

CSC 469 / CSC 2208: Advanced Operating Systems
Assignment 3: Distributed Programming & Fault Tolerance
Client/Server Chat System

Out: November 14th, Due: 11:59 p.m., November 30, 2007

November 15, 2007

1 Introduction

In this assignment, you will implement the client side of a chat system. The chat system consists of two distributed components: chat server and chat client, which may run on different hosts in the network. We provide you the server side of the system. We also provide a skeleton of the chat client that includes a simple user interface to allow users of the chat client to type in control commands and chat messages. **You must add client support for all communication, including requesting chat server location information from a *location server*, the exchange of control messages with the server, and the display of received chat messages. Your chat client must also detect chat server failures and attempt a simple form of recovery.**

The chat server conducts a chat session in chat rooms. At any given time there may be multiple chat clients in a chat session. The chat server is responsible for managing all the chat clients in the session and distributing chat messages. A chat client starts by requesting the communication parameters (server name and port numbers) for the chat server from a *location server*. The client then registers with the chat server to join the chat session. After joining the session, a chat client can allow the user to switch to one of the chat rooms and send and receive chat messages. The communication between the client and server includes two types of messages: control messages and chat messages. The control messages will allow a chat client to perform tasks such as joining and leaving the chat session, creating a chat room, switching to a certain chat room, etc. We will only support public chat messages (that is, messages that are delivered to all users in a given chat room). To send a public chat message, a chat client must be present in one of the chat rooms. The chat client sends all its public chat messages to the chat server, and it is the chat server's job to relay these messages to all the chat clients within that room.

The primary goal of this assignment is for you implement a distributed program, using two important Internet transport protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Specifically, you will learn how to use the Berkeley Socket API to write distributed applications on top of these protocols. In addition, you will learn about service location and fault tolerance in distributed systems.

2 System Overview

In the chat system, we use TCP for the control messages between chat client and server and we use UDP for the public chat messages between chat client and chat server. Communication between the chat client and the location server is via HTTP, which is layered on top of TCP.

The rationale behind this design is from the basic properties of the protocols. TCP is a connection-oriented reliable protocol and states related to each TCP connection need to be maintained at both client

and server side. To allow the chat server to scale (i.e., to accommodate many clients), we use TCP only for mission critical and rare actions, e.g., control messages. Since we view chat messages as non-critical and there will be many of them, we use a lighter weight protocol, UDP, for them. UDP is connection-less and thus there are no connection related states that need to be kept. However UDP is unreliable, which means users's packets can get lost inside the network. Packet loss is considered rare in the our LAN environment; if it does occur, we can fall back on the fact that chat is an interactive application. If a user notices that one of their messages was not delivered to themselves, they can retype it. If a user thinks a message might be missing, because the conversation doesn't make sense, they can type a message asking for clarification.

We now summarize the network-related components needed at the chat server and the chat client.

TCP/UDP chat server: For the chat server, we implemented a TCP server and a UDP server (both in the same single-threaded process). The TCP server is responsible for receiving TCP control messages from clients and sending replies to clients. The UDP server is responsible for receiving chat messages from chat clients and sending chat messages to the clients. If the server is built with `-DUSE_LOCN_SERVER`, it will announce that it is ready to accept chat clients by installing a simple text file at a web server. We expect that there will only be one publicly-announced chat server active at a time, and that some mechanism is in place to restart a failed chat server, possibly on a different host. Although the chat server maintains some internal state about each registered client, this state is lost on failure, and is not available to the new instance of the chat server when it starts accepting clients.

We use a web server for our location service because it simplifies the implementation by allowing us to take advantage of existing well-known (and hopefully reliable and well-maintained) services.

For the chat client, you need to implement the following:

TCP-based client: connects to the location server to obtain the chat server communication parameters; connects to chat TCP server, sends and receives control messages.

UDP-based client/server: sends and receives public chat messages. We refer to the part of the chat client that receives messages as a server – do not confuse this with the centralized chat server itself.

