前期実験I1I2I3

実験開始日：2020年4月13日

実験終了日：2020年5月11日

報告提出日：2020年5月26日

山本桃歌

電気電子工学科3年　23班

学籍番号：03100519

双方向電話（基本課題）

TCPでのソケット通信を用いて双方向電話を作成する。双方向電話を作成するための複数の手順を説明する。まとめるとソケットを作り接続する、双方向通信を始めると同時にrecを起動するようにする、ループ文で音声データを送り合う、受け取ったデータを音声に変える、の四つのステップをこなす必要がある。

ソケットを作り接続する。(クライアント側)

```
char *filename = argv[1]; //IPアドレスの入力
int port_number = atoi(argv[2]); //ポート番号の入力

int s = socket(PF_INET, SOCK_STREAM, 0); // ソケットを作る。SOCK_STREAMでTCPを用いた通信を指定
struct sockaddr_in addr; // 接続するアドレス情報
addr.sin_family = AF_INET; // このアドレスの通信体系はIPv4
int aton = inet_aton(filename, &addr.sin_addr);; // IP アドレスの指定
addr.sin_port = htons(port_number); // ポート番号の指定
int ret = connect(s, (struct sockaddr *)&addr, sizeof(addr)); // 接続する
```

ソケットを作り接続する。(サーバ側)

```
port_number = atoi(argv[1]); //ポート番号の入力

int ss = socket(PF_INET, SOCK_STREAM, 0);

struct sockaddr_in addr; // 最終的にbind に渡すアドレス情報
addr.sin_family = AF_INET; // このアドレスはIPv4 アドレスです
addr.sin_port = htons(port_number); // 待ち受けするポート番号の指定
addr.sin_addr.s_addr = INADDR_ANY; // 自分の持つどのIPアドレスでも待ち受けする
bind(ss, (struct sockaddr *)&addr, sizeof(addr)); // ソケットに情報を割り当てる

listen(ss, 10); // 接続可能宣言  クライアントと通信をするためのソケットを受け取る。

struct sockaddr_in client_addr;
socklen_t len = sizeof(struct sockaddr_in);
printf("waiting for connect...");
int s = accept(ss, (struct sockaddr *)&client_addr, &len); // 接続が来るまで待機
```

接続を開始してからrec を起動する

教科書通りに　rec -t raw -b 16 -c 1 -e s -r 44100 - | ./serv send ポート番号　とターミナルに打ち込んで電話を開始させてしまうと、サーバを立ち上げた直後の音が音が0.8秒分ほど流れてしまう。このことの原因はサーバが上記コードの accept(ss, (struct sockaddr *)&client_addr, &len); の部分が実行されるより前にターミナル側のrec -t raw -b 16 -c 1 -e s -r 44100 -が開始してしまう事である。

これを解決するために、popenというライブラリ関数を使用した。popen() 関数は、プロセスをオープンする関数である。具体的には、 パイプを生成し、フォークを行い、シェルを起動する。パイプは一方向なので第二引き数には読み込み"r"か書き込み"w"のどちらか一方を指定する 生成されるストリームは、この指定に対応して、読み取り専用または 書き込み専用のいずれかとなる。

上記のソケットを作るコードの後に以下のコードを書いた(serv.cとclient.c両方)。つまり fp_in=popen(cmd_rec,"r") でrecへの読み込みモードのコマンドパイプを作り、fread(data_rec, 1, N, fp_in); でパイプからデータを読み込む。これでaccept()が完了した後にrecを始めることができる。

popen()を使うもう一つの利点はターミナルで打つコマンドを減らせることである。この利点を生かして受け取った音声情報をplayするのもpopen(cmd_play,"w")によってプログラムにやらせることができる。

```
FILE  *fp_in;
char  *cmd_rec = "rec -t raw -b 16 -c 1 -e s -r 44100 - 2>/dev/null";
if ( (fp_in=popen(cmd_rec,"r")) ==NULL) {
   perror ("can not rec");
   exit(EXIT_FAILURE);
}
FILE  *fp_out;
char  *cmd_play = "play -t raw -b 16 -c 1 -e s -r 44100 -";
if ( (fp_out=popen(cmd_play,"w")) ==NULL) {
   perror ("can not play");
   exit(EXIT_FAILURE);
}
unsigned char data_rec[N];
unsigned char data_play[N];
while(1){
   int k = fread(data_rec, 1, N, fp_in); //recコマンドのパイプからreadする。
   if (k == 0) break;
   write(s,data_rec,N);

   int n = read(s,data_play,N);
   if (n == -1){perror("read");exit(1);}
```

```
    if (n>0){
        fwrite(data_play, 1, N, fp_out); //playコマンドのパイプへwriteする。
    }
    if(n == 0) printf("END\n");
    if(n == 0) break;
}
```

