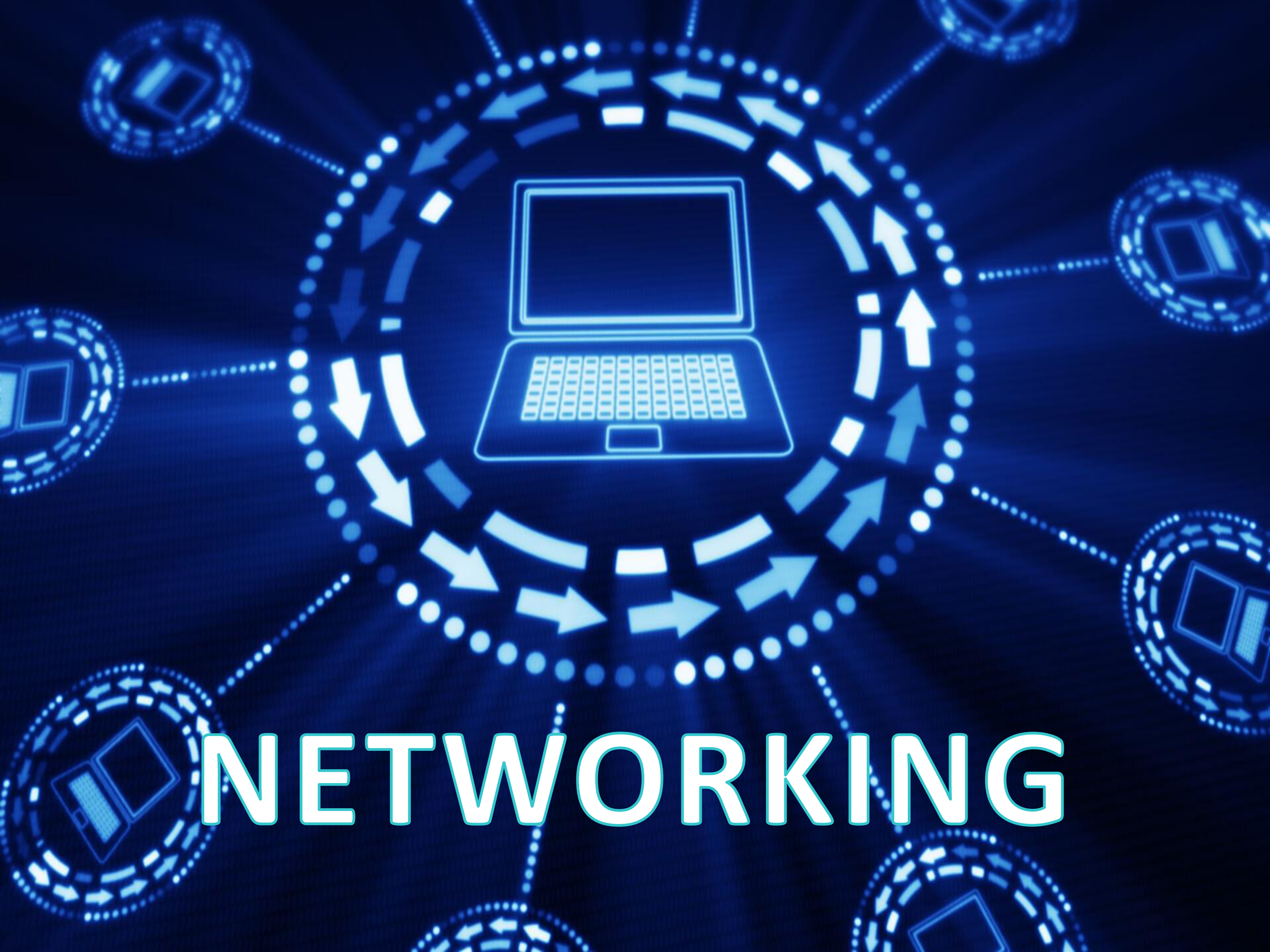

NETWORKING

CLIENT/SERVER MODEL

- How 2 application programs make contact

Server

- Starts first
- Passively waits for contact from a client at a prearranged location
- Responds to requests

Client

- Starts second
- Actively contacts a server with a request
- Waits for a response from server

- Client-server paradigm: form of communication used by all network applications

CHARACTERISTICS OF A CLIENT

- Arbitrary application program
- Becomes client temporarily
- Can also perform other computations
- Invoked directly by user
- Runs locally on user's computer
- Actively initiates contact with a server
- Contacts one server at a time

