Simple Chat

Program Overview

This program consists of a simple chat service that implements peer-to-peer communication organized by a chatroom server. In this case we are taking inspiration of IRC protocol specified in RFC 1459. The client sets up a communication terminal in which commands can be typed to manage and establish connection with other clients connected to the same server.

Usage

The server program is executed using the following command

```
./server <port-number>
```

The client program is executed using the follosing command

```
./client <IP-addr> <port-number> <user-ID>
```

Both executables can be called with the additional flag -@ which enables DEBUG mode. This mode prints to stdout information about the processes taken by each program.
The server runs by itself with no need for input. The client exists in three different modes where command-line input is sent to another host. The three modes are:

- INFO: the user can communicate with the server for managing its state and connections. The following commands are to be used in INFO mode:
  - /wait: sets a listening port awaiting a peer connection with another client, entering the WAIT state. Alternatively, the command /wait <port-number> can be used to set a specific port for waiting.
  - /list: displays clients connected to the server awaiting a connection.
  - /connect <user-ID>: starts a connection with another client if it is waiting. Sets client to the CHAT state.
- WAIT: state of a client awaiting a connection. No messages are sent to other hosts unless it returns to INFO state.
- CHAT: instead of sending messages to the client, the messages are sent to the peer selected by the /connect <user-ID> command.

The function /quit can be used in any state to quit the program succesfully. The Ctrl+C signal is also used for "dropping" a state, returning to the INFO state if in WAIT or CHAT or exiting the program if in the INFO state.

Logic and Design

i. State Management

As mentioned before, clients connected to the server can have any of three different states: INFO, WAIT, or CHAT. Upon connecting to the server, the client is given an INFO state. This means that a connection has been established but no peer-to-peer connection is expected. The WAIT state sets the user as open for

a peer-to-peer connection. All clients in this state are made visible upon requesting the /info command by another client. Lastly, the CHAT state determines that the user has been connected with another user, regardless of who requested the connection.

The Ctrl+C signal is used to return to the INFO state at any time, or quit the program if in the INFO state. Multiple cases are managed by this signal in order to change the state and free the used resources. The server in this implementation is "stateless", as it only keeps track of the state of each client but does not set different resources or measurements depending on the state. The only difference is the visibility to other clients and the ability to connect to it. It is the job of the client to switch from state to state by creating different threads (see part "iv. Client Logic" below) and alocating resources for each state.

ii. Connections

The server represents the orginizing entity in which clients post their usernames and await to be connected to another client. Awaiting clients are made visible as an available client. The list of available clients is requested upon sending the /info command. When a client requests to connect to an awaiting host, the server sends to both hosts the host address information of the other one. Clients in the WAIT state create a thread with a listening port number. Privately, the server keeps track of each client listening port number. When a connect request is made, the requester gets the IP address and the listening port number. The requester then connects to the listening socket using TCP. The requester knows the user ID of the peer, but the peer does not know the name of the requester, thus an initial exchange of IDs is made when the peer-to-peer connection is established. This allows for a true peer-to-peer communication with no "brigding" from the server. To clarify, the server-client connection remains live without caring of the state of the client. The client just sends messages to the peer instead of to the server.

iii. Server Logic

Upon connecting to the server, a unique user ID must be provided, which creates a key in a lexicographical user map. A c++ map allows for easy access and tree rebalancing. The keys (user IDs) have as values a struct containing three integers: the corresponding socket, the state of the client, and the listening port number (Set to 0 for "undefined"). This way of storing user data allows for rapid queries, as the function getpeername(2) provides any host information from an active socket.

As mentioned above, the server uses a struct data for storing client information. This allows us to utilize all of the std::map library to manipulate our connected client information. Additionally, we use a command_hash map which pairs strings of each available command and the corresponding function. Pointers to each function are returned when the key for an existing command is provided. Each server function is mapped to a command string sent by the client. The server implements the aformentioned

functions (i.e. /quit, /list, etc), and also some internal functions that help establish connections and state management. The additional functions are:

- fn_drop: sets the client state of a client to INFO
- fn_nick: sets the nickname of a user, sends to the client a ERR_NICKNAMEINUSE message (see "Message Formats and Errors" below).
- cmd_err: default function for an unkown command. Sends a ERR_UNKNOWNCOMMAND message to the client.

