Redes de Computadores (RCOMP) – 2017/2018

Laboratory Class Script PL21

- Implementing an HTTP server.
- Sample peer-to-peer UDP chat.

1. Practical exercise – implementing an HTTP server – AJAX voting

As presented in previous class, develop a small HTTP server application capable of processing some specific client's requests regarding a voting server application:

- No support for persistent connections, thus, always send to clients the **Connection: close** header line.

- Ignore client's header requests about type, language and encoding, etc.

- **GET /votes** returns a ready to use HTML content with current voting standings and buttons linked to JavaScript functions to cast votes.

- **GET** requests to other URIs are regarded as requests for static files contents stored in **www/** folder. Some common content types (Content-type) should be supported.

- Vote casting is implemented through **PUT** requests to URI **/votes/{N}**, where {N} stands for the candidate's number (1..4). PUT requests are not required to actually carry any data.

- Provided web services (GET /votes and PUT /votes/{N}) are to be consumed by JavaScript running on the browser through **XMLHttpRequest** objects (AJAX).

- The GET /votes request is intended to be used by JavaScript to keep the displayed web page information updated and should be periodically called.

- The PUT /votes/{N} request is used to cast a vote on candidate number {N}.

1.1. The proposed HTTP server implementation in C language

This C implementation is split in two source files, the **http.h** file contains base functions and definitions related to the HTTP protocol (**http.c** contains the corresponding implementations), it's compiled to an object file (**http.o**) that is later linked together with the main application's source file (**http_srv_ajax_voting.c**) compilation's result to build the final executable application.

1.1.1. Some base definitions for HTTP (http.h and http.c)

The header file (http.h) contains definitions to be included (#include) in the main application (**http_srv_ajax_voting.c**). Functions defined in http.h are implemented in http.c.

These files have already been studied, so we will not analyses them thoroughly here. One fundamental pair of functions required to implement any HTTP client or server are those meant to provide reading and writing variable length CR+LF terminated text lines.

```
void readLineCRLF(int sock, char *line);
void writeLineCRLF(int sock, char *line);
```

This is required to handle HTTP message's header. Reading operation is especially delicate because the reader doesn't know the line length in advance, so it must read one byte at a time.

The **CR** byte (Carriage Return) has the **13** decimal value and can be represented in source files by '\r'. The **LF** byte (Line Feed) has the **10** decimal value and can be represented in source files by '\n'.

Beyond these two fundamental functions, **http.h** is focused only on sending HTTP response messages, receiving HTTP messages and sending requests is not implemented here.

1.1.2. The HTTP server - http_srv_ajax_voting.c

To start with, an HTTP server must accept TCP connections from clients, so in essence, it's a TCP server.

One design issue to be addressed in this implementation is that a client's request may be a vote casting, and that changes the current voting standings. So the up to date current vote standings must be accessible when a request is processed and during a request's processing it may be necessary to change that shared information.

With a standard multi-process TCP server implementation, this would require IPC (inter-process communication), for instance using shared memory, also because several processes would be using the same shared memory (current voting standings) a mutual exclusion access mechanisms would be required, typically a semaphore.

To avoid using IPC, the purposed implementation is not entirely multi-process. The processHttpRequest() function is called for each received connection, however, before creating a child process the function checks if it's a vote casting (PUT /votes/{N}). If so, no child process is created and the request is processed within the main process, thus the current voting standings is not actually shared among several processes, it's only updated in the main process.

```
void processHttpRequest(int sock, int conSock) {
  char requestLine[200];

  readLineCRLF(conSock,requestLine);
  if(!strncmp(requestLine,"GET /",5)) {
        if(!fork()) {              // GET requests are processed in background
              close(sock);
              processGET(conSock,requestLine);
              close(conSock);
              exit(0);
              }
        close(conSock);
        return;
        }
  if(!strncmp(requestLine,"PUT /votes/",11)) processPUT(conSock,requestLine);
  else {
        sendHttpStringResponse(conSock, "405 Method Not Allowed", "text/html",
              "<html><body>HTTP method not supported</body></html>");
        puts("Oops, the method is not supported by this server");
        }

  close(conSock);
  }
```

For other requests (GET), a child process is created (fork), when doing so the current voting standings will also be duplicated to the child process (remember the fork() system-call creates an exact copy of the current process).

