ネットワークプログラミング

千代浩司
高エネルギー加速器研究機構
素粒子原子核研究所

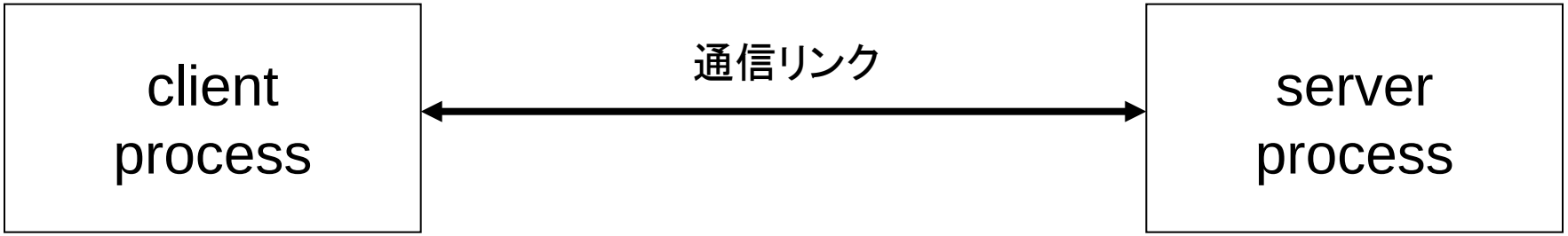
もくじ

- 前提知識
  - TCP/IP (IPアドレス、ポート、TCP)
  - アプリケーションプロトコル
  - ネットワークバイトオーダー
- TCPでデータを読むまでに使う関数
  - socket(), connect(), read()/write()
- プログラムを書くときの情報のありか、エラー処理
  - マニュアルページの読み方
  - エラー捕捉法、メッセージの表示
- 実際にネットワークを使って読むときの注意
  - ソケットレシーブバッファ

もくじ

- 前提知識
  - TCP/IP (IPアドレス、ポート、TCP)
  - アプリケーションプロトコル
  - ネットワークバイトオーダー
- TCPでデータを読むまでに使う関数
  - socket(), connect(), read()/write()
- プログラムを書くときの情報のありか、エラー処理
  - マニュアルページの読み方
  - エラー捕捉法、メッセージの表示
- 実際にネットワークを使って読むときの注意
  - ソケットレシーブバッファ