One TCP connection is established for each round of control message exchange between the chat client and the chat server. On the chat server side, after a control message is received and a reply is sent, the connection is immediately terminated. We do not want to maintain this connection for the entire time that the chat client is registered with the server because once a client has joined, it may only send chat messages and never send another control message until it is ready to quit. To keep the TCP connection around is a waste of resource and will not allow the chat server to scale.

A public chat message is sent from chat clients to the chat server via UDP, and the chat server will then relay the message to all the clients in the appropriate chat room (including the sender client) also via UDP.

You should not expect to detect server failures using messages sent via UDP, since UDP is an unreliable protocol and will not report any errors if a packet cannot be delivered to its destination. The TCP-based control interface, however, will report errors and can be used to detect that the server has failed.

3 Protocol Specification

3.1 HTTP-based chat server location protocol

If the chat client is built with `-DUSE_LOCN_SERVER`, it begins by requesting a text file from a web server using an HTTP GET request. HTTP is a simple, text-based, request-response protocol implemented at the

application level, using TCP/IP for connections. The first step is to open a TCP connection to the web server host. The second step is to send an HTTP request and receive the response from the web server.

An HTTP request has the following format:

```
[METH] [REQUEST-URL] HTTP/[VER]
[header_name_1]: [header_value_1]
[header_name_2]: [header_value_2]
. . .
[header_name_n]: [header_value_n]
<blank line>
[request body, if any]
```

It consists of (i) a request line, (ii) zero or more header lines, (iii) an empty line (always required!), and (iv) an optional message body. Each line (including the empty line) is terminated by `␣CR␣LF␣` (Carriage Return; Line Feed).

[METH] specifies the request method to be invoked at the server, of which there are several. For this assignment, we will only concern ourselves with GET requests, so there will be no request body. The blank line before the (non-existent) body is still required, however.

[REQUEST-URL] specifies the target object in terms of a relative URL. For example, if you had directed your browser to retrieve `http://www.cnn.com`, then, `http` specifies the protocol, `www.cnn.com` specifies the target host and the relative target URL is `/`. On the other hand, if you had directed your browser to retrieve `http://www.cnn.com/index.html` then the relative URL is `/index.html`.

Finally, **[VER]** is the HTTP version, either 1.0 or 1.1. For our purposes, the most important difference is that v1.1 requires a HOST header after the **[METH]** line, for GET requests. We use v1.0 to keep it simple.

To retrieve the chat server location file named `chatserver.txt` from the course pages on the CDF web server, the request can be as simple as:

```
GET /~csc469h/fall/chatserver.txt HTTP/1.0
```

You send the request by writing to the TCP socket you previously `connect()`'d to the web server.

You may find it helpful (or at least interesting) to see the headers that your browser sends along with a request for a Web page. The tools at <http://www.web-sniffer.net/> or at <http://www.ericgiguere.com/tools/http-header-viewer.html> will do so.

An HTTP response is sent from the server to the client after the server has received and processed the client's request and contains the result of the server's request processing. The response has the following format::

```
HTTP/[VER] [STATUS-CODE] [STATUS-TEXT]
[header_name_1]: [header_value_1]
[header_name_2]: [header_value_2]
. . .
[header_name_n]: [header_value_n]
<blank line>
[message body]
```

As before, **[VER]** is the HTTP version number. **[STATUS-CODE]** is a three digit integer code describing the outcome of the request to which the response corresponds. The first digit of the **[STATUS-CODE]** defines the class of the response:

- 1xx: Informational; i.e. the request has been received by the server and is being processed

- 2xx: Successful; i.e. the server has received, understood, and accepted the response and is sending back a response in this message
- 3xx: Redirection; i.e. the client is asked to resubmit the request to another server
- 4xx: Client error; i.e. the client has made an error in the request
- 5xx: Server error; i.e. the server was unable to process the request

