Chapter 6:
Transport Layer

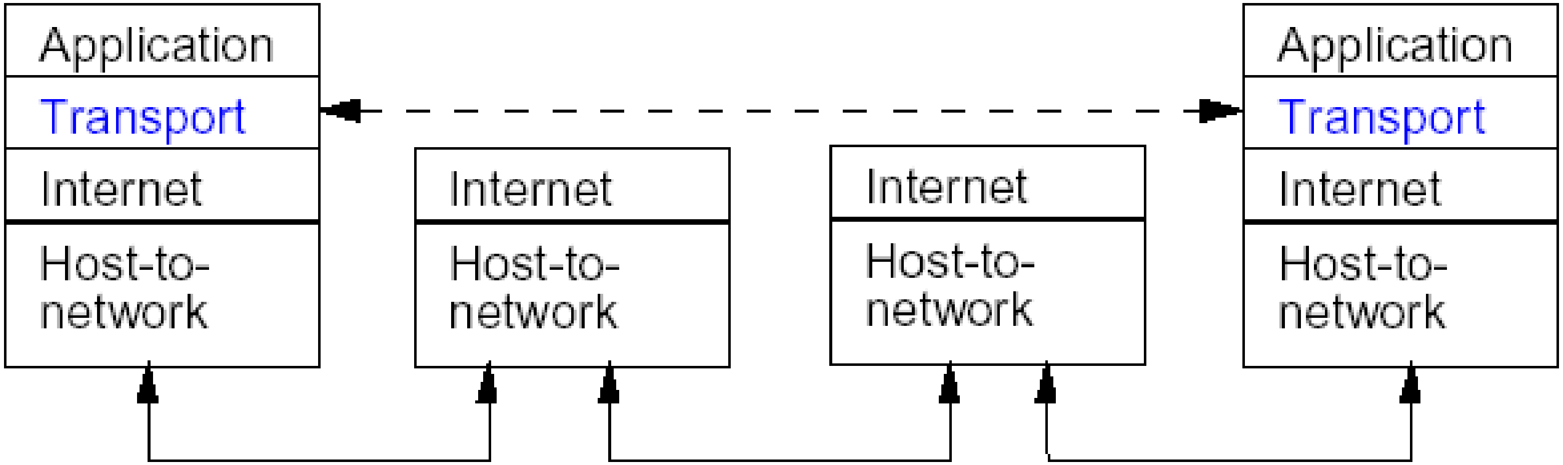
Summary

Part A: Introduction
Part B: Socket
Part C: TCP
Part D: UDP

Part A: Introduction

- The transport layer is an end-to-end layer – this means that nodes within the subnet do not participate in transport layer protocols – only the end hosts.

- As with other layers, transport layer protocols send data as a sequence of packets (segments).



Multiplexing (1)

- The **network layer** provides communication between two hosts.
- The **transport layer** provides communication between two ***processes*** running on different hosts.
- A process is an instance of a program that is running on a host.
- There may be multiple processes communicating between two hosts – for example, there could be a FTP session and a Telnet session between the same two hosts.

Multiplexing (2)

- The transport layer provides a way to multiplex / demultiplex communication between various processes.
- To provide multiplexing, the transport layer adds an address to each segment indicating the source and destination processes.
- Note these addresses need only be unique locally on a given host.
- In TCP/IP these transport layer addresses are called ***port-numbers***.

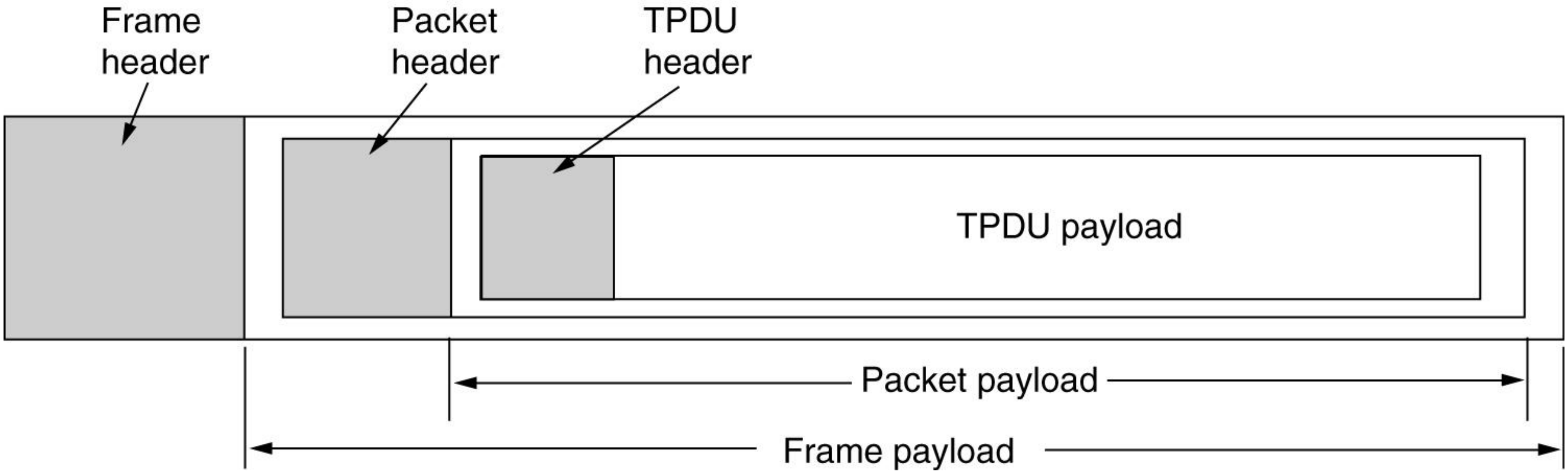
Multiplexing (3)

Session Identifier (5-tuple): uniquely identifies a process-to-process connection in the Internet.

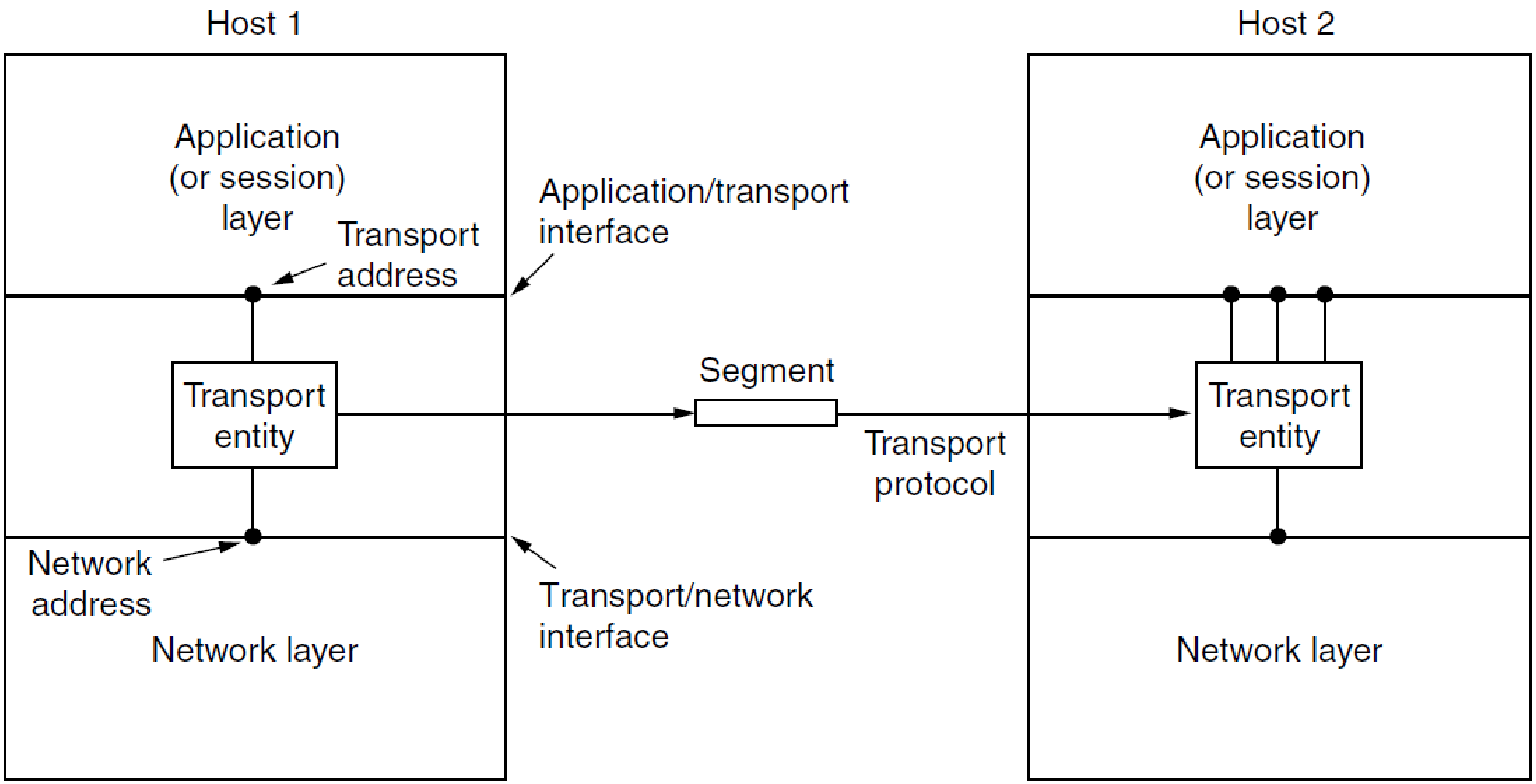
- Sender IP address
- Sender port
- Destination IP address
- Destination port
- Transport layer protocol

TPDU

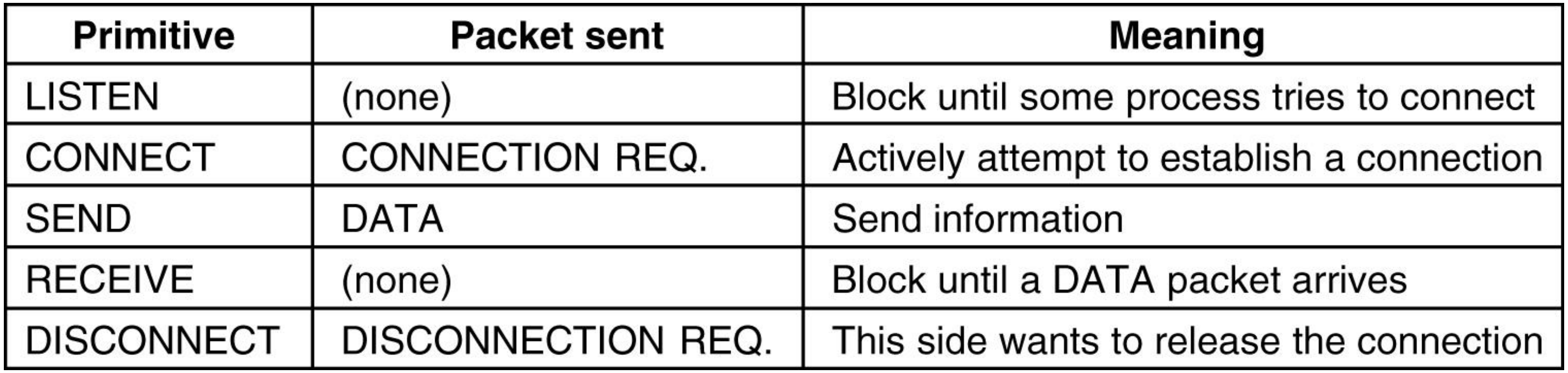
Nesting of Transport Protocol Data Unit (TPDU), packets, and frames.



