

Richard Litt



Scan QR code to add me as a friend.

Week 11 Lecture 2

XMUT-NWEN 241 - 2026 T2

Systems Programming

Richard Littauer

School of Engineering and Computer Science

Victoria University of Wellington

richard.littauer@vuw.ac.nz

Admin

- Exercise 3
  - Due date is 6 December.

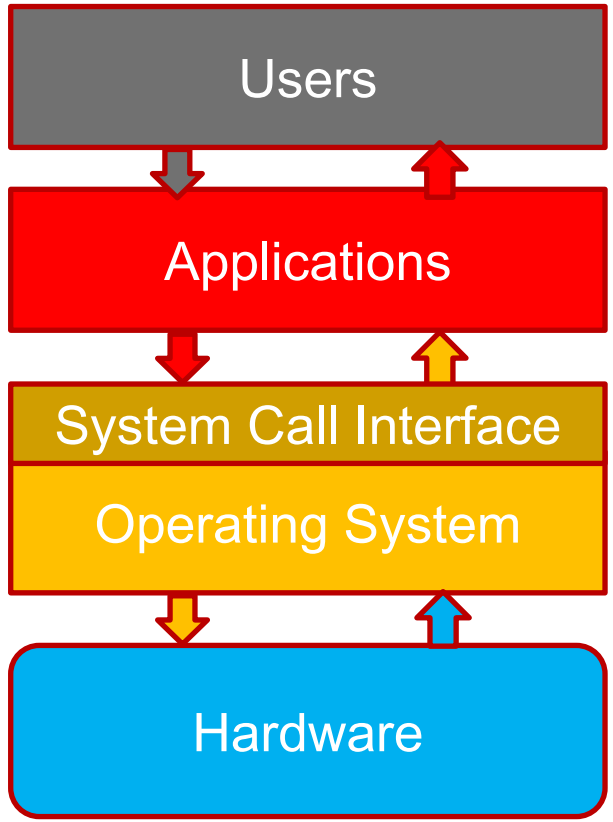
- For the exercises, since we use an automated marking process, please make sure to:
  - Submit C files. In particular no Microsoft Word documents and executable files.
  - No special characters. It breaks the script and makes me sad!

Content

- Pop quiz (not graded)
- Interprocess communication
  - TCP Socket Programming

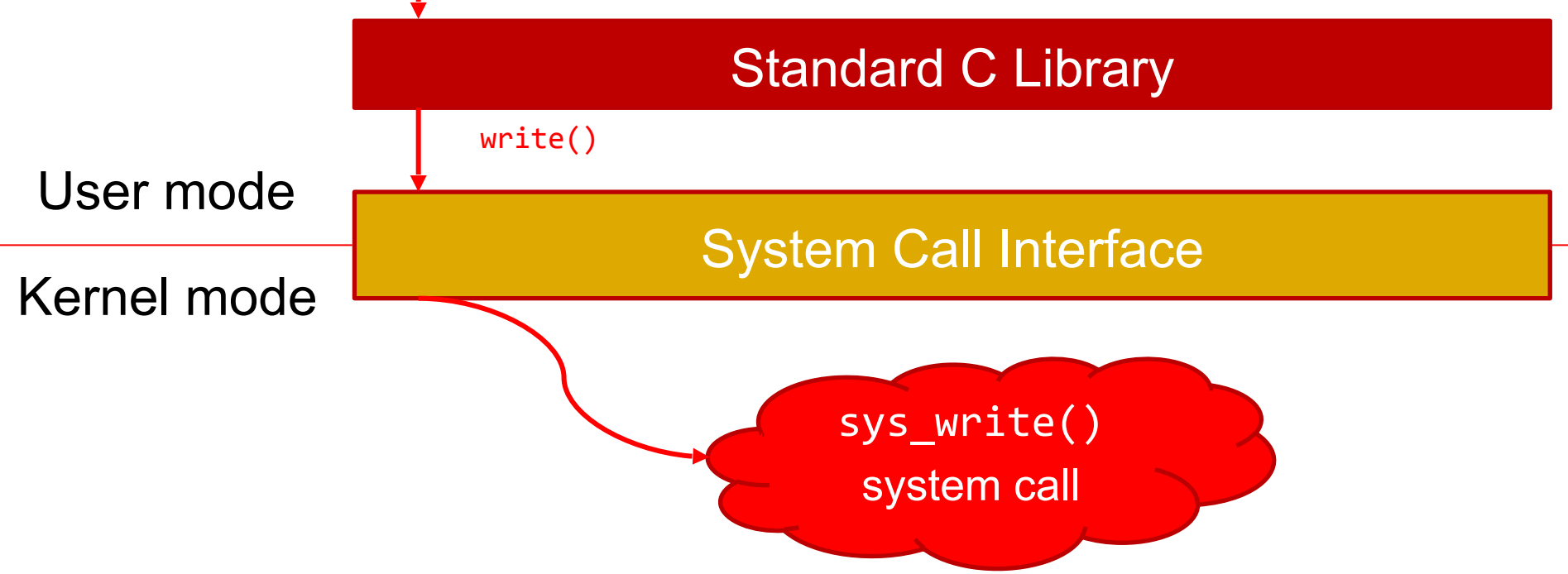
Recall: System calls - What and Why?

- Operating Systems do not allow application software to access system resources directly due to security and reliability issues.
- A program can request the services of system resources from OS through system calls.
- System calls are function invocations made from application into the OS in order to request some service or resource from the operating system.
- Application developers often do not have direct access to system calls but can access them through a system call API, which in turn invokes the system call.



Recall: System call invocation –Example

```
#include <stdio.h>
void main(void)
{
    printf("Hello, world\n");
    exit(0);
}
```



Side Note: How to know which system calls are invoked?

Two commands:

a) `ltrace` – traces call to library functions

b) `strace` -traces system calls

- Details in Linux manual pages :

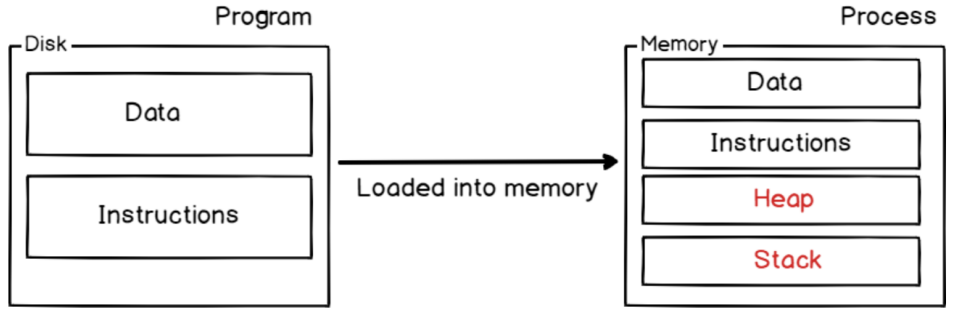
```
$ man ltrace
```

- Usage:

```
$ ltrace ./<program executable file>
$ ltrace -S ./<program executable file>   # also display system calls
```

Recap: What is a process ?

- Program and process are related terms.

