# ソケットプログラミング

# ソケット API

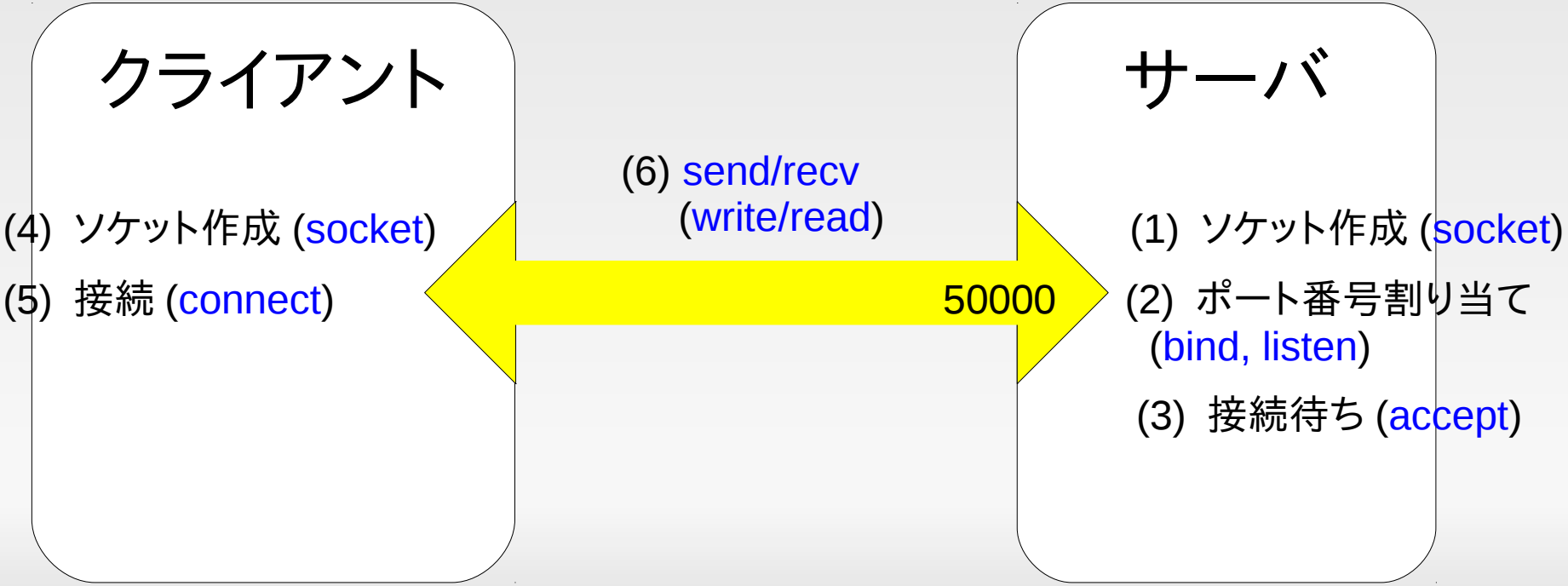
- プロセス間通信の汎用 API
  - プロセス:プログラムのひとつの単位
    ex) ”./a.out” とかやると1つのプロセスが立ち上がる
- ソケット API
  - IPv4
  - IPv6
  - UNIX domain (UNIX 計算機内プロセス間通信 )
- 本実験では IPv4 の TCP および UDP を , ソケット API を通じて行う

