

電気電子情報実験・演習第一 I3 実験レポート

電子情報工学科
03240403 井上聡士

2024 年 6 月 28 日

1 目標

普通の通話アプリ (Zoom、Skype 等) には、以下のような機能が存在する

- 音声通話
- 多人数通話
- マイクのミュート
- カメラのオン・オフ
- 画面共有
- チャット

このうち、カメラ関係の機能は、WSL2 においてカメラのインターフェイスがどうなっているのかわからないことにより断念した。そのため、I3 課題では以下の機能を実装することを目標とした。

- 多人数通話
- ニックネームの設定
- マイクのミュート・アンミュート
- チャット

2 多人数通話

多人数通話をするにあたって、問題点が 4 点存在する。

- そもそもどのように通信を行うか
- 単位時間に送信するデータ量と受信するデータ量が大きく異なり、人数に依存する中どのようにデータの送受信を行うか
- 音声、チャット等様々なデータをどう区別するか
- 受信した音声をどのように再生すれば重ねることが出来るか

2.1 通信方式

そもそも基本的にサーバー・クライアントは1対1で通信を行うため、多人数通話を行うにはサーバー・クライアントの通信を複数行う必要がある。ただし、クライアント(=アプリのユーザー)が複数の他のクライアントに接続するのは、非常に難しく、そもそも通信に負荷がかかるうえ、さらにクライアント同士で接続する際にどちらがサーバーとなるかわからないという問題も存在する。そのため、サーバー側で複数のクライアントを処理し、それぞれのクライアントに複数のクライアント分のデータを送ることで、多人数通話を実現することにした(図1)。サーバーは基本的にデータの仲介のみを行い、それ以外の機能を担わないが、例外として新しいクライアントが接続してきたときに他のクライアントの現在の状況(ニックネーム、ミュート状態)を送る。クライアントで多くの処理を行い、音声、ミュート・アンミュート、チャットの情報を受け取った後にそれを処理する。

さて、既存のクライアントや新しい接続を試みるクライアントが存在するとき、どのように処理すべきか。通常のrecv関数は、基本的に相手から信号を受信するまで待機し続け、その間に他の処理を実行することが出来ない。そこで、selectという関数を使用した。select関数は、複数のfile descriptorに対して、読み込み、書き込み、例外のいずれかが発生するまで待機する関数であり、逆にいえばどれかが一つが変化すれば処理を行うことが出来る。基本的な使用法は、以下の通りである。

```
1 fd_set readfds;
2 int maxfd = 0;
3 while (1) {
4     FD_ZERO(&readfds); // readfdsを初期化
5     FD_SET(fd0, &readfds); // 全てのfile descriptorをreadfdsに追加
6     ...
7     FD_SET(fdN, &readfds);
8     maxfd = max(maxfd, fd0, fd1, ..., fdN); // 最大のfile descriptorを取得
9     select(maxfd + 1, &readfds, NULL, NULL, NULL); // いずれかのfile descriptorが変化するまで待機
10    if (FD_ISSET(fd0, &readfds)) {
11        // fd0に変化があった場合の処理
12    }
13    ...
14    if (FD_ISSET(fdN, &readfds)) {
15        // fdNに変化があった場合の処理
16    }
17 }
```

実際のサーバーのスクリプトでは、新規クライアントからの接続はサーバーのsocketの変化を、既存のクライアントからのデータは各クライアントのsocketの変化を検知することで処理した。

2.2 クライアント側のデータ処理

クライアント側は、通話に参加しているユーザーの数により受信するデータ量が増えるため、I2で行ったようにデータを受信・送信交互に行っても遅延が生じてしまう。そのため、ノンブロッキングモードで入力した音声データの送信と受信したサーバーデータの処理を行う必要がある。

