ネットワーク系演習 II 「ネットワークプログラミング」

1 目的

計算機間でのデータ通信を実現するためのインタフェースであるソケットインタフェースを取り上げ，ネットワークを介したプロセス間通信のインプリメント技法を学習する．また，インターネットにおける WWW (World Wide Web)，電子メール，ファイル転送等のサービスで使用されているプロトコルの動作原理ならびに実現手法を学習する．

2 解説

本実験では，UNIX 上で，TCP (コネクション型) 及び UDP (コネクションレス型) を用いて，ネットワークを介したプロセス間通信プログラムを作成する．プロセス間通信は，標準的なモデルであるクライアント・サーバモデルを用いて実現される．

2.1 クライアント・サーバモデル

クライアントとは，ユーザ側のプロセスであり，ユーザの指示によって，サーバに対してサービス要求を発行する能動的なプロセスである．サーバは，クライアントからの接続を待ち，クライアントからのサービス要求に従ってサービスを実行し，その結果をクライアントに返送する受動的プロセスである (図 1 参照)．

サーバの動作形態には反復サーバと並行サーバがある．反復サーバはクライアントのサービス要求に必要な処理時間が短いと分かっている場合に用いられ，サービスはサーバ自身によって処理される．並行サーバは，クライアントのサービス要求に必要な処理時間が不明である場合や許容される処理待ち時間が小さい場合に用いられ，サーバが別プロセスを起動することによりサービスを行う．図 1 は，反復サーバの例である．

2.2 プロセス制御

並行サーバ実現のためには，サーバが，サービスを行うための別プロセスを起動する必要がある．以下では，UNIX にてプロセス制御を実現するシステムコールを紹介する．

(1) fork() システムコール　UNIX で新しいプロセスを生成するには，fork() システムコールを使用する．

```
int fork();
```

fork システムコールは，それを呼び出したプロセスのコピーを作成する．fork システムコールを呼び出したプロセスのことを親プロセス（parent process）と呼び，新しく作成されたプロセスを子プロセス（child process）と呼ぶ．fork システムコールは親プロセスによって 1 回呼び出されると，親プロセスと子プロセスに対してそれぞれ 1 回ずつ，つまり合計 2 回，その実行から戻る．fork システムコールからの 2 回の戻りが異なっているのは，親プロセスには新しく作成された子プロセスのプロセス ID が返されるのに対して，子プロセスには 0 が返されるという点で

ある．fork システムコールは，失敗すると −1 を返す．もし，子プロセスがその親プロセスのプロセス ID を知りたいときには，getppid システムコールを呼び出せばよい．

fork システムコールの重要な機能の一つに，その呼び出し以前に親プロセスでオープンされていたファイルが，fork 後に子プロセスと共有されるというものがある．これにより，親プロセスで特定のファイルやデバイスをオープンしておき，子プロセスにこれらを渡して処理させるということが簡単に行える．

以下に fork システムコールの利用例を示す．

```
main()
{
    int childpid;

    if ((childpid = fork()) == -1) { /* forkによりプロセスを複製 */
       /* プロセスの複製に失敗 */
       perror("can't fork");
       exit(1);
    } else if (childpid == 0) { /* 親プロセスか子プロセスかを判別 */
       /* 子プロセス */
       printf("child: child pid = %d, parent pid = %d\n",
              getpid(), getppid());
       exit(0);
    } else {
       /* 親プロセス */
       printf("parent: child pid = %d, parent pid = %d\n",
              childpid, getpid());
       exit(0);
    }
}
```

この使用例では，プロセスを複製し，それぞれのプロセスが，自身のプロセス ID と他方のプロセス ID を表示する．

(2) exit() システムコール　プロセスは，exit システムコールを呼び出すことによって終了する．このシステムコールは，呼び出し元に帰らない．exit が呼ばれると，整数の終了ステータスがカーネルに渡され，終了したプロセスの親プロセスから wait() システムコール（後述）によって参照できるようになる．慣例により，正常終了したプロセスは終了ステータスとして 0 を返し，0 でない値は何らかのエラー状態を示す．

(3) exec() システムコール　UNIX においてプログラムを実行する唯一の方法は，すでに存在するプロセスによって exec() システムコールを発行する方法である．exec() システムコールは，それを呼び出したプロセスを新しいプログラムと置き換える．このとき，プロセス ID は変化しない．ここでは，exec() システムコールを呼び出したプログラムのことを呼び出しプロセス，新しく実行されるプログラムのことを新しいプログラムと呼ぶことにする．exec() システムコールでは，新しいプロセスは生成されないことに注意せよ．

exec() システムコールには六つの異なった版があるが，ここではそのうち二つのみを紹介する．

```
int execlp(char *filename, char *arg0, char *arg1, ..., char *argn, (char *) 0);
int execvp(char *filename, char **argv);
```

これら二つの違いは，新しいプログラムに対する引数の与え方にある．execlp() では，引数を一つずつ与える．一方，execvp() では，引数の入った配列をポインタとして引き渡す．

(4) wait() システムコール　あるプロセスは，その子プロセスのいずれかの終了を wait システムコールを用いて待つことができる．

```
int wait(int *status);
```

wait によって返される値は，終了した子プロセスのプロセス ID である．もし，呼び出したプロセスが子プロセスを持たないときには，wait はすぐさま −1 を返す．呼び出したプロセスが，終了していない子プロセスを一つ以上待つならば，そのプロセスは，カーネルによって子プロセスのいずれかが終了するまで待ち状態におかれる．子プロセスが終了して，wait からの制御が戻されるとき，status が NULL でないならば，status に終了したプロセスの終了ステータスが格納され，これによって，付加的な情報も返すことができる．

