ネットワークプログラミング講座–UNIX、Windows & Java 環境における

石井　秀治 (NEC)、
日比野　洋克 (オレンジソフト)

1998 年 12 月 17 日（木）

InternetWeek 98　国立京都国際会館

(社)日本ネットワークインフォメーションセンター編

目次

本セミナーについて(目的、内容) (石井)

本セミナーの目的は、大きく分けて以下の３点です。

- ネットワークプログラムとはどのようなものであるかを理解する（初心者対象）
- Windows プログラミング開発環境及びネットワークプログラミングの基礎を理解する（UNIX プログラマ対象）
- UNIX プログラミング開発環境及びネットワークプログラミングの基礎を理解する（WINDOWS プログラマ対象）

また内容については、一般的で簡単な最初の導入部分についてのみ説明し、難しいことは割愛させていただきます。難しいこと、例えば、ルーティングテーブルの制御、ネットワークインターフェースの制御、ブロードキャスト、マルチキャストといったところは省かせていただきます。

説明順序ですが、まず簡単にインターネットの概要について説明します。それから、主題であるネットワークプログラミングについて、UNIX 編、Windows 編および目次には記述していませんが Java 編を説明します。次にサンプルプログラムを、サーバ側ではなくてクライアントのプログラム、メールの SMTP のクライアントと POP のクライアントの簡単なものについて紹介・解説して、最後にまとめるというふうに進めていきます。

インターネット概要 (石井)

インターネットとは、いろいろな異なったメディア（LAN、ISDN、あるいは ATM など）を統合して論理的なネットワークに見せる技術であると言えます。
インターネットには大きなポイントが２つあります。それは「IP アドレス」と「経路制御」です。
インターネットを図に示すと図１のような感じになります。

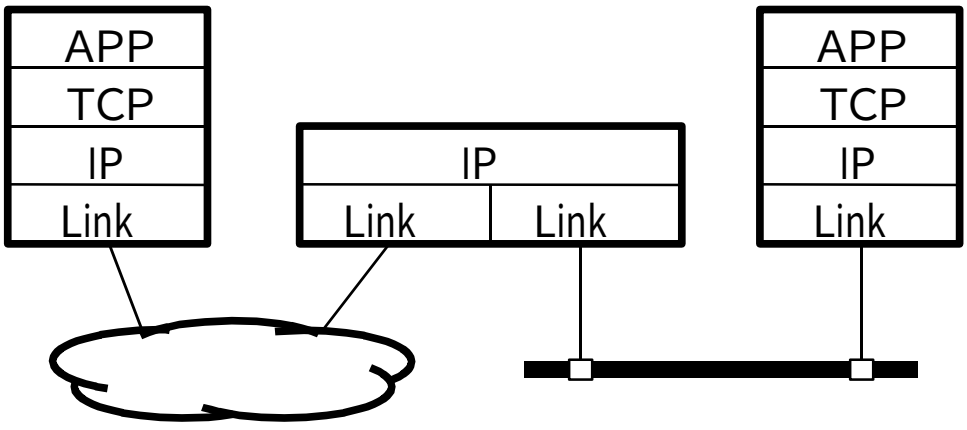


図１インターネット概要

よく TCP/IP という言い方をしますけれども、図２に示すように IP のほかにトランスポートプロトコルである TCP とか UDP というものがあり、それを使って各種アプリケーション（メール、Web、http など）のプログラムをつくっていきます。

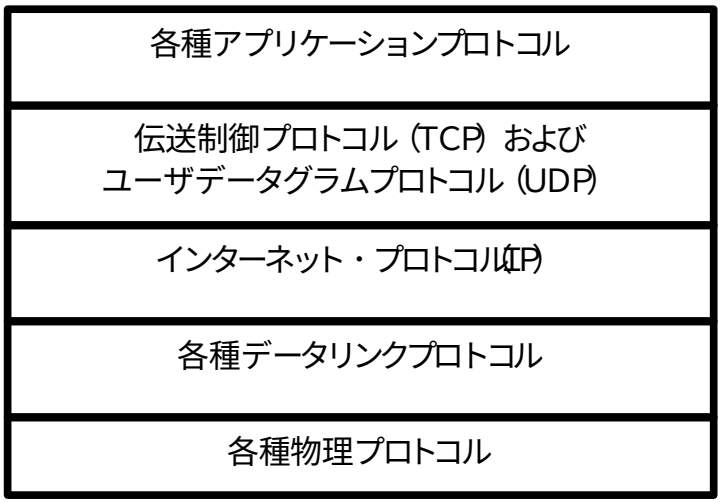


図２ TCP,UDP と IP を中心にしたプロトコールスタック

IP アドレスとホスト名

先ほど出ました IP アドレスについてですが、IP をしゃべるホスト―コンピュータやルータ―は、IP アドレスというのを持っています。IP アドレスは、一言でいうと固定長の論理アドレスです。IPv4 では 32 ビットの長さを持っています。最近よく聞かれる IPv6 では 128 ビットを持っています。ここで紹介するのは v4 についてで、v6 についてはここでは説明しません。IP アドレスは、ホストのネットワークインターフェースを識別するために存在します。その意味は次のとおりです。すばわち、IP アドレスは大抵の場合ホストを識別するためのものですが、例えばホスト（コンピュータ）によってはネットワークインターフェースが複数付く場合があります。その最も顕著なものがルータです。ルータは複数のインターフェースを持っていますから、そのインターフェースごとに別々な IP アドレスがつきます。そういった意味で IP アドレスはネットワークを識別するものです。具体的に文字列であらわすと、192.168.0.3 というように 10 進数４桁を点で区切った表し方をします。

もう一つホストを識別するものとして、こちらのほうが多分わかりやすいと思いますが、ホスト名というのものがあります。これは何かといいますと、読んで字のごとくホストを識別するための名前です。機械とかプログラムは IP アドレスのような数字のほうがプログラムしやすいのですが、人間のほうはこういった文字列のほうがわかりやすいわけです。実際にユーザやアプリケーションが使う名前はこのホスト名（例えば、www.nic.ad.jp）になります。そのために、ホスト名から IP アドレスへ変換するための操作があり、これについては後述します。

トランスポート

次がトランスポートの話です。先ほどのプロトコルスタックで IP の上のところ、その部分をトランスポートといいます。IP ではデータ配送に信頼性がないと書きました。その理由の第一は、IP ではパケットがあまり混んでくると途中のルータで破棄されたり、電気的なノイズでパケットが壊れてしまって破棄され、IP パケットが相手に確実に届くという保証が必ずしもないことです。第二には、IP では送信した順序と違う順序で受信することが起こり得ることです。第三には、IP ではパケットが重複するかもしれないことです。つまり、同じパケットが途中で何らかの形で２つ以上に分かれ、コピーされて届くかもしれません。例えば、タイムアウト時に相手からの確認を待たずに再送した結果、相手に同じものが２個届いてしまうということが起こり得ます。以上の意味から、IP は信頼性がないプロトコルと言えます。

これではとてもアプリケーションのプログラムは作れませんから、例えばパケットが必ず届くようにするとか、順序を保証するとか、２つ届いても大丈夫なようにするために、IP の上のプロトコルである「トランスポートプロトコルで信頼性を確保」します。このトランスポートプロトコルには、TCP と UDP があります。これからこの２つについて簡単に説明します。

TCP

最初によく使われている TCP ですが、これは信頼性のあるデータ配送を提供します。TCP には以下の特徴があります。

- **ストリーム指向**
  何らかの構造を持ったデータが飛ぶのではなく、ただのバイトの列をそのまま書き込むと、相手の方でもそのまバイトの列が同じ順番で出てくる。何かデータに境界があるといったものではなくて、単にずらずらとバイトの列が流れて到達するというものです。
- **バーチャルサーキット（コネクション型）**
  データを転送するときにあらかじめ自分と相手との間でまずコネクションというものを張ります。これは、電話をかけて相手がちゃんといるかどうかを確認するといった手順に相当します。
- **バッファ付き転送**
  転送に当たって、一応バッファリングしておき、ネットワークの状況に応じて少しずつ出したり、一杯出したりということをします。
- **全二重**
  双方向のコネクションがあるということです。

また、TCP では、ユーザから見ると、下位プロトコルを意識せずに通信を行うことが出来ます。つまり、IP では前述したように順序が狂うとか、パケットがなくなってしまうということが起こり得ましたが、TCP ではそういったことを気にせずにデータ通信をすることができます。TCP は以上の機能を提供するプロトコルです。

UDP

もう一方の UDP ですが、これは信頼性のないデータグラム配送を提供します。大雑把に言いますと、UDP は IP パケットにポート番号（このすぐあとで説明します）とチェックサムを付けた程度のもので信頼性がありません。したがって、UDP では信頼性の保証がないため、信頼性を確保するためには UDP のユーザ（プログラマ）の方で何らかの仕組みを組み込む必要があります。TCP の場合は、信頼性を確保するための再送とかパケット順番のチェックの処理にオーバヘッドがかかりますが、UDP の場合はそれがなく、プログラマにそういった機能の実装が全部まかされています。実際、NFS、TFTP、DHCP や DNS プロトコルでは UDP を使用しています。それでは何故 UDP を使うかとい言いますと、例えば LAN だけで使う場合には TCP はちょっと重

過ぎ、もう少しスピードを出すために、TCP のやっていることを自前で全部やるような場合によく UDP が使われます。

ポート

次に先程出てきましたポート番号という概念についてご説明したいと思います。ポート番号というのは、ホスト内での通信端点のことをいいます。通信端点というのはプログラムが通信をするための口のことをいい、それにより同一ホスト内でサービスを区別します。例えば同じホストの中で http、sendmail、pop といったものを区別するためにあるものです。そのポートがいくつかあるわけですが、それぞれに名前すなわち番号がついています。ポート番号は 16 ビットの正の整数でそれぞれ番号がつけられています。その番号は勝手につけていいかというとそうではなく、サービスすなわちメール、http(Web)、FTP あるいは telnet でそれぞれ決まった番号がついています。このポート番号というのは TCP にも UDP にもありますが、トランスポートのプロトコールごとに独立となっています。つまり、TCP の 80 番というポートと UDP の 80 番というポートは別です。同じ 80 番というのを使っていますけれども全く別のものです。なぜかというと、TCP と UDP とはトランスポートプロトコールとしては異なるプロトコールだからです。図３はポートの集まりを簡単に示したものです。

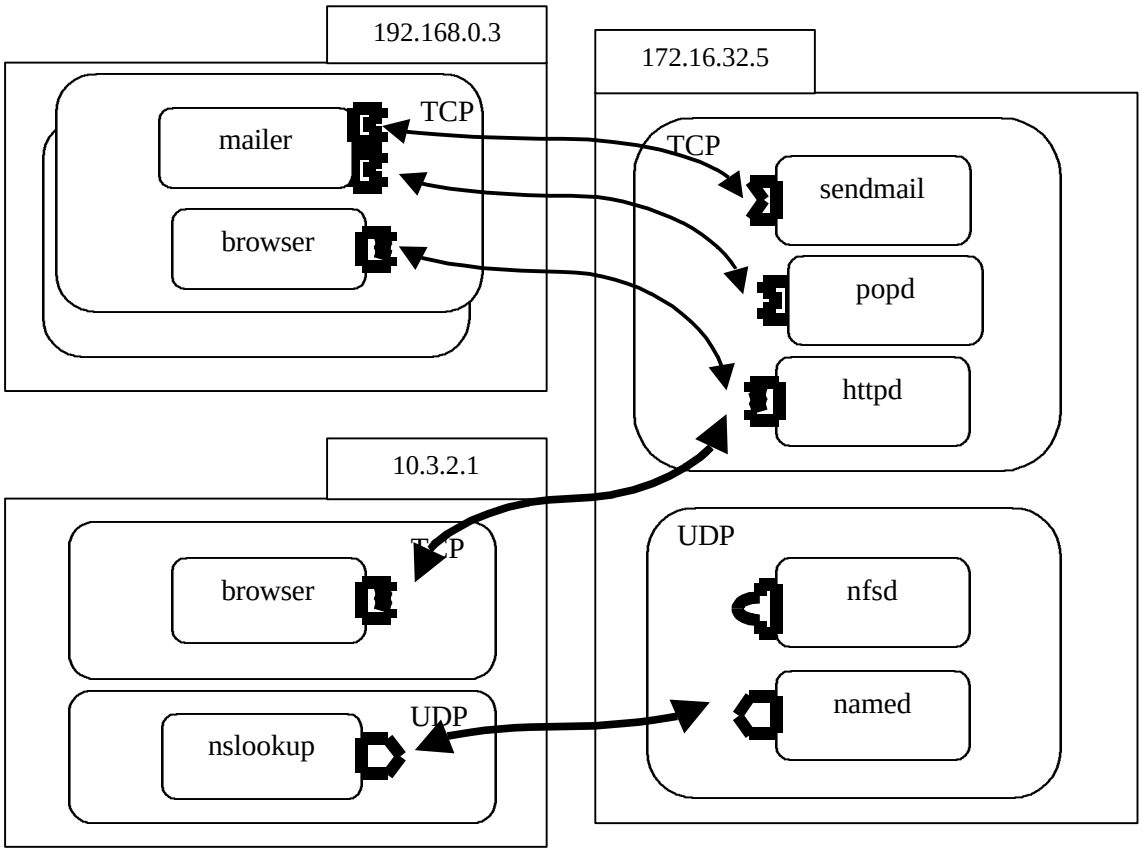


図３ ポートの概念

図３ではホストが３つあり、192.168.0.3 と 10.3.2.1 という２つのホストおよび右側に 172.16.32.5 というホストがあります。192.168.0.3 のホストには TCP だけが存在し、mailer（何かメールを読むためのプログラム）と、browser（Netscape、Internet Explorer といったプログラム）が図３のようにポートを持っています。別に１つのア

プリケーションに１つという制限はないので、図の mailer ように２つでも３つでもたくさんポートを持つことはできます。同じように下の 10.3.2.1 のホストでは、上の browser が TCP のポートを持っていますし、下の nslookup が UDP のポートを持っています。右側はサーバとしていますけれども、TCP では sendmail、popd と httpd というサーバが待機し、UDP では nfsd と named のサーバが待機しています。
実際にどういうふうに通信するかを図３でポート間を結んでいる双方向の線分に示しました。例えば、左側の mailer は、右側の送信側のサーバプログラムである sendmail、とお話しをしますし、それと同時に popd というメールを読み込むところにもコネクションを張ります。また、browser がこの httpd にコネクションを張りにいきます。右側の httpd に注意していただきたいのですが、一つの httpd に別のところ、左の下のホストの browser からやはりこうやって通信することもできます。以上がポートのあらましです。

通信端点
これは、これから出てくるプログラムの話で重要になります。通信相手を指定するという場合に通信端点というものを指定するわけですが、このときの方法をあらわしたのがこの図４です。

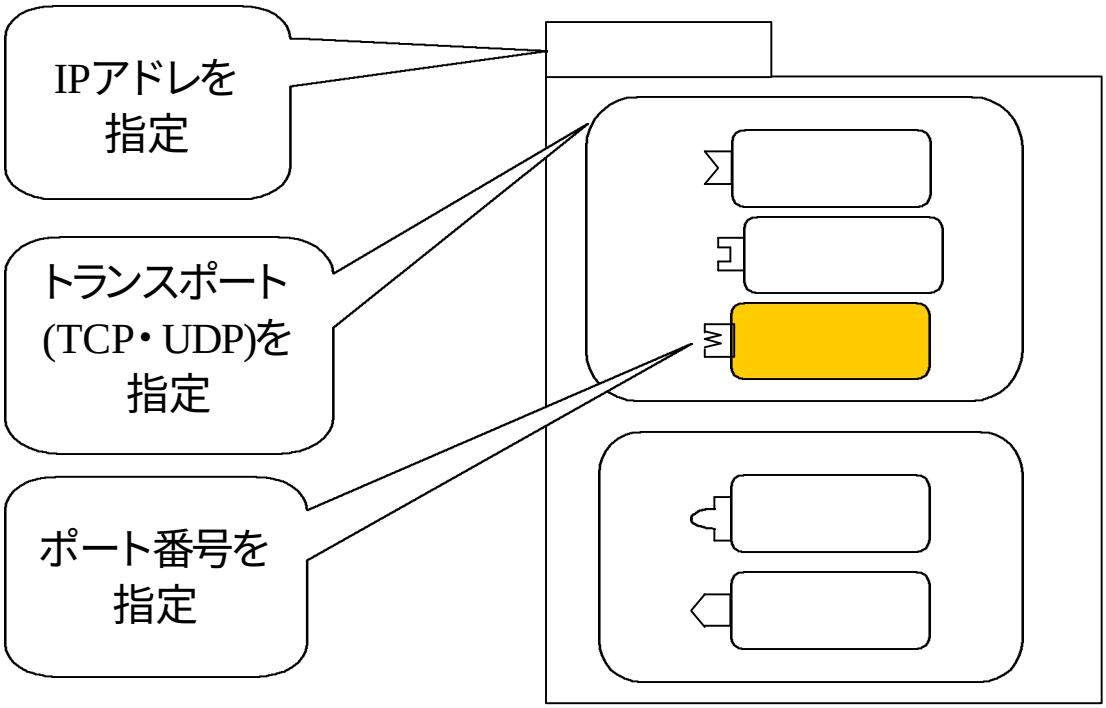


図４ 通信端点の指定方法

図４でも示しているように、通信端点は次に述べる３つで表します。最初に、相手のアドレスです。相手のホスト、機械を指定します。次に、トランスポートを指定します。つまり、TCP の枠の中なのか、UDP の枠の中なのかを指定します。最後にその枠の中の何番のポート番号かを指定します。
以上がインターネットの概要です。

インターネットプログラミング

次に、通信をするための API、プログラムをするためのインターフェースというのが次のように世の中にはたくさんあります。

- **Berkerley socket**
- TLI/XLI
- sunrpc
- **winsock**
- RMI
- **WinInet**

すぐ後で Berkeley socket という BSD 系でよく使われるインターフェースについて説明します。その一つ下の TLI とか XLI というのは System V 系のインターフェースですが、こちらはここでは割愛します。あと、winsock は Windows 関係の API の名前ですが、これは後程説明します。他にもいろいろありますけれども、ここではこの２つを中心に説明します。

UNIX 編（石井）

アプリケーションインターフェース概要（socket）

ここでは UNIX 編ということで、UNIX Berkeley（BSD）系のアプリケーションインターフェースを説明します。まず socket interface の話をします。前述したように、BSD 系の UNIX では、socket と呼ばれる一連のシステムコール群を用いて通信を行います。その特徴は次のとおりです。

- アドレス形式に依存しない
- サーバ・クライアントモデル（後述します）
- プロトコル非依存
  本セミナーでは TCP/IP と銘打っていますが、別に XNS や別のほかのプロトコルで socket を使うこともできます。つまり、socket というもので抽象化して、アドレス形式といったものをプログラマには隠しています。

次にアプリケーションインターフェースにどのようなものがあるかということを大雑把に次に示します。

- socket の生成　**socket**
- 名前付け　**bind**
- 接続受理準備　**listen**
- 接続要求　**connect**
- 接続要求受理　**accept**

最初に socket をつくるためのインターフェースである **socket** があります。それから名前付けと書きました **bind** ですが、これは生成した socket に前述したポート番号（例えば 80 番）をつけたい場合に使うインターフェースです。３番目の **listen** というのは接続受理準備と書いていますが、これはサーバ側で、つまりサービスを受け付けるための準備をするためのインターフェースです。逆に、**connect** というのはクライアント側で相手のサーバに対して、このポート番号に接続してくださいと指示するためのインターフェースです。**accept** というのは、このクライアントからの接続要求である **connect** を待って、来たらそれに対して OK を出してコネクションを張るためのインターフェースです。

次に実際に接続したらデータを転送するわけですが、データ転送に関しては次に示した６つインターフェースが用意されています。

- データ転送　　**read, write,**
  recv, send,
  recvfrom, sendto

UNIX のシステムコールに詳しい方だとお分かりだと思いますが、普通の端末とかファイルの I/O で使う **read** とか **write** のインターフェースをそのまま使うこともできます。そのほかにネットワーク用の通信に特有の制御をしたい場合には、**recv**、**send**、**recvfrom** あるいは **sendto** と呼ばれるインターフェースも用意されています。単純なものであれば、この read、write で単に読み書きをすればよいようになっています。
データ転送が終わったら、セッションを終了します。「切断」と書きましたが、切断に関しては次にすインターフェースが用意されています。

- 切断　　**shutdown, close**

shutdown と **close** がありますが、**close** の方は、**read**、**write** と同じでファイルのクローズと同じですけれども、こういったインターフェースが用意されています。
そのほかに、ここではあまり言及しませんが次のインターフェースが用意されています。