Network, Transport and Application



Transport Service Primitives



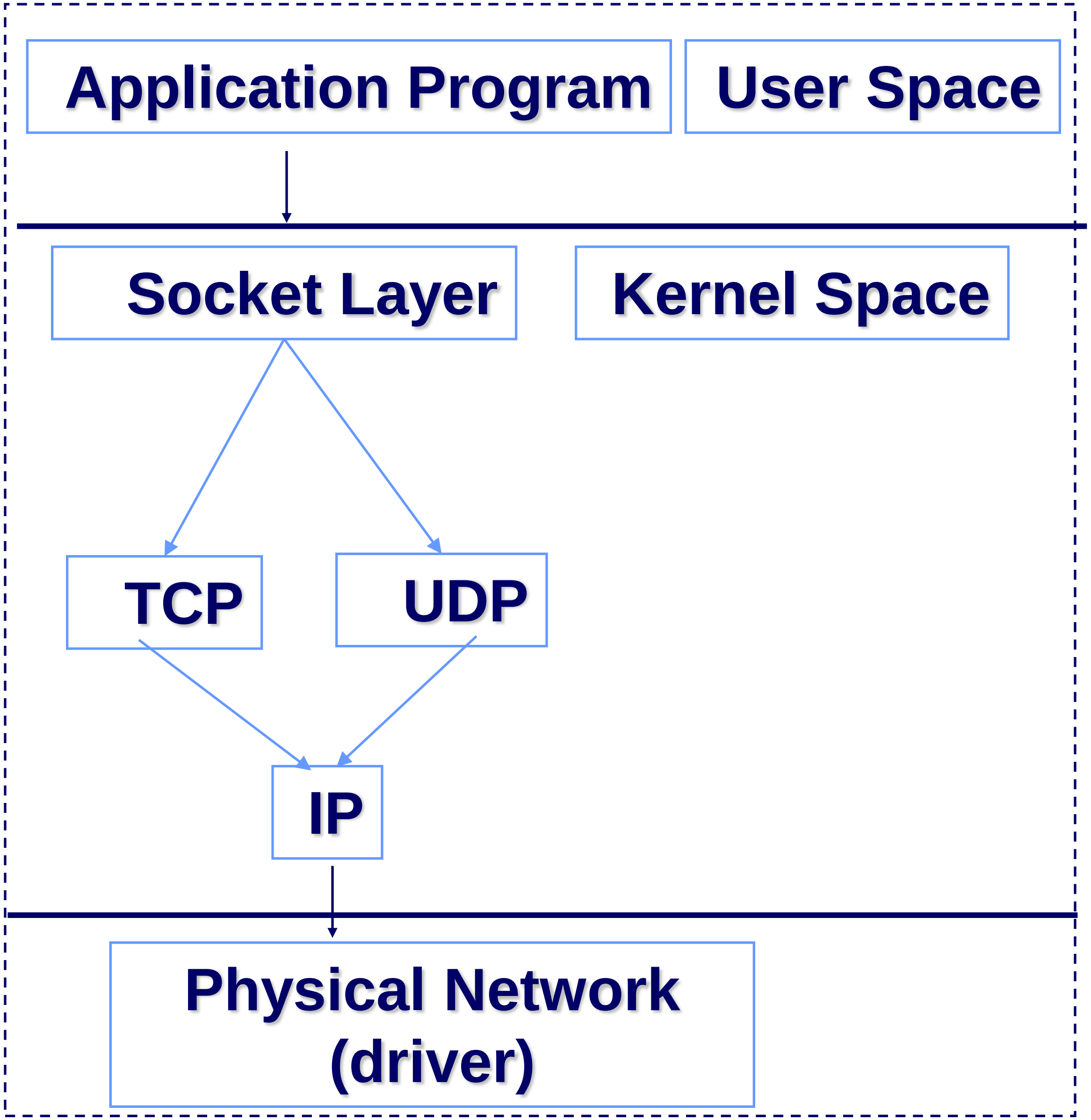
Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

The primitives for a simple transport service.

PART B: SOCKET

- Software interface designed to communicate between the user program and TCP/IP protocol stack
- Implemented by a set of library function calls
- Socket is a data structure inside the program
- Both client and server programs communicate via a pair of sockets

Socket Framework



Application Program

User Space

Socket Layer

Kernel Space

TCP

UDP

IP

Physical Network (driver)

Socket Families

There are several significant socket domain families:

⇨Internet Domain Sockets (AF_INET)

-implemented via IP addresses and port numbers

⇨Unix Domain Sockets (AF_UNIX)

-implemented via filenames (think “named pipe”)

⇨Novell IPX (AF_IPX)

⇨AppleTalk DDS (AF_APPLETALK)

Type of Socket

- Stream (SOCK_STREAM) Uses TCP protocol. Connection-oriented service
- Datagram (SOCK_DGRAM) Uses UDP protocol. Connectionless service
- Raw (SOCK_RAW) Used for testing

Creating a Socket

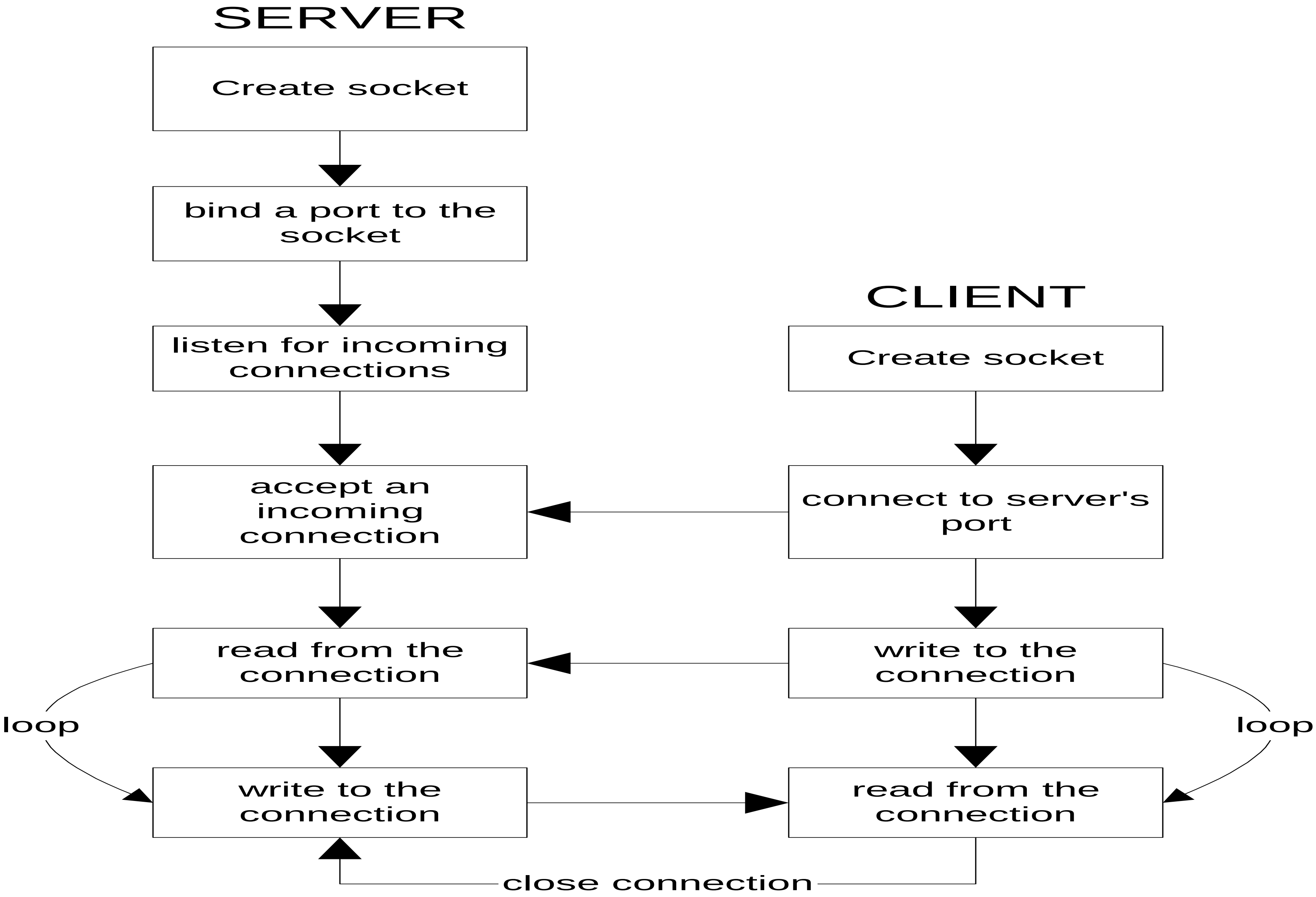
```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- domain is one of the *Protocol Families* (PF_INET, PF_UNIX, etc.)

- type defines the communication protocol semantics, usually defines either:
  - SOCK_STREAM: connection-oriented stream (TCP)
  - SOCK_DGRAM: connectionless, unreliable (UDP)

- protocol specifies a particular protocol, just set this to 0 to accept the default

TCP Client/Server



SERVER

Create socket

bind a port to the socket

listen for incoming connections

accept an incoming connection

read from the connection

write to the connection

CLIENT

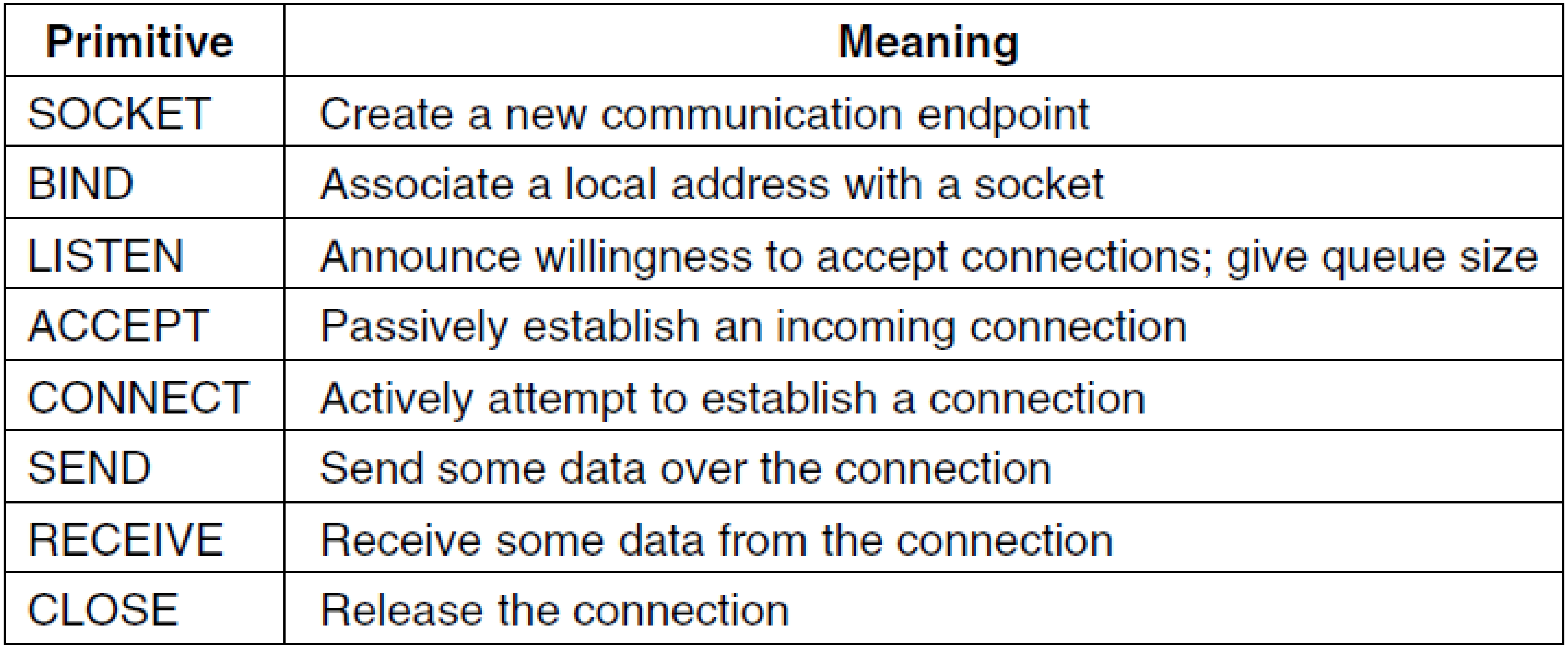
Create socket

connect to server's port

write to the connection

read from the connection

Socket Primitives for TCP



Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

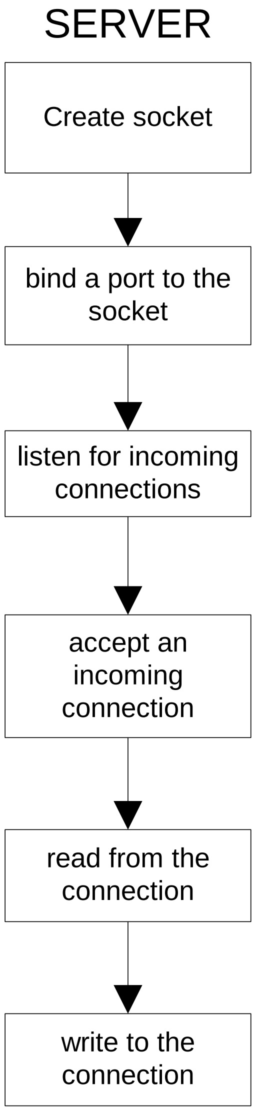
TCP Server

Sequence of calls

sock_init()	Create the socket
bind()	Register port with the system
listen()	Establish client connection
accept()	Accept client connection
read/write	read/write data
close()	shutdown

e ding not in theory

Server Side Socket Details (Functions)


SERVER

Create socket

bind a port to the socket

listen for incoming connections

accept an incoming connection

read from the connection

write to the connection

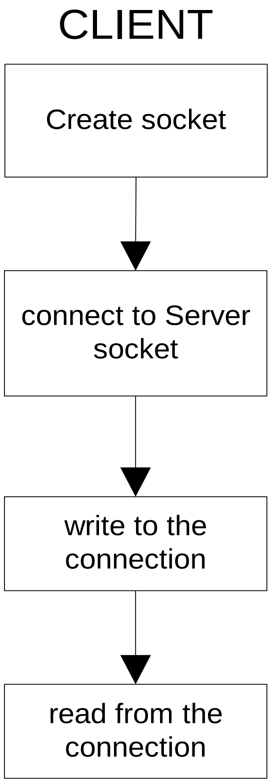
```
int socket(int domain, int type, int protocol)
sockfd = socket(PF_INET, SOCK_STREAM, 0);

