CS 640: Computer Networking

Yu-Chi Lai

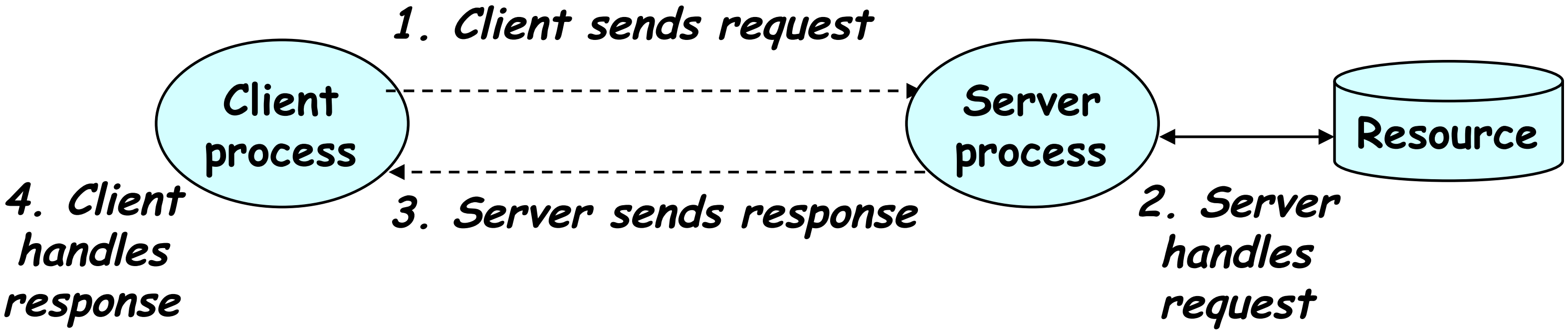
Lecture 3
Network Programming

Topics

- Client-server model
- Sockets interface
- Socket primitives
- Example code for echoclient and echoserver
- Debugging With GDB
- Programming Assignment 1 (MNS)

Client/server model

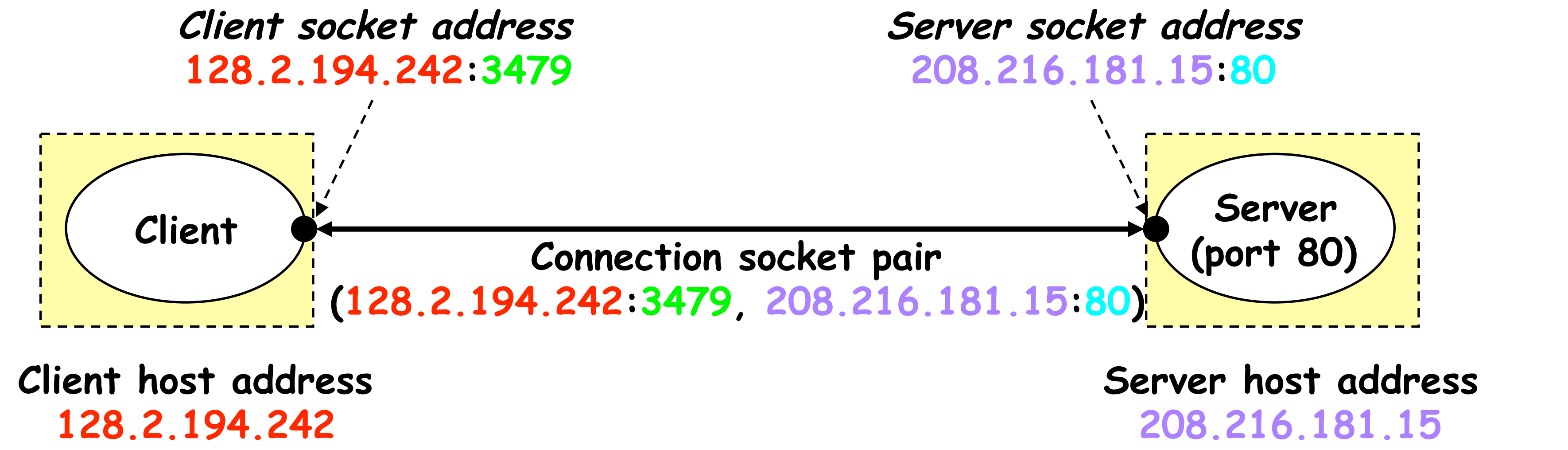
- Client asks (*request*) – server provides (*response*)
- Typically: single server - multiple clients
- The server does not need to know *anything* about the client
  - even that it exists
- The client should always know *something* about the server
  - at least where it is located



Note: clients and servers are processes running on hosts (can be the same or different hosts).

