第6章　ネットワークプログラミングの基礎

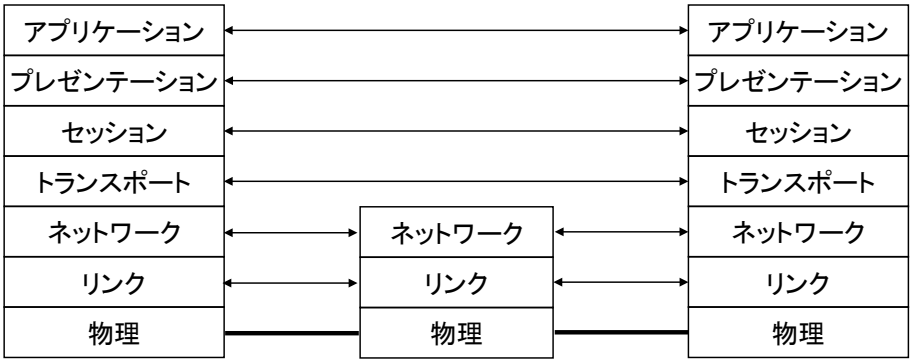
本章からはインターネットを介したプロセス間通信を利用したシステムプログラミングを扱う．3年春学期に「ネットワーク工学 I」を履修していない学生もいると思うので，まずネットワークの基礎，TCP/IP の基礎を復習する．その後，UDP による通信および TCP による通信の基礎について述べる．

6.1　ネットワークの基礎

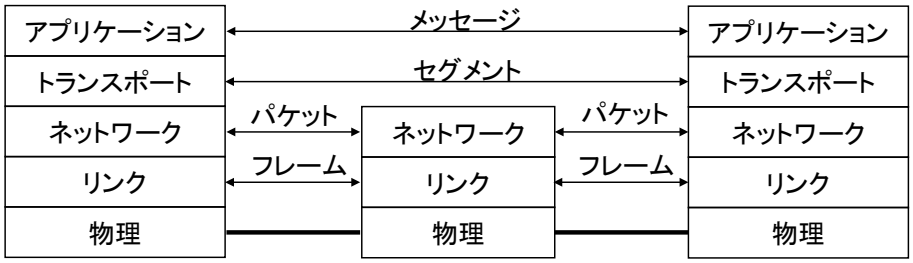
ネットワークを介して通信するにはさまざまな機能や取り決めが必要になる．通信における取り決め (通信規約) を**プロトコル**と呼ぶ．まず物理的にどのような媒体を使うかを決めなければならない．たとえば，より線対，同軸ケーブル，光ファイバー，無線などである．またどのような電圧 (有線媒体の場合) または波長 (光ファイバーや無線の場合) を使うか，情報の符号化にどのような方式を用いるかなども決めなければならない．通信中に符号誤りが発生する場合もあるので，**誤り検出/訂正**を行う機能が必要となる．

ネットワークにおいては通信を行うコンピュータを**ホスト (host)** と呼ぶことが多い．通信を行う2つのホストが1つの媒体で直接接続されていない場合は**ルータ (router)** と呼ばれる装置をいくつか介してこれらのホストが接続される．このような場合，どの経路を利用するかを決定する**経路制御 (routing)** が必要となる．送信ホストは受信ホストのデータ処理能力に合わせて送信スピードを調節する必要がある．これを**流量制御 (flow control)** と呼ぶ．ネットワークには多数のホストが接続され，同時に通信を行っている．多数のホストが一度の大量のデータ通信を行うと，ルータや回線の処理能力がデータ量に追いつかなくなり，データの損失が発生する．このような状態を**輻輳 (ふくそう)(congestion)** と呼ぶ．送信ホストはネットワークが輻輳状態にならないように送信スピードを調節する必要がある．これを**輻輳制御 (congestion control)** と呼ぶ．さらに通信の同期やデータの符号化の取り決めも必要である．アプリケーションプログラムはその機能を実現するためのデータ交換規則を定める必要がある．

以上のように，ネットワークにおいてホスト同士が通信するためにはさまざまな機能が必要となる．これらのすべての機能をアプリケーションプログラムごとに独立に実現していたのでは無駄が多い．そこで必要な機能およびそれを実現するプロトコルを階層的にグループ化したものが**プロトコルの階層化モデル**である．国際標準化機構 (ISO) は **OSI 参照モデル**と呼ばれる7階層モデルを定義している (図 6.1-(a) 参照)．それぞれの階層の役割を以下に示す．



(a) OSI基本参照モデル



(b) インターネットの階層モデル

図 6.1: プロトコル階層

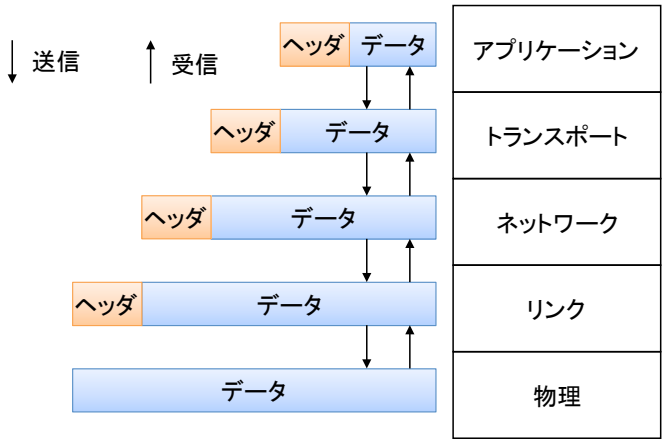


図 6.2: 階層モデルにおけるデータとヘッダ

- アプリケーション層
  Web ページのアクセスやメールの送信などのアプリケーションプログラムの動作を規定する階層．アプリケーション層で送受信されるデータの塊を**メッセージ**と呼ぶことが多い．

- プレゼンテーション層
  データの表現形式の取り決めをする階層．たとえば 32 ビットの整数型のデータは CPU によってデータの表現形式が異なるので，32 ビット整数型データを送受信するには共通形式の定義が必要となる．

- セッション層
  アプリケーションプログラムにとって意味のあるデータ通信単位であるセッションを制御する階層．

- トランスポート層
  end-to-end (アプリケーションプログラム間) のデータ通信を制御する階層．トランスポート層で送受信されるデータの塊を**セグメント**と呼ぶことが多い．

- ネットワーク層
  host-to-host (始点ノードと終点ノード間) のデータ通信を制御する階層．ネットワーク層で送受信されるデータの塊を**パケット**と呼ぶことが多い．

- リンク層
  point-to-point (1 つの媒体で接続された 2 つのノード間) のデータ通信を制御する階層．リンク層で送受信されるデータの塊を**フレーム**と呼ぶことが多い．

- 物理層
  物理的な信号の伝達を制御する階層．

一方，上記の OSI 参照モデルに対してインターネットはプレゼンテーション層とセッション層を省いた 5 階層モデルに基づいている (図 6.1-(b) 参照)．これ以降はインターネットの階層モデルに基づいて説明する．階層モデルでは，論理的には対向する同位の階層間でデータの交換を行うと考える．すなわち，始点ホストのアプリケーション層は終点ホストのアプリケーション層と直接メッセージを交換し，始点ホストのトランスポート層は終点ホストのトランスポート層と直接セグメントを交換すると考える．始点ホストと終点ホストが中継ノード (**ルータ**) を介して接続している場合，ネットワーク層は 1 つのリンクで接続された隣接ノードのネットワーク層とパケットを交換し，リンク層は 1 つのリンクで接続された隣接ノードのリンク層とフレームを交換すると考える．また図 6.1 に示すように，始点ホストと終点ホストはすべての階層を持つが，ルータはネットワーク層までしか持たない．

しかし実際には，始点ノードのアプリケーションプログラムが作成したデータにはアプリケーション層でアプリケーションヘッダが付加されてトランスポート層に渡される (図 6.2 参照)．トランスポート層はアプリケーション層から渡されたデータの塊の全体をデータ[1]とし，これにトランスポート層ヘッダを付加してネットワーク層に渡す．以下の階層で

