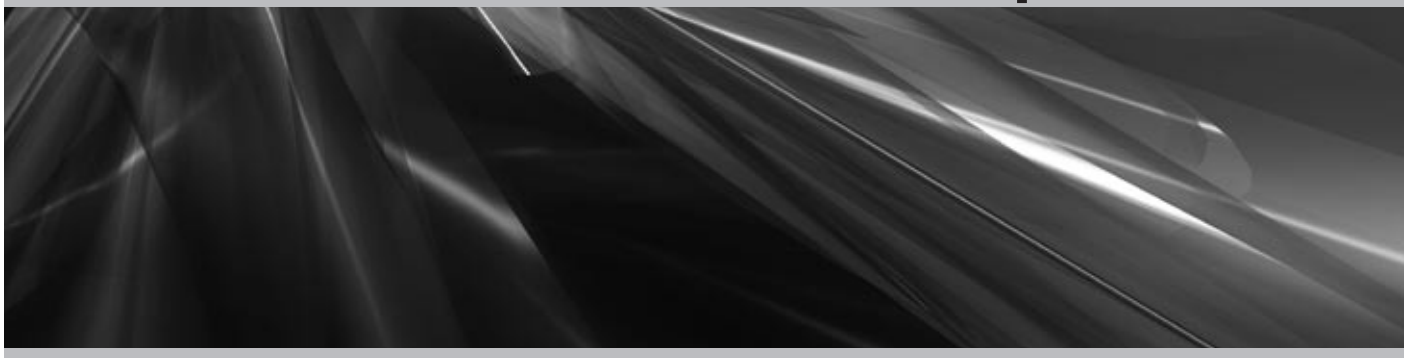
Chapter 3



UDP通信の基礎

UDPの特徴は信頼性を犠牲としたリアルタイム性です。サーバに負荷をかけずに多くの受信者にパケットを送信できるブロードキャストやマルチキャストが利用できるため、音声や映像を転送するアプリケーションなどに使われます。また、DNSに対するqueryにも使われています。

Chapter 3では、まず最初にUDPによる単純な受信サンプルと送信サンプルを示し、次にブロードキャストとマルチキャストによるサンプルプログラムを示します。

3-1 UDPの特徴とプログラミング

ネットワークプログラミングでTCPの次に多い通信方法がUDPだと思われます。

UDPはデータが宛先に届いたかどうかを関知しないため、データの到着を保障しない点がTCPと異なります。そのため、UDPを使った通信を行うプログラムを書く場合には、パケットがネットワークの途中で消えてしまうことも想定しなくてはなりません。確実にデータを届けたいアプリケーションではTCPを使うのが一般的です。

このように紹介するとUDPは使いにくいだけと思うかもしれませんが、もちろん利点もあります。

- **複数の相手に同時にデータを送信できる（ブロードキャスト/マルチキャスト）**
- **TCPよりもリアルタイム性が高い**
- **メッセージの境界が明確**
- **TCPよりもNATを超えやすい（P2Pの場合）**

まず、最初の利点として「複数の相手に同時にデータ送信ができる」ことが挙げられます。Chapter 1で解説したように、IPの通信形態には、「ユニキャスト」「ブロードキャスト」「マルチキャスト」の３種類があります[注3-1]。

TCPでは１対１の通信しかできないため、このうちユニキャストしかサポートしていません。それに対してUDPでは、ひとつデータパケットを送ればネットワークで必要に応じて増やして送ってくるブロードキャストやマルチキャストが利用できます。送信側は受信者数に関係なく必要最小限のパケットだけ送っていれば、あとはネットワークが適切に処理をしてくれるため、送信側のアプリケーションの負荷を大きく軽減できます。

第二の利点としては「リアルタイム性」が挙げられます。TCPはデータの到着を保障するため、ネットワーク内でパケットが消えると再送します。この再送によって、リアルタイム性が損なわれることがあります。また、TCPはネットワークが混雑しているとデータ送信量を減らす「輻輳（ふくそう）制御」を行います。この輻輳制御によっても、TCPのリアルタイム性が損なわれています。

一方、UDPにはリアルタイム性を損なう再送や輻輳制御がないため、UDPはTCPよりもリアルタイム性が高くなります。ただし、UDPには輻輳制御が必要ないわけではないのでご注意ください。TCPは輻輳制御機構がユーザアプリケーションでは関知できない部分（下層レイヤ）で行われるため、アプリケーションに工夫の余地はありません。他方で、UDPではアプリケーションごとに最適な輻輳制御機構を設計し、構築できるという特徴があります。

注3-1：ただし、IPv6にはブロードキャストはありません。代わりに、IPv6には「エニーキャスト」があります（エニーキャストについては本書では割愛します）。

このように、UDPにはTCPにない利点があり、これらを生かしたアプリケーションに使われるのが一般的です。UDPを使ったアプリケーションの例として、映像や音声のストリーミングやIP電話などのVoIP（Voice over IP）などが挙げられます。音声などを使って通話をするアプリケーションは、すべての音が正しく（オリジナルデータに忠実に）届くことよりもリアルタイム性が重視されます。相手の声がわかっても、会話にならないのは困るというわけです。

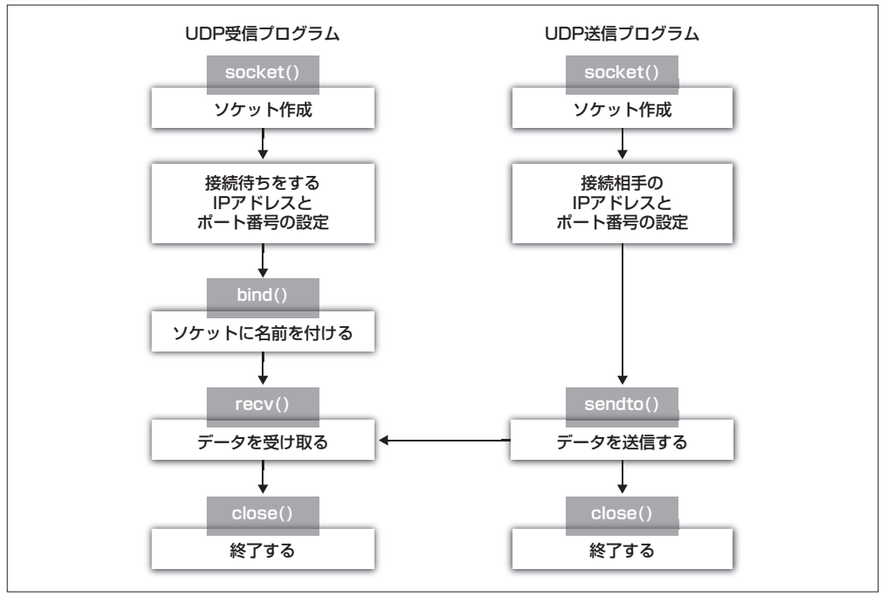
メッセージの境界が明確である点もUDPの特徴のひとつです。TCPの場合は、連続的なストリームとしてデータが扱われるため、アプリケーションが途中で数バイト間違えてデータを読み込んでしまうと、二度とデータの整合性が取れなくなる場合があります（読み込みだけではなく書き込み側が数バイトデータを間違える場合も同様です）。一方、UDPの場合は各メッセージが独立したパケットとして扱われるため、個別のデータ境界が明確となり、途中でデータ読み込みに失敗しても次から復帰することが容易になります。

