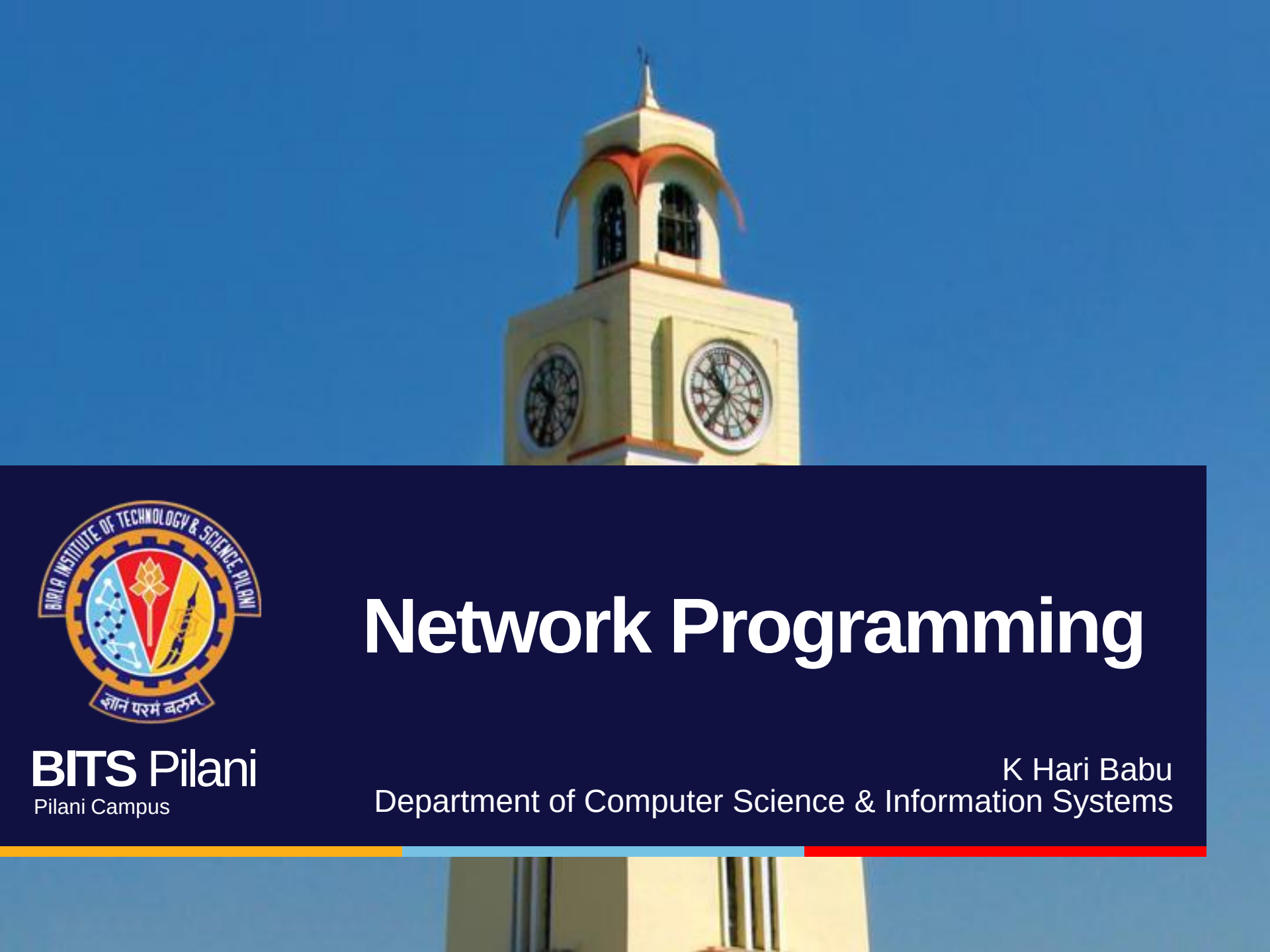


Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

Outline

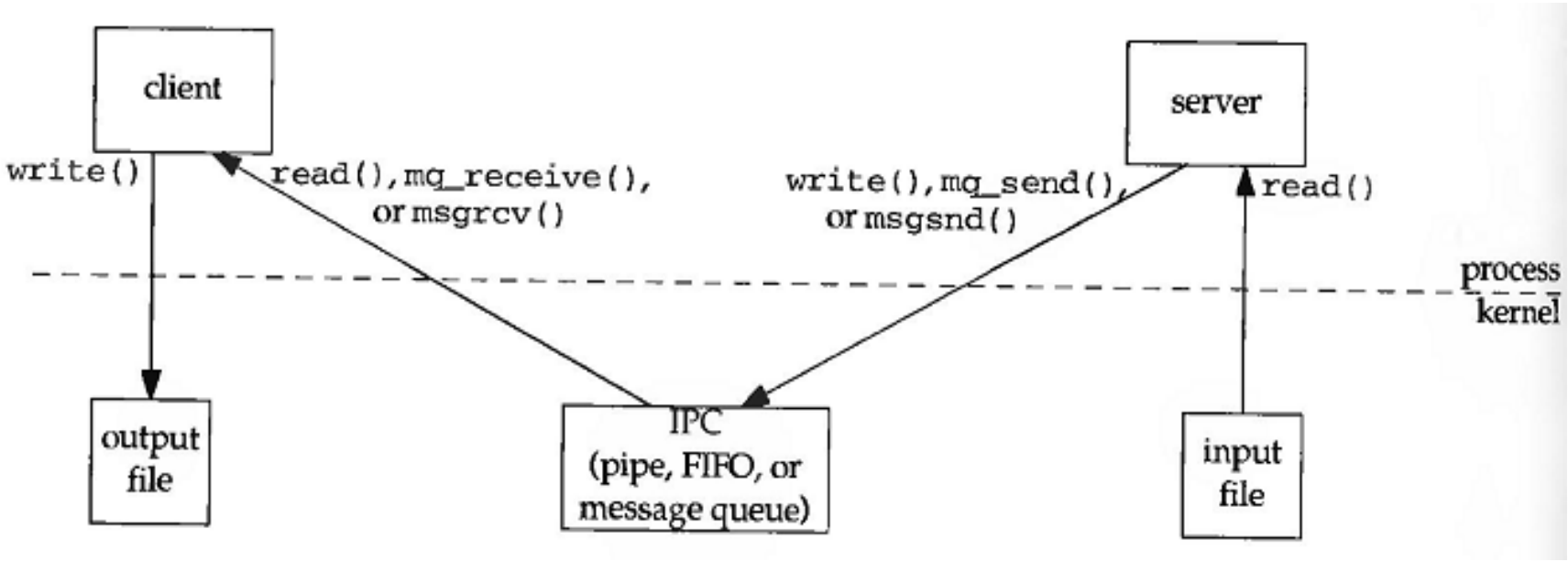
- System V Shared Memory
- Memory mapping
- TCP/IP