2.3　TCP/IP のプロトコル体系

TCP/IP ネットワークにおけるアプリケーションとプロトコルの関係を図 2 に示す．クライアントとサーバのプロセスはアプリケーション層に位置する．そして，これらは，API (Application Program Interface) を介して，トランスポート層の TCP 及び UDP を利用する．なお，詳細については，文献 [1] を参照されたい．

2.3.1　TCP

TCP は，信頼性が高いコネクション指向のストリーム型トランスポート層プロトコルである．通信の信頼性を保証するため，データのビット誤り，消失，重複，到着順序の誤り等を検出するための機能を持つ．但し，コネクション確立・切断のオーバヘッドがあるため，短いメッセージを送信する場合には効率が悪くなる．しかし，アプリケーションで通信の信頼性を保証する必要がないため，一対一通信では TCP が用いられることが多い．

コネクション制御

TCP では，通信の信頼性を保証するためにコネクションを確立する．図 3 に，コネクション確立の手順を示す．SYN はコネクションの初期化セグメント (TCP におけるパケットはセグメントと呼ばれる) を表し，ACK は送達確認セグメントを表している．クライアントは，SYN をサーバに送信することにより，順序番号の初期値や最大セグメントサイズなどを通知する．サーバは，クライアントの SYN に対する ACK を合い乗せして，順序番号の初期値等を通知する SYN を送信する．また，クライアントは，サーバの SYN に対する ACK を送信する．この手続は，三つのセグメントを必要とするため，3-Way-Handshake と呼ばれる．

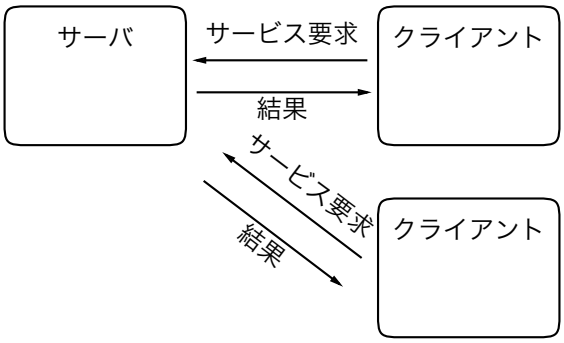

図 1: クライアント・サーバモデル

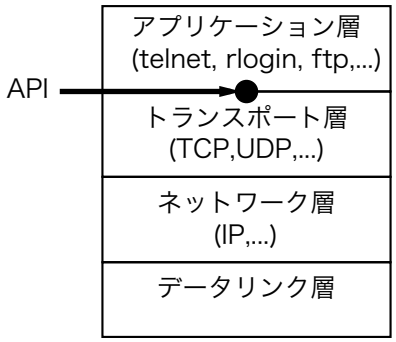

図 2: プロトコル体系

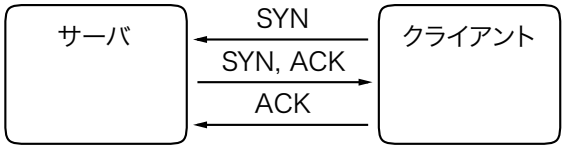

図 3: 3-Way-Handshake

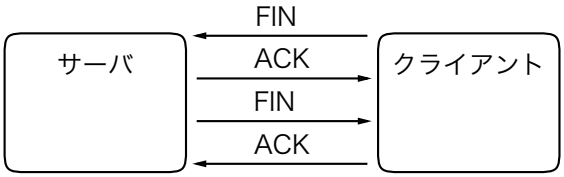

図 4: 緩やかな切断

TCP のコネクションは，全二重であり，クライアント及びサーバのどちらの側からも独立に切断できる．図 4 に，クライアントからコネクションを切断する場合を示す．クライアントは，FIN(コネクションの終了セグメント) をサーバに送信することにより，コネクションの切断を通知する．サーバは，クライアントの FIN に対する ACK を送信する．また，サーバは，FIN をクライアントに送信し，クライアントはサーバの FIN に対する ACK を送信する．この手続は，徐々にコネクションを切断していくことから，緩やかな切断という．

順序制御

順序制御は，オクテット単位の順序番号 (Sequence Number) を用いて行われる．TCP の送信用モジュール (以下，送信 TCP と略す) は，セグメント内の第 1 オクテットの順序番号をプロトコルヘッダに設定する．TCP の受信用モジュール (受信 TCP と呼ぶ) は，プロトコルヘッダの順序番号からデータの重複や到着順序の誤りを検出する．そして，重複したデータを破棄し，次に受信すべきデータの順序番号よりも大きい順序番号を持つデータが先に到着すると，そのデータをコネクション毎の待ち行列に置き，それより小さい順序番号を持つデータの到着を待つ．

誤り検出 (チェックサム)

送信 TCP は，受信 TCP がセグメントのビット誤りを検出できるようにチェックサムをプロトコルヘッダに設定する．チェックサムは，セグメント (厳密には，擬似ヘッダ [1] が付加される) の 16 ビット整数列に対して 1 の補数の和を計算し，その結果の 1 の補数を取ることによって算出される．受信 TCP は，ビット誤りが検出されたセグメントを廃棄する．

送達確認

受信 TCP は，次に受信すべきデータの順序番号を ACK 番号 (Acknowledgment Number) として送信側に通知する機構を持つ．送信 TCP は，ACK 番号を受け取ることで，それより小さな順序番号を持つデータが正しく受信されたことを知る (これにより，再送のために保持していたバッファを解放する).

