

[Introduction to Unix Network Programming]

Reference: Stevens Unix
Network Programming

[Network Programming]

- Key Components:
 - Internet protocols
 - IP, TCP, UDP, etc
 - Sockets
 - API - application programming interface
- Why focus on the Internet?
 - Internet Protocol (IP)
 - IP is standard
 - allows a common namespace across most of Internet
 - reduces number of translations, which incur overhead
 - Sockets
 - reasonably simple and elegant, Unix interface



Network Programming with Sockets

- Socket
 - Host-local, application-created, OS-controlled
Application process can both send and receive
messages to/from another application process
- Sockets API
 - A transport layer service interface
 - Introduced in 1981 by BSD 4.1
 - Implemented as library and/or system calls
 - Similar interfaces to TCP and UDP
 - Also interface to IP (for super-user); “raw sockets”



[Beej's Guide]

- How-to guide on network programming using Internet sockets, or "sockets programming"

<http://beej.us/guide/bgnet/>

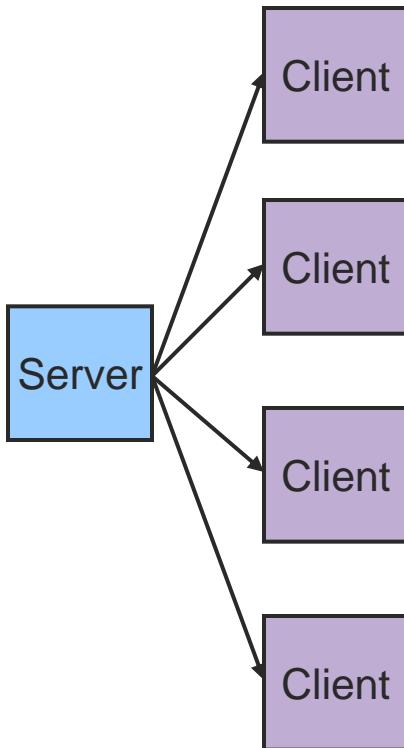


[Outline]

- Client-Sever Model
- TCP Connection
- UDP Services
- Addresses and Data
- Sockets API
- Example



Client-Server Model



- Asymmetric Communication
 - Client sends requests
 - Server sends replies
- Server/Daemon
 - Well-known name
 - Waits for contact
 - Processes requests, sends replies
- Client
 - Initiates contact
 - Waits for response

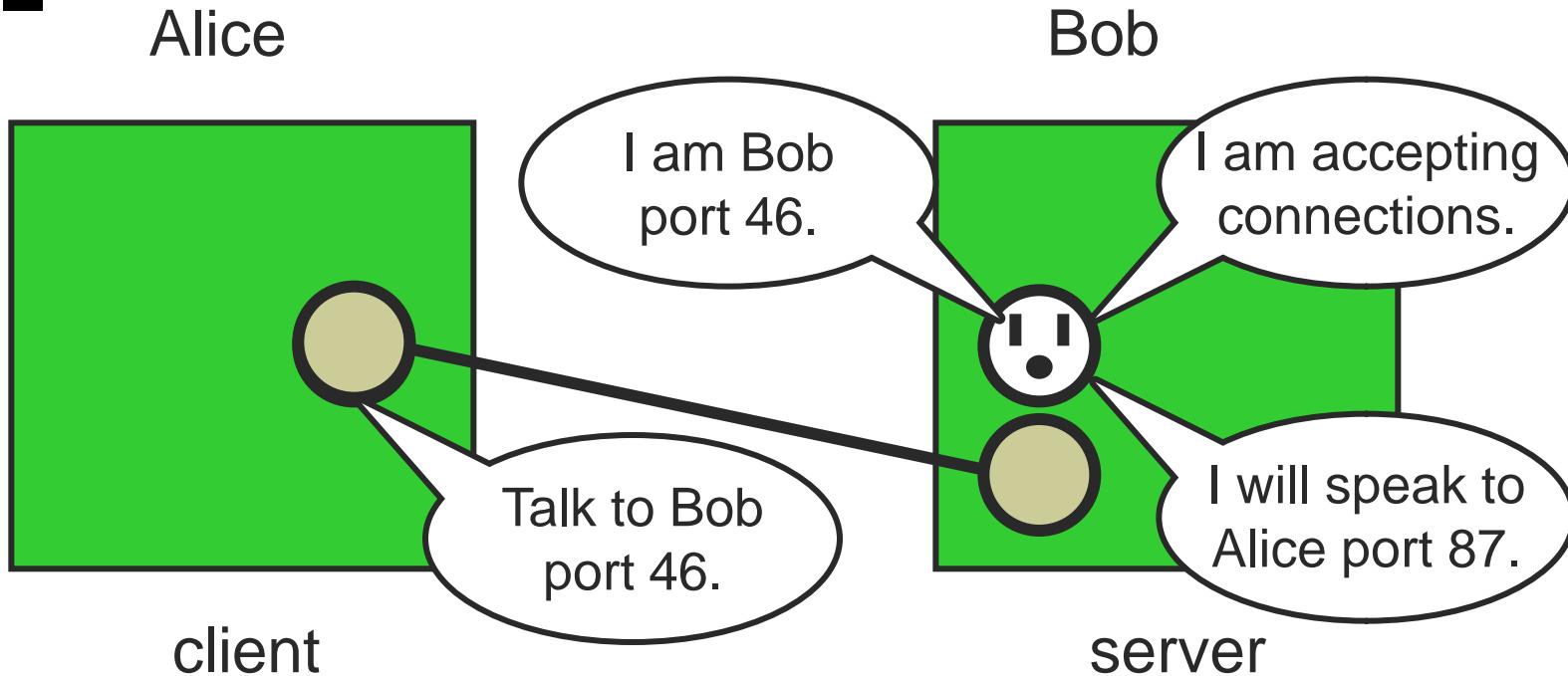


Client-Server Model

- Client contacts server
 - Server process must first be running
 - Server must have created socket that accepts client's contact
- Client: To initiate contact
 - Create client-local TCP socket
 - Specify IP address, port number of server process
 - When client creates socket: client TCP establishes connection to server TCP
- Server: When contacted by client
 - Create new socket for server process to communicate with client
 - Allows server to talk with multiple clients
 - Source port numbers used to distinguish clients



[Example Client-Server Setup]



resulting TCP connection identified by
(Alice:87, Bob:46)



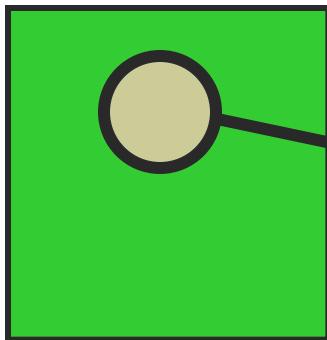
Client-Server Model

- Service Model
 - Concurrent
 - Server processes multiple clients' requests simultaneously
 - Sequential
 - Server processes only one client's requests at a time
 - Hybrid
 - Server maintains multiple connections, but processes responses sequentially
- Client and server categories are not disjoint
 - A server can be a client of another server
 - A server can be a client of its own client
- Examples
 - Web
 - FTP
 - Telnet



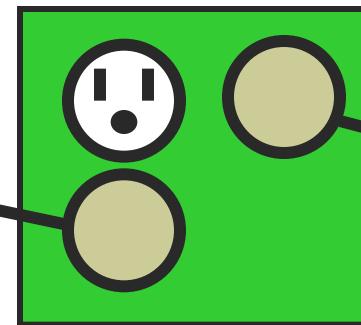
Server Can Be A Client! (And Vice-Versa)