With this system it is very easy to scale this program without making major adjustments to the existing code, since it can manage many more clients and the implementation of new commands. Overall, the design of the server is quite simplistic, mainly taking advantage of powerful c++ libraries such as map. Much of the complexity of this project lies within the implementation of the client.

iv. Client Logic

The client also keeps track of the state, but with a more complex wait. The state defines your active socket, defined as chatfd. Depending on which state you are the active socket represents the socket through which packets will flow (except for the WAIT state; messages are cancelled when waiting). When a signal is called different states trigger different functions, closing all of the appropiate connections and freeing resources. Also, commands are detected at the client. When a command is typed, both the client and the server act separately and in syncronization. A similar command_hash map is used to map command strings and functions.

The program starts a separate thread for a command (except for /list which is done sequentially to avoid printing bugs). For the wait command the program starts a new thread which blocks with the accept() function. For the connect command the program queries the server for an IP address and connects to that address. Upon establishing a peer-to-peer connection, a new thread is created which prints out all incoming messages. This allows for the clients to chat without blocking until a message arrives. All blocking threads are stopped by a SIGINT signal or a /quit command by shuting down the socket, producing an error in the blocking function.

v. Utility Functions

For this application utility functions are used mainly to print error messages and to parse and prepare the IRC formatted packets sent between hosts. The next section explains the protocol and packet formatting.

Message Format and Errors

The protocol implementation is inspired by the RFC 1459 and its description of IRC. Packets are formatted using the following format:

: <prefix> : <suffix>

Where prefix and suffix specify the descriptions for prefixes and suffixes in RFC. The prefix of a packet sent from a client is only the user ID, while a server's preffix is the system name in addition to any message codes (either error or response codes) and the parameter in question (as declared in RFC 1459).

As a sidenote, the /list command responds similarly to the INFO response from an RFC server, by sending packets formated with the RPL_INFO code and a tailing packet with a RPL_ENDOFINFO code to terminate the INFO response.

Error messages from the server are formatted similarly. The implemented error messages are:

- 401 ERR_NOSUCHNICK: No such nickname
- 421 ERR_UNKNOWNCOMMAND: Command not recognised
- 433 ERR_NICKNAMEINUSE: Client ID is already taken

Testing

All sorts of testing were made to test the functionality and error recovery of the application. The following test cases were made succesfully:

- Clients can connect, wait, and request a list of waiting clients.
- The signal Ctrl+C drops a state whenever called, returning to the INFO state.
- The wait command can be used without an argument to open a random available port or be set to a specific port number.
- Clients can chat with other clients concurrently with no bridging, regardless of how many clients are connected, waiting, or chatting at the same time.
- Clients can try to use an existing nickname, connect to an unknown ID, or send commands in the WAIT or CHAT state but it will be unsuccesful, provoking an error message.
- Clients can hop back and forth from states by using commands and the Ctrl+C signal as many times as desired.
- When droping from the CHAT state, the peer also drops simultaneously, and the server modifies state entries, respectively. A communication with the server is resumed.
- The /quit command leaves the program when in any state.

In conclusion, the expected implementation was achieved. Unless something unexpected and unprecedented occurs, the program should work as desired.

Issues

- On ocations, when leaving a chat with another peer the client will receive two SIGINT signals: one for the user client and one for the peer client.
- Rarely, when dropping from the CHAT state, a segmentation fault will occur in the server. This issue has only happened twice before with an unknown cause. Several adjustments have been made to prevent such fault, but a fault of this kind might occur.
- When leaving a CHAT mode abruptly (not by /quit or Ctrl+C), sometimes the peer will take some time to be notified that the connection has ended. This is because the SO_KEEPALIVE option has a certain timer for keeping alive connections. Sometimes the SIGPIPE signal will only occur when the peer attempts to send another message. Pressing the Ctrl+C signal in the peer is advised to avoid waiting for the SIGPIPE signal to happenen.
- In one occation a client, after exiting a CHAT state, remained in a “limbo” state where it was considered to be in the INFO state but not able to send commands to the server.
- If the serveer crashes the clients won’t be notified immediately. The SO_KEEPALIVE signal will take some time to occur.

Acknowledgements

Code snippets and inspiration was taken from Professor Mike Parsa and his course CSE 156, Winter 2021 at UC Santa Cruz, as well as from Professor Wesley Mackey and his course CSE 111, Winter 2020 at UC Santa Cruz.