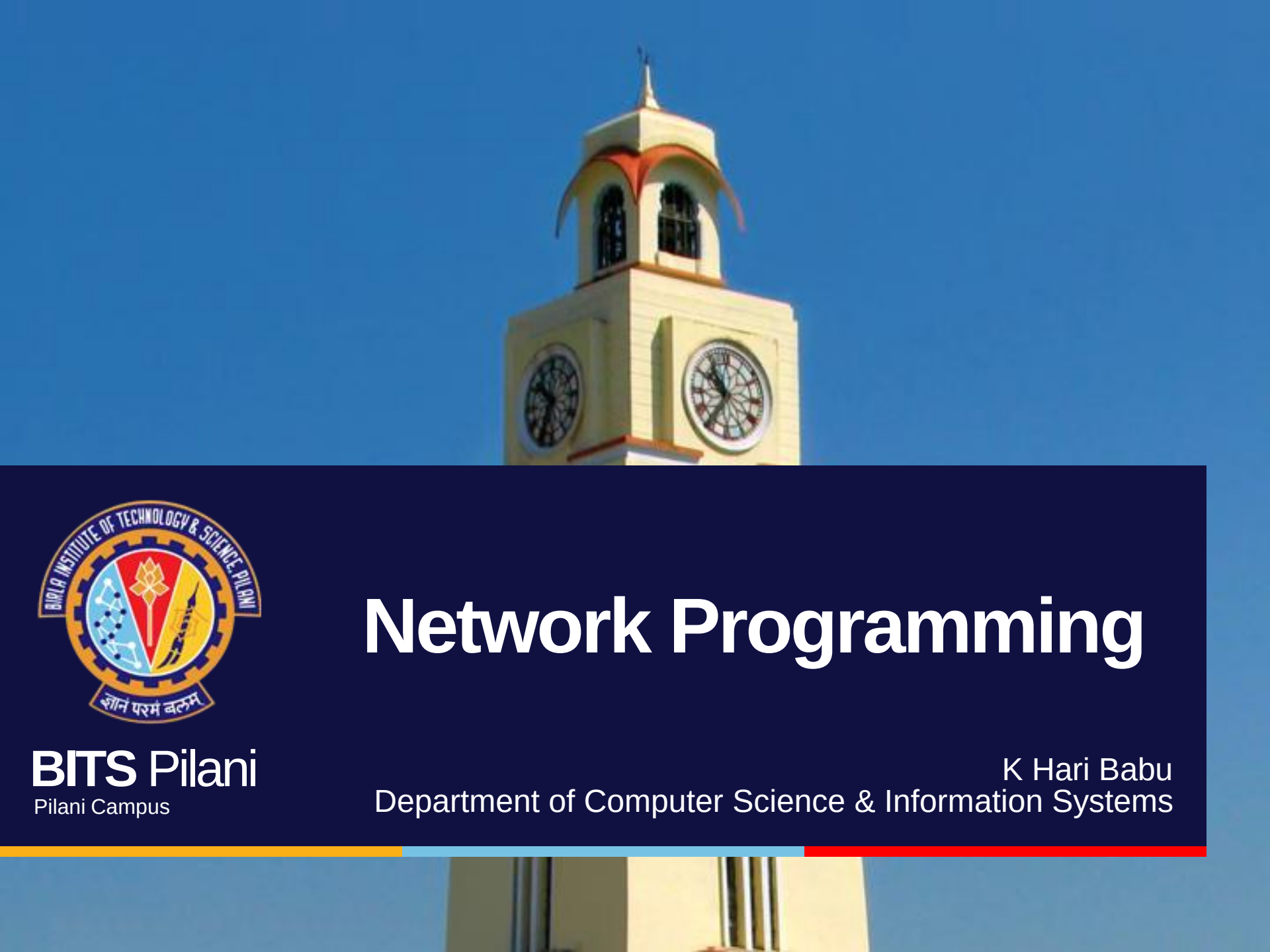


Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

Outline

- Sockets
- TCP Client Server

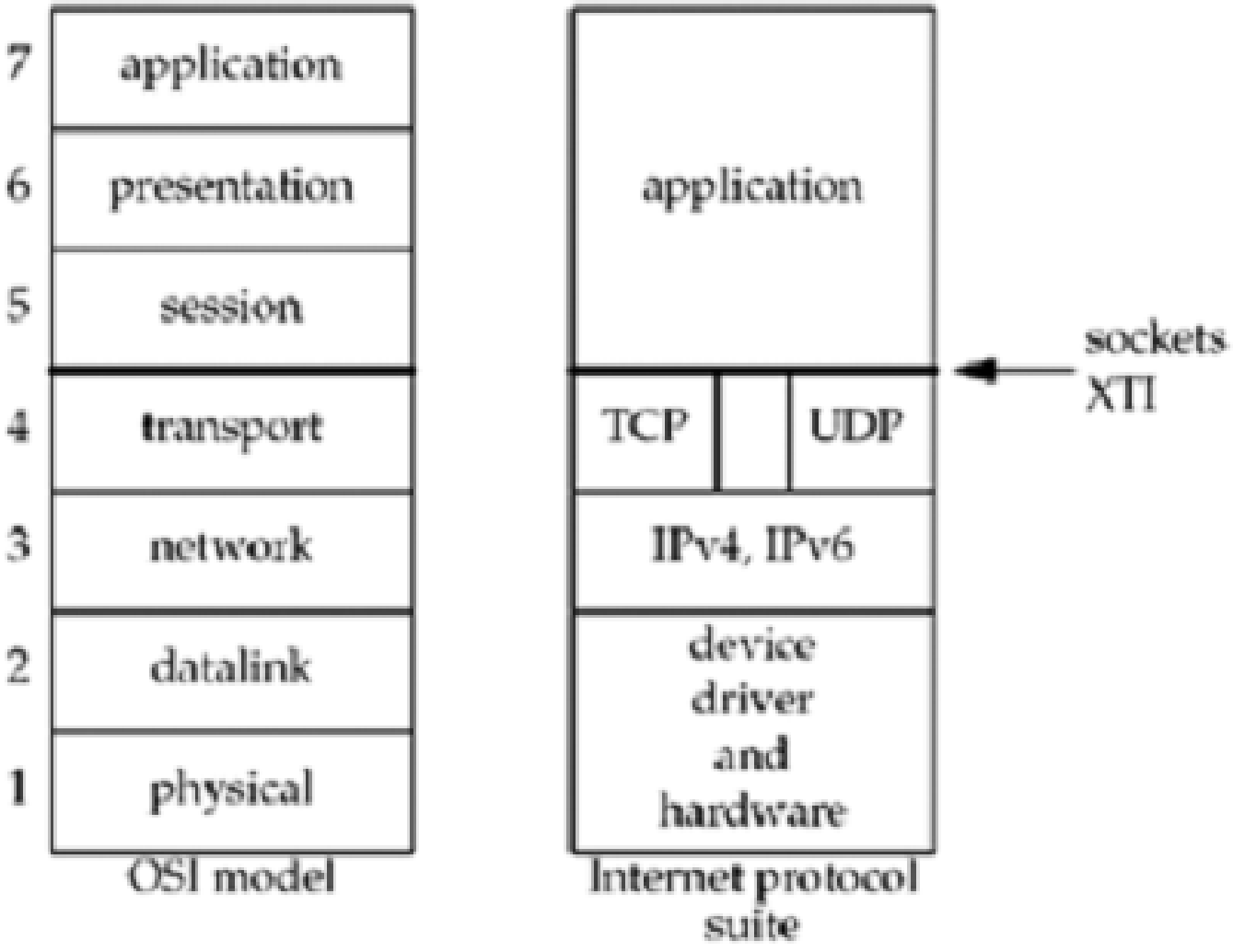
innovate achieve lead

BITS Pilani

Pilani Campus

Sockets

OSI & Internet protocol suite



TCP/IP & Sockets API

- TCP/IP does not include an API definition.
- There are a variety of APIs for use with TCP/IP:
  - Sockets
  - TLI, XTI
  - Winsock
  - MacTCP
- API should have the following functionalities
  - Specify local and remote communication endpoints
  - Initiate a connection
  - Wait for incoming connection
  - Send and receive data
  - Terminate a connection gracefully
  - Error handling

Berkeley Sockets API

- First appeared in 42. BSD in 1983.
  - Supported on every UNIX variant.
  - WinSock API also follows socket API.
- Generic API:
  - support for multiple communication domains which differ in addressing methods.