Finally, [**STATUS-TEXT**] describes the STATUS-CODE with text. Common STATUS-CODE's and corresponding STATUS-TEXT's are:

```
200 OK
301 Moved permanently
302 Moved temporarily
400 Bad request (i.e. syntax error)
401 Unauthorized (i.e. client not authenticated)
403 Forbidden
404 Not found
500 Internal server error
501 Not implemented (i.e. request method not implemented)
503 Service unavailable (i.e. server may be overloaded)
```

Again, there are potentially many headers, but for our purposes we only care about the STATUS and the message body. The body should be the content of the requested file, `chatserver.txt`, which is a simple ASCII string with the following format:

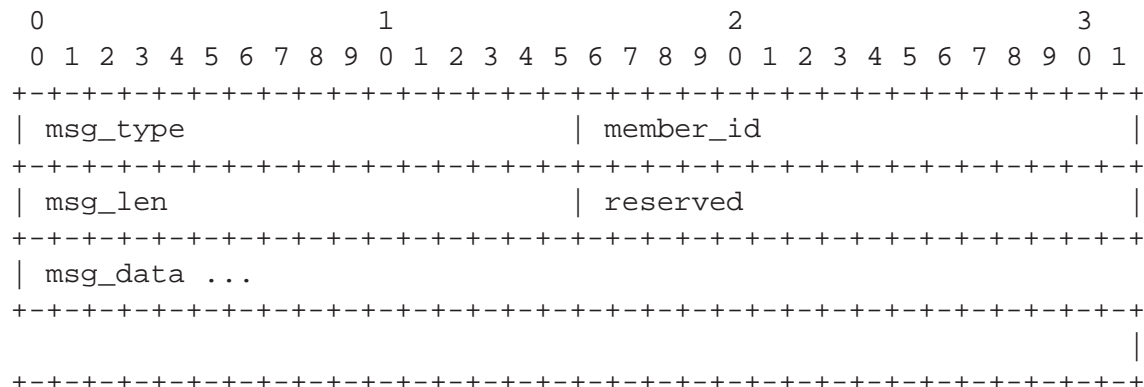
```
<chat server host name> <chat server tcp port> <chat server udp port>
```

Once the chat client knows the host name on which the chat server is running, and the port numbers that the chat server is listening on, it can initiate contact with the chat server.

We provide you with helper functions to construct the HTTP GET request, and to extract STATUS and body from the received request, along with a skeleton function to retrieve the chat server location information in `client_util.c`.

3.2 TCP-based control message protocol

The TCP-based protocol is used between the chat client and the chat server to exchange control messages. Each message exchanged has a common header (two 4-byte words) and may contain extra data in the data area. The data area in a packet is known as the payload in the networking lingo. A control message looks like this:



The header is defined by the following structure in `defs.h`:

```
struct control_msghdr {
    u_int16_t msg_type;
    u_int16_t member_id;
    u_int16_t msg_len;
    u_int16_t reserved;
    caddr_t msgdata[0];
};
```

msg_type represents the type of this message. We will explain each of the defined message types next. **member_id** is the id of the member that sends the message. An id is issued to each client after they have successfully registered. For each message (except `REGISTER_REQUEST`) sent from the client to the server, a valid id must be included. The **member_id** in a message from the server to a client is the **member_id** of the intended client.

msg_len denotes the length (in bytes) of the message including both the header and data.

reserved field is not currently used.

msgdata is a place holder pointing to the data area of the message. (**caddr_t** is defined as `char *`). The specific meaning of the data area depends on the message type.

The following message types are defined in `defs.h`:

1. REGISTER_REQUEST

This is the first message sent from client to server after the client starts. Since at this point, the chat client does not have a valid member id, this field will be ignored by the server. The data area of the message will contain information specific to this client: the UDP port that the client uses to receive public chat messages, the TCP port that client uses to receive private chat messages, and the member name that the client would like to use. This part is defined by another structure in `defs.h`:

```
struct register_msgdata {
    u_int16_t udp_port;
    caddr_t member_name[0];
};
```

The port number in this structure must be in network representation (Big Endian). There are several C library calls which you will find helpful for converting from the host representation of a number to the network representation (e.g. `htons(x)` converts from host to network for a **short** (2 byte) value, while `ntohs(x)` converts from network to host for a **short** value).