2.4 UDP

UDP は信頼性のないコネクションレス型トランスポート層プロトコルである．チェックサムを計算することは可能であるが，データの消失や到着順序の誤り等を検出するための機能は持たない．このため，UDP は，短いメッセージを送信する場合に用いられたり，通信相手を特定する必要がないため，一対多通信で用いられることが多い．

2.5 API

API(Application Program Interface) は，プログラマが利用できるインタフェースのことである．どの API が利用できるかは，使用するオペレーティングシステムに依存する．UNIX にとって最も一般的な API は BSD ソケットである．Windows においても，BSD ソケットと同様な API である WinSock が用意されている．本実験では BSD ソケットを用いる．ソケットを用いたプログラミング技法の詳細については，文献 [2] を参照されたい．

2.5.1 ソケットインタフェース

ソケットは，データを送信及び受信するための通信端点である．ファイルがファイル記述子で操作されるのと同様に，ソケットはソケット記述子を用いて操作される．図 5 に，ソケットを用いたプロセス間通信の例を示す．プロセス間通信を行うためには，通信相手を特定する必要がある．このために，アドレス，ポート番号及びプロトコルが用いられる．アドレスは端末を特定し，ポート番号は端末内のプロセスを特定する．また，ポート番号がプロトコル毎に独立に割り当てられるため，プロトコルも指定する必要がある．

(1) アソシエーション (association)　アソシエーションは，通信するプロセスを完全に指定することのできる五つの組

```
{protocol, local-addr, local-process, foreign-addr, foreign-process}
```

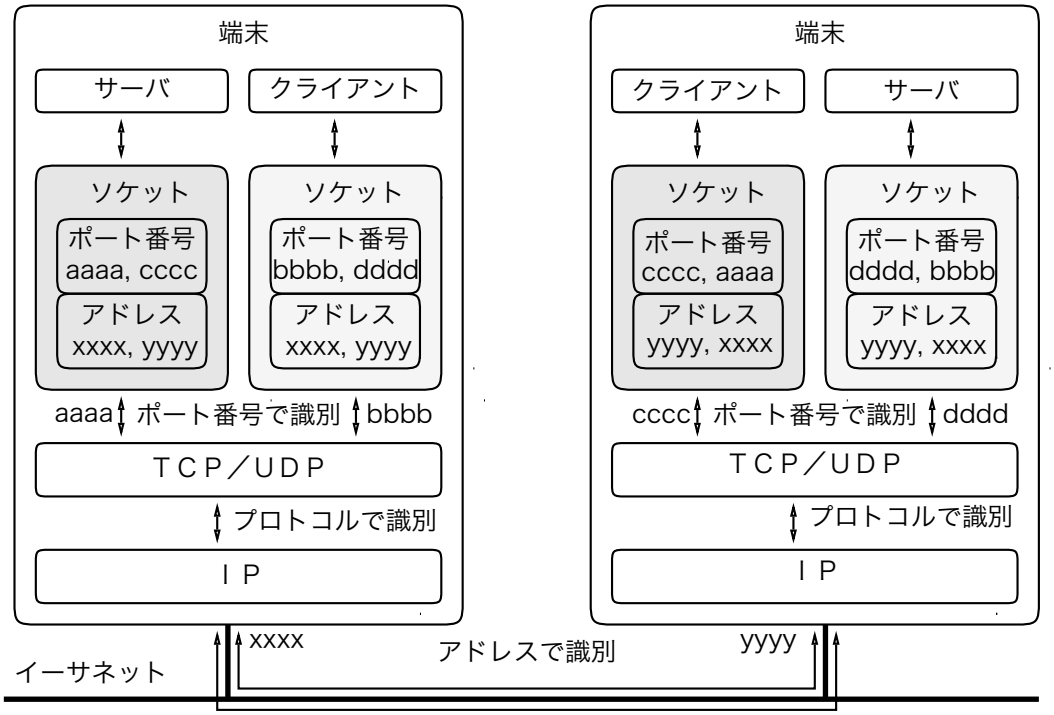

図 5: ソケットを用いたプロセス間通信の例

を示す．通信を行うためには，まずアソシエーションを指定する．その後，プロトコルモジュールにデータの送受信を要求する．これらは，2.5.3 節で示すソケットシステムコールを用いて行う．

protocol は，TCP や UDP 等のプロトコルを示す．また，local-addr と local-process は自アドレスと自ポート番号，foreign-addr と foreign-process は相手アドレスと相手ポート番号を表す．

(2) ソケットアドレス構造体　protocol 以外のアソシエーションの要素は，ソケットアドレス構造体を用いて設定する．ソケットアドレス構造体 (struct sockaddr) は，アドレスファミリ及び local-addr, local-process(または foreing-addr, foreign-process) をメンバに持つ．アドレス体系が複数存在するので，ソケットアドレス構造体はアドレス体系ごとに用意されている．アドレスファミリは，アドレス体系を特定するためのものである．