ソースコードserv.cとclient.cの使い方

serv.cは実行時に第一引数に使いたいポート番号を入れる。

実行例　./serv 50000

client.cは実行時に第一引数にIPアドレスを、第二引数にポート番号を入れる。

実行例　./client 127.0.0.1 50000

スレッドを用いたサーバ（発展課題）

ソースコード

発展課題のソースコードはthread.c, nothread.c, client2.cの三つである。実行の仕方はserv.c,client.cの時と同じである。thread.c, nothread.cはともにクライアントが送った文字列をそのまま送り返すエコー機能を備えている。thread.c, nothread.cは実行時に第一引数に使いたいポート番号を入れる。

実行例　./serv 50000

client2.cは実行時に第一引数にIPアドレスを、第二引数にポート番号を入れる。

実行例　../client 127.0.0.1 50000

課題内容

いままでサーバとクライアントの一対一の通信しか扱ってこなかった。クライアントが複数いる場合について考えてみる。クライアントを立てるときに listen(ss, 10); で第二引数を10としているため10までののクライアントとつなげることができる。しかし二つ目のクライアント以降はサーバと通信ができていないことがわかった。（nothread.cでサーバを立ち上げ、client2.cで二つクライアントを作る。一つつめのクライアントに文字を打ち込むとエコーが返ってくる。しかし二つ目のクライアントに何かを打ち込んでも何も起きない）

そこでスレッド機能をクライアントに入れた。クライアントは複数スレッドを作り、そのスレッド内で個別にクライアントと通信をする。thread.cはスレッドを用いているため複数のクライアントと個別にエコーを送ることができる。

スレッドの作成

pthread_create()でスレ(土)を作成する。実際のコードは以下の通り。

```
pthread_create(pthread_t *thread ,NULL,server,&s) //スレッドの作成
pthread_join(pthread_t *thread,NULL)  //スレッドの終了


void *server(void *arg){ //スレッド内で実行される関数
    int clients;
    int s = *(int *)arg;
    struct sockaddr_in claddr;
```

```
    socklen_t claddr_size = sizeof claddr;
    char buf[2048];
    int r;
    int sen;
    if((clients = accept(s,(struct sockaddr *)&claddr,&claddr_size))<0){perror("accept
error");exit(1);}
    while(1){
        memset(buf,0,sizeof(buf));
        if((r=recv(clients, buf, sizeof(buf), 0))<0){perror("recv error");exit(1);}
         write(0,buf,r);
        if((sen=send(clients,buf,r,0))<0){perror("send error");exit(1);}
    }
    close(clients);
}
```

スレッドを用いた通信

thread.cではサーバは接続してきたクライアントそれぞれに対して、それぞれのスレッド内でaccept()しエコー処理をしている。他のスレッドのクライアントの通信によって他のスレッドは影響されない。

バッファの大きさの影響（考察）

実験8.3が完成し双方通信できるようにした時、二通りの方法でこれを確認した。パソコンを二台用意する方法とターミナルを二つ建てる方法。ターミナルを二つ建て双方通信ができているか確認した時は、「あ」と喋ると遅延ののち「.....あ...あ」と二つ聞こえてきたのでserv.cとclient.cの二つ分の声が聞こえると言うことで8.3の完成を確認した。

なぜ二つ音が聞こえたか

serv.cとclient.cでバッファのサイズの設定が違った。バッファのサイズとは一回の通信で送られるデータの大きさである。バッファサイズをサーバで100,000クライアントで10と設定しまっていたため、サーバから送られる音声とクライアントから送られる音声の遅延に差が出てきてしまったと考えられる。

バッファの容量だけデータがためられてから初めてソケットに送られる。つまりバッファの容量が大きすぎると音声データの入力からソケットに送るまでの間に時間がかかりすぎてしまい遅延が起きる。