最後に、UDPはNAPT（NAT）に強いという特徴もあります。通信を行う両端がNATの裏側に存在した場合、TCPでは接続を確立するのが困難ですが、UDPでSTUN（Simple Traversal of UDP through NATs）などを活用することで通信できる場合があります。その仕組みはプログラミングとは関係が薄いのでここでは割愛します。興味のある方はネットワーク系技術書籍などをご覧ください。

3-2 UDPのプログラミング

UDPでは、TCPのように接続を確立してから通信を始めるようなことは行いません。そのため、どちらがサーバでどちらがクライアントか直感的にわからない場合があります。本書ではUDPに関してはサーバ/クライアントという表現をせずに、「UDP送信プログラム」「UDP受信プログラム」という表現をしていきます（図3-1）。

図3-1　UDP の通信



単純な UDP 受信プログラム

UDP通信を行うプログラムは、受信と送信で異なります。最初にUDP受信プログラムを説明します。

UDP受信プログラムは、特定のIPアドレス＋UDPポート番号でパケットを待ち続けます。手順は以下のとおりです。

- ソケットを作る
- IPアドレスとポートを設定する
- ソケットに名前を付ける（bind()する）
- データを受け取る

UDPを利用したソケットプログラミングでは、データを受信する方法としてrecvfrom()やrecv()システムコールを利用することが一般的です。両者の主な違いは、送信側の情報を毎回取得できるかどうかです。

recvfrom()とrecv()の各システムコールは、以下のように宣言されています。

List 3-1　recvfrom()システムコール

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

ssize_t recvfrom(
  int s,                    /* ソケットのファイルディスクリプタ*/
  void *buf,                /* 受信データを取得するためのバッファ*/
  size_t len,               /* bufのサイズ */
  int flags,                /* 受信するときの挙動を指定するためのフラグ */
  struct sockaddr *from,    /* 宛先に関する情報を取得するためのsockaddr構造体 */
  socklen_t *fromlen        /* fromの大きさ */
);
```

List 3-2　recv()システムコール

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

ssize_t recv(
  int s,                    /* ソケットのファイルディスクリプタ */
  void *buf,                /* 受信データを取得するためのバッファ*/
  size_t len,               /* bufのサイズ */
  int flags                 /* 受信するときの挙動を指定するためのフラグ */
);
```

flagsのパラメータとしては、

- MSG_CMSG_CLOEXEC
- MSG_DONTWAIT
- MSG_ERRQUEUE
- MSG_OOB
- MSG_PEER
- MSG_TRUNC
- MSG_WAITALL

などが指定可能です。これらはビット表現されているので、複数の論理和をとったものを指定できます。詳細はman 2 recvfromをご覧ください。

fromlenはINとOUTの双方で利用されるため、ポインタで渡します。recv()あるいはrecvfrom()を行う前にfromlenのための変数にfromのサイズを代入することでカーネルにfromのサイズを知らせ、カーネルは取得されたfromのサイズをfromlenの実体に代入します。

recv()とrecvfrom()は、成功すると受信したバイト数を返します。エラーの場合には-1を返し、errnoが設定されます。

さっそく簡単なUDP受信プログラムのサンプルを以下に示します。ここでは、recvfrom()システムコールを利用しています。繰り返しますが、コードを簡単にするためエラー処理は省

いてあります。実際にコードを書く場合にはエラー処理も行ってください。

List 3-3　単純な UDP 受信プログラム

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

int
main()
{
  int sock;
  struct sockaddr_in addr;
  struct sockaddr_in senderinfo;
  socklen_t addrlen;
  char buf[2048];
  int n;

  sock = socket(AF_INET, SOCK_DGRAM, 0);                          ❶

  addr.sin_family = AF_INET;
  addr.sin_port = htons(12345);                                   ❷
  addr.sin_addr.s_addr = INADDR_ANY;
  bind(sock, (struct sockaddr *)&addr, sizeof(addr));

  addrlen = sizeof(senderinfo);                                   ❸
  n = recvfrom(sock, buf, sizeof(buf) - 1, 0,
          (struct sockaddr *)&senderinfo, &addrlen);

  write(fileno(stdout), buf, n);

  close(sock);                                                    ❹

  return 0;
}
```

上記UDP受信プログラムは、UDP送信プログラムから送られてきたデータを表示して終了します。Chapter 2で解説したTCPによる通信プログラムよりもシンプルであることがわかるでしょう。

まず最初に、IPv4のUDPソケットを作成しています（❶）。AF_INET＋SOCK_DGRAMの組み合わせがIPv4のUDPであることを示しています。次にUDPソケットに対しての設定値をsockaddr_inに対して代入し（❷）、ソケットをbind（）しています。

データを受信し、受信したデータを表示しています（❸）。recvfrom（）のサイズを「sizeof（buf）－1」に指定しているのは、最後の1バイトを必ず「\0」にするためです。これを行わないと、次にくるprintfと％sの組み合わせでバッファオーバフローを発生させる可能性があります。

最後に、ソケットを閉じています（❹）。

単純な UDP 送信プログラム

UDP送信プログラムは、特定のIPアドレス＋UDPポート番号で待っているアプリケーションに対して、パケットを送信します。

- ■ソケットを作る
- ■宛先を指定して送信する

UDPを利用したソケットプログラミングでは、データを送信する方法としてsendto（）やsend（）システムコールを利用することが一般的です。

sendto（）とsend（）の各システムコールは、以下のように宣言されています。

List 3-4　sendto（）システムコール

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

ssize_t sendto(
  int s,                        /* ソケットのファイルディスクリプタ */
  const void *buf,              /* 送信するデータを含むバッファ */
  size_t len,                   /* bufのサイズ */
  int flags,                    /* 送信するときの挙動を指定するためのフラグ */
  const struct sockaddr *to,    /* 宛先に関する情報を示すsockaddr構造体 */
  socklen_t tolen               /* toの大きさ*/
);
```

