

情報科学演習 C

課題 4

基礎工学部情報科学科計算機コース

学籍番号: 09B22002

安西俊輔

2024 年 8 月 7 日

1 課題 4-1

1.1 実装方針とアルゴリズム

今回作成したのは、複数人がリアルタイムでやりとりできるサーバ・クライアント型のチャットシステムである。仕様としては、各参加者の発言は、チャットサーバを経由してすべての参加者の端末に発言者の名前つきでコピーされ、参加者は途中からでも自由にチャットに参加あるいは離脱することができる。

実装方針としては、どちらのプログラムも変数 state で状態の遷移を管理し、実施要項の 4.3 節に記述されているプロトコル仕様の状態遷移で処理が決定されるようにした。各状態の処理についてそこまでボリュームがなかったため、全て main 関数で状態ごとに分けて記述した。また、クライアントプログラムは状態の数字が昇順で処理が進んでいき、状態変数が 5 または 6 に達したとき終了するが、サーバプログラムは状態を行き交い動作し続けるという仕様であるため、ループを用いた。

1.1.1 クライアントプログラム

まずクライアント側のプログラムのフローチャートを以下に示す。数字は状態番号を示している。

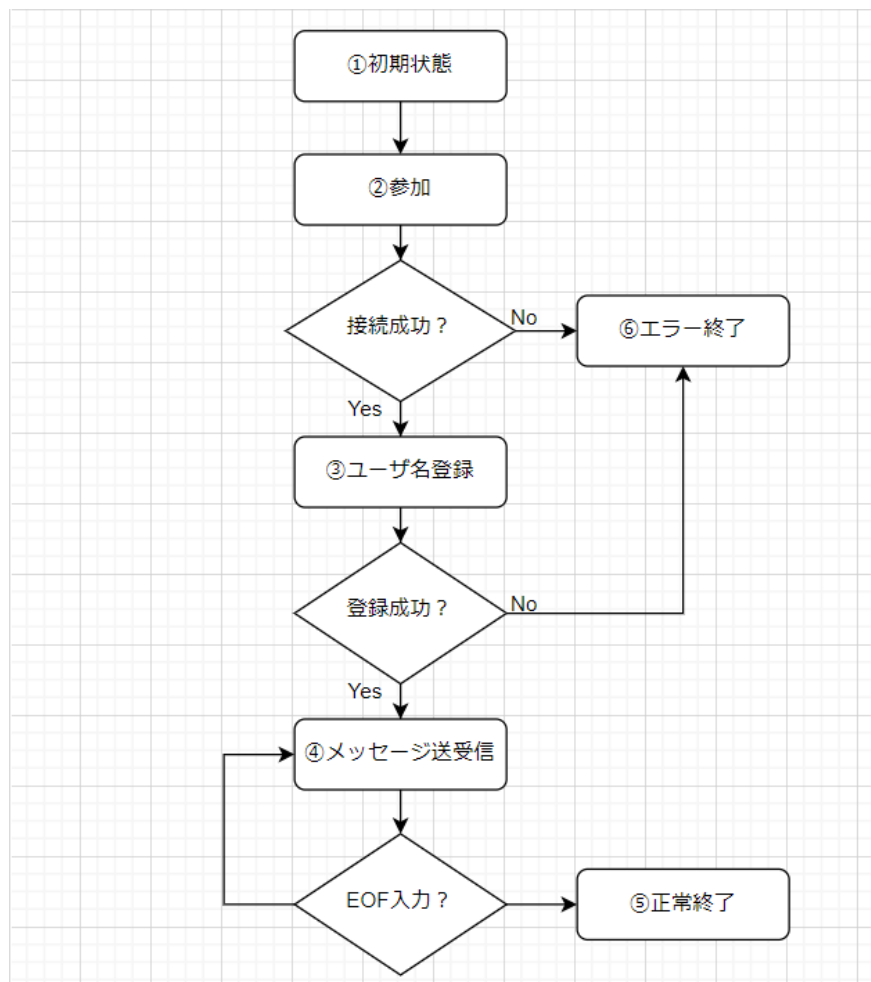


図1 クライアントプログラムのフローチャート

エラーが起きなければ、状態番号は昇順に進む。エラーが起きると状態 6 へ飛び異常終了する。各状態について説明する。各状態の具体的な処理について説明していく。

まず宣言した変数および配列について型と役割をまとめた表を以下に示す。全て main 関数内で宣言した。

変数・配列名	型	役割
state	int	プログラムの状態を管理する変数
sock	int	ソケットファイルディスクリプタ
svr	struct sockaddr_in	サーバーのアドレス情報を保持する構造体
hp	struct hostent*	ホストエントリ情報を指すポインタ
nbytes	int	読み書きされたバイト数を保持する変数
rfd	fd_set	select() で用いるファイル記述子集合
tv	struct timeval	select() の待ち時間を指定する構造体
rbuf	char[1024]	ソケットからの受信データを格納するバッファ
sbuf	char[1024]	ソケットへの送信データを格納するバッファ
name	char[99]	ユーザー名を格納するバッファ

・状態1（ソケットの生成とサーバーへの接続）

ここでは初期設定を行う。以下に抜粋したコードを示す。

```
1     if(state == 1){
2         if(argc != 3){
3             printf("Usage: ./chatclient hostname username\n");
4             exit(1);
5         }
6
7         /* ソケットの生成*/
8         if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
9             perror("socket");
10            exit(1);
11        }
12
13        /* server 受付用ソケットの情報設定*/
14        if ((hp = gethostbyname(argv[1])) == NULL) {
15            fprintf(stderr, "unknown host %s\n", argv[1]);
16            exit(1);
17        }
18        bzero(&svr, sizeof(svr));
19        svr.sin_family = AF_INET;
20        bcopy(hp->h_addr, &svr.sin_addr, hp->h_length);
21        svr.sin_port = htons(10140); /* ポート番号10140 */
22
23        /* サーバーに接続*/
24        if (connect(sock, (struct sockaddr *)&svr, sizeof(svr)) < 0) {
25            perror("connect");
26            exit(1);
27        }
28    }
```

```

29         if(write(1,"connected to server\n",20)<0){
30             perror("write");
31             exit(1);
32         }
33
34         state = 2;
35
36     }

```

2-5 行目：コマンドライン引数が仕様と異なる場合、入力方法を示す。

8-11 行目：ソケットの生成を行う。INET ドメインでストリーム型で指定する。

14-21 行目：サーバのホスト名を取得し、アドレス情報を設定する。ポート番号は 10140 とする。

24-27 行目：サーバに接続する。

29-32 行目：接続が成功したことを示すため write() で標準出力に”connected to server”と表示する。
最後に state を 2 にする。