# クライアントとサーバ

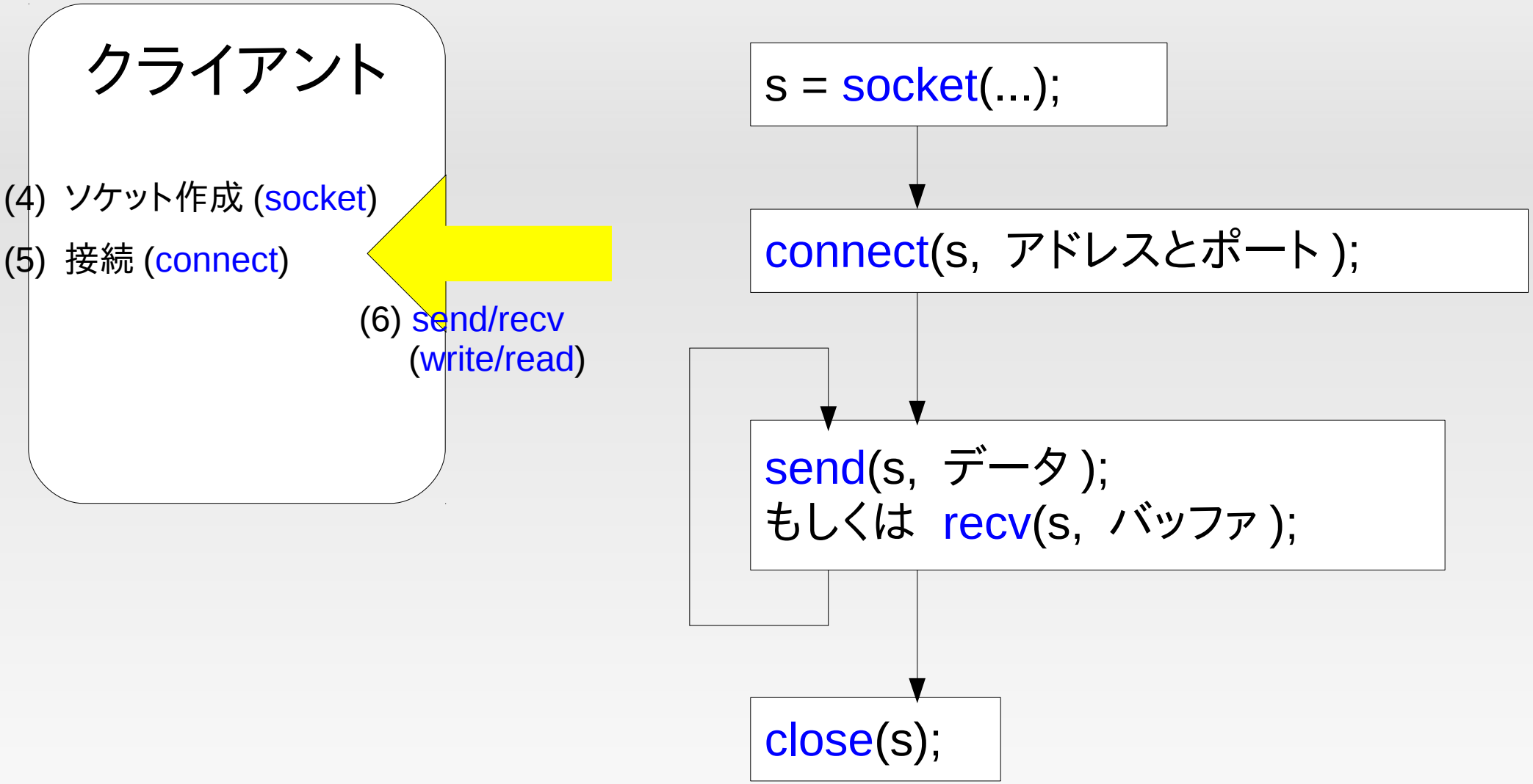
- 電話を用いた比喩
  - サーバ ≈ 電話を待ち受ける人
  - クライアント ≈ 電話をかける人
- 両者では通信開始までの手順が若干異なる

# ソケット API を用いた TCP による通信手順

- ソケット = 接続の「端点」≈ 電話器
- プログラム上はソケット ≈ ファイルディスクリプタ



# TCP クライアント API 概要



# しつこく...

- API を呼び出したら成功を確認すること
- 特にネットワークでは「エラーが日常」
- 詳しくは manual 参照

socket_error.c

```
1  s = socket(...);
2  if (s == -1) {
3      perror(”socket”);
4      exit(1);
5  }
6  if (connect(s, ...) == -1) {
7      perror(”connect”);
8      exit(1);
9  }
```

# ネットワークとファイルの類似

- 実際 UNIX では , send の代わりに write, recv の代わりに read を使っても良い ( ソケットはファイルディスクリプタの一種 )

| | | |
|---|---|---|
| 作成 | open | socket |
| 接続 | N/A | connect |
| 書き込む | write | send |
| 読み込む | read | recv |
| 片付け | close | close |

# socket

- socket( 通信体系の種類 , ソケットの種類 , プロトコル );
- 通信体系の種類 :
  - 我々は「 IPv4 」⇒ PF_INET
- ソケットの種類 :
  - UDP ⇒ SOCK_DGRAM または
  - TCP ⇒ SOCK_STREAM
- プロトコル : 0

# close の挙動に関する注意

- close(s); には二つの効果がある
  - 「もう送りません」宣言 ⇒ 相手が (close 以前に送られたデータをすべて受け取った後 ) end of file (0 バイト ) を受け取る
  - 「もう受けとりません」⇒ 自分がデータを受け取ろうとしてもエラーになる
- しばしば「もう送りません」といいつつまだデータは受け取りたいことがある⇒ shutdown(s, SHUT_WR);