List 3-5　send（）システムコール

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

ssize_t send(
  int s,                        /* ソケットのファイルディスクリプタ */
  const void *buf,              /* 送信するデータを含むバッファ */
  size_t len,                   /* bufのサイズ */
  int flags                     /* 送信するときの挙動を指定するためのフラグ */
);
```

両者の違いは、send（）システムコールがソケットが接続状態（すでにconnect（）した状態）であることを要求する点です。接続状態にあるソケットであればwrite（）システムコールによるデータ送信も可能ですが、send（）とwrite（）の違いはflagsを指定できるか否かになります。そのため、flagsに0を指定したsend（）システムコールは、write（）と同じ挙動になります。

flagsのパラメータとしては、

- MSG_CONFIRM
- MSG_DONTROUTE
- MSG_DONTWAIT
- MSG_EOR
- MSG_MORE
- MSG_NOSIGNAL
- MSG_OOB

などを指定可能です。これらはビット表現されているので、flagsパラメータには複数の論理和をとったものを指定できます。

sendto()とsend()は、成功すると送信されたバイト数を返します。エラーの場合には-1を返し、errnoが設定されます。errnoの値としては以下のものがあり得ます。

表3-2 sendto()とsend()のerrno値(man 2 sendtoより)

errno値	内容
EACCES	ソケット・ファイルへの書き込み許可がなかったか、パス名へ到達するまでのディレクトリのいずれかに対する検索許可がなかった
EAGAINまたはEWOULDBLOCK	ソケットが非停止に設定されており、要求された操作が停止した
EBADF	無効なディスクリプタが指定された
ECONNRESET	接続が接続相手によりリセットされた
EDESTADDRREQ	ソケットが接続型(connection-mode)ではなく、かつ送信先のアドレスが設定されていない
EFAULT	ユーザ空間として不正なアドレスがパラメータとして指定された
EINTR	データが送信される前に、シグナルが発生した
EINVAL	不正な引数が渡された
EISCONN	接続型ソケットの接続がすでに確立していたが、受信者が指定されていた(注3-2)
EMSGSIZE	そのソケット種別ではソケットに渡されたままの形でメッセージを送信する必要があるが、メッセージが大きすぎるため送信できない
ENOBUFS	ネットワークインターフェースの出力キューがいっぱいである(注3-3)
ENOMEM	メモリが足りない
ENOTCONN	ソケットが接続されておらず、接続先も指定されていない
ENOTSOCK	引数 s がソケットでない
EOPNOTSUPP	引数 flagsのいくつかのビットが、そのソケット種別では不適切なものである
EPIPE	接続指向のソケットでローカル側が閉じられている。この場合、MSG_NOSIGNALが設定されていなければ、プロセスにはSIGPIPEも同時に送られる

単純なUDP送信プログラムのプログラムを以下に示します。ここではUDPソケットに対するconnect()を行わないので、sendto()システムコールを利用しています。実行ファイル名を

注3-2：現在のところ、この状況では、このエラーが返されるか受信者の指定が無視されるかのいずれかとなります。
注3-3：一般的には、一時的な輻輳(congestion)のためにインターフェースが送信を止めていることを意味します。通常、Linuxではこのようなことは起こりません。デバイスのキューがオーバーフローした場合にはパケットは黙って捨てられます。

「a.out」としてコンパイルした場合、「./a.out 127.0.0.1」のように宛先IPv4アドレスを指定して実行してください。

List 3-6 単純なUDP送信プログラム

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

int
main(int argc, char *argv[])
{
  int sock;
  struct sockaddr_in addr;
  int n;

  if (argc != 2) {
    fprintf(stderr, "Usage : %s dstipaddr\n", argv[0]);
    return 1;
  }

  sock = socket(AF_INET, SOCK_DGRAM, 0); ―――――――――――――――――――――――――――――――――❶

  addr.sin_family = AF_INET; ―――――――――――――――――――――――――――――――――――――❷
  addr.sin_port = htons(12345);
  inet_pton(AF_INET, argv[1], &addr.sin_addr.s_addr);

  n = sendto(sock, "HELLO", 5, 0, (struct sockaddr *)&addr, sizeof(addr)); ―❸
  if (n < 1) {
    perror("sendto");
    return 1;
  }

  close(sock); ―――――――――――――――――――――――――――――――――――――――――――❹

  return 0;
}
```

上記UDP送信プログラムは、UDP受信プログラムに対してデータを送信して終了します。この動作を確認するためには、あらかじめUDP受信プログラムを起動しておく必要があります。

まず、IPv4のUDPソケットを作成しています(❶)。AF_INET+SOCK_DGRAMの組み合わせがIPv4のUDPであることを示しています。

❷では、「./a.out 127.0.0.1」として実行した場合、127.0.0.1を宛先に設定しています。127.0.0.1とは、localhostという「自分自身」を示す特殊アドレスです。前述したUDP受信プログラムが別ホストで動作している場合には、このIPアドレスを変更してください。

続いて、「HELLO」という5文字を含むパケットを送信しています(❸)。送信先は、❷で設定したsockaddr_inに入っている宛先です。そして最後にソケットを閉じています(❹)。

UDP送信プログラムは、UDPの受信プログラムよりもさらにシンプルです。特別な準備は行わずに、いきなりパケットを投げられるUDPのプロトコルとしての特徴がそのままコードに出ているといえます。ただし、UDPには信頼性がないため、パケットが相手に届かないことがある点には注意しましょう。

getaddrinfo ()を利用したUDP プログラム

inet_pton () はIPv4/IPv6両方に対応していますが、アドレスファミリを指定しなければならない点や、結果出力先をアドレスファミリに応じて変更しなければならない点で、両方に対応したコードが多少書きにくい点があります。

IPv4/IPv6両用のコードを書くには、getaddrinfo () を活用するのが便利です。

UDPパケット送信側サンプルプログラム

以下に、UDPパケット送信側サンプルプログラムを示します。

List 3-7　getaddrinfo ()を利用したUDP 送信プログラム

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int
main(int argc, char *argv[])
{
  int sock;
  struct addrinfo hints, *res;
  int n;
  int err;

  if (argc != 2) {
    fprintf(stderr, "Usage : %s dst\n", argv[0]);
    return 1;
  }

  /* IPアドレス表記+ホスト名両方に対応 */
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC; /* IPv4/IPv6両方に対応 */ ❶
  hints.ai_socktype = SOCK_DGRAM;
  err = getaddrinfo(argv[1], "12345", &hints, &res);
  if (err != 0) {
    printf("getaddrinfo : %s\n", gai_strerror(err));
    return 1;
  }

  sock = socket(res->ai_family, res->ai_socktype, 0); ❷
  if (sock < 0) {
    perror("socket");
    return 1;
  }

  {
    const char *ipverstr;
    switch (res->ai_family) {
      case AF_INET:
        ipverstr = "IPv4";
        break;
      case AF_INET6:
        ipverstr = "IPv6"; ❸
        break;
      default:
        ipverstr = "unknown";
        break;
    }
    printf("%s\n", ipverstr);
  }

  n = sendto(sock, "HELLO", 5, 0, res->ai_addr, res->ai_addrlen); ❹
  if (n < 1) {
    perror("sendto");
    return 1;
  }

  close(sock); ❺
  freeaddrinfo(res);

  return 0;
}
```

まず、AF_UNSPEC＋SOCK_DGRAMでgetaddrinfo () を実行しています (❶)。AF_UNSPECを指定することで、IPv4/IPv6両用になっています。このAF_UNSPECをAF_INETにするとIPv4固定に、AF_INET6にするとIPv6固定にできます。getaddrinfo () の第二引数は宛先ポート番号を文字列表現したものです。

❷では、getaddrinfo () の結果をそのまま利用してソケットを作成しています。こうすることで、getaddrinfo () の結果がIPv4/IPv6どちらであっても、適切なソケットを作成できます。

ここで、解決した宛先アドレスがIPv4かIPv6かをprintf () で表示するコードを追加してみました (❸)。この部分は必ずしも必要ではないのでご注意ください。

❹ではgetaddrinfo () の結果をそのまま利用してsendto () を実行しています。

最後に、ソケットを閉じて、getaddrinfo () によって確保されたaddrinfo構造体のメモリを解放しています (❺)。freeaddrinfo () を行うタイミングはclose () より前でも大丈夫です。

UDPパケット受信側サンプルプログラム

次は、受信側のプログラムです。

List 3-8　getaddrinfo ()を利用したUDP受信プログラム

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int
main()
{
  int sock;
  struct addrinfo hints, *res;
  int err, n;
  char buf[2048];

  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_INET; /* このサンプルはIPv4で作成 */ ❶
  hints.ai_socktype = SOCK_DGRAM;
  hints.ai_flags = AI_PASSIVE;
  err = getaddrinfo(NULL, "12345", &hints, &res);
  if (err != 0) {
    printf("getaddrinfo : %s\n", gai_strerror(err));
    return 1;
  }

  sock = socket(res->ai_family, res->ai_socktype, 0); ❷
  if (sock < 0) {
    perror("socket");
    return 1;
  }

  if (bind(sock, res->ai_addr, res->ai_addrlen) != 0) {
    perror("bind");
    return 1;
  }

  freeaddrinfo(res); ❸

  memset(buf, 0, sizeof(buf));
  n = recv(sock, buf, sizeof(buf), 0); ❹

  printf("%s\n", buf);

  close(sock);

  return 0;
}
```


