

# 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

SiTCP は物理実験のフロントエンドをイーサネットで PC に接続する技術です。

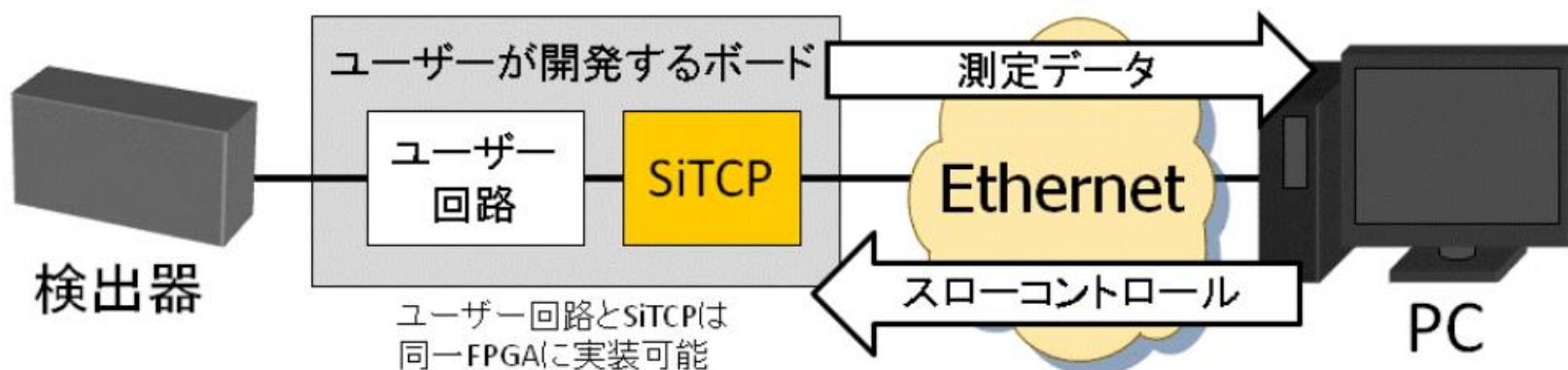


図 1 標準的な SiTCP 使用例

<https://www.sitcp.net/doc/SiTCP.pdf>

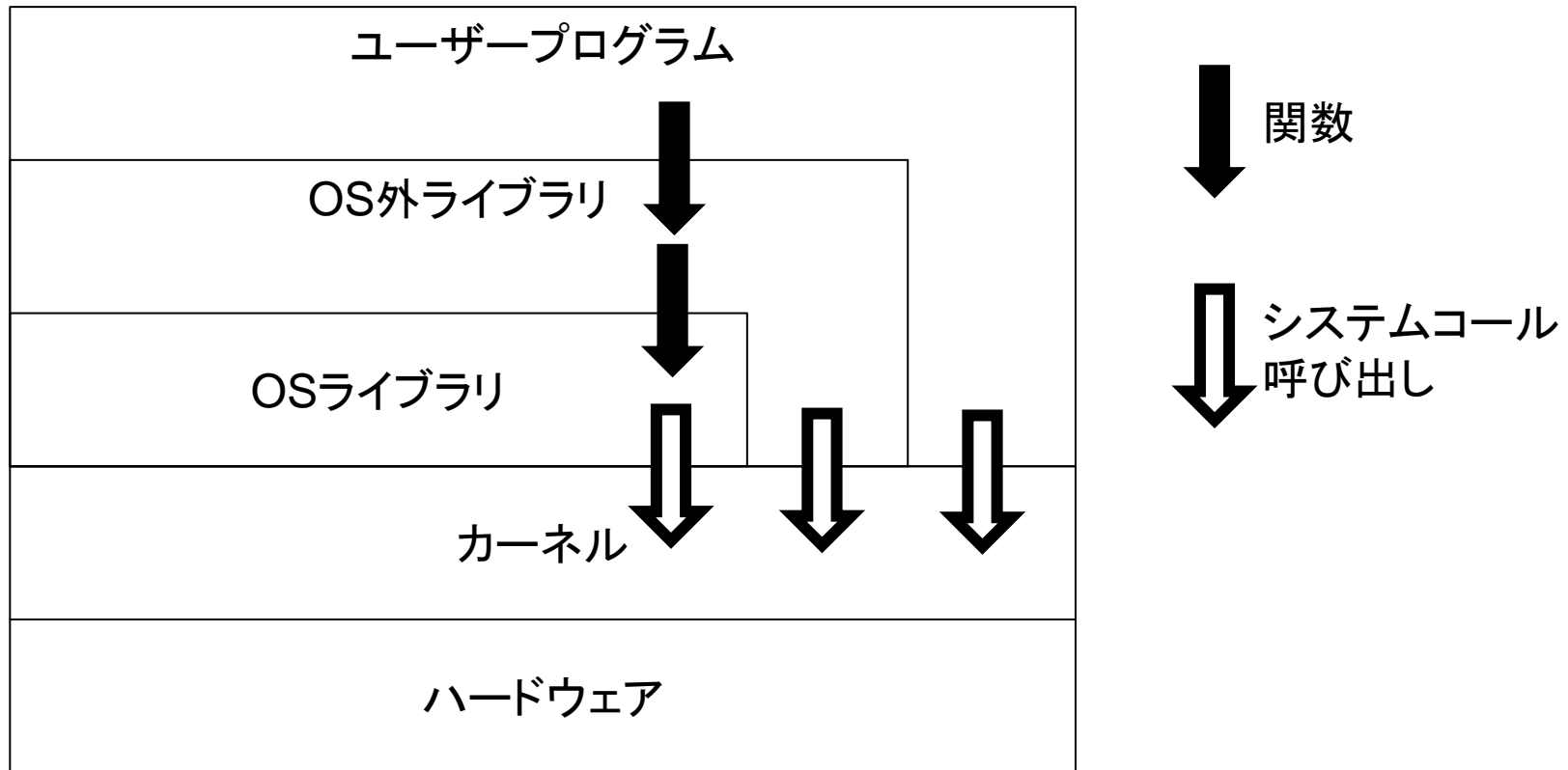
# もくじ

- 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

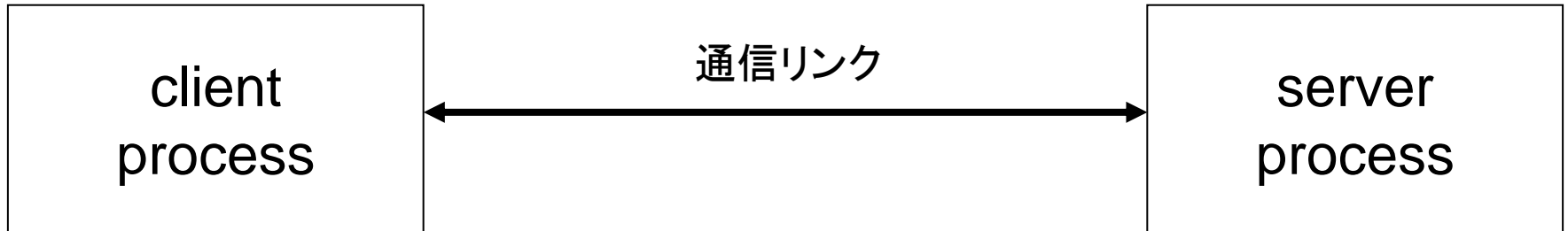
- 前提知識
  - TCP/IP (IPアドレス、ポート、TCP)
  - アプリケーションプロトコル
  - ネットワークバイトオーダー
- TCPでデータを読むまでに使う関数
  - socket(), connect(), read()/write()
- プログラムを書くときの情報のありか、エラー処理
  - マニュアルページの読み方
  - エラー捕捉法、メッセージの表示
- 実際にネットワークを使って読むときの注意
  - ソケットレシーブバッファ