CHARACTERISTICS OF A SERVER

- Special-purpose, privileged program
- Dedicated to providing one service
- Can handle multiple remote clients simultaneously
- Invoked automatically when system boots
- Executes forever
- Needs powerful computer and operating system
- Waits passively for client contact
- Accepts requests from arbitrary clients

SOCKETS

- An interface between applications and network
  - The application creates a socket
  - The socket *type* dictates the style of communications
    - Reliable vs best effort
    - Connection oriented vs connectionless
- Once configured the application can
  - Pass data to the socket for network transmission
  - Receive data from the socket (transmitted through the network by some other host)
- The Berkeley Sockets API, originally developed as part of BSD, is the most popular API for C/C++ over TCP/IP

SOCKETS

- A socket is like a file
  - You can read/write from/to the network like you would a file
- For connection-oriented communication
  - Servers (passive open) do *listen* and *accept* operations
  - Clients (active open) do *connect* operations
  - Both sides can then do *read* and/or *write* (or *send/recv*)
  - Each side must close
- Connectionless uses *sendto* and *recfrom*

SOCKET AND SOCKET LIBRARIES

- In Unix, socket procedures are system calls
- One some other systems, socket procedures are not part of the OS:
  - Instead, they are implemented as a library, linked into the application object code (ex DLL)
  - Typically, this DLL makes calls to similar procedures that are part of the native operating system
  - This can be referred to as a *socket library: "A socket library simulates Berkeley sockets on OS's where the underlying OS networking calls are different from Berkeley sockets"*

TWO ESSENTIAL TYPE OF SOCKETS

- **`SOCK_STREAM`**
  - aka TCP
  - Reliable delivery
  - In-order guarantee
  - Connection-oriented
  - Bidirectional

- **`SOCK_DGRAM`**
  - aka UDP
  - Unreliable delivery
  - No order guarantees
  - No notion of “connection” – app indicates destination for each packet
  - Can send or receive

TCP/IP SOCKETS

- Sockets are able to open a connection on a port and transmit whatever data to and from the recipient
- To establish a connection through a port, the following tuple must be totally defined in the system:

  <protocol, local-addr, local-port, foreign-addr, foreign-port>

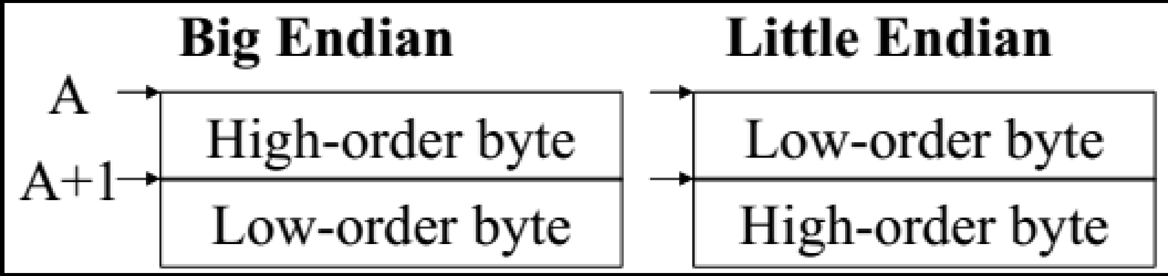
- In a server-client setup, the server provides the local attributes and then waits for a connection from the client

TCP/IP SOCKETS

- A connection from the client provides all the necessary data to fill this tuple
- The client initiates the connection to the remote server, using a known IP address and port number on which the server is listening
- This provides all the data for the local part of the tuple
- One the whole internet, each of these tuples must be unique

NETWORK BYTE ORDERING

- Some computer architectures are big endian and some are little endian
- For example, Intel architectures are little endian



- To be able to communicate on the internet, a network byte ordering has been defined
- Network byte ordering is big endian for 16 and 32-bit integers
- When passing values to be used at the network layers, we need to make the appropriate conversions

NETWORK BYTE ORDERING

- The following library functions handle the potential byte order differences between different computer architectures
- ‘h’ stands for host while ‘n’ stands for network values

```
#include <sys/types.h>
#include <netinet/in.h>

u_long  htonl(u_long hostlong);
u_short htons(u_short hostshort);
u_long  ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

ADDRESS CONVERSION

- Given an IP address stored inside a string, `inter_addr` returns the address in network byte order
- `inet_ntoa` performs the opposite operation
- Every subsequent call to `inet_ntoa` overwrites the statically returned string

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(const char* ptr);
char *inet_ntoa(struct in_addr inaddr);
```