この処理も、select関数を使用して行った。connectしたサーバーのsocketと、起動したsoxの

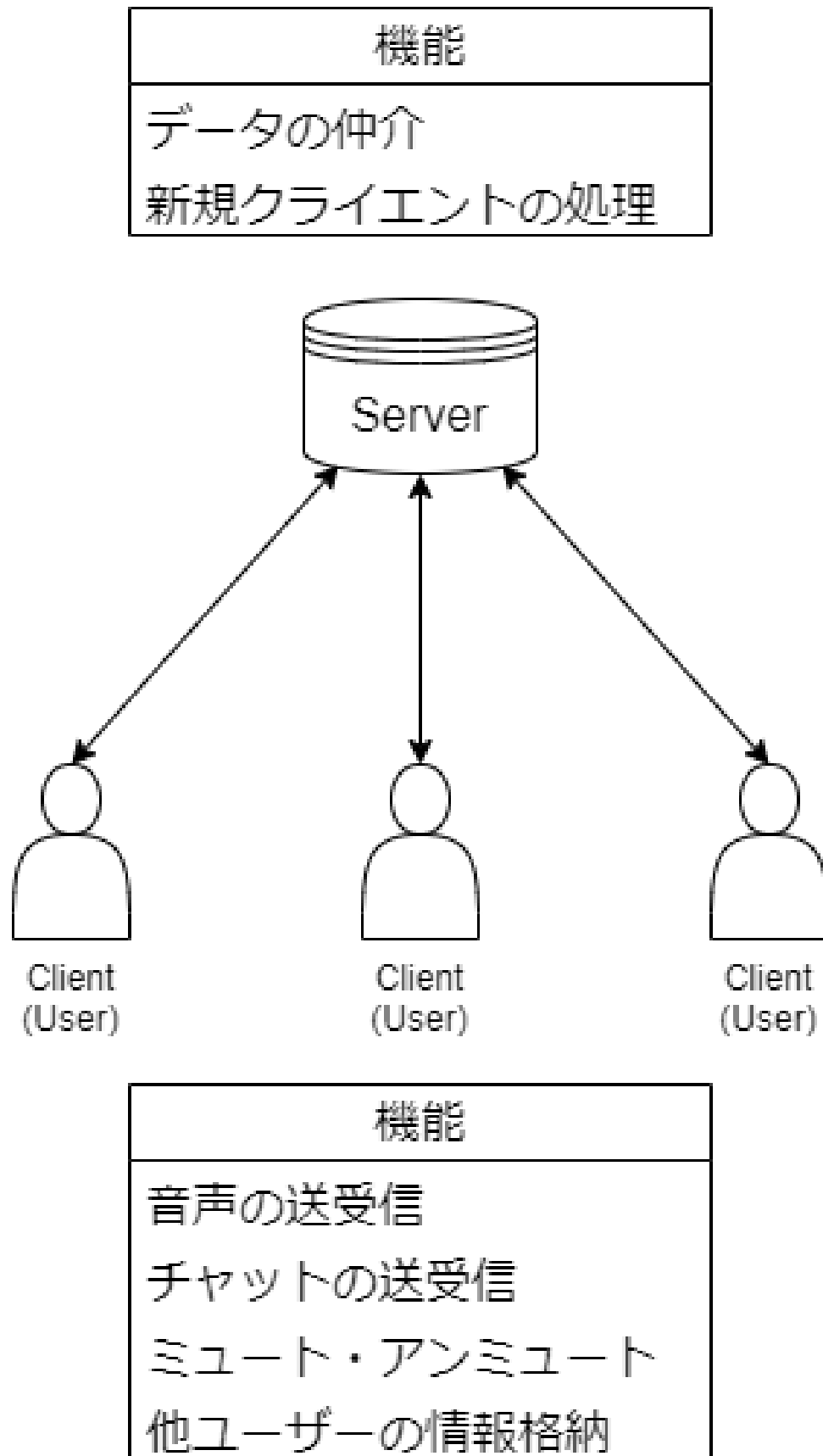


図 1: 通信方式の設計と機能

socket(および後述するが標準入力 stdin の socket である STDIN_FILENO) を select 関数に登録し、どちらかに変化があった場合に処理するようにした。

2.3 データの区別

サーバーからは、自分以外のクライアント分の音声データ、チャットデータ、及びクライアントの状態(ミュート・アンミュート、ニックネーム) 変化のデータが送られてくるが、これを区別する必要がある。そこで、サーバーへの送信データおよびサーバーからの受信データの先頭に、クライアント番号とデータの種別を示す識別番号を付与することで、クライアント側でデータの種別を判別することにした。サーバーへ送信するときは、識別番号をそれぞれ以下の通りに設定した

- 1: マイクからの音声データ
- 2: チャットのデータ
- 3: ミュート・アンミュートのデータ
- 4: ニックネーム・表示名のデータ

サーバーから受信するときは、識別番号をそれぞれ以下の通りに設定した。

- 0: サーバーからのメッセージ (主にサーバーへの接続で帰ってくるメッセージ)
- 1: 新規クライアントの接続・通話への参加
- 2: 他のクライアントの切断・通話からの離脱
- 3: 音声データ
- 4: チャットのデータ
- 5: ミュート・アンミュートの変更
- 6: ニックネーム・表示名の変更

そして、各 TCP パケットの先頭に、サーバーへの送信データは (識別番号)>>を、サーバーからの受信データは (クライアント番号).(識別番号)>>をつけて送信した。

しかし、ここで3つほど問題点が浮上した。

- 付けた修飾子...>>の文字数が不定で、どこまで読み取ればいいのか不明である
- 修飾子...>>が音声データやチャットデータにも含まれてしまうと、音声データが途中で切れたりしてしまい、挙動が不安定になる
- recv で特定の文字数を取得しようとしても、それより短い文字数しか取得できない場合がある

上の二つは、どちらも「recv する文字数が不明である」という共通の問題点に起因している。これを解決するために、まず、修飾子...>>の文字数を統一した。接続するクライアントは最大 30 に設定してあったため、クライアント番号を 2 桁表記 %02d にすることで解決した。次に、2 番目の問題だが、これは事前に読み取るデータの長さを指定することで回避できる。音声データは 1024 バイトずつ送信することに決めていたので、1 回のパケットで送る最大データ量を 1024 バイト + 修飾子として、その送信するデータ量を修飾子の中に記述した。例えば、140 バイトの文字列をサーバーへ送信する場合、2.0140>>と記述した後に 140 字の文字列を送信することで、サーバー側で 140 バイトの文字列として処理でき、仮に 2.0300>>などの文字列がデータ中に現れても問題なく文字として処理できる。

最後に、3 番目の問題は、検証した結果、TCP パケットをすべて受信できる前に `recv` を呼び出したことによるものと判明した。通常 TCP パケットはデータ量が大きいと何回かに分割して送るが、そのうち一つしか受信できてないうちに `recv` を呼ぶと意図していたよりも短いデータしか読み取れない。これを修正するために、パケットのすべてのデータを受け取るまで待つ `recv_all` という関数を定義した。