System V Shared Memory

Shared Memory

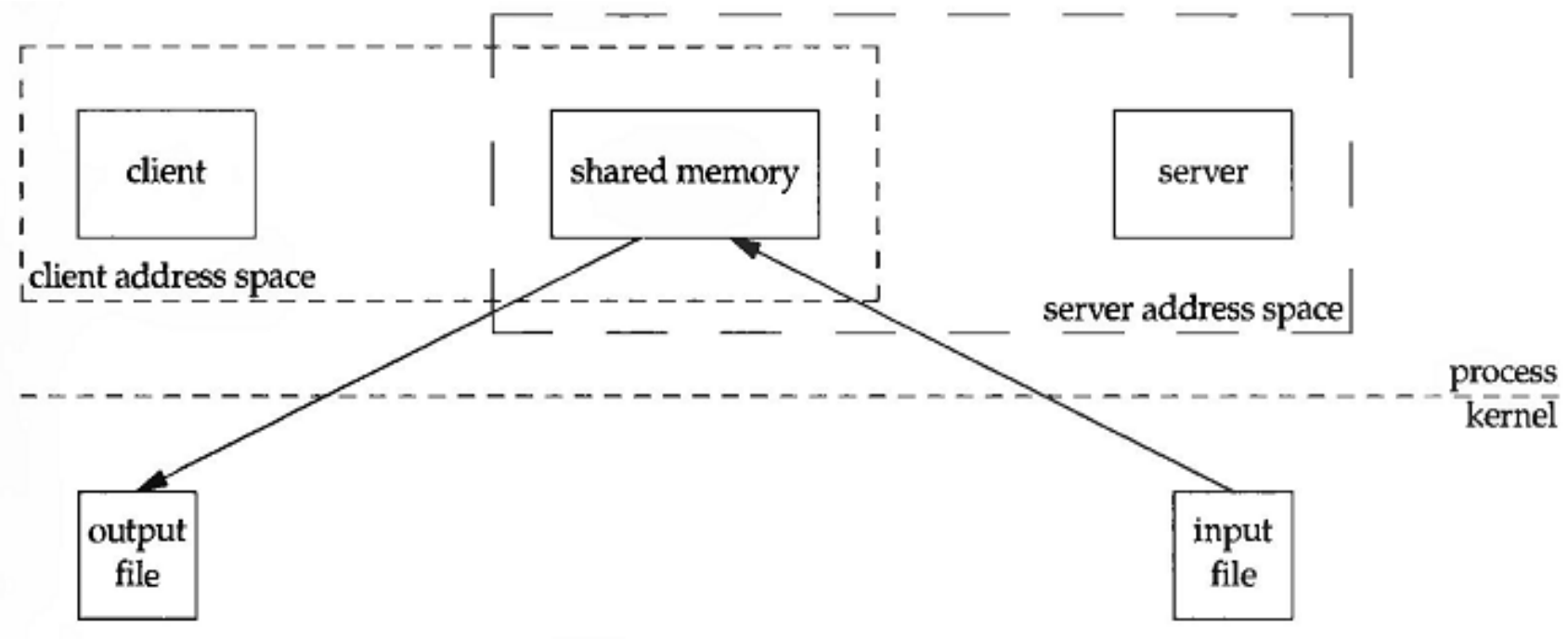
- Shared memory allows two or more processes to share a given region of memory.
- This is the fastest form of IPC, because the data does not need to be copied between the client and the server

Message Passing



- Takes 4 copies to transfer data between two processes

Shared Memory



- Takes only two steps
- Kernel is not involved in transferring data but it is involved in creating shared memory

System V Shared Memory

- For every shared memory segment kernel maintains the following structure.

```
1  struct shmid_ds {
2      struct ipc_perm shm_perm;    /* Ownership and permissions */
3      size_t   shm_segsz;          /* Size of segment in bytes */
4      time_t   shm_atime;          /* Time of last shmat() */
5      time_t   shm_dtime;          /* Time of last shmdt() */
6      time_t   shm_ctime;          /* Time of last change */
7      pid_t    shm_cpid;           /* PID of creator */
8      pid_t    shm_lpid;           /* PID of last shmat() / shmdt() */
9      shmatt_t shm_nattch;         /* Number of currently attached processes */
10 };
```

System V Shared Memory

innovate achieve lead

- Creating or opening shared memory

```
1  #include <sys/types.h>        /* For portability */
2  #include <sys/shm.h>
3  int shmget(key_t  key , size_t  size , int  shmflg );
4  //Returns shared memory segment identifier on success, or -1 on error
```

  - Size is given in bytes.
  - Size is given as zero if we are referencing existing shared memory segment.
  - When a new segment is created, the contents of the segment are initialized with zeros.
  - Flags: IPC_CREAT, IPC_EXCL

Attaching Shared Memory to a Process

innovate achieve lead

- Once a shared memory segment has been created, a process attaches it to its address space by calling shmat.

```
1  #include <sys/types.h>        /* For portability */
2  #include <sys/shm.h>
3  void *shmat(int  shmid , const void * shmaddr , int  shmflg );
4  //Returns address at which shared memory is attached on success,
5  //or (void *) -1 on error
```

  - The address in the calling process at which the segment is attached depends on the addr argument.
  - If addr is 0, the segment is attached at the first available address selected by the kernel.
    - This is the recommended technique.

Table 48-1: *shmflg* bit-mask values for *shmat()*

Value	Description
SHM_RDONLY	Attach segment read-only
SHM_REMAP	Replace any existing mapping at *shmaddr*
SHM_RND	Round *shmaddr* down to multiple of SHMLBA bytes

Detaching Shared Memory from a Process


```
1  #include <sys/types.h>  /* For portability */
2  #include <sys/shm.h>
3  int shmdt(const void * shmaddr );
4  //Returns 0 on success, or -1 on error
```

- this does not remove the identifier and its associated data structure from the system.
- the identifier remains in existence until some process (often a server) specifically removes it by calling shmctl with a command of IPC_RMID.

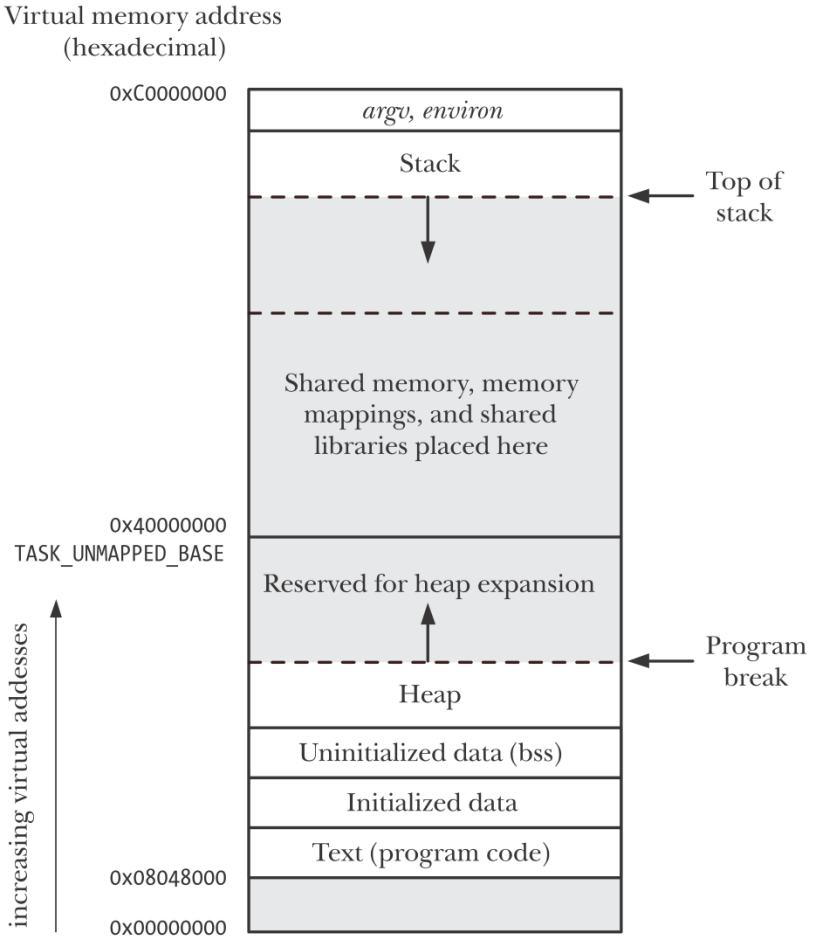
shmctl


```
1  #include <sys/types.h>        /* For portability */
2  #include <sys/shm.h>
3  int shmctl(int  shmid , int  cmd , struct shmid_ds * buf );
4  //Returns 0 on success, or -1 on error
```

- IPC_STAT, IPC_SET same as other System V IPC or XSI IPC.
- IPC_RMID:
  - Remove the shared memory segment set from the system. The segment is not removed until the last process using the segment terminates or detaches it.

Location of Shared Memory in Virtual Memory

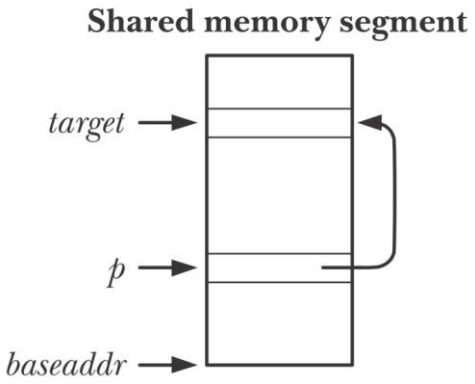
- When shared memory segment is mapped on the process address space using recommended method, it is attached as shown.