Web browser



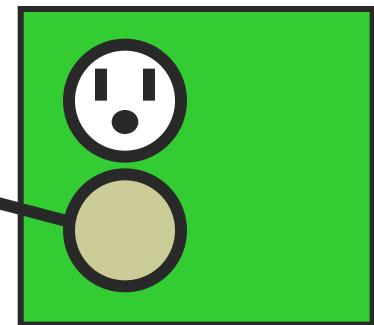
client

Web proxy server



client AND server

Web server



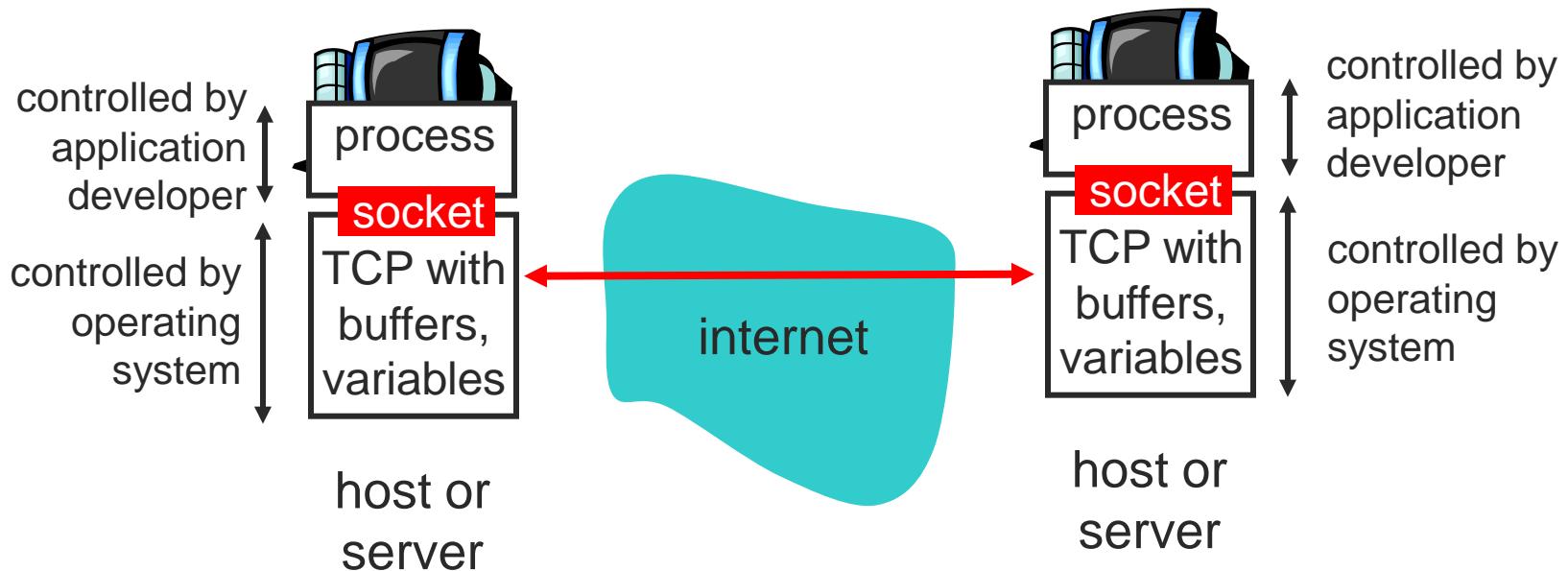
server

- Server: Process that listens and accepts requests
- Client: Process that connects to a listening process



[TCP Connections]

- Transmission Control Protocol (TCP) Service
 - OSI Transport Layer



[TCP Connections]

- Transmission Control Protocol (TCP) Service
 - OSI Transport Layer
 - Service Model
 - Byte stream (interpreted by application)
 - 16-bit port space allows multiple connections on a single host
 - Connection-oriented
 - Set up connection before communicating
 - Tear down connection when done



[TCP Service]

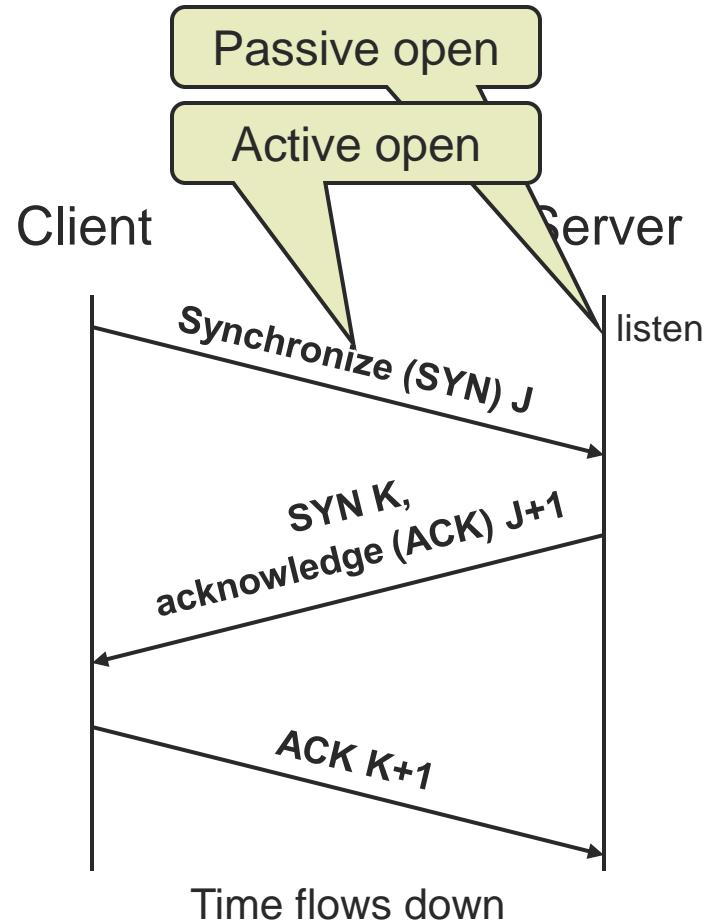
- Reliable Data Transfer
 - Guaranteed delivery
 - Exactly once if no catastrophic failures
- Sequenced Data Transfer
 - In-order delivery
- Regulated Data Flow
 - Monitors network and adjusts transmission appropriately
- Data Transmission
 - Full-Duplex byte stream

- Telephone Call
 - Guaranteed delivery
 - In-order delivery
 - Connection-oriented
 - Setup connection followed by conversation



