■3 群（コンピュータネットワーク） - トランスポートサービス

7 章 トランスポート層プログラミング

（執筆者：村山公保）［2011 年 2 月 受領］

■概要■

この章では，トランスポートサービスの締めくくりとして，TCP，UDP を利用するアプリケーションプログラミングについて解説する．トランスポートサービスを利用するときの API（Application Programming Interface）にはデファクトスタンダードの BSD 系 UNIX のソケットを使用し，プログラミング言語は C 言語を使用する．

ソケットは，System V 系 UNIX の XTI（X/Open Transport Interface）と並んで歴史が古い．ソケットや XTI が設計された時代は，TCP/IP プロトコルスイートがデファクトスタンダードになる前であり，ISO の OSI プロトコルも広く使われると予想されていた．このため，ソケットや XTI は TCP/IP だけではなく OSI プロトコルなど，他のプロトコルにも対応できるように汎用的な設計になっている．アプリケーションを作成するときには，TCP や UDP を利用するうえでの API の書式について理解が必要になる．

性能や効率の良いシステムを開発したい場合には「API の書式の理解」だけでは不十分である．ソケットはトランスポートプロトコルの性質を，ソケット特有のプログラミングスタイルに置き換えたうえで提供する．このため，性能や効率の良いシステムを開発するためには，利用するトランスポートプロトコルの仕組みとソケットの関係について理解し，それぞれが協調動作するようにアプリケーションプログラムを作成する必要がある．

トランスポートプロトコルは，アプリケーションプログラムの作成を手助けするために存在する．アプリケーションによって使われて初めてそのトランスポートプロトコルに実用上の価値が生まれてくる．既存のトランスポートプロトコルを改善したり，新たなトランスポートプロトコルを設計，実装するときには，提案するトランスポートに対するアプリケーションからの要求を既存の API で実装できるのか，新しい API を定義する必要があるのか考える必要がある．そのためには，アプリケーションからトランスポートを利用するプログラミングの方法を理解する必要がある．

以上のことを理解できるようにするため，本章では UNIX と Windows の両方で動作するサンプルプログラムを掲載し，プログラムの内容と，実行結果を基にトランスポート層プログラミングの方法と注意点，性能改善の方法について解説する．

【本章の構成】

本章では，ソケットプログラミングの概要（7-1 節），TCP と UDP に対応したサンプルプログラム（7-2 節），プログラミングの側面から見た TCP と UDP の違い（7-3 節）について解説し，通信性能を考えたプログラミングの方法（7-4 節）について述べる．

■トランスポートサービス - 7 章

7-1 ソケットプログラミングの概要

（執筆者：村山公保）[2011 年 2 月 受領]

まず，ソケットプログラミングの概要を説明する．具体的には，ソケットを識別するソケットディスクリプタとシステムコールの概要について説明し，エラー処理やシグナル，タイムアウト処理などについて説明する．

7-1-1 ソケットが提供する機能

ソケットが利用できる実装では，TCP や UDP はカーネルモジュールとして提供されることが多い．アプリケーションが TCP や UDP を使って通信したい場合には，システムコールを使用してカーネルモジュールのサービスルーチンを利用する．

ソケットはネットワークにおける通信を，メモリを介した関数呼び出しに抽象化する．メッセージを送信するときには，送信したいメッセージをメモリ上に格納し，そのメモリ上の先頭アドレスとバイト数を指定して，送信関数（後述する `send()`，`sendto()`，`sendmsg()`）を呼び出す．すると，トランスポートサービスが働き，メッセージがパケット化されてネットワーク中に送信される．メッセージを受信するときには，受信バッファの先頭アドレスとバッファのバイト数を指定して，受信関数（後述する `recv()`，`recvfrom()`，`recvmsg()`）を呼び出す．すると，届いているパケットのメッセージ部分が受信バッファに格納される．送信関数はトランスポートサービスに対してパケット送信を働きかけるが，受信関数はパケットの受信を働きかけるわけではない．OS の受信バッファにたまったメッセージをアプリケーションのバッファにコピーするだけである．

7-1-2 クライアントとサーバ

ソケットでは，一般的に，クライアント・サーバモデルでアプリケーションプログラムが作成される．使用する命令や書き方はサーバとクライアントで異なっている*．概念的には，サーバは受動的（Passive）で，クライアントは能動的（Active）な処理になる．

サーバは，自分のポート番号を指定して，コネクションの確立受付（TCP），または，要求パケットの到着（UDP）を待つ．サーバは相手を指定してパケットを送ったりしない．

クライアントは，通信相手の IP アドレス，ポート番号を指定して，コネクションの確立要求（TCP），または，要求パケットの送信（UDP）を行う．クライアントが相手を指定してパケットを送らなければ通信は始まらない．

一つのプログラム内で，サーバの機能とクライアントの機能の両方をもたせることもできる．電子メールの MTA（Mail Transfer Agent）や Web プロキシ，SIP User Agent などはそうなっている．

7-1-3 ソケットディスクリプタ

ソケットではソケットディスクリプタと呼ばれる整数値を使用して通信の設定，メッセー

* 本稿はユニキャストを前提として解説する．マルチキャストやブロードキャストを利用するときには，プログラムコードがサーバ，クライアントに明確に分かれない場合もある．

ジの送信・受信を行う．ソケットディスクリプタは socket() でソケットをオープンすると得られる．このディスクリプタに対して IP アドレスやポート番号などの設定，メッセージの送信や受信などの指示を行うことによって，トランスポートサービスを利用することができる．

UNIX 系では，open() で得られるファイルディスクリプタと socket() で得られるソケットディスクリプタは共通の識別子になっている．プログラム実行開始時にディスクリプタの 0，1，2 はそれぞれ標準入力，標準出力，標準エラー出力が割り当てられる．

7-1-4 ソケットシステムコールの概要

socket() でソケットをオープンするときには，ネットワーク層のプロトコルとトランスポート層のプロトコルを指定する．ソケットをクローズするには UNIX 系では close()，Windows 系では closesocket() を使う．TCP ではコネクションの切断など，通信を終了させるために shutdown() が使われることもある．

bind() で自分で使用する IP アドレス，ポート番号を指定し，connect() で相手が使用する IP アドレス，ポート番号を指定する．connect() 時に，自分のホストで使用する IP アドレス，ポート番号が決まっていない場合には，OS が自動的に選択する．