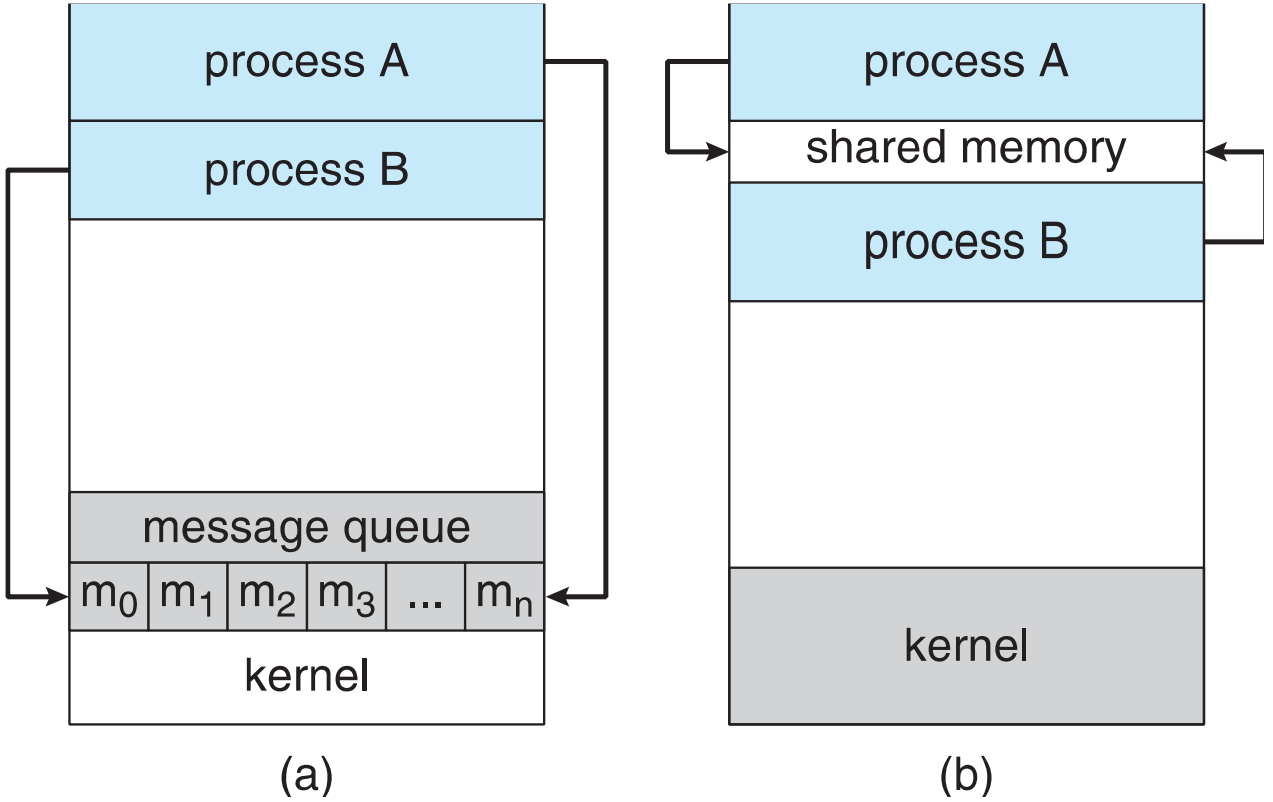


Program is a set of instructions to carry out a specified task	Process is a program in execution

Passive entity	Active entity
Program is a stored in disk and does not require any other resource.	Process requires system resources such as CPU, memory, I/O etc.
Life span - Longer	Life span – limited
Each time a program is run a new process is created.	

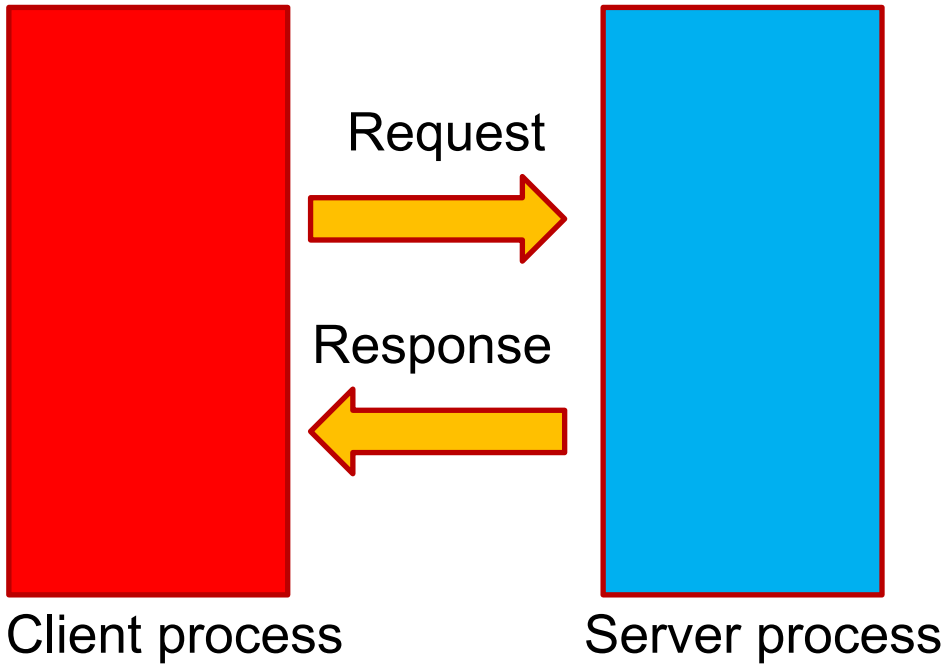
Recap: Interprocess communication

- Cooperating processes need interprocess communication (IPC)
- Two primary models of IPC
  - Message passing
  - Shared memory



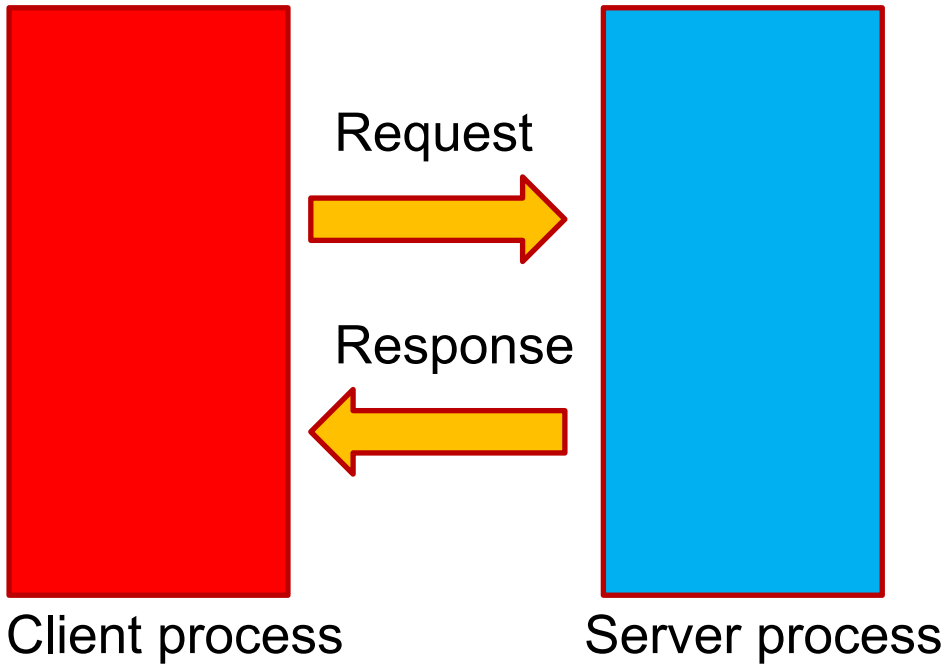
Client-server model

- Most common IPC paradigm
- Based on the producer-consumer model of process cooperation
- Client makes the request for some resource or service to the server process
- Server process handles the request and sends the response (result) back to the client



Client-server model

- Client process needs to know the existence and the address of the server
- However, the Server does not need to know the existence or address of the client prior to the connection
- Once a connection is established, both sides can send and receive information

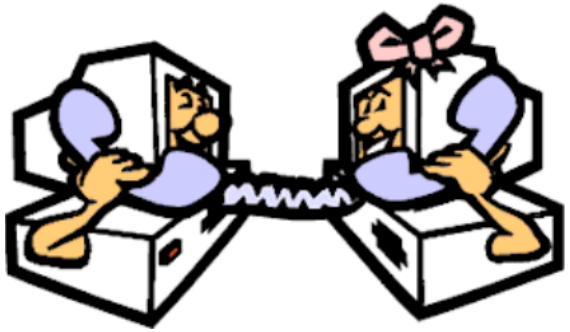


Client-server communication

- Remote Procedure Calls
- Pipes
- Sockets

What is socket?