・状態 2（サーバからの接続承認）

ここではサーバへの接続が受け入れられたかどうかを確認する。以下に抜粋したコードを示す。

```

1     if(state == 2){
2         bzero(rbuf, sizeof(rbuf));
3         if ((nbytes = read(sock, rbuf, 17)) < 0) {
4             perror("read");
5             exit(1);
6         }
7
8         if(strcmp(rbuf, "REQUEST ACCEPTED\n") == 0){
9             if(write(1,"join request accepted\n",22)<0){
10                 perror("write");
11                 exit(1);
12             }
13             state = 3;
14         }else{
15             if(write(1,"join request rejected\n",22)<0){
16                 perror("write");
17                 exit(1);
18             }
19             state = 6;
20         }
21     }

```

2-6 行目：read() でサーバからのメッセージを 17 バイト読み込む。

8-13 行目：メッセージと”REQUEST ACCEPTED\n”を比較し、一致していた場合、標準出力に”join request accepted”と表示し、state を 3 に設定する。

14-20 行目：一致していなかった場合、標準出力に”join request rejected”と表示し、state を 6 に設定する。

・状態3（ユーザ名の登録）

ここではユーザ名を送信し、登録できたか確認する。以下に抜粋したコードを示す。

```
1    if(state == 3){
2        strcpy(name,argv[2]);
3        name[strlen(name)] = '\n';
4
5        if ((nbytes = write(sock, name, sizeof(name))) < 0) {
6            perror("write");
7            exit(1);
8        }
9
10       bzero(rbuf, sizeof(rbuf));
11       if ((nbytes = read(sock, rbuf, 20)) < 0) {
12           perror("read");
13           exit(1);
14       }
15
16       if(strcmp(rbuf, "USERNAME REGISTERED\n") == 0){
17           if(write(1,"user name registered\n",21)<0){
18               perror("write");
19               exit(1);
20           }
21           state = 4;
22       }else{
23           if(write(1,"USERNAME REJECTED\n",18)<0){
24               perror("write");
25               exit(1);
26           }
27           state = 6;
28       }
29   }
```

2-3 行目：コマンドライン引数 argv[2] を配列 name にコピーし、最後に改行文字を入れる。

5-8 行目：取得したユーザ名をサーバに送信する。

10-14 行目：読み込みバッファを初期化し、サーバからのメッセージを20バイト読み込む。

16-21 行目：メッセージと”USERNAME REGISTERED\n”を比較し、一致していた場合、write() で標準出力に”user name registered”と表示し、state を4に設定する。

22-27 行目：一致していなかった場合、write() で標準出力に”USERNAME REJECTED”と表示し、state を6に設定する。

・状態4（メッセージの送受信）

ここではEOFが入力されるまでメッセージの送受信を行う。以下に抜粋したコードを示す。

```
1    if(state == 4){
2        do {
```

```

3      /* 入力を監視するファイル記述子の集合を変数rfds にセットする*/
4      FD_ZERO(&rfd); /* rfd を空集合に初期化*/
5      FD_SET(0, &rfd); /* 標準入力*/
6      FD_SET(sock, &rfd); /* クライアントを受け付けたソケット*/
7
8      /* 監視する待ち時間を1 秒に設定*/
9      tv.tv_sec = 1;
10     tv.tv_usec = 0;
11
12     /* 標準入力とソケットからの受信を同時に監視する*/
13     if (select(sock + 1, &rfd, NULL, NULL, &tv) > 0) {
14         if (FD_ISSET(0, &rfd)) { /* 標準入力から入力があったなら*/
15             /* 標準入力から読み込みクライアントに送信*/
16             if (fgets(sbuf, sizeof(sbuf), stdin) != NULL) {
17                 if ((nbytes = write(sock, sbuf, strlen(sbuf))) < 0) {
18                     perror("write");
19                     exit(1);
20                 }
21             }else{
22                 break;
23             }
24         }
25
26         if (FD_ISSET(sock, &rfd)) { /* ソケットから受信したなら*/
27             /* ソケットから読み込み端末に出力*/
28             bzero(rbuf, sizeof(rbuf));
29             if ((nbytes = read(sock, rbuf, sizeof(rbuf))) < 0) {
30                 perror("read");
31                 exit(1);
32             } else {
33                 write(1, rbuf, nbytes);
34             }
35         }
36     }
37     } while (1);
38     state = 5;
39 }

```

送受信を行う処理は、do-while でループし続け、標準入力に EOF が入力されたときに break する。

4-10 行目：標準入力とソケットを 1 秒間監視するように設定する。

13 行目：select() を用いて、監視している対象に入力があった場合処理を行う。

14-24 行目：標準入力から入力があった場合、入力を読み取る。NULL でないとき、メッセージをサーバに送信する。NULL のとき、break してループから出る。

26-35 行目：ソケットからの入力があった場合、読み込みバッファを初期化してから受信したメッセージを受け取り、標準出力に表示する。

ループを抜けたら state を 5 に設定する。

- ・状態 5、6 (終了)

ここではソケットのクローズを行う。以下に抜粋したコードを示す。

```
1     if(state == 5){
2         close(sock);
3         printf("closed\n");
4         exit(0);
5     }
6
7     if(state == 6){
8         close(sock);
9         printf("closed\n");
10        exit(1);
11    }
```