2.4 音声の入力・出力

音声の入力・出力には、`sox` を使用した。ただし、チャット機能を実装するが故、標準入力が必要であったので、`popen` を使用して独自の file descriptor から入力を取ることで、`sox` と標準入力を同時に行うことが出来るようにした。

出力は、これより難易度が高かった。`sox` のマニュアルを見ると、複数の入力を同時に再生することが可能で、`cat -v` をつけることでそれぞれの入力の音量を独自に調整できるようなので、これを試みた [1]。そこで、クライアント側で、他の各クライアントの音声入力を収納する一時ファイル `tmp01.tmp`、...`tmpN.tmp` を作成し、それを

```
1 $ sox -m -q -t s16 -r 44100 -c 1 tmp01.tmp -t s16 -r 33100 -c 1 tmp02.tmp ... -d
```

として、`popen` で実行するようにした。また、新しいクライアントが入ったときは元の `popen` を `pclose` で閉じて、新しいものを開くようにした。しかし、これはうまく動作しなかった。音声は重なったが、一定の時間再生された後再生されなくなった。これは、`sox` が、起動された時までの `.tmp` ファイルの分の時間しか再生しなかったためであると考えられる。

そこで、この解決策、すなわち `popen` を起動した後のファイルの更新を読み取る仕組みを考えた。このとき、2 通りの似たような方法にたどり着いた。

一つ目は、`tail -f` コマンドによる追跡である [2]。`tail -f` コマンドは、あるファイルを追跡し、随時更新されるたびに新たな出力を生み出し、かつキャンセルされるまで終了しない関数である。`sox` コマンドは、パイプのような形で複数のコマンド出力を取ることも可能であり、以下のように記述される。

```
1 $ sox -m -q -t s16 -r 44100 -c 1 "|tail -f tmp01.tmp" -t s16 ...
```

これを `popen` で実行することで、`sox` が `tmp01.tmp`、`tmp02.tmp`、... の更新を追跡し、新たな音声データが追加されるたびに再生することが出来る。しかし、新たなクライアントが登場し、`pclose` でこのコマンドを閉じようとしても閉じず、そこでフリーズするという問題点が発生した。これは、`pclose` の特性上、実行しているプログラムを `kill` のように強制終了するのではなく、終了するまで待つためである。しかし、`sox` によりパイプされた `tail -f` は、おそらくサブプロセスとして実行され、`sox` が終了しても `tail -f` は終了しないため、`pclose` が終了しないと考えられる。

二つ目は、FIFO(First in, first out) という種類のファイルを使用することである。FIFO は、ファイルシステム上に存在するファイルであり、書き込まれたデータは読み取られるまで保持される [4]。外見上は他の普通のファイルと変わりはないが、`ls -l` コマンドで `p` という文字が先頭についていることで FIFO であることがわかる。FIFO は、`mkfifo` コマンドで作成することが出来る。また、この FIFO は、ファイルの書き込みと読み込みを同時に開く必要があり、同時開かれるまで `open` の行で止まったままになってしまう。そのため、`open` を別スレッドで行うことで、FIFO を開こうとした。`fork` 関数を使用して、まず親プロセスで `sox` を起

動し、子プロセスで FIFO(読み取り)を開き、その後親プロセスで open(書き込み)をすることで、FIFO を同時に開くことが出来た。しかし、これもうまく動作しなかった。検証を重ねていくうちに、おそらく sox コマンドが複数の FIFO の入力に対応していないことが判明した。

そのため、最終的には、一つの sox コマンドですべてのクライアントの音声を再生することは不可能であると判断した。そこで、複数の sox コマンドを起動し、それぞれのクライアントの音声を再生することにした。

この方針は簡単で、popen により複数のコマンドを同時に起動し、それぞれの入力 file descriptor に書き込むことで実現できた。

3 チャット

チャットはいたって簡単である。クライアント側の select で、標準入力も監視し、変更があった際に識別子を付けてサーバーに送信する。受信する際は、読み取った識別子からチャットであることを判断し、それを標準出力に出力する。

4 コマンド

ミュート・アンミュート、および名前の変更は、標準入力で /mute、/unmute、/name <名前> と入力することで行うことが出来る。標準入力のデータを、strcmp で比較し、最初の文字が / であればコマンドと認識し、それ以降の文字列を処理する。

ミュートをしたときには、音声データを単に送信しないようにした。

5 コード

5.1 実行

コンパイルしたサーバー側のファイルを以下のように実行する。

```
1 $ ./multi_server <ポート番号>
```

別のターミナルで、コンパイルしたクライアント側のファイルを以下のように実行する。

```
1 $ ./multi_client <サーバーのIPアドレス> <ポート番号>
```

