Networking

CS351 : Saelee

previously: **intra**-system IPC

networking = **inter**-system IPC

Client-Server Model

can switch roles ...

basic properties:

- asymmetric setup
- asynchronous
- concurrent

concurrency:

- client/server
- multiple clients
- intra-server / intra-client

need “public” server address for setup (client *connects*)

two connection models:

- circuit switched (e.g., phone)
- packet switched (e.g., Internet)

a.k.a.,

- connection-oriented
- connection/*ess*

Internet Protocol (IP) defines mechanisms for ***addressing*** & ***routing*** of packets

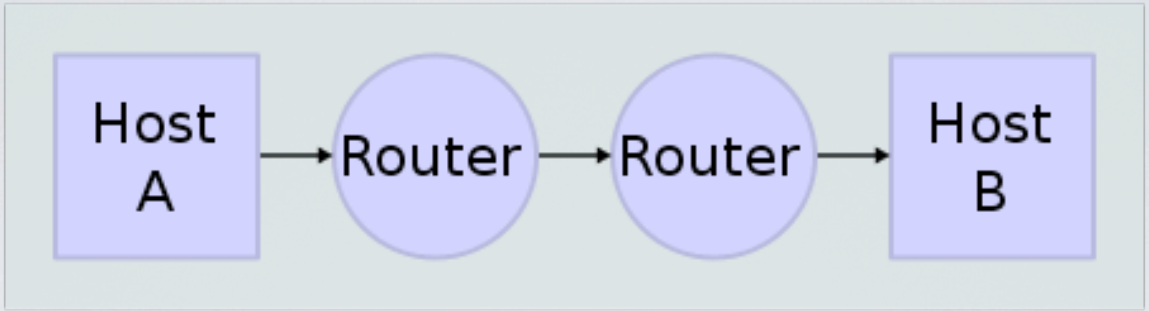
IP address → machine

