

# **Computer Networks**

## **EE-5281**

By Ihsan Ul Haq  
Lec-5 & 6



# UDP Services

- User Datagram Protocol, at OSI transport layer
- Thin layer over IP
- Aspects of services
  - unit of transfer is a datagram (variable length packet)
  - **unreliable**, drops packets **silently**
  - no ordering guarantees
  - no flow control
  - 16-bit port space (**distinct** from TCP ports) allows multiple recipients on a single host

# Addresses and Data

- Internet domain names: human readable
  - mnemonic
  - variable length
    - e.g., www.case.edu.pk, www.carepvtltd.com (FQDN)
- IP addresses: easily handled by routers/computers
  - fixed length
  - tied (loosely) to geography
    - e.g., 131.126.143.82 or 212.0.0.1

# Endianness

- Machines on Internet have different endianness
  - little-endian (Intel, DEC): least significant byte of word stored in lowest memory address
  - big-endian (Sun, SGI, HP): most significant byte...
  - **network byte order** is big-endian
  - use of network byte order
    - imperative for some data (e.g., IP addresses)
    - good form for all binary data (e.g., application-specific)
    - ASCII/Unicode are acceptable alternatives

# Endianness

- 16- / 32-bit conversion (for platform independence)

```
int m, n; // int32
```

```
short int s, t; // int16
```

```
m = ntohl(n) // net-to-host long (32-bit) translation
```

```
s = ntohs(t) // net-to-host short (16-bit) translation
```

```
n = htonl(m) // host-to-net long (32-bit) translation
```

```
t = htons(s) // host-to-net short (16-bit) translation
```

# Socket Address Structures

- Socket address structures (all fields in network byte order except `sin_family`)