Domain	Communication performed	Communication between applications	Address format	Address structure
AF_UNIX	within kernel	on same host	pathname	*sockaddr_un*
AF_INET	via IPv4	on hosts connected via an IPv4 network	32-bit IPv4 address + 16-bit port number	*sockaddr_in*
AF_INET6	via IPv6	on hosts connected via an IPv6 network	128-bit IPv6 address + 16-bit port number	*sockaddr_in6*

- Uses existing I/O programming interface as much as possible.
  - Socket API is similar to file I/O

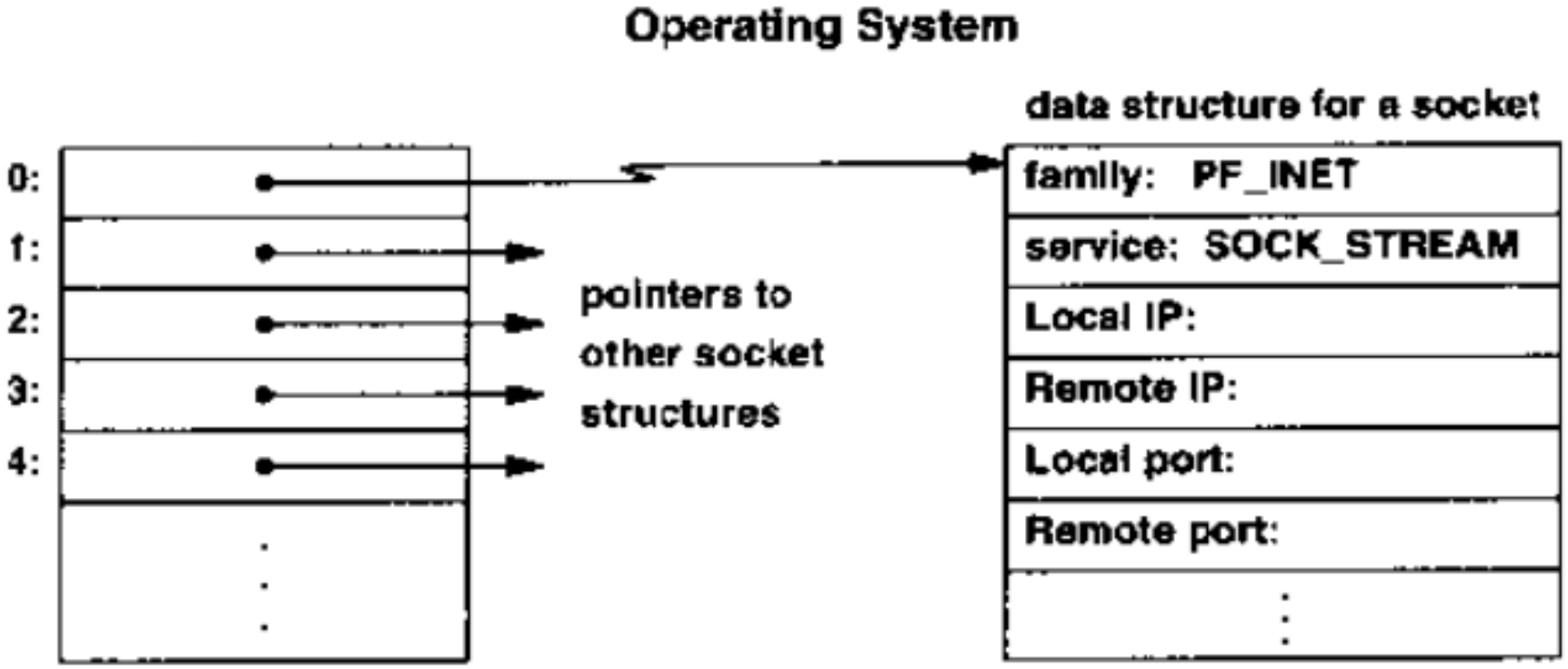
Socket

- A socket is an abstract representation of a communication endpoint.
- Sockets work with Unix I/O services just like files, pipes & FIFOs.
  - read(), write() and close() sys calls work on sockets also
- Sockets have special needs over files:
  - establishing a connection.
  - specifying communication endpoint addresses.

Socket Types

- Two types of sockets
  - Stream Sockets
    - Reliable
    - Bidirectional
    - Byte-stream
    - Connection-oriented
    - Stream sockets operate in connected pairs. (local end point, remote end point)
    - Internet Domain: TCP Socket
  - Datagram Sockets
    - Message boundaries are preserved.
    - No reliability support: out of order, duplicates, or lost datagrams
    - Connectionless socket: no need to be connected to another socket.
    - Internet Domain: UDP Socket

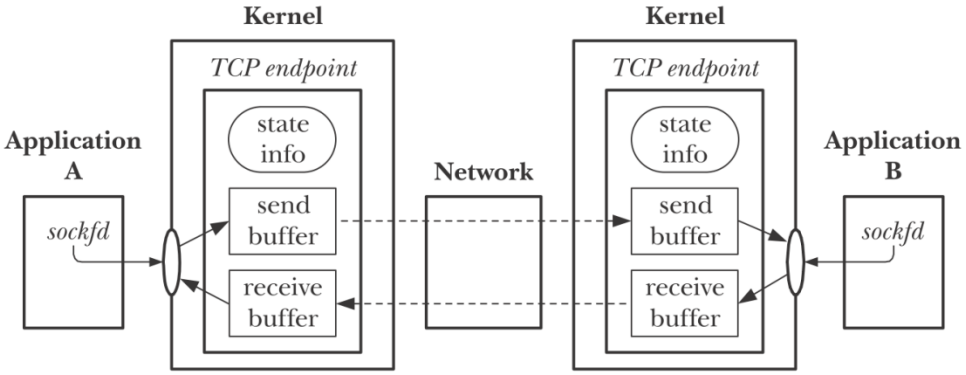
Socket Descriptor Data Structure

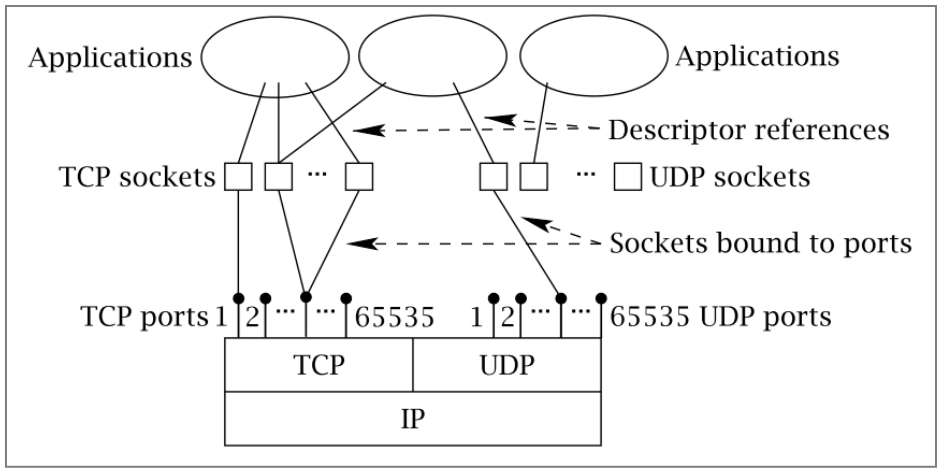


Internet Domain & Sockets

innovate achieve lead

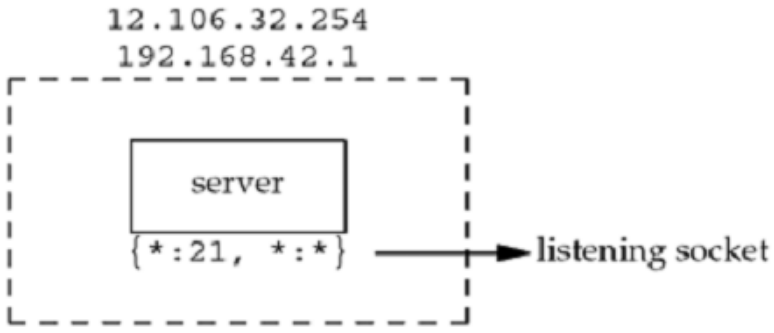
- Endpoint is identified by ip address and port number.
  - A socket needs to be bound to endpoints local and remote.