# プログラム構成層



「試して理解 Linuxのしくみ (武内覚、技術評論社)」

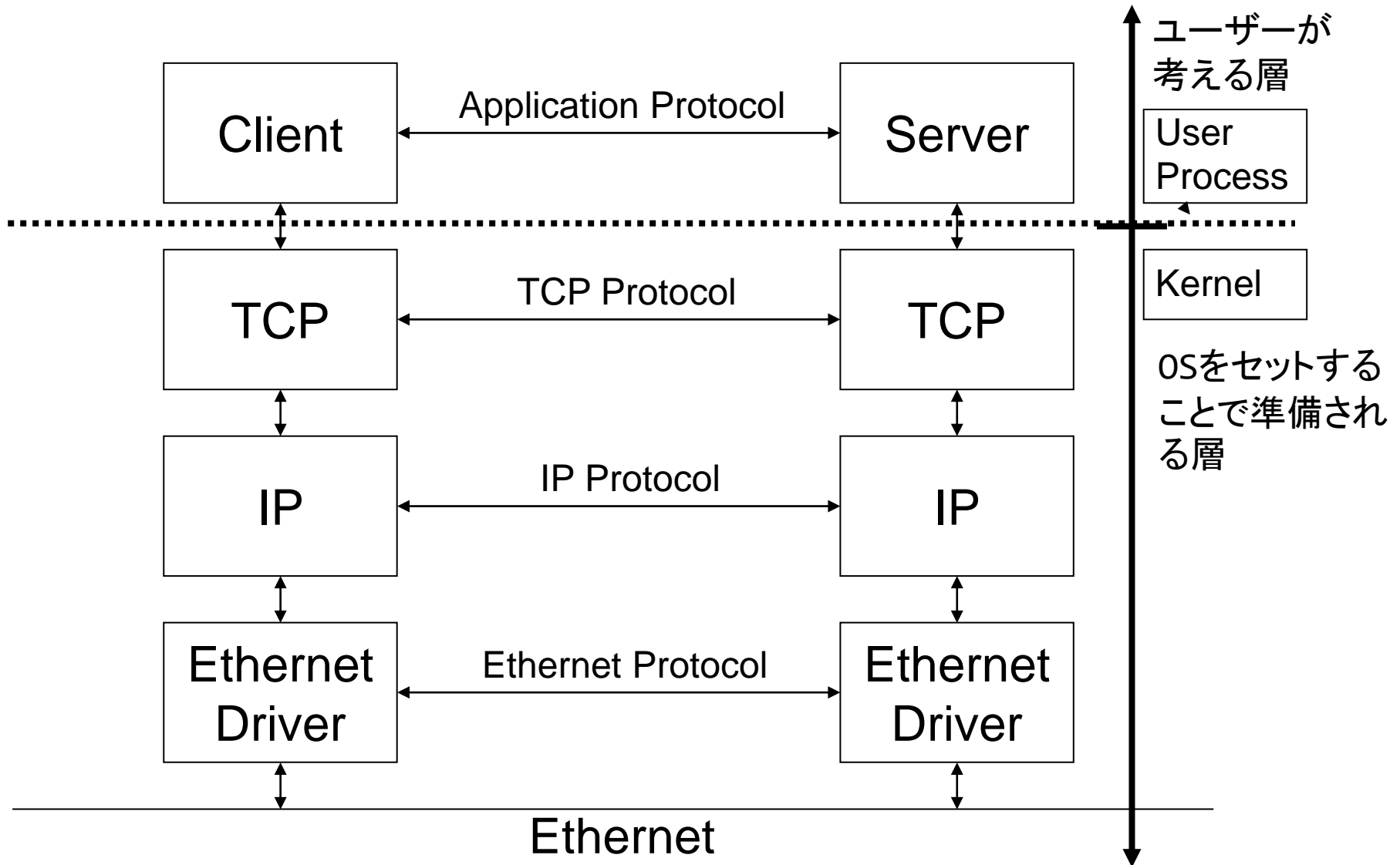
# ネットワークアプリケーション



クライアント	サーバー
Webブラウザ (IE, Edge, Firefox, Chrome など)	Webサーバー (apache, nginxなど)
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

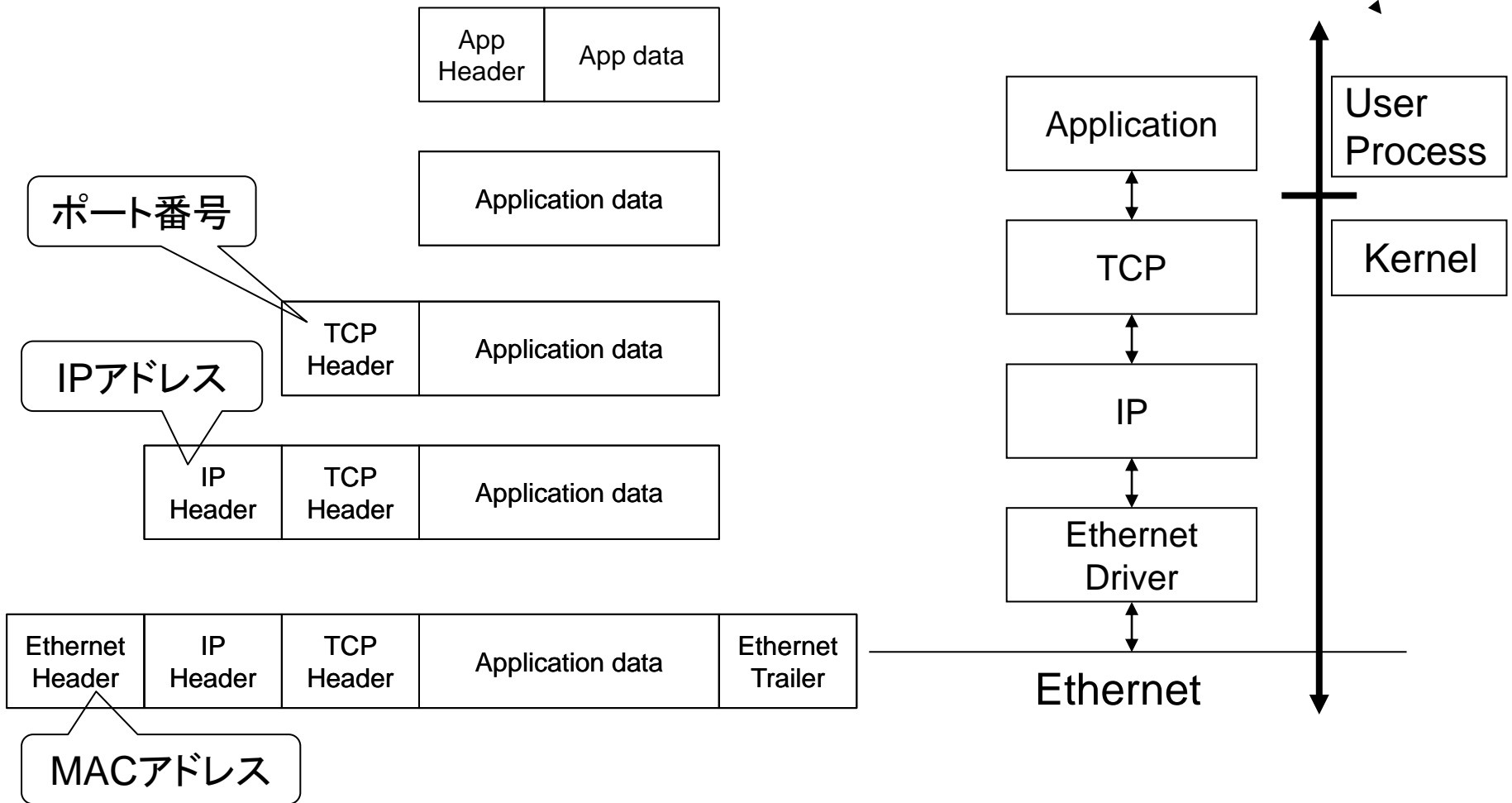
# Ethernet Using TCP



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

Preamble (8)	DST (6)	SRC (6)	Type (2)	data (46 - 1500 )	FCS (4)
-----------------	------------	------------	-------------	----------------------	------------

