CPSC 313 Introduction to Computer Systems

Instructor: Guofei Gu

http://courses.cse.tamu.edu/guofei/csce313

Network Programming

- Network Programming as Programming across Machine Boundaries
- TCP/IP Basics
- The Sockets API
- Reliable Communication Channels: TCP
- Dangerous at any Speed; connectionless communication and UDP
- Server Design
- Reading: R&R, Ch 18 & Ch 22
Naming in a Networked Environment

Reminder:

- Naming within a **single address space**?
  - addresses, duh!

- Naming **across address spaces**?
  - file descriptors
  - filenames (use file system as name space)
  - keys

New:

- Naming **across machine boundaries**?

---

Naming in a Networked Environment

“what”?  

how?  

who?  

where?

“how”?  

TCP  

UDP  

HTTP 1.1

who?  

Telnet server  

Web server  

Printer server

where?

hostX.cs.tamu.edu  

128.144.xxxx.yyy
Protocols for a Networked Environment

Standard Application Protocols

- Standard applications: well documented, standardized and adopted as official TCP/IP suite
- Examples
  - email
  - telnet
  - ftp
  - More?

Let us look more closely at telnet
(multiple choice) telnet application can be used to
  A. Remotely access terminals
  B. Remotely access services
  C. Send an email
  D. Fetch a web page
**Fetching a web page using telnet**

1. Telnet to your favorite Web server:

   `telnet www.tamu.edu 80`  
   Opens TCP connection to port 80  
   (default http server port) at www.tamu.edu.  
   Anything typed in sent  
   to port 80 at www.tamu.edu

2. Type in a GET http request:

   ```
   GET /
   ```  
   By typing this in (hit carriage  
   return twice), you send  
   this minimal (but complete)  
   GET request to http server

3. Look at response message sent by http server!

---

**Scenario: Alice sends message to Bob**

1. Alice uses MUA to compose  
   message and “to”  
   `bob@tamu.edu`
2. Alice’s MUA sends message  
   to her mail server; message  
   placed in message queue
3. Client side of SMTP opens  
   TCP connection with Bob’s  
   mail server
4. MTA (Message Transfer  
   Agent) in mail server sends  
   Alice’s message over the TCP  
   connection
5. Bob’s mail server places the  
   message in Bob’s mailbox
6. Bob invokes his user agent to  
   read message
Sample smtp interaction

S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?  More messages?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection

Sending an email using telnet

- telnet servername 25
- HELO "your own domain name"
- MAIL FROM: <your own email address>
- RCPT TO: <recipient address>
- DATA
- QUIT
What can we learn here?

- telnet goes beyond standard terminal access and provides basic interactive communication facility

- telnet protocol provides maximal flexibility because it only defines interactive communication but not the details of the service accessed

- Users at many sites may want to access the same service at the same time. It is important yet challenging to provide concurrent services

Client-server model

Typical network app has two pieces: client and server.

**Client:**
- initiates contact with server ("speaks first")
- typically requests service from server,
- for Web, client is implemented in browser; for e-mail, in mail reader

**Server:**
- provides requested service to client
- e.g., Web server sends requested Web page, mail server delivers e-mail
TCP/IP BASICS

OSI and Protocol Stack

OSI Model | TCP/IP Hierarchy | Protocols
--- | --- | ---
7th Application Layer | | HTTP, SMTP, POP3, FTP, ...
6th Presentation Layer | Application Layer |
5th Session Layer |
4th Transport Layer | Transport Layer |
3rd Network Layer | Network Layer |
2nd Link Layer | Link Layer |
1st Physical Layer |

Link Layer: includes device driver and network interface card
Network Layer: handles the movement of packets, i.e. Routing
Transport Layer: provides a reliable flow of data between two hosts
Application Layer: handles the details of the particular application
Packet Encapsulation

- The data is sent down the protocol stack
- Each layer adds to the data by prepending headers

Ethernet

- Computer <-> Computer communication on same network
- Each device has unique MAC address (48-bit)
  example: 00-C0-4F-48-47-93

Ethernet Packet:

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Dest. address</th>
<th>Source address</th>
<th>Type</th>
<th>Data</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bytes</td>
<td>6 bytes</td>
<td>6 bytes</td>
<td>2 bytes</td>
<td>64 - 1500 bytes</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

MAC: Media Access Control
ARP: Address Resolution Protocol

- ARP provides mapping
  32bit IP address <-> 48bit MAC address
  128.97.89.153 <-> 00-C0-4F-48-47-93
- ARP cache
  maintains the recent mappings from IP addresses to MAC addresses

