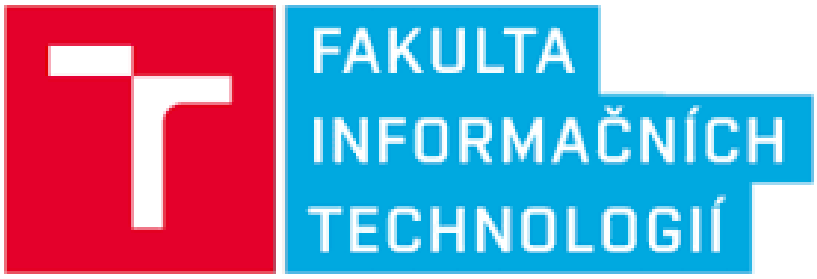
Vysoké učení technické v Brně

Fakulta informačních technologií



Počítačové komunikace a sítě

Dokumentace projektu

Klient pre chat server používajúci

IPK24-CHAT Protokol

Daniel Sehnoutek (xsehno02)

Brno, 31.3.2024

Obsah

Úvod 3
Návrh 4
Implementácia 5
Testovanie 9
Zdroje 12

Úvod

Vitajte v dokumentácii k projektu **Klient pre chat server používajúci IPK24-CHAT Protokol**, odvážny čitatelia. Tento projekt je zameraný na vytvorenie klienta umožňujúceho komunikáciu so vzdialeným serverom pomocou protokolu IPK24-CHAT. Projekt má dve varianty TCP a UDP.

TCP (Transmission Control Protocol) a UDP (User Datagram Protocol) sú dva základné protokoly pre prenos dát medzi klientom a serverom.

TCP (Transmission Control Protocol):

- na prenos dát medzi klientom a serverom treba na nadviazať spojenie
- komunikácia pomocou TCP je spoľahlivá
- TCP zabezpečuje doručenie dát v správnom poradí

UDP (User Datagram Protocol):

- na prenos dát medzi klientom a serverom nie je potreba nadviazať spojenie
- komunikácia pomocou UDP je menej spoľahlivá
- UDP sa používa v oblastiach, kde je potreba minimalizovať oneskorenie

Návrh

Ako prvú vec som si riadne prečítal požiadavky zadania a snažil sa pochopiť problematike daného projektu. Výstupom tejto snahy bolo vytvorenie logických súvislostí, ktoré vyplývali zo zadanie. Tieto logické súvislosti som neskôr spracoval do návrhu rozloženia súborov.

Finálne rozloženie súbor vyzeralo takto:

- **Hlavný priečinok**
  - **program.c** - spúštanie programu
  - **tcp.c** - logika tcp klienta
  - **tcp.h** - hlavičkový súbor pre tcp.c
  - **udp.c** - logika udp klienta
  - **udp.h** - hlavičkový súbor pre udp.c
  - **message_structure.h -** hlavičkový súbor s pomocnými štruktúrami
  - **Makefile a zvyšné súbory z Gitea repozitáru**
- **Testovací priečinok /tests**
  - **súbory s automatickými testami**
  - **textové súbory s častami kódu na manuálne testovanie**
- **Priečinok pre dokumentáciu /docs**
  - **pdf súbor s dokumentáciou projektu**

Implementácia

Najskôr som naimplementoval spustenie skriptu podľa počtu a obsahu vstupných parametrov. Následne vytvoril `štruktúry` a `enumy` pre uchovávanie požadovaných informácii o jednotlivých správach, ktoré som pri programovaní dopĺňal podľa potreby.

```c
// Structures for UDP message arguments
typedef struct {
    char username[USERNAME_MAX_LENGTH + 1];
    char displayName[DISPLAY_NAME_MAX_LENGTH + 1];
    char secret[SECRET_MAX_LENGHT + 1];
} StructAuth;

typedef struct {
    char channelID[CHANNEL_ID_MAX_LENGTH + 1];
    char displayName[DISPLAY_NAME_MAX_LENGTH + 1];
} StructJoin;

typedef struct {
    char displayName[DISPLAY_NAME_MAX_LENGTH + 1];
    char messageContent[MESSAGE_CONTENT_MAX_LENGTH + 1];
} StructErr, StructMsg;

typedef struct {
    uint8_t result;
    uint16_t ref_messageID;
    char messageContent[MESSAGE_CONTENT_MAX_LENGTH + 1];
} StructReply;

// Message structure for UDP
typedef struct {
    uint8_t type;
    uint16_t messageID;
    union {
        StructAuth auth;
        StructJoin join;
        StructErr err;
        StructMsg msg;
        StructReply reply;
    } data;
} Message;
```

```c
// Message types
typedef enum {
    MESSAGE_TYPE_AUTH,
    MESSAGE_TYPE_JOIN,
    MESSAGE_TYPE_ERR,
    MESSAGE_TYPE_BYE,
    MESSAGE_TYPE_MSG,
    MESSAGE_TYPE_REPLY,
    MESSAGE_TYPE_NOT_REPLY,
    MESSAGE_TYPE_NONE,
    MESSAGE_TYPE_WRONG
} MessageType;
```