1-5 行目：state が 5 のとき、これは正常に終了しており、ソケットを閉じて”closed”を表示し、プログラムを終了する。

7-11 行目：state が 6 のとき、これは接続や登録に失敗しており、ソケットを閉じて”closed”を表示し、プログラムを異常終了する。

1.1.2 サーバプログラム

サーバプログラムのフローチャートを以下に示す。数字は状態番号を示している。

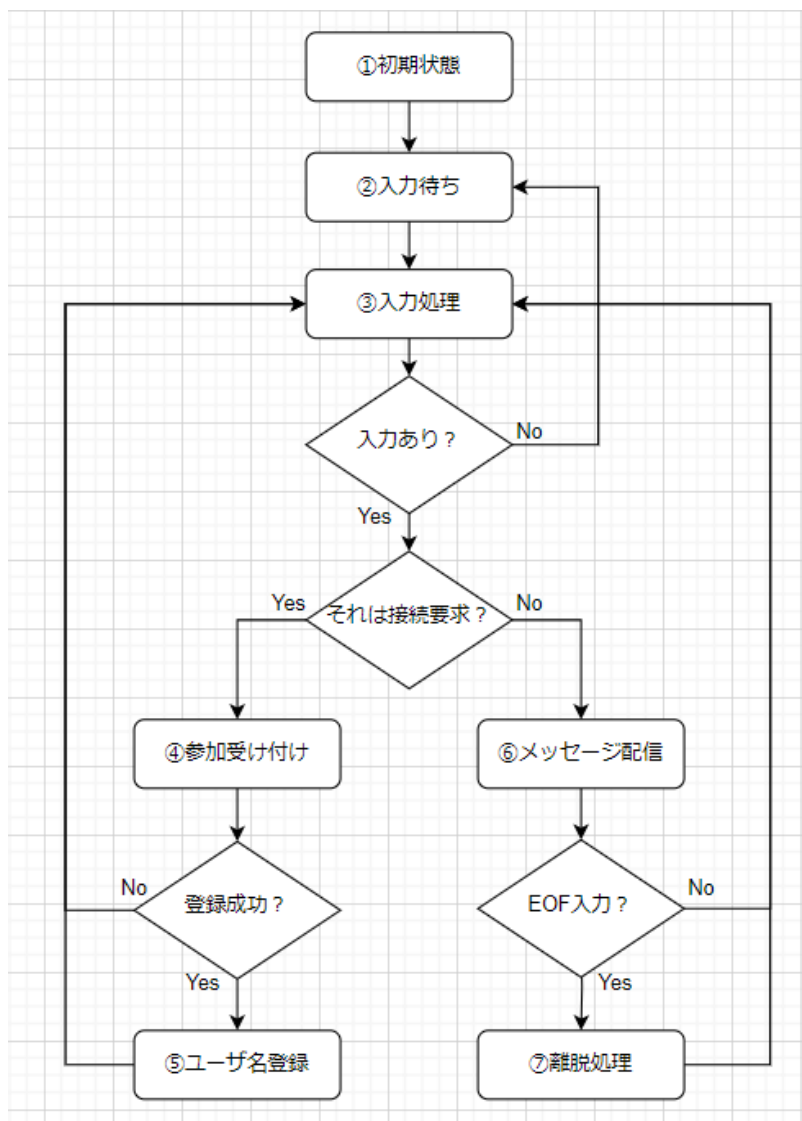


図2 サーバプログラムのフローチャート

実装方針で説明したとおり、サーバプログラムは状態を行き交い、動作し続ける必要があるため、ループを用いている。それ以外の記述の仕方はクライアントプログラムと同様で、変数 `state` の値によってそれぞれの状態ごとの処理を記述している。各状態の具体的な処理について説明していく。

まず宣言した変数および配列について型と役割をまとめた表を以下に示す。

変数・配列名	型	役割
MAXCLIENTS	#define	最大クライアント数の定義
state	int	プログラムの状態を管理する変数
i	int	ループインデックス、およびクライアントインデックス
select_flg	int	select() の戻り値を格納する変数
sock	int	サーバーのソケットファイルディスクリプタ
csock[MAXCLIENTS]	int 配列	クライアントのソケットファイルディスクリプタを格納する配列
svr	struct sockaddr_in	サーバーのアドレス情報を保持する構造体
clt	struct sockaddr_in	クライアントのアドレス情報を保持する構造体
cp	struct hostent*	ホストエントリ情報を指すポインタ
k	int	接続されたクライアントの数を管理する変数
nbytes	int	読み書きされたバイト数を保持する変数
clen	int	クライアントアドレスの長さを保持する変数
reuse	int	ソケットの再利用を設定する変数
rfds	fd_set	select() で用いるファイル記述子集合
tv	struct timeval	select() の待ち時間を指定する構造体
rbuf	char[1024]	ソケットからの受信データを格納するバッファ
name[MAXCLIENTS][99]	char[][]	各クライアントの名前を格納する配列
msg	char[1124]	クライアントからのメッセージを格納するバッファ

表 1 プログラムで宣言された変数と配列

これ以外にも状態内で用いたフラグがあるが、状態ごとの説明で記述する。

・状態 1（初期設定）

ここでは、ソケットの生成やサーバの初期化などを行う。以下に抜粋したコードを示す。

```

1  if (state == 1) {
2      // ソケットの生成
3      if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
4          perror("socket error");
5          exit(1);
6      }
7
8      // ソケットアドレス再利用の指定
9      reuse = 1;
10     if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse)) < 0) {
11         perror("setsockopt error");
12         exit(1);
13     }
14
15     // client 受付用ソケットの情報設定
16     bzero(&svr, sizeof(svr));
17     svr.sin_family = AF_INET;
18     svr.sin_addr.s_addr = htonl(INADDR_ANY);
19     svr.sin_port = htons(10140);

```

```

20
21 // ソケットにソケットアドレスを割り当てる
22 if (bind(sock, (struct sockaddr *)&svr, sizeof(svr)) < 0) {
23     perror("bind error");
24     exit(1);
25 }
26
27 // 待ち受けクライアント数の設定
28 if (listen(sock, 5) < 0) {
29     perror("listen error");
30     exit(1);
31 }
32
33 // 参加クライアント数を初期化
34 k = 0;
35
36 state = 2;
37 }

```

3-6 行目：ソケットの生成を行う。クライアントと同様、INET ドメインでストリーム型で指定する。

9-19 行目：ソケットアドレスの再利用を設定し、サーバのソケットアドレスを初期化する。

22-25 行目：bind() を用いてソケットとアドレスを結びつける。

28-31 行目：listen() を用いて待ち受けクライアント数を設定する。

34 行目：参加クライアント数を格納する変数 k を初期化する。

最後に、state を 2 に設定する。

・状態 2（入力待ち）

ここでは、サーバソケットとすでに接続されている各クライアントソケットを監視する。以下に抜粋したコードを示す。

```

1 if (state == 2) {
2     // 入力を監視するファイル記述子の集合を変数rfds にセットする
3     FD_ZERO(&rfd);
4     FD_SET(sock, &rfd);
5     int max_fd = sock;
6     for (int a = 0; a < k; a++) {
7         FD_SET(csock[a], &rfd);
8         if (csock[a] > max_fd) {
9             max_fd = csock[a];
10        }
11    }
12
13    // 監視する待ち時間を1 秒に設定
14    tv.tv_sec = 1;
15    tv.tv_usec = 0;
16    select_flg = select(max_fd + 1, &rfd, NULL, NULL, &tv);

```

```

17
18     if (select_flg < 0) {
19         perror("select error");
20         exit(1);
21     }
22
23     state = 3;
24 }

```

3-15 行目：標準入力と全てのクライアントソケットを1秒間監視するように設定する。ここで工夫した点が、select() の第一引数には、最も大きいファイル記述子の番号に1足したものを渡さなければいけないため、max_fd という変数を定義し、初期値としてソケットを入れ、それぞれ接続済みのクライアントソケットと比較して大きければ更新を繰り返していくようにし、これによって max_fd に最も大きいファイル記述子の番号が格納されるようにした。