Storing Pointers in Shared Memory

- When multiple processes attach shared memory segments, addresses at which they are attached may be different.
- When storing pointers in shared memory, pointers should store relative offsets instead of absolute references.

```
1  void *p;
2  *p = target;   /* Place pointer in *p (WRONG!) */
3  //The correct approach is to store an offset at *p
4  *p = (target - baseaddr);       /* Place offset in *p */
5  //When dereferencing such pointers, we reverse the above step:
6  target = baseaddr + *p;         /* Interpret offset */
```



Shared Memory Limits

innovate achieve lead

- SHMMNI
  - System limit on no of shared memory identifies
- SHMMIN
  - Minimum size of a shared memory segment
- SHMAX
  - Maximum size of shared memory segment.
- SHMALL
  - System limit of total number of pages of shared memory.

```
1  $ cd /proc/sys/kernel
2  $ cat shmmni
3  4096
4  $ cat shmmax
5  33554432
6  $ cat shmall
7  2097152
```

Table 48-2: System V shared memory limits

Limit	Ceiling value (x86-32)	Corresponding file in /proc/sys/kernel
SHMMNI	32768 (IPCMNI)	shmmni
SHMMAX	Depends on available memory	shmmax
SHMALL	Depends on available memory	shmall

Example

- Transferring data via shared memory
  - Two programs: writer and reader
    - Writer: reads from stdin and copies to shm
    - Reader: copies data from shm to stdout
  - 2 semaphores to control the access to the shared memory in an alternative manner.

```c
/*reader.c*/
#define SEMKEYPATH "./shmserver.c"
#define SHMKEYPATH "./shmserver.c"
#define NUMSEMS 2
#define SIZEOFSHMSEG 50
int main(int argc, char *argv[])
{   void *shm_address;
    struct sembuf operations[2];
semkey = ftok(SEMKEYPATH,1);
shmkey = ftok(SHMKEYPATH,1);
semid = semget( semkey, NUMSEMS, 0666 | IPC_CREAT | IPC_EXCL );
semctl( semid, 0, SETVAL, 1);//for writing
semctl( semid, 1, SETVAL, 0);//for reading
shmid = shmget(shmkey, SIZEOFSHMSEG, 0666 | IPC_CREAT | IPC_EXCL);
shm_address = shmat(shmid, NULL, 0);
for(;;){operations[0].sem_num = 1;
        operations[0].sem_op =  -1;/*access for reading*/
        operations[0].sem_flg = 0;
        rc = semop( semid, operations, 1 );
        printf("Server Received : \"%s\"\n", (char *) shm_address);
        operations[0].sem_num = 0;
        operations[0].sem_op  = 1;/*allow writer*/
        operations[0].sem_flg =0;
        rc = semop( semid, operations, 1 );}
}}
```

```c
/*writer.c*/
#define SEMKEYPATH "./shmserver.c"
#define SHMKEYPATH "./shmserver.c"
#define NUMSEMS 2
#define SIZEOFSHMSEG 50
int main(int argc, char *argv[])
{  struct sembuf operations[2];
    void         *shm_address;
semkey = ftok(SEMKEYPATH,SEMKEYID);
shmkey = ftok(SHMKEYPATH,SHMKEYID);
semid = semget( semkey, NUMSEMS, 0666);
shmid = shmget(shmkey, SIZEOFSHMSEG, 0666);
shm_address = shmat(shmid, NULL, 0);
for(;;){ printf("enter data:\n");
operations[0].sem_num = 0;
operations[0].sem_op =  -1;/*access for writing*/
operations[0].sem_flg = 0;
rc = semop( semid, operations, 1 );
        read(0,shm_address,SIZEOFSHMSEG);
operations[0].sem_num = 1;/* Operate on the second sem     */
operations[0].sem_op =  1;/* Increment the semval by one   */
operations[0].sem_flg = 0;/* Allow a wait to occur         */
}
return 0;
}
```

Memory Mapping

Memory mapping

- mmap() sys call creates a new memory mapping in the calling process's virtual address space.
  - File Mapping
    - Maps a region of a file into the virtual memory
    - Contents of file can be accessed just like accessing memory.
  - Anonymous mapping
    - Doesn't have a corresponding file. Instead pages of the mapping are initialized to 0's.
- Memory in one process's mapping can be shared with mapping in other processes. Page tables point to the same pages of RAM. This can happen when
  - Two processes map the same region of the file.
  - Child inherits the mapping in the parent.

Changes to Memory Mapped

- A process can see the cahnges made by others depending on the type of mapping:
  - Private mapping (MAP_PRIVATE):
    - Modifications are not visible to other processes. If file mapped, they do not go to the file.
    - Changes are private to each process using copy-on-write.
  - Shared mapping (MAP_SHARED)
    - Changes are visible to other processes. If file mapped, changes are written to the file.

Table 49-1: Purposes of various types of memory mappings

Visibility of modifications	Mapping type	
	File	Anonymous
Private	Initializing memory from contents of file	Memory allocation
Shared	Memory-mapped I/O; sharing memory between processes (IPC)	Sharing memory between processes (IPC)

Types of memory mappings

- Private file mapping
  - Memory is initialized from the contents of the file.
  - e.g. initializing text segment, data segment
- Private anonymous mapping
  - To allocate new memory (zero filled) for the process.
- Shared file mapping
  - Memory mapped is initialized with the contents from a file.
  - Changes are visible to all processes.
  - Purposes: Memory mapped I/O, Inter process communication (IPC)
- Shared anonymous mapping
  - Each call to mmap() allocates memory initialized with 0's. Changes are visible.
  - IPC between related processes (parent & child)

mmap()

- mmap() system call creates a new mapping in the virtual address space for the calling process.

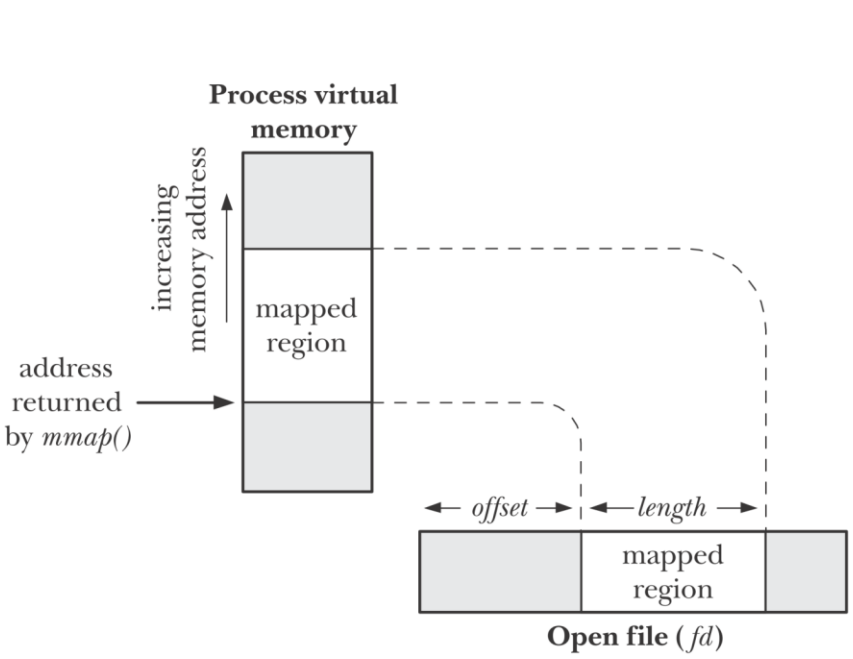
```
1  #include <sys/mman.h>
2  void *mmap(void * addr , size_t  length , int  prot ,
3  int  flags , int  fd , off_t  offset );
4  //Returns starting address of mapping on success,
5  //or  MAP_FAILED  on error
```

