

# ネットワークコンピューティング 第3回

中山 雅哉 ([m.nakayama@m.cnl.t.u-tokyo.ac.jp](mailto:m.nakayama@m.cnl.t.u-tokyo.ac.jp))  
関谷 勇司 ([sekiya@nc.u-tokyo.ac.jp](mailto:sekiya@nc.u-tokyo.ac.jp))

# 演習用ホスト

- ログインできるか確認してみて下さい
  - login1.sekiya-lab.info
  - login2.sekiya-lab.info
- ssh の秘密鍵があるマシンからログインできます
- まだ ssh 公開鍵を送ってない人
  - To : lecture@sekiya-lab.info
  - Subject : SSH public key
  - 本文 : 氏名, 学籍番号, 希望するログイン名
  - 添付ファイル or 本文に貼り付け : ssh 公開鍵

# 授業に関する情報

- 授業スライド、連絡事項、課題等に関する連絡
  - Web
    - <http://lecture.sekiya-lab.info/>
  - Contact E-mail
    - lecture@sekiya-lab.info
- 実習用ホスト
  - login1.sekiya-lab.info
  - login2.sekiya-lab.info

# クラウドコンピューティングの概要



# クラウドコンピューティング？

何をどうやって、どこまでやる？

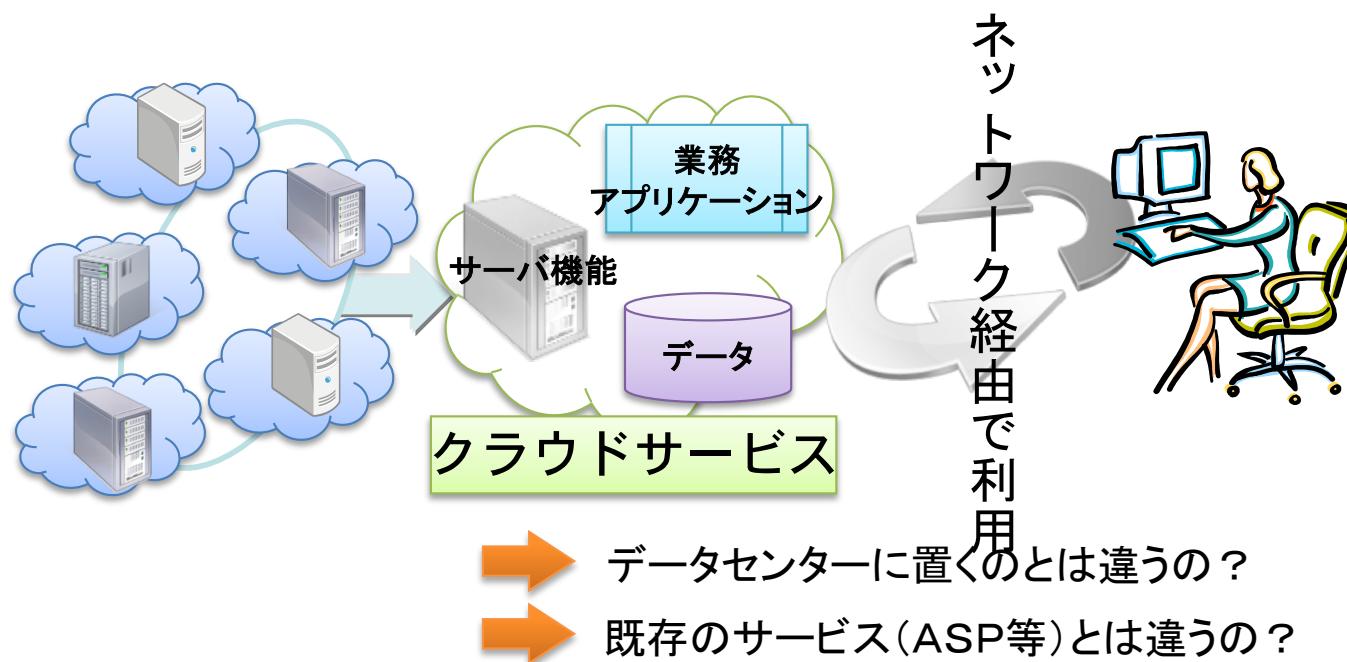
# クラウドの定義

- クラウドの定義
  - Above the Clouds: A Berkeley View of Cloud Computing
    - Electrical Engineering and Computer Sciences University of California at Berkeley, Technical Report No. UCB/EECS-2009-28
  - 規模性
    - 無限であるかのようにリソースを利用することができる
  - 即時性
    - 必要なとき必要なだけ時間単位で利用することができる
  - 可用性
    - 単一障害に影響されずサービスを継続することができる