16 行目：select() を実行し、その戻り値を select_flg に格納する。

18-21 行目：select() でエラーが起きていた場合、メッセージを表示して異常終了する。

最後に state を 3 に設定する。

・状態3（入力処理）

ここでは、入力があったのが接続要求なのかクライアントソケットからなのかによって遷移先を分ける。以下に抜粋したコードを示す。

```

1  if (state == 3) {
2      int c = 0;
3      if (select_flg > 0) {
4          if (FD_ISSET(sock, &rfdsets)) {
5              state = 4;
6              FD_CLR(sock, &rfdsets);
7          } else {
8              for (int a = 0; a < k; a++) {
9                  if (FD_ISSET(csock[a], &rfdsets)) {
10                     i = a;
11                     state = 6;
12                     FD_CLR(csock[a], &rfdsets);
13                     break;
14                 }else{
15                     c++;
16                 }
17             }
18             if(c == k){
19                 select_flg = 0;
20             }
21         }
22     } else {
23         state = 2;
24     }

```

今回作成したプログラムは関数呼び出しを用いていないため、監視する入力が増える場合ひとつひとつ処理していく必要があるため、集合から指定したファイル記述子を消去するマクロ `FD_CLR` を用いて、処理が完了したものから減らしていき、空になったら処理を終了する、という仕組みで行った。

2 行目：この変数は、入力がない、もしくは既に処理が完了したクライアントの数をカウントする変数である。

3 行目：`select_flg` が 0 より大きい、つまり何かしらの入力があったとき 4-20 行目の処理が行われる。

4-6 行目：サーバのソケットから入力があったとき、`state` を 4 にし、集合 `rfd`s からサーバソケットを取り除く。

7-21 行目：これらは何かしらの入力があったが接続要求の入力がなかった、もしくは既にその処理が完了した場合の処理である。まず、for ループで `rfd`s に `csock[a]` があるかチェックする。あった場合、`i` に `a` の値を格納し、`state` を 6 に設定し、集合 `rfd`s からそのクライアントソケットを取り除く。そして、`break` でループを抜ける。これによって複数のクライアントから入力があった場合にも番号が小さいクライアントから順にひとつひとつ処理を行うことができる。14-16 行目では、入力なしまたは完了のクライアントを数える。ループ後にカウンタ `c` が `k` と同値であったら、サーバソケットへの接続要求と既存クライアントからのメッセージ入力のどちらも処理が完了したということであるため、`select_flg` を初期化する。この初期化によって次の main ループで状態 2 に戻る。

・状態 4（接続処理）ここでは、新しく接続要求しているクライアントの処理を行う。以下に抜粋したコードを示す。

```

1  if (state == 4) {
2      clen = sizeof(cld);
3      int new_sock = accept(sock, (struct sockaddr *)&cld, &clen);
4      if (new_sock < 0) {
5          perror("accept error");
6          exit(2);
7      }
8
9      if (k < MAXCLIENTS) {
10         csock[k] = new_sock;
11         if (write(csock[k], "REQUEST ACCEPTED\n", 17) < 0) {
12             perror("write error");
13             exit(1);
14         }
15         state = 5;
16     } else {
17         if (write(new_sock, "REQUEST REJECTED\n", 17) < 0) {
18             perror("write error");
19             exit(1);
20         }
21         close(new_sock);
22         state = 3;
23     }
24 }
```

2-7 行目：`accept()` で新規クライアントの接続を `new_sock` に受け付ける。

9-15 行目：現在のクライアント数 k が MAXCLIENTS 未満であれば、新規クライアントのソケットを `csock[k]` に格納し、クライアントに接続承認メッセージを送信して `state` を 5 に設定する。

16-23 行目：クライアント数が最大数を超過している場合、接続拒否メッセージを新規クライアントに送信し、ソケット `new_sock` を閉じる。そして `state` を 3 に戻す。

・状態 5（ユーザ名登録）

ここでは、新規クライアントのユーザ名登録を行う。以下に抜粋したコードを示す。

```
1  if (state == 5) {
2      int flg = 0;
3      bzero(name[k], sizeof(name[k]));
4      if ((nbytes = read(csock[k], name[k], 99)) < 0) {
5          perror("read error");
6          exit(1);
7      }
8      name[k][strlen(name[k])-1] = '\0'; // 改行文字を削除
9
10     for (int a = 0; a < k; a++) {
11         if (strcmp(name[a], name[k]) == 0) {
12             flg = 1;
13         }
14     }
15
16     if (flg == 0) {
17         if (write(csock[k], "USERNAME REGISTERED\n", 20) < 0) {
18             perror("write error");
19             exit(1);
20         }
21         printf("%s is registered.\n", name[k]);
22         k++;
23         state = 3;
24     } else {
25         if (write(csock[k], "USERNAME REJECTED\n", 19) < 0) {
26             perror("write error");
27             exit(1);
28         }
29         printf("%s is rejected.\n", name[k]);
30         close(csock[k]);
31         csock[k] = 0;
32         state = 3;
33     }
34 }
```

2 行目：ここで用いる `flg` は名前の重複があった場合立てる。

3-8 行目：配列 `name` に新規クライアントから送られてくるユーザ名を読み込み、改行文字を消去する。

10-14 行目：既存クライアントのユーザ名と重複しないか比較し、被っていれば `flg` を立てる。

16-23 行目: flg が 0 のとき、新規クライアントにユーザ登録完了メッセージを送信し、サーバ側にも表示する。
k をインクリメントして状態 3 に戻る。

24-33 行目: flg がたっているとき、ユーザ名拒否メッセージを送信し、ソケットを閉じ、k+1 番目の csock も初期化して状態 3 に戻る。

・状態 6 (メッセージ受信)

ここでは、既存クライアントからのメッセージを全クライアントに送信する。以下に抜粋したコードを示す。

```
1  if (state == 6) {
2      bzero(rbuf, sizeof(rbuf));
3      nbytes = read(csock[i], rbuf, sizeof(rbuf));
4      if (nbytes < 0) {
5          perror("read error");
6          exit(1);
7      } else if (nbytes == 0) {
8          state = 7;
9      } else {
10         snprintf(msg, sizeof(msg), "%s>%s", name[i], rbuf);
11         for (int a = 0; a < k; a++) {
12             if (write(csock[a], msg, strlen(msg)) < 0) {
13                 perror("write error");
14                 exit(1);
15             }
16         }
17         state = 3;
18     }
19 }
```

2,3 行目: クライアント”i”からのメッセージ読み込む。

4-6 行目: 読み込みバイト数 nbytes が 0 より小さいときエラー処理をする。

7,8 行目: nbytes が 0 のとき、クライアントが切断したとみなして state を 7 に設定する。

9-18 行目: nbytes が 0 より大きいとき、クライアント名 name[i] を受信したメッセージに付けて全てのクライアントに送信する。送信完了後、状態 3 にもどる。