Po vytvorení týchto štruktúr som sa snažil prísť na to ako funguje vytvorenie spojenia klienta so serverom vo verzii TCP. Keď sa mi konečne podarilo nadviazať spojenie, poslal som moju prvú správu na skúšobný netcat server, v `tcp_maine` deklaroval pomocné premenné a vytvoril while loop v ktorom som pomocou pollu spracovával správy od uživateľa a od serveru. Spracovávanie prebiehalo následovne:

```
// Message exchange loop
while(true){

    // Checks if u want to terminate program through CTRL+C
    signal(SIGINT, tcp_handler);

    // Wait for events on multiple file descriptors
    int ret = poll(fds, 2, -1); // -1 for indefinite timeout
    if (ret == -1) {
        perror("ERROR: poll");
        exit(EXIT_FAILURE);
    }

    // Incoming message from stdin(user)
    if (fds[0].revents & POLLIN) {

        // Load user message
        fgets(input_line, FULL_MESSAGE_BUFFER + 1, stdin);

        // Process user message and returns a message type
        msg_type = Handle_user_input_tcp(input_line, line_to_send_from_client, sockfd, current_state);
```

```
// Incoming message from server
if (fds[1].revents & POLLIN) {
    ssize_t bytes_received;

    // Read server message
    bytes_received = read(sockfd, received_message, sizeof(received_message));
    if (bytes_received < 0) {
        perror("ERROR: receiving message");
        exit(EXIT_FAILURE);
    }

    // Adjust the length of the received string, removing /r/n from the buffer
    if (bytes_received >= 2) {
        received_message[bytes_received - 2] = '\0'; // Move the terminating null character 2 bytes to the left
    } else {
        // Handle the case where there are less than 2 bytes received
        memset(received_message, '\0', sizeof(received_message));
    }

    // Process server message and returns a message type
    msg_type = Handle_server_messages_tcp(received_message);
```

Buď `štruktúra poll file descriptoru` zistila aktivitu od uživateľa alebo od serveru => prijala správu => správa sa spracovala príslušnou funkciu `Handle_user_input_tcp` pre uživateľa a `Handle_server_messages_tcp` pre server => v týchto funkciách sa so správou vykonali aj potrebné operácie, ako poslanie správy v korektnom formáte, zmena typu správy v premennej `msg_type` a ďalšie podľa typu príhodzej správy. Po dokončení daných funkcií na spracovanie správ následuje vyšetrenie zmeny stavu pre uživateľa aj pre server pomocou konečného automatu.

⇨ Krátka ukážka dvoch častí konečného automatu (1. pre uživateľa, 2. pre server)

```
// Finite state machine for processing user inputs
if (current_state == START_STATE){
    if (msg_type == MESSAGE_TYPE_AUTH) {
        current_state = AUTH_STATE;
    } else if (msg_type == MESSAGE_TYPE_MSG) {
        fprintf(stderr, "ERR: Cannot send message in this state, use /help command\n");
        current_state = START_STATE;
    } else if (msg_type == MESSAGE_TYPE_JOIN){
        fprintf(stderr, "ERR: Cannot join server by channelID in this state, use /help command\n");
        current_state = START_STATE;
    } else if (msg_type == MESSAGE_TYPE_NONE) {
```

```
// Finite state machine for processing server messages
if (current_state == AUTH_STATE){
    if (msg_type == MESSAGE_TYPE_NOT_REPLY) {
        current_state = AUTH_STATE;
    } else if (msg_type == MESSAGE_TYPE_REPLY) {
        current_state = OPEN_STATE;
    } else if (msg_type == MESSAGE_TYPE_ERR) {
        user_BYE(line_to_send_from_client, sockfd);
        current_state = END_STATE;
    } else {
```