アドレスファミリがインターネット (AF_INET) の場合，ソケットアドレス構造体は，以下の struct sockaddr_in で示される (図 6)．なお，struct sockaddr_in はヘッダファイル <netinet/in.h> で定義されている (ヘッダファイルは/usr/include にある)．

```
struct in_addr {
        u_long  s_addr;
};

struct sockaddr_in {
        short           sin_family;     /* AF_INET */
        u_short         sin_port;       /* process, 16bit port number (ネットワークバイト順序) */
```

```
        struct inaddr   sin_addr;       /* addr, 32bit Internet address (ネット
ワークバイト順序) */
        char            sin_zero[8];    /* 未使用 */
};
```

この表記法については，文献 [3] を参照されたい．

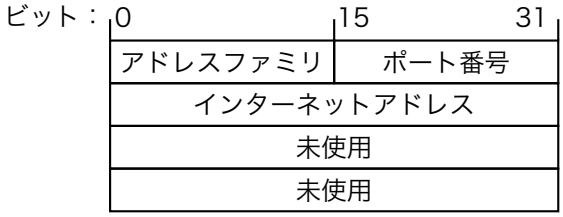

図 6: ソケットアドレス構造体

2.5.2 プロセス間通信

図 7 は，TCP を用いてプロセス間通信を行う場合のシステムコールの流れを表している．TCP のサーバは，socket() を用いて protocol を設定する．また，bind() で local-addr, local-process を設定し，accept() で foreign-addr, foreign-process を確認する．一方，TCP のクライアントは，socket() によって protocol を設定し，connect() で local-addr, local-process, foreign-addr, foreign-process を設定する．

図 8 は，UDP の場合を示す (プログラム例を付録に示す)．UDP においても，TCP の場合と同様に，protocol の設定には socket() を用いる．そして，UDP のサーバは，local-addr, local-process の設定にも bind() を用いる．foreign-addr, foreign-process については recvfrom() で確認する．UDP のクライアントは，local-addr, local-process の設定には bind() を用い，sendto() で foreign-addr, foreign-process を設定する．

2.5.3 ソケットシステムコール

(1) socket() システムコール　socket() はソケットを作成する．そして，そのソケットを識別するためのソケット記述子を返す．

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

int socket(int domain, int type, int protocol)
```

int domain は，通信ドメインを示し，プロトコルファミリを指定する．TCP/IP の場合は PF_INET を指定する．int type は，ソケットタイプ (ストリーム，データグラム等) を指定する．int protocol(プロトコル番号) は，通信ドメイン内のプロトコルを特定するものである．同一通信ドメインに同じソケットタイプをもつプロトコルが存在しないとき，protocol は 0 を指定できる．なお，このシステムコールを用いる場合には，ヘッダファイル <sys/types.h> と <sys/socket.h>

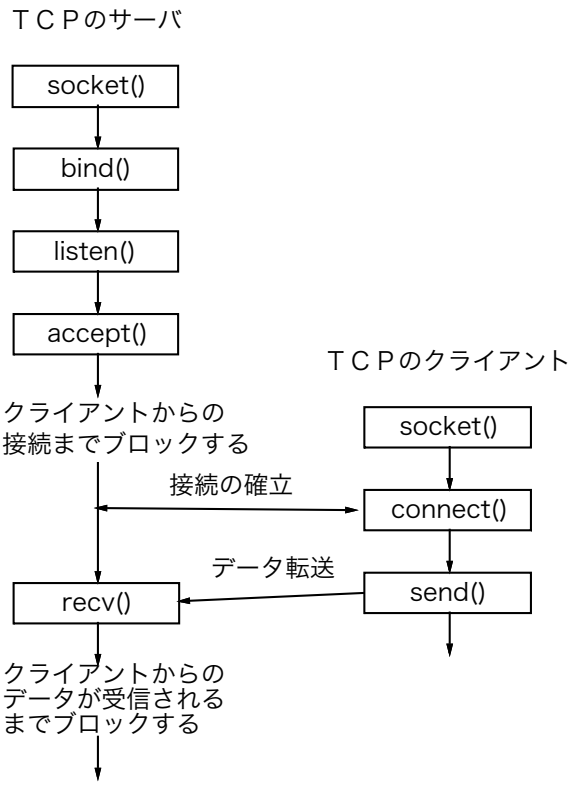

図 7: TCP を用いたプロセス間通信

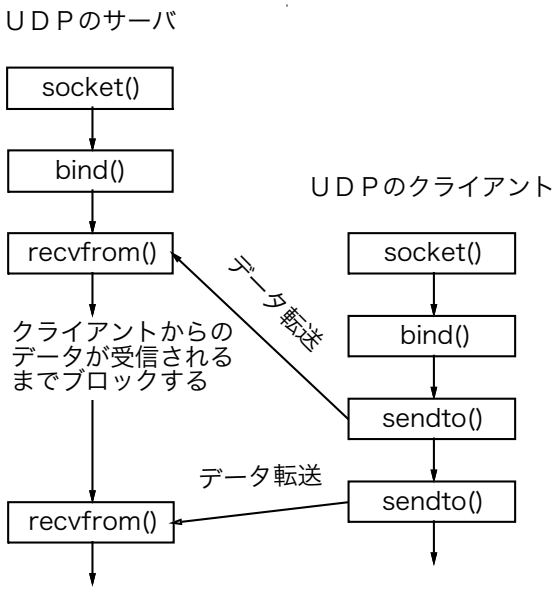

図 8: UDP を用いたプロセス間通信

を取り込む必要がある (以下のシステムコールにおいても同様に，これらのヘッダファイルが必要である).

TCP を用いる場合には，以下のようにソケットを生成する.