Protocol
1. ARP request broadcast on Ethernet
2. Destination host ARP layer responds

IP: Internet Protocol

- Unreliable … connectionless datagram delivery service
- Responsible for routing of data through intermediate networks and computers

IP header:
```
0123 4567 8901 11 2345 6789 2222 0123 2222 4567 2233 8901
```

<table>
<thead>
<tr>
<th>Version</th>
<th>IHL</th>
<th>Type of Service</th>
<th>Total Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification</td>
<td>Flags</td>
<td>Fragment Offset</td>
<td></td>
</tr>
<tr>
<td>Protocol</td>
<td>Header Checksum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source IP address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination IP address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP data payload (many bytes)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IP Routing

- Routing Table
  - Destination IP address
  - IP address of a next-hop router
  - Flags
  - Network interface specification

ICMP: Internet Control Message Protocol

- Used to report problems with delivery of IP Datagrams within an IP network
- Used by Ping, Traceroute commands

Types and Codes
- Echo Request (type=8, code=0)
- Echo Reply (type=0, code=0)
- Destination Unreachable (type=3, code=0)
- Time Exceeded (type=11, code=0): Time-to-Live = 0
TCP: Transmission Control Protocol

- Connection-Oriented, Reliable, Byte Stream Service

Protocol
1. Set up connection
2. Transfer data
3. Close connection

TCP Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0</td>
</tr>
<tr>
<td>Destination Port</td>
<td>1</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>2</td>
</tr>
<tr>
<td>Acknowledgement Number</td>
<td>3</td>
</tr>
<tr>
<td>Data Offset</td>
<td>4</td>
</tr>
<tr>
<td>Concat flags</td>
<td>5</td>
</tr>
<tr>
<td>Urgent pointer</td>
<td>6</td>
</tr>
<tr>
<td>Window</td>
<td>7</td>
</tr>
<tr>
<td>Checksum</td>
<td>8</td>
</tr>
<tr>
<td>Options</td>
<td>9</td>
</tr>
<tr>
<td>TCP Payload</td>
<td>10</td>
</tr>
</tbody>
</table>

Establishing a TCP Connection

- Client sends SYN seq=x
- Host receives SYN segment, sends SYN seq=y, ACK x+1
- Client receives ACK segment
- Client sends ACK y+1
- Host receives ACK segment

Closing a TCP Connection

- Client sends FIN seq=x
- Host receives FIN segment, sends ACK x+1
- Client receives ACK segment
- Client sends FIN seq=y, ACK x+1
- Host receives FIN segment
TCP: Data transfer

HTTP: Hyper Text Transfer Protocol

Steps in Transaction
1. Establish connection
2. Request
   Method <URL> <CR>
3. Response
   Response Code <Data> <CR>
4. Close connection
**HTTP**

- **Common Request Methods**
  - GET, PUT, POST

- **Response Categories**
  - Informational : 100
  - Successful : 200
  - Redirection : 300
  - Client Error : 400 (e.g., 404 Not found)
  - Server Error : 500

---

**Example: Access www.cse.tamu.edu**

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appl</td>
<td>Appl</td>
</tr>
<tr>
<td>HTTP</td>
<td>HTTP</td>
</tr>
<tr>
<td>Transp</td>
<td>Transp</td>
</tr>
<tr>
<td>TCP</td>
<td>TCP</td>
</tr>
<tr>
<td>Net</td>
<td>Net</td>
</tr>
<tr>
<td>IP</td>
<td>IP</td>
</tr>
<tr>
<td>Link</td>
<td>Link</td>
</tr>
<tr>
<td>ethernet</td>
<td>ethernet</td>
</tr>
</tbody>
</table>

- **Http Request**
  - GET "http://www.cse.tamu.edu"<CR>

- **Http Response**
  - 200 "" <CR> <html file in MIME format>

- **Initiate connection (tcpshk)**
- **Package data (add TCP header)**
- **Send http request packet**

- **Assemble response (break into several packets)**
- **Send http response packets**

- **Close connection (tcpshk)**

- **Relay data**
- **ARP to provide IP/MAC translation**
- **Router(s)**
References

- “TCP/IP Illustrated, Volume 1 The Protocols” by W. Richard Stevens
- “Internet Working with TCP/IP Volume 1” by Douglas E. Comer
- “Sams Teach Yourself TCP/IP in 24 Hours” by Joe Casad. Published by Sams. (http://www.informit.com)

SOCKET PROGRAMMING
Specifying a Protocol Interface

- Reality Check: “All” networks today are based on TCP/IP.

