Computer Networks (CS F303)

BITS Pilani
Pilani Campus

Virendra Singh Shekhawat
Department of Computer Science and Information Systems

innovate achieve lead

BITS Pilani

Pilani Campus

Second Semester 2020-2021

Module-2 Application Layer

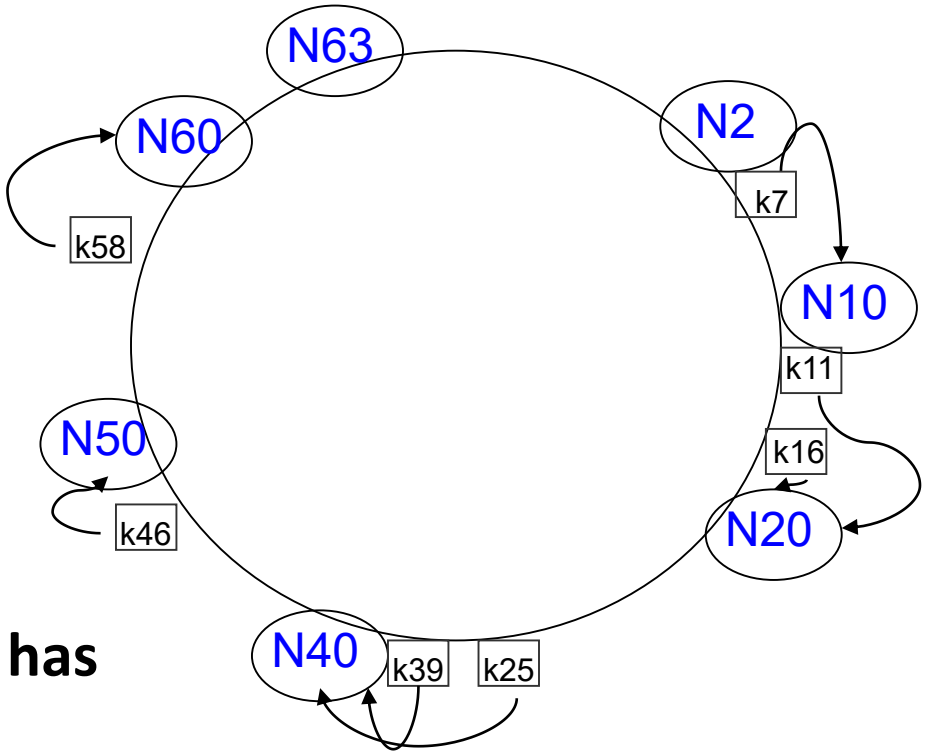
Database Implementation [..2] Circular DHT

- Hash function assigns each "**node**" *and* "**key**" an m-bit *identifier* using a base hash function such as SHA-1
  - Node_ID = hash(IP, Port)
  - Key_ID = hash(original key)