- その他　　**ioctl, socketpair,**
  setsockopt, getsockopt,
  getsockname, getpeername

ioctl は細かい制御を行うものです。

以上が socket interface と呼ばれているものです。

socket を用いた通信

socket を用いた通信の大まかなオーバービュー、流れを、まず TCP の場合について図５に示します。

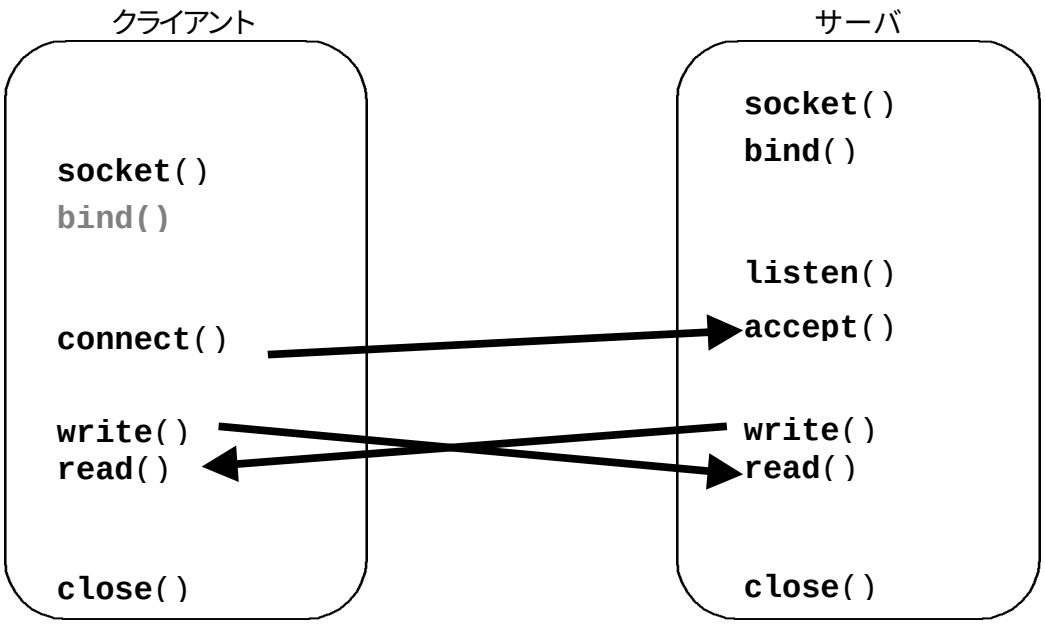


図５ **socket** を用いた通信（**TCP**）

クライアントとサーバの言葉の説明はまだしていませんでしたが、サーバというのはサービスを提供する人で、クライアントというのはそれを使う人です。サーバは要求の受付を待っていて、何かクライアントがサービスを要求してきたらそれに答えるというものです。
まず最初に何をするかというと、クライアントとサーバどちらも socket をつくります。次にクライアントとサーバで同じように、bind によって名前付け操作すなわちポート番号を割り当てる操作を行います。ただし、サーバの場合はポート番号の何番で待ってほしいというのがありますから必ず実行しなくてはいけないのですが、クライアントの方に関しては必ずしも実行する必要がありません。これもあとで説明しますが、特に自分は何番を使いたいということがなければ勝手にシステムの方で空いている番号を割り当ててくれます。
サーバのほうは listen によって準備ができましたといってから、accept というインターフェースを呼ぶと、クライアントから受付が来るまで待ちます。クライアントの方は socket をつくり、bind を行って（これは実行しなくてもいいのですが)、connect というシステムコールを呼びます。そうすると、引数で指定したホストとポート番号に従って相手を見つけて通信をしようとします。ここで相手（サーバ）が待っていると、コネクションが確立されます。あとは、read、write をそれぞれ好きなように実行して、終わったら close します。close の前に shutdown が抜けていますが、ここで必要な場合は shutdown という操作を行います。
以上が TCP の大きな流れです。もう一度整理しますと、クライアント側は、socket(socket,bind)をつくって、接続(connect)して、データ転送(read/write)し、終了(close)。サーバ側は、同じく socket をつくって(socket,bind)、受付準備（listen）をして、それから実際に待って(accep)、データ転送(read/write)して,終わる(close)。そういう形になります。
TCP の場合はこのように手順がかなりあるのですが、UDP の場合はすごくシンプルです。UDP の場合の socket を用いた通信の大まかな流れを図６に示します。

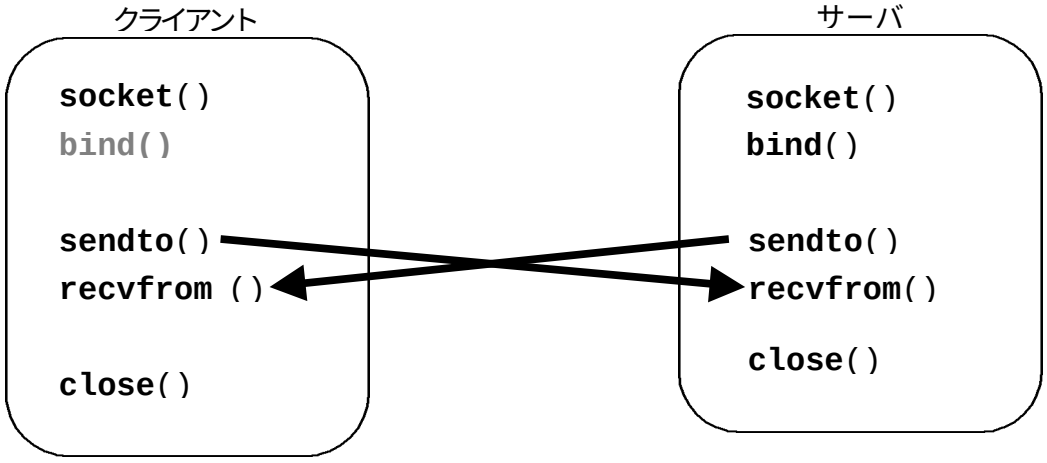


図６ socket を用いた通信（UDP）

socket をつくって bind するところは TCP と一緒です。クライアント側ではポート番号をつけなくてもいいところまでは一緒ですが、あとはいきなり sendto、recvfrom で通信をすることができます。実際に相手をどこで指定するかというと、sendto の引数の中で「このホストのこのポートにつないでください」というのを指定して、そのまま送ってしまいます。何回か sendto、recvfrom でデータ転送を行った後、close するということになります。インターネット概要のところでも説明しましたが、sendto で

送って recvfrom で必ず読めるかというと、UDP の場合は必ずしも recvfrom で読めるとは限りません。つまり、途中で脱落したり、sendto で送った順番に必ずしも受け取れないこともあり得ます。そのために上位のアプリケーションの方でうまく再送して何回か送り直すとか、データの順序をチェックするということを行い、正しくデータ転送が行われるようにする必要があります。

アプリケーションインターフェース詳細（socket）

ここでは socket のインターフェースの詳細を説明します。

socket の生成

socket の生成については socket という関数があります。参考書とかマニュアルを読んでいただければいいかと思いますけれども、この関数の詳細を図７に示します。

socket の生成

■ 通信の口(socket)を作る

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

int socket(proto_family, type, proto);
int sd, proto_family, type, proto;
```

proto_family: プロトコルファミリ
PF_INET　インターネットドメイン
他に PF_UNIX, PF_ISO, PF_INET6 などがある

type:　通信のタイプ
SOCK_STREAM　ストリーム(PF_INET の場合，TCP)
SOCK_DGRAM データグラム(PF_INET の場合，UDP)

proto: プロトコル番号
0 にすると適当な番号に(システムが) 割り当ててくれる

sd:　socket 記述子

■ TCP の socket を生成

```
sd = socket(PF_INET, SOCK_STREAM, 0);
```

■ UDP の socket を生成

```
sd = socket(PF_INET, SOCK_DGRAM, 0);
```

図７ socket の生成

これからは C 言語で説明していきます。include ファイルを２つ用意します。戻り値が整数の関数 socket は、引数を３つ持ちます。socket が成功すると、図７の黄色いものはプロセスだと思ってください。これからの説明では、左側がクライアントで右側がサーバだとします。そうすると、白の四角で書いてある突起で表している socket というものができます。
引数の説明ですが、最初の引数はプロトコルファミリを指定します。今回の場合は、TCP/IP なので、PF_INET、インターネットドメインと呼ばれる定数ですが、これを指定します。アプリケーションインターフェース概要（**socke**t）でプロトコルに依存しないという話をしましたが、例えばここに PF_UNIX,、PF_ISO あるいは PF_INET ６（ＩＰｖ６）などほかのプロトコルファミリを書くこともできます。そうすると、別のプロトコルで socket をつくることができます。２番目の通信のタイプですが、ここまでストリームとかデータグラムということを説明しませんでしたが、ここでは PF_INET（IPv4）で TCP の場合はこの SOCK_STREAM というのを指定してください。UDP の場合は SOCK_DGRAM というものを指定してください。TCP の場合はこの引数ではトランスポートの種類を指定すると思ってください。最後にプロトコル番号ですけれども、説明すると長くなるためここでは省きますが、０にすると適当な番号に割り当ててくれます。つまり TCP とか UDP に対してプロトコル番号というのがそれぞれ決まっています。それを書かなければいけないんですけれども、TCP の場合ここに０と書いていると、システムのほうで勝手に TCP だったら何番というのを割り当ててくれます。逆に、ほかのプロトコルでは、私はよく知らないんですが、通信のタイプが SOCK_STREAM であっても複数プロトコルがあるような場合には、ここで適当な番号をユーザが指定しなければいけません。TCP とか UDP を使う分にはここは０にしていて構いません。sd というのは socket のシステムコールの戻り値です。TCP の socket をつくる場合の例を図７に載せています。UDP の場合は通信のタイプの指定が違っていて、SOCK_DGRAM と指定して呼び出してください。これで socket をつくることができました。

名前付け

次に名前付け、bind の話をします。この関数の詳細を図８に示します。

名前付け

■ socket に名前をつける
通信端点 (IPアドレス,トランスポート,ポート番号)

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

int bind(sd, name, namelen);
int sd, namelen;
struct sockaddr *name;
```

sd socket 記述子
name 名前へのポインタ (sockaddr_in 構造体へのポインタとなる)
namelen 名前の長さ

図８　名前付け

通信端点とは、IP アドレス、トランスポートおよびポート番号からなっています。

■ sockaddr_in 構造体
IPアドレスとポート番号
　トランスポートは，socket 生成時に指定済

```
/usr/include/netinet/in.h

struct sockaddr_in {
   u_char  sin_len;     /* 構造体の長さ */
   u_char  sin_family; /* PF_INET */
   u_short sin_port;   /* ポート番号 */
   struct  in_addr   sin_addr; /*アドレス*/
   char    sin_zero[8]; /*詰物*/
  };

  struct in_addr {
   u_long  s_addr;    /* アドレス */
  };
```

■ アドレス，ポート番号はネットワークバイトオーダで指定する

```
struct sockaddr_in server;
server.sin_len = sizeof(server);
server.sin_family = PF_INET;
server.sin_port = htons(80);          ……①
server.sin_addr = htonl(0x0a030201);/*10.3.2.1*/……②
```

■ アドレス、ポート番号
- アドレスに INADDR_ANY を指定すると，自ホストアドレスと解釈される
- ポート番号に 0 を指定するとシステムが適当なポート番号を割り当ててくれる
- 通常，クライアントは **bind**() を実行しなくてよく、システムが適当なポート番号を割り当ててくれる

図９ sockaddr_in **構造体**

socket **に名前を付けるとは、この通信端点に、図にアンダーラインを引いた** IP **アドレスとポート番号の２つを指定することです。なぜトランスポートは必要ないかといいますと、**socket **の生成で述べたように、既にトランスポートは** socket **をつくる段階で指定しているからで、**socket **を見ればシステムはトランスポートがどれかというのがわかるからです。そのため、ここでは指定しません。**bind **は、**IP **アドレスとポート番号をつけるためのシステムコールです。**bind **の引数は３つあり、**socket **でつくった** socket **記述子（ディスクリプタ）、名前および名前の長さです。「名前へのポインタ」とありますが、名前とは何かというと、具体的には** sockaddr_in **という構造体です。それへのポインタを渡します。長さというのは、**sockaddr_in **構造体の大きさを渡します。**sockaddr_in **構造体の詳細を図９に示します。**

この構造体について簡単に説明します。この構造体には IP **アドレスとポート番号が含まれています。**UNIX **だと** include **の** netinet **の下に** in.h **というファイルがありますが、ここに構造体の定義があります。それで、具体的には図のような構造体メンバーで構成されています。最初のフィールドは構造体の長さが入っています。２番目にフ**

ファミリが入っています。これは PF_INET と指定してください。３番目がポート番号です。４番目に、下に書いてある in_addr 構造体ですが、インターネットアドレスが入っています。最後は長さを合わせるために何か詰め物が入っている。こういった構造体です。実際には、これら３つのメンバーであるファミリ、ポート番号およびインターネットアドレスを設定する必要がありますが、注意しなければいけないのは、アドレスとかポート番号はネットワークバイトオーダで指定する必要があるということです。ネットワークバイトオーダは、そのまま値を入れるのではなく、ある関数を通して設定する必要があります。つまり、長さは sizeof をとって、ファミリは PF_INT、ポート番号は単に 80 と書くのではなくて、htons という関数を通してください（上図①)。それから、アドレスは、これはちょっとお行儀が悪いという方もいらっしゃると思いますが（私もそう思うんですけれども)、例えば htonl 関数を使って設定することができます(上図②)。これとは別な方法で設定することもできます。これはまたあとで説明します。こういった感じでこの構造体を設定します。それで、bind システムコールを呼ぶということになります。ただ、全部きっちり指定しなければいけないかというとそういうことはなく、例えばアドレスに INADDR_ANY という定数がありますが、これを設定すると、システムのほうでこれは自分のホストアドレスだと解釈して、システムが勝手に定義してくれます。だから、自分のアドレスはいくつだと知る必要はありません。２番目に、ポート番号に０を指定すると、やはりシステムが適当なポート番号を割り当ててくれます。サーバの場合これをしてはいけませんが（サーバは例えば自分が何番だというのはわかりますから)、クライアントの場合適当でいい場合は、０というふうに割り当てることができます。
前にも説明しましたが、通常クライアントは bind を実行しなくてよいということです。つまり、socket を実行して、connect を実行すると勝手にシステムのほうで自分のポート番号を割り当ててくれます。
以上で名前をつけることができました。

