システムプログラミング実験
ネットワーク補助資料

柴田祐樹，下川原英理

ver 1.1 2020 年 12 月 11 日 改訂
ver 1.0 2020 年 10 月 7 日 初版

1　はじめに

本資料ではネットワーク実験の課題を解くために参考となる事柄を説明する．この知識は課題を解くのみならず，家や組織のネットワークを設定する際に役に立ち，20 年は不変な知識であろうと考えている．Raspberry Pi の初期設定や操作方法については別資料 RaspberryPi.pdf を参考にしてもらいたい．

2　パケットの配送

本節ではインターネットにおけるパケット配送の仕組みを，郵送システムの例，抽象的な例，実装例（TCP/IP）と合わせて説明する．これら概念を理解することがインターネット，つまり TCP/IP の構造を理解する第一歩である．

まずはインターネットの語源について説明する．情報交換を行うために構築された通信網を単にネットワークと呼ぶ．これらネットワークは衛星，光ケーブル，Ethernet を使うなど様々な形態があり，適材適所であるため一つに統一することは難しい．そこで，この上位に複数のネットワークにまたがり，通信を中継する役割をもたせたルータを配置し，全てのネットワークを相互結合する規格がインターネットである．inter は相互，net はネットワークを意味するため，日本語にするなら相互通信網であろう．図 1 に 5 つのネットワークが相互接合されたインターネットの例を示す．スター型（Switching hub）のネットワークが 3 つ，1 対 1 結合型が一つ，バス型が一つである．

このような仕組みで情報を伝達するために，データはパケットという単位に分けられ少しずつ送られる．多様なネットワークが相互接続されたインターネットでは，ところにより回線速度は異なり，品質も異なるため，全てのデータを一度に連続して送ることはリスクが高い．これに対し，パケットに分けて送ることで，幾らかパケットが損失したとしても再送要求を出すなどの対処が可能とされている．

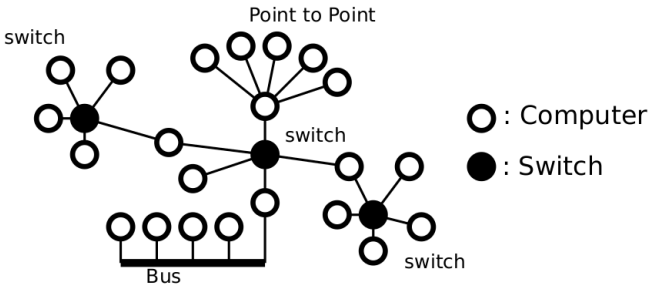
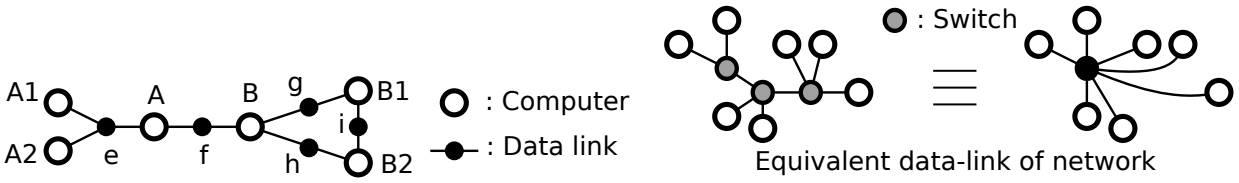


図 1　インターネットの例．白丸は Internet protocol に互換性のある計算機を示す．様々なネットワークが間に経つ計算機で仲介される様子を描いている．



(a) (b)

図 2　IP の説明の例．(a) は仮想化されたリンク，(b) はスイッチングハブによる構成と等価なリンクを示す．白丸はホスト，黒丸はリンク，灰色はスイッチングハブを示す．

上述した例を実現するための規約は単に Internet Protocol (IP) と呼ばれる[*1]．この規約では通信相手がルータであるか普通の計算機であるかを区別しない．インターネットではすべての計算機（ホストと呼ぶ）は対等にルータ（中継器）として振る舞う．計算機の機能をパケットの配送に限定したものをルータと呼ぶが，ホストに対してもパケットの転送が依頼されることは起こりうる．つまり，インターネットに接続できるホストはルータとしての機能を備えている必要が有り，これらに基本的な区別はない [8]．現実の郵送システムでは郵便物は交通網を使い配送されるが，インターネットでは個々のホストがすべて配送を担当するという一般性の高いモデルとなっている．このモデルが現実に採用されているならば，隣の家に郵便物を送ってくれ，などという依頼が一般家庭に来るわけである．もちろん，住民がその要求に必ずしも応じないのと同様に，パケットの転送に対応しないことはインターネット上のホストでも可能であり，その場合は送られてきたパケットをただ無視するか，破棄したことを送り元に通知することが可能である[*2]．

2.1　リンクと経路選択

ここまでの話より，もう少し具体的な例を図 2(a) に描く通信網を使い説明する．この図に置いて丸印はホストを示し，同一の黒丸に接続された端末は同一のネットワークに属することを示す．この例に対し，まずはじめにリンクについて説明する．

IP では，ネットワーク内の通信はそのネットワーク内のプロトコルが担当することを要求し，IP からはネットワーク内の端末は直接通信できるものとみなす．ここでいう直接通信とは IP で定義されるルータを経由せずに通信可能という意味であり，間に別のプロトコルで定義される中継器があっても構わず，その違いは IP では考慮しない．例えば，図 2(b) の左のように幾らかのホストがハブで繋がれていても，それらは IP からは右のようにひとまとめにしてリンク（Data Link）として認識する．ネットワークとは同じリンクで繋がれた集まりである．このように役割を階層分けした結果，IP は図 2(a) の状態だけを考えれば良いことになる．リンクの例として，Ethernet, PPP (Point to point protocol) などが有り，Ethernet は多数のホストをリンクに繋ぐことができるため，図中の e に，PPP は 2 つのホストを繋ぐだけであるため，f や g に用いることができる．

次にパケット転送について説明する．異なるネットワークに属する，つまりリンクを共有しない相手にパケットを送る場合，いずれかのホストにパケットを中継してもらう必要がある．図 2(a) の例で中継を提供する必要があるものは明らかに A, B の 2 つであるが，これらを正しく選ぶために，各端末は送り先に対する中継器とリンクの情報を保持する必要があり，このための参照情報を経路表（Routing table）と呼び，この例では表 1 のように定義する必要が有る．この行では，一列目に送信元，一行目に送信先のホストあるいはリンクを表し，i 行目 j 列目の要素は，i 行目 1 列目のホストが 1 行目 j 列目のホストへパケットを送る場合に次に送る相手，転送先を示す．この経路表は送り先を見つけるために複数回参照される．具体的には，初めに宛先がリンクで繋がっているか確認し，可能ならそのままリンクを使いパケットを送る．リンクで送れない場合には転送してもらうべきルータを探し，該当するものがあればそのルータと繋がるリンクを探す．ルータとつながるリンクすら見つからなければパケットの送信は失敗