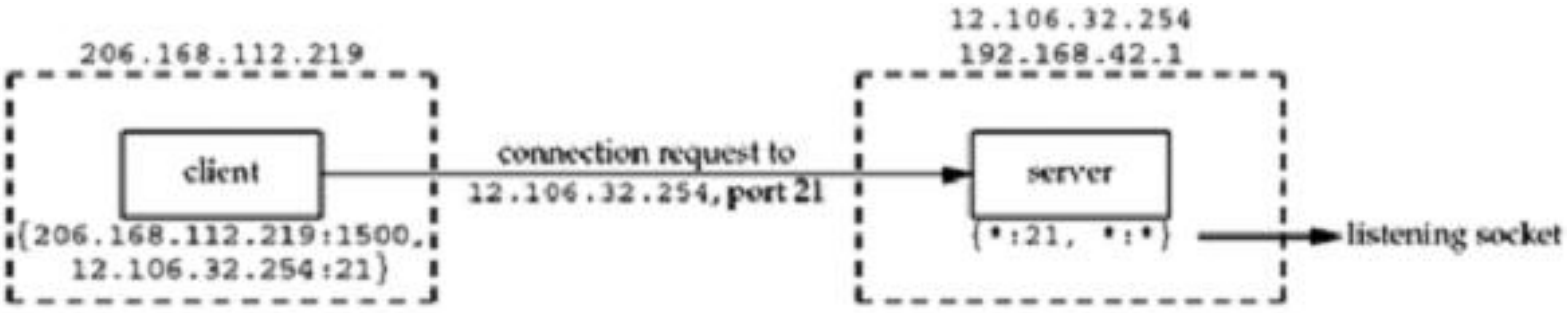
Socket Pair

- The socket pair for a TCP connection is the four-tuple that defines the two endpoints of the connection:
  - the local IP address, local port, foreign IP address, and foreign port.
- A socket pair uniquely identifies every TCP connection on a network.
- We can extend the concept of a socket pair to UDP, even though UDP is connectionless.
- TCP connection for a ftp server:
  - Server has two IP interfaces. * indicates any ip address/any port.

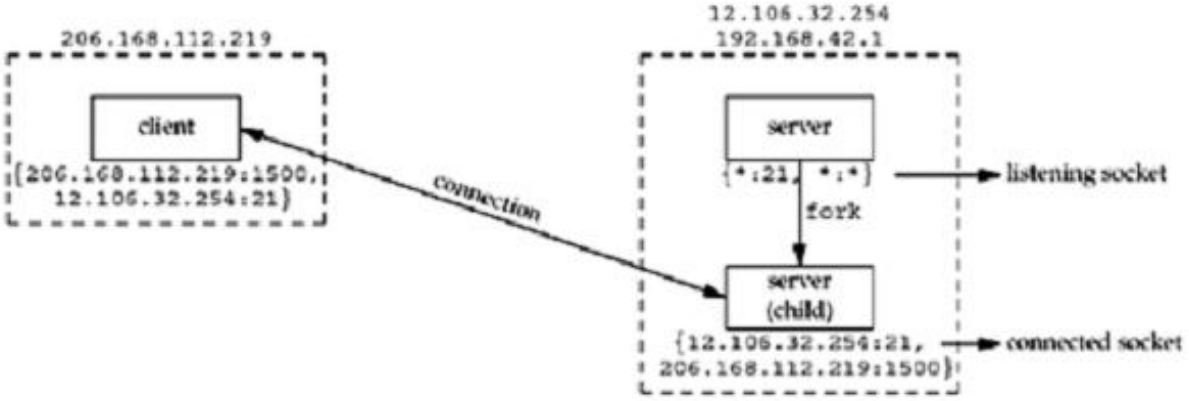


TCP Connection

- A client 206.168.112.219 connects to 12.106.32.254 at port 21.



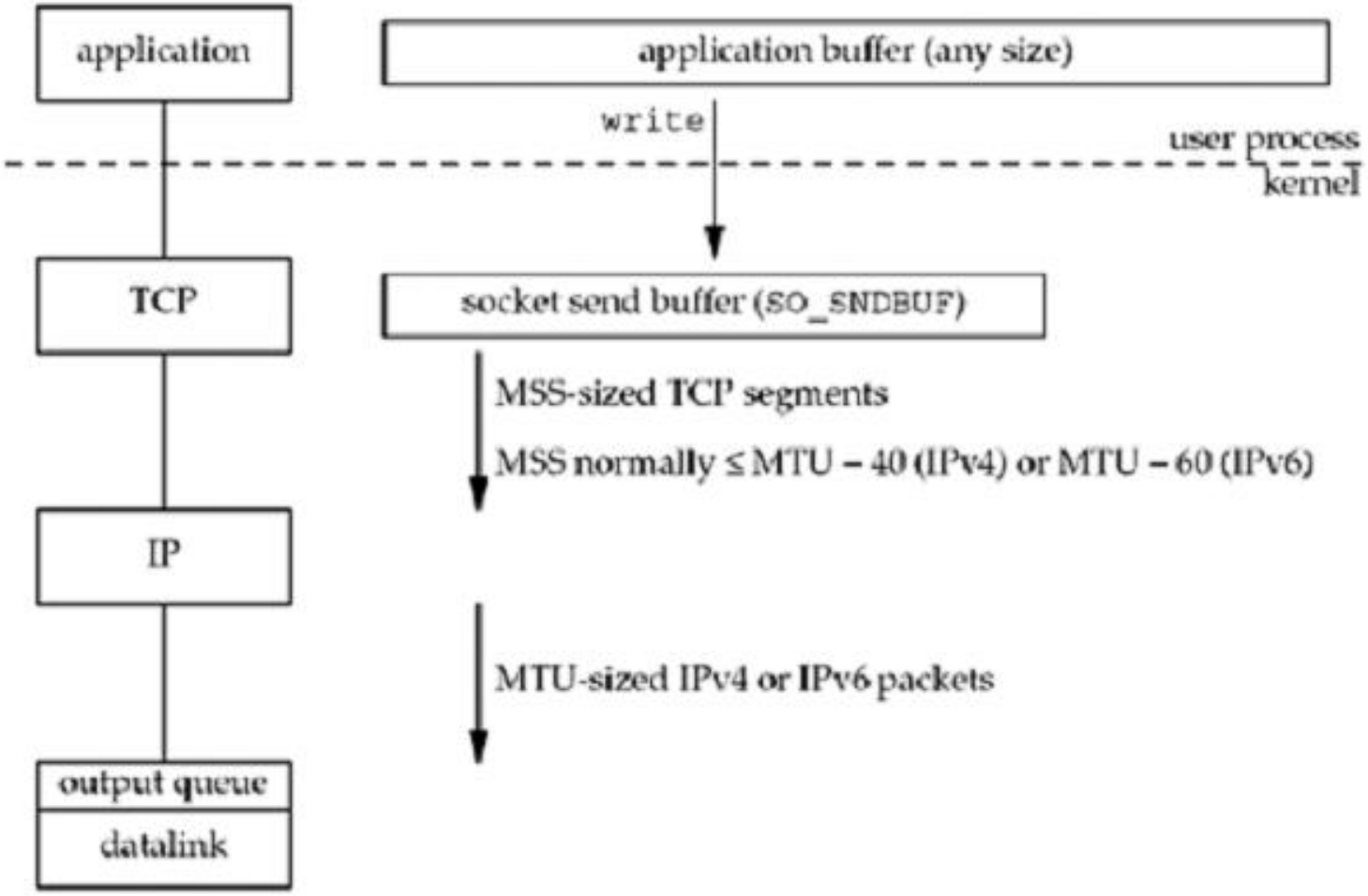
- Server creates a child to handle new cleint.
  - See the new TCP connection pair on server.