5.2 実演

レポートを書く際は4つのターミナルを用意し、サーバーを起動した後3つのクライアントを起動した。尚、別のPCで実行しても動作することは確認済みである。図2は、この時の様子である。サーバーを起動した後、左側のクライアントから順に起動した。新しいクライアント(左から2つめの Client 5)が参加すると、既存の Client 4 には Client 5>> Connected と表示され、接続されたことが表示される。また、参加した Client 5 も、すでにいる Client 4 の情報を受け取っている。

```

(base) inohang@INOHAW:~/projects$ cd experiment-3s/I3/
(base) inohang@INOHAW:~/projects/experiment-3s/I3$ ./multi_client 0.0.0.0 50000
>> Unmuted
Connection established as Client 6
Client 4>> Connected
Client 4>> unmuted
Client 5>> Connected
Client 5>> unmuted
Client 5>> muted
/mute
>> Muted
/name Satoshi
>> Name changed to Satoshi
Hi!
Client 5>> muted
Client 5>> unmuted
Client 4>> Hello!
Client 4>> Disconnected
Client 5>> Disconnected
(base) inohang@INOHAW:~/projects/experiment-3s/I3$

(base) inohang@INOHAW:~/projects/experiment-3s/I3$ ./multi_server 50000
>> Unmuted
Connection established as Client 4
Client 5>> Connected
Client 5>> unmuted
Client 6>> Connected
Client 6>> unmuted
Client 6>> muted
Name: Satoshi
Client 6>> Name changed to Satoshi
Satoshi>> Hi!
Client 5>> muted
Client 5>> unmuted
Hello!
(base) inohang@INOHAW:~/projects/experiment-3s/I3$

```

図 2: 実演した際の様子

最後に参加した Client 6 が /mute コマンドを使用すると、Client 6>> muted と表示され、Client 6 の音声を送信されなくなる。また、name Satoshi とした後は、Client 6>> の代わりに Satoshi>> と表示されている。Hi! と入力すると他のクライアントにも Satoshi>> Hi! と表示されているように、チャットも機能している。

また、クライアントが切断すると、Client 4>> Disconnected のように、切断も表示される。

レポートで紹介した機能は正常に動作しているだろう。

5.3 サーバー側のコード

```

1 #include <arpa/inet.h>
2 #include <errno.h>
3 #include <fcntl.h>
4 #include <netinet/in.h>
5 #include <netinet/ip.h>
6 #include <netinet/tcp.h>
7 #include <netinet/udp.h>
8 #include <signal.h>
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <sys/select.h>
13 #include <sys/socket.h>
14 #include <sys/types.h>
15 #include <unistd.h>
16 #define BUF_READ 1024
17 #define BUF_SEND 1060
18 #define MAX_CLIENTS 30
19
20 // Global
21 int client_sockets[MAX_CLIENTS];
22 int muted[MAX_CLIENTS];
23 char usernames[MAX_CLIENTS][21];
24 int ss;
25
26 int send_all(int *client_sockets, int
    client_exclude, char *msg, int len);
27 int rcv_all(int socket, char *buffer, int
    length);
28 int buf_w_bytes(char *sendbuf, int *len, int
    identifier, int sd, char *buf);
29 int buf_w_str(char *sendbuf, int *len, int
    identifier, int sd, char *str);
30
31 int handle_sigint(int sig);
32
33 int main(int argc, char **argv) {
34     int port;
35     int addrlen, max_sd, sd, len_buf,
        rcv_identifier;
36     struct sockaddr_in addr;
37     fd_set readfds;
38     char readbuf[BUF_READ];
39     char sendbuf[BUF_SEND];
40     char ipbuffer[30];
41     for (int i = 0; i < MAX_CLIENTS; i++) {
42         client_sockets[i] = 0;
43     }
44
45     if (argc < 2) {
46         perror("Usage: ./multi_server <port>");
47         exit(EXIT_FAILURE);
48     }
49     // Server
50     if ((port = atoi(argv[1])) == 0) {
51         perror("port");
52         exit(EXIT_FAILURE);
53     }
54     if ((ss = socket(PF_INET, SOCK_STREAM,
55         0)) == 0) {
56         perror("socket");
57         exit(EXIT_FAILURE);
58     }
59     addr.sin_family = AF_INET;
60     addr.sin_addr.s_addr = INADDR_ANY;
61     addr.sin_port = htons(port);