[*1] 日本語にすれば相互通信網規約であるが，謙虚な日本の風習に置いてこれほど一般製の高い命名を，考えうる通信方法の一つでしかないものに対し行うということは憚れるため，わざわざ片仮名で呼ぶ習慣があるのだろうか，と思う．英語では単に The internet protocol と呼ぶ．

[*2] ただし，最低でも受け取ってから無視するかどうか決定せなければならず，転送要求がされたからと言ってプログラムが停止する様なこともあってはならない．通信規約ではこういった不必要なことに対する対応方法も規定されている．

表 1　図 2 で全端末が通信を可能とするための経路表．i 行目 j 列目の要素は送信元が i 行 1 列目，送信先が j 列一行目のとき次に送るべき相手を示す

	A	A1	A2	B	B1	B2
A	-	e	e	f	B	B
A1	e	-	e	A	A	A
A2	e	e	-	A	A	A
B	f	A	A	-	g	h
B1	B	B	B	g	-	i
B2	B	B	B	h	i	-

表 2　表 1 に対しデフォルトゲートウェイを導入した例.

	Default	A	A1	A2	B	B1	B2
A	B	-	e	e	f	-	-
A1	A	e	-	e	-	-	-
A2	A	e	e	-	-	-	-
B	A	f	-	-	-	g	h
B1	B	-	-	-	g	-	i
B2	B	-	-	-	h	i	-

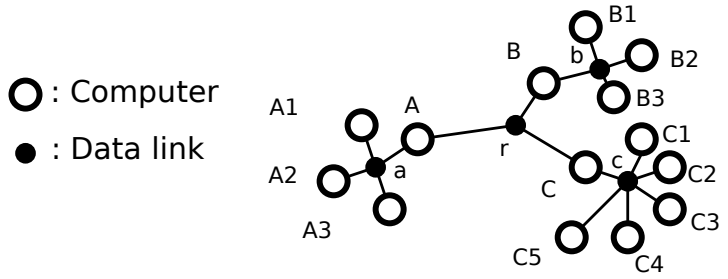


図 3　経路表が図 2 の例に比べより複雑となる例．これに対しまともに経路表を記述したら 14×14 で，対角以外の全ての要素が埋まった随分と大きな表になる．

することとなる．

2.2　効率的な経路指定方法

ここでは効率的な経路表の作成方法を紹介する．表 1 を見てわかるとおり，異なる宛先に対して同じ転送先が多く記述されている．これらはまとめて表した方が良いため，Default gateway (DG) という指定方法が存在する．表 2.2 に DG による指定方法を示す．

DG があれば，図 3 に示すようなもう少し複雑な場合でも，末端のホストは DG を一つ覚えておくだけでよい．たとえば，A1, A2 の DG は RA であるが，RB や RC 側端のホストがどれだけ増えても A1，A2 は DG に RA だけを指定すれば良いため，インターネットの拡張が容易になる．しかしながら，末端にいないホスト，この場合 RA, RB, RC は DG だけ記憶しておくわけには行かないため，このままではすべての端末への経路表を記憶する必要があるため，サブネットという概念を導入することで，宛先をまとめて表現することが IP では可能とされている．ここで，サブネットの概念を IP アドレスを用い説明する前に，文字列の例で一度抽象的に説明しておく．住所区分の概念は一般的には接頭辞を見ることで表現できる．たとえば，図 3 の例では A が接頭辞につくホストへのパケットはすべて A に配送するば良いことは明らかであるため，表 2.2 のような経路表を用意すれば十分である．ただし，* は 0 文字以上からなる任意の文字列を表す．このような転送先を識別するための接頭辞となるものを，TCP/IP ではネットワークアドレスと呼ぶ．また，この方法でまとめられるホストで作られるネットワークをサブネットと呼

表 3　表 2.2 に対しサブネットを導入した例.

	Default	A*	B*	C*	A	B	C
A	-	a	B	C	-	r	r
B	-	A	b	C	r	-	r
C	-	A	B	c	r	r	-
A*	A	a	-	-	-	-	-
B*	B	-	b	-	-	-	-
C*	C	-	-	c	-	-	-

ぶ．これは郵便における日本ー東京都ー日野市ー旭ヶ丘，といった住所区分と同じ仕組みであることは言うまでもない.

2.3　IP による実例

IP アドレスによるネットワークアドレス（接頭辞）とサブネットの指定方法は先の例や住所区分とくらべ，徹底的に一般化されている．まず，アドレスについてだが，使える文字は 0 か 1 のみで，全長 32 の文字列（32bit 整数値）で表現される．しかし 2 進数での記法は長くなるため，人が読みやすい表現方法として，8bit ずつの値，つまり 0-255，をつかい 4 桁で表現したものが，「192.168.1.3」のようなものであり，よく使われる記法である．この記法と 2 進数との対応は以下のとおりである.

192.168.1.3 ≡ 11000000 10101000 00000001 00000011

このアドレスのうちどこまでがネットワークアドレスに対応するかは，二進数の桁数を用いて 192.168.1.3/24 のように「/」以降に接頭辞の長さを書くことで表現される[*3]．また，ネットワークアドレスの範囲に二進数で 1 を置き，それ以降には 0 を置いた数値をサブネットマスクと呼ぶ．例として，192.168.1.3/24 に対応する 2 進数でのネットワークアドレス（Network）とホストのアドレス（Host），サブネットマスクは次のようになる.

$$192.168.1.3/24 \equiv \begin{cases} \text{Network} & = 11000000\ 10101000\ 00000001\ 00000000 \quad (192.168.1.0) \\ \text{Host} & = 11000000\ 10101000\ 00000001\ 00000011 \quad (192.168.1.3) \\ \text{Subnet mask} & = 11111111\ 11111111\ 11111111\ 00000000 \quad (255.255.255.0) \end{cases}$$

ようするに，サブネットマスクとホストアドレスの論理積を取ればネットワークアドレスが得られる．この記法と説明はよく行われるため覚えておいたほうが良い．以下，家庭で使われることはまず無いが，参考までに幾らか例を挙げる.

$$192.168.1.63/30 \equiv \begin{cases} \text{Network} & = 11000000\ 10101000\ 00000001\ 00111100 \quad (192.168.1.60) \\ \text{Host} & = 11000000\ 10101000\ 00000001\ 00111111 \quad (192.168.1.63) \\ \text{Subnet mask} & = 11111111\ 11111111\ 11111111\ 11111100 \quad (255.255.255.252) \end{cases}$$

$$12.168.1.5/12 \equiv \begin{cases} \text{Network} & = 00001100\ 10100000\ 00000000\ 00000000 \quad (12.160.0.0) \\ \text{Host} & = 00001100\ 10101000\ 00000001\ 00000101 \quad (12.168.1.5) \\ \text{Subnet mask} & = 11111111\ 11110000\ 00000000\ 00000000 \quad (255.240.0.0) \end{cases}$$

$$12.168.1.135/26 \equiv \begin{cases} \text{Network} & = 00001100\ 10101000\ 00000001\ 10000000 \quad (12.160.1.128) \\ \text{Host} & = 00001100\ 10101000\ 00000001\ 10000111 \quad (12.160.1.135) \\ \text{Subnet mask} & = 11111111\ 11111111\ 11111111\ 11000000 \quad (255.255.255.192) \end{cases}$$

表 2.3 に図 3 を通信可能とする IP アドレスと経路表の設定方法の一例を示す.

