情報通信応用実験　ネットワークプログラミングの基礎　参考資料

通常，プログラミング言語の処理系はプログラムとファイル上のデータのやり取りをおこなうためのインタフェースを備えている．例えば，C言語の場合，fprintf, fscanf 等の入出力関数がライブラリとして提供されている．では，プロセス間でのデータのやり取りはどのようにして行えばよいのであろうか．本実験では，Unix 上でネットワークを介したプロセス間通信を行うための仕組みについて実際に動作するCプログラムを作成しながら学んでいく．同時に，アプリケーションプロトコル及びパケットキャプチャの実験をとおして，TCP/IP プロトコルの階層構造についても理解する．

1. 基本的概念

1.1 クライアントーサーバモデル

ネットワークアプリケーションプログラムにおける標準的な計算モデルはクライアントーサーバ（Client-server）モデルである（図1）．クライアント，サーバというのは通信を構成する2つのアプリケーションプログラムを指す．通信を能動的に起動する側（サーバに対してサービスを要求する側）をクライアントと呼び，受動的に待つ側（クライアントからの要求に対してサービスを提供する側）をサーバと呼ぶ．ここで，クライアント，サーバという概念は本来プログラム（またはプロセス）に付随するものであることに注意する必要がある．よく”ワークステーション A はファイルサーバである”というような言い方をするが，これは”ワークステーション A でファイルサーバプログラムが実行されている”ことを指している．

クライアントーサーバモデルに基づいたアプリケーションプロセスの典型的な処理の流れは以下のようになる(図 2).

- ある計算機上でサーバプロセスが起動される．サーバプロセスはクライアントプロセスがサービスを要求してくるまで待ち状態になる．
- 同じ計算機上，またはネットワーク接続された別の計算機上でクライアントプロセスが起動される．起動されたクライアントプロセスはある種のサービスに対する要求をサーバプロセスに送信する．
- サーバプロセスはクライアントプロセスからのサービス要求を処理し，処理結果のデータをクライアント

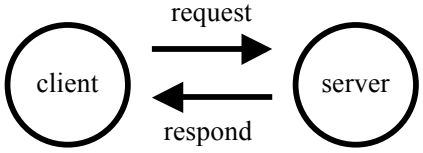

図1　クライアント-サーバモデル

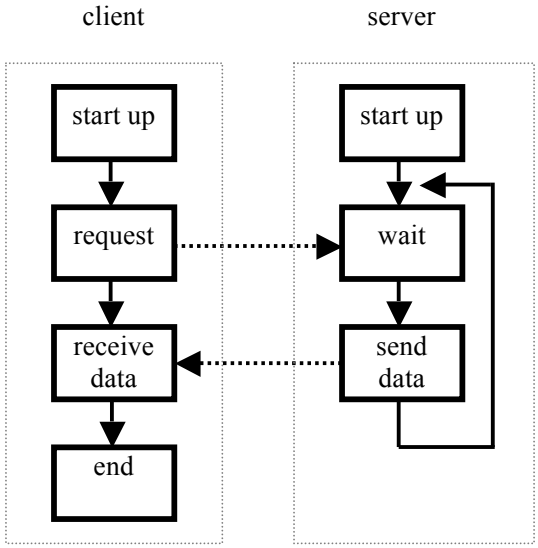

図2　各プロセスの処理の流れ

プロセスへ返送する.

- サービスの提供が終わると，サーバプロセスは再び待ち状態に入り，次のクライアントプロセスからの要求を待つ.

1.2 ネットワークの階層構造

ネットワークシステムは，通常階層的なモデルとして実装されている. 現在インターネットで用いられている TCP/IP の階層構造を図 3 に示す. このような階層構造をとることで，アプリケーションプログラマが実際のネットワーク構造や信頼性の保障などを気にせずに開発を行うことが可能である.

各階層は，プロトコルと呼ばれる通信の取り決めに従って，相手方の同じ階層と仮想的な通信を行う. 例えば，Web サーバと Web ブラウザ(クライアント)の場合，HTTP (Hyper Text Transfer Protocol)というプロトコルを用いて Web ページの転送を行っている. 実際には，各ホストはネットワークインタフェースを通して接続されているため，アプリケーション層で生成された情報は順次下の階層に渡され，ネットワークインタフェースから物理媒体に出力される. 受信側ではこの逆に，ネットワークインタフェース層から順次上に渡され，最終的に宛先アプリケーションに到達する.

各階層の役割と動作は以下のとおりである.

- トランスポート層 … 複数のアプリケーションプロセスを識別し，情報を目的プロセスに配送する. TCP を用いた場合，コネクション型のサービスを提供し，信頼性のある通信（情報の欠落がなく，送信した順序で受信することが保障される）が提供される. UDP はコネクションレスのサービスを提供するため，送信された情報が途中で損失する可能性がある.
- インターネット層 … 送信された情報を，ルータによって接続された複数のネットワークを経由して最終的に宛先ホストに到達させるための機能を持つ.
- ネットワークインタフェース層 … 同一ネットワーク内に存在するホストに情報を転送する機能を持つ.

