

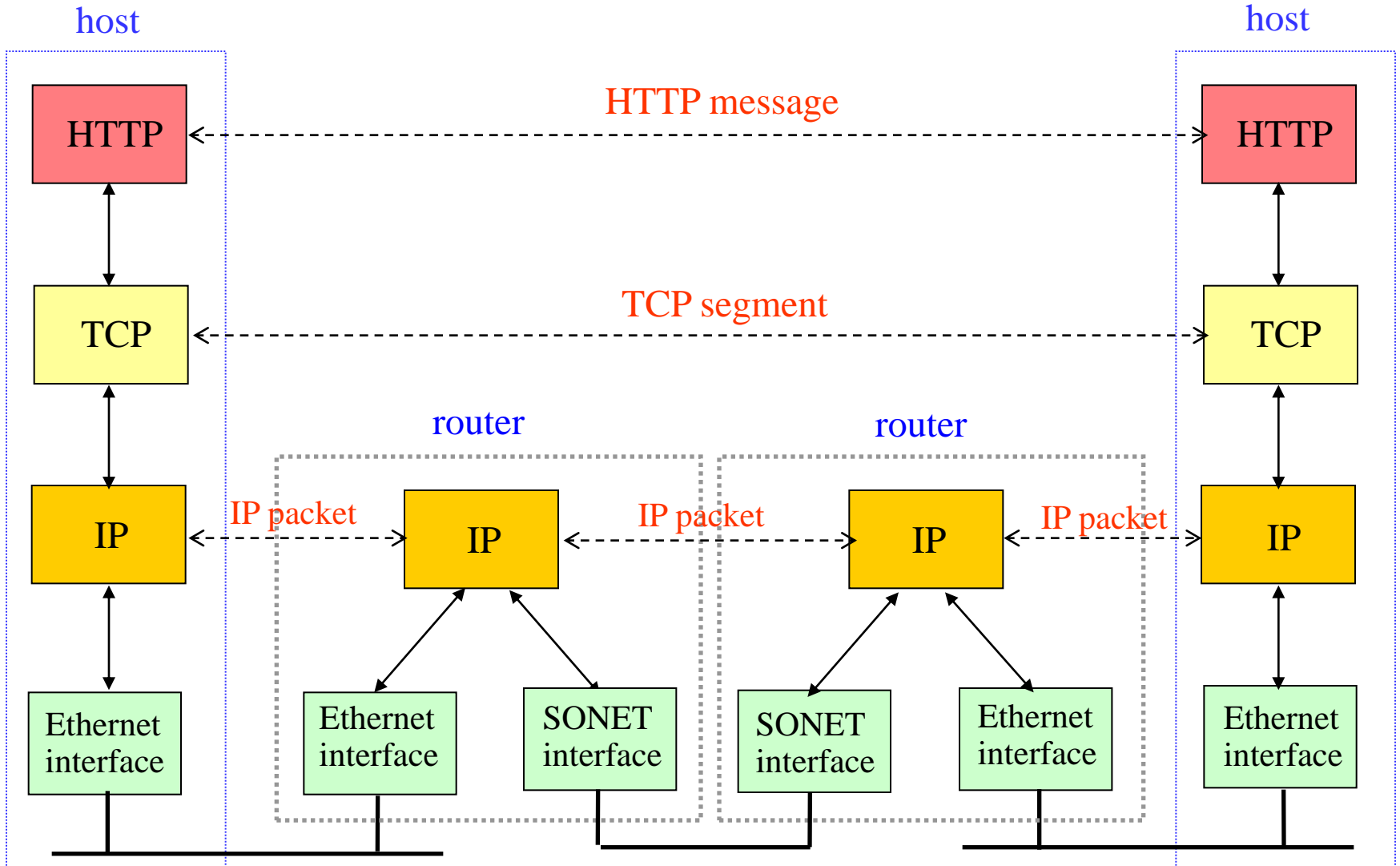
# CMSC 417

## Computer Networks Prof. Ashok K Agrawala

© 2011 Ashok Agrawala  
Set 7

# The Transport Layer

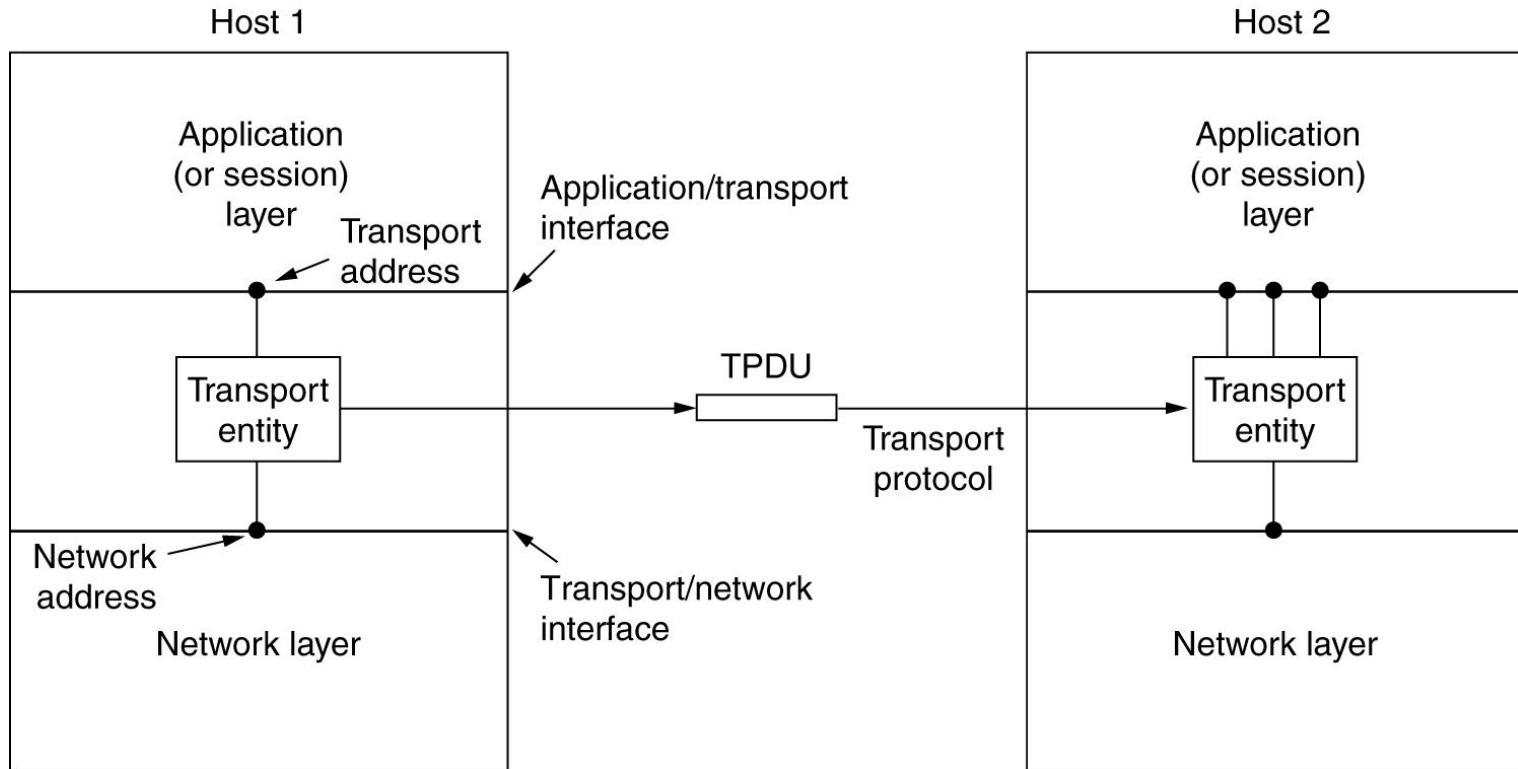
# Message, Segment, Packet, and Frame



# The Transport Service

- Services Provided to the Upper Layers
- Transport Service Primitives
- Berkeley Sockets
- An Example of Socket Programming:
  - An Internet File Server

# Services Provided to the Upper Layers



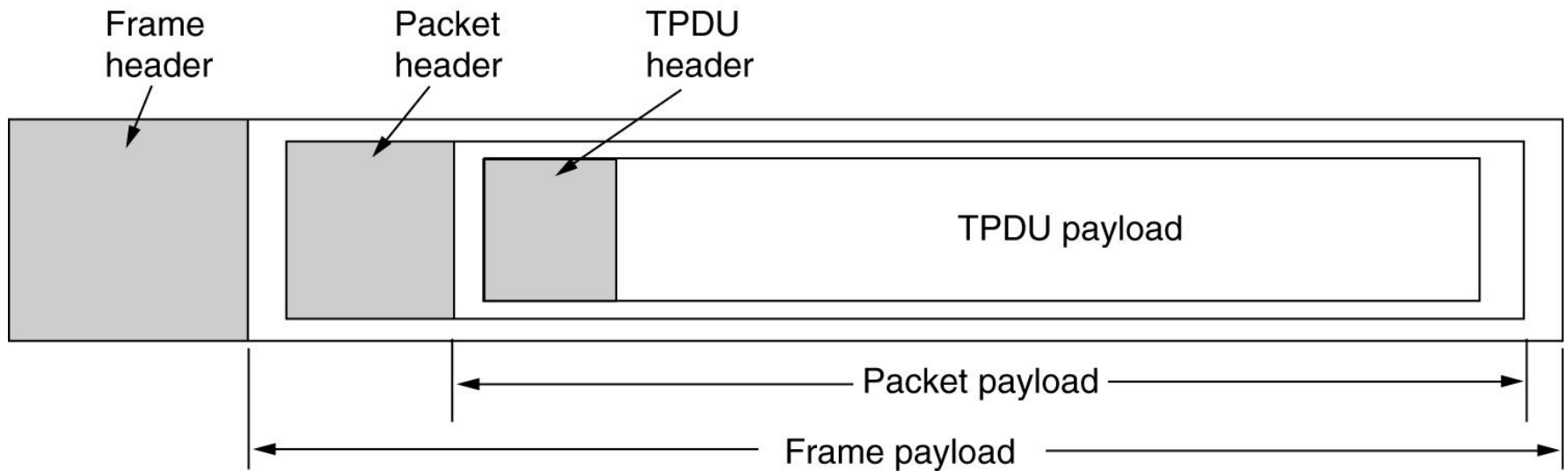
# Transport Service Primitives

<b>Primitive</b>	<b>Packet sent</b>	<b>Meaning</b>
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

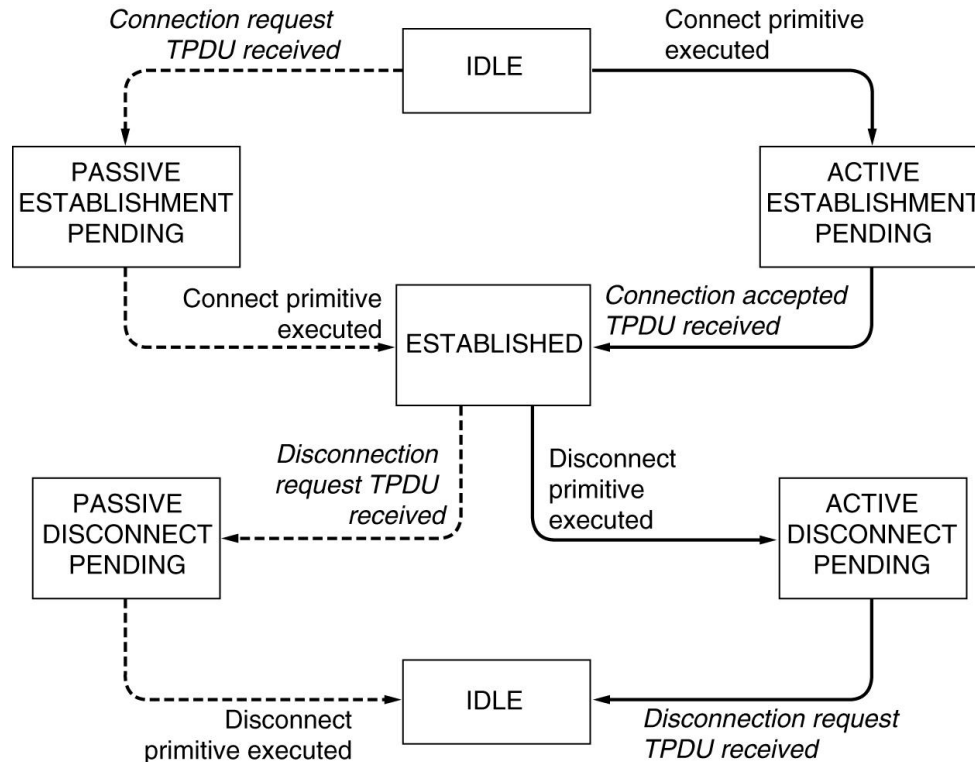
The primitives for a simple transport service.

# Transport Service Primitives (2)

The nesting of TPDU, packets, and frames.



# Transport Service Primitives (3)



A state diagram for a simple connection management scheme. Transitions labeled in italics are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.

# Berkeley Sockets

The socket primitives for TCP.

<b>Primitive</b>	<b>Meaning</b>
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
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

# Example of Socket Programming: An Internet File Server (1)

```
/* 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 */

    . . .
```

Client code using sockets

# Example of Socket Programming: An Internet File Server (2)

. . .

```
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");
```

. . .

Client code using sockets

# Example of Socket Programming: An Internet File Server (3)

...

```
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

# Example of Socket Programming: An Internet File Server (4)

```
#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;        /* holds IP address */
```

. . .

Server code

# Example of Socket Programming: An Internet File Server (5)

...

```
/* 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");
```

...

Server code

# Example of Socket Programming: An Internet File Server (6)

...