ここで言うバッファの大きさとは以下の疑似コードのNである。

```
const int N = 30000; //バッファの大きさを決定
unsigned char data_rec[N];
unsigned char data_play[N];

while(1){
  int k = fread(data_rec, 1, N, fp_in);
  write(s,data_rec,N);

  int n = read(s,data_play,N);
  if (n > 0){ fwrite(data_play, 1, N, fp_out);}
}
```

双方通信の正しい確認方法

実際に耳をすませて音を確認するのではサーバとクライアントのどちらがデータを受け取り音をスピーカーから発生させたかわからないが、ターミナルのUIでスピーカーに送っていることを確認する

ことができる。音をplayする時に [　 |　] となっている部分が [-===|===-]のように音の発生を示す。これが両方のターミナルで発生しているのを確認できれば双方通信ができている確認である。

バッファの大きさによってどれくらい遅延が起きるのか（追加実験）

バッファの大きさによる遅延の大きさはバッファの大きさと線型であると考えた。考えが正しいかどうかを確かめるため実際にバッファの大きさを変えて遅延をはかる実験を試した。

実験方法

双方通信している時に無音の状態から音声をマイクへ入力しターミナルのUIまたはスピーカーから音を確認できた時間を様々なバッファの大きさで測定してバッファサイズと遅延の関係を調べる。

電話機能を維持したまま、サーバから送られてきた音がクライアントまで届き音声がクライアント側から再生されるまでの時間を計るプログラムの作り方がわからなかったため、原始的に自らの耳とストップウォッチで遅延時間を測定した。

マイク付きのイヤホンをパソコンに取り付け、マイクをパソコンにぶつけ「カツン」と音を鳴らす。音を鳴らすと同時にスマホのストップウォッチをスタートさせ、イヤホンから音が聞こえたらストップウィッチを止める。実験に使用したプログラムは提出したコードのclient.c(クライアント)とserv.c(クライアント)である。

測定数値

バッファサイズ N [bit] (サーバとクライアント共に同じサイズ)	遅延時間　測定値 [s]	遅延時間測定値の平均 [s]
1	1.93, 2.09, 1.84, 2.01, 1.80, 1.87,1.97, 1.95, 2.01, 1.81, 2.04, 1.84, 1.90, 1.91	1.92
2	0.82, 0.72, 0.89, 0.84, 0.90, 0.87, 0.79, 0.86, 0.87, 0.58, 0.83, 0.72,0.91, 0.75,0.82, 0.87, 0.86	0.81
3	0.40, 0.41, 0.44, 0.47, 0.50, 0.48, 0.37, 0.48, 0.57, 0.48, 0.51, 0.41, 0.35, 0.37, 0.29, 0.39, 0.36	0.42
4	0.22, 0.38, 0.30, 0.26,0.35, 0.38, 0.31, 0.43, 0.36, 0.34,0.35	0.33
5	0.38, 0.37, 0.32, 0.33, 0.35, 0.35, 0.37, 0.40, 0.43, 0.34, 0.35, 0.27	0.35
8	0.50, 0.36, 0.51, 0.36, 0.37, 0.46, 0.37, 0.45, 0.42, 0.34, 0.40, 0.51, 0.34, 0.38,0.45, 0.30, 0.50	0.41
32	0.36, 0.37, 0.37, 0.39, 0.34, 0.37, 0.39, 0.43, 0.32, 0.37, 0.37, 0.35, 0.43,	0.37