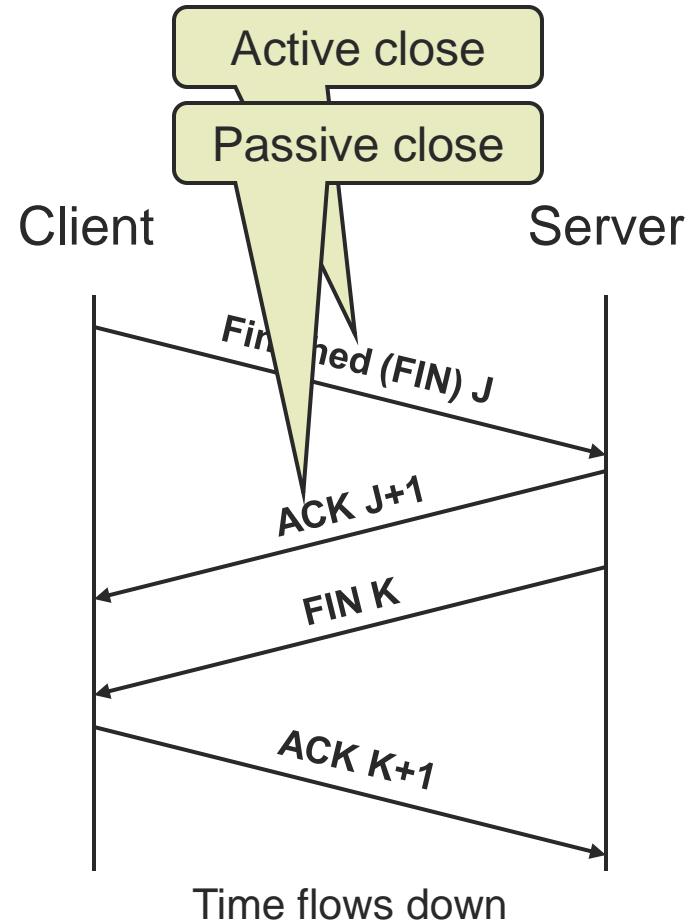
TCP Connection Establishment

- 3-Way Handshake
 - Sequence Numbers
 - J,K
 - Message Types
 - Synchronize (SYN)
 - Acknowledge (ACK)
 - Passive Open
 - Server listens for connection from client
 - Active Open
 - Client initiates connection to server



[TCP Connection Termination]

- Either client or server can initiate connection teardown
- Message Types
 - Finished (FIN)
 - Acknowledge (ACK)
- Active Close
 - Sends no more data
- Passive close
 - Accepts no more data



[UDP Services]

- User Datagram Protocol Service
 - OSI Transport Layer
 - Provides a thin layer over IP
 - 16-bit port space (distinct from TCP ports) allows multiple recipients on a single host



UDP Services

- Unit of Transfer
 - Datagram (variable length packet)
 - Unreliable
 - No guaranteed delivery
 - Drops packets silently
 - Unordered
 - No guarantee of maintained order of delivery
 - Unlimited Transmission
 - No flow control
- Postal Mail
 - Single mailbox to receive all letters
 - Unreliable
 - Not necessarily in-order
 - Letters sent independently
 - Must address each reply



Addresses and Data

- Internet domain names
 - Human readable
 - Variable length
 - Ex: **sal.cs.uiuc.edu**
- IP addresses
 - Each attachment point on Internet is given unique address
 - Easily handled by routers/computers
 - Fixed length
 - Somewhat geographical
 - Ex: **128.174.252.217**



[Byte Ordering]

- Big Endian vs. Little Endian
 - Little Endian (Intel, DEC):
 - Least significant byte of word is stored in the lowest memory address
 - Big Endian (Sun, SGI, HP):
 - Most significant byte of word is stored in the lowest memory address
 - Example: **128 . 2 . 194 . 95**

Big Endian	128	2	194	95
Little Endian	95	194	2	128



[Byte Ordering]

- Big Endian vs. Little Endian
 - Little Endian (Intel, DEC):
 - Least significant byte of word is stored in the lowest memory address
 - Big Endian (Sun, SGI, HP):
 - Most significant byte of word is stored in the lowest memory address
 - Network Byte Order = Big Endian
 - Allows both sides to communicate
 - Must be used for some data (i.e. IP Addresses)
 - Good form for all binary data



Byte Ordering Functions

- 16- and 32-bit conversion functions (for platform independence)
- Examples:

```
int m, n;  
short int s,t;  
  
m = ntohs (n)      // net-to-host long (32-bit) translation  
s = htons (t)      // net-to-host short (16-bit) translation  
n = htonl (m)      // host-to-net long (32-bit) translation  
t = ntohs (s)      // host-to-net short (16-bit) translation
```



Socket Address Structure

- IP address:

```
struct in_addr {  
    in_addr_t s_addr;           /* 32-bit IP address */  
};
```

- TCP or UDP address:

```
struct sockaddr_in {  
    short sin_family;          /* e.g., AF_INET */  
    ushort sin_port;           /* TCP/UDP port */  
    struct in_addr;             /* IP address */  
};
```

- all but **sin_family** in network byte order



Structure: addrinfo

- The `addrinfo` data structure (from `/usr/include/netdb.h`)
 - Canonical domain name and aliases
 - List of addresses associated with machine
 - Also address type and length information

<code>int ai_flags</code>	Input flags
<code>int ai_family</code>	Address family of socket
<code>int ai_socktype</code>	Socket type
<code>int ai_protocol</code>	Protocol of socket
<code>socklen_t ai_addrlen</code>	Length of socket address
<code>struct sockaddr *ai_addr</code>	Socket address of socket
<code>char *ai_canonname</code>	Canonical name of service location
<code>struct addrinfo *ai_next</code>	Pointer to next in list



Address Access/Conversion Functions

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict node,
                const char *restrict service,
                const struct addrinfo *restrict hints,
                struct addrinfo **restrict res);
```

■ Parameters

- **node**: host name or IP address to connect to
- **service**: a port number (“80”) or the name of a service (found /etc/services: “http”)
- **hints**: a filled out struct addrinfo



Example: Server

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;                                // point to the results

memset(&hints, 0, sizeof hints);                         // empty struct
hints.ai_family = AF_UNSPEC;                            // IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM;                        // TCP stream sockets
hints.ai_flags = AI_PASSIVE;                            // fill in my IP for me

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
    exit(1);
}
// servinfo now points to a linked list of 1 or more struct addrinfos
// ... do everything until you don't need servinfo anymore ....

freeaddrinfo(servinfo);                                 // free the linked-list
```



Example: Client

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;           // will point to the results

memset(&hints, 0, sizeof hints);    // make sure the struct is empty
hints.ai_family = AF_UNSPEC;        // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM;    // TCP stream sockets

// get ready to connect
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

// servinfo now points to a linked list of 1 or more struct addrinfos

// etc.
```



Structure: hostent (older)

- The **hostent** data structure (from `/usr/include/netdb.h`)
 - Canonical domain name and aliases
 - List of addresses associated with machine
 - Also address type and length information

```
struct hostent {  
    char* h_name;                      /* official name of host */  
    char** h_aliases;                  /* NULL-terminated alias list */  
    int h_addrtype;                   /* address type (AF_INET) */  
    int h_length;                     /* length of addresses (4B) */  
    char** h_addr_list;                /* NULL-terminated address list */  
#define h_addr h_addr_list[0];/* backward-compatibility */  
};
```



Address Access/Conversion Functions (older)

- All binary values are network byte ordered

```
struct hostent* gethostbyname (const char*  
    hostname);
```

- Translate English host name to IP address (uses DNS)

```
struct hostent* gethostbyaddr (const char* addr,  
    size_t len, int family);
```

- Translate IP address to English host name (not secure)

```
int gethostname (char* name, size_t namelen);
```

- Read host's name (use with `gethostbyname` to find local IP)



Address Access/Conversion Functions

```
char* inet_ntoa (struct in_addr inaddr);
```

- Translate IP address to ASCII dotted-decimal notation (e.g., “128.32.36.37”); not thread-safe

```
in_addr_t inet_addr (const char* strptr);
```

- Translate dotted-decimal notation to IP address; returns -1 on failure, thus cannot handle broadcast value “255.255.255.255”