“port” number → application

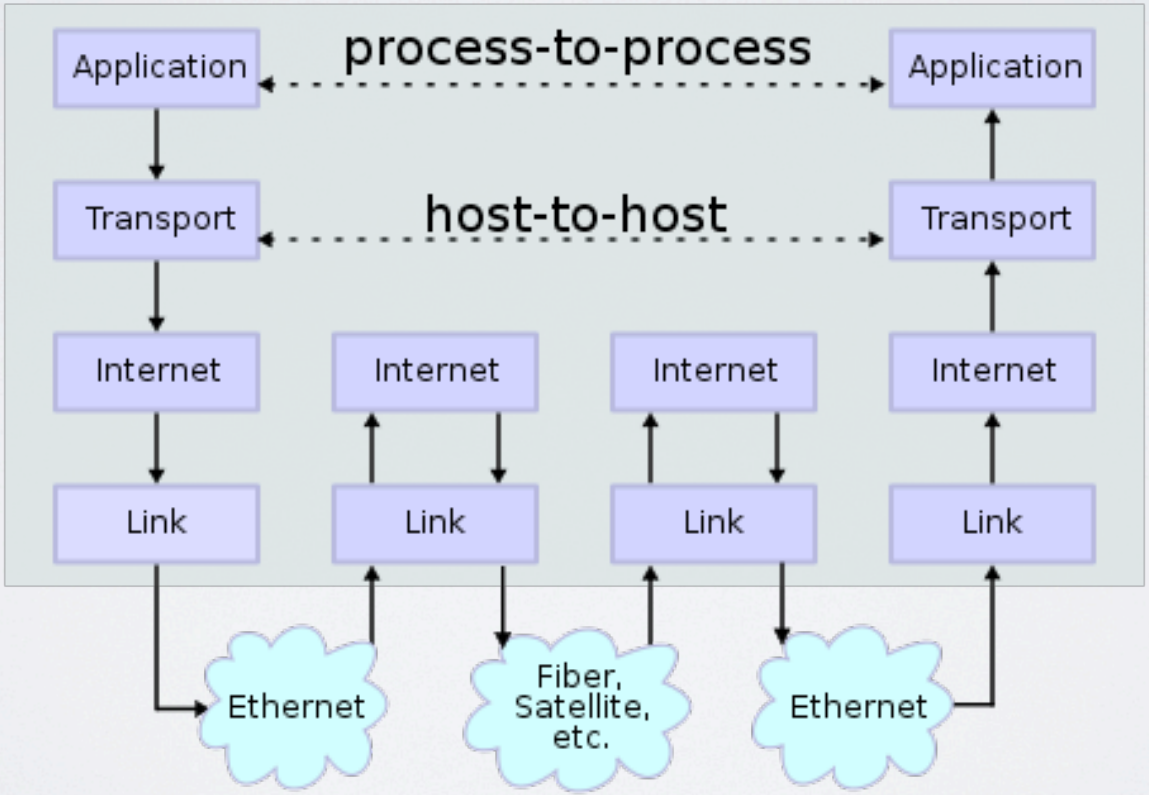
addr :: port = **endpoint**

network stack provides abstractions that hide routing and connection details

Network Topology

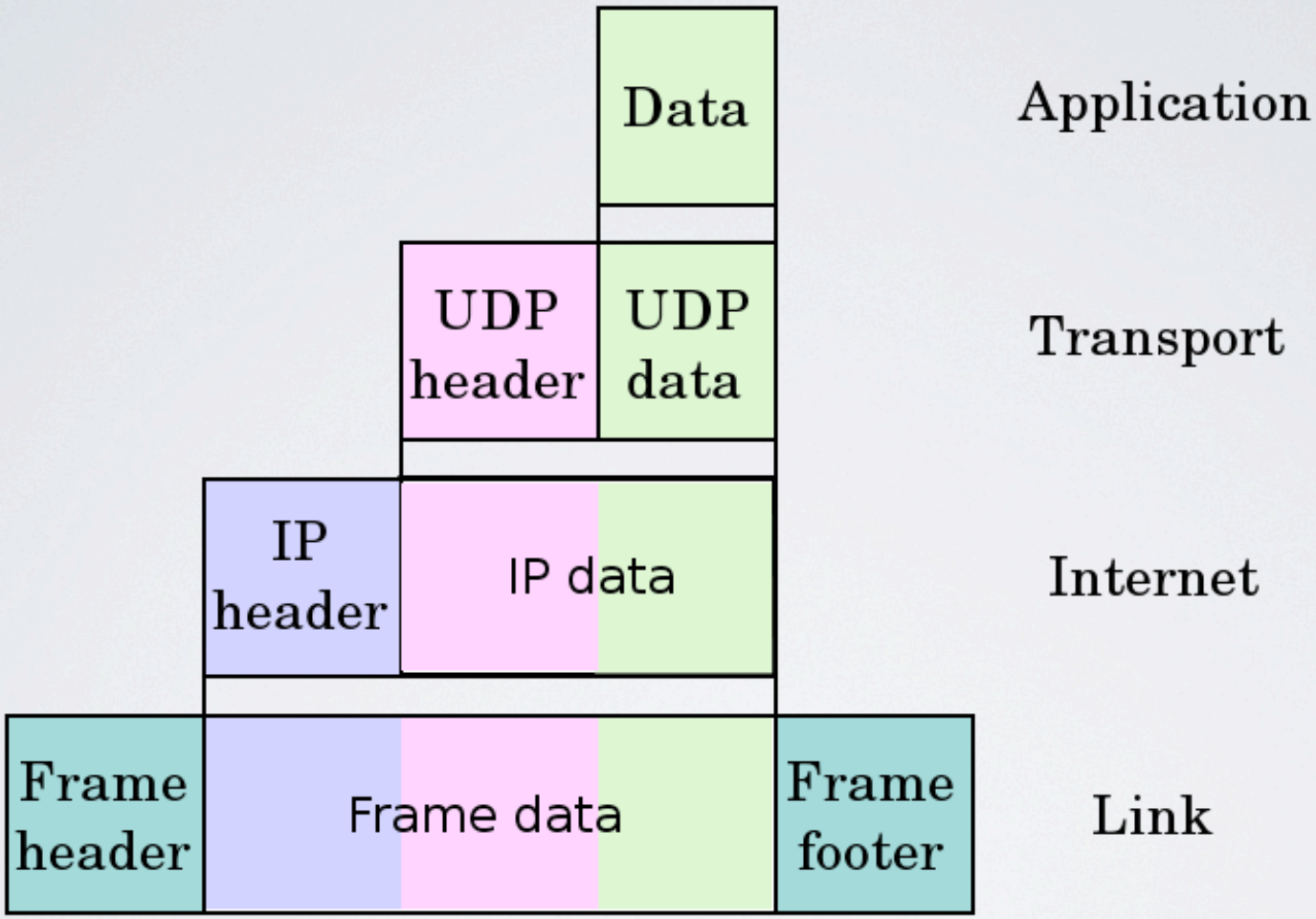


Data Flow

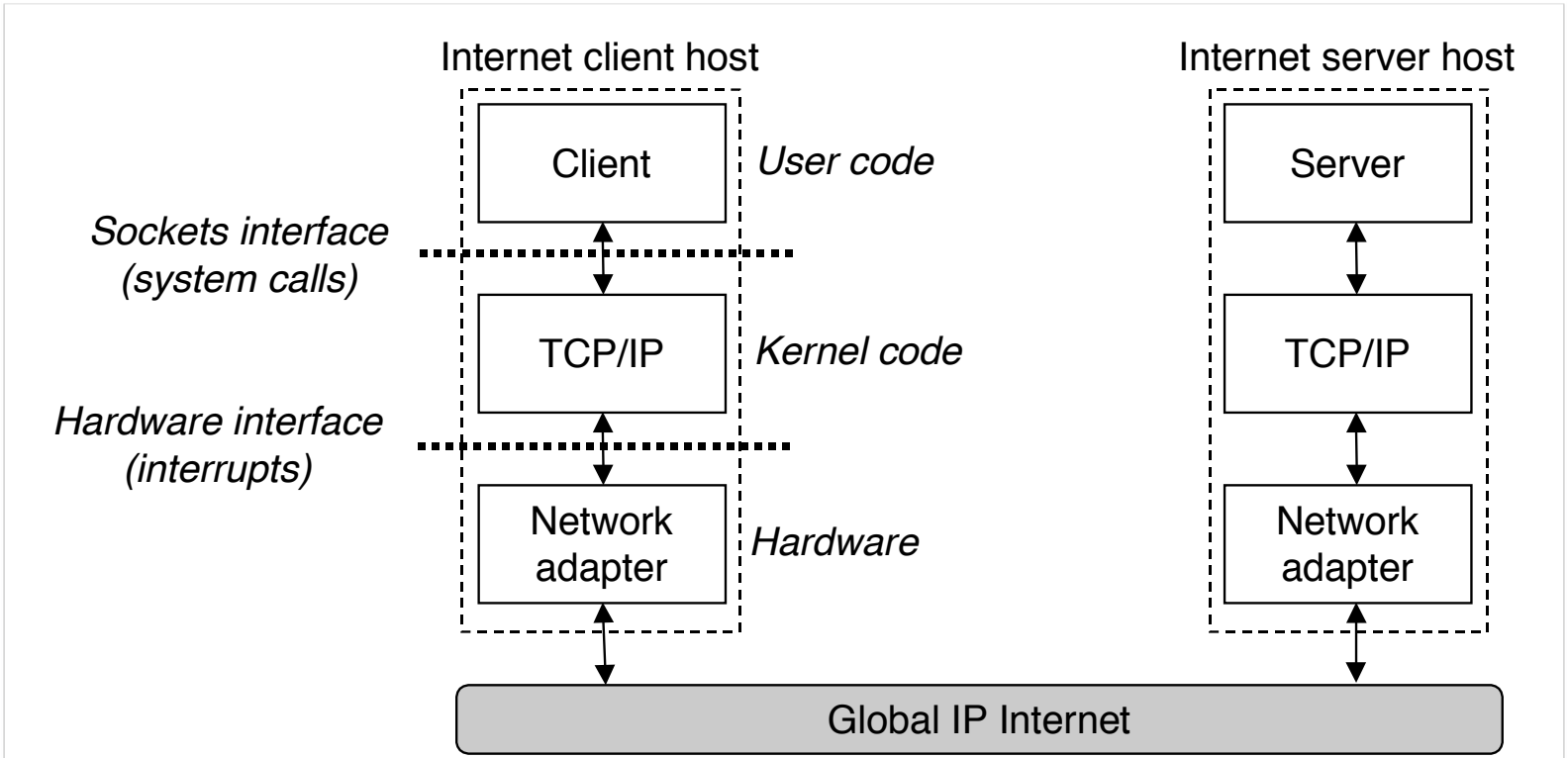


(courtesy Wikimedia Commons)