接続受理の準備

次はサーバ側の話で、接続受理の準備である listen について説明します。図 10 に接続受理の準備の詳細を示します。

接続受理の準備

■ サーバは接続要求受理の準備をおこなう

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

int listen(sd, maxqueue);
int sd, maxqueue;
```

sd　　socket 記述子
maxqueue 受信受付キューの長さ

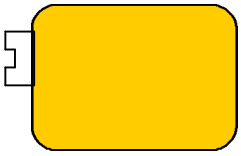


図 10 接続受理の準備

接続受理の準備である listen は、TCP の場合必要ですが、UDP の場合は必要ありません。listen は、サーバが実際にサービスを待ちますよというのをシステムに通知することです。通知されるとシステムは、サービスのための待ち行列をつくります。この待ち行列は何かというと、それは、listen のあと実際に接続要求が来たら accept というインターフェースで受け付けるのですが、その間に同時にいくつかリクエストが来た場合に受け付けるリクエストキューの長さのことをいいます。忙しいサーバ、例えば http など Web 関連の忙しいサーバでは、このリクエストキューの長さを少し大き目に設定します。そうでなければ、大体参考書などでは５とか書いています。実際にどのくらいの長さがよいかは実際に運用してみないとわからない部分もあるのですが、普通のテストプログラムなどあまり忙しくない場合には、５と書くことが多いです。listen は必ず必要です。

接続要求

listen が終わったら、今度はクライアントのほうの話になりまが、接続の要求をします。接続要求の詳細を図 11 に示します。

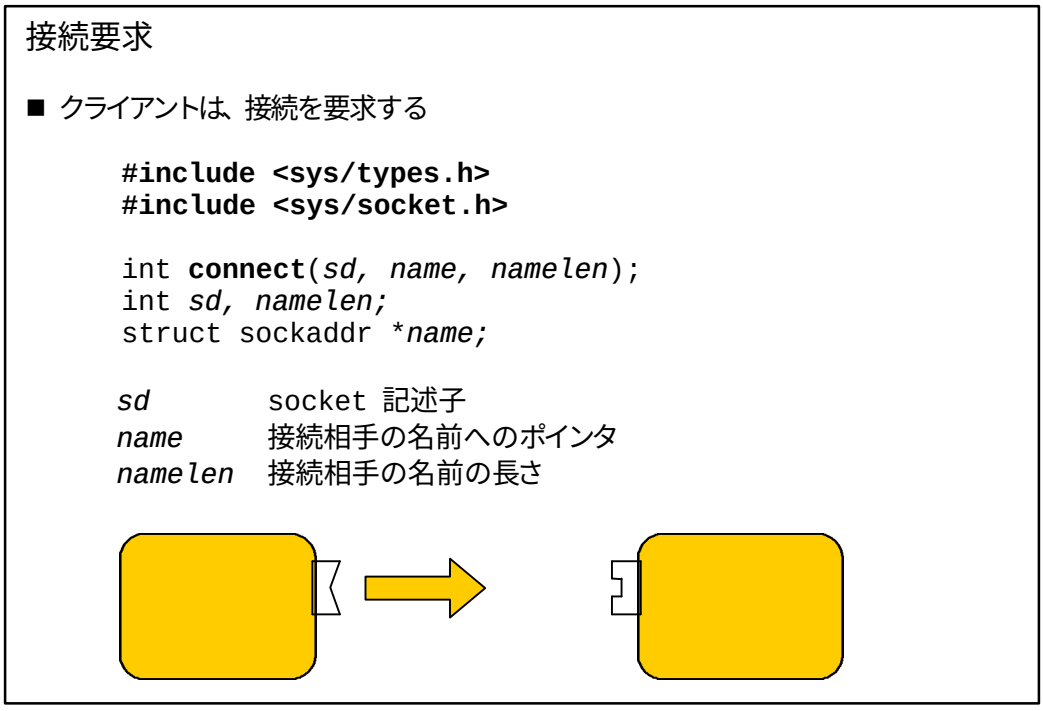


図 11 接続要求

接続要求のためには connect というインターフェースを呼びます。connect は引数を３つ取ります。socket の記述子と、前述した bind と同じですが、名前と名前の長さを渡します。bind のときと同じように、name がポイントする sockaddr 構造体に相手のアドレス、ポート番号およびファミリなどを指定して呼びます。

接続要求の受理

クライアント側の接続要求に対し、サーバ側は accept と呼ばれるインターフェースで接続要求を受け付けます。接続要求の詳細を図 12 に示します。

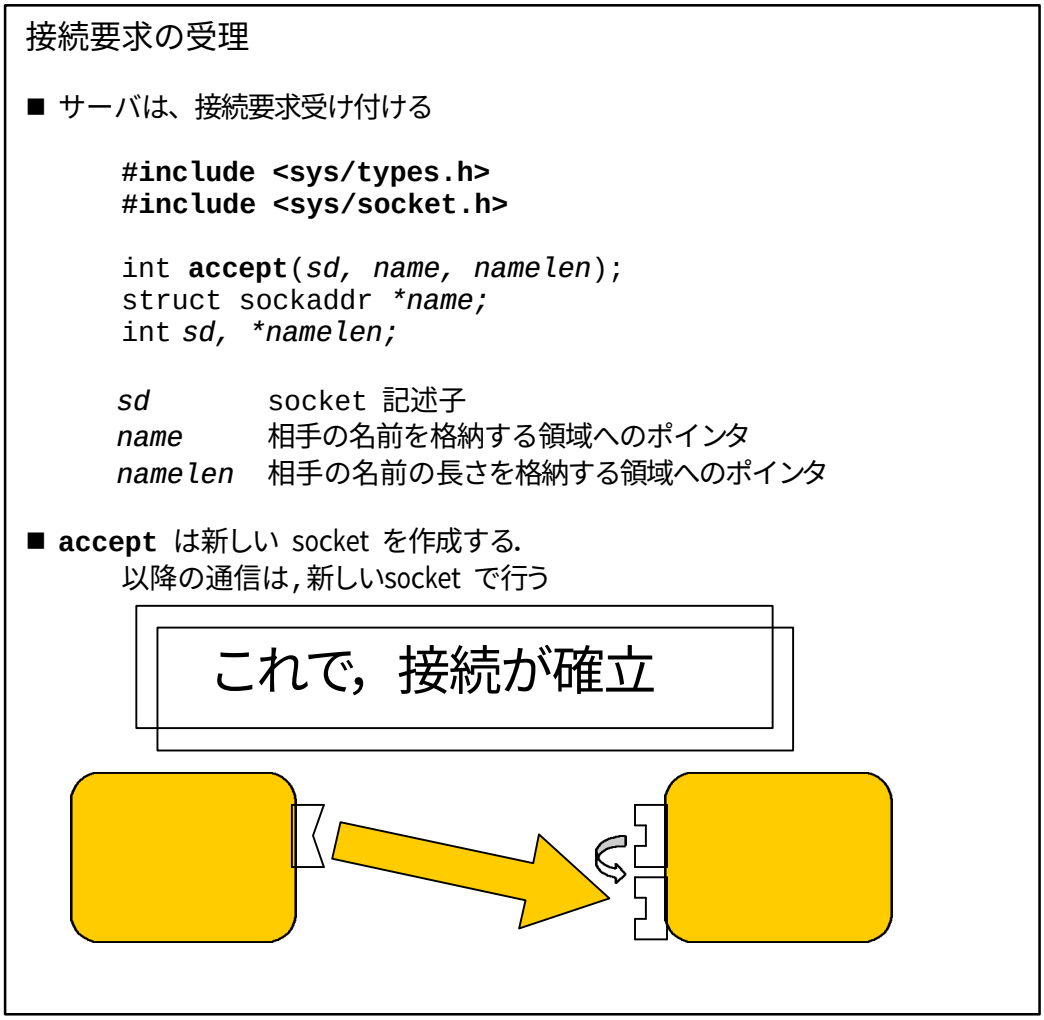


図 12 接続要求の受理

accept には引数が３つあります。socket 記述子と名前と名前の長さです。name に接続してきた相手のアドレスとかポート番号が入ります。それを格納するための sockaddr 構造体へのポインタと、その長さもポインタとして渡します。図 12 のように、今までは上の socket で受け付けてきたのが accept は下の新しい socket の記述子を返し､相手とはこの新しい socket でつながります(図 12 の矢印)。要するに上の socket は要求を受け付けるためだけに使われ、実際のデータ通信は下の新しい socket を使って行われます。以後上の socket はまた別のクライアントからの要求を受け付けるために引き続き使われることとなります。これで接続が確立されました。

データ転送

接続が確立されると、データ転送が始まります。データ転送については前述したように、UNIX の通常の入出力と同じインターフェース（関数）である、read、write を使ってもできますし、あるいは、recv、send といったネットワーク通信専用のインターフェースを使うこともできます。クライアント側は自分でつくった socket を使って

通信しますし、サーバ側は accept インターフェースがつくった（の戻り値の）socket を使って通信をします。最初に TCP に関するシステムコールインターフェースについて説明します。TCP に関するデータ転送の詳細を図 13 に示します。

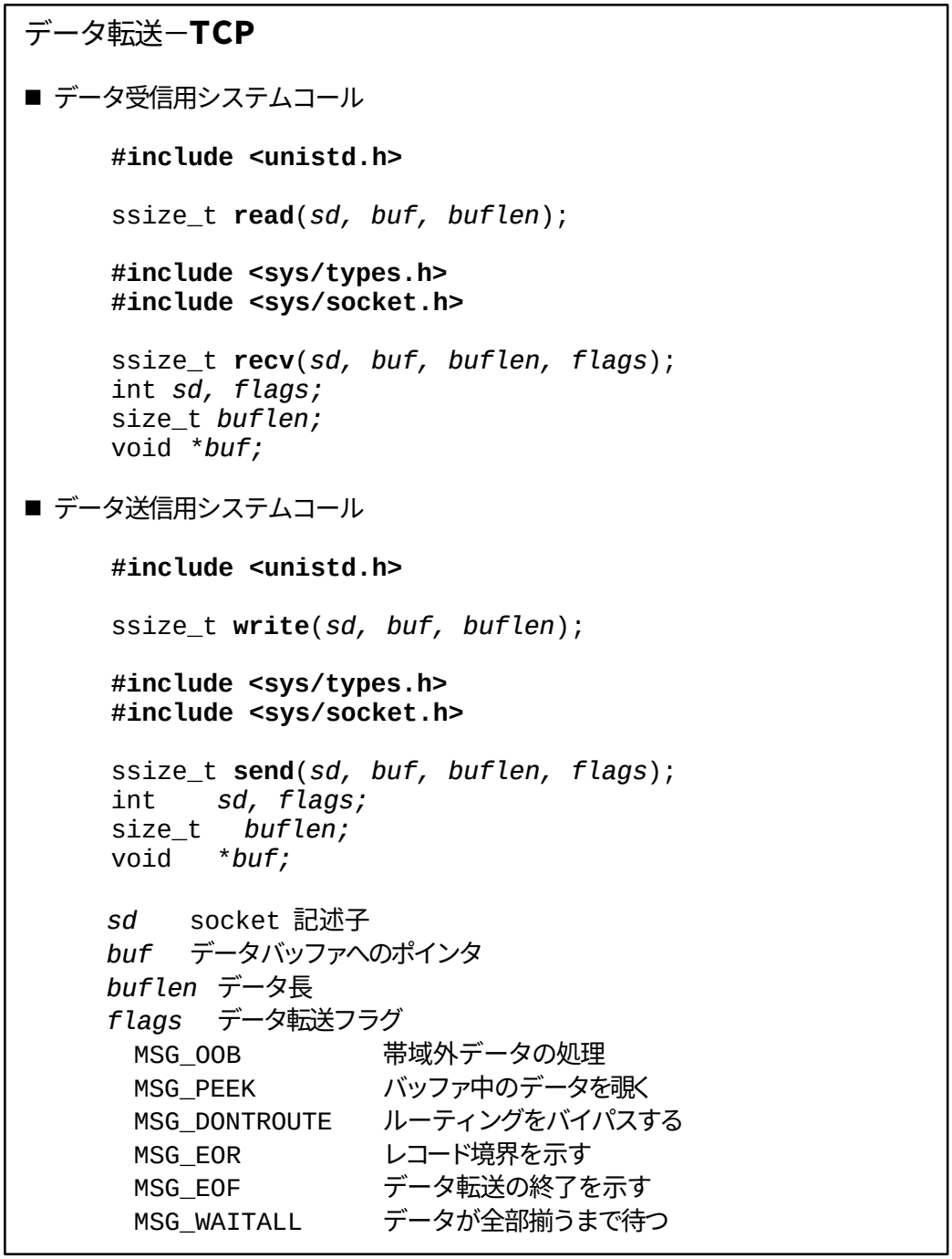


図 13 データ転送－ＴＣＰ

TCP でのデータ受信用システムコールの一つは read、もう一つは recv、この２つがあります。基本的にどちらもデータを受信するものですが、recv のほうが一つ引数が多く、もう少し細かい制御を行うことができます。それから、TCP でのデータ送信用システムコールとしては、受信用と同じように単純な送信だったら write でできます

し、何かちょっと複雑なことをしたければ send というものを使います。それぞれどういう引数があるかというのを簡単に説明しますと、sd、buf、buflen は、それぞれ socket 記述子、データバッファへのポインタ、データ長をそれぞれ表します。read、write はここまでですが、recv あるいは send に関してはデータ転送フラグというのを指定することができます。特にデータ転送フラグの説明に挙げていることを行う必要がなければ、ここは 0 に設定すれば read とか write と同じことになります。データ転送フラグもいくつか挙げましたが、全部説明する時間がないので省略しますが、例えば TCP だと実は帯域外データとか、緊急データというのを指定できます。そういった処理をしてほしいときは、このフラグを指定して細かい制御をすることができます。こういうことを特にする必要がなければ 0 にするか、または read、write を使えばよいわけです。
次に UDP に関するシステムコールインターフェースについて説明します。UDP に関するデータ転送の詳細を図 14 に示します。

データ転送－**UDP**

■ データ転送用システムコール

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

ssize_t sendto(sd, buf, buflen, flags, to, tolen);
ssize_t recvfrom(sd, buf, buflen, flags, from, fromlen);

int        sd, flags, tolen, *fromlen;
void       *buf;
ssize_t    buflen;
struct sockaddr *to, *from;
```

sd	socket記述子
buf	データバッファへのポインタ
to, from	相手の名前へのポインタ
flags	データ転送フラグ
buflen	データ転送長
tolen	名前の長さ
fromlen	名前の長さへのポインタ

図 14 データ転送－UDP

UDP の場合は簡単で、上の２つです。前述しました UDP の概要では、データ転送のところで socket をつくっていきなり送ったり受けたりしていましたが、これが sendto、recvfrom と呼ばれる２つのインターフェースです。引数には、socket、バッファ、バッファの長さと先程説明しましたデータ転送フラグがあります。それから、to（相手）、to（相手）の長さ。長さというのは sockaddr 構造体の長さです。先程の bind や connect で使った socket に名前をつけるための構造体がありましたが、それをそのまま使って一気に送ってしまいます。逆に、受け取るほうも recvfrom で待っていて、相手から sendto が来ると待ちが解け送信されたものが受け取れます。そのときに from と fromlen から相手のアドレスやポート番号がわかるようになっています。

切断

一通り終わった場合は、切断によってデータ転送を終了させます。手順としてはコネクションを切る、もう終わりますよという手続が必要となります。図 15 に切断に関するインターフェースの詳細を示します。

切断

■ 送受信の一方向，あるいは両方向のデータ転送の終了

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

int  shutdown(sd, how);
int  sd, how;

sd    socket 記述子
how   切断方法
      0 データの受信を終了する
      1 データの送信を終了する
      2 データの送受信を終了
```

■ 接続を切断し，ソケットを消滅

```
#include <unistd.h>

int close(sd);
int sd;
sd    socket 記述子
```

■ **shutdown** を利用せず **close** してよい

図 15 切断

まず shutdown というシステムコールインターフェースがあります。これにより、送受信の一方向あるいは両方向のデータ転送を終了させます。一方向というのはどういうことかといいますと、先程も説明しましたように TCP は全二重ですから双方向で通信していますから、ある瞬間からもう片方向は用済みだと認識すれば、例えば受信側はもう閉じていいですよということを教えることができます。それを行うのが shutdown です。引数は２つで、socket とどういう閉じ方をするか（切断方法）というのを指定します。切断方法の指定には 0、1、2 があり、それぞれ受信を終了する、送信を終了する、送受信を終了するという 3 つの終了方法の一つを指定します。shutdown するだけではだめで、そのあと UNIX のファイルの閉じ方と同じように、close というシステムコールを最後に呼ぶ必要があります。普通に使っている分にはあまり shutdown というのは使わずに（もちろん使ってもいいのですが）、いきなり close しても構いません。そうすると、そこでセッションが切れて問題なくコネクションをターミネート、終了させることができます。

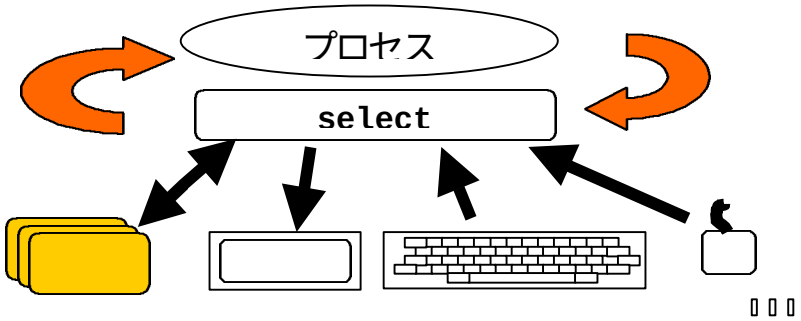
以上が一連のデータ転送の流れです。

入出力の多重化

ここから入出力の多重化ということで、別の話をします。accept、read、recv のシステムコールインターフェースは、データを受信するまでブロック、つまりそこでプログラムがとまってしまいます。しかし、複数の入出力を同時に見張りたいということはよくあるかと思います。例えば、２つ以上のポートを使って待つとか、キーボードの入出力を待ちながらネットワークからも待ちたいといったことがあります。そういった場合どうしたらよいか困ってしまうわけですが、これを解決するのに使うのが select と呼ばれるシステムコールインターフェースです。複数の何か、例えば、ネットワークの TCP や UDP のセッション、TTY（画面）への出力、あるいはマウスのイベント、こういった複数のイベントのうちどれかが発生するまで待ちたい、そういった場合 select によりそれが可能となります。この select システムコールは、複数のデバイスの入出力を待つためのインターフェースです。この詳細を図 16 に示します。

図 16 select

select



- 複数デバイスの入出力待ち

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

int select(nfds, rfds, wfds, efds, tout);
int nfds;
fd_set *rfds, *wfds, *efds;
struct timeval *tout;
```

nfds	調べるファイル記述子の数
rfds	入力用ファイル記述子の指定 (ポインタ)
wfds	出力用ファイル記述子の指定 (ポインタ)
efds	例外用ファイル記述子の指定 (ポインタ)
tout	タイムアウトの指定(ポインタ;NULL: 無限に待つ)
返値	入出力可能なファイル記述子数

- **FD_SETSIZE** マクロ
  システムがサポートする記述子の最大数
- ビット操作マクロ

FD_SET(*fd, &fdset*);	記述子 *fd* を *fdset* にセットする
FD_CLR(*fd, &fdset*);	記述子 *fd* を *fdset* からクリアする
FD_ISSET(*fd, &fdset*);	*fd* が *fdset* にセットされている場合, 0 以外を返し, セットされていない場合 0 を返す
FD_ZERO(*&fdset*);	記述子の集合を空にする

```
int fd;
fd_set fdset;
```

引数はたくさんありますが、まず最初の引数は、いくつのファイルディスクリプタ（ファイル記述子）を見張るのかを指定します。それから、そのあとの 3 つ、2 番目、3 番目、4 番目の引数は、それぞれ入力用、出力用および例外のファイルディスクリプタを指定し、どれを入力、出力として待つかおよびどれをエクセプション（例外）が起こった場合につかまえるかを指示します。この 3 つを使って同時に見張ります。最後に、タイムアウトが指定できて、ずっと無限に待ち続けたくなくて、例えば 10 秒に 1 回は別のことをしたいという場合にはタイムアウトを指定することができます。ここはポインタなので timeval と呼ばれる時間を指定するための構造体を渡し、ここに NULL を渡すと無限に待ちます。select を実行すると、指定したファイル記述子を見張り、どれかイベントというか、何かが入出力可能になると select から返ってきます。そして、そのときに返ってくる値が、実際に入出力可能なファイル記述子の数になります。例えば、キーボードとあるネットワークインターフェースの２つの受信を待っているとします。その時キーボードの入力があると、リターンして、ファイルディスクリプタが１個どれか入力が可能になったということがわかります。そして rfds の中にどのディスクリプタがアベーラブルになったかというのが書かれます。キーボードとネットワークからのパケットが同時に来たような場合には、select からの返り値は２になります。

便利なマクロを説明しますが、それは FD_SETSIZE と呼ばれるマクロです。このマクロは、図 16 に「システムがサポートする記述子の最大数」と書かれていますが、これを select の第１引数に指定してください。いくつ調べるかというところにはこれを入れておいて構いません。2 番目、3 番目、4 番目の引数、つまりどのファイルディスクリプタを見張っておいてほしいかという指定は、実はビットマップで指定します。例えば、select の２番目の引数のところにファイルディスクリプタの 1 番と 3 番を指定する場合は、その 2 番目の引数のビットマップの 1 番目と 3 番目のところに 1 を立てて渡すわけです。そのためにはビット操作が必要になりますが、それを行うためのマクロ、FD_SET と FD_CLR、そして何番目が立っているというのを調べるためのマクロ FD_ISSET があります。あと、ビットマップの配列は、fd_set で指定しますが、これをまず空にするために FD_ZERO のマクロが用意されています。

使用例を説明します。使用例としては、標準入力と socket の sd という変数で指定される 2 つの入力を待つというもので図 17 に示します。

SELECTシステムコール使用例

■ 標準入力と socket sd の入力を待つ

```
fd_set rd;
for (;;) {
    FD_ZERO(&rd);
    FD_SET(fileno(stdin), &rd);
    FD_SET(sd, &rd);
    select(FD_SETSIZE, &rd, NULL, NULL, NULL);
    if  (FD_ISSET(fileno(stdin), &rd)) {
      /* stdin の処理*/
    }
    if (FD_ISSET(sd, &rd)) {
      /* sd の処理*/
    }
```

図 17 select システムコール使用例

まず rd という先ほどのビットマップを宣言します。for 文の中に入り、それを zero にクリアします。それから、ビットを 2 つ立てるのですが、一つは標準入力で立て、それから sd のビットを立てます。これを立ててから、select のシステムコールを呼びます。引数は read だけで write とエクセプションは NULL と指定します。タイムアウトもなしです。上の例では select のエラー処理を省いていますが、select で待って、どちらかに入力があれば select から抜けていきます。そのあとで stdin（標準入）とか、socket を FD_ISSET というマクロで調べて、それぞれ処理を行うといったプログラムです。以上が select の使い方です。こういうように select を使うことによって複数の入出力を同時に見張ることができます。上の例では入力の場合だけを書きましたが、出力の場合も使うことができます。あとエクセプションというのはあまり使いませんが、前述した TCP の緊急データとかそういったときに挙がってきます。

クライアント・サーバモデル

次にクライアント・サーバモデルについて説明します。クライアント・サーバモデルというのは何かといいますと、ネットワークアプリケーションの基本的なモデルであるといえます。先程からサーバ、クライアントという言葉が頻出していますが、定義しますと、サーバとは、公示したサービスに対して要求を待ち、その要求者（クライアントのことですが）に対してサービスを提供するプログラム（プロセス）のことです。デーモンと呼ばれることもあります。クライアントとは、サーバに対してサービスを要求してサービスを受けるプログラムのことを言います。図 18 にサーバ・クライアントモデルの例を示します。

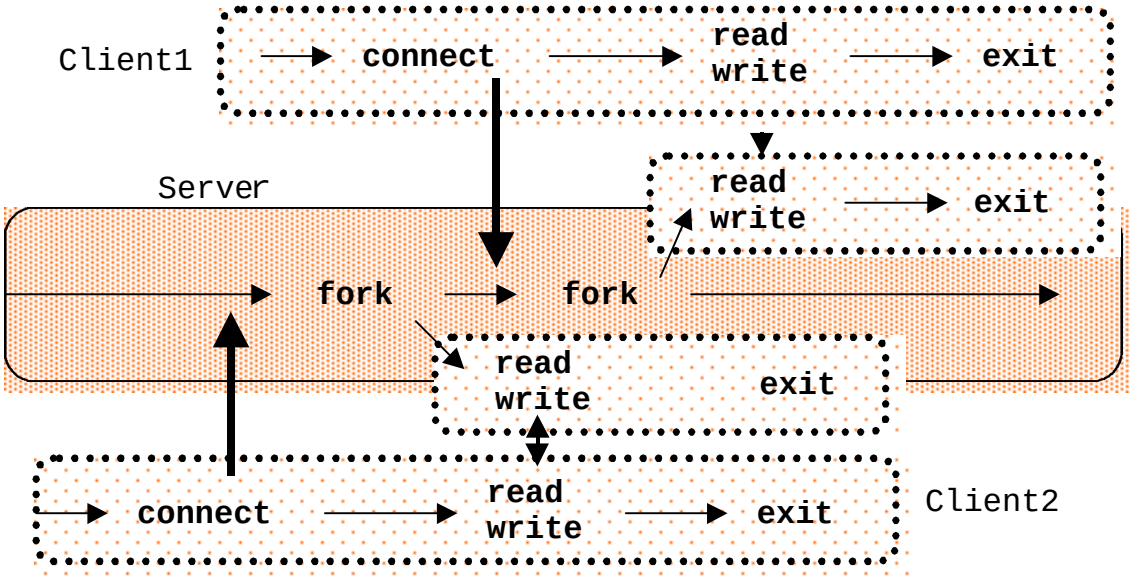


図 18 サーバ・クライアントモデルの例

真ん中にあるのがサーバです。一番上にあるのがクライアント 1 で、一番下にあるのがクライアント 2 です。fork と書いてあるのは UNIX の別プロセスを生成するためのシステムコールです。サーバは、ここには書いていませんが、始めに socket をつくって bind、listen して、accept で待っているという状態です。そうすると、クライアント 2 からまずリクエストが来ます。そうすると、サーバは accept から抜けます。サーバはその後どうするかというと、一つの例として fork というシステムコールを使って自分の分身をつくるわけです。分身はクライアント 2 と何かお話をして、終わったら抜けます。クライアント 2 に対して分身をつくって分身が処理している間に、サーバの本体は fork が終わったあとまた accept で待ちます。その後、クライアント 1 のほうからサービスを要求し、connect してきます。サーバは同じように fork を使って分けます。それで、また分身は read、write して exit する。つまり、一番大もとのサーバのほうは、ずっと要求を待っていて、何か要求が来たら分身を使って分身に作業をさせ、自分はまたひたすら要求を待ち続ける、そういうことを行います。
以上がサーバ・クライアントモデルの構成方法の例です。

