PUBLISH AND CHAT

JAFAR ABBAS (26346650) DECEMBER 9TH 2016

1. Introduction

This report summarizes the details of my implementation of a publish and chat system: an application working on top of UDP, using multiple persistent servers circularly linked to allow multiple clients to register, publish their status and friends and chat together. The programming language used is C++.

2. Assumptions

a) Circular Dependency of Servers

This assumption implies that the chat system will not function if one of the servers is not reachable, because servers must ensure a user is not registered elsewhere before registering.

The servers share a file called "serverconfig.txt" which contains the list of servers available. Example line: *name:server0,status:off,ip:132.205.95.34,port:8888*

Using this information, a server will load with the first available server info, and take the next server in the list as its "next server" to build the circular dependency. It will also change the status to "on".

b) Resending Messages

Messages that haven't been acknowledged are resent within 5 seconds by an independent thread.

Messages that were ignored for over 30 seconds are simply dropped.

c) Error Handling

Since the messages are sent by my own client, I've ensure the messages are sent in a proper format. The fields are assumed to be correct and there's minimal verification.

If a packet is corrupted through the network or doesn't make any sense then the server will simply drop it and send an error message and since it will not be acknowledged, the client will resend the message automatically.

d) Registration

The client's IP address and port is updated upon registration.

No actual authentication is implemented. If a user tries to register with a same name from a different machine, it will be granted access. Registering is taken as the equivalent of logging in.

If you attempt to register in a different server with the same name, you will be redirected to the next server until you reach the server on which you are registered and then granted access.

A client will try 5 times to register, whether he gets referred or not, if a registration is denied. Then will wait and try again later.

e) Publish

It is the user's responsibility for changing his published information while communicating with other users. The client doesn't forcibly stop this, despite the funny things that may happen.

f) Data persistence

Client data is saved into a "clientslist(*servername*).txt" upon update with the following format:

name:{Jafar}status:{off}addr:{132.205.95.34}port:{10000}friends:{Mohammad, Ali}

On launch this file is read to load the previously saved clients' data.

g) Curl.exe

This application verifies the IP address of the machine and prints it on the console upon launch.

h) Closing

Upon closing the client, a publish status off is sent to the server. If the client was chatting with friends, a bye message is sent to each one.

Upon closing the server, the serverconfig.txt is updated to set its status back to "off"

i) Reserved Characters

The following characters are restricted: ^ { }
This is because they are used as delimiters in the serialization process and no extra steps were taken to avoid this pattern elsewhere (i.e. in the name, chat message or friends field).

j) Client port

The client starts from port 10000 and attempts to bind that, if it's taken, it keeps incrementing and trying again.

3. Documentation

a) Program implementation

My implementation incorporates the following important aspects:

A Listener thread continuously listens for messages and launches a handler thread for each message it receives.

A Sender thread will directly send messages pushed into a "to_send" vector.

Resend and Clean up threads will periodically check unacknowledged messages or old messages to resend/delete.

Server-side handlers:

i) Registration and Find Request, these are handlers for requests that don't necessarily require a client to be registered.

ii) Client handler for all other requests allowed by the protocol made by registered clients.

Client-side handlers:

i) Handlers built-in to the functions or "states" of a client, i.e. registering, publishing, chatting. When a client is in a specific state, it will expect one of a few responses and acts accordingly.

ii) Message handler, for printing chat messages, which precedes the above handlers

Serialization: The my_MSG structure is cast to a char array where each field is separated by the following delimiter: ^^^. The message field of my_MSG structure is used for different purposes depending on the message type.

Protocol class: It offers an interface to all the protocol functions needed by the client & server.

Note: Appropriate mutexes are used to protect critical sections.

b) Structures used:

i) my_MSG

The *my_MSG* structure is serialized/deserialized and used for holding and sending messages. The message field is used differently depending on the message. This is the bread and butter of the protocol class.

```
struct my_MSG {
	std::string type = "";
	int         id = -1;
	int         port = -1;
	std::string addr = "";
	std::string name = "";
	std::string message = "";
	int         SERVER_MSG = 0;
	int         MORE_BIT = 0;
	int         OFFSET = 0;
};
```

ii) friend_data

The *friend_data* structure is used client-side to hold the current friends that the client has discovered and is able to chat with.

```
struct friend_data {
	std::string name = "";
	std::string addr = "";
	int port = -1;
};
```