TCP の場合，コネクション受け付け用のソケットと，メッセージ送受信用のソケットは区別される．socket() でオープンしたソケットは，コネクション受け付け用のソケットであり，アプリケーションメッセージの送受信には利用されない．socket() でオープンしたソケットに対して listen() するとコネクションの受付が開始される．TCP コネクションが確立されると新しいソケットが作られ，accept() で作られたソケットのディスクリプタを得ることができる．このディスクリプタを使ってアプリケーションメッセージを送受信することになる．

TCP ではコネクション確立後 accept() しなければサーバ側ではメッセージを送受信できない．accept() しなくても確立できる TCP コネクション数の上限を listen() で設定できる．

メッセージの送信は send()，sendto()，sendmsg()，受信は recv()，recvfrom()，recvmsg() を使用する．

send()，recv() は，TCP のとき，または，UDP で connect() したときに使用する．つまり，コネクション識別子である自 IP，自ポート，相手 IP，相手ポートが固定されているときに使用する．sendto()，recvfrom() は，UDP で相手 IP，相手ポートが固定されていないときに使用する．

send()，sendto()，recv()，recvfrom() は，送受信に使用するバッファは連続するメモリ領域になければならない．sendmsg()，recvmsg() は，複数の不連続なメモリ領域にあるバッファを使って，メッセージを送受信することができる．

ソケットの設定変更には setsockopt()，設定内容の取得には getsockopt() を使う．タイマの設定，バッファサイズの変更，Nagle アルゴリズムの無効化などができる．

7-1-5 ソケットアドレス構造体

通信をするには IP アドレスとポート番号を指定する必要がある．これらの情報を格納する

ために IPv4 では sockaddr_in 構造体，IPv6 では sockaddr_in6 構造体が利用される．しかしながら IPv6 が提案されてから，プログラムコードから IPv4 や IPv6 固有の命令を排除し，IPv4/IPv6 のどちらにも対応する「プロトコル非依存（Protocol-independent）」なプログラムを作成することが奨励されるようになった[1]．そこで登場したのが sockaddr_storage 構造体，addrinfo 構造体と，getaddrinfo()，getnameinfo()，freeaddrinfo() という各種関数である．

sockaddr_storage 構造体は，IP アドレスやポート番号といった通信に必要な情報をバイナリ形式で格納するためのものである．この構造体への値の設定，取り出しは関数を介して行う．

addrinfo 構造体はメンバに sockaddr_storage 構造体へのポインタをもっており*，リスト構造を作って複数の IPv4 アドレスや IPv6 アドレスを扱えるようになっている．

getaddrinfo() は，文字列で記述した IP アドレスやポート番号を addrinfo 構造体に格納する働きがある．getaddrinfo() は内部でメモリ割り当てを行うため，不要になった場合には freeaddrinfo() で割り当てたメモリ領域を解放する必要がある．IP アドレスやポート番号の代わりにドメイン名やサービス名を記述することもできる．ドメイン名に対応する IP アドレス（IPv6 アドレス）が複数ある場合には，それらすべてがリスト構造として格納される．

getnameinfo() を使うと，addrinfo 構造体に格納されているバイナリ型の IP アドレスやポート番号を文字列に変換して取得できる．DNS へ問い合わせてドメイン名などを取得することもできる．

7-1-6 エラー処理について

UNIX 系では，ソケットシステムコールがエラーを返した場合，グローバル変数の errno をチェックすることによりエラーが生じた原因を知ることができる．例えば TCP コネクションが切断された場合，タイムアウトで切断されたのか，RST で切断されたのかを知ることができる．Windows 系の Winsock では，エラーが起きても errno には設定されず，代わりに WSAGetLastError() でエラー情報を得る．UNIX 系でも getaddrinfo() などのライブラリ関数使用時には，エラーが発生しても errno には設定されず，gai_strerror() などの関数でエラー情報を得る必要がある．

7-1-7 シグナルについて

UNIX 系では，TCP コネクションが TCP RST など異常な形で切断され，あとで send() などのシステムコールをすると，SIGPIPE シグナルが発生する．デフォルトではプログラムが強制終了するため，シグナルをマスクするか，シグナルハンドラを用意して SIGPIPE シグナル発生時の処理を記述する必要がある．Linux では，send() 時にフラグに MSG_NOSIGNAL を指定することで，シグナルの発生を抑制することもできる．

* 厳密に言えば sockaddr 構造体へのポインタ．

7-1-8 タイムアウト処理について

TCP や UDP は通信相手がいなくなっていても特別な処理は行わない．これは，一旦始まった通信の途中で，相手システムが障害でダウンした場合でも，永遠に相手からのメッセージを待ち続けるという問題を引き起こすことがある．これらへの対処が必要な場合には，上位層のアプリケーションが行わなければならない．

ソケットの場合，send() や recv() などがブロックする最大時間を設定できる．setsockopt() で SO_RCVTIMEO，SO_SNDTIMEO 指定すると，それぞれ送信時，受信時のタイムアウト時間を設定できる（レベルは SOL_SOCKET）．指定した時間経過後に処理が終わらない場合には処理が中断されるため，戻り値と error などをチェックして適切な処理をするようにする．

TCP のキープアライブ [2] を使用したい場合には，setsockopt() で SO_KEEPALIVE を指定する（レベルは SOL_SOCKET）．

select()，poll() を使ってそのソケットディスクリプタに対する処理が，ブロックするかどうかを調べ，ブロックしない場合のみ送信処理や受信処理をする方法も利用される．

7-1-9 ブロッキングとノンブロッキングについて

ソケットシステムコールの多くはデフォルトでブロッキングモードで動作する．ノンブロッキング処理を行いたい場合にはシステム固有の方法を使って変更することができる．

UNIX 系の場合には fcntl() や ioctl() でそのソケットの設定をノンブロッキングに変更できる．Linux では send() や recv() のフラグに MSG_DONTWAIT を指定すると，システムコールごとにノンブロッキングに変更できる．