[1]ヘッダに対してデータ部をペイロード (payload) と呼ぶこともある

も同様な処理が行われ，最終的には媒体を通じて終点ホストにリンク層フレームが届く．終点ホストでは始点ホストでの処理とは逆に各階層においてヘッダが取り除かれてデータ部が上位層に渡されていき，最終的には始点ホストのアプリケーションプログラムが送信したデータを終点ホストのアプリケーションが受け取ることになる．

6.2 TCP/IP の基礎

インターネットで使用されているプロトコル群を総称して **TCP/IP** と呼ぶことが多い．以降では，ネットワーク層およびトランスポート層のプロトコルを簡単に紹介する．

6.2.1 IP と IP アドレス

IP (Internet Protocol) はインターネットにおけるネットワーク層プロトコルである．上述のように，IP は始点ホストと終点ホスト間のパケット転送制御を行う．現在主に利用されている IP はバージョンが 4 なので，**IPv4** とも呼ばれる．IP はパケットの確実な到達や順序どおりの配送，データの誤りなし，などを保証しない．このような通信サービスを**ベストエフォート (best effort)** と呼ぶ．

IPv4 においては，各ホストは 32 ビットの **IP アドレス**で識別される[2]．IP アドレスを表記する場合は，以下のように 32 ビット (4 バイト) の数値を 1 バイトごとにドット (“.”) で区切って表す．

```
131.113.71.3
```

IP アドレスは**ネットワーク部**と**ホスト部**という 2 つの部分からなる．ネットワーク部はホストが接続されているネットワークセグメントをインターネットの中で一意に識別するための番号である．一方ホスト部は 1 つのネットワークセグメント内でホストを一意に識別するための番号である．ホスト部とネットワーク部の長さの設定はネットワーク管理者に任されている．たとえばネットワーク部の長さが 24 ビット (3 バイト) の場合は以下のように表記する．

```
131.113.71.3/24
```

すなわち，131.113.71 がネットワーク部で 3 がホスト部である．

図 6.3 や図 6.4 に示すように，IP ヘッダには始点 IP アドレスと終点 IP アドレスが格納される．ルータは終点 IP アドレスによって次に転送すべきルータを決定する．このように，各パケットのヘッダに始点アドレスと終点アドレスを格納し，終点アドレスにしたがって経路を選択してパケットを転送する方式を**データグラム**と呼ぶ．

[2]正確には IP アドレスはホストを識別するのではなく，ホストがもつネットワークインタフェースを識別する

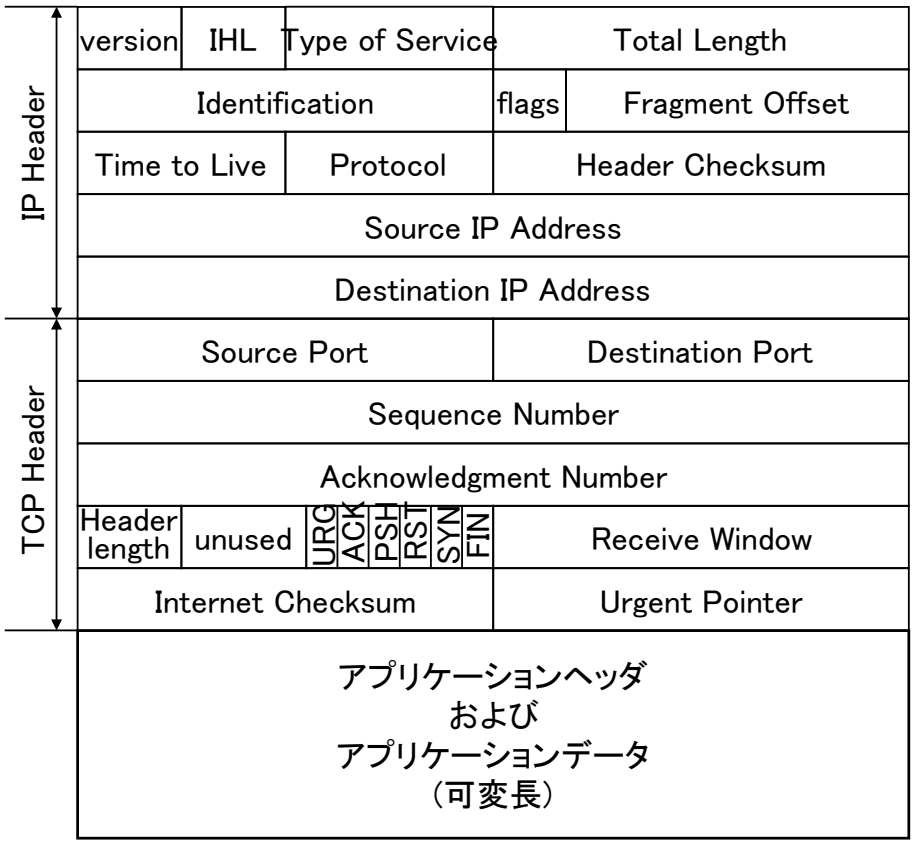


図 6.3: IP ヘッダ，TCP ヘッダ，アプリケーションデータ

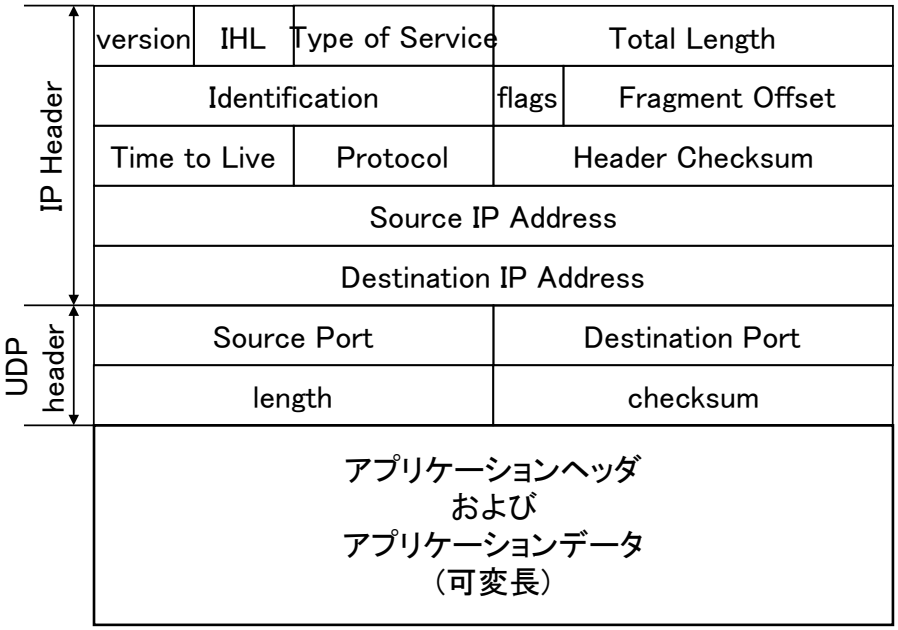


図 6.4: IP ヘッダ，UDP ヘッダ，アプリケーションデータ

```
tera[logex00]% /sbin/ifconfig
eth0      Link encap:Ethernet  HWaddr 40:61:86:DE:06:5E
          inet addr:131.113.108.53  Bcast:131.113.111.255  Mask:255.255.252.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:211998397 errors:0 dropped:0 overruns:0 frame:0
          TX packets:222447451 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:112419414274 (104.6 GiB)  TX bytes:146134790184 (136.0 GiB)
          Interrupt:20 Memory:fb800000-fb820000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:21992518 errors:0 dropped:0 overruns:0 frame:0
          TX packets:21992518 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:12110517799 (11.2 GiB)  TX bytes:12110517799 (11.2 GiB)

tera[logex00]%
```