FCS:Frame Check Sequence

Type 0x8000 IPv4  
0x8006 ARP  
0x8808 Pause  
0x86DD IPv6

DST 送信先MACアドレス  
SRC 送信元MACアドレス

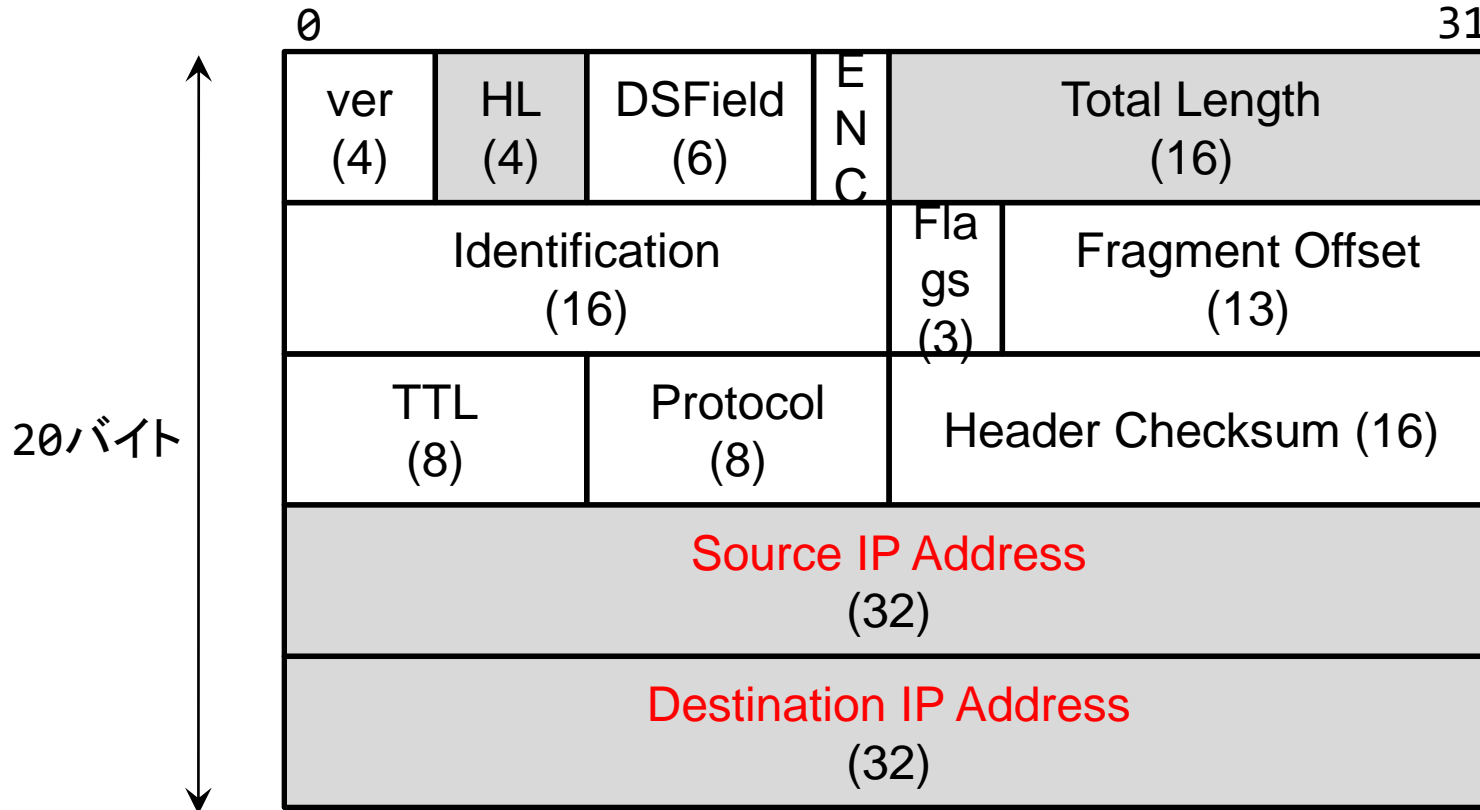
Type 0x8000 IPv4  
0x8006 ARP  
0x8808 Pause  
0x86DD IPv6

データに最小値(46)がある。最小値に足りないデータを送るときはパディングして46バイト以上にする。

# Ethernet

- 長さ制限
  - 1000BaseT イーサネットスイッチ ケーブル長 銅線では100m
  - 多段スイッチでの制限は仕様上はとくにはない
- 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

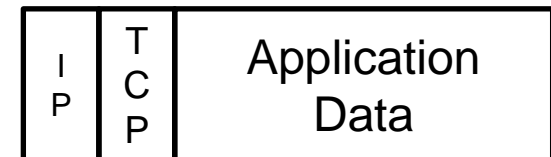


HL: Header Length (単位: 32bit word (4bytes))

Total Length: ヘッダを含む (単位: バイト)