Windows 系でノンブロッキング処理を行いたい場合には，ioctlsocket() でソケットの設定をノンブロッキングに変更するか，WSAAsync で始まる非同期関数を使用する．

■トランスポートサービス - 7 章

7-2 TCP と UDP に対応したサンプルプログラム

（執筆者：村山公保）[2011 年 2 月 受領]

この節では実際に動作するサンプルプログラムを提示し，トランスポートサービスの利用方法について解説する．

7-2-1 サンプルプログラムの概要

サンプルプログラムは，クライアント（file_client0.c）とサーバ（file_server0.c）に分かれており，サーバ側からクライアント側にファイルを転送するプログラムになっている．

プログラム起動時にコマンドラインで必要なパラメータを指定する．サーバを起動するときには，送信するファイル名，自分で使用するポート番号を指定する．クライアントを起動するときには，保存するファイル名，サーバの IP アドレス・ポート番号を指定する．クライアントを起動すると，すぐにサーバからクライアントへ指定されたファイルが転送される．転送終了後，クライアントプログラムは終了し，サーバプログラムは次の要求を待つ．

IPv6 用に追加された getaddrinfo() や getnameinfo() 関数を使用しているため，IPv4, IPv6 の両方に対応している．動作確認は Linux，MacOS X，Windows の最近*のディストリビューションで行った．IPv6 非対応の OS で動作させるには修正が必要になることがある．

TCP と UDP のコーディングの違いや，プロトコル特性の違いを考えやすくするため，サーバもクライアントも TCP と UDP の両方に対応している．TCP/UDP 固有の行は，#ifdef～#else～#endif などによる条件コンパイルで区別している．

パケットの喪失時の再送処理をトランスポートに任せているため，UDP 版でコンパイルして実行すると，保存されるファイルにデータの欠落が発生する可能性がある．

本プログラムではエラー処理を省いている．コメントも通信に関係のある部分のみ記述した．プログラムコードを短くして，ネットワークプログラミングの要点に着目しやすくするためである．何らかの問題があって動作しなくてもエラーメッセージが表示されないため，トラブルシューティングは難しくなっている．

7-2-2 サーバプログラム (file_server0.c)

```
 1 /* file_server0.c: TCP/UDP ファイル転送サーバ Ver. 1.0 2011.01.11 */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <string.h>
 5 #ifdef _WIN32
 6 #include <winsock2.h>
 7 #include <ws2tcpip.h>
 8 #define exit(_Z)   WSACleanup();exit(_Z)
 9 #define close(_Z)  closesocket(_Z);
10 typedef int socklen_t;
11 #else
12 #include <sys/types.h>
13 #include <sys/socket.h>
14 #include <netdb.h>
15 #include <netinet/in.h>
```

* 本稿執筆の 2011 年 1 月時点.

```
16 #include <unistd.h>
17 #endif
18
19 //#define UDP  /* デフォルトはトランスポートに TCP を使用。UDP の場合は//を外す */
20 #define BUFSIZE 8192  /* BUFSIZE >= MSGSIZE + DUMMY */
21
22 #ifdef UDP
23 #define MSGSIZE 1024
24 #define DUMMY 4
25 #else
26 #define MSGSIZE 8192
27 #endif
28
29 enum args {CMD_NAME, READ_FILE, LOCAL_PORT};
30
31 int main(int argc, char *argv[])
32 {
33   struct sockaddr_storage foreign; /* 相手のアドレス情報 */
34   struct addrinfo *local;          /* 自分のアドレス情報 */
35   struct addrinfo hints;           /* getaddrinfo への指示 */
36   socklen_t len;                   /* ソケット構造体の長さ  */
37   int sock0;                       /* コネクションを受け付けるソケット */
38   int sock;                        /* 通信用ソケット */
39   char buf[BUFSIZE];               /* データバッファ */
40   int size;                        /* バッファ内のデータの大きさ */
41   int total;                       /* 送信した総バイト数 */
42   char ip[NI_MAXHOST];             /* 相手の IP アドレス (画面表示用) */
43   char port[NI_MAXSERV];           /* 相手のポート番号 (画面表示用) */
44   FILE *fp;
45
46 #ifdef _WIN32
47   WSADATA wsaData;
48   WSAStartup(MAKEWORD(2, 0), &wsaData);
49 #endif
50
51   if (argc != 3) {
52     fprintf(stderr, "Useage: %s [read file] [local port]\n", argv[CMD_NAME]);
53     exit(1);
54   }
55
56   /* 構造体 local に、トランスポートプロトコル情報、自アドレス、自ポート番号を格納 */
57   memset(&hints, 0, sizeof hints);
58 #ifndef UDP
59   hints.ai_socktype = SOCK_STREAM; /* TCP を使用 */
60 #else
61   hints.ai_socktype = SOCK_DGRAM;  /* UDP を使用 */
62 #endif
63   hints.ai_flags = AI_PASSIVE;     /* ソケットをサーバ用途で使用 */
64   getaddrinfo(NULL, argv[LOCAL_PORT], &hints, &local);
65
66   /* ソケットをオープン */
67   sock0 = socket(local->ai_family, local->ai_socktype, local->ai_protocol);
68
69   /* ソケットに自アドレス、自ポート番号を設定 */
70   bind(sock0, local->ai_addr, local->ai_addrlen);
71 #ifndef UDP
72   listen(sock0, 5); /* コネクション受付開始 (TCP) */
73 #endif
74
75   while (1) {
76     fp = fopen(argv[READ_FILE], "rb");
77
```

```
 78     len = sizeof foreign;
 79 #ifndef UDP /* データ通信用コネクション (TCP) */
 80     sock = accept(sock0, (struct sockaddr *) &foreign, &len);
 81 #else       /* 始まりの合図 (UDP)*/
 82     recvfrom(sock0, buf, sizeof buf, 0, (struct sockaddr *) &foreign, &len);
 83     sock = sock0; /* UDP では使うソケットは 1 つ */
 84 #endif
 85
 86     /* クライアントの IP アドレスとポート番号を表示 */
 87     getnameinfo((struct sockaddr *) &foreign, len, ip, sizeof ip , port, sizeof port,
 88                 NI_NUMERICHOST | NI_NUMERICSERV);
 89     printf("IP=%s PORT=%s\n", ip, port);
 90
 91     total = 0;
 92     while ((size = fread(buf, sizeof buf[0], MSGSIZE, fp)) > 0) {
 93 #ifndef UDP /* 送信処理 (TCP) */
 94       send(sock, buf, size, 0);
 95 #else       /* 送信処理 (UDP) ダミートレイラ (4 バイト) を付ける */
 96       sendto(sock, buf, size + DUMMY, 0, (struct sockaddr *) &foreign, len);
 97 #endif
 98       printf("%d ", size);
 99       total += size;
100       fflush(stdout);
101     }
102     printf("\ntotal = %d byte\n", total);
103
104 #ifndef UDP /* 通信ソケットのクローズ (TCP) */
105     close(sock);
106 #else       /* 終わりの合図 (UDP) */
107     sendto(sock, buf, DUMMY, 0, (struct sockaddr *) &foreign, len);
108 #endif
109     fclose(fp);
110   }
111   /* 無限ループなので以下は処理されない */
112   freeaddrinfo(local); /* getaddrinfo で取得したメモリ領域の開放 */
113   close(sock0);        /* ソケットのクローズ */
114   exit(0);
115
116   return 0;
117 }
```