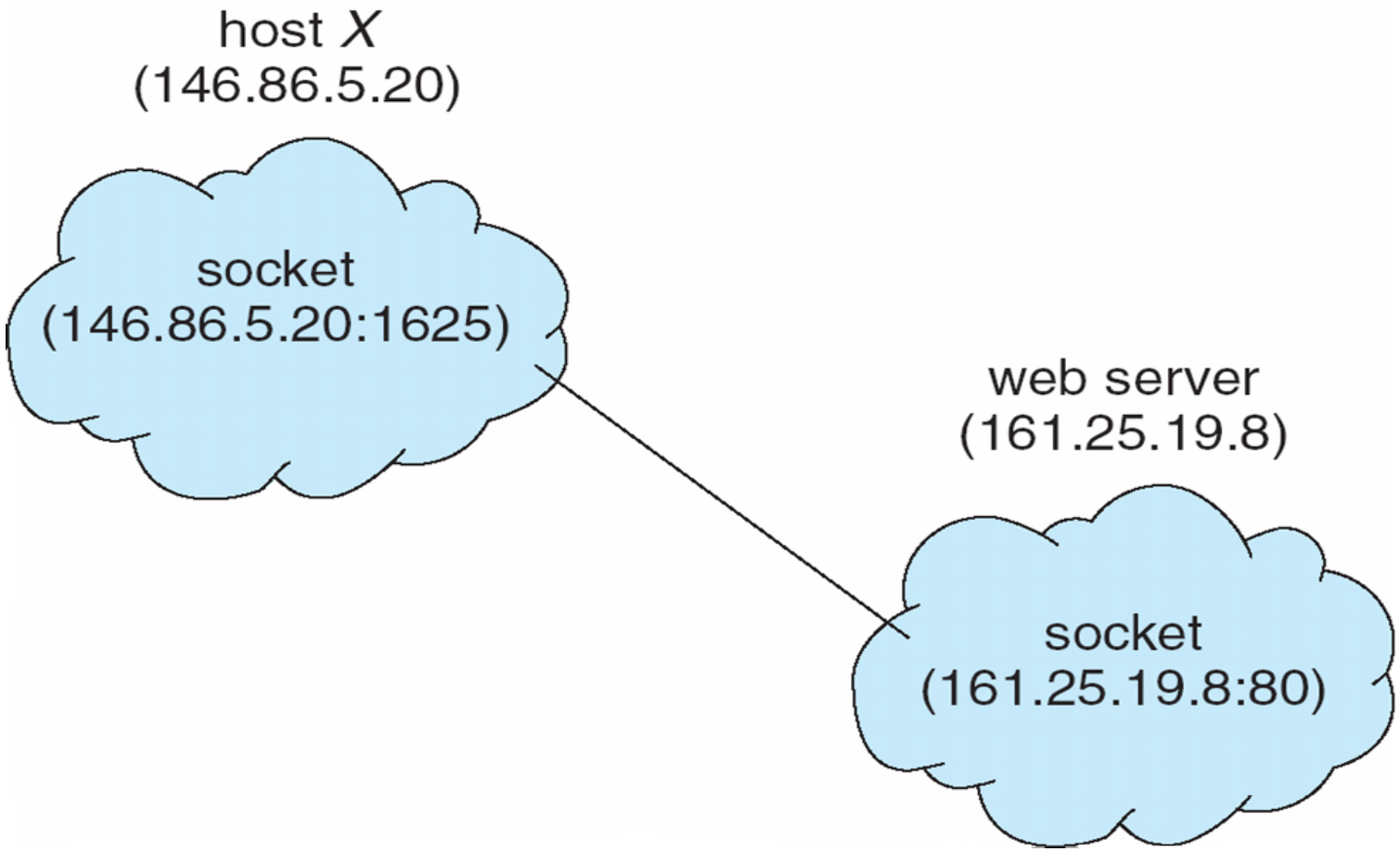
- What do we need to know to allow two processes on a network to communicate?
  - Identity of the communicating machines
    - IP Address
  - Identity of the communicating processes on these machines
    - Port



- Concatenation of IP address and port defines a socket - A socket is defined as an endpoint for communication

- Example: The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8

Socket communication



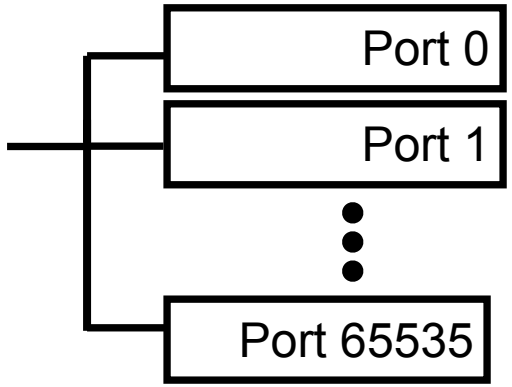
Port numbers

- Each host has 65,536 TCP ports and 65,536 UDP ports

- On Linux, use of ports 1-1023 requires privileges

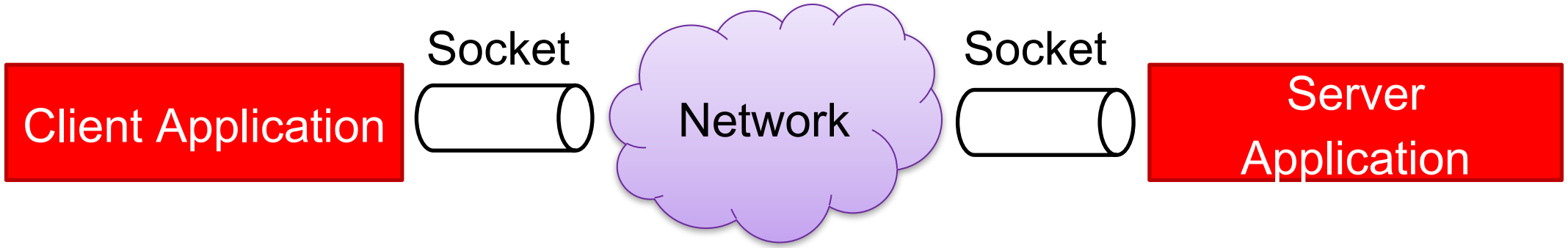
- Some ports are reserved for specific apps
  - 20, 21: FTP
  - 23: Telnet
  - 80: HTTP

- See iana.org for a list of reserved ports



Sockets as programming interface

- An interface between application and network
  - The application creates a socket
  - The socket type dictates the style of communication
  - TCP VS UDP
    - reliable vs. best effort
    - connection-oriented vs. connectionless



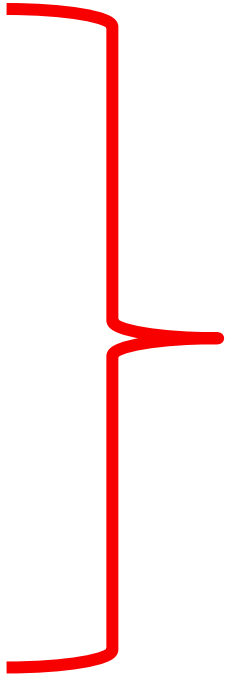
Socket types

- SOCK_STREAM
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional
- SOCK_DGRAM
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of “connection” – app indicates dest. for each packet
  - can send or receive

We will focus on SOCK_STREAM or TCP socket type

3

- socket()
- bind()
- listen()
- accept()
- connect()
- send() / sendto()
- recv() / recvfrom()