int bind(int sockfd, struct sockaddr *server_addr, socklen_t length)
bind(sockfd, &server, sizeof(server));

int listen( int sockfd, int num_queued_requests)
listen( sockfd, 5);

int accept(int sockfd, struct sockaddr *incoming_address, socklen_t length)
newfd = accept(sockfd, &client, sizeof(client)); /* BLOCKS */

int read(int sockfd, void * buffer, size_t buffer_size)
read(newfd, buffer, sizeof(buffer));

int write(int sockfd, void * buffer, size_t buffer_size)
write(newfd, buffer, sizeof(buffer));
```

TCP Client

Sequence of calls

sock_init ()	Create socket
connect()	Set up connection
write/read	write/read data
close()	Shutdown

Client Side Socket Details



CLIENT

Create socket

connect to Server socket

write to the connection

read from the connection

```
int socket(int domain, int type, int protocol)
sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

```
int connect(int sockfd, struct sockaddr *server_address, socklen_t length)
connect(sockfd, &server, sizeof(server));
```

```
int write(int sockfd, void * buffer, size_t buffer_size)
write(sockfd, buffer, sizeof(buffer));
```

```
int read(int sockfd, void * buffer, size_t buffer_size)
read(sockfd, buffer, sizeof(buffer));
```

UDP Clients and Servers

Connectionless clients and servers create a socket using SOCK_DGRAM instead of SOCK_STREAM

Connectionless servers do not call listen() or accept(), and *usually* do not call connect()

Since connectionless communications lack a sustained connection, several methods are available that allow you to *specify a destination address with every call*:

- sendto(sock, buffer, buflen, flags, to_addr, tolen);
- recvfrom(sock, buffer, buflen, flags, from_addr, fromlen);

UDP Server

Sequence of calls

sock_init()	Create socket
bind()	Register port with the system
recfrom/sendto	Receive/send data
close()	Shutdown

→ No Listen & Accept Part

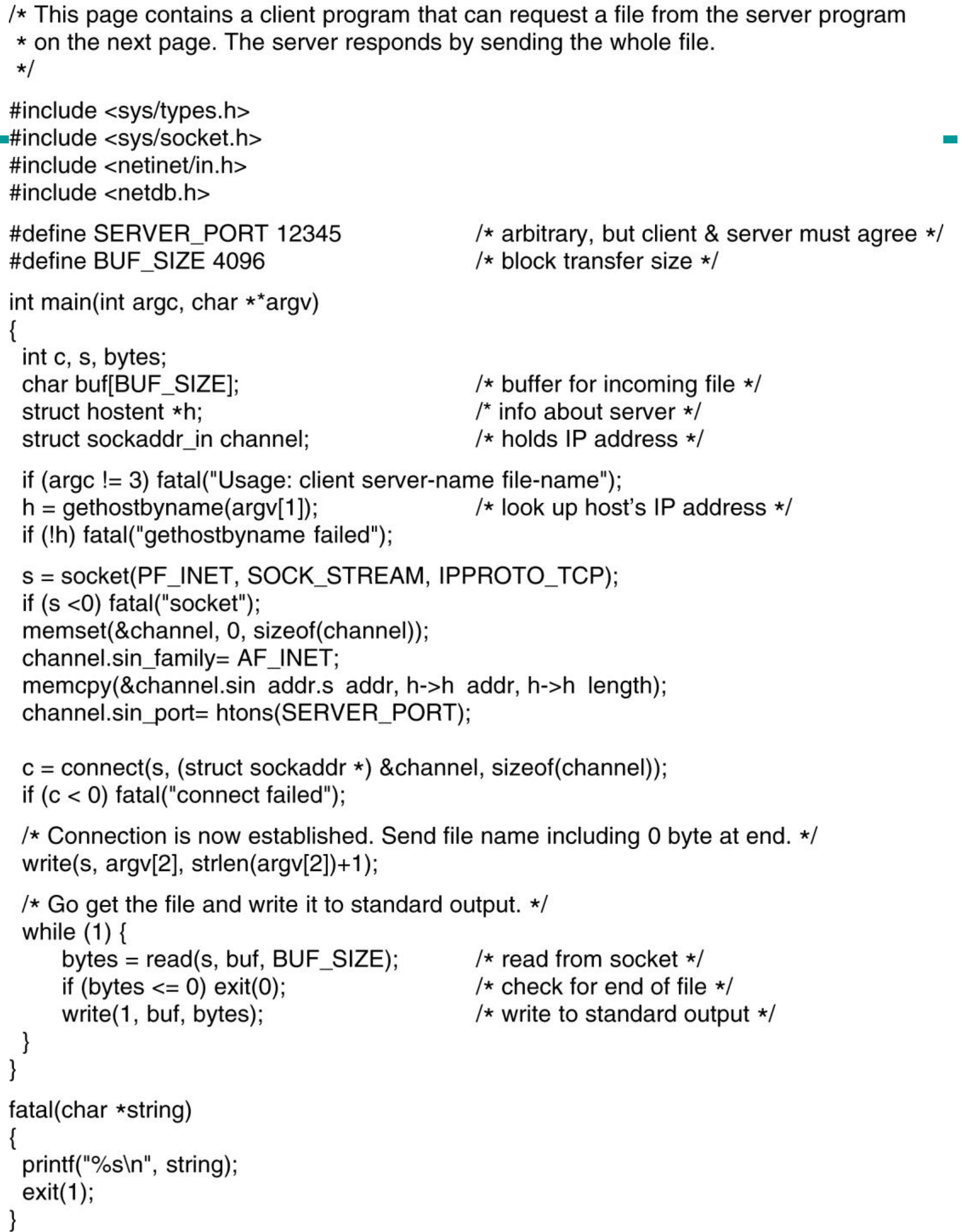
UDP Client

Sequence of calls

socket_init()	Create socket
sendto/recfrom	send/receive data
close()	Shutdown

→ No Connect
Part

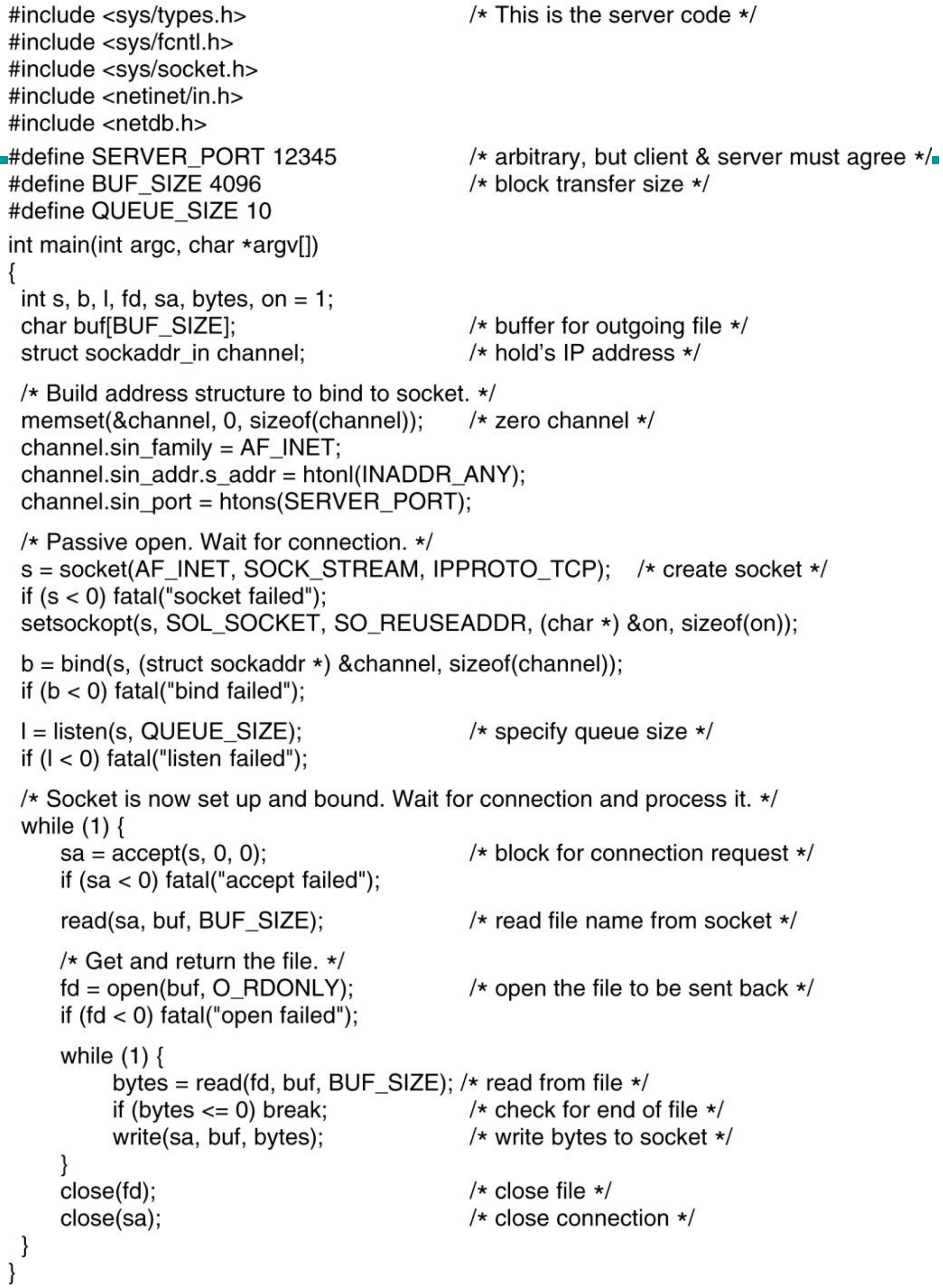
Socket Programming Example: Internet File Server



```
/* This page contains a client program that can request a file from the server program
 * on the next page. The server responds by sending the whole file.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345                 /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096                     /* block transfer size */

int main(int argc, char **argv)
{
  int c, s, bytes;
  char buf[BUF_SIZE];                     /* buffer for incoming file */
  struct hostent *h;                      /* info about server */
  struct sockaddr_in channel;             /* holds IP address */

  if (argc != 3) fatal("Usage: client server-name file-name");
  h = gethostbyname(argv[1]);             /* look up host's IP address */
  if (!h) fatal("gethostbyname failed");

  s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (s <0) fatal("socket");
  memset(&channel, 0, sizeof(channel));
  channel.sin_family= AF_INET;
  memcpy(&channel.sin  addr.s  addr, h->h  addr, h->h  length);
  channel.sin_port= htons(SERVER_PORT);

  c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
  if (c < 0) fatal("connect failed");

  /* Connection is now established. Send file name including 0 byte at end. */
  write(s, argv[2], strlen(argv[2])+1);

  /* Go get the file and write it to standard output. */
  while (1) {
       bytes = read(s, buf, BUF_SIZE);    /* read from socket */
       if (bytes <= 0) exit(0);           /* check for end of file */
       write(1, buf, bytes);              /* write to standard output */
  }
}

fatal(char *string)
{
  printf("%s\n", string);
  exit(1);
}
```

Client code using sockets.