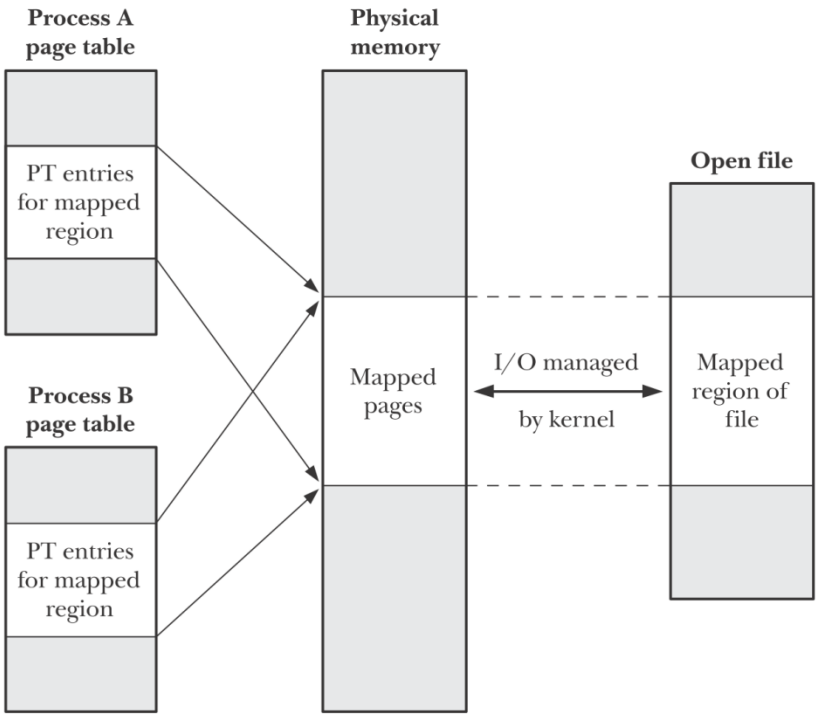
- *addr* refers at which address mapping should start. Recommended is NULL.
- Length is size of mapping in bytes.
- *Prot*: Memory Protection flags:
- *flags*: MAP_PRIVATE/MAP_SHARED
- *fd, offset:* for file mapping

Table 49-2: Memory protection values

Value	Description
PROT_NONE	The region may not be accessed
PROT_READ	The contents of the region can be read
PROT_WRITE	The contents of the region can be modified
PROT_EXEC	The contents of the region can be executed

File mapping





Memory Unmapping

innovate achieve lead

```
1  #include <sys/mman.h>
2  int munmap(void * addr , size_t  length );
3  //Returns 0 on success, or -1 on error
```

- Removes the mapping starting address and for tye corresponding length.