```
int sockfd;
sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

但し，int sockfd はソケット記述子である (以下，同様)．また，PF_INET と SOCK_STREAM はヘッダファイル <sys/socket.h> で定義されている．一方，UDP の場合には,

```
int sockfd;
sockfd = socket(PF_INET, SOCK_DGRAM, 0);
```

とする.

(2) bind() システムコール　bind() は，自アドレスと自ポート番号を設定する.

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

struct sockaddr ∗addr は，ソケットアドレス構造体へのポインタを示す．アドレスファミリが AF_INET の場合は，struct sockaddr_in addr; にインターネットアドレスとポート番号を設定する．int addrlen は，ソケットアドレス構造体の大きさを指定する．これは，複数のソケットアドレス構造体を統一されたインタフェースで扱うために必要となる.

bind() は，ソケットアドレス構造体で指定されたインターネットアドレスとポート番号を local-addr, local-process に設定する．但し，local-addr に特定のアドレスを指定すると，複数のインタフェースからの要求が受付けできなくなる．そこで，通常，bind() では，特定のアドレスを指定せず，INADDR_ANY ((u_long)0x00000000) を指定する．

(3) connect() システムコール　connect() は，相手アドレスと相手ポート番号を設定する．また，bind() により自アドレスと自ポート番号が設定されていなければ，適当な値が設定される．TCP は，connect() により初期化セグメントを送信する．

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

connect() は，ソケットアドレス構造体で指定されたインターネットアドレスとポート番号を foreign-addr, foreign-process に設定する．

(4) listen() システムコール　listen() は，ソケットを接続待ちの状態にする．TCP は，初期化セグメントを受信すると，sockfd と同じ性質のソケットを新たに生成し接続待ち行列に加える．そして，生成したソケットに相手アドレスと相手ポート番号を設定する．

```
int listen(int sockfd, int backlog);
```

int backlog は，accept() による接続の受け付けを待つことのできる接続要求の最大数を指定する．この引数は，現時点で許される最大数である 5 に指定されるのが普通である．

(5) accept() システムコール　accept() は，接続待ち行列からソケットを取り出す．接続待ち行列が空の場合，accept() はブロックする．accept() は，戻り値として，クライアントに接続された終点を持つ新しいソケットの識別子を返す．

```
int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

struct sockaddr ∗addr と int ∗addrlen は，初期化セグメントのソースアドレスとソースポート番号を得るために用いられる．

(6) sendto(), recvfrom() システムコール　sendto(), recvfrom() は，UDP で通信する場合に用いられるデータ送受信用のシステムコールである．

```
int sendto(int sockfd, char *buff, int nbytes, int flags,
                       struct sockaddr *addr, int addrlen);
int recvfrom(int sockfd, char *buff, int nbytes, int flags,
                       struct sockaddr *addr, int *addrlen);
```

sendto() はソケット記述子 (sockfd) を用いてバッファ (buff) のデータを nbytes で指定されたバイト数だけ送信しようとする．そして，実際に送信したバイト数を返す．また，sendto() は struct sockaddr ∗addr でデスティネーションアドレス (foreign-addr) とポート番号 (foreign-process) を指定する．recvfrom() はソケット記述子 (sockfd) から nbytes で指定されたバイト数だけバッファ (buff) にデータを読み込もうとし，実際に読み込んだバイト数を返す．また，recvfrom() は受信したデータグラムのソースアドレスとソースポート番号を struct sockaddr ∗addr に設定する．なお，int flag は，本実験では使用せずに 0 とする (send(), recv() システムコールでも同様)．

(7) send(), recv() システムコール　send(), recv() は，TCP で通信する場合に用いられるデータ送受信用のシステムコールである．また，connect() を実行した場合は UDP でも用いられる．

```
int send(int sockfd, char *buff, int nbytes, int flags);
int recv(int sockfd, char *buff, int nbytes, int flags);
```

これらのシステムコールの sendto(), recvfrom() に対する違いは，struct sockaddr ∗addr と int ∗addrlen を指定しない点にある．

(8) close() システムコール　ソケット記述子を削除する．これにより通信を終結する．

```
int close(int sockfd);
```

(9) setsockopt() システムコール　ソケットオプションを設定する．

```
int setsockopt(int sockfd, int level, int optname, char *optval, int optlen)
```

int level はどの階層を設定対象とするかを示す．また，int optname はオプションの種類を示し，char ∗optval がオプションの値を示す．int optlen は，optval が指すバッファサイズを示す．
socket() を実行した後で，level: SOL_SOCKET, optname: SO_SNDBUF, optval: ウィンドウサイズ として setsockopt() を実行することにより，ウィンドウサイズを指定できる．

(10) select() システムコール　このシステムコールは，複数のソケット記述子を扱わなければならないときに用いられる．

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

int select(int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);

FD_ZERO(fd_set *fdset);          /* fdset のすべてのビットをクリアする */
FD_SET(int fd, fd_set *fdset);   /* fdset 中の fd のビットをセットする */
FD_CLR(int fd, fd_set *fdset);   /* fdset 中の fd のビットをクリアする */
FD_ISSET(int fd, fd_set *fdset); /* fdset 中の fd のビットをテストする */
```