ID Space: 0 to 2^m-1

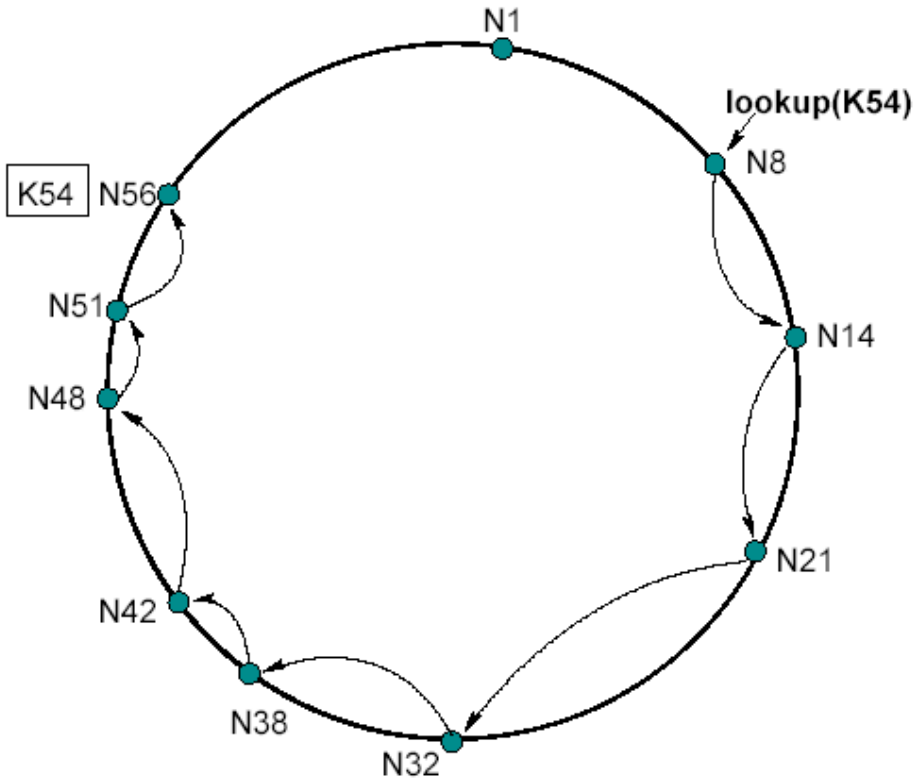
Here: *m = 6*

Range = 64



Assign (key-value) pair to the peer that has the *closest* ID.

Chord Protocol:Lookup Operation Example



Predecessor: pointer to the previous node on the id circle

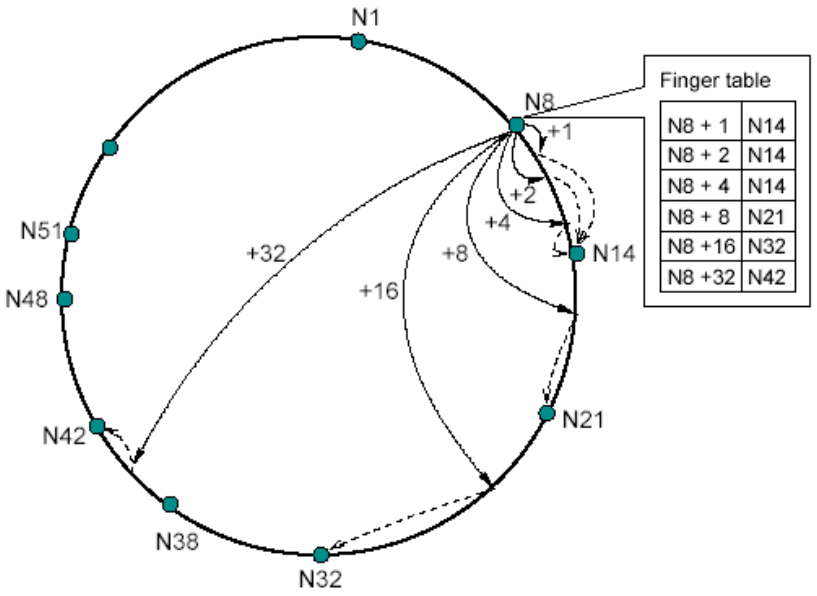
Successor: pointer to the succeeding node on the id circle

- ask node ***n*** to find the successor of ***id***
- If ***id*** between ***n*** and its ***successor***
  return ***successor***
- else forward query to ***n´***s ***successor*** and so on

=>#messages linear in #nodes

Scalable node localization

- Each node n contains a routing table with up-to m entries (**m**: number of bits of the identifier) => **finger table**
- i^{th} entry in the table at node **n** contains the first node **s** that succeds **n** by at least $\mathbf{2^{i-1}}$
  - **s = successor ($n + 2^{i-1}$)**
  - **s** is called the ith finger of node **n**