まず、AI_PASSIVEを利用してgetaddrinfo ()を実行しています(❶)。このgetaddrinfo ()の結果を、socket ()とbind ()の引数としてそのまま利用しています(❷)。

続いてgetaddrinfo ()によって確保されたメモリを解放しています(❸)。recv ()を利用してデータを受信し、それを表示したあと、ソケットを閉じています(❹)。

IPv6が未設定の環境を想定する

List 3-7では、getaddrinfo ()が返す最初の結果を、そのまま利用してソケットを作成し、sendto ()によるパケット送信を行っています。そのため、たとえば、IPv6設定を行っていない環境でgetaddrinfo ()が最初に結果として返したものがIPv6だった場合、sendto ()は失敗してしまいます。このような状況は、IPv4を利用してIPv6の名前解決が可能な環境で発生します。

この問題を回避するために、sendto ()を一度実行してみて、その結果を確認したうえで利用するgetaddrinfo ()の結果を決定するという方法があります。Chapter 2では、TCPとgetaddrinfo ()を解説する際に、getaddrinfo ()の各結果に対してconnect ()を行い、成功したものを最終的に利用していました。UDPソケットにおいても、TCPのサンプルプログラム同様にconnect ()を使う方法もありますが、ここではconnect ()の代わりにsendto ()の結果を確認しています。

なお、次のサンプルプログラムはsendto ()を一度行った直後にclose ()によってソケットを閉じてしまいますが、必要に応じてソケットを閉じずに使うプログラムに変更してください。

List 3-9　IPv6が未設定の環境を想定する

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int
main(int argc, char *argv[])
{
  int sock;
  struct addrinfo hints, *res0, *res;
  int n;
  int err;

  if (argc != 2) {
    fprintf(stderr, "Usage : %s dst\n", argv[0]);
    return 1;
  }

  /* IPアドレス表記+ホスト名両方に対応 */
```

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC; /* IPv4/IPv6両方に対応 */ ❶
hints.ai_socktype = SOCK_DGRAM;
err = getaddrinfo(argv[1], "12345", &hints, &res0);
if (err != 0) {
  printf("getaddrinfo : %s\n", gai_strerror(err));
  return 1;
}

for (res = res0; res != NULL; res = res->ai_next) { ❷

  sock = socket(res->ai_family, res->ai_socktype, 0); ❸
  if (sock < 0) {
    perror("socket");
    return 1;
  }

  n = sendto(sock, "HELLO", 5, 0, res->ai_addr, res->ai_addrlen); ❹
  if (n < 1) {
    perror("sendto");
  }

  close(sock); ❺

  if (n > 0) {
    break; ❻
  }
}

freeaddrinfo(res0);

return 0;
}
```

このサンプルプログラムでは、まずgetaddrinfo ()に渡すaddrinfo構造体のパラメータを設定しています(❶)。ここでは、AF_UNSPECを指定することによってIPv4とIPv6両方に対応し、SOCK_DGRAMにすることによってUDPを指定しています。

❷の部分は、getaddrinfo ()の結果を最初から順番に試すためのforループです。このforループは、sendto ()が成功するまで繰り返されます。

forループのなかでは、まず最初にgetaddrinfo ()の各結果が示すアドレスファミリによるソケットを作成しています(❸)。さらに、そのソケットを利用してsendto ()を行い(❹)、そのままソケットをclose ()しています(❺)。

sendto ()が成功した場合、そのままforループを抜けています(❻)。

3-3 ブロードキャストプログラミング

次に、ひとつのパケットをある特定のネットワークに対する「すべてのホスト」宛に送信する、「ブロードキャストアドレス」に対して送信する方法を説明します。

ブロードキャスト送信プログラム

UDPでブロードキャストを利用したパケット送信をするときに必要な操作は2つあります。

- 1) 宛先IPアドレスをブロードキャストIPアドレスにする
- 2) setsockopt ()を利用してソケットにSO_BROADCASTを設定する(setsockopt ()についてはChapter 6を参照)

何もしない状態のUDPソケットでは、ブロードキャストパケットを送信できません。setsockopt ()を利用してソケットに対してSO_BROADCASTを設定することで、ブロードキャストパケットを送信できるようになります。この設定を行わずにブロードキャストパケットを送ろうとすると、sendto ()が失敗してしまいます。

以下に、ブロードキャストパケットを送信するプログラムを示します。

List 3-10　ブロードキャスト送信プログラム

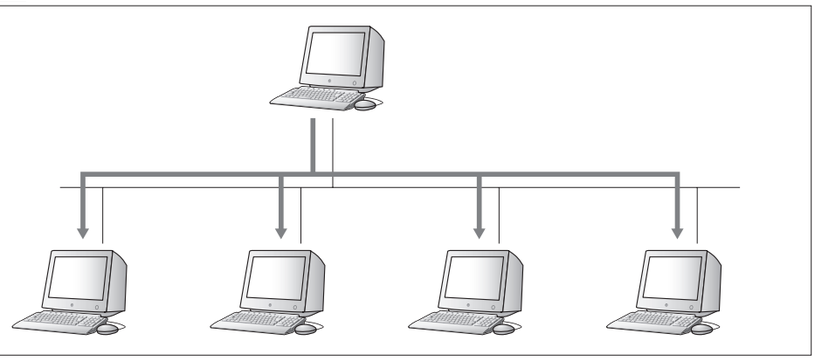
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int
main()
{
  int sock;
  struct sockaddr_in addr;
  int yes = 1;
  int n;

  sock = socket(AF_INET, SOCK_DGRAM, 0);

  addr.sin_family = AF_INET;
  addr.sin_port = htons(12345);
```

```
  inet_pton(AF_INET, "255.255.255.255", &addr.sin_addr.s_addr);

  if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST,
     (char *)&yes, sizeof(yes)) != 0) {
    perror("setsockopt");
    return 1;
  }

  n = sendto(sock, "HELLO", 5, 0, (struct sockaddr *)&addr, sizeof(addr));
  if (n < 1) {
    perror("sendto");
    return 1;
  }

  close(sock);