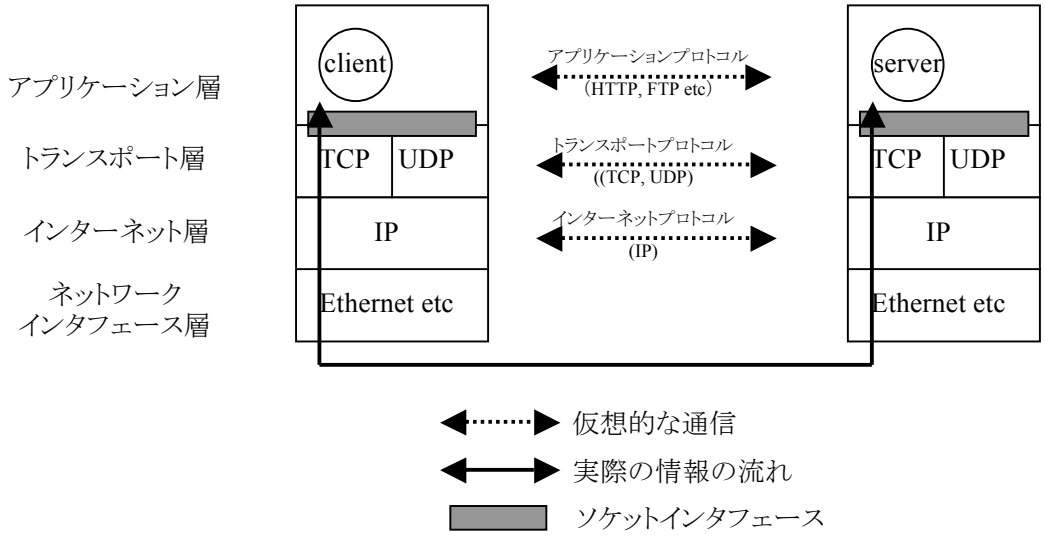

図 3 TCP/IP 階層構造

1.3 サーバの実装方式

クライアントからのサービス要求の仕方によりサーバの実装方式は以下の 2 種類に分類することができる.

- クライアントからのサービス要求をサーバプロセス自身が処理する反復サーバ（iterative server）
- クライアントからのサービス要求の処理を他のプロセスに任せてサーバプロセス自身は次のサービス要求の待ち状態に戻る並行サーバ（concurrent server）

さらに, 使用するトランスポート層プロトコル(TCP, UDP 等)がコネクション指向かコネクションレスかによっても分類することができる. サービス要求の処理方法と使用するトランスポート層プロトコルの組合せにより 4 通りのサーバの実装方式が存在することになる. あるサービスを実現するサーバの実装としてどれを選ぶかは, サーバがクライアントからのサービス要求の処理に要する時間やそのサービスが信頼性を必要とするかどうかに依存する.

2 ソケットによるプロセス間通信

2.1 ネットワークプログラミングインタフェース

プログラマがあるシステムの上でアプリケーションプログラムを開発するときに提供されるインタフェースを API (Application Programming Interface)という. Unix 系の OS では API はシステムコールと呼ばれており, プログラマは関数の形でそれらを利用できる. OS の基本的なサービスであるファイル入出力に関していえば, C 言語のプログラマに対して read や write といったシステムコールが提供されている. アプリケーションプログラムでファイル入出力が必要になったときに read や write を呼び出すことにより制御を OS に渡す. 実際のファイル入出力の制御は OS の内部手続きにより行われる. OS は要求された操作を終えた後, 制御をアプリケーションプログラムに戻す. プログラマの側では, 入出力に関する OS の動作やハードウェアの詳細について知る必要はなく, 通常の C 言語の関数と同じように定義されたシステムコールの外部仕様のみを覚えておけばよい. このようなシステムの階層化設計による情報隠蔽によりプログラマの苦労が軽減される.

ファイルに対する入出力の場合と同様に, ネットワークを介してプロセス間通信を行う場合も, アプリケーションプログラムからトランスポート層プロトコルを利用するための API がシステムコールとしてユーザに提供されている. これにより, アプリケーションプログラマは他の標準関数を利用するのと同様の書式でシステムコールを利用して TCP や UDP にアクセスできる. このとき TCP や UDP の詳細を知っている必要はないのでプログラミングが容易になる. プログラマは API を利用するときに必要になるサーバの IP アドレスや使用するトランスポート層プロトコルに関する情報だけを気にしていればよい.

一般に, どのような API が利用できるかは使用している OS とプログラミング言語に依存する. ネットワークアプリケーションを開発する際に使われる API として Unix 系の OS において一般的なのは, BSD のソケットと System V の TLI (Transport Layer Interface)である. 本実験ではソケットを用いてプロセス間通信を行うための方法について学ぶ.