With this approach, PUT requests from clients are handled one by one by a single process by the order they arrive, and never more than one at the same time. Of course this is not a good solution, it was used here with the single purpose of avoiding harder to read source code.

1.2. The proposed HTTP server implementation in Java language

Unlike the C version, this is a plain typical multi-thread TCP server architecture, capable of handling several requests of any kind at the same time.

Because threads are used, no IPC is required, yet because all threads will be accessing the same data (current voting standings) mutual exclusion is required.

1.2.1. HTTPmessage.java

In HTTP servers and clients, one key concept is undoubtedly the HTTP message, thus, in an object oriented implementation it should be represented by a class.

The provided implementation (**HTTPmessage.java**) is extremely incomplete, nevertheless, it's able to send and receive both HTTP requests and responses, either with or without a content (body).

The class's static elements are private, they include several HTTP constants, methods to read and write HTTP header lines, and some associations between content types and filenames extensions. Methods, readHeaderLine() and writeHeaderLine() implementations are very similar to those used in C language for functions readLineCRLF() and writeLineCRLF().

Each class instance has a few fields to actually store the HTTP message:

```
private boolean isRequest;
private String method;
private String uri;
private String status;
private String contentType;
private byte[] content;
```

The **HTTPmessage(DataInputStream in**) constructor is used to receive an HTTP message from a provided socket's input stream and store it in the newly created instance. The other defined constructor takes no arguments and creates an undefined request message.

The first constructor is the only method for receiving an HTTP message. After reading the first line, checks if it's a request or a response, and stores relevant information for each case. Next, header fields are processed, and mostly ignored, only content length and content type are stored. If there's a content, after the HTTP header, the content is read.

The **send(DataOutputStream out)** public method is used to send the message through a provided socket's output stream.

```
public boolean send(DataOutputStream out) throws IOException {
       if(isRequest) {
              if(method==null||uri==null) return false;
              writeHeaderLine(out, method + " " + uri + " " + VERSION);
              }
       else {
              if(status==null) return false;
              writeHeaderLine(out,VERSION + " " + status);
              }

       if(content!=null) {
            if(contentType!=null) writeHeaderLine(out,CONTENT_TYPE + " " + contentType);
            writeHeaderLine(out,CONTENT_LENGTH + " " + content.length);
            }
       writeHeaderLine(out,CONNECTION + " close");
       writeHeaderLine(out,"");
       if(content!=null) {
            out.write(content,0,content.length);
            }
       return true;
       }
```

If there's a content, then, content-type and content-length header fields are included, and the content itself is sent after the HTML header.

The **setContentFromString(String c, String ct)** public method settles the HTTP message's content-type (ct) and the content from a provided string (c).

The **setContentFromFile(String fname)** public method settles the HTTP message's content by reading it from a provided filename. Returns false if fails to read the file. On success, this method also settles the content type for a few known file extensions.

Most other methods are defined to provide access to object's private elements (encapsulation).

1.2.2. HttpServerAjaxVoting.java

This all static class implements the server loop in the **main()** method, it's an already familiar multi-thread TCP server implementation in Java. For each client´s request, a new instance of the **HttpAjaxVotingRequest** class is created and then launched in background as a thread by calling the **start()** method.

Also implemented in this class, the current vote standings and an HTTP requests counter. These elements are private, they must be accessed by calling a set of synchronized static methods. Thus, these methods ensure mutual exclusion. While a thread is running one of these methods, other threads calling any of these methods will be blocked.

1.2.3. HttpAjaxVotingRequest.java

The purpose of this class is handling HTTP requests from clients, for each accepted TCP connection one instance is created by calling the HttpAjaxVotingRequest() constructor, and then executed as a thread. The constructor receives the connected socket and the base folder from where to fetch files for static contents (GET requests).