Include

sys/types.h

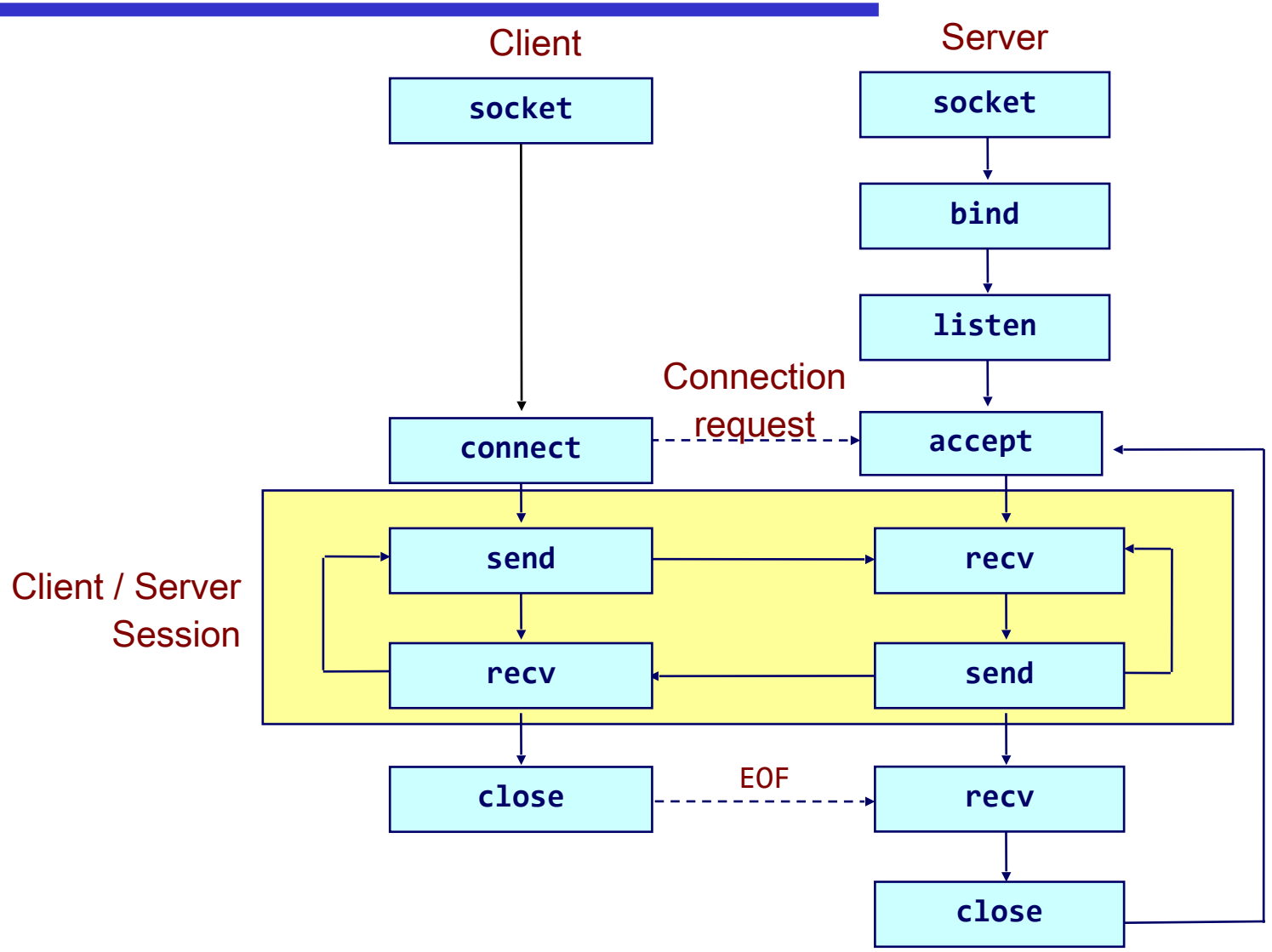
sys/socket.h

TCP Client overview

1) Create a socket with the `socket()` system call

2) Connect the socket to the address of the server using the `connect()` system call

3) Send and receive data



TCP Server overview

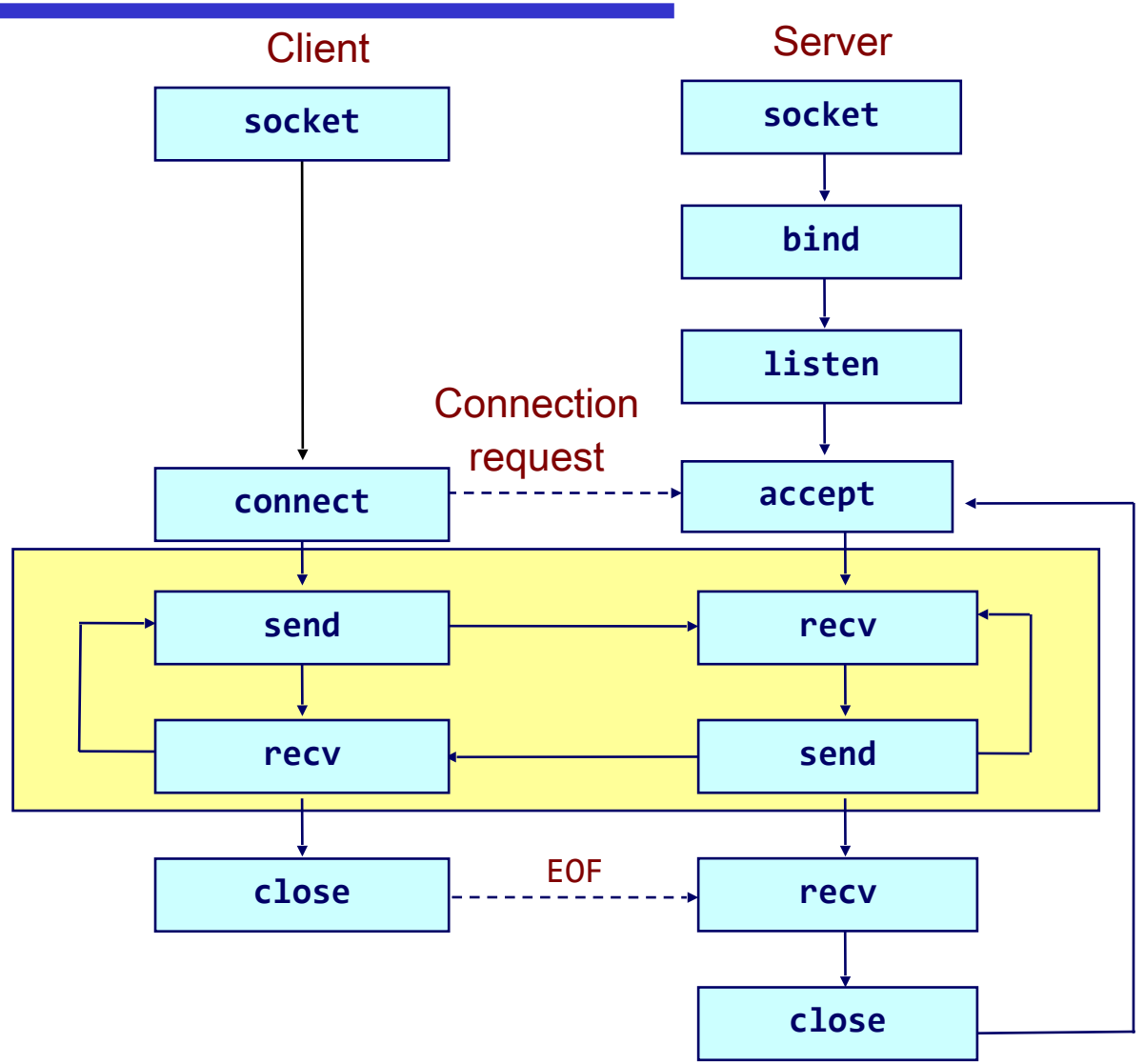
1) Create a socket with the `socket()` system call