Internet Connections (TCP/IP)

- Address the machine on the network
  - By IP address
- Address the process
  - By the "port"-number
- The pair of *IP-address + port* – makes up a "*socket-address*"



Client socket address
128.2.194.242:3479

Server socket address
208.216.181.15:80

Client

Server
(port 80)

Connection socket pair
(128.2.194.242:3479, 208.216.181.15:80)

Client host address
128.2.194.242

Server host address
208.216.181.15

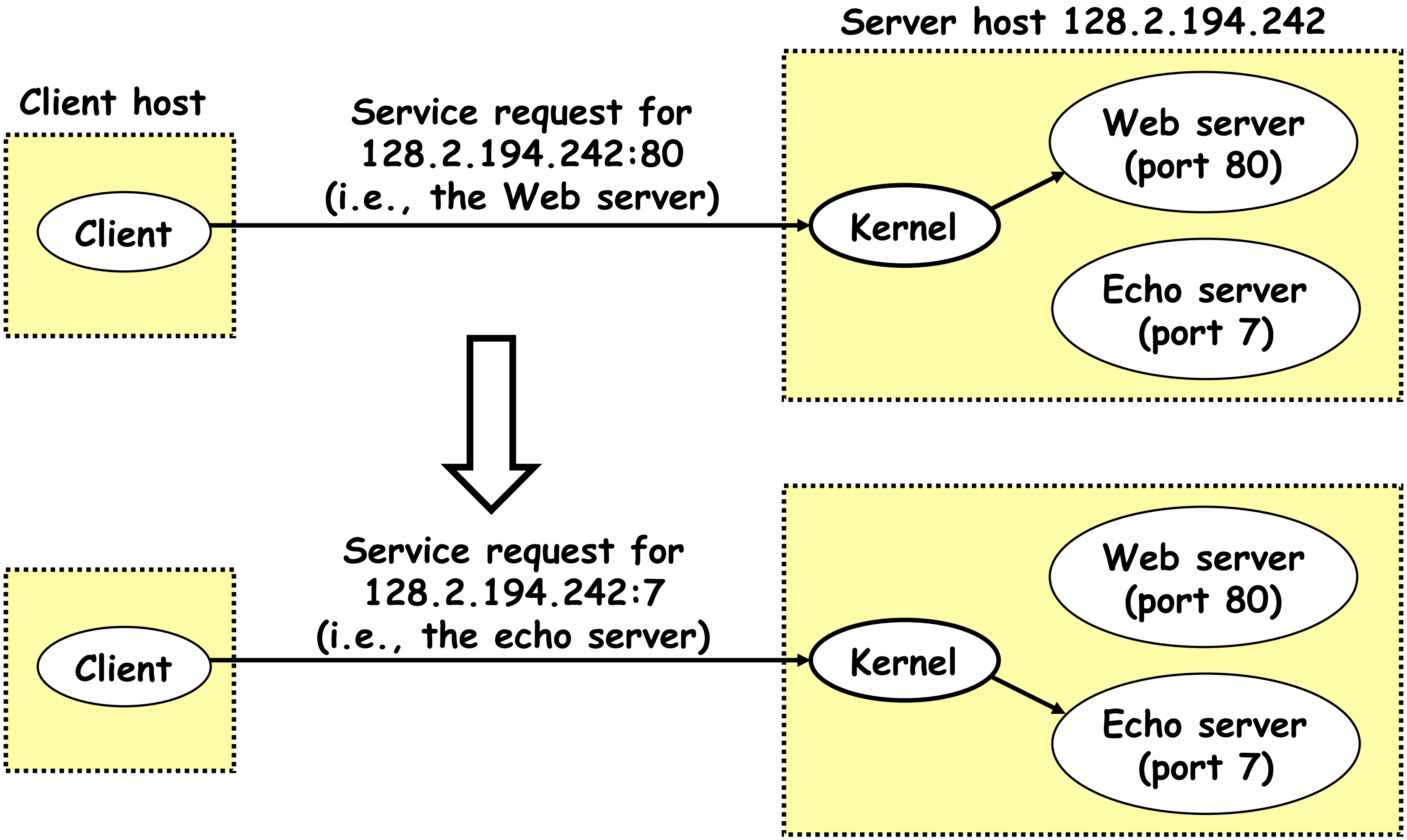
Note: 3479 is an ephemeral port allocated by the kernel