```
/* 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");

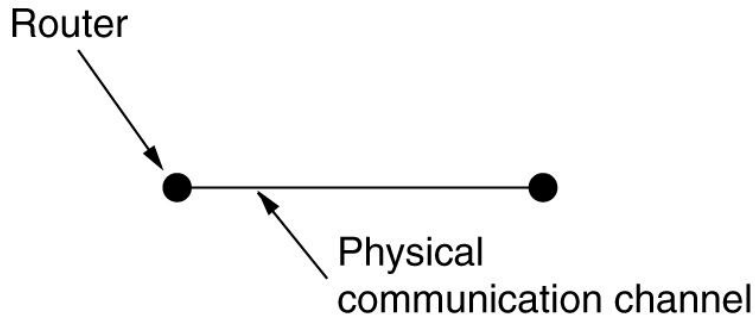
    while (1) {
        bytes = read(fd, buf, BUF_SIZE); /* read from file */
        if (bytes <= 0) break;           /* check for end of file */
        write(sa, buf, bytes);           /* write bytes to socket */
    }
    close(fd);                           /* close file */
    close(sa);          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

Server code

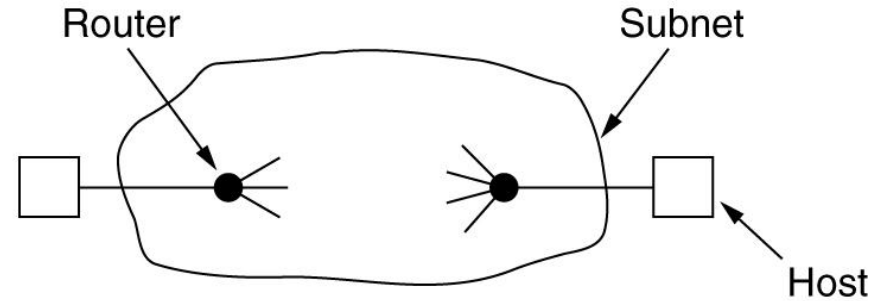
# Elements of Transport Protocols

- Addressing
- Connection Establishment
- Connection Release
- Flow Control and Buffering
- Multiplexing
- Crash Recovery

# Transport Protocol



(a)

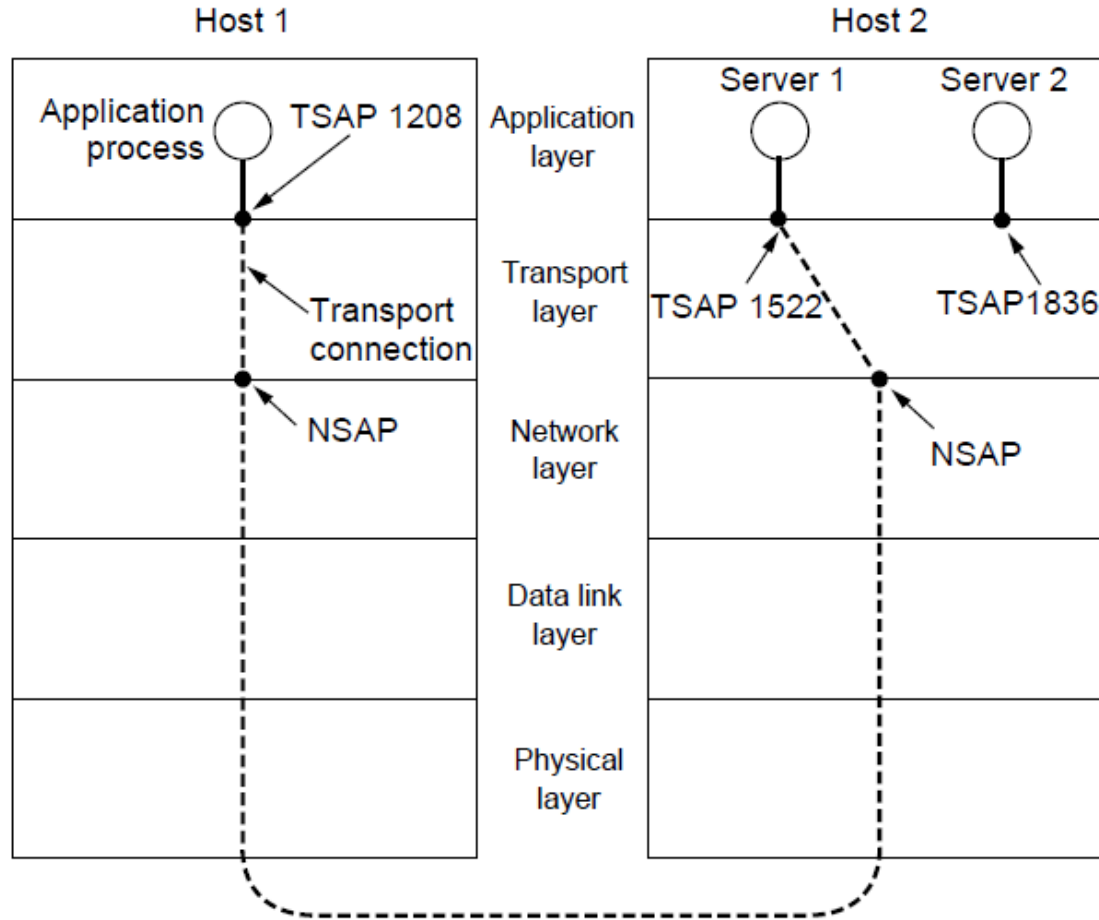


(b)

- (a) Environment of the data link layer.
- (b) Environment of the transport layer.

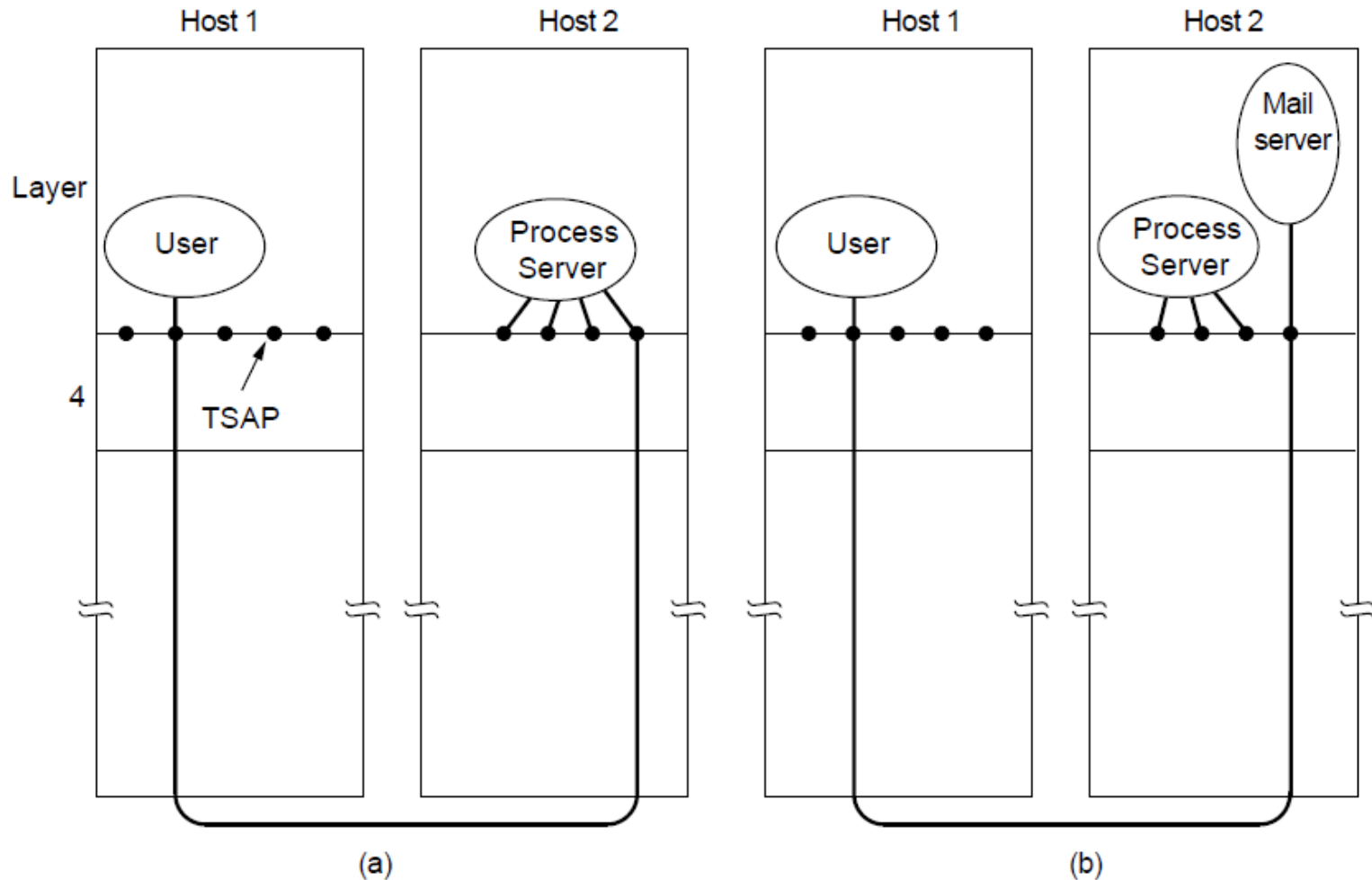
# Addressing (1)

TS,



ons

# Addressing (2)



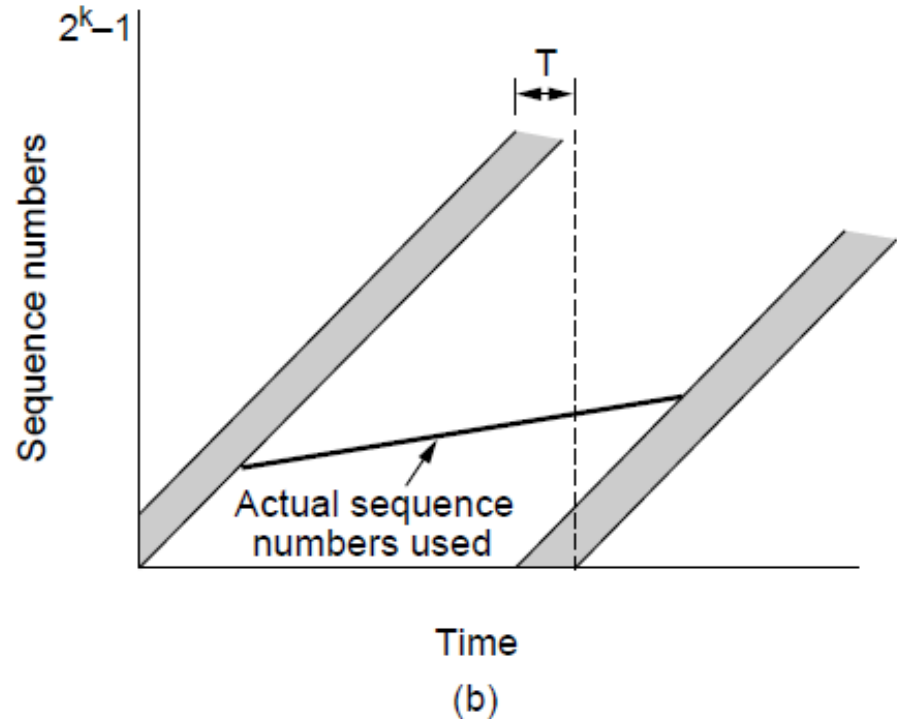
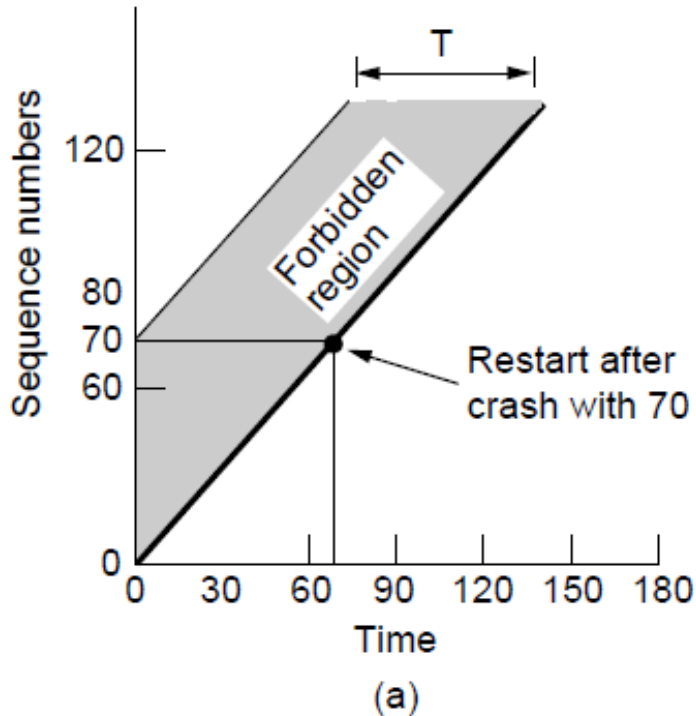
How a user process in host 1 establishes a connection with a mail server in host 2 via a process server.

# Connection Establishment (1)

Techniques for restricting packet lifetime

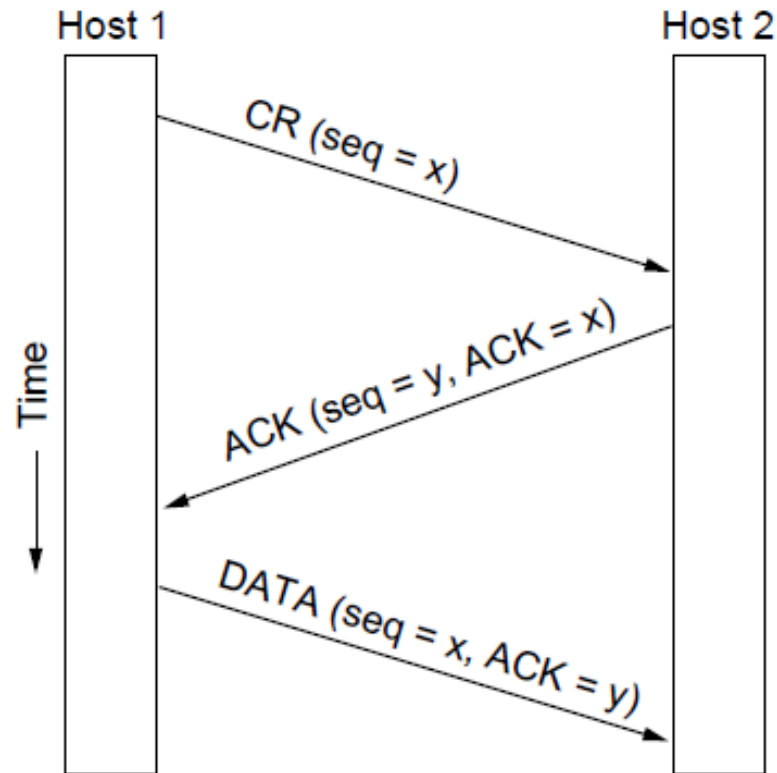
- Restricted network design.
- Putting a hop counter in each packet.
- Timestamping each packet.

# Connection Establishment (2)



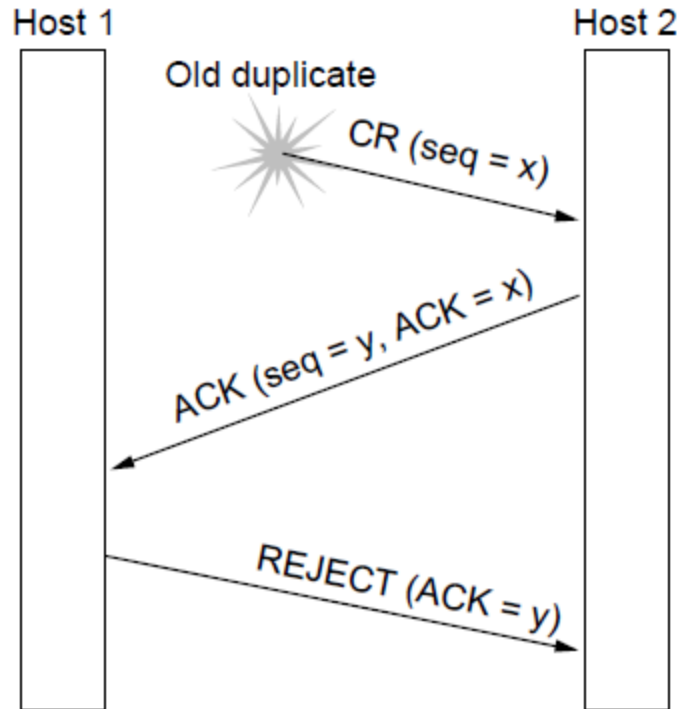
- (a) TPDUs may not enter the forbidden region.
- (b) The resynchronization problem.