# クラウドとは ~クラウド技術概論①~

- クラウドコンピューティング

ネットワーク上に存在するサーバが提供するサービスを、それらのサーバ群を意識することなしに利用できるというコンピューティング形態



# クラウドとは ~クラウド技術概論②~

- クラウドコンピューティングによるサービスと、これまでの共用サービス(ASPやホスティング等)との違い



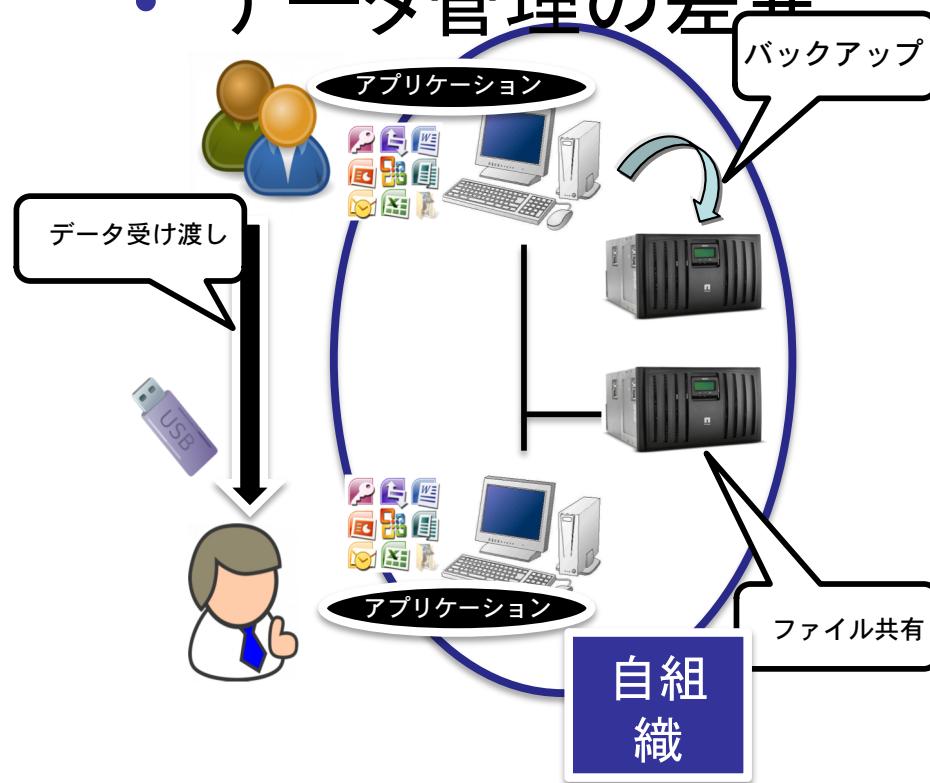
サービス利用者側からみると類似しているが…主に以下の点が相違

- 複数のコンピュータを1つのコンピュータ資源と見なすことができる
- 必要なときに必要なだけ使うことができる

} (システム利用の面で)  
自由度が高い

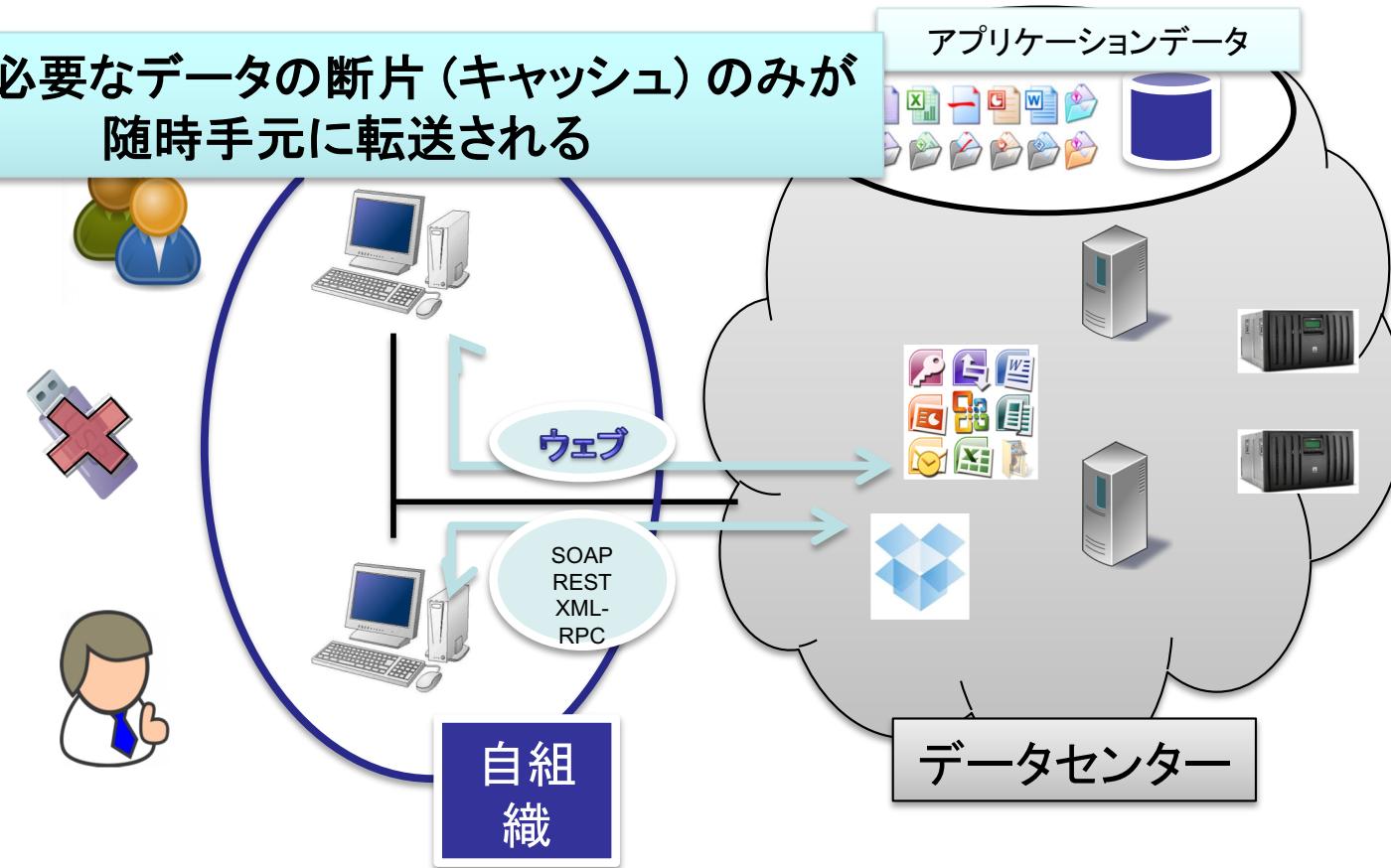
# クラウドによって何が変わるので？

- データ管理の差異



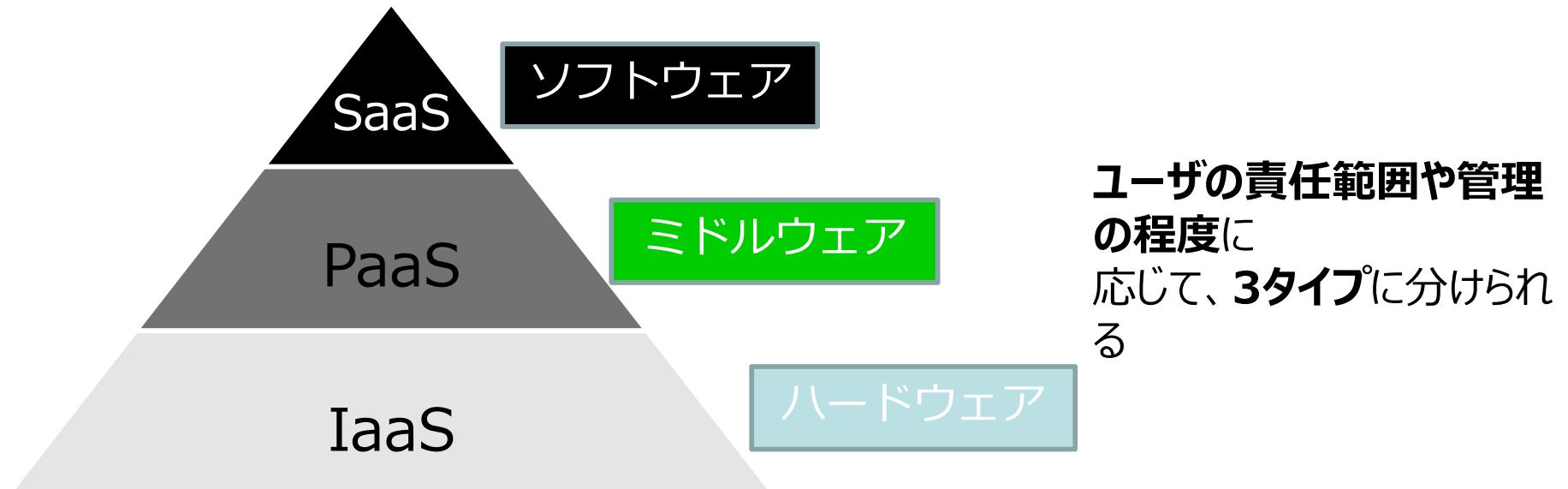
# クラウドによって何が変わらるのか？

操作に必要なデータの断片（キャッシュ）のみが  
随時手元に転送される



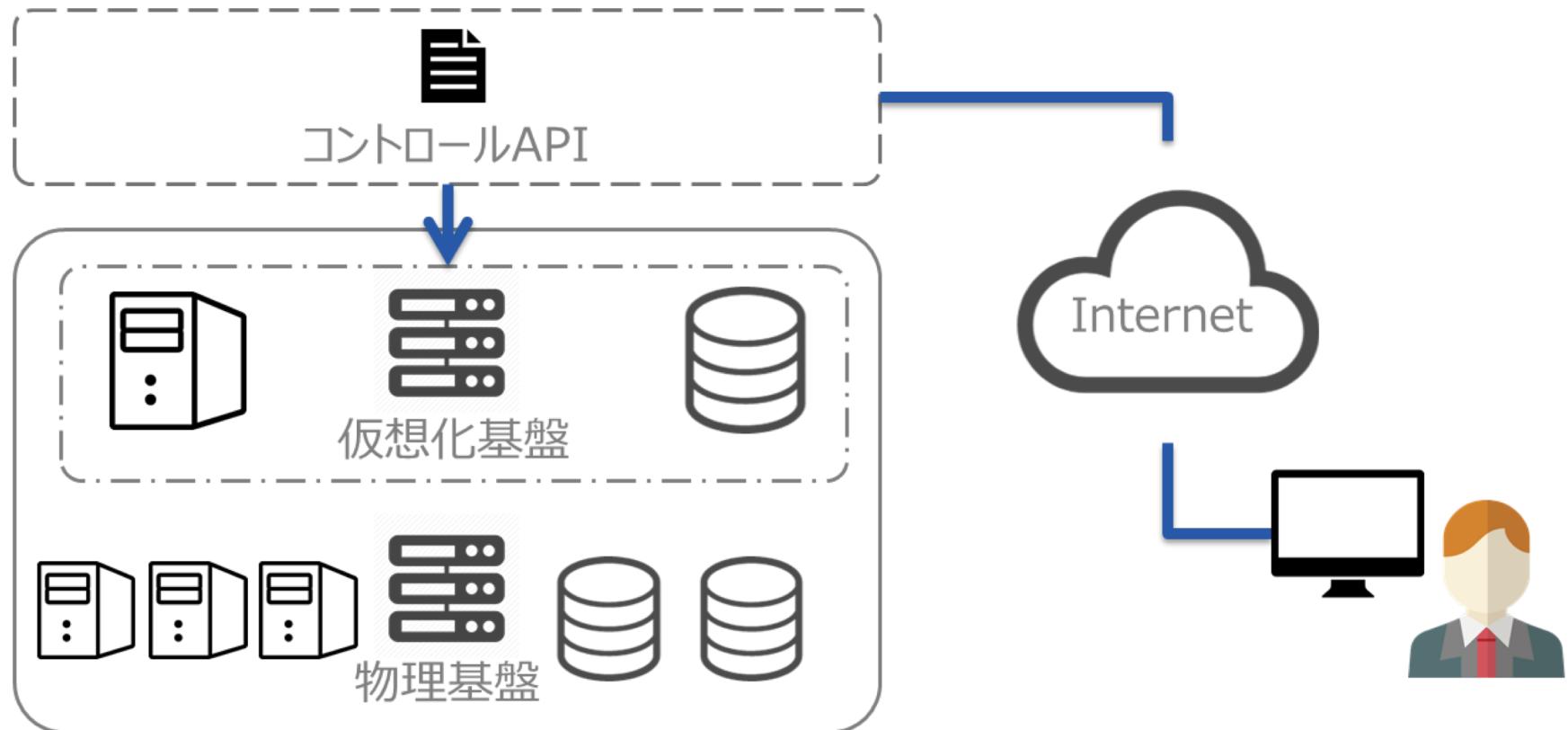
# クラウドコンピューティングとは

- クラウドコンピューティングの概要
  - 従量課金による、インターネット経由のITリソースとアプリケーションのオンデマンド配信全般を指して使用される



# クラウドコンピューティングの アーキテクチャ及び要素技術

# クラウドコンピューティングのアーキテクチャ

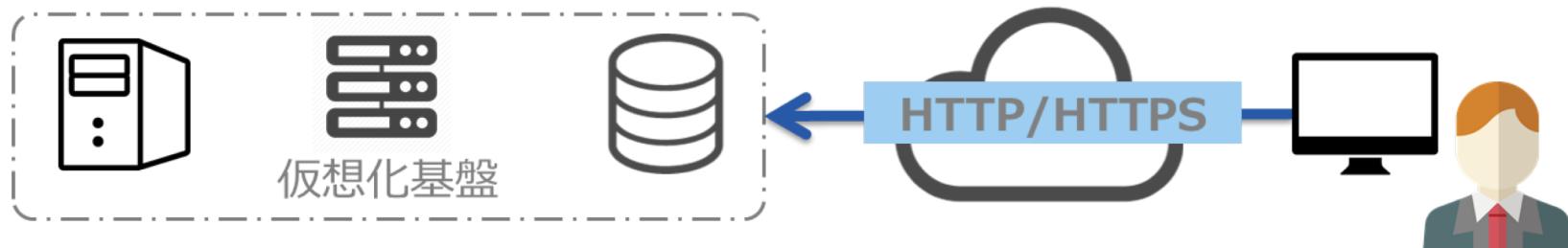


# クラウドコンピューティングの要素技術

- **物理基盤**
  - 仮想基盤を起動するための物理基盤
  - サーバ、ネットワーク、ストレージリソースを用いる
- **仮想化基盤**
  - 物理リソースを分割し、論理的なリソースとして提供
- **コントロールAPI**
  - http/https通信などをもとに、仮想化基盤のリソースを制御するAPI

# クラウドコンピューティングの要素技術

- コントロールAPI
  - APIにより、HTTP/HTTPS経由で仮想化基盤上のリソースを操作するコントロールインターフェイス
  - Webブラウザ経由で、仮想化基盤上リソース(VMなど)の起動/停止やモニタリングなどを行うことができる



# クラウドコンピューティングの要素技術

- クラウド上で提供される論理リソース
  - コンピュートリソース
    - サーバ、DBなど用途に応じた仮想サーバを起動
    - 用途ごとに最適化されたHWリソース(CPU、メモリなど)を利用可能
  - ネットワークリソース
    - コンピュートリソースを配置するための仮想ネットワーク環境を構築
    - パブリック、プライベートなど用途に応じたNW構成が可能
  - ストレージリソース
    - 複数の物理ストレージからなる巨大な仮想ストレージを利用可能

# 著名な商用クラウドの解説



Google Cloud Platform

## 著名な商用クラウドの解説

AWS ? Azure ? GCP ?

# AWSとは？

- AWS
  - Amazon Web Services 提供のパブリッククラウドサービス
  - 業界No1シェア、国内外で多数のユーザを持つ
  - IaaS、PaaS、SaaSと横断的なサービスを提供する
  - 取扱う技術分野においても、コンピューティングはもちろん、ビッグデータ、IoT、マシンラーニングなど幅広いレンジのサービスを提供している



# Azureとは？

- Azure
  - Microsoft提供のパブリッククラウドサービス
  - AWSに次いで、第二位のシェア
  - オンプレミスのWindows環境との協調を強みとしている
  - IaaS分野のみでなく、昨今は分析、IoT、ビッグデータ関連のサービスもリリースしている



## GCPとは？

- GCP (Google Cloud Platform)
  - Google提供のパブリッククラウドサービス
  - Youtube、gmailなど他Googleサービスと同じインフラを使用し、高パフォーマンスを実現
  - 主にApplication開発やデータ解析、機械学習に長けたサービスを豊富に用意



Google Cloud Platform

# 演習マシン

- AWS の上で動いています

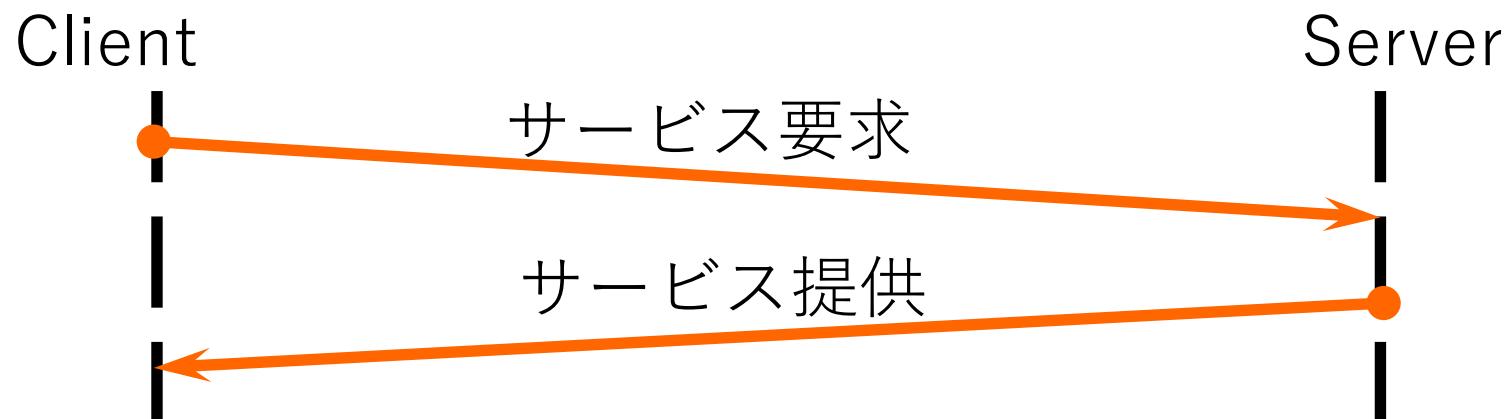
The screenshot shows the AWS EC2 Management Console interface. On the left, there's a sidebar with navigation links like EC2ダッシュボード, イベント, タグ, レポート, 制限, インスタンス, Launch Templates, スポットリクエスト, リザーブドインスタンス, Dedicated Host, イメージ, AMI, and バンドルタスク. The 'Instances' link is currently selected and highlighted in orange. The main content area displays a table of running instances:

Name	インスタンス ID	インスタンスタイプ	アベイラビリティゾーン	インスタンスの状態	ステータスチェック	アラームのステータス	パブリック DNS (IPv4)	IPv4 パブリック IP	IPv6 IP
login2	i-0042d7d3e206e792f	t2.micro	ap-northeast-1a	running	2/2 のチェック	なし	ec2-54-249-36-78.ap-n...	54.249.36.78	2406:da
login1	i-02501f5a0826cbd88	t2.micro	ap-northeast-1a	running	2/2 のチェック	なし	ec2-13-115-209-199.ap...	13.115.209.199	2406:da
Web	i-0eff842378382a22	t2.micro	ap-northeast-1a	running	2/2 のチェック	なし	ec2-13-231-128-203.ap...	13.231.128.203	2406:da