では実際に UNIX でサーバ・クライアントがいかに行われているかについて説明します。図 19 に UNIX でのサーバを示します。

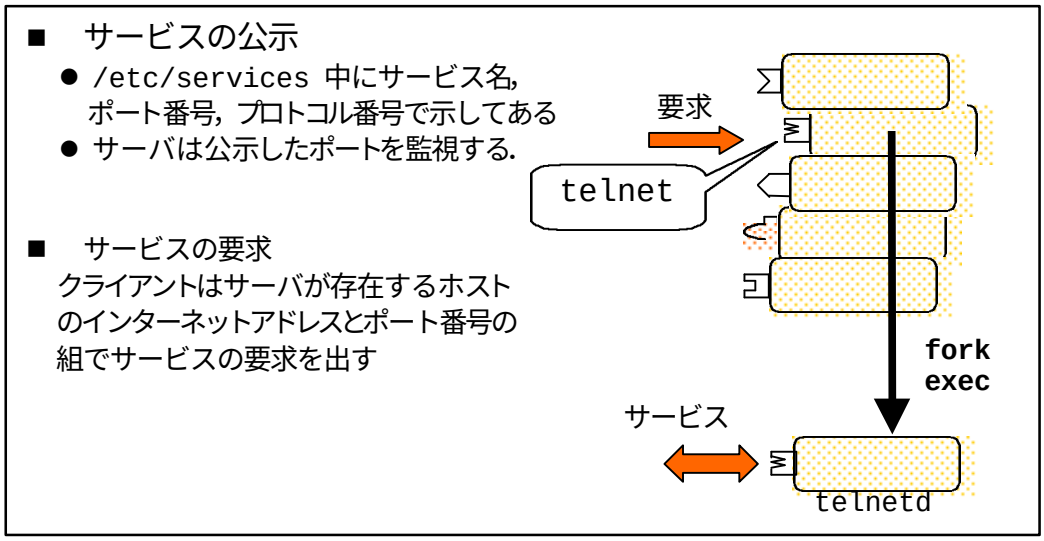


図 19 UNIX でのサーバ

先程公示すると言いましたが、まず/etc/services というファイルがあり、その中にサービス名、ポート番号およびプロトコル番号（TCP か UDP の区別）などが示してあります。そこの中のから自分は telnet サービスを行うといったら、ここのデータベースの中から telnet という文字列をキーにしてポート番号とかプロトコル番号を引いてきます。それで、telnet だと 21 番ですか、そのポートをつくって、bind で名前をつけて待っています。ほかにももちろん http とか smtp とかがあるのですが、こういったサーバが上図のように多数存在しています。だれかクライアントが connect しに来ると、fork exec で分身をつくって、そこでサービスを行います。

inetd

基本的にはこのようなサーバプログラムが多数存在しているわけですが、そうすると、多くのサービスを提供しなければいけないような場合には、プロセス（プログラム）が多数常に見張っていなくてはならなくなり、資源（その一つにメモリ）をむだにすることとなります。そこで inetd（スーパサーバと言われている）と呼ばれるポートを一括して管理してくれるようなプログラムが存在します。図 20 に inetd の概要を示します。

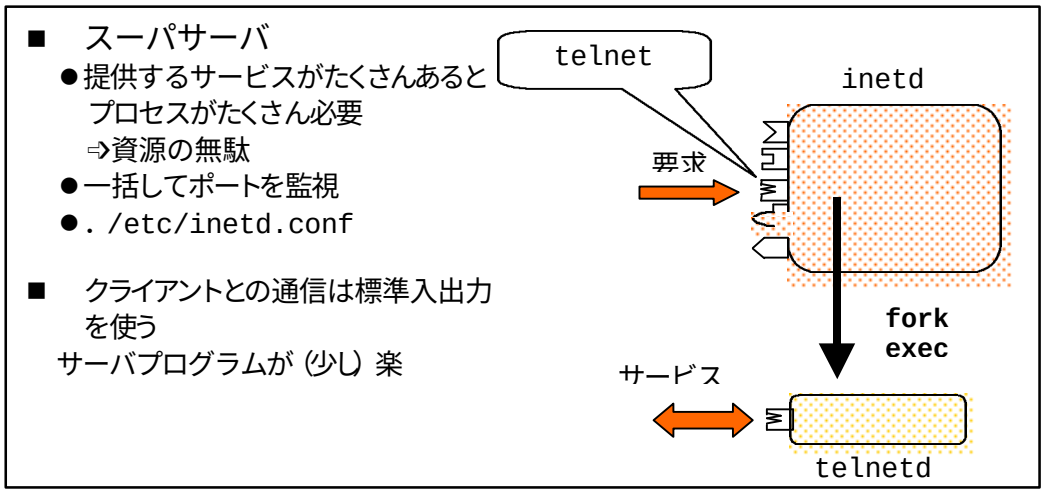


図 20 inetd

inetd は次のことを行います。上図のように大きな inetd という 1 個のプログラムが多数のポートを待っているわけです。つまり、プロセスとしては 1 つですが、ポートをたくさん持っており、それぞれ要求を待っています。例えばだれかから telnet に対して connect 要求があると、前述のように fork exec して、今度は telnetd という自分自身の分身ではなく telnetd という telnet のサービスを行うプロセスを起動・生成し、その telnetd が実際のサービスを行うという動作をします。これにより、一つのプロセスで複数のサービスを提供して、資源もそれほどむだに使わなくて済むようにできます。上図にある inetd.conf というのは、どのサービスを待つとか待たないとか、それを指定するための設定ファイルです。書き方についてはマニュアルを参照してください。そこで、実際にサービスを行うプログラムはクライアントとどうやって通信するかというと、それは標準入出力を使います。その訳は、サーバプログラムとしては前述した socket をつくって、bind、listen、accept してということを全部 inetd が行い、実際にサービスを行うプログラムに残された仕事は渡された socket ディスクリプタで通信するだけになっているからです。しかも、inetd が標準入出力を相手とのコネクションに割り当てるので、そのままいきなり read の 0 番とかいって読むことができます。そういった意味でサーバプログラムを若干楽につくることができます。そういった特徴があります。

添付資料に簡単なサーバの例として、daytimed.c というのを用意しています。基本的には今まで説明したことが書いてありますので、読んでいただければと思います。

ライブラリ

ここまでで一通りネットワークを介して通信するためのプログラムの枠組みは示しましたが、その他にユーティリティ、ライブラリがあります。これを使うと便利ですので、これについて説明しますが、内容は大きく 3 つに分かれます。最初に、「getXbyY」、Y から X を得るといった関数群がいくつかありますが、これについて少し説明します。次に、「インターネットアドレスの操作」に関するライブラリについて説明します。最後に、前述した「ネットワークバイトオーダを変換するためのライブラリ」について説明します。

gethostbyname、gethostbyaddr

まず getXbyY の代表的なもので、gethostbyname と gethostbyaddr と呼ばれるライブラリを紹介します。図 21 に概要を示します。

gethostbyname, gethostbyaddr

■ ホスト名⇒IP アドレス

```
#include <netdb.h>
struct hostent *gethostbyname(name);
char *name;
```

■ IPアドレス⇒ホスト名

```
#include <netdb.h>
struct hostent *gethostbyaddr(addr, len, type);
char *addr;
int len;
int type;
```

■ struct hostent

```
struct hostent {
  char  *h_name;    /*ホスト名*/
  char  **h_aliases;/*ホストの別名*/
  int   h_addrtype; /*アドレスタイプ(AF_INET)*/
  int   h_length;   /*アドレス長*/
  char  **h_addr_list; /*アドレスのリスト*/
};
```

struct hostent

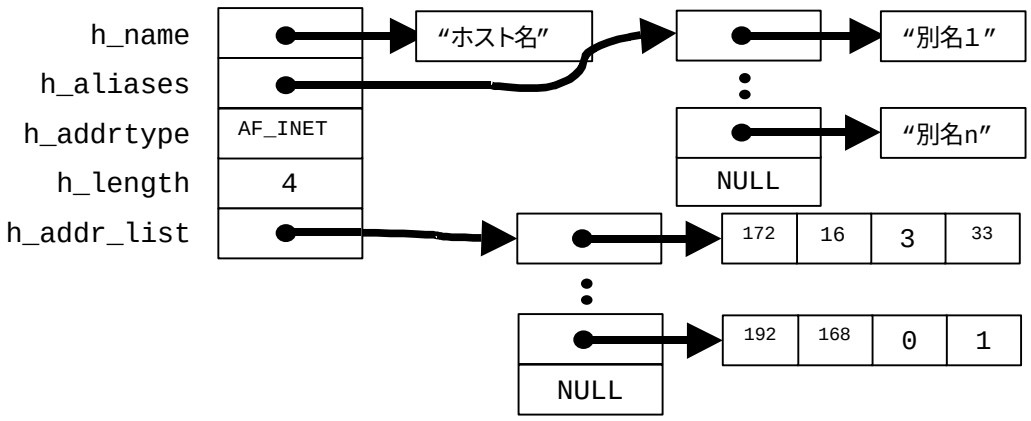


図 21gethostbyname、gethostbyaddr

まず gethostbyname は、上図にも示しているように、ホスト名から IP アドレスを

とってくるためのライブラリです。gethostbyname は多分一番よく使うと思うのですが、ホスト名の文字列を入れると、それに対応する IP アドレスが入った構造体（これはあとで説明します）を返します。逆に IP アドレスからホスト名を返すのが gethostbyaddr で、アドレスとアドレスの長さ、更にタイプを指定すると、やはり同じ構造体ですけれども、そのメンバーの中にホスト名が返ってきます。返ってくる構造体を hostent と呼んでいますが、上図のメンバーから構成されています。すなわち、ホスト名、ホストの別名（実はホストは複数名前を持つことができ、これはニックネームみたいなもので、別名を持っていれば、複数ここに返ってきます)、アドレスタイプ、アドレスの長さやアドレスのリストです。同じホストでも複数のインターフェースを持っている場合にはアドレスを複数持ちますから、それもアドレスのリストとして書くことができます。ポインタがいっぱいあり分かりづらいので、上図に示しました。aliases があれば、h_aliases ポインタの先にそれぞれ別名がたくさんつきます。アドレスタイプは、AF_INET という定数を書きます。アドレスの長さは、32 ビット、4 バイトですので 4 です。アドレスのリストは、面倒ですけれども、2 つポインタをかけると、アドレスの配列があり、そこに返ってきます。これは大抵 1 つしかないのですが、もし複数持つようでしたら、上図のように、複数わかるようになっています。

getnetbyname、getnetbyaddr

同様に、getnetbyname と getnetbyaddr というのもあります。これはホスト名ではなく、ネットワーク名からネットワークアドレスとか、ネットワークアドレスからネットワーク名への変換を行います。getnetbyname と getnetbyaddr の概要を図 22 に示しますが、ほとんど gethostbyname と同様ですので説明を割愛します。

図 22 getnetbyname、getnetbyaddr

getnetbyname,getnetbyaddr

■ ネットワーク名⇒ネットワークアドレス

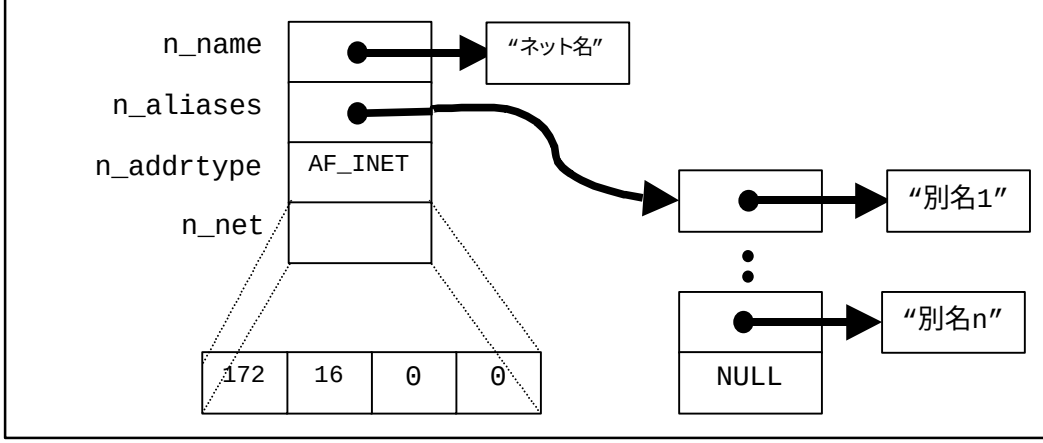
```
#include <netdb.h>
struct netent *getnetbyname(name);
char *name;
```

■ ネットワークアドレス⇒ネットワーク名

```
#include <netdb.h>
struct netent *getnetbyaddr(net, type);
long net;
int type;
```

■ struct netent

```
struct netent {
      char  *n_name;     /*ネットワーク名*/
      char  **n_aliases; /*別名のリスト*/
      int   n_addrtype;  /*アドレスタイプ*/
      unsigned long n_net; /*ネットワークアドレス*/
};
```



getprotobyname、getprotobynumber

次に getprotobyname、getprotobynumber について説明します。図 23 に概要を示します。

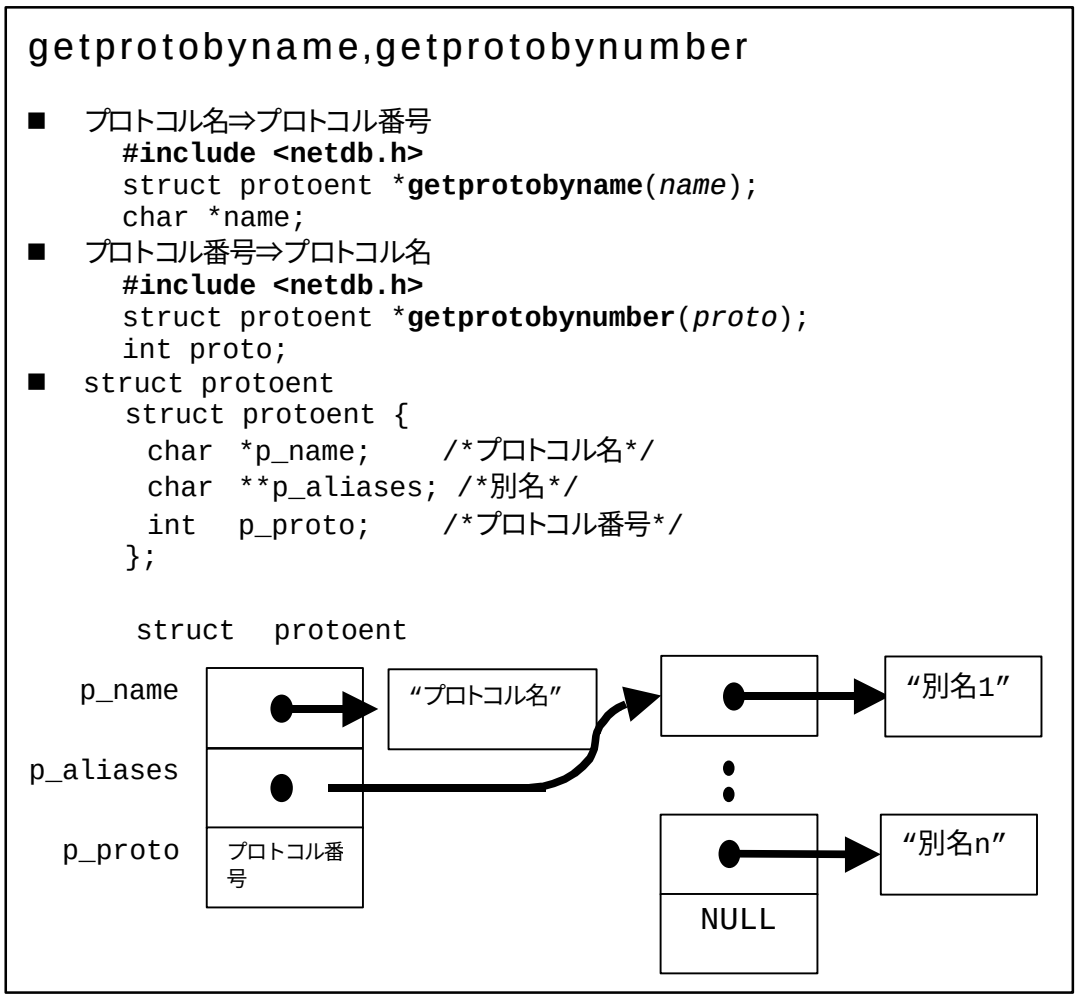


図 23　getprotobyname、getprotobynumber

最初に説明した socket のインターフェースのところで最後の引数は 0 でよいと言いましたが、実は厳密に言いますと、この関数で返ってきた値を socket の最後の引数に渡す必要があります。これは、そのための変換関数です。ここでは 0 でいいということであまり使いませんが、このような関数もあります。返ってくるのが protoent 構造体です。プロトコル名は、例えば TCP とか UDP というのを指定します。プロトコル番号は、5 番か 6 番か知りませんが、返ってきた番号を socket の最後の引数に渡します。

getservbyname、getservbyport

次に、getservbyname ともう一つ重要なのは getservbyport があります。それぞれ、サービス名からポート番号、ポート番号からサービス名へ変換するライブラリです。サービス名というのは、前述したように telnet や http といったサービスの名前です。サービス名から、25 番、80 番あるいは 21 番といったポート番号への変換およびその逆変換するためのライブラリです。詳細を図 24 に示します。

getservbyname,getservbyport

■ サービス名⇒ポート番号

```
#include <netdb.h>
struct servent *getservbyname(name, proto);
char *name;
char *proto;
```

■ ポート番号⇒サービス名

```
#include <netdb.h>
struct servent *getservbyport(port, proto);
int port;
char *proto;
```

■ struct servent

```
struct servent {
        char  *s_name;     /*サービス名*/
        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*/
};
```

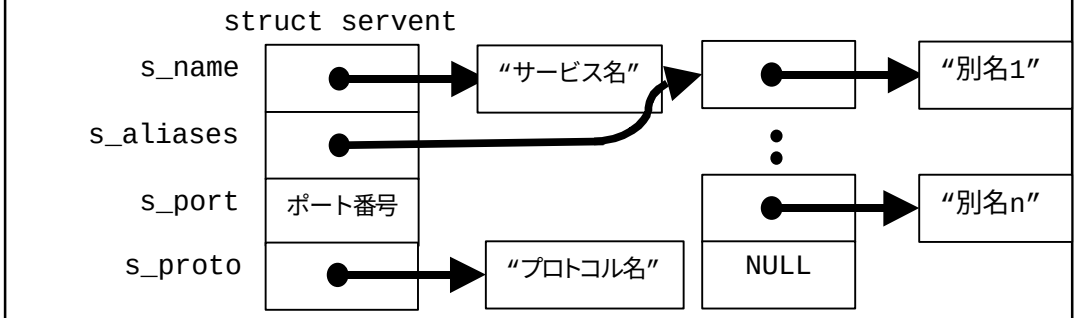


図 24 getservbyname、getservbyport

servent **がそれぞれの関数が返す構造体です。**getservbyname **の場合上図に示すようにサービス名とプロトコル名（**TCP **あるいは** UDP**）を指定すると、ポート番号が返ってきます。例えば, 引数のサービス名に** http **およびプロトコル名に** TCP **を指定して呼び出すと、上図の** servent **構造体のポート番号と書いているところに** 80 **番というのが返ってきます。**

インターネットアドレスの操作

次にインターネットアドレスの操作の関数をいくつかを紹介します。これはアドレス表現から IP アドレスへの変換とその逆変換を行うためのライブラリです。アドレス表現というのは、10.0.0.1 のような IP アドレスを文字列であらわすときの書き方のことを言います。この詳細を図 25 に示します。

