CSE/EE 461
Introduction to Computer-Communication Networks
Autumn 1997
Project 2
Due electronically: 11:30 pm, Tuesday, December 9, 1997

Objectives

- Become more familiar with the Berkeley sockets API.
- Design part of an application-layer protocol using an unreliable datagram service.
- Implement a network chat application.

The code is located in the directory `/cse/courses/cse461/chat/`. You have to do this assignment individually.

1 Preliminaries

This assignment focuses on the problem of maintaining a connection using an unreliable datagram service provided by Berkeley sockets. In particular, the connection will be maintained by your program, not by the TCP/IP infrastructure. You will design a protocol to reliably open and close a connection using unreliable datagrams.

1.1 Sockets for Unreliable Messages

We only describe some aspects of sockets that are necessary for this assignment. In particular, we restrict ourselves to connectionless protocols like UDP. For this project we will not be using the connection oriented socket calls `listen()`, `accept()`, and `connect()`. Instead we will just use `socket()` and `bind()` calls to establish remote contact. The server will bind its socket to anyone who wants to talk to it, while a client will bind its socket to the server. In this application many clients can simultaneously send messages to the server. In return the server can send a message to one or all of its current clients. Messages are sent and received using the `sendto()` and `recvfrom()` socket calls, respectively. These are the functions that you will be using for this assignment. If the descriptions below are not clear, you can also read the online manual by typing `man sendto` and `man recvfrom`.

1.1.1 Send a Message

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

sockfd is the descriptor of the socket 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*`)`. For example the statements

```
 struct sockaddr_in dest_addr;

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

send a string `"hello"` to the destination `dest_addr` using the socket `sender_socket`. (The length of the message is 6 bytes instead of 5 bytes because of the 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 that `sendto()` cannot detect lost or damaged messages. In other words there is no guarantee that the message sent is actually received at the other end.

1.1.2 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 always 0 for our purpose. *source* either is `NULL` or points to a buffer for a socket address, and *source_len* points to an integer specifying the length of this buffer.

The function waits for a message to arrive.[1] When the function returns, *msg* is filled with the received message. If *source* is not `NULL`, *source* will be filled with the socket address of the source of the 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 sockaddr_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 The 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 or modify some functions. We have provided executables called `server_demo` and `chat_demo` that demonstrate what the programs should do. If you compile, you will have primitive executables `server` and `chat` that have part of the functionality of the complete solutions.

The syntax of the server is:

`server` *server_port*

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

```
server: 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 the server on orcas with port number 6000.

The syntax of the client is:

`chat` *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 simply an arbitrarily chosen name. The port number of the

[1] That is, the socket is *blocking*. It is possible to make a socket *non-blocking*, meaning that `recvfrom()` does not wait for a message to arrive and always returns immediately.

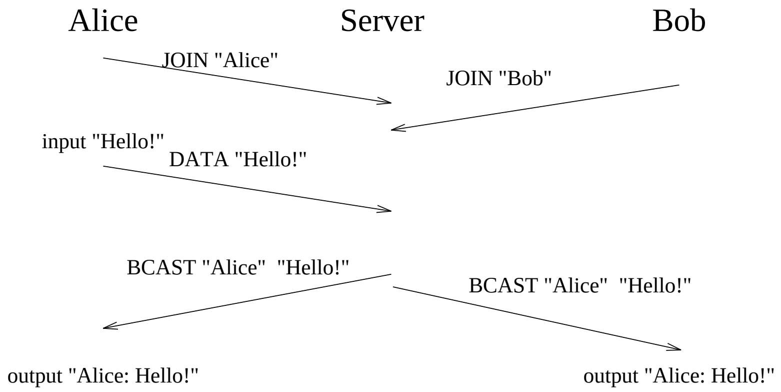

Figure 1: What happens when Alice types "Hello!" (BCAST stands for a BROADCAST message.)

client is automatically selected by the operating system and you don't need to specify it. (The reason for the asymmetry is that the client has to know the server's port number before it can send a 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 in advance.)