・状態 7 (離脱処理)

ここでは、切断したクライアントの処理を行う。以下に抜粋したコードを示す。

```
1  if (state == 7) {
2      printf("%s left the chat.\n", name[i]);
3      close(csock[i]);
4
5      for (int a = i; a < k - 1; a++) {
6          csock[a] = csock[a + 1];
7          strncpy(name[a], name[a + 1], 99);
8      }
9      csock[k - 1] = 0;
```

```

10     k--;
11
12     state = 3;
13 }

```

2,3 行目：サーバ側にクライアント”i”が切断したことを表示し、ソケットを閉じる。

5-8 行目：配列 csock と name について、クライアント”i”よりも後ろに格納されているクライアントソケットを一つ前に格納し直す。これによりクライアント”i”の情報を消去しつつ配列を詰めることができる。

9,10 行目：最も後ろの csock を初期化し、k をデクリメントする。

最後に状態 3 に戻る。

1.2 動作確認

動作確認では、サンプルと作成したプログラムを用いた 3 つの組み合わせ全ての実行結果を示す。成功例として 2 名が参加しメッセージを送受信して離脱する様子と失敗例として 6 人目の登録・既存参加者と重複した名前での登録を示す。

1.2.1 サンプルサーバと作成したクライアント

- ・ 2 名参加、メッセージ送受信、離脱

3 つのターミナルがあり、左がサーバ、右の 2 つがクライアントプログラムを動作している。

```

サーバ: sn-anzai@exp022:~/ensC4$ ./chatserver
mi: connected to server
mi: join request accepted
mi: user name registered
クライアント 1: 4$ gcc -o c client.c
クライアント 1: 4$ ./c exp022 USER1
クライアント 2: 4$ gcc -o c client.c
クライアント 2: 4$ ./c exp022 USER2

```

サーバのホスト名とユーザ名”USER1”, ”USER2”を引数として実行すると、接続とユーザ名登録に成功して、サーバ側には”(ユーザ名) is registered”が、クライアント側には”connected to server”, ”join request accepted”, ”user name registered”が表示された。

```

サーバ: sn-anzai@exp022:~/ensC4$ ./chatserver
mi: connected to server
mi: join request accepted
mi: user name registered
mi: hello, I am USER1.
クライアント 1: sn-anzai@exp021:~/ensC4$ ./c exp022 USER1
クライアント 2: sn-anzai@exp020:~/ensC4$ ./c exp022 USER2

```

クライアント 1 から”Hello, I am USER1”を送信してみる。

```

サーバ: sn-anzai@exp022:~/ensC4$ ./chatserver
mi: connected to server
mi: join request accepted
mi: user name registered
mi: hello, I am USER1.
mi: USER1 >hello, I am USER1.
クライアント 1: sn-anzai@exp021:~/ensC4$ ./c exp022 USER1
クライアント 2: sn-anzai@exp020:~/ensC4$ ./c exp022 USER2

```

すると、クライアント 1 ・ 2 の両方に”USER1>Hello, I am USER1”が出力された。

<pre> miiiiiiiiiiii left the chat. ^C sn-anzai@exp022:~/ensC4\$./chatserver USER1 is registered. USER2 is registered. </pre>	<pre> connected to server join request accepted user name registered hello,I am USER1. USER1 >hello,I am USER1. </pre>	<pre> sn-anzai@exp020:~/ensC4\$./c exp022 USER2 connected to server join request accepted user name registered USER1 >hello,I am USER1. hi,I am USER2. </pre>
---	---	---

次に、クライアント 2 から”Hi, I am USER2”を送信してみる。

<pre> miiiiiiiiiiii is registered. miiiiiiiiiiii left the chat. ^C sn-anzai@exp022:~/ensC4\$./chatserver USER1 is registered. USER2 is registered. </pre>	<pre> connected to server join request accepted user name registered hello,I am USER1. USER1 >hello,I am USER1. USER2 >hi,I am USER2. </pre>	<pre> connected to server join request accepted user name registered USER1 >hello,I am USER1. hi,I am USER2. USER2 >hi,I am USER2. </pre>
--	--	---

すると、クライアント 1・2 の両方に”USER2>Hi, I am USER2”が出力された。

次に、クライアント 2 に Ctrl+D で EOF を入力した。

<pre> miiiiiiiiiiii is registered. miiiiiiiiiiii left the chat. ^C sn-anzai@exp022:~/ensC4\$./chatserver USER1 is registered. USER2 is registered. USER2 left the chat. USER1 left the chat. </pre>	<pre> sn-anzai@exp021:~/ensC4\$./c exp022 USER1 connected to server join request accepted user name registered hello,I am USER1. USER1 >hello,I am USER1. USER2 >hi,I am USER2. </pre>	<pre> connected to server join request accepted user name registered USER1 >hello,I am USER1. hi,I am USER2. USER2 >hi,I am USER2. closed sn-anzai@exp020:~/ensC4\$ </pre>
--	---	--

すると、サーバ側には”USER2 left the chat.”が、クライアント 2 には”closed”が表示され、プログラムが終了した。

<pre> miiiiiiiiiiii is registered. miiiiiiiiiiii left the chat. ^C sn-anzai@exp022:~/ensC4\$./chatserver USER1 is registered. USER2 is registered. USER2 left the chat. USER1 left the chat. </pre>	<pre> sn-anzai@exp021:~/ensC4\$./c exp022 USER1 connected to server join request accepted user name registered hello,I am USER1. USER1 >hello,I am USER1. USER2 >hi,I am USER2. closed sn-anzai@exp021:~/ensC4\$ </pre>	<pre> sn-anzai@exp020:~/ensC4\$./c exp022 USER2 connected to server join request accepted user name registered USER1 >hello,I am USER1. hi,I am USER2. USER2 >hi,I am USER2. closed sn-anzai@exp020:~/ensC4\$ </pre>
--	--	---

同様に、クライアント 1 も EOF を入力することで離脱が完了した。

- ・既存参加者と重複した名前で参加（ユーザ名登録エラー）

<pre> USER2 is registered. USER2 left the chat. USER1 left the chat. USER1 is registered. </pre>	<pre> sn-anzai@exp021:~/ensC4\$./c exp022 USER1 connected to server join request accepted user name registered </pre>	<pre> USER1 >hello,I am USER1. hi,I am USER2. USER2 >hi,I am USER2. closed sn-anzai@exp020:~/ensC4\$./c exp022 USER1 </pre>
--	--	--