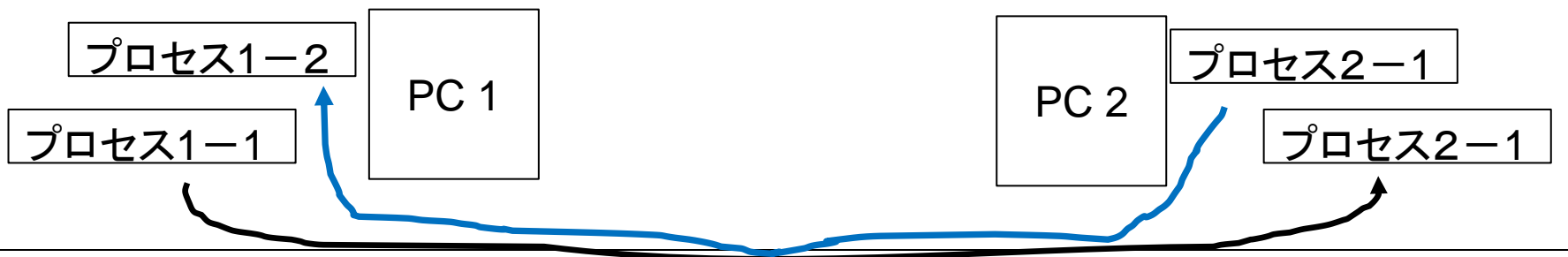
Source IP Address

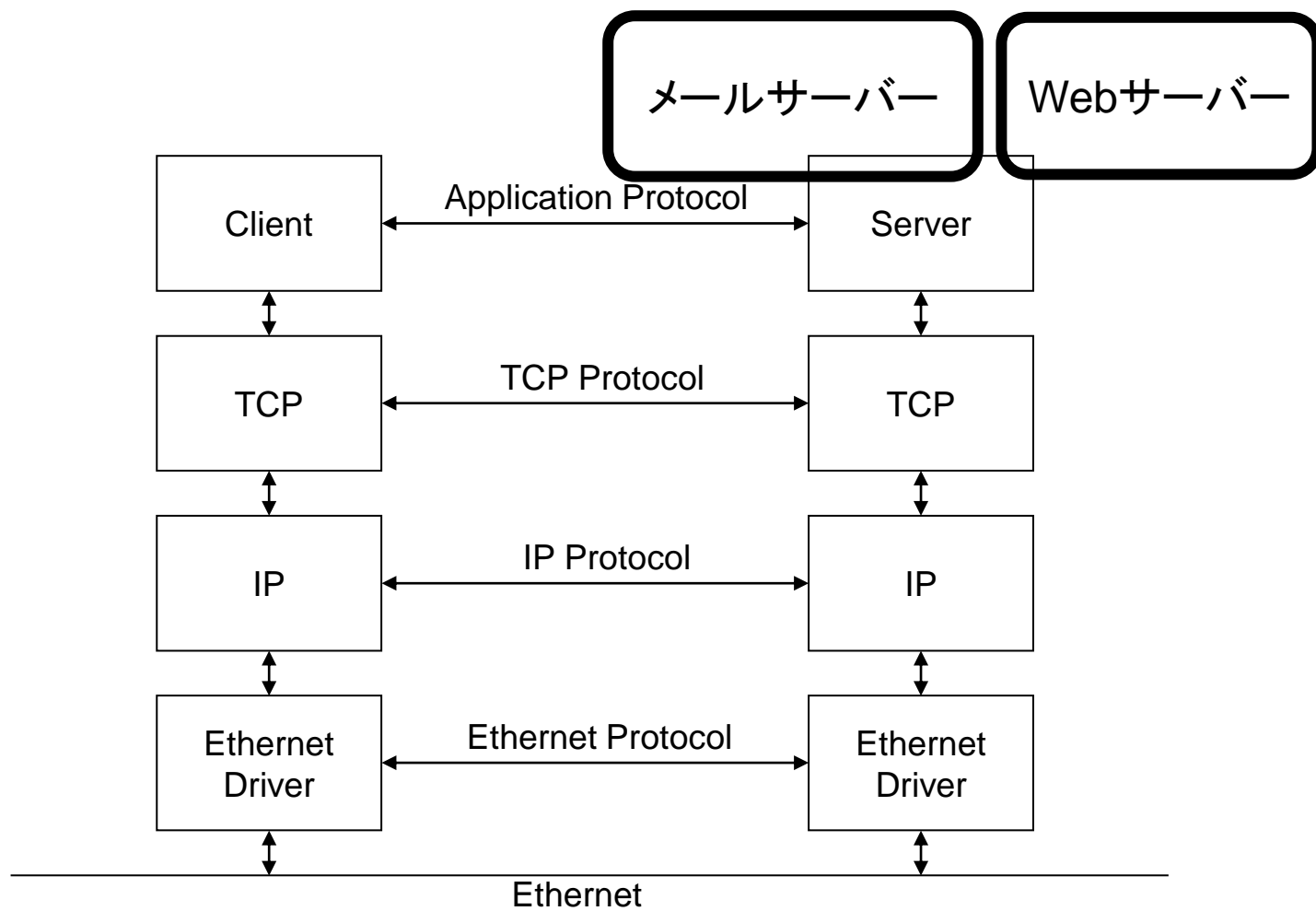
Destination IP Address



# IP

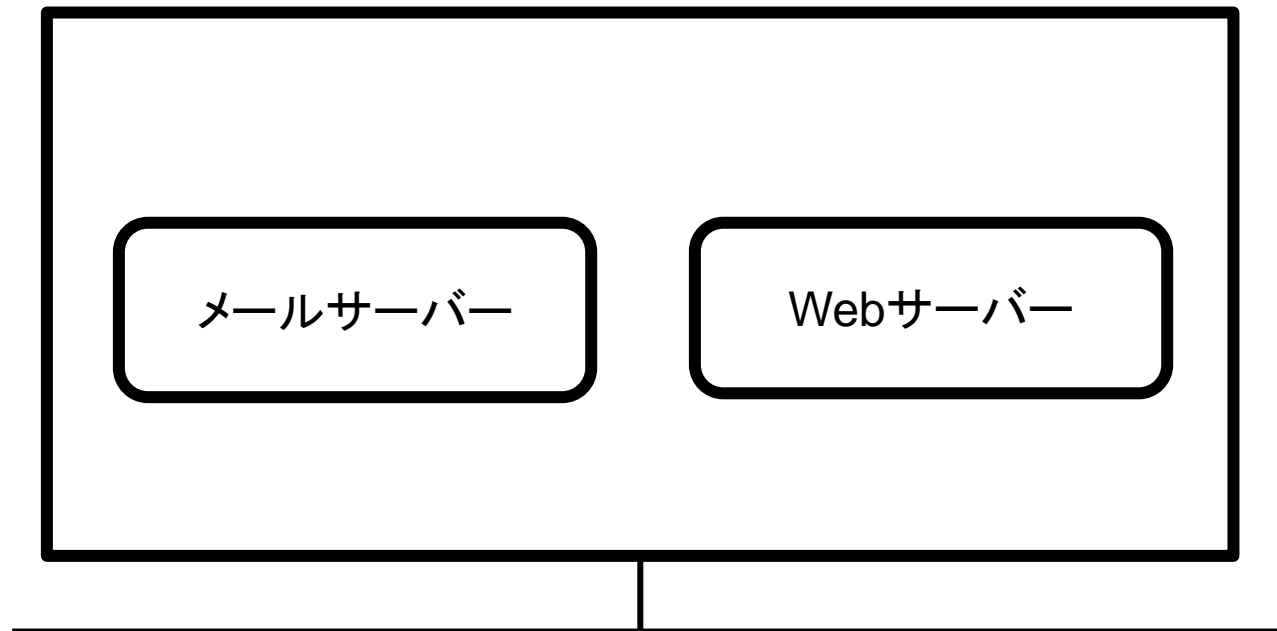
- 遠くまで届くようになった
- PCで動くネットワークプログラムは1個ではない
- 届いたパケットを仕分けるしくみが必要
- 相手方どのプロセスに届けるのか仕分ける仕組みが必要





# ポート番号 (1)

- ひとつのPCで動いている複数のサーバープログラムを識別するしくみ (0~65535)

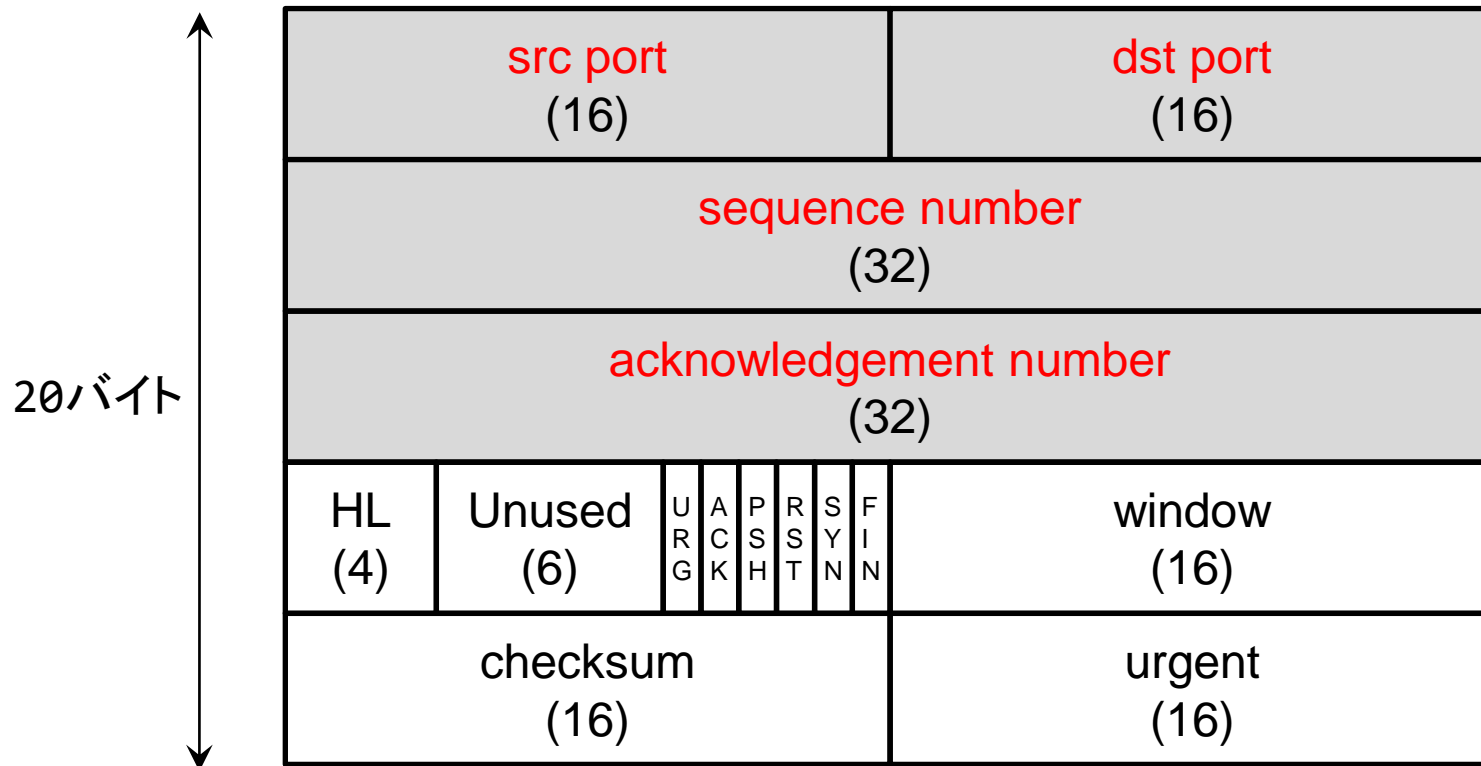


# ポート番号 (2)

- 主要なアプリケーションはポート番号が決まっている



# TCP Header

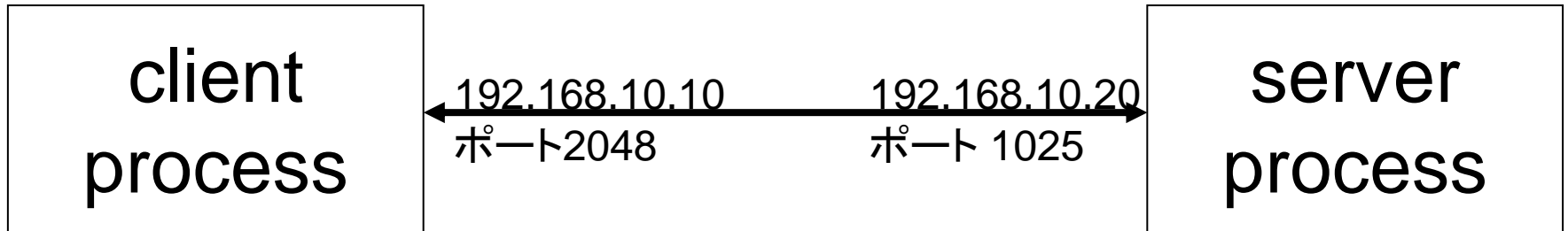


HL: Header Length (単位: 32ビットワード (4バイト))