Note: 80 is a well-known port associated with Web servers

Clients

- Examples of client programs
  - Web browsers, `ftp`, `telnet`, `ssh`
- How does a client find the server?
  - The IP address in the server socket address identifies the host
  - The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
  - Examples of well known ports
    - Port 7: Echo server
    - Port 23: Telnet server
    - Port 25: Mail server
    - Port 80: Web server

Using Ports to Identify Services

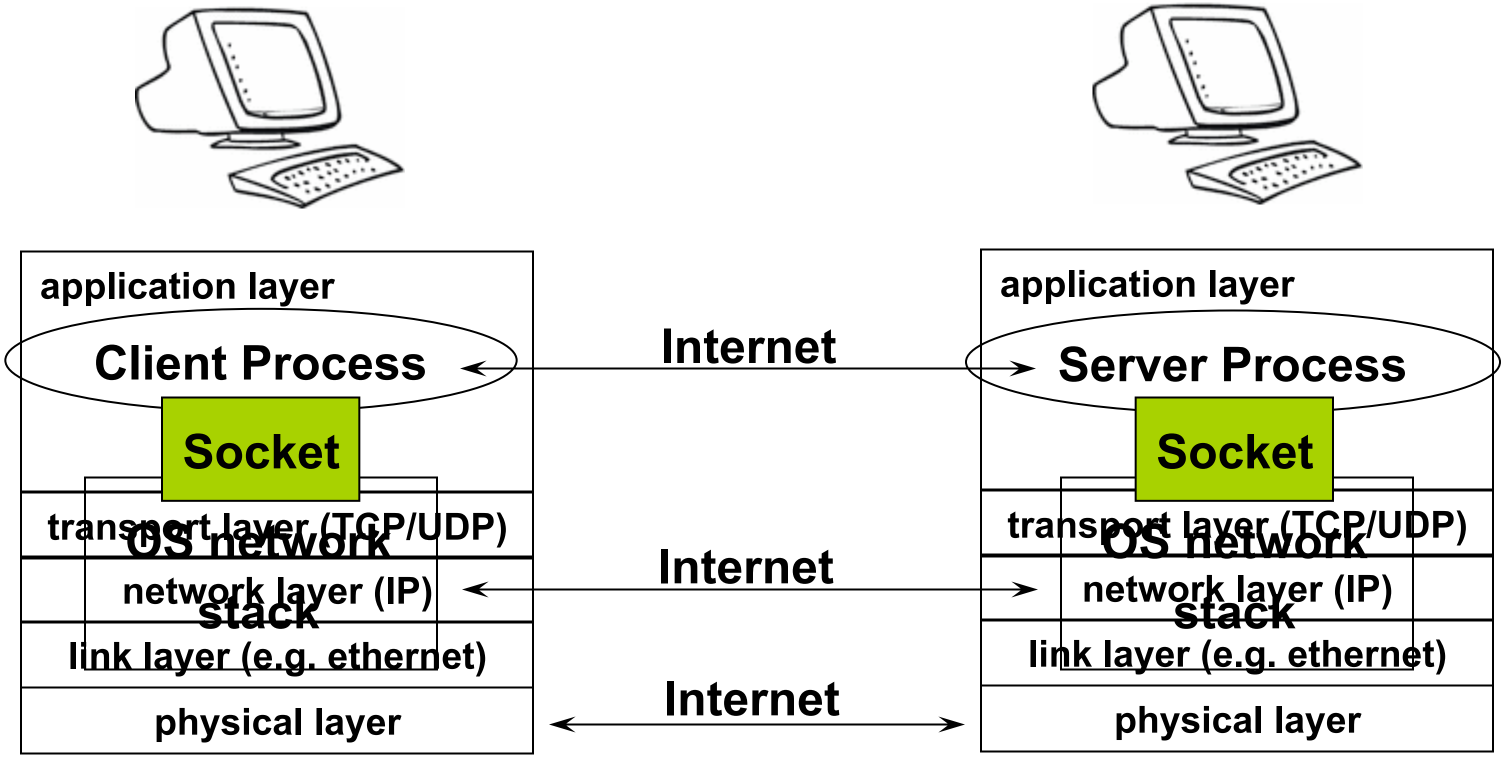


Servers

- Servers are long-running processes (daemons).
  - Created at boot-time (typically) by the init process (process 1)
  - Run continuously until the machine is turned off.
- Each server waits for requests to arrive on a well-known port associated with a particular service.
  - Port 7: echo server
  - Port 23: telnet server
  - Port 25: mail server
  - Port 80: HTTP server
- Other applications should choose between 1024 and 65535

See /etc/services for a comprehensive list of the services available on a Linux machine.

Sockets as means for inter-process communication (IPC)



The interface that the OS provides to its networking subsystem

Sockets

- What is a socket?
  - To the kernel, a socket is an endpoint of communication.
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network.
    - Remember: All Unix I/O devices, including networks, are modeled as files.

- Clients and servers communicate with each by reading from and writing to socket descriptors.

- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors.

Socket Programming Cliches

- Network Byte Ordering
  - Network is big-endian, host may be big- or little-endian
  - Functions work on 16-bit (short) and 32-bit (long) values
  - htons() / htonl() : convert host byte order to network byte order
  - ntohs() / ntohl(): convert network byte order to host byte order
  - Use these to convert network addresses, ports, …

```
struct sockaddr_in serveraddr;
/* fill in serveraddr with an address */
…
/* Connect takes (struct sockaddr *) as its second argument */
connect(clientfd, (struct sockaddr *) &serveraddr,
        sizeof(serveraddr));
…
```

- Structure Casts
  - You will see a lot of ‘structure casts’

Socket primitives

- SOCKET: **int socket(int *domain*, int *type*, int *protocol*);**
  - *domain* := AF_INET (IPv4 protocol)
  - *type* := (SOCK_DGRAM *or* SOCK_STREAM)
  - *protocol* := 0 (IPPROTO_UDP *or* IPPROTO_TCP)
  - *returned*: socket descriptor (*sockfd*), -1 is an error