このシステムコールを用いると，カーネルに対して，指定した複数のイベントのうち，どれかが発生するまで呼び出したプロセスを待ち状態に置くという指示を行うことができる．
select に与える要求としては，例えば,「ファイル記述子の集合 {1,4,5} のいずれかが読み出し可能になったときに知らせよ」,「ファイル記述子の集合 {2,7} のいずれかが書き込み可能になったときに知らせよ」,「ファイル記述子の集合 {1,4} のいずれかに例外状態が発生したときに知らせよ」などが考えられる．readfds，writefds，exceptfds 引数は，検査の対象となるディスクリプタを指定し，それぞれ，読み出しの可能なディスクリプタ，書き込みの可能なディスクリプタ，例外状態の発生しているディスクリプタを指定する．

2.6 アプリケーションプロトコル

ソケット API を利用してアプリケーションを実現するためのプロトコルとして，さまざまなものが考えられ，実際に用いられている．ここでは，その例を示す．

2.6.1 Telnet

Telnet は，遠隔地にある計算機に遠隔ログインするために使用されるプロトコルである．RFC 854 で規定されている．Telnet では，ポート番号 23 を使用する．

多くの Telnet クライアントでは，ポート番号を指定できることから，Telnet 以外のテキストベースのソケット通信（例えば HTTP など）のクライアントとして利用することも可能である．

なお，Telnet は，ログインの際に使用するユーザ名やパスワードを含めた通信内容が平文でやりとりされる．このため，セキュリティの面で問題があるので，オープンなネットワークを介した遠隔ログインには使用すべきではない．

2.6.2 HTTP

HTTP（Hyper Text Transfer Protocol）は，WWW（World Wide Web）サービスを実現するために用いられるプロトコルである．詳細は，文献 [4] が詳しい．

HTTP はリクエストとレスポンスからなる非常に単純なプロトコルである．まず，WWW クライアントは，リクエストとして表示したい Web ページの URL（Uniform Resource Locators）を送信する．これに対して，WWW サーバは，自身が持つ要求された URL に対応するコンテンツ（HTML や画像など）をクライアントに送信する．

一般に HTTP のポート番号は 80 番である．ただし，それ以外のポート番号を利用することも可能である．

HTTP リクエストの 1 行目には，必ず，Method と Request-URL と HTTP-Version からなる Request-Line が存在する．ここで，Method は，リクエストの種類を表している．Request-URL は，リクエストが対象としている URL を示す．また，HTTP-Version は，使用する HTTP のバージョンを表す．そして，2 行目以降に HTTP ヘッダ，CRLF（改行コード）を挟んで，メッセージボディが続く．ただし，メッセージボディは存在しないことがある．

以下に，HTTP リクエストの一例を示す．ここでは，GET メソッドを用いて，接続されたホスト inl.elcom.nitech.ac.jp のファイル index.html を取得しようとしている[1]．

```
GET /index.html HTTP/1.1
Host: inl.elcom.nitech.ac.jp
（改行）
```

このような HTTP リクエストを受信したサーバは，そこに含まれるリクエストを解釈した後，HTTP レスポンスを返送する．レスポンスメッセージの最初の行は，HTTP-Version，ステータスコード番号，それに関連したテキストフレーズからなるステータスラインとなる．それぞれの要素は，スペースにより区切られる．

先の HTTP リクエストに対応した HTTP レスポンスの例を示す．

[1] 2010 年 4 月現在，このホストは存在しません．

```
HTTP/1.1 200 OK
Date: Sat, 28 Jan 2006 05:55:41 GMT
Server: Apache/1.3.26 (Unix)
Last-Modified: Thu, 23 Aug 2001 12:09:12 GMT
ETag: "4217f-2d0-3b84f268"
Accept-Ranges: bytes
Content-Length: 720
Content-Type: text/html

<HTML>
<HEAD>
 <META HTTP-EQUIV="Refresh" CONTENT="1;URL=http://inl.elcom.nitech.ac.jp/index.shtml">
 <META name="robots" content="noindex,nofollow">
 <TITLE>Tasaka Lab.'s Home Page</TITLE>
 （以下，HTML 文章が続く）
```

3 使用機器

- 教育用計算機（Linux）

4 演習内容

※ 各週の演習内容は，演習時間および moodle にてお知らせします．

4.1 反復サーバによる 1 対 1 通信とバイナリデータ伝送

反復サーバとクライアントからなる 1 対 1 の通信プログラムを考える．これにより，ソケットの基本的な使い方を学習する．また，ソケットを用いる上での TCP と UDP との違いを理解する．

(1) 教員があらかじめ用意した，キー入力したもの（標準入力）を送信する client と，それを受信し標準出力へ出力する server をダウンロードせよ．そして，それらをコンパイルし，動作を確認せよ．なお，これらのプログラムは UDP を用いて書かれている．

(2) 教員があらかじめ用意した TCP を用いたサンプルプログラムを元に，上で作成したプログラムを TCP を用いるように変更せよ．そして，TCP と UDP で動作の違いを確かめよ．

(3) 上では標準入出力のデータを扱ったが，ファイルからのバイナリデータを送信するクライアントプログラムと，クライアントからのデータを受け取ってファイルに保存するサーバプログラムを作成せよ．実際にファイルを送受信し，送信したファイルと受信したファイルを比較することで，壊れることなく転送できることを確認せよ．なお，トランスポートプロトコルには TCP を使用すること．