	0.48, 0.38, 0.33, 0.39	
64	0.42, 0.44, 0.47, 0.41,0.50, 0.48, 0.26, 0.47, 0.51, 0.38, 0.40, 0.43, 0.31, 0.40, 0.37, 0.44, 0.47	0.42
100	0.32, 0.36, 0.35, 0.33	0.34
128	0.35, 0.35, 0.38, 0.28, 0.31, 0.34, 0.39, 0.21, 0.41, 0.29, 0.35, 0.27, 0.35, 0.31, 0.37, 0.30, 0.31, 0.44	0.33
1000	0.37, 0.45, 0.37, 0.34	0.38
10,000	0.33, 0.55, 0.49, 0.38, 0.41, 0.40, 0.45, 0.37, 0.37, 0.45, 0.43	0.42
12,345	0.45, 0.31, 0.35, 0.40, 0.38, 0.33, 0.44, 0.45, 0.41	0.39
30,000	0.75, 0.50, 0.61, 0.71, 0.70, 0.70, 0.50, 0.70, 0.44, 0.52, 0.66, 0.61, 0.65, 0.61	0.61
50,000 (クライアント側から聞こえた音)	1.13, 1.04, 1.05, 1.07, 1.07, 1.16	1.08
50,000 (サーバ側から聞こえた音)	1.60, 1.44 ,1.44, 1.42, 1.53, 1.51	1.49
100,000 (クライアント側から聞こえた音)	1.56, 1.74, 1.60, 1.64, 1.41, 1.56, 1.68, 1.59, 1.55, 1.65, 1.57	1.59
100,000 (サーバ側から聞こえた音)	2.65, 2.77, 2.67, 2.74, 2.68, 2.72, 2.66, 2.67, 2.67, 2.55, 2.64	2.67
123,456 (クライアント側から聞こえた音)	1.89, 1.86, 1.97, 1.89, 2.01, 1.88, 1.92, 2.12, 1.88, 2.04, 1.95, 1.88, 2.00, 1.90, 2.03, 1.97, 2.04, 1.87	1.95
123,456 (サーバ側から聞こえた音)	3.28,3.20, 3.26, 3.08, 3.28, 3.24, 3.24, 3.33, 3.32, 3.27, 3.28, 3.24, 3.30, 3.27, 3.34, 3.31, 3.27, 3.1	3.25
200,000 (クライアント側から聞こえた音)	5.05, 4.98, 5.17, 4.93, 4.93, 5.01, 5.34, 5.21	5.07
200,000 (サーバ側から聞こえた音)	5.44, 5.11, 5.74, 5.27, 5.33, 5.29, 5.67, 5.46	5.41
300,000 (クライアント側から聞こえた音)	3.99, 5.20, 5.40, 5.15, 4.01, 4.03, 8.60	4.75
300,000 (サーバ側から聞こえた音)	8.80, 10.21, 10.30, 10.12, 8.68, 8.70	9.62

測定結果

サーバとクライアント間の音声データ通信にかかった遅延時間とは表の通り。ただし N=50,000以降はserv.cとclient.cの大きさが等しかったのにもかかわらず、まずクライアント側から（サーバから送られてきた）音が聞こえ、その後に時間差でサーバ側から音が観測された。右のグラフはそれぞれのNの値における平均遅延時間とNの関係である。遅延時間はN=300,000 の時を除き、電話を立ち上げてからの時間と関係なく一定であった。

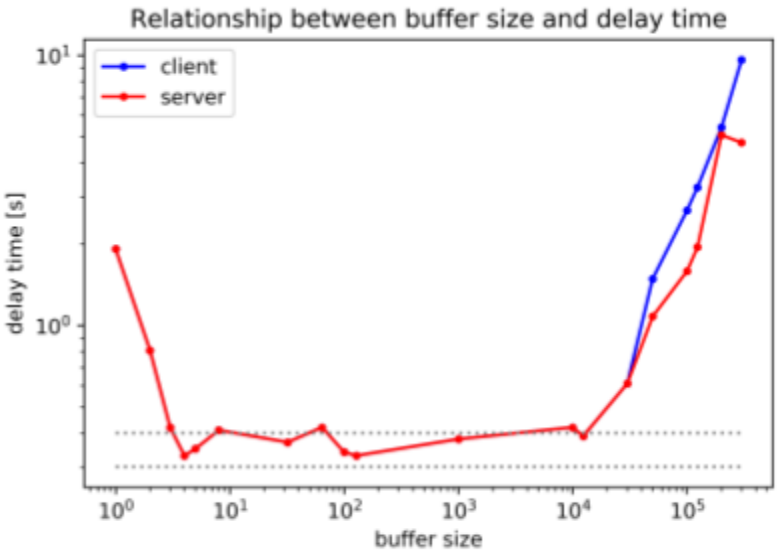

図1　Nの大きさと遅延時間の関係

N=300,000 の時は測定回によって遅延時間が大きく異なったためあまり信用できるデータではないとも考えられる。N=300,000 の時、音の音質が悪くなり、音が途切れ途切れになると同時にターミナルの時計も止まるようになってしまった。試しに「（時計動いてる）あー（時計止まった）いー（時計動いてる）うー（時計止まった）えー（時計動いてる）おー（時計止まった）かー」と時計が動いているか止まっているかに合わせてしゃべるものを変えてみた。すると「うー」と「おー」の時だけ帰ってきた音声が一時止まった。このことから時計が止まっている時は録音に支障があったとも考えられる。