- How to define a network API, then?!
  - **Approach 1**: Define functions that specifically support TCP/IP communication.
    - e.g. `makeTCPConnection(int32 host, int16 portno);`
  - **Approach 2**: Define functions that support network communication in general, and use parameters to handle TCP/IP as a special case.

- The **socket API** provides generalized functions that support network communication using many possible protocols.
- The programmer specifies the type of service required rather than the name of a specific network protocol.

---

The Socket API

- **What does an API need to support?** [Comer & Stevens]
  - allocate local resources (buffers)
  - specify local and remote communication endpoints
  - initiate a connection
  - wait for an incoming connection
  - send or receive data
  - determine when data arrives
  - generate urgent data
  - handle incoming urgent data
  - terminate a connection gracefully
  - handle connection termination from the remote site
  - abort communication
  - handle error conditions or a connection abort
  - release local resources when communication finishes

- **Existing TCP/IP APIs:**
  - **Berkeley Socket API**
    - aka socket API, socket interface, sockets
    - adapted by Linux and others
  - **Windows Sockets**
  - **System V Unix TLI** (Transport Layer Interface)
What is a socket?

- All I/O in Unix is through file descriptors, including communication with a computer across Internet.
- An interface between application and network
  - The application creates a socket
  - The socket type dictates the style of communication
    - 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)

Sockets and File Descriptors

```c
#include <sys/socket.h>

int socket(int family, int type, int protocol)
/* creates communication endpoint and returns file descriptor. */
```

```c
int newsocket;
newsocket = socket(AF_INET, SOCK_STREAM, 0);
```
The socket() Call

```c
#include <sys/socket.h>

int socket(int family, int type, int protocol) /* creates communication endpoint and returns file descriptor. */
```

- Family
  - AF_INET: IPv4 protocols
  - AF_INET6: IPv6 protocols
  - AF_LOCAL: UNIX socket
  - AF_ROUTE: Routing socket
- Type
  - SOCK_STREAM: Stream (TCP) socket
  - SOCK_DGRAM: Datagram (UDP) socket
  - SOCK_RAW: Raw (IP) socket

TCP? UDP?!!

- Connection-oriented Style (SOCK_STREAM) of communication
- Implemented in TCP/IP as the Transport Control Protocol (TCP).
- TCP provides full reliability:
  - verifies that data arrives, with automatic retransmission
  - computes checksums to detect corruption
  - uses sequence numbers to guarantee ordering of received packets
  - automatically eliminates duplicated packets
  - provides flow control (ensures that sender does not send more packets than the receiver can handle)
  - informs sender if network becomes inoperable for any reason
- TCP protects network resources:
  - provides congestion control (throttles transmission when it detects that network is congested)
- What is the cost of all this?! Connection establishment overhead.
TCP? UDP?!!

- **Connectionless Style** (SOCK_DGRAM) of communication
- Implemented in TCP/IP as the User Datagram Protocol (UDP).
- UDP provides no guarantee about reliable delivery:
  - packets can be lost, duplicated, delayed, or delivered out of order
- UDP works well if the underlying network works well, e.g. local network

- **In practice**, programmers use UDP only when:
  1. The application requires that UDP must be used. (The application has been designed to handle reliability and delivery errors.)
  2. The application relies on hardware broadcast or multicast for delivery.
  3. Application runs in reliable, local environment, and overhead for reliability is unnecessary.

---

The Socket’s Local Address

```c
#include <sys/socket.h>

int socket(int family, int type, int protocol)
/* creates communication endpoint and returns file descriptor. */
```

- **newsocket**
  - family: AF_INET
  - service: SOCK_STREAM
  - local IP:
  - remote IP:
  - local port:
  - remote port:

Create new Socket Descriptor:

```c
int newsocket;
newsocket = socket(AF_INET, SOCK_STREAM, 0);
```
Defining the Socket’s Local Address (server)

Create new Socket Descriptor:

```c
int newsocket;
newsocket = socket(AF_INET, SOCK_STREAM, 0);
```

Generalized socket address:

(address family, endpoint address in that family)

Examples:

- Socket Addresses in AF_INET?

```c
struct sockaddr_in {
    u_short sin_family; /* type of address */
    u_short sin_port; /* protocol port number */
    struct in_addr sin_addr; /* IP address (declared to be*/  
                               /* u_long in some systems) */
    u_char sin_len;     /* total length */
    char sin_zero[8];  /* unused (set to zero) */
};
```

