2nd Group Homework Report

TCP Simple Broadcast Chat Server and Client

ECEN 602 Network Programming Assignment 2

Mainly, we worked together, but here is the role what we did.

Minhwan Oh : Developed server programming, debugged for integration

Sanghyeon Lee : Developed client programming, tested for integration

File info

For this program, you can see how TCP Simple Broadcast Chat Server and Client works. There are two main files.

1. Client.c

Path: cd Assignment_2/Client.c

Main feature: Client application for chat room

2. Server.c

Path: cd Assignment_2/Server.c

Main feature: Server application for chat room

3. unp.h

Path: cd Assignment _2/unp.h

Main feature: Packages for socket programming function *Refers to Unix Network Programming library*

4. config.h

Path: cd Assignment_2/config.h

Main feature: Autoheader

5. makefile

Main feature: Compile setting description

Build info

Command $make

Program scenario

This program includes client and server for a TCP simple broadcast chat service, which does the following:

1. Start the server first with the commanding lien: $server <IPAdr> <Port> <Accessible Client Number>, where “server” is the name of the server program, IPAdr is the IPv4 (or IPv6, need to switch several lines in code) address of the server in dotted decimal notation, Port is the port number on which the server is listening, and Accessible Client Number is the maximum number of clients can join server at the same time. The server must support multiple simultaneous connections.

2. Start the client second with a command line: $client <User Name> <IPAdr> <Port>, where client is the name of the client program, User Name is the client's name which will be used in server as a client ID, IPAdr is the IPv4 (or IPv6, need to switch several lines in code) address of the server in dotted decimal notation, and Port is the port number on which the server is listening. Then, client sends JOIN type to server to join the chat session.

3. When server receives the JOIN type header from a client, the server checks the client's username is on the existing client list. If the same user name is assigned on the list, the server sends NAK type header to reject its connection. If the user name is new, the server sends ACK type header to the client to confirm the JOIN request and broadcasts the arrival of new participant to the other client who are already in the chat session by sending ONLINE header message.

4. A client sends a message (e.g. Hi guys!) to server, and the server receives the message, then broadcast it to the other clients by using FWD message which contains the message and its username. If header or attribute type is wrong, DISCARD it

5. If a client doesn't send any message for 10 seconds, the server broadcasts it to the other clients that the user is in IDLE state. However, if the client sends any message again, the IDLE state is over, and the server counts the time again to check the IDLE state of each clients in the chat session.

6. A client leave the session unceremoniously. The server broadcasts OFFLINE message to the other existing clients and cleanup the allocated memory of the left client for new clients can use that user name again.

Program details

1. The communication will work with either IPv4 and IPv6 networks.

1. The server handles all clients' access by using ACK(accepted) and NAK(rejected) messages and broadcasts the clients' status (ONLINE(new client joined), OFFLIEN(client left), and IDLE(didn't move for 10 seconds)) and each other's messages (SEND and FWD) to other clients.

2. The client ask (JOIN) permission to join chat session and gets the client information of chat session (current client list and the number of client).

3. Clients send and receive messages through server's FWD system.

4. The server detects clients' idle status, which clients don't send any message for 10 seconds, and inform it to the other clients. If the idle clients move, the idle status is over and the server initiates the countdown idle timer to zero.

File architecture

1. Server

a) Architecture

- Scenario :

  (1) Setup client_info/user_check/broadcast/ACK/NAK/bind/listen functions

  (2) Then going to infinite loop until the exit

- Main function :

(1) int main(int argc, char **argv): Including main scenario

(2) int check_newFd(int nClient, int nMax_Client, char *username, Client *clientinfo,int connfd): Check new client

(3) void broadcast_all(Message broadcast_Msg,fd_set check_fd, int listenfd, int connfd, int latest_fd): forward clients' status and messages

b) Feature

- Create socket and bind the well-known port of server
- Check new clients and accept or rejects them
- Forward clients' status and messages to the other clients

2. Client

a) Architecture

- Scenario :

(1) Setup JOIN/sendMessage/recvMessage/socket/connect function

(2) Then going to infinite loop until the exit

- Main function :

(1) int main(int argc, char **argv): Including main scenario

(2) int joinServer(char *buf,int sockfd, int size_buf): Make JOIN in SBCP message format

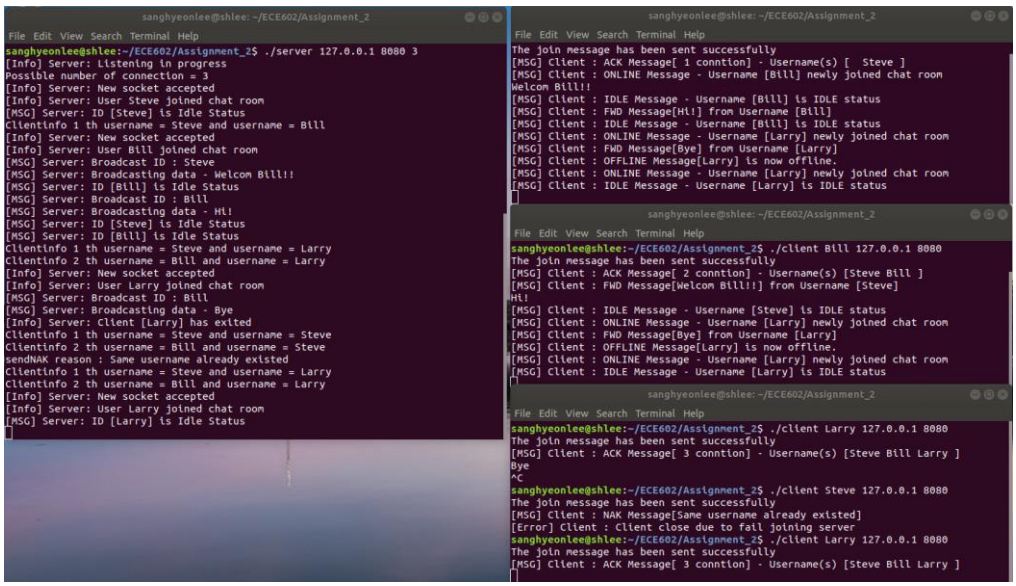
(3) int RecvMsg(int sockfd): handle received messages depend on its header type

b) Feature

- Create socket and bind the well-known port of server
- Monitor headers in received message and print it.

Result

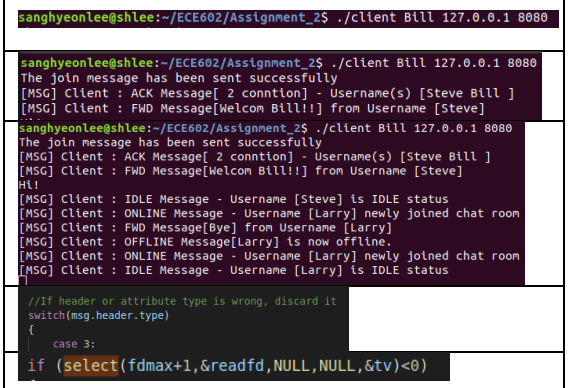
Simulation overview



Requirement

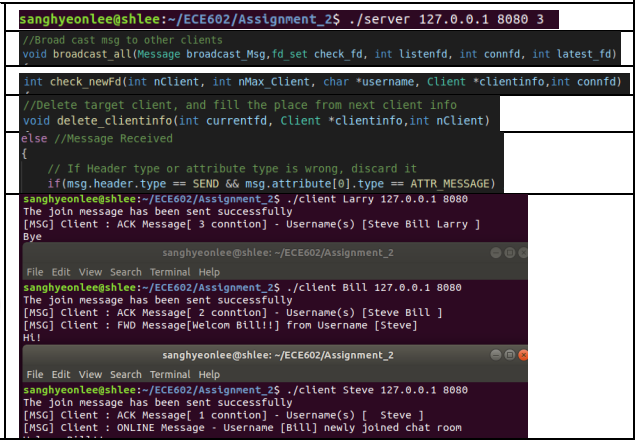
Client

#	Requirement
1	The client should connect to the server using the IP and port supplied on the command line. <./client username server_ip server_port>
2	The client should initiate a JOIN with the server using the username supplied on the command line.
3	The client should display a basic prompt to the operator for displaying received FWD messages, and typed message that are sent (SEND).
4	The client will need to use I/O multiplexing (select) to handle both sending and receiving of messages.
5	The client should discard any message that is not understood.



