our subnet is 172.30.255.255)  ��� Recv\_addr.sin\_addr.s\_addr = inet\_addr("172.30.255.255");  �  �  ��� sendto(sock,sendMSG,strlen(sendMSG)+1,0,(sockaddr \*)&Recv\_addr,sizeof(Recv\_addr));  �  �  ��� recvfrom(sock,recvbuff,recvbufflen,0,(sockaddr \*)&Recv\_addr,&len);  �  ��� cout<<"\n\n\tReceived message from Reader is =>� "<<recvbuff;  �  ���� cout<<"\n\n\tpress any key to CONT...";  ��� \_getch();  �  ��� closesocket(sock);  ��� WSACleanup();  }  � | #include"winsock2.h"  #include<iostream>  #include<conio.h>  �  using namespace std;  �  #define MYPORT 9009��� // the port users will be connecting to  �  �  int main()  {  ��� WSADATA wsaData;  ��� WSAStartup(MAKEWORD(2,2), &wsaData);  �  ��� SOCKET sock;  ��� sock = socket(AF\_INET,SOCK\_DGRAM,0);  �  ��� char broadcast = '1';  �  //���� This option is needed on the socket in order to be able to receive broadcast messages  //�� If not set the receiver will not receive broadcast messages in the local network.  ��� **if(setsockopt(sock,SOL\_SOCKET,SO\_BROADCAST,&broadcast,sizeof(broadcast)) < 0)**  ��� {  ������� cout<<"Error in setting Broadcast option";  ������� closesocket(sock);  ������� return 0;  ��� }  �  ��� struct sockaddr\_in Recv\_addr;��  ��� struct sockaddr\_in Sender\_addr;  �  ��� int len = sizeof(struct sockaddr\_in);  �  ��� char recvbuff[50];  ��� int recvbufflen = 50;  ��� char sendMSG[]= "Broadcast message from READER";  �  ��� Recv\_addr.sin\_family������ = AF\_INET;��������  ��� Recv\_addr.sin\_port�������� = htons(MYPORT);���  ��� Recv\_addr.sin\_addr.s\_addr� = INADDR\_ANY;  �  ��� if (bind(sock,(sockaddr\*)&Recv\_addr, sizeof (Recv\_addr)) < 0)  ��� {  ������� cout<<"Error in BINDING"<<WSAGetLastError();  ������� \_getch();  ������� closesocket(sock);  ������� return 0;  ��� }  �  ��� recvfrom(sock,recvbuff,recvbufflen,0,(sockaddr \*)&Sender\_addr,&len);  �  ��� cout<<"\n\n\tReceived Message is : "<<recvbuff;  ��� cout<<"\n\n\tPress Any to send message";  ��� \_getch();  �  �  ��� if(sendto(sock,sendMSG,strlen(sendMSG)+1,0,(sockaddr \*)&Sender\_addr,sizeof(Sender\_addr)) < 0)  ��� {  ������� cout<<"Error in Sending."<<WSAGetLastError();  ������� cout<<"\n\n\t\t Press any key to continue....";  ������� \_getch();  ������� closesocket(sock);  ������� return 0;  ��� }  ��� else  ������� cout<<"\n\n\n\tREADER sends the broadcast message Successfully";  �  �  ��� cout<<"\n\n\tpress any key to CONT...";  ��� \_getch();  �  ��� closesocket(sock);  ��� WSACleanup();  } |

Multiplexed Chat Server � using select

This example presents an implementation of a client-server chat application working in the command line in an IRC style application with only one room. The server accepts multiple TCP clients and relays messages passed by each client to all other connected clients. The client needs to be able to:

�       read messages from standard input and pass them to the server;

�       read messages relayed from the server and display them to the user;

The motivation of using select lies here in the fact that there are multiple descriptors that need to be watched by both the client and the server. The server also needs a shared state approach where creating a new process to handle each client would only make things complicated.