Transmission Control Protocol (TCP) builds a *connection-oriented*, *reliable* connection on top of IP



(courtesy Wikimedia Commons)


Internet client host
Internet server host
Client
User code
Server
Sockets interface
(system calls)
TCP/IP
Kernel code
TCP/IP
Hardware interface
(interrupts)
Network
adapter
Hardware
Network
adapter
Global IP Internet

goal: *read/write* the network

(… as a file)

I/O mechanism: **FD**

overview:

- obtain FD
- (configure FD)
- read/write

“sockets”

Berkeley/BSD Sockets API

de facto networking API

asymmetrical setup

server:

- get FD: `socket`
- assign name: `bind`
- open for requests: `listen`
- accept a request: `accept`

client:

- get FD: `socket`
- initiate request: `connect`

server first

```c
int socket (
    int domain,    /* domain (AF_UNIX, AF_INET, etc.) */
    int type,      /* SOCK_STREAM, SOCK_DGRAM, etc. */
    int protocol   /* specific to type; usually zero */
    );
```

```c
int bind (
    int socket_fd,                /* socket file descriptor */
    const struct sockaddr *sa,    /* socket address */
    socklen_t sa_len              /* address length */
    );
```

```c
struct sockaddr_un {
    sa_family_t sun_family; /* AF_UNIX */
    char sun_path[N];       /* socket pathname */
};


struct sockaddr_in {
    sa_family_t sin_family;  /* AF_INET */
    in_port_t sin_port;      /* port number (uint16_t) */
    struct in_addr sin_addr; /* IPv4 address */
};

struct in_addr {
    in_addr_t s_addr;   /* IPv4 address (uint32_t) */
};
```

```
int listen (
    int socket_fd,  /* socket file descriptor */
    int backlog     /* maximum connection queue size */
    );
```

```
int accept (
    int socket_fd,              /* socket file descriptor */
    struct_sockaddr *sa,        /* socket address or NULL */
    socklen_t *sa_len           /* address length */
    );
```

`accept` returns a **new FD**!

client *connects*

```c
int connect (
    int socket_fd,                /* socket file descriptor */
    const struct_sockaddr *sa,    /* socket address */
    socklen_t sa_len              /* address length */
    );
```

addresses?

IPv4 = 32bit IP addresses

```
uint32_t addr = (216UL << 24) + (47UL << 16)
                + (152UL << 8) + 36UL;
uint16_t port = 80;
```

machine level byte ordering

little endian vs. big endian

```
struct sockaddr_in sa;
sa.sin_port        = htons(port);
sa.sin_addr.s_addr = htonl(addr);
```

ada.cs.iit.edu

vs.

216.47.150.90

naming registry

Domain **N**ame **S**ystem

```
int getaddrinfo {
    const char *nodename,        /* host name                */
    const char *servname,        /* service name, e.g., http */
    const struct addrinfo *hint, /* query hint               */
    struct addrinfo **infop      /* returned info struct(s)  */
);
```