図 25 インターネットアドレスの操作

インターネットアドレスの操作

■ アドレス表現⇒IPアドレス

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(str, addr);
char *str; /* eg: “10.0.0.1” */
struct in_addr *addr;

in_addr_t inet_addr(str);
char *str; /* eg: “10.0.0.1” */
```

■ IPアドレス⇒アドレス表現

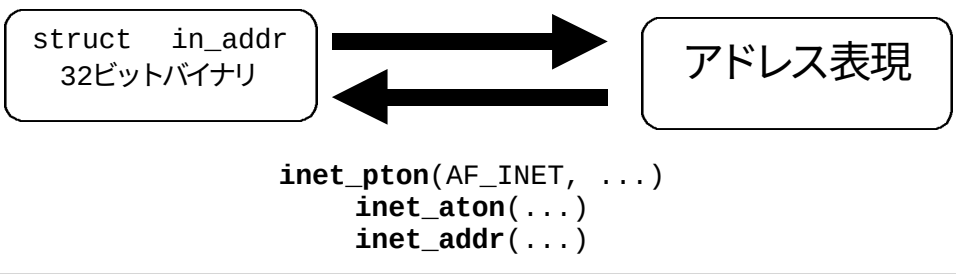
```
#include <sys/types.h>
#include <netine/in.h>
#include <arpa/inet.h>
char *inet_ntoa(addr);
struct in_addr *addr;
```

■ アドレス表現⇒IPアドレス

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_pton(family, str, addr)
int family;  /* AF_INET */
char *str;   /* eg: “10.0.0.1” */
void *addr;
```

■ IPアドレス⇒アドレス表現

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
char *inet_ntop(family, addr, str, len);
int family;  /* AF_INET */
void *addr;
char *str;   /* string */
size_t len;  /* length of str */
```



まず最初に、inet_aton と inet_addr の関数です。共にアドレス表現から IP アドレスへ変換します。最初に inet_aton ですが、これには引数が 2 つあり、最初の引数にアドレス表現を指定し、次に実際の IP アドレス（バイナリ表現）を格納するための構造体を指定します。2 つ目の inet_addr は、引数は 1 つで、アドレス表現を渡すといきなりそのまま関数のリターンバリューとして IP アドレスのバイナリ表現が返ってきます。どちらを使っても構いませんが、inet_aton を使う場合が多いようです。
次に上とは逆に IP アドレスからアドレス表現へ変換する inet_ntoa という関数です。これは in_addr 構造体（connect、bind でも使っていたと思います）で指定された IP アドレスをアドレス表現（文字列）に直すための関数です。
基本的には上の関数群で事足りるのですが、最近 inet_pton という関数、ライブラリができました。この関数もアドレス表現から IP アドレスに変換するものですが、異なる点は、引数にアドレスファミリ（またはプロトコルファミリ）を指定することができる点です。あとの 2 つの引数は inet_aton と同じです（addr の型が違いますが）。アドレスファミリを指定することができることによって、例えば IPv6 のアドレスを変換したいという場合には、PF_INET6 を指定することも可能となります。
同様に inet_pton の逆を行うのが inet_ntop で、これもアドレスファミリが加わっていて、IP アドレスからアドレス表現（文字列）に変換します。IPv4 で使う場合には、アドレスファミリは AF_INET とか PF_INET を指定すれば動きます。
まとめますと、上図の最も下の絵に示しているようになります。inet_ntop と inet_ntoa で 32 ビットのバイナリの表現から普通の ASCII のアドレス表現へ変換することができます。逆に ASCII のアドレス表現からバイナリ表現に変換するには、下に書いてある 3 つの関数、inet_addr、inet_aton および inet_pton のライブラリを使えば変換することができます。

バイトオーダ

最後にバイトオーダについて説明します。図 26 に図示します。

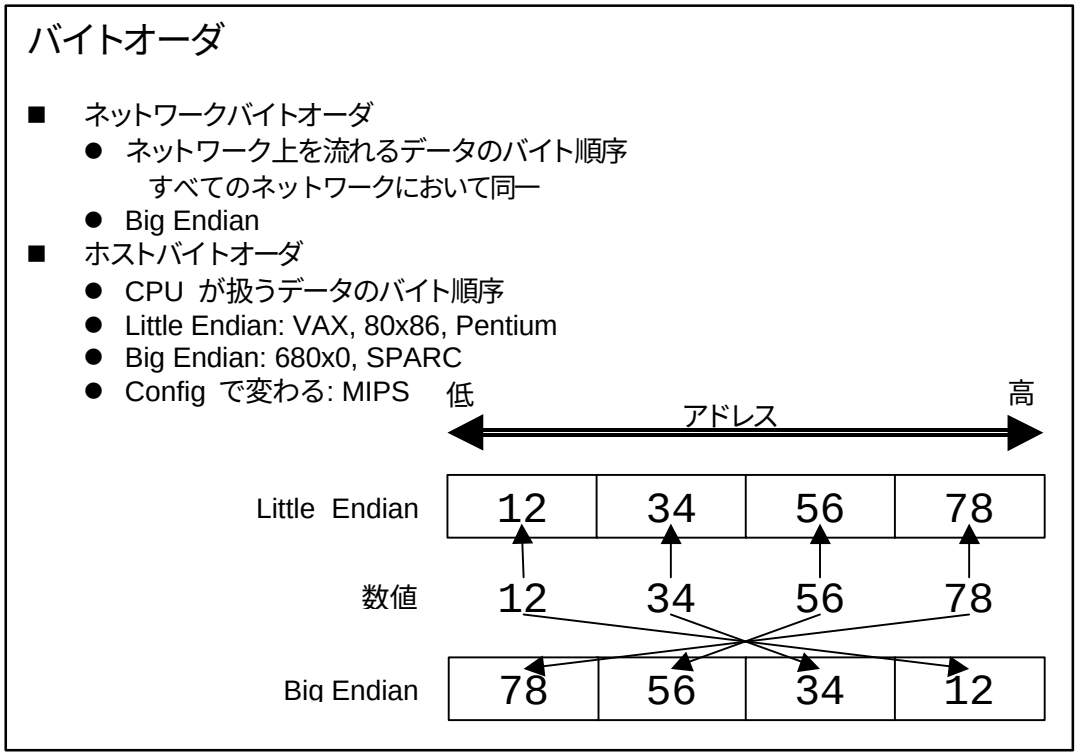


図 26 バイトオーダ

バイトオーダには、ネットワークバイトオーダとホストバイトオーダがあります。

バイトオーダには Big Endian と Little Endian の 2 つがあります（図 26 参照）。ネットワークバイトオーダは、ネットワーク上を流れるデータのバイト順序をいい Big Endian のほうを使います。なお、ネットワークバイトオーダはすべてのネットワークにおいて同一です。ホストバイトオーダは、CPU、ホストが扱うデータのバイト順序のことで、ネットワークバイトオーダと同じように Little Endian と Big Endian と２つの並び方があり、ホストによって異なります。MIPS と呼ばれるチップは config でどちらでも扱えるようになっています。例えば、16 進法の８桁の数字、12、34、56、78 という数値があって、それがメモリ中にどのように格納されるかに、図 26 に示すように 2 通りあります。図 26 にアドレスと書いてあるのがメモリ中のアドレスだと思ってください。低いほうから高いほうへ 0 番地、1 番地、2 番地、3 番地にこの数字を格納するときに、Big Endian と呼ばれるタイプのプロセッサ CPU では、一番上位の桁がアドレスの一番低いところに（図のようにひっくり返って）格納されるものをいいます。逆に、Little Endian では一番下の桁が一番小さいアドレスに格納されます。Endian は合わせておかないと、つまり Little Endian 同士あるいは同じ Endian 同士で話す分にはいいんですが、何の取り決めもなければ、受け取ったほうは別の解釈をすることとなります。例えば、Little Endian の人が Big Endian の人に 16 進法の 8 桁の数字、12、34、56、78 をそのままこの順序で相手に渡すと、向こうでは 78、56、34、12 という数字に解釈されてしまって、全く元とは違ったものになってしまうわけです。それでは困るので、ネットワークを通すときには必ずどっちかに統一しようということで Big Endian に統一することに決まりました。これをネットワークバイトオーダと呼ぼうということになりました。
バイトオーダを変換するための関数があります。図 27 に詳細を示します。

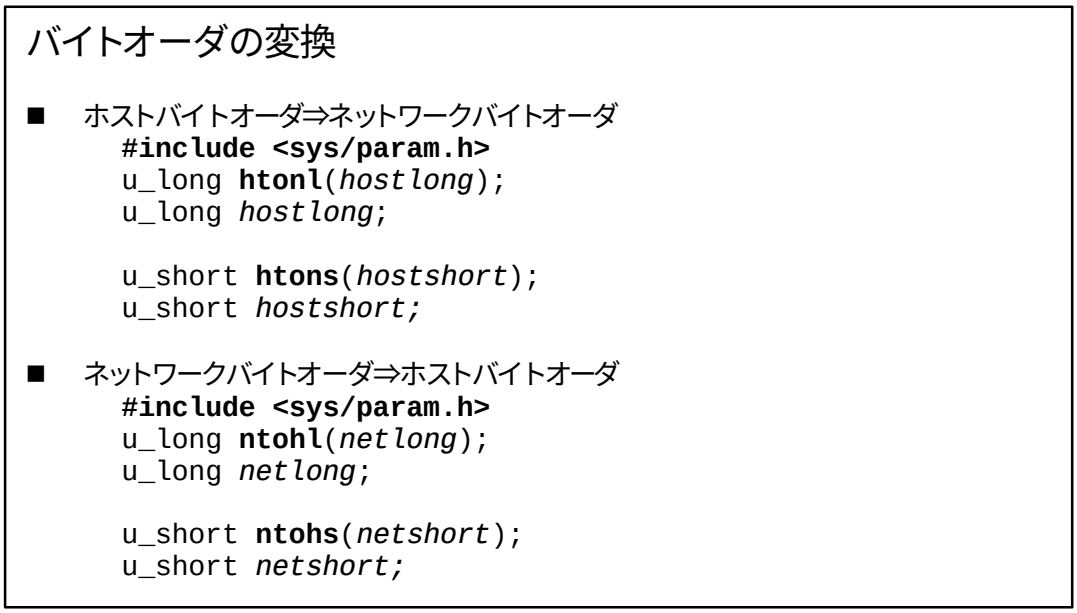


図 27 バイトオーダの変換

ホストバイトオーダからネットワークバイトオーダに変換するための関数に、htonl と htons があります。l と s は何をあらわしているかというと、図 26 では long４バイトの整数を例に挙げましたけれども、short２バイトの場合も同じ問題が起こりますので、l は long４バイトの変換、s は short２バイトの変換を行います。同様に逆のネットワークバイトオーダからホストバイトオーダに変換するための関数に、ntohl と ntohs と呼ばれる変換関数があります。

開発環境

最後に、開発環境としてコンパイルするときの話について軽く触れます。ライブラリの話ですが、BSD 系では特にライブラリの指定は不必要です。ただ BSD 系でも場合によってはレゾルブというネームサーバを使うためのライブラリが必要になる場合もあります。これを特に指定しなくてもいい場合もあるので、必ず必要か、必ずしも必要でないかというのは、各自実際に試されるときにマニュアル等をごらんください。基本的には特に必要ないはずですが。逆に、System V 系ではライブラリの指定が必要な場合があります。socket インターフェースのためのライブラリの指定、-lsocket が必要な場合もありますので、それもマニュアル等をごらんください。Linux ではどうかと言われるとちょっとわからないのですが、多分少なくともライブラリの指定は必要ではないと思います。ちょっと私はわかりません。
これまでは C で説明しましたけれども、perl と呼ばれるインタープリタがあります。これは UNIX でも動くし、この後説明する Windows でも動く結構お手軽な、かつそこそこ強力な処理系言語ですけれども、これを使ってプログラムを書くこともできます。

質疑応答

以上でインターネットの概要と UNIX 編のレクチャーを終わりますが、何かご質問があればお受けしたいと思います。

（質問 1）UDP では TCP に比べてエラー処理などしなければいけないけれども、そのかわり速いというようにおっしゃっていました。実際はエラー訂正などしないといけないわけですから結構大変だと思うんですけれども、大体どれくらい速くなるのかとか。あと、速くなることと簡単に使えるということ以外に、何か UDP のほうを使う利点などもしありましたら教えていただきたいんですけれども、
（石井）まずどのくらい大変かということですね。実際に計ったわけではないですけれども! TCP の場合は UNIX だとカーネルの中でいろいろな処理が行われます。カーネルの中だから重いというわけではないですけれども。ちょっと難しい……。具体的な数字はわからないですね。昔の感覚でいうと、１桁とか２桁という場合もありましたし。その辺はあまり詳しくは分かりません。2 番目の速いこと以外ですが、それはあとはユーザ側で自分のプロトコルを考えて作ることができる、という点です。例えば、TCP よりももう少し速いエラー訂正とか順序を保証するような、信頼性を上げるようなプロトコルを考えて、それをそのまま使うことができます。UDP を使って何か組み込むことができるという利点があります。要はユーザに全部開放するという意味で、UDP の U というのはユーザの意味ですけれども、そういう意味でユーザに開放してくれるというメリットがあり、それで自分で好きなプロトコルをつくることができるということが挙げられます。以上でよろしいでしょうか。

（質問 2）UNIX でのサーバのサービスの公示というところで、/etc/services のファイルがあります。こちらのほうにサーバの場合はポート番号をダブらないようなものを書いていくと思うんですが、これは必須なんでしょうか。要するに、処理系ではユーザアプリケーションでこれを守らないようなアプリを見たことがあるんですが。
（石井）/etc/services のファイルは、getservbyname というライブラリが参照するもので、いきなりポート 80 番とか書くのは別に構いません。何でこういうことをしているかというと、ホスト名と IP アドレスのところでも出ましたけれども、単に名前からポート番号を引くためだけに――だけということはないですけれども、そのためにあるということと、そういった名前とポート番号の関係をプログラムの外に出しておくことで、例えば万が一ポート番号が変わった場合にでもファイル１つ変えれば、アプリケーションプログラムを変えなくてよいという利点があると思います。もちろんプ

ログラムに直接埋め込んでもいいわけです。例えば http が 81 に決まったときに、getservbyname を使っているような場合ですと、ここのファイルを１つ書きかえればいいんですけれども、そうでないと全部リコンパイルしなければいけないということがありますので、なるべくこちらを使ったほうがよいと思います。

（質問 3）プロトコルファミリですけれども、後半のほうで AF_INET という言葉を使っていたのに対して、前半のほうでは PF_INET という言葉でしたが、これはどういう関係でしょうか。
（石井）そこは資料の不備です。基本的というか実際には同じ数字なんですね。PF_INET と AF_INET も一緒なので、こういう言い方をするとあまりよくないのですが、どっちを使っても構わないんです。統一がとれていないというか、基本的にはそれぞれ意味があって、マニュアル等をごらんになるとわかると思いますけれども、socket のところでは PF_INET を使ってくれ、こちらのほうでは、gethostby 何とかでは AF_INET を使ってくれというふうになっています。が、今の UNIX の実装では両方とも同じですので、あまり気にする必要はありません。ただ、将来変わってしまうかもしれませんの、マニュアルのとおりに指定していただければよいかと思います。

Windows 編（日比野）

Windows 編と Java 編のほうのネットワークプログラミングについてお話ししたいと思います。ネットワークプログラミングの基礎みたいな話は大体していただいたので、Windows でどうプログラムを書くかとか、Windows というのはプログラミングスタイルがかなり特殊なので、その辺も少し交えながらお話しします。

Windows の開発環境

Windows の開発環境として一番よく使われるものとして C++（もちろん C も含まれます）があります。Microsoft ということでベンダ依存になってしまいますが、まず Visual C++、これは Microsoft から出ています。それから、C++ Builder という、これは Imprise（昔 Borland といっていた会社です）のほうから出ています。あと何社から出ています。あと RAD（ラド）系と言われるもう少し簡単に使える簡易言語に Delphi（Delphi を簡易言語というと怒る人がいますが）とか Visual Basic、それからその他いろいろな言語が用意されています。後程説明しますが、もちろんこれらはすべてネットワークプログラミングが可能になっています。

Network API

Network API としては次のものがあります。

- Winsock DLL
- Winlnet
- Winsock Control
- Netscape,Internet Explorer

実際は API といいますか、我々アプリケーション屋さんとか皆さんみたいなプログラマが使うライブラリ集といったほうがいいと思うのですが、大体このの４つ（ぐらいほかにもいっぱいあります）があります。あとで詳しくは説明しますので、ここは単純に名前だけ紹介します。まず最初に Winsock DLL というもの。それから、WinInet。それから、Winsock Control（名前が似ていますが、一応最初の Winsock DLL とは違います）。それから、ちょっと意外だと思われるのですが、Netscape とか Internet Explorer みたいなウェブブラウザがアプリケーションから部品として使うことができます。

Windows のライブラリ

これららは実際の名前ですけれども、我々が使う API とかライブラリといっているものに対しいろいろな方法があります。次の 3 つぐらいあります。

- DLL
- COM/OCX
- DDE

DLL といっているもの、それから COM、名前でいうと OCX とか ActiveX という名前でも呼ばれます。それから、DDE というもの、大体 3 種類ぐらいの使い方というか決まりがあります。

最初にまず DLL ですが、これは Dynamic Link Library といい、拡張子が大体 DLL になっています。Windows の OS 自体もこの DLL が集合体となってつくられています。実際にはネットワーク以外の、例えばほかのカーネルとか GDI といったグラフィック周りだったり、そういうものはすべてこの DLL で構成されています。DLL は、名前が Dynamic Link Library ということで、例えば我々が何かつくったアプリケー

ションに共有化されたモジュールがあると思うのですが、それらを一固まりにして、みんなで共有化して使いましょうというのがこの Dynamic Link Library の目的です。ただ、これは、Windows の 3.0、Windows の最初から問題がありまして、バージョンの混乱ということが起こります。要は、例えばどこかのアプリケーションが A という DLL を使って動いていた。ところが、DLL がアップデートされてしまったり、アップデートしたのはいいんですけれども、アップデートしたことでバグが入ってしまった。そうすると、これまで動いていたアプリケーションが動かなくなってしまった、そういうことがよく起こります。実際、最近でいいますと、この間 Visual C（Microsoft の開発系）がバージョン 6.0 に上がりましたが、このときに同じように MFC42 という DLL がアップデートされました。その結果、これまで Visual C の 5.0 とか、Visual C の 4.2 を使っていたアプリケーションは動かなくなりかなり混乱を招いています。実際これは、あとで述べますが、Visual C の 6.0 サービスパックという要するにパッチが出まして、これで一応解決はしたようですけれども、この種の問題はしょっちゅう起こっています。、例えばほかには、Internet Explorer を入れるといろいろ変わってしまうとかいろいろ起こっています。ですから、この辺は実際に開発される方もデバグしたときには気をつけたほうがいいと思います。

それから、COM と OCX。これは同じものだと思っていただいて構わないのですが、最近では ActiveX Control と言われていると思います（よくここの会社は名前をよく変えるので私も覚え切れないのですが）。これは、基本的に Component Object Model というもので、アーキテクチャです。先ほどの DLL は基本的にライブラリ、CAPI と呼べるのですが、こちらのほうはまた別の呼び方を使います。これは、昔 OLE とか言っていたもので、最近はどちらかというと小さい部品（ソフトウエア部品は最近はやりですが）、そういうのを使うためにつくる手順を決めたものです。有名なものでは Internet Explorer の 4.0 などが実際に COM/OCX を実装して使用しています。

Winlnet

それから、WinInet。話がかなり前後してしまって申しわけないのですが、これは以下のネットワークプロトコルを実装・サポートする DLL です。HTTP――要するに Web です。それから、FTP（File Transfer Protocol）。それから、Gopher。最近は Gopher は聞きませんが、大体この３種をサポートしています。ですから、もしこの３種だけしか使わないのであれば、この WinInet を使うというのも一つの選択肢です。

Winsock Control

これは、先ほどお話しした COM（Component Object Model）を使った Control です。何でこういうものが必要かといいますと、Visual C などの低レベル言語では Winsock DLL とかを直接使えばいいわけですが、Basic や Visual Basic for Applications（VBA）、要は Excel、Word あるいは Access のようなものから DLL は非常に呼びにくい。呼ぶことはできるのですけれども、非常に面倒くさい問題がいろいろ発生します。例えば、ポインタをどうするんだとか、そんなものは Visual Basic では扱いにくいので、それを隠ぺいしてしまおうという目的で、この WinsockControl は使われています。ここには MS 系しか紹介していませんが、実際にはもちろん先ほどの Delphi やほかのベンダさんから出ている RAD（ラド）系のツールももちろん使うことができます。

Netscape、Internet Explorer

最後に Netscape とか Internet Explorer の話もしたいのですが、実際に Internet Explorer というのは製品名すなわち OS とかいいますが、実際は COM の集合体です。インストールされると分かりますが、実際の Internet Explorer の exe は非常に小さく数１０K しかありません。本当はどこに入っているんだというと、Windows のシステムディレクトリに非常に多くの OCX が登録されています。これらをアプリケーシ

ョンから使うことができます。例えば我々がアプリケーションを書いたときに、Internet Explorer を呼び出したり、また前の WinInet や Winsock Control を使って実際につくることができます。同じように Netscape も DDE という、またこれもプロトコルですけれども、これを使って使用することができます。実際我々がつくっている mailer も Internet Explorer を使って、例えば HTML の表示とかに Internet Explorer を使っています。Netscape に関しても同じように、DDE というのは手順を決めているだけで、こちらのほうは Netscape 社のほうから手順が公開されていますのでそのドキュメントを読んでいただければ非常にいろいろ使うことができます。

Windows のプログラミングスタイル

話がいきなりまたネットワークから離れてしまいますが、Windows のプログラミングスタイルということをお話しします。基本的に Windows は、OS の名前からも、まず Windows ありきというつくり方になります。すべてウィンドウという GUI が絡んできますので、メニューで書いてどこかで待っている、要するにブロッキングするような処理はまずつくれないと思ったほうがいいです。そういうわけで、基本的にメッセージドリブンなプログラミングスタイルになります。この辺の実際のソースはあとでサンプルソースのときにご説明したいと思いますが、メッセージドリブンでぐるぐる OS から投げてこられるメッセージを待っているというスタイルになります。ほとんどの場合、C で直接 Windows のプログラムを書くということはあまりありません。実際に面倒で、それにほとんど定型的な処理なので、実際はクラスライブラリと言われている C++のクラスライブラリ（Basic は Basic の中で隠されてしまっていますが)、例えば MFC（Microsoft Foundation Classes）という Microsoft がつくっているクラスライブラリが隠ぺいしてくれています。これと関連して、多くの API が実際のウィンドウを必要としています。例えば、ネットワークプログラミングのときに必要になってくる TIMER による一定時間ごとの通知も、ウィンドウにメッセージが投げられてきますので、ウィンドウがないと何もできないというのがほとんどの Windows API の特徴です。ですから、よくあるパターンであとで述べなければならないことですが、実際にはウィンドウなど表示したくないというときには、隠れたウィンドウ、要するに非表示なウィンドウを使ったりするようなちょっとした小細工が必要になります。

Winsock

Winsock（これは DLL のほうです）をメーンに以下説明します。ここでは一番ベーシックなところということで、Winsock DLL についてお話ししたいと思います。ほかの WinInet や Winsock Control などはマニュアルを読めばわかりますので、一応ベーシックなところをまず押さえていきたいと思います。Winsock は、先ほども述べましたように、これは略称ではなくて本当にこういう名前なのですが、Windows Sockets の略称というところから出ています。実際この Winsock 自体にはかなりの歴史があり、Windows 3.0 のころから実装されています。実際、これは Microsoft がつくったというわけではなく、その当時のネットワークベンダがいろいろありましたけれども、そのベンダと Microsoft が協力してつくり上げた API 集です。Windows 95 以降は OS に標準バンドルされたのですが、当時はまだ OS に標準バンドルされていずネットワークカードとかネットワーク TCP/IP の socket のパッケージとかで売られていました。逆に言うと、Windows 3.0 では何もできなくて、Winsock がついたネットワークソフトウエアを買っていたという構成になっていました。現在の最新バージョンは 2.0 になっています。DLL の名前も wsock32.dll になっています。ただ注意していただきたいのは、Windows 95（今も非常によく使われています）で使われているバージョンは 1.1 です。ですから、バージョン 2.0 （Windows NT などではバージョン 2.0 が使われています）に依存したようなプログラムを書こうとすると、Windows 95 ではそのままでは動きません。バージョン 2.0 は Microsoft の Web ページから Windows 95 用

のものがたしかダウンロードできますので、それをダウンロードしていただくとかそういう作業が必要になります。
winsock.dll の API の構造というか構成は、Berkeley-style の API（先ほど石井さんからご説明のあった Berkeley socket）がそのままほぼ移植されています。ですから、socket、connect、listen、accept、recv、send のような API はそのまま使うことができます。
さらに、Windows-style API（これは私が勝手に名前をつけたもの）すなわち Berkeley socket 以外の API が用意されています。これは先ほど説明しましたようにメッセージドリブンな Windows のプログラミングに合わせて追加された API です。ただ、基本的には先ほどの Berkeley socket API に対応するような形で大体拡張されてきています。多くの場合、頭に WSA（Windows Socket Async）という頭文字が追加されています。
それから、その他の API ということで初期化（DLL の場合、ほとんどは初期化をしなければいけない）の API とかエラーコードを取得するなど少しヘルパー的 API が追加されています。

初期化 API

まず簡単な初期化 API として、WSAStartup と WSACleanup があります。まず WSAStartup がやる仕事というのは DLL の初期化を行います。DLL の初期化はスレッドごとに行う必要があります。例えばマルチスレッドを使いたいような場合は、スレッドごとに 1 回初期化をしなければいけません。忘れてはいけないのは、バージョン 2.0、バージョン 1.1 という２種類のバージョンがありますが、ここでバージョンチェックができますから確実にここでバージョンチェックをしていただくというです。
それから、プロセスが終わったからといって何もしないで終わってしまうといろいろリソースの開放が漏れてあとで動かなくなってしまったりすることがありますので、WSACleanup という、要するに winsock.dll の解放処理をしなければいけません。実際にサンプルでも書いていますが、これを大体 winmain という関数の後ろぐらいにつけておけばいいと思います。

Berkeley-styleAPI

これは Berkeley-style の API ということで、UNIX 版とほとんど同じです。ただ、一つ注意していただきたいというか困った点があります。それは、UNIX ですと例えば標準出力、標準入力のようなものと同じように、socket に対する入出力もファイルを使った入出力が使えるわけですが、Windows ではそういうことができない点です。ですから、read、write のような関数は使えないので、その下のレベルといいますか、recv とか send のようなものを使わなければなりません。

Windows-styleAPI

一番面倒といいますか、Windows の特徴的な API があります。Windows-style の API。基本的に、先ほど述べたように Windows というのはメッセージドリブンなので、どこかで例えば先ほどの話で recv などを呼ぶとブロックするということはまず困るわけです。なぜかというと、例えば Window の描画がとまったりといったことが起こるからです。それはプログラム上非常にまずいわけです。そこで、それを防ぐためにどうしたらいいかということで考えられたのが、この Windows-style の API です。大抵 API の名前は頭に WSAAsync とつきます。例として一番よく使われる、絶対使われる次に示す、WSAAsyncSelect という API があります。