7-2-3 クライアントプログラム (file_client0.c)

```
 1 /* file_client0.c: TCP/UDP ファイル転送クライアント Ver. 1.0 2011.01.11 */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <string.h>
 5 #ifdef _WIN32
 6 #include <winsock2.h>
 7 #include <ws2tcpip.h>
 8 #define exit(_Z)   WSACleanup();exit(_Z)
 9 #define close(_Z)  closesocket(_Z);
10 #else
11 #include <sys/types.h>
12 #include <sys/socket.h>
13 #include <netdb.h>
14 #include <netinet/in.h>
15 #include <unistd.h>
16 #endif
17
```

```
18 //#define UDP  /* デフォルトはトランスポートに TCP を使用。UDP の場合は//を外す */
19 #define BUFSIZE 8192
20
21 #ifdef UDP
22 #define DUMMY 4
23 #endif
24
25 enum args {CMD_NAME, WRITE_FILE, FOREIGN_IP, FOREIGN_PORT};
26
27 int main(int argc, char *argv[])
28 {
29   struct addrinfo *foreign;  /* 相手のアドレス情報 */
30   struct addrinfo hints;     /* getaddrinfo への指示 */
31   int sock;                  /* 通信用ソケット */
32   char buf[BUFSIZE];         /* データバッファ */
33   int size;                  /* バッファ内のデータの大きさ */
34   int total;                 /* 受信した総バイト数 */
35   FILE *fp;
36
37 #ifdef _WIN32
38   WSADATA wsaData;
39   WSAStartup(MAKEWORD(2, 0), &wsaData);
40 #endif
41
42   if (argc != 4) {
43     fprintf(stderr, "Useage: %s [write file] [foreign ip] [foreign port]\n", argv[CMD_NAME]);
44     exit(1);
45   }
46   fp = fopen(argv[WRITE_FILE], "wb");
47
48   /* 構造体 foreign に、トランスポートプロトコル情報、相手の IP アドレス・ポート番号を格納 */
49   memset(&hints, 0, sizeof hints);
50   hints.ai_family = PF_UNSPEC;     /* ネットワーク層プロトコルは無指定 */
51 #ifndef UDP
52   hints.ai_socktype = SOCK_STREAM; /* TCP を使用 */
53 #else
54   hints.ai_socktype = SOCK_DGRAM;  /* UDP を使用 */
55 #endif
56   getaddrinfo(argv[FOREIGN_IP], argv[FOREIGN_PORT], &hints, &foreign);
57
58   /* ソケットをオープン */
59   sock = socket(foreign->ai_family, foreign->ai_socktype, foreign->ai_protocol);
60
61   /* 通信相手固定 */
62   connect(sock, foreign->ai_addr, foreign->ai_addrlen);
63 #ifdef UDP /* 始まりの合図 (UDP) */
64   send(sock, buf, DUMMY, 0);
65 #endif
66
67   total = 0;
68   while ((size = recv(sock, buf, BUFSIZE, 0)) > 0) { /* データの受信処理 */
69 #ifdef UDP /* ダミートレイラ (4 バイト) を外す (UDP) */
70     if ((size -= DUMMY) <= 0)
71       break;
72 #endif
73     fwrite(buf, sizeof buf[0], size, fp);
74     printf("%d ", size);
75     total += size;
76     fflush(stdout);
77   }
78   printf("\ntotal = %d byte\n", total);
79
```

```
80   freeaddrinfo(foreign); /* getaddrinfo で取得したメモリ領域の開放 */
81   close(sock);           /* ソケットのクローズ (コネクションの切断) */
82   fclose(fp);
83   exit(0);
84
85   return 0;
86 }
```

7-2-4 コンパイルと実行の方法

(1) コンパイル方法

コンパイルするためには Linux，MacOS X などの UNIX 系 OS では，ターミナルから次のように入力する．

```
cc -o file_server0 file_server0.c
cc -o file_clinet0 file_clinet0.c
```

Windows 系では Visual Studio Tools に含まれるコマンドプロンプトから，次のように入力する（GUI を使用するときにはリンカで入力する依存ファイルに `ws2_32.lib` を追加してからビルドする）．

```
cl ws2_32.lib file_server0.c
cl ws2_32.lib file_clinet0.c
```

本プログラムを無変更でコンパイルすると TCP を使ったプログラムになる．UDP を使う場合には，サーバプログラム (`file_server0.c`) の 19 行目とクライアントプログラム (`file_client0.c`) の 18 行目のコメントアウトを外してからコンパイルする．