  return 0;
}
```

このサンプルによって送信されるパケットの受信は、本Chapterの最初に登場した単純なUDP受信プログラム（List 3-3）で行えます。通常の設定では、ブロードキャストであるかないかに関わらず、指定されたUDPポートに届くパケットはそのまま受信されます。

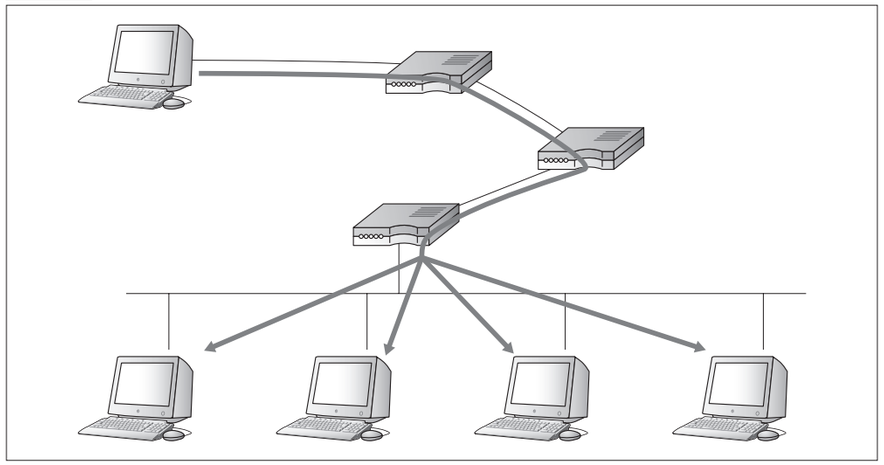
このプログラムで利用している「255.255.255.255」という宛先（IPアドレス）は、サブネット内のすべてのノードを表します。255.255.255.255が宛先のパケットは、ルータを越えません。

図3-2 255.255.255.255 宛のブロードキャスト



ブロードキャストで注意が必要なのは、ルータを越えるブロードキャストです。ルータを越えるブロードキャストである「サブネットブロードキャスト」は、IPアドレスのホスト部をすべて1にしたアドレスです。たとえば、「192.168.0.0/24」というサブネットに対するサブネットブロードキャストは「192.168.0.255」となります。

図3-3 サブネットブロードキャスト



昔は普通にサブネットブロードキャストを使えていましたが、DoS（Denial of Service）攻撃などに多用されるようになったため、現在はルータでのブロードキャストパケット転送を許可しないのが一般的です。送信側ホストからブロードキャストパケットが出ているにも関わらず、受信側ホストにパケットが届かない場合には、受信側のひとつ手前のルータが転送しているかどうかも疑ってみてください。

COLUMN

サブネットブロードキャストを利用したDoS

昔、送信元IPアドレスを攻撃対象となる標的のものに偽装して、まったく関係がない第三者のサブネットブロードキャスト宛にICMP Echo要求を送信するというDoS攻撃が多く発生しました。その仕組みは以下のようなものです。

まず、最初に攻撃者は攻撃を行いたい相手のIPアドレスを偽装してICMP Echo要求パケットをサブネットブロードキャストで送信します。サブネットブロードキャストでICMP Echo要求を受け取った各ホストは、ICMP Echo要求に対して応えます。すると、攻撃者から出されたひとつのパケットが数倍に増えて標的に送信されてしまいます。

このような作業を攻撃者が繰り返すと、標的のネットワーク資源が浪費されてしまい、標的が通信を行えない状態が続いてしまいます。サブネットブロードキャストによって、まったく関係のない第三者が加害者になってしまうため、現在のほとんどのルータではサブネットブロードキャストは無効に設定されています。

3-4 マルチキャストプログラミング

マルチキャストも、ブロードキャスト同様にひとつのパケットが複数人に届くことがあります。ブロードキャストが「送信者が宛先を指定する」のに対して、マルチキャストは「欲しい人に届く」という違いがあります。

ここではマルチキャストプログラミングを知るための基本を解説します。マルチキャストそのものに関する概要は1-4節を、さらに詳細なマルチキャストプログラムについてはChapter 13をそれぞれご覧ください。

マルチキャスト送信プログラム

マルチキャストは非常に複雑で難しい技術です。しかし、マルチキャストの難しさの多くはネットワーク側にあるため、マルチキャストを利用するアプリケーション作成は非常にシンプルです。

以下に、マルチキャスト送信プログラムのサンプルプログラムを示します。

List 3-11　マルチキャスト送信プログラム

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

int
main()
{
  int sock;
  struct sockaddr_in addr;
  int n;

  sock = socket(AF_INET, SOCK_DGRAM, 0);

  addr.sin_family = AF_INET;
  addr.sin_port = htons(12345);
  inet_pton(AF_INET, "239.192.1.2", &addr.sin_addr.s_addr);

  n = sendto(sock, "HELLO", 5, 0, (struct sockaddr *)&addr, sizeof(addr));
  if (n < 1) {
    perror("sendto");
    return 1;
  }

  close(sock);

  return 0;
}
```

ブロードキャストのときとは違い、setsockopt () などによる特殊な処理は必要とせず、送信先のIPアドレスをマルチキャストアドレスにするだけで送信可能です。

ただし、sendto () が正しく動作するためには経路表に適切な経路が記載されている必要があります。経路表に経路がなくてもマルチキャストパケットを送信したい場合には、IP_MULTICAST_IFをsetsockopt()で利用します。IP_MULTICAST_IFに関してはChapter 13で解説します。

IPv6によるマルチキャストパケット送信を行う場合、socket () システムコールの第一引数をAF_INETからAF_INET6に変更し、宛先を示すsockaddr_inをsockaddr_in6に変更したうえで、マルチキャストアドレスもIPv6のマルチキャストアドレスに変更してください。作成したプログラムを実行する際には、IPv6に関する適切な経路が経路表に存在しているかどうかの確認も忘れずに行いましょう。

マルチキャスト受信プログラム

次に、マルチキャストを受信するプログラムです。以下のサンプルプログラムは、前述したマルチキャスト送信プログラムからのパケットを受け取ります。

このサンプルではgetaddrinfo () を利用しています。getaddrinfo () を利用せずに、inet_pton () などを利用すればもう少し簡潔なコードを書くことが可能ですが、IPv6対応を行いやすいように、あえてgetaddrinfo () を利用してみました。

List 3-12　マルチキャスト受信プログラム

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int
main()
{
  int sock;
  struct addrinfo hints, *res;
```

```
int err, n;
struct group_req greq;
char buf[2048];

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
err = getaddrinfo(NULL, "12345", &hints, &res);
if (err != 0) {
  printf("getaddrinfo : %s\n", gai_strerror(err));
  return 1;
}

sock = socket(res->ai_family, res->ai_socktype, 0);
if (sock < 0) {
  perror("socket");
  return 1;
}

if (bind(sock, res->ai_addr, res->ai_addrlen) != 0) {
  perror("bind");
  return 1;
}

freeaddrinfo(res);


memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_DGRAM;
err = getaddrinfo("239.192.1.2", NULL, &hints, &res);
if (err != 0) {
  printf("getaddrinfo : %s\n", gai_strerror(err));
  return 1;
}

memset(&greq, 0, sizeof(greq));

greq.gr_interface = 0; /* 任意のネットワークインターフェースを利用 */

/* getaddrinfo()の結果をgroup_req構造体へコピー */
memcpy(&greq.gr_group, res->ai_addr, res->ai_addrlen);
freeaddrinfo(res);

/* MCAST_JOIN_GROUPを利用してマルチキャストグループへJOIN */
 if (setsockopt(sock, IPPROTO_IP, MCAST_JOIN_GROUP, (char *)&greq,
sizeof(greq)) != 0) {
  perror("setsockopt");
  return 1;
}
```