Popri písaní konečného automatu som doplnil chýbajúce funkčnosti TCP programu o automatické posielanie BYE, ERR správ od uživateľa a korektné vypnutie programu cez signál zo stlačenia CTRL+C.

```
// Signal handeling function, correct termination
void tcp_handler (int signum){
    write(fds[1].fd, "BYE\r\n", 5);
    exit(0);
}

// Sends BYE message from user to a server
void user_BYE(char *line_to_send_from_client, int sockfd){
        size_t message_size;
        ssize_t bytes_sent;

        sprintf(line_to_send_from_client, "BYE\r\n");

        message_size  = strlen(line_to_send_from_client);

        bytes_sent = write(sockfd, line_to_send_from_client, message_size);
        if (bytes_sent < 0) {
            perror("ERROR: sending message");
            exit(EXIT_FAILURE);
        }
}

// Sends ERROR message from user to a server
void ERROR_from_user_to_server(char *line_to_send_from_client, int sockfd){
        size_t message_size;
        ssize_t bytes_sent;

        //ERR FROM {DisplayName} IS {MessageContent}\r\n
        sprintf(line_to_send_from_client, "ERR FROM %s IS Unexpected message in this state\r\n", display_name);

        message_size  = strlen(line_to_send_from_client);

        bytes_sent = write(sockfd, line_to_send_from_client, message_size);
        if (bytes_sent < 0) {
            perror("ERROR: sending message");
            exit(EXIT_FAILURE);
        }
}
```

Hlavičkový súbor `tcp.h` obsahuje prototypy funkcií z `tcp.c`, aby sa zaručila kompatibilita spustenia cez hlavný main v `program.c`.

```c
#ifndef TCP_H
#define TCP_H

#include "message_structure.h"

#include <stdlib.h>
#include <netinet/in.h>


void tcp_main(struct sockaddr_in servaddr, int server_port);

MessageType Handle_user_input_tcp(char *input_line, char *line_to_send_from_client, int sockfd, State current_state);

MessageType Handle_server_messages_tcp(char *received_message);

void tcp_handler (int signum);

void user_BYE(char *line_to_send_from_client, int sockfd);

#endif
```

V udp variante som postupoval podobne, len s využitím iných funkcií pre príjem a poslanie správ a s inými formátmi správ. Bohužiaľ sa mi nepodarilo správne zmeniť a doplniť kód riešeniami pre všetky výzvy, ktoré táto varianta prinášala, ako sú znovuposielanie potvrdzujúcich správ, zapamätávanie ID správ pre korektný chod programu a pár ďalších. Na druhú stranu hlavná štruktúra udp verzie je priložená v súboroch s kódom. Pri spustení je ale vynechaná.

Testovanie

Program som testoval a validoval viacerými spôsobmi:

1. Automatické testy (pomocou assertou, prípadne výpisou, kde to zjednodušilo kontrolu programu)

```c
void test_REPLY_OK(){
    MessageType msg_type;

    // ^REPLY OK IS (.{1,1400})$
    char test_reply_ok[FULL_MESSAGE_BUFFER + 1] = "REPLY OK IS SKUSKA 1 2 3 IDE";

    // Function Handle_server_messages_tcp prints out messages that should be printed out
    printf("Should print: Success: {MessageContent}\n");
    msg_type = Handle_server_messages_tcp(test_reply_ok);

    assert(msg_type == MESSAGE_TYPE_REPLY);
}

void test_REPLY_NOK(){
    MessageType msg_type;

    // ^REPLY NOK IS (.{1,1400})$
    char test_reply_nok[FULL_MESSAGE_BUFFER + 1] = "REPLY NOK IS SKUSKA 1 2 3 IDE";

    // Function Handle_server_messages_tcp prints out messages that should be printed out
    printf("Should print: Failure: {MessageContent}\n");
    msg_type = Handle_server_messages_tcp(test_reply_nok);

    assert(msg_type == MESSAGE_TYPE_NOT_REPLY);
}

void test_MSG(){
    MessageType msg_type;

    // ^MSG FROM ([!-~]{1,20}) IS (.{1,1400})$
    char test_message[FULL_MESSAGE_BUFFER + 1] = "MSG FROM DISPLAYNAME IS SKUSKA 1 2 3 IDE";

    // Function Handle_server_messages_tcp prints out messages that should be printed out
    printf("Should print: {DisplayName}: {MessageContent}\n");
    msg_type = Handle_server_messages_tcp(test_message);

    assert(msg_type == MESSAGE_TYPE_MSG);
}

void test_ERR(){
```

Spustenie testu:

```
Daniel@Ubuntu:~/Desktop/IPK_DEVELOPMENT/IPK-VUT-project1$ ./tcp_server_test
Should print: Success: {MessageContent}
Success: SKUSKA 1 2 3 IDE
Should print: Failure: {MessageContent}
Failure: SKUSKA 1 2 3 IDE
Should print: {DisplayName}: {MessageContent}
DISPLAYNAME: SKUSKA 1 2 3 IDE
Should print: ERR FROM {DisplayName}: {MessageContent}
ERR FROM DISPLAYNAME: SKUSKA 1 2 3 IDE
```