Socket Programming Example: Internet File Server (2)



```
#include <sys/types.h>                       /* This is the server code */
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345                    /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096                        /* block transfer size */
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{
 int s, b, l, fd, sa, bytes, on = 1;
 char buf[BUF_SIZE];                         /* buffer for outgoing file */
 struct sockaddr_in channel;                 /* hold's IP address */

 /* Build address structure to bind to socket. */
 memset(&channel, 0, sizeof(channel));       /* zero channel */
 channel.sin_family = AF_INET;
 channel.sin_addr.s_addr = htonl(INADDR_ANY);
 channel.sin_port = htons(SERVER_PORT);

 /* Passive open. Wait for connection. */
 s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);    /* create socket */
 if (s < 0) fatal("socket failed");
 setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));

 b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
 if (b < 0) fatal("bind failed");

 l = listen(s, QUEUE_SIZE);                  /* specify queue size */
 if (l < 0) fatal("listen failed");

 /* Socket is now set up and bound. Wait for connection and process it. */
 while (1) {
     sa = accept(s, 0, 0);                   /* block for connection request */
     if (sa < 0) fatal("accept failed");

     read(sa, buf, BUF_SIZE);                /* read file name from socket */

     /* Get and return the file. */
     fd = open(buf, O_RDONLY);               /* open the file to be sent back */
     if (fd < 0) fatal("open failed");

     while (1) {
          bytes = read(fd, buf, BUF_SIZE); /* read from file */
          if (bytes <= 0) break;             /* check for end of file */
          write(sa, buf, bytes);             /* write bytes to socket */
     }
     close(fd);                              /* close file */
     close(sa);                              /* close connection */
 }
}
```

Server code using sockets.

PART C: TCP

Introduction

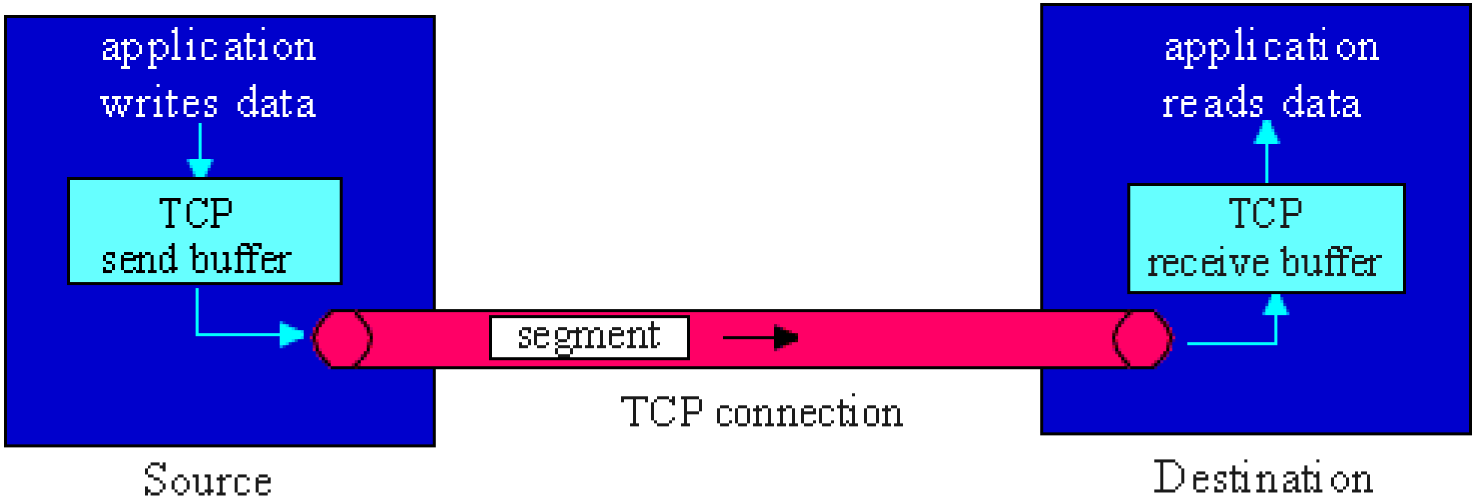
- **connection oriented.**
- **full duplex.**

TCP Services: → Description

- **Reliable transport** — 29 no slide
- **Flow control**
- **Congestion control**

UDP does not provide any of these services

Flow of TCP Segments



TCP Services (1)

TCP converts the unreliable, best effort service of IP into a reliable service, i.e., it ensures that each segment is delivered correctly, only once, and in order.

Converting an unreliable connection into a reliable connection is basically the same problem we have considered at the data link layer, and essentially the same solution is used:

→TCP numbers each segment and uses an ARQ protocol to recover lost segments.

(Automatic Repeat Req)

→Some versions of TCP implement Go Back N and other versions implement Selective Repeat.

TCP Services (2)

However, there are a few important differences between the transport layer and the data link layer.

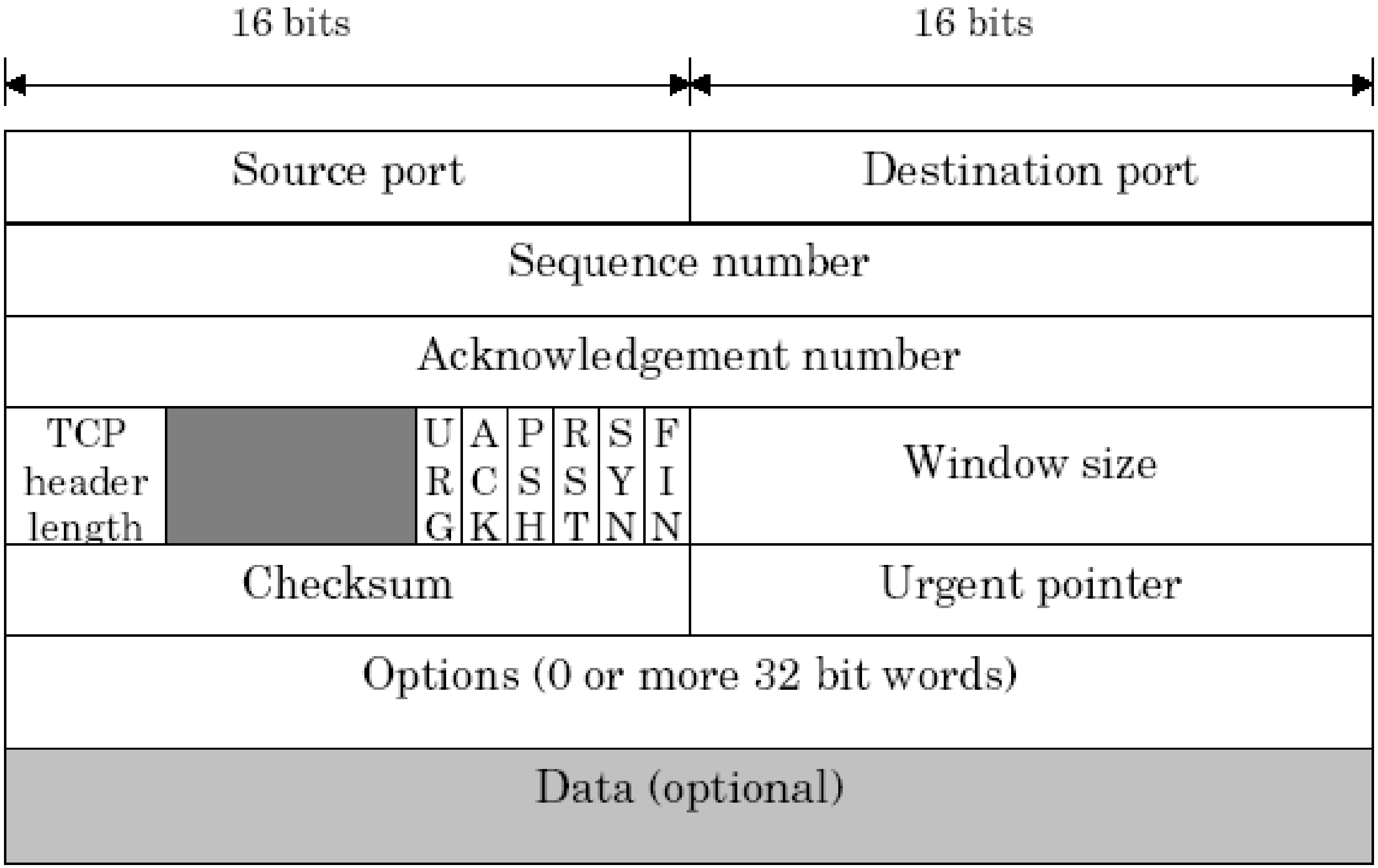
→ At the data link layer, we viewed an ARQ protocol as being operated between two nodes connected by a point-to-point link.

→ At the transport layer, this protocol is implemented between two hosts connected over network.

→ Packets can arrive out-of-order, and packets may also be stored in buffers within the network and then arrive at much later times.

→ The round-trip time will change with different connections and connection establishment is more complicated.

