Socket Programming

CS 353 – Computer Networks Lab

1. Labs 1 and 2 are not graded.
2. All the rest of labs are graded.
3. One week time to evaluation.
4. NetSim software is used.
5. Grading: 50% lab sessions.

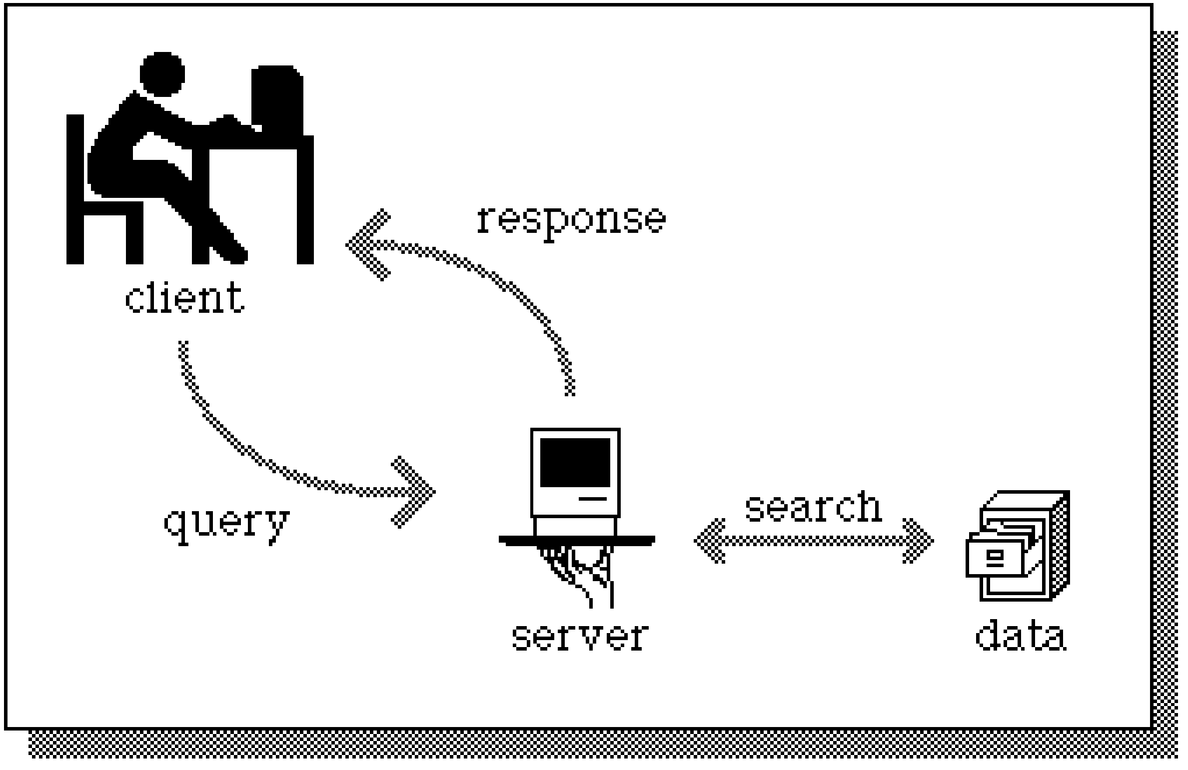
   30% end-sem

   20% lab-quiz

Client-Server Architecture

- In the client server architecture, a machine(refered as client) makes a request to connect to another machine (called as server) for providing some service.
- The services running on the server **run on known ports**(application identifiers) and the client needs to know the address of the server machine and this port in order to connect to the server.
- On the other hand, the server does not need to know about the address or the port of the client at the time of connection initiation.
- The first packet which the client sends as a request to the server contains these informations about the client which are further used by the server to send any information.
- Client acts as the active device which makes the first move to establish the connection whereas the server passively waits for such requests from some client.

Client-Server Architecture



What is a Socket?

- In unix, whenever there is a need for inter process communication within the same machine, we use mechanism like signals or pipes.
- Similarly, when we desire a communication between two applications possibly running on different machines, we need sockets.
- Sockets are treated as another entry in the unix open file table. So all the system calls which can be used for any IO in unix can be used on socket.
- The server and client applications use various system calls to conenct which use the basic construct called socket.
- A socket is one end of the communication channel between two applications running on different machines.

- Steps followed by client to establish the connection:

1) Create a socket
2) Connect the socket to the address of the server
3) Send/Receive data
4) Close the socket

- Steps followed by server to establish the connection:

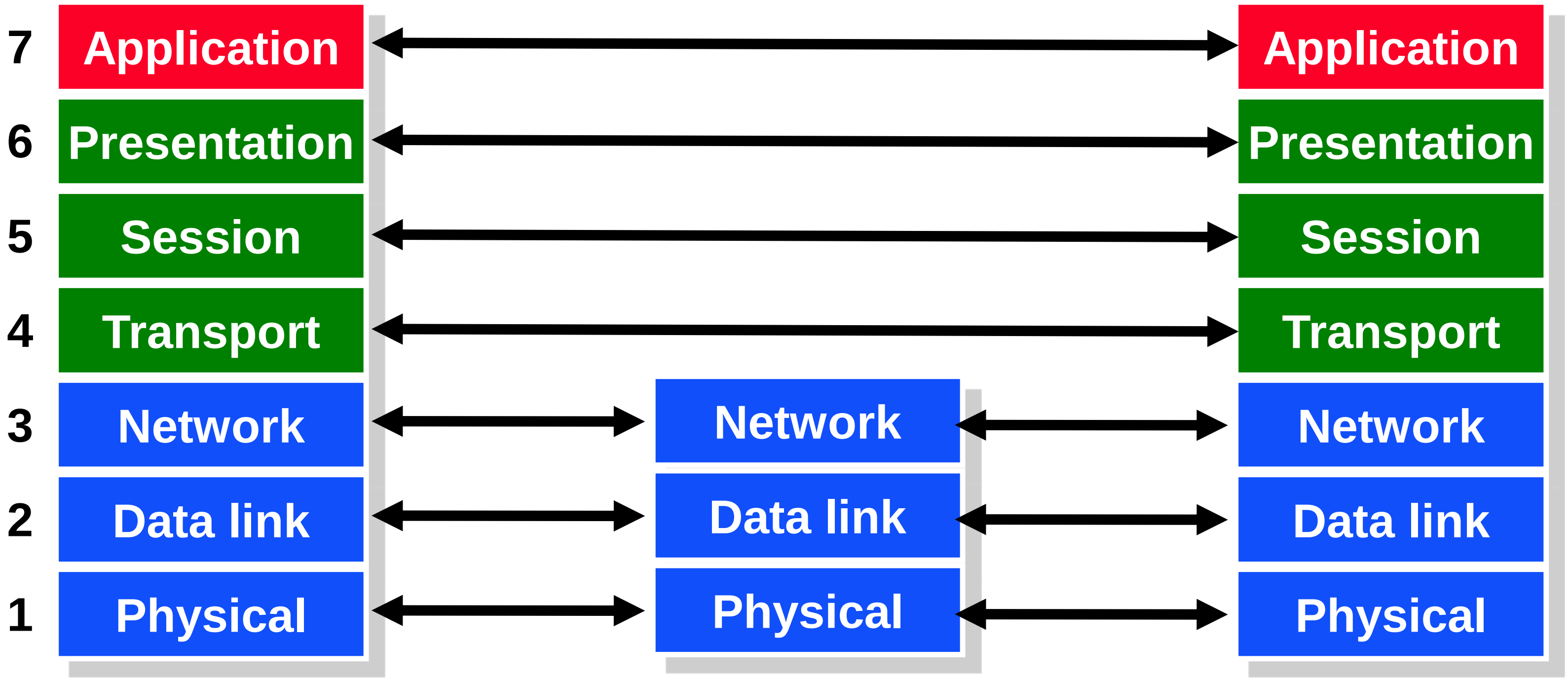
1) Create a socket
2) Bind the socket to the port number known to all clients
3) Listen for the connection request
4) Accept connection request
5) Send/Receive data

Why Socket?

- How can I program a network application?
  - Share data
  - Send messages
  - Finish course projects...