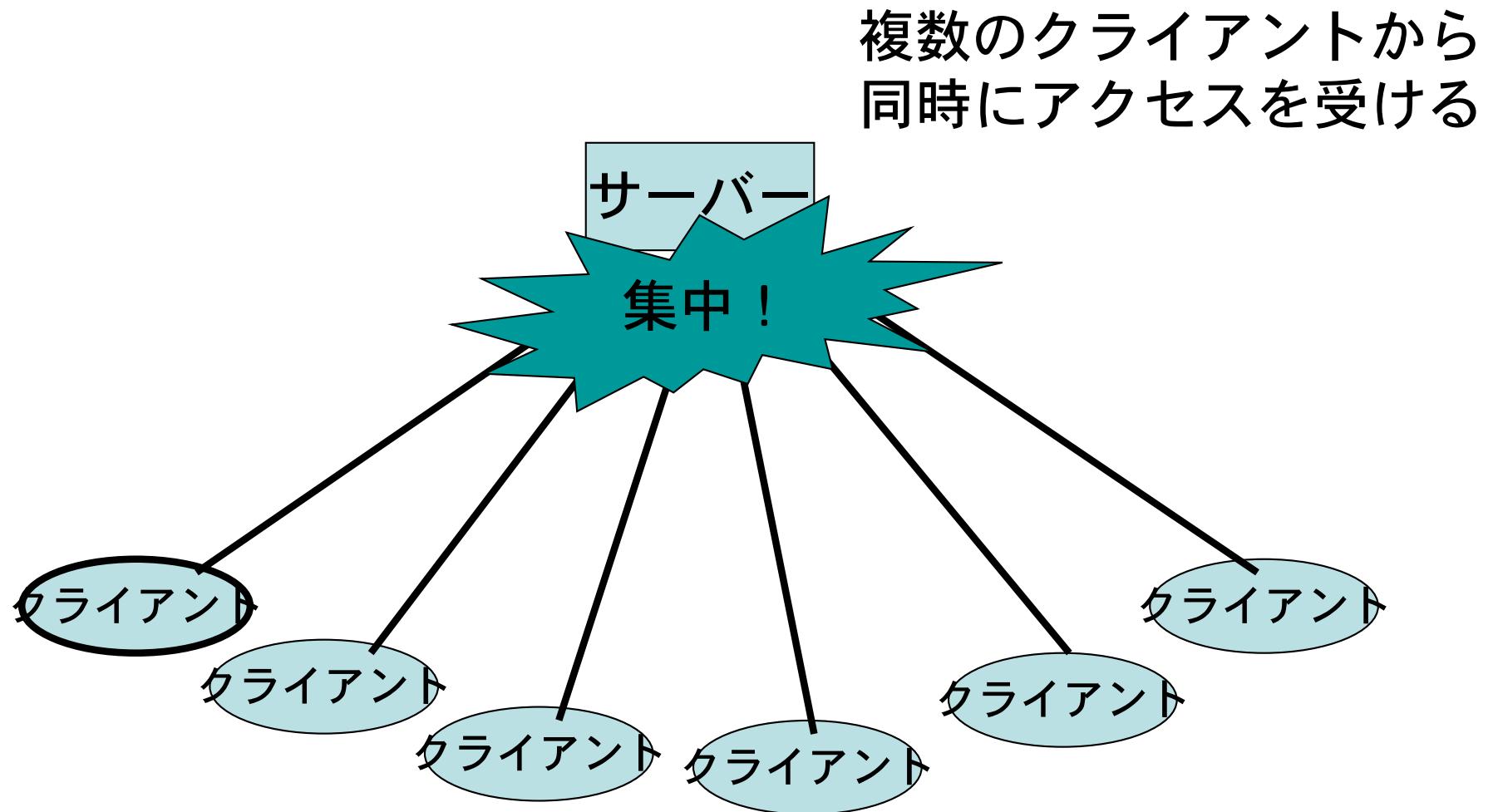
# TCPを用いた クライアントプログラム

# クライアント・サーバモデル

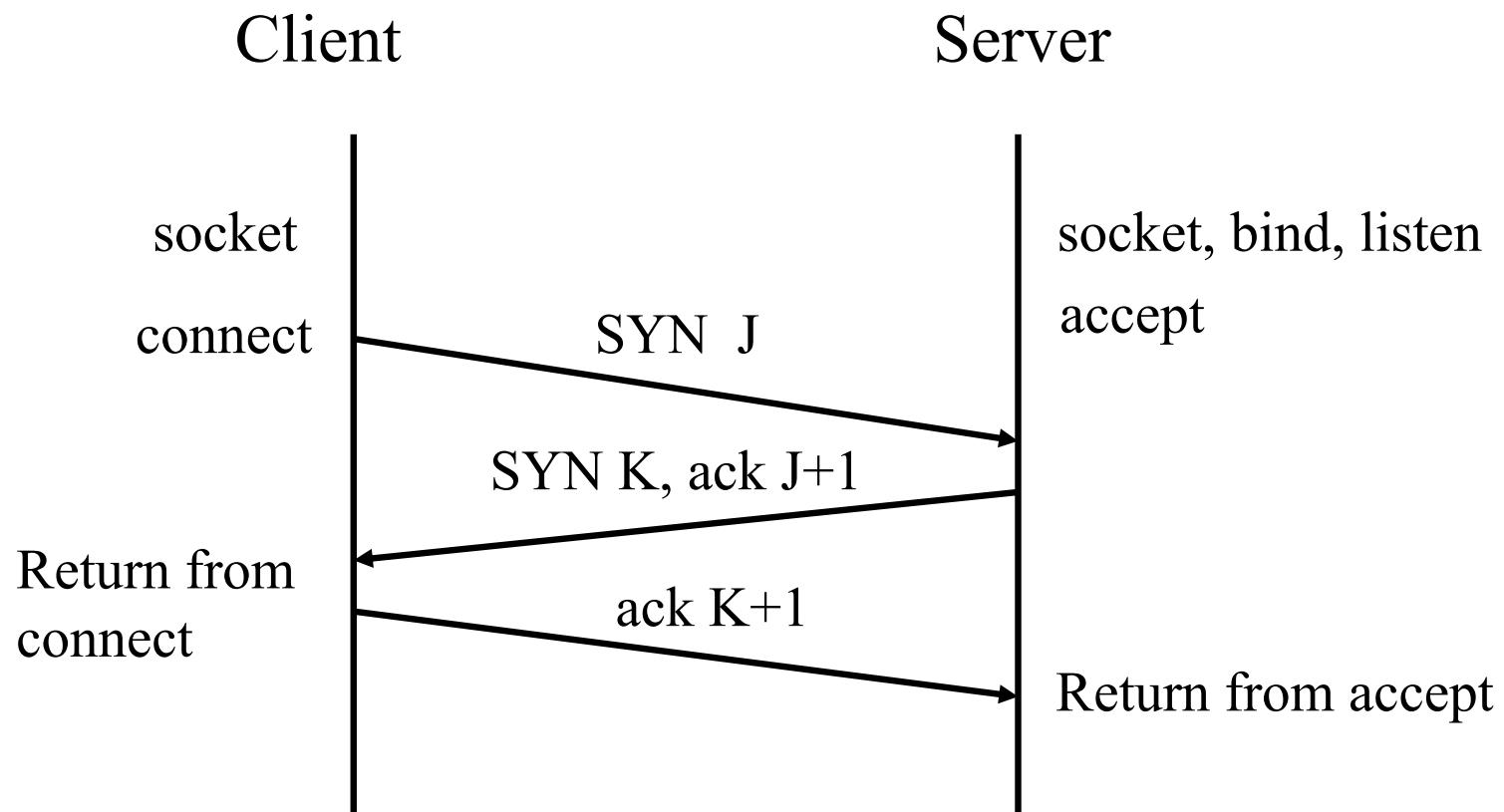
- ネットワークを介したサービスにおける通信モデル
- サーバ
  - 受動的にサービス提供する側、待ってくれる
- クライアント
  - 能動的にサービス提供を促す側、接続しに行く