TCP Header



16 bits

16 bits

Source port

Destination port

Sequence number

Acknowledgement number

TCP header length

URG

ACK

PSH

RST

SYN

FIN

Window size

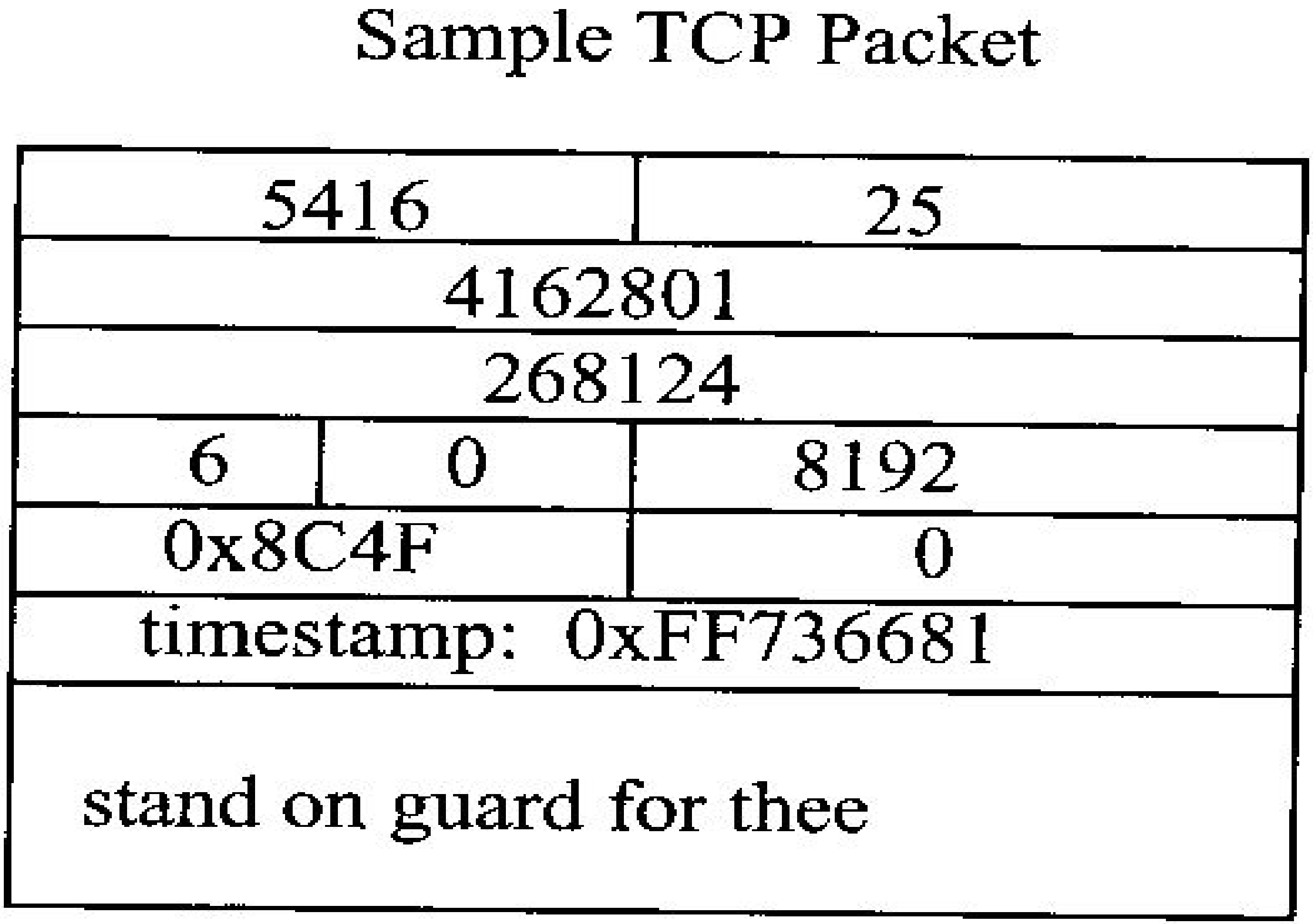
Checksum

Urgent pointer

Options (0 or more 32 bit words)

Data (optional)

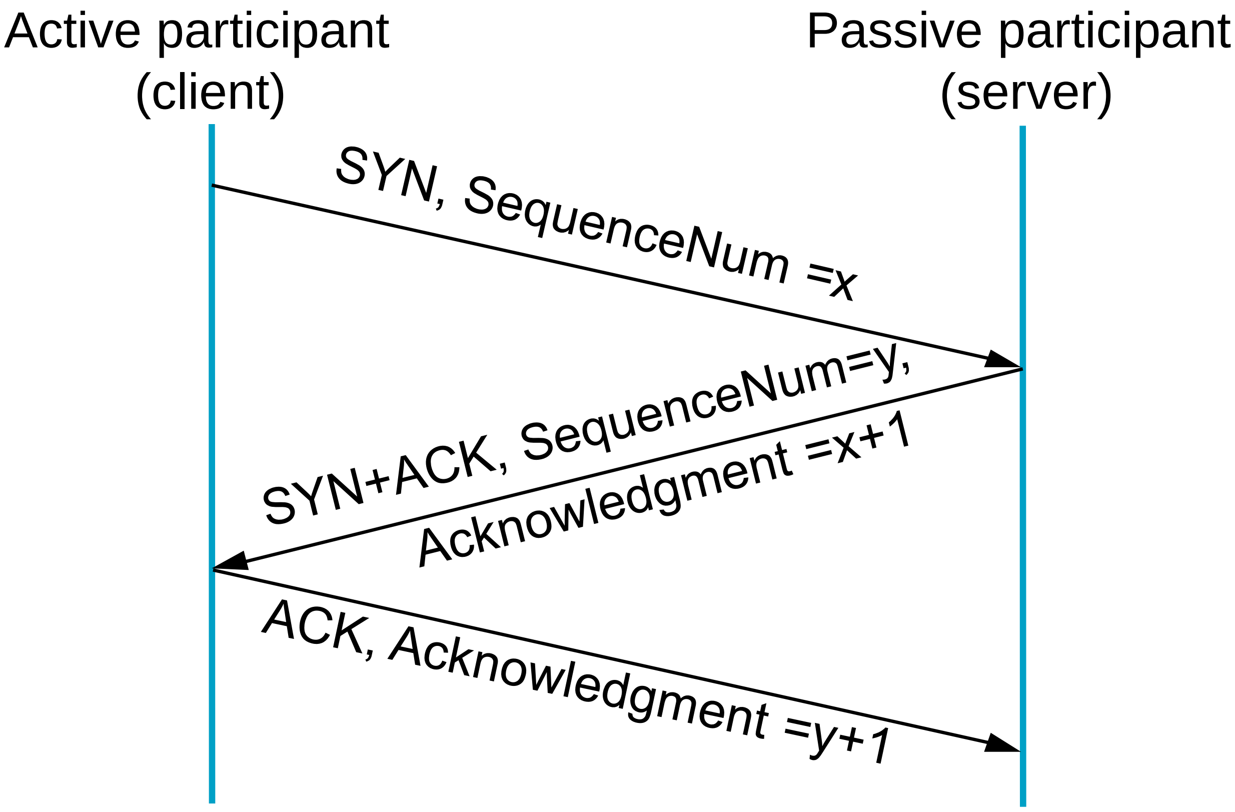
Sample TCP Packet



Sample TCP Packet

<table>
<tr><td colspan="2">5416</td><td>25</td></tr>
<tr><td colspan="3">4162801</td></tr>
<tr><td colspan="3">268124</td></tr>
<tr><td>6</td><td>0</td><td>8192</td></tr>
<tr><td colspan="2">0x8C4F</td><td>0</td></tr>
<tr><td colspan="3">timestamp: 0xFF736681</td></tr>
<tr><td colspan="3">stand on guard for thee</td></tr>
</table>

TCP Connection Establishment

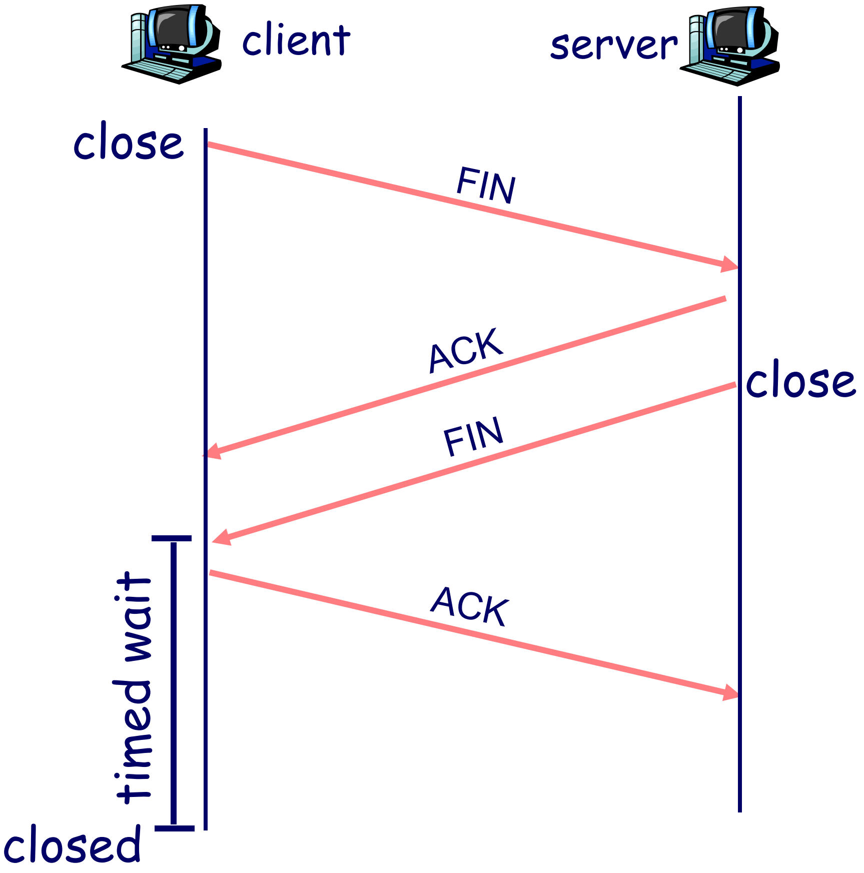


Closing a TCP Connection (1)

client closes socket:
`clientSocket.close();`

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



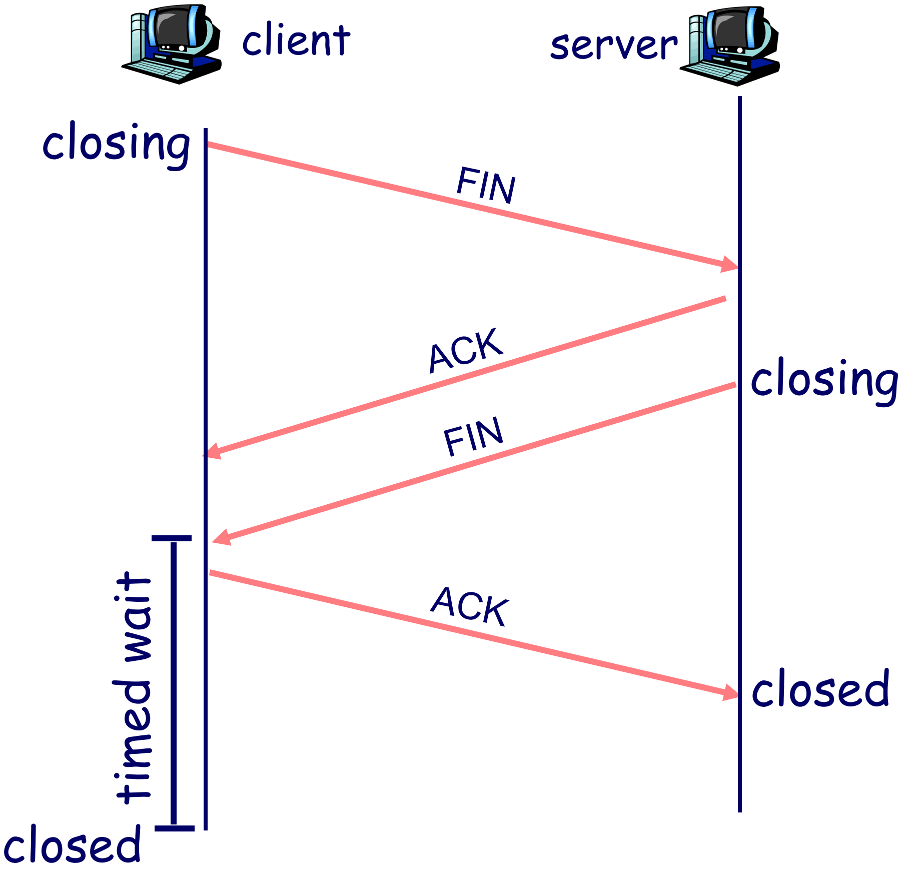
Closing a TCP Connection (2)

Step 3: client receives FIN, replies with ACK.