# connect

- 概念的には ,
  connect(s, IP アドレスとポート );
- しかし「 IP アドレスとポート」を用いるのは IP 通信の場合のみ
- 異なる通信体系 ( したがってアドレスの表現も異なる ) もサポートするため , API は回りくどい

# 具体的には ...

connect_with_error.c

```
1  struct sockaddr_in addr;
2  addr.sin_family = AF_INET;
3  if (inet_aton(“192.168.1.100”, &addr.sin_addr) == 0) {
4    perror(“inet_aton”); exit(1);
5  };
6  addr.sin_port = htons(50000);
7  if (connect(s, (struct sockaddr *)&addr, sizeof(addr)) == -1) {
8    perror(“connect”); exit(1);
9  }
```

- とてもややこしい。(引数が多い、使う関数が多い、など)
- 同種の「回りくどさ」はソケット API 全般の問題 ( 性質 )

# なぜこんなに面倒？

- socket は IPv4 以外の通信 ( したがってアドレス ) をサポートしていることから派生する問題
  - sin_family でそれを明示
  - IPv4 アドレス用構造体 (sockaddr_in) と汎用アドレス用構造体 (sockaddr)
  - それにともなうキャスト ( 強制的な型のごまかし )
  - 構造体のサイズも渡さないといけない
- IP アドレスを文字列ではなく 32 bit 整数にする
- ポート (16bit) を「ネットワークバイト順」にする

connect_with_error.c

```
1  struct sockaddr_in addr;
2  addr.sin_family = AF_INET;
3  if (inet_aton(“192.168.1.100”, &addr.sin_addr) == 0) {
4    perror(“inet_aton”); exit(1);
5  };
6  addr.sin_port = htons(50000);
7  if (connect(s, (struct sockaddr *)&addr, sizeof(addr)) == -1) {
8    perror(“connect”); exit(1);
9  }
```

# 関連マニュアル

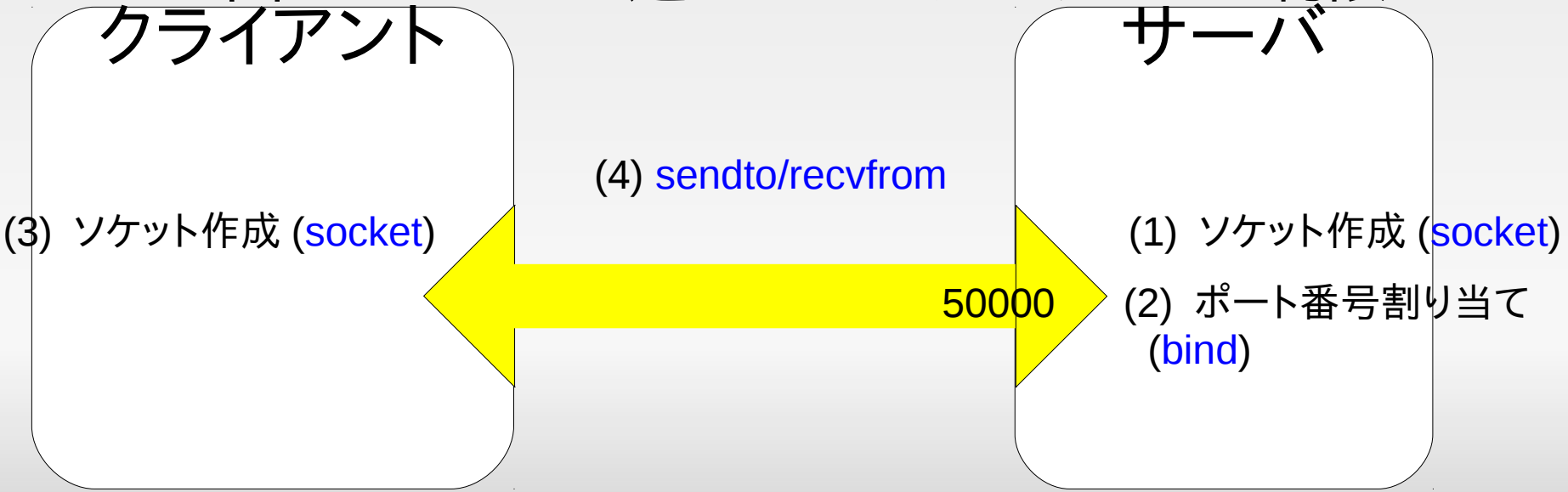
- man 7 ip
- man 7 tcp
- man 7 udp
- 落とし穴 : man socket, man connect,  などでは IPv4  固有の情報 , TCP, UDP 固有の情報が出てこない
- 理由 :  さっきと同じ (socket API は IPv4 だけの API ではない )
- 本棚の書籍も参照