iii) client_data

The *client_data* structure is used server-side to hold the current list of clients the server is serving. When the server loads, it reads the clientslist.txt file and fills up an array of this structure for each client that is registered.

```
struct client_data {
	std::string name = "";
	std::string status = "";
	std::string addr = "";
	int         port = -1;
	std::vector<std::string> friends;
};
```

iii) client_status

The *client_status* structure is used client-side to keep track of the client's own name, address, port, friends and the target server which may be updated upon referral.

```
struct client_status
{
	std::string MY_NAME = "";
	std::string SERVER_ADDRESS = "";
	int 		SERVER_PORT = -1;
	std::string MY_ADDRESS = "";
	int 		MY_PORT = -1;
	std::vector<friend_data> friends;
};
```

iv) server_status

The *server_status* structure is used server-side to keep track of the server's won name, address, port, list of clients registered and the next server it knows about.

```
struct server_status
{
	std::string MY_NAME = "";
	std::string MY_ADDRESS = "";
	std::string NEXT_ADDRESS = "";
	int 		MY_PORT = -1;
	int 		NEXT_PORT = -1;
	std::vector<client_data> clients_registered;
};
```

The above structures are used to hold the relevant data needed by the client/server.

The client doesn't save any data locally, but the server does have persistent data and instead of constantly reading from the file, the above structures are used to represent the information it needs.

The my_MSG structure is one of the most important structures and it is managed by the protocol class to be discussed in the next section.

b) Classes and Implementation

i) Protocol class

This class is the wrapper around the message structure (*my_MSG*) and defines the messages defined by the protocol. Since these messages are different server-side from client-side, the client and server each have their own implementation of this class.

Typical usage of this class involves giving a method some message and it returns the appropriate message to send. Most of the handlers don't care much about what is inside the message itself, it only looks at the type of message.

Common protocol:

The *getId* method is based on `std::chrono::system_clock::now().time_since_epoch()` cast into milliseconds and this works nicely because it can also be used to track how old each message is to know when to drop it or resend.

An array of *messages_pending_reply* is kept by the protocol, so if the protocol generates a message that requires a reply later on, it stores a copy there; a *cleanup* function exists to purge old messages and the *replied_to* method is used to fetch which message is this a reply of, if any. This array is protected by the mutex *mut_msgs*.

Server-side protocol:

```
class protocol {
public:
	protocol(server_status * info) {
		server_info = info;
		last_message = getId();
	}
public:
	my_MSG register_client(my_MSG msg);
	my_MSG deny_register(my_MSG msg);
	my_MSG is_registered_query(my_MSG msg);
	my_MSG is_registered_query_answer(my_MSG msg);
	my_MSG published(my_MSG msg);
	my_MSG unpublished(my_MSG msg);
	my_MSG inform_resp(my_MSG msg, client_data client);
	my_MSG find_resp(my_MSG msg, client_data client);
	my_MSG find_denied(my_MSG msg);
	my_MSG refer(my_MSG msg);
	my_MSG error(my_MSG msg, std::string message);
private:
	std::mutex mut_msgs;
	int last_message;
	server_status* server_info;
	std::vector<my_MSG> messages_pending_reply;

	void newmsg(my_MSG);
	void cleanup();
	my_MSG replied_to(my_MSG);
	int getId();
};
```

Client-side protocol:

```
#define MAX_MESSAGE_LENGTH 140
```

```
class protocol {
public:
	protocol(client_status * info) {
		client_info = info;
		last_message = getId();
	}
	my_MSG receive_fragmented_chat(my_MSG msg);
	std::vector<my_MSG> send_fragmented_chat(friend_data to_friend, std::string
message);
	my_MSG chat(friend_data to_friend, std::string message);
	my_MSG ack(my_MSG msg);
	my_MSG bye(friend_data bye_friend);
	my_MSG register_me();
	my_MSG register_me(my_MSG);
	my_MSG publish(bool status, bool update_friends, bool expect_reply = true);
	my_MSG inform_req();
	my_MSG find_req(std::string name);
	my_MSG find_req(std::string name, my_MSG);
	friend_data extract_friend_data(my_MSG, std::string name);
	client_status extract_my_info(my_MSG);
	bool replied(my_MSG);
	my_MSG error(my_MSG msg, std::string message);
	std::vector<my_MSG> timed_out_msgs();
	bool cleanup();
	void erase_all();

private:
	std::mutex mut_msgs;
	std::mutex mut_defrag;
	int last_message;
	client_status* client_info;
	std::vector<my_MSG> messages_pending_reply;
	std::vector<my_MSG> messages_to_defragment;

	my_MSG replied_to(my_MSG);
	void new_msg(my_MSG);
	void new_fragment(my_MSG msg);
	int getId();
};
```