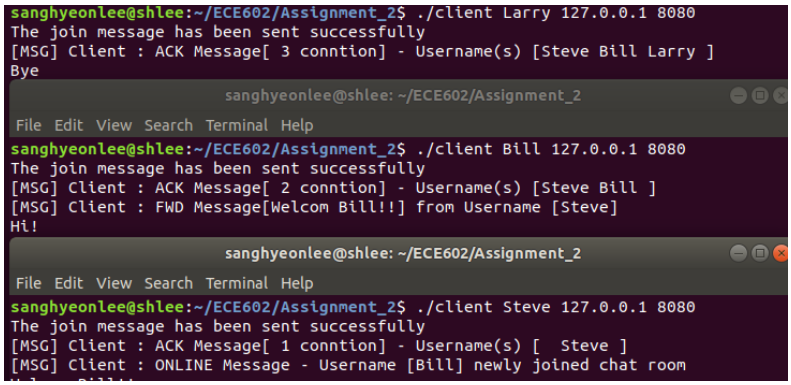
Server

#	Requirement
1	The server should start on the IP and port supplied on the command line. <./server server_ip server_port max_clients>
2	A SEND received by the server will cause a copy of the MESSAGE text to be sent in FWD to all clients except the original sender.
3	A server may only accept JOIN SBCP messages from unknown clients. The server will not allow multiple clients to use the same username.
4	Clients may leave the chat session unceremoniously. The server should cleanup its resources (close socket, make the username available).
5	The server should discard any messages that are not understood.
6	The implementation can be an iterative server. A distinct TCP connection is used for each client-server transaction.



Test cases

(1) Normal operation of the chat client with three clients connected



The maximum number of clients can be set from server input. The clients' information is stored in client_info data. The figure above shows that three clients are connected to the server at the same time.

This can be done by I/O multiplexing which makes it possible that send more than two data through one channel. In select function, the server can monitor file descriptors and check buffer whether it's empty or not by checking FD_ISSET.

(2) Server rejects a client with a duplicate username

```
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./client Larry 127.0.0.1 8080
The join message has been sent successfully
[MSG] Client : ACK Message[ 3 conntion] - Username(s) [Steve Bill Larry ]
Bye
^C
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./client Steve 127.0.0.1 8080
The join message has been sent successfully
[MSG] Client : NAK Message[Same username already existed]
[Error] Client : Client close due to fail joining server
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./client Larry 127.0.0.1 8080
The join message has been sent successfully
[MSG] Client : ACK Message[ 3 conntion] - Username(s) [Steve Bill Larry ]
```

Steve is already on the list of client so that the server sends NAK to the new client and rejected it. Function check_newFd is where the server checks the new username is already used for current clients' username.

(3) Server allows a previously used username to be reused

```
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./client Larry 127.0.0.1 8080
The join message has been sent successfully
[MSG] Client : ACK Message[ 3 conntion] - Username(s) [Steve Bill Larry ]
Bye
^C
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./client Steve 127.0.0.1 8080
The join message has been sent successfully
[MSG] Client : NAK Message[Same username already existed]
[Error] Client : Client close due to fail joining server
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./client Larry 127.0.0.1 8080
The join message has been sent successfully
[MSG] Client : ACK Message[ 3 conntion] - Username(s) [Steve Bill Larry ]
```

On the other hand, Larry who had left before re-enter the chat session and reused its username again because the username Larry was removed when the client left through a function delete_clientinfo.

(4) Server rejects the client because it exceeds the maximum number of clients allowed.

```
sanghyeonlee@shlee: ~/ECE602/Assignment_2
File  Edit  View  Search  Terminal  Help
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./client Steve 127.0.0.1 8080
The join message has been sent successfully
[MSG] Client : ACK Message[ 1 conntion] - Username(s) [  Steve ]
[MSG] Client : ONLINE Message - Username [Bill] newly joined chat room
sanghyeonlee@shlee: ~/ECE602/Assignment_2
File  Edit  View  Search  Terminal  Help
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./client Bill 127.0.0.1 8080
The join message has been sent successfully
[MSG] Client : ACK Message[ 2 conntion] - Username(s) [Steve Bill ]
[MSG] Client : FWD Message[Welcom Bill!!] from Username [Steve]
sanghyeonlee@shlee: ~/ECE602/Assignment_2
File  Edit  View  Search  Terminal  Help
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./client Larry 127.0.0.1 8080
The join message has been sent successfully
[MSG] Client : ACK Message[ 3 conntion] - Username(s) [Steve Bill Larry ]
sanghyeonlee@shlee: ~/ECE602/Assignment_2
File  Edit  View  Search  Terminal  Help
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./client Mark 127.0.0.1 8080
The join message has been sent successfully
[MSG] Client : NAK Message[Server : Full of clients]
[Error] Client : Client close due to fail joining server
sanghyeonlee@shlee:~/ECE602/Assignment_2$
```

The maximum number of client is 3, and currently three clients, Steve, Bill, and Larry are in the chat session. So, the fourth client Mark has rejected due to "full of clients" reason. This can be done with condition functions with the defined maximum client number.

(5) Bonus features

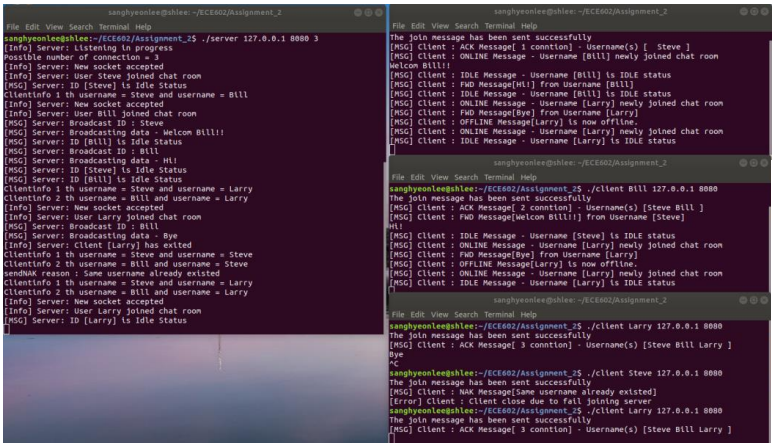
1) Feature 0 IPv4 and IPv6

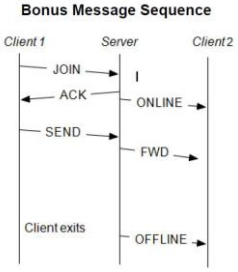
```
//for IPv6
struct hostent*     hret;
```

```
bzero(&servaddr, sizeof(servaddr));
//servaddr.sin_family      = AF_INET;
servaddr.sin_family      = AF_INET6;
servaddr.sin_port        = htons(atoi(argv[2]));
hret = gethostbyname(argv[1]);
inet_pton(AF_INET6, argv[1], &servaddr.sin_addr);
memcpy(&servaddr.sin_addr.s_addr, hret->h_addr,hret->h_length);
```

We used Hosten which is compatible for IPv6 and changed AF_INET to AF_INET6 and inet_pton().