(2) 実行方法

サーバは次のように実行する（Windows 系では行頭の ./ は不要）．

```
./file_server0  読み込むファイル名 ポート番号
```

このサーバは `getaddrinfo()` で得られた `addrinfo` 構造体の先頭のプロトコルファミリーでのみ `bind()` する．このため，IPv6 と IPv4 の両方に対応したシステムの場合，IPv6，もしくは IPv4 のどちらかだけでしか通信できない場合がある．IPv6 射影アドレスに対応しているシステムの場合には，IPv6 と IPv4 のどちらでも通信できる場合がある．

クライアントは次のように実行する．

```
./file_client0  書き込むファイル名 サーバの IP アドレス サーバのポート番号
```

「サーバの IP アドレス」の部分にはホスト名やドメイン名を記述しても動作する．ホスト名やドメイン名を記述した場合，DNS サーバなどに問い合わせて取得できた最初の一つ目のアドレスにのみ接続を試みるようになっている．より実用的なプログラムを目指す場合には，成功するまで順番にアドレスを変えて再試行するようにする．

7-2-5 TCP と UDP での処理内容の違い

本プログラムでは TCP と UDP で処理内容に違いがあるので，それについて説明する．

(1) 通信開始と終了の合図

TCP の場合，TCP のコネクションの確立と切断をファイル転送の開始と終了の合図にしている．これは FTP [3] で利用されている方法である．しかし，UDP はコネクションレスのため TCP と同じ方法が使えない．本プログラムでは，UDP の場合には，ファイルデータの送受信の前後に，開始と終了を表す合図メッセージを送信している（それぞれ `file_client0.c` 64 行目，`file_server0.c` 107 行目）．合図メッセージとファイルデータを識別する必要があるが，本プログラムではメッセージの大きさに着目して区別している．合図メッセージは 4 バイト，ファイルデータは末尾に 4 バイトのダミーデータを付けて「データサイズ＋ 4 バイト」になるようにした*．これにより受信したサイズが 5 以上ならばデータ，4 以下ならば合図だと分かる．ファイルデータの受信時には末尾のダミーデータを取り除いた部分のみをファイルに保存する．この方法はデータグラム型の UDP のみで利用でき，ストリーム型の TCP では利用できない．UDP で送受信するメッセージの大きさの違いにより，データの途中か終了かを判別する方法は TFTP [4] でも利用されている．

(2) メッセージの送信サイズ，受信サイズ

TCP では `send()` も `recv()` も 8192 バイト単位で行うようにしている．これに対して UDP では `send()` は 1028 バイト単位（データ 1024，ダミー 4）で行っている．TCP には IP フラグメントを抑制する機能があり，またフロー制御もあるため，通信効率を高めるために大きめのバッファサイズを指定して送受信を行った．UDP には IP フラグメントを抑制する機能がない．IP フラグメントが発生するとパケットロス時の損失が大きくなるため，Ethernet でフラグメントが発生しないサイズを使用した．ファイルデータ読み書き用のバッファサイズとして切りのよい 1024 バイト（1 K バイト）を使用し，それにダミーデータを加える形にした．

7-2-6 UDP の connect について

クライアントプログラムでは TCP と UDP のプログラムの違いを小さくするために, UDP でも `connect()` を使用して相手のアドレスを固定している．しかしながら, UDP では `connect()` を使わない実装は多い．`connect()` を使わない場合には 62 行目を消去し，64 行目と 68 行目の `send()`，`recv()` をそれぞれ `sendto()`，`recvfrom()` に変更する必要がある．

なお，UDP では `connect()` した場合としていない場合で ICMP エラー†受信時の振る舞いが変化する．`connect()` した場合には，ICMP エラーを受信するとソケットがクローズし，以降 `send()` も `recv()` もできなくなる．`connect()` していない場合には ICMP エラーを受信しても無視されソケットはクローズしないため `recvfrom()` や `sendto()` には影響しない．

---

* UNIX 系では 0 バイトのメッセージを送受信できるためダミーデータを 0 バイトにしても正常に動作するが，Windows 系では無視され送信も受信もできない．

† UDP に関係する ICMP エラーは，ICMP Type 3 (Destination Unreachable)，Code 3 (port unreachable).

■トランスポートサービス - 7 章

7-3 プログラミングの側面から見た TCP/UDP の違い

（執筆者：村山公保）[2011 年 2 月 受領]

この節では，7-2 節で示したプログラムの実行例から，TCP と UDP のプログラミングの違いについて説明する．

7-3-1 実行例

サーバ側では，1 行目はクライアントの IP アドレスとポート番号，2 行目以降は send() で送信したバイト数（send() した回数表示），最終行は送信した総バイト数が表示される．

クライアント側では，1 行目は recv() で受信したバイト数（recv() した回数表示），最終行は受信した総バイト数が表示される．

なお，UDP の場合，実際にはダミーデータを 4 バイト送っている．表示される数字はダミーデータを含まないバイト数になっている．

(1) TCP の場合の実行例

クライアントの実行例

```
$ ./file_clinet0 file 192.168.0.28 55555
1448 1448 8192 1944 4800 4344 1448 1877
total = 25501 byte
```

サーバの実行例

```
$ ./file_server0 file 55555
IP=::ffff:192.168.3.54 PORT=56423
8192 8192 8192 925
total = 25501 byte
```

(2) UDP の場合の実行例

クライアントの実行例

```
$ ./file_clinet0 file 192.168.0.28 55555
1024 1024 1024 1024 1024 1024 1024 1024 102
4 1024 1024 1024 1024 1024 1024 1024 1024 1
024 1024 1024 1024 1024 925
total = 24477 byte
```

サーバの実行例

```
$ ./file_server0 file 55555
IP=::ffff:192.168.3.54 PORT=56442
1024 1024 1024 1024 1024 1024 1024 1024 102
4 1024 1024 1024 1024 1024 1024 1024 1024 1
024 1024 1024 1024 1024 1024 925
total = 25501 byte
```

7-3-2 データグラム型とストリーム型

TCP はストリーム型で UDP はデータグラム型である．sockaddr 構造体の ai_socktype に設定する値がそれぞれ SOCK_STREAM，SOCK_DGRAM になっているのも，このことに由来する．