- BIND: **int bind(int *sockfd*, struct sockaddr *my_addr*, int *addrlen*);**
  - *sockfd* - socket descriptor (returned from socket())
  - *my_addr:* socket address, struct sockaddr_in is used
  - *addrlen* := sizeof(struct sockaddr)

```
struct sockaddr_in  {
  unsigned short  sin_family;  /* address family (always AF_INET) */
  unsigned short  sin_port;    /* port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```

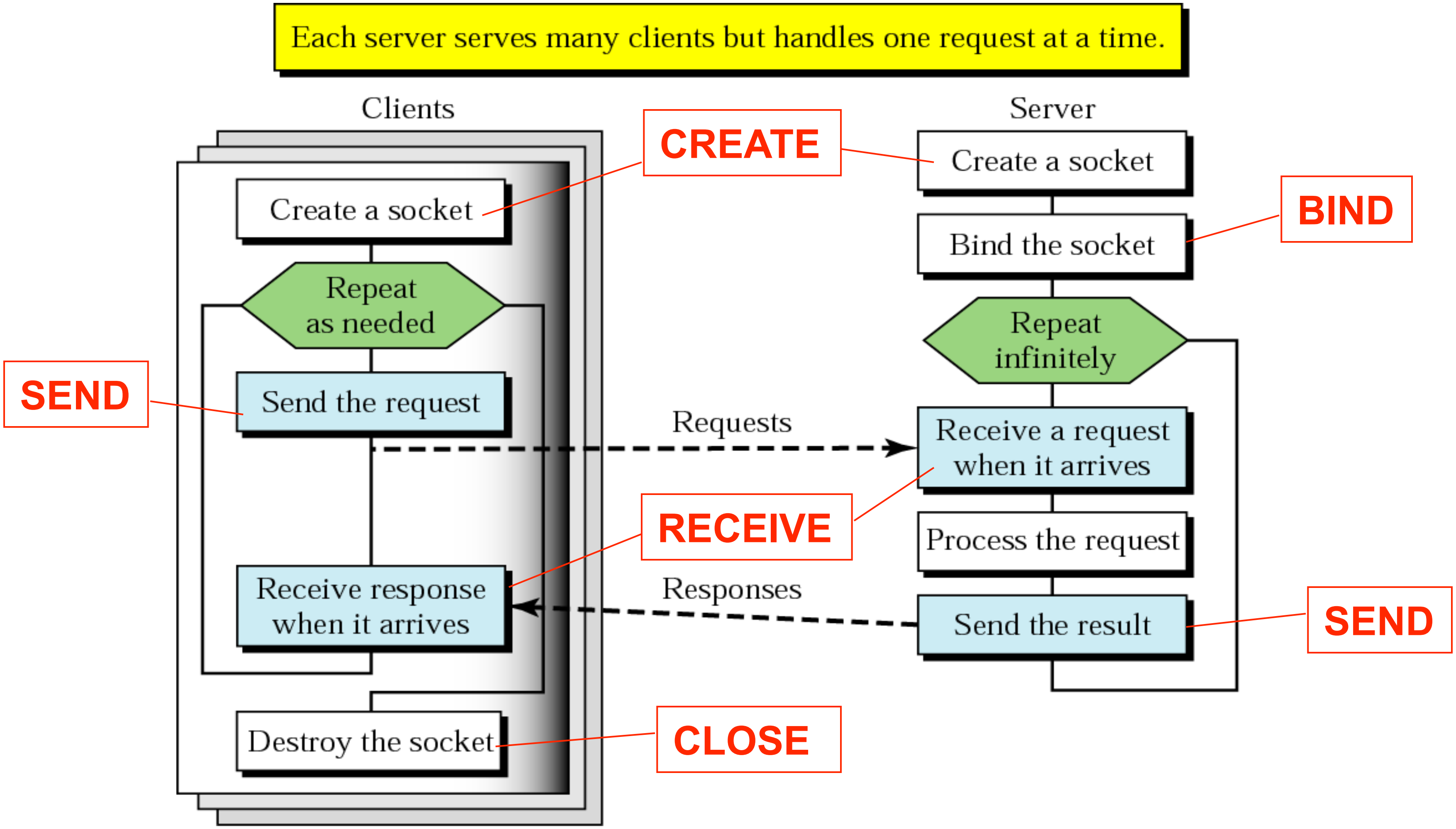
- LISTEN: **int listen(int *sockfd*, int *backlog*);**
  - *backlog*: how many connections we want to queue
- ACCEPT: **int accept(int *sockfd*, void **addr*, int **addrlen*);**
  - *addr*: here the socket-address of the caller will be written
  - *returned:* a new socket descriptor (for the temporal socket)
- CONNECT: **int connect(int *sockfd*, struct sockaddr **serv_addr*, int *addrlen*); //used by TCP client**
  - parameters are same as for bind()