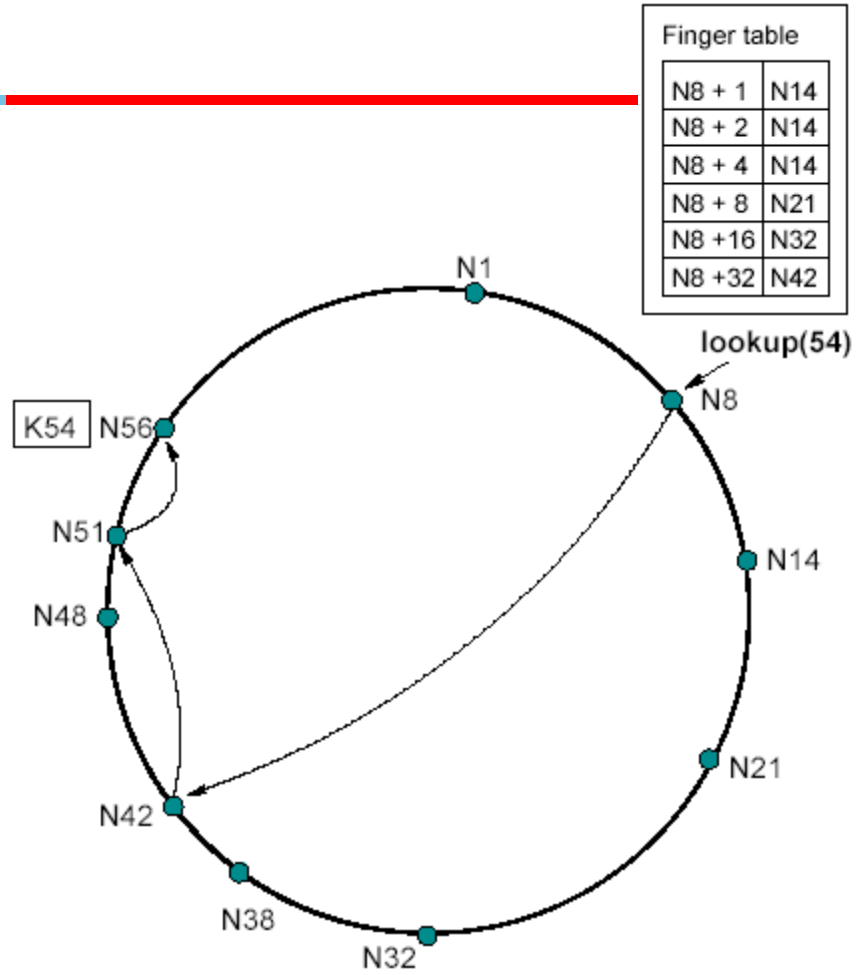


The Chord algorithm – Scalable node localization

- Search in finger table for the node which is

  most immediatly precedes key

- Invoke **find_successor** from that node



Number of messages O(log N)!

Failure Recovery (Peer Churn)

- Key step in failure recovery is maintaining correct successor pointers
- To achieve this, *each node maintains a successor-list* of its *r* nearest successors on the ring
- *If node n notices that its successor has failed*, it replaces it with the first live entry in the list
- The ***stabilize*** will correct finger table entries and successor-list entries pointing to failed node
- Stabilization protocol should be invoked based on the frequency of nodes leaving and joining