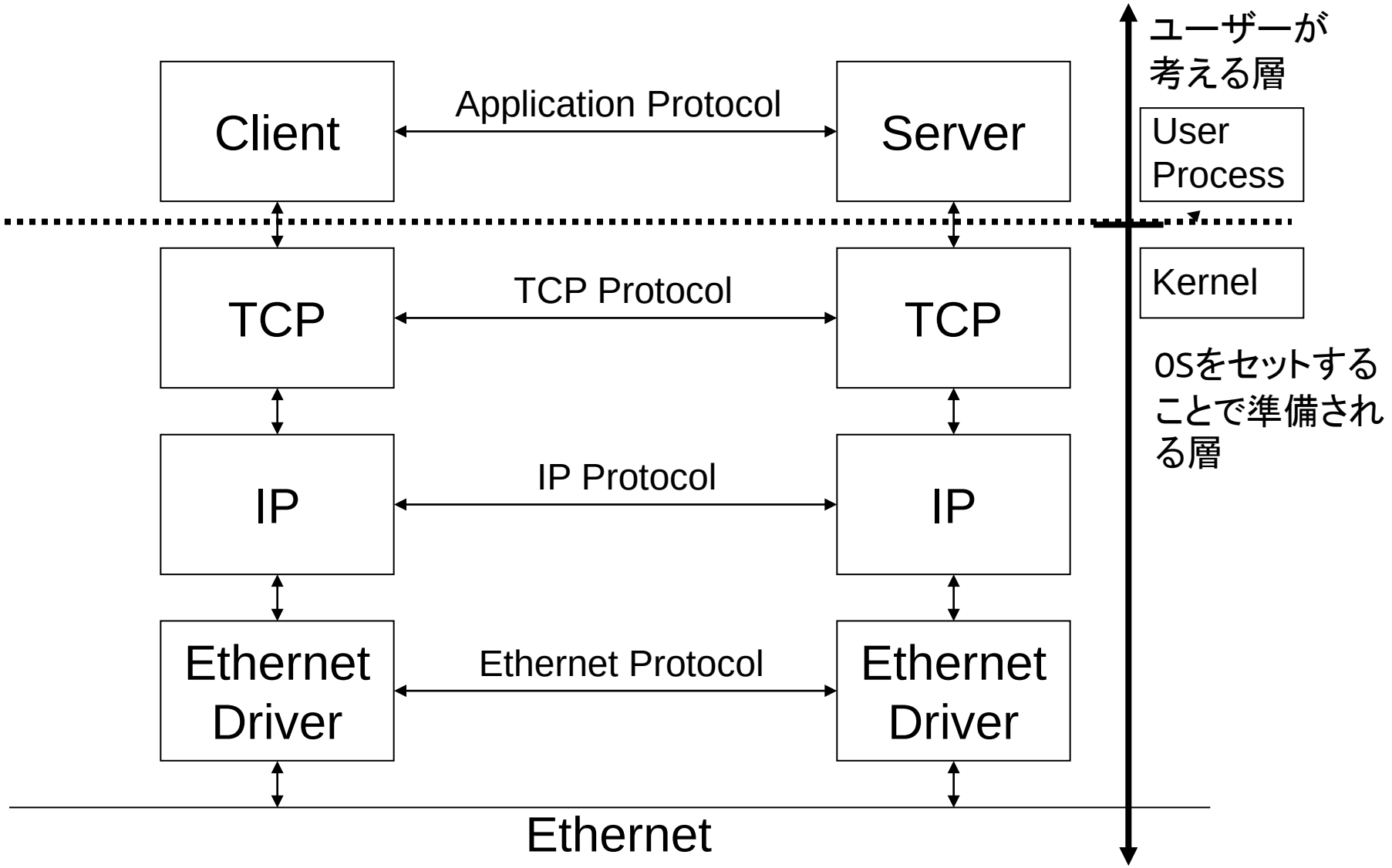
ネットワークアプリケーション



クライアント	サーバー
Webブラウザ	Webサーバー
メール読み書き	メールサーバー
DAQ　読み出しソフトウェア	リードアウトモジュール

Unix Network Programming p. 3

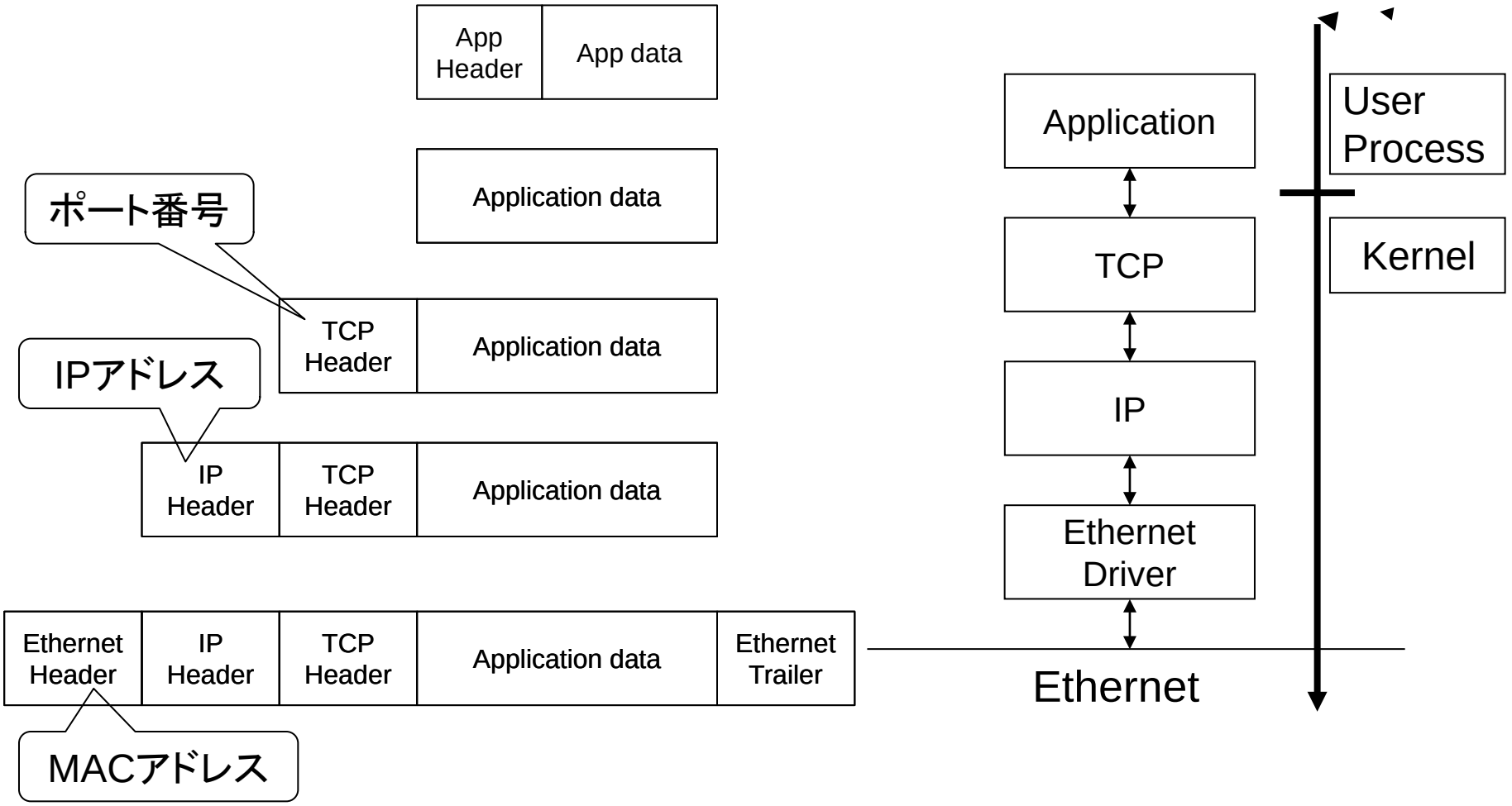
Ethernet Using TCP



出典：Unix Network Programming p. 4

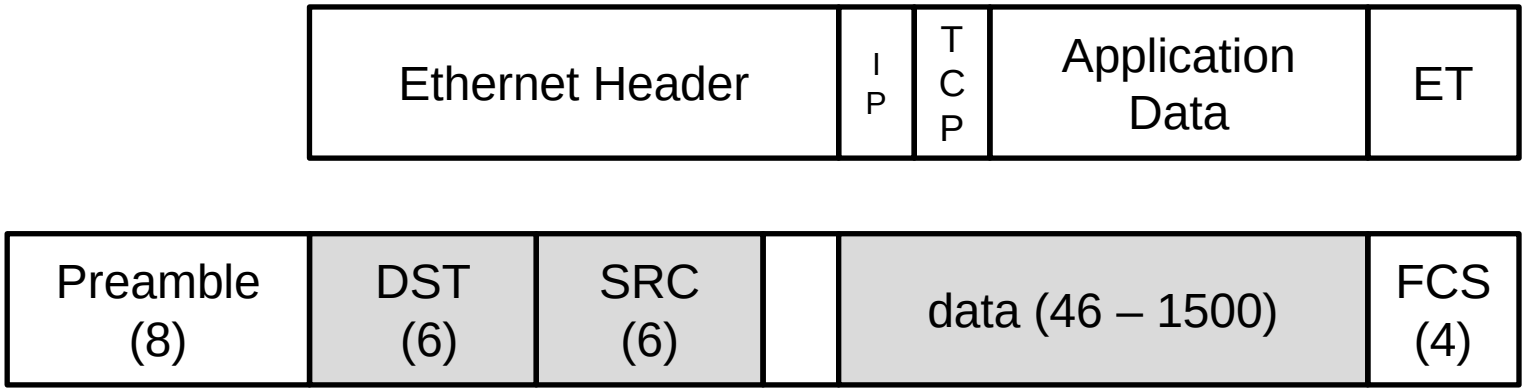
プロトコルスタック縦断

出典:TCP/IP Illustrated Vol. 1 (1st Edition) p. 10



ポート番号、IPアドレス、MACアドレスは始点、および終点

Ethernet Header/Trailer

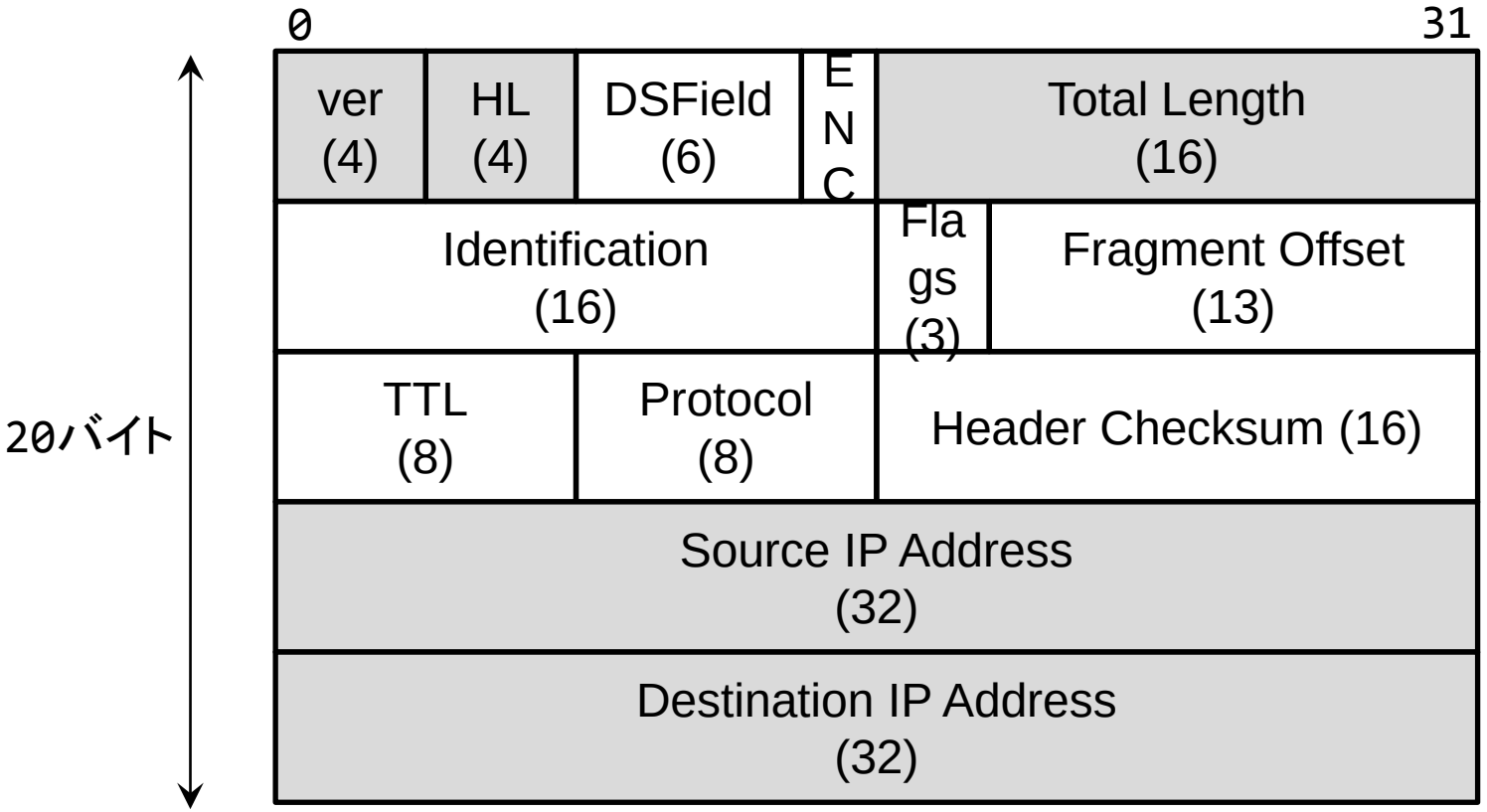



```
Type 0x8000 IPv4
     0x8006 ARP
     0x8808 Pause
     0x86DD IPv6
     0x6003 DECnet Phase IV
```

Ethernet

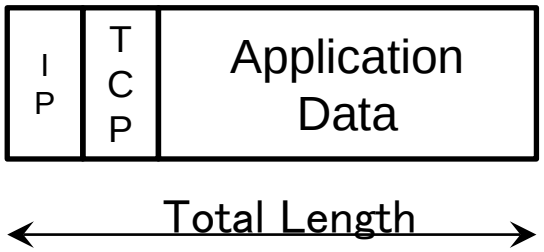
- 長さ制限
  - 1000BaseT イーサネットスイッチ 100m
  - スイッチ多段も制限がある
- Ethernet以外の物理層も使える必要がある
  - MACアドレス以外のアドレスが必要

IP Header



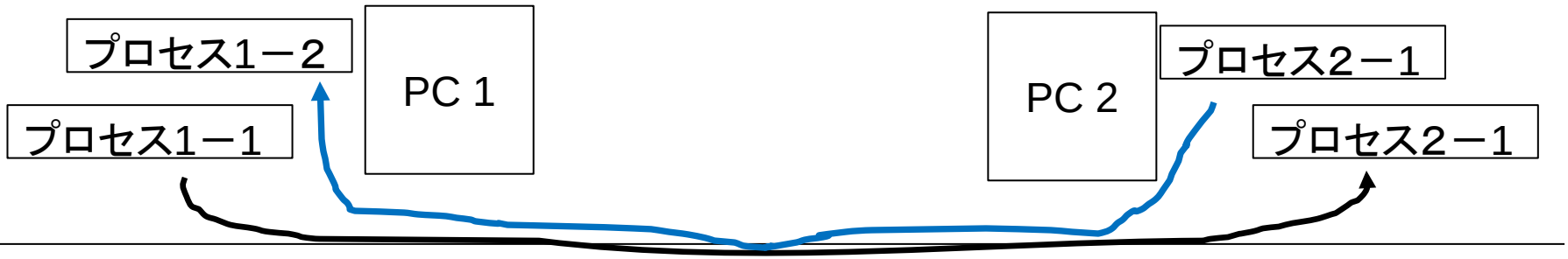
HL: Header Length (単位: 32bit word (4bytes))
Total Length: ヘッダを含む (単位： バイト)

```
% ping 0x7f000001 (IP Address: 32 bit)
PING 0x7f000001 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.053 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.097 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.095 ms
```



