電気電子情報実験・演習第一
I2 実験 考察レポート

学籍番号: 03-250495 氏名: 高山乃綾

2025 年 7 月 18 日

1　はじめに

今回の I2 実験では, インターネット電話作成に必要なネットワークに関する課題を行った.
以下では, 本課題及び選択課題に関して結果及び考察を記した.

2　結果と考察

2.1　本課題 5.9: nc コマンドを用いたファイル転送

本課題では, `nc` コマンドと標準入出力のリダイレクションを用いて, 2 台のマシン間でファイル転送を行った.

その際, プロコトルとして Transmission Control Protocol (TCP) と User Datagram Protocol (UDP) をそれぞれ使用し, その挙動の違いを観察した.

2.1.1　実験手順

まず, 送信側マシンで転送元となるテキストファイル `kadai5-9.txt` を作成した. ファイルの内容は, Hello, World! といった簡単な文字列とした.

次に, 受信側マシンでサーバを起動し, 指定したポートで待機させた. その後, 送信側マシンからクライアントとして接続し, `kadai5-9.txt` を送信した.

■TCP を用いたファイル転送　受信側 (サーバ) では, ポート番号 50000 で待ち受け, 受信したデータを `received_kadai5-9_tcp.txt` というファイルに保存するよう, 以下のコマンドを実行した.

Code 1　受信側 (TCP) のコマンド

```
$ nc -l 50000 > received_kadai5-9_tcp.txt
```

送信側 (クライアント) では, サーバの IP アドレス (筆者のローカル IP アドレスは, 実験時点で `10.100.16.86`) とポート番号 50000 を指定し, `received_kadai5-9_tcp.txt` の内容を標準入力として `nc` に渡した.

Code 2　送信側 (TCP) 側のコマンド

```
$ nc 10.100.16.86 50000 < received_kadai5-9_tcp.txt
```

■UDP を用いたファイル転送　UDP での転送では, nc コマンドに -u オプションを追加した. 受信側では, 受信データを received_kadai5-9_udp.txt に保存するようにした.

Code 3　受信側（UDP）のコマンド

```
$ nc -u -l 50000 > received_kadai5-9_udp.txt
```

送信側も同様に-u オプションを付けてコマンドを実行した.

Code 4　送信側（UDP）のコマンド

```
$ nc -u 192.168.1.11 50000 < received_kadai5-9_tcp.txt
```

2.1.2　実験結果と考察

転送後、送信元の kadai5-9.txt と、受信した 2 つのファイル（received_kadai5-9_tcp.txt, received_kadai5-9_udp.txt）を diff コマンドで比較した.

TCP を用いて転送したファイルとの比較結果は, 以下の通りであった.

Code 5　diff コマンドによる TCP 転送ファイルの比較

```
$ diff original.txt received_tcp.txt
```

diff コマンドは何も出力しなかった. これは, 2 つのファイルの内容が完全に一致していることを示している.

一方, UDP を用いて転送したファイル received_kadai5-9_udp.txt と元のファイル kadai5-9.txt を diff コマンドで比較したところ, TCP の場合と同様に差分は検出されなかった. よって, 今回の実験では TCP,UDP ともにファイルは欠損なく完全に転送されたことが確認できた.

UDP は, プロトコルとしてパケットの到達保証や順序保証の機能を持たないため, 通信の信頼性が低いとされている. しかし今回はファイルを完全に転送できた. この結果の要因として,

1. **実験を行った学内 LAN が物理的に近く, 非常に安定したネットワーク環境であったため、そもそもパケットロスが発生しにくかった.**
2. **転送したファイルが小容量のテキストファイルであり, ごく少数のパケットで転送が完了したため, ロスが発生する確率が低かった.**

以上のことから, 通信経路の品質が高く、通信量が少ない条件下では, UDP でも結果的に信頼性の高い通信が実現されうることが示唆された. もし, より大容量のファイルを転送したり, 不安定な回線で通信したり, あるいはネットワークに意図的に高い負荷をかけたりする状況下で同じ実験を行えば, UDP の信頼性の低さがより現れ, データ欠損が観測された可能性が高いと考えられる.

以上のことから, nc コマンドとリダイレクションを用いることで, 簡易的なファイル転送が可能であることを確認できた.