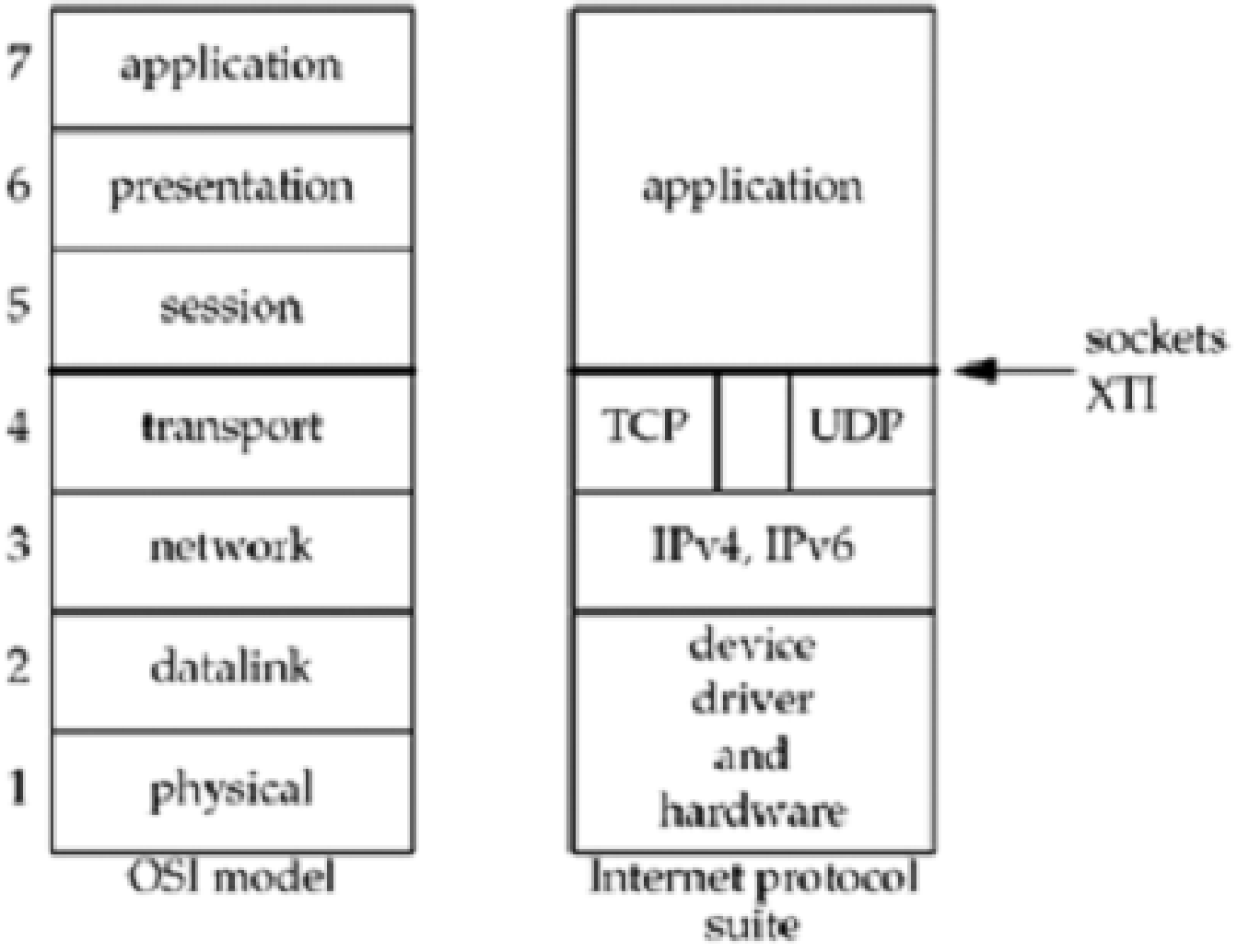
innovate achieve lead

BITS Pilani

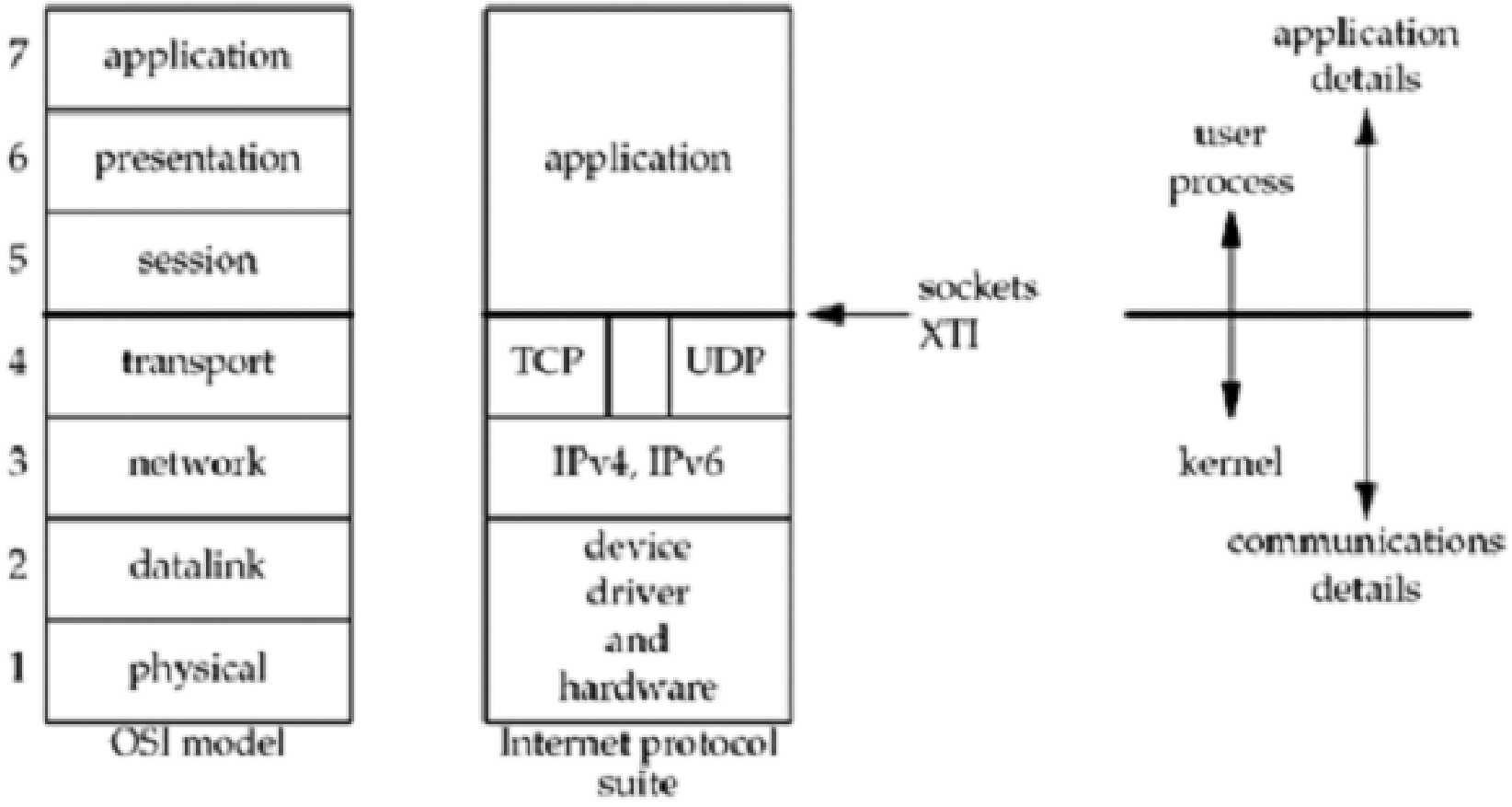
Pilani Campus

TCP/IP

OSI & Internet protocol suite



TCP/IP


7
application
6
presentation
5
session
4
transport
3
network
2
datalink
1
physical
OSI model
application
TCP
UDP
IPv4, IPv6
device driver and hardware
Internet protocol suite
sockets XTI
application details
user process
kernel
communications details

TCP or UDP

- At the internet layer, a destination address identifies a host computer; no further distinction is made regarding which process will receive the datagram
- TCP or UDP add a mechanism that distinguishes among destinations within a given host, allowing multiple processes to send and receive datagrams independently

Why process is not the destination for a message?

- Processes are created and destroyed dynamically
- A process can be repalced with a new process without informing all senders
- Identify a destination by the function rather than the process which implements it
- A process can handle multiple functions, so there should a way to specify which one sender desires

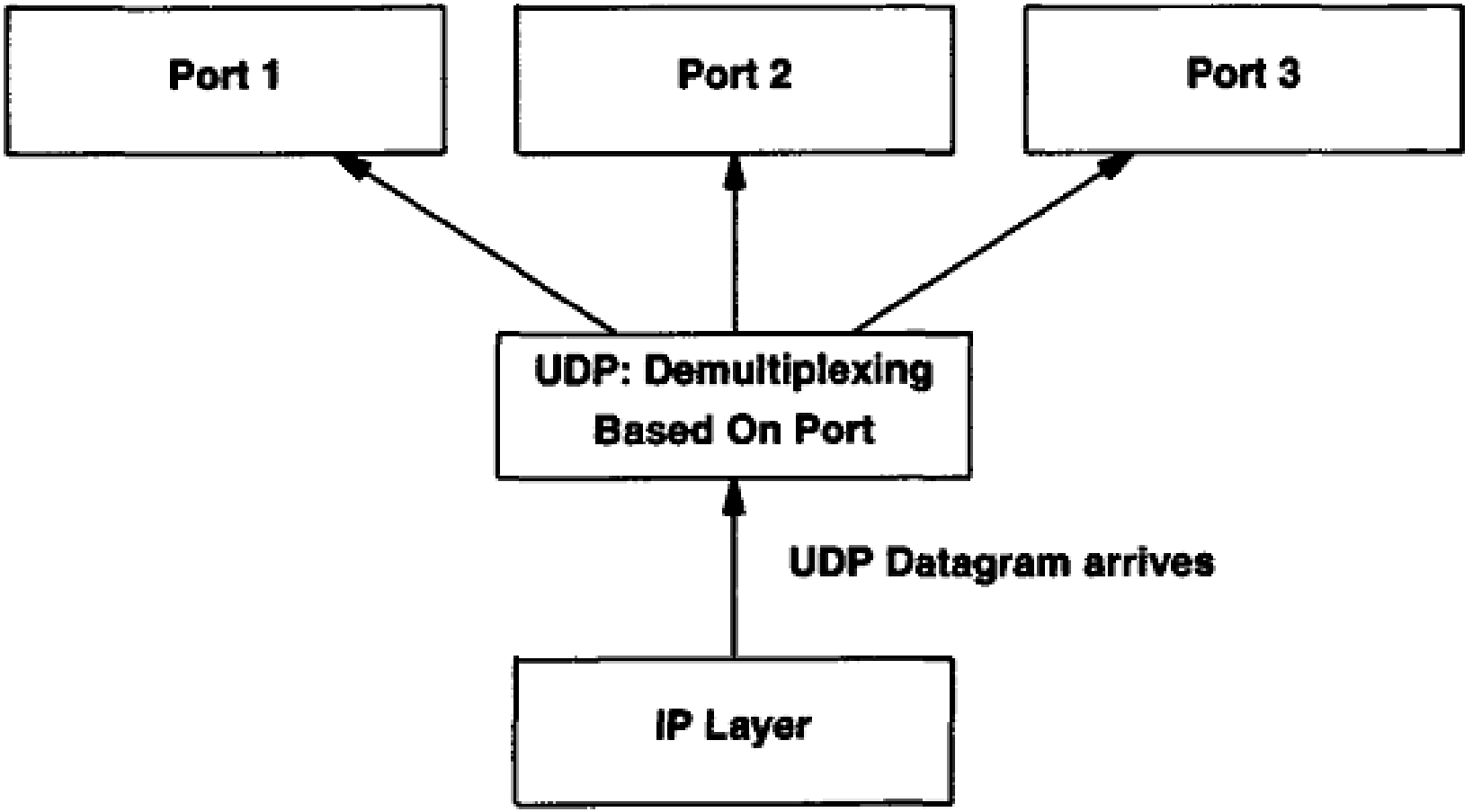
Protocol Ports

- Instead of thinking process as ultimate destination, imagine that each machine contains a set of abstract destination points called protocol ports
- Each protocol port is identified by a positive integer
- Operating systems provide some mechanism that processes use, to specify a port.

Port Numbers

- The port numbers are divided into three ranges by Internet Assigned Numbers Authority
- The well-known ports: 0 through 1023. These port numbers are controlled and assigned by the IANA.
- The registered ports: 1024 through 49151. These are not controlled by the IANA, but the IANA registers and lists the uses of these ports as a convenience to the community.
- The dynamic or private ports, 49152 through 65535. The IANA says nothing about these ports. These are what we call ephemeral ports. (49152 is three-fourths of 65536.)