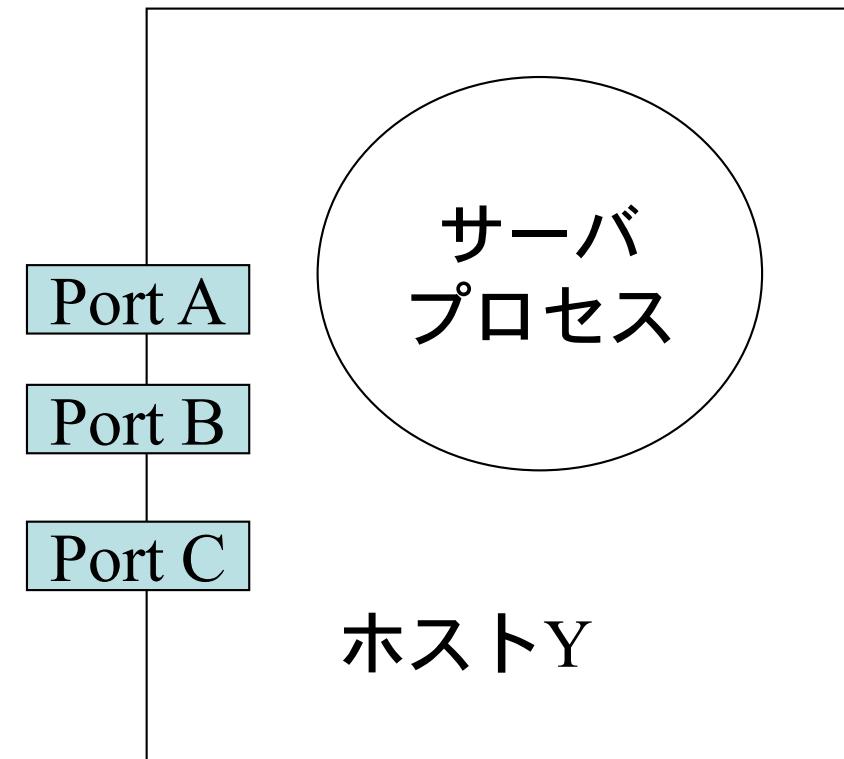
# サーバーとクライアントの違い



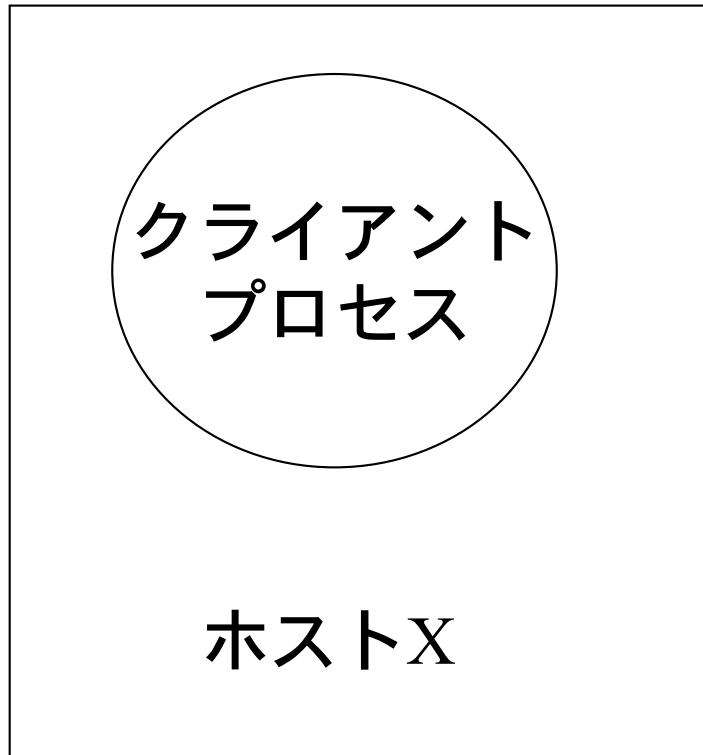
# Establish TCP connection



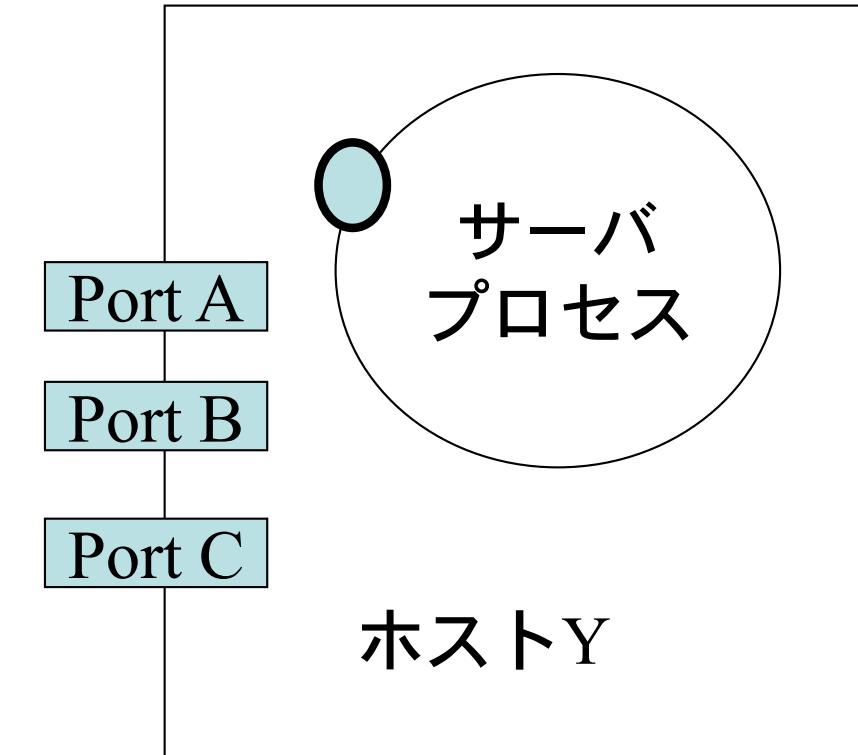
# 初期状態



# サーバがSocketを開いた状態



Socketを開く

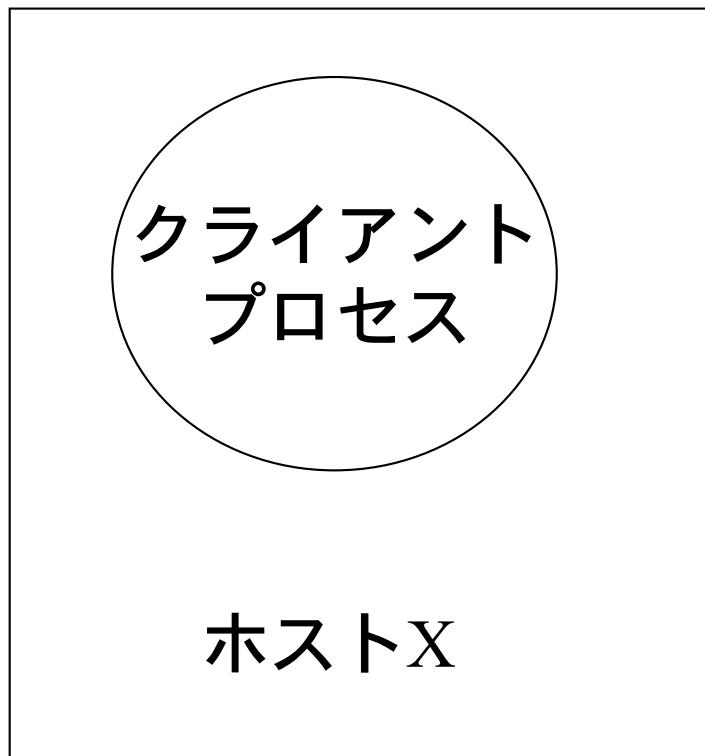


# サーバがbindした状態

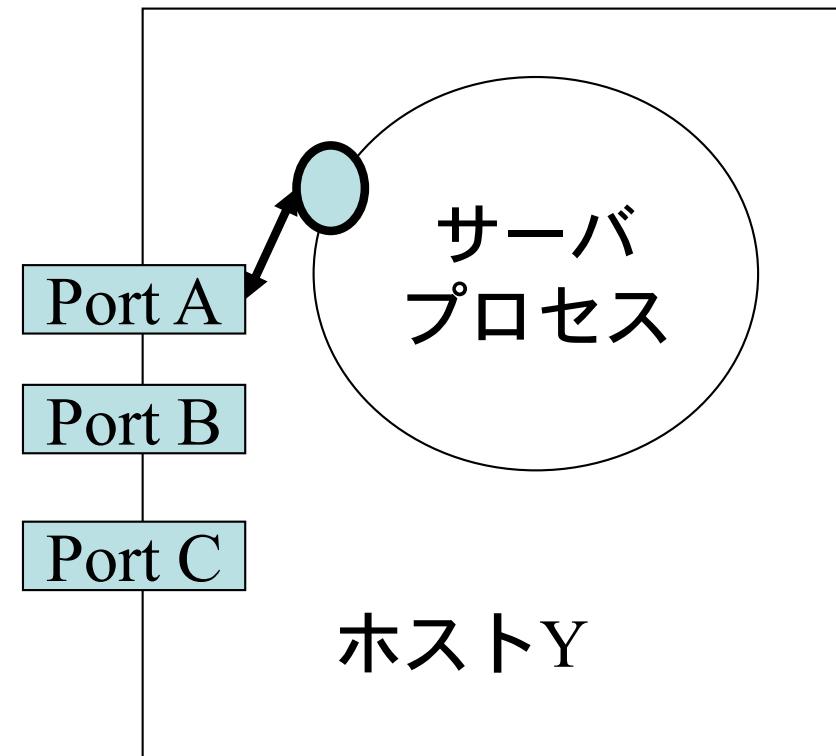
Proto LocalAddress  
**TCP** \*.\*A

ForeignAddress  
\*.\*

State  
**Closed**



IP Address: 10.0.1.1



IP Address: 10.0.2.1

# サーバがlistenした状態

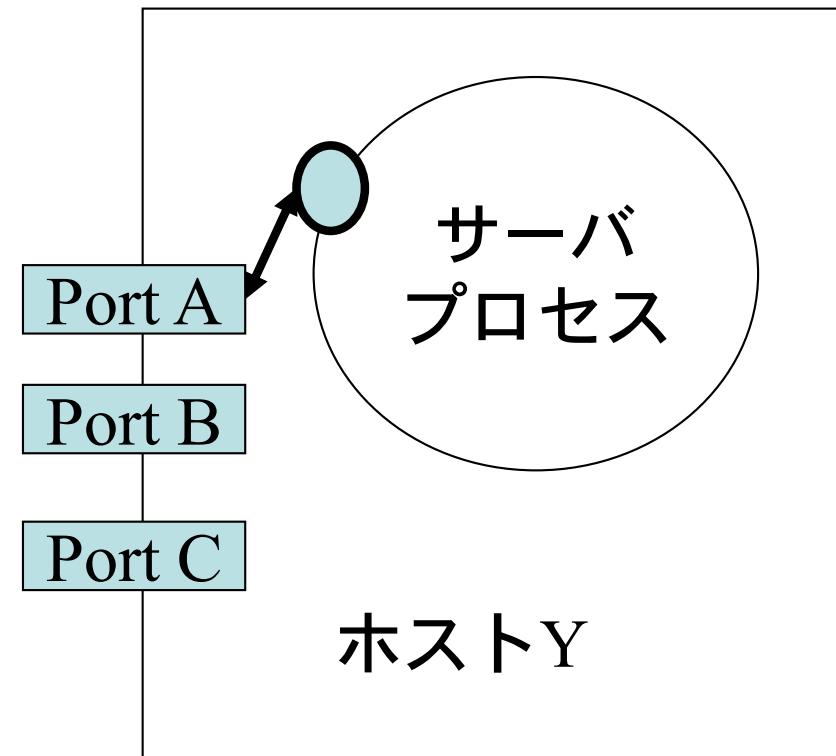
Proto LocalAddress  
**TCP** \*.\*A

ForeignAddress  
\*.\*

State  
**LISTEN**



IP Address: 10.0.1.1



IP Address: 10.0.2.1

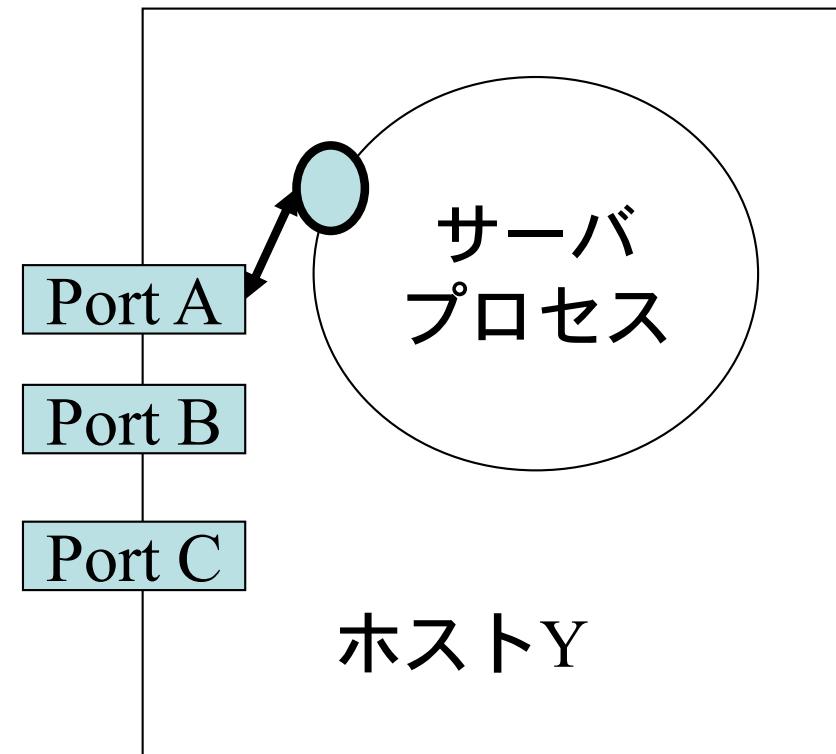
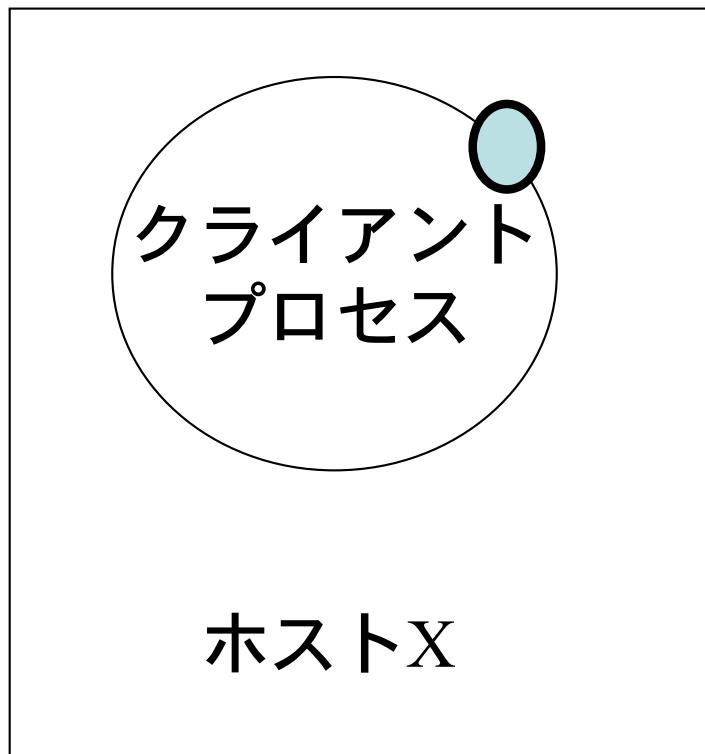
# クライアントがSocketを開いた状態