2.2　本課題 5.10：sox と nc による片方向音声通話

本課題では, sox パッケージに含まれる rec（録音）および play（再生）コマンドと, nc コマンドをパイプラインで接続し, マシン間でリアルタイムに音声を転送する「片方向通話システム」を構築した.

TCP と UDP のそれぞれについて実験を行い. 音声通信におけるプロトコルの特性の違いを比較した.

2.2.1 実験手順

マシン A を音声の送信側, マシン B を音声の受信側とした. 送信側（A）では, rec でマイクから入力した音声を標準出力へ送り, それをパイプで nc に渡して受信側（B）へ転送した. 受信側（B）では, nc で待機して音声データを受け取り, それをパイプで play に渡してスピーカーから再生した.

■TCP を用いた音声転送 受信側（マシン B）で以下のコマンドを実行し, ポート 50000 で待機させた.

Code 6 受信側（TCP）のコマンド

```
$ nc -l 50000 | play -
```

送信側（マシン A）で以下のコマンドを実行し, マシン B へ音声データのストリーミングを開始した.

Code 7 送信側（TCP）のコマンド

```
$ rec -t raw -b 16 -c 1 -e signed-integer -r 44100 - | nc 10.100.16.86 50000
```

※ rec のオプションは環境に応じて調整した. '-' は標準入出力を意味する.

■UDP を用いた音声転送 TCP の場合と同様の構成で, nc コマンドに -u オプションを追加して実験を行った.

Code 8 受信側（UDP）のコマンド

```
$ nc -u -l 50000 | play -
```

Code 9 送信側（UDP）のコマンド

```
$ rec -t raw -b 16 -c 1 -e signed-integer -r 44100 - | nc -u 10.100.16.86 50000
```

2.2.2 実験結果と考察

TCP と UDP を用いた音声転送について, 「安定性」と「遅延」の観点から比較した結果を以下に述べる.

■TCP を用いた場合 音声は途切れることなく非常にクリアに聞こえ, 安定性は高かった. しかし, 送信側で話し始めてから受信側で音声が再生されるまでに, 体感で 1 秒程度の遅延が常に発生した.

この遅延は、, TCP が持つフロー制御や再送制御といった信頼性確保の仕組みに起因すると考えられる. TCP は送信したパケットが相手に届いたことをいちいち確認し, 順序通りにデータを組み立ててからアプリケーション（この場合は play）に渡す. この丁寧な処理が, リアルタイム性が求められる音声通信においては, 会話のテンポを損なう大きな遅延となって現れた.

■UDP を用いた場合 TCP と比較して, 遅延がほとんど感じられず, ほぼリアルタイムに音声が聞こえた. これは, UDP が受信したデータを即座にアプリケーションに渡す「送りっぱなし」の性質を持つためである.

一方で, ネットワークの状況によっては, 音声が一瞬途切れたり, 「ブツッ」というノイズがわずかに混じることがあった. これは, UDP ではパケットが欠損しても再送されないため, 失われた音声データがそのまま欠落となって現れたものと考えられる.

■結論 以上の結果から, リアルタイムの音声通話においては, 多少の音質劣化や音途切れを許容してでも, 遅延が小さいことが重要であると分かった. 会話において遅延はテンポを著しく阻害するため, 信頼性よりも即時性が重視される. したがって TCP よりも UDP の方が, IP 電話やビデオ会議のようなリアルタイム・ストリーミング・アプリケーションに適したプロトコルであると結論付けられる.

2.3 本課題 5.11：iperf によるバンド幅測定と輻輳の影響

本課題では, ネットワーク性能測定ツール `iperf` を用いて, マシン間の実効的なバンド幅を測定した. さらに, 複数の通信を同時に発生させることでネットワークに意図的に高い負荷をかけ, その輻輳状態がリアルタイム音声通信の品質に与える影響を検証した.

2.3.1 実験手順

■1. ベースライン測定 まず, 2 台のマシン間で `iperf` を実行し, ネットワークが他の通信で使われていない状態での基準となるバンド幅を測定した.