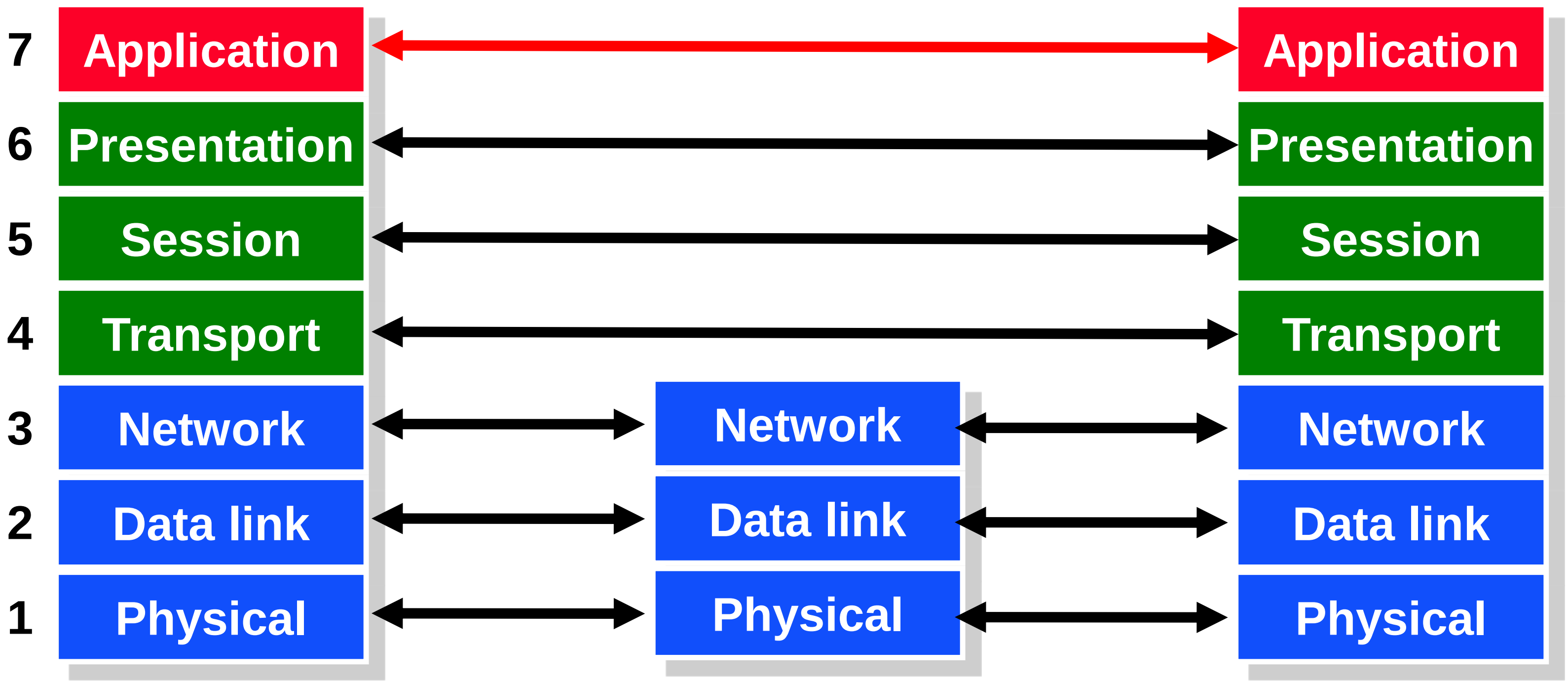
- IPC - Interprocess Communication

Network Layering



Network Layering

- Why layering?



Layering Makes it Easier

- Application programmer
  - Doesn’t need to send IP packets
  - Doesn’t need to send Ethernet frames
  - Doesn’t need to know how TCP implements reliability
- Only need a way to pass the data down
  - Socket is the API to access transport layer functions

What Lower Layer Need to Know?

- We pass the data down. What else does the lower layer need to know?

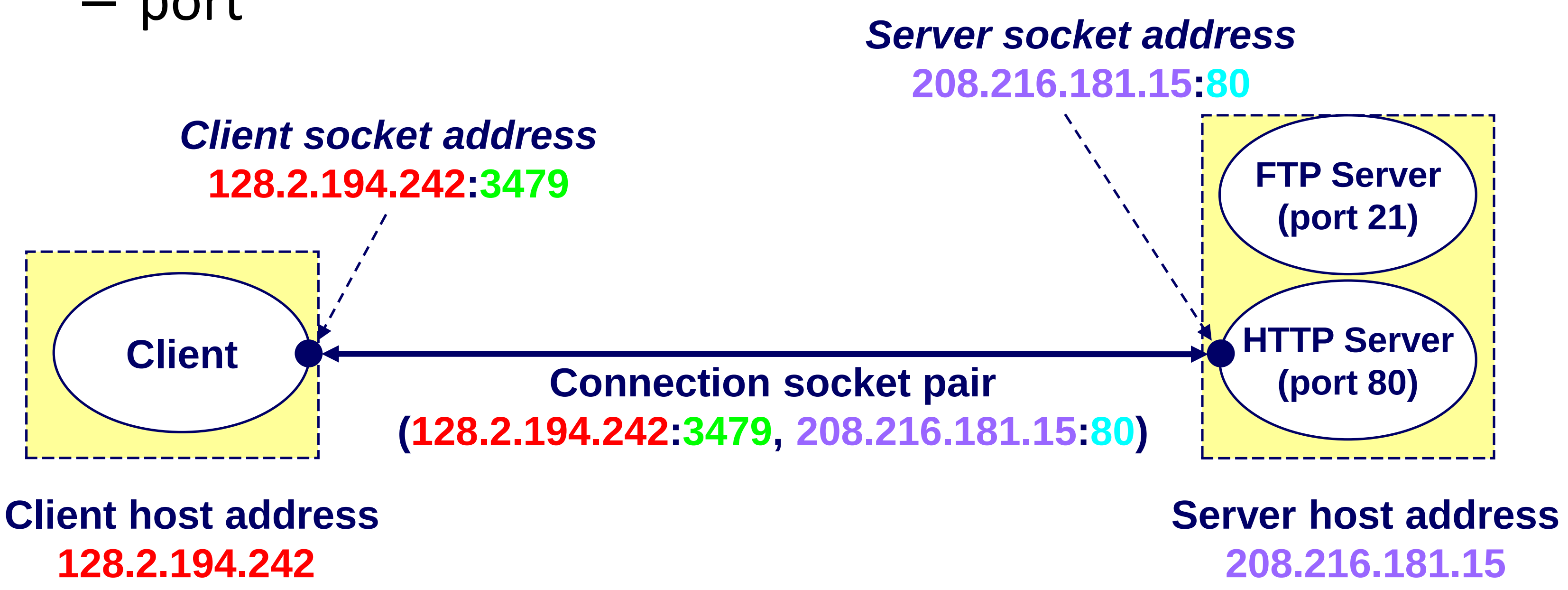
What Lower Layer Need to Know?

- We pass the data down. What else does the lower layer need to know?