図 6.5: ifconfig コマンドの実行例

6.2.2 ifconfig コマンド

自ホストの IP アドレスを調べるには ifconfig コマンドを用いる[3]．図 6.5 は理工学 ITC のログインサーバで ifconfig コマンドを実行した例である．ワークステーション室に設置されているコンピュータでの実行結果はこれとは異なるはずである．図 6.5 の実行結果より以下のことがわかる．このコンピュータは eth0 という名前のネットワークインタフェースをもち，131.113.108.53 という IP アドレスが割り当てられている．Mask:255.255.252.0 はネットワーク部の長さを表している．255.255.252.0 を 16 進数で表すと 0xfffffc00，2 進数で表すと 1111 1111 1111 1111 1111 1100 0000 0000 となる．1 となっている位置がネットワーク部である．すなわちこの IP アドレスのネットワーク部の長さは 22 ビットであることがわかる．

6.2.3 TCP

TCP (Transmission Control Protocol) はインターネットにおけるトランスポート層プロトコルの 1 つである．TCP はパケットの確実な到達および順序どおりの配送，データ誤りなし，などを保証する．TCP はデータ転送に先立って両方のホスト間で制御セグメントの交換を行い，論理的なコネクションを確立する．このように，通信に先立って両方のホスト間に論理コネクションを確立し，通信の信頼性を保証する通信方式を**コネクション指向通信** (connection oriented communication) と呼ぶ．

[3]現在は ifconfig コマンドに代わり，ip コマンドが推奨されている．ifconfig コマンドに相当するのは ip a または ip link である．

6.2.4 UDP

UDP (User Datagram Protocol) はインターネットにおけるトランスポート層プロトコルの 1 つである．UDP は TCP と違い，通信に先立って論理コネクションの確立を行わず，信頼性を保証しない通信サービスを提供する．このような通信方式を**コネクションレス通信** (connectionless communication) と呼ぶ．

6.2.5 ポート

TCP や UDP においては，通信の端点を**ポート**と呼ぶ．ポートは 16 ビットのポート番号で表される．図 6.3 はトランスポート層プロトコルとして TCP が使われるときのパケットフォーマットを示しており，図 6.4 はトランスポート層プロトコルとして UDP が使われるときのパケットフォーマットを示している．TCP ヘッダには始点ポート番号と終点ポート番号が格納される．TCP セグメントを受信したホストは，終点ポート番号によってこのパケットを受信すべきアプリケーションプログラムを決定する．UDP の場合も同様である．

6.2.6 Well-known ポート

インターネットにおける代表的なアプリケーションが使用するポート番号は前もって決まっており，`/etc/services` というファイルに定義されている．たとえば，SMTP (Simple Mail Transfer Protocol) は 25 番，DNS (Domain Name System) は 53 番，HTTP (Hyper Text Transfer Protocol) は 80 番，POP3 (Post Office Protocol) は 110 番，IMAP4+TLS (Internet Message Access Protocol + Transport Layer Security) は 993 番，などである．

ポート番号は 16 ビットの整数であるが，0〜1023 は **well-known port** として特定のアプリケーション用に予約されている．また，1024〜5000 はクライアント側からの接続に使う一時的なポート番号として利用されている．動的に使用したり，プライベートな目的に使うポート番号としては 49152〜65535 が割り当てられている．

6.2.7 ソケット

IP アドレスはインターネット上で一意である[4]．ポート番号はそのホスト内で一意である．したがって，IP アドレスとポート番号の組によってインターネット上で一意な通信端点を表すことができる．IP アドレスとポート番号の組を**ソケット**と呼ぶ (図 6.6 参照)．すなわち，アプリケーションプログラムはソケットを通じて相手のアプリケーションプログラムと通信すると言える．

[4]厳密には，IP アドレスにはグローバルアドレスとプライベートアドレスがあり，グローバルアドレスはインターネット上で一意であるが，プライベートアドレスはそうではない．

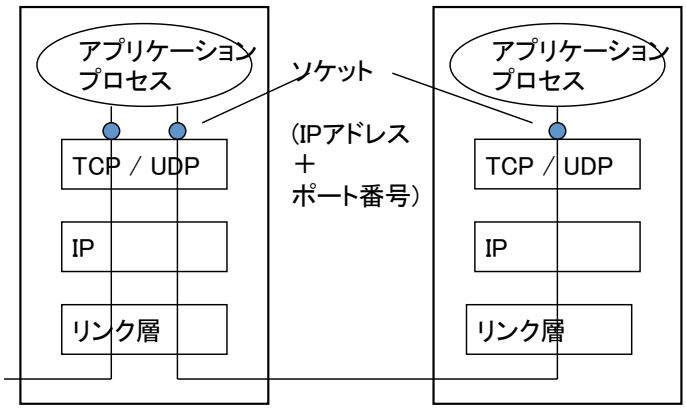


図 6.6: プロセスとソケット

表 6.1: ネットワークプログラミングで使用される型

型	説明
int8_t	8 ビットの整数
uint8_t	8 ビットの符号なし整数
int16_t	16 ビットの整数
uint16_t	16 ビットの符号なし整数
int32_t	32 ビットの整数
uint32_t	32 ビットの符号なし整数
in_addr_t	uint32_t
sa_family_t	uint8_t
in_port_t	uint16_t
socklen_t	uint32_t

6.3 ネットワークプログラミングの注意点

6.3.1 ビット長を明示した型

通常のプログラミングでは `char`，`int`，`long` などの型を用いる．代入の際に左辺と右辺で型があっているかについては注意を払うと思うが，それぞれの型のビット長についてはあまり注意を払わないのではないだろうか．ネットワークプログラミングでは，図 6.3 に示すように 16 ビットのフィールドや 32 ビットのフィールドなどを扱わなければならない．しかし，たとえば `int` 型のビット長はコンピュータのアーキテクチャに依存し，32 ビットだったり 64 ビットだったりする．これでは不便である．そこでネットワークプログラミングでは表 6.1 に示すようなビット長を明示した型を用いる．

6.3.2 バイトオーダ

データをメモリ上にどのように配置するか (バイトオーダ) は CPU のアーキテクチャに依存する．たとえば，32 ビットの CPU で 32 ビットの整数型の数値 (e.g., 0x12345678) を