sequence number, acknowledgement numberを使って信頼性のある通信を行う(後述)。



# IP Address、Port



IPアドレス： IP層

ポート： TCP/UDP層

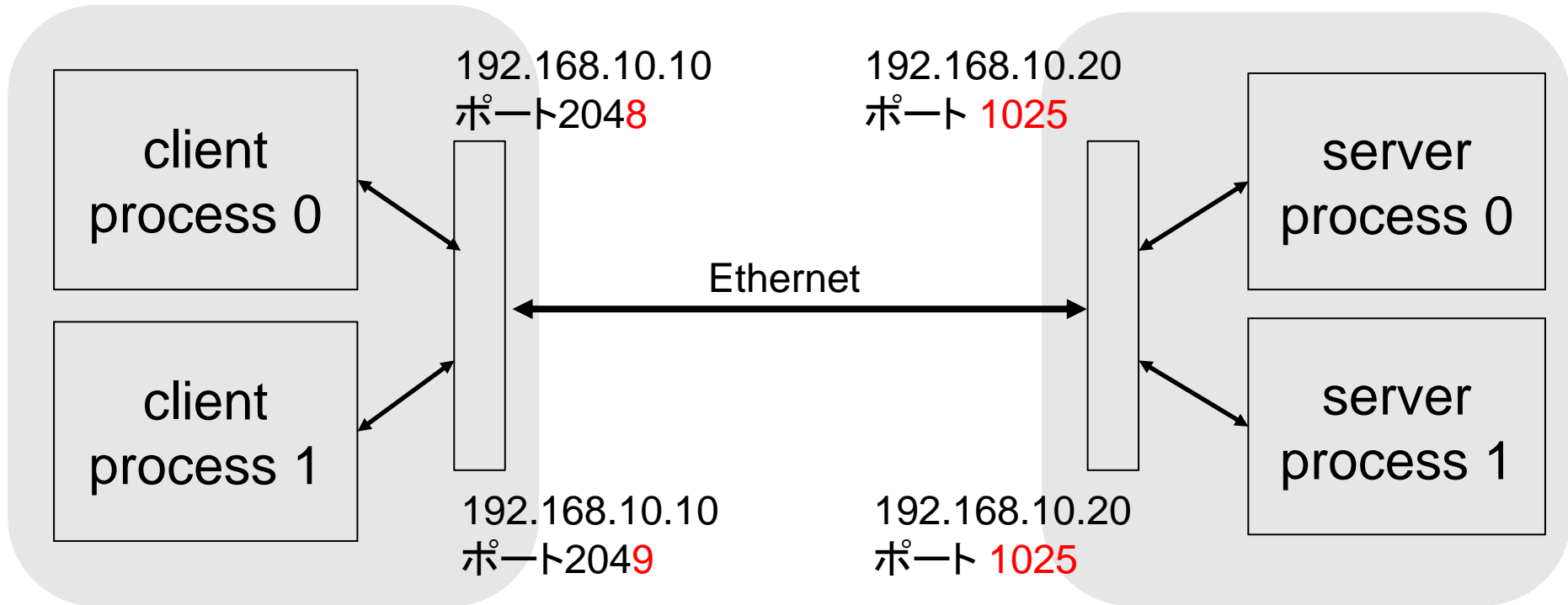
通信の相手方を指定、認識

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)

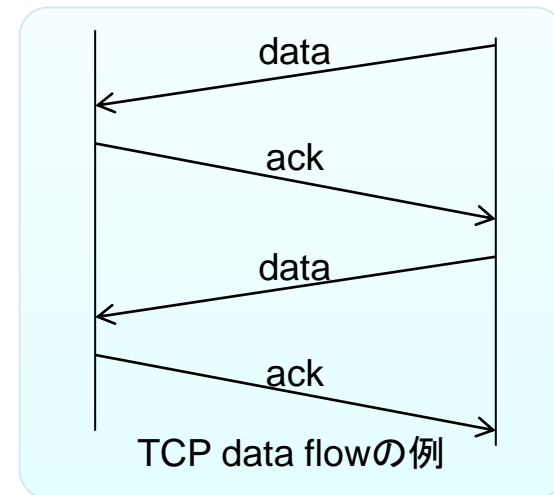
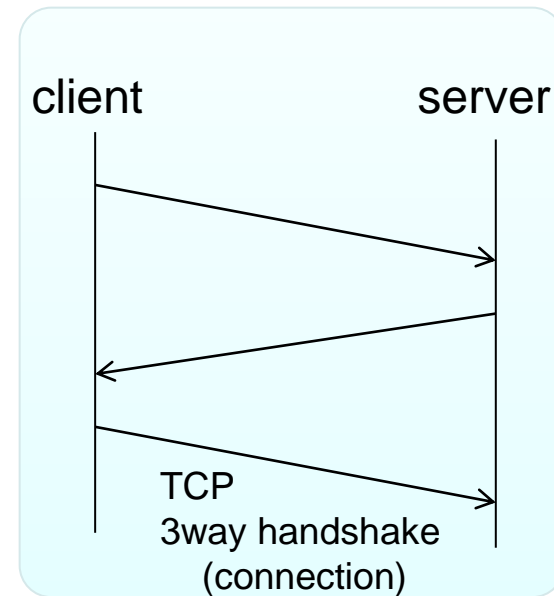
server (192.168.10.20, 1025, 192.168.10.10, 2048)

client (192.168.10.10, 2049, 192.168.10.20, 1025)

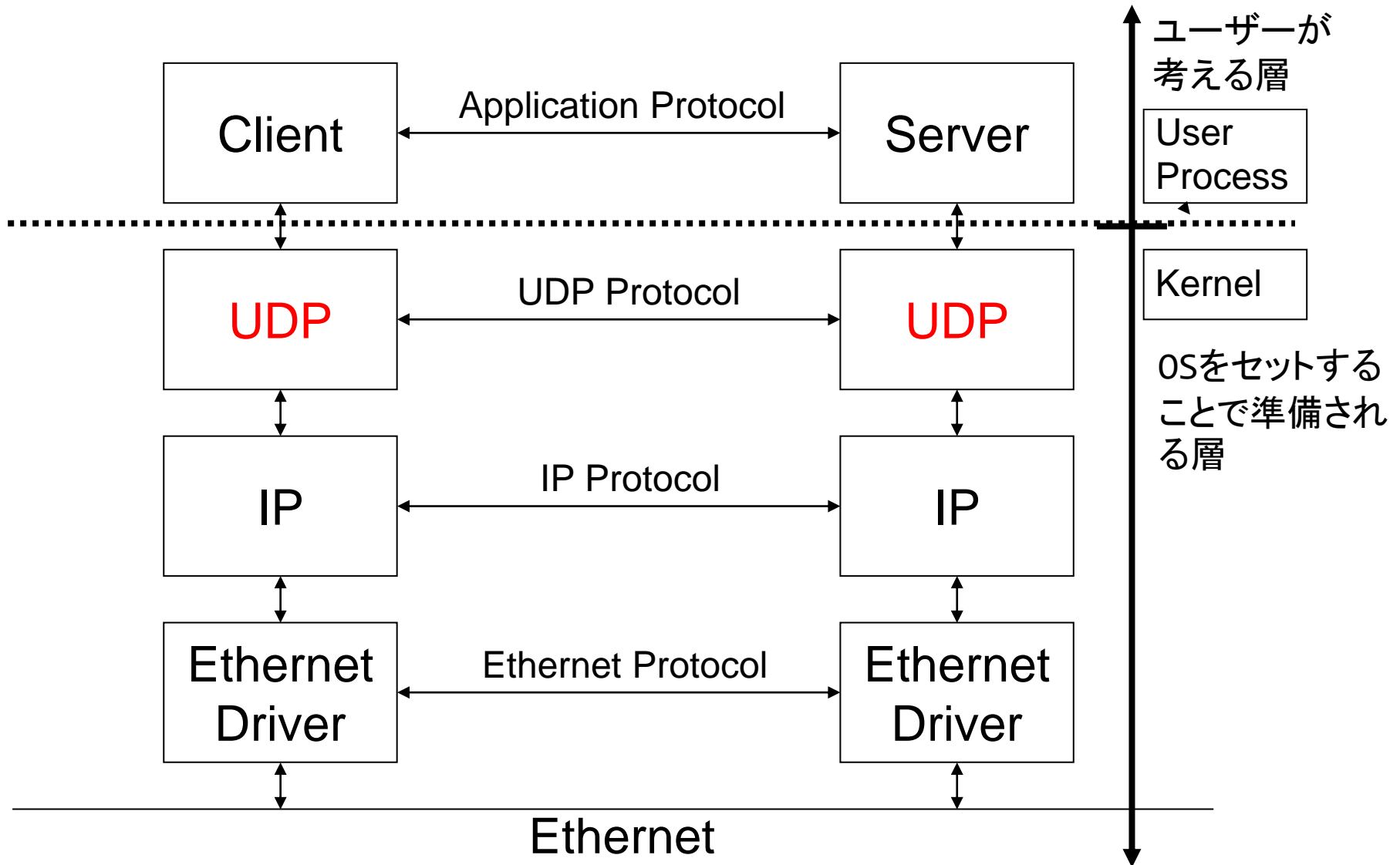
server (192.168.10.20, 1025, 192.168.10.10, 2049)

# TCPとUDP

- TCP (Transmission Control Protocol)
  - 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)。