- SEND: **int send(int *sockfd*, const void **msg*, int *len*, int *flags*);**
  - *msg*: message you want to send
  - *len:* length of the message
  - *flags* := 0
  - *returned:* the number of bytes actually sent
- RECEIVE: **int recv(int *sockfd*, void **buf*, int *len*, unsigned int *flags*);**
  - *buf:* buffer to receive the message
  - *len:* length of the buffer ("don't give me more!")
  - *flags* := 0
  - *returned:* the number of bytes received

- SEND (DGRAM-style): **int sendto(int *sockfd*, const void **msg*, int *len*, int *flags*, const struct sockaddr **to*, int *tolen*);**
  - *msg*: message you want to send
  - *len:* length of the message
  - *flags* := 0
  - *to:* socket address of the remote process
  - *tolen*: = sizeof(struct sockaddr)
  - *returned:* the number of bytes actually sent

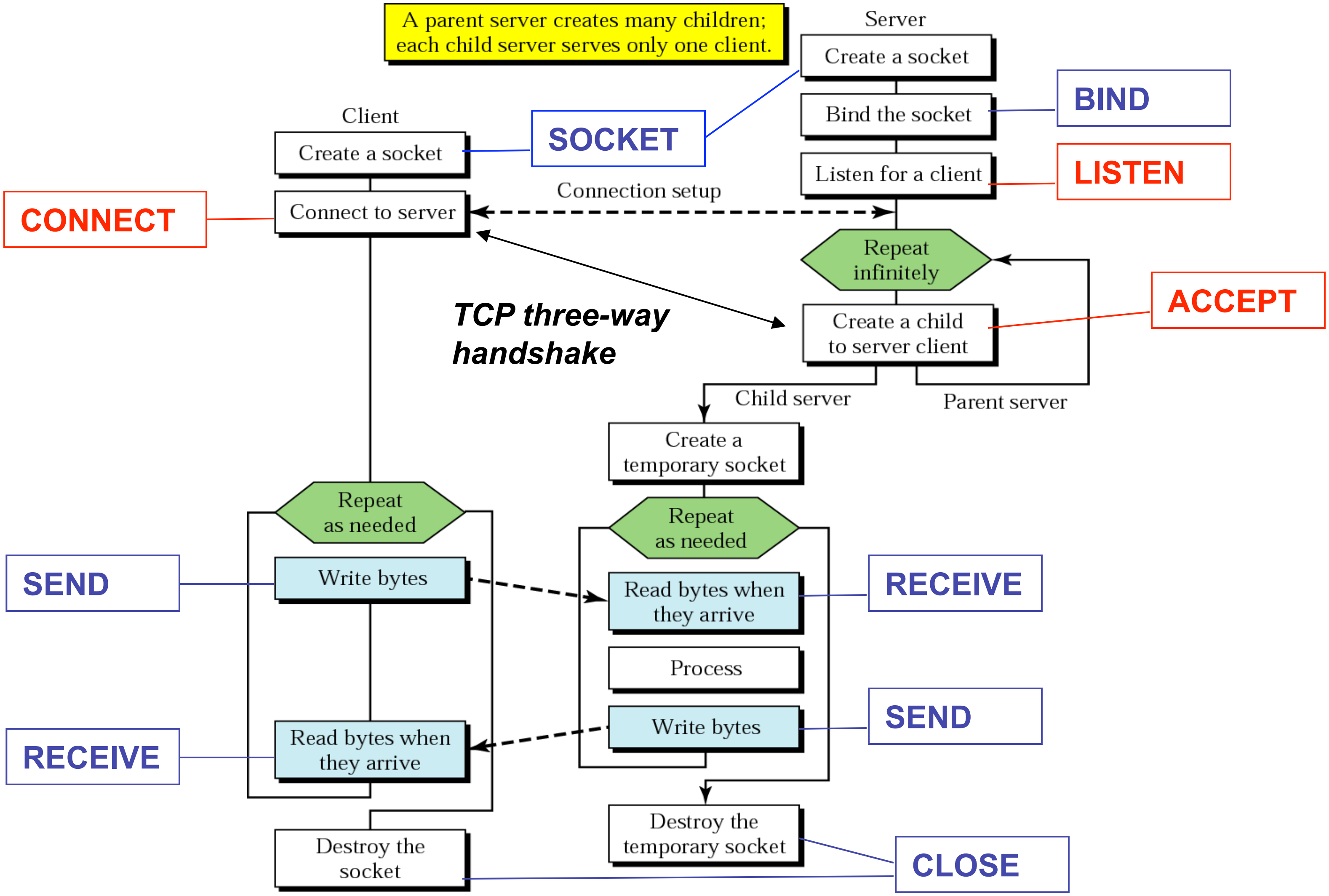
- RECEIVE (DGRAM-style): **int recvfrom(int *sockfd*, void **buf*, int *len*, unsigned int *flags*, struct sockaddr **from*, int **fromlen*);**