推定誤差

マイクに送られた音声がスピーカーになるまでの時間を手で計測したためあまり正確なデータとは言えない。音を聞いてからストップウォッチを止めるまでの時間を考慮すると遅延時間は上記のデータより少し短いと考えられる。複数回数測定市平均を出すことでNの変化による遅延時間の変化を測定しようと考えた。

考察

明らかに N=1,2 の時と N≧30,000 で遅延時間は長くなっていた。それ以外のNの値では遅延時間の平均はは0.3s~0.4sの間にあった。

N=1,2 の時に遅延時間が長かったのは一回の通信で送るデータが短かったために、通信を捨回数がフタタためと考えられる。通信自体に時間がかかるため通信回数が多いのも遅延の原因となってしまう。

N≧30,000 で遅延時間は長くなっていたのは一回の通信で送る情報をrecするまでに時間がかかってしまったことが要因と考えられる。1秒分のデータは44,100bytes×2 であるため44,100bytes×2ごとに1秒遅延が起きる可能性を考え、delay time [s] = buffer size/88200 の補助線を実験結果のグラフに付け加えてみた。（図2）しかしこれはN≧30,000とは完全に重ならなかった。そこでdelay time [s] = buffer size/44100 の補助線を引いた。（図3）これはN≧30,000の時のグラフと重なった。これは偶然なのかきちんとした理由があって重なっているのかはよくわからなかった。