```

```

60     if (bind(ss, (struct sockaddr *)&addr,
61             sizeof(addr)) == -1) {
62         perror("bind");
63         exit(EXIT_FAILURE);
64     }
65     printf("Listening on port %d\n", port);
66     if (listen(ss, 3) < 0) {
67         perror("listen");
68         exit(EXIT_FAILURE);
69     }
70     addrlen = sizeof(addr);
71     printf("Waiting for connections...\n");
72
73     while (1) {
74         FD_ZERO(&readfds);
75         FD_SET(ss, &readfds);
76         max_sd = ss;
77         for (int i = 0; i < MAX_CLIENTS; i++)
78         {
79             sd = client_sockets[i];
80             if (sd > 0) {
81                 FD_SET(sd, &readfds);
82             }
83             if (sd > max_sd) {
84                 max_sd = sd;
85             }
86             int activity = select(max_sd + 1, &
87                                   readfds, NULL, NULL, NULL);
88             if ((activity < 0) && (errno != EINTR
89 )) {
90                 perror("select");
91             }
92
93             // Incoming connection
94             if (FD_ISSET(ss, &readfds)) {
95                 if ((sd = accept(ss, (struct
96 sockaddr *)&addr, (socklen_t *)&addrlen))
97 < 0) {
98                     perror("accept");
99                     exit(EXIT_FAILURE);
100                 }
101                 // Send to preexisting clients
102                 the information of newly connected client
103                 snprintf(readbuf, BUF_READ, "%s:%
104 d:Client %d", inet_ntoa(addr.sin_addr),
105 ntohs(addr.sin_port), sd);
106                 buf_w_str(sendbuf, &len_buf, 1,
107 sd, readbuf);
108                 puts(sendbuf);
109                 if (send_all(client_sockets, sd,
110 sendbuf, len_buf) < 0) {
111                     perror("send_all");
112                 }
113                 snprintf(readbuf, BUF_READ, "%d",
114 sd);
115
116                 buf_w_str(sendbuf, &len_buf, 0,
117 ss, readbuf);
118                 if (send(sd, sendbuf, len_buf, 0)
119 != len_buf) {
120                     perror("send");
121                 }
122                 int is_registered = 0;
123                 // Add new socket to array of
124 sockets
125                 for (int i = 0; i < MAX_CLIENTS;
126 i++) {
127                     if (client_sockets[i] == 0 &&
128 !is_registered) {
129                         client_sockets[i] = sd;
130                         muted[i] = 0;
131                         usernames[i][0] = '\0';
132                         is_registered = 1;
133                     } else if (client_sockets[i]
134 != 0) { // Send old clients to new
135 client
136                         getpeername(
137 client_sockets[i], (struct sockaddr *)&
138 addr, (socklen_t *)&addrlen);
139                         snprintf(readbuf,
140 BUF_READ, "%s:%d", inet_ntoa(addr.
141 sin_addr), ntohs(addr.sin_port));
142                         buf_w_str(sendbuf, &
143 len_buf, 1, client_sockets[i], readbuf);
144                         if (send(sd, sendbuf,
145 len_buf, 0) != len_buf) {
146                             perror("send_all");
147                         }
148                         // Mute status
149                         len_buf = sizeof(char);
150                         buf_w_bytes(sendbuf, &
151 len_buf, 5, client_sockets[i], (char *)&
152 muted[i]);
153                         if (send(sd, sendbuf,
154 len_buf, 0) != len_buf) {
155                             perror("send_all");
156                         }
157                         // Username
158                         if (strcmp(usernames[i],
159 "") != 0) { // If not blank
160                             buf_w_str(sendbuf, &
161 len_buf, 6, client_sockets[i], usernames[
162 i]);
163                             if (send(sd, sendbuf,
164 len_buf, 0) != len_buf) {
165                                 perror("send_all"
166 );
167                             }
168                         }
169                     }
170                 }
171             }
172         }
173     }

```



```

141         // Other sockets (client)
142         for (int i = 0; i < MAX_CLIENTS; i++)
143         {
144             sd = client_sockets[i];
145             if (FD_ISSET(sd, &readfds)) {
146                 readbuf[8] = '\0';
147                 if ((len_buf = recv_all(sd,
148 readbuf, 8)) == 0) {
149                     // Disconnected
150                     getpeername(sd, (struct
151 sockaddr *)&addr, (socklen_t *)&addrlen);
152                     buf_w_str(sendbuf, &
153 len_buf, 2, sd, "");
154                     puts(sendbuf);
155                     if (send_all(
156 client_sockets, sd, sendbuf, len_buf) <
157 0) {
158                         perror("send_all");
159                     }
160                     close(sd);
161                     client_sockets[i] = 0;
162                     muted[i] = 0;
163                     usernames[i][0] = '\0';
164                 } else {
165                     int actual_scanned;
166                     // Get identifier
167                     sscanf(readbuf, "%d.%d>>"
168 , &recv_identifier, &len_buf);
169                     if ((actual_scanned =
170 recv_all(sd, readbuf, len_buf)) !=
171 len_buf) {
172                         fprintf(stderr, "
173 Error: expected %d bytes, got %d bytes\n"
174 , len_buf, actual_scanned);
175                         continue;
176                     }
177                     switch (recv_identifier)
178                     {
179                         case 1: // Sound
180 data
181                             buf_w_bytes(
182 sendbuf, &len_buf, 3, sd, readbuf);
183                             if (send_all(
184 client_sockets, sd, sendbuf, len_buf) <
185 0) {
186                                 perror("
187 send_all");
188                             }
189                             break;
190                         case 2: // Message
191                             buf_w_str(sendbuf
192 , &len_buf, 4, sd, readbuf);
193                             if (send_all(
194 client_sockets, sd, sendbuf, len_buf) <
195 0) {
196                                 perror("
197 send_all");
198                             }
199                         }
200                     }
201                 }
202             }
203         }
204     }
205     close(ss);
206     return 0;
207 }
208
209 int send_all(int *client_sockets, int
210 client_exclude, char *buf, int len) {
211     int flag = 0;
212     for (int i = 0; i < MAX_CLIENTS; i++) {
213         if (client_sockets[i] != 0 &&
214 client_sockets[i] != client_exclude) {
215             if (send(client_sockets[i], buf,
216 len, 0) != len) {
217                 fprintf(stderr, "send_all: %d
218 \n", client_sockets[i]);
219                 flag = 1;
220             }
221         }
222     }
223     return flag;
224 }
225
226 }
227     break;
228     case 3: // Mute &
229 unmute
230         muted[i] = (int)
231 readbuf[0];
232         len_buf = sizeof(
233 char);
234         buf_w_bytes(
235 sendbuf, &len_buf, 5, sd, (char *)&muted[
236 i]);
237         if (send_all(
238 client_sockets, sd, sendbuf, len_buf) <
239 0) {
240             perror("
241 send_all");
242         }
243         break;
244     case 4: // Username
245 change
246         strcpy(usernames[
247 i], readbuf);
248         buf_w_str(sendbuf
249 , &len_buf, 6, sd, usernames[i]);
250         if (send_all(
251 client_sockets, sd, sendbuf, len_buf) <
252 0) {
253             perror("
254 send_all");
255         }
256         break;
257     default:
258         break;
259 }
260     }
261 }
262 }
263 }
264 }
265 }
266 }
267 }
268 }
269 }
270 }
271 }
272 }
273 }
274 }
275 }
276 }
277 }
278 }
279 }
280 }
281 }
282 }
283 }
284 }
285 }
286 }
287 }
288 }
289 }
290 }
291 }
292 }
293 }
294 }
295 }
296 }
297 }
298 }
299 }
300 }
301 }
302 }
303 }
304 }
305 }
306 }
307 }
308 }
309 }
310 }
311 }
312 }
313 }
314 }
315 }