# Connection Establishment (3)



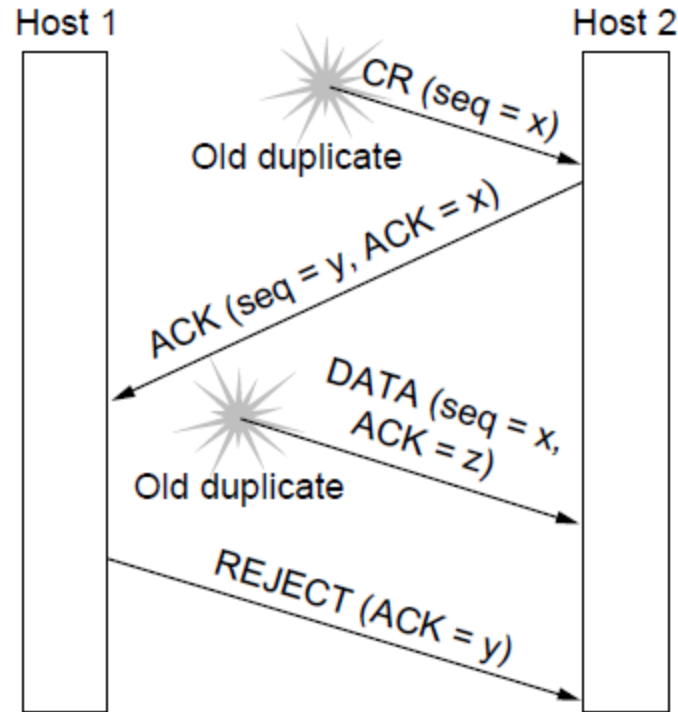
Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. Normal operation.

# Connection Establishment (4)



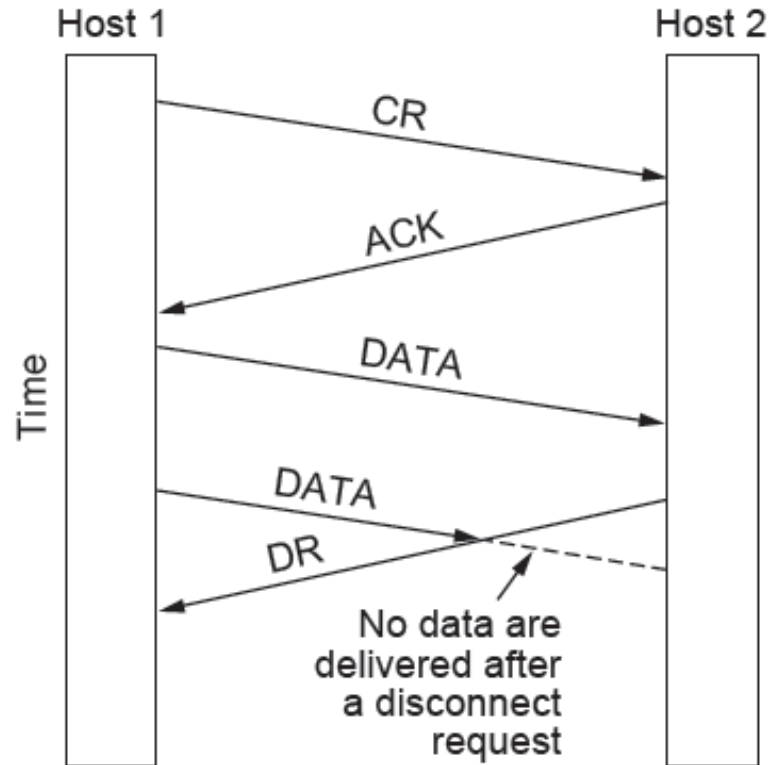
Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. Old duplicate CONNECTION REQUEST appearing out of nowhere.

# Connection Establishment (5)



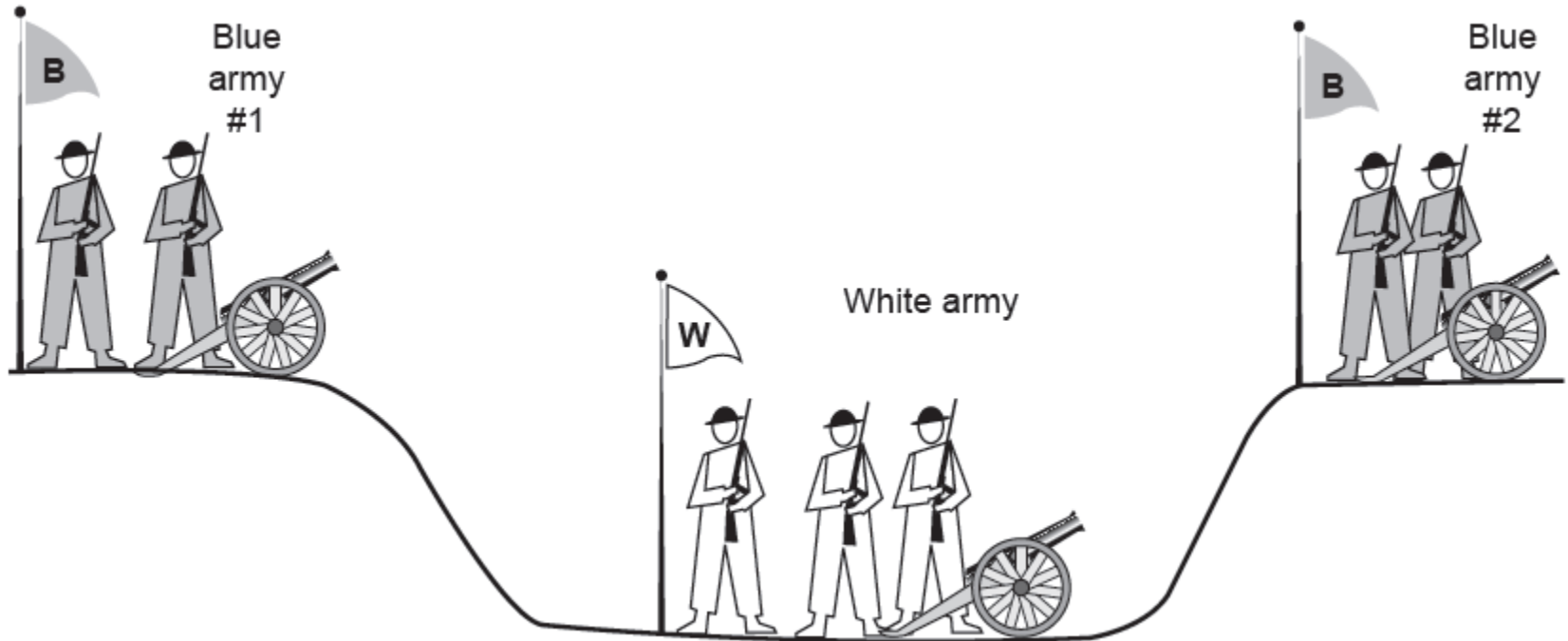
Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. Duplicate CONNECTION REQUEST and duplicate ACK

# Connection Release (1)

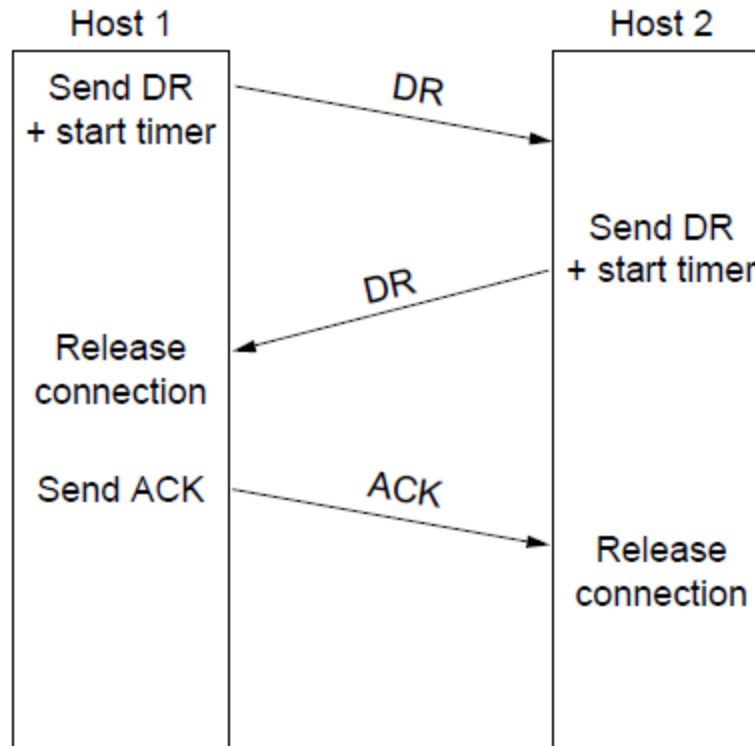


Abrupt disconnection with loss of data

# Connection Release (2)



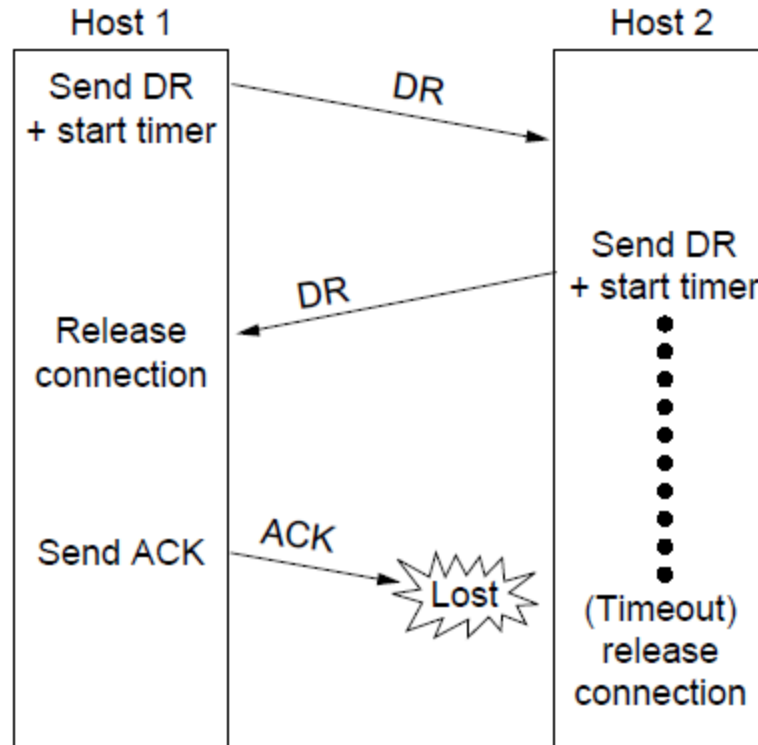
# Connection Release (3)



Four protocol scenarios for releasing a connection.

(a) Normal case of three-way handshake

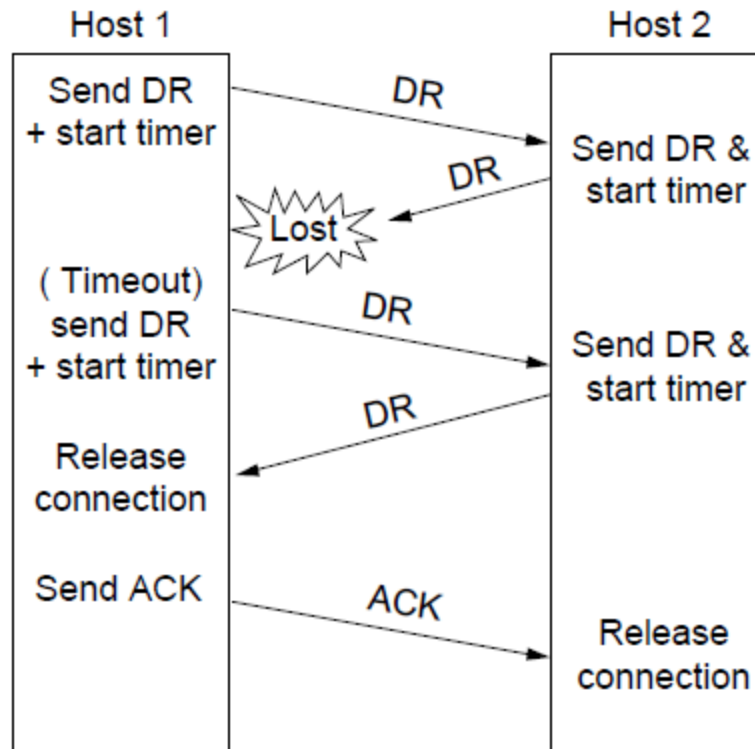
# Connection Release (4)



Four protocol scenarios for releasing a connection.

(b) Final ACK lost.

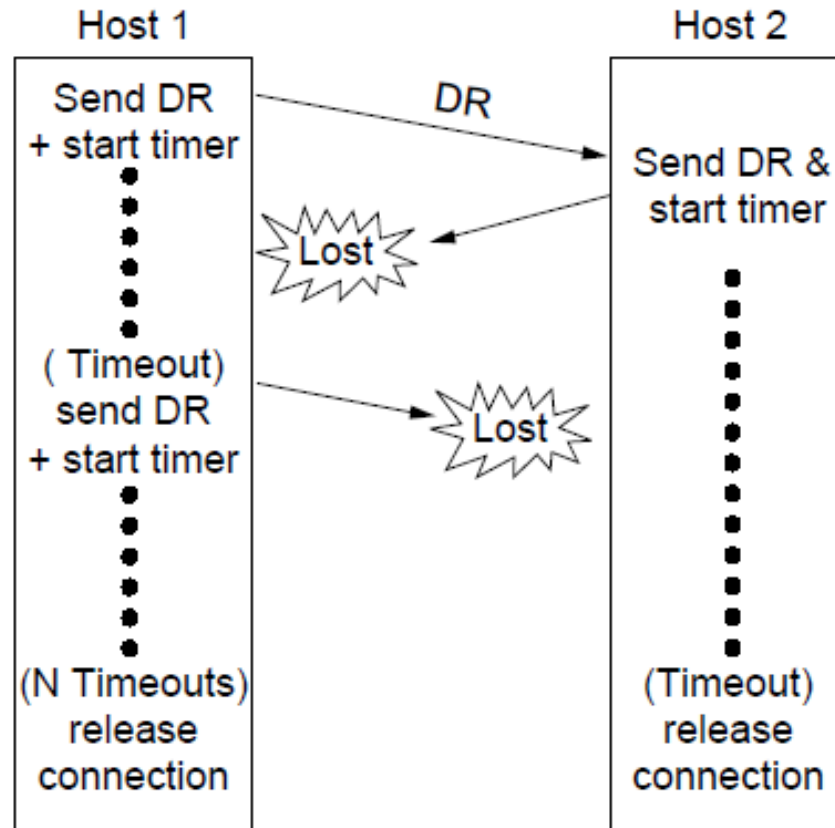
# Connection Release (5)



Four protocol scenarios for releasing a connection.