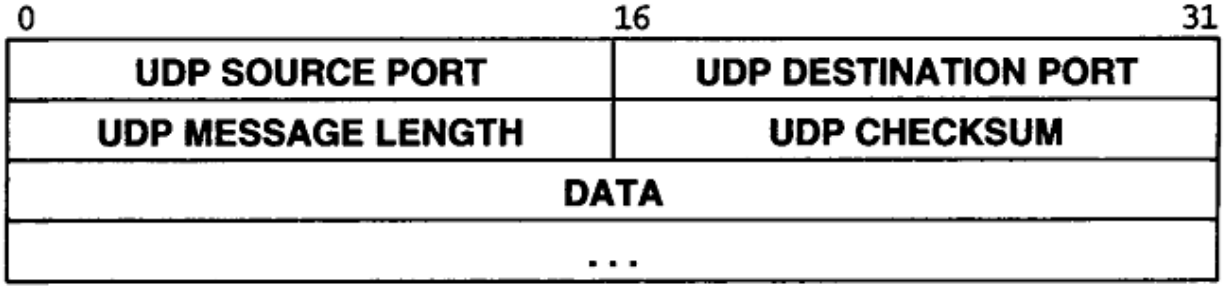
Ports


Port 1
Port 2
Port 3
UDP: Demultiplexing Based On Port
UDP Datagram arrives
IP Layer

UDP (User Datagram Protocol)

- UDP provides an unreliable connectionless delivery service
- UDP uses IP to deliver datagrams to the right host.
- UDP uses *ports* to provide communication services to individual processes.

UDP header



- Header size is 8 bytes
- Lack of reliability:
  - checksum detects an error
  - the datagram is dropped in the network
- If we want to be certain that a datagram reaches its destination, we can build lots of features into our application: acknowledgments from the other end, timeouts, retransmissions, and the like.

Some standard UDP based services and their ports

Decimal	Keyword	UNIX Keyword	Description
0	-	-	Reserved
7	ECHO	echo	Echo
9	DISCARD	discard	Discard
11	USERS	systat	Active Users
13	DAYTIME	daytime	Daytime
15	-	netstat	Network status program
17	QUOTE	qotd	Quote of the Day
19	CHARGEN	chargen	Character Generator
37	TIME	time	Time
42	NAMESERVER	name	Host Name Server
43	NICNAME	whois	Who Is
53	DOMAIN	nameserver	Domain Name Server
67	BOOTPS	bootps	BOOTP or DHCP Server
68	BOOTPC	bootpc	BOOTP or DHCP Client
69	TFTP	tftp	Trivial File Transfer
88	KERBEROS	kerberos	Kerberos Security Service
111	SUNRPC	sunrpc	Sun Remote Procedure Call
123	NTP	ntp	Network Time Protocol

TCP
Transmission Control Protocol

- TCP provides connections between clients and servers.
- TCP uses the connection, not the protocol port, as its fundamental abstraction.
- Connections are identified by a pair of endpoints.
  - Endpoint means (ip, port)
- TCP provides:
  - Connection-oriented
  - Reliable
  - Full-duplex
  - Byte-Stream

Connection-Oriented

- *Connection oriented* means that a virtual connection is established before any user data is transferred.
- A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.
- If the connection cannot be established - the user program is notified.
- If the connection is ever interrupted - the user program(s) is notified.

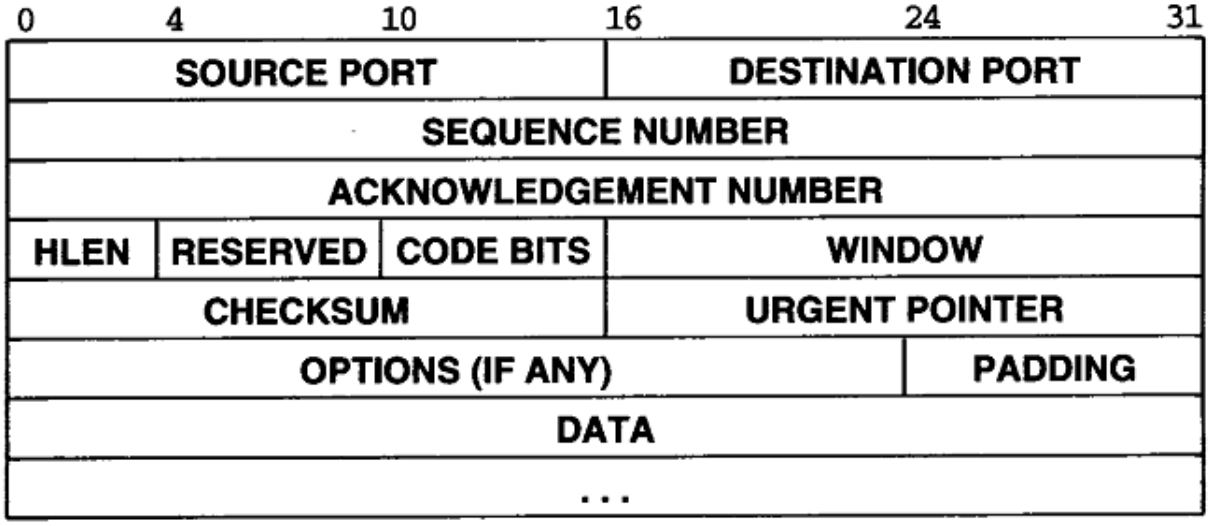
TCP Ports

- Interprocess communication via TCP is achieved with the use of ports (just like UDP).
- UDP ports have no relation to TCP ports (different name spaces).

TCP Segments

- TCP views the data stream as a sequence of bytes that it divides into segments for transmission. Segments carry varying sizes of data.
- The chunk of data that TCP asks IP to deliver is called a *TCP segment*.
- Each segment contains:
  - data bytes from the byte stream
  - control information that identifies the data bytes

TCP Segment Format



Bit (left to right)	Meaning if bit set to 1
URG	Urgent pointer field is valid
ACK	Acknowledgement field is valid
PSH	This segment requests a push
RST	Reset the connection
SYN	Synchronize sequence numbers
FIN	Sender has reached end of its byte stream

TCP Segments

- Segments are exchanged to establish connections, transfer data, send acknowledgements, advertise window sizes, and close connections.
- Because TCP uses piggybacking, acknowledgement can be sent along with data
- TCP advertises how much data it is willing to accept every time it sends segment by specifying its buffer size in the WINDOW field

TCP Connection Establishment

- Three-way handshake
- It accomplishes two important functions.
  - It guarantees that both sides are ready to transfer data (and that they know they are both ready)
  - it allows both sides to agree on initial sequence numbers.
- Sequence numbers are sent and acknowledged during the handshake. Each machine must choose an initial sequence number at random that it will use to identify bytes in the stream it is sending.