2.2 通信を指定するためのパラメータ

ファイルへの入出力を行うプログラムは, データの読み書きをするためのシステムコールを利用するためにファイル記述子だけを知っていればよかった. しかし, ネットワークを介したプロセス間通信を行う場合には, 考慮しなければならないパラメータ数が格段に多くなる. 具体的には以下の 5 個が必要になる.

- 通信に使用するトランスポート層プロトコル （例:TCP, UDP. ）
- 送信元の計算機 (例:送信元の IP アドレス)

- 送信先の計算機 (例:送信先の IP アドレス)
- 送信元のプロセス (例:送信元のポート番号)
- 送信先のプロセス (例:送信先のポート番号)

プロトコル体系によりアドレス等の表現が異なるので, あるプロセス間通信を指定するには, まずそのプロセス間通信で使用されるトランスポート層プロトコルを指定する必要がある. ここでは, 使用するプロトコル体系として TCP/IP を仮定しているので, TCP と UDP のどちらかを指定することになる. TCP はコネクション指向のプロトコルであり, 実際の通信に先立ち接続を確立するための手続きが必要となる. 接続の確立後は, トランスポート層同士で受信確認応答と再送が自動的に行われ, 通信の信頼性(送信した情報(パケット)が送信した順序で相手に受信されること)が保証される. すなわち, 接続を確立した後は, ファイルシステムに対するのと同様なストリーム型のアクセスが可能となる. 一方, UDP はコネクションレスのプロトコルであり, 接続を確立せずにデータグラム(パケット)を送受信する. このため, 通信の信頼性は保証されない.

インターネットに接続されている全ての計算機に対して, 異なる IP アドレスが一意に割り当てられる. このため, 通信を構成している二つのプロセスが実行されている計算機を指定するには, それぞれの IP アドレスを指定すればよい. 現在インターネットで用いられている IP バージョン 4 (IPv4)では, IP アドレスは 32 ビットの数値となる. 視認性の向上のため, 各 8 ビットずつを 10 進数で表し, ピリオドで区切った形式(dotted-decimal)で記述することが可能である (例: 133.10.239.122). 32 ビットの IP アドレスのうち上位ビットは, ネットワークアドレスであり, 下位ビットがネットワーク内のホストアドレスである. ネットワークアドレスとホストアドレスの分離および階層管理によって, 効率的なルーティングが可能となる. 例えば, 首都大学東京日野キャンパスは, 133.10.0.0/16 というクラス B のネットワークアドレスを取得しており, 133.10. xx.xx という IP アドレスを持つ計算機は日野キャンパスにしか存在しない. ここで, "/16"はネットワークアドレスのビット数を表す書式である. さらに, 情報通信実験室は 133.10.235.0/24 というサブネットワークを構成しており, 室内の計算機には全て 133.10.235.xx という IP アドレスが割り当てられている.

最後に, 計算機内には複数のプロセスが同時に存在するため, 通信を構成するプロセスを指定する必要がある. TCP/IP の場合, ポート番号と呼ばれる 2 バイト整数で指定することになっている. 通信に先立って, 各プロセスは自分の利用したいポート番号を OS に申請し, 割り当てられる必要がある. ポート番号のうち, 0 から 1023 まではウェルノウンポート[5]であり, 既存の主要なアプリケーションサービス(http, ftp, telnet 等)のために確保されている. ウェルノウンポートを使用するためには, 管理者権限が必要となる. 1024 から 65535 までのポート番号は一般ユーザが使用可能であるが, このうち 1024 から 49151 までのポート番号は登録済みポート番号であり, 既存のサービスのために予約されている. 49151 から 65535 まではダイナミック/プライベートポートであり, ユーザが自由に使用できる. また, connect()システムコールによる動的なポート番号の割り当てもこの範囲から行われる. 後で紹介するシステムコール群を使う際には, 以上で説明したパラメータのうちいくつかを引数として与えることになる.

2.3 ソケットインタフェースの実装方式

2.2 節で説明したように, プロセス間通信を完全に指定するためには, 使用するプロトコル, 各プロセスを実行している計算機のアドレス, およびプロセスそれ自身(TCP/IP の場合ポート)を指定する必要がある. ソケットインタフェースの設計者は, 単一の関数ではなく複数の関数によってこれら 5 項目を指定するという方針を採用した. 例えば, コネクションレス通信を行うクライアントプログラムでは, 自身の使用する通信の端点を指定するのに必要となる情報(使用するプロトコル, アドレス, ポート番号)を2つの関数 socket(), bind()を呼び出すことにより決定する. ソケットインタフェースを設計する場合に問題になるもう一つの点は, 使用するプロトコルの扱いである. 以下の 2 つの選択肢が存在する.

- プロトコル毎に別の関数を用意する.
- 各プロトコルを汎用的に扱い, 引数として関数に与える.