IP

- 遠くまで届くようになった
- PCで動くネットワークプログラムは1個ではない
- 届いたパケットを仕分けるしくみが必要
- 相手方どのプロセスに届けるのか仕分ける仕組みが必要

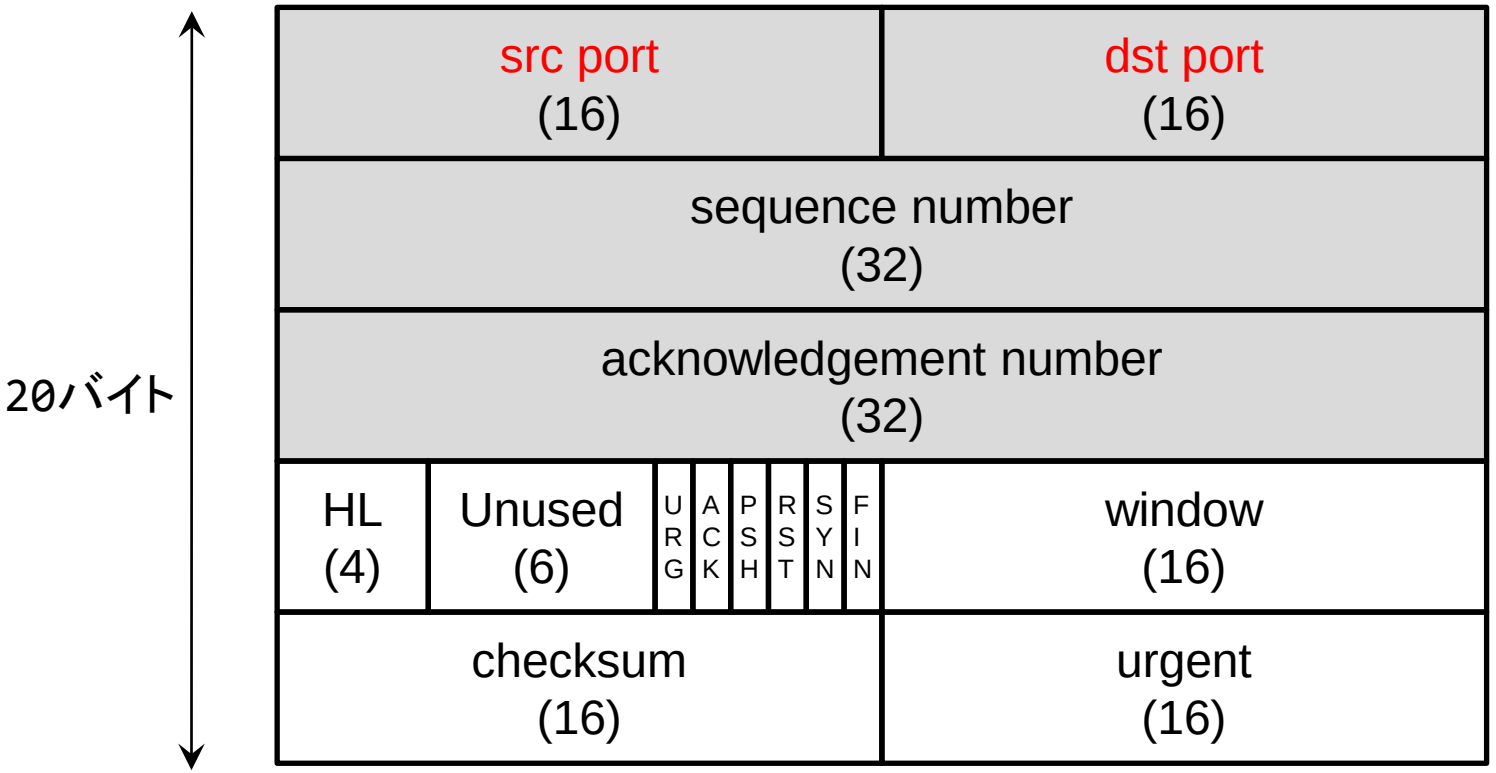


UDP Header

src port (16)	dst port (16)
length (16)	checksum (16)

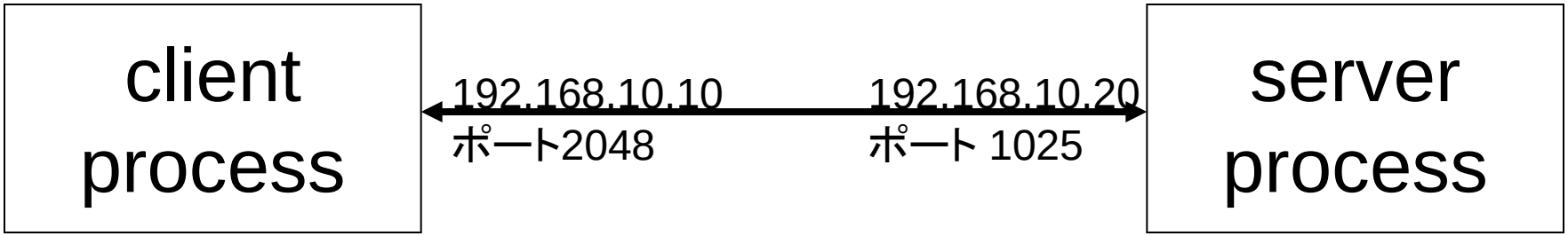
length = udp header length + data length (単位: バイト)
IPヘッダにレングスがあるから本来は不要

TCP Header



HL: Header Length (単位: 32ビットワード (4バイト))

IP Address、Port

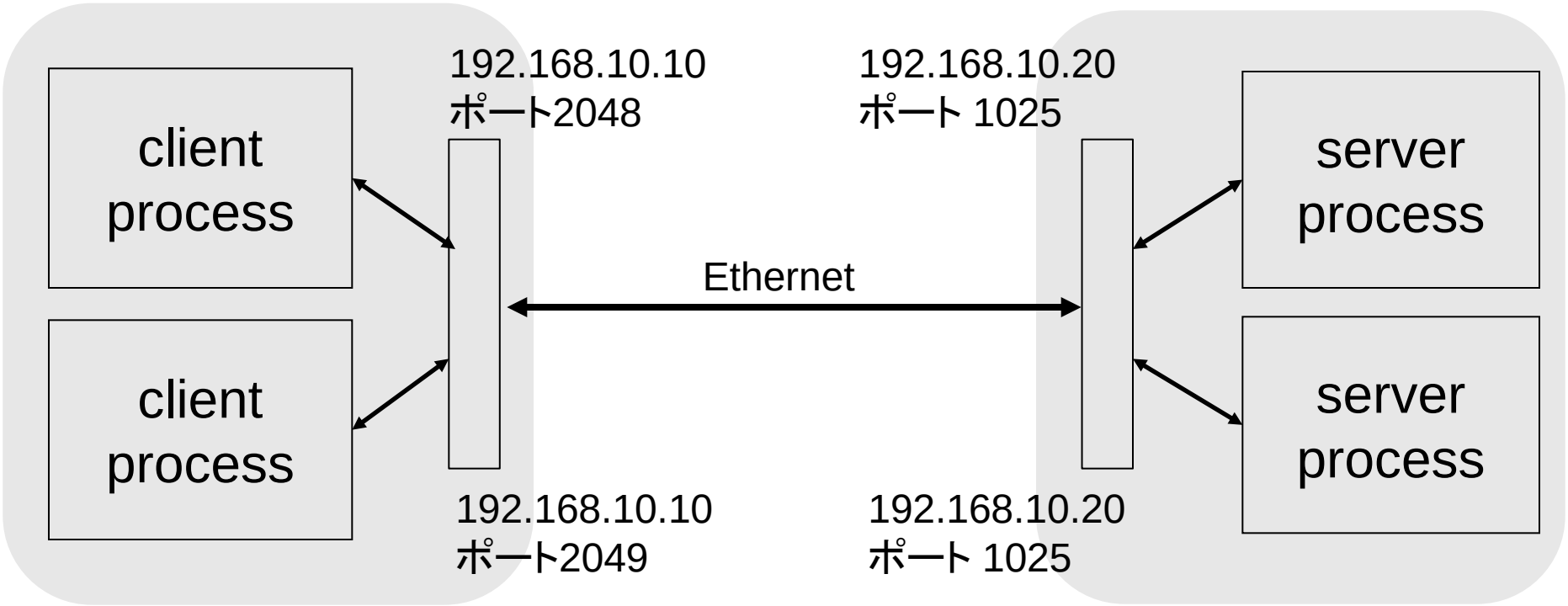


IPアドレス: IP層
ポート: TCP/UDP層

通信の相手方を指定、認識
client側 (192.168.10.10, 24, 192.168.10.20, 1025)
server側 (192.168.10.20, 1025, 192.168.10.10, 24)

