Cpr E 489
Computer Networking and Data Communications
Assignment 11

Objectives

- Familiarize you with Berkeley sockets.
- Develop a simple chat application using sockets.

The code is located on the class web page. You have to do this assignment individually.

1 Preliminaries

In this assignment, we switch our attention from data link layer protocols to transport and application layer protocols.

The Berkeley socket interface is a prevalent application program interface to the transport and network layers in Unix. It supports TCP/IP, Xerox NS, Unix domain, and possibly other protocols. TCP/IP is a suit of protocols consisting of TCP (Transmission Control Protocol, a reliable connection oriented transport protocol) UDP (User Datagram Protocol, an unreliable connectionless network protocol), and IP (Internet Protocol, an unreliable connectionless network protocol. TCP and UDP are built on IP. In this assignment, we will focus on UDP, on which we will build a simple chat application allowing multiple users to talk to one another on the Internet (similar to IRC).

1.1 Clients and Server

Internet communications are often done in a client-server fashion. A server awaits message from the clients and provides services for the clients. For example, when you remote log on a machine, the local machine is the client and the remote machine is the server. Similarly when you send an email message, you are establishing a connection between the local machine (the client) and a server process on the destination machine. Server processes in Unix are often called daemon processes. So there are mail daemon, finger daemon, talk daemon, etc.

In the chat application that we are going to develop, client talk with one another through the server. That is, a client sends a message to the server, who in turn sends the message to *all* the clients. The server keeps track of a list of clients. Details will be explained later.

1.2 Addresses

To communicate you have to specify who you wish to communicate with. Each machine in the Internet has a unique address, e.g. eng3.iastate.edu. However, an Internet address by itself is not enough. For example, there are a mail daemon and a finger daemon on eng3. They reside on the same machine and thus have the same Internet address. Your mail program has to specify that it wants to talk with the mail daemon and not with the finger daemon on the target machine. The way to accomplish this is to use *port numbers.*

The mail daemon and the talk daemon are listening to different ports: the mail daemon listents to port 25, while the talk daemon listens to port 517. These ports are *well known*, so that the application programs know to which port they should send the messages. On most Unix systems, the file /etc/services lists all the well known port numbers.

1.3 Sockets

We only describe some aspects of sockets that are necessary for this assignment. In particular, we restrict ourselves to connectionless protocols like UDP.

Sockets are an abstraction of communication endpoints. You can send and receive messages through sockets. Each socket has a number, or a *socket descriptor*, that identifies it (this is analogous to a file descriptor in Unix). Each internet socket also has an Internet socket address, which is simply an Internet address plus a port number. An internet socket address has type struct socketaddr_in.

The socket() system call creates a socket, and the bind() system call associates an address with the socket. You don't need to understand these two system calls to do the assignment, so we will skip the details here. Messages are sent and received by the sendto() and recvfrom() system calls respectively. These are the functions that you will be using for this assignment. They are analogous to to_physical_layer() and from_physical_layer() which you are used to. If the description below are not clear you can also read the online manual by typing man sendto and man recvfrom.

1.3.1 Send a Message

int sendto(int em sockfd, char **msg*, int *len*, int *flags*, struct sockaddr_in **dest*, int *dest_len*);

sockfd is the socket descriptor through which you want to send the message. *msg* is a pointer to the message, while *len* is the length of the message in bytes. *flags* is always 0 for the purpose of this assignment. *dest* is the destination socket address, and finally *dest_len* is the length of the socket address i.e. sizeof(**dest*). So the statement

sendto(sender_socket, "hello", 6, 0, dest_addr, sizeof(*dest_addr));

sends a string "hello" to the destination dest_addr using the socket sender_socket. (The length of message is 6 bytes instead of 5 because of invisible end of string character.)

How do you know what the destination socket address is? We will explain this later. sendto() returns -1 when there is an error which is detected locally (e.g. *sockfd* is invalid). Note the sendto() cannot detect lost or damaged messages.

