CNline Report

> *2018 Final Project by group 11 b05902019蔡青邑 b05902043 劉鴻慶*

Introduction

This is a project for NTU CSIE Computer Internet course (2018), mainly about building a server-client system with serveral features, including chatting, sending files, and so on.

Instructions on how to run server & clients

> *This part invloving compiling and running, which is what users usually urge to search for, so we put it in the very beginnin of our README.*

To compile the file

You can simply run the `Build.sh` which is requested by our TAs.

Or, usually you should:

1. `cd` to the `src` folder.
2. `make`

To run the server

Simply,

1. `cd` to the `bin` folder
2. `./server [the path to config file]`

while the default configuration file is `server.cfg`, in the `config/` folder.

To run the client

Likwise,

1. `cd` to the `bin` folder
2. `./client`

User & Operator Guide

> *Since there are many many self-defined operations, here we're going to tell you all the how-tos and what-tos for using CNline as a user.*

Server Operation

Actually, you can just leave our server alone after start running it, while it will output all resent status and operation without asking any input.

Client Operation

After running the client,

1. You should see the *Entrance Hall*,

```
----------------------------------------
Entrance Hall
> How can I help you? login/signup/exit
>
```

2. Further, there is *Login Hall*

```
----------------------------------------
Login Hall
>> Please input your account name:
>> Please input your password:
```

3. After that there's *User Menu*:

```
----------------------------------------
User Menu
>>> What do you want to do?
add_friend/chat/logout/send_file/view_friend_list/block/unblock
>>>
```

Actually , other than these there main menus, there are a lot of other menus you may encounter. Nevertheless, just follow the instructions, you will be notified and have a second chance if you do anything wrong.

However, something we must tell you here is that, if you want to quit a chatting room, please type `exit()`, then the process will let you leave.

Protocol Specification

> *There are two designing purpose for protocol, one is to declare some common identifier for options for all users and programmer, the other is to specify a generally-used packet format with header and body so that both client side and server side can communicate smoothly and detect error or loss.*

Option Identifier

Basically, they are just some pre-defined value we use to avoid meaningless variable delclaring and establishing some common language. All spedifier is defined in c language at the file `include/common.h`.

```
//for identifying the message is a response or request
typedef enum {
  DATUM_PROTOCOL_MAGIC_REQ = 0x90,
  DATUM_PROTOCOL_MAGIC_RES = 0x91,
} datum_protocol_magic;

//for identifying the main purpose of operation for the message
typedef enum {
  DATUM_PROTOCOL_OP_LOGIN = 0x00,
  DATUM_PROTOCOL_OP_SEND_MESSAGE = 0x01,
  DATUM_PROTOCOL_OP_SEND_FILE = 0x02,
  DATUM_PROTOCOL_OP_REQ_LOG = 0x03,
  DATUM_PROTOCOL_OP_SIGN_UP = 0x04,
  DATUM_PROTOCOL_OP_ADD_FRIEND = 0x05,
  DATUM_PROTOCOL_OP_LOGOUT = 0x06,
  DATUM_PROTOCOL_OP_LISTEN = 0x07,
  DATUM_PROTOCOL_OP_REQ_LIST = 0x08,
  DATUM_PROTOCOL_OP_BLOCK = 0x09,
  DATUM_PROTOCOL_OP_UNBLOCK = 0x10,
} datum_protocol_op;//

//for identify the message is an ACK, NAK, or just a fragment
typedef enum {
  DATUM_PROTOCOL_STATUS_OK = 0x00,
  DATUM_PROTOCOL_STATUS_FAIL = 0x01,
  DATUM_PROTOCOL_STATUS_MORE = 0x02,
} datum_protocol_status;
```

Packet Format

Similarly, all format is written in c language at the file `include/common.h`, and they can be roughly classified into two types as follows:

- Header
    - `magic`: for `datum_protocol_magic` from the above identifier.
    - `op`: place for `datum_protocol_op` from the above.
    - `status`: for `datum_protocol_status`.
    - `client_id`: for specifying the client-side number.
    - `datalen`: to specify the size for the following body part (in the next type).

```
  //common header
  typedef union {
    struct {
      uint8_t magic;
      uint8_t op;
      uint8_t status;
      uint16_t client_id;
      //datalen = the length of complete message - the length of
  common header
      uint32_t datalen;
    } req;
    struct {
      uint8_t magic;
      uint8_t op;
      uint8_t status;
      uint16_t client_id;
      //datalen = the length of complete message - the length of
  common header
      uint32_t datalen;
    } res;
    uint8_t bytes[9];
  } datum_protocol_header;
```

- Complete Message (with header)

  In fact, in our protocol, there can be very many different varieties for a complete message. However, they are all of the same style—a mesage containing a header and a body—which is:

```
typedef union {
  struct {
    datum_protocol_header header;//header structure that we
already showed you
    struct {
      char some_body_content[100];
      char some_important_things[100];
    } body; //the body part, the main content(depend on the
message type)
  } message;//message = header + body
  uint8_t bytes[sizeof(datum_protocol_header) + 100 * 2];//for
memcpy
} datum_sample_message;
```

  and for example, here one specific message format for add-friend request:

```
typedef union {
  struct {
    datum_protocol_header header;
    struct {
      char name[USER_LEN_MAX];
    } body;
  } message;
  uint8_t bytes[sizeof(datum_protocol_header) + USER_LEN_MAX];
} datum_protocol_add_friend;//some more meaning full union name
```

  while the `USER_LEN_MAX` is 30 in our case.