```
  memset(buf, 0, sizeof(buf));
  n = recv(sock, buf, sizeof(buf), 0);

  printf("%s\n", buf);

  close(sock);

  return 0;
}
```

先ほど触れたように、マルチキャストパケットを受け取るには、マルチキャストグループに「参加(JOIN)」しなくてはなりません。マルチキャストグループへの参加には、IPPROTO_IP＋MCAST_JOIN_GROUPでsetsockopt()を利用します。

MCAST_JOIN_GROUPで利用するgroup_req構造体は、/usr/include/netinet/in.h内で以下のように宣言されています。

List 3-13　group_req 構造体

```
/* Multicast group request.  */
struct group_req
{
  uint32_t gr_interface;              /* インターフェース番号 */
  struct sockaddr_storage gr_group;  /* グループアドレス */
};
```

setsockopt()は、bind()を行ったあとで利用する必要があります。そしてマルチキャストグループへの参加を行うためには、参加するグループを表わすマルチキャストアドレスと利用するインターフェースを指定しなくてはなりません。

インターフェースに0を指定することにより、明示的に利用するインターフェースを指定しないこともできます。ただし、bind()を行ったインターフェースと、MCAST_JOIN_GROUPをgr_interface=0で指定したときのインターフェースが一致する保障はありません。

一度JOINしたマルチキャストグループから「脱退(LEAVE)」するには、setsockopt()にMCAST_LEAVE_GROUPを指定しますが、ソケットをclose()することで同様の効果を得られます。

MCAST_JOIN_GROUPは、IPv4/IPv6両方で利用可能です。group_req構造体のメンバがインターフェース番号やsockaddr_storage構造体である点からも、それがうかがえます(sockaddr_storageに関してはChapter 11を参照)。

これまで、旧APIであるIP_ADD_MEMBERSHIPはip_mreq構造体を利用し、IPV6_ADD_MEMBERSHIPはipv6_mreqを利用するなど、プロトコル依存の構造体を利用していました。

List 3-14　プロトコルに依存した構造体

```
/* IPv4 multicast request.  */
struct ip_mreq
{
  struct in_addr imr_multiaddr;  /* マルチキャストグループのIPアドレス */
  struct in_addr imr_interface;  /* インターフェースのローカルIPアドレス */
};

/* Likewise, for IPv6.  */
struct ipv6_mreq
{
  struct in6_addr ipv6mr_multiaddr; /* マルチキャストグループのIPv6 IPアドレス */
  unsigned int ipv6mr_interface;    /* ローカルインターフェース */
};
```

上記のようにsetsockopt ()に渡す引数が異なっていたので、IP_ADD_MEMBERSHIPとIPV6_ADD_MEMBERSHIPを両方扱うのは面倒だったのですが、MCAST_JOIN_GROUPを使うことによってIPv4/IPv6両用のマルチキャストコードが書きやすくなっています。IP_ADD_MEMBERSHIPとIPV6_ADD_MEMBERSHIPに関するサンプルプログラムは、Chapter 13をご覧ください。

また、setsockopt ()は、bind ()を行ったあとで利用する必要があります。そしてマルチキャストグループへの参加を行うためには、参加するグループを表すマルチキャストアドレスと利用するインターフェースを指定しなくてはなりません。

インターフェースに0を指定することにより、明示的に利用するインターフェースを指定しないこともできます。ただし、bind ()を行ったインターフェースとMCAST_JOIN_GROUPをgr_interface=0で指定したときのインターフェースが一致する保証はありません。

一度JOINしたマルチキャストグループから「脱退(LEAVE)」するには、setsockoptにIP_DROP_MEMBERSHIPを指定します。ソケットをclose ()することでも同様の効果を得られます。

3-5 UDPソケットの「名前」とbind()

ここまで紹介したUDPサンプルプログラムでは、データ送信側はbind ()を行わず、データ受信側だけがbind ()を行っています。しかし、bind ()の役目は「名前を付ける」ことであって、bind ()そのものが送受信の可否を決定付けるものではありません。bind ()を行わなくてもUDPでデータを受信できますし、送信側で意図的にbind ()することもあります。

では、ソケットに対して「名前を付ける」とはどのようなことなのでしょうか？　そのことを説明するために、以下のようなサンプルプログラムを作成しました。

List 3-15　ソケットの名前を取得する

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

void
print_my_port_num(int sock)
{
  struct sockaddr_in s;
  socklen_t sz = sizeof(s);
  getsockname(sock, (struct sockaddr *)&s, &sz);
  printf("%d\n", ntohs(s.sin_port));
}

int
main()
{
  int sock;
  struct sockaddr_in addr;
  int n;

  sock = socket(AF_INET, SOCK_DGRAM, 0);

  print_my_port_num(sock);

  addr.sin_family = AF_INET;
  addr.sin_port = htons(12345);
  inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr.s_addr);

  n = sendto(sock, "HELLO", 5, 0, (struct sockaddr *)&addr, sizeof(addr));
  if (n < 1) {
    perror("sendto");
    return 1;
  }

  print_my_port_num(sock);

  close(sock);

  return 0;
}
```

このサンプルプログラムは、getsockname ()システムコール(Chapter 6を参照)を利用してソケットに付いている「名前」を取得し、そのうちのポート番号部分をprintf ()で表示しています。この「名前の確認」は、以下の2カ所で行われています。

■ socket ()システムコールによってソケットを作成した直後

■ sendto () の直後

このサンプルプログラムを実行すると、最初の部分では「0」というポート番号が表示され、次の部分では何らかのポート番号が表示されます。これにより、作成されたソケットから送信されるUDPパケットの送信元ポート番号は、sendto () システムコールを利用したあとに決定されているのがわかります。

このようなタイミングで送信元ポート番号が付くのは、ポート番号をbind () で指定していないソケットを使おうとすると、カーネル内部で自動的にポート番号が割り当てられるためです。このサンプルではbind () を利用していませんが、「名前が付く」タイミングをこれで理解してもらえるかと思います。

では、次はbind () を行った場合にどうなるかを見てみましょう。

List 3-16　bind () を使った名前確認

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

void
print_my_port_num(int sock)
{
  struct sockaddr_in s;
  socklen_t sz = sizeof(s);
  getsockname(sock, (struct sockaddr *)&s, &sz);
  printf("%d\n", ntohs(s.sin_port));
}

int
main()
{
  int sock;
  struct sockaddr_in addr;
  struct sockaddr_in myname;
  int n;

  sock = socket(AF_INET, SOCK_DGRAM, 0);

  print_my_port_num(sock);

  addr.sin_family = AF_INET;
  addr.sin_port = htons(12345);
  inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr.s_addr);

  myname.sin_family = AF_INET;
  myname.sin_addr.s_addr = INADDR_ANY;
```



```
  myname.sin_port = htons(54321);
  bind(sock, (struct sockaddr *)&myname, sizeof(myname));

  print_my_port_num(sock);

  n = sendto(sock, "HELLO", 5, 0, (struct sockaddr *)&addr, sizeof(addr));
  if (n < 1) {
    perror("sendto");
    return 1;
  }

  print_my_port_num(sock);

  close(sock);