# Ethernet Using UDP



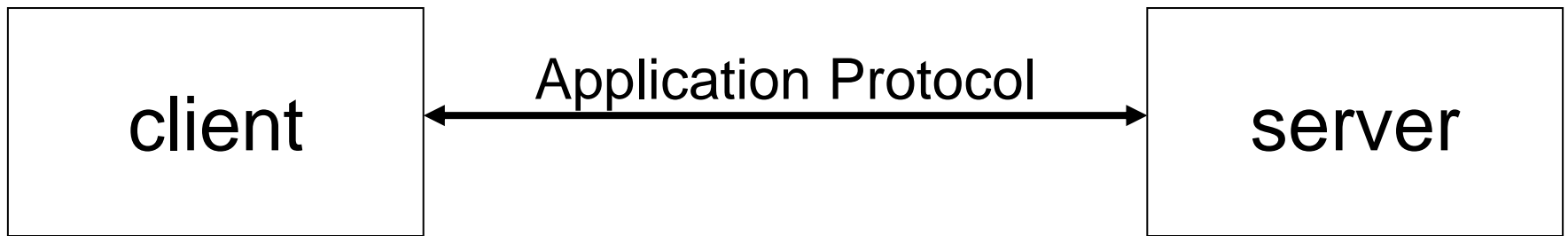
出典: Unix Network Programming p. 4

# 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

# 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。

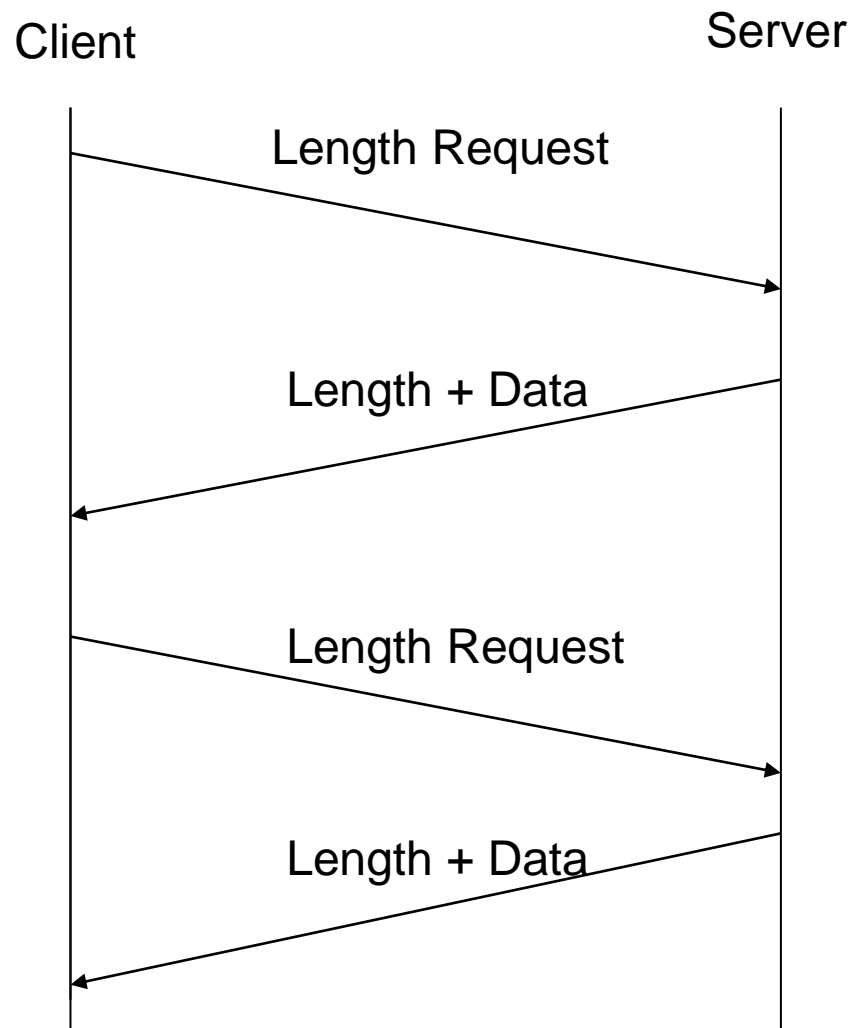
Unix Network Programming p. 3

# 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

# 通信プロトコルの例：実験システム

- 無手順（接続後すぐにデータを送り始める）
- ポーリングで読み取り（右図）





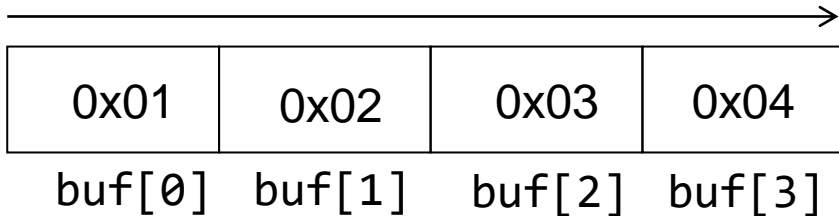
# ネットワークバイトオーダー (1)

- `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

アドレス



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

Intel CPUはlittle endian

```

#include <stdio.h>

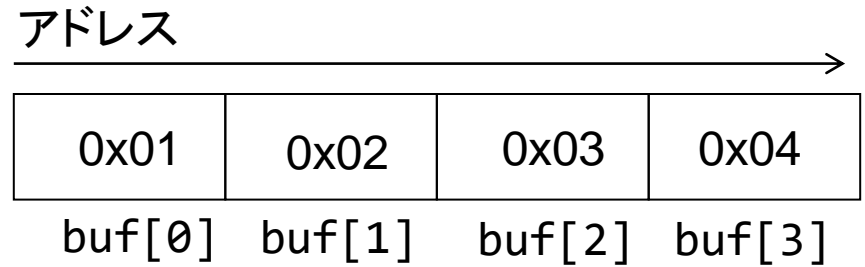
int main(int argc, char *argv[])
{
    unsigned char buf[4];
    unsigned int *int_p;
    unsigned int i;

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

    int_p = (unsigned int *) &buf[0];
    i = *int_p;
    printf("0x %08x = %d\n", i, i);

    return 0;
}

```



0x 04030201 = 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)

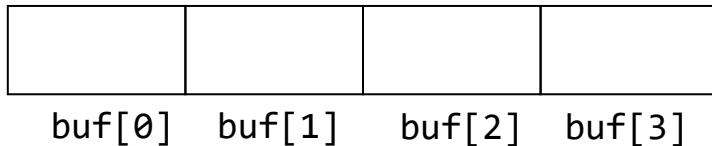
```
u_num.c[0]: 0xbfbfe850 0x04  
u_num.c[1]: 0xbfbfe851 0x03  
u_num.c[2]: 0xbfbfe852 0x02  
u_num.c[3]: 0xbfbfe853 0x01
```

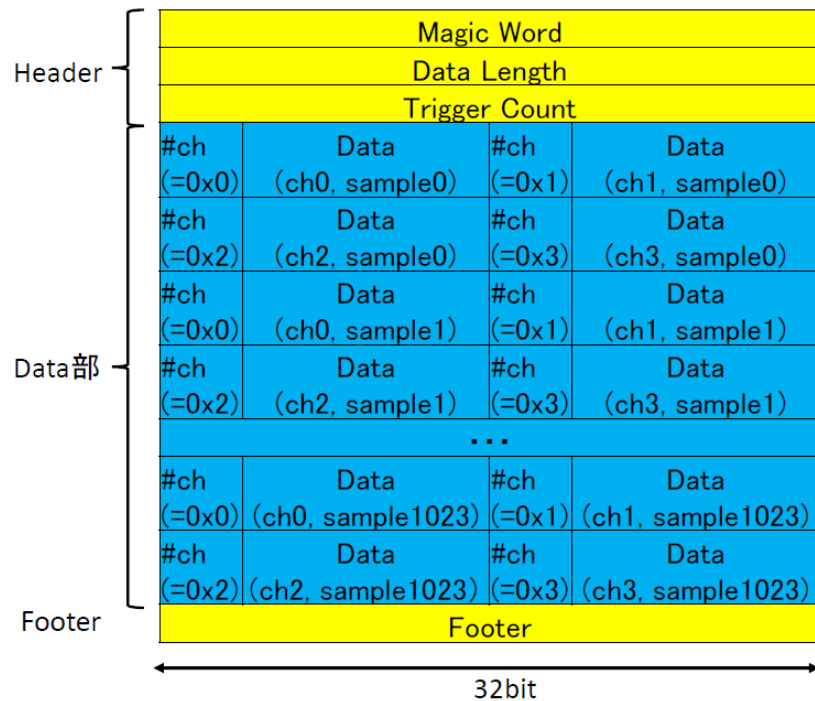
出力 (arm)

```
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 $\leftrightarrow$ ネットワークバイトオーダー変換関数
  - htonl (host to network long) (long: 4バイト整数)
  - htons (host to network short) (short: 2バイト整数)
  - ntohl (network to host long)
  - ntohs (network to host short)
- $\text{buf}[0]*256*256*256 + \text{buf}[1]*256*256 + \text{buf}[2]*256 + \text{buf}[3]$ とする方法もある





「Header」  
 ・Magic Word  
 常に 0x01234567

・Datalength  
 Data部のバイト長

・Trigger Count  
 1イベントのデータを送るごとに+1されていく。

「Data部」  
 ・各データは16bit  
 (上位4bitはch番号、下位12bitにデータ値)  
 ・1イベントはsample0から順々にsample1023まで  
 1024sampleを送る  
 ・各sampleはch0からch3までの4ch分を送る

「Footer」  
 ・Footer  
 常に 0x89ABCDEF  
 ※全てビッグエンディアン

3

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 */
```