BYTE OPERATIONS

- Whenever a series of bytes have to be copied, use `bcopy`
- `bcopy` is better suited than `strcpy` since a series of bytes might contain '`/0`' inside it.
- When using network structures, be sure to apply `bzero` before using them

```
#include <string.h>

void bcopy(char *src, char *dest, int nbytes);
void bzero(char *dest, int nbytes);
int bcmp(char *ptr1, char *ptr2, int nbytes);
```

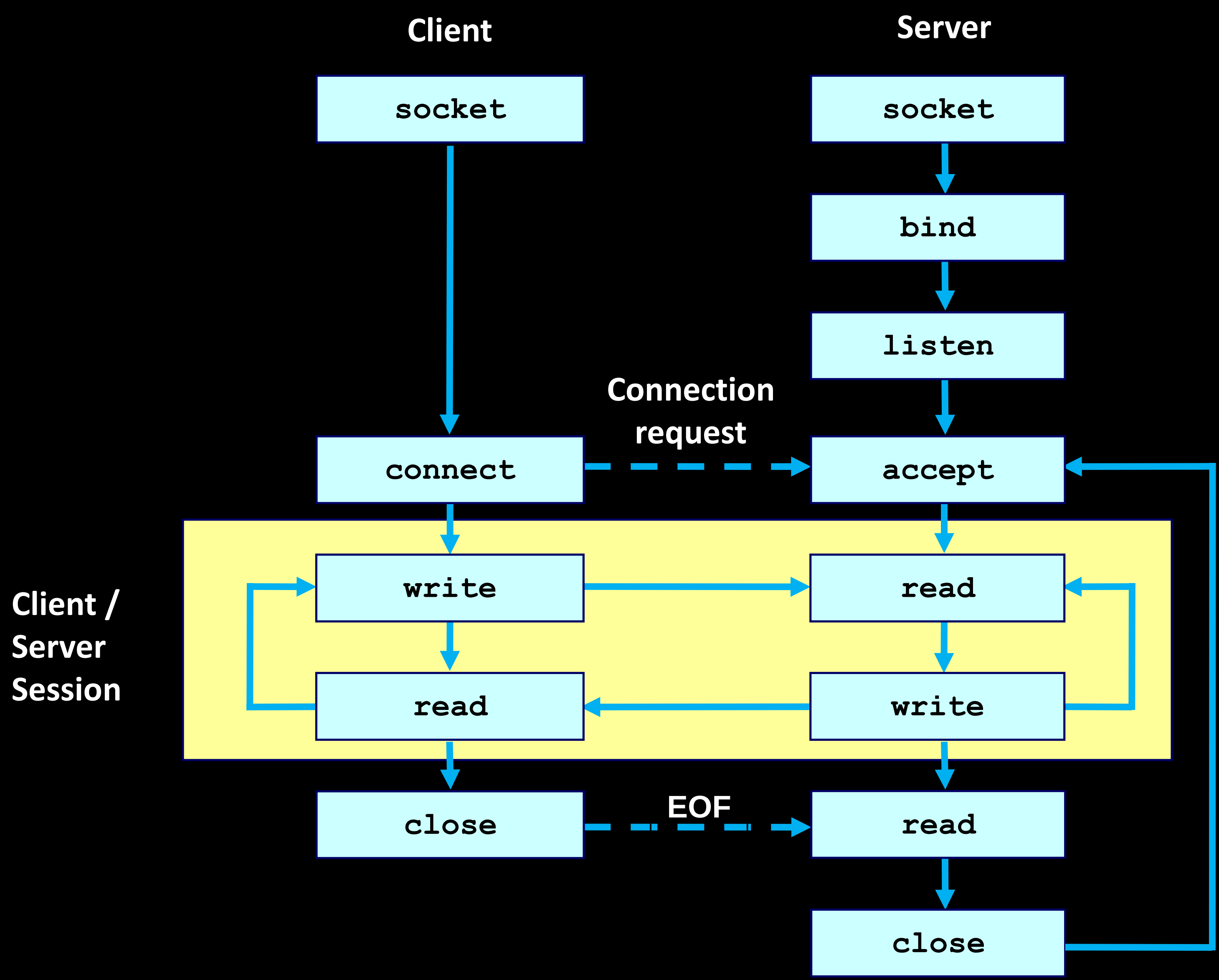
SOCKET ACCESS

- When using sockets apply the following to your programs:

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
```

- To compile programs using sockets on SunOS systems use: `gcc -lnsl -lsocket myprog.c`
- Some programs such as `ping`, `netstat` and `traceroute` are available in the directory `/usr/sbin` or `/bin`

TCP CLIENT SERVER



STEP 1: SETUP SOCKET

- Both client and server need to setup the socket

```
int socket(int domain, int type,
           int protocol);
  return socket descriptor or -1 on error
```

- `domain:`
  - `AF_INET` (`AF_INET6` for IPv6)
- `type:`
  - `SOCK_STREAM` for TCP
  - `SOCK_DGRAM` for UDP
- `protocol`
  - `0`

SOCKET OPTIONS

```
int setsockopt(int sockfd, int level,
               int option, const void *val,
               socklen_t len);
int getsockopt(int sockfd, int level,
               int option, void *restrict val,
               socklen_t restrict lenp);
     Return 0 if OK, 1 on error
```

- The `level` argument identifies the protocol to which the option applies (eg `IPPROTO_TCP`)
- If option is generic then `SOL_SOCKET` is used
- The `val` argument points to a data structure

SOME SOCKET OPTIONS

`SO_ACCEPTCONN`	Return whether a socket is enabled for listening
`SO_DEBUG`	Debugging in network drivers
`SO_DONTROUTE`	Bypass normal routing
`SO_ERROR`	Return and clear pending socket errors
`SO_KEEPALIVE`	Periodic keep-alive messages
`SO_RCVBUF`	Size of the receive buffer
`SO_RCVTIMEO`	Timeout value for a socket receive call
`SO_REUSEADDR`	Reuse address in bind
`SO_SENDBUF`	Size of the send buffer
`SO_SNDTIMEO`	Timeout value for a socket send call

STEP 2: BINDING

- Only server needs to bind (optional for client)

```
int bind(int sockfd,
         struct sockaddr *myaddr,
         int addrlen);
   Returns -1 on error
```

- `sockfd`:
  - File descriptor `socket()` returned
- myaddr:
  - `struct sockaddr_in` for IPv4
  - Cast (`struct sockaddr_in*`) to (`struct sockaddr *`)
- `addrlen`
  - Size of `sockaddr_in`

STEP 2: BINDING

- All sockets of all protocols and type use `struct sockaddr` as a base communication structure
- This structure is a general structure which is then applied type-casted to the specific protocol or type required

```
struct sockaddr
{
    u_short sa_family;    // AF_xxxx
    char    sa_data[14]; // protocol specific
}
```

STEP 2: BINDING

- For internet access, `struct sockaddr_in` is used. It is applied wherever `struct sockaddr` is required.
- Be sure to `bzero()` the structure before using it

```
struct sockaddr_in
{
    short    sin_family;          // AF_INET
    u_short sin_port;             // 16-bi port no.
    struct in_addr sin_addr; // 32-bit IP in NBO
    char     sin_zero[8];         // Set to 0
}

struct in_addr {
    u_long  s_addr;  // 32-bit IP in NBO
}
```

STEP 2: BINDING

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

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

bzero((char *)&saddr, 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

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

STEP 3: LISTENING (SERVER)

- Now we can listen

```
int listen(int sockfd, int backlog);
    Returns -1 on error
```

- After a server binds a specific socket to an IP address and a port, it registers the socket to `listen()` for connections
- The backlog specifies the number of pending connections to queue until the server can handle them
- If the backlog is full, new connection requests are simply ignored

STEP 4: ACCEPT (SERVER)

- Server must explicitly accept incoming connections

```
int accept(int sockfd,
           struct addr *client,
           int *addrlen);
     Return new socket descriptor or
     -1 on error
```

- `addr:`
  - Pointer to store client address
- `addrlen:`
  - Pointer to store the returned size of `addr`, should be `sizeof(*addr)`

STEP 4: ACCEPT

- `accept()` blocks until a connection request exists on the queue of pending connections
- `accept()` completes the foreign part of the socket tuple for the server
- All connection requests are accepted, so it is up to the server to close a connection from an unwanted client
- `accept()` returns a new socket descriptor to access the new connection
- The original socket is left open to be able to accept more connections

SERVER

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

if((sockfd = socket(AF_INET, SOCK_STREAM, 0) < 0) {    // Create socket
  printf("Error creating socket\n");

bzero((char *)&saddr, 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");

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

DOMAN NAME SYSTEM

- If a connection is required to “www.slashdot.org”, the name needs to be converted to an IP address