# send/recv に関する注意

- 要求したバイト数 { 受け取れる･送れる } とは限らない
  - recv(s, buf, 1000000, 0);  で 1000000 バイト必ず受け取れるわけではない
  - 「何バイト受け取れたのか」は返り値でわかる
  - send も同様
  - 参考 : read/write も同様だった
- N バイト ( もしくは接続が切れるかエラーになるまで ) きっちり { 送る･受け取る } 関数を書いてみよ

# ソケット API を用いた UDP による通信手順

- TCP との API 上の違い :
  - connect/accept/listen が不要 ( 比喩 : 電話 vs 手紙 )
  - close に意味はない
  - send の代わりに sendto で毎回宛先を指定
  - recv の代わりに recvfrom で送信元を取得できる
  - 1 回の sendto で送れるデータのサイズに制限がある



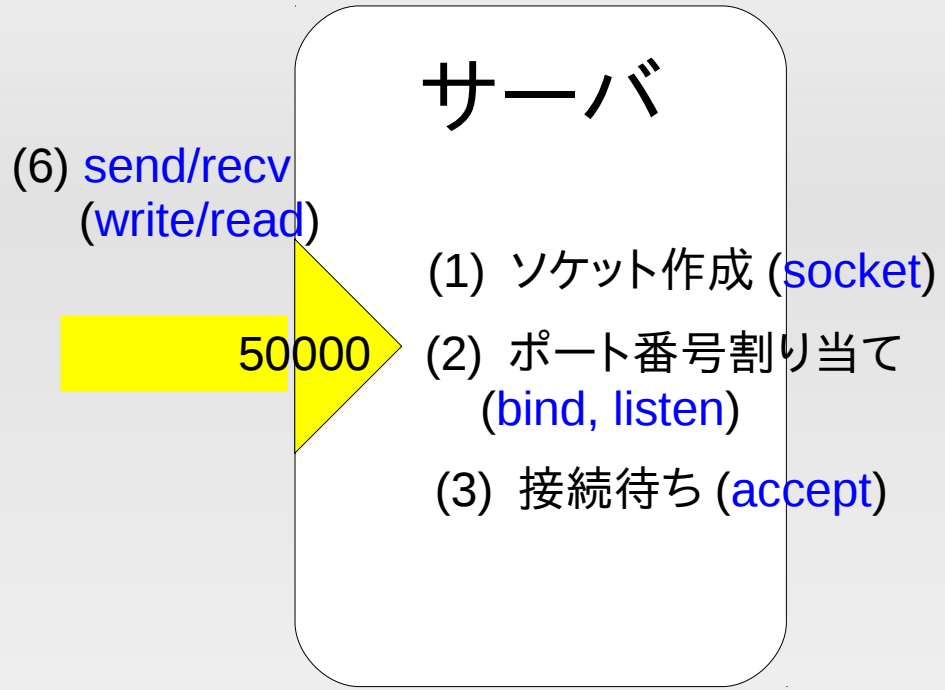
# UDP

- 一見 API の種類が少なくて簡単そうだがそうとは限らない
  - メッセージが到着しない可能性がある
  - 通信開始･終了のプロトコルは自分で作る必要がある
  - いつになったらメッセージを送り始めて良いの？
  - いつになったら終了して良いの？「これが最後のメッセージ」みたいなデータを明示的に送る . Close しても何も起きない

# TCP vs UDP ( よくある勘違い )

- ( 嘘ではないがざっくりすぎる理解 ) TCP は信頼性を保証するために大きなオーバーヘッドを払っている . だから遅い
- ( 大きな勘違い ) 自分の作った電話ではなぜか 1-2 秒音が遅れてやってくる . これは TCP が遅いせい
- 自作の pingpong プログラムで TCP でのメッセージの往復がどのくらいの時間であったか測ったはず . それを踏まえて考えること

# TCP サーバ API




```
ss = socket(...);
bind(ss, アドレスとポート );
listen(ss, queue 長 );
s = accept(ss, ...);
send(s, データ );
もしくは recv(s, バッファ );
close(s);
```

# bind

- bind(ss, IP アドレス + ポート );
- 最終的に待ち受ける (connect の目標となる )IP アドレス , ポート番号を宣言する
- 引数は ,connect と似た状況で ,sockaddr* 型の引数に sockaddr_in* を渡す
- 「どの IP アドレスで connect を受け付けるか」も指定可能だが多くの場合 IPADDR_ANY( どのアドレスでも受け付ける ) を指定すれば足りる