Defining the Socket’s Local Address (server)

```c
#include <sys/socket.h>
int bind(int socket, const struct sockaddr * address,  
         socketlen_t address_len);
/* associates socket file descriptor with 
   communication endpoint address. */
```

Examples:

- Socket Addresses in AF_INET?

```c
struct sockaddr_in {
    u_short sin_family; /* type of address */
    u_short sin_port; /* protocol port number */
    struct in_addr sin_addr; /* IP address (declared to be*/  
                               /* u_long in some systems) */
    u_char sin_len;     /* total length */
    char sin_zero[8];  /* unused (set to zero) */
};
```
Host vs. Network Byte Order

- Big-endian vs. Little-endian.
- Network representation requires big-endian.
- Portability?!

```c
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint32_t ntohl(uint32_t netlong);
uint16_t htons(uint16_t hostshort);
uint16_t ntohs(uint16_t netshort);
```

Associate port 8652 with a socket:

```c
struct sockaddr_in server;
int sock = socket(AF_INET, SOCK_STREAM, 0);
server.sin_family = AF_INET;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons((short)8652);
bind(sock, &server, sizeof(server));
```

Prepare to Accept Incoming Connections (server)

```c
#include <sys/socket.h>

int listen(int socket, int backlog);
/* specify willingness to accept incoming connections at given socket, with given queue limit.
Connections are then accepted with accept(). */
```

When request for connection from client comes in, both client and server execute hand-shake procedure to set up the connection.

When server is busy, two things can happen
- Connection request is queued (as long as backlog not exceeded)
- Connection request is refused (client receives ECONNREFUSED error)
Handle Incoming Connections (server)

```c
#include <sys/socket.h>
int accept(int socket,
            struct sockaddr * address,
            socklen_t * address_len);
/* Accept a connection on a socket.
   Create a new socket with same properties of "socket" and
   return new file descriptor. */
```

- Extract first connection request on queue of pending connections.
- Create new socket with same properties of given socket.
- Returns new file descriptor for the socket.
- Blocks caller if no pending connections are present in queue.
- New socket may not be used to accept more connections.
- Original socket `socket` remains open.
- Argument `address` (result parameter) is filled in with the address of connecting entity.
- Argument `address_len` (value-result parameter) initially contains length of space pointed to by `address`. On return it contains length of actual length of space.

In the meantime, at the Client’s End (client)

```c
#include <sys/socket.h>
int connect(int socket,
            struct sockaddr * address,
            socklen_t address_len);
/* Initiate a connection on a socket.
   Structure `address` specifies the
   other endpoint of the connection. */
```

Note difference between SOCK_DGRAM and SOCK_STREAM sockets:

- For SOCK_STREAM sockets, `connect` attempts to establish connection with other socket.
  - Generally, stream sockets may connect only once.

- For SOCK_DGRAM sockets, `connect` specifies the peer with which to associate socket.
  - Datagram sockets may dissolve association by connecting to invalid address, such as null address.

- For `address`, fill in `family`, `address`, and `port number`.
Connection Establishment: Summary

**Client**
- `socket()`
- `connect()`
- `read()` / `write()`
- `close()`

**Server**
- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `read()` / `write()`
- `close()`

---

#include <netdb.h>
struct hostent *
gethostbyname(char * name);

#include <arpa/inet.h>
in_addr_t
inet_addr(char * cp);

char *
inet_ntoa(struct in_addr in);
/* return dot notation from IP address */

Addressing: IP Address vs. Dotted vs. Name

**Internet Address Dot Notation**
- "128.10.2.3"

**Internal conversion**

**IP Address (IPv4)**
- <some 32 bit number>

**Host name resolution using Domain Name Service (DNS)**

#include <netdb.h>
struct hostent *
gethostbyname(char * name);
struct hostent *
gethostbyaddr(void * addr,
socklen_t len, int type);
/* return info about names and addresses */

**Host Name**
- "merlin.cs.purdue.edu"
Host Names and IP Addresses

Host Information:
```c
#include <netdb.h>

struct hostent {  
    char * h_name;       /* canonical name of host */  
    char ** h_aliases;    /* alias list */  
    int h_addrtype;       /* host address type */  
    int h_length;         /* length of address */  
    char ** h_addr_list;  /* list of addresses */  
};
```