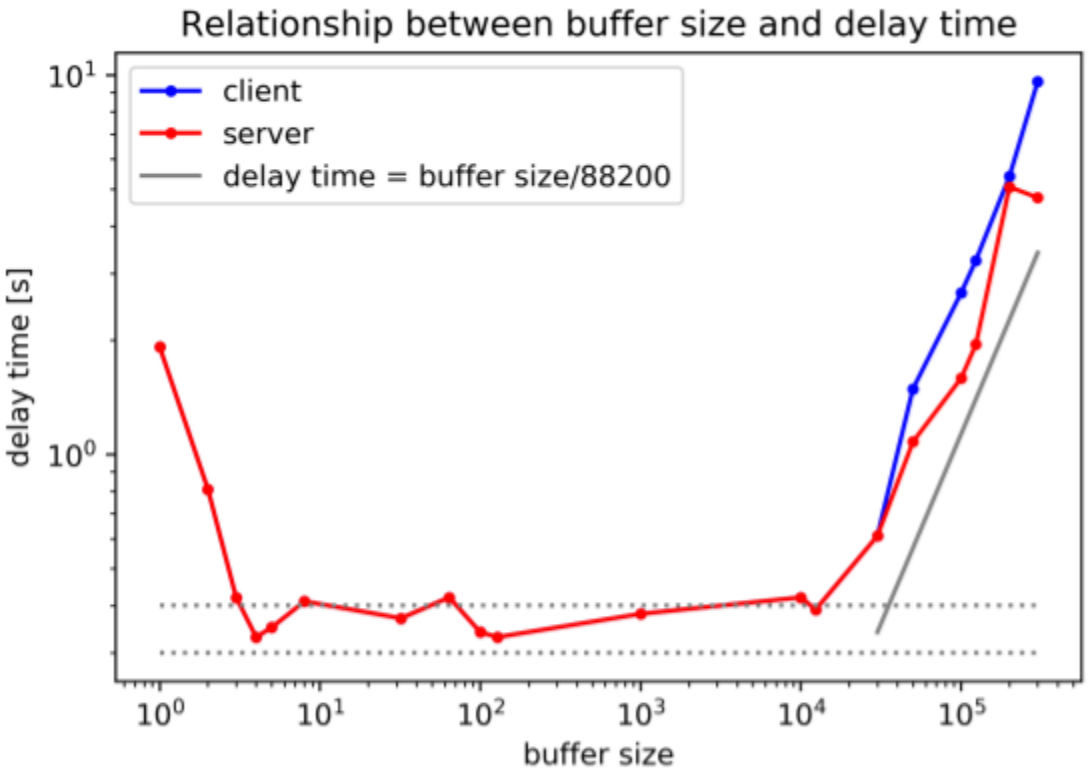

図2　delay time [s] = buffer size/88200 の補助線付き

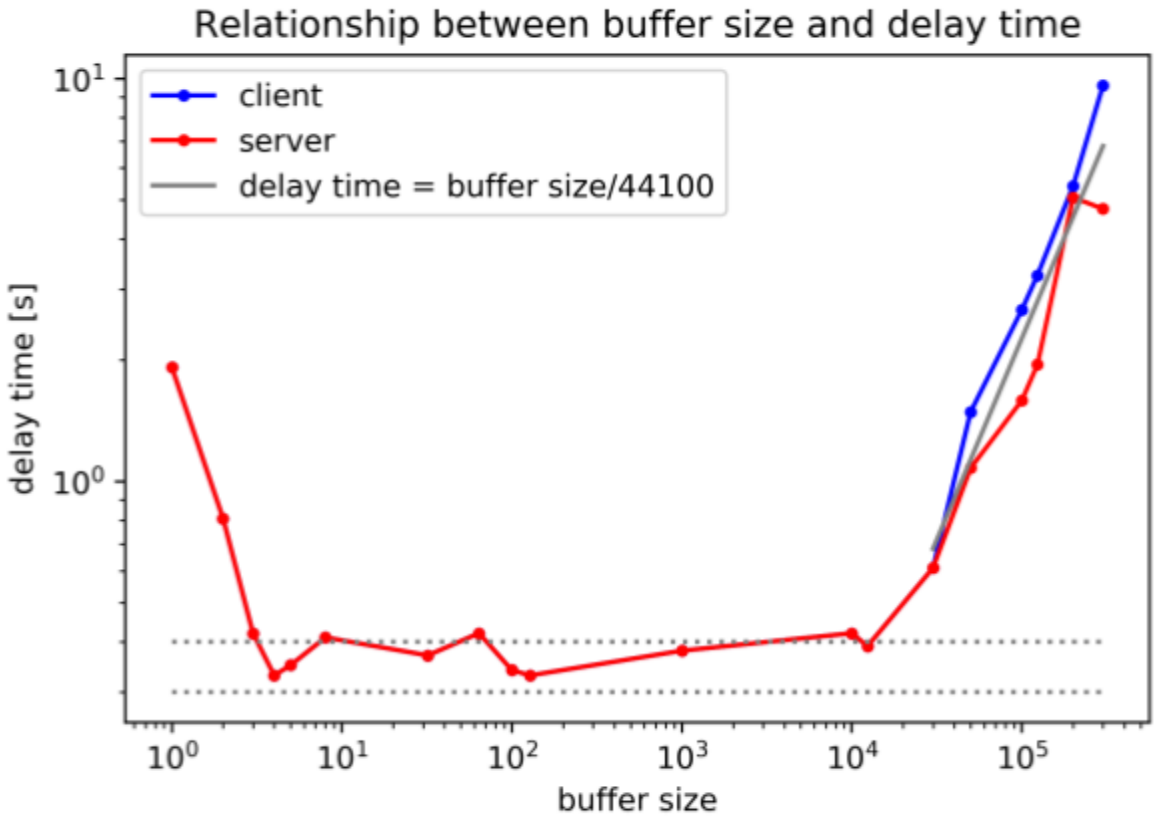

図3　delay time [s] = buffer size/44100 の補助線付き

References（参考文献）

双方向電話（基本課題）

popen:

Man page of POPEN https://linuxjm.osdn.jp/html/LDP_man-pages/man3/popen.3.html

スレッドを用いたサーバ（発展課題）

pthread_create:

Man page of PTHREAD_CREATE https://linuxjm.osdn.jp/html/glibc-linuxthreads/man3/pthread_create.3.html

pthread_createしたらjoinを忘れない https://www.hakodate-ct.ac.jp/~tokai/tokai/gtkmm/etc/p4.htm