# Bind でありがちなエラー： Address already in use

- 注 : もちろん perror で表示されるので , 心がけはいつもと同じ
- 意味 : そのポートはすでに使われている
- 理由 :
  - 実際に他のプロセスが使用中の可能性もある
  - が , おそらく ,「さっきまで自分のプログラムが使っていた」( しばらくは同じポートを再利用できない )

# ポート番号の再利用

- OS はあるポートを使っているソケットが close された後 , 数分間そのポート番号を再利用不可とする
- 理由 : すぐに再利用してしまうと , 以前の接続のためのパケットが混入してくる可能性がある

# 安全な ( 空いている ) ポート番号の割り当て

- bind をポート番号 =0 で呼び出す
  - 実際のポート番号 0 を使うのではなく「適当な空きポート番号」が割り当てられる
- 残る問題 : どうやって割り当てられたポートを知るか？
  - getsockname(ss, …)
  - ... は sockaddr* 型の引数 . いつも通り実際に渡すのは ,sockaddr_in*

# Listen

- listen(ss, qlen);
- qlen の意味は , 未処理の connect 要求をいくつまで (OS が ) 蓄えるか ( それ以上になったらクライアントに即座にエラーを返す )
- この実験ではさして重要ではない (10 程度にしておけば十分 )

# Accept

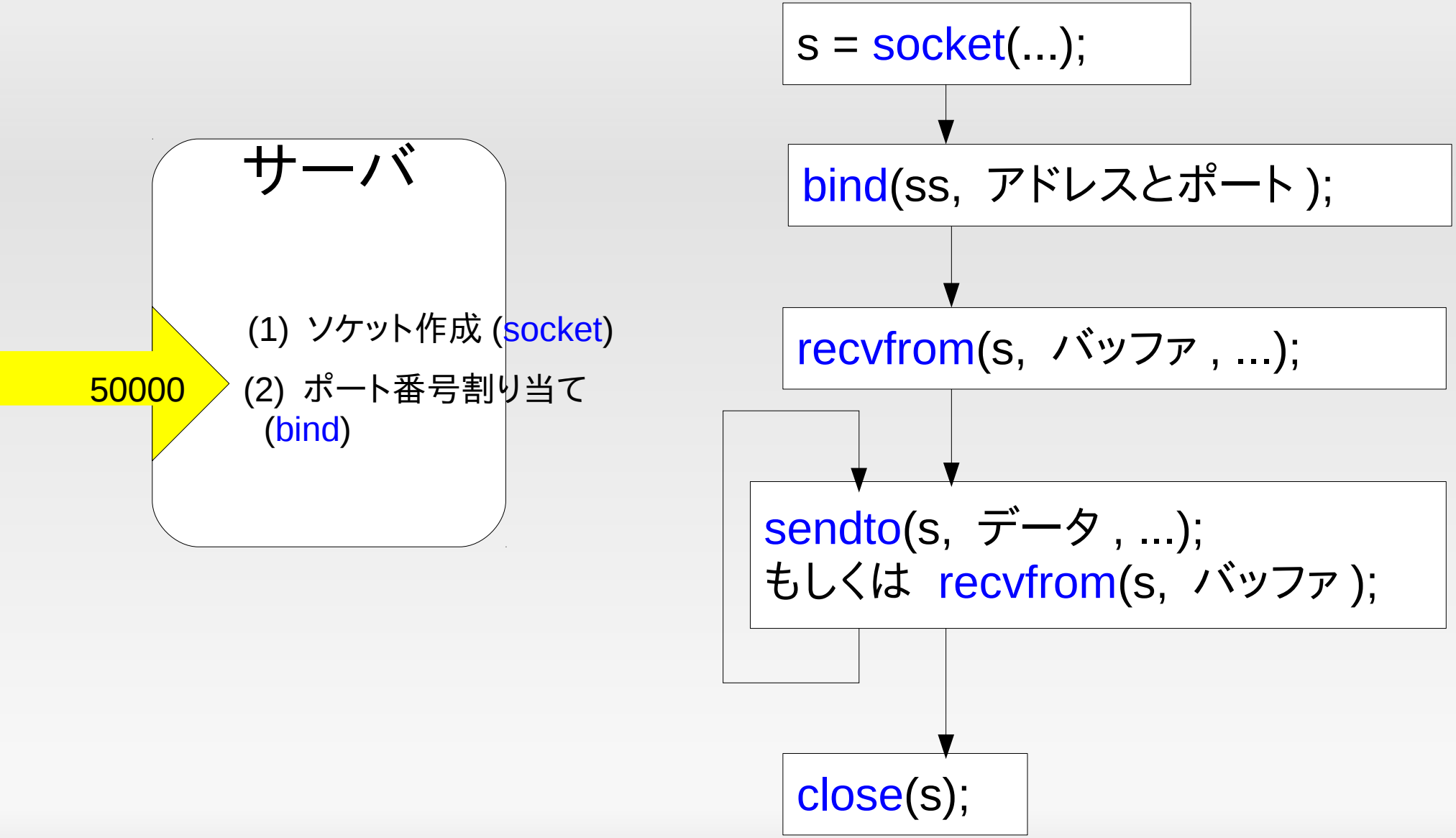
- cs = accept(ss, ...);
- クライアントからの connect を待つ
- 成功したら「新しいソケットを返す」
- 注意：クライアントと通信するのはこの新しいソケット．元々の ss で通信するのではないので間違えないように
- ... に , 接続してきたクライアントの IP アドレスとポートが返ってくる ( 興味がなければ NULL でも可 )
  - 引数は connect と似ているがさらにややこしい