(c) Response lost

# Connection Release (6)



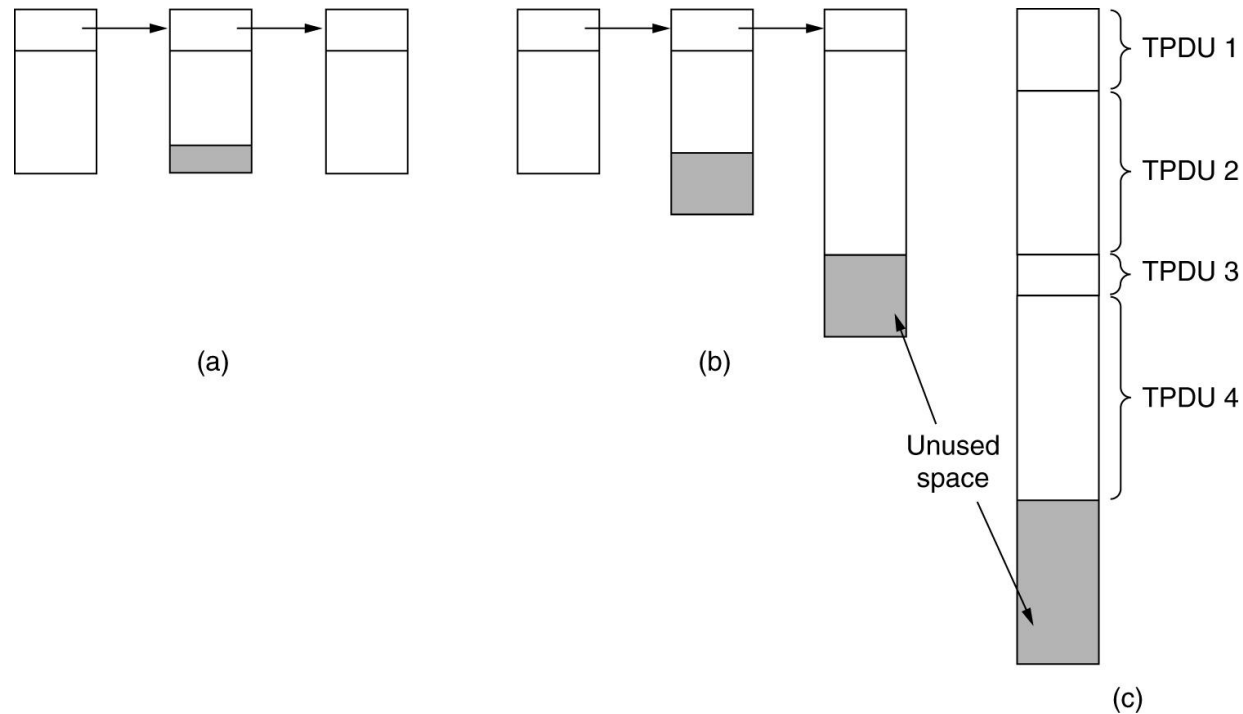
Four protocol scenarios for releasing a connection.

(d) Response lost and subsequent DRs lost.

# Flow Control

- Use Sliding Window
- Buffering
  - Sender buffers all TPDU's until acknowledged
  - TPDU lost by the network
    - Unreliable service
    - Receiver not having buffer
- How should buffers be managed
  - Dedicate
  - Acquire when needed
- Traffic
  - Low bandwidth, Bursty – buffer at sender – acquire at receiver
  - High bandwidth, smooth – Buffer at both ends
- Exchange Buffer information

# Buffer Management



- (a) Chained fixed-size buffers. (b) Chained variable-sized buffers.  
(c) One large circular buffer per connection.

# Flow Control and Buffering (2)

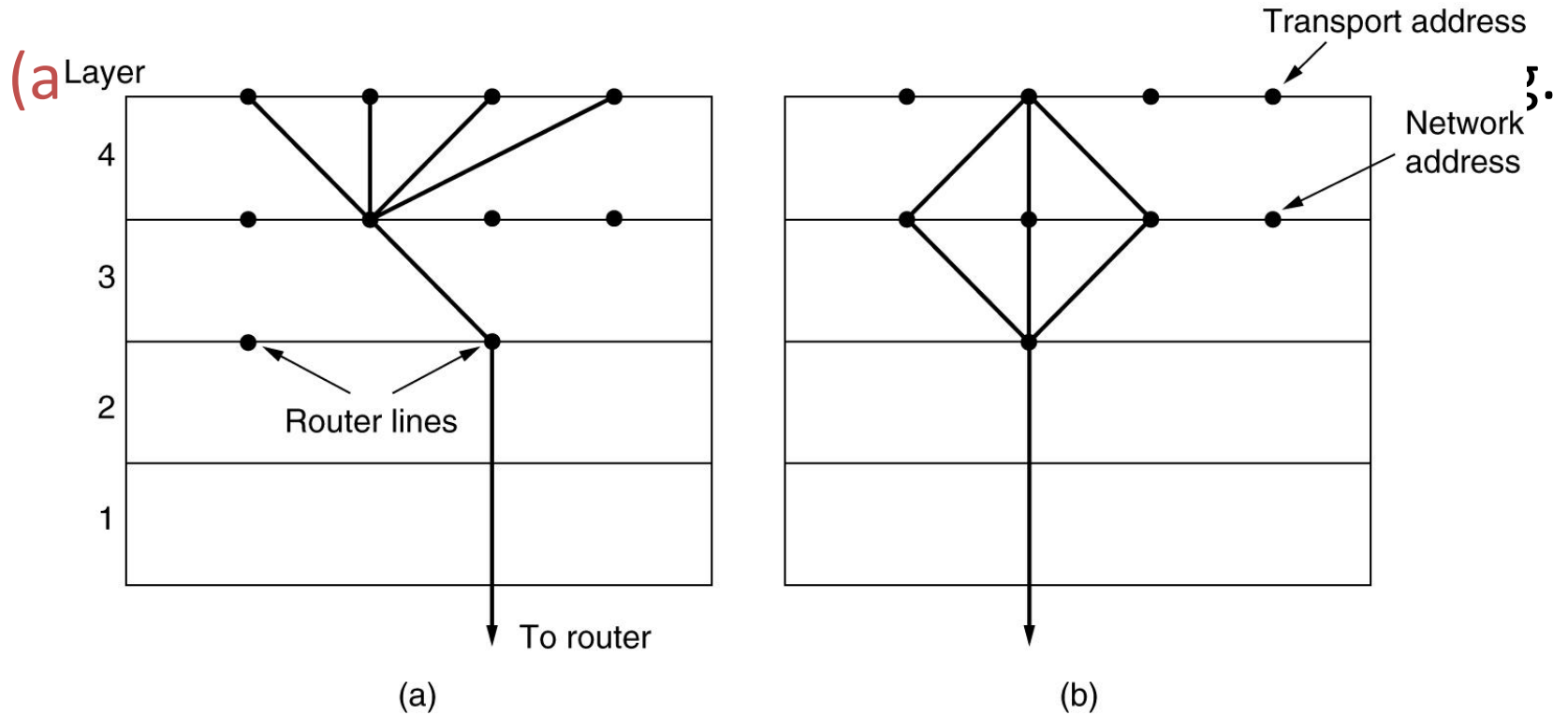
Dynamic buffer allocation. The arrows show the direction of transmission. An ellipsis (...) indicates a lost TPDU.

	<u>A</u>	<u>Message</u>	<u>B</u>	<u>Comments</u>
1	→	< request 8 buffers >	→	A wants 8 buffers
2	←	<ack = 15, buf = 4 >	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0 >	→	A has 3 buffers left now
4	→	<seq = 1, data = m1 >	→	A has 2 buffers left now
5	→	<seq = 2, data = m2 >	•••	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3 >	←	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3 >	→	A has 1 buffer left
8	→	<seq = 4, data = m4 >	→	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2 >	→	A times out and retransmits
10	←	<ack = 4, buf = 0 >	←	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1 >	←	A may now send 5
12	←	<ack = 4, buf = 2 >	←	B found a new buffer somewhere
13	→	<seq = 5, data = m5 >	→	A has 1 buffer left
14	→	<seq = 6, data = m6 >	→	A is now blocked again
15	←	<ack = 6, buf = 0 >	←	A is still blocked
16	•••	<ack = 6, buf = 4 >	←	Potential deadlock

# Throughput limits

- Buffers
- Bandwidth – subnet's carrying capacity
  - $K$  TPDU's per second
  - $X$  paths then total of  $XK$
- Flow control to manage
  - Manage window size
    - If network can handle  $c$  TPDU's/sec and Cycle time is  $r$  then the window size should be  $cr$

# Multiplexing



Upward Multiplexing

Downward Multiplexing

# Crash Recovery

- Network Failures
  - Transport layer handles
    - Connectionless
    - Connection oriented
- Host Crashes
  - Server crash and may reboot
    - Send broadcast asking clients to inform of prior connections ( stop and wait protocol)
      - Client – one TPDU outstanding or none outstanding

# Crash Recovery

Strategy used by receiving host

← First ACK, then write →
← First write, then ACK →
b

Strategy used by sending host	First ACK, then write			First write, then ACK		
	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

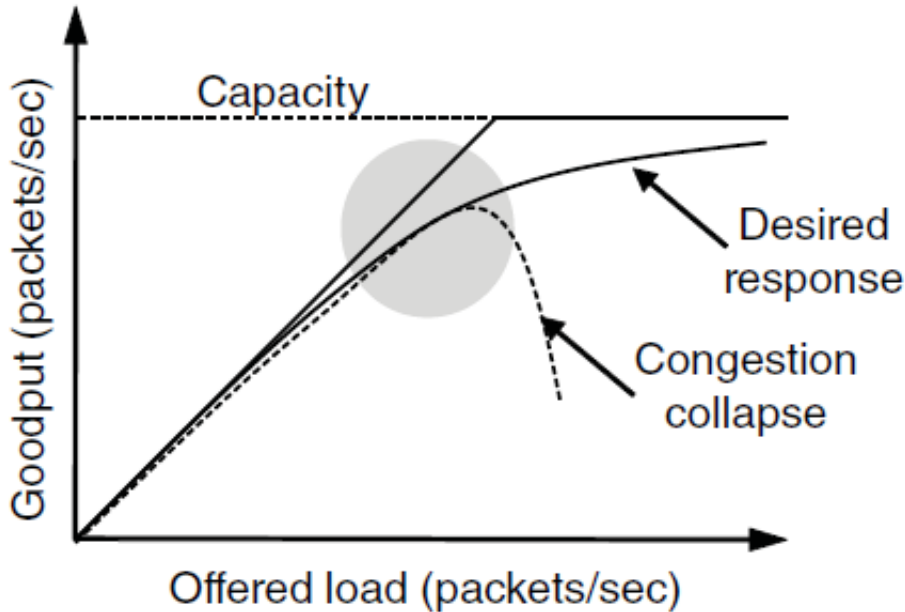
OK = Protocol functions correctly  
 DUP = Protocol generates a duplicate message  
 LOST = Protocol loses a message

Recovery from layer N Crash can only be done at layer N+1

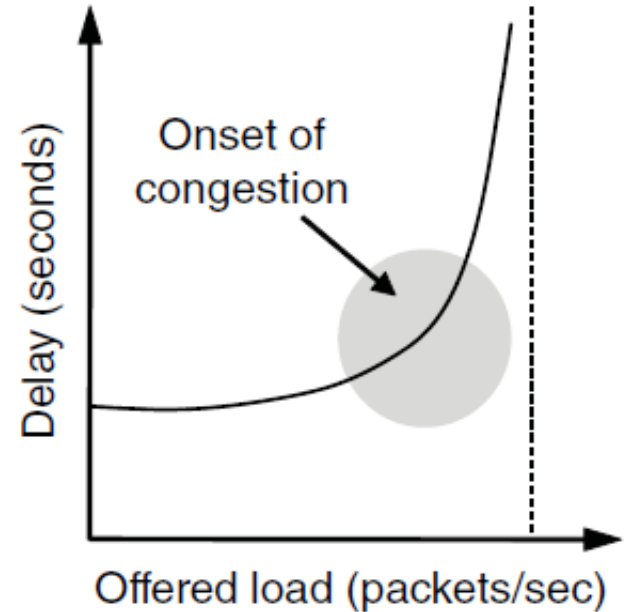
# Congestion Control

- Desirable bandwidth allocation
- Regulating the sending rate

# Desirable Bandwidth Allocation (1)



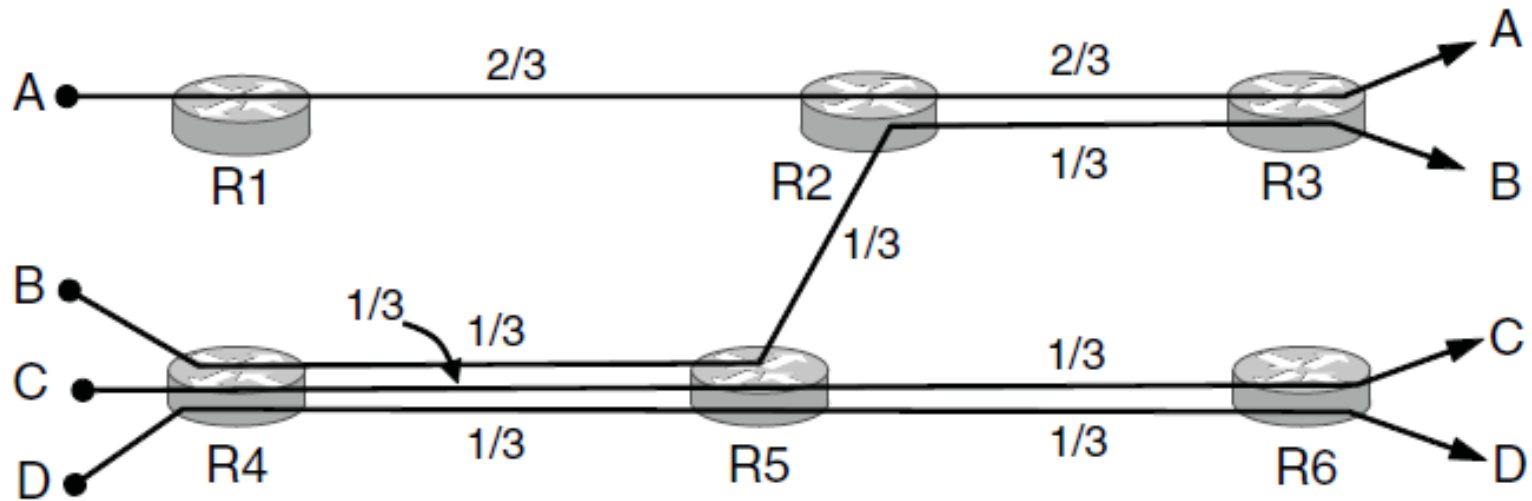
(a)