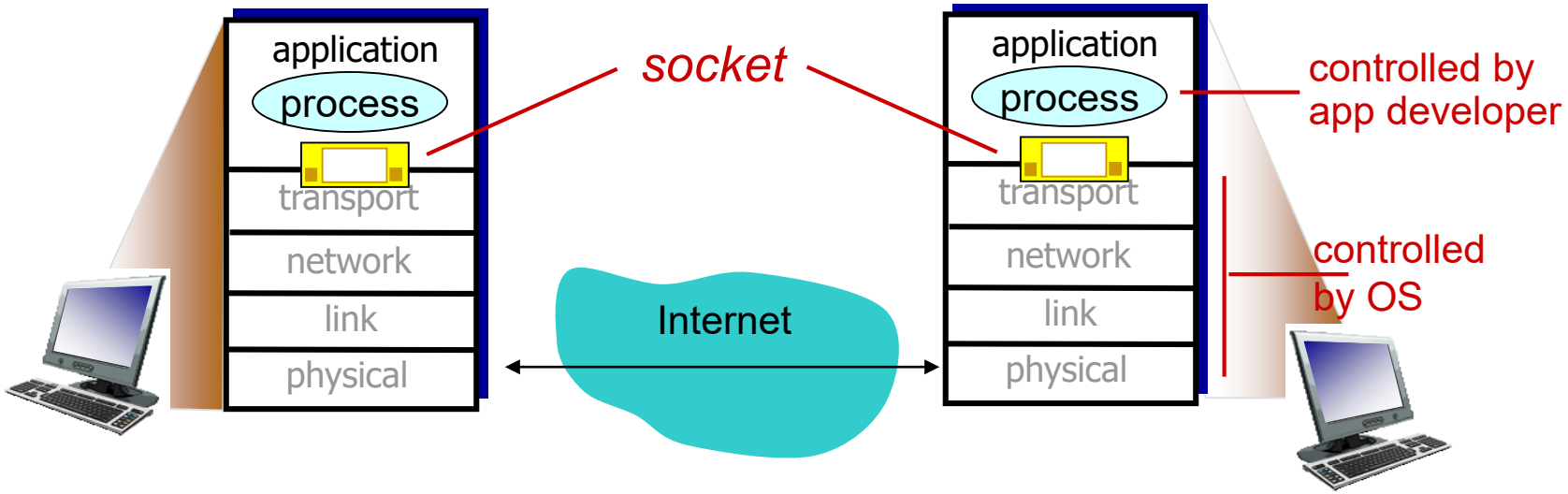
Next…

- Creating network Applications
  - Socket Programming
    - TCP vs. UDP Sockets
- Transport Layer
  - Transport Layer Services
    - Multiplexing/Demultiplexing
      - Connectionless and Connection Oriented
        - » TCP and UDP
    - Reliable data transfer (Protocol design)
    - Flow control
    - Congestion control

Socket Programming [.1]

- 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 other by reading from and writing to socket descriptors.



Socket Programming [..2]

Two socket types for two transport services:

- *UDP:* unreliable **datagram**
- *TCP:* reliable, **byte stream-oriented**

Application Example:

1. Client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

Socket Programming with UDP

UDP: no "connection" between client & server

- No handshaking before sending data
- Sender explicitly attaches IP destination address and port # to each packet
- Receiver extracts sender IP address and port# from received packet

Note: Transmitted data may be lost or received out-of-order

Socket Programming with TCP

Client contacts server by:

- Creating TCP socket, specifying IP address, port number of server process
- Server must have created socket (door) that welcomes client's contact
- Client TCP establishes connection to server TCP

- When contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - Allows server to talk with multiple clients

Application viewpoint:

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Socket Structure [.1]

```
struct sockaddr
{
        unsigned short int sa_family;      // address family, AF_xxx
        char sa_data[14] ;                 // 14 bytes of protocol address
}
```

- **sa_family** – this remains AF_INET for stream and datagram sockets
- **sa_data** - contains destination address and port number for the socket

Socket Structure [..2]

- Parallel structure to **sockaddr**

```
struct sockaddr_in
{
        short int sin_family;                // Address family (e.g., AF_INET)
        unsigned short int sin_port;         // Port number (e.g., htons (2240)
        struct in_addr sin_addr;             // Internet address
        unsigned char sin_zero[8];           // same size as sockaddr
}
 struct in_addr
                  {     unsigned long s_addr;
                   }
```

- **sin_zero** is just used to pad the structure to the length of a structure **sockaddr** and hence is set to all zeros with the function memset()
- Important – you can cast **sockaddr_in** to a pointer of type **struct sockaddr** and vice versa
- **sin_family** corresponds to sa_family and should be set to “AF_INET”.
- **sin_port** and **sin_addr** must be in NBO

NBO & HBO Conversion Functions

- Two types that can be converted
  - short (two bytes)
  - long (two bytes)