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 */  
};
```

# Address Conversion

- All binary values used and returned by these functions 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 Conversion

**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

**char\* inet\_ntoa (struct in\_addr inaddr);**

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

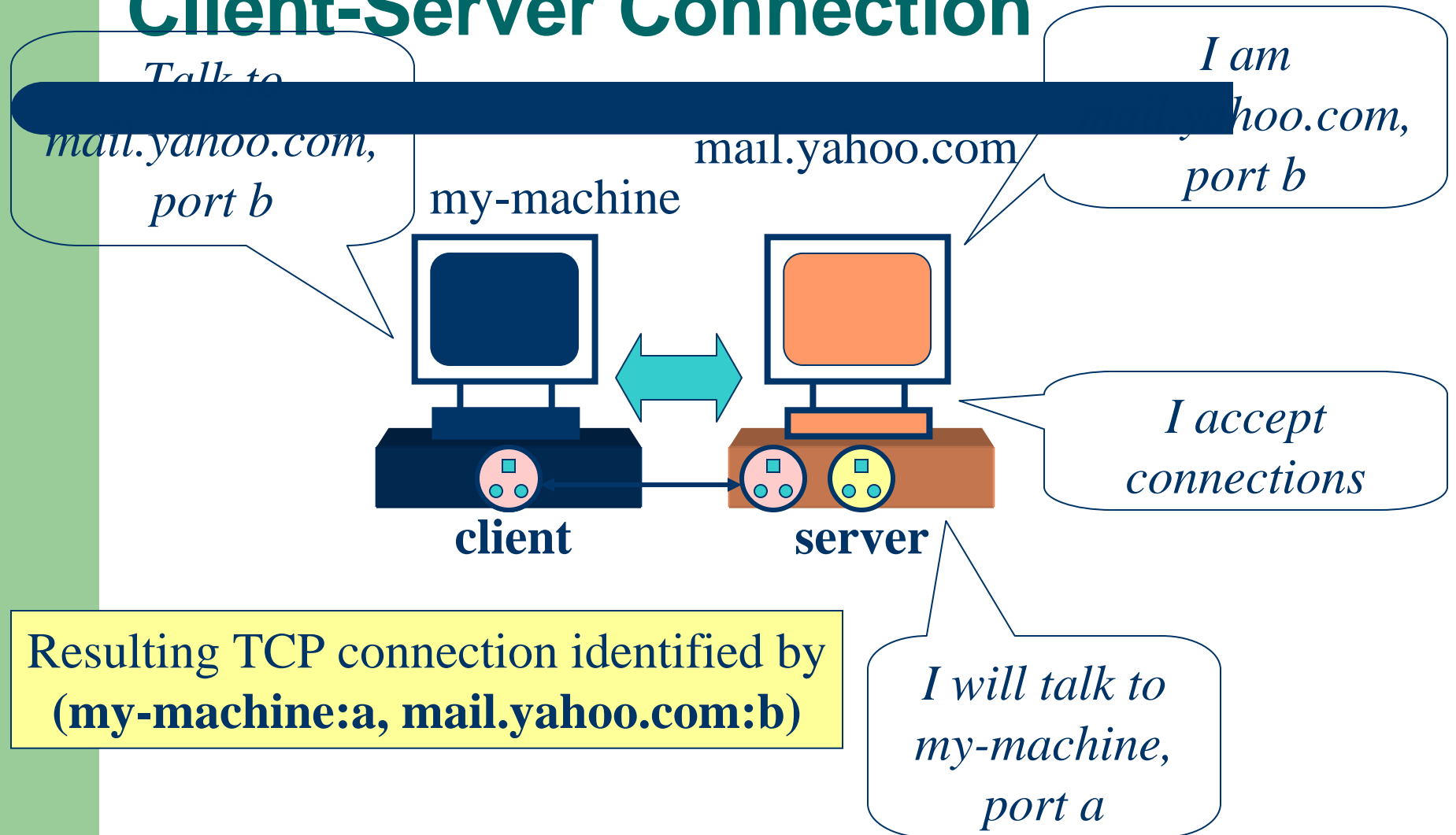
# Sockets API

- Basic Unix concepts
- Creation and setup
- Establishing a connection (TCP only)
- Sending and receiving data
- Tearing down a connection (TCP only)
- Advanced sockets

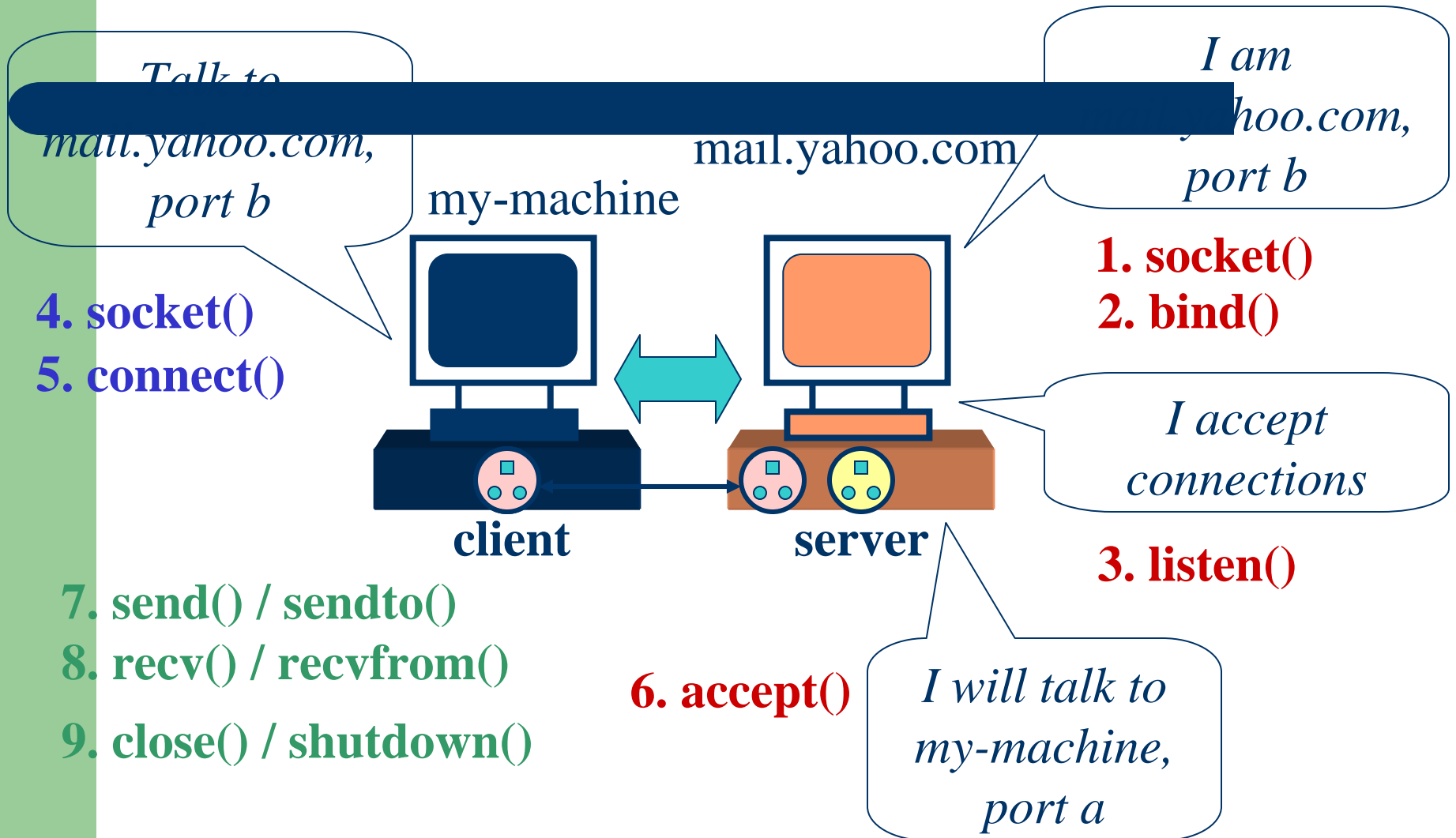
# Basic UNIX Concepts

- I/O
  - per-process table of I/O channels
  - table entries can describe files, sockets, devices, pipes, etc.
  - unifies I/O interface
  - table entry/index into table called “**file descriptor**”
- Error model
  - “standardization” of return value
    - 0 on success, -1 on failure
    - NULL on failure for routines returning pointers
  - **errno** variable

# Client-Server Connection



# Client-Server Connection



# Socket Creation and Setup

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

# Creating Sockets - `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) or protocol family

- `AF_INET` for IPv4
- other possibilities: `AF_INET6` (IPv6), `AF_UNIX`, `AF_OSI` 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

- Usually already defined by domain & type, typically 0 (default)

# Naming and Identifying Sockets - 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**)

# TCP and UDP Port Namespaces

- Allocated and assigned by the Internet Assigned Numbers Authority (IANA)
  - see RFC 1700
  - <ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers>
- **1-512** standard services (see `/etc/services`); super-user only
- **513-1023** registered and controlled, also used for identity verification; super-user only
- **1024-49151** registered services/ephemeral ports
- **49152-65535** private/ephemeral ports