2) Bind the socket to an address using the `bind()` system call

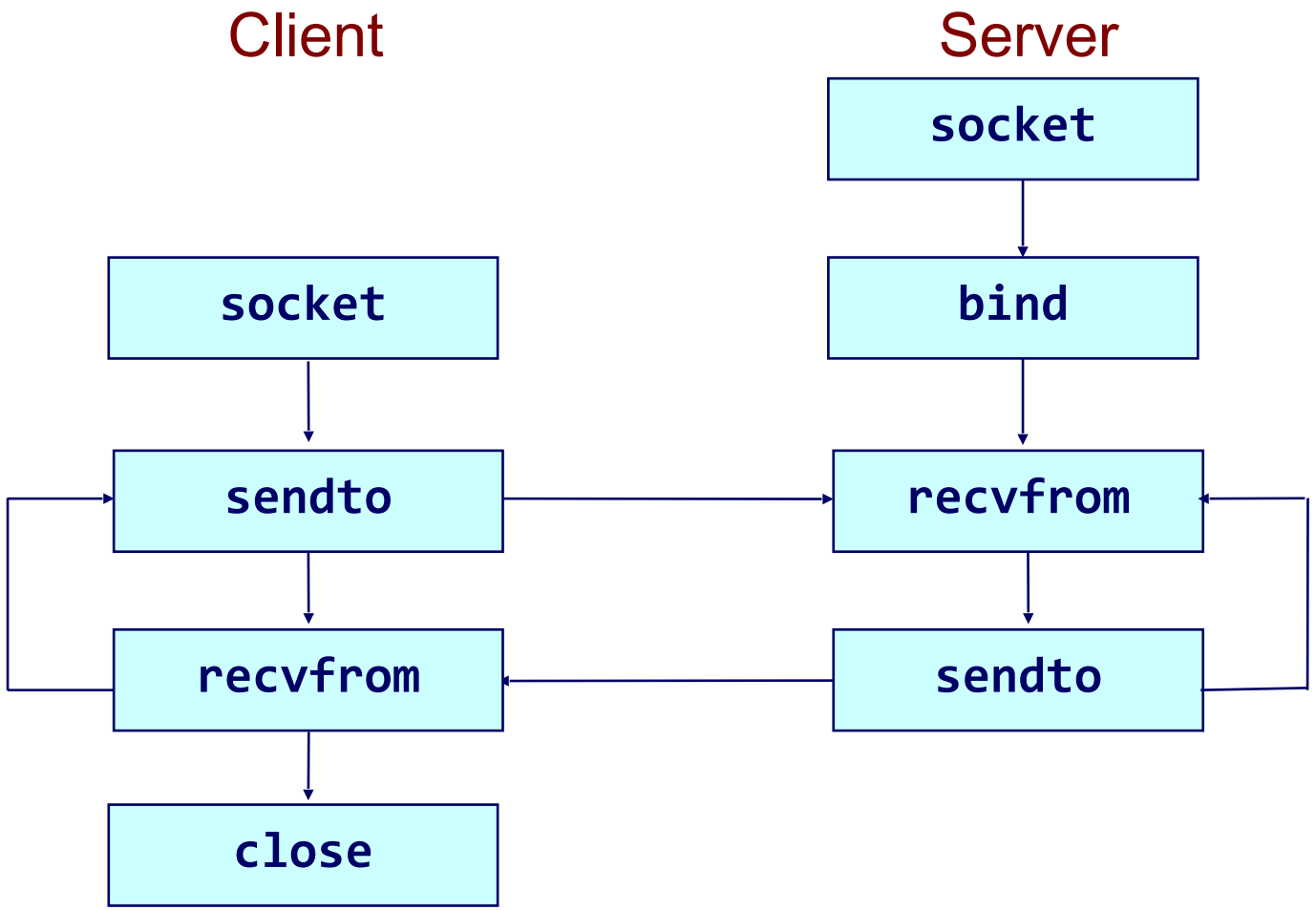
3) Listen for connections with the `listen()` system call

4) Accept a connection with the `accept()` system call

5) Send and receive data



Client-server communication overview - UDP



Server: step 1

- Create a socket with the `socket()` system call

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

  - *domain* – communication domain (protocol family) such as `AF_INET` (IPv4) or `AF_INET6` (IPv6)
  - *type* – communication semantics such as `SOCK_STREAM` (TCP) or `SOCK_DGRAM` (UDP)
  - *protocol* specifies the protocol, usually 0.

  - Creates an endpoint of communication.
  - If successful, returns socket file descriptor, otherwise, returns -1

Server: step 1 example

- Create TCP socket

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd == -1) {
    printf("Error creating socket\n");
    exit(1);
}
```

- Create UDP socket

```
int fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd == -1) {
    printf("Error creating socket\n");
    exit(1);
}
```

Demo: create a socket

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

int main(void)
{
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd == -1) {
        printf("Error creating socket\n");
        exit(1);
    }
    printf("socket returned %d\n", fd);
    return 0;
}
```

Demo: create a socket

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(void)
{
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd == -1) {
        printf("Error creating socket\n");
        exit(1);
    }
    printf("socket returned %d\n", fd);
    return 0;
}
```

Why it matters

- `socket` returns a file descriptor, like `open` and `pipe`.
- 0, 1 and 2 are already taken: stdin, stdout and stderr.
- It isn't connected to anything yet.

Output

```
$ ./prog
socket returned 3
```

Server: step 2

- Bind the socket to an address using the `bind()` system call

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

  - *sockfd* is the socket file descriptor (returned by `socket()`)
  - *addr* is a pointer to the structure *struct sockaddr* (generic data type for address) which contains the host IP address and port number to bind to
  - *addrlen* is the length of what `addr` points to

  - Binding means associating and reserving a port number for use by the socket
  - If successful, returns 0, otherwise, returns -1

struct sockaddr_in

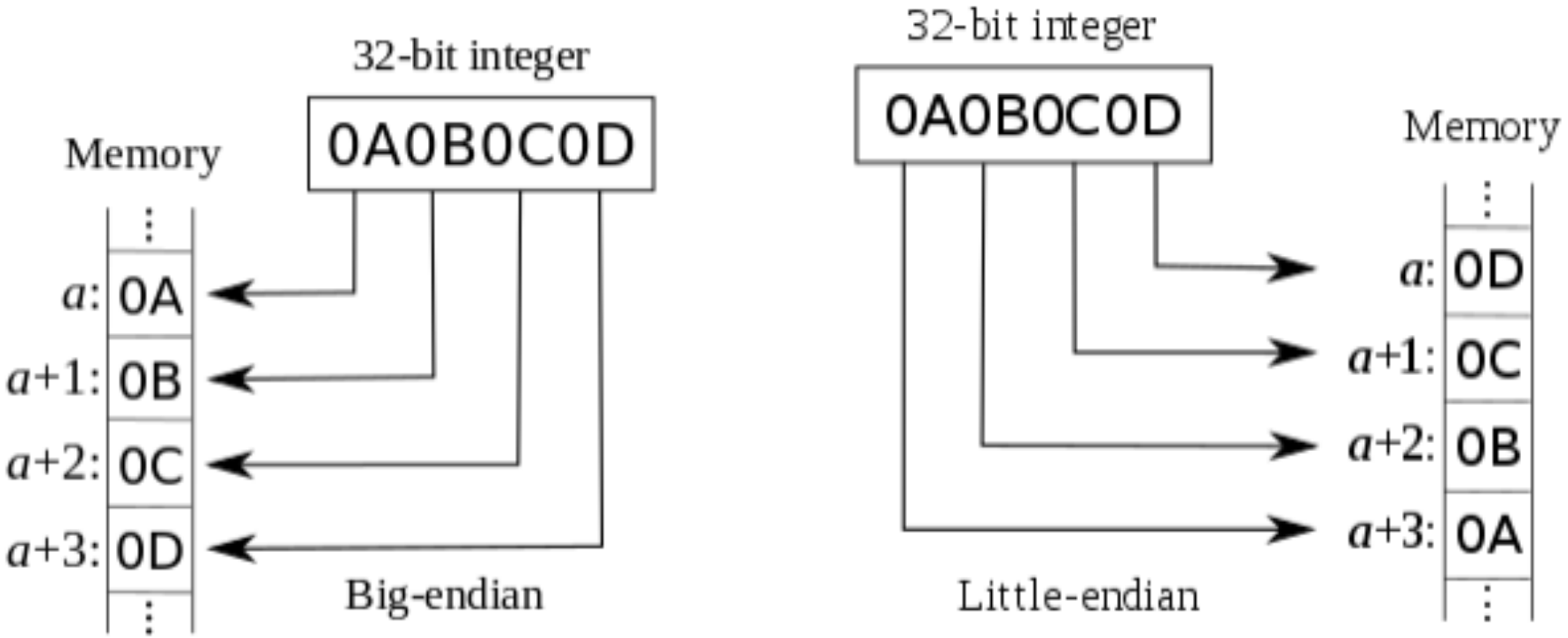
- struct sockaddr_in in IPv4 (included in <netinet/in.h> header)

```
struct sockaddr_in {
    short sin_family;          // AF_INET
    unsigned short sin_port;   // port number
    struct in_addr sin_addr;   // Internet address in
                               //network byte order
};

struct in_addr {
    uint32_t s_addr;          // IPv4 address in network
                              //byte order
};
```

Host and network byte order

- Little-endian and big-endian issue?



Host and network byte order

- Byte ordering also matters in network communication
  - Host and network may differ in byte ordering
  - Host byte order may be little-endian or big-endian
  - Network byte order is always big-endian

- Functions for converting between host and network byte order:

```
uint32_t htonl(uint32_t hostlong);  //host to network long
uint16_t htons(uint16_t hostshort); //host to network short
uint32_t ntohl(uint32_t netlong);   //network to host long
uint16_t ntohs(uint16_t netshort);  //network to host short
```

The l means 32 bits.
The s means 16 bits.

Demo: byte order

```
#include <stdio.h>
#include <arpa/inet.h>

int main(void)
{
    unsigned short port = 1234;
    printf("host order:    %d (hex %x)\n", port, port);
    printf("network order: %d (hex %x)\n",
           htons(port), htons(port));
    return 0;
}
```

Demo: byte order

```c
#include <stdio.h>
#include <arpa/inet.h>

int main(void)
{
    unsigned short port = 1234;
    printf("host order:    %d (hex %x)\n", port, port);
    printf("network order: %d (hex %x)\n",
           htons(port), htons(port));
    return 0;
}
```

Why it matters

- 1234 is `04 D2` in hex.
- `htons` swapped the bytes: `D2 04`.
- On a big-endian machine, `htons` changes nothing.

Output

```
$ ./prog
host order:    1234 (hex 4d2)
network order: 53764 (hex d204)
```

Server: step 2 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; // any address
addr.sin_port = htons(1234);  // port 1234

if (bind(fd, (struct sockaddr *)&addr, sizeof(addr))<0) {
    printf("Error binding socket\n");
    exit(1);
}
```

```
struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
};

struct in_addr {
    uint32_t s_addr;
};
```

Demo: bind

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; // any address
addr.sin_port = htons(1234);  // port 1234

if (bind(fd, (struct sockaddr *)&addr, sizeof(addr))<0) {
    printf("Error binding socket\n");
    exit(1);
}
printf("bound to port 1234. Press return to quit.\n");
getchar();
```

Terminal 1

```
$ ./prog
```

Terminal 2

```
$ ./prog
```

Demo: bind

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; // any address
addr.sin_port = htons(1234);  // port 1234

if (bind(fd, (struct sockaddr *)&addr, sizeof(addr))<0) {
    printf("Error binding socket\n");
    exit(1);
}
printf("bound to port 1234. Press return to quit.\n");
getchar();
```

Why it matters

- Only one socket can have a port.
- The second copy fails while the first is running.
- Quit the first, and the second works.

Terminal 1

```
$ ./prog
bound to port 1234. Press return to quit.
```

Terminal 2

```
$ ./prog
Error binding socket
```

Server: step 3

- Listen for connections with the `listen()` system call

```
int listen(int sockfd, int backlog);
```

  - *`sockfd`* is the socket file descriptor (returned by `socket()`)
  - *`backlog`* is the maximum number of pending connections to allow for this socket
    - `SOMAXCONN` is defined as the number of maximum pending connections allowed by the operating system
  - If successful, returns 0, otherwise, returns -1

Server: step 3 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…

if(listen(fd, SOMAXCONN) < 0) {
    printf("Error listening for connections\n");
    exit(1);
}
```

Demo: listen

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
… bind as before …

if(listen(fd, SOMAXCONN) < 0) {
    printf("Error listening for connections\n");
    exit(1);
}
printf("listening on port 1234. Press return to quit.\n");
getchar();
```

Terminal 1

```
$ ./prog
```

Terminal 2

```
$ nc -v 127.0.0.1 1234
```

Demo: listen

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
… bind as before …

if(listen(fd, SOMAXCONN) < 0) {
    printf("Error listening for connections\n");
    exit(1);
}
printf("listening on port 1234. Press return to quit.\n");
getchar();
```

Why it matters

- `nc` is a ready-made TCP client.
- The OS finished the connection and queued it.
- `accept` will take it from the queue.

Terminal 1

```
$ ./prog
listening on port 1234. Press return to quit.
```

Terminal 2

```
$ nc -v 127.0.0.1 1234
Connection to 127.0.0.1 port 1234 [tcp/search-agent] succeeded!
```

Server: step 4

- Accept a connection with the `accept()` system call

```
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
```

  - *sockfd* is the socket file descriptor (returned by `socket()`)
  - *addr* is a pointer to the structure `struct sockaddr` which will contain the details of the peer socket
  - *addrlen* is a pointer to the length of what `addr` points to

  - If successful, returns non-negative socket file descriptor, otherwise, returns -1

Server: step 4 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…
struct sockaddr_in client_addr;
socklen_t addrlen = sizeof(client_addr);

int client_fd = accept(fd, (struct sockaddr *)&client_addr,
                       &addrlen);
if(client_fd < 0) {
    printf("Error accepting connection\n");
    exit(1);
}
```

Demo: accept

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
… bind and listen as before …
struct sockaddr_in client_addr;
socklen_t addrlen = sizeof(client_addr);

printf("waiting for a client\n");
int client_fd = accept(fd, (struct sockaddr *)&client_addr,
                       &addrlen);
…
printf("client connected from %s:%d\n",
       inet_ntoa(client_addr.sin_addr),
       ntohs(client_addr.sin_port));
printf("listening socket: %d, client socket: %d\n",
       fd, client_fd);
```

Terminal 1

```
$ ./prog
```

Terminal 2

```
$ nc 127.0.0.1 1234
```

Demo: accept

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
… bind and listen as before …
struct sockaddr_in client_addr;
socklen_t addrlen = sizeof(client_addr);

printf("waiting for a client\n");
int client_fd = accept(fd, (struct sockaddr *)&client_addr,
                       &addrlen);
…
printf("client connected from %s:%d\n",
       inet_ntoa(client_addr.sin_addr),
       ntohs(client_addr.sin_port));
printf("listening socket: %d, client socket: %d\n",
       fd, client_fd);
```

Why it matters

- `accept` returned a new socket, 4, for this client.
- Socket 3 is still listening.
- The OS picked the client's port.

Terminal 1

```
$ ./prog
waiting for a client
client connected from 127.0.0.1:59526
listening socket: 3, client socket: 4
```

Terminal 2

```
$ nc 127.0.0.1 1234
```

Server: step 5 send

- Send and receive data in TCP

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

  - *sockfd* is the socket file descriptor (returned by accept())
  - *buf* is a pointer to buffer to be sent
  - *len* is the length of buffer to be sent
  - *flags* is bitwise OR of zero or more options

- Used in connection-oriented sockets (TCP)
- If successful, returns number of characters sent, otherwise, returns -1
- `send(sockfd, buf, len, 0);` is equivalent to `write(sockfd, buf, len);`

Server: step 5 sendto

- Send and receive data in UDP

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

  - *sockfd* is the socket file descriptor (returned by socket())
  - *buf* is a pointer to buffer to be sent
  - *len* is the length of buffer to be sent
  - *flags* is bitwise OR of zero or more options
  - *dest_addr* is a pointer to the structure `struct sockaddr` which contains the address of the peer socket
  - *addrlen* is the length of what `dest_addr` points to
- Used in non-connection-oriented sockets (UDP)
- If successful, returns number of characters sent, otherwise, returns -1

Server: step 5 example using send( )

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…
int client_fd = accept(fd, (struct sockaddr *)& client_addr,
                       &addrlen);
…

char msg[] = "hello, world";
ssize_t r = send(client_fd, msg, strlen(msg), 0);
if(r < 0) {
    printf("Error sending message\n");
    close(client_fd);
    exit(1);
}
```

Demo: send

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
… bind, listen and accept as before …

char msg[] = "hello, world";
ssize_t r = send(client_fd, msg, strlen(msg), 0);
if(r < 0) {
    printf("Error sending message\n");
    close(client_fd);
    exit(1);
}
printf("sent %zd bytes\n", r);
```

Why it matters

- `strlen` doesn't count the `'\0'`, so it isn't sent.
- `nc` prints exactly the 12 bytes.
- There's no newline, so the prompt follows straight after.