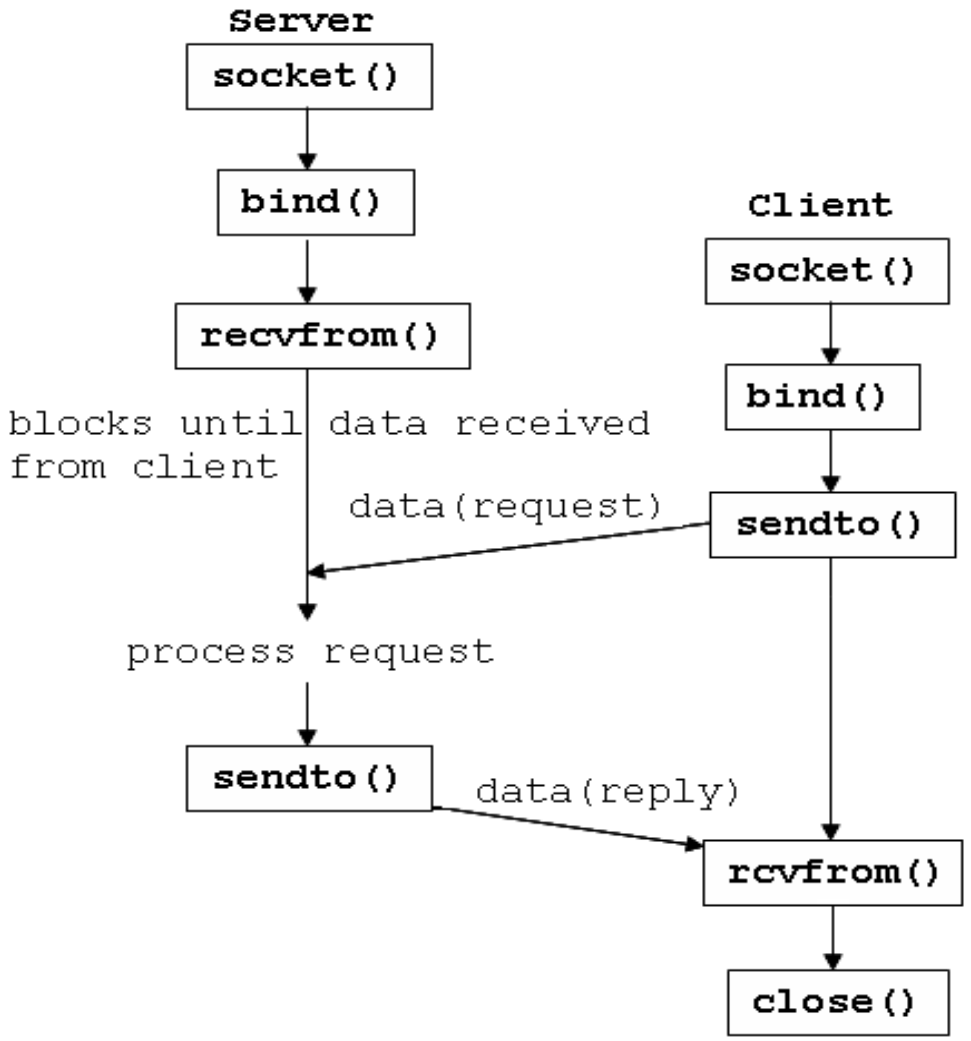
- Primary conversion functions
  - htons() // host to network short
  - htonl() // host to network long
  - ntohs // network to host short
  - ntohl() // network to host long

- **Very Important:** Even if your machine is Big-Endian m/c, but you put your bytes in NBO before putting them on to the network for portability