4.2 並行サーバの実現

4.1 節で作成したサーバプログラム（TCP を用いたもの）を拡張して，複数のクライアントからの接続を許可する並行サーバを実現する．

(1) 4.1 節 (2) で作成された TCP を用いた標準入出力を扱うサーバプログラムを拡張し，client 側から送信された文字列を返送し，client 側の標準出力に表示されるようにせよ．なお，server では，ただ client から送信されてきた文字列を返送するのではなく，なんらかの加工（大文字小文字の変換，プロンプトの付与など）を行うこと．

(2) (1) で作成したプログラムをもとに，select() を用いて，一定時間接続要求がなかったときにプログラムが自動的に終了するように改良せよ．

(3) (1) で作成したプログラムをもとに，fork() によるマルチプロセス化により，複数クライアントからの接続を処理する並行サーバを実現せよ．

4.3 HTTP プロトコルの実装

既存アプリケーションプロトコルを用いたプログラミング手法の学習として，本演習テーマでは HTTP を対象とする．

課題 1：Telnet を用いた HTTP の振舞い確認

Telnet を用いて，HTTP サーバにアクセスせよ．そして，情報のやりとりを調べよ．具体的には次の場合にサーバがどのような振舞いをするか確認せよ．

- 正常に情報を取得できる場合
- GET 以外のメソッドを使う場合
- リクエストヘッダの中身を変えた場合
- 存在しないファイルへアクセスした場合

課題 2：HTTP クライアント作成

サンプルプログラムをもとに，単純な Web クライアントを作成せよ．コマンドラインから入力した URL に対して，GET メソッドによりリクエストを送信し，応答結果を標準出力に表示させよ．

課題 3：HTTP サーバ作成

(1) GET メソッドを用いた HTTP リクエストに応答できる HTTP サーバプログラムを作成せよ．動作確認に用いるクライアントとしては，課題 2 で作成した HTTP クライアントならびに一般的な Web ブラウザ（Mozilla Firefox や Internet Explorer）を用いよ．クライアントからリクエストされたファイルを返送できるように拡張せよ．まずは，text/html のファイルタイプのみを対象とし，非持続的コネクションを考え，エラー処理を考えなくても良い．

(2) 画像ファイルを含んだ Web ページを扱えるよう，画像ファイルに対応できるようにせよ．

(3) 不正なリクエストに対して，エラーコードを返送するように拡張せよ．

(4) 可能であれば拡張機能を追加せよ．例えば，持続的コネクションをサポートできるように拡張することなどが考えられる．

5　考察及び検討事項

各実験項目において作成したプログラムの設計方針・動作検証方法・動作検証結果をまとめること．

また，1 週目の実験に関して，TCP と UDP の違いを考察せよ．2 週目の実験に関しては，select や fork が並行サーバを実現する上で必要となる理由を説明せよ．

6　参考文献

[1] 田坂修二：“情報ネットワークの基礎”，数理工学社（2003）．

[2] W. リチャード・スティーヴンス著，篠田陽一訳：“UNIX ネットワークプログラミング”，トッパン (1992).

[3] B. W. カーニハン，D. M. リッチー著，石田晴久訳：“プログラミング言語 C 第 2 版”，共立出版 (1989).

[4] Studying HTTP, （http://www.studyinghttp.net）．

付録

A. ライブラリとシステムコール

ネットワークプログラミングで用いられる C のマクロと関数について説明する．また，計時の際に用いるシステムコールについても示す.

(1) バイト順序変換ルーチン　次の四つの関数は，異なったアーキテクチャあるいは異なったプロトコル間で生じる可能性があるバイト順序の相違を取り扱う.

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

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

htonl()	長い整数をホストのバイト順序からネットワークのバイト順序に変換する
htons()	短い整数をホストのバイト順序からネットワークのバイト順序に変換する
ntohl()	長い整数をネットワークのバイト順序からホストのバイト順序に変換する
ntohl()	長い整数をネットワークのバイト順序からホストのバイト順序に変換する

これらの関数は，長い整数が 32 ビット，短い整数が 16 ビットであることを仮定している．例えば，ポート番号をソケットアドレス構造体に設定するとき htons() を用いる.

(2) バイト操作ルーチン　ソケットアドレス構造体の内部には，複数バイトのフィールドが存在する．4.3BSD では，ユーザ定義のバイト列を操作する次の三つのルーチンを定義している．ここでユーザ定義のバイト列とは，C 言語の標準的な文字列 (必ずヌル文字で終端されている) ではないことを意味する.

```
bcopy(char *src, char *dest, int nbytes);
bzero(char *dest, int nbytes);
int bcmp(char *ptr1, char *ptr2, int nbytes);
```

bcopy() は，指定された数のバイトを src から dest にコピーする．bzero() は，指定された数のヌルバイトを dest から始まる領域にセットする．bcmp() は，任意のバイト列を比較する．これら三つの関数は，主にソケットアドレス構造体を操作するために用いられる.