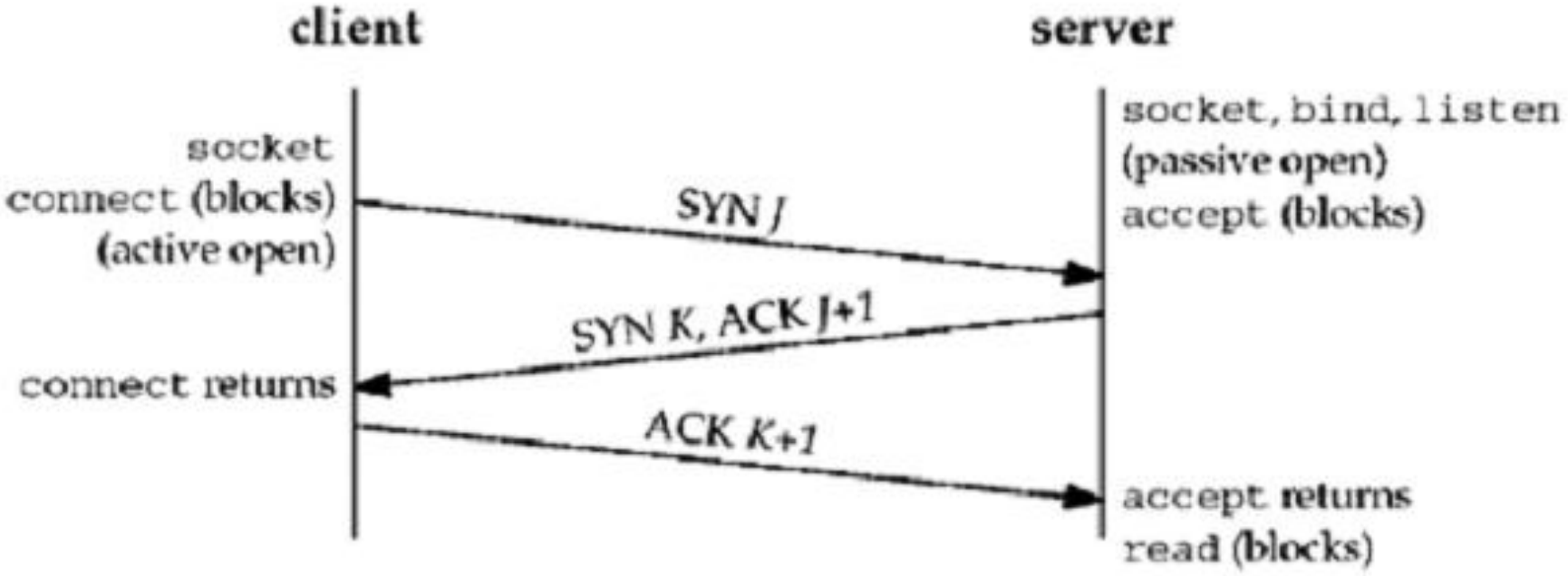
TCP Connection Establishment

- When a client requests a connection, it sends a “SYN” segment (a special TCP segment) to the server port.
- SYN stands for synchronize. The SYN message includes the client’s ISN.
- ISN is Initial Sequence Number.
- Every TCP segment includes a Sequence Number that refers to the first byte of data included in the segment.
- Every TCP segment includes a Request Number (Acknowledgement Number) that indicates the byte number of the next data that is expected to be received.
  - All bytes up through this number have already been received.

TCP Connection Establishment

- A server accepts a connection.
  - Must be looking for new connections!
- A client requests a connection.
  - Must know where the server is!
- A client starts by sending a SYN segment with the following information:
  - Client's ISN (generated pseudo-randomly)
  - Maximum Receive Window for client.
  - Optionally (but usually) MSS (largest datagram accepted).
  - No payload! (Only TCP headers)
- When a waiting server sees a new connection request, the server sends back a SYN segment with:
  - Server's ISN (generated pseudo-randomly)
  - Request Number is Client ISN+1
  - Maximum Receive Window for server.
  - Optionally (but usually) MSS
  - No payload! (Only TCP headers)

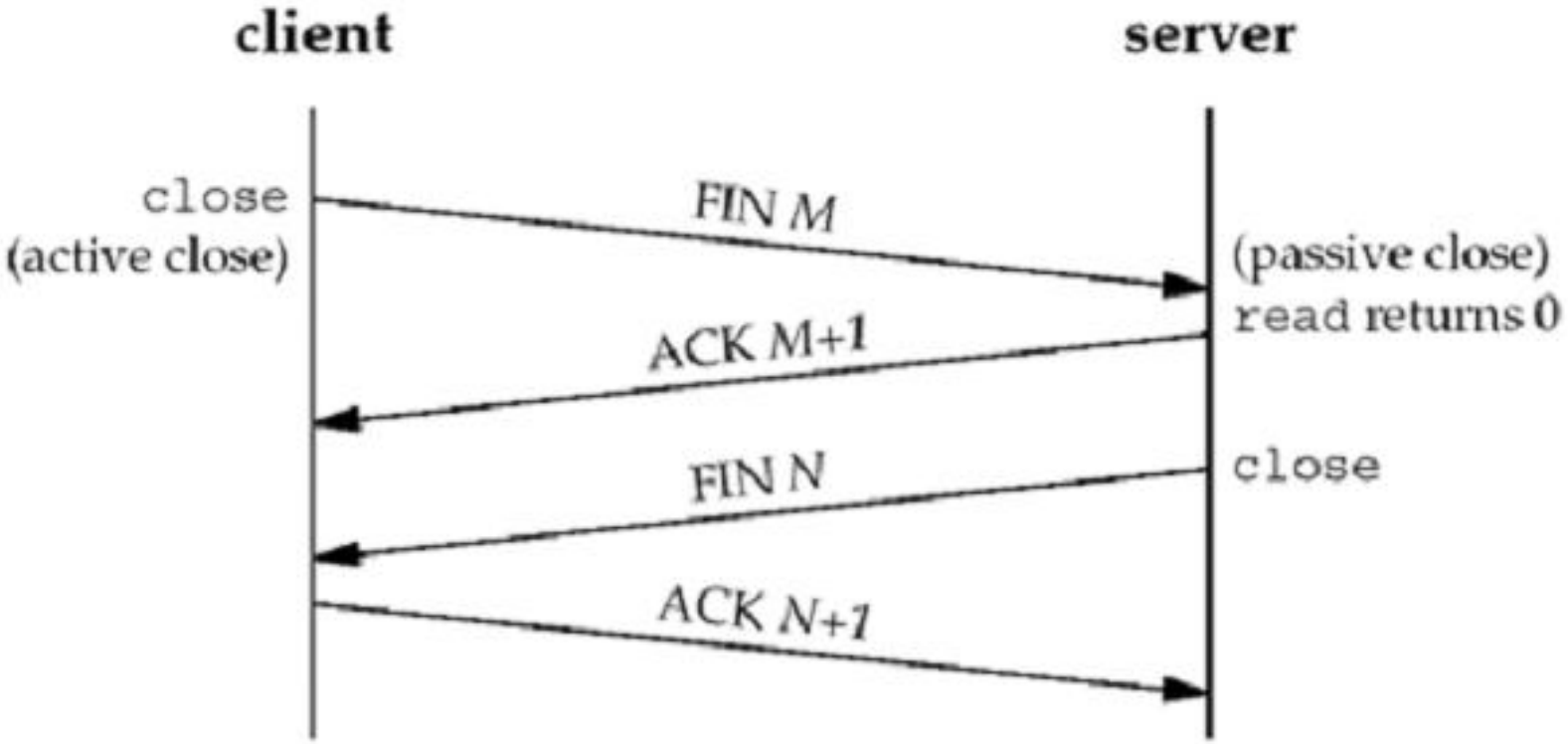
TCP Connection Establishment



Connection Termination

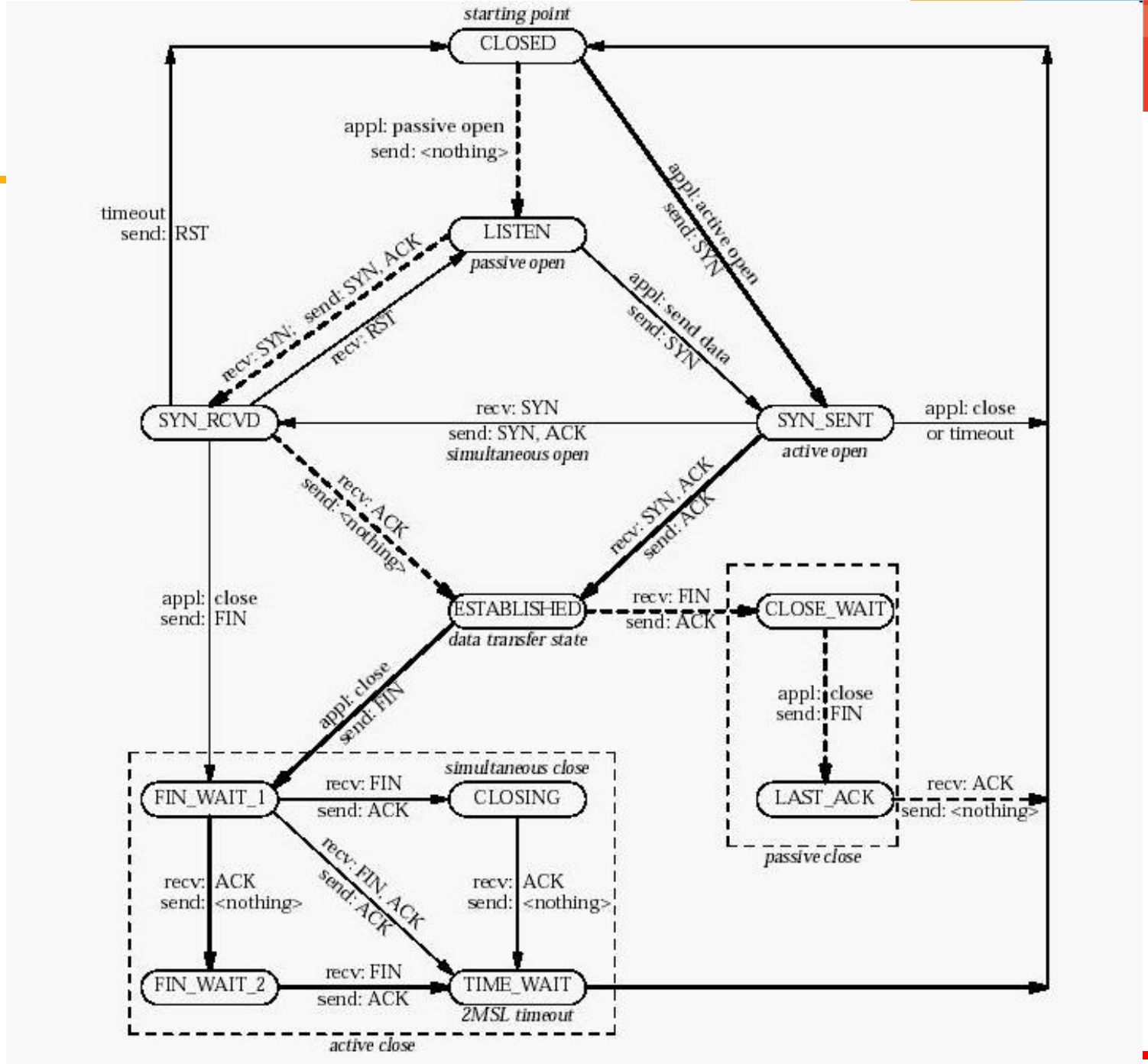
- The TCP layer can send a RST segment that terminates a connection if something is wrong.
- Usually the application tells TCP to terminate the connection gracefully with a FIN segment.
- Either end of the connection can initiate termination.
- A FIN is sent, which means the application is done sending data.
- The FIN is ACK'd.
- The other end must now send a FIN.
- That FIN must be ACK'd.