Unix Network Programming p. 3

IP Address、Port、Multiplexing, DeMultiplexing

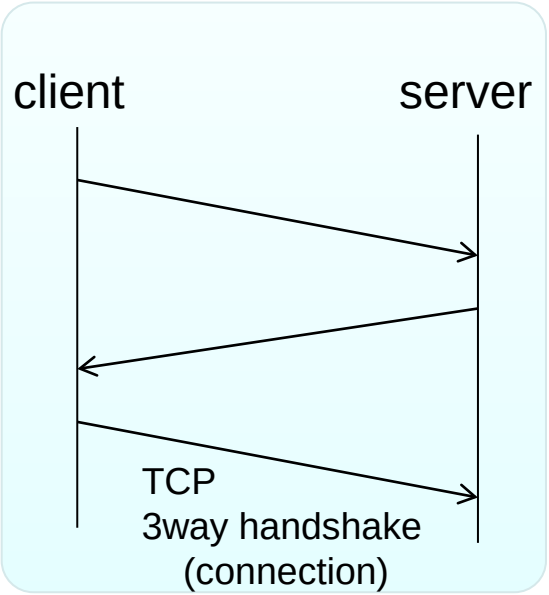


通信の相手方を指定、認識

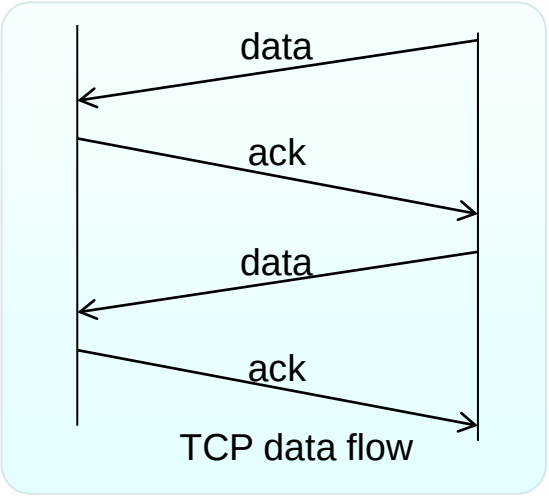
```
client (192.168.10.10, 2048, 192.168.10.20, 1025)
server (192.168.10.20, 1025, 192.168.10.10, 2048)

client (192.168.10.10, 2049, 192.168.10.20, 1025)
server (192.168.10.20, 1025, 192.168.10.10, 2049)
```

TCPとUDP

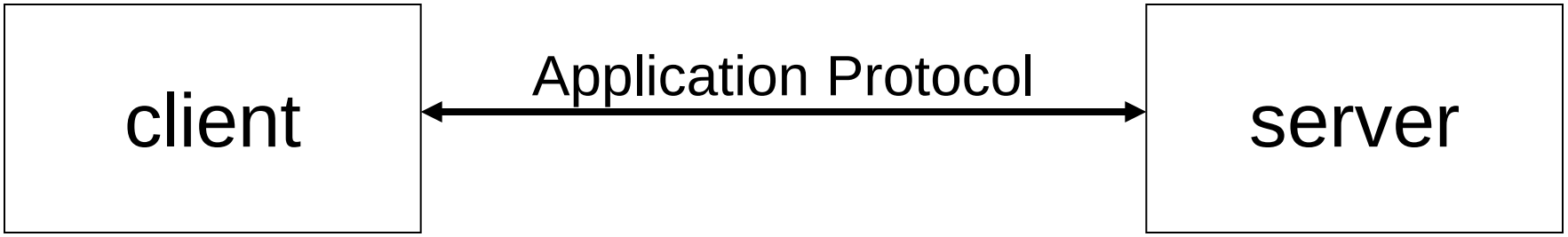
- TCP (Transmission Control Protorol)
  - コネクション型通信(まず最初に接続を確立)
  - データが届いたか確認しながら通信する
    - 届いていなければ再送する
  - 実験ではデータ転送に使う



- UDP (User Datagram Protocol)
  - コネクションレス型通信(接続を確立せずデータを送る)
  - データが届いたかどうかの確認が必要ならそれはユーザーが行う
  - 実験ではスローコントロールに使う(データ転送に使う場合もある)。



Network Application: Client - Server



ネットワークを通じて通信するプログラムを書くにはまずクライアントおよびサーバー間の通信プロトコルを策定する必要がある。

Unix Network Programming p. 3

通信プロトコルの例

- TCP
  - SMTP（メール）
  - HTTP（ウェブ）
  - 実験データ転送
- UDP
  - DNS (ホスト名からIPアドレスへ変換など)
    - TCPへのフォールバックあり
  - NTP (時刻サーバーと時刻の同期)
  - リードアウトモジュールスローコントロール

nc (netcat)

- nc - arbitrary TCP and UDP connections and listens
  - 標準入力をネットワークへ (標準入力は通常はキーボード)
  - ネットワークからのデータを標準出力へ (標準出力は通常は画面)

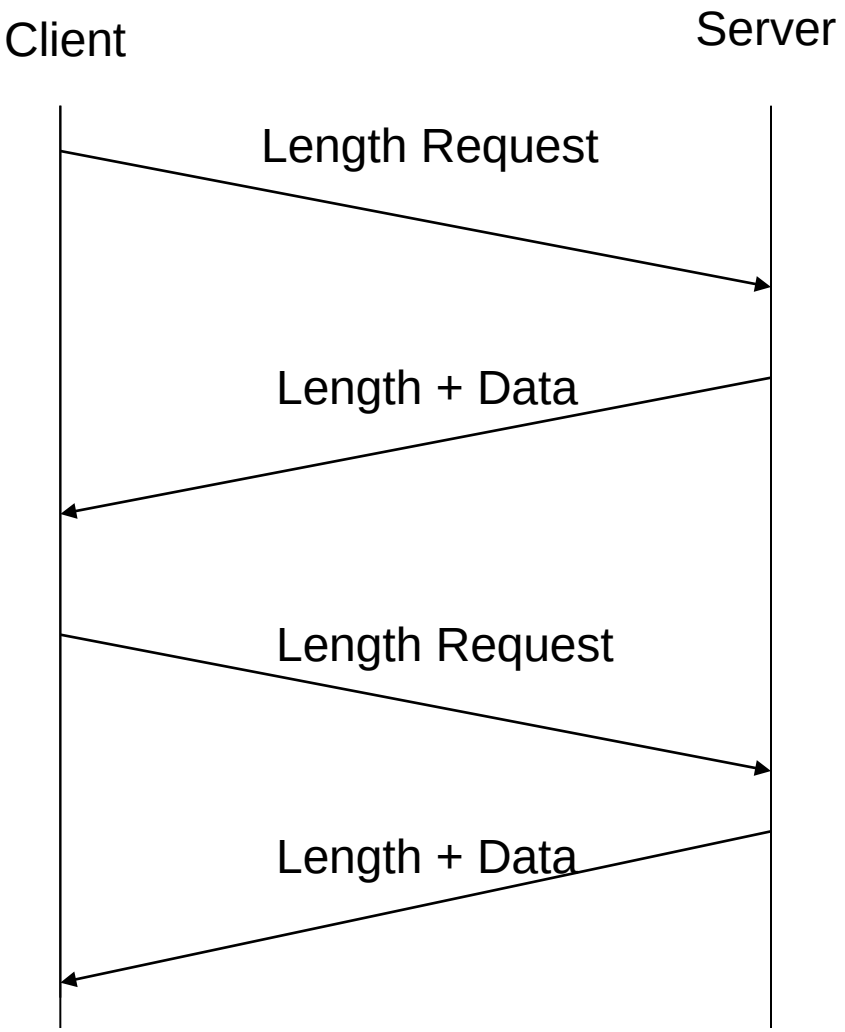
```
% nc   -l   1234
(これで待機して別の端末から)

Hello, worldと表示される
```

```
% nc   127.0.0.1   1234
Hello, worldと入力、エンターキー
```

- リードアウトモジュール 192.168.10.16、ポート24からTCPでデータがくるとして
  - nc 192.168.10.16 24 > datafile (標準出力をファイルへリダイレクト)
  - nc 192.168.10.16 24 | prog_histo (パイプでプログラムへ)
  - nc 192.168.10.16 24 | tee datafile | prog_histo
  - "|" のバッファサイズ 64kbytes

通信プロトコルの例：実験システム

- 無手順（データをどんどん送る）
- ポーリングで読み取り

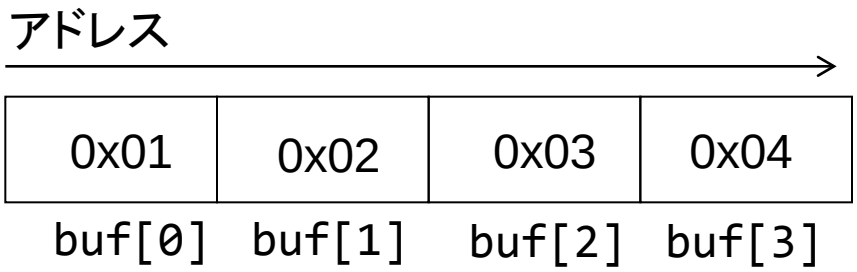


ネットワークバイトオーダー (1)