1.3.2 Receive a Message

int recvfrom(int *sockfd*, char **msg*, int *len*, int *flags*, struct sockaddr_in **source*, int **source_len*);

sockfd is the descriptor of the socket from which you want to receive a message. *msg* points to a message buffer and *len* indicates the length of the buffer in bytes. Again *flags* is 0 for our purpose. *source* either is NULL or points to a buffer for a socker address, and *source_len* points to an integer specifying the length of this buffer.

The function waits for a message to arrive. (This means a socket is blocking. It is possible to make it non_blocking where it does not wait for the message and returns immediately.) When the function returns, *msg* is filled with the received message, and **source_len* will indicate the length of the socket address. recvfrom() returns the length of the received message in bytes. For example:

char msg[8]; struct socketaddr_in source_addr; int source_len,n;

source_len = sizeof(source_addr); n = recvfrom(receiver_socket, msg, 8, 0, &source_addr, &source_len);

After the function returns, msg will be "hello" and n will be 6, if receiver socket corresponds to dest_addr above.

2 A Chat Application

2.1 How to use it

The chat application that you will be building consists of two executables, server and chat (which is the client). Part of the code has already been written for you, and you are supposed to fill in some functions. The executables of the solutions are provided in the same directory and they are called 489server and 489chat. The syntax of the server is:

489server *server_port*

where *server_port* is the port number that the server process will be using. In the real application, the port number is often hardcoded in the program and is well-known. For this assignment, you can pick a number between 5001 and 65535. If you get an error message:

489server: cannot bind local address: Address already in use

that means somebody else (probably one of your classmates) is using that port number. Just pick another one. Suppose you have started a server on eng3 with port number 6000.

The syntax of the client is:

489chat *server_hostname server_port user_name*

where server_hostname is the name of the machine that the server is running, server_port is the port number that the server is using, and user_name is an arbitrarily chosen name. The port number of the client is automatically selected by the operating system and you don't need to specify it. (the reason for asymmetry is that the client has to know the server's port number before it can send the message to the server. However when the server gets the client's message, it can tell the client's socket address by the recvfrom() system call, so the server does not need to know the client's port number is advance.)

As an example, you are on bart:

489chat eng3 6000 Pete

Note that if you type a wrong port number, no error message will be printed out; it simply sends a message to a port that no servers are listening.

To run the program outside the particular domain, specify the domain to which the address belong. For the example shown above you may need to use eng3.iastate.edu instead of eng3 as you are running the client on bart.ee.iastate.edu. Or suppose if you server is running on some machine abc in cs department then run it as,

489chat abc.cs.iastate.edu 6000 Bob

Now Pete and Bob can talk with one another and others can also join then by invoking the chat program and specifying their name. Press Control-D when a client is done. Up to 16 clients can connect to the same server. (16 is just an arbitrary number hardcoded in the program.)

2.2 How it works

Figure 1 depicts what happens when Pete and Bob join, and Pete subsequently types "Hello!".

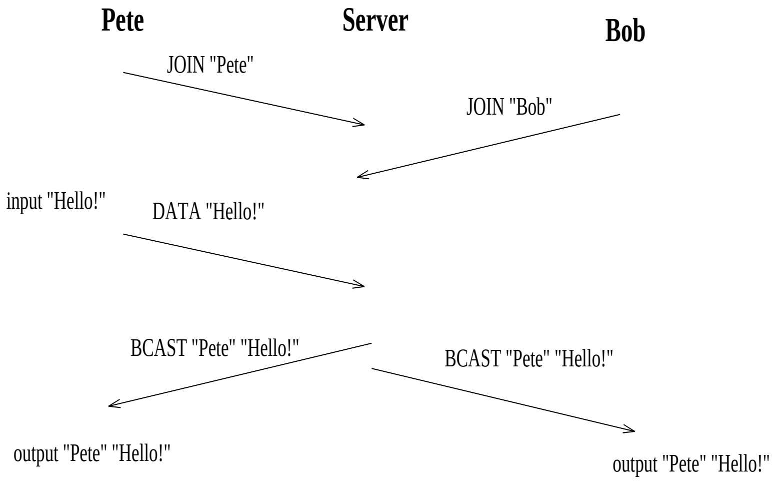

Figure 1: What happens when Pete types "Hello!" (BCAST stands for BROADCAST message)