Example: Translate a host name into IP address for use in `connect()` call:
```c
struct hostent * hp;  
struct sockaddr_in the_server;  
  
if ((hp = gethostbyname("www.cs.tamu.edu")) == NULL)  
    fprintf(stderr, "Failed to resolve host name\n");  
else  
    memcpy((char*)&the_server.sin_addr.s_addr,  
           hp->h_addr_list[0], hp->h_length);  
```

Example TCP Client: DAYTIME client [Comer]

```c
#define LINELEN 128  
/* forward */  
int connectTCP(const char * host, const char * service);  
/* main program */  
int main(argc, char * argv) {  
    char * host    = "localhost";  
    char * service = "daytime";  
    if (argc > 1) host    = argv[1];  
    if (argc > 2) service = argv[2];  
    int s = connectTCP(host, service);  
    while ( (int n = read(s, buf, LINELEN)) > 0) {  
        buf[n] = '\0';  
        (void) fputs(buf, stdout);  
    }  
}  
```
Example TCP Client: (cont...)

```c
#define LINELEN 128

int connectTCP(const char * host, const char * service) {
    struct sockaddr_in sin; /* Internet endpoint address */
    sin.sin_family = AF_INET;
    /* Map service name to port number */
    if (struct servent * pse = getservbyname(service, "tcp") )
        sin.sin_port = ntohs(pse->s_port);
    else if ( (sin.sin_port = htons((unsigned short)atoi(service))) == 0 )
        errexit("can't get <&s> service entry\n", service);
    /* Map host name to IP address, allowing for dotted decimal */
    if (struct hostent * phe = gethostbyname(host) )
        memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
    else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
        errexit("can't get <&s> host entry\n", host);
    /* Allocate socket */
    int s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) errexit("can't create socket: %s\n", strerror(errno));
    /* Connect the socket */
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        errexit("can't connect to %s.%s: %s\n", host, service, strerror(errno));
    return s;
}
```

Connection Oriented Example
Connection oriented Server

```c
#define SERVER_PORT 3005
#define BUFFER_LENGTH 250
#define FALSE 0

void main() {
  int sd=-1, sd2=-1;
  int rc, length, on=1;
  char buffer[BUFFER_LENGTH];
  fd_set read_fd;
  struct timeval timeout;
  struct sockaddr_in serveraddr;

  do {
    sd = socket(AF_INET, SOCK_STREAM, 0);
    // test error: sd < 0)
    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(SERVER_PORT);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
    // test error rc < 0
    rc = listen(sd, 10);
    // test error rc< 0
    printf("Ready for client connect().\n");
    sd2 = accept(sd, NULL, NULL);
    // test error sd2 < 0
    timeout.tv_sec  = 30;
    timeout.tv_usec = 0;
    FD_ZERO(&read_fd);
    FD_SET(sd2, &read_fd);
    rc = select(sd2+1, &read_fd, NULL, NULL, &timeout);
    // test error rc < 0
    length = BUFFER_LENGTH;
    rc = recv(sd2, buffer, sizeof(buffer), 0);
    // test error rc < 0 or rc == 0 or rc < sizeof(buffer
    rc = send(sd2, buffer, sizeof(buffer), 0);
    // test error rc < 0
  } while (FALSE);
  if (sd != -1)
    close(sd);
  if (sd2 != -1)
    close(sd2);
}
```

Connection oriented Server

```c
print("Ready for client connect().\n");

sd2 = accept(sd, NULL, NULL);
// test error sd2 < 0

timeout.tv_sec = 30;
timeout.tv_usec = 0;
FD_ZERO(&read_fd);
FD_SET(sd2, &read_fd);
rc = select(sd2+1, &read_fd, NULL, NULL, &timeout);
// test error rc < 0

length = BUFFER_LENGTH;

rc = recv(sd2, buffer, sizeof(buffer), 0);
// test error rc < 0 or rc == 0 or rc < sizeof(buffer

rc = send(sd2, buffer, sizeof(buffer), 0);
// test error rc < 0
}
while (FALSE):
  if (sd2 != -1)
    close(sd2);
  if (sd2 != -1)
    close(sd2);
```
Connection oriented Client

```c
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
#define FALSE           0
#define SERVER_NAME     "ServerHostName"

void main(int argc, char *argv[]) {
    int    sd=-1, rc, bytesReceived;
    char   buffer[BUFFER_LENGTH];
    char   server[NETDB_MAX_HOST_NAME_LENGTH];
    struct sockaddr_in serveraddr;
    struct hostent *hostp;
    do   {
        sd = socket(AF_INET, SOCK_STREAM, 0);
        // test error sd < 0
        if (argc > 1)  strcpy(server, argv[1]);
        else  strcpy(server, SERVER_NAME);
        memset(&serveraddr, 0, sizeof(serveraddr));
        serveraddr.sin_family      = AF_INET;
        serveraddr.sin_port        = htons(SERVER_PORT);
        serveraddr.sin_addr.s_addr = inet_addr(server);
        if (serveraddr.sin_addr.s_addr == (unsigned long)INADDR_NONE)      {
            hostp = gethostbyname(server);
            if (hostp == (struct hostent *)NULL) {
                printf("Host not found --> ");
                break;
            }
            memcpy(&serveraddr.sin_addr, hostp->h_addr, sizeof(serveraddr.sin_addr));
        }
        rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
        // test error rc < 0
        memset(buffer, 'a', sizeof(buffer));
        rc = send(sd, buffer, sizeof(buffer), 0);
        // test error rc < 0
        bytesReceived = 0;
        while (bytesReceived < BUFFER_LENGTH) {
            rc = recv(sd, & buffer[bytesReceived], BUFFER_LENGTH - bytesReceived, 0);
            // test error rc < 0 or rc == 0
            bytesReceived += rc;
        }
    } while (FALSE);
    if (sd != -1)
        close(sd);
}
```
Connectionless Example