- How to identify the destination process?
  - Where to send the data? (Addressing)
  - What process gets the data when it is there? (Multiplexing)

Identify the Destination

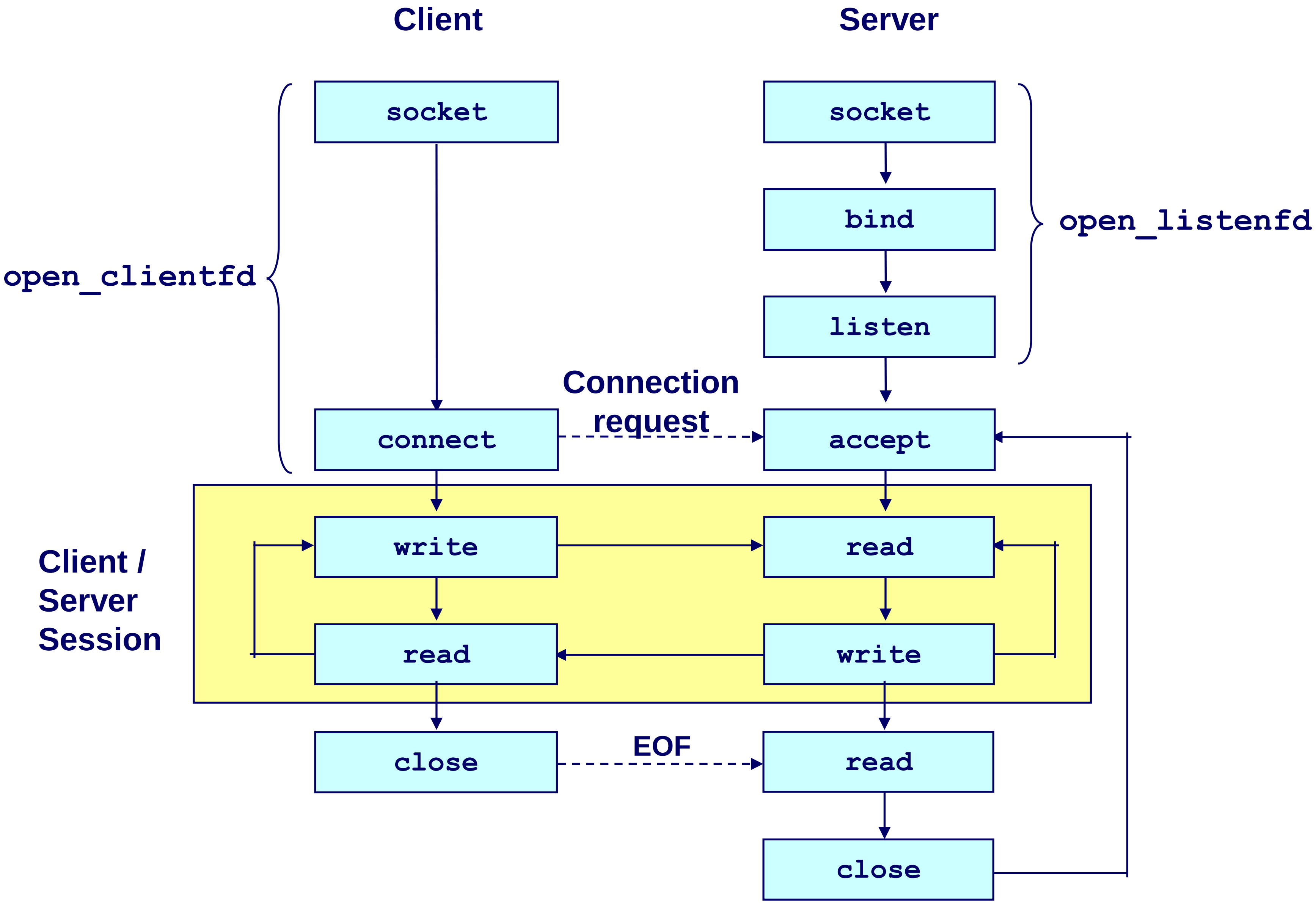
- Addressing
  - IP address
  - hostname (resolve to IP address via DNS)
- Multiplexing
  - port



Sockets

- How to use sockets
  - Setup socket
    - Where is the remote machine (IP address, hostname)
    - What service gets the data (port)
  - Send and Receive
    - Designed just like any other I/O in unix
    - send -- write
    - recv -- read
  - Close the socket

Overview



Client
Server
socket
socket
bind
open_listenfd
listen
open_clientfd
Connection
request
connect
accept
write
read
Client /
Server
Session
read
write
close
EOF
read
close

Basic data structures used in Socket programming

- Socket Descriptor: A simple file descriptor in Unix.

  Int
- Socket Address: This construct holds the information for socket address

```
struct sockaddrs {
  unsigned short    sa_family;    // address family, AF_xxx
 or PF_xxx
  char    sa_data[14];  // 14 bytes of protocol address
};
```

- AF stands for Address Family and PF stands for Protocol Family. In most modern implementations only the AF is being used.