```
int WSAAsyncSelect(SOCKET s, HWND hWnd, UINT wMsg, long lEvent);
```
イベントマスク(lEvent)で指定したイベントが発生すると、ウインドウ(hWnd)にメッセージwMsgを送信する。

パラメータとしては socket ディスクリプタの s、hWnd、wMsg、lEvent という４つのパラメータを渡しますが、意味的にはイベントマスクにどういうイベントが欲しいか、すなわち繋がったとか、何かデータが来たとか、何かデータを送りたい（データを送りたいというのはあまりないかもしれませんが）など、どういうイベントを自分が受け取りたいかというのを指定し、そのイベントが発生したときに、そのイベントを指定した hWnd（ウィンドウ）のほうに通知してもらう、要するにメッセージを送ってもらうということをお願いするわけです。ここで 1 回 WSAAsyncSelect を出した後、上記のイベントが発生すると、指定したウィンドウにそれぞれイベントを通知するメッセージが送られてきます。そのメッセージを受け取り、Windows のほうがそれを処理していくという流れになります。

Hangarian notation

また少し寄り道をしますが、先ほど hWnd とか lEvent とか変な頭文字がついていました。Windows のコーディングスタイルの話なのでここではあまり関係ないのですが、API とか見ているとわけがわからなくなるので少し説明します。Hungarian notation といって（ハンガリアン記法とか日本語ではいうことが多いと思います）、先頭のほうに型をつけます。例えば hWnd は HANDLE なので hWnd。それから、long 値だと l をつけたり、pointer だったら p をつけたり、C++ですが参照などは r をつけたりというふうに、その変数が見たところですぐどういう型かというのが一目でわかるようにするというのが大体約束になっています。ですから、Windows のプログラムを書くときは、この辺を気をつけてみると割合読みやすいかと思います。それから、クラスのメンバーには m_とかをつけることが一般的です。実際にはこれを守っていないものもありますが、多くの場合は m_hWnd とか m_nSize ような感じで付けていきます。特にこれは MFC のようなクラスライブラリでは顕著に使われている例です。

イベントとWSAAsyncSelect

WSAAsyncSelect のイベントを以下に一部列挙します。

- FD_ACCEPT　　接続確認通知
- FD_CLOSE　　ソケットが閉じられたときの通知
- FD_CONNECT　　接続結果通知
- FD_OOB　　帯域外のデータ到達通知
- FD_WRITE　　書き込み準備完了通知
- FD_READ　　読み込み準備完了通知

大体上の６種類ぐらいのイベントが挙がってきます。ほとんど先ほどの select のイベントに対応しています。しかし、select のイベントが呼んだらもうすぐに戻ってこない（タイムアウトしたら戻ってきますが）のに対し、これはを呼んでもプログラム自体にはすぐ戻ってきますので、その間にほかの処理を行うというプログラミングができます。もう一度同じことを繰り返しますが、WSAAsyncSelect というのは select の非同期版、それから通知が Windows の Message になっているというところが select との大きな違いです。

WSAAsyncGetXtByY、WSACancelAsyncRequest

名前引きの関数がいろいろあります。Berkeley socket に対応したものが以下のように全部揃っています。

- WSAAsyncGetHostByAddr (gethostbyaddr)
- WSAAsyncGetHostByName (gethostbyname)
- WSAAsyncGetServByName (getservbyname)
- WSAAsyncGetProtoByName (getprotobyname)
- WSAAsyncGetProtoByNumber (getprotobynumber)
- WSAAsyncGetServByName (getservbyname)
- WSAAsyncGetServByPort (getservbyport)

これも WSAAsyncSelect と同じように、この関数を呼び出してもすぐ処理が戻ってきます。名前引きとかが完了すると Windows のほうに通知が発生します。すなわち非同期に動作させて Windows の処理をとめないということになっています。大体同じパラメータなので個々の使い方の説明は省略しますが、例として以下の１個だけ示します。

```
HANDLE WSAAsyncGetHostByName
    HWND hWnd, // window handle
    unsigned int wMsg,    // message
    const char * name,    // [in]host name
    char * buf,    // [out] HOSTENT
    int buflen //    length of buf
    );
```

最初の引数は、通知がほしいウィンドウ。それから、どういうメッセージが欲しいか。これはメッセージの種類によっていろいろ分けなければいけませんので、自分が勝手に使える範囲のメッセージ（メッセージはアンサリング・ウィンドウで示されます)、つまり自分の使える範囲がありますので、その範囲から使うということになります。
それから、host name、HOSTENT、char *buf の長さです。
上に挙げた GetHostByAddr などを使いますと、どうしても DNS に名前を聞きに行ったりしますので、場合によってはすごく時間がかかったり、失敗するのに時間がか

かったりします。その時ユーザがキャンセルボタンを押してキャンセルさせたいということがあると思いますが、そのために WSACancelAsyncRequest という関数が用意されていて、コールすると先ほどの API コールはキャンセルされます。それによって応答性が速くなるということになります。注意していただきたいのは、WSAAsyncGetXtByY 以外の非同期系 API はキャンセルできません。例えば、WSAAsyncSelect などをキャンセルしたいと思ってもできません。そのときは単純に切ってしまって構わないと思います。

エラー処理

エラー処理ということで、WSAGetLastErro と WSASetLastError があります。これは名前どおりエラーコードを取得するものとエラーコードをセットするものです。ほとんどの場合、自分でエラーコードを設定することはないと思います。これらが用意されているのは Winsock DLL を実際につくるベンダさんのためだと思いますが、これを使って一応設定することが可能です。
エラーコードは UNIX とは異なりますが、値自体は同じです。ただ、エラーコードを定義している（winsock.h で定義）シンボル自体、WSAE というのが先頭についていますので、UNIX から移植するときはこの辺を注意して置きかえないといけないことがあります。

例：CSocket

少し例を挙げます。私も実際に使っていますが、CSocket というクラスがあります。これは先ほども出ましたが Microsoft のフレームワークである Microsoft Foundation Classes という C++用のクラスライブラリで実装されているクラスです。これは CSocket とはいうものの、実際は CAsyncSocket と CSocket という２種類のクラスがあります。CAsyncSocket がベースで CSocket が派生したクラスです。ソースは示しませんが、Visual C++をインストールするとこのあたりのソースは全部ついてきますので、Visual C++を持っている方はソースを見ていただきたいと思います。ポイントは次のとおりです。先ほど説明しましたようにすべて非同期系の API 自体がウィンドウを必要とするので非表示のウィンドウを１個つくっています。要するに、CSocket というものをつくりますと、実際は MFC の中ではその CSocket に対応した見えない１個のウィンドウをつくります。そのウィンドウに対して先ほどのいろいろな通知メッセージを送ってもらい、いろいろな処理をしていくということをしています。これと似た処理として先ほど説明しました WM_TIMER なども同じようなことをよくします。だいたいこのからくりは、ほかのケースでもよく使い、一応 Windows のプログラミングとしては常道ですので覚えておいて損はないと思います。

CAsyncSocket

Csocket のベースクラスである CAsyncSocket ですが、こちらのほうはほとんど先ほどの非同期の API を軽くラップしただけのクラスです。実際にコールバック関数というのものが用意されていて、先ほどの隠れたウィンドウを持っています。イベントを全部このクラスでとることができます。すなわち、このクラスに先ほどの OnAccept/OnClose、OnConncet/OnOutOfBandData および OnRecieve/OnSend などのコールバック関数が用意されていますので、例えばデータを受信するには OnRecieve のところに受信したときのコードつまり受信したデータを処理するコードを書いていくという流れになります。もちろん、これは全て非同期型なので、例えばフローチャート的に 1 から 2、3、4 のような順番的なコーディングはできません。ある程度状態を持ったコーディングスタイルになると思います。

CSocket

CAsyncSocket から派生した CSocket というクラスがあります。こちらのほうは同期型です。先ほどの CAsyncSocket のほうが非同期型で、呼んでもすぐ戻ってくるので実際はコールバック関数で処理をしなければいけないというプログラミングスタイルに対して、Csocket のほうは一回例えば connect というのを呼ぶと、connect するまで戻ってきません。そのため、フローチャート的な順番にシーケンスを持っているプログラムを書く場合はこちらのほうが非常に楽になります。そうするとウィンドウの表示とかはどうなるのかという疑問が多分発生し先ほどの話と少し矛盾しているんじゃないかと思われるかも知れませんが、ウィンドウの再表示といった処理は、CSocket のライブラリの中で処理しています。ですから、通信中は画面の表示だけは何とかしてくれますが、表示以外は全て止まってしまうことになります。
それから、CArchive との連動ということがあります。先ほど UNIX ではファイルと socket が仮想化されて read/write の呼び出しを使うことができ、Windows ではできないという話をしましたが、それに対応する形で socket ライブラリにより、その仮想化の仕組みが一応 MFC には用意されています。これが CArchive というものです。CArchive を使うことで、ファイル、socket の通信、メモリの中のデータなどを全て仮想化して１個のストリームとして扱うことができるようになっています。このため、これを連動させることで比較的デバイスに依存しないコーディングができるようになっています。

問題点 1

さて、よい話はここまででやはりいろいろ問題があります。ここで挙げる問題点はあくまで MFC の Csocket についてです。まず、Windows ではマルチスレッドがほとんど必須に近いのですが、スレッド間で Csocket や CasyncSocket といったオブジェクトを受け渡しすることはできません。MFC を作った人たちの言い分はパフォーマンスを上げるためだということなので、これは「ああ、そうですか」ということで終わってしまい仕方ありません。これを回避するために Attach/Detach といって、CSocket が持っている SOCKET ハンドル（先ほどの SOCKET ディスクリプタ）を付けたり離したりしてやっていくことが可能です。SOCKET ディスクリプタに関してはスレッド間で受け渡すことができ問題はないのですが、しかしあくまでこの MFC がつくっている socket クラスのオブジェクトはスレッド間で渡すことはできないという制限は残ります。以下はコーディングの例です。