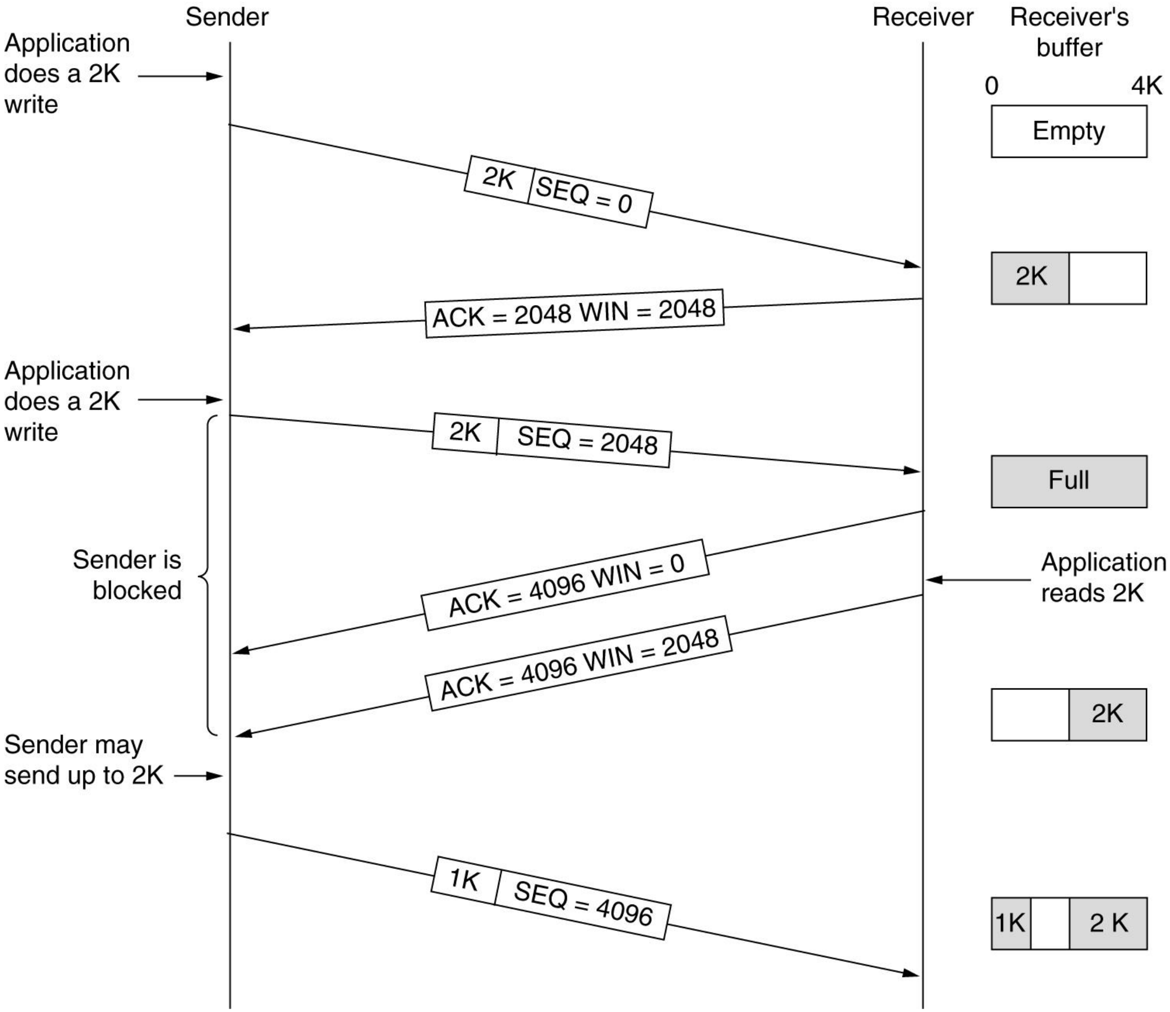
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



TCP Transmission Policy



Window management in TCP.

Flow Control and Congestion Control

- In a network, it is often desirable to limit the rate at which a source can send traffic into the subnet.

- If this is not done and sources send at too high of a rate, then buffers within the network will fill-up resulting in long delays and eventually packets being dropped.

- Moreover as packets gets dropped, retransmission may occur, leading to even more traffic.

- When sources are not regulated this can lead to ***congestion collapse*** of the network, where very little traffic is delivered to the destination.

Flow Control and Congestion Control

Two different factors can limit the rate at which a source sends data.

- inability of the destination to accept new data. Techniques that address this: ***flow control***.
- number of packets within the subnet. Techniques that address this: ***congestion control***.

Flow Control and Congestion Control

- Flow control and congestion control can be addressed at the transport layer, but may also be addressed at other layers.

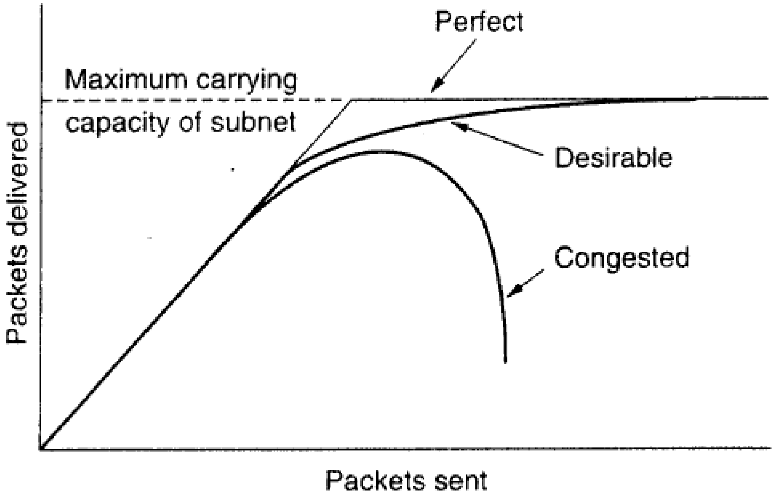
- For example, some DLL protocols perform flow control on each link. And some congestion control approaches are done at the network layer.

- Both flow control and congestion control are part of TCP.

Congestion

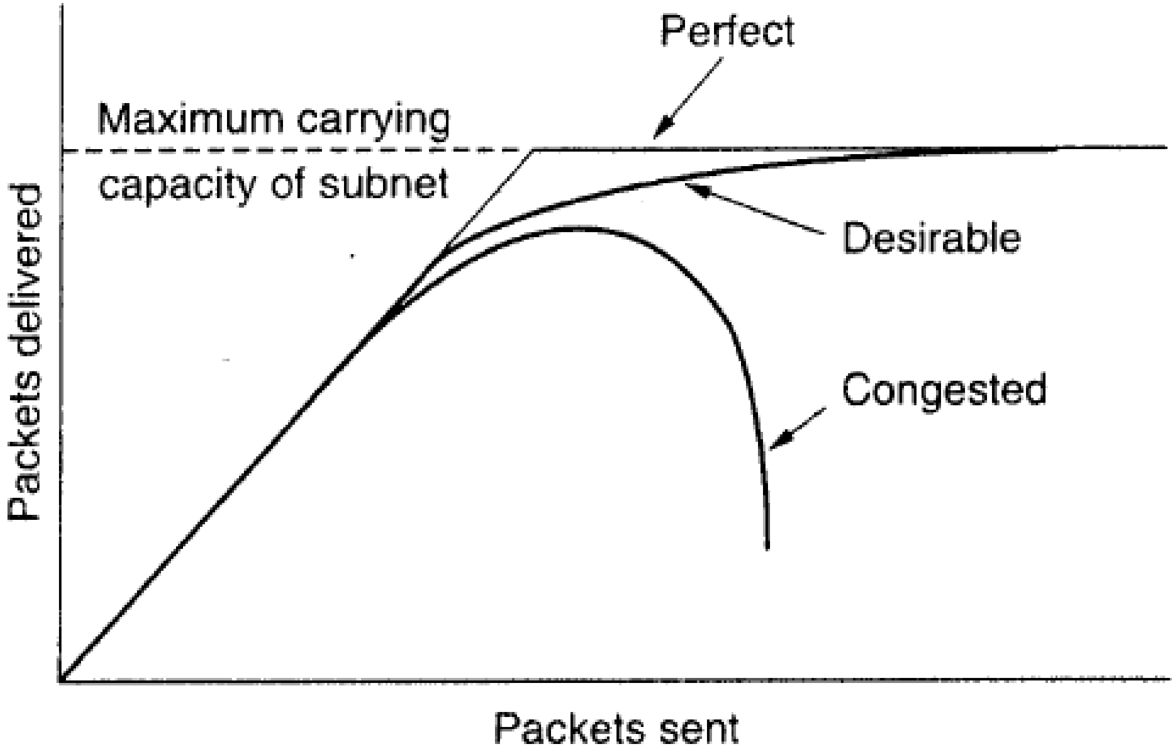
Congestion arises when the total load on the network becomes too large. This leads to queues building up and to long delays

If sources retransmit messages, then this can lead to even more congestion and eventually to ***congestion collapse***.
Notice, as the offered load increase, the number of packets delivered at first increases, but at high enough loads, this rapidly decreases.



Congestion

Notice, as the offered load increase, the number of packets delivered at first increases, but at high enough loads, this rapidly decreases.



Approaches to Congestion Control

Congestion control may be addressed at both the network level and the transport layer.

At the network layer possible approaches include:

Packet dropping → when a buffer becomes full a router can drop waiting packets - if not coupled with some other technique, this can lead to greater congestion through retransmissions.

Packet scheduling → certain scheduling policies may help in avoiding congestion - in particular scheduling can help to isolate users that are transmitting at a high rate.

Approaches to Congestion Control

Dynamic routing → when a link becomes congested, change the routing to avoid this link - this only helps up to a point (eventually all links become congested) and can lead to instabilities

Admission control/Traffic policing - Only allow connections in if the network can handle them and make sure that admitted sessions do not send at too high of a rate - only useful for connection-oriented networks.

Approaches to Congestion Control

An approach that can be used at either the network or transport layers is

Rate control → this refers to techniques where the source rate is explicitly controlled based on feedback from either the network and/or the receiver.

For example, routers in the network may send a source a "choke packet" upon becoming congested. When receiving such a packet, the source should lower it rate.

Approaches to Congestion Control

These approaches can be classified as either "congestion avoidance" approaches, if they try to prevent congestion from ever occurring, or as "congestion recovery" approaches, if they wait until congestion occurs and then react to it. In general, “better to prevent than to recover."

Different networks have used various combinations of all these approaches.

Traditionally, rate control at the transport layer has been used in the Internet, but new approaches are beginning to be used that incorporate some of the network layer techniques discussed above.