Name	Purpose
AF_UNIX, AF_LOCAL	Local communication
AF_INET	IPv4 Internet protocols
AF_INET6	IPv6 Internet protocols

Basic data structures used in Socket programming

- Socket Descriptor: A simple file descriptor in Unix.

  Int
- Socket Address: This construct holds the information for socket address

```
struct sockaddrs {
   unsigned short    sa_family;    // address family, AF_xxx
  or PF_xxx
   char    sa_data[14];  // 14 bytes of protocol address
};
```

- AF stands for Address Family and PF stands for Protocol Family. In most modern implementations only the AF is being used.

Name	Purpose
AF_UNIX, AF_LOCAL	Local communication
AF_INET	IPv4 Internet protocols
AF_INET6	IPv6 Internet protocols

Step 1 – Setup Socket

- **Both client and server need to setup the socket**
  - *int socket(int domain, int type, int protocol);*
- *domain*
  - AF_INET -- IPv4 (AF_INET6 for IPv6)
- *type*
  - SOCK_STREAM -- TCP
  - SOCK_DGRAM -- UDP
- *protocol*
  - 0
- For example,
  - *int sockfd = socket(AF_INET, SOCK_STREAM, 0);*

Step 2 (Server) - Binding

- **Only server need to bind**
  - *int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);*
- *sockfd*
  - file descriptor socket() returned
- *my_addr*
  - struct sockaddr_in for IPv4
  - cast (struct sockaddr_in*) to (struct sockaddr*)

```
struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};
struct in_addr {
    unsigned long s_addr;  // load with inet_aton()
};
```

What is that Cast?

- bind() takes in protocol-independent (struct sockaddr*)

```
struct sockaddr {
    unsigned short    sa_family;   // address family
    char              sa_data[14]; // protocol address
};
```

  - There are structs for IPv6, etc.

IP Addresses

Assuming that we are dealing with IPv4 addresses, the address is a 32bit integer. Remembering a 32 bit number is not convenient for humans. So, the address is written as a set of four integers seperated by dots, where each integer is a representation of 8 bits.

The representation is like a.b.c.d, where a is the representation of the most significant byte. The system call which converts this representation into Network Byte Order is:

int inet_aton(const char *cp, struct in_addr *inp);

inet_aton() converts the Internet host address cp from the standard numbers-and-dots notation into binary data and stores it in the structure that inp points to. inet_aton returns nonzero if the address is valid, zero if not.

For example, if we want to initialize the sockaddr_in construct by the IP address and desired port number, it is done as follows:

```
struct sockaddr_in sockaddr;
sockaddr.sin_family = AF_INET;
sockaddr.sin_port = htons(21);
inet_aton("172.26.117.168", &(sockaddr.sin_addr));
memset(&(sockaddr.sin_zero), '\0', 8);
```

Step 2 (Server) - Binding contd.

- *addrlen*
  - size of the sockaddr_in

```
struct sockaddr_in saddr;
int sockfd;
unsigned short port = 80;

if((sockfd=socket(AF_INET, SOCK_STREAM, 0) < 0) {        // from back a couple slides
printf("Error creating socket\n");
...
}

memset(&saddr, '\0', sizeof(saddr));             // zero structure out
saddr.sin_family = AF_INET;                       // match the socket() call
saddr.sin_addr.s_addr = htonl(INADDR_ANY);       // bind to any local address
saddr.sin_port = htons(port);                     // specify port to listen on

if((bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) { // bind!
printf("Error binding\n");
...
}
```

What is htonl(), htons()?

- Byte ordering
  - Network order is big-endian
  - Host order can be big- or little-endian
    - x86 is little-endian
    - SPARC is big-endian
- Conversion
  - *htons(), htonl()*: host to network short/long
  - *ntohs(), ntohl()*: network order to host short/long
- What need to be converted?
  - Addresses
  - Port
  - etc.

Little & Big Endian

- Some systems (like x8086) are Little Endian i-e. least signficant byte is stored in the higher address, whereas in Big endian systems most significant byte is stored in the higher address.
- Consider a situation where a Little Endian system wants to communicate with a Big Endian one, if there is no standard for data representation then the data sent by one machine is misinterpreted by the other.
- So standard has been defined for the data representation in the network (called Network Byte Order) which is the Big Endian.
- The system calls that help us to convert a short/long from Host Byte order to Network Byte Order and vice-versa are

1) htons() -- "Host to Network Short"
2) htonl() -- "Host to Network Long"
3) ntohs() -- "Network to Host Short"
4) ntohl() -- "Network to Host Long"

Step 3 (Server) - Listen

- **Now we can listen**
  - *int listen(int sockfd, int backlog);*
- *sockfd*
  - again, file descriptor socket() returned
- *backlog*
  - number of pending connections to queue
- For example,
  - *listen(sockfd, 5);*

Step 4 (Server) - Accept

- **Server must explicitly accept incoming connections**
  - *int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)*
- *sockfd*
  - again... file descriptor socket() returned
- *addr*
  - pointer to store client address, (struct sockaddr_in *) cast to (struct sockaddr *)
- *addrlen*
  - pointer to store the returned size of addr, should be sizeof(*addr)
- For example
  - *int isock=accept(sockfd, (struct sockaddr_in *) &caddr, &clen);*