(b)

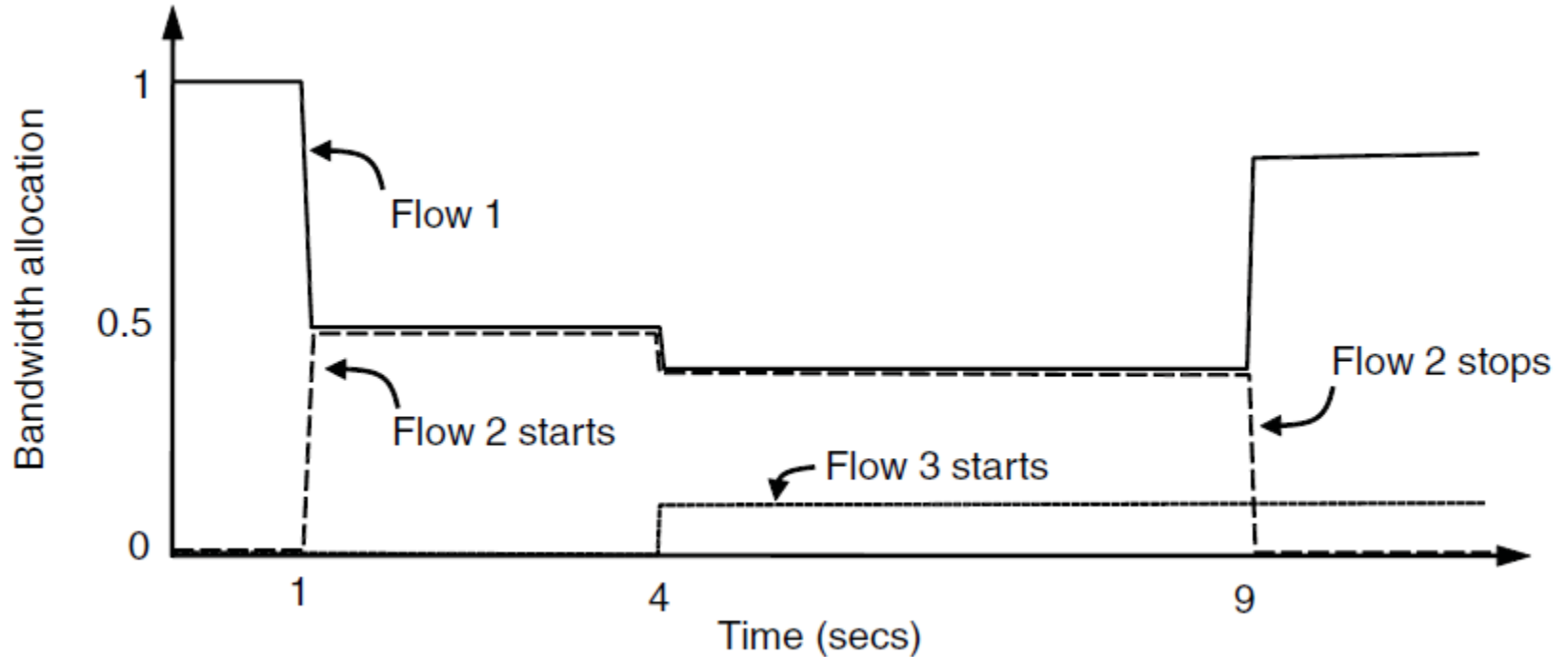
(a) Goodput and (b) delay as a function of offered load

# Desirable Bandwidth Allocation (2)



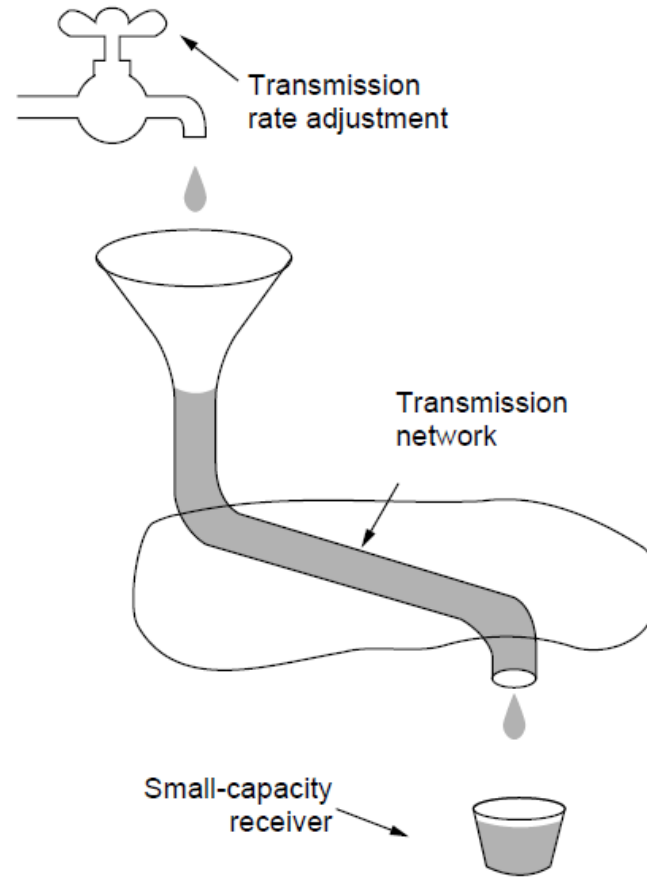
Max-min bandwidth allocation for four flows

# Desirable Bandwidth Allocation (3)



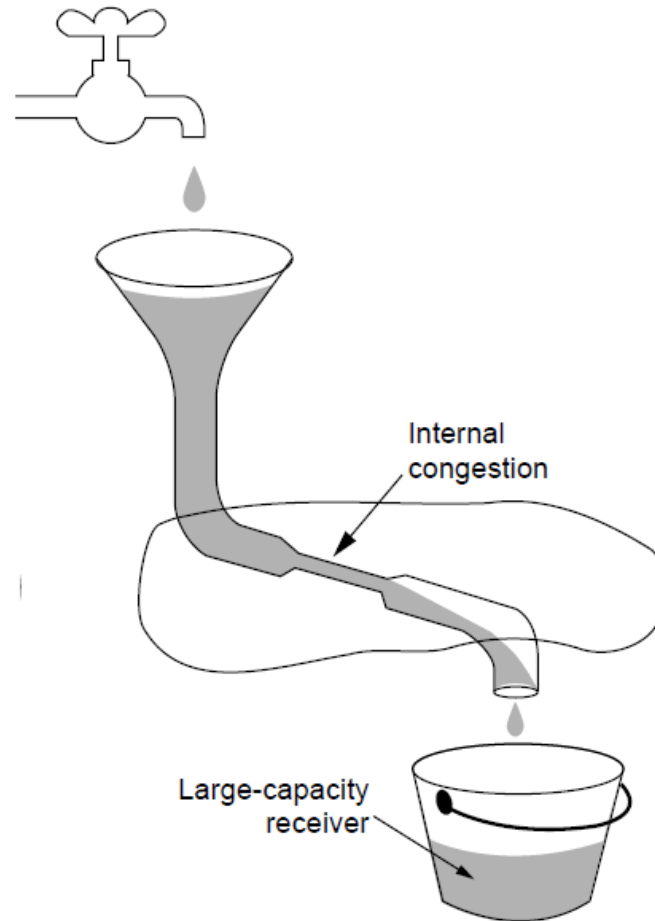
Changing bandwidth allocation over time

# Regulating the Sending Rate (1)



A fast network feeding a low-capacity receiver

# Regulating the Sending Rate (2)



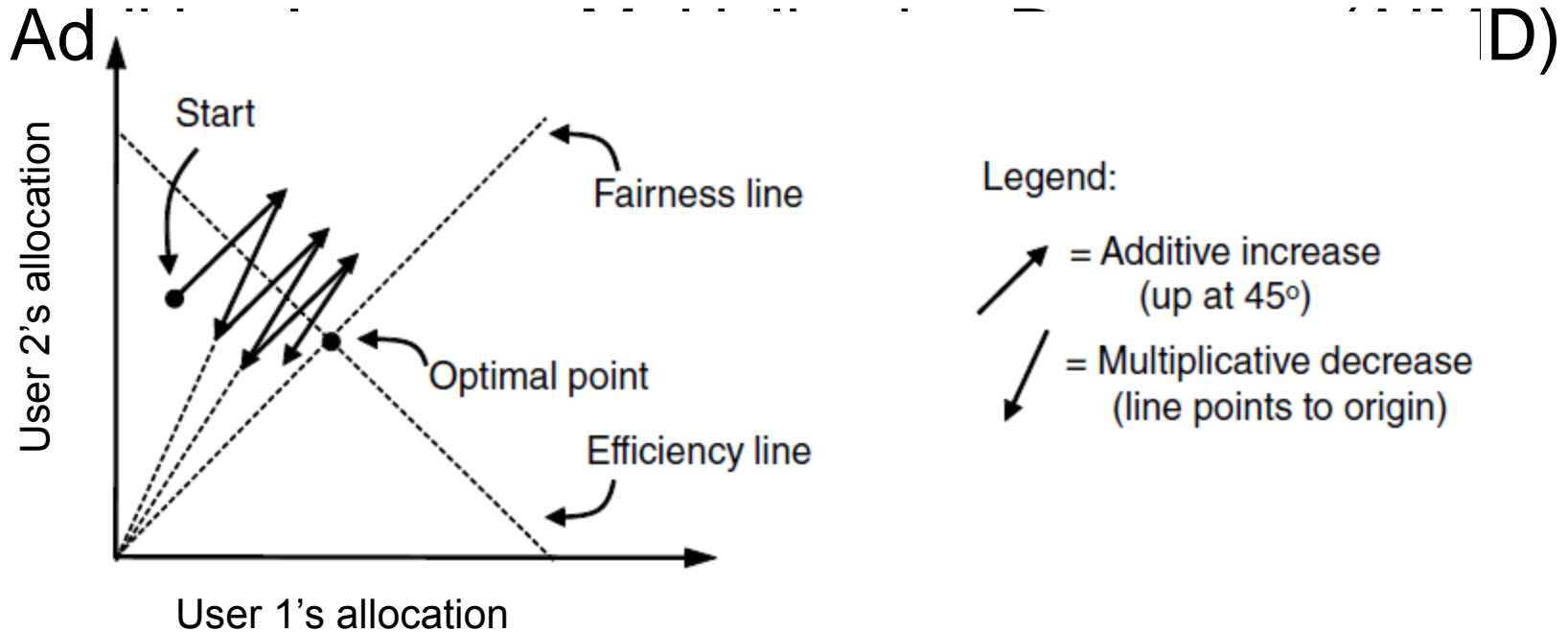
A slow network feeding a high-capacity receiver

# Regulating the Sending Rate (3)

<b>Protocol</b>	<b>Signal</b>	<b>Explicit?</b>	<b>Precise?</b>
XCP	Rate to use	Yes	Yes
TCP with ECN	Congestion warning	Yes	No
FAST TCP	End-to-end delay	No	Yes
CUBIC TCP	Packet loss	No	No
TCP	Packet loss	No	No

Some congestion control protocols

# Regulating the Sending Rate (4)



# A Simple Transport Protocol

- The Example Service Primitives
- The Example Transport Entity
- The Example as a Finite State Machine

Similar to TCP but simpler

# Service Primitives

- Connect
  - Parameters – local and remote TSAPs
  - Caller is blocked
  - If connection succeeds the caller is unblocked and transmission starts
- Listen – specifies a TSAP to listen to
- Disconnect
- Send
- Receive
- \*\* Library procedures

# Service Primitives

- Connnum=LISTEN(local)
- Connnum=Connect(local,remote)
- Status = Send(Connnum, buffer,bytes)
  - No Connection, illegal buffer address, negative count
- Status = Receive(Connnum, buffer, bytes)
- Status = Disconnect(Connnum)

# The Transport Entity

- Use connection oriented, reliable network service
- Transport Entity is part of the user process
- Network Layer interface
  - To\_net and from\_net
  - Parameters –
    - Connection Identifier
    - Q bit – control message
    - M bit – more data from this message to follow
    - Packet Type
    - Pointer to data

# The Example Transport Entity

The network layer packets used in our example.

<b>Network packet</b>	<b>Meaning</b>
CALL REQUEST	Sent to establish a connection
CALL ACCEPTED	Response to CALL REQUEST
CLEAR REQUEST	Sent to release a connection
CLEAR CONFIRMATION	Response to CLEAR REQUEST
DATA	Used to transport data
CREDIT	Control packet for managing the window

# The Example Transport Entity (2)

Each connection is in one of seven states:

1. Idle – Connection not established yet.
2. Waiting – CONNECT has been executed, CALL REQUEST sent.
3. Queued – A CALL REQUEST has arrived; no LISTEN yet.
4. Established – The connection has been established.
5. Sending – The user is waiting for permission to send a packet.
6. Receiving – A RECEIVE has been done.
7. DISCONNECTING – a DISCONNECT has been done locally.