- unsigned char buf[10];
  アドレスはbuf[0], buf[1], buf[2]の順に大きくなる
- unsigned char buf[10];
  write(sockfd, buf, 10);
  とするとbuf[0], buf[1], buf[2] …の順に送られる。
- read(sockfd, buf, 10);
  きた順にbuf[0], buf[1], buf[2]に格納される。

ネットワークバイトオーダー(2)

0x 01 02 03 04 の順に送られてきたデータをread(sockfd, buf, 4)で読んだ場合



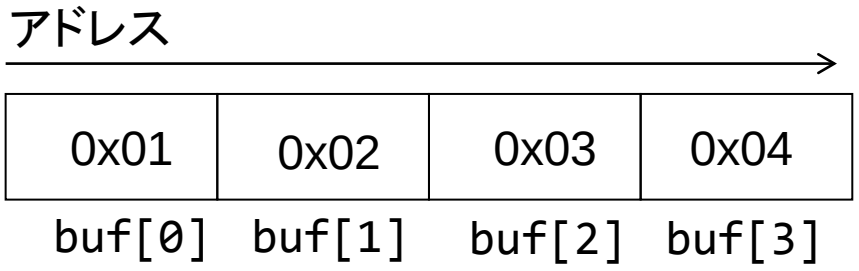
intとしての解釈

big endian　0x 01020304 = 16909060

little endian　0x 04030201 = 67305985

ネットワークバイトオーダーはbig endian

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned char buf[4];
    unsigned int *int_p;
    unsigned int i;

    buf[0] = 0x01;
    buf[1] = 0x02;
    buf[2] = 0x03;
    buf[3] = 0x04;

    int_p = (unsigned int *) &buf[0];
    i = *int_p;
    printf("%d¥n", i);

    return 0;
}
```



実行結果: 67305985

ネットワークバイトオーダー (3)

```
// intがどういう順番でメモリーに
// 入っているか調べるプログラム

#include <stdio.h>

int main(int argc, char *argv[])
{
        int i;

        union num_tag {
                unsigned char c[sizeof(int)];
                unsigned int    num;
        } u_num;

        u_num.num = 0x01020304;

        for (i = 0; i < sizeof(int); i++) {
                printf("u_num.c[%d]: %p 0x%02x ¥n", i, &u_num.c[i], u_num.c[i]);
        }
        return 0;
}
```

出力 (i386)

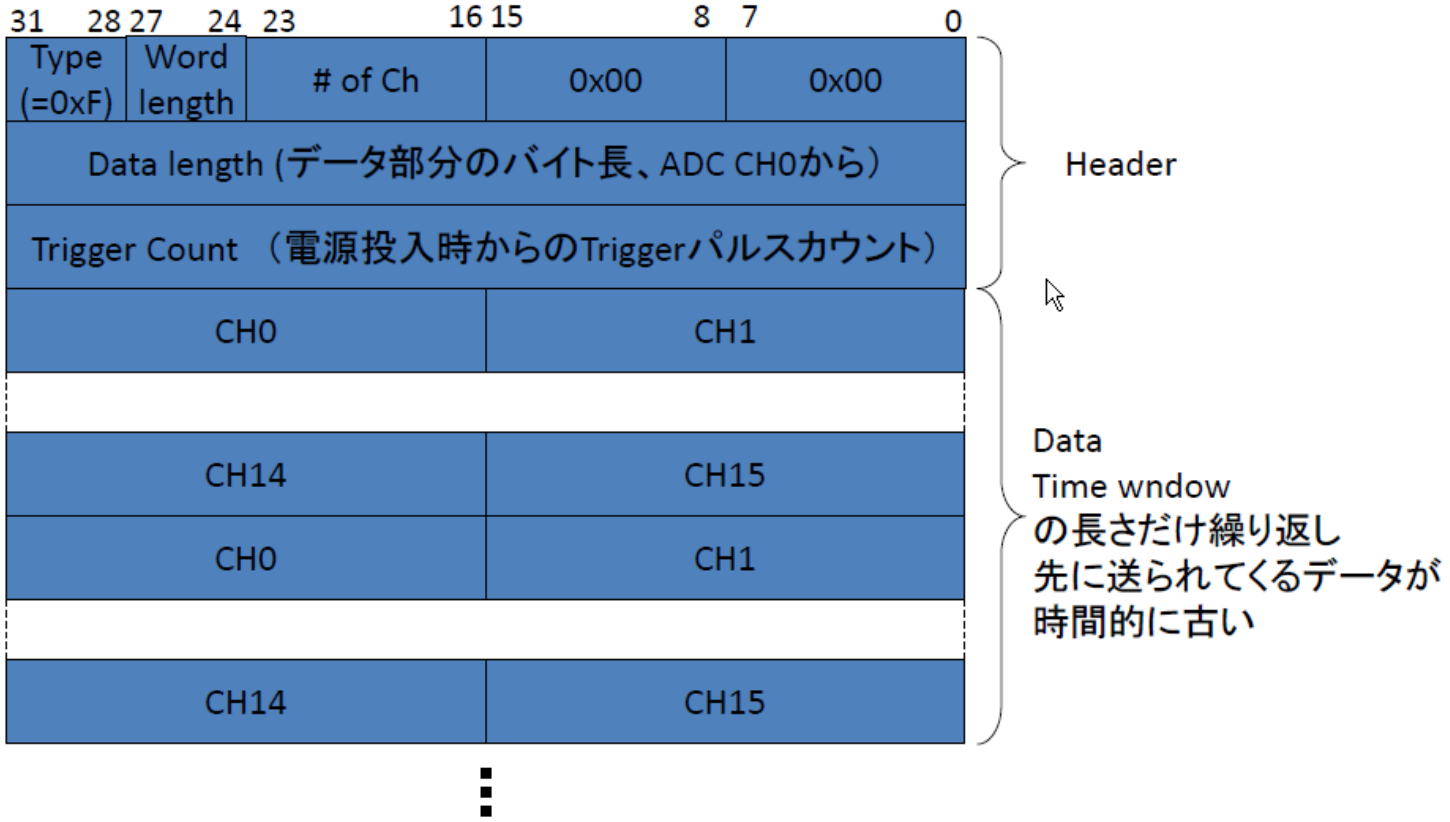
```
u_num.c[0]: 0xbfbfe850 0x04
u_num.c[1]: 0xbfbfe851 0x03
u_num.c[2]: 0xbfbfe852 0x02
u_num.c[3]: 0xbfbfe853 0x01
```

出力 (arm)

```
u_num.c[0]: 0xbe8d76c4 0x04
u_num.c[1]: 0xbe8d76c5 0x03
u_num.c[2]: 0xbe8d76c6 0x02
u_num.c[3]: 0xbe8d76c7 0x01
```

ネットワークバイトオーダー(4)

- ホストオーダー⇔ネットワークバイトオーダー変換関数
  - htonl (host to network long)
  - htons (host to network short)
  - ntohl (network to host long)
  - ntohs (network to host short)



Data length (複数バイト)を取り出すことを考える。
まずバイトオーダーを仕様などで確認する。ネットワークバイトオーダーだった場合

```
unsigned char header_buf[12];
int *int_p;
int data_length;
// header_buf にヘッダデータをいれる。いれたあとdata lengthを取り出す処理:
int_p = (int *) &header_buf[4];
data_length = *int_p;
data_length = ntohl(data_length);
```

もくじ

- 前提知識
  - TCP/IP (IPアドレス、ポート、TCP)
  - アプリケーションプロトコル
  - ネットワークバイトオーダー
- TCPでデータを読むまでに使う関数
  - socket(), connect(), read()/write()
- プログラムを書くときの情報のありか、エラー処理
  - マニュアルページの読み方
  - エラー捕捉法、メッセージの表示
- 実際にネットワークを使って読むときの注意
  - ソケットレシーブバッファ

ネットワークの読み書き

- ファイルの読み

```
FILE *fp = fopen("filename", "r");
n = fread(buf, (size_t) 1, sizeof(buf), fp);
```

- ネットワークの場合

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
connect(sockfd, &remote_addr, sizeof(remote_addr));
n = read(sockfd, buf, sizeof(buf));
```

socket()

```
int sockfd;

sockfd = socket(AF_INET, SOCK_STREAM, 0); /* TCP */
sockfd = socket(AF_INET, SOCK_DGRAM,  0); /* UDP */
```

- ソケットを作る
- まだどこにも接続していない
- TCP, UDPの指定をする

connect()

```
struct sockaddr remote_addr;

connect(sockfd, &remote_addr, remote_addr_len);
```

- 通信相手に接続する
  - 接続に必要な情報: IPアドレス、ポート番号
  - IPアドレス、ポートはsockaddr構造体を使って指定する（これが一番大変）
  - sockaddr構造体へIPアドレス、ポート情報を割り当てる方法
    - 構造体メンバーへ値を代入
    - getaddrinfo()の利用

connect()

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

int    connect ( int    sockfd, const    struct sockaddr *serv_addr,   socklen_t addrlen);
```

struct sockaddr: 総称ソケットアドレス構造体
アドレス、ポートの情報を格納する構造体
（IPv4, IPv6その他のアドレス体系でも使えるように）

connect()では通信相手を指定するためにsockaddrを使用する。

connect()