Proto LocalAddress  
**TCP** \*.\*A

ForeignAddress  
\*.\*

State  
**LISTEN**

Socketを開く



# connect() -> accept() した状態

Proto LocalAddress

TCP \*.\*A

TCP 10.0.2.1.A

ForeignAddress

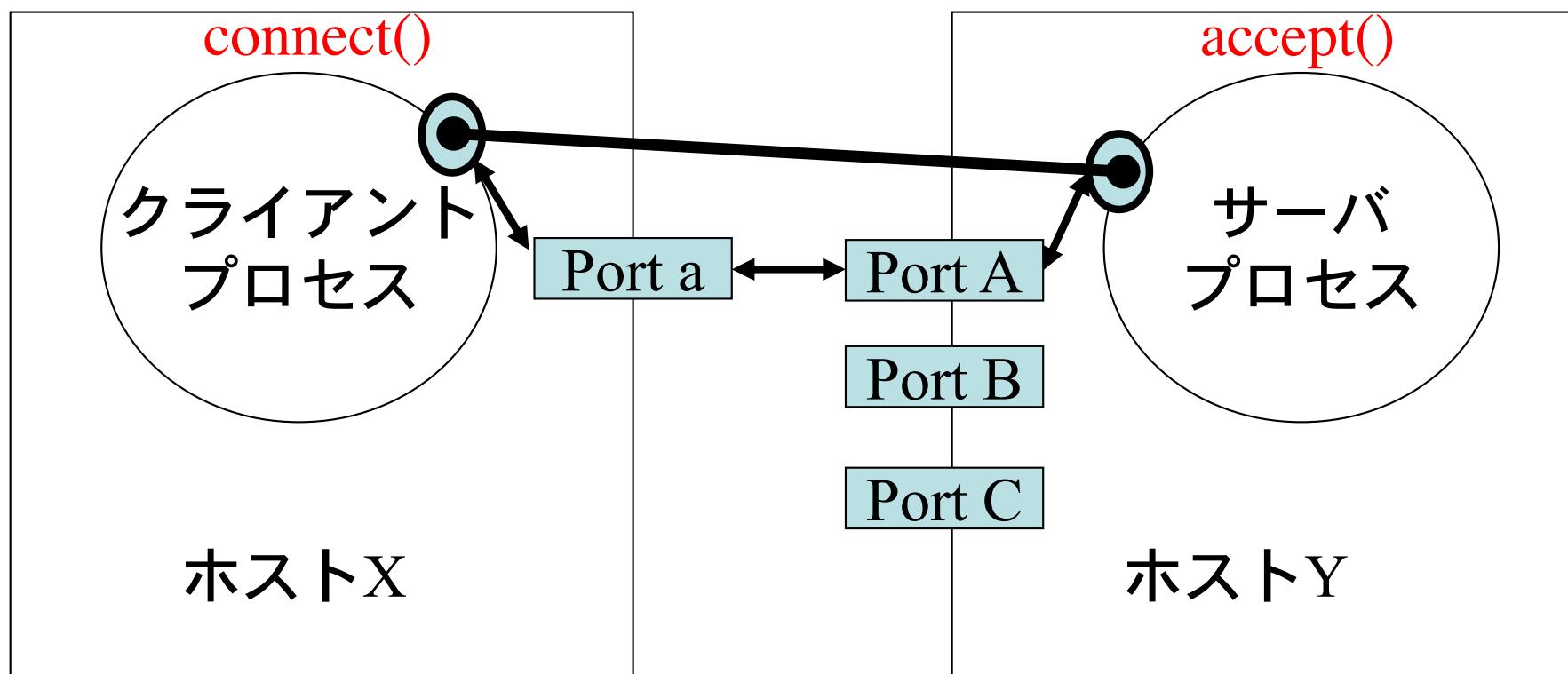
\*.\*

10.0.1.1.a

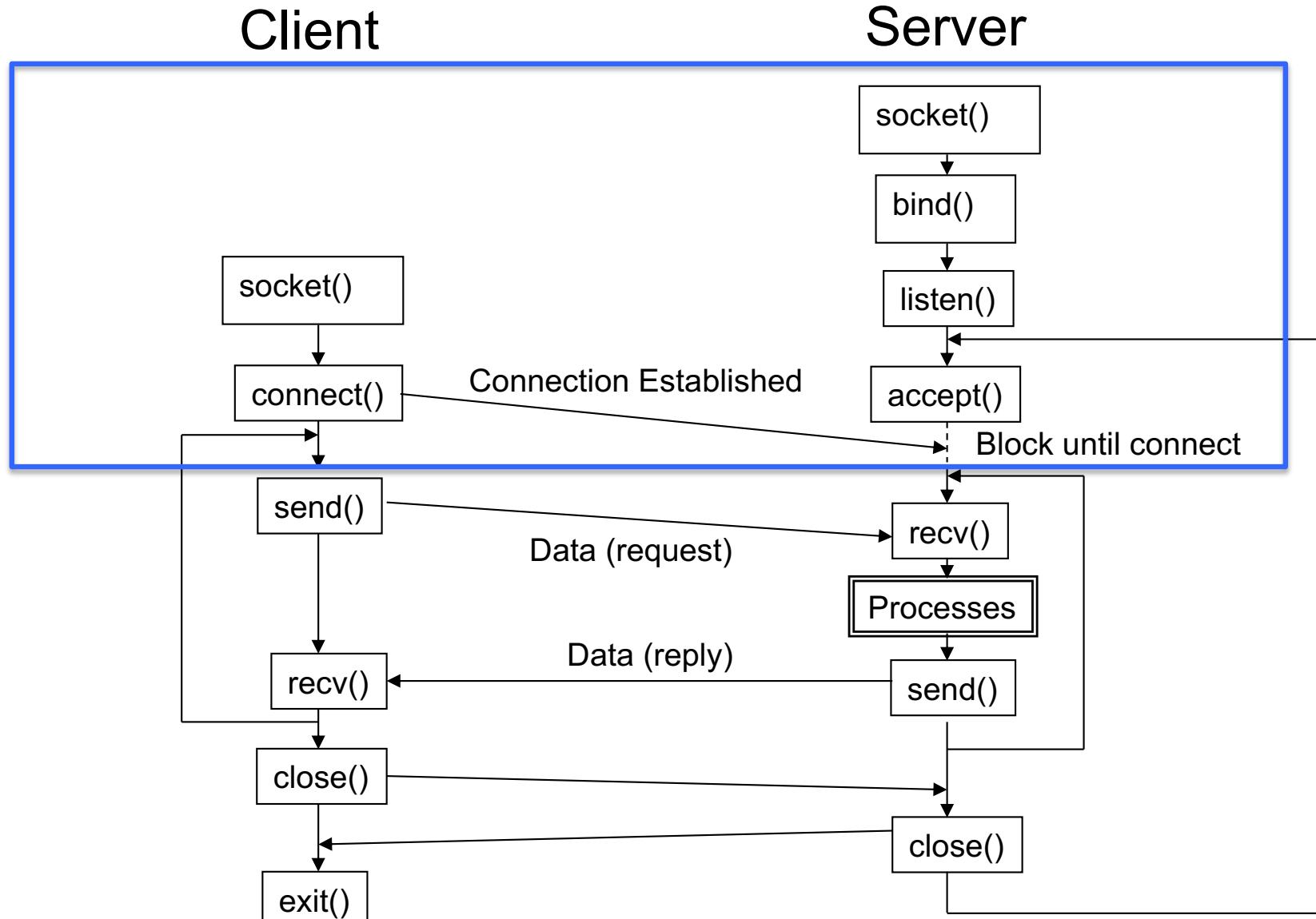
State

LISTEN

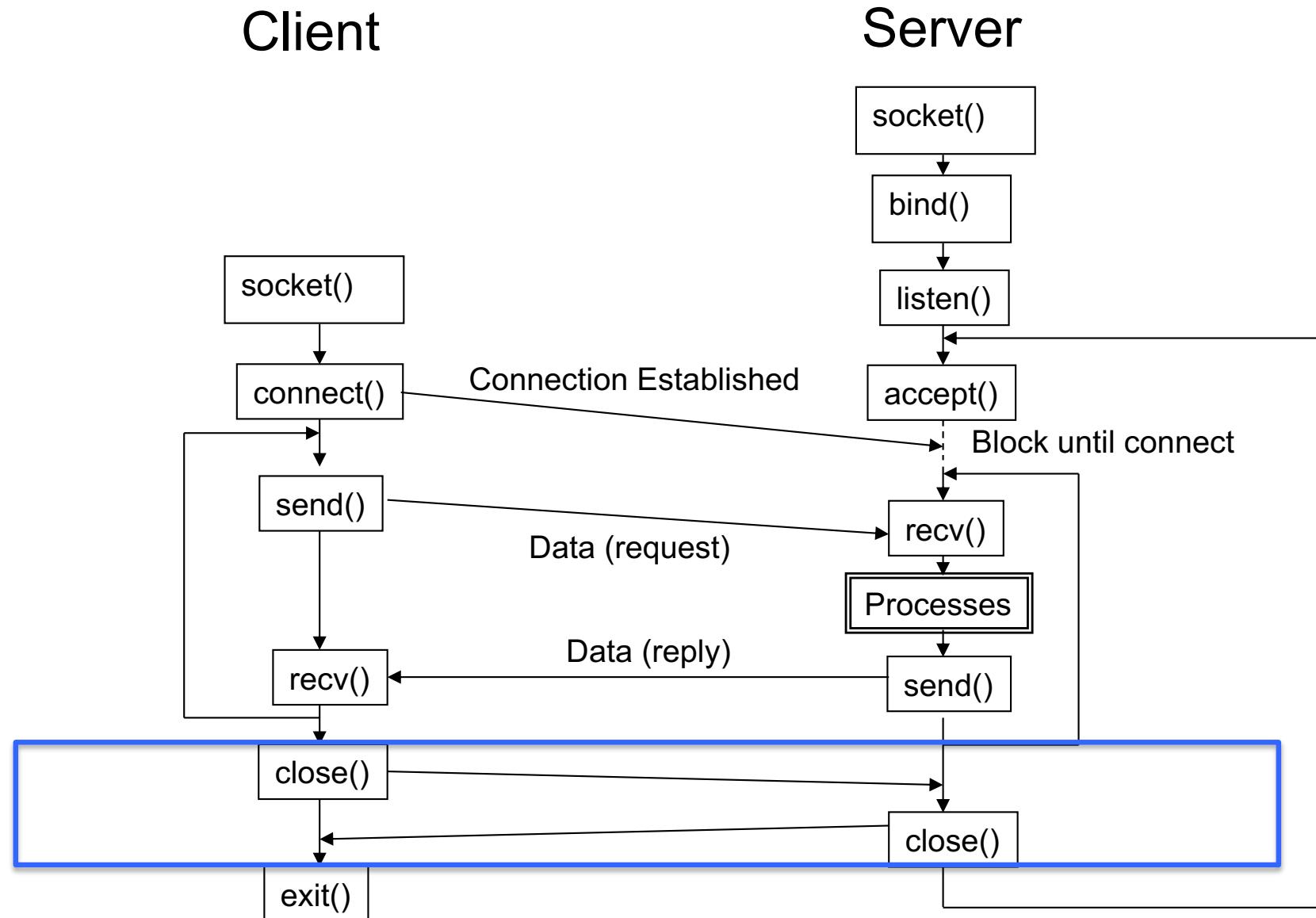
ESTABLISHED



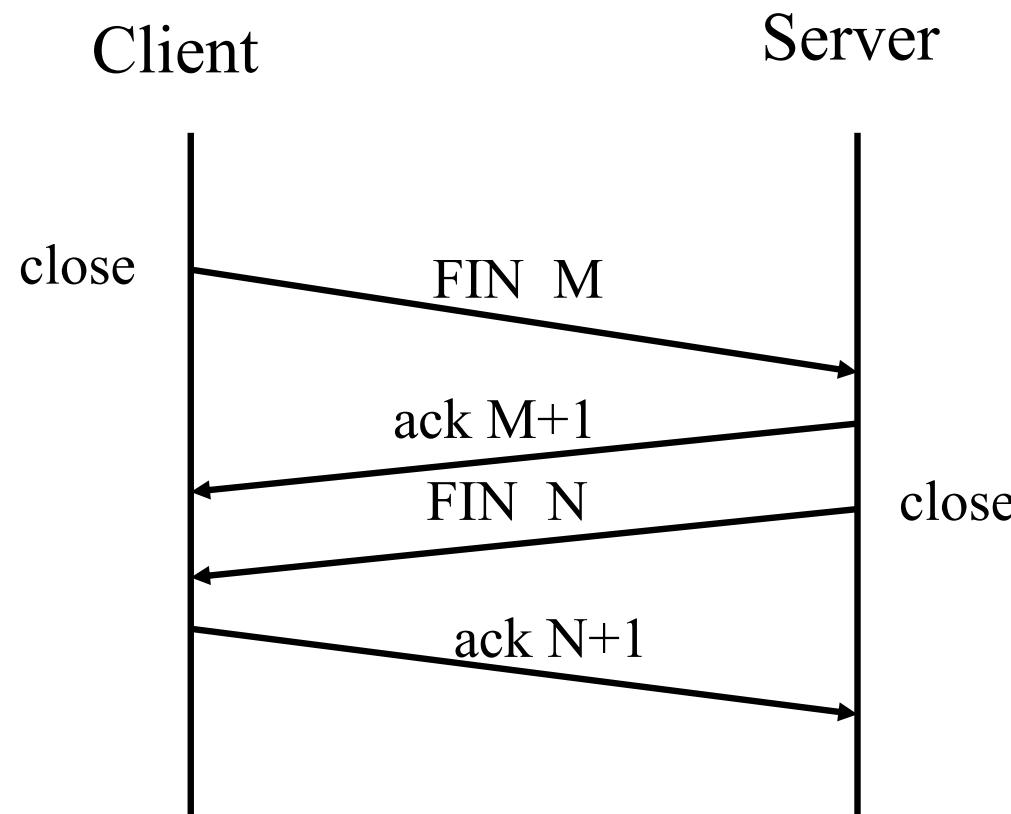
# Basic Scenario (TCP)



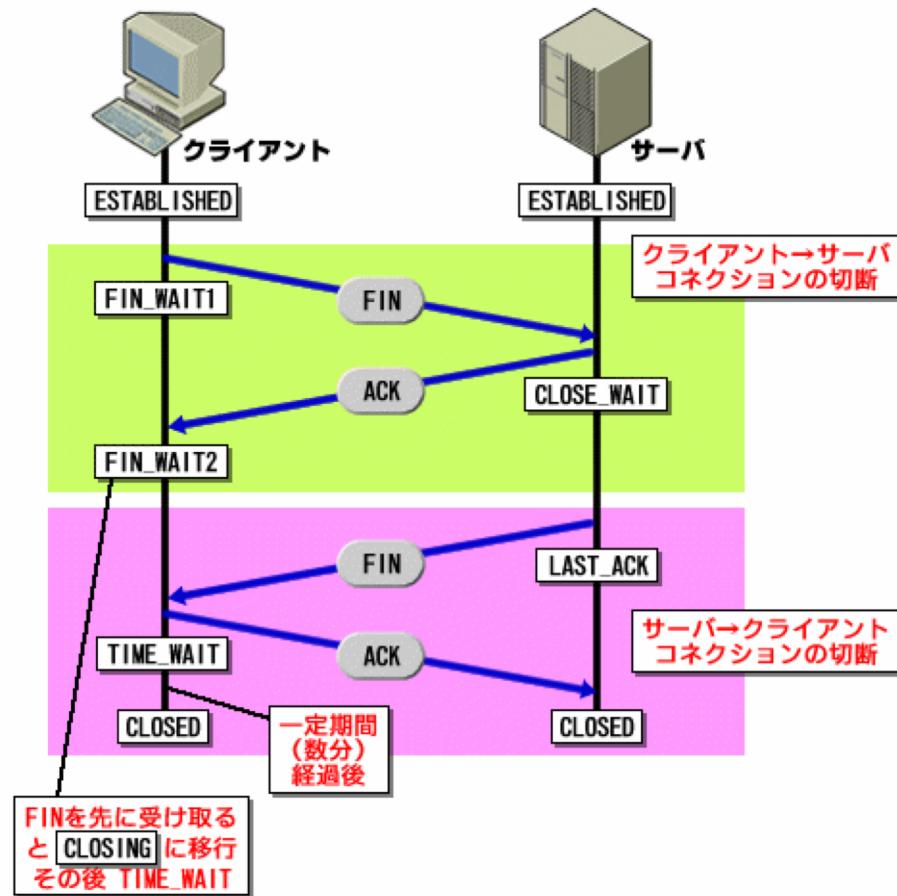
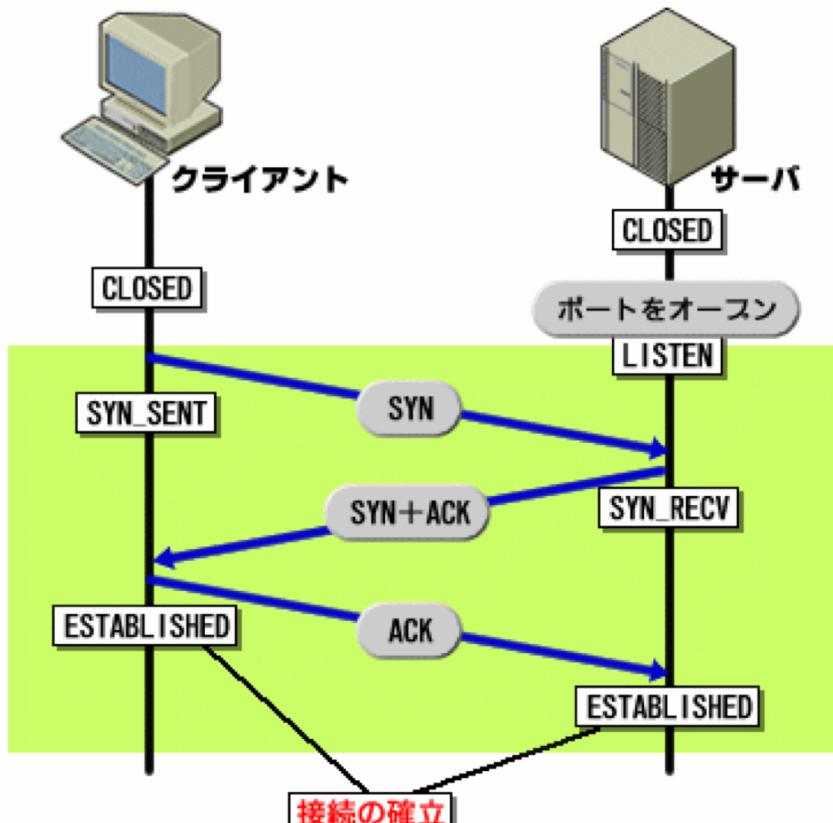
# Basic Scenario (TCP)