IP アドレスには上記の他に利便性のための特別なアドレスが用意されていて，その一つがすべてが 0 の 0.0.0.0 で表されるメタアドレスと呼ばれるものである．これは，同一リンク内のみで通信する場合に使えるもので，自分自身を示す IP アドレスとみなされる．実際にリンク内でのみ通信を行う場合は IP アドレスは使われないが，この

[*3] これを Classless Inter-Domain Routing (CIDR) [4] 記法という.

表 4　表 2.2 に対しサブネットを導入した例．上部は IP アドレス，下部は経路表を示す．

Host	IP address	Host	IP address	Host	IP address
A	192.168.0.0	B	192.168.0.4	C	192.168.0.8
A1	192.168.0.1	B1	192.168.0.5	C1	192.168.0.9
A2	192.168.0.2	B2	192.168.0.6	C2	192.168.0.10
A3	192.168.0.3	B3	192.168.0.7	C3	192.168.0.11
				C4	192.168.0.12
				C5	192.168.0.13

	Default	192.168.0.0/30	192.168.0.4/30	192.168.0.8/29	192.168.0.0	192.168.0.4	192.168.0.8
A	-	-	192.168.0.4	192.168.0.8	-	r	r
B	-	192.168.0.0	-	192.168.0.8	r	-	r
C	-	192.168.0.0	192.168.0.4	-	r	r	-
A*	192.168.0.0	a	-	-	-	-	-
B*	192.168.0.4	-	b	-	-	-	-
C*	192.168.0.8	-	-	c	-	-	-

IP アドレスはインターネット層を扱うアプリケーションが，自身の IP アドレスが知れない場合であっても，MAC アドレスを指定するなどのリンクの直接的な操作を介さずに同一リンク内で通信を行うために使う．

また，ブロードキャストアドレスというものがあり，これは 255.255.255.255 と表される．このアドレスはリンク内のすべてのホストに同じパケットを送るためのアドレスである．リンク内の通信に IP を使う必要なこちらに置いても必ずしも無いが，先ほどと同様にアプリケーションからリンクのアドレスを指定する手間を省くために存在する．

これらブロードキャストアドレスとメタアドレスはインターネットにおけるホストの指定に使われることはない．この他にも，IP アドレスにはループバックアドレス，プライベート IP アドレスなど幾らか特殊なものが有る [3]．しかし，これらは重要であるが，ここで述べるほど複雑なものでもなく，明快な説明が Web 上に多く存在するため各自調べておいてもらいたい．

2.4　慣習

IP の設定方法には幾らか慣習が有り，そのなかで実用上混乱するため覚えて置いたほうが良いものが幾らか有る．

1. 有向ブロードキャストアドレス（Directed broadcasts）が有り，これはホストに使わない
2. ネットワークアドレスはホストに使わない
3. ホストが接続するネットワークごとに IP アドレスを設定する

まず 1 についてだが，255.255.255.255 の他に，ネットワークアドレスとサブネットマスクの否定の論理和を取ったアドレスを，そのネットワークに限ったブロードキャストアドレスと見做すという慣習が，攻撃に利用される危険性の高さから現在実質使われていないが存在する [6][*4]．例えば第 2.3 節の例に有向ブロードキャストアドレスとサブネットマスクを併記すれば，以下のようになる．

$$
192.168.1.3/24 \equiv \left\{ \begin{array}{ll}
\text{Network} & = 11000000\ 10101000\ 00000001\ 00000000 \quad (192.168.1.0) \\
\text{Host} & = 11000000\ 10101000\ 00000001\ 00000011 \quad (192.168.1.3) \\
\text{Subnet mask} & = 11111111\ 11111111\ 11111111\ 00000000 \quad (255.255.255.0) \\
\text{Directed broadcasts} & = 11000000\ 10101000\ 00000001\ 11111111 \quad (192.168.1.255)
\end{array} \right.
$$

[*4] 規約の悪用は不毛であるため危険性の具体的な説明はここでは省く．

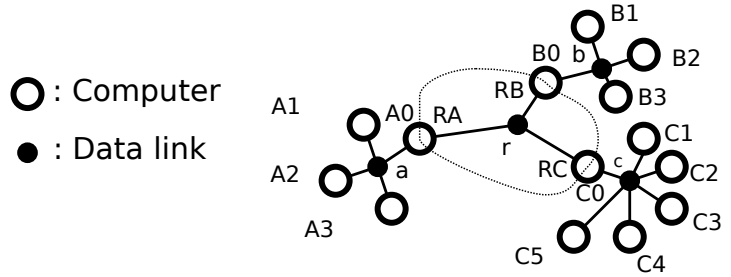


図 4　多重にアドレスを割り当てる方法を図 3 の例に適用した例.

表 5　図 4 に対応した経路表.

	Default	R*	A*	B*	C*
R*	-	r	RA	RB	RC
A*	A0	-	a	-	-
B*	B0	-	-	b	-
C*	C0	-	-	-	c

$$12.168.1.5/12 \equiv \begin{cases} \text{Network} & = 00001100\ 10100000\ 00000000\ 00000000 & (12.160.0.0) \\ \text{Host} & = 00001100\ 10101000\ 00000001\ 00000101 & (12.168.1.5) \\ \text{Subnet mask} & = 11111111\ 11110000\ 00000000\ 00000000 & (255.240.0.0) \\ \text{Directed broadcasts} & = 00001100\ 10101111\ 11111111\ 11111111 & (12.183.255.255) \end{cases}$$

上記に加えて，ネットワークアドレスもまた有向ブロードキャストアドレスとして用いるという慣習がかつてあったが，同様に危険だという理由で現在は使われていない [7]．つまりかつては自身のネットワーク外からの有向ブロードキャストキャストに備えるために，ネットワークアドレスと有向ブロードキャストアドレスの 2 つはホストに用いないという慣習があったが，現在はこのパケットが送られてくることはないためこれらもホストに用いることができる．

最後に，慣習 3 は経路表縮小に貢献するため IP アドレスの無駄遣いではあるが有用であり広く用いられる．図 3 を図 4 のようにルータに対し接続するネットワーク毎にそれぞれ適したアドレスを割り当てることで，経路表表 2.2 を表 2.4 のように書き換えることができる．この方式は経路表を小さくすることができるため主記憶の小さいルータでは利点が有るが，一度決めたサブネットの大きさは 2 の冪乗でしか指定できないため多くの場合に無駄が生じる．IP version 6 のアドレス数ならば地球上の石の数より多いため[*5]，このような無駄は許容されるが，IP version 4 でそれは実用上無視できない.