```
struct hostnet {
 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)`

CLIENT

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

if((sockfd=socket(AF_INET, SOCK_STREAM, 0) < 0)      // create socket
 printf("Error creating socket\n");

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

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);
saddr.sin_port = htons(port);                   // specify port to connect to

// connect
if((connfd=connect(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0)
    printf("Cannot connect\n");
```

SENDING AND RECEIVING DATA

- The usual `read` and `write` system calls are used to access sockets
- The only difference in these system calls is that the socket descriptor is used instead of the normal file descriptor
- All writes on a socket will block until the data will be sent through the other host, but not until that process reads it
- All reads from a socket will block until some data is available. `read` will return the number of bytes read, which may be less than requested

SENDING AND RECEIVING DATA

```
int recv(int sockfd, void *buff, size_t mbytes,
         int flags)

int send(int sockfd, void *buff, size_t mbytes,
         int flags)
```

- Same as `read()` and `write()` but for flags
  - `MSG_DONTWAIT` (non-blocking send)
  - `MSG_OOB` (out of band data, 1 byte sent ahead
  - `MSG_PEEK` (look, but don't remove)
  - `MSG_WAITALL` (don't give me less than max)
  - `MSG_DONTROUTE` (bypass routing table)

TCP FRAMING

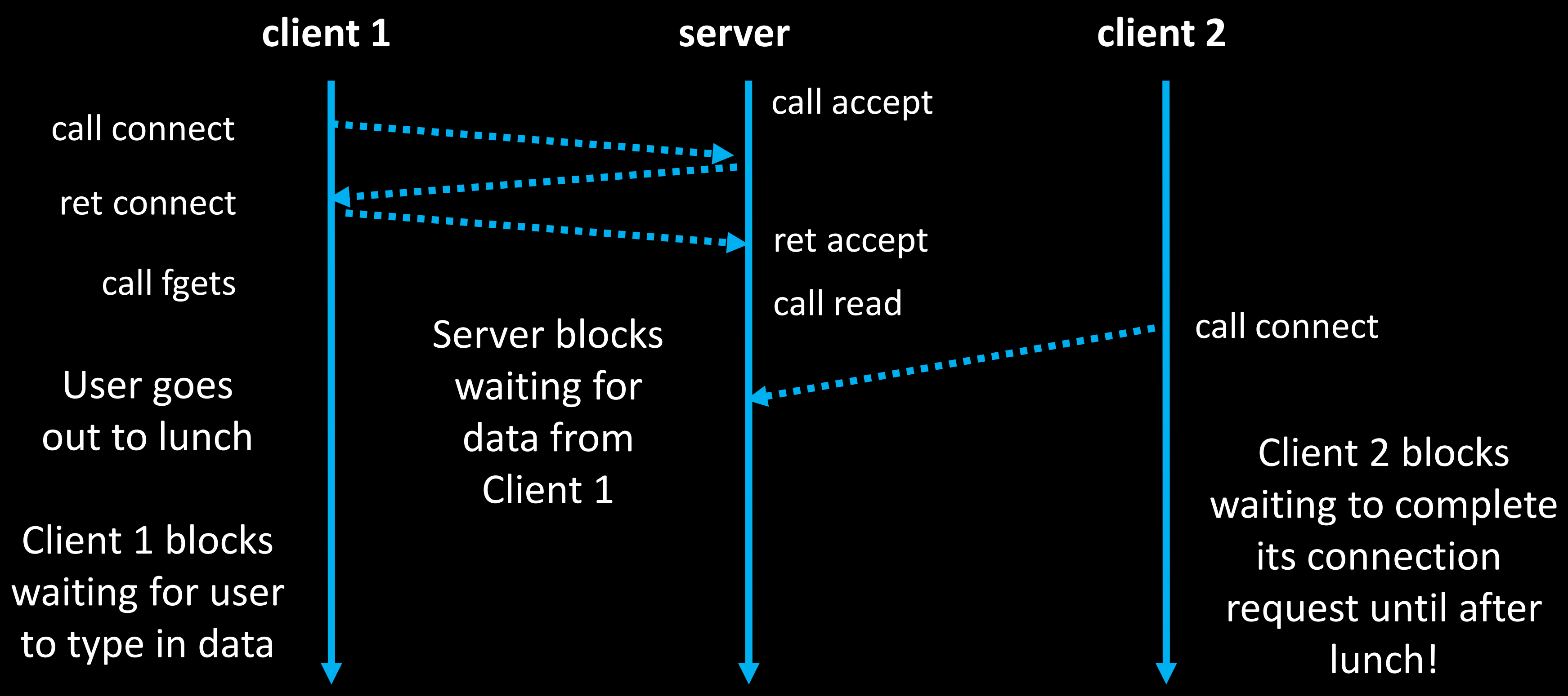
- TCP does not guarantee message boundaries
  - For example, strings in IRC commands are generally terminated by a newline
  - But you may not get one at the end of `read()`:
    - Example, one send (“`Hello\n`”), multiple receives (“`He`”, “`llo\n`”)
  - If you don’t get entire line from one `read()`, use a buffer

CLOSING THE SOCKET

- Calling `close()` will terminate the connection
- If there is any pending data on the socket, the system will try to send it through
- Any process (client or server) trying to access, read from or write to a broken stream will receive the SIGPIPE signal
- The SIGPIPE signal normally terminates the program. Handling this signal makes your program fault tolerant to an abrupt loss of connection

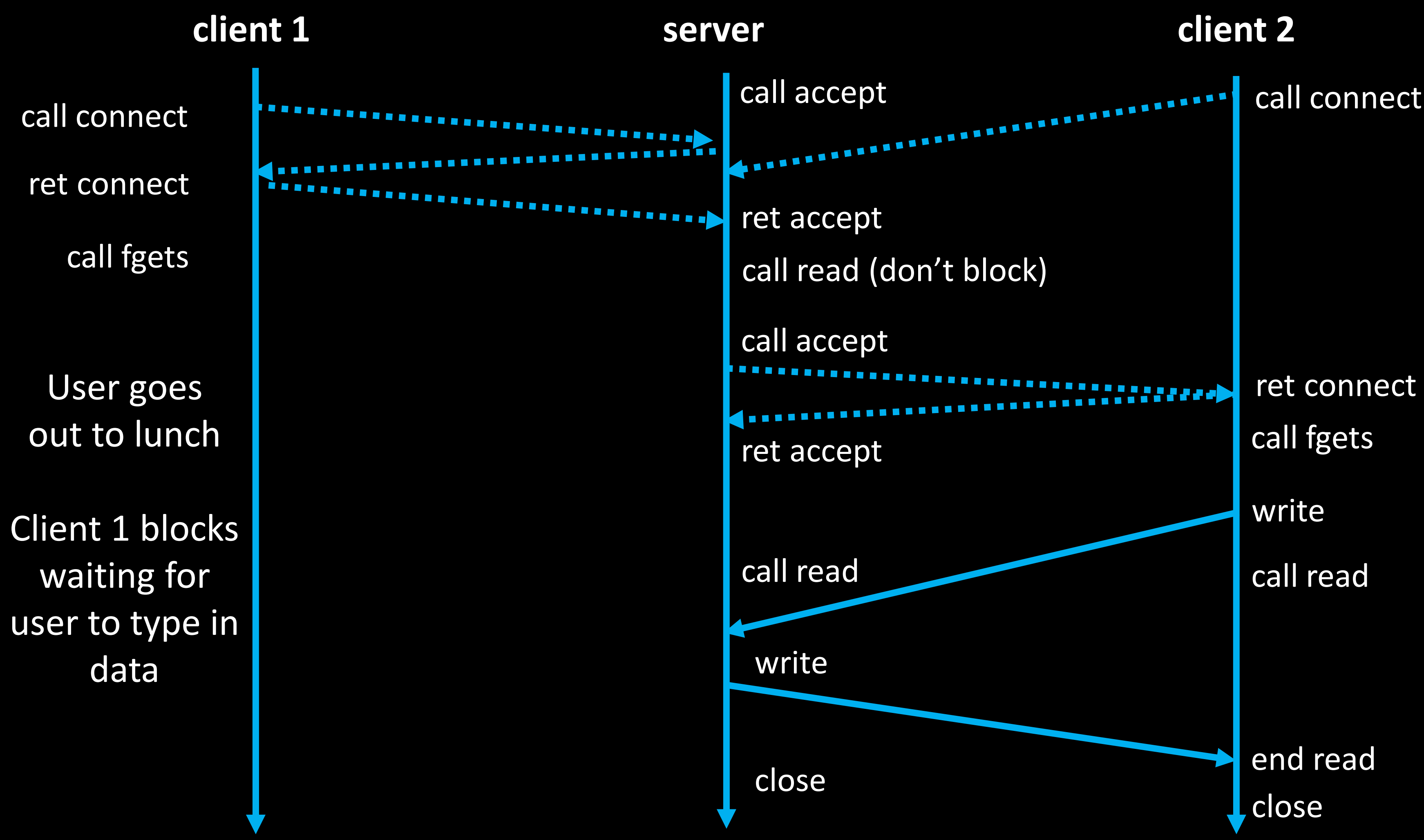
```
int close(int sockfd);
    Returns -1 on error
```

SERVER FLAW



client 1
server
client 2
call accept
call connect
ret connect
ret accept
call fgets
call read
Server blocks waiting for data from Client 1
call connect
User goes out to lunch
Client 2 blocks waiting to complete its connection request until after lunch!
Client 1 blocks waiting for user to type in data

CONCURRENT SERVERS



ITERATIVE AND CONCURRENT SERVERS

- There are two types of servers depending on their behaviour after the `accept` call:
  - Concurrent: After `accept`, a new child is forked which closes the original socket and handles the new connection using the new socket. Meanwhile the parent closes the new socket and calls `accept` again to wait for a new connection
  - Iterative: After `accept`, the server handles the new connections, closes it and then call `accept` again

SELECT

- Monitor multiple descriptors
  - Setup sets of sockets to monitor
  - `select()`: blocking until something happens
  - "Something" could be:
    - Incoming connection: `accept()`
    - Clients sending data: `read()`
    - Pending data to send: `write()`
    - Timeout

SELECT

- Allowing address reuse