# Waiting for Connections - 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 un-**accept()**ed connection queue (connection backlog); kernel will cap, thus better to set high

# Contact the Peer - 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

# Welcome a Connection - accept()

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

Accept a new connection (first one off queue of pending connections). Returns file descriptor or -1. Also sets **errno**.

**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)



# **Sending and Receiving data**

# Send the Data - 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. Also sets **errno** on failure.

**sockfd** : socket file descriptor (returned from **socket** )

**buf** : data buffer

**nbytes** : number of bytes to try to write

- some reasons for failure or partial writes:
  - process received interrupt or signal
  - kernel resources unavailable (e.g., buffers)

**int send (int sockfd, char\* buf, size\_t nbytes , int flags);**

# Receive the Data - 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. Also 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

**int recv (int sockfd, char\* buf, size\_t nbytes , int flags);**

# Send Data to Someone - 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. Also 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**)

# Receive Data from Someone - 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. Also 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**)



# Tearing Down a Connection

# Good Bye - close()

**int close (int sockfd);**

Closes a socket and deletes descriptor from system tables. 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.

# Close in My Way - 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.

# Advanced Sockets

- Managing multiple connections
  - `fork()/exec()`: multiple server processes
  - `pthread_create()`: multi-threaded server process
  - (no calls): event-based server process
- Detecting data arrival
  - `select()` and `poll()` functions
- Synchronous vs. asynchronous connections
- Other socket options

# Example of Use

- Taken from Beej's Guide to Network Programming (see the course web page)
- Client-server example using TCP
- For each client
  - Server forks new process to handle connection
  - Sends "Hello, world"

## server.c (from Beej)