```
int inet_aton (const char* strptr, struct in_addr  
inaddr);
```

- Translate dotted-decimal notation to IP address; returns 1 on success, 0 on failure



[Sockets API]

- Basic Unix Concepts
- Creation and Setup
- Establishing a Connection (TCP)
- Sending and Receiving Data
- Tearing Down a Connection (TCP)
- Advanced Sockets



[Basic Unix Concepts]

■ Input/Output – I/O

- Per-process table of I/O channels
- Table entries describe files, sockets, devices, pipes, etc.
- Unifies I/O interface
- Table entry/index into table called “file descriptor”

■ Error Model

- Return value
 - 0 on success
 - -1 on failure
 - NULL on failure for routines returning pointers
- **errno** variable



[Socket Creation and Setup]

- Include file `<sys/socket.h>`
- Create a socket
 - `int socket (int family, int type, int protocol);`
 - Returns file descriptor or -1.
- Bind a socket to a local IP address and port number
 - `int bind (int sockfd, struct sockaddr* myaddr, int addrlen);`
- Put socket into passive state (wait for connections rather than initiate a connection).
 - `int listen (int sockfd, int backlog);`



Functions: socket

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

- Create a socket.
 - Returns file descriptor or -1. Also sets **errno** on failure.
 - **family**: address family (namespace)
 - **AF_INET** for IPv4
 - other possibilities: **AF_INET6** (IPv6), **AF_UNIX** or **AF_LOCAL** (Unix socket), **AF_ROUTE** (routing)
 - **type**: style of communication
 - **SOCK_STREAM** for TCP (with **AF_INET**)
 - **SOCK_DGRAM** for UDP (with **AF_INET**)
 - **protocol**: protocol within family
 - typically 0



[Example: socket]

```
int sockfd, new_fd; /* listen on sock_fd, new
                      connection on new_fd */
struct sockaddr_in my_addr;           /* my address      */
struct sockaddr_in their_addr; /* connector addr */
int sin_size;

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
}
```



Function: bind

```
int bind (int sockfd, struct sockaddr*  
          myaddr, int addrlen);
```

- Bind a socket to a local IP address and port number
 - Returns 0 on success, -1 and sets **errno** on failure
 - **sockfd**: socket file descriptor (returned from **socket**)
 - **myaddr**: includes IP address and port number
 - IP address: set by kernel if value passed is **INADDR_ANY**, else set by caller
 - port number: set by kernel if value passed is 0, else set by caller
 - **addrlen**: length of address structure
 - = **sizeof (struct sockaddr_in)**



[Example: bind]

```
my_addr.sin_family = AF_INET;      // host byte order
my_addr.sin_port = htons(MYPORT); // short, network
                                  // byte order
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);

// automatically fill with my IP
bzero(&(my_addr.sin_zero), 8); // zero struct

if (bind(sockfd, (struct sockaddr *)&my_addr,
          sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
}
```



[TCP and UDP Ports]

- Allocated and assigned by the Internet Assigned Numbers Authority
 - see **RFC 1700** (for historical purposes only)

1-512	<ul style="list-style-type: none">■ standard services (see /etc/services)■ super-user only
513-1023	<ul style="list-style-type: none">■ registered and controlled, also used for identity verification■ super-user only
1024-49151	<ul style="list-style-type: none">■ registered services/ephemeral ports
49152-65535	<ul style="list-style-type: none">■ private/ephemeral ports



Reserved Ports

Keyword	Decimal	Description	Keyword	Decimal	Description
-----	-----	-----	-----	-----	-----
	0/tcp	Reserved	time	37/tcp	Time
	0/udp	Reserved	time	37/udp	Time
tcpmux	1/tcp	TCP Port Service	name	42/tcp	Host Name Server
tcpmux	1/udp	TCP Port Service	name	42/udp	Host Name Server
echo	7/tcp	Echo	nameserver	42/tcp	Host Name Server
echo	7/udp	Echo	nameserver	42/udp	Host Name Server
systat	11/tcp	Active Users	nicname	43/tcp	Who Is
systat	11/udp	Active Users	nicname	43/udp	Who Is
daytime	13/tcp	Daytime (RFC 867)	domain	53/tcp	Domain Name Server
daytime	13/udp	Daytime (RFC 867)	domain	53/udp	Domain Name Server
qotd	17/tcp	Quote of the Day	whois++	63/tcp	whois++
qotd	17/udp	Quote of the Day	whois++	63/udp	whois++
chargen	19/tcp	Character Generator	gopher	70/tcp	Gopher
chargen	19/udp	Character Generator	gopher	70/udp	Gopher
ftp-data	20/tcp	File Transfer Data	finger	79/tcp	Finger
ftp-data	20/udp	File Transfer Data	finger	79/udp	Finger
ftp	21/tcp	File Transfer Ctl	http	80/tcp	World Wide Web HTTP
ftp	21/udp	File Transfer Ctl	http	80/udp	World Wide Web HTTP
ssh	22/tcp	SSH Remote Login	www	80/tcp	World Wide Web HTTP
ssh	22/udp	SSH Remote Login	www	80/udp	World Wide Web HTTP
telnet	23/tcp	Telnet	www-http	80/tcp	World Wide Web HTTP
telnet	23/udp	Telnet	www-http	80/udp	World Wide Web HTTP
smtp	25/tcp	Simple Mail Transfer	kerberos	88/tcp	Kerberos
smtp	25/udp	Simple Mail Transfer	kerberos	88/udp	Kerberos



Functions: listen

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

- Put socket into passive state (wait for connections rather than initiate a connection)
 - Returns 0 on success, -1 and sets **errno** on failure
 - **sockfd**: socket file descriptor (returned from **socket**)
 - **backlog**: bound on length of unaccepted connection queue (connection backlog); kernel will cap, thus better to set high
 - Example:

```
if (listen(sockfd, BACKLOG) == -1) {  
    perror("listen");  
    exit(1);  
}
```



[Establishing a Connection]

- Include file `<sys/socket.h>`

```
int connect (int sockfd, struct  
sockaddr* servaddr, int addrlen);
```

- Connect to another socket.

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

- Accept a new connection. Returns file descriptor or -1.



Functions: connect

```
int connect (int sockfd, struct  
            sockaddr* servaddr, int addrlen);
```

- Connect to another socket.
 - Returns 0 on success, -1 and sets **errno** on failure
 - **sockfd**: socket file descriptor (returned from **socket**)
 - **servaddr**: IP address and port number of server
 - **addrlen**: length of address structure
 - = **sizeof (struct sockaddr_in)**
- Can use with UDP to restrict incoming datagrams and to obtain asynchronous errors



[Example: connect]

```
their_addr.sin_family = AF_INET; /* interp'd by host */
their_addr.sin_port = htons (PORT);
their_addr.sin_addr = *((struct in_addr*)he->h_addr);

bzero (&(their_addr.sin_zero), 8);
/* zero rest of struct */

if (connect (sockfd, (struct sockaddr*)&their_addr,
             sizeof (struct sockaddr)) == -1) {
    perror ("connect");
    exit (1);
}
```



Functions: accept

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

- Block waiting for a new connection
 - Returns file descriptor or -1 and sets **errno** on failure
 - **sockfd**: socket file descriptor (returned from **socket**)
 - **cliaddr**: IP address and port number of client (returned from call)
 - **addrlen**: length of address structure = pointer to **int** set to **sizeof (struct sockaddr_in)**
- **addrlen** is a **value-result** argument
 - the caller passes the size of the address structure, the kernel returns the size of the client's address (the number of bytes written)



[Example: accept]

```
sin_size = sizeof(struct sockaddr_in);
if ((new_fd = accept(sockfd, (struct sockaddr*)
                      &their_addr, &sin_size)) == -1) {
    perror("accept");
    continue;
}
```

- How does the server know which client it is?
 - `their_addr.sin_addr` contains the client's IP address
 - `their_addr.port` contains the client's port number

```
printf("server: got connection from %s\n",
       inet_ntoa(their_addr.sin_addr));
```



[Functions: accept]

■ Notes

- After `accept()` returns a new socket descriptor, I/O can be done using `read()` and `write()`
- Why does `accept()` need to return a new descriptor?



Sending and Receiving Data

```
int write (int sockfd, char* buf, size_t  
nbytes);
```

- Write data to a stream (TCP) or “connected” datagram (UDP) socket.
 - Returns number of bytes written or -1.