図 7･1 は，7-3-1 項の実行例を得たときの，send()，recv() とパケットの流れの関係を図示したものである．send() で送信したメッセージサイズ，実際にネットワーク中を流れたメッセージのサイズ，recv() で受信したメッセージサイズを記述している．

TCP では send() と recv() が 1 対 1 に対応しない．TCP は，MSS（Maximum Segment Size）を決定し，Nagle アルゴリズム，スロースタート，ウィンドウ制御，順序制御，再送制

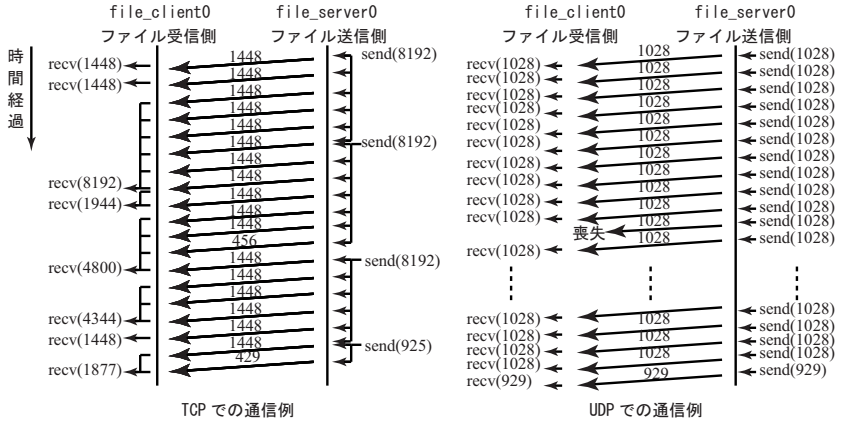

図 7·1　パケットの流れ方の例

御があるため，複数回分の send() のメッセージがまとめられて一つの TCP セグメントで送信されたり，1 回の send() で指定したメッセージが複数の TCP セグメントに分割されて送信されることがある．

recv() は，OS のバッファに届いているメッセージをアプリケーションのメモリに格納する．recv() がパケット到着よりも早い周期で行われると受信メッセージは 1MSS に近くなり，パケット到着よりも遅い周期で行われると受信サイズは recv() で指定したバッファサイズに近くなる．

これに対して UDP では，send() と recv() が 1 対 1 に対応する．複数の UDP データグラムに分割されたり，結合されたりしない*．UDP では信頼性が提供されないため，send() されたメッセージが recv() できるとは限らず，欠落する場合がある．UDP を使用した場合，コンピュータ内部で UDP データグラムが喪失することがある．このため，同一のホスト内でローカルループバックによる通信を行っても，UDP ではパケットが欠落することがある．

7-3-3 TCP のコネクション切断と信頼性

「TCP のコネクションの切断」をファイル転送の終了の合図に使うのは，ファイルの受信側では問題とならないが，ファイルの送信側では問題となる可能性がある．recv() は戻り値をチェックするとコネクションが正常切断したか分かる．しかし，send() は分かるとは限らない．

send() は，送信メッセージが OS のバッファにコピーできれば正常値を返す．つまり，send() が終了しても，パケット送信が実際に行われているとは限らない．OS のバッファに格納されたままの状態の場合もあり，send() の戻り値をチェックしてもそのメッセージが相手に届いたかどうか分からない．

* Linux では send() や sendto() で MSG_MORE フラグを付けると，複数回の send() で一つの UDP データグラムを送ることができる．

同様に close() の戻り値をチェックしても，すべてのメッセージが相手に届いたかどうか分からない．デフォルト設定では，未送信のデータが存在していても close() はブロックせず，コネクション切断処理はバックグラウンドで行われる．

TCP のコネクションが正常に切断されたかどうかを知るためには，setsockopt() で SO_LINGER を指定し，長めのタイムアウト時間（例えば 10 分など）を設定する（レベルは SOL_SOCKET）．その後 close() すると，コネクション切断処理が終了するまでブロックし，戻り値でコネクションが正常に切断できたかどうかを知らせてくれる．タイムアウト時間までにコネクションが正常に切断できない場合にはエラーを返す．

7-3-4 TCP のコネクション切断と TIME_WAIT

TCP ではコネクション切断時に**図 7･2** の左のようにパケットが流れる．先に切断を開始した側の状態遷移が TIME_WAIT になり，TCB（Transmission Control Block）を 2 MSL（Maximum Segment Lifetime）の間保持しなければならない[5]．莫大な数のコネクションの接続・切断が短期間に集中して行われると，TIME_WAIT で保持する TCB がシステムリソースを圧迫し，運用上の問題を引き起こすことがある．

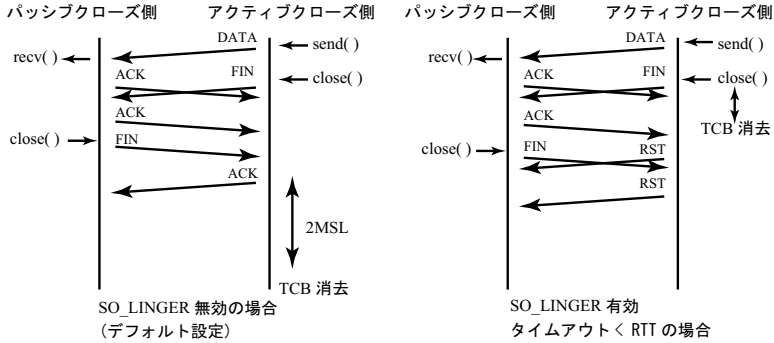

図 7･2　SO_LINGER でタイムアウト時間を短くした場合

これを避けようと setsockopt() で短いタイムアウト時間を設定して SO_LINGER を設定し（レベルは SOL_SOCKET），即座に TCB を消去するような実装が行われることがある．RTT よりもタイムアウト時間が短いと図 7･2 の右のようにパケットが流れ，あとからコネクションを切断した側には TCP RST が返されることになり，TCP コネクションは正常には切断されない．SO_LINGER でタイムアウト時間を 0 に設定して close() すると，FIN セグメントは流れず，RST が流れる．送信バッファ内に未送信のデータセグメントが存在していても，すべてが廃棄され，信頼性は保証されなくなるため，注意が必要である．