# Terminate TCP connection

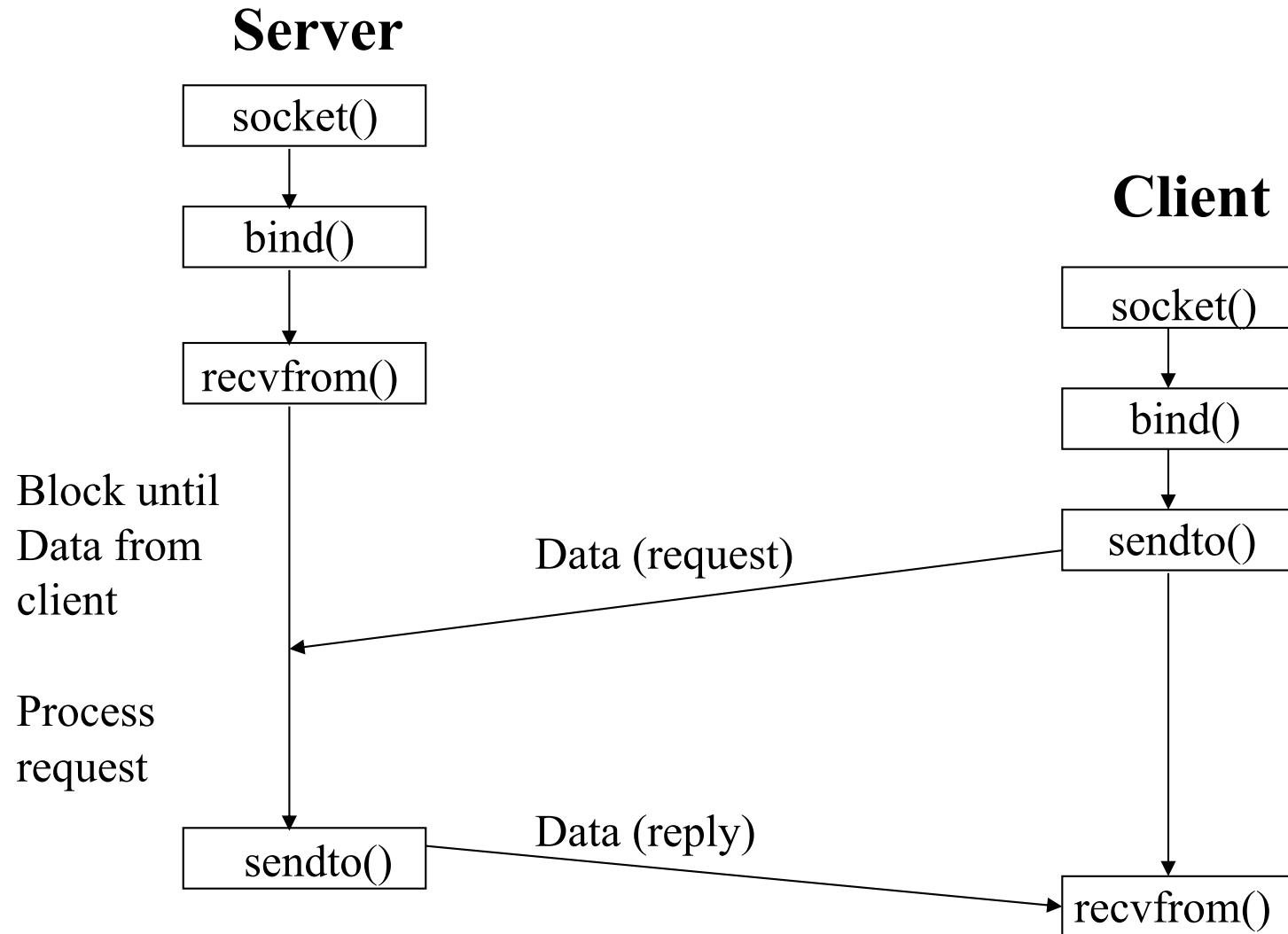


# コネクション確立と終了のまとめ

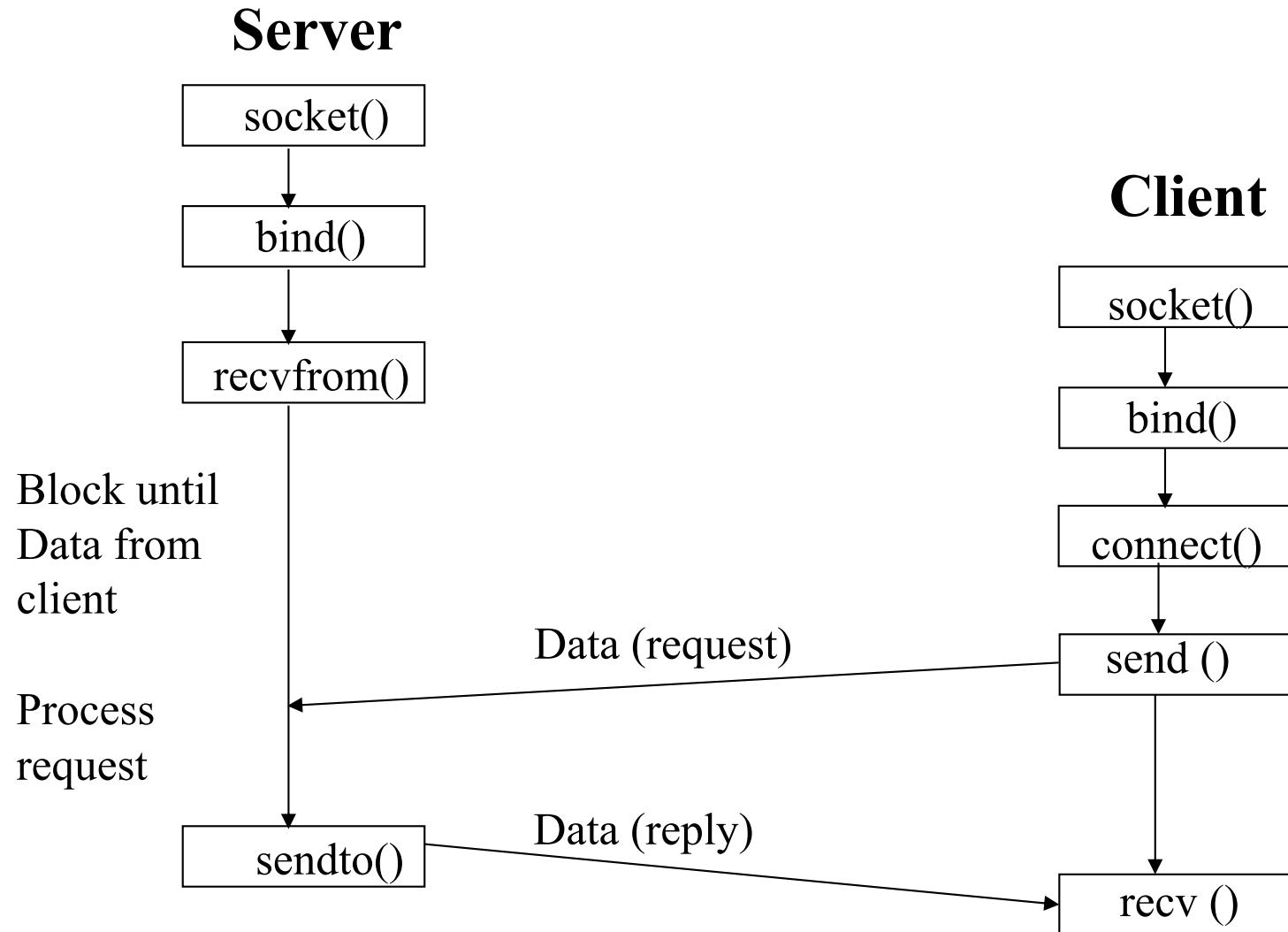


出展 : <http://www.atmarkit.co.jp/fnetwork/netcom/netcom05/netcom01.html>

# Datagram example (UDP)



# Datagram example2 (UDP)



# TCP を使ったクライアント作成

- 必要な関数
  - socket()
  - connect();
  - bind(); <- 使わなくても可
- 必要な構造体
  - sockaddr\_in
  - sockaddr\_in6
  - sockaddr\_storage
- その他の関数
  - memset();
  - htons(); ntohs();
  - inet\_pton(); inet\_ntop();
  - perror();

# socket() 関数

- int socket(int domain, int type, int protocol)
- ネットワーク通信のための出入口を作成
  - domain にはプロトコルファミリを指定
    - AF\_INET IPv4プロトコル
    - AF\_INET6 IPv6プロトコル
    - AF\_LOCAL UNIX Domain Socket
  - Typeにはソケットのタイプ(以下のどれか)
    - SOCK\_STREAM ストリームソケット (TCP)
    - SOCK\_DGRAM データグラムソケット (UDP)
    - SOCK\_RAW rawソケット
  - Protoには raw ソケット以外、通常0

# socket() 関数

- 返り値
  - 成功: ソケットディスクリプタが返る
  - 失敗: -1が返る
    - ソケットディスクリプタはファイルディスクリプタの友達
- 実際のコードでは…
  - `fd = socket(AF_INET, SOCK_STREAM, 0)`

# connect 関数

- ```
int connect( int sockfd,
              const struct sockaddr *addr,
              socklen_t addrlen);
```
- 通信先の情報を与える
- 引数
  - sockfd : ソケットディスクプリタ
  - addr : 通信先の情報が入った sockaddr\_in / sockaddr\_in6
  - addrlen : sockaddr の長さ
- 返り値
  - 成功 0
  - 失敗 -1

# sockaddr\_in / sockaddr\_in6 構造体

- sockaddr\_in

```
struct sockaddr_in
{
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;           /* Port number.
    */
    struct in_addr sin_addr;      /* Internet
    address. */
    /* Pad to size of `struct sockaddr'. */
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           __SOCKADDR_COMMON_SIZE -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
};

#define __SOCKADDR_COMMON(sa_prefix) \
    sa_family_t sa_prefix##family
```

- sockaddr\_in6

```
struct sockaddr_in6
{
    __SOCKADDR_COMMON (sin6_);
    in_port_t sin6_port;          /* Transport layer port
    #
    uint32_t sin6_flowinfo;       /* IPv6 flow information
    */
    struct in6_addr sin6_addr;    /* IPv6 address */
    uint32_t sin6_scope_id;       /* IPv6 scope-id */
};
```

# sockaddr\_storage 構造体

- ソケットの情報を一般化した形
  - ソケットを使った通信のためのテンプレート
  - 利用するプロトコルに依存しない
  - sockaddr\_in / sockaddr\_in6 の union や cast として用いられる
  - 通信時に IPv4/IPv6 がまだ決定していないが socket を作る場合に用いられる

```
struct sockaddr_storage
{
    __SOCKADDR_COMMON (ss_);
    /* Address family, etc. */
    __ss_aligntype __ss_align; /* Force desired alignment. */
    char __ss_padding[_SS_PADSIZE];
};
```