Currently there are four types of messages sent between the server and clients, JOIN, DATA, BROADCAST and LEAVE. Each message is tagged with a number representing it's type. Message of the same type has equal lengths; this just makes the code a bit easier to write. When a client starts to run, it send a JOIN message to the server. A JOIN message has the form:

MSG_TYPE_JOIN	*user_name*

Figure 2: Format of JOIN message

where MSG_TYPE_JOIN is a constant, and *user_name* is the user's name got from the command line. The first field tells the server that it's a JOIN message. The *user_name* field is always 32 bytes long even if the user's name is shorter, so that all JOIN messages have the same length.

The server maintains a table of clients in an array. (In real application, this should probably be a hash table to support a large number of clients.) Each entry in the table is a client's name and its socket address. Upon receiving the JOIN message, the server adds the client to the table.

When a user has something to say, a DATA message is sent to the server. A DATA message has the form:

MSG_TYPE_DATA	*data*

Figure 3: Format of DATA message

where again MSG_TYPE_DATA is some constant (different from MSG_TYPE_JOIN) and *data* is the string that user typed. The *data* field is always 256 bytes long. When the server receives the DATA message, it first checks whether the sender is in the client table. If not, it will discard the message. Otherwise, it goes through each entry in the client table, and sends each client a BROADCAST message:

MSG_TYPE_BROADCAST	*user_name*	*data*

Figure 4: Format of BROADCAST message

where a *user_name* is the name of the user that sent the DATA message. When a client receives the BROADCAST message, it simply prints out *user_name* and data.

Finally, when a client quits, it sends to the server a LEAVE message of the form:

MSG_TYPE_LEAVE

Figure 5: Format of LEAVE message

That is, the message only contains a number. Upon receipt of LEAVE message, the server removes the client's entry from the table.

3 The Assignment

3.1 Part 1

There are four files, chat.h, server.c, chat.c and common.c.

- server.c and chat.c implements the server and client respectively.
- common.c contains the functions which may be used by either the server or the client.
- chat.h is a header file used by the three .c files. In particular, it defines the message types.

Implement the functions that are left out in server.c and chat.c:

1. send_data() and send_leave() in chat.c, which sends a DATA and a LEAVE message to the server respectively.
2. process_message_from_server() in chat.c which checks whether a broadcast message from the server to the has arrived and if so, prints out the message.
3. part of process_data() in server.c which sends a BROADCAST message to all the clients.

The executables of your server and clients will be called server and chat respectively. Recall that the executables of the solutions are given to you as 489server and 489chat. We suggest you concentrate on writing chat first so that it works with 489server, and once it works, write server.

3.2 Part 2

Recall that the application is built on UDP, which is unreliable. (However, as you will find out, UDP is often pretty reliable in a local area network. The most common problem of UDP is packet loss due to buffer overflow, which is unlikely to happen is our chat application. Nevertheless, imagine some messages are lost.)

It is ok for some DATA or BROADCAST messages to get lost, because such errors are transient. Loss of LEAVE message will cause the server to believe a client is still active even though it has in fact left. So some extraneous BROADCAST messages will be wasted. But we can live that.

A more serious problem is the loss of a JOIN message. The client thinks it has joined the group, but the server does not get the message. So DATA messages sent by the client will be discarded by the server, and the server will not send BROADCAST message to the client. Extend your programs to make the JOIN message reliable. To accomplish this, you need to change both the server and the client:

- Change the server so that it sends an ack to the client when it receives a JOIN message.
- Change the client so that it can time out. The pseudo code is as follows:

```
repeat
    send a JOIN message to the server
    wait at most 1 second for a message
until a message has arrived and the message is an ack
```

Hint: You might want to use the socket_timeout() function defined in common.c.

In this implementation, the server may receive duplicate JOIN messages. Don't worry about it. The solution to this part is not implemented in 489server and 489chat.