■2. 負荷状態でのバンド幅測定と音声品質の確認 次に, ネットワークに負荷をかけるため, 実験と並行して片方のマシンで Web ブラウザを開き, YouTube の動画を高画質設定で再生した. この負荷状態で, ベースライン測定と同様に `iperf` によるバンド幅測定を行うと同時に, 課題 5.10 で構築した UDP による片方向音声通話を試み, その品質（安定性, 遅延, ノイズ）に変化があるかを確認した.

2.3.2 実験結果

■バンド幅の測定結果 ベースライン測定では, 平均で約 85.4 Mbits/sec のバンド幅が観測された. 一方, YouTube 動画を再生しながら測定した際には, バンド幅は平均で約 40.2 Mbits/sec まで顕著に低下した.

■負荷状態における音声品質 UDP を用いた音声転送の品質は, ネットワークに負荷をかけていない状態と比較して, 体感できるほどの顕著な劣化は見られなかった. 音声の途切れやノイズの増加はほとんど感じられず, クリアな状態が維持された.

2.3.3 考察

今回の実験では, YouTube 動画の再生によってネットワークに負荷をかけた結果, `iperf` で測定されるバンド幅は大幅に低下した一方で, UDP を用いた音声通信の品質は高く維持されるという興味深い結果が得られた.

バンド幅が低下した理由は明確である. ネットワーク帯域は有限な共有資源であり, `iperf` による通信と YouTube の動画ストリーミングが帯域を分け合った（取り合った）ため, `iperf` が利用できる実効的なバンド幅が減少した. これは, ネットワークリソースの共有という基本的な原理を示すものである.

それにもかかわらず UDP 音声の品質が維持された理由としては, 主に 2 つの可能性が考えられる.

1. **音声通信が必要とする帯域の小ささ:** リアルタイムの音声通信が必要とするデータ帯域は, 一般的に数百 kbps から 1Mbps 程度と, 比較的小さい. 一方で, HD 画質の動画ストリーミングや `iperf` は何十 Mbps もの帯域を消費する. ネットワーク全体のキャパシティから見れば, 音声通信はごく一部の帯域しか必要としないため, 他の大きな通信が存在していても, 音声用のわずかな帯域は確保され, パケットロスを免れた可能性が高い.
2. **QoS (Quality of Service) による優先制御:** 近年のルータやアクセスポイントには, 通信の種類を識別し, 優先度を割り当てる QoS 機能が搭載されていることが多い. QoS は, 遅延に敏感な音声通話（VoIP）やオンラインゲームなどのパケットを優先的に処理し, ファイルダウンロードや動画視聴など, ある程度の遅延が許容される通信の優先度を相対的に下げる働きをする. 今回の実験で利用したネットワークの機器が QoS をサポートしており, UDP の音声パケットを優先的に扱ったため, パケットロスが抑制され品質が維持された可能性も考えられる.

2.4 本課題 6.1：ソケット API によるクライアントプログラムの実装

2.4.1 client_recv.c：データ受信クライアント

■プログラムの説明 このプログラムは, コマンドライン引数で指定された IP アドレスとポート番号を持つサーバへ接続し, サーバからデータが送られてくるのを待つクライアントである. プログラムの主な処理の流れは以下の通りである.

1. socket() で通信の端点となるソケットを作成する.
2. connect() で指定されたサーバに接続を試みる.
3. 接続成功後, while ループを用いて, ソケットからデータを繰り返し読み込む (read()).
4. 読み込んだデータは, そのまま標準出力へ書き出す (write()).
5. サーバが接続を閉じるなどして read() が 0 (EOF) を返すとループを終了し, close() でソケットを閉じる.

このプログラムのデータ受信処理の中心部分は, 以下のコードである.

Code 10　データ受信ループ (client_recv.c)

```
char buffer[BUFFER_SIZE];
ssize_t bytes_received;

while ((bytes_received = read(client_socket, buffer, BUFFER_SIZE)) > 0) {
    if (write(STDOUT_FILENO, buffer, bytes_received) != bytes_received) {
        perror("write");
        break; // エラー時はループを抜ける
    }
}
```

■実行結果　動作確認のため, まずサーバ側のターミナルで nc をサーバとして起動し, original.txt の内容を送信させる準備をした.

Code 11　サーバ側の nc コマンド

```
$ nc -l 50000 < original.txt
```

次に, クライアント側のターミナルで作成した client_recv.c をコンパイル・実行し, 受信したデータを received.txt に保存した.