  - *buf:* buffer to receive the message
  - *len:* length of the buffer ("don't give me more!")
  - *from*: socket address of the process that sent the data
  - *fromlen*:= sizeof(struct sockaddr)
  - *flags* := 0
  - *returned:* the number of bytes received

- CLOSE: **close (*socketfd*);**

Client+server: connectionless

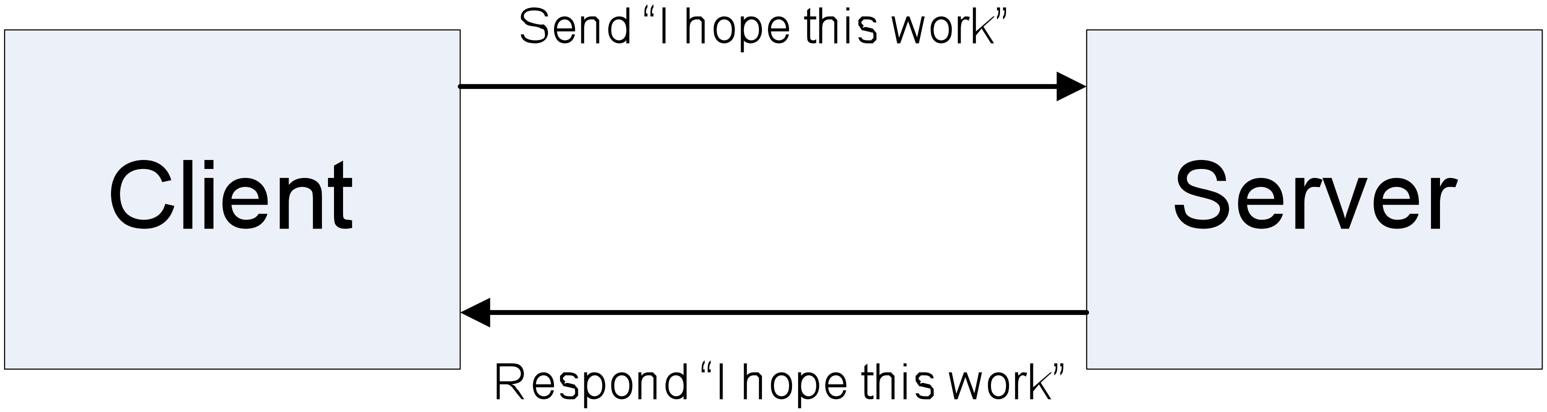


Client+server: connection-oriented



Concurrent server

Echo Client-Server



#include's

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(),
                            sendto(), and recvfrom() */
#include <arpa/inet.h>  /* for sockaddr_in and
                   inet_addr() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */
#include <netdb.h>      /* Transform the ip address
                          string to real uint_32 */

#define ECHOMAX 255     /* Longest string to echo */
```