```
#include   <netinet/in.h>
struct sockaddr_in {
    sa_family_t      sin_family;          /* AF_INET */
    in_port_t        sin_port;            /* 16 bit TCP or UDP port number
    struct  in_addr  sin_addr;            /* 32 bit IPv4 address */
    char             sin_zero[8]          /* unused */
};
struct  in_addr {
    in_addr_t   s_addr;
};
```

Example:

```
struct   sockaddr_in   servaddr;
char    *ip_address    = "192.168.0.16";
int      port          = 13;                  /* daytime */
servaddr . sin_family = AF_INET;
servaddr . sin_port    = htons(port);
inet_pton(AF_INET,  ip_address,  &servaddr.sin_addr); /* need error check */
```

socket() + connect()

```
struct sockaddr_in servaddr;
int     sockfd;
char    *ip_address = "192.168.0.16";
int     port = 13;  /* daytime */

if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    peror("socket");
    exit(1);
}

servaddr.sin_family = AF_INET;
servaddr.sin_port   = htons(port);
if (inet_pton(AF_INET, ip_address, &servaddr.sin_addr) <=0) {
    fprintf(stderr, "inet_pton error for %s¥n", ip_address);
    exit(1);
}

if (connect(sockfd, (struct sockaddr *) &servaddr,
        sizeof(servaddr)) < 0) {
    perror("connect");
    exit(1);
}
```

長過ぎるので普通はなにかしたいところ

getaddrinfo()

```
char *host      = "192.168.10.16";
char *port_name = "1234";

int r;
struct addrinfo hint, *result;

memset(&hint, 0, sizeof(hint)); /* 構造体変数の初期化 */

hint.ai_family   = AF_INET;     /* IPv4 */
hint.ai_socktype = SOCK_STREAM; /* TCP  */

r = getaddrinfo(host, port_name, &hint, &result);

connect(sockfd, result->ai_addr, result->ai_addrlen);

freeaddrinfo(result);
```

注：エラー処理をしていないので
このままではだめです

connect_tcp()

```
if (   (sockfd = connect_tcp(ip_address, port)) < 0) {
    fprintf("connect error");
    exit(1);
}
```

と書けるようにまとめておくと使いまわしがきく（かもしれない）。

DAQ-Middleware Sockライブラリでは

```
try {
        // Create socket and connect to data server.
        m_sock = new DAQMW::Sock();
        m_sock->connect(m_srcAddr, m_srcPort);
} catch (DAQMW::SockException& e) {
        std::cerr << "Sock Fatal Error : " << e.what() << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
} catch (...) {
        std::cerr << "Sock Fatal Error : Unknown" << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
}
```

read()

```
unsigend char buf[64];
int n;

n = read(sockfd, buf, read_bytes);
```

- 読むデータがない場合には読めるまで待つ
- 読めたバイト数が返る

  n = 0: EOF

  n < 0: エラー
- ネットワークの場合 : read_bytesで指定したバイト数読めるとはかぎらない (最大値はread_bytes バイト)
  - 指定した分だけまだデータが来ていないなど
- 指定したバイト数必ず読みたい場合はそのようにプログラムする必要がある。

```
/* Read "n" bytes from a descriptor. */
ssize_t readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;
    ssize_t nread;
    char    *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;      /* and call read() again */
            else
                return(-1);
        } else if (nread == 0)
            break;              /* EOF */

        nleft -= nread;
        ptr   += nread;
    }
    return(n - nleft);      /* return >= 0 */
}
```

write()

```
unsigned char buf[64];
int n;

n = write(sockfd, buf, write_bytes);
```

- 書けたバイト数が返る
- エラーの場合は -1 が返る

もくじ

- 前提知識
  - TCP/IP (IPアドレス、ポート、TCP)
  - アプリケーションプロトコル
  - ネットワークバイトオーダー
- TCPでデータを読むまでに使う関数
  - socket(), connect(), read()/write()
- プログラムを書くときの情報のありか、エラー処理
  - マニュアルページの読み方
  - エラー捕捉法、メッセージの表示
- 実際にネットワークを使って読むときの注意
  - ソケットレシーブバッファ

```
/* sample.c */
int main(int argc, char *argv[])
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);

    return 0;
}

% make sample
cc     sample.c   -o sample
sample.c: In function 'main':
sample.c:4: error: 'AF_INET' undeclared (first use in this function)
sample.c:4: error: (Each undeclared identifier is reported only once
sample.c:4: error: for each function it appears in.)
sample.c:4: error: 'SOCK_STREAM' undeclared (first use in this function)
make: *** [sample] Error 1
```

- インクルードファイルが足りない。どのファイルをインクルードすればよいのか？
- エラーチェックがない。でもどういうエラーが返ってくるのか？

プログラムを書く際の情報のありか

- Manual Page (man コマンド)

```
% man read
BASH_BUILTINS(1)                                          BASH_BUILTINS(1)

NAME
       bash,  :,  .,  [, alias, bg, bind, break, builtin, caller, cd, command,
       compgen, complete, compopt,  continue,  declare,  dirs,  disown,  echo,
       enable,  eval,  exec, exit, export, false, fc, fg, getopts, hash, help,
       history, jobs, kill, let, local, logout, mapfile, popd, printf,  pushd,
       pwd,  read, readonly, return, set, shift, shopt, source, suspend, test,
       times, trap, true, type, typeset, ulimit, umask, unalias, unset, wait -
       bash built-in commands, see bash(1)

BASH BUILTIN COMMANDS
```

/usr/share/man/man1/read.1.gz と /usr/share/man/man2/read.2.gzがある。上の内容は/usr/share/man/man1/read.1.gzのほう。

Manual Pages

- セクション
  - 1 (Utility Program)
  - 2 (System call)
  - 3 (Library)
  - 4 (Device)
  - 5 (File format)
  - 6 (Game)
  - 7 (Misc.)
  - 8 (Administration)
- セクションはman man するとでてくる。
  - read()の場合は man 2 read

ライブラリ関数/システムコール

- システムコール: カーネルが提供する機能
- ファイル、ネットワーク関連ではread(), write()など
- Cの関数としてよびだせる

- fopen(), fread(), fgets()などはライブラリ関数
  - 使いやすいように
  - バッファリング機能の提供
- manコマンドで出てくるファイルの先頭
  - SOCKET(2): 2: システムコール
  - FOPEN(3): 3: ライブラリ関数

Manual Pages

- Header

  READ(3P) POSIX Programmer's Manual READ(3P)

  READ(2) Linux Programmer's Manual READ(2)

- SYNOPSIS
- DESCRIPTION
- RETURN VALUE
- SEE ALSO
- EXAMPLE

% man socket

SOCKET(2) Linux Programmer's Manual SOCKET(2)

NAME

socket - create an endpoint for communication

SYNOPSIS

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

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

DESCRIPTION

socket() creates an endpoint for communication and returns a descriptor.

Manual Pages(例題)

READ(2) Linux Programmer's Manual READ(2)

NAME

read - read from a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION

read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

:

RETURN VALUE

:

ERRORS

:

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001.

NOTES

:

SEE ALSO

システムコールのエラーの捕捉（1）

- エラーの捕捉は必須
  - ユーザー（あるいは自分）が悩む
  - エラーが起きたときの処理は状況でかわるが、プログラムを停止させるのがよい場合が多い。
- 大部分のシステムコールはエラーだと -1 を返す
- 大域変数errnoにエラー原因の番号が設定される
  - #include <errno.h>
  - エラーが起きたときに設定される。エラーがおこる前は前のerrnoが残っている
- どんなエラーがあるかはマニュアルページのERRORSに書いてある。

man socketで出てくる例:

RETURN VALUE

On success, a file descriptor for the new socket is returned. On error, -1 is returned, and errno is set appropriately.

ERRORS

EACCES Permission to create a socket of the specified type and/or protocol is denied.

EAFNOSUPPORT
The implementation does not support the specified address family.

EINVAL Unknown protocol, or protocol family not available.

EMFILE Process file table overflow.

ENFILE The system limit on the total number of open files has been reached.

ENOBUFS or ENOMEM
Insufficient memory is available. The socket cannot be created until sufficient resources are freed.

EPROTONOSUPPORT
The protocol type or the specified protocol is not supported within this domain.

Other errors may be generated by the underlying protocol modules.

fopen() の例 (ライブラリ関数)

```
RETURN VALUE
Upon successful completion fopen(), fdopen() and
freopen() return a FILE pointer.  Otherwise, NULL is
returned and errno is set to indicate the error.
```

システムコールのエラーの捕捉(2)

- errnoは単なる数字で人間には意味がわかりにくい
- errnoから文字列へ変換する関数
  - perror()
  - err()

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0) {
    perror("socket error");
    exit(1);
}
```

エラー時にはperror()で指定した文字列 + ": " と、errnoに対応する文字列が表示される。

システムコールのエラーの捕捉(3)

```
#include <err.h>

err(eval, const char *fmt, . . .);
```

- 機能的には perror() + exit(eval)
- fmtにはprintf()で使うフォーマット指定子を使える
- 関数の最後が . . . なのは可変長関数であることを示す。例: printf("%d %d¥n", 10, 20);

```
if (connect(sockfd, result->ai_addr, result->ai_addrlen) < 0) {
    err(1, "connect for %s port %s", host, port_name);
}
```