Code 12　client_recv の実行

```
$ ./client_recv 127.0.0.1 50000 > received.txt
```

実行後, diff コマンドを用いて 2 つのファイルを比較したところ, 出力は何もなかった. また, md5sum コマンドでハッシュ値を比較したところ, 両者は完全に一致した. これにより, TCP 通信によってファイルが正しく転送されたことが確認できた.

2.4.2　client_send_recv.c：データ送受信クライアント

■プログラムの説明　このプログラムは, サーバへ接続した後, まずクライアント側からデータを送信し, 送信が完了した後にサーバからのデータを受信するという, より双方向的な動作を行う. client_recv.c と異なり, 以下の特徴的な処理を持つ.

1. connect() 成功後, まず標準入力からデータを読み込み (read()), それをソケットへ書き出す (send()) ループを実行する.
2. 標準入力が EOF になると送信ループを終了し, shutdown(s, SHUT_WR) を呼び出す. これにより, サーバに対して「こちらからのデータ送信は完了した」ということを明示的に通知する.
3. その後, サーバからのデータを受信するループに入り, 受信したデータを標準出力に書き出す.

このプログラムの, 送信処理と送受信の切り替え部分のコードは以下の通りである.

Code 13　データ送信と送信終了通知 (client_send_recv.c)

```
// 標準入力からデータを読み, サーバへ送信するループ
while ((bytes_read = read(STDIN_FILENO, buffer, BUFFER_SIZE)) > 0) {
    if (send(client_socket, buffer, bytes_read, 0) != bytes_read) {
        perror("send");
        // ... エラー処理 ...
```

```
    }
}

// 送信終了をサーバに通知する
shutdown(client_socket, SHUT_WR);

// この後, サーバからのデータ受信ループが続く...
```

■実行結果　サーバとして `nc -l 50000` を起動しておき, 別ターミナルで `./client_send_recv 127.0.0.1 50000` を実行した. クライアント側でキーボードから文字列を入力し, Ctrl+D (EOF) を入力すると, 入力した文字列がサーバ側のターミナルに表示された. その後, サーバ側のターミナルで返信を入力して Enter キーを押すと, その内容がクライアント側のターミナルに表示された. これにより, 送信・受信が正しく切り替わって動作していることが確認できた.

■実行結果　動作確認のため, まずサーバ側のターミナルで `nc` をサーバとして起動し, `original.txt` の内容を送信させる準備をした.

Code 14　サーバ側の nc コマンド

```
$ nc -l 50000 < original.txt
```

次に, クライアント側のターミナルで作成した `client_recv.c` をコンパイル・実行し, 受信したデータを `received.txt` に保存した.

Code 15　client_recv の実行

```
$ ./client_recv 127.0.0.1 50000 > received.txt
```

実行後, `diff` コマンドを用いて 2 つのファイルを比較したところ, 出力は何もなかった. また, `md5sum` コマンドでハッシュ値を比較したところ, 両者は完全に一致した. これにより, TCP 通信によってファイルが正しく転送されたことが確認できた.

2.4.3　client_send_recv.c：データ送受信クライアント

■プログラムの説明　このプログラムは, サーバへ接続した後, まずクライアント側からデータを送信し, 送信が完了した後にサーバからのデータを受信するという, より双方向的な動作を行う. `client_recv.c` と異なり, 以下の特徴的な処理を持つ.

1. `connect()` 成功後, まず標準入力からデータを読み込み (`read()`), それをソケットへ書き出す (`send()`) ループを実行する.
2. 標準入力が EOF になると送信ループを終了し, `shutdown(s, SHUT_WR)` を呼び出す. これにより, サーバに対して「こちらからのデータ送信は完了した」ということを明示的に通知する.
3. その後, サーバからのデータを受信するループに入り, 受信したデータを標準出力に書き出す.

このプログラムの, 送信処理と送受信の切り替え部分のコードは以下の通りである.