前者の方針を採用した場合, 通信の端点を作る関数として socket()に相当する機能を持つ多数の関数をプロトコル毎に用意する必要があるが, ソケットインタフェースの設計者はこのような煩雑さを避けるため後者の方針を採用した. 現在, 我々が利用できる socket()の外部仕様は以下のようになっている.

```
int socket(int family, int type, int protocol);
```

family　プロトコルファミリ
type　サービス型
protocol　プロトコル番号(0 の場合, OS が適当なプロトコルを勝手に選択する.)

上で示したように, socket()の第一引数で使用するプロトコルファミリを指定する. 実際のプログラム中では PF というプレフィックスで始まる以下の定数を引数として与える.

PF_INET　Internet プロトコル

第二引数では通信の型を指定する. 第一引数の場合と同様, 以下の定数のうち1つを指定できる.

SOCK_STREAM　ストリーム型(コネクション指向)
SOCK_DGRAM　データグラム型(コネクションレス)

本実験では, TCP を使用する場合には SOCK_STREAM を, UDP を使用する場合には SOCK_DGRAM を指定する. 第三引数では使用するプロトコルに固有の整数を渡す. Unix 系の OS では, 各プロトコルに割り当てられた整数は /etc/protocols に記述されている. TCP/IP では, ストリーム型およびデータグラム型のプロトコルはそれぞれ1種類ずつしか存在しないため, 第三引数として 0 を与えることも可能である. socket()は呼び出しが成功するとソケット記述子と呼ばれる整数を返す. ソケット記述子はソケットを扱う他の関数でソケットを指定するために使用される.

ここまでで, 使用するプロトコルに関する指定が完了しているが, 通信の端点を指定するためには, さらにアドレスとプロセスの指定を行う必要がある. そのために bind()関数が用意されている.

```
int bind(int socket, struct sockaddr *addr, int addrlen);
```

socket　ソケット記述子.
addr　アドレスとプロセスを指定するための構造体.
addrlen　第二引数のバイト長.

bind()関数は第一引数で指定したソケット記述子に対応するソケットにローカルアドレスとプロセスを指定するための情報を割り当てる. これらの情報の受け渡しは, 以下のソケットアドレス構造体へのポインタという形で行われる.

```
struct sockaddr {
  u_short sa_family; /*アドレスファミリ(AF_xxx)*/
  char sa_data[14]; /*プロトコル特有のアドレス情報*/
};
```

sockaddr 構造体は，ヘッダファイル<sys/socket.h>内で定義されている．プロトコルファミリに対応したアドレスファミリは AF というプレフィックスで始まる定数で指定される．sa_family はアドレスファミリを指定するためのメンバであり，本実験では PF_INET に対応するアドレスファミリ(AF_INET)のみを使用する．

AF_INET　Internet プロトコルに対応するアドレスファミリ

sa_data にはアドレスやポートに関する情報が格納されている．sa_data の解釈の仕方は使用するアドレスファミリに依存する．AF_INET を使用する場合はヘッダファイル<netinet/in.h>で定義される以下の構造体が用いられる．

```
struct in_addr {
  u_long s_addr; /*32 ビットの IP アドレス*/
};

struct sockaddr_in {
  short               sin_family;   /*AF_INET */
  u_short             sin_port;     /*16 ビットのポート番号(ネットワークバイト順序)*/
  struct in_addr      sin_addr;     /*32 ビットの IP アドレス (ネットワークバイト順序)*/
  char                sin_zero[8];  /*未使用*/
};
```

よく知られているように，複数のバイトで構成されるデータのメモリへの格納の仕方はコンピュータシステムにより異なる．例えば，2 バイト整数の配置方法として，アドレスの昇順に低位バイト，高位バイトと並べる方法（リトルエンディアン）とその逆の方法（ビッグエンディアン）とがある．ネットワークプログラムでで規定されたパケットフォーマットで使用されるバイト順序（ネットワークバイト順序）は各コンピュータシステムでのバイト順序とは独立に規定されている．例えば，TCP/IP では 2 バイト整数，4 バイト整数ともにビッグエンディアンを採用しているので，上で定義した sockaddr_in 中のメンバ sin_port, sin_addr.s_addr はビッグエンディアンとして解釈される．ネットワークを介したプロセス間通信を行うプログラムでは，ローカルシステムで採用されているバイト順序とネットワークバイト順序が異なる場合にバイト順序を変換する関数を使用する必要がある．

bind()に限らず，アドレス情報を引数としてとるシステムコールへの情報の受け渡しは，全てアドレス構造体へのポインタ型の引数を介して行われる．Internet ファミリを使用するときには，sockaddr_in へのポインタを sockaddr へのポインタ型に型変換(キャスト)して情報を受け渡す．この様な実装になっているのは，アドレスファミリ毎に別の名前のシステムコールを用意することによって生じる煩雑さをなくすためである．

