Assignment 3: Distributed Chat System

Yu-Ching Chen g3yuch
Harris Fok g5harris

CSC469 - Operating Systems Design and Implementation

Stage 1: TCP Control Protocol

Implementation of the TCP control protocol was relatively straightforward, and more or less follows the structure detailed in the assignment specification. This stage of implementation covers the handle_XXX_req() functions and the send and receive control message functions. All of the request handlers follow the same basic structure: attempt to send the request to the server, receive the response, then handle the response in some way.

Attempts for server communication is contained in the function `attempt_request()`, which will be covered in more detail in Stage 4. Within this function, `connect_tcp()`, `send_ctrl_msg()`, and `receive_ctrl_msg()` are all called. The first establishes a new TCP connection (more details in Stage 4), and the latter two send and receive from the server.

send_ctrl_msg()

This function is responsible for creating and sending the control message. It takes the target file descriptor, the control message type, the message payload if applicable, and the appropriate message size as inputs. These values are obtained from the request handler functions, and passed down through `attempt_request()`. The function then simply creates the message header, writes to the socket.

receive_ctrl_msg()

This function used to be a lot more complex (reading just the header size to determine the full message length, then malloc'ing and reading the exact desired length), but since messages have limited length, it was easier and cleaner to just malloc and read the max message length. The buffer created will eventually be freed in the request handler functions before those return.

Regarding the sending of control messages, the data field of those messages are relatively straightforward, but server registration follows a more complicated system. Thus, in `handle_register_req()`, the register message data is filled out before the request is made to the server.

When handling the response, each function generally follows this format: if response is NULL, that means there was a connection error. Otherwise, if the response indicated a success, print the success text, or if it failed, print the failure text.

Stage 2: UDP Message System

During the initial process of the main chat program, the program calls a function to create a udp message system receiver. In this function, a message channel for communicating with another process is created using the IPC method messaging queue with the token 42 (random number to generate a key with ftok on shared file). Then the main chat client forks a new process, the child process then execs a client recv program in a xterm terminal.

client_recv.c

The newly created process, the client receiver then opens communication with the main chat client completing the IPC messaging queue with key generated from ftok also using the token 42 (agreement on the serialization). After the communication link is set up, the program creates a udp server starting at port 7000 and trying 200 times incrementally until a socket is binded. When successful, a message is sent back to the main client with the port number, but if unsuccessful, a message is sent back to the main client with an error signal. Then the program begins to receive messages from both the main client and the udp socket, both non-blocking.

Stage 3: Location Server

The work for this stage is essentially entirely contained within `retreive_chatserver_info()`, in client_util.c. This is called from client_main.c with the host name, TCP port, and UDP port variable addresses as parameters. The function in client_util.c then creates a TCP connection in a similar way to client_main.c, connects to the location server, and then makes the HTTP request. Writing and reading from the server is already done for us, so this made things fairly simple. Finally, if the response code is 200 OK, and only if it's that, we use the provided function to skip the headers and then parse the body with `strtok()`. All of this is performed again on each reconnect attempt, in case the location server gets updated.

Stage 4: Fault Recovery

There are two places in our client where errors can occur and be detected: whenever a general TCP request is made, and whenever our client tries to connect/register at a server. In the first case, our client will try to make the request again if a failure has been detected (covered by `attempt_request()`). In the second, the client attempts to fully reconnect to the server (covered by `reconnect()`). Additionally, the client will probe the server every 10 seconds using `heartbeat()` (which also doubles for `MEMBER_KEEP_ALIVE` functionality) to check if the server is still alive, and will `reconnect()` if it isn't.

attempt_request()

As mentioned above, `attempt_request()` works with `connect_tcp()`, `send_ctrl_msg()`, and `receive_ctrl_msg()` to make a request to the server. The main purpose of this function is to make multiple attempts to communicate with the server in case it fails the first time (or if the CONNECTION IS RESET BY PEER). If the connection fails, or we run out of attempts, then NULL is returned - and when the request handlers receive a NULL response, the user is notified of a server disconnect, and `reconnect()` is called.

connect_tcp()

It is important to note that `connect_tcp()` reestablishes a socket fd and makes a new connection with the server every time it's called. The server configuration is performed in `init_client()`, but that is the only function that isn't performed every time. For connecting, we used `select()`, and made the socket non-blocking (so that we can bypass `connect()` and use `select()`). This was done because the default timeout interval of `connect()` is far too long, and we were basically stuck whenever the location server pointed to tomlin.cdf.toronto.edu. `select()` lets us use our own timeout interval (set to 10 seconds), which made the connection process a lot more intuitive. Finally, whenever the connection fails, this function will return -1, which in turn causes `attempt_request()` to return NULL, and as mentioned previously, this is interpreted as a server disconnect.

reconnect()

This function works in a similar manner to `attempt_request()` - it uses a loop to perform multiple attempts at reconnecting. In our submission, we have `MAX_RETRIES` set to 5, but this can be set to a different value which will result in a more or less resilient client, or set to -1 to retry infinitely. At its core, this function calls `init_client()` to attempt to reconnect. Basically everything within `init_client()` needs to be done on every connect, except opening the receiver, which is only done once. If we reconnect successfully, the client will try to join the channel it was previously in (stored in `last_channel`), otherwise it will notify the user that they are no longer in a channel. If we don't connect successfully and run out of attempts, then we give up, and `shutdown_clean()` is called. Overall, when it works, this system smoothly recovers from any server errors, and the user experience is left mostly unperturbed.

heartbeat()

To allow checking for server status the user input must not block, to solve this problem a separate thread was created for user input. Creating a new thread also introduces synchronization problem with reading and writing to the user input buffer. Thus our design introduced a locking flag that only allows users to input when the buffer is empty and read only when the buffer is full and locking when it is satisfied. The input thread generates input when the buffer is not full, and checks every 0.01 seconds when it is full. While the input thread generates input, the main thread checks for buffer availability and handles it when the buffer is available,

else it checks every 0.01 seconds until it is available or until 10 seconds has passed, then it checks the server status to see if a reconnect is needed.

Name retry system

We've included a system for when the desired name is taken at the server. This can happen if two users pick the same name, but can also happen if a client reconnects to a server where someone else has picked the same name, or if a client doesn't unregister (uses ctrl+c) and tries to reconnect with the same name. The system appends "(*n*)" to the end the user's name, where *n* is a number that increments from 1 for each attempt. The way this system works is by adding an extra case in handle_register_req() that detects if we got a name-taken error. This extra case returns -3, and is then handled in init_client() where the handler was originally called. The client then loops through a number of conditional checks, then modifies the user's name, and attempts re-registering again and again until we finally get in, or get disconnected. Nevertheless, this system works surprisingly well, as it requires no user intervention, and can avert a lot of pain from having to manually change to a new name and connect again every time.

Unimplemented Features

Nothing we can think of!