```
#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.c (from Beej) – socket()

```
main()
```

```
{
```

```
    int sockfd, new_fd; /* listen on sockfd, new connection on  
                        new_fd */
```

```
    struct sockaddr_in my_addr; /* my address */
```

```
    struct sockaddr_in their_addr; /* connector addr */
```

```
    int sin_size; /* address size passed to accept() */
```

```
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
```

```
    {
```

```
        perror("socket");
```

```
        exit(1);
```

```
    } /* if(socket) ends */
```

## server.c (from Beej) – 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 (w/o Beej's bug) */
bzero(&(my_addr.sin_zero), 8); /* zero the rest of the struct */
```

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

# server.c (from Beej) – listen()/accept()

```
if (listen(sockfd, BACKLOG) == -1) {  
    perror("listen");  
    exit(1);  
} /* if(listen) ends */  
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;  
    } /* if(accept) ends */  
    printf("server: got connection from %s\n",  
        inet_ntoa(their_addr.sin_addr));
```

## server.c (from Beej) – close()

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

## client.c (from Beej)

```
#include <stdio.h>
#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.c (from Beej)

```
int main (int argc, char* argv[])  
{
```

```
    int sockfd, numbytes;  
    char buf[MAXDATASIZE];  
    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);  
    }
```

# hostent Data Structure

- Canonical domain name and aliases
- 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 */  
};
```

# client.c (from Beej) – socket()/connect()

```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)  
{
```

```
    perror ("socket"); exit (1);
```

```
} /* if(socket) ends */
```

```
their_addr.sin_family = AF_INET; /* interpreted 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 the rest of struct */
```

```
if (connect (sockfd, (struct sockaddr*)&their_addr, sizeof  
    (struct sockaddr)) == -1) {
```

```
    perror ("connect"); exit (1);
```

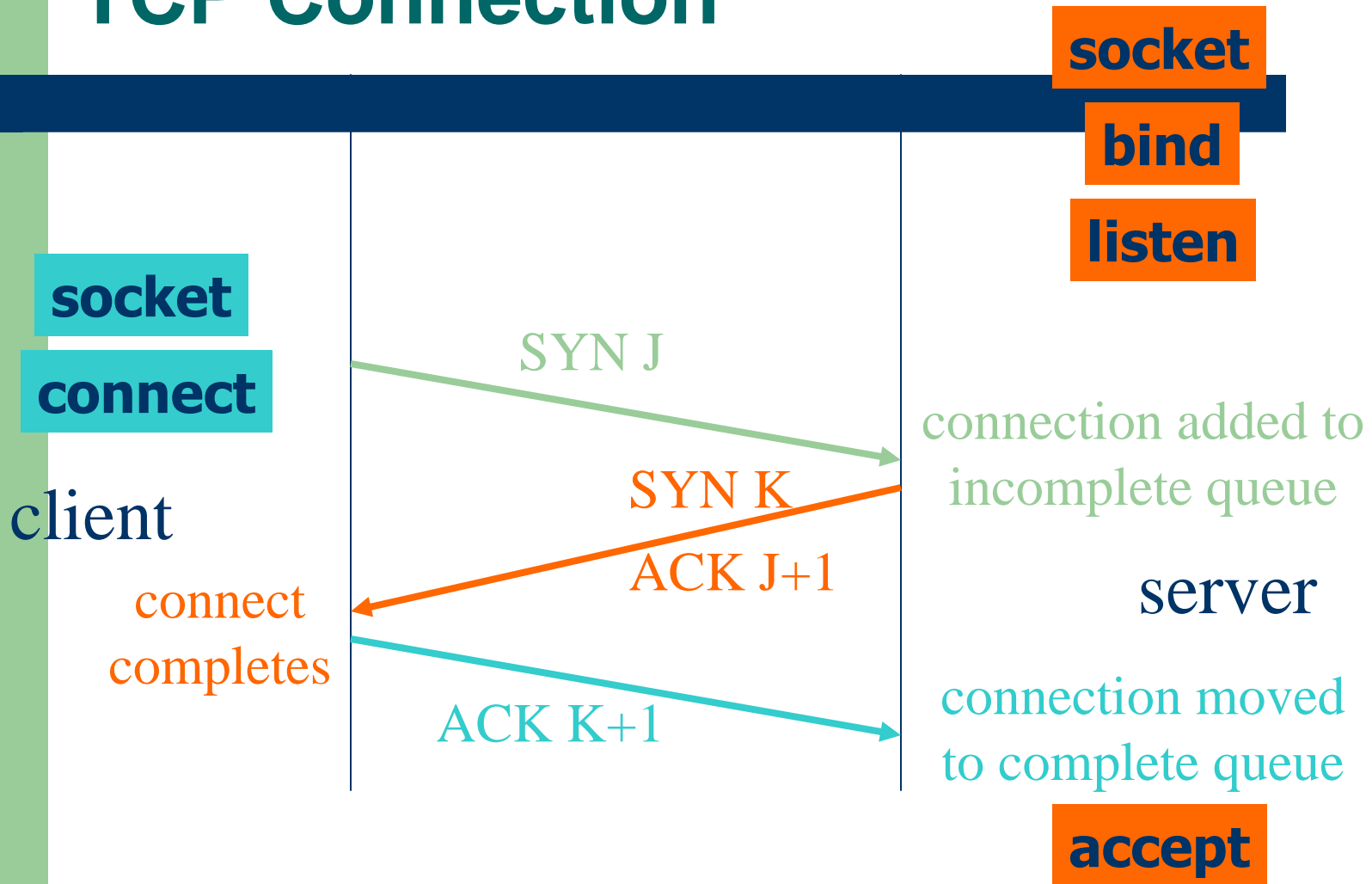
```
} /* if(connect) ends */
```

## client.c (from Beej) – recv()/close()