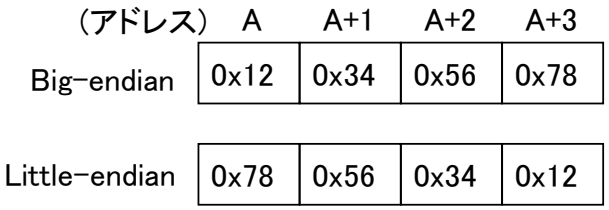


図 6.7: big-endian と little-endian

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

uint32_t
htonl(uint32_t hostlong);   // 32 ビット，ホスト -> ネットワーク

uint16_t
htons(uint16_t hostshort);  // 16 ビット，ホスト -> ネットワーク

uint32_t
ntohl(uint32_t netlong);    // 32 ビット，ネットワーク -> ホスト

uint16_t
ntohs(uint16_t netshort);   // 16 ビット，ネットワーク -> ホスト
```

図 6.8: バイトオーダ変換のための関数

メモリ上 (e.g., A 番地から A+3 番地) に配置する場合，図 6.7 に示す 2 種類がある．最上位のバイトを最初に配置する方式を **big-endian** と呼び，その逆を **little-endian** と呼ぶ．1 台のマシンの中でデータを交換する際にはバイトオーダに注意する必要はないが，ネットワークを介して異なったアーキテクチャによるコンピュータ間でデータを交換する際には，データのバイトオーダを統一しておく必要がある．インターネットにおいては，データは big-endian で表現するようになっている．IP アドレスやポート番号を扱う場合には注意が必要である．そこで，CPU 固有のバイトオーダとネットワーク共通のバイトオーダとを変換する関数が用意されている (図 6.8 参照)．すなわち，送信側では送信データのバイトオーダをネットワークオーダに変換し，受信側では受信データのバイトオーダをネットワークバイトオーダからをホストオーダに変換しなければならない．

6.4 UDP による通信の基礎

6.4.1 全体のシーケンス

便宜上，クライアント・サーバモデルで UDP 通信を行う場合のシーケンスを図 6.9 に示す．クライアント，サーバともにまず `socket()` システムコールにより通信端点であるソ

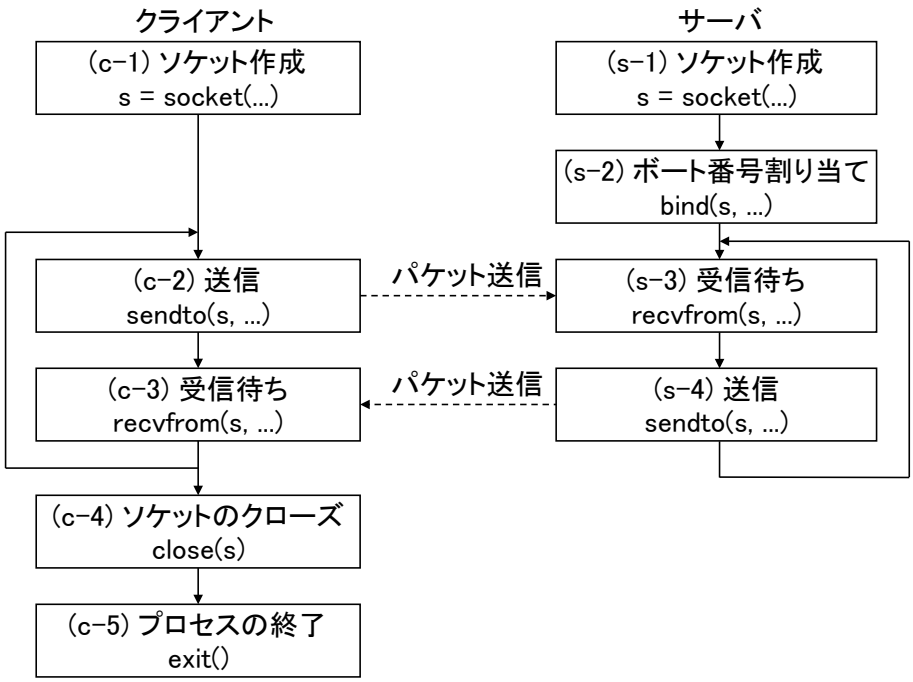


図 6.9: UDP 通信のための手順

ケットを作成する (図 6.9-(c-1), (s-1)). クライアントはサーバのソケットを指定してデータを送信するので，サーバのソケットはクライアントプロセスから識別できるようにしなければならない．すなわち，サーバのソケットには識別のためにポート番号を割り当てる必要がある．そこでサーバでは `bind()` システムコールによって自身のソケットにポート番号を割り当てる (図 6.9-(s-2)). 一方，クライアントのソケットに関しては他のプロセスから識別される必要はないため，`bind()` システムコールを呼び出す必要はない．このような場合，クライアントのソケットにはオペレーティングシステムが自動的にポート番号を割り当てる．

次にサーバは `recvfrom()` システムコールを呼び出してクライアントからのパケット受信を待つ (図 6.9-(s-3)). クライアントは `sendto()` システムコールを呼び出してサーバにパケットを送信する (図 6.9-(c-2)). サーバからクライアントへパケットを送信する場合，クライアントは `recvfrom()` システムコールでパケット受信を待ち，サーバは `sendto()` システムコールでパケットを送信する (図 6.9-(c-3), (s-4)).

通信が終わったら，クライアントは `close()` システムコールによってソケットを閉じ (図 6.9-(c-4))，`exit()` システムコールによってプロセスを終了する．サーバは他のクライアントからのパケットを待つ (図 6.9-(s-3)).

6.4.2 `socket()`：ソケットの生成

通信端点であるソケットを生成するには `socket()` システムコールを使用する．`socket()` システムコールの構文を図 6.10 に示す．`domain` は通信が行われるドメインを指定する．インターネットの場合，`PF_INET` という値を用いる[5]．`type` はプロトコルのタイプを指定

[5] `AF_INET` でもよい

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

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

図 6.10: socket() の構文

する．TCP の場合は SOCK_STREAM を指定し，UDP の場合は SOCK_DGRAM を指定する．上記で指定した domain および type に該当するプロトコルが複数ある場合は protocol でプロトコルを指定する．通常は該当するプロトコルは 1 つしかないので 0 とする．

socket() は返り値としてソケット記述子 (socket descriptor) を返す．この値はファイル入出力におけるファイル記述子 (file descriptor) と似ており，以降のネットワークに対する入出力の際の入出力先を指定するために用いられる．

6.4.3 ソケットアドレス構造体

通信端点であるソケットの識別子としてソケットアドレス構造体が定義されている．通信のためのシステムコールはインターネットプロトコル群以外もサポートできるように一般的に設計されているが，本書ではインターネットに特化して述べる．

インターネットにおけるソケットアドレス構造体である struct sockaddr_in の定義を図 6.11 に示す．struct sockaddr_in は<netinet/in.h>というヘッダファイルで定義されているので，struct sockaddr_in を使用する場合には “#include <netinet/in.h>” が必要となる．struct sockaddr_in において，sin_family はアドレスファミリーを示し，インターネットの場合は AF_INET という値が使用される．sin_port はポート番号を表す 16 ビットの符号なし整数である．IP アドレスは struct in_addr 型の sin_addr で表され，この構造体には s_addr という 32 ビットの符号なし整数のメンバのみが含まれている．自ホストに割り当てられている IP アドレスを使用する場合は，sin_addr には INADDR_ANY という値を設定する．sin_len はアドレス長を示すためのメンバであるが，sin_family で AF_INET を指定しているので，特に指定する必要はない．sin_zero[8] には 0 を代入しておく．

6.4.4 IP アドレスの表現形式の変換

いわゆるドット (“.”) 表記の文字列で表した IP アドレスをバイナリ形式に変換したり，その逆を行う関数が用意されている (図 6.12)．inet_aton() は cp が指すメモリオブジェクトに格納されている文字列で表した IP アドレスを，ネットワークバイトオーダのバイナリ形式に変換して pin が指すメモリオブジェクトに格納する．正常の場合は inet_aton() は 1 を返し，文字列表記が不正の場合は 0 を返す．

```
/* Internet address */
struct in_addr {
    in_addr_t s_addr;
};

/* Socket address, internet style. */
struct sockaddr_in {
    uint8_t        sin_len;
    sa_family_t    sin_family;  // アドレスファミリー (AF_INET)
    in_port_t      sin_port;    // ポート番号
    struct in_addr sin_addr;    // IP アドレス
    char           sin_zero[8];
};
```

図 6.11: sockaddr_in 構造体の定義

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

int
inet_aton(const char *cp, struct in_addr *pin);

char *
inet_ntoa(struct in_addr in);
```

図 6.12: IP アドレスの表現形式変換のための関数

inet_ntoa() は in に設定されたバイナリ形式の IP アドレスを文字列に変換してメモリオブジェクトに格納し，このメモリオブジェクトへのポインタを返す．

6.4.5 bind()：ソケットへのアドレスの割り当て

図 6.13 に bind() システムコールの構文を示す．bind() は自ホストのソケットにソケットアドレス (IP アドレスとポート番号の組) を割り当てる．s は socket() の返り値であるソケット記述子である．addr は，インターネットの場合には struct sockaddr_in のメモリオブジェクトへのポインタとする．IP アドレスの割り当てをオペレーティングシステムに任せる場合は，addr->sin_addr.s_addr に INADDR_ANY をセットする．IP アドレスやポート番号をセットする際には，htonl() 関数や htons() 関数を使用してネットワークバイトオーダに変換することを忘れてはならない．