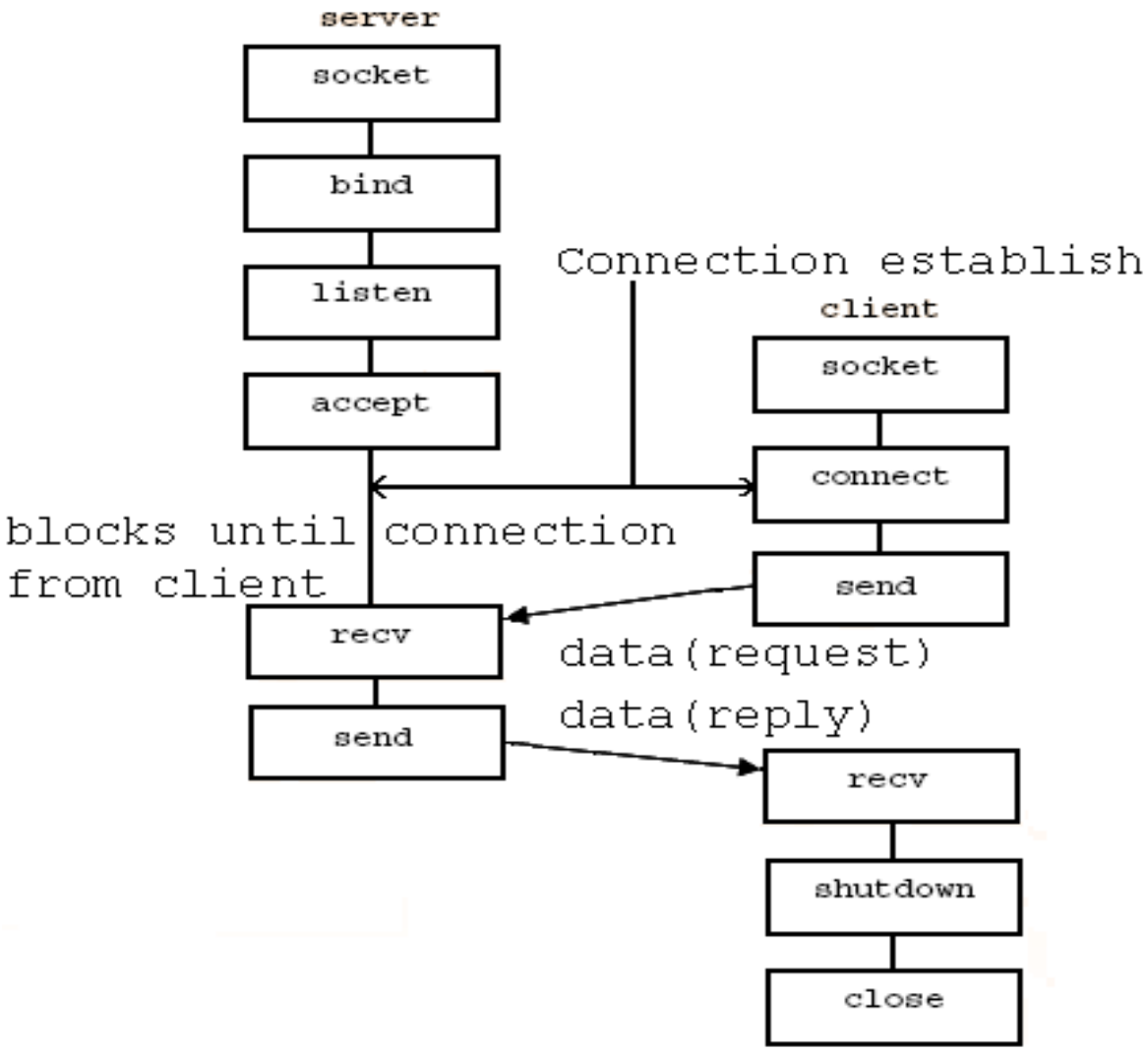
Primary Socket System Calls

- socket() - create a new socket and return its descriptor
- bind() - associate a socket with a port and address
- listen() - establish queue for connection requests
- accept() - accept a connection request
- connect() - initiate a connection to a remote host
- recv() - receive data from a socket descriptor
- send() - send data to a socket descriptor
- close() - “one-way” close of a socket descriptor

Socket System Calls: Connectionless (e.g., UDP)



Socket System Calls: Connection-Oriented (e.g., TCP)



Socket System Calls [.1]

- **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)

Socket System Calls [..2]

- **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()

Socket System Calls [...3]

- **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

Socket System Calls [….4]

- **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*);**

Byte ordering routines


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

u_long htonl(u_long hostlong);      /* host-to-network, long integer */

u_short htons(u_short hostshort);   /* host-to-network, short integer */

u_long ntohl(u_long netlong);       /* network-to-host, long integer */

u_short ntohs(u_short netshort);    /* network-to-host, short integer */
```

Address conversion routines

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

unsigned long inet_addr(char *ptr);
```

converts a char string IP address to its 32-bit network byte-order integer equivalent.

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

Simple TCP Server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_PORT    5888

int main()
 {    int    sockfd, connfd,clilen,n;
      char      buf[256];
      struct sockaddr_in servaddr, cliaddr;

 sockfd = socket( AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0)
      { printf(" Server socket error");
        exit(1);  }
      servaddr.sin_family = AF_INET;
      servaddr.sin_port = htons(SERVER_PORT);
      servaddr.sin_addr.s_addr =
   htonl(INADDR_ANY);

if(bind(sockfd,(struct
   sockaddr*)&servaddr,sizeof(servaddr) <0 )
      { printf("Server Bind Error"); exit(1); }
```