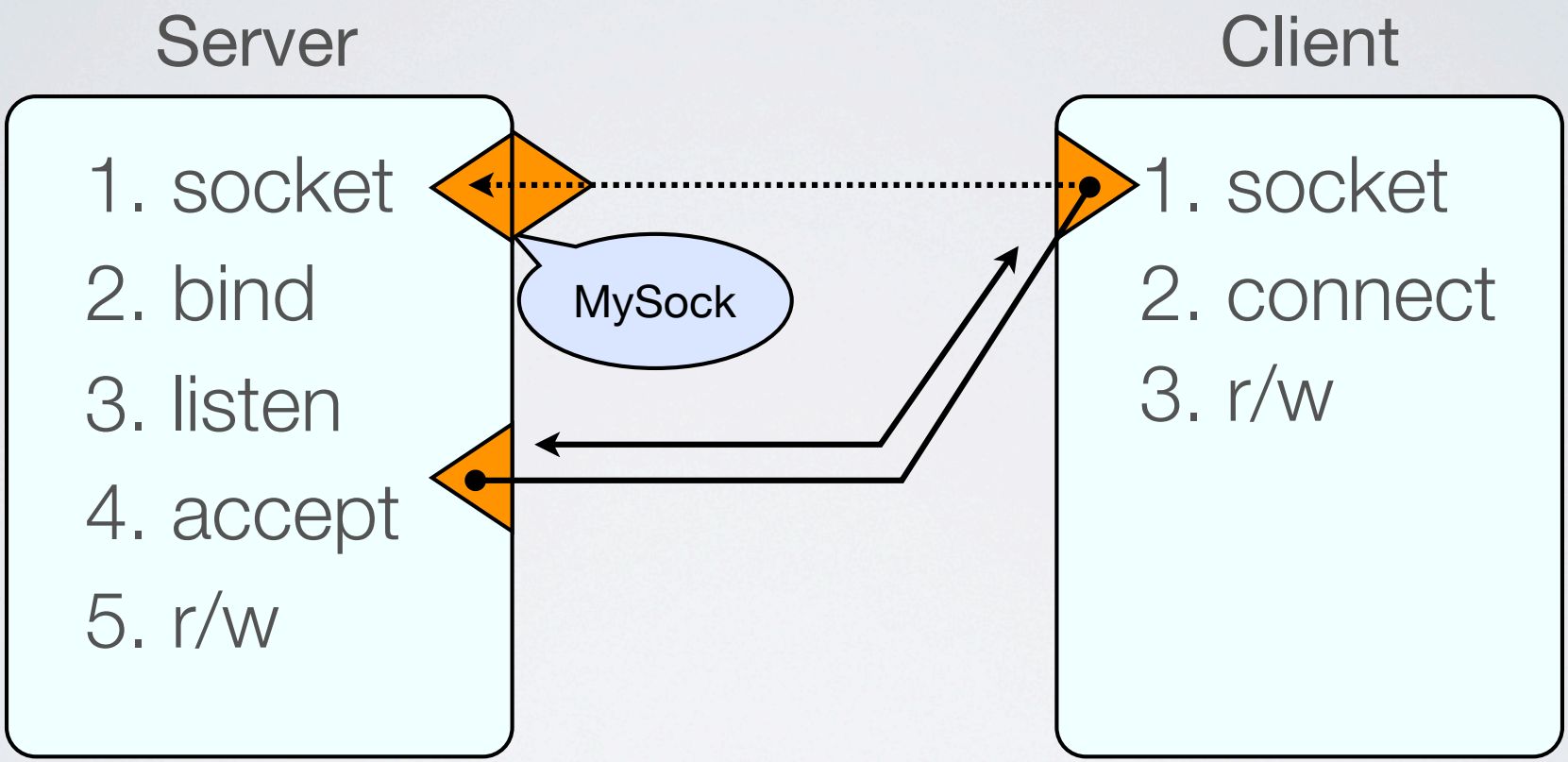
```c
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    socklen_t ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;
    struct addrinfo *ai_next;
};
```

```c
int main (int argc, char **argv) {
    struct addrinfo hint, *infop;

    memset(&hint, 0, sizeof(hint)); /* zero out hint */

    hint.ai_family = AF_INET;
    hint.ai_socktype = SOCK_STREAM;

    getaddrinfo(argv[1], argv[2], &hint, &infop);

    /* navigate linked list of addrinfo structs */
    for ( ; infop != NULL; infop = infop->ai_next) {
        struct sockaddr_in *sa = infop->ai_addr;
        printf("%s port: %d\n", inet_ntoa(sa->sin_addr),
               ntohs(sa->sin_port));
    }

    freeaddrinfo(infop); /* avoid memory leak! */
}
```

```
$ ./getaddrinfo ada.cs.iit.edu http
216.47.150.90 port: 80
```

```
$ ./getaddrinfo www.cnn.com http
157.166.224.26 port: 80
157.166.226.25 port: 80
157.166.226.26 port: 80
157.166.255.18 port: 80
157.166.255.19 port: 80
157.166.224.25 port: 80
```