The thread's execution is enforced by calling the start() method which in turn calls the run() method. Once the socket's input (inS) and output (outS) streams are obtained, an HTTP message is received (request) and a response is created.

```
HTTPmessage request = new HTTPmessage(inS);
HTTPmessage response = new HTTPmessage();
if(request.getMethod().equals("GET")) {
     if(request.getURI().equals("/votes")) {
            response.setContentFromString(
                  HttpServerAjaxVoting.getVotesStandingInHTML(), "text/html");
            response.setResponseStatus("200 Ok");
            }
     else {
            String fullname=baseFolder + "/";
            if(request.getURI().equals("/")) fullname=fullname+"index.html";
            else fullname=fullname+request.getURI();
            if(response.setContentFromFile(fullname)) {
                  response.setResponseStatus("200 Ok");
                  }
            else {
                  response.setContentFromString(
                        "<html><body><h1>404 File not found</h1></body></html>",
                        "text/html");
                  response.setResponseStatus("404 Not Found");
                  }
            }
     response.send(outS);
     }
else { // NOT GET
     if(request.getMethod().equals("PUT")
            && request.getURI().startsWith("/votes/")) {
            HttpServerAjaxVoting.castVote(request.getURI().substring(7));
            response.setResponseStatus("200 Ok");
            }
     else {
            response.setContentFromString(
                  "<html><body><h1>ERROR: 405 Method Not Allowed</h1></body></html>",
                  "text/html");
            response.setResponseStatus("405 Method Not Allowed");
            }
     response.send(outS);
     }
```

The GET /votes request has a special treatment as it returns an HTML content produced by method getVotesStandingInHTML() of the HttpServerAjaxVoting class

containing the current voting standings and buttons attached to JavaScript functions to cast votes (HTTP request PUT /votes/{N}).

1.3. The HTML root document (www/index.html file)

This document's content is provided by the server for GET requests to URI / or URI /index.html.

```
<html><head><title>HTTP demo</title>
<script src="rcomp-ajax.js"></script>
</head>
<body bgcolor=#C0C0C0 onload="refreshVotes()"><h1>HTTP server demo - Voting with AJAX</h1>
<h3>Java version</h3>
<hr>
<center><table width=60% border=1 cellpadding=20 cellspacing=20><tr>
<td height="300" align=left width=50% valign="top">
<big><div id="votes">Please wait, loading voting results ...</div></big>
</td></tr></table></center>
<hr>
<center><table border=0><tr><td align=center>Image contents are supported:<br><br><img src=http2.png><br>(http2.png)</td>
<td align=center><img src=http.gif><br>(http.gif)</td></tr></table></center>
</body></html>
```

It retrieves JavaScript code defined in file **www/rcomp-ajax.js**, and once loaded runs the **refreshVotes()** JavaScript function for the first time. The document's area identified by name votes ( ) is updated by this function by making an HTTP request GET /votes.

1.4. JavaScript functions and AJAX (www/rcomp-ajax.js file)

The **refreshVotes()** JavaScript function is called once the main HTML document is loaded.

```
function refreshVotes() {
   var request = new XMLHttpRequest();
        var vBoard=document.getElementById("votes");
        request.onload = function() {
            vBoard.innerHTML = this.responseText;
            setTimeout(refreshVotes, 2000);
            };
        request.ontimeout = function() {
            vBoard.innerHTML = "Server timeout, still trying ...";
            setTimeout(refreshVotes, 100);
        };
        request.onerror = function() {
            vBoard.innerHTML = "No server reply, still trying ...";
            setTimeout(refreshVotes, 5000);
        };
   request.open("GET", "/votes", true);
   request.timeout = 5000;
   request.send();
   }
function voteFor(option) {
   var request = new XMLHttpRequest();
   request.open("PUT", "/votes/" + option , true);
   request.send();
   var vBoard=document.getElementById("votes");
   vBoard.innerHTML = vBoard.innerHTML + "<p>Casting your vote ...";
   }
```

The **refreshVotes()** function starts an HTTP GET /votes request, call-back functions schedule a subsequent call to the same function, so the content is

constantly updated with a periodicity equal to the time it takes to get a response, plus 2 seconds for a successful response. On success, the **votes** area of the document is replaced with the retrieved plain text content, if fact an HTML content.

The **timeout** property of the **XMLHttpRequest** is settled to 5 seconds, otherwise if the server doesn't respond to a request, the content would stop being refreshed.

This **voteFor()** function is attached to buttons to cast votes by issuing an HTTP **PUT /votes/{N}** request.

1.5. Compiling and testing

Select a host to run the server, say **ssh4.dei.isep.ipp.pt** (a CNAME for **vsrv27.dei.isep.ipp.pt**). You could use any other host, including your workstation.

This host's address in the laboratories network is 10.8.0.83 (IPv4) and fd1e:2bae:c6fd:1008::83 (IPv6), but it can also be referred by the DNS name **labs-vsrv27.dei.isep.ipp.pt**.