# State Transitions

- A primitive is executed
- A packet arrives
- A timer expires

# The Example Transport Entity (3)

```
#define MAX_CONN 32                /* max number of simultaneous connections */
#define MAX_MSG_SIZE 8192          /* largest message in bytes */
#define MAX_PKT_SIZE 512          /* largest packet in bytes */
#define TIMEOUT 20
#define CRED 1
#define OK 0

#define ERR_FULL -1
#define ERR_REJECT -2
#define ERR_CLOSED -3
#define LOW_ERR -3

typedef int transport_address;
typedef enum {CALL_REQ,CALL_ACC,CLEAR_REQ,CLEAR_CONF,DATA_PKT,CREDIT} pkt_type;
typedef enum {IDLE,WAITING,QUEUED,ESTABLISHED,SENDING,RECEIVING,DISCONN} cstate;

/* Global variables. */
transport_address listen_address; /* local address being listened to */
int listen_conn;                 /* connection identifier for listen */
unsigned char data[MAX_PKT_SIZE]; /* scratch area for packet data */

struct conn {
    transport_address local_address, remote_address;
    cstate state;                /* state of this connection */
    unsigned char *user_buf_addr; /* pointer to receive buffer */
    int byte_count;              /* send/receive count */
    int clr_req_received;        /* set when CLEAR_REQ packet received */
    int timer;                   /* used to time out CALL_REQ packets */
    int credits;                 /* number of messages that may be sent */
} conn[MAX_CONN + 1];           /* slot 0 is not used */
```

# The Example Transport Entity (4)

```
void sleep(void);                /* prototypes */
void wakeup(void);
void to_net(int cid, int q, int m, pkt_type pt, unsigned char *p, int bytes);
void from_net(int *cid, int *q, int *m, pkt_type *pt, unsigned char *p, int *bytes);

int listen(transport_address t)
{ /* User wants to listen for a connection. See if CALL_REQ has already arrived. */
  int i, found = 0;

  for (i = 1; i <= MAX_CONN; i++)          /* search the table for CALL_REQ */
    if (conn[i].state == QUEUED && conn[i].local_address == t) {
      found = i;
      break;
    }

  if (found == 0) {
    /* No CALL_REQ is waiting. Go to sleep until arrival or timeout. */
    listen_address = t; sleep(); i = listen_conn ;
  }
  conn[i].state = ESTABLISHED;             /* connection is ESTABLISHED */
  conn[i].timer = 0;                       /* timer is not used */
}
```

# The Example Transport Entity (5)

```
listen_conn = 0; /* 0 is assumed to be an invalid address */
to_net(i, 0, 0, CALL_ACC, data, 0); /* tell net to accept connection */
return(i); /* return connection identifier */
}
```

```
int connect(transport_address l, transport_address r)
{ /* User wants to connect to a remote process; send CALL_REQ packet. */
  int i;
  struct conn *cptr;

  data[0] = r; data[1] = l; /* CALL_REQ packet needs these */
  i = MAX_CONN; /* search table backward */
  while (conn[i].state != IDLE && i > 1) i = i - 1;
  if (conn[i].state == IDLE) {
    /* Make a table entry that CALL_REQ has been sent. */
    cptr = &conn[i];
    cptr->local_address = l; cptr->remote_address = r;
    cptr->state = WAITING; cptr->clr_req_received = 0;
    cptr->credits = 0; cptr->timer = 0;
    to_net(i, 0, 0, CALL_REQ, data, 2);
    sleep(); /* wait for CALL_ACC or CLEAR_REQ */
    if (cptr->state == ESTABLISHED) return(i);
    if (cptr->clr_req_received) {
      /* Other side refused call. */
      cptr->state = IDLE; /* back to IDLE state */
      to_net(i, 0, 0, CLEAR_CONF, data, 0);
      return(ERR_REJECT);
    }
  }
  else return(ERR_FULL); /* reject CONNECT: no table space */
}
```

# The Example Transport Entity (6)

```
int send(int cid, unsigned char bufptr[], int bytes)
{ /* User wants to send a message. */
  int i, count, m;
  struct conn *cptr = &conn[cid];

  /* Enter SENDING state. */
  cptr->state = SENDING;
  cptr->byte_count = 0; /* # bytes sent so far this message */
  if (cptr->clr_req_received == 0 && cptr->credits == 0) sleep();
  if (cptr->clr_req_received == 0) {
    /* Credit available; split message into packets if need be. */
    do {
      if (bytes - cptr->byte_count > MAX_PKT_SIZE) { /* multipacket message */
        count = MAX_PKT_SIZE; m = 1; /* more packets later */
      } else { /* single packet message */
        count = bytes - cptr->byte_count; m = 0; /* last pkt of this message */
      }
      for (i = 0; i < count; i++) data[i] = bufptr[cptr->byte_count + i];
      to_net(cid, 0, m, DATA_PKT, data, count); /* send 1 packet */
      cptr->byte_count = cptr->byte_count + count; /* increment bytes sent so far */
    } while (cptr->byte_count < bytes); /* loop until whole message sent */
  }
}
```

# The Example Transport Entity (7)

```
    cptr->credits -- ;                /* each message uses up one credit */
    cptr->state = ESTABLISHED;
    return(OK);
} else {
    cptr->state = ESTABLISHED;
    return(ERR_CLOSED);              /* send failed: peer wants to disconnect */
}
}
```

```
int receive(int cid, unsigned char bufptr[], int *bytes)
{ /* User is prepared to receive a message. */
    struct conn *cptr = &conn[cid];

    if (cptr->clr_req_received == 0) {
        /* Connection still established; try to receive. */
        cptr->state = RECEIVING;
        cptr->user_buf_addr = bufptr;
        cptr->byte_count = 0;
        data[0] = CRED;
        data[1] = 1;
        to_net(cid, 1, 0, CREDIT, data, 2);    /* send credit */
        sleep();                               /* block awaiting data */
        *bytes = cptr->byte_count;
    }
    cptr->state = ESTABLISHED;
    return(cptr->clr_req_received ? ERR_CLOSED : OK);
}
```

# The Example Transport Entity (8)

```
int disconnect(int cid)
{ /* User wants to release a connection. */
  struct conn *cptr = &conn[cid];

  if (cptr->clr_req_received) { /* other side initiated termination */
    cptr->state = IDLE; /* connection is now released */
    to_net(cid, 0, 0, CLEAR_CONF, data, 0);
  } else { /* we initiated termination */
    cptr->state = DISCONN; /* not released until other side agrees */
    to_net(cid, 0, 0, CLEAR_REQ, data, 0);
  }
  return(OK);
}

void packet_arrival(void)
{ /* A packet has arrived, get and process it. */
  int cid; /* connection on which packet arrived */
  int count, i, q, m;
  pkt_type ptype; /* CALL_REQ, CALL_ACC, CLEAR_REQ, CLEAR_CONF, DATA_PKT, CREDIT */
  unsigned char data[MAX_PKT_SIZE]; /* data portion of the incoming packet */
  struct conn *cptr;

  from_net(&cid, &q, &m, &ptype, data, &count); /* go get it */
  cptr = &conn[cid];
```



# The Example Transport Entity (10)

```
}  
void clock(void)  
{ /* The clock has ticked, check for timeouts of queued connect requests. */  
  int i;  
  struct conn *cptr;  
  for (i = 1; i <= MAX_CONN; i++) {  
    cptr = &conn[i];  
    if (cptr->timer > 0) { /* timer was running */  
      cptr->timer--;  
      if (cptr->timer == 0) { /* timer has now expired */  
        cptr->state = IDLE;  
        to_net(i, 0, 0, CLEAR_REQ, data, 0);  
      }  
    }  
  }  
}
```

# The Example as a Finite State Machine

The example protocol as a finite state machine. Each entry has an optional predicate, an optional action, and the new state. The tilde indicates that no major action is taken. An overbar above a predicate indicate the negation of the predicate. Blank entries correspond to impossible or invalid events.

		State						
		Idle	Waiting	Queued	Established	Sending	Receiving	Dis- connecting
Primitives	LISTEN	P1: ~/Idle P2: A1/Estab P2: A2/Idle		~/Estab				
	CONNECT	P1: ~/Idle P1: A3/Wait						
	DISCONNECT				P4: A5/Idle P4: A6/Disc			
	SEND				P5: A7/Estab P5: A8/Send			
	RECEIVE				A9/Receiving			
Incoming packets	Call_req	P3: A1/Estab P3: A4/Queue'd						
	Call_acc		~/Estab					
	Clear_req		~/Idle		A10/Estab	A10/Estab	A10/Estab	~/Idle
	Clear_conf							~/Idle
	DataPkt						A12/Estab	
Clock	Credit				A11/Estab	A7/Estab		
	Timeout			~/Idle				

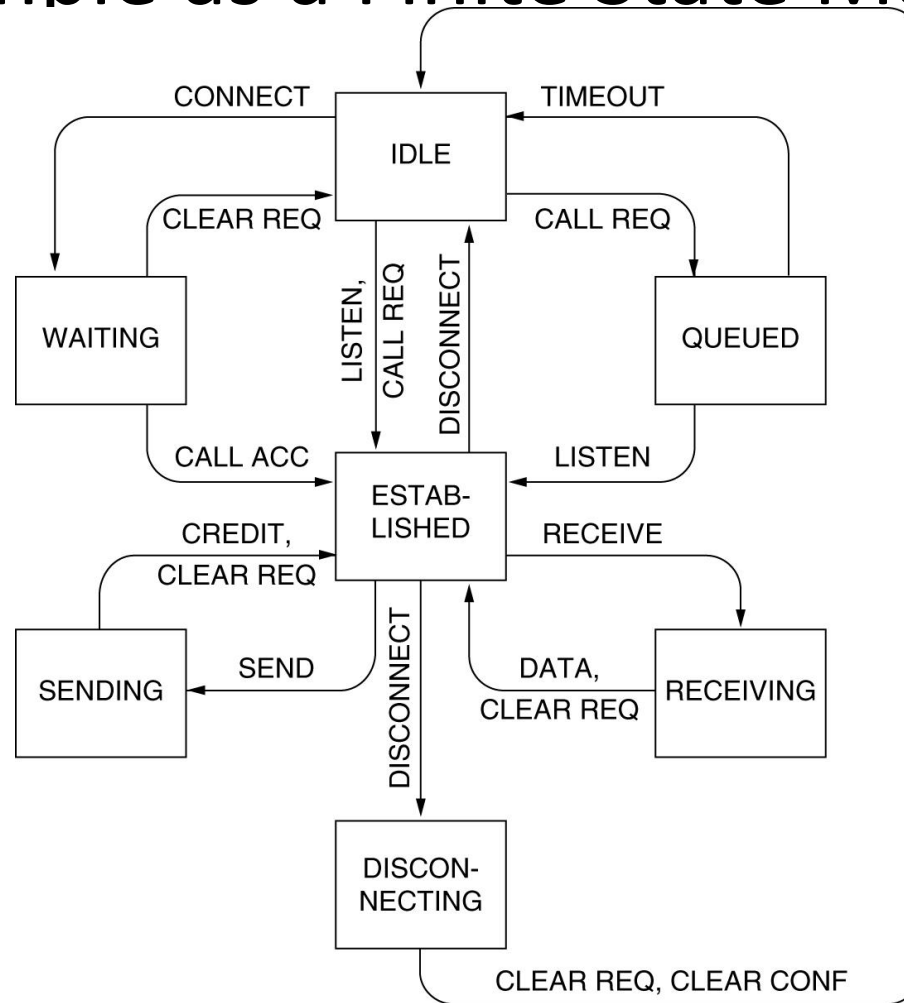
## Predicates

P1: Connection table full  
 P2: Call\_req pending  
 P3: LISTEN pending  
 P4: Clear\_req pending  
 P5: Credit available

## Actions

A1: Send Call\_acc      A7: Send message  
 A2: Wait for Call\_req    A8: Wait for credit  
 A3: Send Call\_req      A9: Send credit  
 A4: Start timer        A10: Set Clr\_req\_received flag  
 A5: Send Clear\_conf    A11: Record credit  
 A6: Send Clear\_req     A12: Accept message

# The Example as a Finite State Machine (2)



The example protocol in graphical form. Transitions that leave the connection state unchanged have been omitted for simplicity.

# The Internet Transport Protocols: UDP

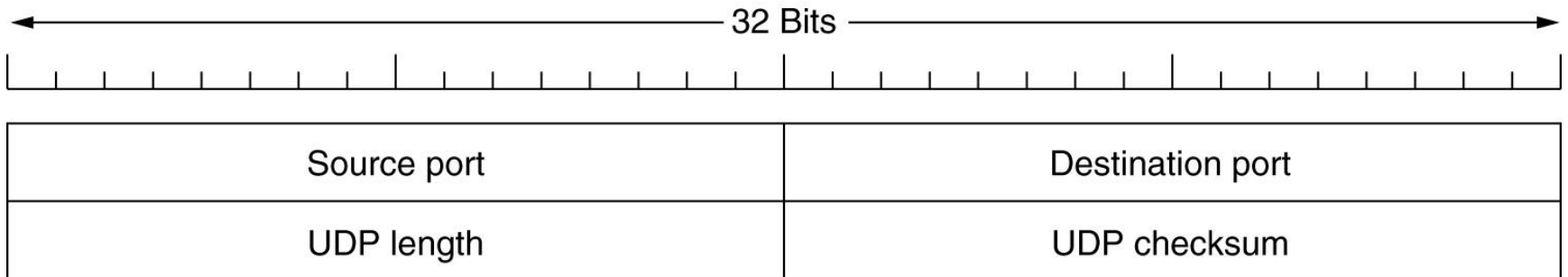
- Introduction to UDP
- Remote Procedure Call
- The Real-Time Transport Protocol

# User Datagram Protocol

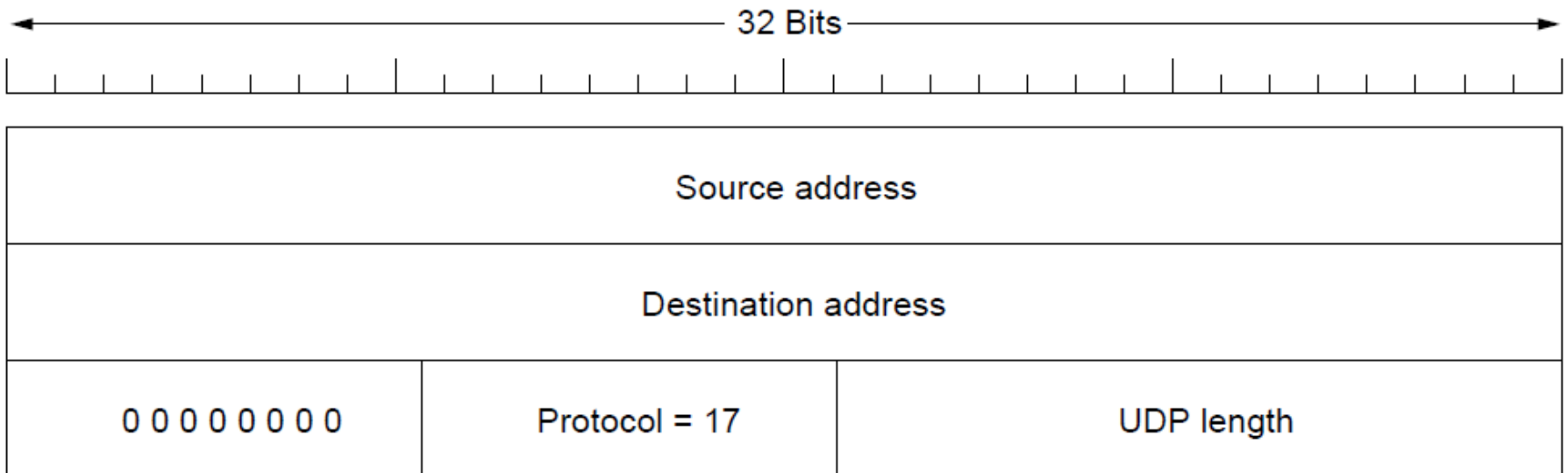
- Connectionless
- Does not do
  - Flow control
  - Error control
  - Retransmissions
- Useful in client-server situations
- Sends segments consisting of an 8-byte header followed by the payload

# Introduction to UDP

The UDP header.