エラーの場合は
sample: connect for localhost port 10: Connection refused
のように プログラム名 ： fmtで指定した文字列 ： errnoが示す失敗した理由を表示して、終了する。

TCPでconnectするまで (1)

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

#include <err.h>
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int usage(void)
{
    char *msg = "Usage: ./sample remote port";
    fprintf(stderr, "%s¥n", msg);

    return 0;
}
```

TCPでconnectするまで (2)

```
int main(int argc, char *argv[])
{
    char *host;
    char *port_name;
    int   r, sockfd;
    struct addrinfo hint, *result;

    /* program argument */
    if (argc != 3) {
        usage();
        exit(EXIT_FAILURE); /* EXIT_FAILURE == 1 in stdlib.h */
    }
    host      = argv[1];
    port_name = argv[2];
```

TCPでconnectするまで (3)

```
/* Create socket */
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0) {
       err(EXIT_FAILURE, "socket()");
   }

   /* Prepare addrinfo for IP address and port */
   memset(&hint, 0, sizeof(hint));
   hint.ai_family   = AF_INET;
   hint.ai_socktype = SOCK_STREAM;
   r = getaddrinfo(host, port_name, &hint, &result);
   if (r != 0) {
       fprintf(stderr, "getaddrinfo: %s¥n", gai_strerror(r));
       exit(EXIT_FAILURE);
   }
```

TCPでconnectするまで (4)

```
 /* Connect to remote host */
    if (connect(sockfd, result->ai_addr, result->ai_addrlen) < 0) {
        err(EXIT_FAILURE, "connect for %s port %s", host, port_name);
    }

    /* do read/write */

    return 0;
}
```

connect_tcp()

```
if ((sockfd = connect_tcp(ip_address, port)) < 0) {
    fprintf("connect error");
    exit(1);
}
```

と書けるようにまとめておくと使いまわしがきく（かもしれない）。

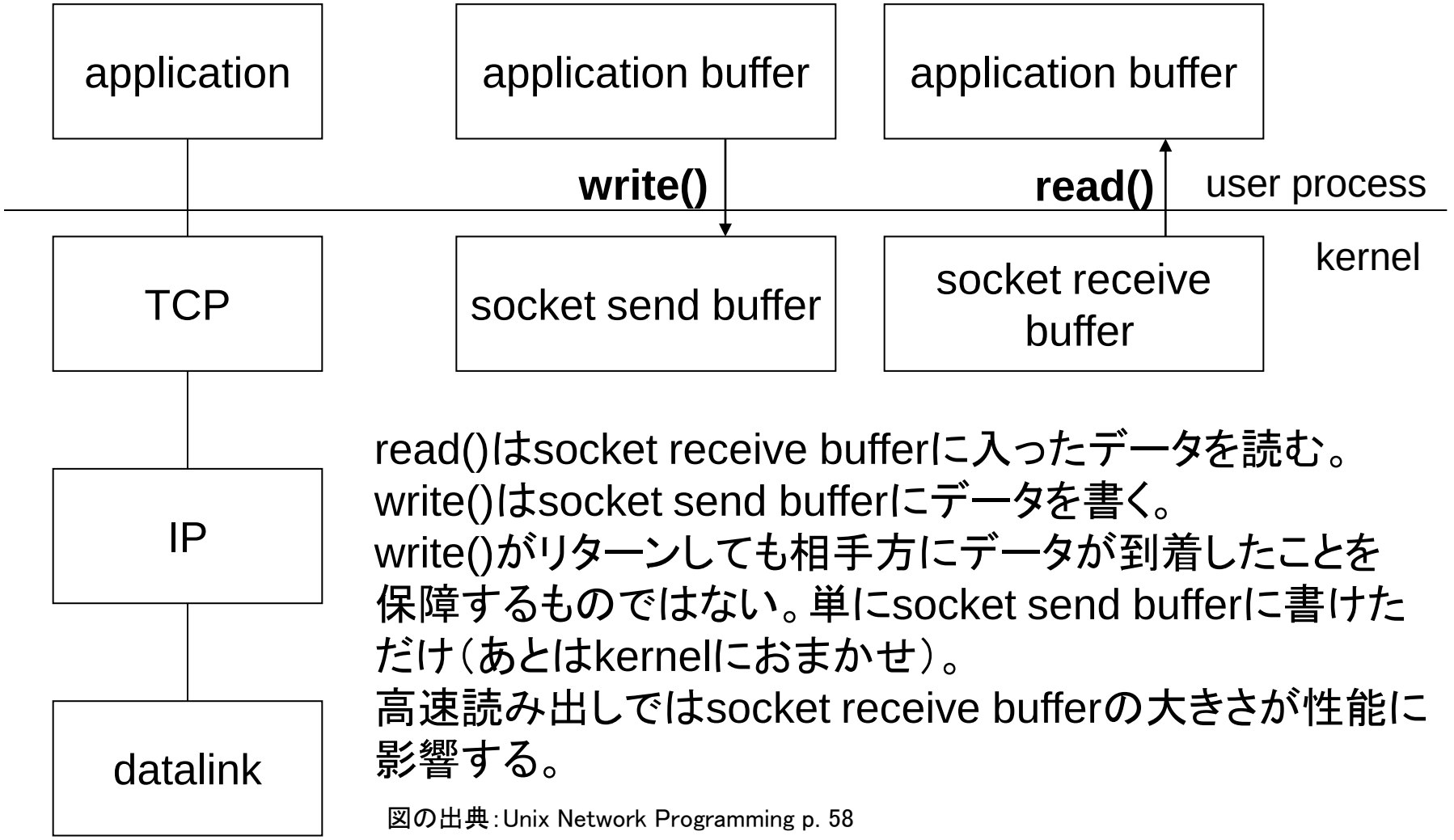
もくじ

- 前提知識
  - TCP/IP (IPアドレス、ポート、TCP)
  - アプリケーションプロトコル
  - ネットワークバイトオーダー
- TCPでデータを読むまでに使う関数
  - socket(), connect(), read()/write()
- プログラムを書くときの情報のありか、エラー処理
  - マニュアルページの読み方
  - エラー捕捉法、メッセージの表示
- 実際にネットワークを使って読むときの注意
  - ソケットレシーブバッファ

read()、write()

- ソケットファイルディスクリプタをread(), write()するとデータの受信、送信ができる。実際の動作は:
- read()
  - 通信相手方からのデータがソケットレシーブバッファに入っている。そのデータを読む。
- write()
  - ソケットセンドバッファにデータを書く。書いたデータが通信相手方に送られる。

TCP Input/Output



read()はsocket receive bufferに入ったデータを読む。
write()はsocket send bufferにデータを書く。
write()がリターンしても相手方にデータが到着したことを保障するものではない。単にsocket send bufferに書けただけ（あとはkernelにおまかせ）。
高速読み出しではsocket receive bufferの大きさが性能に影響する。

図の出典：Unix Network Programming p. 58

ソケットバッファに関する関数

- 現在のソケットバッファの大きさを取得する

```
int so_rcvbuf;
socklen_t len;

len = sizeof(so_rcvbuf);
/* レシーブバッファの大きさ*/
getsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &so_rcvbuf, &len);
/* センドバッファの大きさ */
getsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &so_rcvbuf, &len);
```

- レシーブバッファにあるデータバイト数

```
int nbytes;
nbytes = recv(sockfd,  buf,  sizeof(buf), MSG_PEEK|MSG_DONTWAIT);
```

あるいは

```
ioctl(sockfd, FIONREAD, &nbytes);
```

socket send/receive bufferの大きさの調整

- 受信に関してはLinuxでは自動調節機能がある

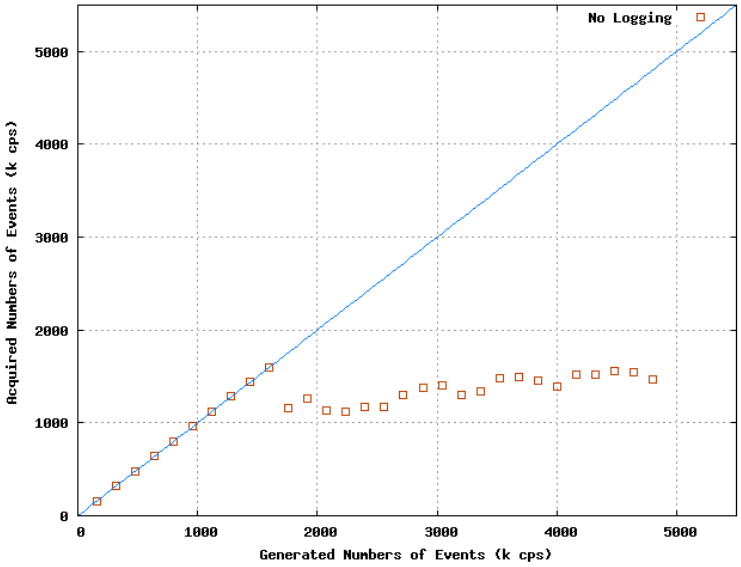
```
% cat /proc/sys/net/ipv4/tcp_rmem
4096    87380   4194304
最小値   初期値   最大値