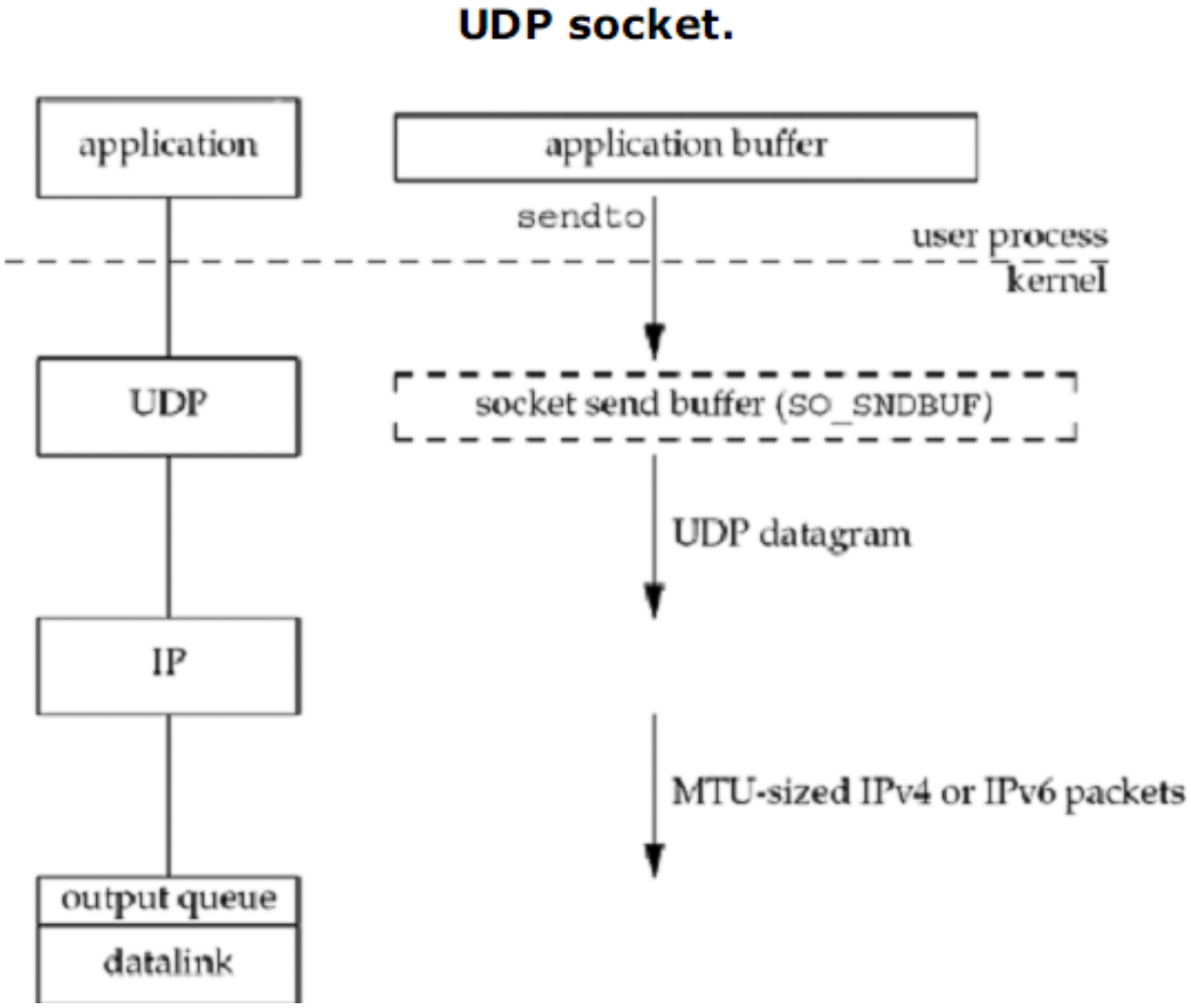
TCP Connection

- Client opens one more connection to the same server. What makes it a different TCP connection?

Writing to TCP Socket



Writing to UDP Socket



Creating a Socket


```
1  #include <sys/socket.h>
2  int socket(int  domain , int  type , int  protocol );
3  //Returns file descriptor on success, or -1 on error
```

- Domain argument specifies the communication domain.
- Type argument specifies the socket type
- Protocol is usually 0. Its value is automatically from first two arguments.
  - But for raw sockets, protocol value can be specified.
- The **`socket()`** system call returns a socket descriptor (small integer) or -1 on error.
- **`socket()`** allocates resources needed for a communication endpoint - but it does not deal with endpoint addressing.

Specifying an Endpoint Address

- Remember that the sockets API is generic.
- There must be a generic way to specify endpoint addresses.
- Internet Domain requires an IP address and a port number for each endpoint address.
- Other domains (families) may use other schemes.
- Data types for specifying address structure

```
1   sa_family_t //address family
2   socklen_t   //length of struct
3   in_addr_t   //IPv4 address
4   in_port_t   //IP port number
```

Generic Socket Address Structure

- General socket address structure acts as template.
- All sys calls take this as the input.

```
1  struct sockaddr {
2      sa_family_t sa_family;  /* Address family (AF_* constant) */
3      char        sa_data[14];/* Socket address (size varies
4                                according to socket domain) */
5  };
```

- For IPv4 domain, socket address structure is

```
1  struct sockaddr_in {
2      sa_family_t     sin_family;
3      in_port_t       sin_port;
4      struct in_addr  sin_addr;
5      char            sin_zero[8];
6  };
```

```
1  sa_family_t //address family
2  socklen_t   //length of struct
3  in_addr_t   //IPv4 address
4  in_port_t   //IP port number
```

```
2  struct in_addr {
3      in_addr_t       s_addr;
4  };
5
```

Specifying End Point Address

- Bind call binds an address to a socket

```
1  #include <sys/socket.h>
2  int bind(int  sockfd , const struct sockaddr * addr ,
3  socklen_t  addrlen );
4  //Returns 0 on success, or -1 on error
```

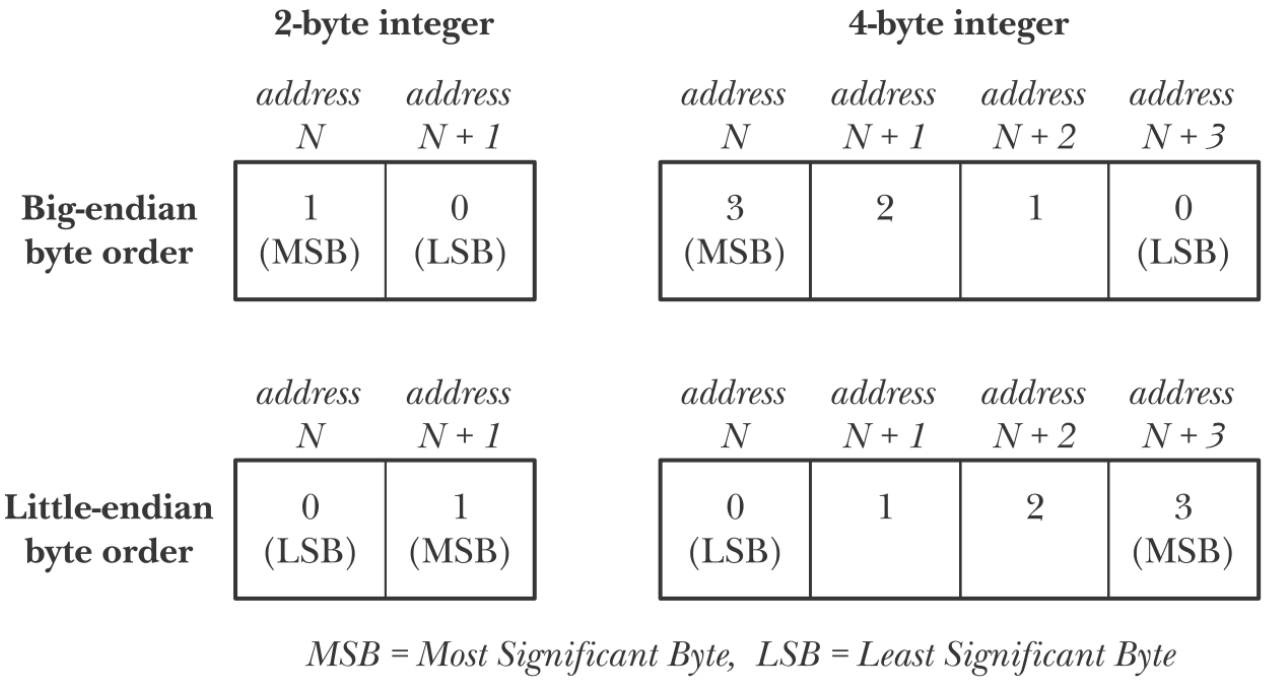
- Bind is used to bind the socket to a local end point.
  - *addr* is the pointer to an address structure that contains the endpoint address.

process specifies		result
IP address	**port**	
Wildcard	0	kernel chooses IP addr and port
Wild card	nonzero	kernel chooses IP, process specifies port
local IP addr	0	process specifies IP, kernel chooses port
local IP addr	nonzero	process specifies IP and port

Wildcard specified as INADDR_ANY

Byte Order

- Different hardware architectures in the network. Passing integers across the network needs to be in standard format: Network Byte Order (NBO): bigendian byte order.