```
SOCKET hSocket;
CSocket socket;
socket.Attach(hSocket);
…
hSocket = socket.Detach();
```

socket の Attach をして何か処理をして、socket の Detach をしてハンドルを返してもらい、またどこかに渡すという処理が必要になります。

問題点 2

2 番目の問題点ですが、これは私にとってはかなり深刻な問題でした。それは、Visual C の 6.0 ではマルチスレッドで CSocket を使おうとすると落ちることです。これについては Microsoft さんからもサポート情報として流れていたのですが、単純に Visual C の 5.0 とか 4.2 を使ったプログラムを普通に移行するだけできれいに落ちてくれました。これに関する情報を以下に示します。

http://support.microsoft.com/support/kb/articles/q193/1/01.asp
Q193101 BUG: Unhandled Exception Using MFC Sockets in Visual C++ 6.0
対策も書いてあります。
SP1(Service Pack 1)を待ちましょう。
...と思ったら直っていなかった。

上に示しているようにその対策方法も書いてあり、それを対策すれば動くということは確認しました。私がこれを書いているのは少し前だったので、Service Pack 1、要するにパッチの１では直っているだろうなと思ったのですが、上の最後の行に書きましたが直っていませんでした。それで、MS のことですから、バージョン３までいけばきっと直ると思いますので、Service Pack 3 まで我慢してくださいということになります。ちなみに弊社、オレンジソフトのほうでは、Visual C の 6.0 には移行していません。何回か Service Pack 1 とかも挑戦してみたのですが、かなりこの辺は非常にコアなところです。そこで、socket といったオブジェクトの管理あたりのコアなところをいろいろソースを見てみました。見た目はあまり変わっていないのですが、実際に中のメモリ管理だったり、オブジェクトの管理方法はかなり変わっていました。そのあたりで CSocket というものがおかしくなってしまうというところを見ていると、かなり心配になってしまい、現状は私自身も Service Pack 3 まで待っているところです。

問題点 3

先ほど説明したように AsyncSocket とか CSocket というのは非同期の API を使うことで GUI がとまらないでうまく動くという話しをしたのですが、１個だけ同期の API が使われています。これは私も理由はよくわからないですが、gethostbyaddr で使われています。ほとんどの場合、これはすぐ返ってくるのでいいだろうという判断だと思うのですが、一応これが嫌だという方は、connect0 という用意されているメソッドを使わないで、自分で書いてしまうのがいいと思います。connect を自分で書くのはそれほど難しいことではありません。先ほどのように派生させて書けばできると思います。

開発スタイル

開発スタイルとして、今お話しした Berkeley-style と Windows-style という大きく分けて２種類の API ファミリといいますか API のグループがあり、それとマルチスレッドを使うか、シングルスレッドで書くかといった２つのプログラミングスタイルの組み合わせでプログラミングのやり方、設計のやり方が考えられます。以下に４つ単純に組み合わせて示します。

ａ．.Berkeley-style API+シングルスレッド
ｂ．Windows-style API+シングルスレッド
ｃ．Berkeley-style API +マルチスレッド
ｄ．Windows-style API +マルチスレッド

書籍等ではb,dを勧めることが多いが、私はcがお勧め。

1 番目が Berkeley-style の API を使ってシングルスレッドで書きましょう。それから 2 番目が、Windows-style の API を使ってシングルスレッドで書きましょう。それから Berkeley-style の API を使ってマルチスレッドで書きましょう。それから、最後は Windows-style の API を使ってマルチスレッドで書きましょうと。本当に 4 種類組み合わせました。私が変なのですが、大体多くの Windows のネットワークの書籍ですと、大体は b と d、要するに Windows-style の API を使えと書いている書籍がほとんどです。逆に言うと、こういうふうに書かないと変なやつだとかというのが結構多いのですが、私としては逆に c、Berkeley-style の API＋マルチスレッドが比較的お勧めできます。もちろんいろいろ実装するアプリケーションによっては、b とか d を使わなければいけない場合もありますが、多くの場合、特に smtp、pop、imap などキャラクタベースといいますか、そういう対話的なプロトコルに関してですと、c のスタイルで十分書けると思います。

Berkeley-style のAPI＋シングルスレッド

Berkeley-style の API＋シングルスレッドは、ブロッキングしてしまいますから、ウィンドウが止まります。したがって基本的にはこれは使えません。しかし、Windows 95 とか Windows 98、Windows NT のコンソールアプリケーションの作成にこれを使うことができます。ウィンドウのないアプリケーションに対してはこの組み合わせを採用することは可能で、Berkeley-style の API とシングルスレッドで書けばいいと思います。

Windows-style のAPI＋シングルスレッド

それから 2 番目の Windows-style の API＋シングルスレッドですが、これは当然 Windows-style、非同期 API を使っていますのでブロッキングしません。従って、表示が止まってしまうようなことも起こりません。ただ、例えば複数のセッション、pop だったら pop がたくさんアカウントを持っていて、こちらを受けて、あちらを受けて、またこちらを受けるようなことをやっていくとわけがわからなくなってしまいますから、そういうことを始めるとかなり面倒なプログラムになる傾向があります。もちろん要領よくうまく作ればすっきり書くことができますけれども、傾向としては複雑になりがちす。それから、これは非常に問題だと思いますが、UNIX などのソースの移植がすごく面倒になります。UNIX 系ではやはり順番にシーケンスを流すようにコーディングするのが当たり前ですが、Windows-style ではイベントドリブンでウィンドウのメッセージごとにハンドラを書いていくというスタイルになります。そうすると、ほとんど全面的な書き直しに近いようなことになってしまいます。当然書き直しをするということはバグも入りますし、設計もやり直しということになりますので（もちろんいろいろな場合がありますけれども）、基本的に面倒ということになってしまうと思われます。

Berkeley-style のAPI＋マルチスレッド

先ほどお勧めと言った Berkeley-style＋マルチスレッドです。まずバックグラウンドで通信スレッドを動作させ、つまりバックグラウンドで勝手に通信させ、フォアグラウンドであるメーンのスレッドのほうでウィンドウ処理に専念させるというスタイルです。要するに、通信の処理と画面表示・操作を分離してしまうというものです。これによって、先ほどのブロッキングの問題は起こりませんし、UNIX からの移植が面倒ということもありません。実際、UNIX からの移植はとても簡単だと思います。もちろん read とか write のようなところは書き直さなければいけませんが、それ以外は問題が起こりません。実際に OpenLDAP（オープンソースです）や PGP などではソースが UNIX と Windows で共有化されており、多分今説明したスタイルで書いているということになります。

ただ、マルチスレッド特有のプログラミングの面倒さはあります。マルチスレッドというのは、つくるのは簡単ですけれどもデバグが大変です。非同期にスレッドが動いていますし、共有メモリとかを使いますので、共有メモリへの書き方の順番が間違っているとおかしいことが起こるとか、親のスレッドが死んでしまったのに子供のスレッドが生きていてそれがゾンビみたいになってしまうとか、いろいろ面倒なことが発生します。ですから、通信のところは簡単になるのですが、逆にスレッド特有の問題が発生してくるので、両者でトレードオフの関係が発生することになります。

それから、Windows 3.1 にはマルチスレッドはありませんので使えません。Windows 3.1 は勘定に入れないほうがいいと思いますが、どうしても無視できない人とか、仕事でやらなければいけない人は、先ほどの Windows-style で書かなければいけないということになってしまいます。

Windows-style のAPI＋マルチスレッド

これは最後の組み合わせで、Windows-style で書いてマルチスレッドで見せるというものです。これは当然マルチスレッドになっているので Windows プログラムの構造は比較的シンプルになります。それから、Csocket を使えば、Csocket は非同期系 API を使っていますので、結果的にマルチスレッドになります。ただ、わざわざスレッドを分けて、それからウィンドウをつくってといった処理をしなければいけないようなアプリケーションはあまりないのではないかと思います。

以上で Windows 関係の話を終わらせていただきます。

質疑応答

（質問 1）UNIX 系で育ったもので fork をついつい使いたいと思うんですけれども、Windows にはありませんね?

（日比野）ええ。ありません。特にマルチプロセス、プロセスを上げればいいというのが UNIX の方の考えだと思うんですけれども、当然その辺の継承関係がすごく面倒です。ですから、ほとんど実質的に使えないと思ったほうがいいと思います。それから、子プロセスの終了ががわかりません。わからないというのはうそですけれども、わかりにくいわけです。だから、あまり使えません。スレッドに変えるのがいいと思います。

（質問 2）socket を共有しようというのははなから考えてはいけないと?

（日比野）僕はあまりやりませんす。ですから、その辺はスレッドにしてしまいましょうというのが多くの考え方だと思います。

Java 編（日比野）

次に、Java についての話をします。Java についてはあまり書いていません。というのは、ネットワークと Java は非常に親和性が高く、Windows の話と比べ、特にこういう最初の部分の話をするところがあまりないからです。

概要

Java は標準で socket をサポートしています。ネットワークプログラミングとして socket をサポートしています。しかし、文字列の扱い方に注意しなければいけないということがあります。socket の標準サポートということで、Java では、java.net.Socket と java.net.ServerSocket（これはサーバサイド側の socket です）という大体 2 種類のクラス（ほかにも多数ありますが）を使うことになります。

プログラミングスタイル

Java のプログラミングスタイルは先ほどの Windows と違って、同期型です。処理が完了するまで呼び出しは戻ってきません。ですから、当然 Java でも例えば Applet を使う、要するに画面を使うときには同期型を呼んでしまうと画面が止まってしまいます。ですから、ではマルチスレッドを組み合わせて使うというのが必須になります。ただ、幸いマルチスレッドも Java では標準でサポートされていますので、共有メモリなどいろいろ面倒な話はありますが、C++とか先ほどの Windows の世界と比べると比較的簡単だと思います。

文字列（String）

Java の文字列は、内部的には Unicode を使っています。簡単に説明しますと、16 ビットのワードで表現されています。そうすると、あとで説明しますが、smtp だったり pop みたいなのは byte 列で流れてきますので、Unicode と byte 列の相互変換が必要になります。というわけで、Java の世界と他の世界すなわちメールの世界、Web の世界あるいは他のいろいろな世界と通信するためには、Unicode と byte の変換が必要になってきます。もちろんこれらの変換ロジックは Java が標準に持っています。String クラスの中の getBytes というメソッドです。使用例を以下に示します。

```
String#getBytes(String enc)
String command = “RETR 1¥r¥n”;
out_stream.write(command.getBytes());
```

上の“RETR 1…”というのは pop のコマンドで、これをコマンドとして流したいときは、out_stream（出力するための Stream）で write するときは byte 列に変換して流し込まなければいけません。これを忘れてしまいますと、Unicode で先ほど言ったように 16 ビットですから 0 が入ることになるわけです。ですから、実際は例えば a はヘキサで 41 だと思いますが、そうすると Unicode では 0041 となり、これは C の世界でいうと NULL 文字が入りおかしくなってしまいます。

接続

接続の方法ですが、これは非常に簡単です。Java の文法は説明しませんが、socket クラスをホスト名（例えばオレンジソフトなら mail.oangesoft.co.jp）とポートナンバーを入れて生成すれば、socket が生成され、先ほどの socket で接続まで行ってくれます。以下のとおりです。

```
int POP3_PORT = 110;
Socket socket = new Socket(“hostname”, POP3_PORT);
```

読み込み／書き込み

読み込み／書き込みについては、UNIX 系と同じようにファイルと socket などの入出力関係は 1 個に仮想化されています。そういうわけで、InputStream とか OutputStream を使うことで、１個の byte 列として処理することができます。以下にコーディング例を示します。

```
import java.io.*;
… // 接続処理
BufferedInputStream in_stream = BufferedInputStream(socket. getInputStream());
BufferedOutputStream out_stream = BufferedOutputStream (socket.
getOutputStream());
int ch = in_stream.read();
out_stream.write(ch);
import java.io.*;
… // 接続処理
BufferedInputReader reader = BufferedInputReader(new InputStreamReader(socket.
getInputStream()));
BufferedOutputWriter writer = BufferedOutputWriter(new OutputStreamWriter(socket.
getOutputStream()));
int ch = reader.read();
int ch = reader.write();
```

上図の socket.getInputStream あるいは socket.getOutputStream というメソッドを呼び出すことで、socket に対応した Stream（入力あるいは出力の byte の並び）のオブジェクトを取得することができます。BufferedInputStream あるいは BufferedOutputStream というのは何かというと、名前で想像がつくかもしれませんが、バッファリングしてくれるものです。バッファリングをしないと、1 文字 send したら 1 文字ネットワークに出てしまいますので、ネットワーク的に非常に効率が悪いことになってしまいます。また、読み込みのほうも、まとめて読み込んだバッファからデータを読み込んで、データがなくなったらまたネットワークから取り出すという処理をしてくれます。そのためこの BufferedInputStream あるいは BufferedOutputStream を使うことによって、ある程度バッファリングしてまとめて読み込みあるいは送り出しができます。読み込み、書き込みのほうは単純で、in_stream.read、out_stream.write でできます。これらの戻り値はインテジャで、また read でデータがないと－1 が返ってきます。これは UNIX 系の名残みたいなものです。

以下は読み込み／書き込みの２番目の方法です。

```
import java.io.*;
… // 接続処理
BufferedInputReader reader = BufferedInputReader(new InputStreamReader(socket.
getInputStream()));
BufferedOutputWriter writer = BufferedOutputWriter(new OutputStreamWriter(socket.
getOutputStream()));
int ch = reader.read();
int ch = reader.write();
```

先ほどは stream というのを使いましたが、これは、もう一つの方法の Writer/Reader 系を使います。これは JDK の 1.1 位（定かなバージョンは忘れてしまいましたが）から入ってきています。こちらも同じような使い方で、BufferedInputReader、BufferedOutputWriter というバッファリングをしたもの、プラス InputStreamReader、

OutputStreamWriter を使って入出力をおこなうことができます。これを使うと、文字列の変換（キャラクタセットの変換）を自動的におこないますが、少し面倒といいますか問題があり、両方使わなければいけないことが多いと思います。（これについてはあとでお話しします）

Reader(Writer)とStream

私自身特に mailer をつくっていることもあって特にメッセージ系に限ったことではないのですが、、先ほどの Reader、Writer でよく気がつく、問題だと思う点があります。それは次のとおりです。メールというのは実際に本文の中には日本語が入っていたり、ハングルが入っていたり、中国語が入っていたり、ヨーロッパ圏の latin1 とか latin2 みたいなキャラクタがいろいろ入ってきます。そういう状況では、ずっと同じキャラクタが来るわけではないので Reader 系とか Writer 系の自動変換は実質的には使えないわけです。ではどうすればいいんだということになってくるんですが、先ほどの Stream を使いまして、ただの byte 列として 1 回読み込んでしまいます。読み込んで、それをあとでメッセージを解析して実際にそれに対応したキャラクタセットで変換する（少々面倒くさいですが）ことが必要になります。これはメール、すなわち RFC821 や MIME メッセージみたいなものに加えて、例えば http などでも当然 MIME を使っていますので、同じようなことが多分発生していると思います。ただ、http に関していいますと、Java が標準で持っていますので、その辺は隠されていると思います。メールに関していいますと、SMTP と IMAP については拡張の API が Sun から出ていますが、POP などではありませんので、やはりこの辺を意識して処理する必要があると思います。

Java 編は以上です。次に、実際のサンプル、それから実際にどういうふうに作成するかということを紹介します。

実例紹介

まず、サンプルプログラムは以下の通りです。

- SMTP
  SIMPLE MAIL TRANSFER PROTOCOL
  RFC821
- POP3
  POST OFFICE PROTOCOL
  RFC1939

SMTP と POP というメールに関するプロトコルですけれども、SMTP は送る側（本当は送る側だけではないですが）、Simple Mail Transfer Protocol、RFC でいうと 821 というところで規定されているプロトコルです。それから、POP3 は最近非常に皆さんもメールでお使いだと思いますが、Post Office Protocol というプロトコルです。こちらのほうは RFC の 1939 で定義されています。多くのインターネットのプロトコルに関しては全て RFC というもので定義されていますので、メールをつくりたいとか、例えば HTTP の Web ブラウザをつくりたいとかという場合は、こういう RFC という文献がありますので、大体全て英語ですがこちらのほうを参照してください。

SMTP

まず、SMTP の大体の流れを図 28 に示します。

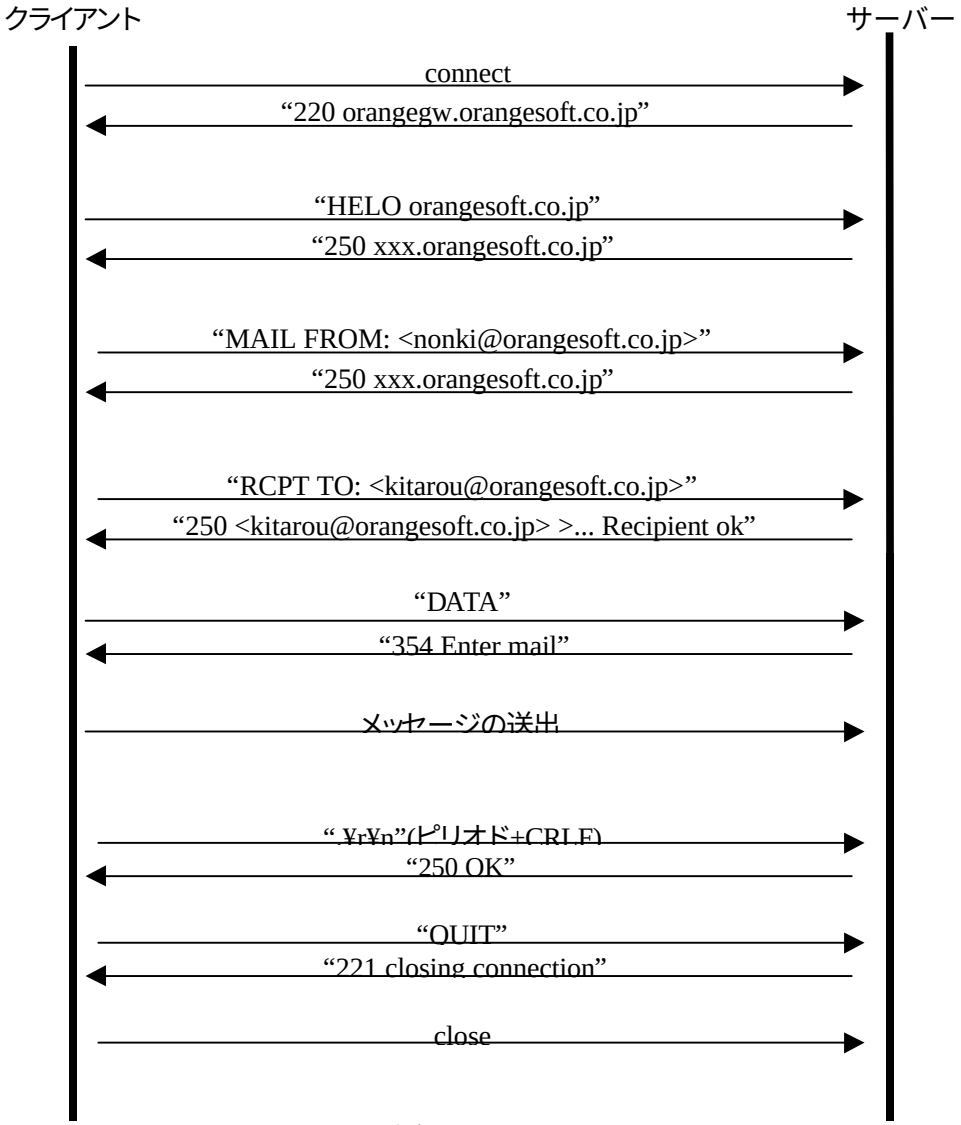


図 28 SMTP

最初にまず、当然 socket レベルといいますか、ネットワークレベルで connect します。そうしますと、最初にサーバ側から、これは例ですけれども、こういう文字列が返ってきます。このレスポンスコードの先頭のほうに 3 桁の数字が入ってきます。250 や 220 です。これがレスポンスコードです。このコードを調べて、エラーが起こったとか、いろいろな処理を続けていきます。最初にそれを調べて、220 は受け入れがオーケーなので、その次に HELO、こんにちはということで、あいさつをします。通信できますよ、という返事をもらい、ここから実際のメールを 1 通ずつ送り出すわけです。これは送り元がだれか。今回は私ですけれども、nonki@orangesoft.co.jp が FROM です。それから、kitarou@orangesoft.co.jp というものに送りますので、RCPT TO:<kitarou@orangesoft.co.jp>と書きます。そうすると、いいですよと返ってきます。先頭が 2 だとオーケー、成功応答なので続けていきます。

ここまででだれが送るか、だれに送るかというやり取りが終わりましたので、実際のメールを送ります。DATA ということでデータを送ります。354 というのは継続したデータを送ってくださいねということです。それで、そのあとにメッセージを送って

いきます。それで、メッセージを 1 個送り終わりましたら、最後にピリオド＋改行の文字を送ります。これでメールを 1 通送り終わったということになります。実際に 2 通、3 通送りたい場合は、MAIL FROM から始めてもらえればいいわけです。それで、全部送り終わったら、ここで QUIT を出して終わります。これで、ネットワークを落としておしまいということになります。
先ほどのは簡単なシーケンスですけれども、SMTP コマンドは多数あります。以下に抜粋を示します。

コマンド一覧(抜粋)

- HELO　使用開始
- MAIL　メール送信の開始
- RCPT　送り先メールアドレスの指定
- DATA　メール本文の送信開始
- RSET　リセット
- NOOP　何もしない
- QUIT　接続の終了

先ほどの HELO、MAIL、RCPT、DATA そして RSET はありませんでしたがリセットです。それから、NOOP、これは何もしません。NOOP を送ると 220 が返ってきます。それから、QUIT。あといろいろありますが、今回は抜粋ということで省略します。
SMTP に関していいますと、先頭文字 3 桁の文字で成功か失敗かの応答がわかります。xyz、最も左の桁は 1、2、3、4、5 です。多くは 2 です。2 で肯定的。3 が、中間的というのは変な言い方ですが、要するに前のデータの応答などは 3 を返します。それから、エラーが起こった場合には 4 とか 5 が返ってきますので、こちらのほうでチェックしなければいけません。複数行の応答が返ってこなければいけないときがありますが、そういう場合は 123 という数字が返ってきたその後ろにハイフンがくっついてきます。ですから、123 でハイフンがついてきた場合は、ハイフンがなくなる、要するにスペースになるまでずっと読んでいくという処理が必要になってきます。

POP３

POP３ の流れを図 29 に示します。

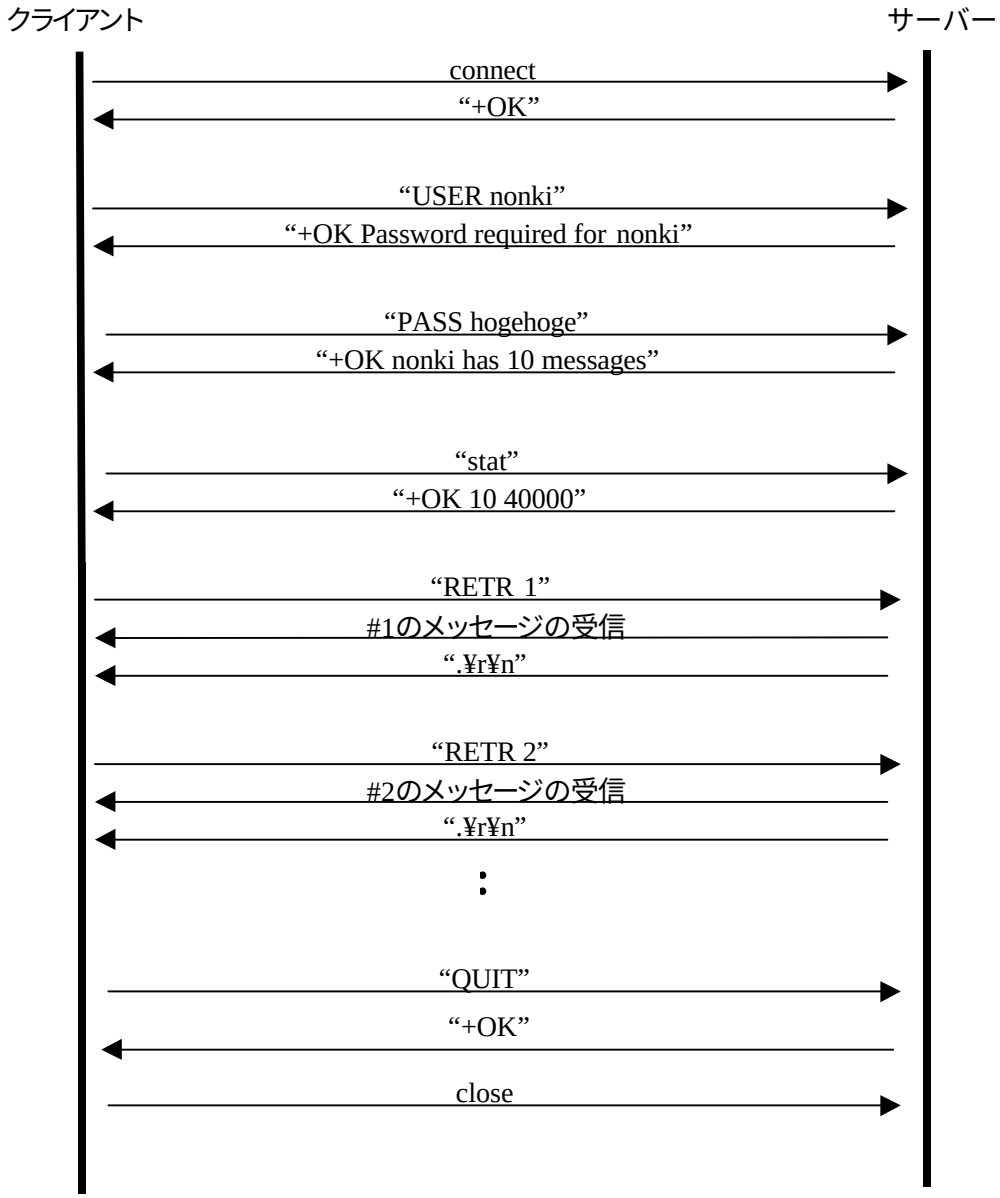


図 29 POP３

POP のほうは要するにテキストベースのプロトコルなので大体似たようなものです。応答コードが、先ほどは数字だったのですが、それが文字列、+OK とか、エラーの例は書いていませんが、-ERR を返してきます。やはり同じように接続するとサーバからオーケーが返ってきます。それから、POP の場合はメールを拾うわけですから、ユーザ名、それからパスワードを渡さなければいけません。まずユーザは私ですから、nonki。いいよと、オーケー。それからパスワード。実際はクリアで渡ってしまいますけれども、これは暗号化した方法もありますが、今回は説明しません。hogehoge と出し、これでパスワードが合っているとオーケーが返ってきます。それで、stat というコマンドを使って、今実際にメールが何件入っているかというのがわかります。ここ

では 10 件。それから、その次の 40000 は、総バイト数を返してくれます。RETR 1 というのはリトリーブ 1、要するに 1 番目のメッセージをくださいというコマンドです。これを渡しますと、POP サーバからは 1 番目のメッセージを送りだしてきます。メッセージの終わりも、先ほどの SMTP と同じようにピリオドだけの改行文字で表現されています。ですから、.¥r¥n が返ってきたら一応メッセージが終わりだということがわかります。それから、同じように２通目、３通目と繰り返して続けてきます。全部処理が終わったら QUIT を出して、オーケー応答をもらって閉じるという処理になっていきます。
POP3 の全てのコマンドを以下に示します。

コマンド一覧

- **STAT** メールボックスのメール数とサイズの所得
- **LIST** ここのメールのサイズの所得
- RETR 指定したメールの取出し
- DELE 指定したメールの削除
- NOOP 何もしない
- RSET 操作の取消し
- QUIT 接続の終了
- TOP 指定したメッセージのヘッダと本文を指定した行数所得する
- UIDL ここのメールのサーバ内でのIDを所得する
- USER ユーザ認証時のユーザ名の送信
- PASS ユーザ認証時のパスワードの送信
- APOP MD5で暗号化されたユーザ認証

レスポンス

- +OK 成功 -ERR 失敗

POP3 というのは、おもしろいプロトコルで、かなりのコマンドがオプショナルなコマンドです。例えば PASS というコマンドもオプショナルで、実際は PASS を実装しなくても POP サーバと言い張ることもできます。おかげでメールサーバ、メールクライアントをつくる人はいろいろ苦労するわけです。いずれにしろ、これらのコマンドを使っていろいろ処理をするということになります。
レスポンスは、先程の SMTP の文字コード化された親切なコードと比べて非常に不親切です。成功したか失敗したかしかわかりません。エラーの場合はエラーの理由を教えてくれますので、それを見るということになります。

特徴

今までの２つのプロトコルの特徴は、やはりテキストベースだということです。他のプロトコルの多く、特にインターネットのプロトコルはテキストベースのプロトコルがほとんどです。ほとんどの場合、telnet でそのサーバにつなげて、実際の動作を確認するようなことができます。ですから、デバグするようなときとか、実際のサーバの動作を確認するようなときには、実際のアプリケーションももちろん必要ですけれども、実際に自分で telnet でポートを、例えば POP ですと 110 ですけれども、110 につなげて実際に手で打って確認するということができます。これは非常にいい確認法だと思います。むやみにプロバイダでやってはいけませんが、自分のところの会社でやるのは構わないと思います。
それから、行単位の処理ということで、転送データ量が予測できません。何バイト来るよというのがわかりませんから、実際は改行がセパレータ（CRLF(0x0d,0x0a)）なので改行が来るまで待っているようなことをしなければいけません。
補足ですが、セパレータがピリオドと CRLF だと言いましたが、では本文の中にピリ

オドと改行だけがあるようなメールがあったらどうなるんだという疑問もあるかもしれませんが、そういうメールはメールの中、メッセージの中ではピリオド、ピリオドというふうに 1 個追加しています。これによってピリオドだけの行というのはメールの中には発生しません。それで、実際に表示するときにはそれを抜くということをしなければいけないということになります。

サンプルソースコード

ここでサンプルソースコードのテキストを説明したいと思います。
POP を使ってメールを受信するプログラムで、コメントも入れて全部で 230 行あります。
ボトムアップに書いているので、メーンから簡単に説明します。main から、上から説明します。まず概略ですけれども、open_popserver という関数を呼んで、まず引数に mailserver という、これはホスト名ですけれども、これを渡してあげて、pop サーバとまずコネクションを張りまして、open_popserver の中で例の socket とか connect とかそういうのを呼んでいます。pop_login という関数でユーザ名とパスワードを引数に渡して、実際にその認証を行います。