再度クライアント 1 で”USER1”というユーザ名で参加した後、クライアント 2 で同名の”USER1”で参加を試みる。

<pre> USER2 left the chat. USER1 left the chat. USER1 is registered. USER1 is rejected. </pre>	<pre> sn-anzai@exp021:~/ensC4\$./c exp022 USER1 connected to server join request accepted user name registered </pre>	<pre> sn-anzai@exp020:~/ensC4\$./c exp022 USER1 connected to server join request accepted USERNAME REJECTED sn-anzai@exp020:~/ensC4\$ </pre>
--	--	---

すると、サーバには”USER1 is rejected”が、クライアント 2 には”USERNAME REJECTED”が表示されプログラムが終了した。

- ・ 6 人目参加（接続要求エラー）

右のサーバに対して 5 つのクライアントで参加した後、左のクライアントで、重複していない名前に参加を試みる。すると、クライアントに”join request rejected”が表示されプログラムが終了した。

```
Your Hardware Enablement Stack (HWE) is supported until April 2025.
sn-anzai@exp020:~$ cd ensC4
sn-anzai@exp020:~/ensC4$ ./a.out exp014 chibimaruko
connected to server
join request rejected
closed
sn-anzai@exp020:~/ensC4$
```

```
sn-anzai@exp014:~/ensC4$ ./chatserver
ruffy is registered.
tanjiro is registered.
doraemon is registered.
naruto is registered.
goku is registered.
```

以上から、サンプルサーバと作成したクライアント間で正常に動作していることがわかる。

1.2.2 作成したサーバとサンプルクライアント

1.2.1 節と同様、左からサーバ、クライアント 1、クライアント 2 で動作確認を行った。

- ・ 2 名参加、メッセージ送受信、離脱

```
sn-anzai@exp022:~/ensC4$ gcc -o s server.c
sn-anzai@exp022:~/ensC4$ ./s
USER1 is registered.
USER2 is registered.
```

```
sn-anzai@exp021:~/ensC4$ ./chatclient exp022 USER1
ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted
user name registered
```

```
sn-anzai@exp020:~/ensC4$ ./chatclient exp022 USER2
ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted
user name registered
```

サーバのホスト名とユーザ名”USER1”, ”USER2”を引数として実行すると、接続とユーザ名登録に成功して、サーバ側には”(ユーザ名) is registered”が、クライアント側には”connected to server”, ”join request accepted”, ”user name registered”が表示された。

```
sn-anzai@exp022:~/ensC4$ gcc -o s server.c
sn-anzai@exp022:~/ensC4$ ./s
USER1 is registered.
USER2 is registered.
```

```
sn-anzai@exp021:~/ensC4$ ./chatclient exp022 USER1
ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted
user name registered
hello,I am USER1.
USER1>hello,I am USER1.
```

```
sn-anzai@exp020:~/ensC4$ ./chatclient exp022 USER2
ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted
user name registered
USER1>hello,I am USER1.
```

クライアント 1 から”Hello, I am USER1”を送信してみると、クライアント 1 ・ 2 の両方に”USER1>Hello, I am USER1”が出力された。

```
sn-anzai@exp022:~/ensC4$ ./s
k=0 strlen=6
USER1 is registered.
^C
sn-anzai@exp022:~/ensC4$ gcc -o s server.c
sn-anzai@exp022:~/ensC4$ ./s
USER1 is registered.
USER2 is registered.
```

```
sn-anzai@exp021:~/ensC4$ ./chatclient exp022 USER1
ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted
user name registered
hello,I am USER1.
USER1>hello,I am USER1.
USER2>hi,I am USER2.
```

```
sn-anzai@exp020:~/ensC4$ ./chatclient exp022 USER2
ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted
user name registered
USER1>hello,I am USER1.
hi,I am USER2.
USER2>hi,I am USER2.
```

クライアント 2 から”Hi, I am USER2”を送信してみると、クライアント 1 ・ 2 の両方に”USER2>Hi, I am USER2”が出力された。

```

sn-anzai@exp022:~/ensC4$ gcc -o s server.c
sn-anzai@exp022:~/ensC4$ ./s
USER1 is registered.
USER2 is registered.
USER1 left the chat.
USER2 left the chat.

USER1>hello,I am USER1.
USER2>hi,I am USER2.
sn-anzai@exp021:~/ensC4$ ./chatclient exp022 USER1
ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted

ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted
user name registered
USER1>hello,I am USER1.
hi,I am USER2.

```

クライアント 1・2 に EOF を入力すると離脱が完了した。

- ・既存参加者と重複した名前で参加（ユーザ名登録エラー）

```

sn-anzai@exp022:~/ensC4$ ./s
USER1 is registered.
USER2 is registered.
USER1 left the chat.
USER2 left the chat.
USER1 is registered.
USER1 is rejected.

USER2>hi,I am USER2.
sn-anzai@exp021:~/ensC4$ ./chatclient exp022 USER1
ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted
user name registered

connected to exp022
join request accepted
user name registered
USER1>hello,I am USER1.
hi,I am USER2.
USER2>hi,I am USER2.
sn-anzai@exp020:~/ensC4$ ./chatclient exp022 USER1

```

再度クライアント 1 で”USER1”というユーザ名で参加した後、クライアント 2 で同名の”USER1”で参加を試みる。

```

USER2 is registered.
USER1 left the chat.
USER2 left the chat.
USER1 is registered.
USER1 is rejected.

sn-anzai@exp021:~/ensC4$ ./chatclient exp022 USER1
ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted
user name registered

sn-anzai@exp020:~/ensC4$ ./chatclient exp022 USER1
ICS Exercises C sample program chatclient.c
connected to exp022
join request accepted
USERNAME REJECTED
sn-anzai@exp020:~/ensC4$

```

すると、サーバには”USER1 is rejected”が、クライアント 2 には”USERNAME REJECTED”が表示されプログラムが終了した。

- ・6 人目参加（接続要求エラー）

右のサーバに対して 5 つのクライアントで参加した後、左のクライアントで、重複していない名前で参加を試みる。

```

join request accepted
user name registered
^C
sn-anzai@exp015:~/ensC4$ ./chatclient exp014 pancake
ICS Exercises C sample program chatclient.c
connected to exp014
join request rejected
sn-anzai@exp015:~/ensC4$

sn-anzai@exp014:~/ensC4$ gcc -o server new-server.c
sn-anzai@exp014:~/ensC4$ ./server
donut is registered.
pudding is registered.
cookie is registered.
candy is registered.
chocolate is registered.