Summary


Server
1. socket
2. bind
3. listen
4. accept
5. r/w
MySock
Client
1. socket
2. connect
3. r/w

```c
#define SOCKETNAME "mysock.sock"

int main () {
    struct sockaddr_un sa;
    char buf[100];
    strcpy(sa.sun_path, SOCKETNAME);
    if (fork() == 0) {
        /* child --- client */
        int fd_con;
        fd_con = socket(AF_UNIX, SOCK_STREAM, 0);
        connect(fd_con, &sa, sizeof(sa));
        write(fd_con, "Hello!", 7);
        read(fd_con, buf, sizeof(buf));
        printf("Client got %s\n", buf);
        close(fd_con);
    } else {
        /* parent --- server */
        int fd_srv, fd_client;
        fd_srv = socket(AF_UNIX, SOCK_STREAM, 0);
        bind(fd_srv, &sa, sizeof(sa));
        listen(fd_srv, SOMAXCONN);
        fd_client = accept(fd_srv, NULL, 0);
        read(fd_client, buf, sizeof(buf));
        printf("Server got %s\n", buf);
        write(fd_client, "Goodbye!", 9);
        close(fd_srv);
        close(fd_client);
    }
}
```

```c
if (fork() == 0) {
   /* child --- client */
    int fd_con;
    fd_con = socket(AF_UNIX, SOCK_STREAM, 0);
    while (connect(fd_con, &sa, sizeof(sa)) == -1) {
        sleep(1);
        continue;
    }
    write(fd_con, "Hello!", 7);
    read(fd_con, buf, sizeof(buf));
    printf("Client got %s\n", buf);
    close(fd_con);
}
```

```
Server got Hello!
Client got Goodbye!
```

application level **protocol**

“handshake” procedure

methods & arguments

Hyper**T**ext **T**ransfer **P**rotocol

verbs: GET, PUT,
POST, DELETE

nouns: URLs

```c
#define REQUEST "GET / HTTP/1.0\r\n\r\n"

int main (int argc, char **argv) {
    struct addrinfo *infop = NULL, hint;
    int fd_skt;
    char buf[1000];
    ssize_t nread;

    /* get address of webserver */
    memset(&hint, 0, sizeof(hint));
    hint.ai_family = AF_INET;
    hint.ai_socktype = SOCK_STREAM;
    getaddrinfo("www.iit.edu", "80", &hint, &infop);

    /* get socket and establish connection to webserver */
    fd_skt = socket(infop->ai_family, infop->ai_socktype, infop->ai_protocol);
    connect(fd_skt, infop->ai_addr, infop->ai_addrlen);

    /* send HTTP GET request to server */
    rio_writen(fd_skt, REQUEST, strlen(REQUEST));

    /* read (partial) server response and print to stdout */
    nread = rio_readn(fd_skt, buf, sizeof(buf));
    write(1, buf, nread);

    close(fd_skt);
}
```

```
HTTP/1.1 200 OK
Date: Wed, 18 Nov 2009 02:25:03 GMT
Server: Apache/1.3.31 (Unix) PHP/4.3.7
Connection: close
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<link rel="stylesheet" type="text/css" href="profiles/css/iit_homepage_style.css" />
...
```

“echo” server

```
void echo (int fd) {
    ssize_t n;
    char buf[MAXLINE];

    n = read(fd, buf, MAXLINE);
    printf("Server received %d bytes\n", n);
    write(fd, buf, n);
}
```

(short counts!)

```c
void echo (int fd) {
    ssize_t n;
    char buf[MAXLINE];
    rio_t rio;

    rio_readinitb(&rio, fd);
    while((n = rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("Server received %d bytes\n", n);
        rio_writen(fd, buf, n);
    }
}
```

```c
int main(int argc, char **argv) {
    int fd_srv, fd_client, clen;
    struct sockaddr_in saddr, caddr;

    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl(INADDR_ANY);
    saddr.sin_port = htons(atoi(argv[1]));

    fd_srv = socket(AF_INET, SOCK_STREAM, 0);
    bind(fd_srv, &saddr, sizeof(saddr));
    listen(fd_srv, SOMAXCONN);

    while (1) {
        clen = sizeof(caddr);
        fd_client = accept(fd_srv, &caddr, &clen);
        printf("Connection from %s\n", inet_ntoa(caddr.sin_addr));
        echo(fd_client);
        close(fd_client);
    }
}
```