```
int read (int sockfd, char* buf, size_t  
nbytes);
```

- Read data from a stream (TCP) or “connected” datagram (UDP) socket.
 - Returns number of bytes read or -1.



Functions: write

```
int write (int sockfd, char* buf, size_t nbytes);
```

- Write data to a stream (TCP) or “connected” datagram (UDP) socket
 - Returns number of bytes written or -1 and sets **errno** on failure
 - **sockfd**: socket file descriptor (returned from **socket**)
 - **buf**: data buffer
 - **nbytes**: number of bytes to try to write
 - Example:

```
if((w = write(fd, buf, sizeof(buf))) < 0) {  
    perror("write");  
    exit(1);  
}
```



Functions: write

```
int write (int sockfd, char* buf, size_t nbytes);
```

- Notes

- **write** blocks waiting for data from the client
- **write** may not write all bytes asked for
 - Does not guarantee that **sizeof(buf)** is written
 - This is not an error
 - Simply continue writing to the device
- Some reasons for failure or partial writes
 - Process received interrupt or signal
 - Kernel resources unavailable (e.g., buffers)



Example: writen

```
/* Write "n" bytes to a descriptor */
ssize_t writen(int fd, const void *ptr, size_t n) {
    size_t nleft;
    ssize_t nwritten;
    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1); /* error, return -1 */
            else
                break; /* error, return amount written so far */
        }
        else
            if (nwritten == 0)
                break;
        nleft -= nwritten;
        ptr += nwritten;
    }
    return(n - nleft); /* return >= 0 */
}
```

`writen` returned
a potential error

0 bytes were
written

Update number
of bytes left to
write and
pointer into
buffer



Functions: read

```
int read (int sockfd, char* buf, size_t nbytes);
```

- Read data from a stream (TCP) or “connected” datagram (UDP) socket
 - Returns number of bytes read or -1, sets **errno** on failure
 - Returns 0 if socket closed
 - **sockfd**: socket file descriptor (returned from **socket**)
 - **buf**: data buffer
 - **nbytes**: number of bytes to try to read
 - Example

```
if((r = read(newfd, buf, sizeof(buf))) < 0) {  
    perror("read"); exit(1);  
}
```



Functions: read

```
int read (int sockfd, char* buf, size_t nbytes);
```

■ Notes

- **read** blocks waiting for data from the client
- **read** may return less than asked for
 - Does not guarantee that **sizeof(buf)** is read
 - This is not an error
 - Simply continue reading from the device



Example: readn

```
/* Read "n" bytes from a descriptor */
ssize_t readn(int fd, void *ptr, size_t n) {
    size_t nleft;
    ssize_t nread;
    nleft = n;
    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1); /* error, return -1 */
            else
                break; /* error, return amt read */
        }
        else
            if (nread == 0)
                break; /* EOF */
            nleft -= nread;
            ptr += nread;
    }
    return(n - nleft); /* return >= 0 */
}
```

read returned
a potential error

0 bytes were
read

Update number
of bytes left to
read and
pointer into
buffer



Sending and Receiving Data

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

- Write data to a stream (TCP) or “connected” datagram (UDP) socket.
 - Returns number of bytes written or -1.

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

- Read data from a stream (TCP) or “connected” datagram (UDP) socket.
 - Returns number of bytes read or -1.



Functions: send

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

- Send data un a stream (TCP) or “connected” datagram (UDP) socket
 - Returns number of bytes written or -1 and sets **errno** on failure
 - **sockfd**: socket file descriptor (returned from **socket**)
 - **buf**: data buffer
 - **nbytes**: number of bytes to try to write
 - **flags**: control flags
 - MSG_PEEK: get data from the beginning of the receive queue without removing that data from the queue



Functions: send

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

- Example

```
len = strlen(msg);  
bytes_sent = send(sockfd, msg, len, 0);
```



Functions: recv

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

- Read data from a stream (TCP) or “connected” datagram (UDP) socket
 - Returns number of bytes read or -1, sets **errno** on failure
 - Returns 0 if socket closed
 - **sockfd**: socket file descriptor (returned from **socket**)
 - **buf**: data buffer
 - **nbytes**: number of bytes to try to read
 - **flags**: see man page for details; typically use 0



Functions: recv

```
int read (int sockfd, char* buf, size_t nbytes);
```

■ Notes

- **read** blocks waiting for data from the client but does not guarantee that **sizeof(buf)** is read
- Example

```
if((r = read(newfd, buf, sizeof(buf))) < 0) {  
    perror("read"); exit(1);  
}
```



[Sending and Receiving Data]

- Datagram sockets aren't connected to a remote host
 - What piece of information do we need to give before we send a packet?
 - The destination/source address!



Sending and Receiving Data

```
int sendto (int sockfd, char* buf,  
           size_t nbytes, int flags, struct  
           sockaddr* destaddr, int addrlen);
```

- Send a datagram to another UDP socket.
 - Returns number of bytes written or -1.

```
int recvfrom (int sockfd, char* buf,  
             size_t nbytes, int flags, struct  
             sockaddr* srcaddr, int* addrlen);
```

- Read a datagram from a UDP socket.
 - Returns number of bytes read or -1.



Functions: sendto

```
int sendto (int sockfd, char* buf, size_t nbytes,  
           int flags, struct sockaddr* destaddr, int  
           addrlen);
```

- Send a datagram to another UDP socket
 - Returns number of bytes written or -1 and sets **errno** on failure
 - **sockfd**: socket file descriptor (returned from **socket**)
 - **buf**: data buffer
 - **nbytes**: number of bytes to try to read
 - **flags**: see man page for details; typically use 0
 - **destaddr**: IP address and port number of destination socket
 - **addrlen**: length of address structure
 - = **sizeof (struct sockaddr_in)**



Functions: sendto

```
int sendto (int sockfd, char* buf, size_t nbytes,  
           int flags, struct sockaddr* destaddr, int  
           addrlen);
```

- Example

```
n = sendto(sock, buf, sizeof(buf), 0, (struct  
    sockaddr *) &from, fromlen);  
  
if (n < 0)  
    perror("sendto");  
    exit(1);  
  
}
```



Functions: recvfrom

```
int recvfrom (int sockfd, char* buf, size_t  
nbytes, int flags, struct sockaddr* srcaddr,  
int* addrlen);
```

- Read a datagram from a UDP socket.
 - Returns number of bytes read (0 is valid) or -1 and sets **errno** on failure
 - **sockfd**: socket file descriptor (returned from **socket**)
 - **buf**: data buffer
 - **nbytes**: number of bytes to try to read
 - **flags**: see man page for details; typically use 0
 - **srcaddr**: IP address and port number of sending socket (returned from call)
 - **addrlen**: length of address structure = pointer to **int** set to **sizeof (struct sockaddr_in)**



Functions: recvfrom