Figure 1 gives a simplified code snippet of how a REGISTER REQUEST message might be built, to illustrate the use of the fields in the control message header. When the server receives this message, it will decide whether to accept the client or not and reply with one of the following two messages:

2. REGISTER_SUCC

This message is sent from server to the client as a successful response to the REGISTER_REQUEST message. The **member_id** field will contain the id that the server generated for this client. Every client will receive a different id, which will be used for all the later control messages sent from client to server. There is no extra data in this message.

3. REGISTER_FAIL

This message is sent from the server to the client as a failed response to the REGISTER_REQUEST

```

...

/* allocate a block of memory to hold the message */
char *buf = (char *)malloc(1024);

/* common header pointer */
struct control_msghdr *cmh;

/*register data area pointer */
struct register_msgdata *rdata;

/* initialize the block to all 0 */
bzero(buf, 1024);

/* by casting, cmh points to the beginning of the memory block, and header
 * field values can then be directly assigned */
cmh = (struct control_msghdr *)buf;

/* message type */
cmh->msg_type = REGISTER_REQUEST;

/* rdata points to the data area */
rdata = (struct register_msgdata *)cmh->msgdata;

/* udp port: required */
rdata->udp_port = htons(7777);

/* member name */
strcpy((char *)rdata->member_name, "Smokey");

/* message length */
cmh->msg_len = sizeof(struct control_msghdr) +
              sizeof(struct register_msgdata) +
              strlen("Smokey");

/* send the message */
write(socket_fd, buf, cmh->msg_len);

...

```

Figure 1: Example of how to construct a REGISTER_REQUEST message.

message. The **member_id** field will not contain valid information. The **msgdata** area will contain an ASCII string that explains the reason for failure. For example, it could be that the member name a client wants to use has already been used by others. The client may choose to print this info out.

4. ROOM_LIST_REQUEST

This message is sent from client to server to request a list of current room names in the session. There should be no extra data in this message.

5. ROOM_LIST_SUCC

Succeeded request reply to ROOM_LIST_REQUEST from server to client. The data area contains an ASCII string list of the room names and number of members in each of the room. The list is in the following format:

[room1 (n1)] [room2 (n2)] ...

6. ROOM_LIST_FAIL

Failed request reply to ROOM_LIST_REQUEST from server to client. The data area contains an ASCII string with the reason for failure.

7. MEMBER_LIST_REQUEST

This message is sent from client to server to request a list of current members in a particular room. The **msgdata** must include the room's name.

8. MEMBER_LIST_SUCC

Succeeded request reply to MEMBER_LIST_REQUEST from server to client. The data area contains an ASCII string list of the member names in the requested room. It has the following format:

(member1) (member2) ...

9. MEMBER_LIST_FAIL

Failed request reply to MEMBER_LIST_REQUEST from server to client. The data area contains an ASCII string with the reason for failure.

10. SWITCH_ROOM_REQUEST

This message is sent from client to server to switch to a room. The **msgdata** must include the room's name.

11. SWITCH_ROOM_SUCC

Succeeded request reply to SWITCH_ROOM_REQUEST from server to client. There is no additional message data in this message.

12. SWITCH_ROOM_FAIL

Failed request reply to SWITCH_ROOM_REQUEST from server to client. The data area contains an ASCII string with the reason for failure.

13. CREATE_ROOM_REQUEST

This message is sent from client to server to create a room. The **msgdata** must include the room's name.

14. CREATE_ROOM_SUCC

Succeeded request reply to CREATE_ROOM_REQUEST from server to client. There is no additional message data in this message.

15. CREATE_ROOM_FAIL

Failed request reply to CREATE_ROOM_REQUEST from server to client. The data area contains an ASCII string with the reason for failure.

16. QUIT REQUEST

Client sends this message to server to leave the session. No additional data is needed. No reply is sent from the server.