```
 listen(sockfd, 5);

for(; ; )  {
clilen= sizeof(cliaddr);
connfd=accept(sockfd, (struct sockaddr *)
   &cliaddr, &clilen);

if(connfd<0)
  {  printf("Server Accept error \n"); exit(1); }

   printf("Client IP: %s\n",
   inet_ntoa(cliaddr.sin_addr));
   printf("Client Port: %hu\n",
   ntohs(cliaddr.sin_port));

   n = read(connfd, buf,256);
   printf("Server read: \"%s\" [%d chars]\n", buf,
   n);

   write(connfd, "Server Got Message", n);
   close(connfd);
  }
}
```

Simple TCP Client

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_PORT   5888
int main()
{   int   sockfd, clifd,len;
    char     buf[256];
    struct sockaddr_in servaddr;
 sockfd = socket( AF_INET, SOCK_STREAM, 0);
    if (sockid < 0) { printf(" Server socket error");  exit(1); }

    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERVER_PORT);
    servaddr.sin_addr.s_addr = inet_addr("172.24.2.4");

connect(sockfd,(struct sockaddr*)&servaddr, sizeof(servaddr))

  print("Enter Message \n");
   fgets(buf,256,stdin);
   write(sockfd,buf,strlen(buf));

   read(sockfd,buf,256);
  printf("Client Received%s\n",buf);
  Close(sockfd);
 }
```

Simple UDP Server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_PORT    9988
int main()
 {   int   sockfd, clilen;
    char     buf[256];
    struct sockaddr_in servaddr, cliaddr;
sockfd = socket( AF_INET, SOCK_DGRAM, 0);
     servaddr.sin_family = AF_INET;
     servaddr.sin_port = htons(SERVER_PORT);
     servaddr.sin_addr.s_addr =htonl(INADDR_ANY);
if (bind(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr)) <0 )
      { printf("Server Bind Error"); exit(1); }
for(; ; )
{ clilen= sizeof(cliaddr);
 recvfrom(sockfd,buf,256,0,(struct sockaddr*)&cliaddr,&clilen);

 printf("Server Received:%s\n",buf);

sendto(sockfd,"Server Got Message",18, 0,(struct sockaddr*)&cliaddr,clilen);
 }
}
```

Simple UDP Client


```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_PORT   9988
#define SERVER_IPADDR  "172.24.2.4"
int main()
{   int   sockfd,len;
     char     buf[256];
    struct sockaddr_in ,cliaddr,servaddr;

     servaddr.sin_family = AF_INET;
     servaddr.sin_port = htons(SERVER_PORT);
     servaddr.sin_addr.s_addr = inet_addr(SERVER_IPADDR);

sockfd = socket( AF_INET, SOCK_DGRAM, 0);
     cliaddr.sin_family = AF_INET;
     cliaddr.sin_port = htons(0);
     cliaddr.sin_addr.s_addr =htonl(INADDR_ANY);
 bind(sockfd,(struct sockaddr*)&cliaddr,sizeof(cliaddr));
 printf("Enter Message\n");   fgets(buf,255,stdin);
  len= sizeof(server);

 sendto(sockfd,buf,strlen(buf), 0,(struct sockaddr*)&seraddr,len);

 recvfrom(sockfd,buf,256,0,NULL,NULL);
    printf("Clinet Received: %s \n",buf);
 close(sockfd);
}
```

Not mandatory

Q.4 Answer the following questions based on socket programming concepts: [3+2=5M]

a) What is the purpose of **bind()** system call? Describe the requirement of binding sockets with respect to client and server for both TCP and UDP sockets.

Sol: The bind() system call binds a particular port to the socket. In other words, bind() assigns a name to the socket.

The bind() is useful in following manner:

i) Servers register their well-known address with the system. It tells the system "this is my address and any messages received for this address are to be given to me." Both connection-oriented and connectionless servers need to do this before accepting client requests.

ii) A client can register a specific address for itself. (Client side bind() is not mandatory for both TCP and UDP. The operating system binds the requested socket to a random local port when the response received from the server.)

iii) A connectionless client needs to assure that the system assigns it some unique address, so that the other end (the server) has a valid return address to send its responses to. This corresponds to making certain an envelope has a valid return address, if we expect to get a reply from the person we sent the letter to.

< 1 mark for bind() definition, 2 for its requirement>

b) What is the correct order in which a server process must invoke the system calls **accept()**, **bind()**, **listen()**, and **recv()** according to UNIX socket API. Which one of these is/are a blocking call(s)?

Sol: The correct order is **bind()**, **listen()**, **accept()**, and **recv()** <1 mark for correct order, no partial marks>

For a blocking system call, the caller can't do anything until the system call returns. So based on this definition **accept()** and **recv()** are blocking calls. <0.5 + 0.5 for **accept()** + **recv()**, no marks if your answer includes **bind()**>