以上の慣習は広く広まっており，民間用の機器ではこれら慣習を踏まえた設定しかできないものが多いので気をつけてもらいたい．Linux ならば無視した一般的な設定が可能である.

2.5　Linux での IP

Linux なら，以下のコマンドで IP の設定と確認ができる.

- ip addr: IP アドレス設定の表示
- ip route: 経路表の表示
- ip route add: 経路表の要素の追加
- ip route delete: 経路表の要素の削除
- ip addr add dev n1: ネットワークアダプタ n1 へ IP アドレスを追加.
- ip addr delete dev n1: ネットワークアダプタ n1 から IP アドレスを削除.

[*5] 地球の質量は 6×10^{24}kg ほどであり，砂粒一つが 10^{-6}kg だとしても，6×10^{30} 個の砂粒が存在可能である．これに対し IP version 6 のアドレス数は $2^{128}\approx3.4\times10^{38}$ である.

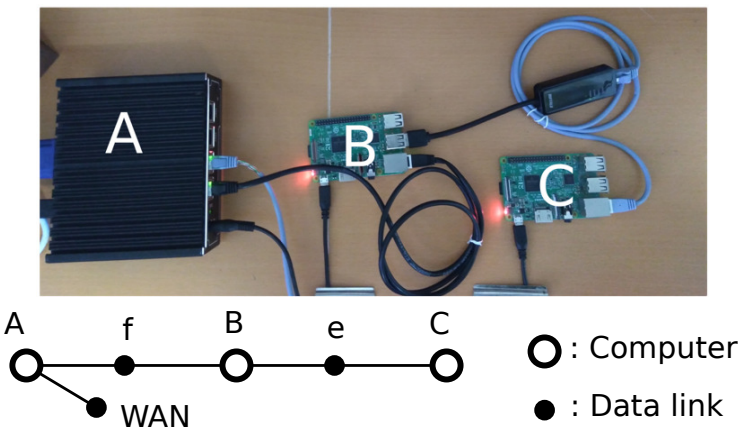


図 5　Raspberry Pi による実験例.

表 6　Raspberry Pi による実験用アドレスと経路表.

	Address	Default	A	B	C
A	192.168.1.2	WAN	-	f	B
B	10.8.8.8	A	f	-	e
C	172.19.4.1	B	-	e	-

以下，上述したコマンドを用い，図 5 のように構成された RasPi のシステムおよび下部に描かれたその論理構成に対し，IP アドレスと経路表を表 2.5 の通り設定して実験を行う手順とその結果を示す.

まずは各ホストに接続されたネットワークアダプタを列挙するために，コマンド ip addr を実行する．Dynamic Host Configuration Protocol (DHCP) [5] が Raspberry Pi では初期状態で有効になっており，DHCP 付きのルータに接続していれば IP アドレス等が自動設定されるため，図 6 の様な出力がなされるはずである．さらに経路情報を見るために図中では ip route を続けてした結果を示している．この結果からは，10 行目より IP アドレスに 192.168.0.2 が設定されていることがわかり，デフォルトゲートウェイは 17 行目より 192.168.0.1 であることが分かる．また，18 行目より 192.168.0.0 - 192.168.0.255 の範囲の IP は eth0 に繋がるリンクに存在すると仮定していることも分かる．なお，wlan0 は WiFi 用のアダプタであり，WiFi を使っている場合はこちらに設定が行われるはずである.

次に，自動設定が行われると実験に支障を来すため，これをコマンド sudo systemctl disable dhcpcd を実行した後，コマンド reboot により RasPi を再起動して，自動設定を無効化する．この状態で調べた ip addr と ip route の結果を図 7 に示す．ループバックを除き何も設定されていないことが分かる.

この状態で，次のコマンド,

```
ip link set eth0 up
ip addr add 172.19.4.1 dev eth0
ip route add 10.8.8.8 dev eth0 scope link
ip route add default via 10.8.8.8
```

を図 5 中 C の RasPi で実行し，また，次のコマンド

```
ip link set eth0 up
ip link set eth1 up
ip addr add dev eth0 10.8.8.8
ip addr add dev eth1 10.8.8.8
ip route add 192.168.1.2 dev eth0 scope link
ip route add 172.19.4.1 dev eth1 scope link
```

```
$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
    1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
    default qlen 1000
    link/ether b8:27:eb:d8:90:8c brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.2/24 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::ba27:ebff:fed8:908c/64 scope link
        valid_lft forever preferred_lft foreverreboo
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether b8:27:eb:8d:c5:d9 brd ff:ff:ff:ff:ff:ff
$ ip route
default via 192.168.0.1 dev eth0
192.168.0.1/24 dev eth0 proto kernel scope link
```

図 6　Dynamic Host Configuration Protocol により設定された Raspberry Pi のネットワーク設定例.

```
$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
    1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether b8:27:eb:d8:90:8c brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether b8:27:eb:8d:c5:d9 brd ff:ff:ff:ff:ff:ff
$ ip route
```

図 7　自動設定なしの Raspberry Pi のネットワーク設定例.

```
1  $ ping 192.168.1.2 −c 4
2  PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
3  64 bytes from 192.168.1.2: icmp_seq=1 ttl=63 time=1.17 ms
4  64 bytes from 192.168.1.2: icmp_seq=2 ttl=63 time=1.40 ms
5  64 bytes from 192.168.1.2: icmp_seq=3 ttl=63 time=1.38 ms
6  64 bytes from 192.168.1.2: icmp_seq=4 ttl=63 time=1.39 ms
7
8  −−− 192.168.1.2 ping statistics −−−
9  4 packets transmitted, 4 received, 0% packet loss, time 3004ms
10 rtt min/avg/max/mdev = 1.179/1.340/1.407/0.103 ms
11
12 $ ping 10.8.8.8 −c 4
13 PING 10.8.8.8 (10.8.8.8) 56(84) bytes of data.
14 64 bytes from 10.8.8.8: icmp_seq=1 ttl=64 time=0.658 ms
15 64 bytes from 10.8.8.8: icmp_seq=2 ttl=64 time=0.643 ms
16 64 bytes from 10.8.8.8: icmp_seq=3 ttl=64 time=0.616 ms
17 64 bytes from 10.8.8.8: icmp_seq=4 ttl=64 time=0.629 ms
18
19 −−− 10.8.8.8 ping statistics −−−
20 4 packets transmitted, 4 received, 0% packet loss, time 3099ms
21 rtt min/avg/max/mdev = 0.616/0.636/0.658/0.029 ms
```