■トランスポートサービス - 7 章

7-4 通信性能を考えたプログラミングの方法

（執筆者：村山公保）[2011 年 2 月 受領]

TCP は，Web，ファイル転送，電子メール，遠隔ログイン，リモートデスクトップ，ストリーミングなど，広範囲な用途に利用されている．通信の形式ごとにアプリケーション作成上の注意点がある．本章では TCP の通信をバルクデータ転送形式，リクエスト・レスポンス形式，イベント駆動形式の三つに分類し，それぞれで通信性能を考えたプログラミングの方法について述べる．

7-4-1 バルクデータ転送形式

バルクデータ転送形式では**図 7･3** のようにパケットが流れる*．大きなデータ（最低でも 1 MSS を越える）を転送するときのデータ転送であり，FTP や HTTP によるファイルやデータの転送，SMTP で添付ファイルを含むような大きな電子メールを転送するときにはこの形式になる．本章のプログラムもバルクデータ転送形式になる．

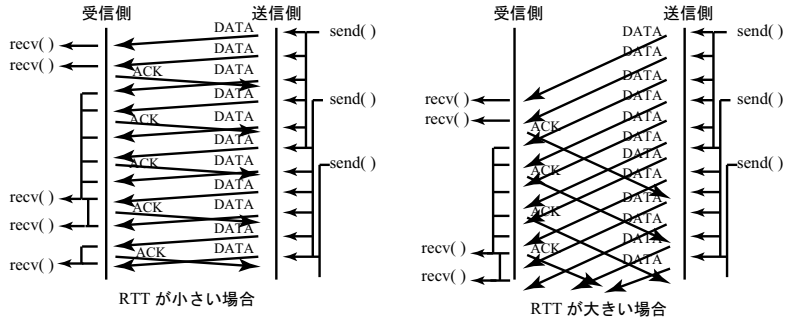

図 7･3　バルクデータ転送のパケットの流れの例

帯域遅延積が大きい通信路（Long Fat Pipe）で高速な通信を実現するためには TCP のウィンドウを大きくする必要がある．ソケットの場合，ウィンドウの最大値は TCP のバッファサイズと連動する．`setsockopt()` で SO_SNDBUF，SO_RCVBUF を指定すると，それぞれ送信バッファ，受信バッファの大きさを変更できる（レベルは SOL_SOCKET）．TCP ヘッダで通知するウィンドウは SO_RCVBUF が関係するが，送信側の輻輳ウィンドウは SO_SNDBUF で指定した値が上限値になるため，受信側の SO_RCVBUF と送信側の SO_SNDBUF の両方の値を調整する必要がある．バッファサイズを 65535 よりも大きな値に設定すると，プロトコルスタックが対応している場合には TCP のウィンドウスケールオプションが利用される．なお，TCP の送受信バッファサイズを自動的にチューニングする手法が提案されており[6]，デフォルト設定で自動チューニングが有効になっている OS も存在する．このような OS では，アプリケーションが TCP バッファサイズを設定することは奨励されない．

このデータ転送では注意しなければならない点がある．バルクデータ転送では，毎回同じ

* スロースタートは考慮してない．

メッセージサイズで send() することが多い．例えば，今回のサンプルプログラムでは 8192 バイト単位で send() している．このときに使用するメッセージサイズによっては通信性能が極端に低下する可能性がある．これは，階層化の問題として古くから知られており[7]，TCP モジュールとソケットのメモリ管理方式が異なるために発生する OS 実装上の問題である．512 バイトという無難そうに思えるサイズを使用しても通信性能が極端に低下する事例があった[8]ため，組込みシステムなど，メモリリソースが少ないシステムであったとしても，メッセージサイズは MSS よりも充分に大きなサイズにすることが望ましい．

より高速化のためには，バッファのメモリ上のアライメント（Alignment）についても考慮した方がよい場合がある．仮想記憶システムの場合，ページ単位でバッファを使用した方が高速化が期待できる．例えば，ページサイズが 4096 バイトのシステムで，バッファサイズを 8192 バイトで使用する場合を考える．次のプログラムはアライメントしていない場合（p1）と，ページ境界になるようにアライメントした場合（p2）の例である．

```
#define BUFSIZE   8192
#define ALIGNMENT 4096
char buf[BUFSIZE + ALIGNMENT], *p1, *p2;
p1 = buf;
p2 = buf + (ALIGNMENT - ((unsigned long) buf % ALIGNMENT));
```

アライメントしていない p1 をバッファの先頭アドレスとして使うと三つのページを使用する可能性が大きくなるが，p2 を使えば二つのページで済むことになる．アライメントをするとメモリ使用量は増えるが，アドレス変換，キャッシュ，DMA などを考慮するとアライメントによって性能が向上する場合もあると考えられる．アライメントをテストする機能はネットワークベンチマークソフトのオプションとして古くから実装されていた [9]．

最適なバッファサイズやアライメントの値は使用するハードウェア，OS，コンパイラ，ライブラリによって異なるため，テストツール [10] などを使って実機でテストをすることが重要になる．

7-4-2 リクエスト・レスポンス形式

リクエスト・レスポンス形式とは，お互いのプログラムが小さなメッセージ（1 MSS 以下）を送りあって通信を行う形式である．これは，SMTP や SIP，FTP で，制御メッセージをやりとりするときの形式である．HTTP で，クライアント側でキャッシュされているページにアクセスしたときもこの形式になる．この通信では気をつけなければならないことがある．それは一つのメッセージを 1 回のシステムコールで送信しないとパフォーマンスが低下するということである．

図 7･4 の左側は，メッセージを常に 1 回のシステムコールで実行した例である．TCP のピギーバックが働くためパケット数が減り，効率の良い通信ができている．

図 7･4 の右側は，2 番目のメッセージを 3 回のシステムコールに分けて送信した例である（※の部分）．この部分は Nagle アルゴリズムの影響で，最初の send() メッセージだけが先に送信され，残りの 2 回分の send() メッセージは確認応答パケットが返ってくるまで遅延してから送信される．受信側ではメッセージ全体がそろわないと処理ができないため，大き