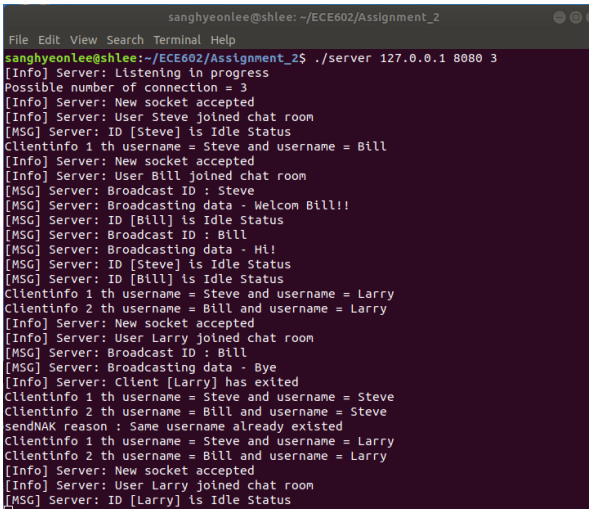
2) Feature 1 ACK, NAK, ONLINE, OFFLINE





We made SBCP header types not only JOIN, SEND, and FWD, but also ACK, NAK, ONLINE, OFFLINE, and IDEL. ACK is accept header for a new client to join the chat session. NAK is reject header which returns reasons why the new client cannot join the chat session. ONLINE header is status which is broadcasted to other clients by server. OFFLINE header is also status to inform the other clients that a client has left. Even though client left unceremoniously, server broadcasts the other with OFFLINE header.

3) Feature 2 IDLE



```
sanghyeonlee@shlee: ~/ECE602/Assignment_2
File  Edit  View  Search  Terminal  Help
sanghyeonlee@shlee:~/ECE602/Assignment_2$ ./server 127.0.0.1 8080 3
[Info] Server: Listening in progress
Possible number of connection = 3
[Info] Server: New socket accepted
[Info] Server: User Steve joined chat room
[MSG] Server: ID [Steve] is Idle Status
Clientinfo 1 th username = Steve and username = Bill
[Info] Server: New socket accepted
[Info] Server: User Bill joined chat room
[MSG] Server: Broadcast ID : Steve
[MSG] Server: Broadcasting data - Welcom Bill!!
[MSG] Server: ID [Bill] is Idle Status
[MSG] Server: Broadcast ID : Bill
[MSG] Server: Broadcasting data - Hi!
[MSG] Server: ID [Steve] is Idle Status
[MSG] Server: ID [Bill] is Idle Status
Clientinfo 1 th username = Steve and username = Larry
Clientinfo 2 th username = Bill and username = Larry
[Info] Server: New socket accepted
[Info] Server: User Larry joined chat room
[MSG] Server: Broadcast ID : Bill
[MSG] Server: Broadcasting data - Bye
[Info] Server: Client [Larry] has exited
Clientinfo 1 th username = Steve and username = Steve
Clientinfo 2 th username = Bill and username = Steve
sendNAK reason : Same username already existed
Clientinfo 1 th username = Steve and username = Larry
Clientinfo 2 th username = Bill and username = Larry
[Info] Server: New socket accepted
[Info] Server: User Larry joined chat room
[MSG] Server: ID [Larry] is Idle Status
```

IDLE is the feature that client doesn't send any message for 10 seconds, server counts the time and inform the others that the client is in IDLE status (i.e. sleep mode). Once the IDLE status is informed server doesn't send it again. However, when the IDLE client send messages, the idle timer initiated to zero. So, if the client doesn't send any message again for 10 seconds, the server broadcasts IDLE status again.

```
struct timeval tv;
tv.tv_sec = 0;
tv.tv_usec = 100000;

fdmax = sockfd;
if (select(fdmax+1,&readfd,NULL,NULL,&tv)<0)
```

Select function check any changes of file descriptors. While it handles multiple I/O, it returns changed file descriptor. However, if there is no change, it become block state (infinite waiting) if select function value is

(select(nFd, fd, NULL, NULL, NULL). Thus, we set timeinterval argument timeout tv which can make select function to be called in every tv period time. If there is no change, we consider it as the client doesn't do anything for tv time and it becomes IDLE status. Our time interval to wake up is 10ms, then if the count of 10ms is up to 100 which means 10s, we sends IDLE message to server.

Makefile

==========

```
PROGS =         server client
CLEANFILES = *.o
CC = gcc
CFLAGS = -I../lib -g -O2 -D_REENTRANT -Wall

all:        ${PROGS}

server:  server.o
                ${CC} ${CFLAGS} -o $@ server.o
client:   client.o
                ${CC} ${CFLAGS} -o $@ client.o

clean:
                rm -f ${PROGS} ${CLEANFILES}
```