3.3 UDP-based public chat message format

Public chat messages are sent from a client to the server. The server then relays messages to all the clients in the sender's room. The format for each message is defined as follows:

```
struct chat_msghdr {
    union {
        char member_name[MAX_MEMBER_NAME_LEN];
        u_int16_t member_id;
    } sender;
    u_int16_t msg_len;
    caddr_t msgdata[0];
};
```

When a client sends a chat message to the server, it must include its **member_id** in the **union** field, and the **msgdata** field would contain the actual message. **msg_len** is the total length (in bytes) of the message (header and the data). The chat message relayed by the server to the clients would have the union field initialized to the member name of the original sender, so that when a client receives this chat message, it will know who it is from.

3.4 User Interface

The focus of this assignment is on distributed programming and not on fancy user interface design. We provide you with a minimal implementation of an interface that does a reasonable job. You are, however, free to embellish it once you have things working. The usage of the chat client is as follows:

- To start, the user executes a single binary, “chatclient”, along with a set of command line arguments. However, within chatclient we start a second binary. As provided, the Makefile builds a chatclient executable that accepts the following command line arguments:

- h host name of the chat server
- t TCP port of the chat server
- u UDP port of the chat server
- n member name the client chooses to use

The options related to the chat server allow you to develop and test with your own chat server, rather than using the location server and contacting the publicly-announced chat server. When you are ready, modify the Makefile so that the client is built with **-DUSE_LOCN_SERVER**. When the client is built in this way, it will obtain the chat server information from the location server, and the only command line argument permitted is “-n member_name”.

- The chat client uses the following windows:

1. A user input window. This window displays a prompt `[membername] >` similar to a UNIX shell. We use the window that `chatclient` was started in for this. The user can either type in a control command or a chat message. A line that starts with “!” will be interpreted as a command, and all other input will be considered chat messages.
2. A public chat message display window. This window will display chat messages received from the chat server (including the messages sent by this user). For each message displayed, the sender’s member name also needs to be shown along with the message. We use a simple method of generating a second window from inside a C program. We fork and then exec an `xterm` using the `-e` option. For example, `xterm -e mybinary` launches a new `xterm` and starts executing `mybinary` inside it. For the chat client, we execute the “receiver” binary. The chat receiver must perform setup to allow it to receive messages from the chat server. This means it must create a UDP socket and bind it to a local port. For robustness, you should let the kernel choose the port number, and communicate the port chosen back to the main `chatclient` binary, which will send it to the chat server as part of the `REGISTER_REQUEST`. We use a simple local IPC message queue, for communication between the two chat client processes. The same message channel is also used to tell the receiver process to exit when the user enters the “quit” command.

- The control commands that a user can type are:

!r This will cause the chat client to retrieve the list of rooms in the session.

!c room_name This will cause the chat client try to create a room with **room_name** on the server.

!m room_name This will cause the client to retrieve the list of member names in **room_name**.

!s room_name This will cause the client to try to switch to **room_name**.

!q This will cause the client to quit from the session. A `QUIT_REQUEST` message will be sent to the server and no corresponding reply message is sent from the server to the client.

Users are given warnings on inputting non-conformant commands. White spaces (space, carriage return, tab, etc.) and non displayable characters are not allowed in **room_name**.

- Chat messages that a user types: Anything that does not start with a “!” is a chat message. A chat message can contain arbitrary ASCII characters, but is only one line. That is, in the control window, a user terminates a chat message with the “Enter” key.
- When the `chatclient` quits (the user types `!q`), all windows associated with `chatclient` must be destroyed and the resources allocated to them (e.g., TCP, UDP socket) must be released.

We provide code for basic user input handling. You must fill in the functions that deal with each type of control entry, or chat message entry.