```

すると、クライアントに”join request rejected”が表示されプログラムが終了した。

以上から、作成したサーバとサンプルクライアント間で正常に動作していることがわかる。

1.2.3 作成したサーバとクライアント

1.2.1 節, 1.2.2 節と同様、左からサーバ、クライアント 1、クライアント 2 で動作確認を行った。

- ・2 名参加、メッセージ送受信、離脱

```

^C
sn-anzai@exp028:~/ensC4$ ./s
USER1 is registered.
USER2 is registered.

sn-anzai@exp003:~/ensC4$ ./c exp028 USER1
connected to server
join request accepted
user name registered

sn-anzai@exp004:~/ensC4$ ./c exp028 USER2
connected to server
join request accepted
user name registered

```

サーバのホスト名とユーザ名”USER1”, ”USER2”を引数として実行すると、接続とユーザ名登録に成功して、サーバ側には”(ユーザ名) is registered”が、クライアント側には”connected to server”, ”join request

accepted”, ”user name registered”が表示された。

```
sn-anzai@exp028:~/ensC4$ ./s
USER1 is registered.
USER2 is registered.
[

Hello, i am USER1
USER1>Hello, i am USER1
USER2>Hi, i am USER2
[

USER1>Hello, i am USER1
Hi, i am USER2
USER2>Hi, i am USER2
[
```

クライアント 1 から”Hello, I am USER1”を送信してみると、クライアント 1・2 の両方に”USER1>Hello, I am USER1”が出力された。また、クライアント 2 から”Hi, I am USER2”を送信してみると、クライアント 1・2 の両方に”USER2>Hi, I am USER2”が出力された。

- ・既存参加者と重複した名前で参加（ユーザ名登録エラー）

```
USER1 left the chat.
USER2 left the chat.
USER1 is registered.
[

connected to server
join request accepted
user name registered
[

Hi, i am USER2
USER2>Hi, i am USER2
closed
sn-anzai@exp004:~/ensC4$ ./c exp028 USER1
```

再度クライアント 1 で”USER1”というユーザ名で参加した後、クライアント 2 で同名の”USER1”で参加を試みる。

```
USER2 is registered.
USER1 left the chat.
USER2 left the chat.
USER1 is registered.
USER1 is rejected.
[

closed
sn-anzai@exp003:~/ensC4$ ./c exp028 USER1
connected to server
join request accepted
user name registered
[

sn-anzai@exp004:~/ensC4$ ./c exp028 USER1
connected to server
join request accepted
USERNAME REJECTED
closed
sn-anzai@exp004:~/ensC4$
```

すると、サーバには”USER1 is rejected”が、クライアント 2 には”USERNAME REJECTED”が表示されプログラムが終了した。

- ・6 人目参加（接続要求エラー）

右のサーバに対して 5 つのクライアントで参加した後、左のクライアントで、重複していない名前で参加を試みる。

```
Your Hardware Enablement Stack (HWE) is supported until April 2025.
sn-anzai@exp007:~$ cd ensC4
sn-anzai@exp007:~/ensC4$ ./a.out exp002 tanjiro
connected to server
join request rejected
closed
sn-anzai@exp007:~$
sn-anzai@exp007:~/ensC4$ [

sn-anzai@exp002:~/ensC4$ gcc -o news new-server.c
sn-anzai@exp002:~/ensC4$ ./news
anzai is registered.
shun is registered.
roland is registered.
ryoma is registered.
gunshin is registered.
```

すると、クライアントに”join request rejected”が表示されプログラムが終了した。

以上から、作成したサーバと作成したクライアント間で正常に動作していることがわかり、サンプルと入れ替えた状態でも問題なく動作できることが分かった。

2 課題 4-2

2.1 実装した機能

実装した機能は、

- ・各発言の先頭に時刻を表示する機能
- ・参加や離脱が生じたら、誰が参加したか、誰が離脱したか、現在の参加者総数は何人であるなどといった情報が

全参加者の端末に表示される機能
の2つである。

2.1.1 実装方針とアルゴリズム

・時刻表示機能

ライブラリ `time.h` を用いて時刻を取得し、`state6` でメッセージを全ユーザに送信するバッファに時刻情報を格納した配列を加えてから送信する。以下に追加・変更した部分のプログラムについて説明する。

まず、追加した時刻を取得する関数を抜粋したコードを示す。

```
1 #include <time.h>
2
3 void get_time_string(char *buffer, size_t size) {
4     time_t now = time(NULL);
5     struct tm *t = localtime(&now);
6     strftime(buffer, size, "%Y-%m-%d %H:%M:%S", t);
7 }
```

`time` 関数でカレンダー時間を取得し、`localtime`, `strftime` で文字列に変換する。

次に、時刻を送信する機能を追加した `state6` を以下に示す。

```
1 if (state == 6) {
2     bzero(rbuf, sizeof(rbuf));
3     nbytes = read(csock[i], rbuf, sizeof(rbuf));
4     if (nbytes < 0) {
5         perror("read");
6         exit(1);
7     } else if (nbytes == 0) {
8         state = 7;
9     } else {
10        char time_str[100];
11        get_time_string(time_str, sizeof(time_str));
12        snprintf(msg, sizeof(msg), "%s %s>%s", time_str, name[i], rbuf);
13        for (int a = 0; a < k; a++) {
14            if (write(csock[a], msg, strlen(msg)) < 0) {
15                perror("write");
16                exit(1);
17            }
18        }
19        state = 3;
20    }
21 }
```

10,11 行目を追加し、12 行目を一部変更した。既存参加者からメッセージを受信したとき、`get_time_string` 関数を呼び出して時刻を文字列で取得し、その文字列・メッセージを送信した参加者の名前・メッセージ内容を `snprintf` で配列 `msg` にまとめ、`write()` で送信する。

・参加者情報表示機能

まず、参加時に送信する機能を追加した state5 を抜粋したコードを示す。

```
1  if (state == 5) {
2      int flg = 0;
3      bzero(name[k], sizeof(name[k]));
4      if ((nbytes = read(csock[k], name[k], 99)) < 0) {
5          perror("read error");
6          exit(1);
7      }
8      name[k][strlen(name[k])-1] = '\0'; // 改行文字を削除
9
10     for (int a = 0; a < k; a++) {
11         if (strcmp(name[a], name[k]) == 0) {
12             flg = 1;
13         }
14     }
15
16     if (flg == 0) {
17         write(csock[k], "USERNAME REGISTERED\n", 20);
18         printf("%s is registered.\n", name[k]);
19
20         // 参加メッセージを作成
21         snprintf(msg, sizeof(msg), "USER JOINED: %s (Total users: %d)\n", name[k], k + 1)
22             ;
23         for (int a = 0; a <= k; a++) {
24             if (write(csock[a], msg, strlen(msg)) < 0) {
25                 perror("write");
26                 exit(1);
27             }
28
29             k++;
30             state = 3;
31         } else {
32             write(csock[k], "USERNAME REJECTED\n", 19);
33             printf("%s is rejected.\n", name[k]);
34             close(csock[k]);
35             csock[k] = 0;
36             state = 3;
37         }
38     }
```