V.V.I

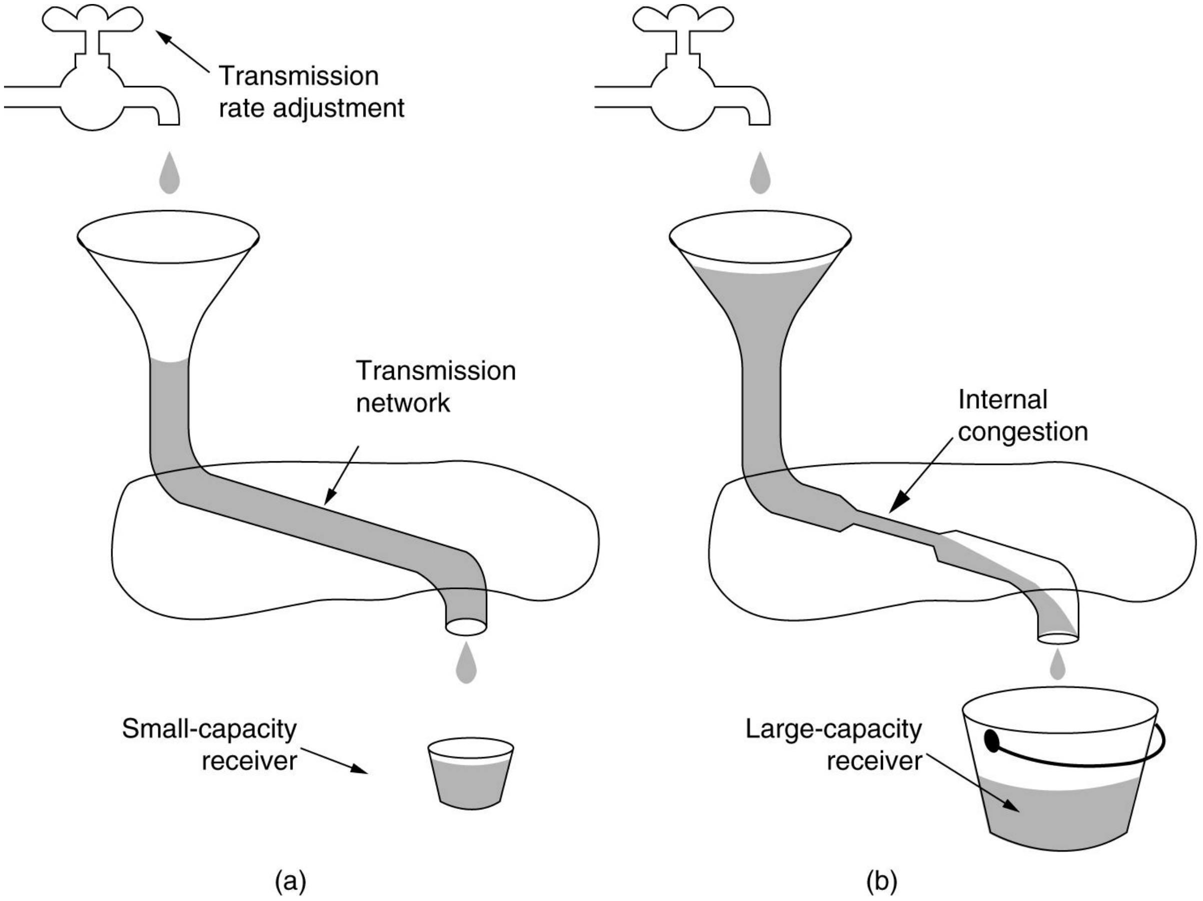
Congestion Control in TCP

TCP implements end-to-end congestion control. TCP detects congestion via the ACK's from the sliding-window ARQ algorithm used for providing reliable service.

When the source times out before receiving an ACK, the most likely reason is because a link became congested. TCP uses this as an indication of congestion. In this case, TCP will slow down the transmission rate.

TCP controls the transmission rate of a source by varying the window size used in the sliding window protocol.

TCP Flow control versus Congestion Control



Slow Start

Initial CW = 1.

After each ACK, CW += 1;

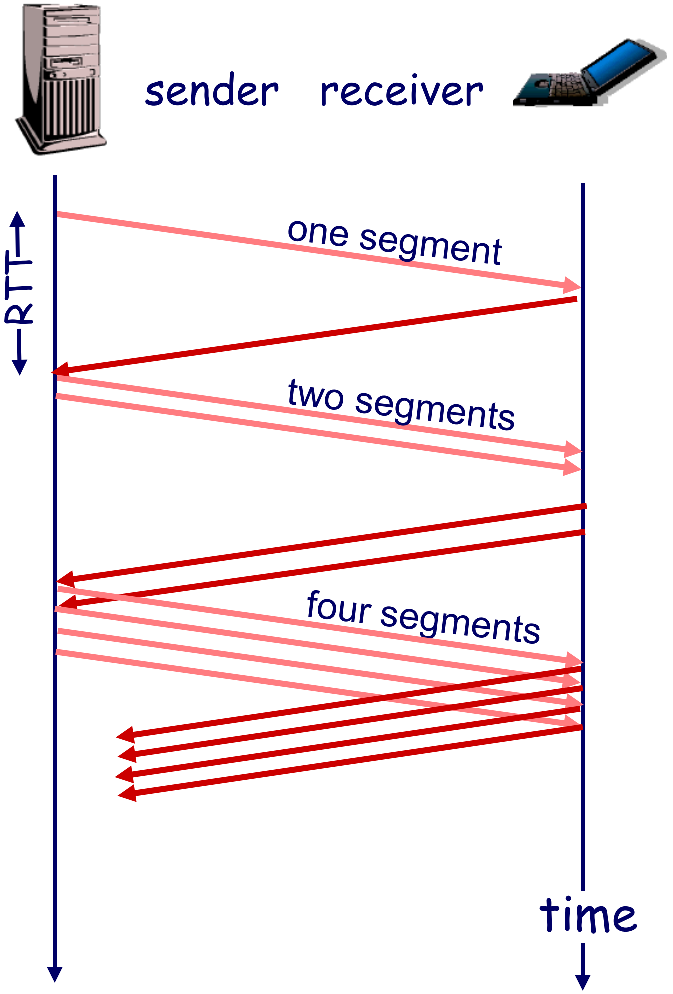
Continue until:

- Loss occurs OR
- CW > slow start threshold

Then switch to congestion avoidance

If we detect loss, cut CW in half

Exponential increase in window size per RTT

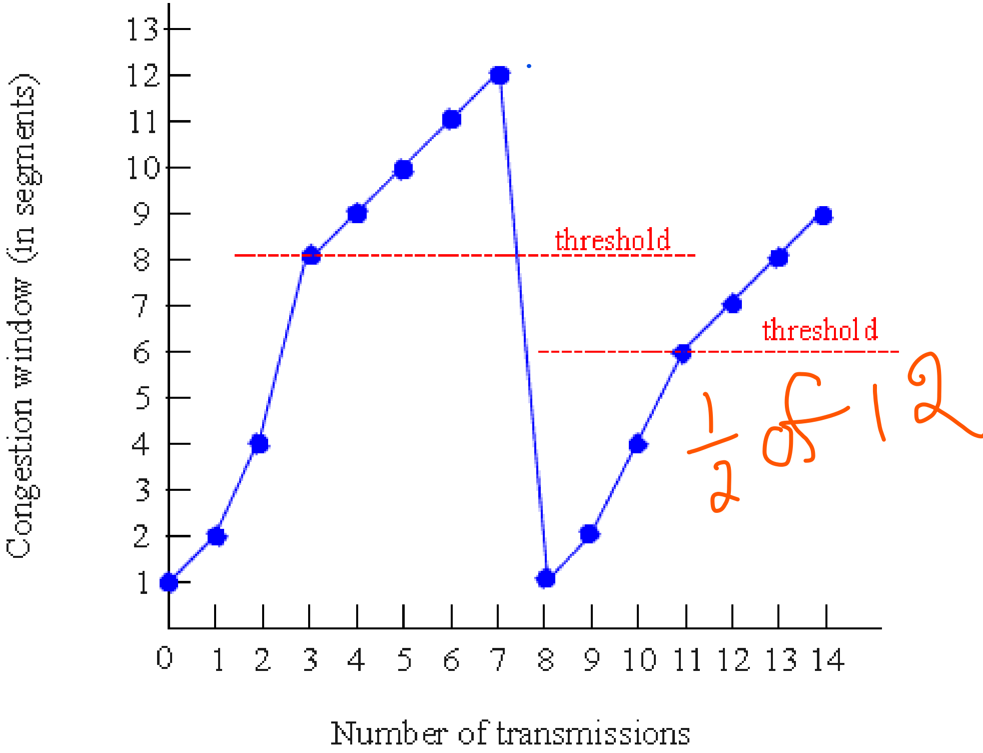


Congestion Avoidance

```
Until (loss) {
 after CW packets ACKed:
   CW += 1;
}
ssthresh = CW/2;
Depending on loss type:
  SACK/Fast Retransmit:
    CW/= 2; continue;
  Course grained timeout:
    CW = 1; go to slow start.
```

TCP Reno: CW = CW/2 after loss

TCP Tahoe: CW=1 after a loss



How are losses recovered?

Say packet is lost (data or ACK!)

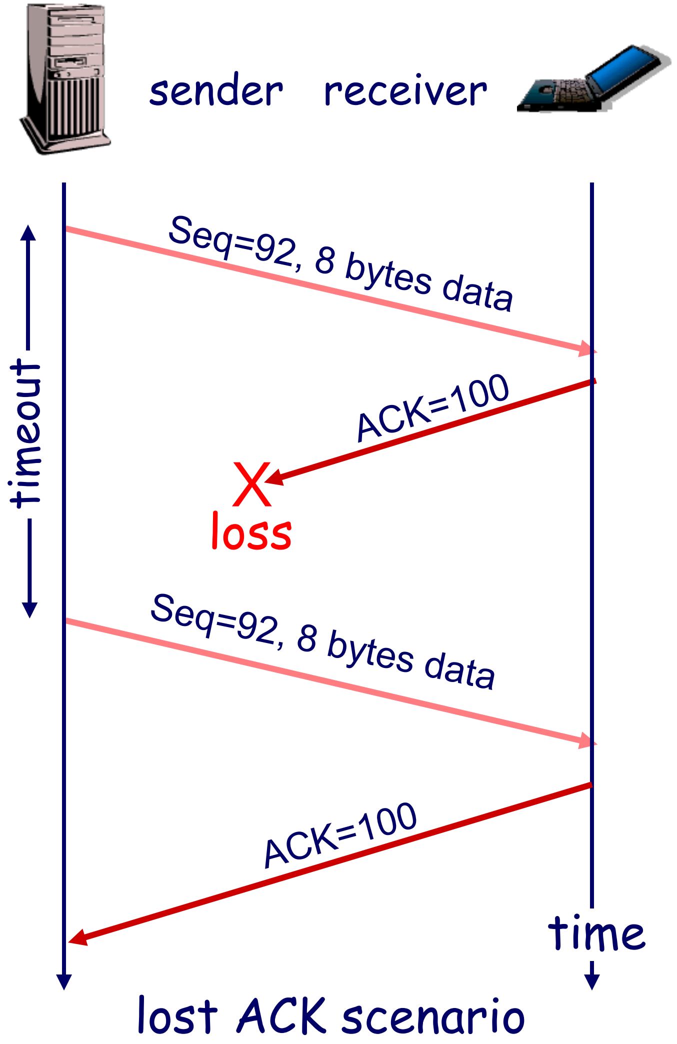
Coarse-grained Timeout:

- Sender does not receive ACK after some period of time
- Event is called a retransmission time-out (RTO)
- RTO value is based on estimated round-trip time (RTT)
- RTT is adjusted over time using exponential weighted moving average:

  RTT = (1-x)*RTT + (x)*sample

  (x is typically 0.1)

First done in TCP Tahoe



lost ACK scenario

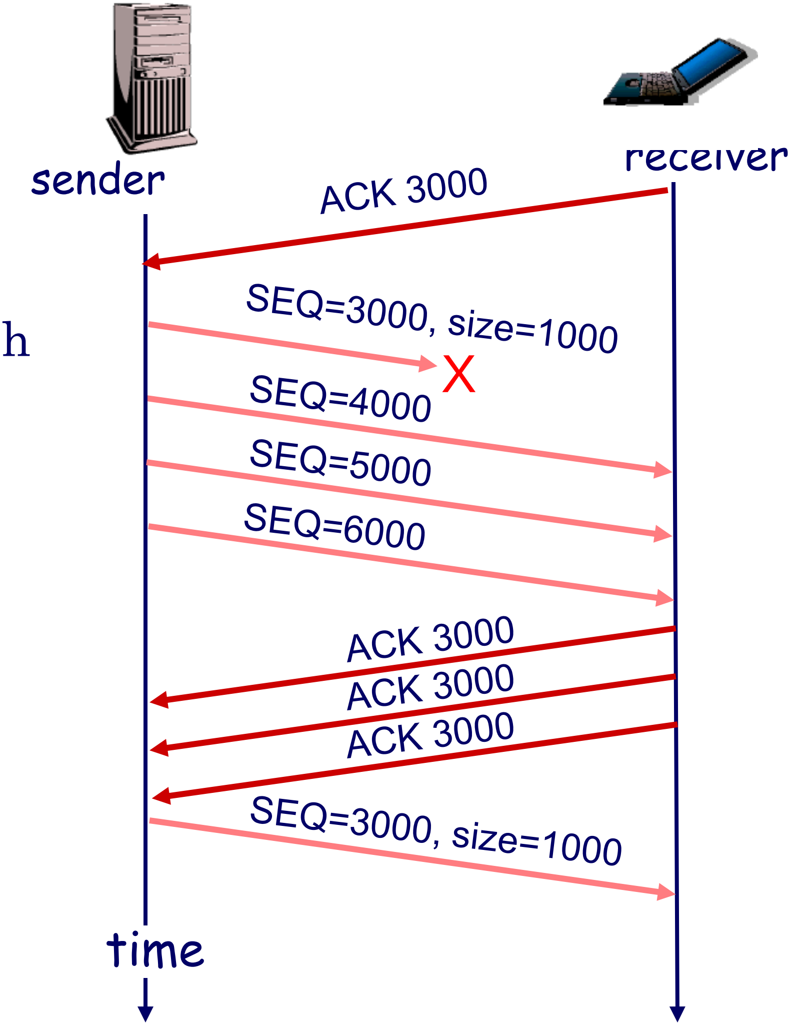
Fast Retransmit (TCP Reno)