# accept の引数

```
1  sockaddr_in addr;
2  socklen_t len = sizeof(addr);
3  cs = accept(ss, (struct sockaddr *)&addr, &len);
```

- 第 2 引数 &addr の役割
  - addr に，接続してきたクライアントのアドレスを入れてもらう
- 第 3 引数 & len の役割
  - addr に受け入れ可能サイズを教える (2 行目 )
  - len に , 接続してきたクライアントのアドレスのサイズを入れてもらう
- UDP の recvfrom も似たパターン

# UDP サーバ API


サーバ
(1) ソケット作成 (socket)
50000
(2) ポート番号割り当て
(bind)
s = socket(...);
bind(ss, アドレスとポート );
recvfrom(s, バッファ , ...);
sendto(s, データ , ...);
もしくは recvfrom(s, バッファ );
close(s);

# N バイト「確実に」受け取るループ

- エラーが発生するか，相手が接続を切るか，N バイト受け取るまでループする

```c
int recv_all(int s, char *buf, int len) {
  int received = 0;
  while (received < len) {
    int n = recv(s, buf + received, len - received, 0);
    if (n == -1) {
      perror("recv");
      exit(1);
    } else if (n == 0) {
      fprintf(stderr, "EOF!!\n");
      return received;
    }
    received += n;
  }
  return received;
}
```

# send も同様に

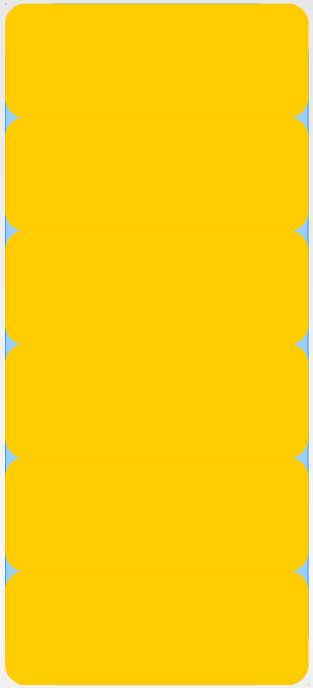
- N バイト確実に送る関数を書いてみよ

# sox を使う上での注意 (8.1 → 8.2)

- rec/play ではパイプを使ってデータをやり取りする

```
$ rec -t raw - | ./serv_send 50000
```

```
1  write(1, ..., ...);
```



パイプのバッファ

```
1  s = accept(ss, ...);
2  while(1) {
3    int read_size = read(0, buffer, N);
4    write(s, buffer, read_size);
5  }
```

# 理解の助け

- ソケット API は汎用的な「プロセス間通信」の API を意図したもの
- IPv4 以外の通信体系も ( 少しパラメータを変えて ) ほぼ同じ API で用いることができるように設計されている
- ⇒API がややこしく見える
  - パラメータが多い , 回りくどい
  - パラメータの型が不自然

- 以下は connect ( やこれから出てくる多数のソケット関連 API) がなぜこんな汚いパラメータの渡し方になっているのかの詳細説明
  - 「ともかくこうすればいい」と教科書丸呑みする分には必ずしも必要ないが
  - C 言語でよく使われる「手口」として理解しておくことは有用
- 「場合によってパラメータの型 ( 種類 ) が異なるような API をどう設計するか」という問題

# 「場合に応じて異なる種類のパラメータ」を受け取る汎用 API の形

- 例題：異なる種類の「図形」がある
  - 三角形
  - 円
- 「図形の面積」を求める汎用 API を作りたい
  - area(...);
  - 三角形でも円でも機能するようにしたい

# 三角形と円 ( 素直な定義 )

- typedef struct triangle {
  double px, py, qx, qy, rx, ry;
  } triangle;
- typedef struct circle {
  double cx, cy, r;
  } circle;