Connectionless Server

```c
#define SERVER_PORT 3555
#define BUFFER_LENGTH 100
#define FALSE 0

void main() {
    int sd=-1, rc;
    char buffer[BUFFER_LENGTH];
    struct sockaddr_in serveraddr;
    struct sockaddr_in clientaddr;
    int clientaddrlen = sizeof(clientaddr);

    do {
        sd = socket(AF_INET, SOCK_DGRAM, 0);
        if (sd < 0)
            // test error rc < 0

        memset(&serveraddr, 0, sizeof(serveraddr));
        serveraddr.sin_family = AF_INET;
        serveraddr.sin_port = htons(SERVER_PORT);
        serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

        rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
        // test error rc < 0
    }

    while (rc < 0);
```

### Connectionless Server

```c
rc = recvfrom(sd, buffer, sizeof(buffer), 0, (struct sockaddr *)&clientaddr, &clientaddrlen);
// test error rc < 0

printf("server received the following: <%s>\n", buffer);
printf("from port %d and address %s\n", ntohs(clientaddr.sin_port), inet_ntoa(clientaddr.sin_addr));

rc = sendto(sd, buffer, sizeof(buffer), 0, (struct sockaddr *)&clientaddr, sizeof(clientaddr));
// test error rc < 0
```

### Connectionless Client

```c
#define SERVER_PORT     3555
#define BUFFER_LENGTH    100
#define FALSE              0
#define SERVER_NAME     "ServerHostName"

void main(int argc, char *argv[]) {
int    sd, rc;
char   server[NETDB_MAX_HOST_NAME_LENGTH];
char   buffer[BUFFER_LENGTH];
struct hostent  *hostp;
struct sockaddr_in serveraddr;
int    serveraddrlen = sizeof(serveraddr);

do   {
    sd = socket(AF_INET, SOCK_DGRAM, 0);
    // test error sd < 0

    if (argc > 1) strcpy(server, argv[1]);
    else strcpy(server, SERVER_NAME);

    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin_family      = AF_INET;
    serveraddr.sin_port        = htons(SERVER_PORT);
    serveraddr.sin_addr.s_addr = inet_addr(server);
    if (serveraddr.sin_addr.s_addr == (unsigned long) INADDR_NONE)   {
        hostp = gethostbyname(server);  // get the IP address from the dotted decimal
        if (hostp == (struct hostent *) NULL)   {
            printf("Host not found --> ");
            break;
        }
    }
    
if (sd != -1)
    close(sd);
}
```
Connectionless Client

```c
memcpy(&serveraddr.sin_addr, hostp->h_addr, sizeof(serveraddr.sin_addr));

memset(buffer, 0, sizeof(buffer));
strcpy(buffer, "A CLIENT REQUEST");
rc = sendto(sd, buffer, sizeof(buffer), 0, (struct sockaddr *)&serveraddr,
sizeof(serveraddr));
// test error rc < 0

rc = recvfrom(sd, buffer, sizeof(buffer), 0,
(struct sockaddr *)&serveraddr,
& serveraddrlen);
// test error rc < 0
printf("client received the following: <%s>\n", buffer);
} while (FALSE);
if (sd != -1)
close(sd);
```
Server Software Design: Issues

- **Concurrent vs. Iterative Servers:**
  The term *concurrent server* refers to whether the server permits multiple requests to proceed concurrently, **not** to whether the underlying implementation uses multiple, concurrent threads of execution.
  Iterative server implementations are easier to build and understand, but may result in poor performance because they make clients wait for service.