追加したのは 21-27 行目の部分である。接続要求とユーザ名登録に成功した後、全既存参加者に新しく参加したユーザ名と合計参加者数を送信する。

次に、離脱時に送信する機能を追加した state7 を抜粋したコードを示す。

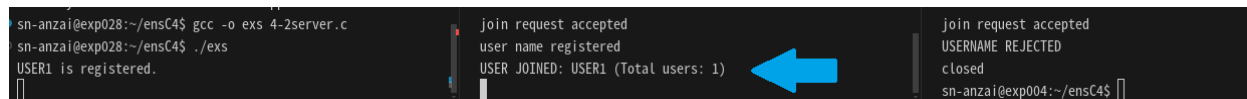
```
1 if (state == 7) {
2     printf("%s left the chat.\n", name[i]);
3     close(csock[i]);
4
5     // 離脱メッセージを作成
6     snprintf(msg, sizeof(msg), "USER LEFT: %s (Total users: %d)\n", name[i], k - 1);
7     close(csock[i]);
8
9     for (int a = i; a < k - 1; a++) {
10         csock[a] = csock[a + 1];
11         strncpy(name[a], name[a + 1], 99);
12     }
13     csock[k - 1] = 0;
14     k--;
15     for (int a = 0; a < k; a++) {
16         if (write(csock[a], msg, strlen(msg)) < 0) {
17             perror("write");
18             exit(1);
19         }
20     }
21
22     state = 3;
23 }
```

追加したのは 6 行目,15-20 行目の部分である。state7 に入った時点で離脱処理であるため、離脱者以外の全参加者に離脱者のユーザ名と残りの合計参加者数を送信する。

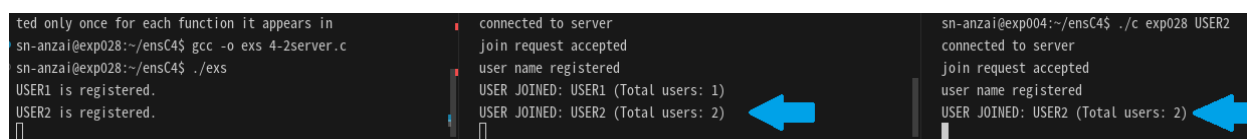
2.2 動作確認

4-1 と同様、左からサーバ、クライアント 1、クライアント 2 で動作確認を行った。

- ・参加時情報表示機能



クライアント 1 が”USER1”というユーザ名で実行すると、接続成功のメッセージのあとに”USER JOINED : USER1 (Total users: 1)”が表示された。



クライアント 2 が”USER2”というユーザ名で実行すると、クライアント 1 と 2 両方に”USER JOINED :

USER2 (Total users: 2)”が表示された。

- ・時刻表示機能

```
4-2server.c:101:17: note: each undeclared identifier is reported only once for each function it appears in
sn-anzai@exp028:~/ensC4$ gcc -o exs 4-2server.c
sn-anzai@exp028:~/ensC4$ ./exs
USER1 is registered.
USER2 is registered.

join request accepted
user name registered
USER JOINED: USER1 (Total users: 1)
USER JOINED: USER2 (Total users: 2)
Hello, I am USER1
2024-08-06 12:43:15 USER1>Hello, I am USER1

sn-anzai@exp004:~/ensC4$ ./c exp028 USER2
connected to server
join request accepted
user name registered
USER JOINED: USER2 (Total users: 2)
2024-08-06 12:43:15 USER1>Hello, I am USER1
```

”Hello, I am USER1”というメッセージをクライアント 1 から送信した。すると、時刻”2024-08-06 12:43:15”がメッセージの前に追加された。

```
ted only once for each function it appears in
sn-anzai@exp028:~/ensC4$ gcc -o exs 4-2server.c
sn-anzai@exp028:~/ensC4$ ./exs
USER1 is registered.
USER2 is registered.

USER JOINED: USER1 (Total users: 1)
USER JOINED: USER2 (Total users: 2)
Hello, I am USER1
2024-08-06 12:43:15 USER1>Hello, I am USER1
2024-08-06 12:43:32 USER2>Hi, I am USER2

user name registered
USER JOINED: USER2 (Total users: 2)
2024-08-06 12:43:15 USER1>Hello, I am USER1
Hi, I am USER2
2024-08-06 12:43:32 USER2>Hi, I am USER2
```

同様に、”Hi, I am USER2”というメッセージをクライアント 2 から送信した。すると、時刻”2024-08-06 12:43:32”がメッセージの前に追加された。

- ・離脱時情報表示機能

```
sn-anzai@exp028:~/ensC4$ gcc -o exs 4-2server.c
sn-anzai@exp028:~/ensC4$ ./exs
USER1 is registered.
USER2 is registered.
USER1 left the chat.

USER JOINED: USER2 (Total users: 2)
Hello, I am USER1
2024-08-06 12:43:15 USER1>Hello, I am USER1
2024-08-06 12:43:32 USER2>Hi, I am USER2
closed
sn-anzai@exp003:~/ensC4$

USER JOINED: USER2 (Total users: 2)
2024-08-06 12:43:15 USER1>Hello, I am USER1
Hi, I am USER2
2024-08-06 12:43:32 USER2>Hi, I am USER2
USER LEFT: USER1 (Total users: 1)
```

クライアント 1 に Ctrl+D で EOF を入力すると、クライアント 2 に離脱を知らせる”USER LEFT: USER1 (Total users: 1)”が表示された。

以上から、追加した機能が正常に動作していることが分かる。

3 考察

今回のようなチャットシステムを作る際に、サーバクライアント型が最も最適なのかを考察した。調べてみると、ピアツーピアというサーバがなくクライアント同士でやりとりする方式があることが分かった。ピアツーピアのメリットは、サーバを介さずに端末同士で直接やりとりをするため、サーバの運用コストがないことや、特定のノードがダウンしてもネットワーク全体が使えなくなることはないことである。デメリットとしては、ノード同士の通信のため、どうしてもセキュリティ面ではサーバクライアント方式に劣ってしまうことである。LINE のトーク履歴やデータファイルなどのやりとりはピアツーピア方式が採用されているようなので、重要性が高いものの通信には有効であると考えられる。

4 感想・謝辞

通信のプログラムをデバッグするときは、printf() があまり使えないため苦戦した。分からない部分はサンプルサーバとサンプルクライアントの挙動を見ながら合わせられたため、やりやすかった。先生方、TA の方々、半年間ありがとうございました。