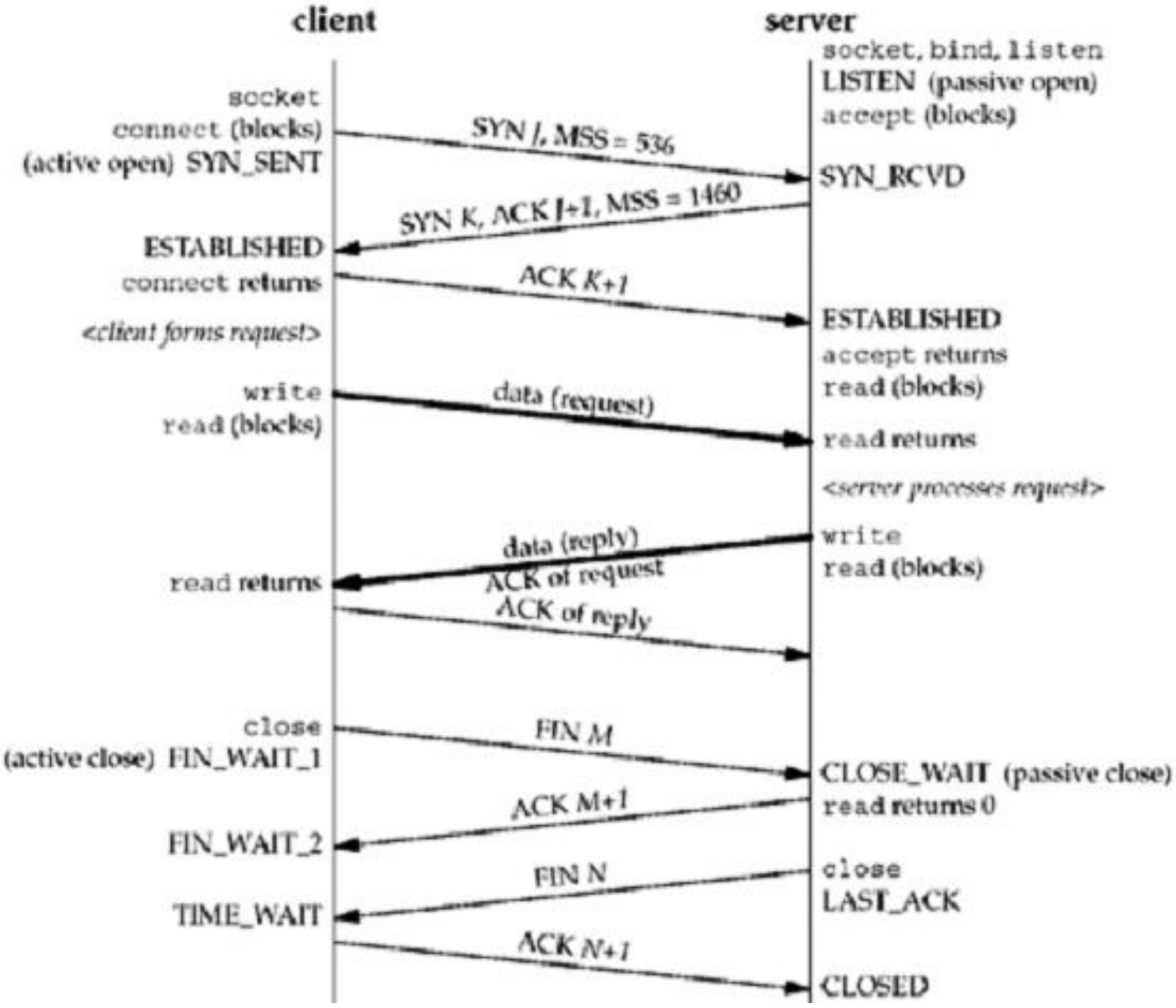
Connection Termination



TCP Connection State Diagram

- There are 11 different states defined for a connection
  - based on the current state and the segment received in that state.
- One reason for showing the state transition diagram is to show the 11 TCP states with their names. These states are displayed by netstat, which is a useful tool when debugging client/server applications


starting point
CLOSED
appl: passive open
send: <nothing>
appl: active open
send: SYN
timeout
send: RST
LISTEN
passive open
recv: SYN; send: SYN, ACK
recv: RST
appl: send data
send: SYN
SYN_RCVD
recv: SYN
send: SYN, ACK
simultaneous open
SYN_SENT
active open
appl: close
or timeout
recv: ACK
send: <nothing>
recv: SYN, ACK
send: ACK
appl: close
send: FIN
ESTABLISHED
data transfer state
recv: FIN
send: ACK
CLOSE_WAIT
appl: close
send: FIN
LAST_ACK
recv: ACK
send: <nothing>
passive close
appl: close
send: FIN
simultaneous close
FIN_WAIT_1
recv: FIN
send: ACK
CLOSING
recv: FIN, ACK
send: ACK
recv: ACK
send: <nothing>
recv: ACK
send: <nothing>
FIN_WAIT_2
recv: FIN
send: ACK
TIME_WAIT
2MSL timeout
active close


client
server
socket, bind, listen
LISTEN (passive open)
accept (blocks)
socket
connect (blocks)
(active open) SYN_SENT
SYN J, MSS = 536
SYN_RCVD
SYN K, ACK J+1, MSS = 1460
ESTABLISHED
connect returns
ACK K+1
<client forms request>
ESTABLISHED
accept returns
read (blocks)
write
read (blocks)
data (request)
read returns
<server processes request>
write
read (blocks)
data (reply)
ACK of request
read returns
ACK of reply
close
(active close) FIN_WAIT_1
FIN M
CLOSE_WAIT (passive close)
read returns 0
ACK M+1
FIN_WAIT_2
FIN N
close
LAST_ACK
TIME_WAIT
ACK N+1
CLOSED

What is the purpose of TIME_WAIT?

- Once a TCP connection has been terminated (the last ACK sent) there is some unfinished business:
  - What if the ACK is lost? The last FIN will be resent and it must be ACK'd.
  - What if there are lost or duplicated segments that finally reach the incarnation of the previous connection after a long delay?
- The MSL is the maximum amount of time that any given IP datagram can live in a network

Q&A

Next Time

innovate achieve lead

- Please read through R1: chapters 47-48

innovate achieve lead

BITS Pilani

Pilani Campus

Thank You