Network Byte Order Functions

innovate achieve lead

```
1  #include <arpa/inet.h>
2  uint16_t htons(uint16_t  host_uint16 );
3  //Returns host_uint16 converted to network byte order
4  uint32_t htonl(uint32_t  host_uint32 );
5  //Returns host_uint32 converted to network byte order
6  /uint16_t ntohs(uint16_t  net_uint16 );
7  //Returns net_uint16 converted to host byte order
8  uint32_t ntohl(uint32_t  net_uint32 );
9  //Returns net_uint32 converted to host byte order
```

'**h**' : host byte order

'**n**' : network byte order

'**s**' : short (16bit)

'**l**' : long (32bit)

bind() Example

innovate achieve lead

```
1  int mysock,err;
2  struct sockaddr_in myaddr;
3
4  mysock = socket(PF_INET,SOCK_STREAM,0);
5  myaddr.sin_family = AF_INET;
6  myaddr.sin_port = htons( portnum );
7  myaddr.sin_addr = htonl( ipaddress);
8
9  err=bind(mysock, (sockaddr *) &myaddr, sizeof(myaddr));
```

- Note the usage of byte conversion.
- Nit the casting internet domain address to generic address structure.
- There are a number of uses for **`bind()`**:
  - Server would like to bind to a well known address (port number).
  - Client can bind to a specific port.
  - Client can ask the O.S. to assign *any available* port number.

IPv4 Address Conversion

- The inet_pton() and inet_ntop() functions allow conversion of both IPv4 and IPv6 addresses between binary form and dotted-decimal or hex-string notation.
- The p in the names of these functions stands for “presentation,” and the n stands for “network.” The presentation form is a human-readable string.

```
1  #include <arpa/inet.h>
2  int inet_pton(int  domain , const char * src_str , void * addrptr );
3  //Returns 1 on successful conversion, 0 if src_str is not in
4  //presentation format, or -1 on error
5  const char *inet_ntop(int  domain , const void * addrptr ,
6  char * dst_str , size_t  len );
7  //Returns pointer to dst_str on success, or  NULL  on error
```