server.c

==========

```
#include    "unp.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/select.h>

//Protocol Version
#define VERSION 3

#define JOIN 2
#define SEND 4
#define FWD 3
#define ACK 7
#define NAK 5
#define ONLINE 8
#define OFFLINE 6
#define IDLE 9

#define MAX_USERNAME 16
#define MAX_MESSAGE 512
#define MAX_REASON 32
#define CLIENT_COUNT 2

//SBCP Attribute Type
```

```
#define ATTR_USERNAME 2
#define ATTR_MESSAGE 4
#define ATTR_REASON 1
#define ATTR_CLIENT_COUNT 3

//Typedef for convenience
typedef struct sockaddr_in sockaddr;

//SBCP Header
struct Header_SBCP
{
	unsigned int vrsn : 9;
	unsigned int type : 7;
	unsigned int length : 16;
};
typedef struct Header_SBCP Header;

//SBCP Attribute
struct Attribute_SBCP
{
	unsigned int type : 16;
	unsigned int length : 16;
	char payload[MAX_MESSAGE];
};
typedef struct Attribute_SBCP Attribute;


/**************************************
 *
 *	Structure : SBCP Message
 *	Attribute[0] : It contains message
 *	Attribute[1] : It contains username
 *
**************************************/
struct Message_SBCP
{
	Header header;
	Attribute attribute[2];
};
typedef struct Message_SBCP Message;

struct Client_Info
{
		char username[16];
		unsigned int fd_num;
};
typedef struct Client_Info Client;

//Delete target client, and fill the place from next client info
void delete_clientinfo(int currentfd, Client *clientinfo,int nClient)
{
		int i;
		int move =0;

		for(i = 0; i<nClient; i++)
		{
			if(move == 1 )
			{
				clientinfo[i-1].fd_num = clientinfo[i].fd_num ;
				strcpy(clientinfo[i-1].username,clientinfo[i].username);
			}

			if(clientinfo[i].fd_num == currentfd)
			{
				clientinfo[i].fd_num = 0;
				bzero(clientinfo[i].username,sizeof(clientinfo[i].username));
```

```
                                        move = 1;
                        }
                }

}

void find_username(int currentfd, Client *clientinfo, int nClient, char *username)
{
        int i;

        for(i = 0; i < nClient; i++)
        {
                if( clientinfo[i].fd_num == currentfd)
                {
                        strcpy(username, clientinfo[i].username);
                }
        }
}

int check_newFd(int nClient, int nMax_Client, char *username, Client *clientinfo,int connfd)
{
        int i;
        int result    = 0;

        if( nClient >= nMax_Client)
        {
                result = 2;
        }
        else
        {
                for(i =0; i < nClient; i++)
                {
                        printf("Clientinfo %d th username = %s and username = %s \n",i+1,
clientinfo[i].username, username);
                        if(strcmp(clientinfo[i].username, username) ==0)
                        {
                                //Duplicated username
                                result = 1;
                        }
                }

                if( result != 1)
                {
                        strcpy(clientinfo[i].username,username);
                        clientinfo[i].fd_num = connfd;
                }
        }

        return result;
}

//Broad cast msg to other clients
void broadcast_all(Message broadcast_Msg,fd_set check_fd, int listenfd, int connfd, int latest_fd)
{
        int i;
      for( i = 0; i <= latest_fd; i++)
      {
                //check the broadcasting fd is valid or not
            if (FD_ISSET(i, &check_fd))
            {
                // except the listener and ourselves
                if (i != listenfd && i != connfd)//dont broadcast to the original sender and the server
                {
                    if (send(i,(void *) &broadcast_Msg,sizeof(broadcast_Msg),0) < 0)
                    {
                        printf("[Error] Server : Fail to broadcast message\n");
```

```
                    }
                }
            }
        }
}

//send ACK message to the new client
void sendACK(int connfd, int nClient, char *username, Client *clientinfo)
{
        Message msg;
                char a[2];
                char clientlist[MAX_MESSAGE];
                strcat(clientlist, "\b\b");
                for(int i=0 ; i<nClient ; i++)
                {
                        strcat(clientlist, clientinfo[i].username);
                        strcat(clientlist, " ");
                }

        msg.header.vrsn = VERSION;
        msg.header.type = ACK;

        msg.attribute[0].type = ATTR_CLIENT_COUNT; //the payload in the attribute is message
                sprintf( msg.attribute[0].payload , "%d", nClient ) ;
        msg.attribute[0].length = 4 + strlen(msg.attribute[0].payload ); //default 4 bytes + length of client count value

                msg.attribute[1].type = ATTR_USERNAME;
                strcpy(msg.attribute[1].payload, clientlist);
                msg.attribute[1].length = 4 + strlen(msg.attribute[1].payload);

                if(send(connfd,(void *) &msg,sizeof(msg),0) < 0)
                {
                        printf("[Error] Server : Fail to send ACK message\n");
                }
}

//send NAK message to the new client
void sendNAK(int connfd,int code)
{

        Message msg;

                //Reason Attribute Payload should be equal or less than 32
                char strReason[32];

                msg.header.vrsn =3;
        msg.header.type =NAK;
        msg.attribute[0].type = ATTR_REASON;

                switch(code)
                {
                        case 1:
                                //the flag to mark this NAK is for username existed
                                strcpy(strReason,"Same username already existed");//
                                break;
                        case 2:
                                strcpy(strReason,"Server : Full of clients");//
                                break;
                        default:
                        break;
                }

                strcpy(msg.attribute[0].payload, strReason);
        msg.attribute[0].length = 4 + strlen(strReason);//the length of the payload is depends on the message        we write

                printf("sendNAK reason : %s \n",strReason);
```

```
		msg.header.length = 8 + strlen(strReason);

	if(send(connfd,(void *) &msg,sizeof(msg),0) < 0)
		{
			printf("[Error] Server : Fail to send NAK message\n");
		}
		//This socket should be closed due to inproper condition
	close(connfd);
}

int main(int argc, char **argv)
{

		int 					listenfd, connfd, nMaxfd, nMaxi, nMax_Client,
nReady, i;
		int 				nClient,nClientinfo,recv_check, result;
		char 				broadID[MAX_USERNAME];
		pid_t 				childpid;
		socklen_t 		Client_Length;
		sockaddr 		cliaddr, servaddr;
		fd_set 				check_fd, latest_fd;
		Client 				*clientinfo;
	struct hostent*	hret;

		Message 		msg;
		Message 		broadcast_Msg;

		//As assignment, set argc as 4
		if (argc != 4)
		{
			printf("[Error] Server : Follow those command - ./Server <Server_IP> <Server_Port> <Number of
Clients> \n");
			printf("[Error] Server : Program Exit");
			return 0;
		}

		nMax_Client = atoi(argv[3]);

		if( nMax_Client < 1)
		{
			printf("[Error] Server : nMax_Client should be greater than 0 \n");
		}
		else
		{
			//Allocate client info array
			clientinfo = (struct Clientinfo *)malloc(nMax_Client * sizeof(Client));
		}


		listenfd = socket(AF_INET, SOCK_STREAM, 0);
		//listenfd = socket(AF_INET6, SOCK_STREAM, 0);

		//allow multiple connections
		int opt=1;
		if( setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, (char *) &opt,sizeof(opt))<0)
		{
			printf("[Error] Server: Multiple connections on server socket can't create\n");
		}

		bzero(&servaddr, sizeof(servaddr));
		servaddr.sin_family		= AF_INET;
		//servaddr.sin_family		= AF_INET6;
		servaddr.sin_port		= htons(atoi(argv[2]));
	hret = gethostbyname(argv[1]);
		//inet_pton(AF_INET6, argv[1], &servaddr.sin_addr);
	memcpy(&servaddr.sin_addr.s_addr, hret->h_addr,hret->h_length);
```

```
	bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

	listen(listenfd, LISTENQ);
	printf("[Info] Server: Listening in progress \n");

	nMaxfd = listenfd;
	printf("Possible number of connection = %d \n", nMaxfd);

	/*********************************************************
	 *
	 * FD is file descriptor, and can use it to include select.h
	 * below is FD structure function
	 *
	 * #define FD_SET(fd, fdsetp)   __FD_SET (fd, fdsetp)
	 * #define FD_CLR(fd, fdsetp)   __FD_CLR (fd, fdsetp)
	 * #define FD_ISSET(fd, fdsetp)__FD_ISSET (fd, fdsetp)
	 * #define FD_ZERO(fdsetp)                    __FD_ZERO (fdsetp)
	 *
	 *
	 *********************************************************/

	//FD initialization
	FD_ZERO(&check_fd);
	FD_ZERO(&latest_fd);
	FD_SET(listenfd, &latest_fd);

	// Network variables initialization
	nClient = 0;
	nClientinfo = 0;
	recv_check = 0;

	for ( ; ; )
	{

		check_fd = latest_fd;		/* structure assignment */
		nReady = select(nMaxfd+1, &check_fd, NULL, NULL, NULL);

		if( nReady < 0)
		{
			printf("[Error] Server: select function doesn't work properly");
		}

		for(i = 0 ; i <= nMaxfd ; i++)
		{
			if(FD_ISSET(i, &check_fd)) //Check whether the fd is valid or not
			{
				if(i == listenfd) //Data received on main server, it menas new socket is
coming
				{
					Client_Length = sizeof(cliaddr);
					connfd = accept(listenfd, (struct sockaddr *) &cliaddr,
&Client_Length); //Accept Client

					if( connfd < 0)
					{
						printf("[Error] Server: Failed to accept Socket\n");
					}
					else
					{

						FD_SET(connfd, &latest_fd); //add to FD set

						recv_check = recv(connfd,(Message
*)&msg,sizeof(Message),0);
```

```
                                                result = check_newFd(nClient,
nMax_Client,msg.attribute[0].payload,clientinfo,connfd);

                                                if(result == 0)
                                                {
                                                        nClient++;

                                                        if(connfd > nMaxfd)
                                                        {
                                                                nMaxfd = connfd; //new largest
fds
                                                        }

                                                        //printf("[Info] Server: Maxfd = %d Connfd
value is %d \n\n", nMaxfd, connfd);
                                                        sendACK(connfd, nClient,
msg.attribute[0].payload,clientinfo);
                                                        printf("[Info] Server: New socket
accepted\n");
                                                        printf("[Info] Server: User %s joined chat
room \n",msg.attribute[0].payload);

                                                        msg.header.vrsn = VERSION;
                                msg.header.type = ONLINE;
                                msg.attribute[0].type = ATTR_USERNAME;
                                strcpy(msg.attribute[0].payload,msg.attribute[0].payload);

                                broadcast_all(msg,latest_fd,listenfd,connfd, nMaxfd);

                                                }
                                                else
                                                {

                                                        sendNAK(connfd,result);
                                                        FD_CLR(connfd, &latest_fd);//clear
accept_socket_fd out of
                                                }

                                        }

                                }
                                else    //Data received
                                {
                                        recv_check = recv(i,(Message *)&msg,sizeof(Message),0);

                                        if(recv_check <= 0) //Client has exited
                                        {

                                                msg.header.vrsn = VERSION;
                                                msg.header.type = OFFLINE;
                                                msg.attribute[0].type = ATTR_USERNAME;
                                                find_username(i, clientinfo, nClient, broadID);
                                                strcpy(msg.attribute[0].payload, broadID);
                                                msg.attribute[0].length = 4 + strlen(broadID);



        bzero((char*)&msg.attribute[1].payload,sizeof(msg.attribute[1].payload));

                                                broadcast_all(msg,latest_fd,listenfd,i, nMaxfd);

                                                //Remove Client, Clean Resources and Inform Other
Clients
                                                FD_CLR(i,&latest_fd);

                                                delete_clientinfo(i, clientinfo, nClient);
```

```
                                                        //Remove Client Name from list
                                                        close(i);
                                                        nClient--;
                                                        printf("[Info] Server: Client [%s] has
exited\n",broadID );

                                                }
                                                else //Message Received
                                                {
                                                        // If Header type or attribute type is wrong, discard it

                                                        if(msg.header.type == SEND && msg.attribute[0].type
== ATTR_MESSAGE) //Check if Forward
                                                        {

                                                                printf("[MSG] Server: Broadcast ID : %s
\n",broadID );
                                                                printf("[MSG] Server: Broadcasting data
- %s\n", msg.attribute[0].payload);

                                                                msg.header.type = FWD;
                                                                msg.attribute[1].type =
ATTR_USERNAME;

                                                                find_username(i, clientinfo, nClient,
broadID);
                                                                strcpy(msg.attribute[1].payload, broadID);

                                                                msg.attribute[1].length = 4 +
strlen(broadID);

                                                                broadcast_all(msg,latest_fd,listenfd,i,
nMaxfd);
                                                        }
                                                        else if(msg.header.type == IDLE)
                                                        {

                                                                msg.header.type = IDLE;
                                                                msg.attribute[0].type =
ATTR_USERNAME;

                                                                find_username(i, clientinfo, nClient,
broadID);
                                                                strcpy(msg.attribute[0].payload, broadID);

                                                                msg.attribute[0].length = 4 +
strlen(broadID);

                                                                broadcast_all(msg,latest_fd,listenfd,i,
nMaxfd);
                                                                printf("[MSG] Server: ID [%s] is Idle
Status\n", broadID);
                                                        }

                                                }
                                        }

                                }
                        }
                }
                close(listenfd);
                return 0;
}
```