# inet\_pton 関数

- int inet\_pton(int af, const char \*src, void \*dst);
- IPv4/IPv6 のアドレスをテキスト形式から sockaddr\_in / sockaddr\_in6 に入る形式に変換
- 使用例
  - inet\_pton(AF\_INET, “133.11.206.165”, &dst.sin\_addr)
  - inet\_pton(AF\_INET6, “2001:200:180:452::3:80”, &dst.sin6\_addr);
- 返り値
  - 成功 1
  - 失敗 0, -1

# bind 関数

- ```
int bind( int sockfd,
           const struct sockaddr *addr,
           socklen_t addrlen);
```
- 通信元の情報を与える
- 引数
  - sockfd : ファイルディスクリプタ
  - addr : 通信元の情報が入った sockaddr\_in / sockaddr\_in6
  - addrlen : sockaddr の長さ
- 返り値
  - 成功 0
  - 失敗 -1

# ネットワーク・バイト・オーダ

- Network Byte Order
- CPUアーキテクチャによって、バイトの並びが違う
  - Big Endian (SPARC, PowerPC等)
  - Little Endian (Intel等)
- ネットワーク上に流すバイト順は統一が必要
- htons()/htonl()/ ntohs()/ntohl()を利用

# エラー処理

- perror() 関数
  - システムコールやライブラリ関数の呼び出しにおいて、最後に発生した エラーに関する説明メッセージを生成し、標準エラー出力に出力する
- 使用例

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>
int main() {
    int sock;
    sock = socket(1000, 1000, 1000);
    if (sock < 0) {
        perror("socket");
        return 0;
    }
    return 1;
}

% gcc -o sock_error sock_error.c
% ./sock_error
socket: Invalid argument
```

## コマンドライン引数:argc,argv

- コマンドの引数を用いるには?
  - cat -n file
  - cat file1 file2 file3

# コマンドライン引数の利用法

- main( int argc, char \*\*argv)
- main( int argc, char \*argv[])
  
- argc には引数の数
- argv[0] にはコマンド名
- argv[1] には1番目の引数
- argv[2] には2番目の引数

# コマンドライン引数 サンプルプログラム

```
#include<stdio.h>

int main(int argc, char *argv[]){
    int i;
    for(i = 0; i < argc; i++){
        printf("arg[%d]: %s\n", i, argv[i]);
    }
}

% gcc -o arg_example arg_example.c
% ./arg_example ABC DEF 123
arg[0]: ./arg_example
arg[1]: ABC
arg[2]: DEF
arg[3]: 123
```

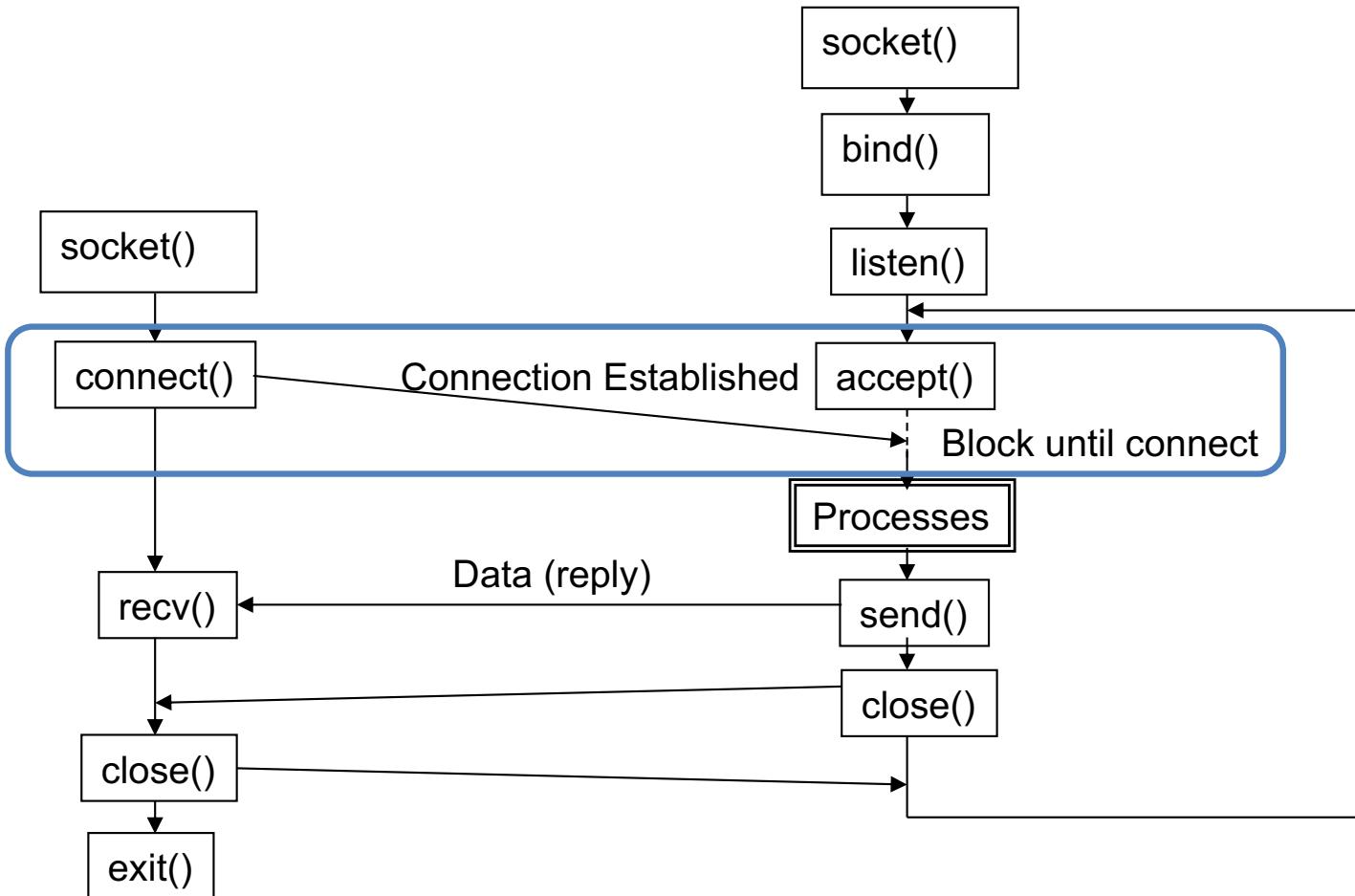
# サンプルプログラムと実行結果

- daytime プロトコル
  - 現在の時刻をテキスト形式で返すサービス
  - TCP port 13番
- テスト用サーバ (login2.sekiya-lab.info)
  - IPv4 : 54.249.36.78
  - IPv6 : 2406:da14:941:3c03::20
- 実行例

```
% gcc -o tcp_client tcp_client.c  
% ./tcp_client  
Tue Apr 24 09:21:32 2018
```

# Daytime における TCP 通信の流れ

Client



Server

# daytime クライアントサンプルプログラム（抜粋）

```
deststr  = "130.69.250.51" ;
destport = 13;

sock = socket(AF_INET, SOCK_STREAM, 0);
server.sin_family = AF_INET;
server.sin_port = htons(destport);
s = inet_pton(AF_INET, deststr, &server.sin_addr);

connect(sock, (struct sockaddr *)&server,
        sizeof(server));