innovate achieve lead

BITS Pilani

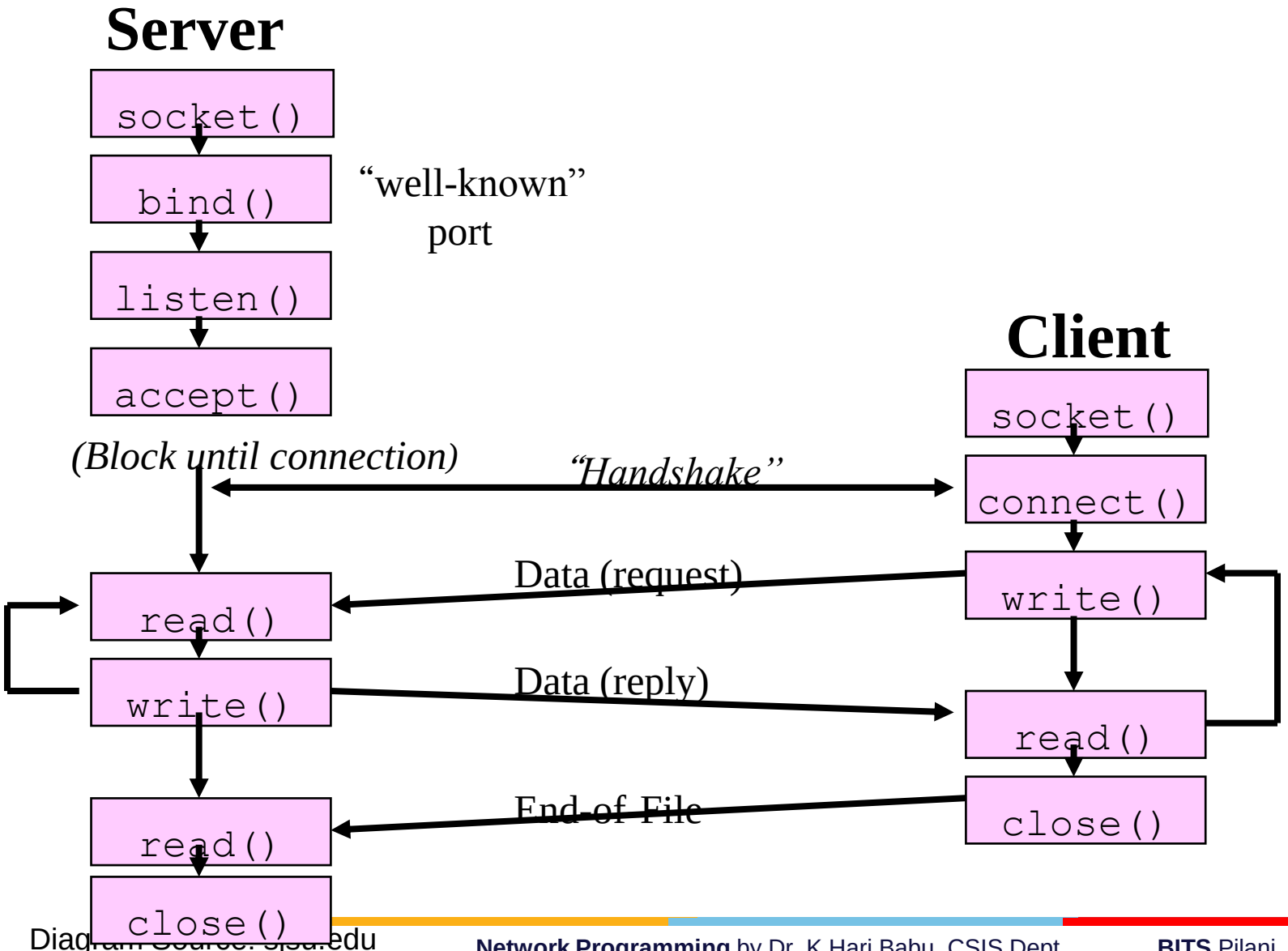
Pilani Campus

Client Server Programming

Active & Passive Sockets

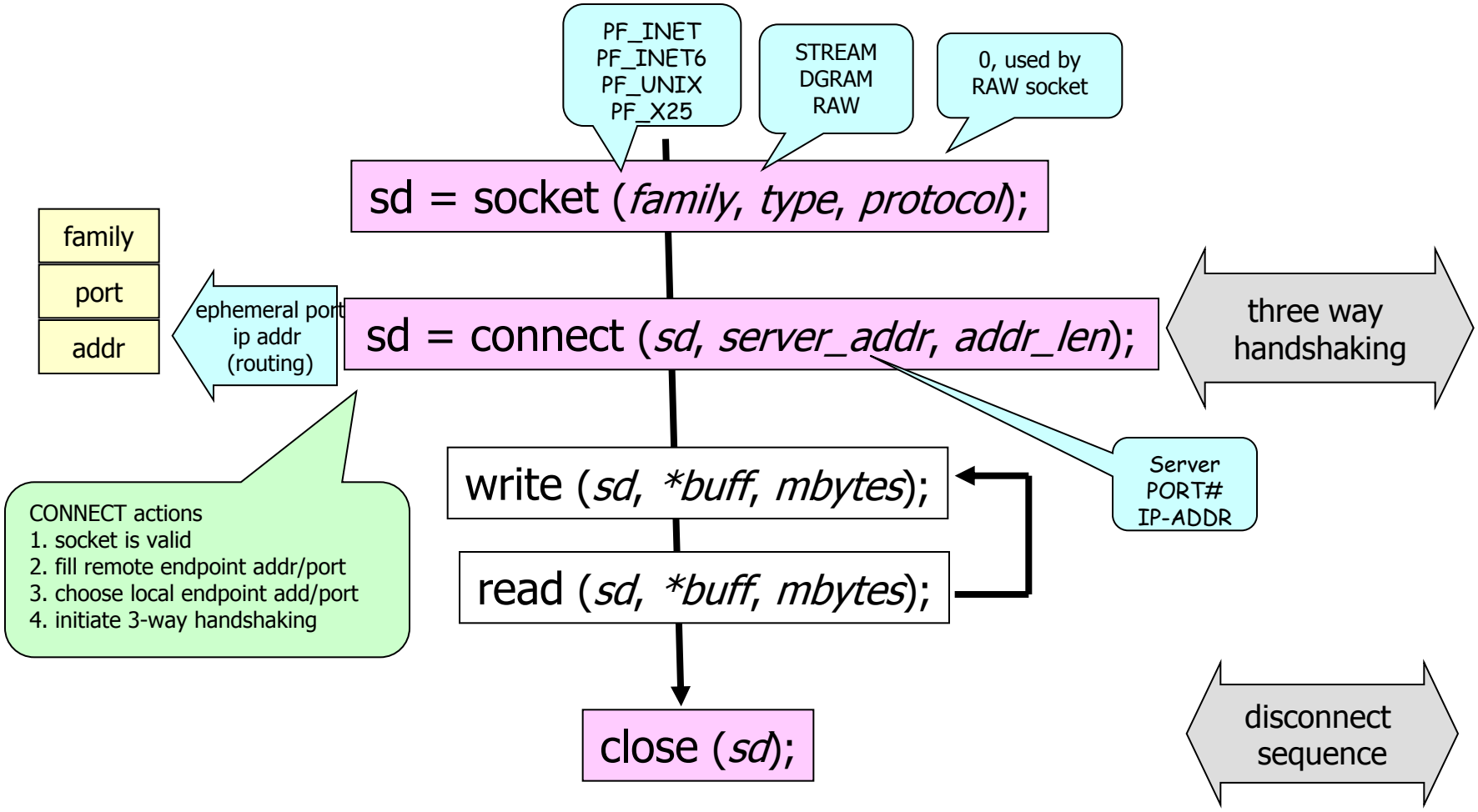
- By default, a socket that has been created using socket() is active.
  - An active socket can be used in a connect() call to establish a connection to a passive socket.
  - This is referred to as performing an active open.
- A passive socket (also called a listening socket) is one that has been marked to allow incoming connections by calling listen().
  - Accepting an incoming connection is referred to as performing a passive open.
- client:
  - which does active socket open
- Server:
  - which does passive socket open.

TCP Client Server



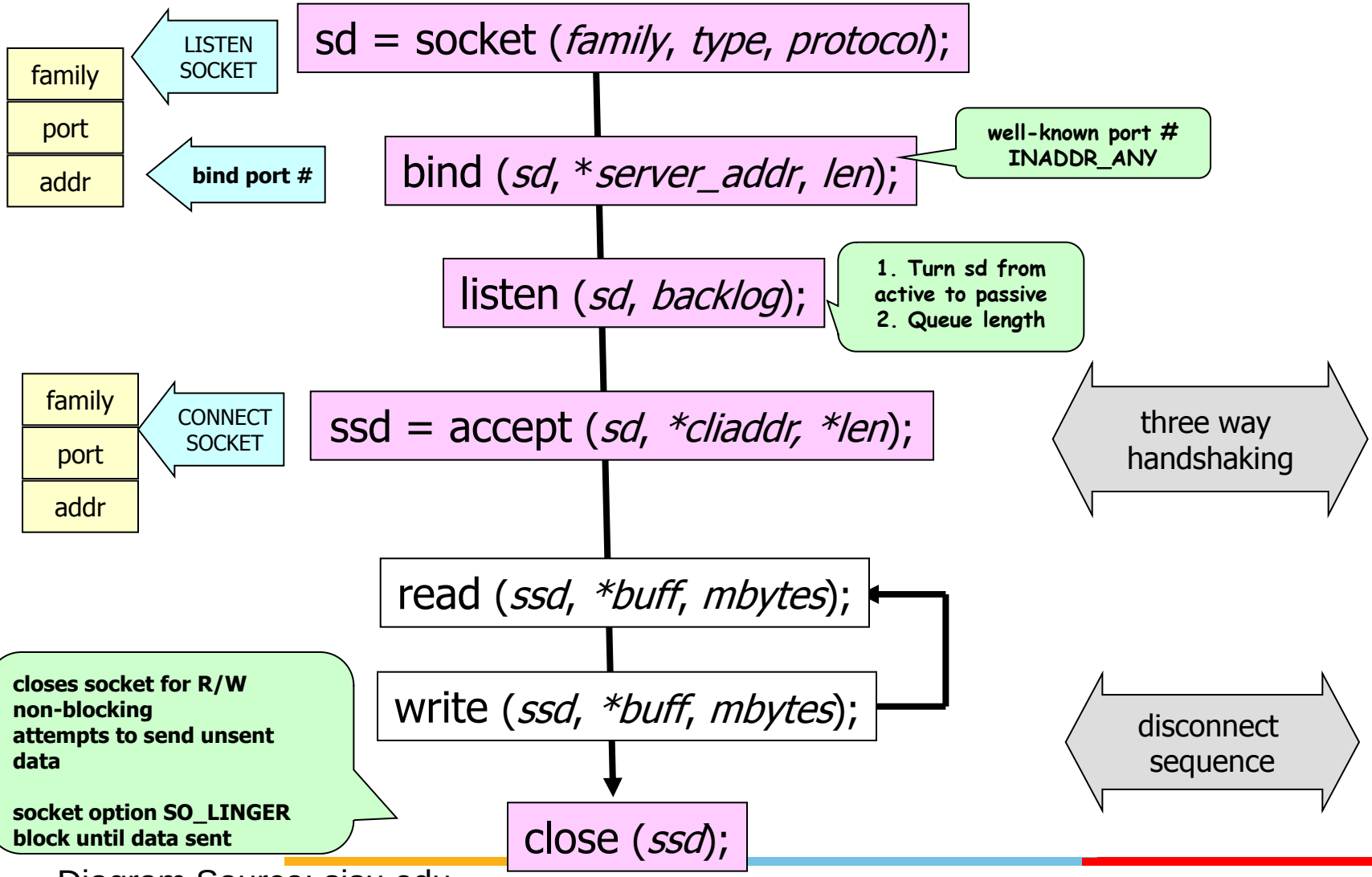
- Diagram Source: sjsu.edu

TCP Client



- Diagram Source: sjsu.edu

TCP Server



- Diagram Source: sjsu.edu

connect() - connect to server

- The connect() system call connects the active socket referred to by the file descriptor sockfd to the listening socket whose address is specified by addr and addrlen.

```
1  #include <sys/socket.h>
2  int connect(int  sockfd , const struct sockaddr * addr,
3  socklen_t  addrlen );
4  //Returns 0 on success, or -1 on error
```

  - *sockfd* is socket descriptor from `socket()`
  - *servaddr* is a pointer to a structure with:
    - *port number* and *IP address*
    - must be specified (unlike `bind()`)
  - *addrlen* is length of structure
  - client doesn't need `bind()`
    - OS will pick ephemeral port
  - returns socket descriptor if ok, -1 on error

connect() - connect to server

- Errors
  - If the server's TCP response to client TCP's SYN segment is RST, then there is no process is waiting for incoming connections.
    - Hard error
  - Three conditions that generate RST are
    - when a SYN arrives for a port that has no listening server
    - when TCP wants to abort an existing connection
    - when TCP receives a segment for a connection that does not exist
  - If client's SYN request elicits ICMP "destination unreachable" message, kernal saves the message but keeps on sending the SYN segment.
    - Soft error