client.c

==========

```
#include    "unp.h"
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <unistd.h>
#include <time.h>



//Protocol Version
#define VERSION 3

#define MAX_MESSAGE                     512

#define JOIN 2
#define SEND 4
#define FWD 3
#define ACK 7
#define NAK 5
#define ONLINE 8
#define OFFLINE 6
#define IDLE 9

//SBCP Attribute Type
#define ATTR_USERNAME 2
#define ATTR_MESSAGE 4
#define ATTR_REASON 1
#define ATTR_CLIENT_COUNT 3

//SBCP Header
struct Header_SBCP
{
      unsigned int vrsn : 9;
      unsigned int type : 7;
      unsigned int length : 16;
};
typedef struct Header_SBCP Header;

//SBCP Attribute
struct Attribute_SBCP
{
      unsigned int type : 16;
      unsigned int length : 16;
      char payload[MAX_MESSAGE];
};
typedef struct Attribute_SBCP Attribute;

/***************************************
 *
 *   Structure : SBCP Message
 *   Attribute[0] : It contains message as SendMsg, and contains username when client joins server
 *   Attribute[1] : It contains username
 *
***************************************/
struct Message_SBCP
{
      Header header;

      Attribute attribute[2];
```

```
};
typedef struct Message_SBCP Message;

// Request to join server
int joinServer(char *buf,int sockfd, int size_buf)
{
        int status = 0;
    Message msg;

    msg.header.vrsn = VERSION; //set the protocal version is 3
    msg.header.type = JOIN; // join message is type 2
    msg.attribute[0].type = ATTR_USERNAME; //username used in chatting
    msg.attribute[0].length = size_buf + 4;

    bzero((char*)&msg.attribute[0].payload,sizeof(msg.attribute[0].payload));
    bzero((char*)&msg.attribute[1].payload,sizeof(msg.attribute[1].payload));

        strcpy(msg.attribute[0].payload,buf);

    msg.header.length = ( 8 + size_buf );

    if (send(sockfd,&msg,sizeof(msg),0) < 0)
    {
       printf("[Error] Client : Failed to join\n");
    }
    printf("The join message has been sent successfully\n");

        sleep(1);
        status = RecvMsg(sockfd);

        return status;
}


void SendMsg(int sockfd)
{
    Message msg;
    int size_buf;
    char buf[MAX_MESSAGE];

    int i;

    fgets(buf,sizeof(buf)-1,stdin);

    size_buf = strlen(buf)-1;

    if (buf[size_buf]=='\n')
    {
                buf[size_buf]='\0';
    }

    msg.header.vrsn = VERSION;
    msg.header.type = SEND;
    msg.attribute[0].type = ATTR_MESSAGE;
    msg.attribute[0].length = ( 4 + size_buf );

    bzero((char*)&msg.attribute[0].payload,sizeof(msg.attribute[0].payload));
    bzero((char*)&msg.attribute[1].payload,sizeof(msg.attribute[1].payload));

        for ( i = 0; i < size_buf; i++ )
        {
        msg.attribute[0].payload[i] = buf[i];
        }

        // We should consider sizeof attribute[0] and attribute[1]
        // So, 8 is default value for type and length field, plus consider attribute[0] payload
```

```
		msg.header.length = ( 8 + size_buf );

		if (send(sockfd,&msg,sizeof(msg),0)<0)
	{
			printf("[Error] Client : Failed to write to socket\n");
		}
}

void IdleMsg(int sockfd)
{
	Message msg;
	int size_buf;

	msg.header.vrsn = VERSION;
	msg.header.type = IDLE;
	msg.attribute[0].type = ATTR_MESSAGE;
	msg.attribute[0].length = 4;

	bzero((char*)&msg.attribute[0].payload,sizeof(msg.attribute[0].payload));
	bzero((char*)&msg.attribute[1].payload,sizeof(msg.attribute[1].payload));

		//IdleMsg doesn't send any payload
		msg.header.length = 8;

		if (send(sockfd,&msg,sizeof(msg),0)<0)
	{
			printf("[Error] Client : Failed to write to socket\n");
		}
}

//read message from the server
int RecvMsg(int sockfd)
{
	Message msg;
	int status = 0;
	if(recv(sockfd, (Message *) &msg, sizeof(msg),0) < 0)
		{
			printf("[Error] Client : Failed to receive data from server");
		}

		//If header or attribute type is wrong, discard it
		switch(msg.header.type)
		{
			case 3:
				if(msg.attribute[0].type == ATTR_MESSAGE && msg.attribute[1].type ==
ATTR_USERNAME)
				{
					printf("[MSG] Client : FWD Message[%s] from Username [%s] \n",
msg.attribute[0].payload, msg.attribute[1].payload);
					status=0;
				}
				break;

			case 5:
				if( msg.attribute[0].type == ATTR_REASON)
				{
					printf("[MSG] Client : NAK Message[%s] \n", msg.attribute[0].payload);
					status=1;
				}
				break;
			case 6:
				if( msg.attribute[0].type == ATTR_USERNAME)
				{
					printf("[MSG] Client : OFFLINE Message[%s] is now offline. \n",
msg.attribute[0].payload);
```

```
                                        status=0;
                                }
                                break;
                        case 7:
                                if(msg.attribute[0].type == ATTR_CLIENT_COUNT && msg.attribute[1].type ==
ATTR_USERNAME)
                                {
                                        printf("[MSG] Client : ACK Message[ %s conntion] - Username(s) [    %s]
\n", msg.attribute[0].payload, msg.attribute[1].payload);
                                        status=0;
                                }
                                break;
                        case 8:
                                if( msg.attribute[0].type == ATTR_USERNAME)
                                {
                                        printf("[MSG] Client : ONLINE Message - Username [%s] newly joined
chat room \n", msg.attribute[0].payload);

                                        status=0;
                                }
                                break;
                        case 9:
                                if( msg.attribute[0].type == ATTR_USERNAME)
                                {
                                        printf("[MSG] Client : IDLE Message - Username [%s] is IDLE status \n",
msg.attribute[0].payload);

                                        status=0;
                                }
                                break;

                        default:
                                //The client should discard any message that is not understood
                                break;
                }
        return status;
}

int main(int argc, char **argv)
{
                int                                             sockfd;
                int                                     connfd;
                struct sockaddr_in      servaddr;
                //for IPv6
                struct hostent* hret;
                fd_set readfd;
        int fdmax;
                char username[MAX_MESSAGE];
                int status = 0;
                int Time_Check = 0;

                if (argc != 4)
                {
                        printf("[Error] Client : Follow those command - ./Client <Username> <Server_IP> <Server_Port>
\n");
                        printf("[Error] Client : Program Exit");
                        return 0;
                }

                strcpy(username, argv[1]);
                if(strlen(username) > 16)
                {
                        printf("[Error] Client : Username should be less or qual to 16\n");
                        return 0;
                }
```