2.4 クライアントーサーバプログラムの構造

ここでは，クライアントーサーバモデルに基づいたプログラムの実際の構造について説明する．説明で使用されているシステムコールの使用の詳細については付録を参照せよ．

[コネクションレスの場合]

コネクションレスの通信を行う場合のクライアントプログラム，サーバプログラムの典型的な動作の流れは以下のとおりである．データを転送するのに相手のアドレスを指定する必要があるため，ソケットへの読み書きには sendto(), recvfrom()システムコールを用いる．

サーバ側の動作は以下のようになる．

1. socket()システムコールによりソケットを生成する．
2. bind()システムコールにより，ソケットにローカルな（自身を特定する）IP アドレスとポート番号を割り当てる．
3. recvfrom()システムコールにより，ソケットからバッファ内にデータを読み込む．データが到着していない場合は，新たにデータが到着するまでサーバプロセスをブロックする（ブロッキング通信）．
4. バッファ内のデータを処理する．並行サーバの場合は fork()システムコールを呼び出して処理を子プロセスに任せる．
5. sendto()システムコールにより，処理結果をソケットに書き込んでクライアントに送る

クライアント側の動作は以下のようになる．

1. socket()システムコールによりソケットを生成する．
2. bind()システムコールにより，ソケットにローカルな IP アドレスとポート番号を割り当てる．
3. sendto()システムコールにより，ソケットにデータを書き込みサーバに処理を依頼する．
4. recvfrom()システムコールにより，サーバの処理結果を受け取る

[コネクション指向の場合]

コネクション指向の通信を行う場合のサーバプログラムの典型的な動作の流れは以下のようになる．

1. socket()システムコールによりソケットを生成する．
2. bind()システムコールにより，ソケットにローカルな IP アドレスとポート番号を割り当てる．
3. listen()システムコールにより，クライアントからのコネクション要求に対する待ち行列を生成し，ソケットがコネクション要求を受け付け可能な状態にする．
4. accept()システムコールにより，クライアントからのコネクション要求を待ち行列から取り除きコネクションを確立する．待ち行列が空の場合は新たなコネクション要求が到着するまでサーバプロセスをブロックする．accept()システムコールの呼び出しが成功した場合，新たなソケット記述子が返される．以降，確立されたコネクションに対する通信はその新しいソケット記述子を用いて行う．
5. read()システムコールにより，ソケットからバッファ内にデータを読み込む．並行サーバの場合は，fork()システムコールを呼び出して子プロセスを生成し，子プロセスがデータの読み込みを行う．
6. バッファ内のデータを処理する．
7. write()システムコールにより，処理結果をソケットに書き込んでクライアントに送る．

クライアント側の動作はサーバの場合よりも簡単である．

1. socket()システムコールによりソケットを生成する.
2. connect()システムコールにより, サーバ側のソケットとの接続を要求する.
3. write()システムコールにより, ソケットにデータを書き込みサーバに処理を依頼する.
4. read()システムコールにより, サーバの処理結果を受け取る.

コネクション指向の場合, クライアントは bind()システムコールを呼び出す必要がない. これは, コネクションを確立する際に connect()システムコールがソケットアドレス(IP アドレス, ポート番号)の割り当てを自動的に行うためである. このとき割り当てられるポート番号は 49152 から 65535 までの適当な値が選択される.

3. サンプルプログラム

DAYTIME サービスを実現するサーバのサンプルプログラムを以下に示す. 本プログラムはコネクション指向通信を用いており, 実装方式は反復型である. また, 本来 DAYTIME サービスは well-known service として定義されているが, ルート権限を持たないユーザが実行可能であるよう, /etc/services に記されている値(一般的に 13)に 50000 を加えた値をローカルのポート番号としている. なお, 網掛はコメントアウトされた部分であることを示している.

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <sys/types.h>
 4:  #include <netdb.h>
 5:  #include <sys/socket.h>
 6:  #include <netinet/in.h>
 7:  #include <string.h>
 8:  #include <time.h>
 9:
10:
11:  int main(int argc, char *argv[])
12:  {
13:    struct servent     *servp;
14:    struct protoent    *protop;
15:
16:    char *service = "daytime";
17:    char *protocol = "tcp";
18:
19:    int family = PF_INET;
20:    int type = SOCK_STREAM;
21:
22:    struct sockaddr_in sin, fsin;
23:
24:    int msd, ssd;
25:    int alen;
26:    int portbase = 50000;
27:    int rval;
28:
29:
30:  /* --- not used by daytime server
31:    char buf[1024];
32:    int n;
33:  --- */
34:
35:    char   *timep;
36:    time_t now;
37:
38:
39:  /* set up the parameters for TCP server */
```