な遅延となっている．

このような遅延を避けるためには，メッセージを必ず 1 回のシステムコールで送信する必要がある．1 区切りのアプリケーションメッセージ全体を連続するメモリ領域にコピーしてから send() するか，複数のメモリ領域を指定できる sendmsg() を使用する．

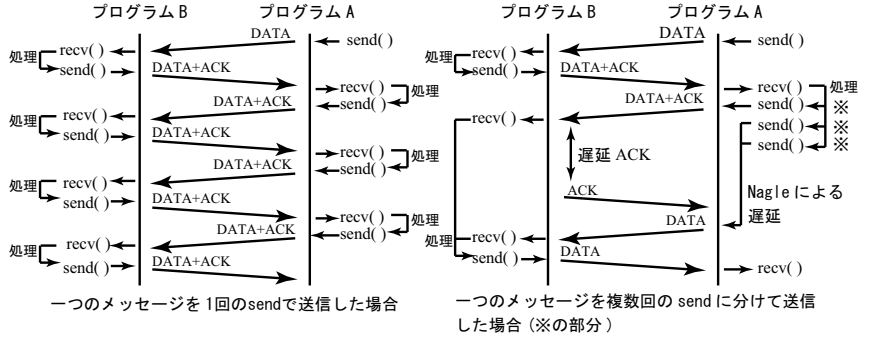

図 7･4　リクエスト・レスポンス形式のパケットの流れの例

7-4-3 イベント駆動形式

イベント駆動形式とは，遠隔ログインやリモートデスクトップなど，イベントにより送信データが生じる通信のことである．イベントが発生しなければ無通信状態になる．即時性が要求されることが多く，遅延の小さな通信が望まれる．イベントが間欠的に発生する場合，通信に必要な帯域の上限が決まっているといえる．音声や動画などのライブ配信もこの形式と考えることができる．

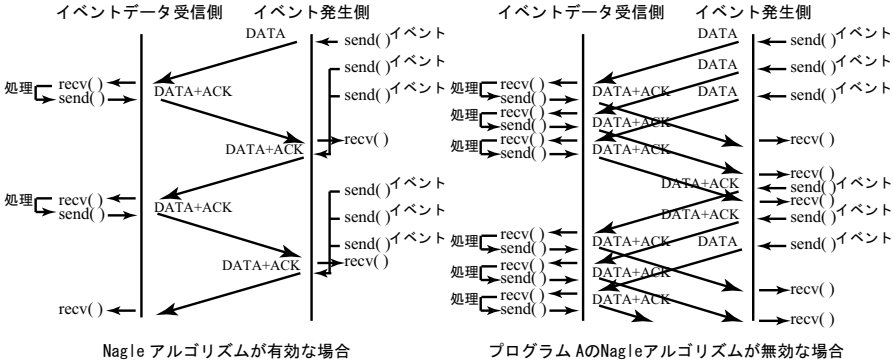

図 7･5　イベント駆動のパケットの流れの例

この形式では TCP の Nagle アルゴリズムが悪影響を及ぼす．**図 7･5** はイベント発生ごとに send() する図である．図 7･5 の左を見ると，Nagle アルゴリズムの影響で，いくつかのメッセージが遅延しており，即時性が損なわれている．

これは Nagle アルゴリズムを無効にすることで回避できる場合がある．図 7･5 の右は Nagle アルゴリズムを無効にした場合である．Nagle アルゴリズムを無効にするには `setsockopt()` で `TCP_NODELAY` を指定すればよい（レベルは `IPPROTO_TCP`）．

Nagle アルゴリズムは，シリーウィンドウシンドロームを回避するために誕生したため，Nagle アルゴリズムを無効にしたい場合には慎重に検討したうえで行わなければならない．

シリーウィンドウシンドロームは，バルクデータ転送で発生する可能性が大きいため，バルクデータ転送を行う場合には Nagle アルゴリズムを無効にしてはならない．HTTP や SMTP は，少量のデータを送るときや制御のときにはリクエスト・レスポンス形式になり，大きなデータを転送するときにはバルクデータ転送になるなど，瞬間瞬間でデータ転送の形式が切り替わる．このようなシステムでは，Nagle アルゴリズムを無効にしない方が安全である．

イベント駆動形式は，使用する通信帯域の上限が，イベントの内容によっておさえられる場合が多く，シリーウィンドウシンドロームが発生する可能性が小さいため，Nagle アルゴリズムを無効にした方がより良い効果を生む場合が多いと考えられる．

■参考文献

1) R. Gilligan, S. Thomson, J. Bound, J. McCann, W. Stevens: “Basic Socket Interface Extensions for IPv6”, RFC 3493, 2003.
2) R. Braden: “Requirements for Internet Hosts – Communication Layers”, RFC 1122, 1989.
3) J. Postel, J. Reynolds: “File Transfer Protocol”, RFC 959, 1985.
4) K. Sollins: “The TFTP Protocol (Revision 2)”, RFC 1350, 1992.
5) J. Postel: “Transmission Control Protocol”, RFC 793, 1981.
6) J. Semke, J. Mahdavi, and M. Mathis, “Automatic TCP buffer tuning”, ACM SIGCOMM’98, pp.315-323, Aug. 1998.
7) Crowcroft, J. I. Wakeman, Zheng Wang, D. Sirovica: “Is Layering Harmful ?”, IEEE Network Magazine, vol.6, pp.20-24, Jan. 1992.
8) 村山公保，西田佳史，尾家祐二：“トランスポートプロトコル”，岩波講座インターネット第 3 巻，岩波書店, 2001.
9) Silicon Graphics, Inc.: “TTCP.C modified in 1989 at Silicon Graphics, Inc.”, ttcp.c, 1989
10) S. Parker, C. Schmechel: “Some Testing Tools for TCP Implementors”, RFC 2398, 1998.