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
hret = gethostbyname(argv[2]);
memcpy(&servaddr.sin_addr.s_addr, hret->h_addr, hret->h_length);
///Input Port data
servaddr.sin_port = htons(atoi(argv[3]));

connfd = connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
if (connfd < 0)
{
        printf("[Error] Client : Failed to connect\n");
}

status = joinServer(username,sockfd,strlen(username));
if( status == 1)
{
        close(sockfd);
printf("[Error] Client : Client close due to fail joining server\n");
        return 0;
}
//Clear the socket set
FD_ZERO(&readfd);
FD_SET(0,&readfd);
FD_SET(sockfd,&readfd);


for(;;)
{
        //For IDLE Message
        struct timeval tv;
        tv.tv_sec = 0;
        tv.tv_usec = 100000;

        fdmax = sockfd;
        if (select(fdmax+1,&readfd,NULL,NULL,&tv)<0)
        {
                printf("[Error] Client : Failed to select\n");
        }
        else
        {

                if (FD_ISSET(sockfd,&readfd))
                {
                        if(RecvMsg(sockfd) ==1)
                        {
                                close(sockfd);
                                break;
                        }
                }

                if (FD_ISSET(0,&readfd))
                {
                        SendMsg(sockfd);
                        Time_Check = 0;
                }

                //IDLE status is more than 10s
                //It only sends one time after this client is free
                if( Time_Check == 100)
                {
                        IdleMsg(sockfd);
                }
                Time_Check++;
        }
```

```
                        FD_SET(0,&readfd);
                        FD_SET(sockfd,&readfd);
                }

                close(sockfd);
        printf("[Info] Client : Client close\n");

                return 0;
}
```

unp.h

==========

```
/* include unph */
/* Our own header.    Tabs are set for 4 spaces, not 8 */

#ifndef                 __unp_h
#define                 __unp_h

#include                "config.h"              /* configuration options for current OS */
                                                                                        /* "../config.h" is generated by
configure */

/* If anything changes in the following list of #includes, must change
     acsite.m4 also, for configure's tests. */

#include                <sys/types.h>           /* basic system data types */
#include                <sys/socket.h>          /* basic socket definitions */
#if TIME_WITH_SYS_TIME
#include                <sys/time.h>            /* timeval{} for select() */
#include                <time.h>                /* timespec{} for pselect() */
#else
#if HAVE_SYS_TIME_H
#include                <sys/time.h>            /* includes <time.h> unsafely */
#else
#include                <time.h>                /* old system? */
#endif
#endif
#include                <netinet/in.h>          /* sockaddr_in{} and other Internet defns */
#include                <arpa/inet.h>           /* inet(3) functions */
#include                <errno.h>
#include                <fcntl.h>               /* for nonblocking */
#include                <netdb.h>
#include                <signal.h>
#include                <stdio.h>
#include                <stdlib.h>
#include                <string.h>
#include                <sys/stat.h>            /* for S_xxx file mode constants */
#include                <sys/uio.h>                     /* for iovec{} and readv/writev */
#include                <unistd.h>
#include                <sys/wait.h>
#include                <sys/un.h>                      /* for Unix domain sockets */

#ifdef                  HAVE_SYS_SELECT_H
# include               <sys/select.h>          /* for convenience */
#endif

#ifdef                  HAVE_SYS_SYSCTL_H
#ifdef                  HAVE_SYS_PARAM_H
# include               <sys/param.h>           /* OpenBSD prereq for sysctl.h */
```

```
#endif
# include       <sys/sysctl.h>
#endif

#ifdef          HAVE_POLL_H
# include       <poll.h>        /* for convenience */
#endif

#ifdef          HAVE_SYS_EVENT_H
# include       <sys/event.h>   /* for kqueue */
#endif

#ifdef          HAVE_STRINGS_H
# include       <strings.h>             /* for convenience */
#endif

/* Three headers are normally needed for socket/file ioctl's:
 * <sys/ioctl.h>, <sys/filio.h>, and <sys/sockio.h>.
 */
#ifdef          HAVE_SYS_IOCTL_H
# include       <sys/ioctl.h>
#endif
#ifdef          HAVE_SYS_FILIO_H
# include       <sys/filio.h>
#endif
#ifdef          HAVE_SYS_SOCKIO_H
# include       <sys/sockio.h>
#endif

#ifdef          HAVE_PTHREAD_H
# include       <pthread.h>
#endif

#ifdef HAVE_NET_IF_DL_H
# include       <net/if_dl.h>
#endif

#ifdef HAVE_NETINET_SCTP_H
#include        <netinet/sctp.h>
#endif

/* OSF/1 actually disables recv() and send() in <sys/socket.h> */
#ifdef          __osf__
#undef          recv
#undef          send
#define         recv(a,b,c,d)   recvfrom(a,b,c,d,0,0)
#define         send(a,b,c,d)   sendto(a,b,c,d,0,0)
#endif

#ifndef         INADDR_NONE
/* $$.Ic INADDR_NONE$$ */
#define         INADDR_NONE     0xffffffff  /* should have been in <netinet/in.h> */
#endif

#ifndef         SHUT_RD                                 /* these three POSIX names are new */
#define         SHUT_RD                 0       /* shutdown for reading */
#define         SHUT_WR                 1       /* shutdown for writing */
#define         SHUT_RDWR       2       /* shutdown for reading and writing */
/* $$.Ic SHUT_RD$$ */
/* $$.Ic SHUT_WR$$ */
/* $$.Ic SHUT_RDWR$$ */
#endif

/* *INDENT-OFF* */
#ifndef INET_ADDRSTRLEN
/* $$.Ic INET_ADDRSTRLEN$$ */
```

```
#define         INET_ADDRSTRLEN             16      /* "ddd.ddd.ddd.ddd\0"

1234567890123456 */
#endif

/* Define following even if IPv6 not supported, so we can always allocate
     an adequately sized buffer without #ifdefs in the code. */
#ifndef INET6_ADDRSTRLEN
/* $$.Ic INET6_ADDRSTRLEN$$ */
#define         INET6_ADDRSTRLEN    46      /* max size of IPv6 address string:
                                               "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx" or
                                               "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:ddd.ddd.ddd.ddd\0"
                                                1234567890123456789012345678901234567890123456 */
#endif
/* *INDENT-ON* */

/* Define bzero() as a macro if it's not in standard C library. */
#ifndef         HAVE_BZERO
#define         bzero(ptr,n)            memset(ptr, 0, n)
/* $$.If bzero$$ */
/* $$.If memset$$ */
#endif

/* Older resolvers do not have gethostbyname2() */
#ifndef         HAVE_GETHOSTBYNAME2
#define         gethostbyname2(host,family)     gethostbyname((host))
#endif

/* The structure returned by recvfrom_flags() */
struct unp_in_pktinfo {
   struct in_addr       ipi_addr; /* dst IPv4 address */
   int                                  ipi_ifindex;/* received interface index */
};
/* $$.It unp_in_pktinfo$$ */
/* $$.Ib ipi_addr$$ */
/* $$.Ib ipi_ifindex$$ */

/* We need the newer CMSG_LEN() and CMSG_SPACE() macros, but few
     implementations support them today.    These two macros really need
       an ALIGN() macro, but each implementation does this differently. */
#ifndef         CMSG_LEN
/* $$.Im CMSG_LEN$$ */
#define         CMSG_LEN(size)          (sizeof(struct cmsghdr) + (size))
#endif
#ifndef         CMSG_SPACE
/* $$.Im CMSG_SPACE$$ */
#define         CMSG_SPACE(size)        (sizeof(struct cmsghdr) + (size))
#endif