図 8　Ping を実行した結果．試行回数は 4 である．

```
ip route add default via 192.168.1.2
```

を B の RasPi で実行する．コマンド ip link set はアダプタを利用可能状態とするために必要である．さらに，A において，

```
ip link set enp4s0 up
ip addr add 192.168.1.2 dev enp4s0
ip route add 10.8.8.8 dev enp4s0 scope link
ip route add 172.19.4.1 via 10.8.8.8
```

を実行する．この状態で，C から ping 192.168.1.2 -c 4 と ping 10.8.8.8 -c 4 を実行した結果を図 8 に示す．この結果より，B のホストを経由しなければならない 1 つめの結果において，Time to Live [10] の値が 2 つめの結果より減少していることから，確かに一度中継されて応答が返ってきていることを確認できる．また，10, 21 行目の統計値を比較すれば，中継を要した 10 行目の結果のほうが長い時間となっていて，確かに中継処理が何かしら行われていることを確認できる．

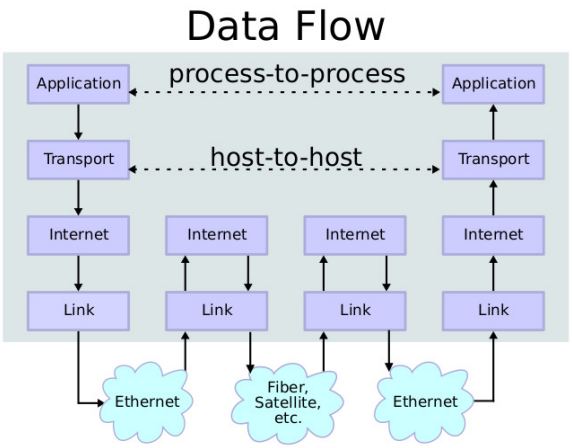


図 9　TCP/IP 階層の概念図 [12].

3　階層モデル

前節まででリンクとそれらを相互結合したインターネットについて説明したが，この内容だけでは物理的な伝送方式や，欠落した情報の再送要求，また実際のアプリケーションの組み方等が定義されていない．これらを実用するために他の必要な機能を揃えたモデルに，TCP/IP プロトコル・スタック [12] がある．

TCP/IP の階層モデルにおいては，物理層は普通含まれないが，ここでは参考のために物理層を含めて考えることにする．このモデルに関する文献は多量に存在するためここでは詳細は省くが，TCP/IP で発生するデータの流れの概略は図 9 に示したとおりであり，前節で説明していたのはこの内 Internet，Link と書かれた部分である．この他に通信の信頼性を向上させ，プロセスを識別するための Transport 層，アプリケーション毎の実装をおこなう Application 層がある．よく言われるように各層に属するプロトコルは他の層の通信状況を関知しなくてよいため，論理的に各層からはそれぞれ図 10 に示すとおりに見えると言える．物理（Physical）層では使う電磁波，端末の形状，ケーブルの長さや導線に使う物質まで規定する必要がある．その次のリンク層では物理的な要請からは離れて，ネットワーク内のホスト間通信のみを規定する．さらに上位のインターネット層では，ネットワーク内の Data link はすでに確立されているものとみなし，ホスト間のパケットの転送やその経路維持のみを管理する．

もう一つ上位の層に Transport 層があり，同一ホスト内に複数存在するプロセスを識別し，それらの間の通信を確立する．図 10 の Transport 層中の丸はホスト，その中の四角はプロセスを示し，色はアプリケーション毎に塗られている．このとき，ホストまでデータを運ぶことはインターネット層の役目であるため，この層ではプロセス間のデータ転送のみを扱うことになる．この層は連続して送られてきたパケットを記憶域に保持しアプリケーションが利用するのを待機したり，紛失したパケットの再送要請を出すなど通信の信頼性確保においても貢献する．

そして最上位の Application 層では Transport 層で接続されたプロセスのうち，互換性のあるアプリケーションを識別して，それらアプリケーションに固有の通信を行う．トランスポート層はどんなプロセス同士も接続するため，アプリケーション層では通信内容がアプリケーションの動作に適しているか常に検証をし誤動作を防ぐように設計しなくてはならない．

より詳細な内容はそれぞれについて他の文献を参考にしてもらいたい．また，他の文献にもを通して置かれるのが良いだろう．

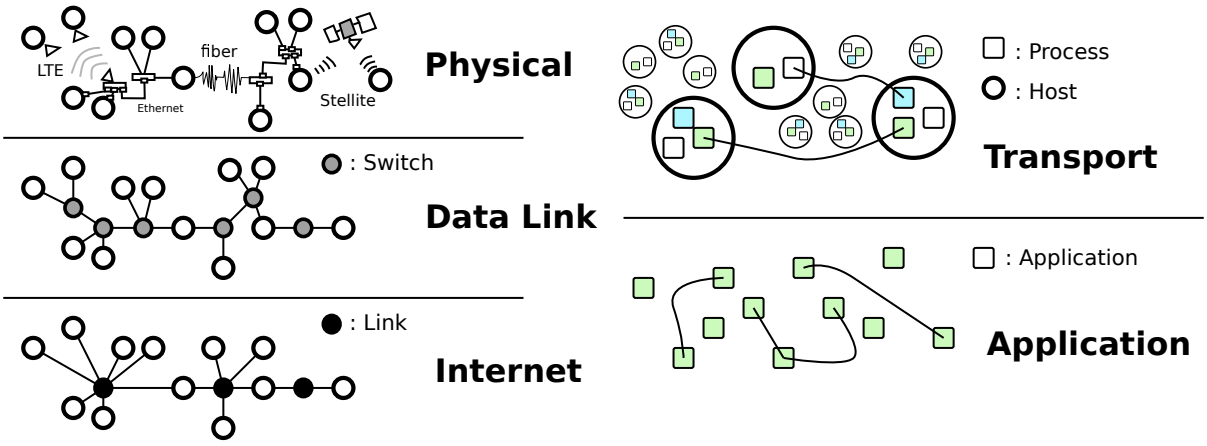


図 10　TCP/IP の各階層における対応範囲の遷移.

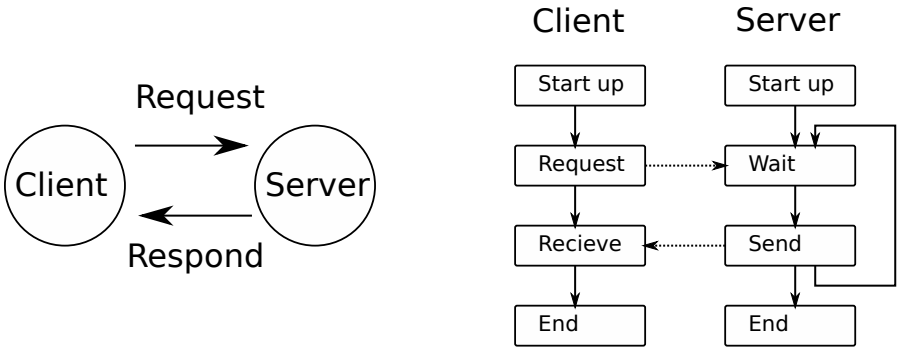


図 11　クライアント・サーバモデル概念図.