```
if ((numbytes = recv (sockfd, buf, MAXDATASIZE,  
0)) == -1) {  
    perror ("recv");  
    exit (1);  
} /* if(recv) ends */
```

```
buf[numbytes] = '\0';  
printf ("Received: %s", buf);  
close (sockfd);  
return 0;  
} /* main function ends */
```

# TCP Connection



# TCP Connection

socket

bind

listen

socket

connect

client

write

read

close

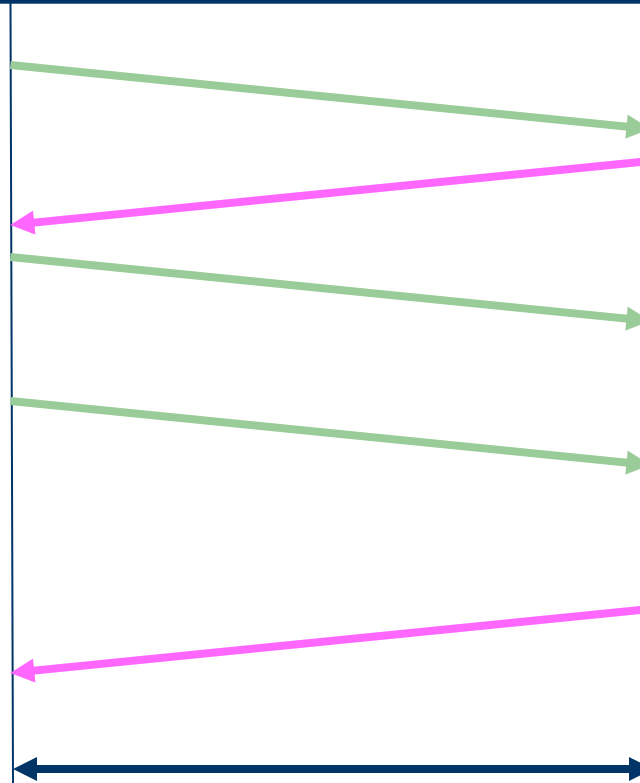
server

accept

read

write

close



# UDP Connection

socket

bind

server

recvfrom

sendto

socket

sendto

client

recvfrom

close



# Food for Thought

Framing messages on a byte stream ... ?

- Problem
  - pass logical messages using a TCP connection
  - **read()** may return partial or multiple messages
  - how can receiver identify the end of a message?
- Try to come up with two or three methods
- Hints
  - string storage in C and Pascal
  - format strings with **printf()**