As an example, you can run on `sanjuan`:

```
chat orcas 6000 Alice
```

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

You can even compile and run the the chat program outside the CS domain, for example on the MSCC machines, on `homer.u`, etc. In those cases, you will need to state that orcas is in the CS domain, for example

```
chat orcas.cs 6000 Bob
```

or if you run the program outside the UW domain:

```
chat orcas.cs.washington.edu 6000 Calvin
```

Now `Alice`, `Bob` and `Calvin` can talk with one another. 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.)

In addition to being able to broadcast messages, there are two other capabilities, a command to list of all the current members of the chat group, a command to send a private message to a single member of the chat group, and a command to leave the group. These commands are entered by first typing a backslash.

- List command: `\list`
- Private message: `\p\Alice\Hello Alice, I want to tell you something private...`
- Quit command: `\quit`

As a consequence broadcast messages never begin with a backslash.

2.2 How it works

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

There are a number of types of messages that can be used in this application. For example, a message of type `msg_join` can be used for Alice and Bob to join the chat group. As you can see from the `chat.c` code a message of this type is used in the function `send_join`. As you can see in `msgs.h` we have a number of different message types defined for this project. The first field always contains a short integer indicating the type of the message. This allows the receiver of the message to determine its type easily.

The server maintains a table of clients in an array.[2] 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 two fields, the first for its `type` and the second, `data`, for its data. 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 that has three fields: `type`, `name`, and `data`. The `name` is limited to 32 bytes and the `data` to 256 bytes. String functions are used to extract the actual name and message of the chat participant.

The main function of the server is to broadcast messages sent to it. However, from time to time the server may get a message requesting a list of all chat participants, a request to send a private message, a join message, or a leave message. These are handled appropriately. Because there is a single command line for each client, we use the backslash (\) to indicate the beginning of a non-broadcast command. A simple way to handle this is to use the string function `strtok()` that supports string parsing. Please look up the manual page for `strtok()` for more details.

3 The Assignment

3.1 Part 1

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

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

In `chat.c` implement the following:

1. `send_join()` that allows the client to join the chat group. The function is now implemented in a very simple way by sending just one message to the server. If this message is lost then the join does not happen. Modify this function so that the client reliably joins the chat group. The server should send acknowledgements and there should be client timeouts to make this happen reliably. See the `socket_timeout()` function in `common.c`.
2. `send_data()` that sends a data message from client to the server. This function does not have to be reliable.
3. `process_message_from_server()` checks whether a message from the server has arrived and if so prints it. If the message is private, that should be indicated. The function `socket_ready()` defined in `common.c` will allow the client to check if there is a message without blocking on a call to `recvfrom()`.
4. `send_leave()` that, as reliably as possible, lets the client leave the chat group. Again, in this function you will have to use acknowledgements and timeouts to achieve reliability. You may want to read section 6.2.3 (pages 498-502) in Tanenbaum to get ideas.
5. `send_list_req()` that sends a request from the client to the server requesting a list of the chat participants. This is unreliable.
6. `send_pri_data()` that sends a private message to the server who will relay it to the proper client. This is unreliable.

[2] In a real application, this should probably be a hash table to support a large number of clients.

In `server.c` implement the following.

1. `process_join()` that adds the client to the chat group on the server. This function is now implemented in a very simple way. You will need to modify this function so that joining is done reliably in coordination with the client.
2. `process_data()` that accepts a message from a client and broadcasts it to all the other clients in the chat group. This is unreliable.
3. `process_leave()` that removes a client from the group. This function should be done reliably in coordination with the client.
4. `process_list_req()` that sends a list of the current chat group participants to the client that requested it. This is unreliable.
5. `process_pri_req()` that relays a private message from one client to another. This is unreliable.

3.2 Part 2 (Optional for extra credit)

Design any improvements to the user interface that you desire. You should only attempt to do this part if you have completed all of part 1. In addition it would be a good idea to check with the instructor about the design of your improvements before starting.

4 Criteria for Grading

You will want to implement all the functions. Perhaps the most difficult part is implementing the reliable join and leave functions. So it might be wise to implement everything else first. You should include comments in the code or in the README file that explains how the reliable join and leave functions are implemented.

5 Turn in

Just type `make turnin` to submit your assignment. No late assignments are accepted.