3.5 Snapshot of a chat session

We present a snapshot of a chat session below to give you an idea of what a working system may look like. The chat server is running on machine `werewolf.cdf.toronto.edu` and waiting on TCP port 4000 and UDP port 4000.

```
redwolf% ./chatclient -h werewolf.cdf.toronto.edu -t 4000 -u 4000 -n Smokey
[Smokey]> !r
[football (0)] [basketball (0)] [soccer (0)] [hockey (0)]
```

```
[Smokey] !s football
switch to room: football
[Smokey]> !r
[football (1)] [basketball (0)] [soccer (0)] [hockey (0)]
```

A user starts chatclient on machine redwolf with nickname Smokey and finds there are four rooms in the session (apparently whoever created the rooms is a crazy sports fan) and she happily switches to her favorite room.

Later another user starts chatclient on machine seawolf with nickname JRB. JRB is also a football fan.

```
seawolf% ./chatclient -h werewolf -t 4000 -u 4000 -n JRB
[JRB]> !s football
switch to room: football
[JRB]> !r
[football (2)] [basketball (0)] [soccer (0)] [hockey (0)]
[JRB]> !m football
members: (Smokey) (JRB)
```

Then on machine seawolf, JRB enters a message in the control window:

```
[JRB]> did you see the Steelers last weekend?
```

On both JRB and Smokey's chat message displaying window:

```
JRB::
did you see the Steelers last weekend?
```

In response, Smokey types a chat message in her control window on redwolf:

```
[Smokey]> yeah, Parker put on quite a show in the 2nd half!
```

This message is shown in both JRB and Smokey's chat message displaying window:

```
JRB::
did you see the Steelers last weekend?
Smokey::
yeah, Parker put on quite a show in the 2nd half!
```

Smokey now quits the session (instead of hitting Ctrl-C), because she is heading out to Shoeless Joe's to watch this week's game:

```
[Smokey]> !q
```

JRB then sadly finds out that his friend is gone:

```
[JRB]> !m football
members: (JRB)
```

4 Implementation Stages

We recommend that you use the following guidelines in tackling this assignment. Based on the structure of the system, we divide the assignment into four stages. Before you move on to a next stage, you want to make sure that the code you write for the current stage is fully tested and free of bugs. However, in implementing later stages, you may wish add to or modify your implementation of an earlier stage.

Stage 1: Implement the basic control protocol. This includes the implementation of a TCP client to support all the control messages. You probably want to start by supporting the REGISTER_REQUEST message. The important thing is to have the first message type supported, the rest of the message types in the protocol can be done similarly. By the end of this stage, when chatclient starts, it joins the chat session and displays a prompt in the control window, which will then allow a user to type in various control commands to request room info, to create new rooms, to switch to a room and to quit. You may defer parts of the full protocol implementation until later stages. For instance, you may choose to put an arbitrary port number in your REGISTER_REQUEST message for the sake of implementing this stage and later on fix it with the right port number when you have implemented code for the chat receiver process to initialize a UDP socket and pass the port number back to the main chat process.

Stage 2: Implement the public chat messages sending, receiving and displaying. The things involved in this stage are: building a UDP client to send chat messages to the chat server (in `client.main.c`); building a UDP server to receive chat messages (in `client.recv.c`); having the receiver process tell the client process what UDP port number to use, and completing the previous REGISTER_REQUEST. Remember that the receiver process must check for messages from the chat server, and from the main chat client process, in case the main process wants to tell it to quit. Thus, you should not block when checking for either type of message. By the end of this stage, you should be able to exchange chat messages using your local version of the server. You can also look up the `chatserver.txt` file, and enter the chat server parameters as command line arguments when you start `chatclient`, to exchange messages through the public chat server.

Stage 3: Implement the use of the location server, rather than command line parameters, to obtain the chat server host name and port numbers. This step should be fairly simple, since you already know how to open a TCP connection to a server from Stage 1, and we provide functions to build and parse the text-based HTTP request/response. Remember that web servers typically listen on port 80.