Receiver expects N, gets N+1:

- Immediately sends ACK(N)
- This is called a **duplicate ACK**
- Does NOT delay ACKs here!
- Continue sending dup ACKs for each subsequent packet (not N)

Sender gets 3 duplicate ACKs:

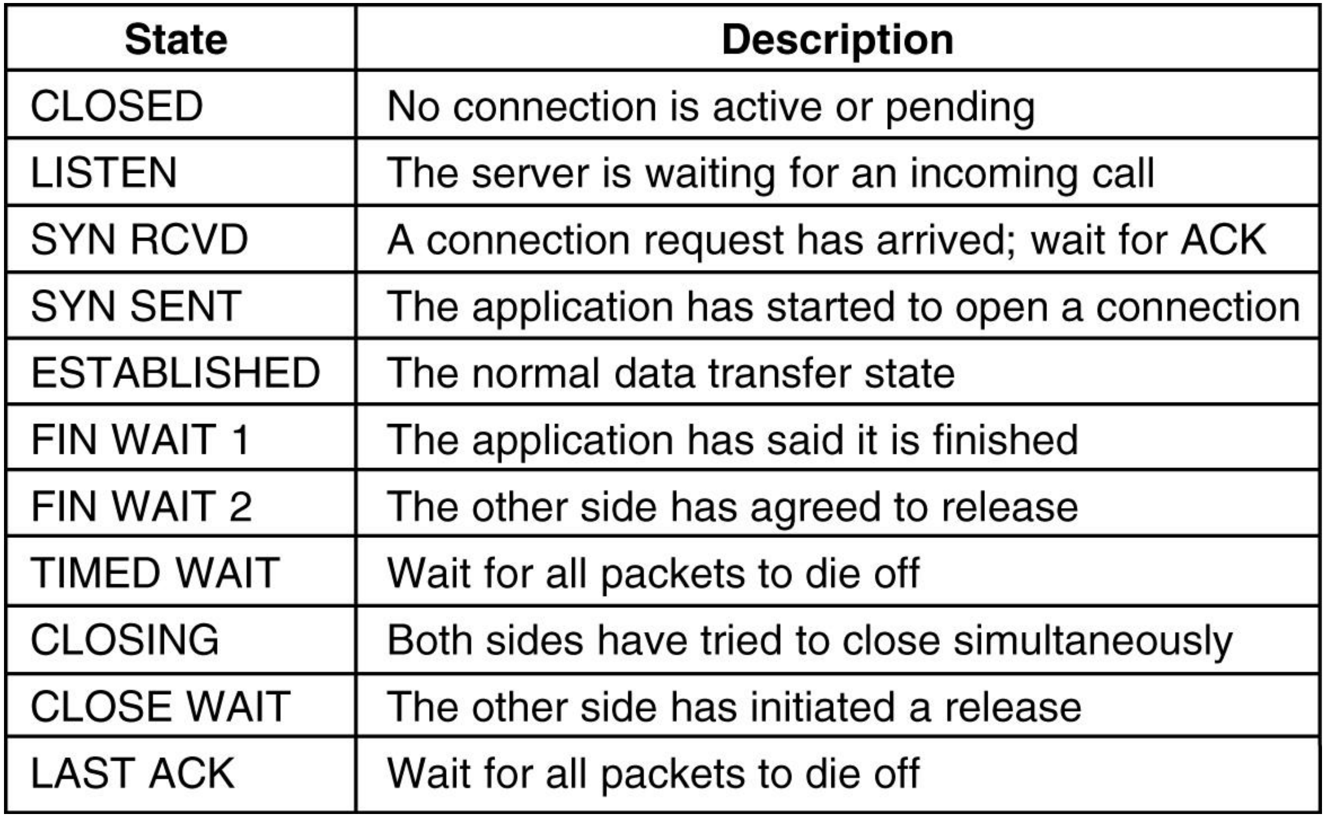
- Infers N is lost and resends
- 3 chosen so out-of-order packets don't trigger Fast Retransmit accidentally
- Called "fast" since we don't need to wait for a full RTT



Introduced in TCP Reno

TCP Connection Management Modeling

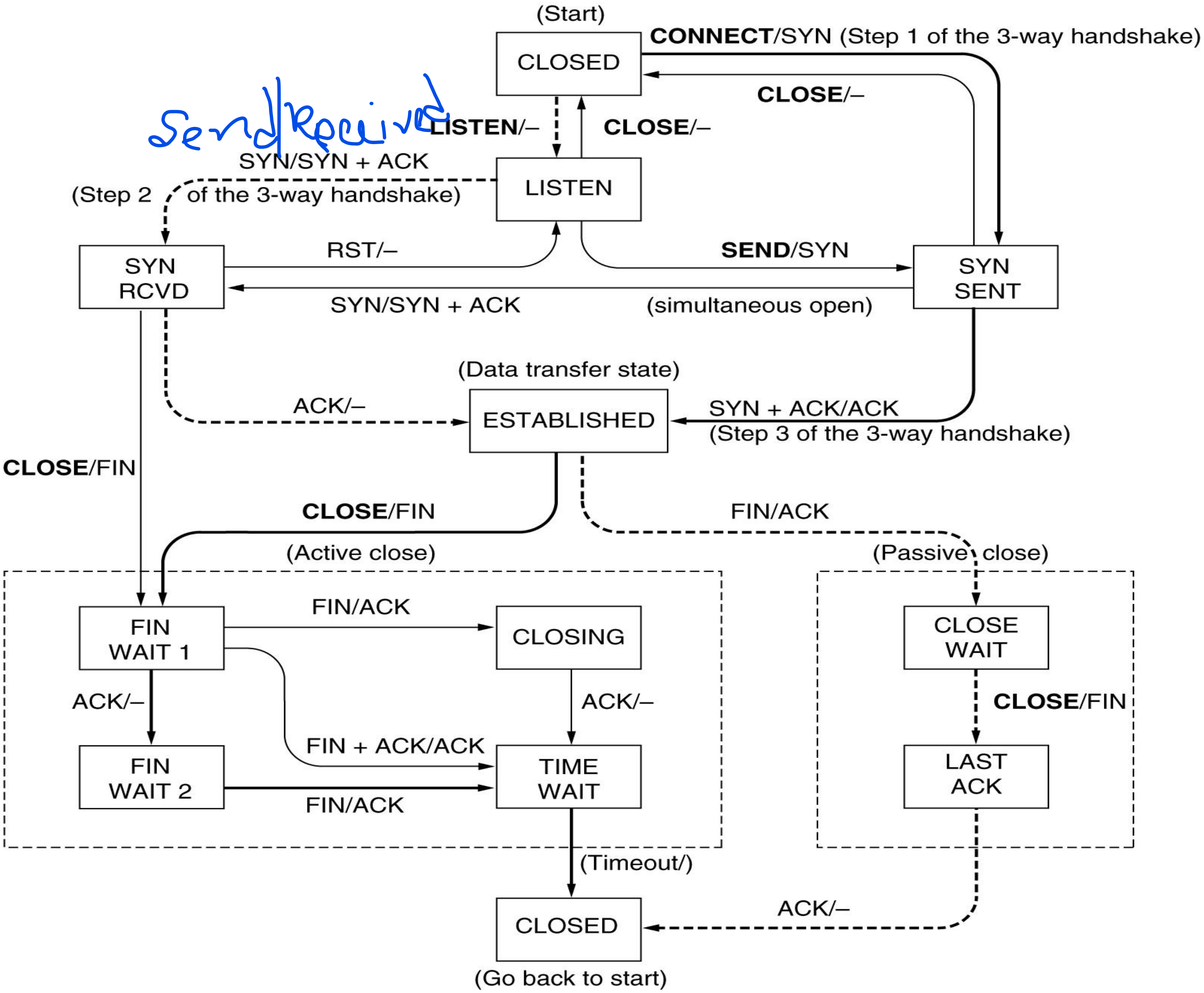
The states used in the TCP connection management finite state machine.



State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIMED WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

No Need to Memorize

TCP Connection Management Modeling (2)



PART D: UDP

- **Unreliable**
- **Connectionless**
- **No TCP's flow control;**
- **Applications where prompt delivery more important than accurate delivery (speech, video, …)**

UDP

Datagram protocol → it does not have to establish a connection to another machine before sending data.

1. Takes the data an application provides
2. Packs it into a UDP packet
3. Hands it to the IP layer.
4. The packet is then put on the wire, and that is where it ends.

There is no way to guarantee that the packet will reach its destination.

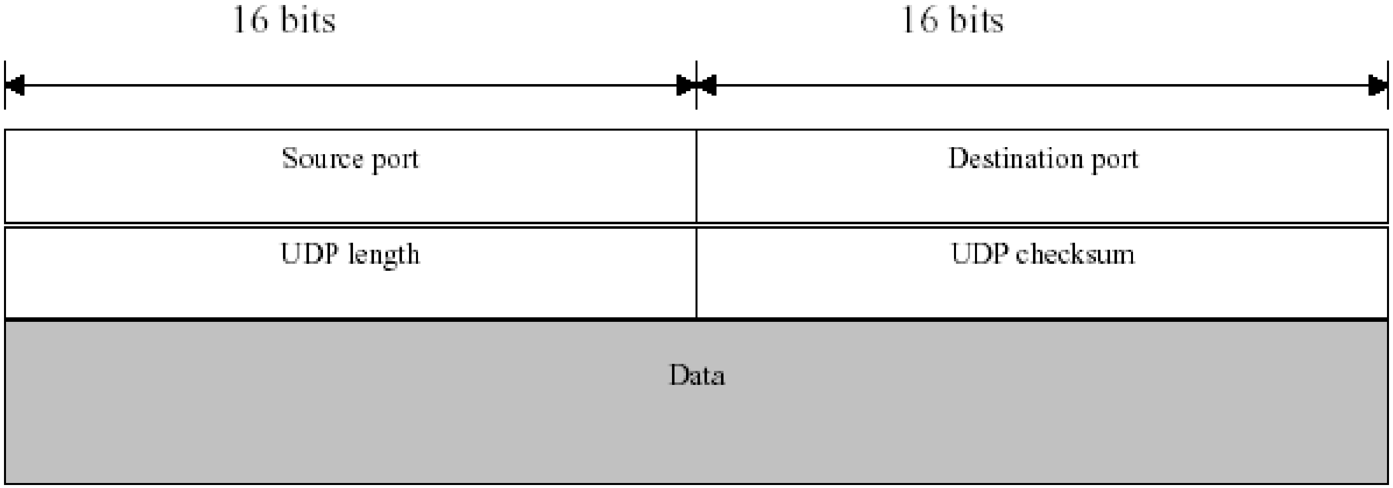
UDP Packet

Basically all UDP provides is a multiplexing capability on top of IP.

The length field gives the length of the entire packet including the header.

The checksum is calculated on the entire packet (and also on part of the IP header).

Calculating the checksum is optional, and if an error is detected the packet may be discarded or may be passed on to the application with an error flag.


16 bits
16 bits
Source port
Destination port
UDP length
UDP checksum
Data