- 最初にソケットを作る
- まだどこにも接続していない
- IPv4 (AF\_INET)
- TCP (SOCK\_STREAM)
- UDP (SOCK\_DGRAM)の指定をする
- 第3引数はTCP, UDPでは0を指定する
  - RAW Socketなどを使うときに0以外を使う (pingなど)

# connect()

```
struct sockaddr remote_addr;
```

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

- remote\_addr: 接続する相手を指定
  - 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)
  - IPv4ではstruct sockaddr\_in 構造体を使う
  - 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()

```
#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 */

memset(&servaddr, 0, sizeof(servaddr)) /* initialize */
servaddr . sin_family = AF_INET;
servaddr . sin_port = htons(port); /* network byte order */
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) {
    perror("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";
```

注: エラー処理をしていないので  
このままではだめです

```
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);
```

# 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)。

私が自分で作って使っているもの

<https://github.com/h-sendai/socketkit>

自分用なのでドキュメントがありません

# 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()

```
unsigned 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？
- `man socket`

# 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 file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file

# 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)。

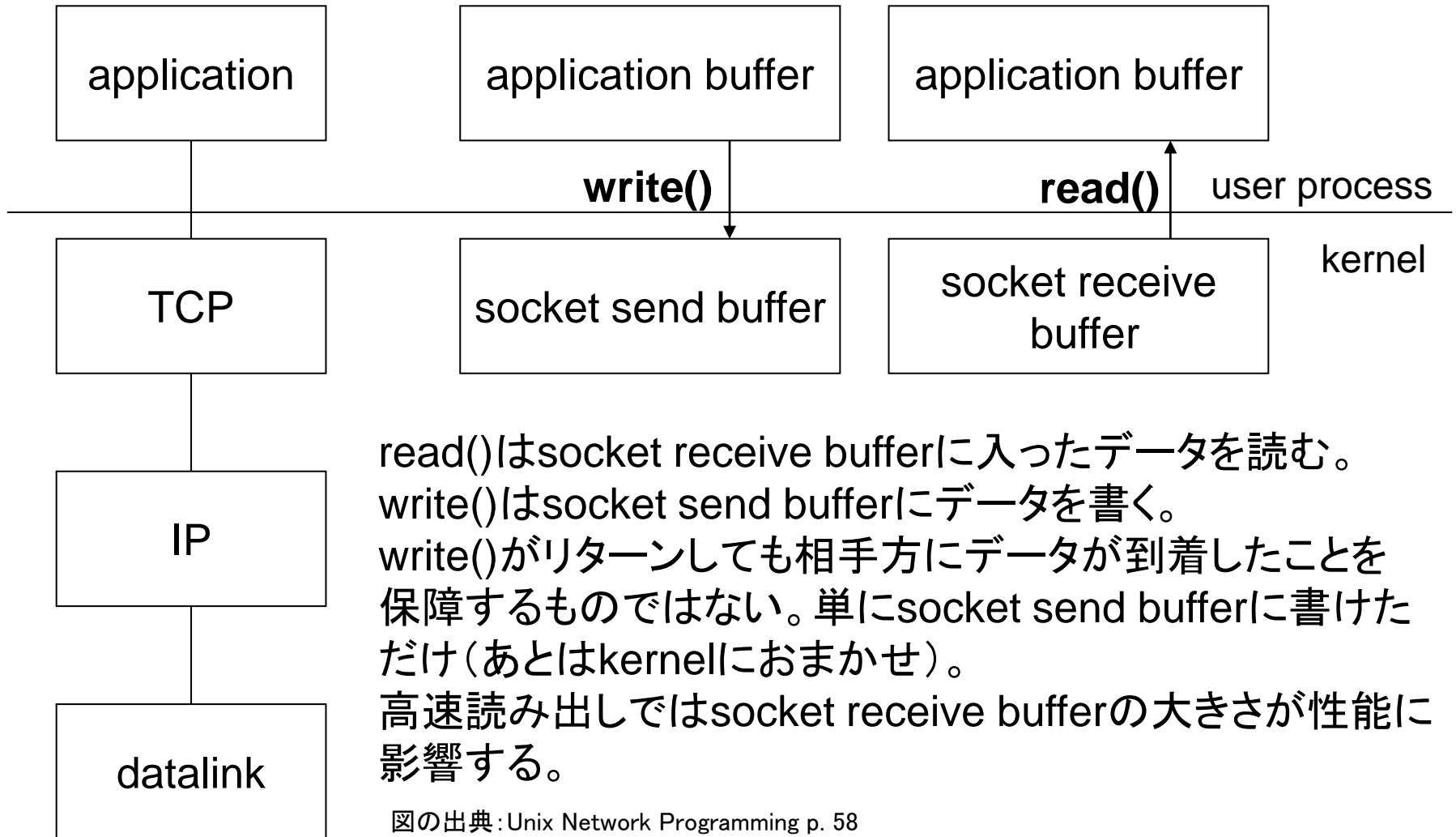
# もくじ

- 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
  - イーサネットスイッチのメモリー不足

# 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





# 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
```

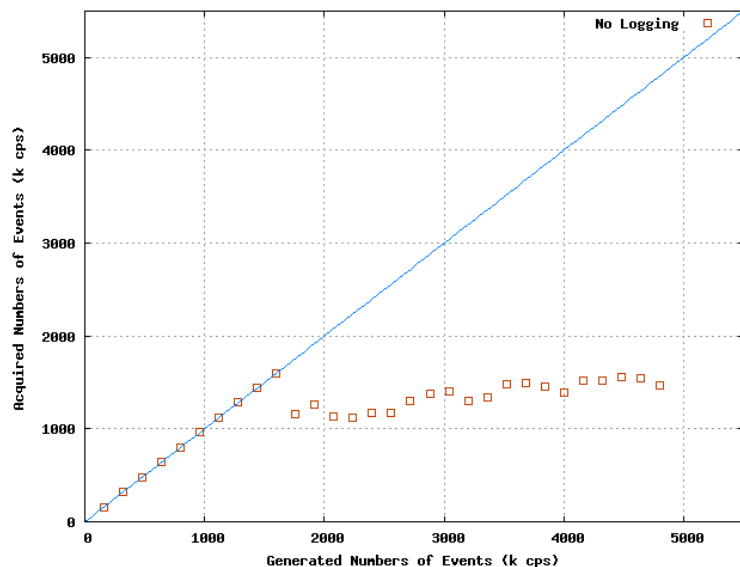
```
echo $so_rcvbuf_max > /proc/sys/net/core/wmem_max
echo $so_sndbuf_max > /proc/sys/net/core/rmem_max
read min init max          < /proc/sys/net/ipv4/tcp_rmem
echo $min $init $so_rcvbuf_max > /proc/sys/net/ipv4/tcp_rmem
read min init max          < /proc/sys/net/ipv4/tcp_wmem
echo $min $init $so_sndbuf_max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

- setsockopt()を使うとプログラム内で設定できる。

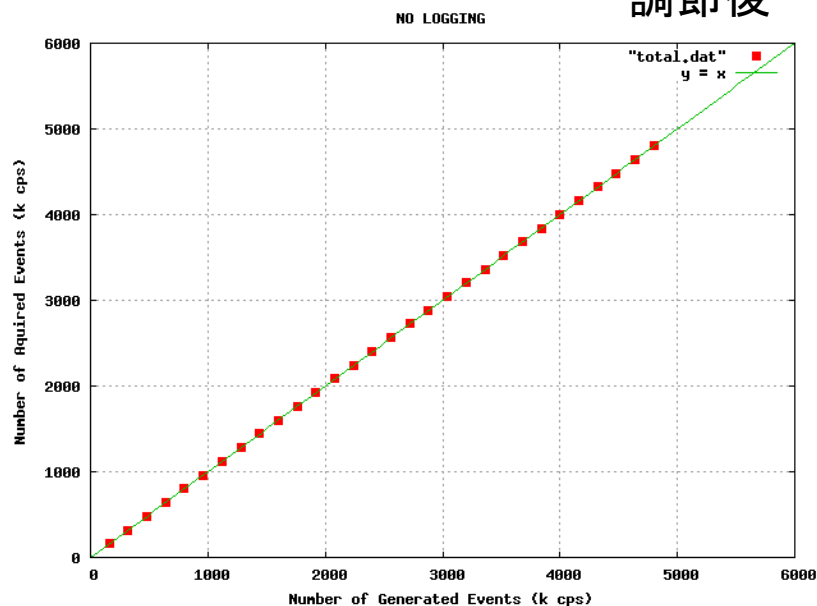
# ソケットレシーブバッファの大きさ調節による改善例

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



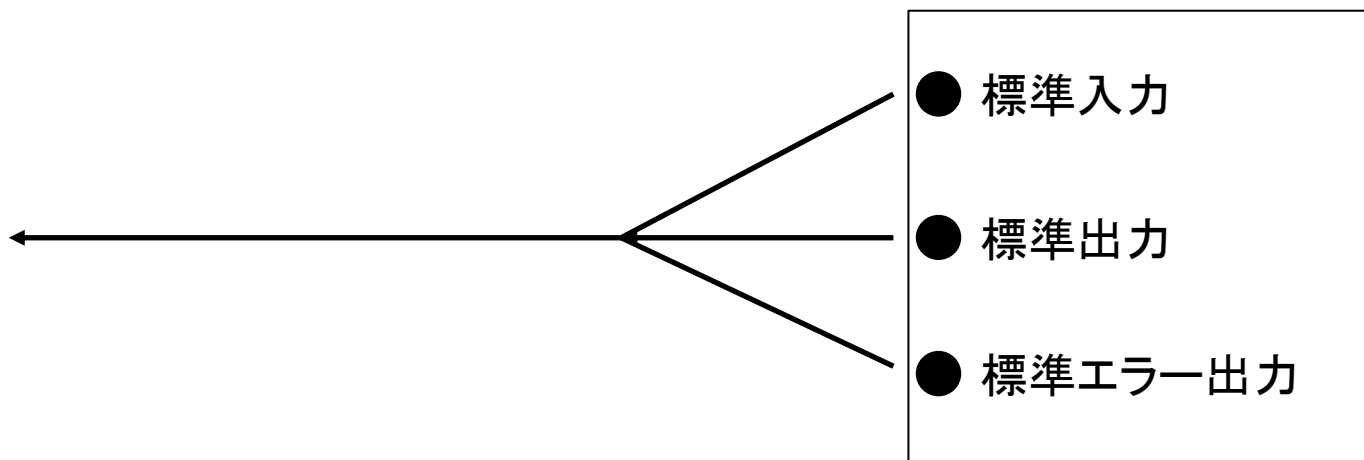
多重読み出しで複数モジュールから読み出し  
各モジュールは同一レートでデータを送ってくるようにセット  
読むモジュール数を1, 2, 3, と増加させていった。

# データを送ってくるサーバー (既製品)

- xinetd内蔵サーバー daytime (port 13)、chargen (port 19)
- セットアップ (Scientific Linux, CentOSの場合)
  - rpm -q xinetd で入っているかどうか確認
  - yum install xinetd でインストール
  - /etc/xinetd.d/daytime-stream、/etc/xinetd.d/chargen-streamで  
disable = no  
に変更して service xinetd restart
- nc localhost 13  
すると現在日時が表示される
- nc localhost 19 でデータがずらずらでてくる

# xinetdの利用

- xinetdを設定すると、標準入力、標準出力、標準エラー出力を使うプログラムを即座にネットワーク対応することが可能



Unix Network Programming p. 376

# xinetdの利用

1. /etc/xinetd.d/mycmd:

```
service mycmd
{
    port                = 60000
    socket_type         = stream
    type                = UNLISTED
    wait                = no
    user                = username
    server               = /home/username/bin/mycmd
    disable              = no
}
```

2. /home/username/bin/mycmdの用意

```
chmod +x mycmd
```

3. sudo service xinetd restart

4. nc localhost 60000

```
/home/user/bin/mycmd
```

```
#!/bin/sh
```

```
read line
```

```
now=$(date)
```

```
echo "$now: hello"
```

```
echo "user input: $line"
```

```
% nc localhost 60000
```

```
12345 (と入力。これがline変数に入る)
```

```
Mon Aug 31 09:50:28 JST 2015: hello
```

```
user input: 12345
```

# イーサネットスイッチのメモリー不足

- イーサネットスイッチの動作
  - いったんスイッチ内メモリーにパケットを保存
  - 届け先が分かっているならそのポートへ、わからないなら全部のポートへフォワード(次回からわかるようになる)
  - いったんメモリーに保存するのでトリガーと同時にドンとくるとメモリーに入りきらない。入りきらない分は消える
  - データ消失！
- 対応
  - レートが許すなら1GbEではなくて100Mbpsに落とす
  - 高いスイッチを買う(メモリー量は非公開な場合が多いが)

# Linuxのしくみ



実験と図解で学ぶOS、仮想マシン、コンテナの  
基礎知識【増補改訂版】

2022年10月17日紙版発売

武内覚 著

B5変形判／336ページ

定価(本体3530円＋税)

ISBN 978-4-297-13148-7

<https://gihyo.jp/book/2022/978-4-297-13148-7>



# Linuxプログラミング



ふつうのLinuxプログラミング 第2版  
Linuxの仕組みから学べるgccプログラミングの王道

青木 峰郎 著

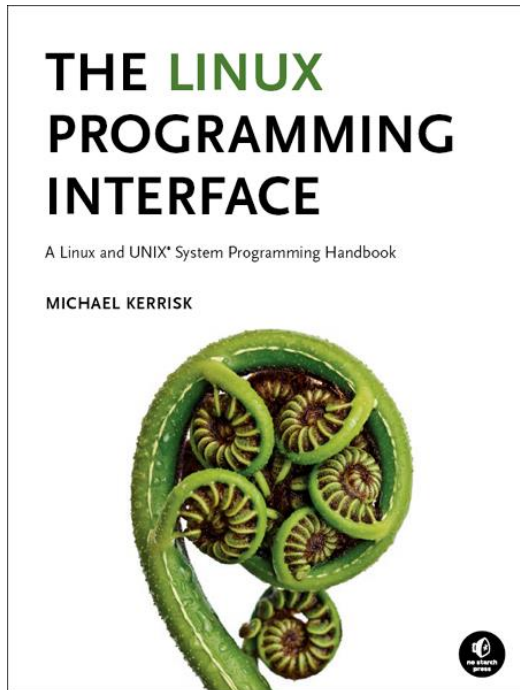
ISBN

978-4-7973-8647-9

<https://www.sbcr.jp/products/4797386479.html>



# Linux System Programming



*The Linux Programming Interface*

Michael Kerrisk

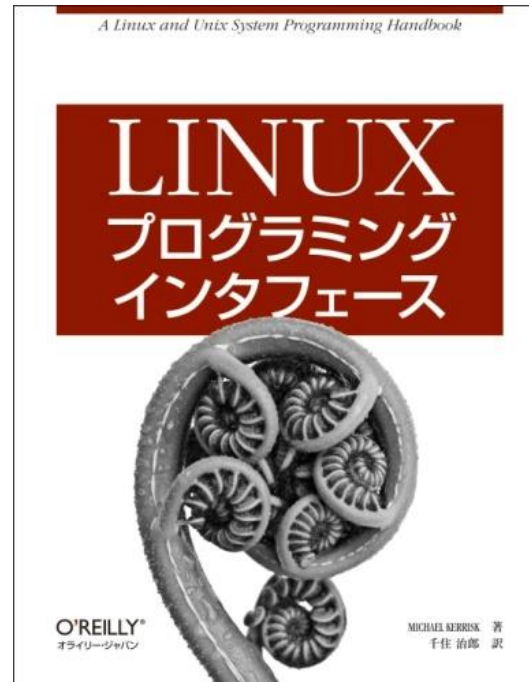
No Starch Press

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。

# ネットワークプログラミング参考書 (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

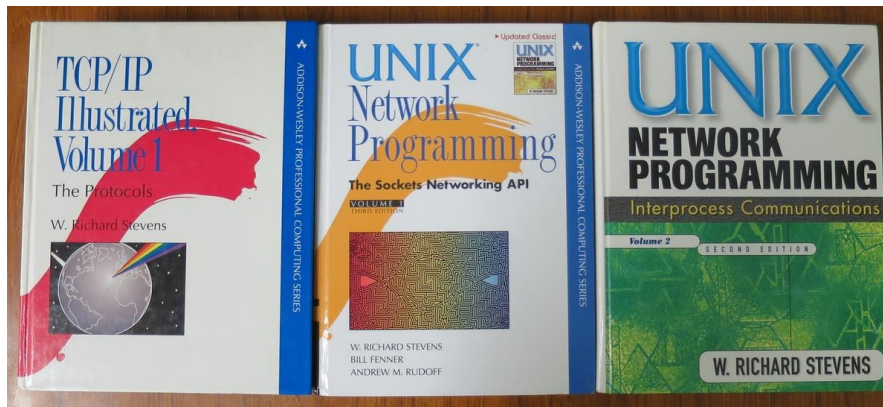
<http://ssl.ohmsha.co.jp/cgi-bin/menu.cgi?ISBN=4-274-06519-7>

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)

# ネットワークプログラミング参考書 (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)



# 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