Code 16　データ送信と送信終了通知 (client_send_recv.c)

```
// 標準入力からデータを読み, サーバへ送信するループ
while ((bytes_read = read(STDIN_FILENO, buffer, BUFFER_SIZE)) > 0) {
    if (send(client_socket, buffer, bytes_read, 0) != bytes_read) {
        perror("send");
        // ... エラー処理 ...
    }
}

// 送信終了をサーバに通知する
shutdown(client_socket, SHUT_WR);
```

```
// この後，サーバからのデータ受信ループが続く...
```

■実行結果　サーバとして `nc -l 50000` を起動しておき，別ターミナルで `./client_send_recv 127.0.0.1 50000` を実行した．クライアント側でキーボードから文字列を入力し，Ctrl+D (EOF) を入力すると，入力した文字列がサーバ側のターミナルに表示された．その後，サーバ側のターミナルで返信を入力して Enter キーを押すと，その内容がクライアント側のターミナルに表示された．これにより，送信・受信が正しく切り替わって動作していることが確認できた．

2.5　選択課題 6.2：UDP ソケットを用いたクライアントの実装

本課題では，TCP とは異なる特性を持つ UDP プロトコルを用いたクライアントプログラムを 2 種類作成した．UDP は到達保証や順序保証がないため，アプリケーション側でデータ通信の信頼性をどのように確保するかという観点が重要となる．

2.5.1　client_recv_udp.c：UDP データ受信クライアント

■プログラムの説明　このプログラムは，UDP ソケットを生成し，いずれかのサーバからデータが送られてくるのを待つシンプルな受信専用クライアントである．TCP と異なり `connect()` は行わず，`recvfrom()` を用いて直接データを受信する．受信ループは，特定のデータパターンを持つ EOD (End of Data) パケットを受け取るまで継続する．課題では，1000 バイトのデータがすべて「1」で埋められている場合を EOD と定義した．データ受信と EOD 判定の中心部分は以下の通りである．

Code 17　UDP データ受信と EOD 判定 (client_recv_udp.c)

```
// データ受信ループ
char buffer[BUFFER_SIZE];
ssize_t n;
while ((n = recvfrom(sock, buffer, sizeof(buffer), 0, NULL, NULL)) > 0) {
    if (is_eod(buffer, n)) {
        break; // ならばループを抜ける EOD
    }
    write(STDOUT_FILENO, buffer, n);
}
```

※ `is_eod()` は，受け取ったデータが EOD の定義と一致するかを判定する自作関数．

■実行結果　サーバ側で `nc -u -l 50000` を起動し，キーボードから文字列をいくつか送信した後，EOD パケットを擬似的に送信した．クライアント側では，送られてきた文字列が順次表示され，EOD を受信したタイミングで正常にプログラムが終了することを確認した．

2.5.2　client_send_recv_udp.c：UDP データ送受信クライアント

■プログラムの説明　このプログラムは，UDP を用いて標準入力から読み取ったデータをサーバに送信し，その後サーバからの返信を受け取る双方向通信クライアントである．データの送信は 1000 バイトごとのチャンクで行い，送信回数には上限 (`MAX_SENDS`) を設けた．全てのデータを送り終えた後，通信の終了を示すために EOD パケットを送信する．その後，受信モードに切り替わり，サーバからの返信を EOD が送られてくるまで待機する．データ送信と EOD 送信部分は以下のコードである．

Code 18　データと EOD の送信 (client_send_recv_udp.c)

```
// 標準入力からデータを読み込み，バイトずつ送信 1000
while (sends < MAX_SENDS && (n = read(STDIN_FILENO, buf, CHUNK_SIZE)) > 0) {
    sendto(sock, buf, n, 0, (struct sockaddr *)&addr, sizeof(addr));
    sends++;
```

```
}

// 送信終了後，パケットを送信 EOD
memset(buf, 1, CHUNK_SIZE);
sendto(sock, buf, CHUNK_SIZE, 0, (struct sockaddr *)&addr, sizeof(addr));

// この後，サーバからのデータ受信ループに移行する
```

■実行結果　UDP のエコーサーバ（受信したデータをそのまま送り返すサーバ）を相手に実行した．クライアント側で入力したファイルの内容が，EOD を送信した後にサーバから送り返され，標準出力に正しく表示されることを確認した．

2.5.3　考察：UDP の信頼性の欠如と対策

課題で指摘されている通り，UDP には TCP のような到達保証・順序保証がないため，様々な問題が発生しうる．今回実装したプログラムが直面する可能性のある問題と，その対策について考察する．