2. Posielanie správ pomocnému serveru netcat

```
Daniel@Ubuntu:~/Desktop/IPK_DEVELOPMENT/IPK-VUT-project1$ ./ipk24chat-client -t
tcp -s localhost
ahoj, toto je sprava
ERR: Cannot send message in this state, use /help command
/auth name secret dname
^CDaniel@Ubuntu:~/Desktop/IPK_DEVELOPMENT/IPK-VUT-project1$
```

```
Daniel@Ubuntu:~/Desktop/IPK_DEVELOPMENT/IPK-VUT-project1$ nc -l -p 4567
AUTH name AS dname USING secret
BYE
Daniel@Ubuntu:~/Desktop/IPK_DEVELOPMENT/IPK-VUT-project1$
```

3. Pomocou wiresharku (iba prvotné správy)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	49108 → 4567 [SYN] Seq=0 W
2	0.000039681	127.0.0.1	127.0.0.1	TCP	54	4567 → 49108 [RST, ACK] Se

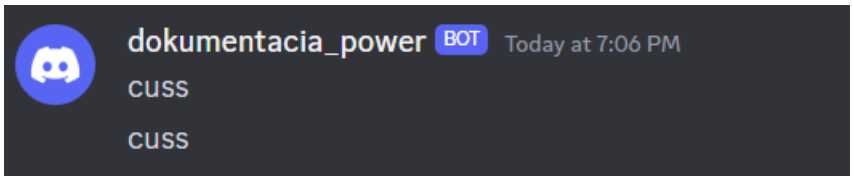
```
▸ Frame 1: 74 bytes on wire (592 bits), 74 bytes capt
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:0(
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: :
▸ Transmission Control Protocol, Src Port: 49108, Dst
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45
0010  00 3c a0 3c 40 00 40 06  9c 7d 7f 00 00 01 7f
0020  00 01 bf d4 11 d7 9e 67  9f 8e 00 00 00 00 a0
0030  ff d7 fe 30 00 00 02 04  ff d7 04 02 08 0a 45
0040  0f 7a 00 00 00 00 01 03  03 07
```

4. Manuálnymi testami s pomocnými výpismy, ktoré niektoré trochu opakované sú uložené v priečinku /tests

```
1   // BUFFERS FOR USER COMMANDS/MESSAGES TESTING
2   char test_auth[FULL_MESSAGE_BUFFER + 1] = "/auth username secret JOZEF_PRV
3   char test_join[FULL_MESSAGE_BUFFER + 1] = "/join channelID";
4   char test_rename[FULL_MESSAGE_BUFFER + 1] = "/rename ROBERT_DRUHY";
5   char test_help[FULL_MESSAGE_BUFFER + 1] = "/help";
6   char test_message[FULL_MESSAGE_BUFFER + 1] = "Hola, que pasa?";
7
8   // HOW THEY SHOULD BE SEND TO THE SERVER
9   // ERR FROM {DisplayName} IS {MessageContent}\r\n
10  // REPLY {"OK"|"NOK"} IS {MessageContent}\r\n
11  // AUTH {Username} AS {DisplayName} USING {Secret}\r\n
12  // JOIN {ChannelID} AS {DisplayName}\r\n
13  // MSG FROM {DisplayName} IS {MessageContent}\r\n
14
15  // FUNCTIONS TO BE YOU WHILE MANUALLY TESTING USER COMMANDS/MESSAGES
16  Handle_user_input_tcp(test_auth, line_to_send_from_client, sockfd);
17  printf("%s", display_name);
18  Handle_user_input_tcp(test_join, line_to_send_from_client, sockfd);
19  Handle_user_input_tcp(test_rename, line_to_send_from_client, sockfd);
20  Handle_user_input_tcp(test_help, line_to_send_from_client, sockfd);
21  Handle_user_input_tcp(test_message, line_to_send_from_client, sockfd);
22  user_BYE(line_to_send_from_client, sockfd);
23  printf("%s", display_name);
```

5. Finálnu verziu na “discord” server priloženom v zadaní



Zdroje

- IPK Prezentácie

- IPK Prednášky

- VUT discord

- poll.h documentation

https://pubs.opengroup.org/onlinepubs/009695399/basedefs/poll.h.html

- tcp client

https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/

- udp client

https://www.geeksforgeeks.org/udp-client-server-using-connect-c-implementation/