# 面積

- area(f);
- 直面する問題 : f の型を何にしたらいい？
- 「 triangle または circle 」などという器用な型は書けない

# 解決法

- area のパラメータの型は何かへの「ポインタ」とする ( 何でもよい . 意図を表すために figure*)
  - double area(figure * f);
- area を呼び出す方も triangle/circle の「ポインタ ( アドレス ) 」を渡す
  - triangle t;

    …

    area(&t);　　/*　注 : figure* ← triangle* */
  - circle c;

    …

    area(&c);　/*　注 : figure* ← circle* */

# どちらを受け取ったか分かるようにする (データのタグ付け)

- typedef struct figure {
  int kind; /* triangle: 0, circle 1 */
  } figure;
- typedef struct triangle {
  int kind; /* 0 */
  double px, py, ...;
  } triangle;
- typedef struct circle {
  int kind; /* 1 */
  double cx, cy, r;
  } circle;

# area の中身 ( タグによる場合分け )

- area(figure * f) {
  if (f->kind == 0) {
  triangle * t = f; /* 注 : triangle* ← figure* */
  ...;
  } else {
  circle * c = f; /* 注 : circle* ← figure* */
  ...;
  }
  }

# コンパイラ警告の消し方

- 異なるポインタ型間で代入やパラメータ渡しをしているところで警告が出る
  - エラーにならないところがポイント
- コンパイラを説得する : キャスト
  - ( 型 ) 式
  - 「式」の本来の型を無視して「型」だと思う
- area(&c) → area((figure *)&c);
- circle * c = f; → circle * c = (circle *)f;

# ( 本題に戻り )connect の引数

- IP アドレス + ポートを表す構造体 ：sockaddr_in (≈ triangle や circle に相当 )
- すべての通信体系のための , 汎用的なアドレス構造体 ：sockaddr (≈ figure 相当 )
- テンプレート (connect 以外にも似た場面あり )
  - struct sockaddr_in a;
    a.sin_family = AF_INET; /* kind 相当 */
    a.sin_addr.s_addr = IP アドレス ;
    a.sin_port = ポート ;
    connect(s, (sockaddr*)&a, ...);

# 結局何が問題で , 何が解決だったのか？

- C 言語の表面的には ,
- 問題： 変数 ( 関数のパラメータ ) の型を一つに決めなくてはならない (⇒ 故に複数の型を受け取る関数は作れないように見える )
- 解決： 実は引数の型がポインタ (xxx*) であれば , どんなポインタを代入 ( 渡 ) してもエラーではない ( 警告で済む )
  - 「 A* ← B* 」は「一応合法」
  - さらに , キャストをすれば警告もでない

# 「ポインタ」でないといけないのか？

- 素朴な疑問： 要するに変数の型が違っても OK ってこと？ じゃ，以下はダメなの？
- area(figure f) { … }
  circle c;
  …
  area(c);
  /* または */
  area((figure)c);
- 答え： ダメ ( エラーになる )

# なぜポインタは OK でポインタじゃないと NG なのか？

- つまらない答え： それが C 言語の仕様だから
- もう少し「納得できる」答え：
  - C 言語の仕組みを想像する
  - 実は「ポインタ = アドレス」で ,A* であろうが B* であろうがその表現型式は同じ (= アドレス )
  - ⇒ A* も B* も保持できる変数を作ることに何の苦労もいらない
  - ポインタでない場合 , そのサイズおよび種類 ( 特に , 浮動小数点数であるか否か ) によって変数用に確保すべきバイト数やレジスタの種類が異なる
  - ⇒ A も B も保持できる変数を作るのは面倒

# 注 1

- ここで示した問題「多様な種類のデータに同じ API を適用したい」はよく現れる問題
- 問題の根源に見える , 「変数の型を決めて , 異なる種類の代入が行われないようにする」のは , プログラムの間違いを検出するためにも重要
  - C 言語の解決策 : 安全でない「抜け道」を用意 ( ポインタ型は型が違っていても代入できる )
  - より最近の言語の解決策 : クラスとその継承 , 型パラメータ (C++ テンプレートなど )

# 注 2

- C 言語で同じ事をやるもう少し「教科書的」方法は union を使うこと
- typedef struct figure {
  int kind; /* 0 : circle, 1 : triangle */
  union {
    circle c;
    triangle t;
  } f;
} figure;
- あとから種類 ( 例 :rectangle) を追加するときに figure を修正できるならこれで OK

# 関連してヤになる話

- ソケットが「 IP に限らない」汎用 API であるせいで ,
  - man socket
  - man connect
- etc. では IPv4 に固有の情報 (sockaddr_in など ) は得られない