Terminal 1

```
$ ./prog
sent 12 bytes
```

Terminal 2

```
$ nc 127.0.0.1 1234
hello, world$
```

Server: step 5 receive

- Send and receive data

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

  - *sockfd* is the socket file descriptor (returned by accept())
  - *buf* is a pointer to buffer to be received
  - *len* is the length of buffer to be received
  - *flags* is bitwise OR of zero or more options

- Used in connection-oriented sockets (TCP)
- If successful, returns number of characters received, otherwise, returns -1
- If peer socket is shutdown/closed, will return 0
- `recv(sockfd, buf, len, 0);` is equivalent to `read(sockfd, buf, len);`

Server: step 5 receive from

- Send and receive data

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

  - *sockfd* is the socket file descriptor (returned by socket())
  - *buf* is a pointer to buffer to be received
  - *len* is the length of buffer to be received
  - *flags* is bitwise OR of zero or more options
  - *src_addr* is a pointer to the structure `struct sockaddr` which will contain the details of the peer socket
  - *addrlen* is a pointer to the length of what src_addr points to
- Used in non-connection-oriented sockets (UDP)
- If successful, returns number of characters received, otherwise, returns -1
- A return value of 0 means an empty message (UDP has no connection to close)

Server: step 5 example using recv()

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…
int client_fd = accept(fd, (struct sockaddr *)& client_addr,
                       &addrlen);
…

char incoming[100];
ssize_t r = recv(client_fd, incoming, 100, 0);
if(r <= 0) {
    printf("Error receiving message\n");
    close(client_fd);
    exit(1);
}
// Do something with receiving message
printf("Received message: %s", incoming);
```

Demo: recv

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
… bind, listen and accept as before …

char incoming[100];
ssize_t r = recv(client_fd, incoming, 100, 0);
if(r <= 0) {
    printf("Error receiving message\n");
    close(client_fd);
    exit(1);
}
// Do something with receiving message
printf("Received message: %s", incoming);
```

Why it matters

- `recv` copies bytes. It doesn't add a `'\0'`.
- `printf("%s")` runs on into old memory.
- It may look fine on another run. It's still a bug.

Terminal 1

```
$ ./prog
Received message: hello
<junk>
```

Terminal 2

```
$ nc 127.0.0.1 1234
hello
```

Demo: recv twice

```
… bind, listen and accept as before …

char incoming[100];
recv(client_fd, incoming, 100, 0);
printf("Received message: %s", incoming);

recv(client_fd, incoming, 100, 0);
printf("Received message: %s", incoming);
```

Terminal 1

```
$ ./prog
```

Terminal 2

```
$ nc 127.0.0.1 1234
hello
hi
```

Demo: recv twice, fixed

```
… bind, listen and accept as before …

char incoming[100];
ssize_t r = recv(client_fd, incoming, 99, 0);
incoming[r] = '\0';
printf("Received message: %s", incoming);

r = recv(client_fd, incoming, 99, 0);
incoming[r] = '\0';
printf("Received message: %s", incoming);
```

Why it matters

- `hi\n` only overwrote 3 bytes. `lo\n` was left over.
- Fix: put `'\0'` at `incoming[r]`.
- Receive at most 99, to leave room for it.

Before the fix

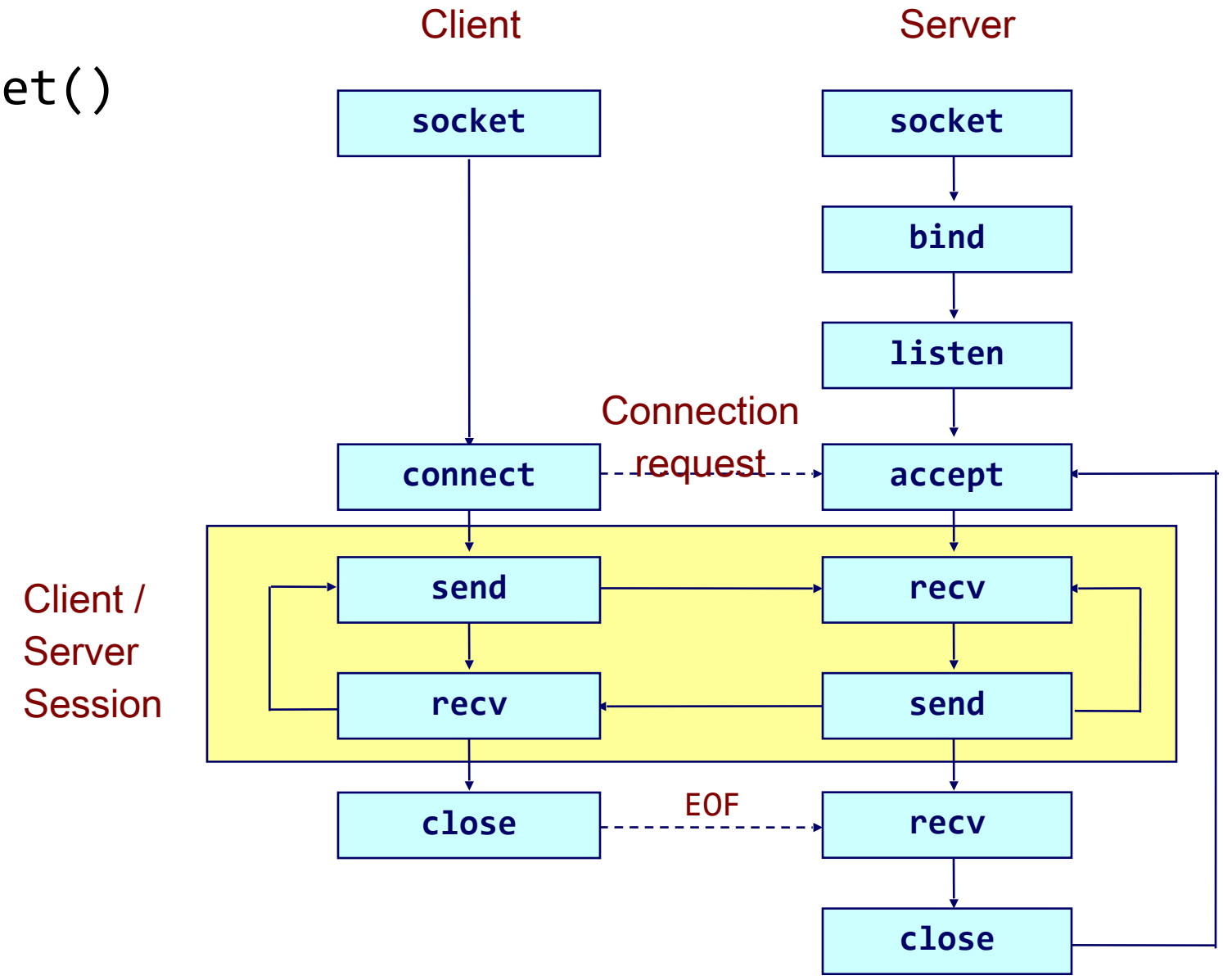
```
Received message: hello
<junk>
Received message: hi
lo
<junk>
```

After the fix

```
Received message: hello
Received message: hi
```

Client: step 1

- Create a socket with the `socket()` system call
- Same as server step 1



Client: step 2

- Connect the socket to the address of the server using the `connect()` system call
  - This step is only required for connection-oriented sockets (TCP)

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

  - *`sockfd`* is the socket file descriptor (returned by socket())
  - *`addr`* is a pointer to the structure `struct sockaddr` which contains the address of the server socket
  - *`addrlen`* is the length of what `addr` points to

  - If successful, returns 0, otherwise, returns -1

Client: step 3

- Send and receive data

- Same as server step 5

Demo: a client

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = inet_addr("127.0.0.1"); // this computer
addr.sin_port = htons(1234);  // port 1234

if (connect(fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
    printf("Error connecting\n");
    exit(1);
}

char incoming[100];
ssize_t r = recv(fd, incoming, 99, 0);
…
incoming[r] = '\0';
printf("Received message: %s\n", incoming);
close(fd);
```

Why it matters

- `connect` to the server's IP address and port.
- 127.0.0.1 always means this computer.
- With no server running, `connect` fails.

Terminal 1

```
$ ./server
sent 12 bytes
```

Terminal 2

```
$ ./client
Received message: hello, world

(server not running)
$ ./client
Error connecting
```

Closing a socket

- Socket must be closed after its use

```
int shutdown(int sockfd, int how);
```

```
int close(int sockfd);
```

  - *sockfd* is the socket file descriptor (returned by socket())
  - *how* can either be SHUT_RD (further receptions disallowed), SHUT_WR (further transmissions disallowed), or SHUT_RDWR (further receptions and transmissions disallowed)

  - If successful, returns 0, otherwise, returns -1

Demo: recv returns 0