■起こりうる問題とプログラムの挙動

1. **EOD パケットの欠損:** クライアントまたはサーバが送信した EOD パケットがネットワークの途中で失われた場合，受信側は通信の終わりを検知できない．その結果，`recvfrom()` でデータを待ち続け，プログラムが応答しなくなる．
2. **データパケットの欠損:** ファイルやメッセージの途中のパケットが失われた場合，その部分は受信されず，データが欠損する．今回のプログラムでは欠損を検知する仕組みがないため，壊れたデータがそのまま出力されてしまう．
3. **パケットの順序入れ替わり:** ネットワークの経路差により，後から送信したパケットが先に到着することがある．例えば，データパケットより先に EOD パケットが到着した場合，受信側はデータがまだ残っているにも関わらず通信を終了してしまい，データの一部しか得られない．

■考えられる対策　これらの問題に対処するためには，アプリケーション層で独自の信頼性メカニズムを実装する必要がある．

1. **タイムアウトの実装:**　EOD 欠損によるハングを防ぐため，`recvfrom()` にタイムアウトを設定する．`setsockopt()` でソケットオプション `SO_RCVTIMEO` を設定すれば，一定時間データが来ない場合に `recvfrom()` がエラーを返すようにできる．これにより，無限に待ち続ける事態を避けられる．
2. **シーケンス番号と ACK:** TCP のように，各パケットにシーケンス番号（通し番号）を付与する．受信側は，番号が飛んでいたらパケットロスを検知できる．また，受信したパケットの番号を ACK（確認応答）として送信者に返すことで，送信者はデータが確実に届いたかを確認できる．届いていない場合は再送する，という再送制御も実装できる．
3. **バッファリングと並べ替え:** 順序入れ替わりに対処するため，受信側で一度パケットをバッファに溜め，シーケンス番号を元に正しい順序に並べ替えてからアプリケーションに渡す．

もちろん，これらの対策をすべて実装すると TCP に近づいていき，UDP のシンプルさや低遅延という利点は損なわれていく．したがって，実際のアプリケーションでは，リアルタイム性を優先して多少のパケットロスは許容する（例: 音声通話）か，信頼性が必須な場合は素直に TCP を使うか，あるいは用途に応じて必要な信頼性機能だけを実装する，という設計上の判断が求められるだろう．

2.6　本課題 8.1：データ送信サーバの実装

■プログラムの説明　本課題で作成した `serv_send.c` は，指定したポート番号でクライアントからの接続を待ち受け，接続が確立したクライアントに対して一方的にデータを送信する機能を持つ TCP サーバである．プログラムの主な処理の流れは以下の通りである．

1. `socket()` で，接続待ち受け専用のソケット (`ss`) を生成する．
2. `bind()` で，待ち受けソケットに特定のポート番号と IP アドレス (`INADDR_ANY`) を割り当てる．これにより，どのネットワークインターフェースからの接続も受け付けることが可能となる．
3. `listen()` で，ソケットがクライアントからの接続を受け付け可能な状態であることを宣言する．

4. accept() で, 実際にクライアントからの接続要求が来るまで処理を停止 (ブロック) して待機する.
5. 接続が確立すると, accept() は通信専用の新しいソケット (s) を返す. その後, 標準入力から読み取ったデータを, この新しいソケットを通じてクライアントに送信 (write()) する.
6. 標準入力が EOF に達し, 全てのデータを送信し終えたら, 通信ソケット (s) と待ち受けソケット (ss) をそれぞれ close() してプログラムを終了する.

このサーバプログラムの中心部分である, 接続の待機と確立を行うコードは以下の通りである.

Code 19　接続の待機と確立 (serv_send.c)

```
// listen()で接続待ち受けを宣言した後...

// クライアントからの接続を受け入れる (accept)
struct sockaddr_in client_addr;
socklen_t len = sizeof(struct sockaddr_in);
int s = accept(ss, (struct sockaddr *)&client_addr, &len);
if (s < 0) {
    perror("accept");
    // ... エラー処理 ...
}

// 接続後、この新しいソケット 's' を用いてデータの送受信を行う
```

■実行結果と考察　動作確認のため, 2 つのターミナルを用意した. まずサーバ側で, 作成した serv_send をポート 50000 番で起動し, リダイレクションで send_file.txt を標準入力に指定した.