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

int
bind(int s, const struct sockaddr *addr, socklen_t addrlen);
```

図 6.13: bind() の構文

bind() は成功すると 0 を返し，エラーの場合は-1 を返す．エラーの理由は errno に設定される．

図6.14 にUDP 通信の場合の socket() と bind() の使用例を示す．05 行目は SOCK_DGRAM を指定してソケットをオープンしている．12 行目は自分のソケットアドレスである myskt の領域を 0 で初期化している．13〜15 行目で myskt の各メンバの値を設定している．ここではバイトオーダに気をつけなければならない．17 行目で bind() を呼び出している．このとき，&myskt の型を struct sockaddr *型にキャストしなければならない．

6.4.6 recvfrom()：パケット受信待ち

図 6.15 に recvfrom() システムコールの構文を示す．recvfrom() は指定したソケットにおいてパケットの受信を待つ．s は socket() の返り値であるソケット記述子である．buf はパケットのデータ部 (図 6.4 の「アプリケーションヘッダおよびアプリケーションデータ」部) が格納されるメモリオブジェクトへのポインタである．len は buf が指す領域のサイズである．flags は通常は 0 でよい．from が NULL ポインタでなく，かつソケットがコネクション指向でない場合，from が指すメモリオブジェクトに送信側のソケットの情報が設定される．すなわち，図 6.4 の UDP ヘッダにある Source Port フィールド，および IP ヘッダにある Source IP Address フィールドの値が from の指す領域に設定される．fromlen が指すメモリオブジェクトには，recvfrom() を呼び出す際には from が指すメモリオブジェクトのサイズを設定しておく．bind() の実行後には送信側のソケットアドレスの実際のサイズが設定されている．

recvfrom() は実際に受信したデータサイズを返す．エラーの場合は-1 を返し，errno に理由が設定される．

図6.16 に recvfrom() の使用例を示す．recvfrom() を呼び出す前に socket() や bind() を実行しておかなければならない．recvfrom() を呼び出すときに気をつけなければならないことは，送信側のソケットアドレスを格納するオブジェクトのサイズの渡し方である．09 行目で送信側のソケットアドレスである skt のサイズを sktlen に設定している．そして sktlen のポインタを recvfrom() の最後の引数として渡している．recvfrom() の実行後には，送信側のソケットアドレスの実際のサイズが sktlen に返される．また，skt に送信側のソケットアドレスが返される．送信側のポート番号や IP アドレスを扱う際には，ntohs() や ntohl() を使用してホストバイトオーダに変換しなければならないことを忘れないように．

```
01:    int s;                    // ソケット記述子
02:    in_port_t myport;         // 自ポート番号
03:    struct sockaddr_in myskt; // 自ソケット
04:
05:    if ((s = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
06:        perror("socket");
07:        exit(1);
08:    }
09:
10:    // myport の設定
11:
12:    memset(&myskt, 0, sizeof myskt);
13:    myskt.sin_family = AF_INET;
14:    myskt.sin_port = htons(myport);
15:    myskt.sin_addr.s_addr = htonl(INADDR_ANY);
16:
17:    if (bind(s, (struct sockaddr *)&myskt, sizeof myskt) < 0) {
18:        perror("bind");
19:        exit(1);
20:    }
```

図 6.14: socket() と bind() の使用例

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

ssize_t
recvfrom(int s, void * restrict buf, size_t len, int flags,
    struct sockaddr * restrict from, socklen_t * restrict fromlen);
```

図 6.15: recvfrom() の構文

```
01:    int s, count;
02:    struct sockaddr_in myskt; // 自ソケット
03:    struct sockaddr_in skt;   // 相手のソケット
04:    char rbuf[512];           // 受信用バッファ
05:    socklen_t sktlen;
06:
07:    // s = socket(...); および bind(s, &myskt, ..); を実行する
08:
09:    sktlen = sizeof skt;
10:    if ((count = recvfrom(s, rbuf, sizeof rbuf, 0,
11:             (struct sockaddr *)&skt, &sktlen)) < 0) {
12:        perror("recvfrom");
13:        exit(1);
14:    }
15:    // skt.sin_port に送信側のポート番号が設定されている
16:    // skt.sin_addr.s_addr に送信側の IP アドレスが設定されている
17:    //     バイトオーダに気をつけること
18:    // sktlen に相手側ソケットアドレスの実際のサイズが返される
```

図 6.16: recvfrom() の使用例

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

ssize_t
sendto(int s, const void *msg, size_t len, int flags,
    const struct sockaddr *to, socklen_t tolen);
```

図 6.17: sendto() の構文

受信パケットの各ヘッダフィールドと recvfrom() の引数の値との関係を，図 6.4 を用いて説明する．recvfrom() の引数である s は受信側のソケットであり，bind() によってソケットアドレス (IP アドレスとポート番号の組) が割り当てられている．すなわち，IP ヘッダの “Destination Address” フィールドが受信側ソケットの IP アドレスと一致し，かつ UDP ヘッダの “Destination Port” の値が受信側ソケットのポート番号と一致する UDP セグメントのみが受信されることになる．このとき，受信パケットの IP ヘッダの “Source IP Address” の値と UDP ヘッダの “Source Port” の値が from の指すメモリオブジェクトに設定され，送信側のソケットアドレスを知ることができる．from の内容を送信先のソケットアドレスとして指定して sendto() を呼び出すことにより，UDP セグメントを送信してきた相手へ UDP セグメントを送信することができる．

6.4.7 sendto()：パケット送信

図 6.17 に sendto() システムコールの構文を示す．s は socket() の返り値であるソケット記述子である．msg は送信すべきデータが格納されているメモリオブジェクトへのポインタであり，図 6.4 の「アプリケーションヘッダおよびアプリケーションデータ」部に格納される．len は送信データのサイズである．flags は通常は 0 でよい．to は，インターネットの場合は struct sockaddr_in のメモリオブジェクトへのポインタである．to->sin_family には AF_INET を設定する．to->sin_addr.s_addr に送り先の IP アドレスを設定し，to->sin_port に送り先のポート番号を設定する．すなわち，これらの値は図 6.4 の UDP ヘッダ部の Destination Port フィールドおよび IP ヘッダ部の Destination IP Address フィールドに設定される．tolen は to が指すメモリオブジェクトのサイズである．

sendto() は実際に送信したデータサイズを返り値として返す．エラーの場合は -1 を返し，エラーの理由は errno に設定される．

図 6.18 に sendto() の使用例を示す．sendto() を呼び出す前に，自分のソケットをオープンし，送信データの準備をしておかなければならない．11〜13 行目で送信相手のソケットアドレスである skt を設定している．このとき，バイトオーダの変換をしなければならないことを忘れないように．

送信パケットのフィールドと sendto() の引数の関係を，図 6.4 を用いて説明する．sendto() の引数である s は送信側のソケットであるが，bind() を呼び出していないのでソケットア