```

```

216 }
217
218 int recv_all(int socket, char *buffer, int
length) {
219     int total_received = 0;
220     int bytes_left = length;
221     int n;
222
223     while (total_received < length) {
224         n = recv(socket, buffer +
total_received, bytes_left, 0);
225         if (n <= 0) {
226             // Handle error or disconnection
227             return n;
228         }
229         total_received += n;
230         bytes_left -= n;
231     }
232
233     return total_received;
234 }
235
236 int buf_w_bytes(char *sendbuf, int *len, int
identifier, int sd, char *buf) {
237     if (*len > BUF_READ) {
238         fprintf(stderr, "%d bytes exceeds
buffer size\n", *len);
239         return -1;
240     }
241     snprintf(sendbuf, BUF_SEND, "%d.%02d.%04
d>>%s", identifier, sd, *len);
242     bcopy(buf, sendbuf + 11, *len);
243     *len += 11;
244     return 0;
245 }
246
247 int buf_w_str(char *sendbuf, int *len, int
identifier, int sd, char *str) {
248     if (strlen(str) + 1 > BUF_READ) {
249         fprintf(stderr, "String %s exceeds
buffer size\n", str);
250         return -1;
251     }
252     snprintf(sendbuf, BUF_SEND, "%d.%02d.%04
ld>>%s", identifier, sd, strlen(str) + 1,
str);
253     *len = 11 + strlen(str) + 1;
254     return 0;
255 }
256
257 int handle_sigint(int sig) {
258     for (int i = 0; i < MAX_CLIENTS; i++) {
259         if (client_sockets[i] != 0) {
260             close(client_sockets[i]);
261         }
262     }
263     close(ss);
264     exit(0);
265 }

```

multi_server.c

5.4 クライアント側のコード

```

1  #include <arpa/inet.h>
2  #include <errno.h>
3  #include <fcntl.h>
4  #include <netinet/in.h>
5  #include <netinet/ip.h>
6  #include <netinet/tcp.h>
7  #include <netinet/udp.h>
8  #include <signal.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <sys/select.h>
13 #include <sys/socket.h>
14 #include <sys/stat.h>
15 #include <sys/types.h>
16 #include <sys/wait.h>
17 #include <unistd.h>
18 #define BUF_READ 1024
19 #define BUF_SEND 1060
20 #define MAX_CLIENTS 30
21 #define COMMAND_LENGTH 500
22
23 typedef struct clientsettings {
24     int id;
25     char ip[17];
26     unsigned int port;
27     char username[21];
28     FILE *fp;
29     unsigned int volume;
30     int is_muted;
31 } ClientSettings;
32
33 // Global variables
34 ClientSettings *clients[MAX_CLIENTS];
35 int s;
36
37 // Function prototypes
38 ClientSettings *client_setup(int sd, char *
str);
39 int client_free(int sd);
40 ClientSettings *client_get(int sd);
41 int recv_all(int socket, char *buffer, int