- **Connection-Oriented vs. Connectionless Access:**
  Connection-oriented (TCP, typically) servers are easier to implement, but have resources bound to connections.
  Reliable communication over UDP is not easy!

- **Stateful vs. Stateless Servers:**
  How much information should the server maintain about clients? (What if clients crash, and server does not know?)

---

**Server Design Combinations**

- **Iterative**
  - simple
  - diagnostic programs

- **Concurrent**
  - multi-user (time slice, multi-core)
  - multi-processor, multi-process, multi-thread, async IO
  - performance

- **Connectionless (UDP)**
  - simple
  - fast

- **Connection (TCP)**
  - reliable pipe
  - 3 way handshaking (SYN attack)
  - Nothing going on when idle
    - resource exhaustion
    - client crash – unintentional or intentional
    - memory leak
Iterative UDP Server

- Other clients block while one request is processed, not for a full connection time.
- Each subsequent recvfrom can be from a different client.
- Server identifies client by from_addr.
- Not reliable, but no connection overhead.

Iterative TCP Server

- Work on new sock until close then, go back to accept.
Example: Iterative, Connection-Oriented Server

```c
int passiveTCPsock(const char * service, int backlog) {
    struct sockaddr_in sin; /* Internet endpoint address */
    memset(&sin, 0, sizeof(sin)); /* Zero out address */
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    /* Map service name to port number */
    if (struct servent * pse = getservbyname(service, "tcp") )
        sin.sin_port = pse->s_port;
    else if ((sin.sin_port = htons((unsigned short)atoi(service))) == 0)
        errexit("can't get <\s service entry\n", service);
    /* Allocate socket */
    int s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) errexit("can't create socket: %s", strerror(errno));
    /* Bind the socket */
    if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        errexit("can't bind to ...");
    /* Listen on socket */
    if (listen(s, backlog) < 0)
        errexit("can't listen on ...");
    return s;
}
```

Example: Iterative, Connection-Oriented Server

```c
int main(int argc, char * argv[]) {
    char * service = "daytime"; /* service name or port number */
    int    m_sock, s_sock;      /* master and slave socket */
    service = argv[1];
    m_sock = passiveTCPsock(service, 32);
    for (;;) {
        s_sock = accept(m_sock, (struct sockaddr *)&fsin, sizeof(fsin));
        if (s_sock < 0) errexit("accept failed: %s", strerror(errno));
        time_t now;
        time(&now);
        char * pts = ctime(&now);
        write(s_sock, pts, strlen(pts));
        close(s_sock);
    }
}
```
Possible Server Deadlocks

- A deadlock occurs when
  - The client waits for the server
  - The server waits for the client
- A simple deadlock

```
Server     Client
accept()  <------>  connect()
     |                  |
recv()     recv()
```

Concurrent UDP Server

M1. sock = socket (PF_INET, SOCK_DGRAM, 0);
M2. bind (sock, localaddr, addrlen)
  - For well-known service, use getservbyname (name, protocol) to get port number from service name
  - Often use wildcard address (INADDR_ANY) for host IP
M3. recvfrom (sock, buf, buflen, flags, from_addr, from_addrlen)
  - Create a new process (fork) or thread (pthread_create)

S1. Process the request from the master
  - Send the reply to the client using sendto (sock, buf, buflen, flags, toaddr, toaddrlen)
  - Socket number (sock) is inherited from the master
S2. exit

S3. slave spawn by master
slave exit after one request
Concurrent TCP Server

Example: Concurrent, Connection-Oriented Server
Cleaning Child Processes

- The life cycle of a child process
  - creation
  - exit (in zombie state)
    - Maintain information of child process (PID, resource utilization, exit status)
  - waited by its parent
- Zombie processes take up system resources
- How to clean zombie processes?
  - Kernel sends SIGCHLD signal to parent process

Handling Interrupted System Calls

- When a process is blocked in a slow system call (e.g., accept) and the process catches a signal and the signal handler returns, the system call can return an error of EINTR
  - Some kernels restart interrupted system calls others don’t.

```c
sock = accept(msock, (struct sockaddr *)&fsin, &alen);
if (sock < 0) {
  if (errno == EINTR)
    continue;
  errexit("accept: %s
", strerror(errno));
}
```