```
01:    int s, count, datalen;
02:    struct sockaddr_in skt;   // 相手のソケット
03:    char sbuf[512];           // 送信用バッファ
04:    in_port_t port;           // 送信相手のポート番号
05:    struct in_addr ipaddr;    // 送信相手の IP アドレス
06:
07:    // s = socket(...); を実行する
08:    // sbuf[] に送信データを準備し，datalen にデータ長を設定する
09:    // port, ipaddr を設定する
10:
11:    skt.sin_family = AF_INET;
12:    skt.sin_port = htons(port);
13;    skt.sin_addr.s_addr = htonl(ipaddr.s_addr);
14:    if ((count = sendto(s, sbuf, datalen, 0,
15:             (struct sockaddr *)&skt, sizeof skt)) < 0) {
16:        perror("sendto");
17:        exit(1);
18:    }
```

図 6.18: sendto() の使用例

ドレス (IP アドレスとポート番号の組) は割り当てられていない．このような場合はオペレーティングシステムが自動的に IP アドレスとポート番号を割り当てる．すなわち，送信パケットの IP ヘッダの “Source IP Address” フィールドには，自ホストの IP アドレスが設定される．また UDP ヘッダの “Source Port” フィールドには，オペレーティングシステムが割り当てたポート番号が設定される．一方，sendto() の引数である to は受信側のソケットアドレスであり，to によって指定された IP アドレスが IP ヘッダの “Destination IP Address” フィールドに設定され，to によって指定されたポート番号が UDP ヘッダの “Destination Port” フィールドに設定される．

6.4.8 close()：ソケットのクローズ

ソケットを閉じる際には close() システムコールを呼び出す．引数は socket() の返り値であるソケット記述子である．

6.5 TCP による通信の基礎

6.5.1 全体のシーケンス

クライアント・サーバモデルで TCP 通信を行う場合の基本的なシーケンスを図 6.19 に示す．クライアント，サーバともにまず socket() システムコールにより通信端点であるソケットを作成する (図 6.19-(c-1), (s-1))．第 6.4.1 節で述べたようにサーバ側のソケットにはポート番号を割り当てる必要があるので，サーバは bind() システムコールによって自身のソケットにポート番号を割り当てる (図 6.19-(s-2))．

次にサーバは listen() システムコールを呼び出し，クライアントからの TCP コネクション確立要求受信の準備を行う (図 6.19-(s-3))．そしてサーバはクライアントからのコネクション確立要求の受信を待つ (図 6.19-(s-4))．一方，クライアントは TCP コネクションを確立するためサーバ側のソケットを指定して connect() システムコールを呼び出す (図 6.19-(c-2))．この結果，クライアントとサーバ間で制御セグメントがやり取りされ，TCP コネクションが確立する．

その後，データを送信する場合は send() システムコールを呼び出し (図 6.19-(c-3), (s-6))，受信する場合は recv() システムコールを呼び出す (図 6.19-(c-5), (s-5))．送受信を繰り返して処理が終了すると，クライアント，サーバともに close() システムコールを呼び出し，TCP コネクションを終了するとともにソケットを閉じる (図 6.19-(c-5), (s-7))．クライアントは exit() システムコールを呼び出し，プロセスを終了する (図 6.19-(c-6))．サーバは次のコネクション確立要求を待つため図 6.19-(s-4) に戻る．

6.5.2 listen()：コネクション確立要求の受信準備

図 6.20 に listen() システムコールの構文を示す．listen() システムコールは，TCP コネクション確立要求の制御セグメント受信のための準備を行う．s は socket() システ

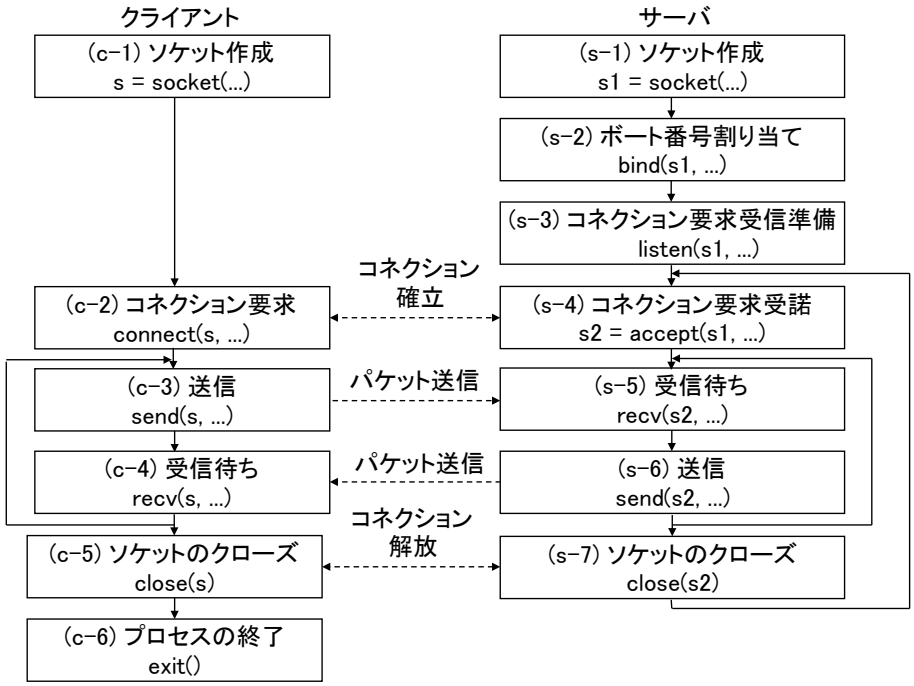


図 6.19: TCP 通信の基本的なシーケンス

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

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

図 6.20: `listen()` の構文

ムコールの返り値であるソケット記述子である．`backlog` はコネクション確立要求の制御セグメントの待ち行列のサイズである．通常は 5 程度を指定しておけばよい．

正常の場合，`listen()` は 0 を返す．エラーの場合，`listen()` は-1 を返し，エラーの理由は `errno` に設定される．

6.5.3 `accept()`：コネクション確立要求の受諾

図 6.21 に `accept()` システムコールの構文を示す．`accept()` システムコールはコネクション確立要求の制御セグメントを処理し，TCP コネクションを確立する．`listen()` システムコールによって作られた待ち行列にすでにコネクション確立要求セグメントがある場合をそれを即座に処理する．ない場合は到着を待つ．`s` は `socket()` システムコールの返り値であるソケット記述子であり，すでに `bind()` システムコールによってポート番号が割り当てられ，`listen()` システムコールによってコネクション確立要求受信の準備が終わっている必要がある．`addr` が示す領域に，コネクション要求セグメントを送信した側のソケットのアドレスが設定される．上述のように，インターネットの場合は `struct`

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

int
accept(int s, struct sockaddr * restrict addr,
    socklen_t * restrict addrlen);
```

図 6.21: accept() の構文

sockaddr の代わりに struct sockaddr_in 型が用いられる．addrlen は addr が指すメモリオブジェクトのサイズを示す．

accept() は返り値として新しいソケット記述子を返す．TCP コネクション確立後の送受信 (send() および recv() システムコール) の際には，accept() の返り値であるソケット記述子を用いる．エラーの際には accept() は-1 を返す．エラーの理由は errno に設定される．

図6.22 に socket()，bind()，listen()，accept() の使用例を示す．06 行目では SOCK_STREAM を指定してソケットをオープンしている．次に自分のソケットアドレスを myskt に設定し，13 行目で bind() を実行している．17 行目で listen() を実行しているが，第 2 引数は適当な正の整数を指定しておけばよい．23 行目で accept() を呼び出しているが，ここで気をつけなければならないことは，第 3 引数である sktlen に事前に相手のソケットアドレスである skt のサイズを代入しておかなければならないことである．TCP コネクションが確立すると accept() の実行が終了し，skt には相手のソケットアドレスが返され，sktlen には相手のソケットアドレスサイズが返される．また accept() は新しいソケット記述子 (s2) を返すので，以降の send()，recv() の呼び出しの際には，s2 を使用する．

コネクション確立要求の制御セグメントのフィールドの値と accept() の引数との関係を，図 6.3 を用いて説明する．accept() の引数である s は bind() によってソケットアドレスが割り当てられている．すなわち IP ヘッダの “Destination IP Address” フィールドの値がこのソケットの IP アドレスと一致し，TCP ヘッダの “Destination Port” フィールドの値がこのソケットのポート番号と一致するコネクション確立要求セグメントのみを受け付ける．そして accept() の引数である addr が指すメモリオブジェクトには IP ヘッダの “Source IP Address” フィールドの値，および TCP ヘッダの “Source Port” フィールドの値が設定される．

6.5.4 connect()：コネクション確立要求

図 6.23 に connect() システムコールの構文を示す．connect() システムコールは TCP コネクション確立要求セグメントを送信し，TCP コネクションを確立する．s はソケット記述子である．name が指すメモリオブジェクトに，TCP コネクションを確立する相手のソケットのアドレスを設定する．上述のように，インターネットの場合は struct sockaddr

```
01:    int s, s2;
02:    struct sockaddr_in myskt;  // 自分のソケットアドレス
03:    struct sockaddr_in skt;    // 相手のソケットアドレス
04:    socklen_t sktlen;
05:
06:    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
07:        perror("socket");
08:        exit(1);
09:    }
10:
11:    // myskt の各メンバの設定をする
12:
13:    if (bind(s, (struct sockaddr *)&myskt, sizeof myskt) < 0) {
14:        perror("bind");
15:        exit(1);
16:    }
17:    if (listen(s, 5) < 0) {   // 第２引数には適当な正の整数を指定する
18:        perror("listen");
19:        exit(1);
20:    }
21:
22:    sktlen = sizeof skt;
23:    if ((s2 = accept(s, (struct sockaddr *)&skt, &sktlen)) < 0) {
24:        perror("accept");
25:        exit(1);
26:    }
27:    // 相手のソケットアドレスが skt に返される
28:    // また相手のソケットアドレスサイズが sktlen に返される
29:    //     バイトオーダに注意すること
30:    // 以降の send() や recv() には s ではなく，s2 を使用する
```