```
/*
 * メールを受信する(POP)/socket 版
 *
 * Copyright 1998, Shuji Ishii, shuji@ccs.mt.nec.co.jp
```

```
void main(int argc, char **argv)
{
    POPSERVER *pop;
    char *pass;
    int msgcnt;
    int i;

    if ((pop = open_popserver("mailserver")) == NULL) {
        fprintf(stderr, "error: %s¥n", *argv);
        exit(2);
    }

    if (pop_login(pop, "shuji", "password")) == POP_ERR) {
        fprintf(stderr, "error: pop_login¥n");
        exit(3);
    }

    if ((msgcnt = pop_msgcnt(pop)) < 0) {
        fprintf(stderr, "error: pop_msgcnt¥n");
        exit(4);
    }

    for (i = 1; i <= msgcnt; i++) {
        if (pop_retr(pop, i, stdout) == POP_ERR) {
            fprintf(stderr, "error: pop_retr¥n");
            break;
        }
    }

    close_popserver(pop);
    exit(0);
```

上に戻りまして、使っているデータ構造を簡単に説明します。構造体は 1 つで、_popserver という構造体があります。それで、基本的にはファイル構造体というスタンダード I/O のライブラリを使っていまして、それの in と out、入出力用のディスクリプタのポインタと、あと result という 256 バイトとっていますけれども、そこの結果を格納する領域を確保した、こういったものを使います。OK と ERR はこういうふうに定義しています。

まず、pop サーバに接続する先程の関数ですが、hostent とか protoent とか servent とか一通りそろっています。あと sockaddr_in というので、これで相手の pop サーバのポート番号とかアドレスを指定します。

基本的にはコメントを読んでいきますけれども、まず gethostbyname で pop サーバ……。引数の説明をします。引数はサーバの名前です。サーバのホスト名から実際のアドレスを取るために gethostbyname を呼びます。エラーだったら下にありますけれども、エラーハンドラ、goto err に飛ぶ。次に、プロトコル番号を、0 で書けばいいと何回も言いましたけれども、ここでは TCP のプロトコル番号を調べています。これは pp というところに返ってきます。最後に、pop3 サービスというポート番号を調べています。これは pop3 という名前で tcp というので取ってきています。ここの malloc は、上で定義した構造体をそのまま取っている。

60 行目で socket をこういうふうにつくっています。PF_INET で SOCK_STREAM で、最初は 0 でもよいと書いたところはこういうふうに pp で p_proto というメンバーに入っているプロトコル番号を渡してあげます。socket をつくってから sockaddr_in の構造体を設定して、順番はどうでもいいですけれども、ここでとりあえず気持ちの問題ですけれども、0 でクリアしてから長さとファミリとポート番号——これは sp です。これで sockaddr_in 構造体を設定して、connect します。s は上でつくった socket です。それで、sin。キャストしているのは、もともとはここは struct sockaddr という構造体へのポインタを渡すようになっているのでこうなっています。あと size を渡し connect します。エラーがあったらエラーへ飛びます。

```
#define POP_RESULT_LEN (256)

typedef struct _popserver {
    FILE *in, *out;     /* POP サーバ入出力用 */
    char result[POP_RESULT_LEN]; /* 結果格納領域 */
} POPSERVER;

#define POP_OK (0)
#define POP_ERR (-1)

static int transaction(POPSERVER *pop, char *command);

/*
 * pop サーバに接続する.
 * 接続に成功すれば POPSERVER 構造体へのポインタを返す
 * エラーの場合 NULL を返す
 */
static POPSERVER *open_popserver(char *popserver)
{
    struct hostent *hp;    /* ホストエントリ */
    struct protoent *pp;   /* プロトコルエントリ */
    struct servent *sp;    /* サービスエントリ */
    int s = -1;         /* ソケット */
    struct sockaddr_in sin;   /* サーバ */
    POPSERVER *pop = NULL;

    /* pop サーバのエントリを取得する */
    if ((hp = gethostbyname(popserver)) == NULL) {
        goto err;
    }
    /* tcp のエントリを取得する */
    if ((pp = getprotobyname("tcp")) == NULL) {
        goto err;
    }
    /* pop3 サービスのエントリを取得する */
    if ((sp = getservbyname("pop3", "tcp")) == NULL) {
        goto err;
    }

    if ((pop = (POPSERVER *)malloc(sizeof(POPSERVER)))
== NULL) {
        goto err;
    }

    /* socket を作る */
    if ((s = socket(PF_INET, SOCK_STREAM, pp->p_proto))
< 0) {
        goto err;
    }
```

次は、スタンダード I/O の fgets などを使えるようにするために fdopen で入出力側をこのように設定しています。

ここには transaction という関数を 1 個用意していますけれども、transaction という関数はコマンドを送ってその結果をもらう、それをまとめた関数です。つなぐといきなり何かメッセージを出してきますので、まずそれをとりあえず読み込むためにこれを呼んでいます。読み込んだらこの pop という構造体へのポインタを返します。エラーハンドラはエラーハンドラでいいので、エラーと返すということです。

```
/* sockaddr_in 構造体を設定する */
    memset(&sin, 0, sizeof(sin));
    sin.sin_len = sizeof(sin);
    sin.sin_family = PF_INET;
    sin.sin_port = sp->s_port;
    memcpy(&sin.sin_addr, *(hp->h_addr_list), sizeof(sin.sin_addr));

    /* pop サーバに接続する */
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        goto err;
    }

    /* pop サーバへの通信路を fdopen で open する */
    pop->in = fdopen(s, "r");
    pop->out = fdopen(s, "w");

    /* greeting message を読み込む */
    transaction(pop, NULL);

    return pop;

     err:
    if (pop) {
        free(pop);
    }
    if (s >= 0) {
        close(s);
    }
    return NULL;
```

先に close を説明しますが、close は簡単です。fclose して解放する。それだけです。中心的なのはこの transaction というところで、ここも簡単です。基本的には POPSERVER をあらわすこの構造体と command と。上では一番最初は NULL を渡していましたけれども、command がない場合は単に読み込むだけということをします。command があればこういうふうに fprintf を使って command を%s で指定して、セパレータが CRLF でしたから、これをつけて渡す。flush というのはスタンダード I/O のライブラリの関数で、バッファリングされている場合がありますから、それを flush するということをします。

```
* pop サーバへの接続を閉じる
 */
static void close_popserver(POPSERVER *pop)
{
    fclose(pop->in);
    fclose(pop->out);
    free(pop);
}

/*
 * コマンドを送信し、結果が +OK なら POP_OK を、それ以外なら POP_ERR を
 * 返す
 */
static int transaction(POPSERVER *pop, char *command)
{
    if (command != NULL) {
        fprintf(pop->out, "%s¥r¥n", command);
        fflush(pop->out);
    }
    fgets(pop->result, POP_RESULT_LEN, pop->in);

    return (pop->result[0] == '+') ? POP_OK : POP_ERR;
}

/*
```

fgets というところで読み込んで、ちゃんとエラー処理をしていないですけれども、それで result、さっきの構造体の中にあった領域に読み込む。リターンバリューが一応+OK とか-ERR と出るはずなので、それに従って POP_ERR とか POP_OK を返します。

あとは transaction というのを使ってやり取りします。ですから、例えば login する場合には USER というコマンドを送って、エラーだったら ERR を返し、そのままオーケーだったら pass というコマンドでパスワードを送ります。セキュリティというか安全性のためにパスワードがメモリの中に入っているので、消していますが、基本的には transaction という今の関数を使って送っています。

メッセージのカウントも一緒です。STAT を呼びます。STAT の場合は全部のメッセージの数と全体の大きさが返ってくるので、それを解析しています。実際にはメッセージの数を返しています。
以上が fetchmail です。

```
* 現在のメッセージ数を返す
 */
static int pop_msgcnt(POPSERVER *pop)
{
    int msgcnt, ttlsize;

                /* STAT コマンド */
    if (transaction(pop, "STAT") == POP_ERR) {
        return -1;
    }
                /* 結果を解析 */
    if (sscanf(pop->result, "+OK %d %d", &msgcnt, &ttlsize) != 2) {
        return -1;
    }

    return msgcnt;
}

/*
```

今度は送信するほうです。これも概略を説明したいと思います。構成は一緒ですけれども、main です。ちょっと違うのはヘッダーというのをあらかじめここで定義していますけれども、From:<shuji>、@が右側にないですけれども、メールアドレスです。これは日比野さんの nonki というアドレスに、Subject は test で、Mime はこれでというのを設定しておきます。メッセージそのものは標準入力から読むというものです。基本的に構成は一緒で、open_smtp.server で open してから、ひたすら transaction というのを、同じような関数ですけれども、呼んで HELO とか MAIL とか recvto とか data と呼んで終わる。そういったプログラムです。

```
void main(int argc, char **argv)
{
    MAIL mail;
    SMTPSERVER *smtp;
    static char *headers[] = {
        "From: <shuji>",
        "To: <nonki>",
        "Subject: test",
        "Mime-Version: 1.0",
        "Content-type: text/plain; charset=us-ascii",
        "Content-Transfer-Encoding: 7bit",
        NULL,
    };
mail.headers = headers;
    mail.body = stdin;

    if ((smtp = open_smtpserver("mailserver")) == NULL) {
        fprintf(stderr, "open_smtpserver: localhost¥n");
        exit(1);
    }

    if (transaction(smtp, "HELO localhost", NULL) == SMTP_ERR) {
        fprintf(stderr, "transaction: HELO; %s¥n", smtp->result);
        goto quit;
    }
    if (transaction(smtp, "MAIL FROM: <shuji>", NULL) == SMTP_ERR)
{
        fprintf(stderr, "transaction: MAIL FROM; %s¥n", smtp->result);
        goto quit;
    }
    if (transaction(smtp, "RCPT TO: <nonki>", NULL) == SMTP_ERR) {
        fprintf(stderr, "transaction: RCPT TO; %s¥n", smtp->result);
        goto quit;
    }
    if (transaction(smtp, "DATA", &mail) == SMTP_ERR) {
        fprintf(stderr, "transaction: DATA; %s¥n", smtp->result);
        goto quit;
    }
    if (transaction(smtp, "QUIT", NULL) == SMTP_ERR) {
        fprintf(stderr, "transaction: QUIT; %s¥n", smtp->result);
        goto quit;
    }

 quit:
    close_smtpserver(smtp);
    exit(0);
}

/* ----- end of delivermail-socket.c
```

質疑応答

（質問 1）今回のチュートリアルとちょっと外れてしまいますが、例えば TCP を自分でつくりたいと思ったとき、IP の上から直接プログラムを組みたいというときに、そのときのプログラムインターフェースというのは Windows とか、あと Berkeley の BSD とかで用意されているんでしょうか。

（石井）UNIX では、Berkeley 系ではあります。例えば直接 IP のパケットをいじるとかそういうことですよね。それで、今回は本当は時間があれば紹介したかったんですが、一応 2 つありまして、一つは BPF（Berkeley Packet Filter）と呼ばれるものがあります。それは IP だけではなくて、本当に Ethernet フレームを自分でつくる、あるいは流れているのを取ってくるという、それはそれで結構危険なインターフェースがあって、いたずらし放題です。それはここの参考文献で挙げているところにある「インターネットを 256 倍……」というアスキーで出している本がありますが、その Vol.2 で結構詳しくプログラミングが解説されているので、これは結構よいと思います。2 つ目は、BPF のほかにもう一つ、IP ヘッダというか、IP の上で TCP 以外にほかに何か追加するといったスタイルの API というかインターフェースも用意されています。それは、参考文献の 66 ページの、スライドの 131 ページの上の、英語ですけれども、UNIX Network Programming Volume 1 Second Edition に載っていますので、そこを読んでいただければよいと思います。Windows ではありますか。私も知りたかったりですが。ドライバをつくらないとだめですか。

（日比野）そうですね。基本的にドライバをつくらなければいけないことになると思います。ただ、一つだけわかっているのは、今回お話ししませんでしたけれども、Winsock のバージョン 2 は実際は TCP だけではなく、ほかのネットワークにも対応するようなアーキテクチャになっているんですね。ですから、ちょっとご質問とは違うかもしれませんけれども、実際に Winsock は上の API になっていて、下は自分でドライバを書くという流れになっていくと思います。ただ、何にしてもやはりドライバレベルの話になると思います。

まとめ（石井、日比野）

インターネットの概要などについてお話ししました。UNIX、Berkeley 系ですが、socket interface についてお話しし、select とか、サーバの構造、fork とか inetd のやり方について説明しました。Windows の方は、Windows-style と Berkeley-style の２種類があります。それぞれシングルスレッド、マルチスレッドの 2 種類あり、計 4 種類やり方がありますが、日比野さんは Berkeley-style でマルチスレッドがお勧めということでした。Windows-style の API というのはメッセージベースの非同期 API であるということです。Java の方は、Socket をもともと持っているということと、同期型の API ということをご説明しました。smtp と pop のプロトコルについても少しお話ししました。

参考文献は、次のものがよいかと思います。

- “UNIX Network Programming Volume 1 Second Edition, Networking APIs: Sockets and XTI” , W.Richard Stevens 著, Prentice Hall
- “インターネットを256倍使うための本 Vol. 2”, 志村 拓, 榊 隆, 岩井 潔 共著, アスキー出版局
- マニュアルページ(**man** コマンド)
- WinSock2.0プログラミング ソフトバンク(ISBN4-7973-0688-2)
- インターネットRFC事典　アスキー(ISBN4-7561-1888-7)

次は Resources です。

- Java House Mailing List　　http://java-house.etl.go.jp/ml/
- Javaカンファレンス　　http://www.java-fj.or.jp/

添付資料1

簡単なサーバの例：daytimed.c

```
 1 /*
 2  * daytimed.c
 3  */
 4
 5 #include <stdio.h>
 6 #include <string.h>
 7 #include <unistd.h>
 8 #include <sys/types.h>
 9 #include <sys/socket.h>
10 #include <sys/time.h>
11 #include <sys/wait.h>
12 #include <signal.h>
13 #include <netinet/in.h>
14 #include <netdb.h>
15 #include <time.h>
16
17 static void print_daytime(int sd)
18 {
19      char strbuf[64];
20      struct timeval tv;
21
22      gettimeofday(&tv, NULL);
23      strftime(strbuf, sizeof(strbuf), "%a %b %d %T %Y¥r¥n",
24               localtime(&tv.tv_sec));
25      write(sd, strbuf, strlen(strbuf));
26      exit(0);
27 }
28
29 void sigchld(int sig)
30 {
31      (void)wait(NULL);
32 }
33
34 main(void)
35 {
36      int sd, snew;
37      struct sockaddr_in sin;
38
39      signal(SIGCHLD, sigchld);
40
41      if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
42           perror("socket");
43           exit(1);
44      }
45
46      sin.sin_len = sizeof(sin);
47      sin.sin_family = PF_INET;
```

```
48       sin.sin_port = htons(8888);
49       sin.sin_addr.s_addr = INADDR_ANY;
50
51       if (bind(sd, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
52            perror("bind");
53            exit(2);
54       }
55
56       listen(sd, 5);
57       for (;;) {
58            if ((snew = accept(sd, NULL, NULL)) < 0) {
59                 perror("accept");
60                 exit(3);
61            }
62            if (fork() != 0) {
63                 /* 親 */
64                 close(snew);
65            } else {
66                 /* 子 */
67                 close(sd);
68                 print_daytime(snew);
69            }
70       }
71 }
72 /* ----- end of daytimed.c ----- */
```