/* POSIX requires the SUN_LEN() macro, but not all implementations DefinE
     it (yet).    Note that this 4.4BSD macro works regardless whether there is
     a length field or not. */
#ifndef         SUN_LEN
/* $$.Im SUN_LEN$$ */
# define        SUN_LEN(su) \
                (sizeof(*(su)) - sizeof((su)->sun_path) + strlen((su)->sun_path))
#endif

/* POSIX renames "Unix domain" as "local IPC."
     Not all systems DefinE AF_LOCAL and PF_LOCAL (yet). */
#ifndef         AF_LOCAL
#define AF_LOCAL    AF_UNIX
#endif
#ifndef         PF_LOCAL
#define PF_LOCAL    PF_UNIX
#endif
```

```
/* POSIX requires that an #include of <poll.h> DefinE INFTIM, but many
     systems still DefinE it in <sys/stropts.h>.  We don't want to include
     all the STREAMS stuff if it's not needed, so we just DefinE INFTIM here.
     This is the standard value, but there's no guarantee it is -1. */
#ifndef INFTIM
#define INFTIM          (-1)    /* infinite poll timeout */
/* $$.Ic INFTIM$$ */
#ifdef  HAVE_POLL_H
#define INFTIM_UNPH                                     /* tell unpxti.h we defined it */
#endif
#endif

/* Following could be derived from SOMAXCONN in <sys/socket.h>, but many
     kernels still #define it as 5, while actually supporting many more */
#define LISTENQ                 1024    /* 2nd argument to listen() */

/* Miscellaneous constants */
#define MAXLINE                 4096    /* max text line length */
#define BUFFSIZE        8192    /* buffer size for reads and writes */

/* Define some port number that can be used for our examples */
#define SERV_PORT                9877                   /* TCP and UDP */
#define SERV_PORT_STR "9877"                            /* TCP and UDP */
#define UNIXSTR_PATH  "/tmp/unix.str"           /* Unix domain stream */
#define UNIXDG_PATH             "/tmp/unix.dg"          /* Unix domain datagram */
/* $$.ix [LISTENQ]~constant,~definition~of$$ */
/* $$.ix [MAXLINE]~constant,~definition~of$$ */
/* $$.ix [BUFFSIZE]~constant,~definition~of$$ */
/* $$.ix [SERV_PORT]~constant,~definition~of$$ */
/* $$.ix [UNIXSTR_PATH]~constant,~definition~of$$ */
/* $$.ix [UNIXDG_PATH]~constant,~definition~of$$ */

/* Following shortens all the typecasts of pointer arguments: */
#define SA      struct sockaddr

#ifndef HAVE_STRUCT_SOCKADDR_STORAGE
/*
  * RFC 3493: protocol-independent placeholder for socket addresses
  */
#define __SS_MAXSIZE    128
#define __SS_ALIGNSIZE  (sizeof(int64_t))
#ifdef HAVE_SOCKADDR_SA_LEN
#define __SS_PAD1SIZE   (__SS_ALIGNSIZE - sizeof(u_char) - sizeof(sa_family_t))
#else
#define __SS_PAD1SIZE   (__SS_ALIGNSIZE - sizeof(sa_family_t))
#endif
#define __SS_PAD2SIZE   (__SS_MAXSIZE - 2*__SS_ALIGNSIZE)

struct sockaddr_storage {
#ifdef HAVE_SOCKADDR_SA_LEN
        u_char          ss_len;
#endif
        sa_family_t     ss_family;
        char            __ss_pad1[__SS_PAD1SIZE];
        int64_t         __ss_align;
        char            __ss_pad2[__SS_PAD2SIZE];
};
#endif

#define FILE_MODE       (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
                                        /* default file access permissions for new files */
#define DIR_MODE        (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)
                                        /* default permissions for new directories */

typedef void    Sigfunc(int);           /* for signal handlers */
```

```
#define                 min(a,b)  ((a) < (b) ? (a) : (b))
#define                 max(a,b) ((a) > (b) ? (a) : (b))

#ifndef                 HAVE_ADDRINFO_STRUCT
# include               "../lib/addrinfo.h"
#endif

#ifndef                 HAVE_IF_NAMEINDEX_STRUCT
struct if_nameindex {
   unsigned int     if_index;    /* 1, 2, ... */
   char              *if_name;     /* null-terminated name: "le0", ... */
};
/* $$.It if_nameindex$$ */
/* $$.Ib if_index$$ */
/* $$.Ib if_name$$ */
#endif

#ifndef                 HAVE_TIMESPEC_STRUCT
struct timespec {
   time_t               tv_sec;             /* seconds */
   long                         tv_nsec;    /* and nanoseconds */
};
/* $$.It timespec$$ */
/* $$.Ib tv_sec$$ */
/* $$.Ib tv_nsec$$ */
#endif
/* end unph */

                                        /* prototypes for our own library functions */
int                             connect_nonb(int, const SA *, socklen_t, int);
int                             connect_timeo(int, const SA *, socklen_t, int);
int                      daemon_init(const char *, int);
void                     daemon_inetd(const char *, int);
void                     dg_cli(FILE *, int, const SA *, socklen_t);
void                     dg_echo(int, SA *, socklen_t);
int                             family_to_level(int);
char                    *gf_time(void);
void                     heartbeat_cli(int, int, int);
void                     heartbeat_serv(int, int, int);
struct addrinfo *host_serv(const char *, const char *, int, int);
int                             inet_srcrt_add(char *);
u_char    *inet_srcrt_init(int);
void                     inet_srcrt_print(u_char *, int);
void                     inet6_srcrt_print(void *);
char      **my_addrs(int *);
int                             readable_timeo(int, int);
ssize_t                  readline(int, void *, size_t);
ssize_t                  readn(int, void *, size_t);
ssize_t                  read_fd(int, void *, size_t, int *);
ssize_t                  recvfrom_flags(int, void *, size_t, int *, SA *, socklen_t *,
                                struct unp_in_pktinfo *);
Sigfunc *signal_intr(int, Sigfunc *);
int                             sock_bind_wild(int, int);
int                             sock_cmp_addr(const SA *, const SA *, socklen_t);
int                             sock_cmp_port(const SA *, const SA *, socklen_t);
int                             sock_get_port(const SA *, socklen_t);
void                    sock_set_addr(SA *, socklen_t, const void *);
void                    sock_set_port(SA *, socklen_t, int);
void                    sock_set_wild(SA *, socklen_t);
char                    *sock_ntop(const SA *, socklen_t);
char                    *sock_ntop_host(const SA *, socklen_t);
int                             sockfd_to_family(int);
void                     str_echo(int);
void                     str_cli(FILE *, int);
int                             tcp_connect(const char *, const char *);
```

```
int                     tcp_listen(const char *, const char *, socklen_t *);
void                    tv_sub(struct timeval *, struct timeval *);
int                     udp_client(const char *, const char *, SA **, socklen_t *);
int                     udp_connect(const char *, const char *);
int                     udp_server(const char *, const char *, socklen_t *);
int                     writable_timeo(int, int);
ssize_t                 writen(int, const void *, size_t);
ssize_t                 write_fd(int, void *, size_t, int);