Returned by the LinuxKernel if a system call was interrupted prematurely with a Signal before it was able to complete. The system call is usually retried.
Summary of Connection-Oriented server

1. We must catch SIGCHLD signal when forking child processes
2. A SIGCHLD handler must be coded correctly using wait3() or waitpid() to prevent zombies from being left behind
3. We must handle interrupted system calls when we install signal handlers

Example: Concurrent, Connection-Oriented Server

```c
int main(int argc, char * argv[]) {
    char * service = argv[1]; /* service name or port number */
    int m_sock, s_sock; /* master and slave socket */
    service = argv[1];
    m_sock = passiveTCPsock(service, 32);
    signal(SIGCHLD, cleanly_terminate_child);
    for (;;) {
        s_sock = accept(m_sock, (struct sockaddr*)&fsin, sizeof(fsin));
        if (s_sock < 0)
            if (errno == EINTR) continue;
        else errexit(
            "accept failed: %s",
            strerror(errno));
        if (fork() == 0) { /* child */
            close(s_sock);
            /* handle request here . . . */
        }
        close(m_sock);
    }
    void cleanly_terminate_child(int sig) {
        int status;
        while (wait3(&status, WNOHANG, NULL) >= 0)
    }
```
Using Thread

Example: **Concurrent, Connection-Oriented Server**

```c
int passiveTCPsock(const char * service, int backlog) {
    // Implementation
}

int main(int argc, char * argv[]) {
    char * service = "daytime"; /* service name or port number */
    int m_sock, s_sock;      /* master and slave socket */
    service = argv[1];
    m_sock = passiveTCPsock(service, 32);

    // Create slave threads
    pthread_t th; pthread_attr_t ta;
    pthread_attr_init(&ta);
    pthread_attr_setdetachstate(&ta, PTHREAD_CREATE_DETACHED);
    for (;;) {
        s_sock = accept(m_sock, (struct sockaddr *)&fsin, sizeof(fsin));
        if (s_sock < 0)
            if (errno == EINTR) continue;
        else errexit("accept failed: %s", strerror(errno));
        pthread_create(&th, &ta, handle_request, (void*)s_sock);
    }
}

int handle_request(int fd) {
    /* handle the request ... */
    close(s_sock);
}
```
Non-Blocking I/O: Using select Function

- Wait for any one of multiple IO events to occur
- Return if one or more of the IO events occur or a specified amount of time has passed
- Example: we may call return to wait for the following events
  - Any of the descriptors in the set [1,4,5] are ready for reading
  - Any of the descriptors in the set [2,7] are ready for writing
  - Any of the descriptors in the set [1,4] have an exception condition pending
  - T seconds have passed

Example: Concurrent, Connection-Oriented Server

```c
int passiveTCPsock(const char * service, int backlog);

int main(int argc, char * argv[]) {
    char * service = "daytime"; /* service name or port number */
    int m_sock, s_sock; /* master and slave socket */
    service = argv[1];
    int m_sock = passiveTCPsock(service, 32);
    FD_set rfds, afds;
    int nfds = getdtablesize();
    FD_ZERO(&afds); FD_SET(m_sock, &afds);
    for (;;)
        memcpy(&rfds, &afds, sizeof(rfds));
        select(nfds, &rfds, 0, 0, 0);
        if(FD_ISSET(m_sock, &rfds) {
            s_sock = accept(m_sock,(struct sockaddr*)&fsin, sizeof(fsin));
            FD_SET(s_sock, &afds);
            for(int fd = 0; fd < nfds; fd++)
                if (fd != m_sock && FD_ISSET(fd, &rfds)) {
                    /* handle request . . . */
                    close(fd);
                    FD_CLR(fd, &afds);
                }
        }
    close();
}
```
**Multiprotocol Server**

- A single thread of execution that handles both TCP and UDP requests
- Useful when one service can be accessed by either TCP or UDP
  - DAYTIME, TIME, ECHO
- Advantages?
  - resource usage
  - code maintenance

**Multiprotocol DAYTIME Server**

![Diagram of Multiprotocol DAYTIME Server process]

- Both TCP and UDP sockets are watched with the `select` function.
- The server first accepts incoming connections using `accept`.
- Following the `accept` operation, the server processes the request through `process_request`.
- Upon processing, the socket is closed.
- After processing the response, the server sends the response using `sendto`.
Concurrent Multiservice TCP Server

- Master socket, one for each service
- Socket for individual slave connections

SELECT