The client-side supports fragmented chat messages, so it requires an extra array to hold these fragments until they can be combined.

ii) Common implementation

For the server/client, I did not find any necessity to implement it within a class. In the main file, all the needed functions and structures were declared and defined.

```
#define BUFLEN 1024

protocol * protocol_manager;

SOCKET s;
struct sockaddr_in server, si_send, si_recv;
int slen, recv_len;
char buf[BUFLEN];
char message[BUFLEN];
WSADATA wsa;

std::vector<my_MSG> messages_to_send;
std::vector<my_MSG> messages_received;
std::vector<my_MSG> temp;   // temp holder for messages_to_send
std::mutex    mut_send;
std::mutex    mut_recv;

void send(my_MSG);
void receive(my_MSG);
void sender();
void listener();
void getMyExternalIP();

BOOL WINAPI ConsoleCtrlEventHandler(DWORD dwCtrlType);
```

Each client/server has a *protocol_manager* that it uses to process messages, generate replies and create messages for whatever the entity wants from the protocol.

The *ConsoleCtrlEventHandler* handles closing events, such as sending bye messages to friends and setting status as off.

The *getMyExternalIp* method verifies the IP of the machine it's running on, using the curl.exe application.

The *buf* and *message* buffers are used for receiving and sending bytes from sockets.

The *sender*, *listener*, are each running in their own thread.

The *send* and *receive* methods simply push or pop the appropriate array of messages.

The *mut_send* and *mut_recv* protect the messages received and to send.

iii) Server implementation

```
server_status        my_status;
std::mutex           mut_clients;
std::mutex           mut_client_file;

// My functions
void initializeConnection();
void loadServersList();
void closeServer();
void loadClientsData();
void registerClient(my_MSG);
bool updateClientsData(my_MSG, bool update_ip_port_only = false);
void saveClientsData(client_data, bool);
void registrationHandler(my_MSG);
bool findRequestHandler(my_MSG);
void clientHandler(sockaddr_in, my_MSG);
void printClientsRegistered();
void getMyExternalIP();
void deserialize(char*, my_MSG*);
void serialize(char*, my_MSG*);
```

Handlers are launched as a separate thread by the listener thread. Handlers that don't require registration are called first, i.e. *registrationHandler* and *findRequestHandler* and if the message is from a registered client then *clientHandler* would be called for the other message types.

The *printClientsRegistered* conveniently prints the current list of clients periodically (i.e. it runs on its own thread). The rest of the methods are pretty straightforward and do as their names imply.

The *mut_clients* protects the list of clients embedded into the *server_status* structure.

The *mut_client_file* protects the clientslist.txt.

iv) Client implementation

```
#define START_PORT 10000

client_status client_info;
std::vector<friend_data> friends_available;
std::mutex    mut_friends;

bool registered, published, chat_mode, finished = false; // States

// My functions
void loadServersList();
void initializeConnection();
void resend_old_messages();
void erase_ignored_messages();
void deserialize(char*, my_MSG*);
void serialize(char*, my_MSG*);
void message_handler(my_MSG);
void closeClient();
void cleanup();
void send_chat_message(friend_data);
void receive_chat_message(my_MSG);

// UI
void myInterface();
void getRegistered();
void getPublished();
void getFriend();
void getMyInfo();
void getChatting();
void requestStatusAndFriends(bool*, bool*);
std::string requestFindFriend();
```

The *resend_old_messages* and *erase_ignored_messages* run on their own threads to periodically resend unacknowledged messages and to delete very old messages.

The *mut_friends* protects the list of friends in *friends_available*.

The *mut_client_file* protects the clientslist.txt.

4. Contribution & Conclusion

All code was done solely by me. This helped write the server and client in unison.

In summary, the protocol class acts as a manager and it is the only entity that creates, modifies, deletes messages (my_MSG). It is also the main class that even reads the contents of messages. The program mainly deals with the message type field.

Having a separate listener, sender and handler threads made my program very responsive. Using time cast into milliseconds for my unique ID was very convenient because it served as dual purpose to identify how old messages were.