memset(buf, 0, sizeof(buf));
n = read(sock, buf, sizeof(buf));
printf("%s\n", buf);
```

# 課題 1

# 課題 1

- サンプルプログラムを参考に daytime クライアントを作成
  - 引数でアドレスを指定できるようにする
  - エラー処理をする
  - IPv4 だけではなく IPv6 にも対応する
- プログラム環境
  - login1.sekiya-lab.info
    - 13.115.209.199 / 2406:da14:941:3c03::10
- テスト用 daytime サーバ
  - login2.sekiya-lab.info
    - 54.249.36.78 / 2406:da14:941:3c03::20

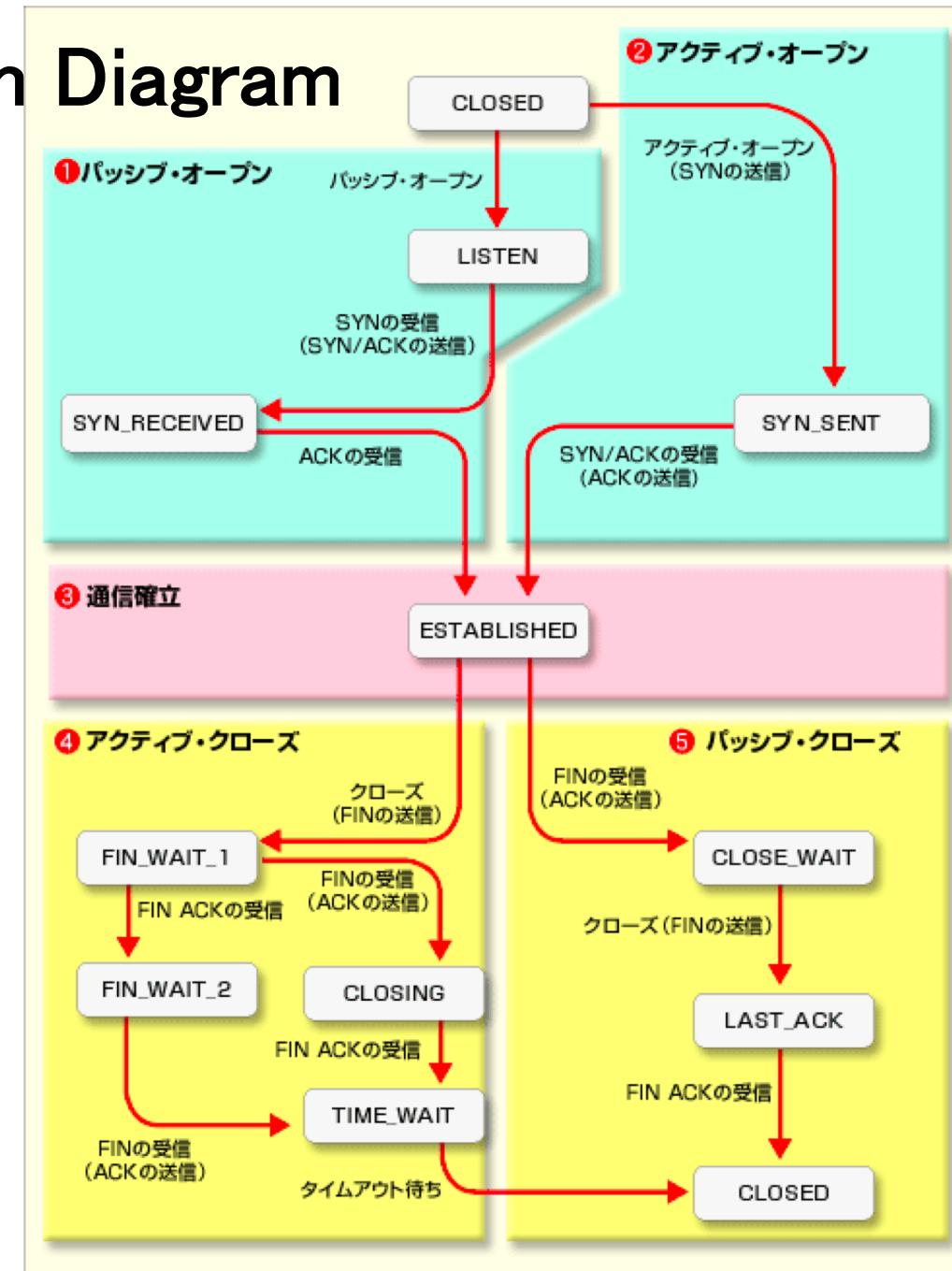
# 課題 1

- 締め切り : 2018/05/9 (水) 23:59 JST
- 提出方法 :
  - 以下の情報をメールにて [lecture@sekiya-lab.info](mailto:lecture@sekiya-lab.info) まで提出
    - Subject : 課題1提出
    - 学籍番号
    - 氏名
    - 作成したクライアントのプログラムリスト
    - 作成したクライアントの実行結果
  - 本文貼り付けでもレポートとして PDF ファイルでもどちらでも構いません

# TCP を用いたサーバプログラミング

# TCP State Transition Diagram (簡易版)

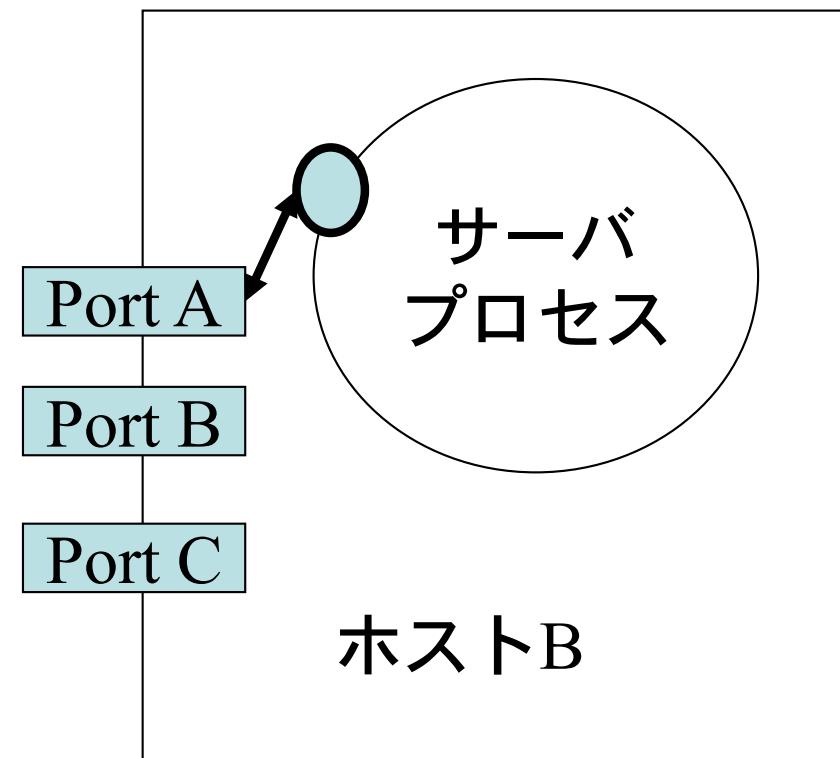
- パッシブオープン
- アクティブオープン
- アクティブクローズ
- パッシブクローズ



# bind()

Proto	Local Address
tcp	0.0.0.0A

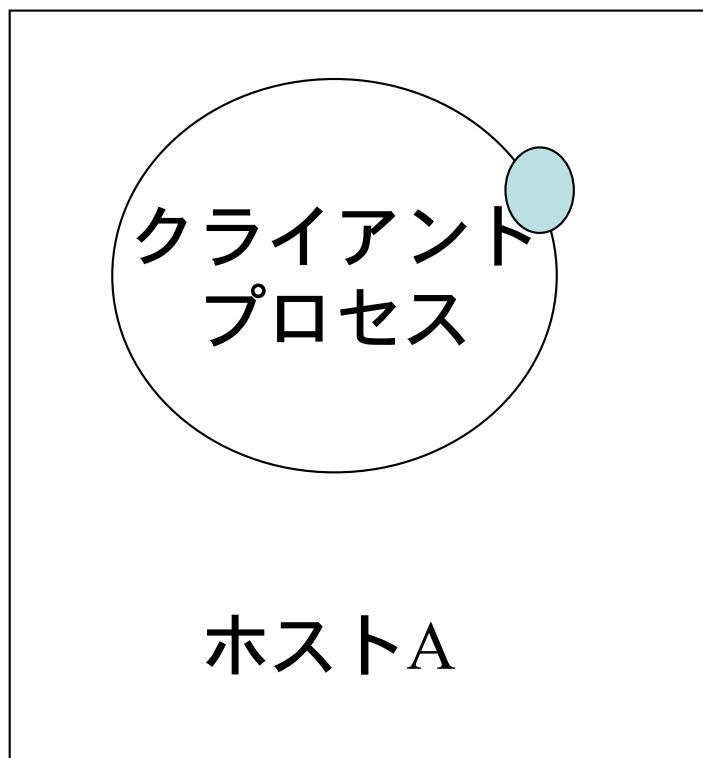
Foreign Address	State
0.0.0.0*	CLOSED



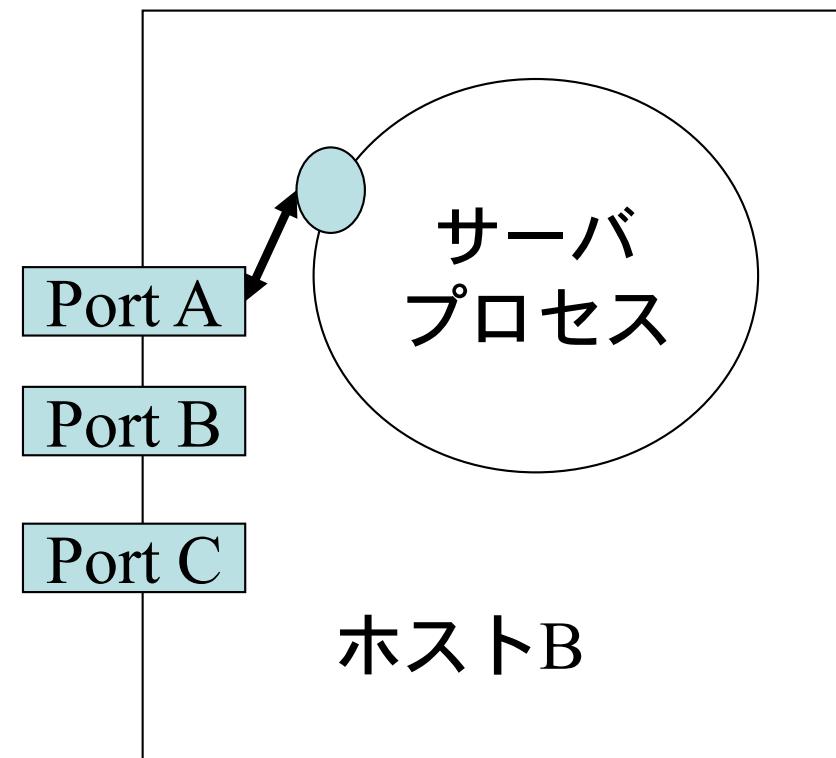
# Listen()

Proto Local Address  
tcp 0.0.0.0A

Foreign Address 0.0.0.0\*  
State LISTEN



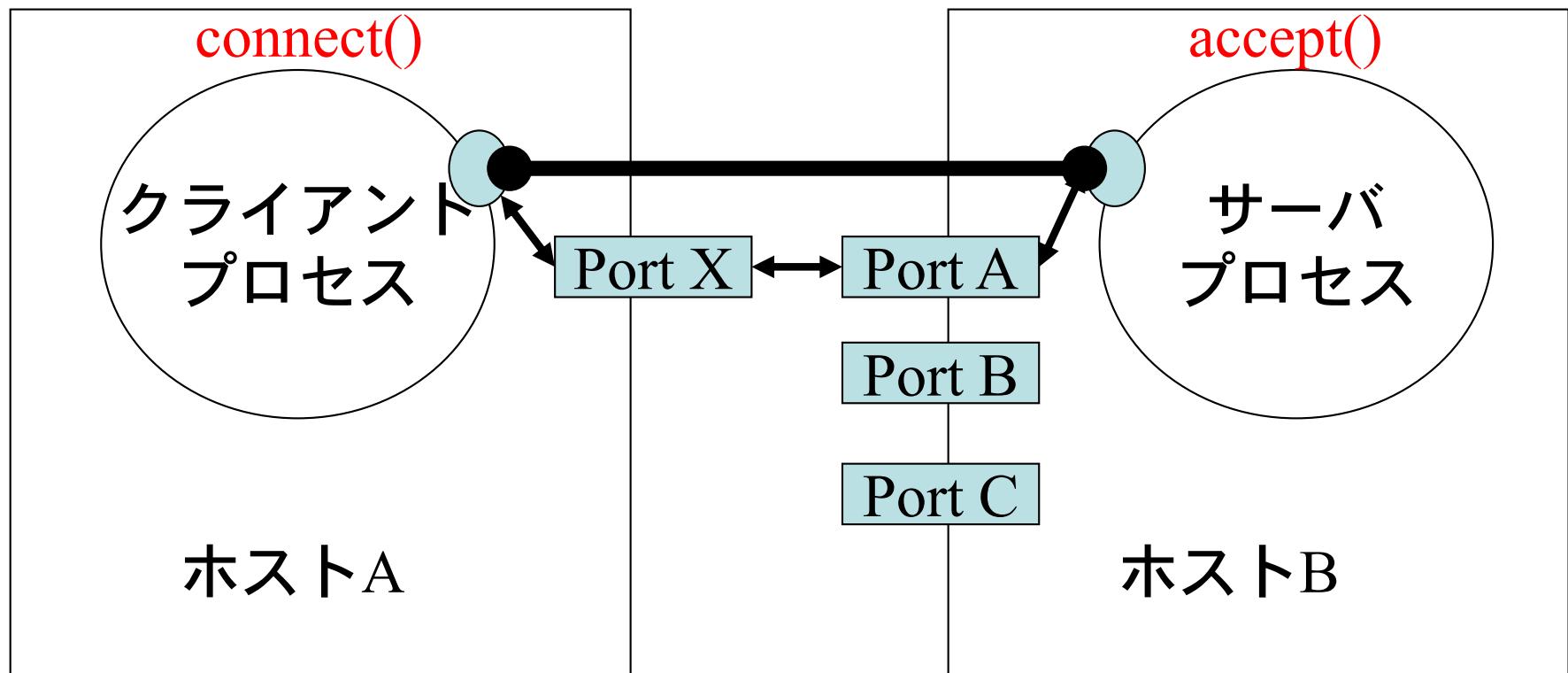
IP Address: 133.11.205.162



IP Address: 133.11.205.161

# connect() -> accept()

Proto	Local Address	Foreign Address	State
tcp	133.11.205.161.A	133.11.205.162.X	ESTABLISHED



IP Address: 133.11.205.162

IP Address: 133.11.205.161

# bind 関数

- ```
int bind( int sockfd,
           const struct sockaddr *addr,
           socklen_t addrlen);
```
- 通信元の情報を与える
- 引数
  - sockfd : ソケットディスクリプタ
  - addr : 通信元の情報が入った sockaddr\_in / sockaddr\_in6
  - addrlen : sockaddr の長さ
- 返り値
  - 成功 0
  - 失敗 -1

# listen() 関数

- int listen(int sockfd, int backlog)
- 用意した socket を待ち受け準備状態にする
  - ソケットの状態を CLOSED から LISTEN へ
- backlog は connect に来たクライアントの待機数
  - accept()されるまで backlog キューに保持
  - accept() した順に、順次処理
  - キューがあふれると、その要求は無視
- 返り値
  - 成功 0
  - エラー -1

# accept() 関数

- `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`
  - listen キューから接続要求を取り出して、クライアントと通信を開始 => 新しいソケットディスクリプタを作成
  - 新しいソケットディスクリプタを利用してサーバ側での通信処理が始まる
- 実際のコードでは…
  - `cli = accept( sockfd, (struct sockaddr *)&cliaddr, &clilent)`
- 返り値
  - 成功 新しいソケットディスクリプタ
  - エラー -1

# write() 関数

- `ssize_t write(int sockfd, const void *buf, size_t nbyte)`
  - ソケットディスクリプタ(ファイルディスクリプタ)に対してデータ列を書き込む
- 返り値
  - 成功 書き込んだデータサイズ
  - 失敗 -1

# サーバサンプルプログラム (抜粋 - エラー処理なし)

```
sock = socket(AF_INET, SOCK_STREAM, 0);
addr.sin_family = AF_INET;
addr.sin_port = htons(12345);
addr.sin_addr.s_addr = INADDR_ANY;

bind(sock, (struct sockaddr *)&addr, sizeof(addr));
listen(sock, 5);

while (1) {
    len = sizeof(client);
    cli = accept(sock, (struct sockaddr *)&client, &len);
    write(cli, "HELLO\n", 7);
    close(cli);
}
```

# 接続してきたクライアントの確認

- `cli = accept(sock, (struct sockaddr *)&client, &len);`
- ここで新しいソケットディスクリプタ (cli) が返される
- 同時に、`sockaddr_in client` に情報が入る
- `sockaddr_in client` 構造体に、接続してきたクライアント(相手)に関する情報が格納される
  - サーバは、つないできたクライアントに関する情報を得ることができる

# sockaddr\_in 構造体

```
struct sockaddr_in
{
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;                      /* Port number. */
    struct in_addr sin_addr;                  /* Internet address. */

    /* Pad to size of `struct sockaddr'. */
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           __SOCKADDR_COMMON_SIZE -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
};

#define __SOCKADDR_COMMON(sa_prefix) \
    sa_family_t sa_prefix##family
```

# サーバサンプルプログラム (抜粋 - エラー処理なし)

```
while (1) {
    len = sizeof(client);
    cli = accept(sock, (struct sockaddr *)&client, &len);

    printf("accepted connection from %s, port=%d\n",
           (char *)inet_ntop(AF_INET, &client.sin_addr,
                           &cli_addr, sizeof(cli_addr)),
           ntohs(client.sin_port));

    write(cli, "HELLO\n", 7);
    close(cli);
}
```

# 実行例

- クライアント

```
sekiya% telnet -4 lecture-server.nc.u-tokyo.ac.jp 12345
Trying 133.11.205.161...
Connected to lecture-server.nc.u-tokyo.ac.jp.
Escape character is '^]'.
HELLO
Connection closed by foreign host.
```

- サーバ

```
sekiya@LECTURE-CLIENT:~/EXAMPLE$ ./tcp_server
accepted connection from 130.69.251.130, port=49381
```