# /etc/rc.local あたりに書いておく

so_rcvbuf_max=$((16*1024*1024)) # 16MB
so_sndbuf_max=$((16*1024*1024)) # 16MB

echo $so_rcvbuf_max > /proc/sys/net/core/wmem_max
echo $so_sndbuf_max > /proc/sys/net/core/rmem_max
read min init max              < /proc/sys/net/ipv4/tcp_rmem
echo $min $init $so_rcvbuf_max > /proc/sys/net/ipv4/tcp_rmem
read min init max              < /proc/sys/net/ipv4/tcp_wmem
echo $min $init $so_sndbuf_max > /proc/sys/net/ipv4/tcp_wmem
```

- setsockopt()を使うとプログラム内で設定できる。

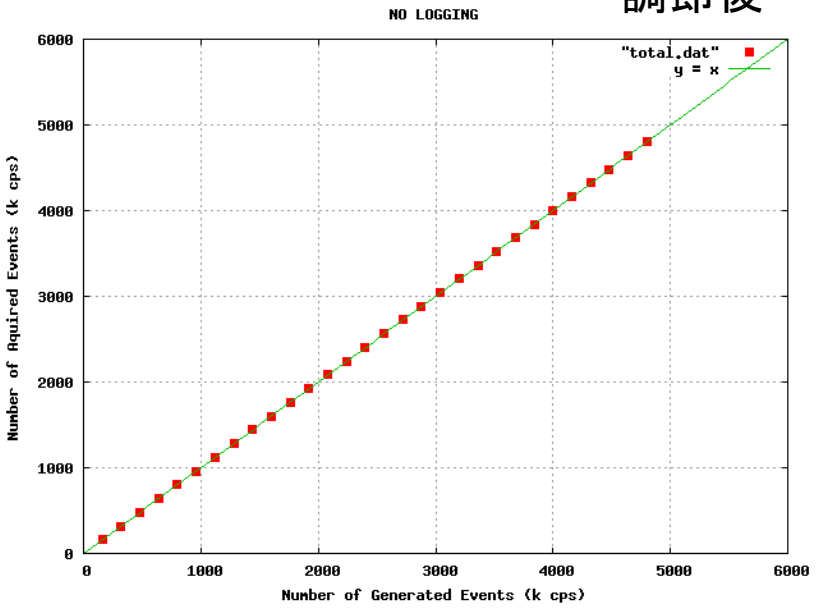
ソケットレシーブバッファの大きさ調節による改善例

多重読み出しを行うときにはデフォルト値を大きくしておかないと性能がでないことがある

調節前



調節後



多重読み出しで複数モジュールから読み出し
各モジュールは同一レートでデータを送ってくるようにセット
読むモジュール数を1, 2, 3, と増加させていった。

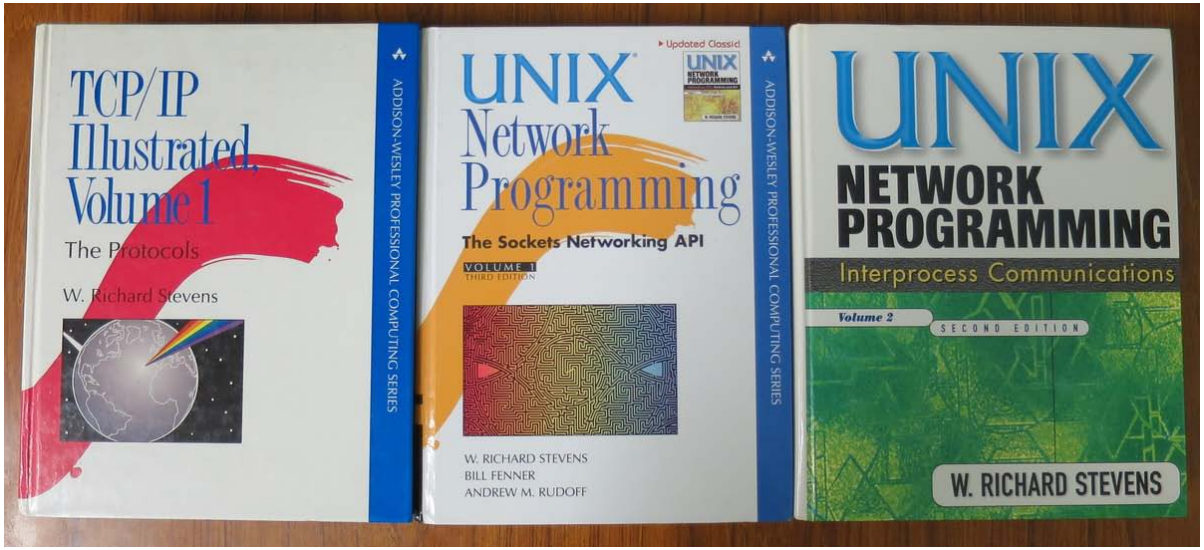
参考書 (軽量型)

TCP/IP ソケットプログラミングC言語編
Michael J. Donahoo, L. Calvert
小高知宏監訳
オーム社
ISBN4-274-06519-7

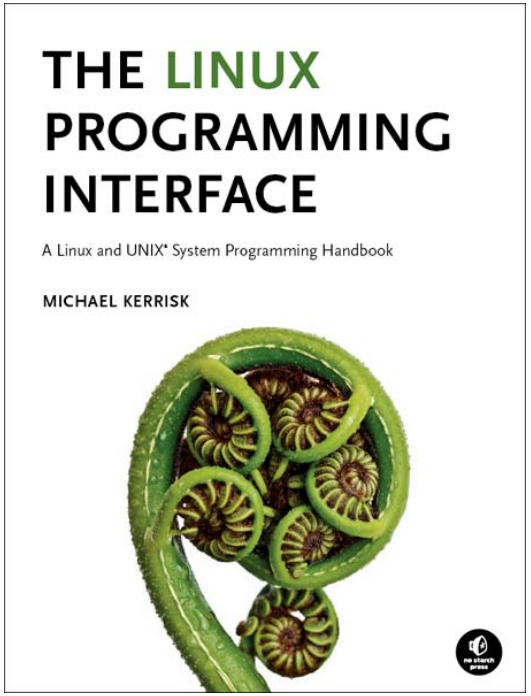
http://ssl.ohmsha.co.jp/cgi-bin/menu.cgi?ISBN=4-274-06519-7
38ページまで読めばクライアントが書けるようになる。
例題がそのままひとつのプログラムとして動かすことができる（説明のために断片化していない）

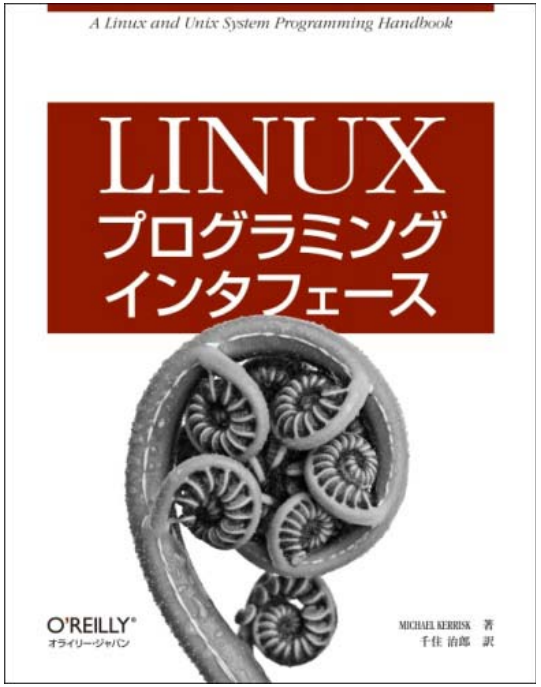
参考書 (本格的)

- Protocol
  - TCP/IP Illustrated, Volume 1　2nd edition (Fall, Stevens)
- Programming
  - Unix Network Programming Volume 1 (3rd edition) (Stevens, Fenner, Rudoff) (ソケット)
  - Unix Network Programming Volume 2 (2nd edition) (Stevens) (Inter Process Communications)



Linux System Programming





The Linux Programming Interface
Michael Kerrisk
No Starch Press
ISBN 978-1-59327-220-3
1552 pages
published in October 2010
http://man7.org/tlpi/

翻訳
Linuxプログラミングインターフェイス
Michael Kerrisk 著、千住 治郎 訳
ISBN978-4-87311-585-6
1604 ページ

システムコールプログラミングの話だけではなくたとえばシェアードライブラリの作り方およびsonameなどの話も書かれています。

まとめ

- 前提知識
  - TCP/IP (IPアドレス、ポート、TCP)
  - アプリケーションプロトコル
  - ネットワークバイトオーダー
- TCPでデータを読むまでに使う関数
  - socket(), connect(), read()/write()
- プログラムを書くときの情報のありか、エラー処理
  - マニュアルページの読み方
  - エラー捕捉法、メッセージの表示
- 実際にネットワークを使って読むときの注意
  - ソケットレシーブバッファ