# IPv4 固有の API 情報の得方

- 答え 1:  本実験の範囲内ではほぼ教科書にある
- 答え 2: man 7 ip, man 7 tcp, man 7 udp などで必要な情報は ( 不親切だが ) 得られる
- 答え 3:  本  TCP/IP ソケットプログラミング

# さらなる注意点

- IP アドレス： 文字列ではなく ,32bit の表現に変換
  - × a.sin_addr.s_addr = ”133.11.238.11”;
  - ○ a.sin_addr.s_addr = inet_addr(”133.11.238.11”);
  - ○ inet_aton(”133.11.238.11”, &a.sin_addr);
- ポート番号： ネットワークバイトオーダで表現された 16 bit 整数 (short)
  - × a.sin_port = 50000;
  - ○ a.sin_port = hton(50000);

# bind

- bind(ss, IP アドレス + ポート );
- 最終的に待ち受ける (connect の目標となる )IP アドレス , ポート番号を宣言する
- 引数は ,connect と似た状況で ,sockaddr* 型の引数に sockaddr_in* を渡す
- 「どの IP アドレスで connect を受け付けるか」も指定可能だが多くの場合 IPADDR_ANY( どのアドレスでも受け付ける ) を指定すれば足りる

# Bind でありがちなエラー： Address already in use

- 注 : もちろん perror で表示されるので , 心がけはいつもと同じ
- 意味： そのポートはすでに使われている
- 理由 :
  - 実際に他のプロセスが使用中の可能性もある
  - が , おそらく ,「さっきまで自分のプログラムが使っていた」 ( しばらくは同じポートを再利用できない )

# ポート番号の再利用

- OS はあるポートを使っているソケットが close された後 , 数分間そのポート番号を再利用不可とする
- 理由 : すぐに再利用してしまうと , 以前の接続のためのパケットが混入してくる可能性がある
- 現在使用可能なポートを OS に割り当ててもらう方法は後述

# Listen

- listen(ss, qlen);
- qlen の意味は , 未処理の connect 要求をいくつまで (OS が ) 蓄えるか ( それ以上になったらクライアントに即座にエラーを返す )
- この実験ではさして重要ではない (10 程度にしておけば十分 )

# Accept

- cs = accept(ss, ...);
- クライアントからの connect を待つ
- 成功したら「新しいソケットを返す」
- 注意：クライアントと通信するのはこの新しいソケット．元々の ss で通信するのではないので間違えないように
- ... に , 接続してきたクライアントの IP アドレスとポートが返ってくる ( 興味がなければ NULL でも可 )
  - 引数は connect と似ているがさらにややこしい

# accept の引数

- sockaddr_in addr;
  socklen_t len = sizeof(addr);
  cs = accept(ss, (struct sockaddr *)&addr, &len);
- 第 2 引数 &addr の役割
  - addr に , 接続してきたクライアントのアドレスを入れてもらう
- 第 3 引数 & len の役割
  - addr に受け入れ可能サイズを教える (2 行目 )
  - len に , 接続してきたクライアントのアドレスのサイズを入れてもらう
- UDP の recvfrom も似たパターン

# 空きポート番号の割り当て

- bind をポート番号 =0 で呼び出す
  - 実際のポート番号 0 を使うのではなく「適当な空きポート番号」が割り当てられる
- 残る問題 : どうやって割り当てられたポートを知るか？
  - getsockname(ss, …)
  - ... は sockaddr* 型の引数 . いつも通り実際に渡すのは ,sockaddr_in*

# 空きポート番号の割り当て

- bind をポート番号 =0 で呼び出す
  - 実際のポート番号 0 を使うのではなく「適当な空きポート番号」が割り当てられる
- 残る問題 : どうやって割り当てられたポートを知るか？
  - getsockname(ss, …)
  - ... は sockaddr* 型の引数 . いつも通り実際に渡すのは ,sockaddr_in*

# 空きポート番号の割り当て

- bind をポート番号 =0 で呼び出す
  - 実際のポート番号 0 を使うのではなく「適当な空きポート番号」が割り当てられる
- 残る問題 : どうやって割り当てられたポートを知るか？
  - getsockname(ss, …)
  - ... は sockaddr* 型の引数 . いつも通り実際に渡すのは ,sockaddr_in*

# 空きポート番号の割り当て

- bind をポート番号 =0 で呼び出す
  - 実際のポート番号 0 を使うのではなく「適当な空きポート番号」が割り当てられる
- 残る問題 : どうやって割り当てられたポートを知るか？
  - getsockname(ss, …)
  - ... は sockaddr* 型の引数 . いつも通り実際に渡すのは ,sockaddr_in*