```
 40:    sin.sin_family = AF_INET;
 41:    sin.sin_addr.s_addr = INADDR_ANY;
 42:
 43:  /* try to resove port number from service name */
 44:    servp = getservbyname(service, protocol);
 45:    if (servp != NULL)
 46:      {
 47:        sin.sin_port = htons(ntohs((u_short) servp->s_port) + portbase);
 48:      }
 49:    else
 50:      {
 51:        fprintf(stderr, "Unknown service¥n");
 52:        exit(EXIT_FAILURE);
 53:      }
 54:
 55:  /* try to resove protocol number of TCP from protocol name */
 56:    protop = getprotobyname(protocol);
 57:    if (protop == NULL)
 58:     {
 59:        fprintf(stderr, "Unknown protocol¥n");
 60:        exit(EXIT_FAILURE);
 61:      }
 62:
 63:  /* generate socket */
 64:    msd = socket(family, type, protop->p_proto);
 65:    if (msd < 0)
 66:      {
 67:        fprintf(stderr, "Can't create socket¥n");
 68:        perror("");
 69:        exit(EXIT_FAILURE);
 70:      }
 71:
 72:  /* bind socket to local address */
 73:    rval = bind(msd, (struct sockaddr *) &sin, sizeof(sin));
 74:    if ( rval < 0)
 75:      {
 76:        fprintf(stderr, "Can't bind¥n");
 77:        perror("");
 78:        exit(EXIT_FAILURE);
 79:      }
 80:
 81:  /* listen to the socket */
 82:    rval = listen(msd, 5);
 83:    if ( rval < 0)
 84:      {
 85:        fprintf(stderr, "Can't listen¥n");
 86:        perror("");
 87:        exit(EXIT_FAILURE);
 88:      }
 89:
 90:  /* infinite loop  */
 91:    while (1)
 92:      {
 93:  /* if connection is accepted,
 94:           new socket is generated to communicate to client */
 95:
 96:        alen = sizeof(fsin);
 97:        ssd = accept(msd, (struct sockaddr *) &fsin, &alen);
 98:        if (ssd < 0)
 99:     {
100:       fprintf(stderr, "Can't accept¥n");
101:       perror("");
102:       exit(EXIT_FAILURE);
103:     }
104:
105:
106:  /* --- daytime server does not receive data from client,
107:              and transmits data only once.
```

```
108:
109:       while(1) {
110:       n = read(ssd, buf, sizeof(buf));
111:      if (n == 0)
112:             {
113:       printf("connection closed.¥n");
114:       break; // break from intrenal loop
115:     }
116:        else if (n < 0)
117:     {
118:       fprintf(stderr,"Can't read¥n");
119:       perror("");
120:       exit(EXIT_FAILURE);
121:     }
122:  ---   */
123:
124:  /* obtain current time, covert to string, and send to client */
125:        time(&now);
126:        timep = ctime(&now);
127:
128:        rval = write(ssd, timep, strlen(timep));
129:        if ( rval < 0)
130:    {
131:       fprintf(stderr, "Can't Write¥n");
132:       perror("");
133:       exit(EXIT_FAILURE);
134:     }
135:
136:  /* --- daytime server does not repeat transmission
137:     }
138:  --- */
139:
140:  /* close socket and go back to accept */
141:        close(ssd);
142:      }
143:
144:     return 0;
145:  }
```

参考文献

[1] B. W. Kernighan, D. M. Ritchie (石田晴久訳), プログラミング言語 C 第 2 版 ANSI 規格準拠, 共立出版, 1989 年.
[2] W. R. Stevens（篠田陽一訳）, UNIX ネットワークプログラミング, トッパン, 1992 年.
[3] C. Comer（村井純, 楠本博之訳）, TCP/IP によるネットワーク構築 Vol. Ⅰ -原理・プロトコル・アーキテクチャ- 第 3 版, 共立出版, 1997 年.
[4] C. Comer（村井純, 楠本博之訳）, TCP/IP によるネットワーク構築 Vol. Ⅲ -クライアント-サーバプログラミングとアプリケーション- BSD ソケットバージョン, 共立出版, 1996 年.
[5] http://www.key.ne.jp/RFC/Assigned_Numbers/well-known.html.
[6] マルチメディア通信研究会, 通信プロトコル事典, アスキー, 1996 年.

付録 A: ソケットシステムコール群

プログラム作成に必要となるシステムコール, ライブラリ関数, 構造体について解説する.

[socket システムコール]

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int family, int type, int protocol);
```

family プロトコルファミリ(PF_INET 等).
type サービス型(SOCK_STREAM, SOCK_DGRAM 等).
protocol プロトコル番号. 0 の場合, OS が適当なプロトコルを勝手に選択する.

通信で使用されるソケットを生成し, ソケット記述子を返す. 呼び出しが成功した場合ソケット記述子(0 以上)を, エラーが起きた場合-1 を返す.