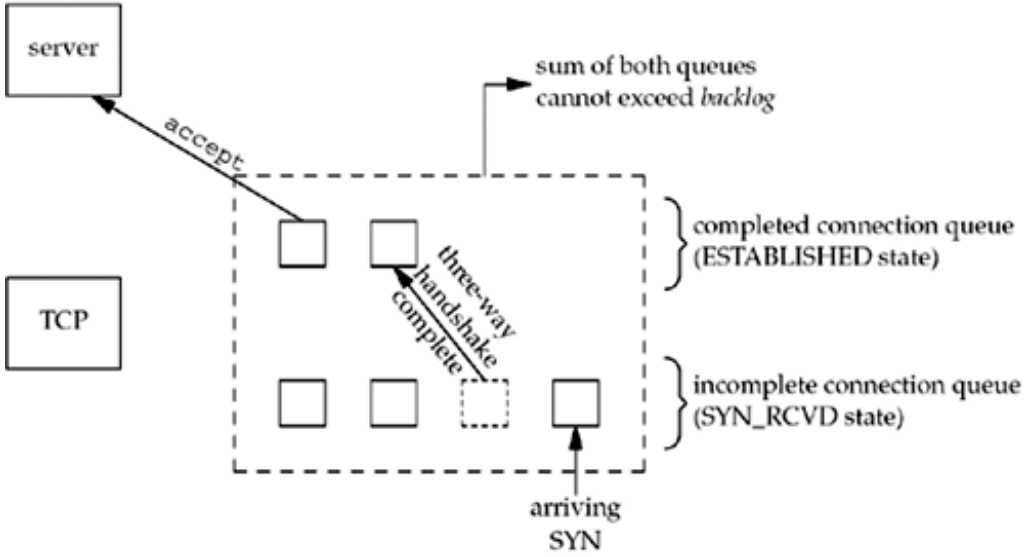
listen() - change socket state to passive

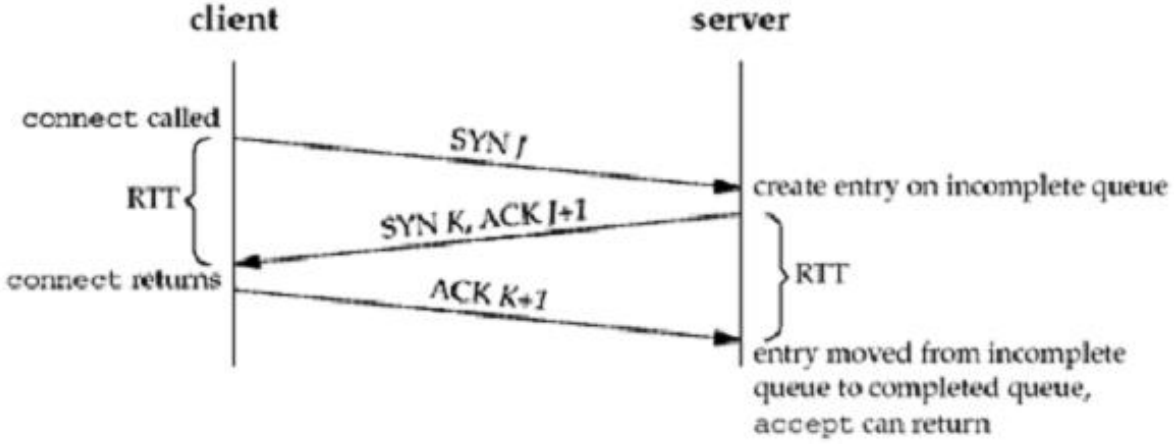
- The listen() system call marks the stream socket referred to by the file descriptor sockfd as passive.
  - The socket will subsequently be used to accept connections from other (active) sockets.

```
1  #include <sys/socket.h>
2  int listen(int  sockfd , int  backlog );
3  //Returns 0 on success, or -1 on error
```

- *sockfd* is socket descriptor from `socket()`
- *backlog* is maximum number of connections that the server should queue for this socket
  - historically 5
  - rarely above 15 on a even moderate Web server!

listen()


server
accept
sum of both queues
cannot exceed backlog
TCP
three-way handshake complete
completed connection queue
(ESTABLISHED state)
incomplete connection queue
(SYN_RCVD state)
arriving SYN


client
server
connect called
SYN J
RTT
create entry on incomplete queue
SYN K, ACK J+1
RTT
connect returns
ACK K+1
entry moved from incomplete
queue to completed queue,
accept can return

accept() - return next completed connection

- The accept() system call accepts an incoming connection on the listening stream socket referred to by the file descriptor sockfd.
  - If there are no pending connections when accept() is called, the call blocks until a connection request arrives.

```
1  #include <sys/socket.h>
2  int accept(int  sockfd , struct sockaddr * addr,
3  socklen_t * addrlen );
4  //Returns file descriptor on success, or -1 on error
```

- *sockfd* is socket descriptor from `socket()`
- *cliaddr* and *addrlen* return protocol address from client
- returns brand new descriptor, created by OS
  - if used with `fork()`, can create concurrent server

close() - close socket fd

- *sockfd* is socket descriptor from `socket()`
- closes socket for reading/writing
  - returns (doesn't block)
  - attempts to send any unsent data
  - socket option SO_LINGER
    - block until data sent
    - or discard any remaining data
  - Returns -1 if error

Descriptor Reference Counts

- For every socket a reference count is maintained, as to how many processes are accessing that socket
- When close() is called on socket descriptor reference count is decreased by 1
- When close() is called on socket descriptor, TCP 4 packet termination sequence will be initiated only if the reference count goes to zero.

getsockname() and getpeername() Functions

- getsockname return the local endpoint address associated with a socket
- getpeername return the foreign protocol address associated with a socket

```
1  #include <sys/socket.h>
2   int getsockname(int sockfd, struct sockaddr *localaddr,
3   socklen_t *addrlen);
4   int getpeername(int sockfd, struct sockaddr *peeraddr,
5   socklen_t *addrlen);
```