Download provided source files, the **Makefile** and the **www** folder. As usual, to build run the **make** command.

Ask the class's teacher for a unique port number to be used by your server (**MY-PORT-NUMBER**).

Start the C version of the HTTP server:

```
./http_srv_ajax_voting MY-PORT-NUMBER
```

a) In a workstation connected to the laboratories network, start a standard web browser and open the URL:

   http:// labs-vsrv27.dei.isep.ipp.pt:MY-PORT-NUMBER

   The main HTML page should be displayed, check that the **HTTP accesses counter** in the page is increasing every two seconds, this means JavaScript is refreshing the content as expected. You may ask nearby colleagues to also open your URL in their browsers, then the counter will increase much faster.

b) Test vote casting, ask colleagues using your server to do the same. You may also open another browser window to access your URL.

c) Use a postman application to cast votes on the second candidate (**PUT /votes/2**).

d) The Java version includes a DemoConsumer casting 200 votes on the first candidate, compile it and run it on another host:

   ```
   java DemoConsumer labs-vsrv27.dei.isep.ipp.pt MY-PORT-NUMBER
   ```

   It simply performs two hundred HTTP PUT /votes/1 requests to the server, check the results on the web page.

e) Force a server crash (press CTRL+C on the server's console). Check the browser's web page again.

f) Start the server again. Due to the server's forced crash, it may take some time before the local port number is available again.

g) Once the server finally starts, check the browser's web page again. It should be recovered, of course all counters are back to the starting point.

Stop the server's C version and start the Java version:

```
java HttpServerAjaxVoting MY-PORT-NUMBER
```

Repeat the same tests as before for the C version.

2. Peer-to-peer network applications

Peer-to-peer network applications have some unique distinct characteristics. First of which is, they do not clearly follow the client-server architecture, there are no distinct client and server applications, there is only one application with several identical instances running on the network.

From this some issues arise:

- How does one peer-to-peer application know where the other is?

- Which application takes the initiative to contact the other?

Although the client-server architecture is not directly present, one can always see a peer-to-peer application as being both a client and a server.

The problem with locating partner peer-to-peer applications can be seen as similar to locating a server application and this has already been addressed by using UDP broadcast or multicast.

By sending periodic UDP requests to a broadcast or multicast address, a peer-to-peer application can get a list of potential partners. If a **broadcast** address is used, there is, however, an unavoidable limitation: **only applications on the local network will be detected**.

One concern that this type of application will have to take is about distinguishing between other instances' requests and its own requests. This can be accomplished, for instance, by checking the request's source address to see if it matches a local interface address.

2.1. Practical exercise – implementing a peer-to-peer UDP chat

Implement a peer-to-peer UDP application with features similar to the previously developed TCP chat client and server, following these design guidelines:

- The application requests a nickname to the user.

- At startup the application sends a **peer-start-announcement** (a single byte UDP datagram with value one) to the broadcast address.

- Also, when a **peer-start-announcement** is received, a **peer-start-announcement** should be sent back to the source address (not to broadcast address).

- Each application maintains an active peers' list, created from **peer-start-announcements** it has received.

- When an application wants to exit it should send a **peer-exit-announcement** (a single byte UDP datagram with value zero) to each of the active peers in the list. Applications that receive the **peer-exit-announcement** will then remove that peer from the active peers' list.

- Other received UDP datagrams that are not peer start or exit announcements are supposed to be text messages to be printed on the console.

- When the user types a text line, it should be sent to all active partners.

- If the text line entered by the user is **EXIT**, then, before exiting, the application should send a **peer-exit-announcements** to all active peers in the list.

- Also, if the text line entered by the user is **LIST**, then a list of the active peers' IP addresses should be printed to the console.

One could argue that it would be rather easier if all messages were sent to the broadcast address. Nevertheless this is **not acceptable** because broadcast traffic must be reduced as far as possible. Broadcast and multicast traffic disables layer two segmentation and is rather **harmful for network performance**.

The suggested guidelines reduce broadcast to the minimal, each application sends a single broadcast UDP datagram only at startup.