EchoClient.cpp -variable declarations

```
int main(int argc, char *argv[])
{
    int sock;                          /* Socket descriptor */
    struct sockaddr_in echoServAddr; /* Echo server address */
    struct sockaddr_in fromAddr;      /* Source address of echo */
    unsigned short echoServPort =2000;     /* Echo server port */
    unsigned int fromSize;           /* address size for recvfrom() */
    char *servIP=“172.24.23.4”;   /* IP address of server */
    char *echoString=“I hope this works”;    /* String to send to
    echo server */
    char echoBuffer[ECHOMAX+1];      /* Buffer for receiving
    echoed string */
    int echoStringLen;                /* Length of string to echo */
    int respStringLen;                /* Length of received response */
```

EchoClient.c - creating the socket

```
/* Create a datagram/UDP socket  and
   error check */
sock = socket(AF_INET, SOCK_DGRAM,
   0);
if(sock <= 0){
    printf("Socket open error\n");
   exit(1);
  }
```

EchoClient.cpp – sending

```
/* Construct the server address structure */
memset(&echoServAddr, 0, sizeof(echoServAddr)); /* Zero out
    structure */
echoServAddr.sin_family = AF_INET; /* Internet addr family */
inet_pton(AF_INET, servIP, &echoServAddr.sin_addr); /* Server IP
    address */
echoServAddr.sin_port = htons(echoServPort); /* Server port */

/* Send the string to the server */
echoStringLen = strlen(echoString);
 sendto(sock, echoString, echoStringLen, 0, (struct sockaddr *)
    &echoServAddr, sizeof(echoServAddr);
```

EchoClient.cpp – receiving and printing

```
/* Recv a response */
fromSize = sizeof(fromAddr);
recvfrom(sock, echoBuffer, ECHOMAX, 0, (struct sockaddr *)
    &fromAddr, &fromSize);

/* Error checks like packet is received from the same server*/
…

/* null-terminate the received data */
echoBuffer[echoStringLen] = '\0';
printf("Received: %s\n", echoBuffer); /* Print the echoed arg */
close(sock);
exit(0);
} /* end of main () */
```

EchoServer.cpp - creating socket

```
int main(int argc, char *argv[])
{
    int sock;                        /* Socket */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned int cliAddrLen;         /* Length of incoming message */
    char echoBuffer[ECHOMAX];        /* Buffer for echo string */
    unsigned short echoServPort =2000;     /* Server port */
    int recvMsgSize;                 /* Size of received message */

   /* Create socket for sending/receiving datagrams */
   sock = socket(AF_INET, SOCK_DGRAM, 0);
   if(sock <= 0){
       printf("Socket open error\n");
       exit(1);
  }
```

EchoServer.cpp – binding

```
/* Construct local address structure*/
   memset(&echoServAddr, 0, sizeof(echoServAddr));  /* Zero out structure
  */
  echoServAddr.sin_family = AF_INET; /* Internet address family */
  echoServAddr.sin_addr.s_addr =htonl(INADDR_ANY);
  echoServAddr.sin_port = htons((uint16_t) echoServPort); /* Local port */


  /* Bind to the local address */
  int error_test = bind(sock, (struct sockaddr *) &echoServAddr,
  sizeof(echoServAddr));
  if(error_test < 0){
    printf("Binding error\n");
    exit(1);
  }
```

EchoServer.cpp – receiving and echoing

```
for (;;) /* Run forever */
   {
      cliAddrLen = sizeof(echoClntAddr);

      /* Block until receive message from a client */
      recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX, 0,
         (struct sockaddr *) &echoClntAddr, &cliAddrLen);

      printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

      /* Send received datagram back to the client */
      sendto(sock, echoBuffer, recvMsgSize, 0,
          (struct sockaddr *) &echoClntAddr, sizeof(echoClntAddr);
   }

} /* end of main () */
```

Error handling is must

Socket Programming Help

- man is your friend
  - man accept
  - man sendto
  - Etc.
- The manual page will tell you:
  - What #include<> directives you need at the top of your source code
  - The type of each argument
  - The possible return values
  - The possible errors (in errno)

Debugging with gdb

- Prepare program for debugging
  - Compile with "`-g`" (keep full symbol table)
  - Don't use compiler optimization ("`-O`", "`-O2`", …)
- Two main ways to run gdb
  - On program directly
    - `gdb progname`
    - Once gdb is executing we can execute the program with:
      - `run args`
  - On a core (post-mortem)
    - `gdb progname core`
    - Useful for examining program state at the point of crash
- Extensive in-program documentation exists
  - `help` (or `help <topic>` or `help <command>`)

More information…

- Socket programming
  - W. Richard Stevens, UNIX Network Programming
  - Infinite number of online resources
  - http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html
- GDB
  - Official GDB homepage: http://www.gnu.org/software/gdb/gdb.html
  - GDB primer: http://www.cs.pitt.edu/~mosse/gdb-note.html

Project Partners

- If you don't have a partner
  - Stay back after class

- Now…
  - Overview of PA 1