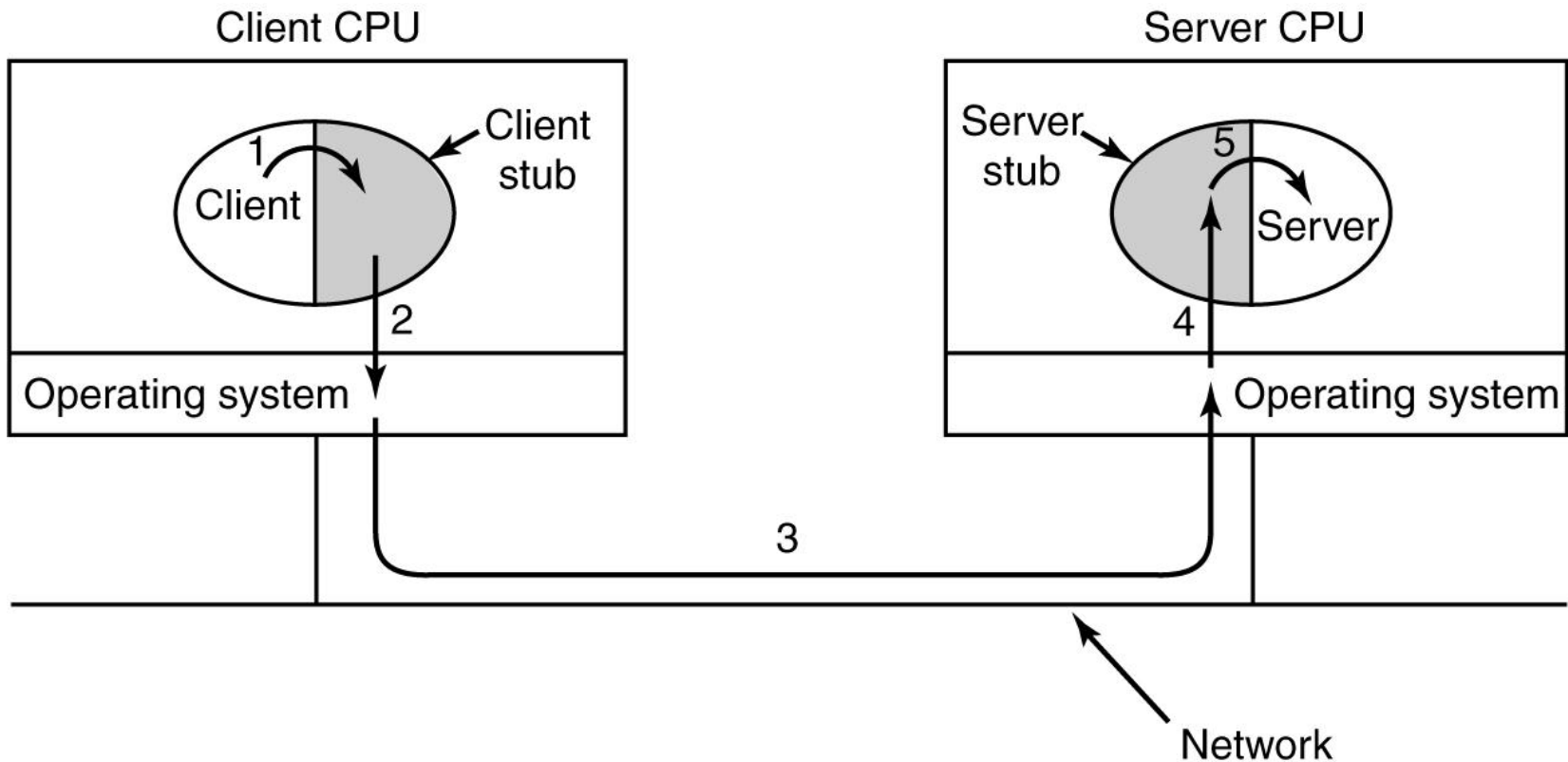
# Introduction to UDP (2)



The IPv4 pseudoheader included in the UDP checksum.

# Remote Procedure Call

Steps in making a remote procedure call. The stubs are shaded.

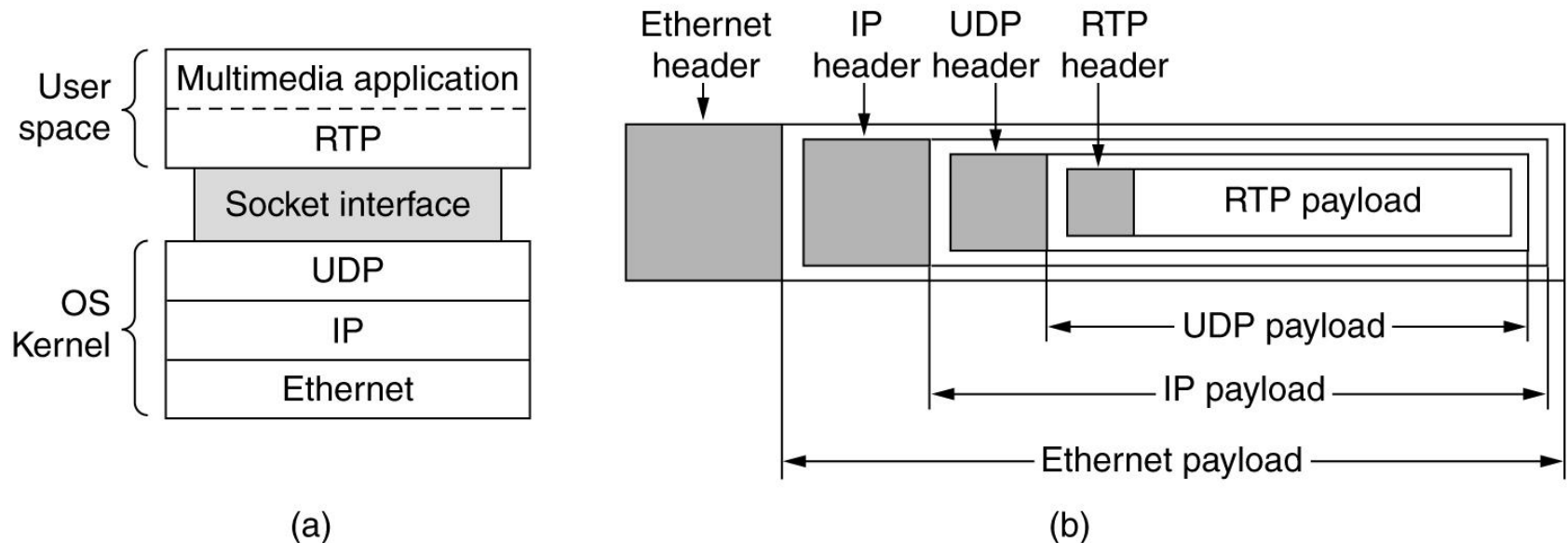


# Limitations of RPC

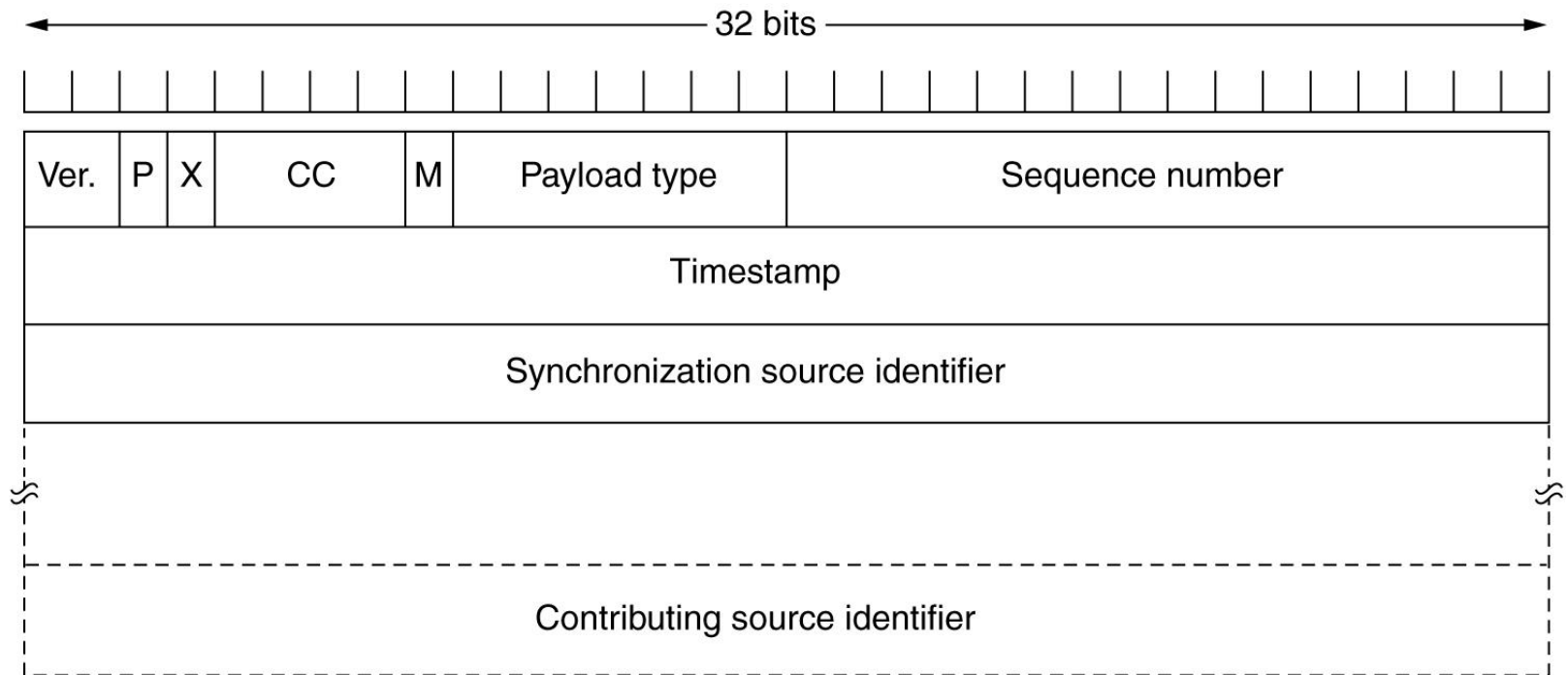
- Pointers
- Weakly Typed languages – variable length arrays
- Not possible always to deduce parameter types
- Global variables

# The Real-Time Transport Protocol

(a) The position of RTP in the protocol stack. (b) Packet nesting.



# The Real-Time Transport Protocol (2)



# RTP Header Fields

- Ver – 2
- P – Packet padded to multiple of 4 bytes
- X – extension header present
- CC – number of contributing sources
- M bit – Application specific marker
- Payload Type – encoding used
- Sequence Number
- Time stamp – produced by the source
- Synchronizations Source Identifier – which stream the packet belongs to

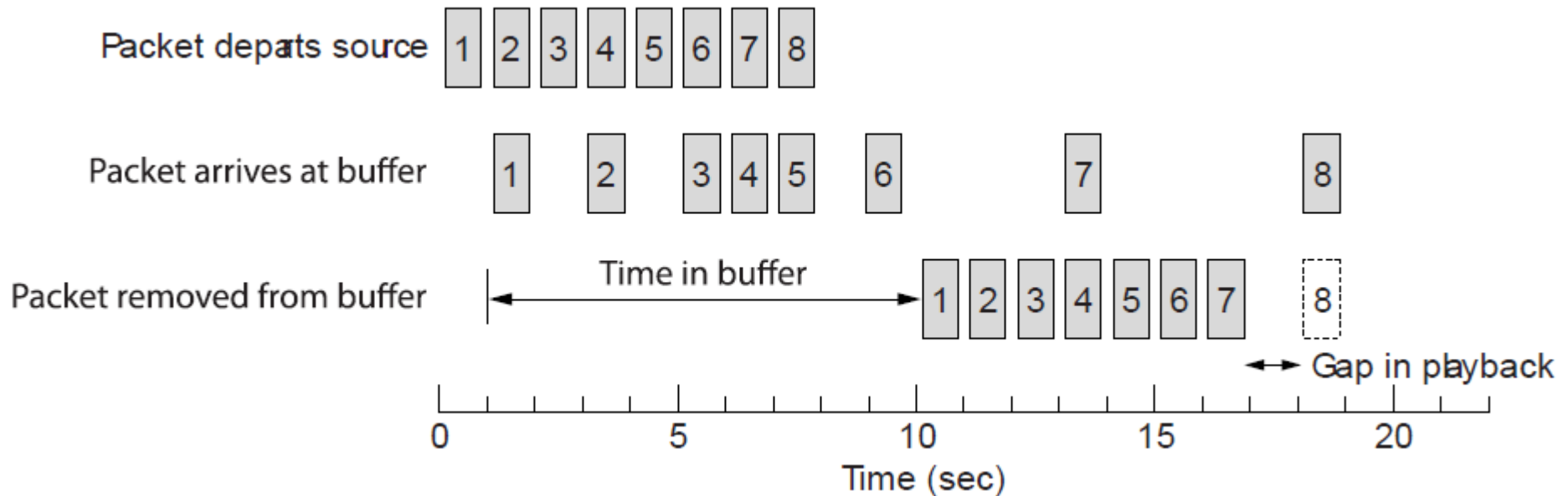
# RTP Profiles

- RTP payloads may contain multiple samples coded in any way the application wants
- Profiles – to support interworking
  - Single Audio Stream
  - Multiple encoding formats may be supported
    - 8-bit pcm samples at 8KHz
    - Delta encoding
    - Predictive encoding
    - MP3
    - ...