  return 0;
}
```

このサンプルプログラムでは、以下の3カ所でソケットの「名前」を確認しています。

■ 1) socket () システムコールによってソケットを作成した直後
■ 2) bind () の直後
■ 3) sendto () を行ったあと

実際には、1) のタイミングでは「0」と出力されますが、2) と3) のタイミングでは「12345」と出力されます。ここから、最初のソケット作成直後以外は、bind () によって割り当てたポート番号が自分の「名前」の一部として登録されていることがわかると思います。このように、bind () は送信元のポート番号をソケットに対して関連付けます。

何げなく使いがちなbind () システムコールですが、意図的にbind () することで指定どおりの送信元ポート番号でのUDP通信が可能になります。

送信元ポート番号を指定するということは、逆に考えるとデータを受信する待ち受けポート番号の指定でもあるということです。そう考えると、なぜ受信側サンプルプログラムで毎回bind () を行っているのかが見えてきます。送信側プログラムと受信側プログラムは、何らかの方法で利用するUDPポート番号に関しての合意形成があらかじめ確立されており、受信側はそれに従ったポート番号でbind () を行っているというわけです。この「合意形成」としては、たとえば、静的に「このポート番号を使う」とプログラム内に埋め込んだり、プロトコルの中に規定としてポート番号が記述されていたり、動的にポート番号などを割り当てるなどの方法があります。

送信用UDPソケットに対するbind () は、ポート番号を指定するだけではなく、送信元IPアドレスを指定するためにも利用可能です。たとえば、インターフェースが2つ以上存在していて、送信元のIPアドレスを特定のIPアドレスにしたい場合にも、bind () を行ってからsendto () を行うという方法が使えます。

UDPでの「返信」

UDPソケットに対するbind ()の意味がわかったところで、次はやり取りを行うUDPソケットのサンプルプログラムを示したいと思います。

以下のサンプルは、UDPのポート12345で待ち続けます。待ち続けているUDPポート12345にデータが到着し、recvfrom ()によってデータを受け取ると、受け取ったUDPパケットの送信IPアドレス＋送信元ポートが示す送信者に対してsendto ()が行われます。このサンプルプログラムは、このように受け取った相手に対してUDPでそのまま返信するという動作をwhile (1)によって繰り返し続けます。

List 3-17　UDP パケットのやりとりを行うサンプルプログラム

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int
main()
{
  int sock;
  struct sockaddr_in addr;
  struct sockaddr_in senderinfo;
  socklen_t addrlen;
  char buf[2048];
  char senderstr[16];
  int n;

  /* AF_INET+SOCK_DGRAMなので、IPv4のUDPソケット */
  sock = socket(AF_INET, SOCK_DGRAM, 0);

  /* 待ち受けポート番号を12345にするためにbind()を行う */
  addr.sin_family = AF_INET;
  addr.sin_port = htons(12345);
  addr.sin_addr.s_addr = INADDR_ANY;
  bind(sock, (struct sockaddr *)&addr, sizeof(addr));

  while (1) {
    memset(buf, 0, sizeof(buf));

    /* recvfrom()を利用してUDPソケットからデータを受信 */
    addrlen = sizeof(senderinfo);
    n = recvfrom(sock, buf, sizeof(buf) - 1, 0,
                 (struct sockaddr *)&senderinfo, &addrlen);
```



```
    /* 送信元に関する情報を表示 */
    inet_ntop(AF_INET, &senderinfo.sin_addr, senderstr, sizeof(senderstr));
    printf("recvfrom : %s, port=%d\n", senderstr, ntohs(senderinfo.sin_port));

    /* UDPで返信 */
    sendto(sock, buf, n, 0, (struct sockaddr *)&senderinfo, addrlen);

    /* 送信元に関する情報をもう一度表示 */
    printf("sent data to : %s, port=%d\n", senderstr,
           ntohs(senderinfo.sin_port));
  }

  /* このサンプルコードはここへは到達しません */
  close(sock);

  return 0;
}
```

次は、UDPの返信アプリケーションに対してUDPパケットを送信して、返信を受け取る側のプログラムです。このサンプルを実行するには、たとえば「./a.out 127.0.0.1」のように実行時にUDP返信アプリケーションのIPv4アドレスを指定してください。

List 3-18　UDP パケットの送信&受信プログラム

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void
print_my_port_num(int sock)
{
  struct sockaddr_in s;
  socklen_t sz = sizeof(s);
  getsockname(sock, (struct sockaddr *)&s, &sz);
  printf("%d\n", ntohs(s.sin_port));
}

int
main(int argc, char *argv[])
{
  int sock;
  struct sockaddr_in addr;
  struct sockaddr_in senderinfo;
  socklen_t senderinfolen;
  int n;
  char buf[2048];
```

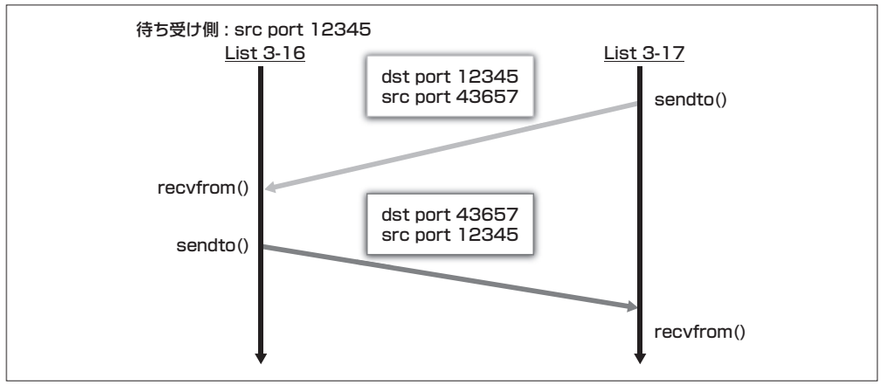
```
if (argc != 2) {
  fprintf(stderr, "Usage : %s dstipaddr\n", argv[0]);
  return 1;
}

sock = socket(AF_INET, SOCK_DGRAM, 0);

addr.sin_family = AF_INET;
addr.sin_port = htons(12345);
inet_pton(AF_INET, argv[1], &addr.sin_addr.s_addr);

n = sendto(sock, "HELLO", 5, 0, (struct sockaddr *)&addr, sizeof(addr));
if (n < 1) {
  perror("sendto");
  return 1;
}

print_my_port_num(sock);

memset(buf, 0, sizeof(buf));
senderinfolen = sizeof(senderinfo);
recvfrom(sock, buf, sizeof(buf), 0,
        (struct sockaddr *)&senderinfo, &senderinfolen);

printf("%s\n", buf);

close(sock);