```

```

length);
42 int buf_w_bytes(char *sendbuf, int *len, int
    identifier, char *buf);
43 int buf_w_str(char *sendbuf, int *len, int
    identifier, char *str);
44 void handle_sigint(int sig);
45
46 int main(int argc, char **argv) {
47     signal(SIGINT, handle_sigint);
48     if (argc < 3) {
49         perror("Usage: ./multi_client <ip> <
port>");
50     }
51     fd_set readfds;
52     int port = atoi(argv[2]);
53     char *ip = argv[1];
54     FILE *fp_in;
55     int max_sd, fd_in, len_buf;
56     int scan_type, scan_client, scan_len;
57     int is_muted = 0;
58     char buf[BUF_READ];
59     char sendbuf[BUF_SEND];
60     char buf_identifier[12];
61     buf_identifier[11] = '\0';
62
63     if ((s = socket(PF_INET, SOCK_STREAM, 0))
== 0) {
64         perror("socket");
65         exit(EXIT_FAILURE);
66     }
67     struct sockaddr_in addr;
68     addr.sin_family = AF_INET;
69     if (inet_aton(ip, &addr.sin_addr) == 0) {
70         perror("inet_aton");
71         return 1;
72     }
73     addr.sin_port = htons(port);
74     int ret = connect(s, (struct sockaddr *)&
addr, sizeof(addr));
75     if (ret == -1) {
76         perror("connect");
77         return 1;
78     }
79
80     if ((fp_in = popen("rec -t s16 -c 1 -r
44100 -q --buffer 1024 -", "r")) == NULL)
    {
81         perror("popen");
82         return 1;
83     }
84     fd_in = fileno(fp_in);
85
86     // Send initial information (mute, user
name) first
87     len_buf = sizeof(char);
88     buf_w_bytes(sendbuf, &len_buf, 3, (char
*)&is_muted);
89     send(s, sendbuf, len_buf, 0);
90     fprintf(stderr, is_muted ? ">> Muted\n" :
">> Unmuted\n");
91     for (int i = 0; i < argc; i++) {
92         if (strcmp(argv[i], "-u") == 0) {
93             if (strlen(argv[i + 1]) > 20) {
94                 fprintf(stderr, "Username too
long\n");
95             } else {
96                 buf_w_str(sendbuf, &len_buf,
4, argv[i + 1]);
97                 send(s, sendbuf, len_buf, 0);
98                 fprintf(stderr, ">> Name
changed to %s\n", argv[i + 1]);
99             }
100             break;
101         }
102     }
103
104     while (1) {
105         FD_ZERO(&readfds);
106         // Incoming data from server
107         FD_SET(s, &readfds);
108         // Incoming data from stdin
109         FD_SET(STDIN_FILENO, &readfds);
110         // Incoming data from sox (mic)
111         FD_SET(fd_in, &readfds);
112         if (s > STDIN_FILENO && s > fd_in) {
113             max_sd = s;
114         } else if (fd_in > STDIN_FILENO &&
fd_in > s) {
115             max_sd = fd_in;
116         } else {
117             max_sd = STDIN_FILENO;
118         }
119         int activity = select(max_sd + 1, &
readfds, NULL, NULL, NULL);
120         if ((activity < 0) && (errno != EINTR
)) {
121             perror("select");
122         }
123         // Incoming data from server
124         if (FD_ISSET(s, &readfds)) {
125             int actual_scanned;
126             if (recv_all(s, buf_identifier,
11) == 0) {
127                 fprintf(stderr, "Server
disconnected\n");
128                 break;
129             }
130             sscanf(buf_identifier, "%d.%d.%d
>>", &scan_type, &scan_client, &scan_len
);
131             if ((actual_scanned = recv_all(s,
buf, scan_len)) != scan_len) {
132                 fprintf(stderr, "[%s]",
buf_identifier);

```

```

133         fprintf(stderr, "Error:
expected %d bytes, got %d bytes\n",
scan_len, actual_scanned);
134     }
135     switch (scan_type) {
136     case 0: // message from
server
137         fprintf(stderr, "
Connection established as Client %s\n",
buf);
138         break;
139     case 1: // new client
140         client_setup(scan_client,
buf);
141         fprintf(stderr, "%s>>
Connected\n", client_get(scan_client)->
username);
142         break;
143     case 2: // client
disconnected
144         fprintf(stderr, "%s>>
Disconnected\n", client_get(scan_client
)->username);
145         client_free(scan_client);
146         break;
147     case 3: // client data
148         if (client_get(
scan_client)->fp < 0) {
149             fprintf(stderr, "%s
not ready\n", client_get(scan_client)->
username);
150             break;
151         }
152         fwrite(buf, sizeof(char),
scan_len, client_get(scan_client)->fp);
153         break;
154     case 4: // Client chat
155         fprintf(stderr, "%s>> %s"
, client_get(scan_client)->username, buf);
156         break;
157     case 5: // Mute change
158         client_get(scan_client)->
is_muted = buf[0];
159         fprintf(stderr, "%s>> %s\
n", client_get(scan_client)->username,
client_get(scan_client)->is_muted ? "
muted" : "unmuted");
160         break;
161     case 6: // Name change
162         strcpy(client_get(
scan_client)->username, buf);
163         fprintf(stderr, "Name: %s
\n", client_get(scan_client)->username);
164         if (strcmp(client_get(
scan_client)->username, "") == 0) {
165             snprintf(client_get(
scan_client)->username, 21, "Client %d",
scan_client);
166         }
167         fprintf(stderr, "Client %
d>> Name changed to %s\n", client_get(
scan_client)->id, buf);
168         break;
169     default:
170         break;
171     }
172     }
173     // Input from STDIN
174     if (FD_ISSET(STDIN_FILENO, &readfds))
{
175         fgets(buf, BUF_READ, stdin);
176         if (strncmp(buf, "/", 1) == 0) {
177             // Commands
178             if (strncmp(buf, "/mute", 5)
== 0) {
179                 is_muted = 1;
180                 len_buf = sizeof(char);
181                 buf_w_bytes(sendbuf, &
len_buf, 3, (char *)&is_muted);
182                 send(s, sendbuf, len_buf,
0);
183                 fprintf(stderr, ">> Muted
\n");
184             } else if (strncmp(buf, "/"
unmute", 7) == 0) {
185                 is_muted = 0;
186                 len_buf = sizeof(char);
187                 buf_w_bytes(sendbuf, &
len_buf, 3, (char *)&is_muted);
188                 send(s, sendbuf, len_buf,
0);
189                 fprintf(stderr, ">>
Unmuted\n");
190             } else if (strncmp(buf, "/"
name", 5) == 0) {
191                 buf[strlen(buf) - 1] = '
\0';
192                 if (strlen(buf + 6) > 20)
{
193                     fprintf(stderr, "Name
too long\n");
194                 } else {
195                     buf_w_str(sendbuf, &
len_buf, 4, buf + 6);
196                     send(s, sendbuf,
len_buf, 0);
197                     fprintf(stderr, ">>
Name changed to %s\n", buf + 6);
198                 }
199             } else {
200                 fprintf(stderr, "Unknown
command\n");
201             }
202         } else {