2.2. Peer-to-peer UDP chat – suggested C language version – udp_chat.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <strings.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 PORT_NUMBER "9999"
#define BCAST_ADDRESS "255.255.255.255"
#define MAX_PEERS 100

#define GETS(B,S) {fgets(B,S-2,stdin);B[strlen(B)-1]=0;}

int main(int argc, char **argv)  {
struct sockaddr_storage peerAddr[MAX_PEERS];
char peerActive[MAX_PEERS];

struct sockaddr_storage bcastAddr, currPeerAddr;
socklen_t addrLen;
int i, err, sock;
char nick[BUF_SIZE], linha[BUF_SIZE], buff[BUF_SIZE];
struct addrinfo  req, *list;
fd_set rfds;

bzero((char *)&req,sizeof(req));
req.ai_family = AF_INET;        // we use broadcast, so there is no point in supporting IPv6
req.ai_socktype = SOCK_DGRAM;   // UDP
err=getaddrinfo(BCAST_ADDRESS, PORT_NUMBER , &req, &list);
if(err) { printf("Failed to get the broadcast address, error: %s\n",gai_strerror(err));
          exit(1); }
addrLen=list->ai_addrlen;
memcpy(&bcastAddr,list->ai_addr,addrLen); freeaddrinfo(list);

bzero((char *)&req,sizeof(req));
req.ai_family = AF_INET;        // we use broadcast, so there is no point in supporting IPv6
req.ai_socktype = SOCK_DGRAM;   // UDP
req.ai_flags = AI_PASSIVE;      // local address
err=getaddrinfo(NULL, PORT_NUMBER , &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);

i=1;setsockopt(sock,SOL_SOCKET, SO_BROADCAST, &i, sizeof(i));          // enable broadcast

for(i=0;i<MAX_PEERS;i++) peerActive[i]=0; // to start, all peers inactive

// ACTION STARTS

printf("Enter nickname: ");GETS(nick,BUF_SIZE);

buff[0]=1;           // send a peer start announcement to broadcast address
sendto(sock, &buff, 1, 0, (struct sockaddr *) &bcastAddr, addrLen);
for(;;) {
    FD_ZERO(&rfds);
    FD_SET(0,&rfds); FD_SET(sock,&rfds);
    select(sock+1,&rfds,NULL,NULL,NULL);
```

```
    if(FD_ISSET(0,&rfds)) // user wrote something on the console
        {
        GETS(linha,BUF_SIZE);
        if(!strcmp(linha,"EXIT")) break;
        if(!strcmp(linha,"LIST"))  {
              printf("Active peers list:");
              for(i=0;i<MAX_PEERS;i++)
                  if(peerActive[i])
                       {
                       getnameinfo((struct sockaddr *) &peerAddr[i], addrLen, buff,
                                        BUF_SIZE,NULL,0,NI_NUMERICHOST|NI_NUMERICSERV);
                       printf(" %s",buff);
                       }
              printf("\n");
              }
        else  {
              sprintf(buff,"(%s) %s",nick,linha);
              for(i=0;i<MAX_PEERS;i++)  // send the text line to all active peers
                if(peerActive[i])
                   sendto(sock,&buff,strlen(buff),0, (struct sockaddr *) &peerAddr[i], addrLen);
              }
        }
    if(FD_ISSET(sock,&rfds))   // there is a UDP datagram to receive
       {
       err=recvfrom(sock, &buff, BUF_SIZE, 0, (struct sockaddr *) &currPeerAddr, &addrLen);
       if(err>0)
          {
          if(buff[0]==1)   // is a peer start announcement
             {
             for(i=0;i<MAX_PEERS;i++)
                if(peerActive[i])
                   if(!memcmp(&peerAddr[i],&currPeerAddr,addrLen)) break;
             if(i==MAX_PEERS)
                { // new peer
                for(i=0;i<MAX_PEERS;i++) if(!peerActive[i]) break;
                if(i==MAX_PEERS) puts("Sorry, no space for more peers");
                else
                      {
                      peerActive[i]=1;
                      memcpy(&peerAddr[i],&currPeerAddr,addrLen);
                      buff[0]=1; // send back a peer start announcement
                      sendto(sock, &buff, 1, 0, (struct sockaddr *) &currPeerAddr, addrLen);
                      }
                 }
              }
          else
               if(buff[0]==0)   // is a peer exit announcement
                  {
                  for(i=0;i<MAX_PEERS;i++)
                      if(peerActive[i])
                          if(!memcmp(&peerAddr[i],&currPeerAddr,addrLen)) break;
                  if(i<MAX_PEERS) peerActive[i]=0;
                  }
               else         // is a text message
                    {
                    buff[err]=0; // null terminate the string
                    puts(buff);
                    }
          }
       }
    }