Demo

serial execution stinks

desire **concurrent**
client handling

1. multiple processes

shared open files

```
while (1) {
    clen = sizeof(caddr);
    fd_client = accept(fd_srv, &caddr, &clen);
    printf("Connection from %s\n", inet_ntoa(caddr.sin_addr));

    if (fork() == 0) {
        close(fd_srv);
        echo(fd_client);
        close(fd_client);
        exit(0);
    }

    close(fd_client);
}
```

(child reaping still necessary)

Demo

pros/cons

1. concurrency achieved
2. kernel-level concurrency
3. simple & elegant
4. robust!

pros/**cons**

1. processes are “heavy-weight” (lot of overhead)
2. intra-server data sharing is difficult (need IPC!)

2. I/O multiplexing

FD “juggling”

track sets of connected FDs — monitor for read/write “readiness” (i.e. no blocking)

```c
int select (
      int nfds,   /* # FDs in each set to check */
      fd_set *readset,    /* read set or NULL */
      fd_set *writeset,   /* write set or NULL */
      fd_set *errorset,   /* error set or NULL */
      struct timeval *timeout  /* time-out or NULL */
    );
```

`fd_set` = bit set

“value-result” arguments

```
int select (
    int nfds,    /* # FDs in each set to check */
    fd_set *readset,     /* read set or NULL */
    fd_set *writeset,    /* write set or NULL */
    fd_set *errorset,    /* error set or NULL */
    struct timeval *timeout  /* time-out or NULL */
    );
```

`fd_set` Macros

- `FD_ZERO (fd_set *fdset);`
  - Clear entire `fdset`
- `FD_SET (int fd, fd_set *fdset);`
  - Set bit for `fd` in `fdset`
- `FD_CLR (int fd, fd_set *fdset);`
  - Clear bit for `fd` in `fdset`
- `FD_ISSET (int_fd, fd_set *fdset);`
  - Is `fd` in `fdset` set?

```c
int fd_srv,
    fd_client;

fd_set read_set,
       vr_set;   /* value-return set */

/* socket, bind, listen */
fd_srv = socket(AF_INET, SOCK_STREAM, 0);
bind(fd_srv, ...);
listen(fd_srv, SOMAXCONN);

/* clear out read_set */
FD_ZERO(&read_set);

/* add server FD to read_set */
FD_SET(fd_srv, &read_set);
```

```c
int fd_hwm = fd_srv;   /* set FD "high water mark" */

while (1) {
    vr_set = read_set;

    /* block until an FD is readable */
    select(fd_hwm+1, &vr_set, NULL, NULL, NULL);

    /* loop over all FDs */
    for (fd = 0; fd <= fd_hwm; fd++) {
        if (FD_ISSET(fd, &vr_set)) {
            if (fd == fd_srv) {
                /* accept a new client */
                fd_client = accept(fd_srv, NULL, 0);
                FD_SET(fd_client, &read_set);
                if (fd_client > fd_hwm)
                    fd_hwm = fd_client;
            } else {
                /* handle incoming client data */
                echo(fd);
                FD_CLR(fd, &read_set);
                if (fd == fd_hwm)
                    fd_hwm--;
                close(fd);
            }
        }
    }
}
```

pros/cons

1. single logical control flow
2. easier to debug (?!)
3. no process overhead
4. implicit data sharing

pros/**cons**

1. complicated!
2. not very robust
3. no true parallelism

multiple processes

vs.

I/O multiplexing

multiple processes

middle ground: multi**thread**ing

I/O multiplexing

```c
void echo (int fd) {
    ssize_t nread;
    char buf[80];
    nread = read(fd, buf, 80);
    printf("Server received %d bytes\n", nread);
    write(fd, buf, nread);
}

void *thread (void *vargp) {
  int fd = *((int *)vargp);
  pthread_detach(pthread_self());
  free(vargp);
  echo(fd);
  close(fd);
  return NULL;
}

int main(int argc, char **argv) {
  int fd_skt, *fd_client, clen;
  pthread_t tid;

  /* socket setup */
  while (1) {
    fd_client = malloc(sizeof(int));
    *fd_client = accept(fd_skt, (struct sockaddr *)&caddr, &clen);
    printf("Connection from %s\n", inet_ntoa(caddr.sin_addr));
    pthread_create(&tid, NULL, thread, fd_client);
  }
}
```