System & Program Design

> *Other than protocols, here is how we design the system and how our programs run. Every function, structure, and behavior are followed by the below 3 principle:*
>
> - *To ensure no loss or error between communication, if there are, they should be detected and dealed.*
> - *Server side should be parted from client side, except those protocols.*
> - *The main function should be seldome(or never) modified or changed, compared to those others.*

Files and Folders

Before compiling the folders and files should be like:

```
.
├── README.md
├── bin
├── cdir
├── config
│   ├── client.cfg
│   └── server.cfg
├── include
│   ├── client.h
│   ├── common.h
│   └── server.h
├── sdir
└── src
    ├── Makefile
    ├── client.c
    ├── client_main.c
    ├── common.c
    ├── server.c
    └── server_main.c
```

- `README.md`: the thing you are reading now.
- `bin`: where all the executable files will be put at.
- `config`: configuration text files for determining things like port, IP, datbase location, etc.
- `include`: place we put all protocols and common functions.
- `cdir`/`sdir`: where client/server user data are saved.

- `src`: all the files to be compiled, including
  - `client.c`/`server.c`: all functions that are used by client or server respectively.
  - `client_main.c`/`server_main.c`: containing only the main functions itself, both no more than 10 lines.
  - `common.c`: functions that are commonly used by client and server, ie. `recv_message()`, `send_message`, or `complete_message_with_header`
  - `Makefile`: for `make`.

How we determine ACK/NAK and Why using it

- We determine the ACK/NAK by the flag `op` in our header structure, while `DATUM_PROTOCOL_STATUS_OK` stands for ACK and `DATUM_PROTOCOL_STATUS_FAIL` stands for NAK.
- The reason we use ACK/NAK even though TCP has already done it is that not every request sent by client is valid.

  For example, there can be a login request with wrong password, a add-friend request with a user name never signed up, or a send-file request with the receiver offline.

  Thus, telling the source client whether its request is valid can be necessary and time-saving, prevending some needless action and function calls.

  (Beside, we use timeout on ACK/NAK to determine whether to reconnect to server or checking the Internet connection).

Server-side Designing

In the `server_main.c`, there are only:

```
#include "server.h"
int main(int argc, char** argv)
{
  server_info* box = 0;
  server_init(&box, argc, argv);
  if (box)
    server_run(box);
  server_destroy(&box);
  return 0;
}
```

However, most of the functions are done in `server_run` and written in `server.c`.

Generally, our server is react-triggered, meaning it do something whenever there's some request received. And for different requests sent by serveral clients, it deal with them according to the type of that request.

The whole procedure is simply reading the header, and then read the message body depends on what the header said and send a ACK/NAK to the source client, before react to it.

Client-side Designing

Likewise, the main function is simple as `client_main.c` shows:

```
#include "client.h"
int main()
{
    struct socket_information *server;
    server = client_init();
    client_run(server);
    client_destroy(server);
    return 0;
}
```

and most of the functions are done in `client_run` and written in `client.c`.

While in `client_run`, the process is actually written as:

```
int client_run(socket_info *server)
{
    int reconnect_flag = 0;
    user_info *cur_user = NULL;
    while(1)
    {
        if(!reconnect_flag)
            cur_user = client_main_menu(server);//main_menu
        else if(reconnect_flag)
        {
            reconnect(cur_user, server);
            reconnect_flag = 0;
        }
        if(cur_user->login_status == USER_MAIN_OPT_EXIT)
            break;
        else if(cur_user->login_status ==
USER_MAIN_OPT_LOG_IN_SUCCESS)
        {
            reconnect_flag = client_user_menu(cur_user,
server);//user_menu
        }
    }
    return 0;
}
```

there are only two functions, `client_main_menu()` and `client_user_menu()`, that asking for user instruction and sending request to user, the rest is just looping over it and providing exit and reconnect options.

Some details for each request/ response implementation

- **log in/signup**

  Client asks for user name and password, then waits server for response after sending the request. If successfully receive ACK, then **login()** would lead user to **client_user_menu()**.

- **client_user_menu**

  In general, its a function for reading various user options and make each correspondent function calls. However, it also open a thread and connect a new socket for receiving other requests sent by server, like receiving messages and receiving file segments. The reason for doing so is thus we can send request and receive request at the same time. When open this thread and socket, client send a listen-request to server, so server can identify the use of this socket and ACKs it.

- **send file**

  To attain multiple files sent at the same time, we open a new thread and a new socket for each send-file request ACKed by the server, and then cut them into fragment for further sending process.

  During the sending process, server constantly receiving the file segment from this new socket and transfer them to another receiving-client's listening socket until the source client close the socket.

- **chat**

  After entering chat room, anything user types would instantly be sent as a send-message request.

  Beside, the moment user enters chat, client would request the corresponding chat log from the server. each time the thread opened by **client_user_menu** receives something from the current chatting room, it would directly print it on the screen.

  Simultaneously, server record any send-message requested in the corresponding user chat log, so that they won't be lost even if server was temporarily shut down.

Bonus

> *We've also done the following features, which is not stated in the requirement as some bonus part.*

- **auto-reconnect**

  Whenever the timeout reached for server's ACK response, client check the connection of the server by the `connect()` function, and if the connection is broken. It would tell the user, and keep trying reconnecting until server's back again or user close the process. If reconnecting sucessfully, user don't have to sign in again, which will be done by the reconnect function.

- **block/unblock**

  User can block and unblock other users to avoid uncomfortable add-friend or sending request.

- **logout**

  As not stated in the requirement, we allowing user to login/logout infinitely many times wthout reopen or close the client.

- **add friend**

  We build friend system so that user can add-friend and viewing them by a view-list request.

- **SHA256 hash encryptionfor pwd**

  Actually, every password sent by the client is encrypted and thus protected by SHA256, even the server can't know the actual password of any user.

- **view on/off-line status of friend list**

  While there is friend list, we also track on all the online status of your friend, so you know which one can response your message quikly.