return 0;
}
```

このサンプルでは、sendto () に利用したソケットをそのまま使ってrecvfrom () を行っています。これにより、UDP返信アプリケーションからの「返信」が、そのままこのソケットに到着します。

また、UDP返信アプリケーションがsendto () していると主張するprintf () の出力結果と比べられるように、getsockname()によって取得した自分のソケットに割り当てられた送信元ポート番号がprintf () で表示されるようにしてあります。

これら2つのサンプルコードの動作は図3-4のようになります。

図3-4　UDP 返信サンプルの動作



待ち受け側であるList 3-17のプログラムがUDPポート番号12345で待ち受けています。List 3-18のプログラムが待ち受けを行っているホストのUDPポート番号12345に対してsendto () を行います。このとき、3-18側の送信元UDPポート番号は43657[注3-4]であるとします。

3-18側からのUDPパケットを、recvfrom () によって受け取った3-17側は、3-18のホスト宛のUDPパケットをポート番号43657で送信します。3-18側は、recvfrom () によって3-17側からのUDPパケットを受け取ります。

さて、このようなUDPによる「返信」ですが、さまざまなところで活用されています。たとえば、ユーザがホストの名前解決を行うときのDNSへのQueryはUDPで行われますが、上記サンプルに近い形で「DNSサーバはQueryを受け取った送信元に対してUDPで返信」が行われます。

また、たとえばUPnP (Universal Plug and Play) のSSDP (Simple Service Discovery Protocol) のようにマルチキャストで機器発見用のQueryがサブネット上に送信され、ネットワーク上の機器側はマルチキャストで送信されたUDPパケットの送信元に対して「UDPで返信を行う」という用途もあります。

3-6 ソケットバッファ

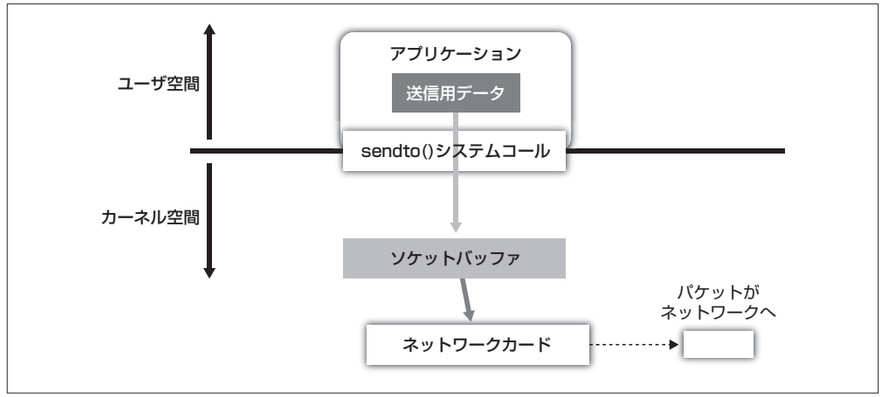
Linuxのカーネルは、sendto () が利用された瞬間に直接ネットワークにパケットを送信しているわけではありません。また、ネットワークからのパケット受信が、そのままrecvfrom () と直結しているわけでもありません。

ユーザからのデータはLinuxカーネルに渡されるとソケットバッファという内部バッファに

注3-4：43657は仮の値です。プログラム実行ごとに3-17側の送信元ポート番号は変化します。

格納され、準備が整った時点でネットワークに送信されます。また、Linuxカーネルはパケットを受信するとソケットバッファに格納します。

図3-5　送信用ソケットバッファと sendto ()システムコール



ソケットバッファは、ネットワークカードのキューに対してパケットの追加を行ったり、逆にキューからデータを取得します。パケットの送受信などのネットワークが関係する処理のタイミングは、ネットワークやネットワークカードの状態に応じて変化します。

このタイミングは、アプリケーションが動作しているプロセスがLinuxカーネルのスケジューラによってCPUを利用できるタイミングとは独立したタイミングで動作しています。そのため、アプリケーションの扱いたいデータをネットワーク経由で送受信するには、Linuxカーネル内にソケットバッファのように一時的に格納できる場所が求められます。

音声や動画などのマルチメディアデータをやり取りするUDPアプリケーションを書いているとき、ソケットバッファの存在がパケット喪失数に反映される場合があります。これは、アプリケーションがrecv () などでカーネル内のソケットバッファからデータを取り出したり、send () などでデータをカーネルに渡すタイミングと実際のパケット処理タイミングがずれたり、ネットワーク上でのデータ到着がバースト状に発生した場合に発生しがちです。

ソケットバッファを設定するには、setsockopt () をSOL_SOCKET＋SO_RCVBUFもしくはSO_SNDBUFで利用します。SO_RCVBUFが受信用のバッファで、SO_SNDBUFが送信用のバッファです。

List 3-19　ソケットバッファの設定

```
int bufsz = 100000;
if (setsockopt(sock, SOL_SOCKET, SO_RCVBUF, (char *)&bufsz, sizeof(bufsz)) !=
0) {
  perror("setsockopt");
  ....エラー処理
}
```


ソケットに設定されているソケットバッファの大きさを知るには、getsockopt () システムコールで SO_SNDBUFやSO_RCVBUFを指定して値を取得します。setsockopt () でSO_SNDBUFやSO_RCVBUFを指定しない状態でのデフォルト値はシステム内で一定です。Linuxカーネルに設定されているデフォルト値を知りたい場合には、以下のコマンドで知ることも可能です。

書き込みバッファ

```
sysctl net.core.wmem_default
```

読み込みバッファ

```
sysctl net.core.rmem_default
```

なお、SO_SNDBUFやSO_RCVBUFに大きめの値を設定したい場合には、ソケットバッファの最大値にもご注意ください。Linuxカーネルの/net/core/sock.cでは、SO_RCVBUFの設定時に以下のように実装されています。

List 3-20　ソケットバッファの最大値

```
case SO_RCVBUF:

/* Don't error on this BSD doesn't and if you think
   about it this is right. Otherwise apps have to
   play 'guess the biggest size' games. RCVBUF/SNDBUF
   are treated in BSD as hints */

  if (val > sysctl_rmem_max)
    val = sysctl_rmem_max;
```

このように、SO_SNDBUFやSO_RCVBUFに与えられた設定値が大きいと、エラーにならずにLinuxカーネルに設定されているソケットバッファ最大値に自動的に変更されます。

ソケットバッファの最大値は以下のコマンドで調べられます。

書き込みバッファ

```
sysctl net.core.wmem_max
```

読み込みバッファ

```
sysctl net.core.rmem_max
```

ソケットバッファに設定可能な最小値にも注意が必要です。/net/core/sock.cでは下限値に関して以下のように処理しています。

List 3-21　ソケットバッファの最小値

```
/*
 * We double it on the way in to account for
 * "struct sk_buff" etc. overhead.   Applications
 * assume that the SO_RCVBUF setting they make will
 * allow that much actual data to be received on that
 * socket.
 *
 * Applications are unaware that "struct sk_buff" and
 * other overheads allocate from the receive buffer
 * during socket buffer allocation.
 *
 * And after considering the possible alternatives,
 * returning the value we actually used in getsockopt
 * is the most desirable behavior.
 */

if ((val * 2) < SOCK_MIN_RCVBUF)
  sk->sk_rcvbuf = SOCK_MIN_RCVBUF;
else
  sk->sk_rcvbuf = val * 2;
break;
```

SOCK_MIN_SNDBUFとSOCK_MIN_RCVBUFは、Linuxカーネルソースの/include/net/sock.hにて以下のように設定されています。

List 3-22　SOCK_MIN_SNDBUF と SOCK_MIN_RCVBUF の既定値

```
#define SOCK_MIN_SNDBUF 2048
#define SOCK_MIN_RCVBUF 256
```

このように、setsockopt ()が成功したように見えても、アプリケーションが設定したいと考えているソケットバッファの大きさが必ずしも設定されているわけではないのでご注意ください。正確にソケットバッファの設定値を知る場合には、setsockopt ()のあとにgetsockopt ()で実際の設定値を確認する必要があります。

3-7 Chapter 3のまとめ

Chapter 3では、UDPによる通信の基礎として、ユニキャスト、ブロードキャスト、マルチキャストについて簡単なサンプルプログラムを示しました。インターネットでは、TCP通信だけでなく、このUDP通信が活用される場面も多々あります。マルチキャストプログラミングについては、Chapter 13で詳しく紹介します。