Stage 4: Implement detection of, and recovery from, server failures. Your goal is to make this as transparent as possible to the user, however, you may want to let the user know that you have switched to a new server. At a minimum, you should notice that the old server is no longer working, use the location server to obtain information for the new server, and re-register with the new server. Some additional points you should consider are:

- How and when will you detect failure? It is unlikely you will see errors on your UDP sockets (you should check anyway), so a failed server will probably only be detected during a control exchange. An active user will notice that her own messages are not being delivered, and may initiate a control action (like listing the members in the room she is in). However, other users may only read the messages that others type, and not notice the difference between a quiet period in the conversation and a failed server that is not delivering any more messages. Can you use the existing control interface to silently check for server failure and initiate recovery without requiring a control input from the user? How often should you do so? Remember the chat server expects control messages to be rare relative to chat messages, but users would like to not miss large parts of a conversation.

- How can you help to deal with loss of server state when a new server takes over? For example, when you re-register, your client is not in any chat room, so you may want to automatically perform a `ROOM_SWITCH_REQUEST` to put your user back in the same chat room that they were in on the old server. But what if that room doesn't exist on the new server? Worse, what if your user's member name was registered on the new server by a different client *before* your chat client discovered the failure and re-registered?

5 Resources

General Information: There is a wealth of information available online on network programming. Several useful starting points are listed on the course assignments page. Keep in mind however, that these tutorials are often specific to a particular system that the author uses, and the exact details of the function calls used and the headers required may differ on CDF. For that reason, you should also refer frequently to the man pages for the C library functions and system calls you will be using. The man pages will provide detailed function specifications, information about error codes, and the header files you need to include. Finally, the books on network programming by renowned Unix expert W. Richard Stevens come highly recommended (e.g., *Unix Network Programming: Networking APIs: Sockets and XTI (Volume 1)*).

System Calls: We list below some system calls that you might expect to use for this assignment. You will not need all of these, and you might choose to use some that are not listed here, but this provides a starting point:

- *Basic socket related:* `socket`, `bind`, `connect`, `listen`, `accept`, `read`, `write`, `recvfrom`, `sendto`
- *Get protocol address associated with a socket:* `getsockname`, `getpeername`
- *Byte ordering:* `ntohs`, `ntohl`, `htons`, `htonl`
- *Block memory operations:* `bzero`, `bcopy`, `memset`
- *Name resolution related:* `gethostbyname`, `gethostbyaddr`
- *I/O multiplexing:* `select`, `FD_ZERO`, `FD_ISSET`, `FD_SET`

Many of these are used in the server code that we provide.

Starter Code: Get the file `A3.tgz` from `/u/csc469h/fall/pub/A3` on CDF. The README file contains a description of all the files included. All of your code should go in `client.h`, `client_main.c`, `client_recv.c`, and `client_util.c`.

6 Debugging

We distribute the chat server source code to you. The purpose of this is two-fold:

1. By examining the source code, you can get a feeling of how the server is designed and implemented, and hopefully this will help you designing your chat client and writing the code.
2. You can compile the source code to get a local version of the chatserver binary. You can debug your client code by having your client interact with your local copy of chatserver.

The chatserver is executed like this:

```
./chatserver -t <tcp port> -u <udp port> [-f <log file name> -s <sweep interval(mins)> -r <room file name>]
```

The parameters in square brackets are optional. The server TCP port and UDP port must be specified. Since on one machine, a given port number for each protocol can be bound to only one process, an error will be returned when multiple processes try to bind to the same port. In the server implementation, if the command line specified port has been used, we let the kernel dynamically choose a port for us from the pool of available ports. When -f is used, the server will log detailed information on each message exchanged between the server and the client. For each registered client, the server has to maintain states for this client, however, a client may die before sending an explicit QUIT_REQUEST message. In the server implementation, if -s is used, we periodically check the status of each client, if a client has been quiet (didn't send any messages to the server) for **sweep_interval** (minutes), the server will remove the client from the session. If -s is not used, the server will maintain a registered client's information throughout the lifetime of the server. As an aside, if -s is used, the server will also remove the rooms that do not have any members for **sweep_interval** (minutes). Option -r will specify a room config file which contains room names. This will allow the server to create rooms according to the file before any client joins. No rooms will be available at server start time if this option is not set.