```
int recvfrom (int sockfd, char* buf, size_t  
nbytes, int flags, struct sockaddr* srcaddr,  
int* addrlen);
```

- Example

```
n = recvfrom(sock, buf, 1024, 0, (struct sockaddr  
*)&from,&fromlen);  
if (n < 0) {  
    perror("recvfrom");  
    exit(1);  
}
```



[Tearing Down a Connection]

`int close (int sockfd);`

- Close a socket.
 - Returns 0 on success, -1 and sets `errno` on failure.

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

- Force termination of communication across a socket in one or both directions.
 - Returns 0 on success, -1 and sets `errno` on failure.



Functions: close

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

- Close a socket
 - Returns 0 on success, -1 and sets **errno** on failure
 - **sockfd**: socket file descriptor (returned from **socket**)
- Closes communication on socket in both directions
 - All data sent before **close** are delivered to other side (although this aspect can be overridden)
- After **close**, **sockfd** is not valid for reading or writing



Functions: shutdown

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

- Force termination of communication across a socket in one or both directions
 - Returns 0 on success, -1 and sets `errno` on failure
 - `sockfd`: socket file descriptor (returned from `socket`)
 - `howto`:
 - `SHUT_RD` to stop reading
 - `SHUT_WR` to stop writing
 - `SHUT_RDWR` to stop both