Code 20　サーバ側の起動コマンド

```
$ ./serv_send 50000 < send_file.txt
```

次にクライアント側で, 課題 6.1 で作成した client_recv を実行し, サーバへ接続した.

Code 21　クライアント側の実行コマンド

```
$ ./client_recv 127.0.0.1 50000 > received_file.txt
```

クライアントの実行後, サーバ側からファイルの内容が送信され, クライアント側で received_file.txt として保存された. diff コマンドで両ファイルを比較したところ差分はなく, md5sum の値も一致したことから, 自作のサーバ・クライアント間で TCP 通信が正しく行われたことが確認できた.

また, 課題文で指摘されている通り, rec と play コマンドを用いて片方向の音声通話を試みた.

- サーバ側: rec ... | ./serv_send 50000
- クライアント側: ./client_recv ... | play -

この構成で実行すると, サーバを起動してからクライアントが接続するまでの間にサーバ側で録音された古い音声が, クライアント側で再生された. つまり, 遅延が生じていた. これは, rec が serv_send の accept() による待機状態とは無関係に録音を開始し, パイプのバッファにデータを溜め込んでしまうためである. このバッファリング問題は, 次の課題で解決を目指す.

2.7　本課題 8.2 & 8.3：リアルタイム音声サーバと双方向電話への発展

本課題 8.1 で作成した serv_send.c では, パイプで渡された rec の音声データがバッファリングされ, クライアント側で古い音が再生されるという問題が確認された. 本節では, まずこの問題を解決し (本課題 8.2), 次にこれまでの知見を統合して双方向のインターネット電話を完成させる (本課題 8.3).

2.7.1　バッファリング問題の解決 (serv_send2.c)

■プログラムの説明　古い音声がバッファに溜まる問題の根源は, serv_send がクライアントの接続を待っている (accept() でブロックしている) 間にも, rec コマンドが録音を続けてしまうことにあった. この問題を解決するため,

serv_send2.c では, クライアントからの接続が確立した後に, はじめて録音プロセスを開始する手法を採用した. 具体的には, C の標準ライブラリ関数である popen() を用いている. popen() は, プログラム内から外部コマンドを実行し, その標準入出力をパイプとしてファイルポインタ経由で操作する機能を持つ. これにより, accept() が成功して通信専用ソケットが準備できたタイミングで rec を起動し, その出力を直接読み取ってクライアントに送信できるため, バッファに古いデータが溜まる問題が解消される. この解決策の中心部分は以下のコードである.

Code 22　popen による rec の遅延起動 (serv_send2.c)

```
// ... accept()で接続が確立した後 ...

// popen で rec コマンドを起動し, その標準出力をパイプで受け取る
FILE *rec_fp = popen("rec -q -t raw -b 16 -c 1 -e s -r 44100 -", "r");
if (!rec_fp) {
    perror("popen");
    // ... エラー処理 ...
}

// rec の出力 ( パイプ ) からデータを読み込み, ソケットへ書き出すループ
char buffer[BUFFER_SIZE];
size_t nread;
while ((nread = fread(buffer, 1, BUFFER_SIZE, rec_fp)) > 0) {
    if (write(s, buffer, nread) < 0) {
        perror("write");
        break;
    }
}
pclose(rec_fp); // rec プロセスを終了
```

■実行結果　serv_send2.c をサーバとして起動し, しばらく時間をおいてからクライアントで接続して音声を再生した. serv_send.c の時とは異なり, 古い音声が再生されることはなく, クライアントが接続した時点の「ライブ」な音声が聞こえることを確認した. これにより, popen() を用いた遅延起動がバッファリング問題の有効な解決策であることが示された.

2.7.2　双方向インターネット電話の実装 (i1i2i3_phone.c)

■プログラムの説明　本課題の最終目標として, これまでの技術を統合し, 2 台のマシン間で双方向の会話が可能なインターネット電話プログラム (i1i2i3_phone.c) を実装した. このプログラムは, 1 つのプログラムがクライアントとサーバの両方の役割を同時にこなす. すなわち, (1) 相手からの音声を指定のポートで待ち受けて再生しつつ, (2) こちらからの音声をマイクから拾って相手に送信する, という動作を並行して行う. この並行処理を実現するために, 以下の重要な技術を用いた.