```
… bind, listen and accept as before …

char incoming[100];
ssize_t r;
while ((r = recv(client_fd, incoming, 99, 0)) > 0) {
    incoming[r] = '\0';
    printf("Received message: %s", incoming);
}
printf("recv returned %zd\n", r);
close(client_fd);
close(fd);
```

Terminal 1

```
$ ./prog
```

Terminal 2

```
$ nc 127.0.0.1 1234
hello
how are you
^C
```

Demo: recv returns 0

```
… bind, listen and accept as before …

char incoming[100];
ssize_t r;
while ((r = recv(client_fd, incoming, 99, 0)) > 0) {
    incoming[r] = '\0';
    printf("Received message: %s", incoming);
}
printf("recv returned %zd\n", r);
close(client_fd);
close(fd);
```

Why it matters

- The loop runs while `recv` returns more than 0.
- Ctrl-C closed the connection, so `recv` returned 0.
- Then the server closes its own sockets.

Terminal 1

```
$ ./prog
Received message: hello
Received message: how are you
recv returned 0
```

Terminal 2

```
$ nc 127.0.0.1 1234
hello
how are you
^C
```

Extra: an echo server

```
… socket, bind and listen as before …
while (1) {
    printf("waiting for a client\n");
    int client_fd = accept(fd, (struct sockaddr *)&client_addr,
                           &addrlen);
    printf("client connected\n");

    char incoming[100];
    ssize_t r;
    while ((r = recv(client_fd, incoming, 100, 0)) > 0) {
        send(client_fd, incoming, r, 0);
    }
    printf("client left\n");
    close(client_fd);
}
```

Why it matters

- Back to `accept` after each client, as in the overview diagram.
- It sends back exactly `r` bytes, so no `'\0'` is needed.
- One client at a time. A second one waits.

Terminal 1

```
$ ./prog
waiting for a client
client connected
client left
waiting for a client
```

Terminal 2

```
$ nc 127.0.0.1 1234
one
one
```

Extra: many clients at once

```
… socket, bind and listen as before …
while (1) {
    int client_fd = accept(fd, (struct sockaddr *)&client_addr,
                           &addrlen);
    if (fork() == 0) {
        printf("process %d: client connected\n", getpid());
        … the same recv and send loop …
        printf("process %d: client left\n", getpid());
        close(client_fd);
        exit(0);
    }
    close(client_fd);
}
```

Why it matters

- Each client gets its own child process.
- The parent goes straight back to `accept`.
- Two clients are served at the same time.

Terminal 1, two clients

```
$ ./prog
process 18394: client connected
process 18402: client connected
process 18402: client left
process 18394: client left
```

Extra: Error binding socket

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…

int yes = 1;
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));

… bind, listen and accept as before …
printf("client connected, closing it\n");
close(client_fd);
```

Why it matters

- Rerun a server straight away, and `bind` can fail.
- The side that closed first keeps the port for a while: about 30 seconds here.
- `SO_REUSEADDR` lets `bind` use it anyway.

Without SO_REUSEADDR

```
$ ./prog
Error binding socket
```

With SO_REUSEADDR

```
$ ./prog
client connected, closing it
```

Extra: TCP is a stream

```
… socket, bind and listen as before …
int client_fd = accept(fd, (struct sockaddr *)&client_addr,
                       &addrlen);
printf("client connected, waiting 5 seconds\n");
sleep(5);

char incoming[100];
ssize_t r = recv(client_fd, incoming, 99, 0);
incoming[r] = '\0';
printf("one recv got %zd bytes:\n%s", r, incoming);
```

Why it matters

- TCP carries a stream of bytes, not separate messages.
- Two lines were typed, and one `recv` got both.
- Programs mark where messages end, for example with a newline.

Terminal 1

```
$ ./prog
client connected, waiting 5 seconds
one recv got 12 bytes:
hello
world
```

Terminal 2

```
$ nc 127.0.0.1 1234
hello
world
```

Extra: a web server

```
… socket, bind and listen as before …
while (1) {
    int client_fd = accept(fd, (struct sockaddr *)&client_addr,
                           &addrlen);
    char request[1000];
    ssize_t r = recv(client_fd, request, 999, 0);
    if (r > 0) {
        request[r] = '\0';
        printf("%s", request);
        char reply[] = "HTTP/1.0 200 OK\r\n\r\nHello from a C program!\n";
        send(client_fd, reply, strlen(reply), 0);
    }
    close(client_fd);
}
```

Why it matters

- A browser is just a TCP client.
- HTTP is text: a request, then a reply.
- Try http://localhost:1234 in a browser.

Terminal 1

```
$ ./prog
GET / HTTP/1.1
Host: 127.0.0.1:1234
User-Agent: curl/8.7.1
Accept: */*
```

Terminal 2

```
$ curl http://127.0.0.1:1234
Hello from a C program!
```

Extra: a UDP server

```
int fd = socket(AF_INET, SOCK_DGRAM, 0);
… bind as before. No listen, no accept …

while (1) {
    char incoming[100];
    struct sockaddr_in client_addr;
    socklen_t addrlen = sizeof(client_addr);
    ssize_t r = recvfrom(fd, incoming, 99, 0,
                         (struct sockaddr *)&client_addr, &addrlen);
    incoming[r] = '\0';
    printf("Received message: %s", incoming);
    sendto(fd, "got it\n", 7, 0, (struct sockaddr *)&client_addr, addrlen);
}
```

Why it matters

- `SOCK_DGRAM`: no connection, no `listen`, no `accept`.
- `recvfrom` says who sent each message.
- `sendto` replies to that address.

Terminal 1

```
$ ./prog
Received message: hello
Received message: again
```

Terminal 2

```
$ nc -u 127.0.0.1 1234
hello
got it
again
got it
```

Review question 1

```
int a = socket(AF_INET, SOCK_STREAM, 0);
int b = socket(AF_INET, SOCK_STREAM, 0);
printf("%d %d\n", a, b);
```

- What does this print?

Review question 2

```
printf("%x\n", htons(0x1234));
```

- What does this print on a little-endian machine?

Review question 3

```
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = 8080;
bind(fd, (struct sockaddr *)&addr,
     sizeof(addr));
listen(fd, SOMAXCONN);
```

- A client connects to port 8080 and gets Connection refused. Why?

Review question 4

```
ssize_t r;
while ((r = recv(client_fd, buf, 99, 0)) > 0) {
    buf[r] = '\0';
    printf("%s", buf);
}
printf("done\n");
```

- When does this print done?

Review question 5

```
int client_fd = accept(fd,
        (struct sockaddr *)&client_addr,
        &addrlen);
char msg[] = "hello";
send(fd, msg, strlen(msg), 0);
```

- The client never gets hello. What's wrong?

Review question 6

```
char buf[5];
recv(client_fd, buf, 5, 0);
printf("%s\n", buf);
```

- The client sends hello. What's wrong?

More questions: sockets and ports (1)

- What two things make up a socket address?
- What does 127.0.0.1 mean?
- Can two servers on one machine both use port 80?

More questions: sockets and ports (2)

- Which port does a web server usually use?
- A client never calls bind. Where does its port come from?
- Why are well-known port numbers useful?

More questions: TCP and UDP

- Name two things TCP guarantees that UDP doesn't.
- Which socket type is TCP: SOCK_STREAM or SOCK_DGRAM?
- A game sends each player's position 30 times a second. TCP or UDP?

More questions: the server (1)

- Put these in order: accept, bind, listen, socket.
- Which server call waits until a client connects?
- What does accept return?

More questions: the server (2)

- What's the difference between listen and accept?
- What does INADDR_ANY mean?
- Why do we cast &addr to (struct sockaddr *)?

More questions: byte order

- What is network byte order?
- What does htons stand for?
- A port number arrives from the network. Which function makes it print correctly?

More questions: the client

- Which two calls does a TCP client need before it can send?
- connect fails with Connection refused. What's the likely reason?
- What would you change in 52_client.c to reach a server on another machine?

More questions: send, recv and close

- recv returns 0. What happened?
- send(fd, buf, len, 0) does the same as which file I/O call?
- Why do you have to add the '\0' yourself after recv?

Next Lecture

- (Tutorial Style Lecture): Interprocess Communication (TCP)

- System Calls and Process Management