- `shutdown` overrides the usual rules regarding duplicated sockets, in which TCP teardown does not occur until all copies have closed the socket

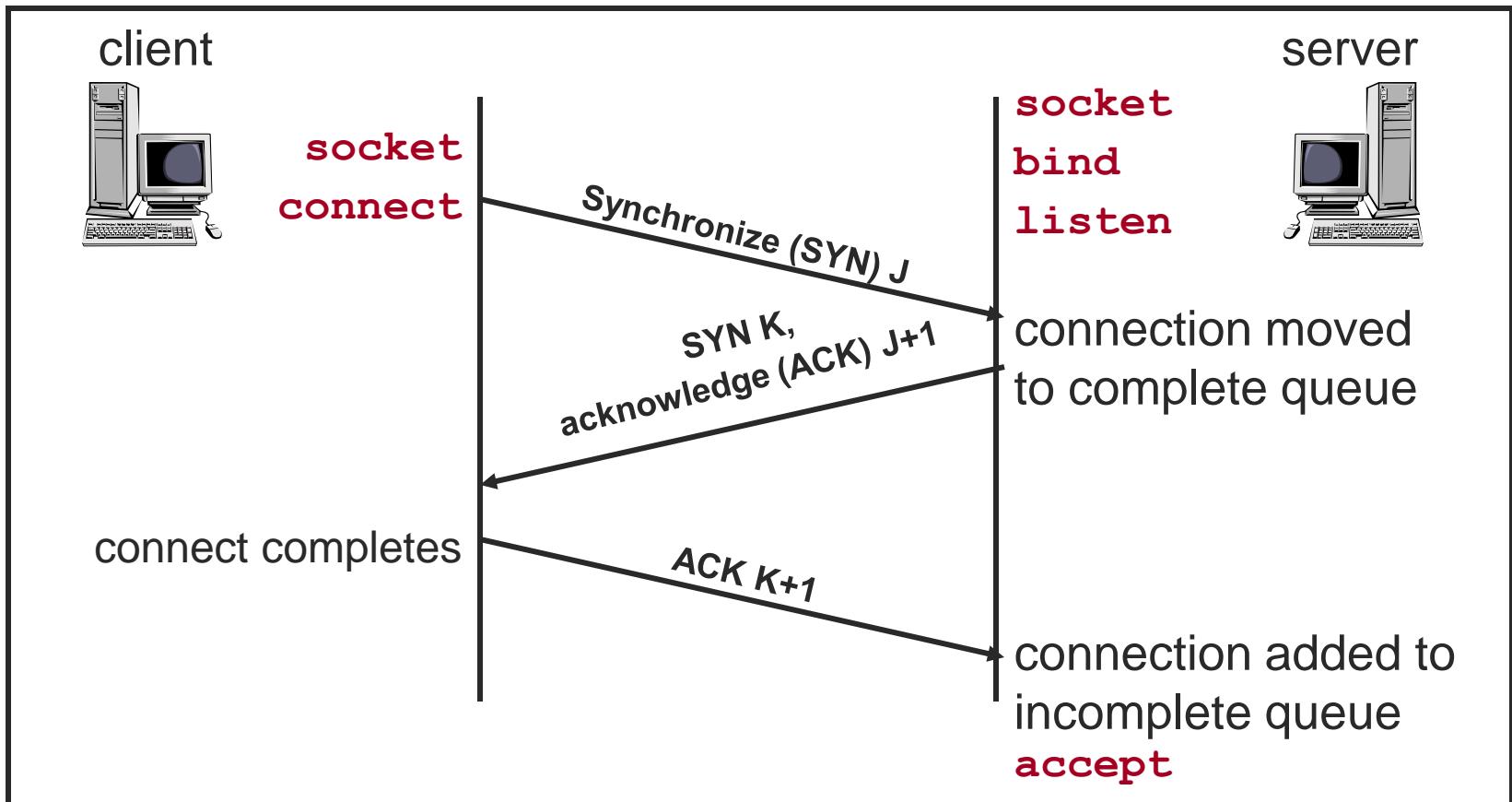


[Note on **close** vs. **shutdown**]

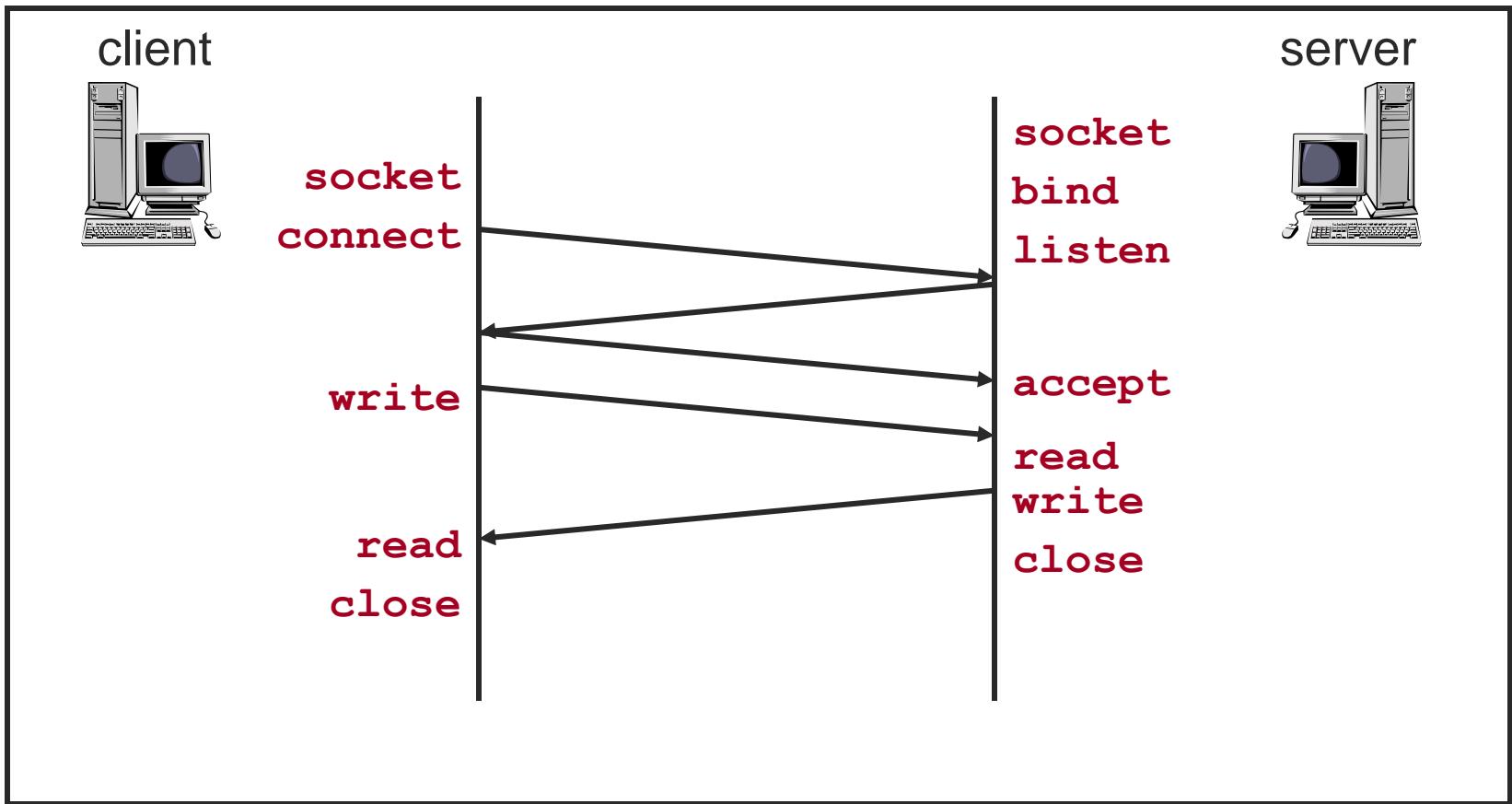
- **close()** : closes the socket but the connection is still open for processes that shares this socket
 - The connection stays opened both for read and write
- **shutdown()** : breaks the connection for all processes sharing the socket
 - A read will detect **EOF**, and a write will receive **SIGPIPE**
 - **shutdown()** has a second argument how to close the connection:
 - 0 means to disable further reading
 - 1 to disable writing
 - 2 disables both



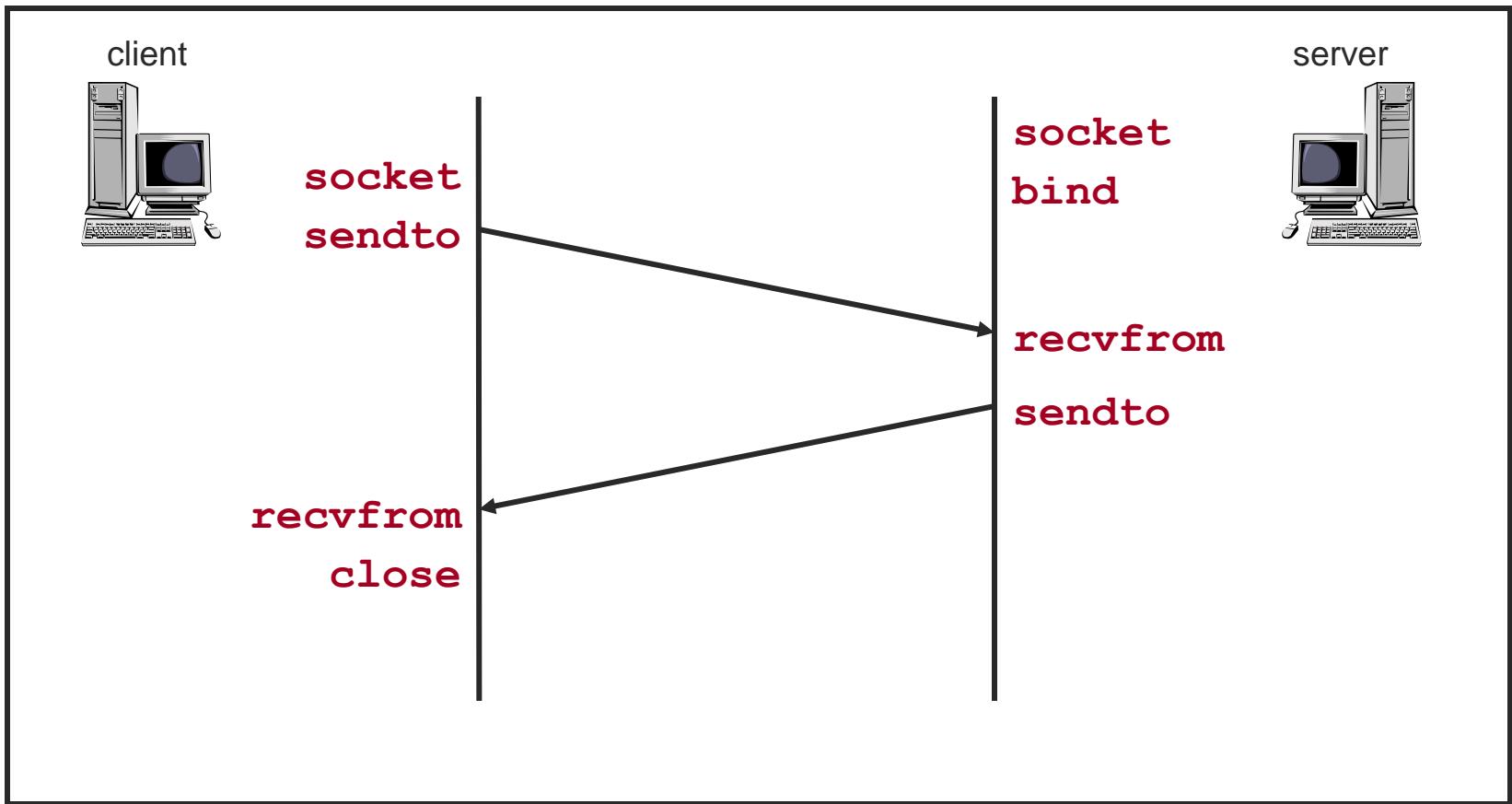
TCP Connection Setup



[TCP Connection Example]



[UDP Connection Example]



Examples

- Taken from Beej's Guide to Network Programming:
<http://beej.us/guide/bgnet/>
- Structure
 - One server on a machine
 - Server handles multiple clients using `fork()`
- Basic routine
 - Server waits for a connection
 - `accept()`s the connection
 - `fork()`s a child process to handle the client



[Example: TCP]

- Client-Server example using TCP
 - For each client
 - server forks new process to handle connection
 - sends “**Hello, world**”



[server]

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#define PORT 3490 /* well-known port */
#define BACKLOG 10 /* how many pending
                           connections queue
                           will hold */
```



