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

Laboratory Class 12 (PL12 – 2.5 hours)

Implementing a peer-to-peer UDP chat application (C and Java).

1. Peer-to-peer network applications

Peer-to-peer network applications have some unique distinct characteristics. They do not clearly follow the client-server architecture; this means the application is at the same time a client and a server. In the peer-to-peer model, there aren't two distinct applications like a client application and a server application, there is only one application. Nevertheless, there will be several identical instances of the same peer-to-peer application running on the network, and they will exchange information between them.

From this concept, some issues arise:

- How does one peer-to-peer application know where the other is?
- Which application takes the initiative of contacting 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, may be seen as rather 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 same local network (broadcast domain) will be detected**.

One concern that this kind 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. Practical exercise – implementing a peer-to-peer UDP chat

Implement a peer-to-peer UDP application, with features like those of the previously developed TCP chat client and server, following these new protocol design guidelines:

- The application requests a nickname to the user.
- At start-up, the application sends a **peer-start-announcement** (a single byte UDP datagram with value one) to the broadcast address.
- When a **peer-start-announcement** is received, a **peer-start-announcement** should be sent back to the source address only (unicast, not to the broadcast address).
- Each application maintains an active peers' list, created from **peer-start-announcements** it has received so far.
- When an application wants to exit, it must send the **peer-exit-announcement** (a single byte UDP datagram with value zero), it must be sent 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, which 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.
- If the text line entered by the user is **LIST**, then a list of the active peers' IP addresses should be printed at the console.

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

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

2.1. Peer-to-peer UDP chat – 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);
}
```

2.2. Peer-to-peer UDP chat – 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 is accessed through static synchronized methods to ensure mutual exclusion.