4　クライアント・サーバモデル

アプリケーション層ではアプリケーションを実装するわけであるが，そのうち，ネットワークアプリケーションプログラムにおける標準的な計算モデルとしてクライアント・サーバ（Client-server）モデル [11] がある．図 11 に概念図を示す．クライアント，サーバというのは通信を構成する 2 つのアプリケーションプログラムを指す．通信を能動的に起動する側（サーバに対してサービスを要求する側）をクライアントと呼び，受動的に待つ側（クライアントからの要求に対してサービスを提供する側）をサーバと呼ぶ．ここで，クライアント，サーバという概念は本来プログラム（またはプロセス）に付随するものであることに注意する必要がある．よく「ワークステーション A はファイルサーバである」というような言い方をするが，これは「ワークステーション A でファイルサーバプログラムが実行されている」ことを指している[*6]．

図 11 右に示すとおり，クライアント－サーバモデルに基づいたアプリケーションプロセスの典型的な処理の流れは以下のようになる．

1. ある計算機上でサーバプロセスが起動される．サーバプロセスはクライアントプロセスがサービスを要求してくるまで待ち状態になる．
2. 同じ計算機上，またはネットワーク接続された別の計算機上でクライアントプロセスが起動される．起動されたクライアントプロセスはある種のサービスに対する要求をサーバプロセスに送信する．
3. サーバプロセスはクライアントプロセスからのサービス要求を処理し，処理結果のデータをクライアントプロセスへ返送する．
4. サービスの提供が終わると，サーバプロセスは再び待ち状態に入り，次のクライアントプロセスからの要求

[*6] 情報通信システム工学科，酒井和也，2015.

を待つ.

5 ソケット通信

本節では Linux（Unix）上で実際にアプリケーション層で動作する通信プログラムを開発するための事柄を説明する.

プログラマがあるシステムの上でアプリケーションプログラムを開発するときに提供されるインタフェースを API (Application Programming Interface) という．Unix 系の OS では API はシステムコールと呼ばれており，プログラマは関数の形でそれらを利用できる．OS の基本的なサービスであるファイル入出力に関していえば，C 言語のプログラマに対して read や write といったシステムコールが提供されている．アプリケーションプログラムでファイル入出力が必要になったときに read や write を呼び出すことにより制御を OS に渡す．実際のファイル入出力の制御は OS の内部手続きにより行われる．OS は要求された操作を終えた後，制御をアプリケーションプログラムに戻す．プログラマの側では，入出力に関する OS の動作やハードウェアの詳細について知る必要はなく，通常の C 言語の関数と同じように定義されたシステムコールの外部仕様のみを覚えておけばよい．このようなシステムの階層化設計による情報隠蔽によりプログラマの苦労が軽減される.

ファイルに対する入出力の場合と同様に，ネットワークを介してプロセス間通信を行う場合も，アプリケーションプログラムからトランスポート層プロトコルを利用するための API がシステムコールとしてユーザに提供されている．これにより，アプリケーションプログラマは他の標準関数を利用するのと同様の書式でシステムコールを利用して TCP や UDP にアクセスできる．このとき TCP や UDP の詳細を知っている必要はないのでプログラミングが容易になる．プログラマは API を利用するときに必要になるサーバの IP アドレスや使用するトランスポート層プロトコルに関する情報だけを気にしていればよい.

一般に，どのような API が利用できるかは使用している OS とプログラミング言語に依存する．本実験ではソケットという API 群を用いてプロセス間通信を行うための方法について学ぶ.

5.1 通信に必要な情報

ファイル入出力ではファイルパスだけを知っていればよかったが，インターネットを介した通信のためには以下に示す複数の情報が必要となる.

1. トランスポート層プロトコル　（例：TCP，UDP）
2. 送信元のホスト（IP アドレス）
3. 送信先のホスト（IP アドレス）
4. 送信元のプロセス（ポート番号）
5. 送信先のプロセス（ポート番号）

プロトコル体系によりアドレス等の表現が異なるので，あるプロセス間通信を指定するには，まずそのプロセス間通信で使用されるトランスポート層プロトコルを指定する必要がある．ここでは，使用するプロトコル体系として TCP/IP を仮定しているので，TCP と UDP のどちらかを指定することになる．TCP はコネクション指向のプロトコルであり，実際の通信に先立ち接続を確立するための手続きが必要となる．接続の確立後は，トランスポート層同士で受信確認応答と再送が自動的に行われ，通信の信頼性（送信した情報（パケット）が送信した順序で相手に受信されること）が保証される．すなわち，接続を確立した後は，ファイルシステムに対するのと同様なストリーム型のアクセスが可能となる．一方，UDP はコネクションレスのプロトコルであり，接続を確立せずにデータグラム（パケット）を送受信する．このため，通信の信頼性は保証されない.

ホストを指定するためには IP アドレスを用い，加えて，ホスト内に存在する複数のプロセスの内目的のものを指定する必要が有る．TCP/IP の場合，ポート番号と呼ばれる 2 バイト整数でプロセスを指定することになっている．通信に先立って，各プロセスは自分の利用したいポート番号を OS に申請し，割り当てられる必要がある．ポート番

号のうち，0 から 1023 まではウェルノウンポートであり，既存の主要なアプリケーションサービス（HTTP, FTP, Telnet 等）のために確保されている．ウェルノウンポートを使用するためには，管理者権限が必要となる．1024 から 65535 までのポート番号は一般ユーザが使用可能であるが，このうち 1024 から 49151 までのポート番号は登録済みポート番号であり，既存のサービスのために予約されている．49151 から 65535 まではダイナミック/プライベートポートであり，ユーザが自由に使用できる [13].

5.2 ソケットを用いたクライアント・サーバモデルのプログラム構造

ここでは，クライアント – サーバモデルに基づいたプログラムの実際の構造について説明する．説明で使用されているシステムコールの仕様の詳細については付録を参照せよ．

5.2.1 コネクションレスの場合

コネクションレスの通信を行う場合のクライアントプログラム，サーバプログラムの典型的な動作の流れは以下のとおりである．UDP ではコネクションレス通信を用いる．データを転送するのに相手のアドレスを指定する必要があるため，ソケットへの読み書きには sendto と recvfrom を用いる．各関数の概略については本書末尾の付録を参考にしてもらいたい．

サーバ側の動作は以下のようになる．

1. socket により通信に使うソケットを生成する．
2. bind により通信に使う IP アドレスとポート番号を指定する．
3. recvfrom によりソケットから記憶域内へデータを読み込む．データが到着していない場合は，新たにデータが到着するまでサーバプロセスをブロックする（ブロッキング通信）．
4. 受け取ったデータに従い，所定の処理を行う．
5. sendto により応答をソケットに書き込むことでクライアントへ処理結果を送る．
6. 3 から 5 を繰り返し複数のクライアントへ対応する．