```
int sock, opts=1;

sock = socket(...);

setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opts,
sizeof(opts));
```

- Then we set the sockets to non-blocking

```
// Get current options
if((opts = fcntl(sock, F_GETFL)) < 0)
      printf("Error...\n");

// Don't clobber your old settings
opts = (opts | O_NONBLOCK);
if(fcntl(sock, F_SETFL, opts) < 0) {
      printf("Error...\n");

bind(...);
```

CONCURRENCY: STEP 2

- Monitor sockets with `select()`

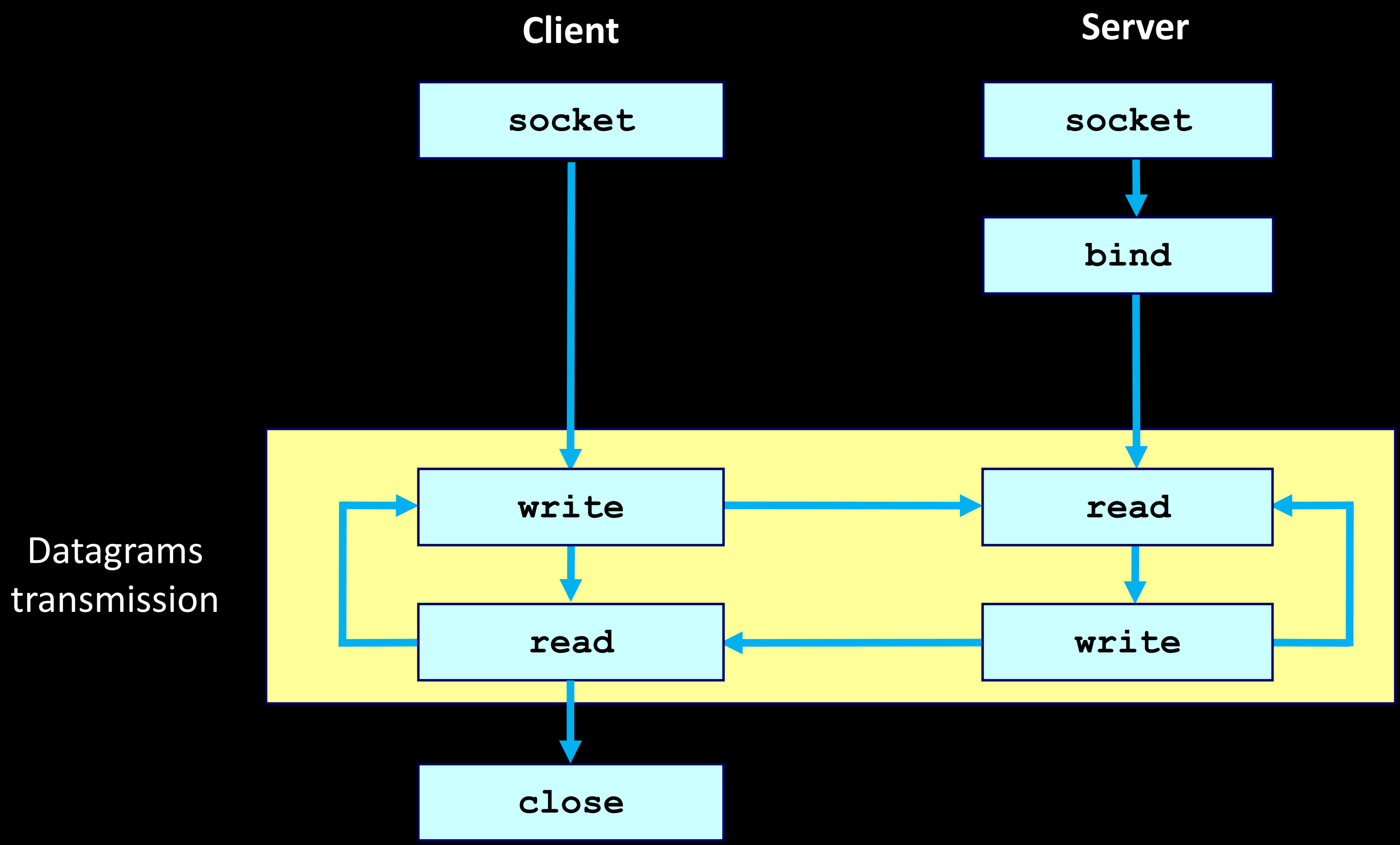
```
int select(int maxfd, fd_set *readfds,
           fd_set *writefds, fd_set *exceptfds,
           const struct timespec *timeout);
```

- `maxfd`: maximum file descriptors + 1
- `fd_set`: bit vector with `FD_SETSIZE` bits
  - `readfds`: bit vector of read descriptors to monitor
  - `writefds`: bit vector of write descriptors to monitor
  - `exceptfds`: set to NULL
- `timeout`: how long to wait without activity before returning

BIT VECTORS

- `void FD_ZERO(fd_set *fdset);`
  - clear out all bits
- `void FD_SET(int fd, fd_set *fdset);`
  - set one bit
- `void FD_CLR(int fd, fd_set *fdset);`
  - clear one bit
- `int FD_ISSET(int fd, fd_set *fdset);`
  - test whether `fd` bit is set

UDP CLIENT SERVER



- No “handshake”
- No simultaneous close
- No need for threading for concurrent servers

SENDING AND RECEIVING DATAGRAMS

```
int recvfrom(int sockfd, void *buff, size_t mbytes,
             int flags, struct sockaddr *from,
             socklen_t *addrlen);

int sendto(int sockfd, void *buff, size_t mbytes,
           int flags, const struct sockaddr *to,
           socklen_t addrlen);
```

- Same as `recv()` and `send()` but for addr