(3) アドレス変換ルーチン　インターネットアドレスは通常 192.43.235.1 のように dotted-decimal 形式で書かれる．次のルーチンは，dotted-decimal 形式と in_addr 構造体で用いられる 32 ビットのアドレス形式間の相互変換を行う.

```
#include <sys/socket>
#include <netinet/in.h>
#include <arpha/inet.h>

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

inet_addr() は，dotted-decimal 形式の文字列を 32 ビットのインターネットアドレスに変換する．inet_ntoa() は逆の変換を行う．

(4) gettimeofday() システムコール　日付及び時間を獲得する．

```
struct timeval {
        long    tv_sec;         /* 秒 */
        long    tv_usec;        /* マイクロ秒 */
};

#include <sys/time.h>

int gettimeofday(struct timeval *tp, struct timezone *tzp)
```

struct timeval の long tv_sec と long tv_usec は，1970 年 1 月 1 日から経過した秒時間及びマイクロ秒時間を示す．struct timezone は，夏時間の修正のために用意されているが，今回は用いない．

B. サンプルプログラム

UDP を用いてプロセス間通信を行う場合のプログラムの一例を以下に示す．この例では，クライアントは標準入力からデータを入力してサーバに送信し，サーバは受信したデータを標準出力に出力するようにしている．

```
/* UDP のサーバプログラム */

/* 以下のヘッダファイルは/usr/include にある */
#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>

#define BUFFSIZE 512                /* メッセージの最大長 */

main(argc, argv)
int     argc;
char    *argv[];
{
    int     port = 6666;            /* 自ポート番号に 6666 を使用 */
    int     sockfd;                 /* ソケット記述子 */
    struct sockaddr_in      addr, my_addr;
                                    /* インタネットソケットアドレス構造体 */
    int     addrlen;
    char    buff[BUFFSIZE];         /* 受信バッファ */
    int     nbytes;                 /* 受信メッセージ長 */
```

```
    /* ソケットの生成 */
    if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");          /* ソケットの生成失敗 */
        exit(1);
    }

    /* 自アドレスと自ポート番号の設定 */
    bzero((char *) &my_addr, sizeof(my_addr));      /* 0 クリア */
    my_addr.sin_family = AF_INET;                   /* アドレスファミリ */
    my_addr.sin_addr.s_addr = htonl(INADDR_ANY);    /* アドレス */
    my_addr.sin_port = htons(port);                 /* ポート番号 */
    if (bind(sockfd, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0) {
        perror("bind");            /* 自アドレスと自ポート番号の設定失敗 */
        exit(1);
    }

    addrlen = sizeof (addr);
    bzero(buff, sizeof(buff));     /* 受信バッファの 0 クリア */

    /* 受信 */
    if ((nbytes = recvfrom(sockfd, buff, BUFFSIZE, 0,
                           (struct sockaddr *)&addr, &addrlen)) < 0) {
        perror("recvfrom");        /* 受信失敗 */
        exit(1);
    }

    printf("%s\n", buff);          /* 受信したメッセージを標準出力に出力 */
    close(sockfd);                 /* ソケットの終了 */

}

/* UDP のクライアントプログラム */

/* 以下のヘッダファイルは/usr/include にある */
#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>
#include        <netdb.h>

#define BUFFSIZE 512               /* メッセージの最大長 */

main(argc, argv)
int     argc;
char    *argv[];
{
    char    *host;                 /* 相手ホスト名 */
    int     port = 6666;           /* 相手ポート番号は 6666 */
    int     sockfd;                /* ソケット記述子 */
    struct sockaddr_in      addr, my_addr;
                                   /* インタネットソケットアドレス構造体 */
    int     addrlen;
    char    buff[BUFFSIZE];        /* 送信バッファ */
    int     nbytes;                /* 送信メッセージ長 */
    struct hostent  *hp;           /* 相手ホストエントリ */

    if (argc == 2){
        host  = argv[1];           /* 相手ホスト名の設定 */
    } else {
        fprintf(stderr,"usage: %s host \n",argv[0]);
        exit(1);
    }
```

```
    /* 相手ホストエントリの取得 */
    if ((hp = gethostbyname(host)) == NULL) {
        perror("gethostbyname");  /* 相手ホストエントリ取得失敗 */
        exit(1);
    }

    /* 相手アドレスと相手ポート番号の設定 */
    bzero((char *)&addr, sizeof (addr));            /* 0 クリア */
    addr.sin_family = AF_INET;                      /* アドレスファミリ */
    bcopy(hp->h_addr, (char *)&addr.sin_addr, hp->h_length);
                                                    /* アドレス */
    addr.sin_port = htons(port);                    /* ポート番号 */

    /* ソケットの生成 */
    if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");         /* ソケットの生成失敗 */
        exit(1);
    }

    /* 自アドレスと自ポート番号の設定 */
    bzero((char *) &my_addr, sizeof(my_addr));      /* 0 クリア */
    my_addr.sin_family = AF_INET;                   /* アドレスファミリ */
    my_addr.sin_addr.s_addr = htonl(INADDR_ANY);    /* アドレス */
    my_addr.sin_port = htons(0);                    /* ポート番号 */
    if (bind(sockfd, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0) {
        perror("bind");           /* 自アドレスと自ポート番号の設定失敗 */
        exit(1);
    }

    addrlen = sizeof (addr);
    fgets(buff, BUFFSIZE, stdin); /* 送信メッセージの取得 */
    nbytes = strlen(buff);        /* 送信メッセージ長の設定 */

    /* 送信 */
    if (sendto(sockfd, buff, nbytes, 0,
               (struct sockaddr *)&addr, addrlen) != nbytes) {
        perror("sendto");         /* 送信失敗 */
        exit(1);
    }

    close(sockfd);                /* ソケットの終了 */
}
```