buff[0]=0;
for(i=0;i<MAX_PEERS;i++)  // send exit announcement to all active peers
    if(peerActive[i])
        sendto(sock, &buff, 1, 0, (struct sockaddr *) &peerAddr[i], addrLen);
close(sock);
exit(0);
}
```

2.3.Peer-to-peer UDP chat – suggested Java language version – UdpChat.java

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

class UdpChat {

private static final String BCAST_ADDR = "255.255.255.255";
private static final int SERVICE_PORT = 9999;

private static HashSet<InetAddress> peersList = new HashSet<>();

public static synchronized void addIP(InetAddress ip) { peersList.add(ip);}

public static synchronized void remIP(InetAddress ip) { peersList.remove(ip);}

public static synchronized void printIPs() {
		for(InetAddress ip: peersList) {
			System.out.print(" " + ip.getHostAddress());
			}
		}

public static synchronized void sendToAll(DatagramSocket s, DatagramPacket p) throws Exception {
		for(InetAddress ip: peersList) {
			p.setAddress(ip);
			s.send(p);
			}
		}

	static InetAddress bcastAddress;
	static DatagramSocket sock;

	public static void main(String args[]) throws Exception {
		String nick, frase;
		byte[] data = new byte[300];
		byte[] fraseData;
		int i;
		DatagramPacket udpPacket;

		try { sock = new DatagramSocket(SERVICE_PORT); }
		catch(IOException ex) {
			System.out.println("Failed to open local port");
			System.exit(1); }

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

		System.out.print("Nickname: "); nick = in.readLine();

		bcastAddress=InetAddress.getByName(BCAST_ADDR);
		sock.setBroadcast(true);
		data[0]=1;
		udpPacket = new DatagramPacket(data, 1, bcastAddress, SERVICE_PORT);
		sock.send(udpPacket);

		Thread udpReceiver = new Thread(new UdpChatReceive(sock));
		udpReceiver.start();

		while(true) { // handle user inputs
			frase=in.readLine();
			if(frase.compareTo("EXIT")==0) break;
			if(frase.compareTo("LIST")==0) {
				System.out.print("Active peers:");
				printIPs();
				System.out.println("");
				}
			else {
				frase="(" + nick + ") " + frase;
				fraseData = frase.getBytes();
```

```
                            udpPacket.setData(fraseData);
                            udpPacket.setLength(frase.length());
                            sendToAll(sock,udpPacket);
                            }
                    }
            data[0]=0; // announce I'm leaving
            udpPacket.setData(data);
            udpPacket.setLength(1);
            sendToAll(sock,udpPacket);
            sock.close();
            udpReceiver.join(); // wait for the UdpChatReceive thread to end
            }
    }

class UdpChatReceive implements Runnable {
    private DatagramSocket s;

    public UdpChatReceive(DatagramSocket udp_s) { s=udp_s;}

    public void run() {
            int i;
            byte[] data = new byte[300];
            String frase;
            DatagramPacket p;
            InetAddress currPeerAddress;

            p=new DatagramPacket(data, data.length);

            while(true) {
                    p.setLength(data.length);
                    try { s.receive(p); }
                    catch(IOException ex) { return; }
                    currPeerAddress=p.getAddress();

                    if(data[0]==1) { // peer start
                            UdpChat.addIP(p.getAddress());
                            try { s.send(p); }
                            catch(IOException ex) { return; }
                            }
                    else
                    if(data[0]==0) { // peer exit
                            UdpChat.remIP(p.getAddress());
                            }
                    else {          // chat message
                            frase = new String( p.getData(), 0, p.getLength());
                            System.out.println(frase);
                            }
                    }
            }
    }
```

- A thread is started to receive UDP datagrams from the network, it receives text messages and peer announcements (start and exit). While text messages are printed to the console, received announcements are used to add and remove peers from the list.

- The main thread reads the keyboard and sends UDP datagrams to peers in the list. Both threads need to use the static peers' list, so it's accessed through static synchronized methods to ensure mutual exclusion.