クライアント側の動作は以下のようになる．

1. socket により通信に使うソケットを生成する．
2. sendto によりデータを引数に指定したサーバへ送信する．利用するポート番号は OS が適切なものを自動的に選ぶが，これより前に bind を呼び出し指定することもできる．
3. recvfrom によりソケットに到着したデータを読み取る．通常は，sendto への応答が格納される．

5.2.2 コネクション指向の場合

コネクション指向の通信を行う場合のサーバプログラムの典型的な動作の流れは以下のようになる．コネクションレスの場合に比べ，接続を確立するための手続きが増えているが，接続確立後はアドレスなどを指定する必要がない．各関数の概略については本書末尾の付録を参考にしてもらいたい．

1. socket により通信に使うソケットを生成する．
2. bind により通信に使う IP アドレスとポート番号を指定する．
3. listen によりクライアントからの接続要求に対する待ち行列の生成を OS に依頼する．また，これによりソケットが接続要求を受付可能な状態になる．
4. accept により待ち行列に格納されているクライアントからの接続要求に対応するソケットを生成し，接続を確立する．待ち行列がからである場合は新たな接続要求が来るまで処理を停止する．呼び出しが成功した場合はこの接続に対応したソケットの記述子が返される．以降クライアントとの通信には最初に生成したソケットではなく，ここで生成されたソケットを用いる．
5. read によりソケットからデータを記憶域に読み込む．

6. 読み込んだデータに従い処理を行う．
7. write により処理結果をクライアントに送る．

クライアント側の動作は次のとおりである．サーバに比べ簡単である．

1. socket によりソケットを生成する．
2. connect によりサーバ側のソケットへ接続を試みる．
3. write によりソケットへデータを書き込むことでサーバへ処理を依頼する．
4. read によりサーバの処理結果を受け取る．

5.2.3 参考プログラム

以上説明した方式の参考として，サーバ側から時刻を取得するプログラムを以下の通り用意している．

- udp_time_client.cpp: UDP を用いた時刻取得を行うクライアントプログラム
- udp_time_server.cpp: 上記クライアントに応答するサーバプログラム
- tcp_time_client.cpp: TCP を用いた時刻取得を行うクライアントプログラム
- tcp_time_server.cpp: 上記クライアントに応答するサーバプログラム

上記プログラムは全て以下のコマンドでコンパイル可能である．

```
g++ −std=c++14 udp_time_client.cpp −o udp_time_client
g++ −std=c++14 udp_time_server.cpp −o udp_time_server
g++ −std=c++14 tcp_time_client.cpp −o tcp_time_client
g++ −std=c++14 tcp_time_server.cpp −o tcp_time_server
```

何度もこのコマンドを記述するのは大変なので，make.sh などとしてテキストファイルに保存し，

```
bash make.sh
```

のように実行すると良い．本プログラムで通信相手の IP アドレスは 127.0.0.1 となっているが，適宜ここを書き換えて通信相手を変更してもらいたい．また，ポート番号も必要に応じて変更してもらいたい．

参考文献

[1] Internet protocol suite, Wikipedia. https://en.wikipedia.org/wiki/Internet_protocol_suite (Accessed 2020/11/30)
[2] RFC1878. https://tools.ietf.org/html/rfc1878 (accessed Dec., 8th, 2020)
[3] RFC5735. https://tools.ietf.org/html/rfc5735 (Accessed 2020/12/8)
[4] RFC4632. https://tools.ietf.org/html/rfc4632 (Accessed 2020/12/8)
[5] RFC2131. https://tools.ietf.org/html/rfc2131 (Accessed 2020/12/8)
[6] RFC 2644. https://tools.ietf.org/html/rfc2644 (Accessed 2020/12/8)
[7] RFC 1812. https://tools.ietf.org/html/rfc1812#page-93 (Accessed 2020/12/8)
[8] RFC1122. https://tools.ietf.org/html/rfc1122#section-1.1.1 (Accessed 2020/12/8)
[9] iproute2, Wikipedia. https://en.wikipedia.org/wiki/Iproute2 (Accessed 2020/12/8)
[10] Time to live, Wikipedia. https://ja.wikipedia.org/wiki/Time_to_live (Accessed 2020/12/8)
[11] Client-sever model, Wikipedia. https://en.wikipedia.org/wiki/Client%E2%80%93server_model (Accessed 2020/12/8)
[12] Comparison of TCP/IP and OSI layering, Wikipedia. https://en.wikipedia.org/wiki/Internet_protocol_suite#Comparison_of_TCP/IP_and_OSI_layering (Accessed 2020/12/8)

[13] List of TCP and UDP port numbers, Wikipedia. https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers (Accessed 2020/12/8)

付録: 関数

プログラム作成に必要となるシステムコール，ライブラリ関数，構造体について解説する．

システムコール

- socket
  通信で使用されるソケットを生成し，ソケット記述子を返す．
  - ヘッダ: ⟨sys/types.h⟩, ⟨sys/socket.h⟩
  - Prototype: int socket(int family, int type, int protocol);
  - 引数 family: プロトコルファミリ（PF_INET 等）．
  - 引数 type: サービス型．TCP なら SOCK_STREAM, UDP なら SOCK_DGRAM.
  - 引数 protocol: プロトコル番号．0 の場合，OS が適当なプロトコルを勝手に選択する．
  - 戻り値: 呼び出しが成功した場合ソケット記述子（0 以上），エラーが起きた場合 –1.
- bind
  ソケットが使う IP アドレスとポート番号を指定する．ここで指定した IP アドレスとポートの組に対して送られてきたパケットのみを呼び出し元のプロセスは読み取る．
  - ヘッダ: ⟨sys/types.h⟩, ⟨sys/socket.h⟩
  - Prototype: int bind(int socket, struct sockaddr *addr, int addrlen);
  - 引数 socket: ソケット記述子．システムコール socket で取得したもの．
  - 引数 addr: IP アドレスとポート番号が格納された sockaddr 型の構造体へのポインタ．
  - 引数 addrlen: 第 2 引数のバイト長．
  - 戻り値: 呼び出しが成功した場合 0 を，エラーが起きた場合 –1.