図 6.22: socket(), bind(), listen(), accept() の使用例

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

int
connect(int s, const struct sockaddr *name, socklen_t namelen);
```

図 6.23: connect() の構文

の代わりに struct sockaddr_in 型が用いられる．namelen は name が指すメモリオブジェクトのサイズを示す．

正常な場合は connect() は 0 を返し，エラーの場合は-1 を返す．エラーの理由は errno に設定される．

図 6.24 に socket() と connect() の使用例を示す．06 行目では SOCK_STREAM を使用して socket() を実行している．13 行目では相手のソケットアドレスである skt の領域を 0 で初期化している．14〜16 行目で相手のソケットアドレスを設定している．このとき，バイトオーダに気をつけなければならない．そして 17 行目で connect() を呼び出している．TCP コネクションが確立すると connect() の実行が終了する．

コネクション確立要求の制御セグメントのフィールドの値と connect() の引数との関係を，図 6.3 を用いて説明する．connect() の引数である s には bind() によってソケットアドレスが割り当てられていない．するとオペレーティングシステムが自動的に IP ヘッダの “Source IP Address” フィールドに自ホストの IP アドレスを設定し，TCP ヘッダの “Source Port” フィールドには適当な値を選択して設定する．そして connect() の引数である name によって指定された IP アドレスおよびポート番号が IP ヘッダの “Destination IP Address” フィールドおよび TCP ヘッダの “Destination Port” フィールドにそれぞれ設定される．

6.5.5 recv()：パケットの受信待ち

図 6.25 に recv() システムコールの構文を示す．recv() システムコールは TCP コネクションでのデータ受信を待つ．s は socket() または accept() システムコールの返り値であるソケット記述子である．buf は受信データを格納するメモリオブジェクトを指す．len は buf が指すメモリオブジェクトのサイズを示す．flags は 0 でよい．

recv() は実際に受信したデータのバイト数を返す．エラーの場合は-1 を返す．エラーの理由は errno に設定される．

recv() の引数である s はすでに TCP コネクションの端点となっている．すなわち，recv() は自ソケットの IP アドレスの値およびポート番号の値が，IP ヘッダの “Destination IP Address” フィールドの値および TCP ヘッダの “Destination Port” の値と一致し，さらに相手ソケットの IP アドレスの値およびポート番号の値が，IP ヘッダの “Source IP Address” の値および TCP ヘッダの “Source Port” の値と一致する TCP セグメントのみを受信する (図 6.3 参照)．

```
01:    int s;
02:    struct sockaddr_in skt;  // 相手のソケットアドレス
03:    in_port_t port;          // 相手のポート番号
04:    struct in_addr ipaddr;   // 相手の IP アドレス
05:
06:    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
07:        perror("socket");
08:        exit(1);
09:    }
10:
11:    // port, ipaddr を設定する
12:
13:    memset(&skt, 0, sizeof skt);
14:    skt.sin_family = AF_INET;
15:    skt.sin_port = htons(port);
16:    skt.sin_addr.s_addr = htonl(ipaddr.s_addr);
17:    if (connect(s, (struct sockaddr *)&skt, sizeof skt) < 0) {
18:        perror("connect");
19:        exit(1);
20:    }
```

図 6.24: socket(), connect() の使用例

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

ssize_t
recv(int s, void *buf, size_t len, int flags);
```

図 6.25: recv() の構文

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

ssize_t
send(int s, const void *msg, size_t len, int flags);
```

図 6.26: send() の構文

6.5.6 send()：パケット送信

図 6.26 に send() システムコールの構文を示す．send() システムコールは TCP コネクションにおいてデータを送信する．s は socket() または accept() システムコールの返り値であるソケット記述子である．msg は送信データを格納したメモリオブジェクトへのポインタである．len は送信データのバイト数である．flags は 0 でよい．

send() は実際に送信したデータのバイト数を返す．エラーの場合は-1 を返す．エラーの理由は errno に設定される．

send() の引数である s はすでに TCP コネクションの端点となっている．すなわち，send() は自ソケットの IP アドレスの値およびポート番号の値を，IP ヘッダの “Source IP Address” フィールドおよび TCP ヘッダの “Source Port” フィールドに設定し，さらに相手ソケットの IP アドレスの値およびポート番号の値を，IP ヘッダの “Destination IP Address” フィールドおよび TCP ヘッダの “Destination Port” フィールドに設定し，TCP セグメントを送信する (図 6.3 参照)．

6.6 TCP と UDP における受信データの取り扱いの違い

図 6.27 に受信バッファの状態の例を示す．受信データは左側から受信バッファに入り，右側から recv() や recvfrom() によってプロセスに読み込まれるとする．この例では，送信側は最初に 500 バイトのデータを送信し，次に 1,000 バイトのデータを送信し，次にまた 500 バイトのデータを送信した状態である．ここでこの通信が TCP によるものか UDP によるものかで受信側での受信データの扱い方に差がある．

まずこれを TCP 通信であるとする．TCP はバイトストリームと呼ばれる通信サービスを提供する．したがって，socket() の引数である type には SOCK_STREAM が使われる．この例では，送信側は上述のように 500 バイト，1,000 バイト，500 バイトと 3 つの TCP セグメントを送信したわけだが，受信側においてはこのような TCP セグメントの境界は保存されず，単に現在は 2,000 バイトのデータが受信バッファに蓄えられていることになる．受信するプロセスは recv() を呼び出して 1 バイトずつ読み込んでもいいし，1,000 バイトずつ読み込んでもいい．

次にこれを UDP 通信であるとする．UDP では受信側において UDP セグメントの境界が保存される．したがって，受信プロセスがたとえば recvfrom() で 1,024 バイトのバッファサイズを指定した場合，最初の recvfrom() は 500 バイトのデータが読み込まれ，次

図 6.27: 受信データと受信バッファ

は 1,000 バイトのデータが読み込まれ，その次は 500 バイトのデータが読み込まれることになる．

6.7 ホスト名から IP アドレスへの変換

ユーザはホストを www.tera.ics.keio.ac.jp のようなホスト名で識別しているが，IP においてはホストを IP アドレスによって識別している．文字列であるホスト名を IP アドレスに変換する機構が **DNS** (Domain Name System) である．DNS においては，各組織は**ネームサーバ**と呼ばれるサーバを立ち上げ，自組織内のホスト名と IP アドレスの対応関係のデータベースを管理する．ネームサーバに DNS プロトコルで問い合わせることにより，ホスト名に対応した IP アドレスを得ることができる．DNS の詳細はここでは省略する．

DNS の仕組みを利用してホスト名から IP アドレスを得るライブラリ関数が gethaddrinfo() である．図 6.28 に getaddrinfo() の構文を示す．node と service でホストとサービスを指定する．たとえば node にはホスト名 (e.g., “www.ics.keio.ac.jp”) を指定し，service にはサービス名 (e.g., “http”) を指定する．node と service のどちらかは NULL でもよい．hints はノード名やサービス名をソケットアドレスに変換するためのヒントを指定する．ヒントには，希望するアドレスファミリー (e.g., hints.ai_family = AF_INET)，推奨のソケット型 (e.g., hints.ai_socktype = SOCK_STREAM) などがある．また，ai_flags に AI_PASSIVE フラグを指定し，かつ node が NULL の場合，返されるソケットアドレスはコネクションを accept() するためのソケットを bind() するのに適したものになる．逆に hints.ai_flags に AI_PASSIVE フラグが設定されていない場合，返されるソケットアドレスは，connect()，sendto() での使用に適したものになる．hints は NULL でもよい．getaddrinfo() 関数は addrinfo 構造体のリストのメモリ領域を確保し，res にリストの先頭へのポインタを入れて返す．返される addrinfo 構造体がリストになる理由としては，たとえば指定したホストが複数の IP アドレスを持つ場合などである．この構造体リスト領域の使用が済んだら，freeaddrinfo() 関数でこのメモリ領域を解放しなければならない．

図 6.29 にクライアント側における getaddrinfo() の使用例を示す．この例は，まず接続先のホスト名 (例: “www.inl.ics.keio.ac.jp” とサービス名 (例：“http”) を指定して接続先のソケットアドレスを得，次に得られたソケットアドレスを使用して socket() や connect() を呼び出すものである．呼び出し手順としては，まず char *node が指す領域にホスト名の文字列を格納し，char *service が指す領域にサービス名の文字列を格納しておく (コメント部分)．次にヒント (struct addrinfo hints) のソケットタイプ (hints.ai_socktype) に SOCK_STREAM を設定する (08-09 行目)．そして 10 行目で getaddrinfo() を呼び出している．エラーの場合，gai_strerror() 関数を呼び出してエラーの原因を表示している (11 行目)．getaddrinfo() が正常に終了すると res->ai_addr

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

int
getaddrinfo(const char *node, const char *service,
            const struct addrinfo *hints, struct addrinfo **res);

void
freeaddrinfo(strcut addrinfo *res);

struct addrinfo {
     int ai_flags;
     int ai_family;
     int ai_socktype;
     int ai_protocol;
     size_t ai_addrlen;
     struct sockaddr *ai_addr;
     char *ai_canonname;
     struct addrinfo *ai_next;
};
```