[bind システムコール]

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int socket, struct sockaddr *addr, int addrlen);
```

socket　ソケット記述子.
addr　IP アドレスとポート番号が格納された sockaddr 型の構造体へのポインタ.
addrlen　第 2 引数のバイト長.

ソケットに対してローカル IP アドレスとポート番号を指定する. 呼び出しが成功した場合 0 を, エラーが起きた場合-1 を返す. IP アドレスとして INADDR_ANY を指定した場合, そのホストの全ての IP アドレスに自動的に割り当てられる.

[listen システムコール]

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int socket, int queuelen);
```

socket　ソケット記述子.
queuelen　コネクション要求の待ち行列の長さ(通常, 5 以下の値を指定する).

サーバにより呼び出され, サーバ側のソケットがクライアントからの要求を受付可能な状態にする. また, あるソケットに対するコネクション要求の待ち行列の長さも指定する. 呼び出しが成功した場合 0 を, エラーが起きた場合-1 を返す.

[accept システムコール]

```
#include <sys/types.h>
#include <sys/socket.h>

int accpet(int socket, struct sockaddr *addr, int *addrlen);
```

socket　ソケット記述子.
addr　クライアントのアドレス.
addrlen　第 2 引数のバイト長.

TCP を使用している場合に用いられる. サーバにより呼び出され、クライアントからのコネクション要求を待ち行列から取り除く. 待ち行列が空の場合は次の要求が到着するまで待つ. その後, その要求に対して新しいソケットを生成し, そのソケットに対するソケット記述子を返す. また, クライアントのアドレス情報が, 引数で指定された領域に格納される. ポインタ addrlen で示されたアドレスには, addr に書き込み可能なバイト長を格納しておく必要がある. エラーが起きた場合-1 を返す.

[connect システムコール]

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int socket, struct sockaddr *addr, int addrlen);
```

socket　ソケット記述子.
addr　サーバの IP アドレスやポート番号を格納.
addrlen　第 2 引数のバイト長.

クライアントプロセスにより呼び出され, (TCP を使用している場合)サーバとの接続を確率する. UDP を用いている場合は、単に connect 呼び出し以降のサーバのアドレス指定の省略が可能になるだけである. 呼び出しが成功した場合 0 を, エラーが起きた場合-1 を返す.

[read システムコール]

```
int read(int socket, char *buf, int buflen);
```

socket　ソケット記述子.
buf　入力データを読み込むためのバッファ.
buflen　バッファ長.

ソケットからデータを読み込む. 入力を得ると読み込んだバイト数を返し, ソケットのファイル終了状態を検出すると 0 を返す. エラーが起きた場合-1 を返す.

[write システムコール]

```
int write(int socket, char *buf, int buflen);
```

socket　ソケット記述子.
buf　出力データが入っているバッファ.
buflen　バッファ長.
ソケットにデータを書き込む. 出力に成功すると書き込んだバイト数を返し, エラーが起きた場合-1 を返す.

[sendto システムコール]

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int socket, char *msg, int msglen, int flags,
           struct sockaddr *to, int tolen);
```

socket　ソケット記述子.
msg　メッセージへのポインタ.
msglen　メッセージのバイト長.
flags　制御ビット (通常は 0 とすれば良い)
to　送信先アドレスへのポインタ.
tolen　送信先アドレスのバイト長.
指定された受信者に対してメッセージ送信を行う. 送信先アドレスを引数で指定する. 呼び出しが成功した場合送られたバイト数を, エラーが起きた場合-1 を返す.

[recvfrom システムコール]

```
#include <sys/types.h>
#include <sys/socket.h>

int recvfrom(int socket, char *buf, int buflen, int flags,
             struct sockaddr *from, int *fromlen);
```

socket　ソケット記述子.
buf　メッセージを保持するバッファへのポインタ.
msglen　バッファ長.
flags　制御ビット (通常は 0 とすれば良い)
from　送信元アドレスへのポインタ.
fromlen　送信元アドレスのバイト長へのポインタ.
ソケットに到着したメッセージを得る. また, 送信元アドレスの情報が, 引数で指定された領域に格納される. 呼び出しが成功した場合受け取ったバイト数を, エラーが起きた場合-1 を返す. ポインタ fromlen で示されたアドレスには, 送信元アドレス(from)に書き込み可能なバイト長を格納しておく必要がある. 呼び出し終了後, from には送信元アドレスの情報が, *fromlen には送信元アドレスのバイト長が格納される.

[close システムコール]

```
int close(int socket);
```

socket　ソケット記述子.
通信を終了させ, ソケットを取り除く. 呼び出しが成功した場合1を, エラーが起きた場合-1 を返す.

[fork システムコール]

```
int fork();
```

プロセスの複製を作る. 並行サーバがクライアントからの要求を処理するための並行プロセスを生成するのに使われる. 呼び出しが成功した場合, 子プロセスに対しては 0 を返し, 親プロセスに対しては子プロセスのプロセス識別子を返す. エラーが起きた場合 -1 を返す.