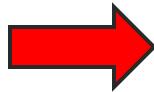
[server]

```
main()
{
    int sockfd, new_fd;      /* listen on sock_fd, new
                               connection on new_fd */
    struct sockaddr_in my_addr;  /* my address         */
    struct sockaddr_in their_addr; /* connector addr   */
    int sin_size;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) ===
1) {
        perror("socket");
        exit(1);
}
```



[server]



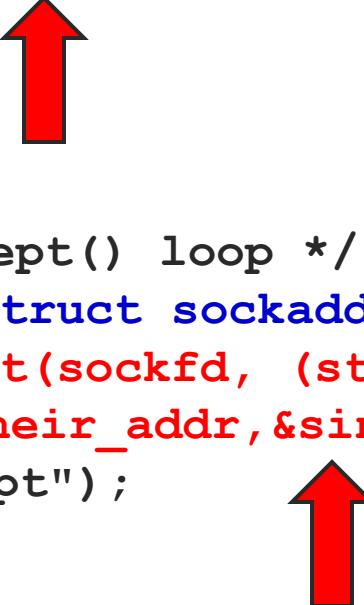
```
my_addr.sin_family = AF_INET; /* host byte order */
my_addr.sin_port = htons(MYPORT); /* short, network
                                byte order */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
/* automatically fill with my IP */
bzero(&(my_addr.sin_zero), 8); /* zero struct */

if (bind(sockfd, (struct sockaddr *)&my_addr,
         sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
}
```



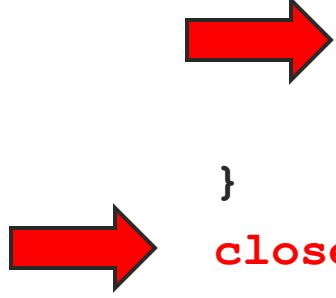
[server]

```
if (listen(sockfd, BACKLOG) == -1) {  
    perror("listen");  
    exit(1);  
}  
  
while(1) /* main accept() loop */  
{  
    sin_size = sizeof(struct sockaddr_in);  
    if ((new_fd = accept(sockfd, (struct sockaddr*)&their_addr,  
                         &sin_size)) == -1) {  
        perror("accept");  
        continue;  
    }  
    printf("server: got connection from %s\n",  
          inet_ntoa(their_addr.sin_addr));  
}
```



[server]

```
if (!fork()) { /* this is the child process */
    if (send(new_fd,"Hello, world!\n", 14, 0)
        == -1)
        perror("send");
    close(new_fd);
    exit(0);
}
close(new_fd); /* parent doesn't need this */
/* clean up all child processes */
while(waitpid(-1,NULL,WNOHANG) > 0);
}
```



client

```
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define PORT 3490          /* well-known port */
#define MAXDATASIZE 100    /* max number of bytes we
                           can get at once */
```



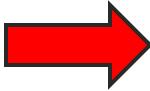
[client]

```
int main (int argc, char* argv[]) {
    int sockfd, numbytes;
    char buf[MAXDATASIZE + 1];
    struct hostent* he;
    struct sockaddr_in their_addr;
    /* connector's address information */
    if (argc != 2) {
        fprintf (stderr, "usage: client hostname\n");
        exit (1);
    }
    if ((he = gethostbyname (argv[1])) == NULL) {
        /* get the host info */
        perror ("gethostbyname");
        exit (1);
    }
```

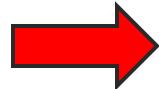


[client]

```
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) == -1) {  
    perror ("socket");  
    exit (1);  
}  
  
their_addr.sin_family = AF_INET; /* interp'd by host */  
their_addr.sin_port = htons (PORT);  
their_addr.sin_addr = *((struct in_addr*)he->h_addr);  
bzero (&(their_addr.sin_zero), 8);  
/* zero rest of struct */  
if (connect (sockfd, (struct sockaddr*)&their_addr,  
            sizeof (struct sockaddr)) == -1) {  
    perror ("connect");  
    exit (1);  
}
```



client



```
if ((numbytes = recv (sockfd, buf, MAXDATASIZE, 0))  
    == -1) {  
    perror ("recv");  
    exit (1);  
}  
  
buf[numbytes] = '\0';  
printf ("Received: %s", buf);  
close (sockfd);  
return 0;  
}
```



[Example: UDP]

- Client-Server example using UDP
 - For each client
 - **listener** sits on a machine waiting for an incoming packet on port 4950
 - **talker** sends a packet to that port, on the specified machine, that contains whatever the user enters on the command line



[listener]

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define MYPORT "4950" // the port users will be connecting to
#define MAXBUFSIZE 100
```



[listener]

```
// get sockaddr, IPv4 or IPv6:  
void *get_in_addr(struct sockaddr *sa) {  
    if (sa->sa_family == AF_INET) {  
        return &(((struct sockaddr_in*)sa)->sin_addr);  
    }  
    return &(((struct sockaddr_in6*)sa)->sin6_addr);  
}
```



[listener]

```
int main(void) {  
    int sockfd, rv, numbytes;  
    struct addrinfo hints, *servinfo, *p;  
    struct sockaddr_storage their_addr;  
    char buf[MAXBUFSIZE], s[INET6_ADDRSTRLEN];  
    size_t addr_len;  
    memset(&hints, 0, sizeof hints);  
    hints.ai_family = AF_UNSPEC;  
 hints.ai_socktype = SOCK_DGRAM;  
    hints.ai_flags = AI_PASSIVE; // use my IP  
  
    if ((rv = getaddrinfo(NULL, MYPORT, &hints, &servinfo)) == -1) {  
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));  
        return 1;  
    }
```



[listener]

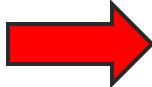
```
// loop through all results and bind to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
                          p->ai_protocol)) == -1) {
        perror("listener: socket");
        continue;
    }

    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("listener: bind");
        continue;
    }
    break;
}
```



[listener]

```
if (p == NULL) {  
    fprintf(stderr, "listener: failed to bind socket\n");  
    return 2;  
}  
freeaddrinfo(servinfo);  
printf("listener: waiting to recvfrom...\n");  
  
addr_len = sizeof their_addr;  
if ((numbytes = recvfrom(sockfd, buf, MAXBUFSIZE-1 , 0,  
    (struct sockaddr *)&their_addr, &addr_len)) == -1) {  
    perror("recvfrom");  
    exit(1);  
}
```



[listener]

```
printf("listener: got packet from %s\n",
inet_ntop(their_addr.ss_family,
          get_in_addr((struct sockaddr *)&their_addr),
          s, sizeof s));
printf("listener: packet is %d bytes long\n", numbytes);
buf[numbytes] = '\0';
printf("listener: packet contains \"%s\"\n", buf);

close(sockfd);

return 0;
}
```



talker

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT "4950"      // port users will connect to
```



[talker]

```
int main(int argc, char *argv[]) {
    int sockfd, rv, numbytes;
    struct addrinfo hints, *servinfo, *p;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;

    if ((rv = getaddrinfo(argv[1], SERVERPORT, &hints,
                          &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }
```



[talker]

```
// loop through all the results and make a socket
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
                          p->ai_protocol)) == -1) {
        perror("talker: socket");
        continue;
    }
    break;
}

if (p == NULL) {
    fprintf(stderr, "talker: failed to bind socket\n");
    return 2;
}
```



talker

```
if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
                      p->ai_addr, p->ai_addrlen)) == -1) {
    perror("talker: sendto");
    exit(1);
}

freeaddrinfo(servinfo);

printf("talker: sent %d bytes to %s\n", numbytes, argv[1]);
close(sockfd);

return 0;
}
```



Connected Datagram Sockets

- **talker** calls **connect()** and specifies **listener**'s address
 - **talker** may only send to and receive from the address specified by **connect()**
 - Don't have to use **sendto()** and **recvfrom()**
 - Simply use **send()** and **recv()**



[Framing]

- Goal
 - Framing messages on a byte stream
- Given a TCP message stream, how can we pass logical messages?
 - Note:
 - read may return partial or multiple messages
 - Questions:
 - How can we determine the end of a message?
- Hint
 - string storage in C and Pascal
 - format strings with printf



[Framing Problem]

■ Approach

- Think about the problem for a minute or two
- Introduce yourself to 2-3 people near you
(form groups of 3-4)
- Discuss the problem and agree on a solution