|  |  |
| --- | --- |
| **Multiplexed chat SERVER** | **Chat CLIENT** |
| /\*  // Multiperson chat server using select  // Server part  \*/    #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>    //#define PORT 9034�� // port we're listening on  fd\_set master;�� // master file descriptor list  fd\_set read\_fds; // temp file descriptor list for select()  struct sockaddr\_in myaddr;���� // server address  struct sockaddr\_in remoteaddr; // client address  int fdmax;������� // maximum file descriptor number  int listener;���� // listening socket descriptor  int newfd;������� // newly accept()ed socket descriptor  char buf[256], tmpbuf[256];��� // buffer for client data  int nbytes, ret;  int yes=1;������� // for setsockopt() SO\_REUSEADDR, below  int addrlen;  int i, j, crt, int\_port,client\_count=0;      struct sockaddr\_in getSocketName(int s, bool local\_or\_remote) {  � struct sockaddr\_in addr;  � int addrlen = sizeof(addr);  � int ret;    � memset(&addr, 0, sizeof(addr));  � ret = (local\_or\_remote==true?getsockname(s,(struct sockaddr \*)&addr,(socklen\_t\*)&addrlen):  ������� getpeername(s,(struct sockaddr \*)&addr, (socklen\_t\*)&addrlen) );  � if (ret < 0)  ��� perror("getsock(peer)name");  � return addr;  }    char \* getIPAddress(int s, bool local\_or\_remote) {  � struct sockaddr\_in addr;  � addr = getSocketName(s, local\_or\_remote);  � return inet\_ntoa(addr.sin\_addr);  }    int getPort(int s, bool local\_or\_remote) {  � struct sockaddr\_in addr;  � addr = getSocketName(s, local\_or\_remote);  � return addr.sin\_port;  }    // send to everyone  void sendToALL(char \* buf, int nbytes) {  � int j, ret;  � for(j = 0; j <= fdmax; j++) {  ��� if (FD\_ISSET(j, &master))  ����� // except the listener and ourselves  ����� if (j != listener && j != crt)  ������� if ( send(j, buf, nbytes, 0) == -1)  ������� � perror("send");  � }  � return;  }      int main(int argc, char \*\*argv)  {  ��  ����if (argc < 2 ) {  ����� printf("Usage:\n%s <portno>\n",argv[0]);  ����� exit(1);  ��� }  ��  ����int\_port = atoi(argv[1]);  ��  ����FD\_ZERO(&master);��� // clear the master and temp sets  ��� FD\_ZERO(&read\_fds);    ��� // get the listener  ��� if ((listener = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) {  ������� perror("socket");  ������� exit(1);  ��� }    ��� // lose the pesky "address already in use" error message  ��� if (setsockopt(listener, SOL\_SOCKET, SO\_REUSEADDR, &yes, sizeof(int) ) == -1) {  ������� perror("setsockopt:");  ������� exit(1);  ��� }    ��� // bind  ��� memset(&myaddr, 0, sizeof(myaddr));  ��� myaddr.sin\_family = AF\_INET;  ��� myaddr.sin\_addr.s\_addr = INADDR\_ANY;  ��� myaddr.sin\_port = htons(int\_port);  ��� //� ��memset(&(myaddr.sin\_zero), '\0', 8);  ��� if (bind(listener, (struct sockaddr \*)&myaddr, sizeof(myaddr)) == -1) {  ������� perror("bind:");  ������� exit(1);  ��� }  ��� // listen  ��� if (listen(listener, 10) == -1) {  ������� perror("listen");  ������� exit(1);  ��� }  ��� // add the listener to the master set  ��� FD\_SET(listener, &master);  ��� // keep track of the biggest file descriptor  ��� fdmax = listener; // so far, it's this one  ��� // main loop  ��� for(;;) {  ������� read\_fds = master; // copy it  ������� if (select(fdmax+1, &read\_fds, NULL, NULL, NULL) == -1) {  ����������� perror("select");  ����������� exit(1);  ������� }  �������  ������� // run through the existing connections looking for data to read  ������� for(i = 0; i <= fdmax; i++) {  ������� � if (FD\_ISSET(i, &read\_fds)) { // we got one!!  ������� ��� crt = i;  ������� ��� if (i == listener) {  ������� ����� // handle new connections  ������� ����� addrlen = sizeof(remoteaddr);  �������������� � if ((newfd = accept(listener, (struct sockaddr \*)&remoteaddr,(socklen\_t\*)&addrlen)) == -1){  �������������� ��� perror("accept");  �������������� � } else {  �������������� ��� FD\_SET(newfd, &master); // add to master set  �������������� ��� if (newfd > fdmax) {��� // keep track of the maximum  �������������� ����� fdmax = newfd;  �������������� ��� }  �������������� ��� printf("selectserver: new connection from %s on "  ���������������������� �� "socket %d\n", getIPAddress(newfd, false),newfd);  �������������� ��  �������������� ����client\_count++;  �������������� ��� sprintf(buf,"Hi-you are client :[%d] (%s:%d) connected to server %s\nThere are %d clients connected\n",  ���������������������� ��� newfd, getIPAddress(newfd,false), getPort(newfd, false),  ���������������������� ��� getIPAddress(listener, true), client\_count);  �������������� ��� send(newfd,buf,strlen(buf)+1,0);  �������������� � }  ������� ��� } else {  ������� ����� // handle data from a client  ������� ����� if ((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0) {  �������������� // got error or connection closed by client  �������������� if (nbytes == 0) {  �������������� � // connection closed  �������������� � printf("<selectserver>: client %d forcibly hung up\n", i);  �������������� }  �������������� else  �������������� � perror("recv");  �������������� client\_count--;  �������������� close(i); // bye!  �������������� FD\_CLR(i, &master); // remove from master set  ������� ����� }  ������� ����� else {  �������������� // we got some data from a client  �������������� // check for connection close request  �������������� buf[nbytes]=0;  �������������� //printf("%s\n",buf);  �������������� if ( (strncasecmp("QUIT\n",buf,4) == 0)) {  �������������� � sprintf(buf,"Request granted [%d] - %s. Disconnecting...\n",i,getIPAddress(i,false));  �������������� � send(i,buf, strlen(buf)+1,0);  �������������� � nbytes = sprintf(tmpbuf,"<%s - [%d]> disconnected\n",getIPAddress(i,false), i);  �������������� � sendToALL(tmpbuf,nbytes);  �������������� � client\_count--;  �������������� � close(i);  �������������� � FD\_CLR(i,&master);  �������������� }  �������������� else {  �������������� � nbytes = sprintf(tmpbuf, "<%s - [%d]> %s",getIPAddress(crt, false),crt, buf);  �������������� � sendToALL(tmpbuf, nbytes);  �������������� }  ������� ����� }  ������� ��� }  ������� � }  ������� }  ��� }  ��� return 0;  } | /\*  // Multiperson chat using select  // Client part  \*/  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <netdb.h>    fd\_set read\_fds,master; // temp file descriptor list for select()  int sock;������� //socket  struct sockaddr\_in servaddr;  char buf[256];� // buffer for client data  int nbytes, ret, int\_port;    int main(int argc, char \*\*argv)  {  ���  ����if (argc < 3 ) {  ����� printf("Usage:\n%s <hostname or IP address> <portno>\n",argv[0]);  ����� exit(1);  ��� }  ���  ����  ����int\_port = atoi(argv[2]);  ���  ����int ipaddr = inet\_addr(argv[1]);  ��� // check if address is a hostname  ���  ����printf("%s => %d ip address\n",argv[1],ipaddr);  ��� if (ipaddr == -1 ) {  ����� struct in\_addr inaddr;  ����� struct hostent \* host = gethostbyname( argv[1] );  ����� if (host == NULL ) {  �������������� printf("Error getting the host address\n");  �������������� exit(1);  ����� }  ����� memcpy(&inaddr.s\_addr, host->h\_addr\_list[0],sizeof(inaddr));  ����� printf("Connecting to %s ...\n",inet\_ntoa( inaddr) );  ����� memcpy(&ipaddr, host->h\_addr\_list[0],sizeof(unsigned long int)) ;  ���}    ��� // get the socket  ��� if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) {  ������� perror("socket");  ������� exit(1);  ��� }  ���  ����memset(&servaddr,0, sizeof(servaddr));  ��� servaddr.sin\_family = AF\_INET;  ��� servaddr.sin\_addr.s\_addr = ipaddr;  ��� servaddr.sin\_port = htons( int\_port );  ��� // connect to server  ��� if (connect(sock, (struct sockaddr \*)&servaddr, sizeof(servaddr)) < 0 ) {  ����� perror("connect");  ����� exit(1);  ��� }  �  ����// add the listener to the master set  ��� FD\_ZERO(&read\_fds);��� // clear the set  ��� FD\_ZERO(&master);  ��� FD\_SET(0, &master);  ��� FD\_SET(sock, &master);    ��� for(;;) {  ����� read\_fds = master;  ����� if (select(sock+1, &read\_fds, NULL, NULL, NULL) == -1) {  ������� ������� perror("select");  �������������� exit(1);  ����� }  �����  ������// check if read from keyboard  ������if ( FD\_ISSET(0, &read\_fds) ) {  �������������� nbytes = read(0, buf,sizeof(buf)-1);  �������������� ret = send(sock, buf, nbytes,0);  �������������� if (ret <= 0 ){  �������������� ������� perror("send");  ���������������������� exit(1);  �������������� }  ������� //else� printf("WARNING Not all data has been sent: %d bytes out of %d\n", ret, nbytes);  ����� }  �����  ������// check if read from server  ����� if ( FD\_ISSET(sock, &read\_fds) ) {  �������������� nbytes = read(sock, buf, sizeof(buf)-1);  �������������� if (nbytes <= 0) {  �������������� ������� printf("Server has closed connection... closing...\n");  ���������������������� exit(2);  �������������� }  �������������� write(1,buf, nbytes);  ���� }  ��� }  ���  ����return 0;  } |