Put Server Together

```
struct sockaddr_in saddr, caddr;
int sockfd, clen, isock;
unsigned short port = 80;

if((sockfd=socket(AF_INET, SOCK_STREAM, 0) < 0) {   // from back a couple slides
        printf("Error creating socket\n");
        ...
}

memset(&saddr, '\0', sizeof(saddr));            // zero structure out
saddr.sin_family = AF_INET;                             // match the socket() call
saddr.sin_addr.s_addr = htonl(INADDR_ANY);    // bind to any local address
saddr.sin_port = htons(port);                           // specify port to listen on

if((bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) { // bind!
        printf("Error binding\n");
        ...
}

if(listen(sockfd, 5) < 0) {             // listen for incoming connections
        printf("Error listening\n");
        ...
}

clen=sizeof(caddr)
if((isock=accept(sockfd, (struct sockaddr *) &caddr, &clen)) < 0) {  // accept one
        printf("Error accepting\n");
        ...
}
```

What about client?

- Client need not bind, listen, and accept
- **All client need to do is to connect**
  - *int connect(int sockfd, const struct sockaddr *saddr, socklen_t addrlen);*
- For example,
  - *connect(sockfd, (struct sockaddr *) &saddr, sizeof(saddr));*

Domain Name System (DNS)

- What if I want to send data to “www.slashdot.org”?
  - DNS: Conceptually, DNS is a database collection of host entries

```
struct hostent {
    char        *h_name;      // official hostname
    char        **h_aliases;   // vector of alternative hostnames
    int   h_addrtype;    // address type, e.g. AF_INET
    int   h_length;      // length of address in bytes, e.g. 4 for IPv4
    char        **h_addr_list;          // vector of addresses
    char        *h_addr;                // first host address, synonym for h_addr_list[0]
};
```

- hostname -> IP address
  - *struct hostent *gethostbyname(const char *name);*

- IP address -> hostname
  - *struct hostent *gethostbyaddr(const char *addr, int len, int type);*

Put Client Together

```
struct sockaddr_in saddr;
struct hostent *h;
int sockfd, connfd;
unsigned short port = 80;

if((sockfd=socket(AF_INET, SOCK_STREAM, 0) < 0) {   // from back a couple slides
        printf(“Error creating socket\n”);
        ...
}

if((h=gethostbyname(“www.slashdot.org”)) == NULL) { // Lookup the hostname
        printf(“Unknown host\n”);
        ...
}

memset(&saddr, '\0', sizeof(saddr));                // zero structure out
saddr.sin_family = AF_INET;                         // match the socket() call
memcpy((char *) &saddr.sin_addr.s_addr, h->h_addr_list[0], h->h_length); // copy the address
saddr.sin_port = htons(port);                       // specify port to connect to

if((connfd=connect(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) { // connect!
        printf(“Cannot connect\n”);
        ...
}
```