図 6.28: getaddrinfo() の構文

がソケットアドレスの領域を指し，その中にソケットタイプ，IP アドレス，ポート番号などが設定されている．その結果，IP アドレスやポート番号などの具体的な値を気にせずに socket() や connect() を使用することができる (16 行目および 21 行目)．

図 6.30 にサーバ側における getaddrinfo() の使用例を示す．まず serv が指す領域にサービス名 (例："http") を格納しておく．次にヒントとして，accept() で使用することを示すため，hints.ai_flags に AI_PASSIVE フラグをセットする (08 行目)．また，TCP で使用することを示すため，hints.ai_socktype に SOCK_STREAM を代入する (09 行目)．そして 10 行目で getaddrinfo() を呼び出す．このとき，第 1 引数には NULL を指定し，IP アドレスは任意であることを示す．次に，getaddrinfo() が返したソケットアドレスを使用して socket() と bind() を呼び出す (15 行目および 20 行目)．次に 25 行目で freeaddrinfo() でソケットアドレスの領域を解放する．次に 27 行目で listen() を呼び出し，33 行目で accept() を呼び出している．ここで注意して欲しい点は，accept() の第 2 引数の型が struct sock_storage のポインタである点である．詳しい説明は省略するが，IP (Internet Protocol) にはバージョン 4 (IPv4) とバージョン 6 (IPv6) があり，アドレス長が違う．IPv4 のアドレス長は 32 ビットであり，IPv6 のアドレス長は 128 ビットである．struct sock_storage は IP のバージョンに依存しないプログラミングを可能にするために定義された．

```
01:    struct addrinfo hints, *res;
02:    char *node, *serv;
03:    int err, sd;
04:
05:    // nodeが指す領域にホスト名を格納 (e.g., "www.inl.ics.keio.ac.jp")
06:    // servが指す領域にサービス名を格納 (e.g., "http")
07:
08:    memset(&hints, 0, sizseof hints);
09:    hints.ai_socktype = SOCK_STREAM;    // TCPを使用
10:    if ((err = getaddrinfo(node, serv, &hints, &res)) < 0) {
11:        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(err));
12:        exit(1);
13:    }
14:    // res->ai_addrがソケットアドレスの領域を指す (struct sockaddr *)
15:
16:    if ((sd = socket(res->ai_family, res->ai_socktype,
                res->ai_protocol)) < 0) {
17:        perror("socket");
18:        exit(1);
19:    }
20:
21:    if (connect(sd, res->ai_addr, res->ai_addrlen) < 0) {
22:        perror("connect"):
23:        exit(1);
24:    }
25:
26:    freeaddrinfo(res);   // getaddrinfo() が確保した領域を解放
27:    . . .
```

図 6.29: getaddrinfo() の使用例：クライアント側

```
01:    struct addrinfo hints, *res;
02:    struct sockarr_storage sin;
03:    char *serv;
04:    int sd, sd1, err, sktlen;
05:
06:    // servが指す領域にサービス名(e.g., "http")を格納しておく.
07:    memset(&hints, 0, sizeof hints);
08:    hints.ai_flags = AI_PASSIVE;      // accept()で使用することを示す
09:    hints.ai_socktype = SOCK_STREAM;  // TCPで使用することを示す
10:    if ((err = getaddrinfo(NULL, serv, &hints, &res)) < 0) {
11:        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(err));
12:        exit(1);
13:    }
14:
15:    if ((sd = socket(res->ai_family, res->ai_socktype,
             res->ai_protocol)) < 0) {
16:        perror("socket");
17:        exit(1);
18:    }
19:
20:    if (bind(sd, res->ai_addr, res->ai_addrlen) < 0) {
21:        perror("bind");
22:        exit(1);
23:    }
24:
25:    freeaddrinfo(res);
26:
27:    if (listen(sd, 5) < 0) {
28:        perror("lisen");
29:        exit(1);
30:    }
31:
32:    sktlen = sizeof (struct sockaddr_storage);
33:    if ((sd1 = accept(sd, &sin, &sktlen)) < 0) {
34:        perror("accept");
35:        exit(1);
36:    }
37:    . . .
```

図 6.30: getaddrinfo() の使用例：サーバ側

練習問題

1. 以下に示すような手順で UDP で通信するサーバとクライアントを，隣にすわっている人と分担して作成しなさい．

   - サーバ：
     (a) ソケットをオープンする．
     (b) ソケットにポート番号 49152 を割り当てる．
     (c) クライアントからのパケット受信を待つ．
     (d) パケットを受信したら，受信データサイズ，クライアントの IP アドレス，ポート番号，受信メッセージを表示する．
     (e) クライアントから受信したメッセージをそのままクライアントに送信する．
     (f) (c) に戻り，次のパケット受信を待つ．
   - クライアント：
     (a) ソケットをオープンする．
     (b) 標準入力からサーバの IP アドレスを文字列表記で受け取る．
     (c) サーバに送信する文字列を標準入力から受け取る．EOF が入力されたら (h) に進む．
     (d) サーバのポート番号 49152 にパケットを送信する．
     (e) サーバからのパケット受信を待つ．
     (f) パケットを受信したら，受信データのサイズ，サーバの IP アドレス，ポート番号，受信メッセージを表示する．
     (g) (c) に戻り，次の送信の準備をする．
     (h) ソケットをクローズしてプロセスを終了する．

2. 以下に示すような手順で TCP で通信するサーバとクライアントを，隣にすわっている人と分担して作成しなさい．

   - サーバ：
     (a) ソケットをオープンする．
     (b) ソケットにポート番号 49152 を割り当てる．
     (c) クライアントからのコネクション確立要求を待つ．
     (d) コネクションが確立したら，クライアントの IP アドレス，ポート番号を表示する．
     (e) クライアントからのパケット受信を待つ．
     (f) クライアントから受信したメッセージを表示する．
     (g) クライアントから受信したメッセージをそのままクライアントに送信する．
     (h) (e) に戻り，次のパケット受信を待つ．ただしクライアントから受信した文字列が “FIN” であった場合は通信用のソケットを閉じ，(c) に戻って次のコネクション確立を待つ．

- クライアント：
  (a) ソケットをオープンする．
  (b) 標準入力からサーバの IP アドレスを文字列表記で受け取る．
  (c) サーバの 49152 番ポートとの間にコネクションを確立する．
  (d) サーバに送信する文字列を標準入力から受け取る．ただし，EOF が入力された場合は “FIN” という文字列を送信する．
  (e) サーバにデータを送信する．
  (f) サーバからの受信を待つ．
  (g) パケットから受信したメッセージを表示する．
  (h) (d) に戻り，次の送信の準備をする．ただし，サーバから受信した文字列が “FIN” であった場合はソケットをクローズしてプロセスを終了する．