[getsockname, getpeername システムコール]

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockname(int socket, struct sockaddr *addr, int *addrlen);
int getpeername(int socket, struct sockaddr *addr, int *addrlen);
```

socket　ソケット記述子.
addr　ソケットのローカルまたはリモートの IP アドレスなど.

addrlen　　第 2 引数のバイト長.

これら 2 種類の関数は、ソケットのローカル(getsockname)またはリモート(getpeername)の IP アドレスやポート番号などを返す. 呼び出しが成功した場合 0 を, エラーが起きた場合-1 を返す.

[バイト順序変換ルーチン]

```
#include <sys/types.h>
#include <netinet/in.h>

u_long htonl(ulong hostlong);
U_short htons(u_short hostshort);
u_long ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

hostlong　ホストで扱われるデータ(unsigned long).
hostshort　ホストで扱われるデータ(unsigned short).
netlong　ネットワークで扱われるデータ(unsigned long).
netshort　ネットワークで扱われるデータ(unsigned short).

これらの関数は, 異なったアーキテクチャ, 異なったプロトコル間でのバイト順序の相違を吸収するために使用する. ホストのバイト順序からネットワークのバイト順序へ(hton), またはその逆(ntoh)へ, 長い(l)または短い(s)整数を変換する.

[アドレス変換ルーチン]

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned_long inet_addr(char *ptr);
char *inet_ntoa(struct in_addr inaddr);
```

ptr　dotted-decimal 形式の文字列.
inaddr　4 バイトの Internet アドレス.

inet_addr は dotted-decimal 形式の IP アドレス("133.86.20.1"など)を 4 バイトの Internet アドレスに変換する. inet_ntoa はその逆の操作を行う.

[gethostbyname ライブラリ関数]

```
#include <netdb.h>

struct hostent *gethostbyname(char *name);
```

name　ホスト名

ホスト名からそのホストに割り当てられた IP アドレス等の情報を含む構造体を獲得する. 呼び出しが成功した場合 hostent 構造体へのポインタを返し, エラーが起きた場合 NULL を返す. hostent 構造体は, netdb.h の中で下記のとおり定義されている.

```
struct hostent {
  char *h_name;              /*ホストの正式名*/
  char **h_aliases;          /*別名のリスト */
  int  h_addr_types;/*ホストアドレスのタイプ*/
  int  h_length;             /*アドレス長*/
  char **h_addr_list;        /*ネームサーバからのアドレスのリスト*/
};
```

[getservbyname ライブラリ関数]

```
#include <netdb.h>

struct servent *getservbyname(char *name, char *proto);
```

name　サービス名.
proto　プロトコル名.

サービス名からそのサービスに割り当てられたポート番号等の情報を含む構造体を獲得する. 呼び出しが成功した場合 servent 構造体へのポインタを返し, エラーが起きた場合 NULL を返す. servent 構造体は, 下記のとおり定義されている.

```
struct servent {
  char  *s_name;             /*サービスの正式名*/
```

```
  char **s_aliases;        /*別名のリスト */
  int  s_port;             /*ポート番号*/
  char *s_proto;           /*プロトコル名*/
};
```

[getprotobyname ライブラリ関数]

```
#include <netdb.h>

struct protoent *getprotobyname(char *name);
```

name　　プロトコル名.

プロトコル名からそのプロトコルに割り当てられたプロトコル番号等の情報を含む構造体を獲得する. 呼び出しが成功した場合 protoent 構造体へのポインタを返し, エラーが起きた場合 NULL を返す. protoent 構造体は下記のとおり定義されている.

```
struct protoent {
  char  *p_name;           /*プロトコルの正式名*/
  char **p_aliases;        /*別名のリスト */
  int  p_proto;            /*プロトコル番号*/
};
```

[exit ライブラリ関数]

```
#include <stdlib.h>

void exit(int status);
```

status　　終了ステータス

プロセスを正常に終了させ, status を親プロセスへ返す. status として, EXIT_SUCCESS (正常終了)と EXIT_FAILURE(異常終了)が定義されている.

[perror ライブラリ関数]

```
#include <stdlib.h>

void perror(const char *str);
```

str　　表示文字列

直前のエラーに対応するエラーメッセージを生成し, 指定された文字列とともに標準エラー出力に出力する.

[time システムコール]

```
#include <time.h>

time_t time(time_t *timer);
```

timer　　時刻格納場所へのポインタ

現在時刻を取得し, 指定されたアドレスに格納する. 呼び出しが成功した場合には現在時刻を返し, エラーが起きた場合には-1 を返す. 指定されたアドレスに格納される値は返戻値と同一である. なお, 現在時刻は UNIX 時刻, すなわち 1970/01/01 00:00:00(UTC)からの経過秒数で表わされる.

[ctime ライブラリ関数]

```
#include <time.h>

char *ctime(const time_t *timer);
```

timer　　時刻格納場所へのポインタ

timer に格納された UNIX 時刻を, ホストに設定された地域の現地時刻として"曜日　月　日　時:分:秒　年"の形式の文字列に変換し, 文字列へのポインタを返す.