#ifdef                  MCAST
int                     mcast_leave(int, const SA *, socklen_t);
int                     mcast_join(int, const SA *, socklen_t, const char *, u_int);
int                     mcast_leave_source_group(int sockfd, const SA *src, socklen_t srclen,
                                                 const SA *grp,
socklen_t grplen);
int                     mcast_join_source_group(int sockfd, const SA *src, socklen_t srclen,
                                                const SA *grp,
socklen_t grplen,
                                                const char *ifname,
u_int ifindex);
int                     mcast_block_source(int sockfd, const SA *src, socklen_t srclen,
                                           const SA *grp, socklen_t
grplen);
int                     mcast_unblock_source(int sockfd, const SA *src, socklen_t srclen,
                                             const SA *grp, socklen_t
grplen);
int                     mcast_get_if(int);
int                     mcast_get_loop(int);
int                     mcast_get_ttl(int);
int                     mcast_set_if(int, const char *, u_int);
int                     mcast_set_loop(int, int);
int                     mcast_set_ttl(int, int);

void                    Mcast_leave(int, const SA *, socklen_t);
void                    Mcast_join(int, const SA *, socklen_t, const char *, u_int);
void                    Mcast_leave_source_group(int sockfd, const SA *src, socklen_t srclen,
                                                 const SA *grp,
socklen_t grplen);
void                    Mcast_join_source_group(int sockfd, const SA *src, socklen_t srclen,
                                                const SA *grp,
socklen_t grplen,
                                                const char *ifname,
u_int ifindex);
void                    Mcast_block_source(int sockfd, const SA *src, socklen_t srclen,
                                           const SA *grp, socklen_t
grplen);
void                    Mcast_unblock_source(int sockfd, const SA *src, socklen_t srclen,
                                             const SA *grp, socklen_t
grplen);
int                     Mcast_get_if(int);
int                     Mcast_get_loop(int);
int                     Mcast_get_ttl(int);
void                    Mcast_set_if(int, const char *, u_int);
void                    Mcast_set_loop(int, int);
void                    Mcast_set_ttl(int, int);
#endif

uint16_t                in_cksum(uint16_t *, int);

#ifndef                 HAVE_GETADDRINFO_PROTO
int                     getaddrinfo(const char *, const char *, const struct addrinfo *,
                                    struct addrinfo **);
void                    freeaddrinfo(struct addrinfo *);
char                    *gai_strerror(int);
#endif
```

```
#ifndef		HAVE_GETNAMEINFO_PROTO
int			getnameinfo(const SA *, socklen_t, char *, size_t, char *, size_t, int);
#endif

#ifndef		HAVE_GETHOSTNAME_PROTO
int			gethostname(char *, int);
#endif

#ifndef		HAVE_HSTRERROR_PROTO
const char	*hstrerror(int);
#endif

#ifndef		HAVE_IF_NAMETOINDEX_PROTO
unsigned int	 if_nametoindex(const char *);
char			*if_indextoname(unsigned int, char *);
void			 if_freenameindex(struct if_nameindex *);
struct if_nameindex *if_nameindex(void);
#endif

#ifndef		HAVE_INET_PTON_PROTO
int			inet_pton(int, const char *, void *);
const char	*inet_ntop(int, const void *, char *, size_t);
#endif

#ifndef		HAVE_INET_ATON_PROTO
int			inet_aton(const char *, struct in_addr *);
#endif

#ifndef		HAVE_PSELECT_PROTO
int			pselect(int, fd_set *, fd_set *, fd_set *,
				const struct timespec *, const sigset_t *);
#endif

#ifndef		HAVE_SOCKATMARK_PROTO
int			sockatmark(int);
#endif

#ifndef		HAVE_SNPRINTF_PROTO
int			snprintf(char *, size_t, const char *, ...);
#endif

				/* prototypes for our own library wrapper functions */
void		Connect_timeo(int, const SA *, socklen_t, int);
int			Family_to_level(int);
struct addrinfo *Host_serv(const char *, const char *, int, int);
const char		*Inet_ntop(int, const void *, char *, size_t);
void			 Inet_pton(int, const char *, void *);
char			*If_indextoname(unsigned int, char *);
unsigned int		If_nametoindex(const char *);
struct if_nameindex	*If_nameindex(void);
char	**My_addrs(int *);
ssize_t		Read_fd(int, void *, size_t, int *);
int			Readable_timeo(int, int);
ssize_t		Recvfrom_flags(int, void *, size_t, int *, SA *, socklen_t *,
			struct unp_in_pktinfo *);
Sigfunc *Signal(int, Sigfunc *);
Sigfunc *Signal_intr(int, Sigfunc *);
int			Sock_bind_wild(int, int);
char		*Sock_ntop(const SA *, socklen_t);
char		*Sock_ntop_host(const SA *, socklen_t);
int			Sockfd_to_family(int);
int			Tcp_connect(const char *, const char *);
int			Tcp_listen(const char *, const char *, socklen_t *);
int			Udp_client(const char *, const char *, SA **, socklen_t *);
int			Udp_connect(const char *, const char *);
int			Udp_server(const char *, const char *, socklen_t *);
```

```
ssize_t         Write_fd(int, void *, size_t, int);
int             Writable_timeo(int, int);

                        /* prototypes for our Unix wrapper functions: see {Sec errors} */
void           *Calloc(size_t, size_t);
void            Close(int);
void            Dup2(int, int);
int             Fcntl(int, int, int);
void            Gettimeofday(struct timeval *, void *);
int             Ioctl(int, int, void *);
pid_t           Fork(void);
void           *Malloc(size_t);
int             Mkstemp(char *);
void           *Mmap(void *, size_t, int, int, int, off_t);
int             Open(const char *, int, mode_t);
void            Pipe(int *fds);
ssize_t         Read(int, void *, size_t);
void            Sigaddset(sigset_t *, int);
void            Sigdelset(sigset_t *, int);
void            Sigemptyset(sigset_t *);
void            Sigfillset(sigset_t *);
int             Sigismember(const sigset_t *, int);
void            Sigpending(sigset_t *);
void            Sigprocmask(int, const sigset_t *, sigset_t *);
char           *Strdup(const char *);
long            Sysconf(int);
void            Sysctl(int *, u_int, void *, size_t *, void *, size_t);
void            Unlink(const char *);
pid_t           Wait(int *);
pid_t           Waitpid(pid_t, int *, int);
void            Write(int, void *, size_t);

                        /* prototypes for our stdio wrapper functions: see {Sec errors} */
void            Fclose(FILE *);
FILE           *Fdopen(int, const char *);
char           *Fgets(char *, int, FILE *);
FILE           *Fopen(const char *, const char *);
void            Fputs(const char *, FILE *);

                        /* prototypes for our socket wrapper functions: see {Sec errors} */
int             Accept(int, SA *, socklen_t *);
void            Bind(int, const SA *, socklen_t);
void            Connect(int, const SA *, socklen_t);
void            Getpeername(int, SA *, socklen_t *);
void            Getsockname(int, SA *, socklen_t *);
void            Getsockopt(int, int, int, void *, socklen_t *);
#ifdef          HAVE_INET6_RTH_INIT
int             Inet6_rth_space(int, int);
void           *Inet6_rth_init(void *, socklen_t, int, int);
void            Inet6_rth_add(void *, const struct in6_addr *);
void            Inet6_rth_reverse(const void *, void *);
int             Inet6_rth_segments(const void *);
struct in6_addr *Inet6_rth_getaddr(const void *, int);
#endif
#ifdef          HAVE_KQUEUE
int             Kqueue(void);
int             Kevent(int, const struct kevent *, int,
                       struct kevent *, int, const struct timespec *);
#endif
void            Listen(int, int);
#ifdef          HAVE_POLL
int             Poll(struct pollfd *, unsigned long, int);
#endif
ssize_t         Readline(int, void *, size_t);
ssize_t         Readn(int, void *, size_t);
ssize_t         Recv(int, void *, size_t, int);
```

```
ssize_t         Recvfrom(int, void *, size_t, int, SA *, socklen_t *);
ssize_t         Recvmsg(int, struct msghdr *, int);
int                     Select(int, fd_set *, fd_set *, fd_set *, struct timeval *);
void            Send(int, const void *, size_t, int);
void            Sendto(int, const void *, size_t, int, const SA *, socklen_t);
void            Sendmsg(int, const struct msghdr *, int);
void            Setsockopt(int, int, int, const void *, socklen_t);
void            Shutdown(int, int);
int                     Sockatmark(int);
int                     Socket(int, int, int);
void            Socketpair(int, int, int, int *);
void            Writen(int, void *, size_t);

void            err_dump(const char *, ...);
void            err_msg(const char *, ...);
void            err_quit(const char *, ...);
void            err_ret(const char *, ...);
void            err_sys(const char *, ...);

#endif          /* __unp_h */
```