  IP アドレスとして INADDR_ANY を指定した場合，そのホストの全ての IP アドレスに自動的に割り当てられる．
- listen
  サーバにより呼び出され，サーバ側のソケットがクライアントからの要求を受付可能な状態にする．また，あるソケットに対するコネクション要求の待ち行列の長さも指定する．呼び出しが成功した場合 0 を，エラーが起きた場合-1 を返す．
  - ヘッダ: ⟨sys/types.h⟩, ⟨sys/socket.h⟩
  - Prototype: int listen(int socket, int queuelen);
  - 引数 socket: ソケット記述子．システムコール socket で取得したもの．
  - 引数 queuelen: コネクション要求の待ち行列の長さ（通常，5 以下の値を指定する）．
  - 戻り値: 呼び出しが成功した場合 0 を，エラーが起きた場合 –1
- accept
  TCP を使用している場合に用いられる．サーバにより呼び出され、クライアントからのコネクション要求を待ち行列から取り除く．待ち行列が空の場合は次の要求が到着するまで待つ．その後，その要求に対して新しいソケットを生成する．また，クライアントのアドレス情報が，引数で指定された領域に格納される．ポインタ addrlen で示されたアドレスには，addr に書き込み可能なバイト長を格納しておく必要がある．
  - ヘッダ: ⟨sys/types.h⟩, ⟨sys/socket.h⟩
  - Prototype: int accpet(int socket, struct sockaddr *addr, int *addrlen);
  - 引数 socket: ソケット記述子．

  - 引数 addr: クライアントの情報を格納するための構造体へのポインタ.
  - 引数 addrlen: 第 2 引数のバイト長.
  - 戻り値: 生成されたソケットに対するソケット記述子．エラーが起きた場合 –1.
- connect
  クライアントプロセスにより呼び出され，（TCP を使用している場合）サーバとの接続を確率する．UDP を用いている場合は、単に connect 呼び出し以降のサーバのアドレス指定の省略が可能になるだけである.
  - ヘッダ: ⟨sys/types.h⟩, ⟨sys/socket.h⟩
  - Prototype:
  - 引数: ソケット記述子.
  - 引数 addr: サーバの IP アドレスとポート番号を格納した構造体へのポインタ.
  - 引数 addrlen: 第 2 引数のバイト長.
  - 戻り値: 呼び出しが成功した場合 0 を，エラーが起きた場合 –1
- read
  ソケットからデータを読み込む．入力を得ると読み込んだバイト数を返す.
  - ヘッダ:
  - Prototype: int read(int socket, char *buf, int buflen);
  - 引数 socket: ソケット記述子.
  - 引数 buf: 入力データを読み込むための記憶域へのポインタ.
  - 引数 buflen: 第 2 引数に指定した記憶域のバイト数.
  - 戻り値: 読み込んだバイト数．ただし，ソケットのファイル終了状態が検出されたときは 0，エラーが起きた場合は –1.
- write
  ソケットにデータを書き込む.
  - ヘッダ:
  - Prototype: int write(int socket, char *buf, int buflen);
  - 引数 socket: ソケット記述子.
  - 引数 buf: 出力データが記載されている記憶域へのポインタ.
  - 引数 buflen: 第 2 引数に指定した出力データのバイト数．記憶域の大きさを指定するわけではないことに注意.
  - 戻り値: 出力に成功すると書き込んだバイト数．エラーが起きた場合は –1.
- sendto
  指定された受信者に対してメッセージ送信を行う．送信先アドレスを引数で指定する．主に UDP で使う.
  - ヘッダ: ⟨sys/types.h⟩, ⟨sys/socket.h⟩
  - Prototype: int sendto(int socket, char *msg, int msglen, int flags, struct sockaddr *to, int tolen);
  - 引数 socket: ソケット記述子.
  - 引数 msg: 送信データが格納された記憶域へのポインタ
  - 引数 msglen: 送信データのバイト数．記憶域の大きさではないことに注意.
  - 引数 flags: 制御用整数値．通常は 0 で良い.
  - 引数 to: 送信先の情報が格納された構造体へのポインタ
  - 引数 tolen: to のバイト数.
  - 戻り値: 呼び出しが成功した場合送られたバイト数．エラーが起きた場合 –1.
- recvfrom
  ソケットに到着したメッセージを得る．また，送信元アドレスの情報が，引数で指定された領域に格納される．ポインタ fromlen で示されたアドレスには，送信元アドレス (from) に書き込み可能なバイト長を格納しておく必要がある．呼び出し終了後，from には送信元アドレスの情報が，*fromlen には送信元アドレスのバ

イト長が格納される．
  - ヘッダ: ⟨sys/types.h⟩, ⟨sys/socket.h⟩
  - Prototype: int recvfrom(int socket, char *buf, int buflen, int flags, struct sockaddr *from, int *fromlen);
  - 引数 socket: ソケット記述子．
  - 引数 buf: 受信内容を格納する記憶域へのポインタ．
  - 引数 msglen: buf に指定した記憶域の大きさ．
  - 引数 flags: 制御用整数値．通常は 0 で良い．
  - 引数 from: 送信元の情報を格納するための構造体へのポインタ．
  - 引数 fromlen: 構造体 from のバイト数．
  - 戻り値: 呼び出しが成功した場合受け取ったバイト数．エラーが起きた場合 −1．
- close
  通信を終了させ，ソケットをシステムから取り除く．この関数を呼び出さずにソケットの生成を続けていると，メモリリークが発生するので注意する．
  - ヘッダ: ⟨sys/types.h⟩, ⟨sys/socket.h⟩
  - Prototype: int close(int socket);
  - 引数 socket: ソケット記述子．
  - 戻り値: 呼び出しが成功した場合 1 を，エラーが起きた場合 −1

ライブラリ

- inet_addr
  Dotted-decimal 形式の IP アドレス（“133.86.20.1” など）を 4 バイト整数値形式へ変換する．
  - ヘッダ: ⟨sys/types.h⟩, ⟨netinet/in.h⟩, ⟨arpa/inet.h⟩
  - Prototype: unsigned_long inet_addr(char *ptr);
  - 引数 ptr: Dotted-decimal 形式で記載された IP アドレスの文字列へのポインタ．
  - 戻り値: IP アドレスの整数値表現．
- inet_ntoa
  4 バイト整数値形式の IP アドレスを Dotted-decimal 形式の IP アドレスの文字列表現へ変換する．
  - ヘッダ: ⟨sys/types.h⟩, ⟨netinet/in.h⟩, ⟨arpa/inet.h⟩
  - Prototype: char *inet_ntoa(struct in_addr inaddr);
  - 引数 inaddr: IP アドレスの整数値表現を格納するための変数へのポインタ．
  - 戻り値: Dotted-decimal 形式の IP アドレスの文字列へのポインタ．
- time
  現在時刻を取得し，指定されたアドレスに格納する．呼び出しが成功した場合には現在時刻を返す．指定されたアドレスに格納される値は返戻値と同一である．なお，現在時刻は UNIX 時刻，すなわち 1970/01/01 00:00:00(UTC) からの経過秒数で表わされる．
  - ヘッダ: ⟨time.h⟩
  - Prototype: time_t time(time_t *timer);
  - 引数 timer: 時刻格納場所へのポインタ．
  - 戻り値: 呼び出しが成功した場合には現在時刻．エラーが起きた場合には −1．
- ctime
  UNIX 時刻を，ホストに設定された地域の現地時刻として” 曜日　月 日　時：分：秒　年” の形式の文字列に変換し，
  - ヘッダ:⟨time.h⟩

– Prototype: char *ctime(const time_t *timer);
– 引数 timer: 時刻格納場所へのポインタ.
– 戻り値: 生成された文字列へのポインタ.