  - `recvfrom` fills in address of where packet came from
  - `sendto` requires address of where packet is being sent

UNIX DOMAIN SOCKETS

- Use the same connection protocol as internet sockets, yet are used to communicate on the same system
- Another form of IPC limited to the same host machine
- All data flow is reliable since data is redirected in the kernel
- Instead of IP addresses and port, pathname of files are used
- The file referenced is created in some system, yet this is not necessary

UNIX DOMAIN SOCKETS

- One cannot `open` a socket file using the open system call
- A socket file has type `S_IFSOCK` and can be tested with the `S_ISSOCK()` macro in conjunction with the `fstat()` system call
- `struct sockaddr_un` is used whenever `struct sockaddr` is required
- All Unix domain sockets use the `<sys/un.h>` header file
- `sun_path` is a null terminated string which is used for the path to be used

UNIX DOMAIN SOCKETS

- To create a UNIX socket, the socket call is used with family set to `AF_UNIX`, type to `SOCK_STREAM` and protocol equal to 0
- The usual calls to `bind`, `connect`, `listen` and `accept` are used to open stream connection using `struct sockaddr_un`

```
struct sockaddr_un
{
    short sun_family;     // AF_UNIX
    char  sun_path[108]; // pathname
}
```

UNIX DOMAIN SOCKETS

- A connection is opened between two sockets, where each socket is associated with a different pathname
- On some systems, socket creation is allowed depending on access rights to the pathname's directory
- Closing a UNIX socket will remove the file from the system
- `read`, `write` and all other system calls we used for internet sockets are valid for UNIX sockets