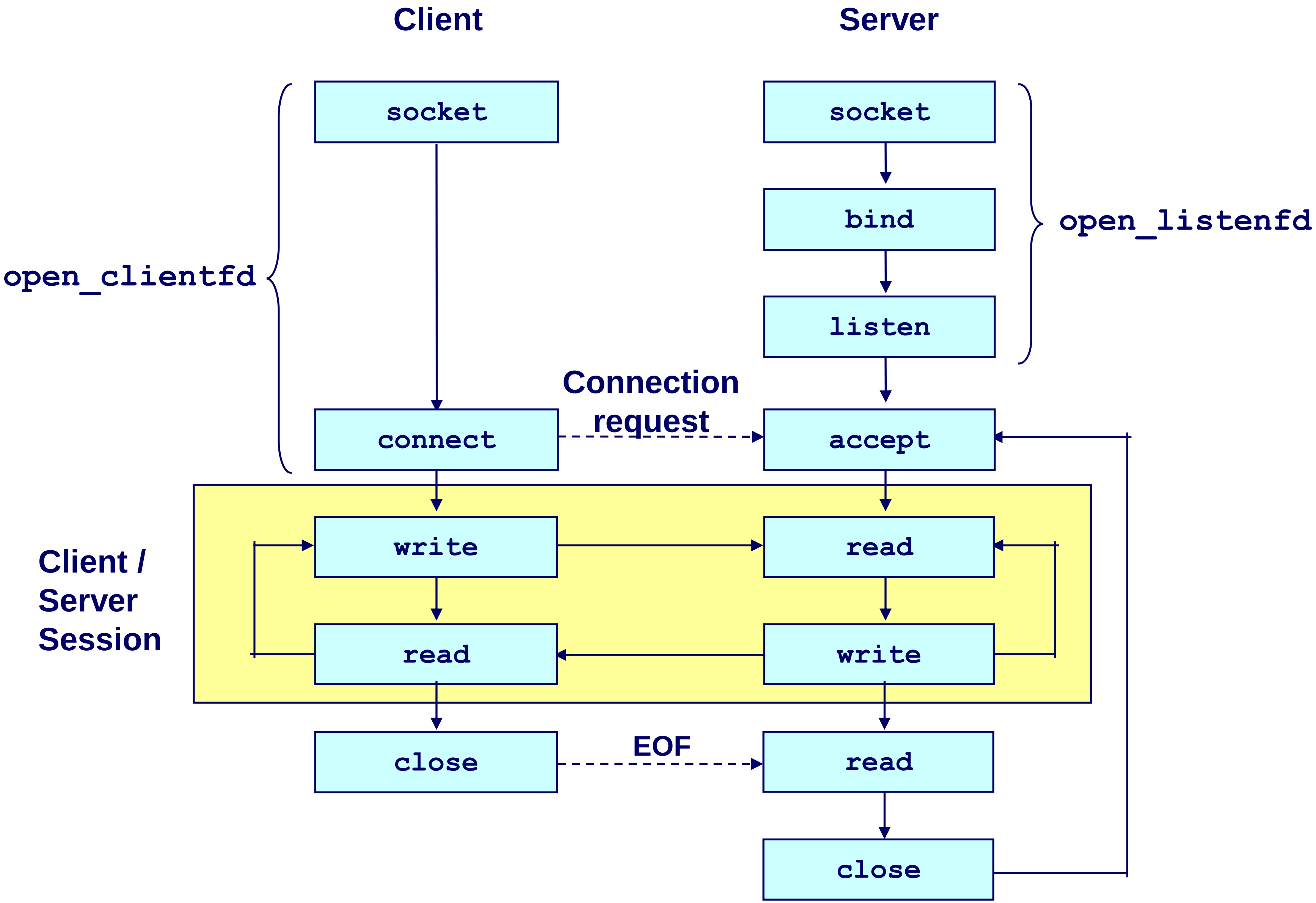
We Are Connected

- Server accepting connections and client connecting to servers
- Send and receive data
  - *ssize_t read(int fd, void *buf, size_t len);*
  - *ssize_t write(int fd, const void *buf, size_t len);*
- For example,
  - *read(sockfd, buffer, sizeof(buffer));*
  - *write(sockfd, “hey\n”, strlen(“hey\n”));*

TCP Framing

- TCP does NOT guarantee message boundaries
  - IRC commands are terminated by a newline
  - But you may not get one at the end of read(), e.g.
    - One Send “Hello\n”
    - Multiple Receives “He”, “llo\n”
  - If you don’t get the entire line from one read(), use a buffer