The chatserver limits the maximum number of rooms and maximum number of members in the chat session by using pre-defined constants. To terminate chatserver, hit Ctrl-C. You should not run chatserver when you are not working on your assignment.

Testing failure detection and recovery: begin with a local version of the chat server using port numbers that you select. Run the chat client under GDB, so you can better control where the client is at the time the server fails (that is, when you kill it). Make sure your client notices that the server has failed, then restart the server with the same port numbers and step the client through the recovery. Once that is working, it should be straightforward to query the location server for the new chat server parameters, rather than assuming they are known ahead of time. To fully test with a chat client and server that you control, you would need to modify the way your chat server announces itself (since you do not have write permission on the 469 public.html directory), and the way your chat client locates the server. Make sure the client code you submit uses the current strategy to locate the server.

IPC message queue cleanup: the client code given out uses a unique temporary file name to create the IPC message queue key (this allows you to run multiple copies of the chat client from the same directory, without having them inadvertently all share the same control ; receiver message queue). If the client does not exit cleanly however, the temporary file is not deleted and the message queue is not destroyed. Files in /tmp are cleaned up nightly, but IPC message queues are not, and there is a system-wide limit on the number that are allowed. Please use the **ipcs** and **ipcrm** commands to find and delete your message queues as follows:

```
redwolf% ipcs -q

----- Message Queues -----
key          msqid      owner      perms      used-bytes  messages
0x2a0b0126  12681231  demke     600        0           0

redwolf% ipcrm -q 12681231
```

7 Revised Submission Instructions

Submit a softcopy of your code **and documentation in .txt or .pdf format** using the CDF submit command:

```
redwolf% tar -czf a3.tgz client.h client_main.c client_util.c \
client_recv.c a3_doc.pdf
redwolf% submit -c csc469h -a A3 a3.tgz
```

WARNING: The files you submit *must* compile with the original Makefile and **defs.h** file. To make it easy for you to test this, we are providing a script to check your submission:

```
redwolf% /u/csc469h/fall/pub/A3/check_submission.sh a3.tgz
Your submission compiled without errors.
redwolf%
```

Submit a hardcopy printout of (i) your **documentation** and (ii) your code, formatted with a C pretty-printer that numbers lines. One option is to use `enscript`, but you may use another tool if you prefer. To do this with `enscript`:

```
enscript --mark-wrapped-lines=box --header='demke|$n|page $% of $=' -C \
--highlight=c -p A3.ps client.h client_main.c client_recv.c client_util.c
```

This leaves the output of `enscript` in the file **A3.ps**, which you can then print out and hand in to the CSC469 drop box in BA 2220. (Replace 'demke' in the command above with the cdf login ids of your team).

8 Evaluation

The assignment will primarily be graded on: (i) overall design of the system; (ii) how closely your code implements the described protocol; (iii) how much of the described functionality you are able to support, and how correct your implementation is; and (iv) coding style. Design and style are important components in this assignment. Significant lack of documentation or elegance within your code will affect your grade.

Your grade will be based on the following:

- Correctness and completeness (65%). To get some credit, you must at the minimum be able to join the system, switch to a room and send and receive chat messages (25%). Handling all other control messages in a graceful manner earns you another 15%. Detection and recovery from server failure is worth the final 25%. **Your implementation must match the user interface specified in Section 3.4 exactly. If you change the interface, our testcases will fail, and your code will be considered incorrect. The TA will mark the assignment by running and testing the code. If your code does not compile, you will not receive any marks for correctness and completeness.**
- Design (20%). Elegance, robustness of design and handling of error conditions and special cases will be considered.
- Documentation (15%) You should develop your code in a good style. You should, for example, split up code into modules, make use of header files, use reasonable variable names and comment portions of your code that reflect design choices and non-intuitive ideas, properly indent your code and use empty lines to make your code more readable. **Your written documentation must describe your design, and what has and hasn't been done. Marks will be given for the clarity of this documentation – organize it neatly, use proper English, and spellcheck!**