- **UDP プロトコルの採用:** 会話のリアルタイム性を最優先するため, 遅延の少ない UDP プロトコルを選択した.
- **ノンブロッキング I/O と select():** そのままでは read() や recvfrom() で処理がブロックされてしまうため, 標準入力とソケットの両方をノンブロッキングモードに設定した. その上で select() システムコールを用いることで, 「マイクから入力があったか」「ネットワークからデータが届いたか」をブロックせずに監視し, データが準備できた方から処理を行う高度な I/O 多重化を実現した.

プログラムのメインループは以下のようになっており, select() で入力の発生を待機し, FD_ISSET() でどちらからの入力かを判別して処理を振り分けている.

Code 23　select() を用いた I/O 多重化 (i1i2i3_phone.c)

```
while (running) {
    FD_ZERO(&rfds);
    if (!stdin_eof) FD_SET(STDIN_FILENO, &rfds); // 標準入力を監視
    FD_SET(sockfd, &rfds);                       // ソケットを監視
```

```
        // タイムアウト付きで入力の発生を待つ
        int nready = select(maxfd + 1, &rfds, NULL, NULL, &tv);

        // ... キープアライブなどの処理 ...

        // 標準入力にデータがあれば読み込んでソケットへ送信
        if (!stdin_eof && FD_ISSET(STDIN_FILENO, &rfds)) {
            // ... read()と sendto()による送信処理 ...
        }

        // ソケットにデータがあれば読み込んで標準出力へ書き出し
        if (FD_ISSET(sockfd, &rfds)) {
            // ... recvfrom()と write()による受信処理 ...
        }
    }
```

■実行結果と結論　2 台のマシンで互いを宛先としてプログラムを起動し, `rec` と `play` をパイプで接続したところ, 双方向で遅延の少ない音声会話が可能となった. 単純なソケット API の学習から始まり, TCP と UDP の特性の理解, バッファリングのような実践的な問題の解決を経て, 最終的に `select()` を用いた高度な I/O 制御による双方向電話アプリケーションを完成させることができた. これにより, ネットワークプログラミングの基礎的な概念から応用までを一貫して学ぶことができた.

まとめ

本レポートでは, 第 5 日から第 8 日までの課題を通じ, インターネットの基本的な通信原理から, C 言語のソケット API を用いたクライアント・サーバアプリケーションの実装までを段階的に学んだ.

初めに, `nc` や `iperf` といった既存のコマンドを用いることで, TCP と UDP のプロトコルの特性差（信頼性と遅延のトレードオフ）や, ネットワーク帯域が共有資源であることを体験的に理解した. 次に, ソケット API に入り, まずクライアントプログラム (`client_recv.c`) を実装することで, ネットワーク通信の基本的なプログラミングの流れを習得した.

さらに, サーバ側の実装 (`serv_send.c`) では, `bind`, `listen`, `accept` というサーバ特有の API を学ぶとともに, `rec` コマンドとの連携で発生するバッファリングに起因した音声遅延という実践的な問題に直面した. この問題に対し, `popen()` を用いて外部プロセスを遅延実行させる手法 (`serv_send2.c`) で改善を図った.

最終的に, これら全ての知識と技術を統合し, 双方向のリアルタイム音声通話アプリケーションの基礎を完成させた. これにより, 単純なデータ転送から, リアルタイム性が要求される高度なアプリケーションの実装まで, ネットワークプログラミングにおける一連の重要な概念と技術を体系的に身につけることができた.

参考文献

[1] 株式会社アットマークテクノ. “6.2. ソケットプログラミング”. Armadillo-600 シリーズ ソフトウェアマニュアル. **https://manual.atmark-techno.com/armadillo-guide/armadillo-guide-2_ja-1.0.0/ch06.html**

[2] 筑波大学 情報学群 情報科学類. “システムプログラミング講義補助ページ”. **https://www.coins.tsukuba.ac.jp/~yas/coins/syspro-2005/**

[3] Google. 大規模言語モデル「Gemini」. 本レポートの構成相談, 文章校正, および LaTeX コード生成支援に利用.