Revisited

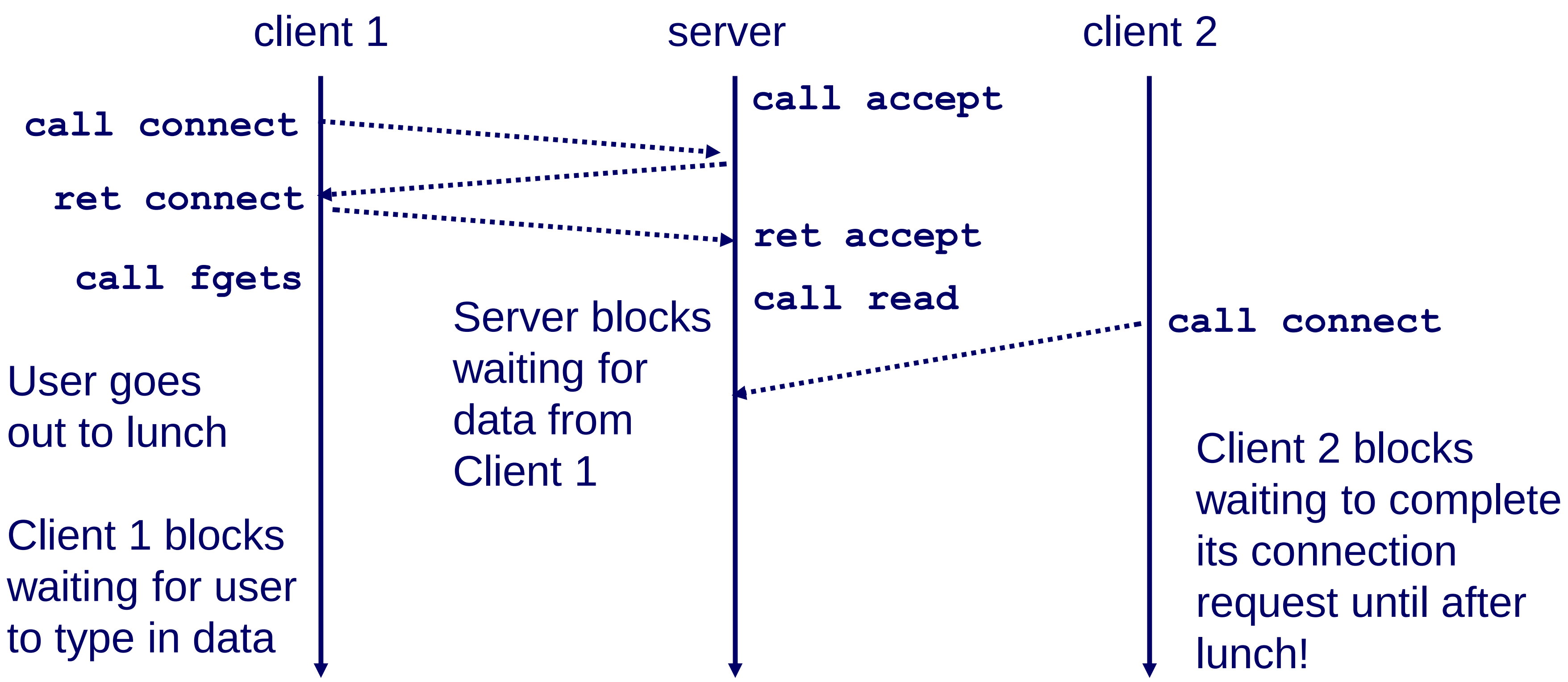

Client
Server
socket
socket
bind
listen
open_listenfd
open_clientfd
Connection request
connect
accept
Client / Server Session
write
read
read
write
EOF
close
read
close

Close the Socket

- Don’t forget to close the socket descriptor, like a file
  - *int close(int sockfd);*

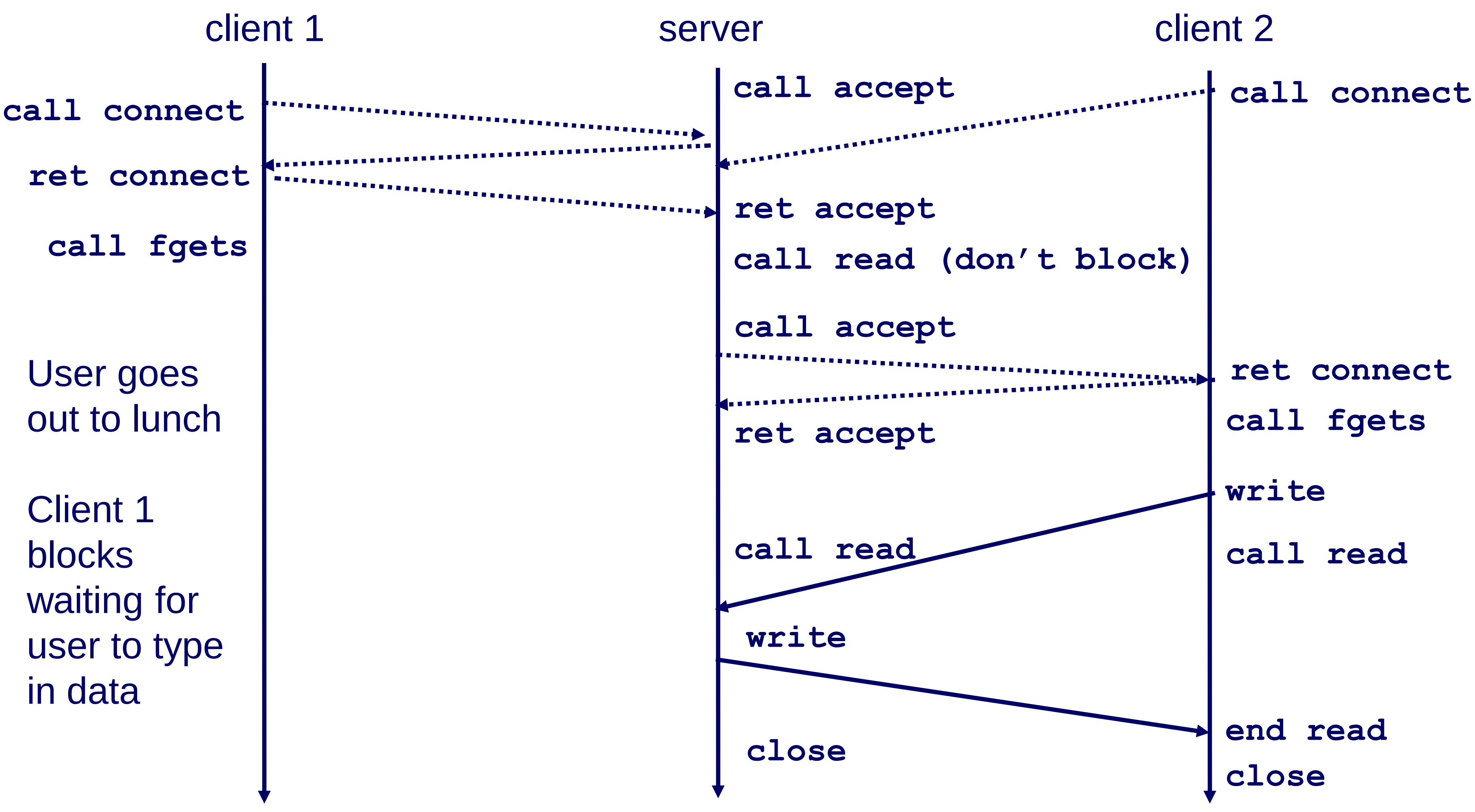
- Now server can loop around and accept a new connection when the old one finishes
- What’s wrong here?

Server Flaw



Taken from D. Murray, R. Bryant, and G. Langale 15-441/213 slides

Concurrent Servers



Taken from D. Murray, R. Bryant, and G. Langale 15-441/213 slides