```

```

202         buf_w_str(sendbuf, &len_buf,
203         2, buf);
204         send(s, sendbuf, len_buf, 0);
205     }
206 }
207 // Input from sox (mic)
208 if (FD_ISSET(fd_in, &readfds)) {
209     if ((len_buf = read(fd_in, buf,
210     BUF_READ)) > 0) {
211         if (!is_muted) {
212             buf_w_bytes(sendbuf, &
213             len_buf, 1, buf);
214             send(s, sendbuf, len_buf,
215             0);
216         }
217     }
218 }
219 handle_sigint(SIGINT);
220 ClientSettings *client_setup(int sd, char *
221 str) {
222     ClientSettings *client = malloc(sizeof(
223     ClientSettings));
224     for (int i = 0; i < MAX_CLIENTS; i++) {
225         if (clients[i] == NULL) {
226             clients[i] = client;
227             break;
228         }
229     }
230     client->id = sd;
231     char *token = strtok(str, ":");
232     strcpy(client->ip, token);
233     token = strtok(NULL, ":");
234     client->port = atoi(token);
235     snprintf(client->username, 21, "Client %d",
236     client->id);
237     client->volume = 100;
238     client->is_muted = 0;
239     if (client->fp != NULL) {
240         fprintf(stderr, "Audio output for %s
241         already exists\n", client->username);
242     } else {
243         if ((client->fp = popen("play -t s16
244         -c 1 -r 44100 -q --buffer 1024 -", "w"))
245         == NULL) {
246             fprintf(stderr, "Failed to open
247             audio output for %s\n", client->username);
248         }
249     }
250     return client;
251 }
252 int client_free(int sd) {
253     ClientSettings *client = NULL;
254     int i;
255     for (i = 0; i < MAX_CLIENTS; i++) {
256         if (clients[i] != NULL && clients[i]
257         ]->id == sd) {
258             client = clients[i];
259             break;
260         }
261     }
262     if (client == NULL) {
263         return -1;
264     }
265     if (client->fp != NULL) {
266         pclose(client->fp);
267         client->fp = NULL;
268     }
269     free(client);
270     clients[i] = NULL;
271 }
272 ClientSettings *client_get(int sd) {
273     for (int i = 0; i < MAX_CLIENTS; i++) {
274         if (clients[i] != NULL && clients[i]
275         ]->id == sd) {
276             return clients[i];
277         }
278     }
279     return NULL;
280 }
281 int recv_all(int socket, char *buffer, int
282 length) {
283     int total_received = 0;
284     int bytes_left = length;
285     int n;
286     while (total_received < length) {
287         n = recv(socket, buffer +
288         total_received, bytes_left, 0);
289         if (n <= 0) {
290             // Handle error or disconnection
291             return n;
292         }
293         total_received += n;
294         bytes_left -= n;
295     }
296     return total_received;
297 }
298 int buf_w_bytes(char *sendbuf, int *len, int
299 identifier, char *buf) {
300     if (*len > BUF_READ) {
301         fprintf(stderr, "%d bytes exceeds
302         buffer size\n", *len);
303         return -1;
304     }
305     snprintf(sendbuf, BUF_SEND, "%d.%04d>>",

```

```

    identifier, *len);
299     bcopy(buf, sendbuf + 8, *len);
300     *len += 8;
301     return 0;
302 }
303
304 int buf_w_str(char *sendbuf, int *len, int
    identifier, char *str) {
305     if (strlen(str) + 1 > BUF_READ) {
306         fprintf(stderr, "String %s exceeds
            buffer size\n", str);
307         return -1;
308     }
309     snprintf(sendbuf, BUF_SEND, "%d.%04ld>>%s",
        identifier, strlen(str) + 1, str);
310     *len = 8 + strlen(str) + 1;
311     return 0;
312 }
313
314 void handle_sigint(int sig) {
315     for (int i = 0; i < MAX_CLIENTS; i++) {
316         client_free(i);
317     }
318     close(s);
319     exit(0);
320 }
```

multi_client.c

参考文献

- [1] Bagwell, C. (n.d.). sox(1) - Linux man page. die.net. Retrieved June 29, 2024, from <https://linux.die.net/man/1/sox>
- [2] GeeksforGeeks. (2023, December 13). Tail command in Linux with examples. GeeksforGeeks. <https://www.geeksforgeeks.org/tail-command-linux-examples/>
- [3] pclose(3) - Linux man page. (n.d.). die.net. Retrieved June 30, 2024, from <https://linux.die.net/man/3/pclose>
- [4] GeeksforGeeks. (2023, September 9). Named Pipe or FIFO with example C program. GeeksforGeeks. <https://www.geeksforgeeks.org/named-pipe-fifo-example-c-program/>