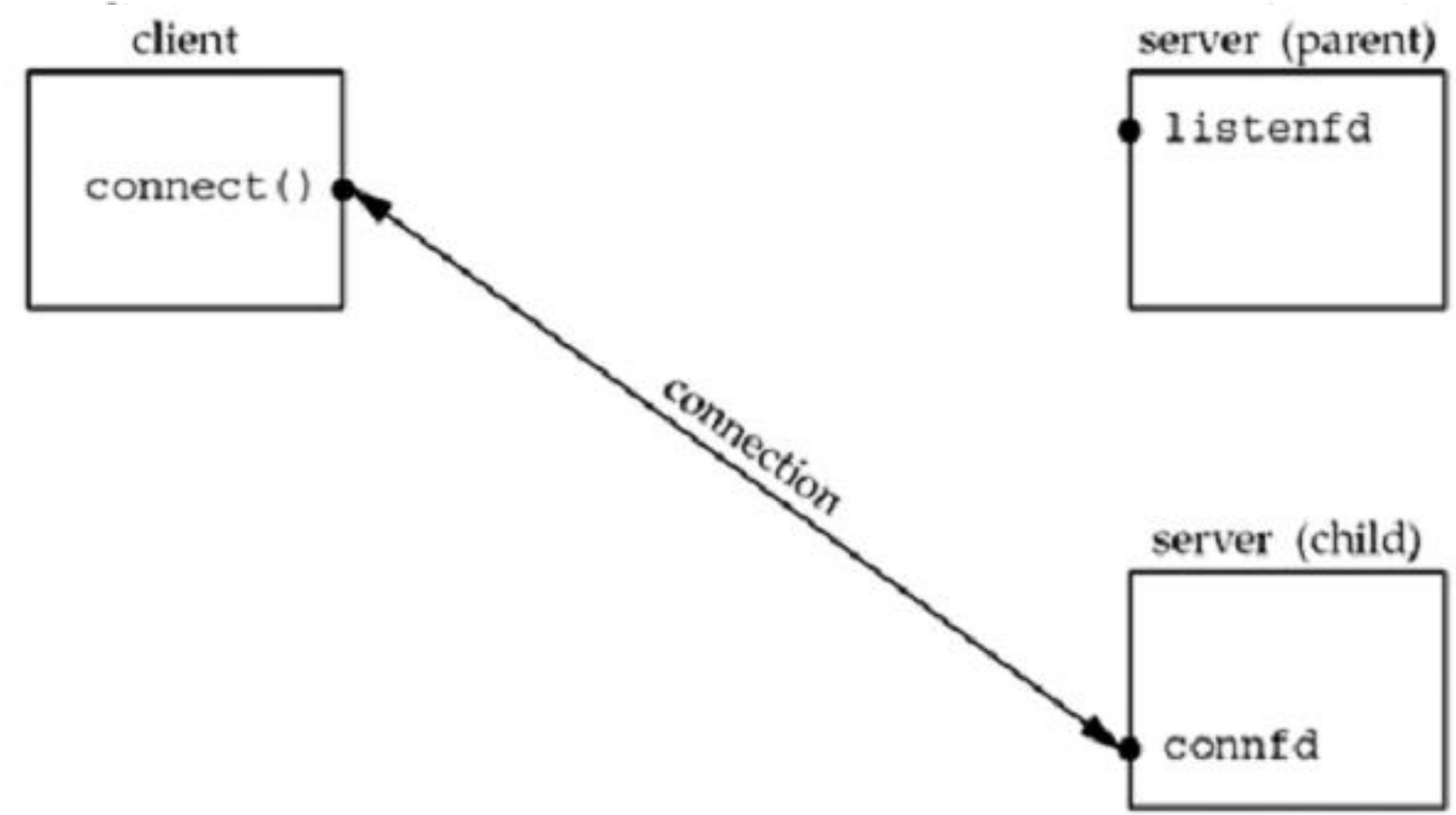
TCP Echo Client

- TCP client and server using echo protocol

```
 1  int main(int argc, char **argv)
 2  {
 3   int     sockfd;
 4   struct sockaddr_in servaddr;
 5   if (argc != 2)
 6      err_quit("usage: tcpcli <IPaddress>");
 7   sockfd = Socket(PF_INET, SOCK_STREAM, 0);
 8   bzero(&servaddr, sizeof(servaddr));
 9   servaddr.sin_family = AF_INET;
10   servaddr.sin_port = htons(SERV_PORT);
11   inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
12   connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
13   str_cli(stdin, sockfd);
```

```
17  void str_cli(FILE *fp, int sockfd)
18  {
19      char sendline[MAXLINE], recvline[MAXLINE];
20      while (Fgets(sendline, MAXLINE, fp) != NULL) {
21       write(sockfd, sendline, strlen (sendline));
22       if (wead(sockfd, recvline, MAXLINE) == 0)
23           err_quit("str_cli: server terminated prematurely");
24       wputs(recvline, stdout);
25  }}
```

TCP Concurrent Server



TCP Concurrent Server


```
1  int main(int argc, char **argv)
2  {  int     listenfd, connfd;
3   pid_t   childpid;
4    socklen_t clilen;
5    struct sockaddr_in cliaddr, servaddr;
6   listenfd = Socket (AF_INET, SOCK_STREAM, 0);
7   bzero(&servaddr, sizeof(servaddr));
8   servaddr.sin_family = AF_INET;
9   servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
10  servaddr.sin_port = htons (SERV_PORT);
11  bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
12  listen(listenfd, LISTENQ);
13   for ( ; ; )  {
14   clilen = sizeof(cliaddr);
15   connfd = accept(listenfd, (struct sokaddr *) &cliaddr, &clilen);
16   if ( (childpid = fork()) == 0) { /* child process */
17       close(listenfd);    /* close listening socket */
18   str_echo(connfd);   /* process the request */
19   exit (0);
20   }
21  close(connfd);          /* parent closes connected socket */
22  }
23  }
```

str_echo function

innovate achieve lead

```
1   void  str_echo(int sockfd)
2     {
3         ssize_t n;
4         char    buf[MAXLINE];
5       again:
6         while ( (n = read(sockfd, buf, MAXLINE)) > 0)
7             Write(sockfd, buf, n);
8         if (n < 0 && errno == EINTR)
9             goto again;
10        else if (n < 0)
11            err_sys("str_echo: read error");
12    }
```

TCP Concurrent Server

- Handling zombies
  - while ((pid = waitpid(-1, &stat, WNOHANG)) > 0) in SIGCHLD signal handler
- Handling interrupted system calls
  - when writing network programs that catch signals, we must be cognizant of interrupted system calls, and we must handle them
  - Slow system call is any system call that can block forever

Handling interrupted system calls

innovate achieve lead

```
for ( ; ; ) {
clilen = sizeof (cliaddr);
if ( (connfd = accept (listenfd, (SA *)
  &cliaddr, &clilen)) < 0) {
    if (errno == EINTR)
    continue;              /* back to for () */
    else
            err_sys ("accept error");
        }
```

- Another option:
  - Use sigaction() with SA_RESTART flag.

Termination of Server Process

- FIN is sent to client
- Client tcp sends ACK to server
- What if client application doesn't take not of it, and sends data to server?

SIGPIPE Signal

- When a process writes to a socket that has received an RST, the SIGPIPE signal is sent to the process.
- The default action of this signal is to terminate the process, so the process must catch the signal to avoid being involuntarily terminated.

Crashing of Server Host

- Nothing is sent to client
- Client will try to reach the host, but will get errors such as ETIMEDOUT, EHOSTUNREACH, ENETWORKUNREACH

Crashing and Rebooting of Server Host

- When client sends packets, server will respond with RST

Shutdown of Server Host

- Init sends SIGTERM to all processes
- Then sends SIG KILL to all processes
- Fin is sent to the client

Exercise

innovate achieve lead

- Write a TCP client and server that fulfills the following requirements.
- Server.c:
  - server should take port number on command-line and listen on that port.
  - server should create a child to handle a new client.
  - When a client sends a command such as 'ps', server should execute the command and send the output to the client.
  - it should take care of zombies.
- Client.c:
  - client takes ip address and port number of the server on command-line.
  - client sets up a connection to the server.
  - client takes a command from the user and sends it to the server.
  - client waits for the reply and prints the reply on the standard output.

```
1  if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
2  printf("socket() failed");
3  memset(&echoServAddr, 0, sizeof(echoServAddr));
4  echoServAddr.sin_family = AF_INET;
5  echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);
6  echoServAddr.sin_port = htons(echoServPort);
7  if (bind
8  (servSock, (struct sockaddr *) &echoServAddr,
9   sizeof(echoServAddr)) < 0)
10 printf("bind() failed");
11 if (listen(servSock, MAXPENDING) < 0)
12 printf("listen() failed");
```

```
14      for (;;) {
15      clntLen = sizeof(echoClntAddr);
16      if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr,
17                      &clntLen)) < 0)//2M
18      printf ("accept() failed");
19      printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));
20      ret=fork(); //3M
21          if(ret==0){
22              close(listenfd);
23              read(clntSock, buff, size);
24              int p[2];
25              pipe(p);
26              ret=fork();
27              if(ret==0)
28              {   close(1);
29                  dup(p[1]);
30                  execv(buff);
31                  }
32              read(p[0],buff,size);
33              write(clntSock,buff,size);
34              }
35      close(clntSock);
36  }
```

```
1  //Step 1: Set up Address Structure
2  bzero(&Server_Address, sizeof(Server_Address));
3  Server_Address.sin_family = AF_INET;
4  Server_Address.sin_port = htons(port);
5  temp = inet_addr(Address);
6  if (temp != INADDR_NONE){
7      Server_Address.sin_addr.s_addr = temp;
8  }else{
9      printf ("Invalid IP Address.");
10 }
11 //Step 2: Create a Socket
12 mysocket = socket(PF_INET, SOCK_STREAM, 0);
13 if (mysocket == -1) printf ("socket()");
14
15 //Step 3: Connect to Server
16 result = connect(mysocket, (struct sockaddr *) &Server_Address,
17 sizeof(Server_Address) );
18 if (result == -1) printf ("connect()");
19 while(1){
20 scanf("%s", cmd);
21 if(strcmp(cmd,"exit")==0)
22     exit(0);
23 write(mysocket, cmd,(strlen(cmd));
24 read(mysocket,buff,size);
25 printf("%s", buff);}}
```

Acknowledgements

Q&A

innovate achieve lead

BITS Pilani

Pilani Campus

Thank You