# RTCP – Real-time Transport Control Protocol

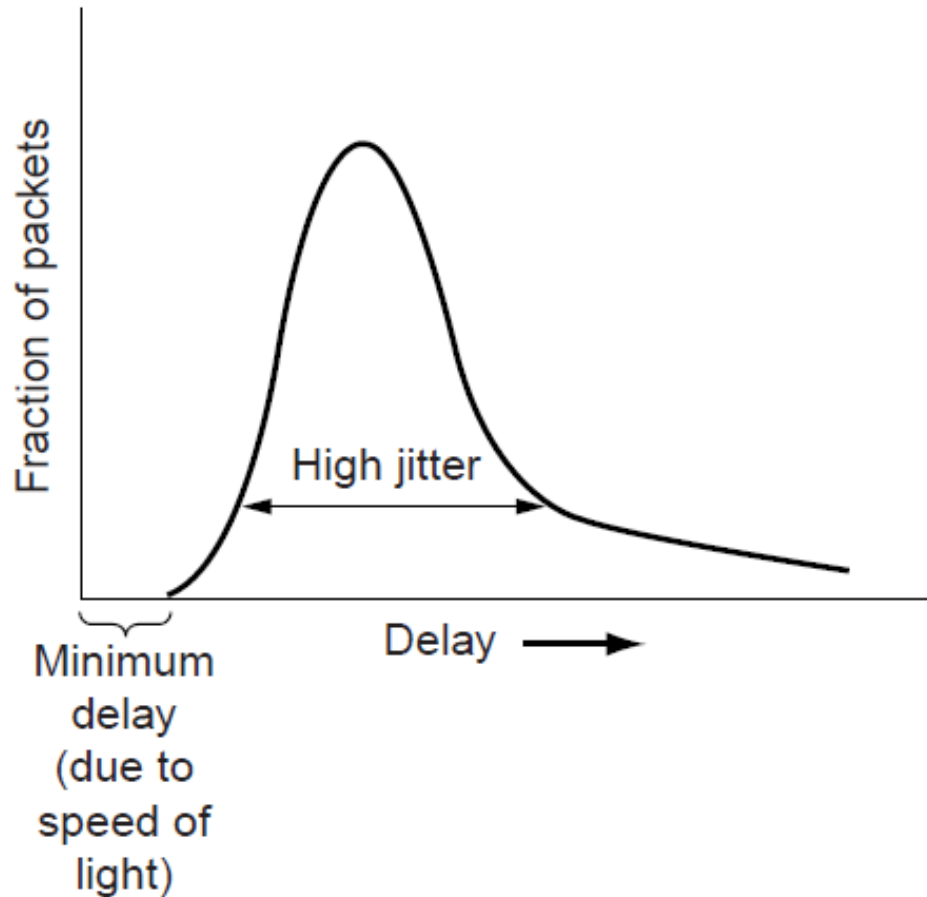
- Control Protocol for RTP
- Does not transport any data
- Handles:
  - Feedback
    - Delay
    - Jitter
    - Bandwidth
    - Congestion, etc.
  - Synchronization
    - Interstream Synchronization – Different clocks, drifts, etc.
  - User Interface

# Real-Time Transport (3)



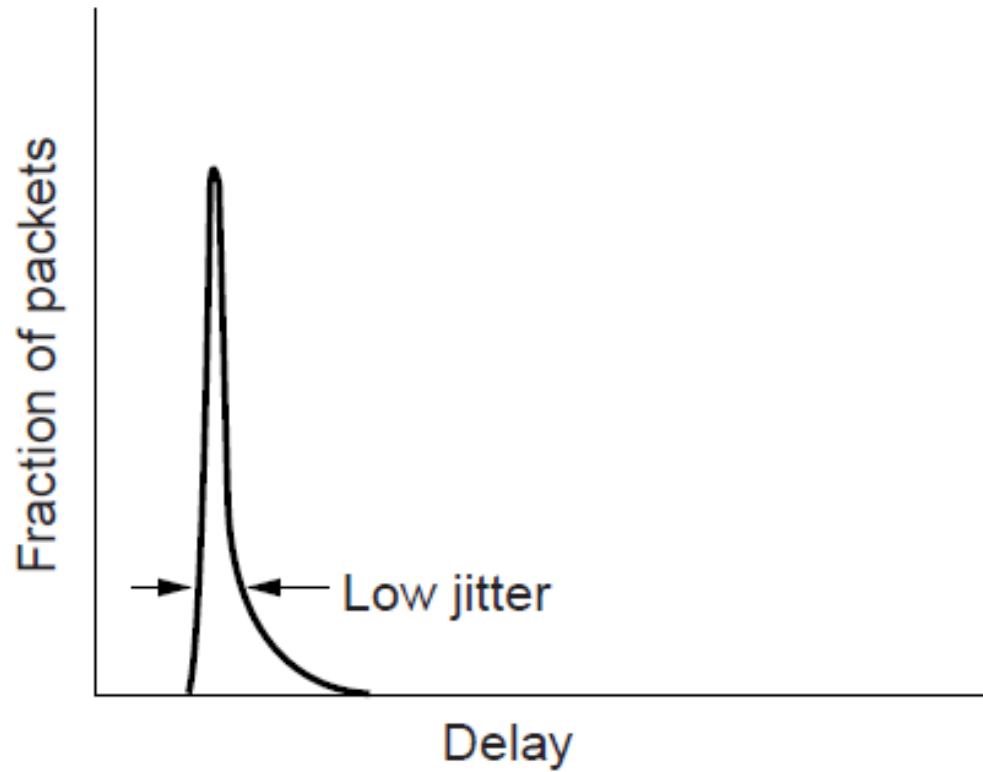
Smoothing the output stream by buffering packets

# Real-Time Transport (3)



High jitter

# Real-Time Transport (4)



Low jitter

# The Internet Transport Protocols: TCP

- Introduction to TCP
- The TCP Service Model
- The TCP Protocol
- The TCP Segment Header
- TCP Connection Establishment
- TCP Connection Release
- TCP Connection Management Modeling
- TCP Transmission Policy
- TCP Congestion Control
- TCP Timer Management
- Wireless TCP and UDP
- Transactional TCP

# TCP

- Reliable end-to-end byte stream over an unreliable internetwork
- Dynamically adapt to the properties of the network
- Robust
  
- RFC 793 – 1122 and 1323

# TCP Transport Entity

- Implemented as
  - Library procedures
  - User process, or
  - Part of the kernel
- Accepts data streams from local processes
- Breaks them into segments of >64KB (usually 1460 bytes)
- Sends each piece in a separate IP packet
- On receive side – reconstruct the original byte stream and give to a process.
- Must recover from errors – time outs, retransmissions, etc.

# TCP Service Model

- End points called sockets
  - Socket number
    - IP address of the host
    - 16-bit number (called the *port*)
- Connection Oriented – Full Duplex, Point-to-Point
  - Establish a connection between sockets
  - An socket may be used for multiple connections at the same time
  - Connection (*socket1, socket2*)

# Berkeley Sockets

The socket primitives for TCP.

<b>Primitive</b>	<b>Meaning</b>
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
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

# The TCP Service Model

Some assigned ports.

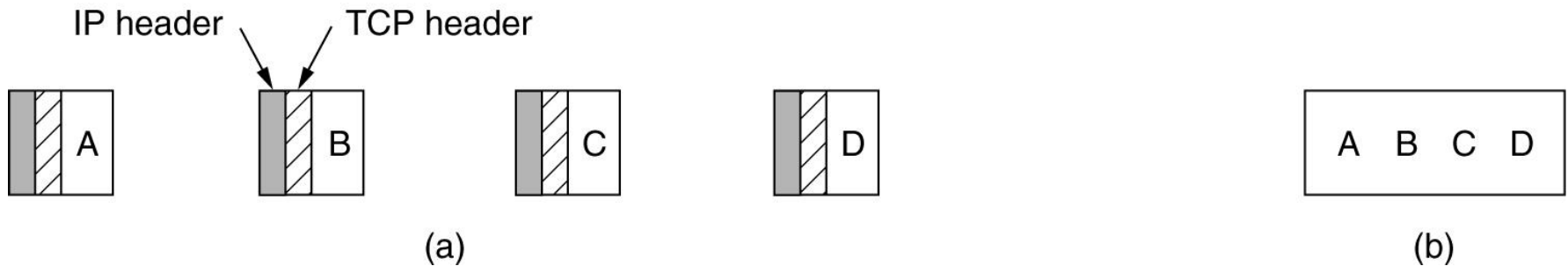
Port	Protocol	Use
21	FTP	File transfer
23	Telnet	Remote login
25	SMTP	E-mail
69	TFTP	Trivial File Transfer Protocol
79	Finger	Lookup info about a user
80	HTTP	World Wide Web
110	POP-3	Remote e-mail access
119	NNTP	USENET news

Ports below 1024 are well-known ports

# Internet Daemon

- Attaches to multiple well-known ports and waits
- When a connection comes in it forks off a new process and executes the appropriate daemon
- That daemon handles the request

# The TCP Service Model



(a) Four 512-byte segments sent as separate IP datagrams.

(b) The 2048 bytes of data delivered to the application in a single READ CALL.

# TCP Service

- No message boundaries are preserved
- Send data as received or buffer it
- PUSH Flag
  - Send data now
  - Useful in sending command from terminal
- Urgent Data
  - DEL or CTRL-C to break off a remote computation
  - Use URGENT flag – Transmit everything right now
    - Receiving application is interrupted

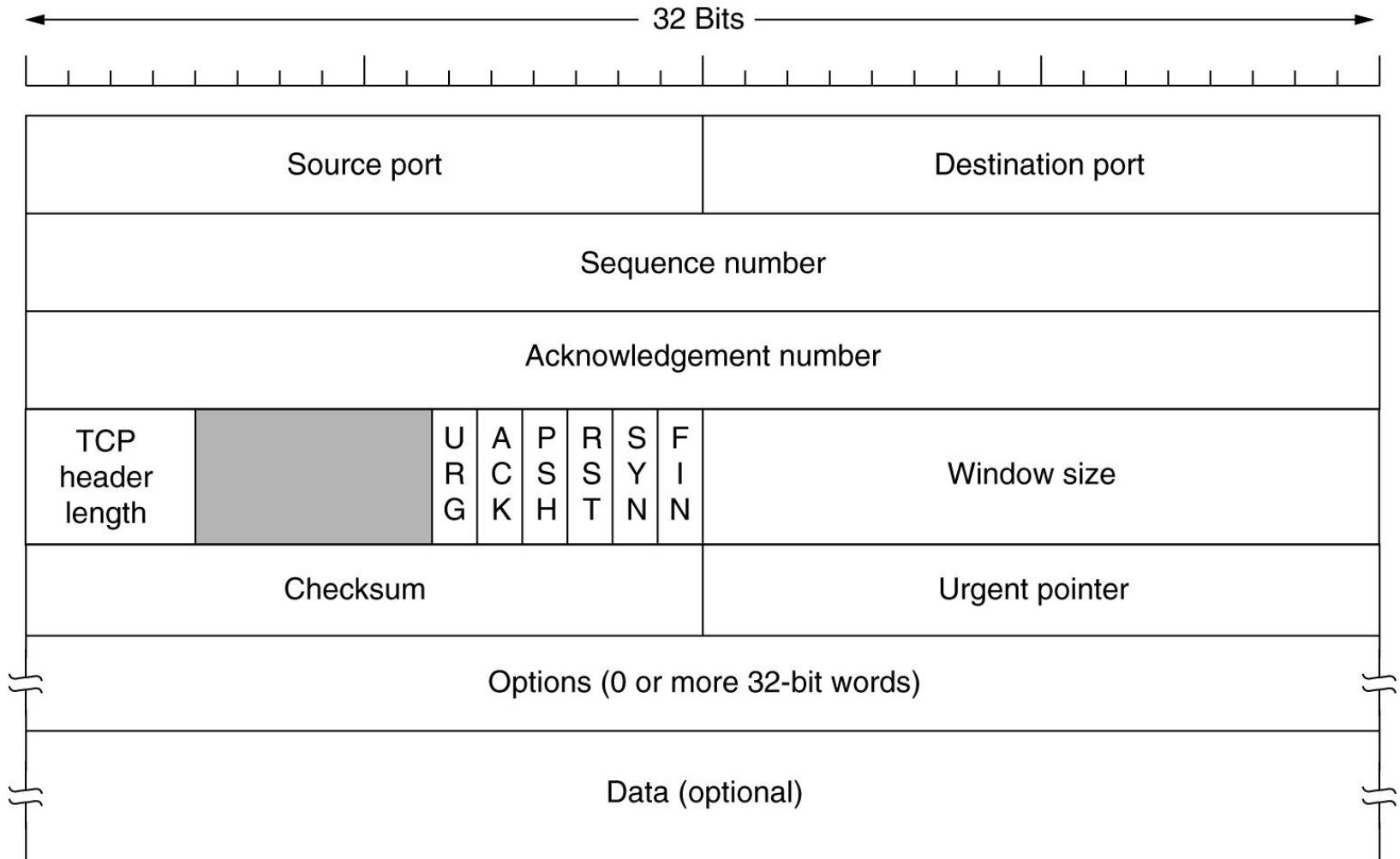
# TCP Protocol

- Exchange segments
  - 20 byte header (plus optional parts)
  - 0 or more data bytes
- Accumulate data from several writes into one segment
- May split data from one write into multiple segments
- Each segment, including the header, must be <65515 byte IP payload
- Each network has MTU- Maximum Transfer Unit
  - Each segment must be less than or equal to MTU

# TCP Protocol

- TCP uses sliding window protocol
- Sequence numbers are for bytes not segments
- Sending – start a timer
- Receiving – send an ack with sequence number = next sequence number expected

# The TCP Segment Header

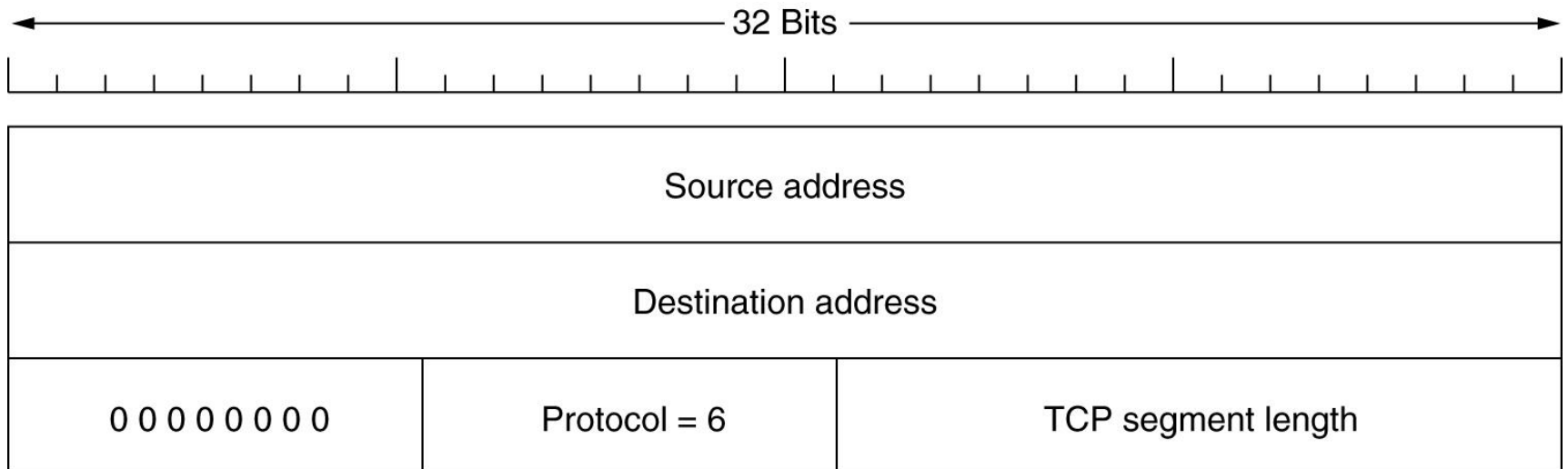


# TCP Segment Header

- Source and Destination Port
- Sequence number
- Acknowledgement Number
  - Next byte expected
- TCP Header Length – Number of 32 bit words in TCP header
- Flags
  - URG – set to 1 if Urgent Pointer is in use
    - Used to indicate a byte offset from the current seq no at which urgent data care there
  - ACK – set to 1 when ack no is valid
  - PSH – Push bit
  - RST – Reset
  - SYN – Used for connection establishment
  - FIN – Used to close a connection

# The TCP Segment Header (2)

- The pseudoheader included in the TCP checksum
- Checksum the header, the data and the pseudoheader
- Add all 16-bit words in 1's complement and then take 1's complement of the sum
- To check calculate on the entire segment and result should be 0.



# TCP Window

- Window size tells how many bytes may be sent starting at the byte acknowledged
  - If 0 means do not send now.
  - May send a segment with same ack no and non-zero window size.

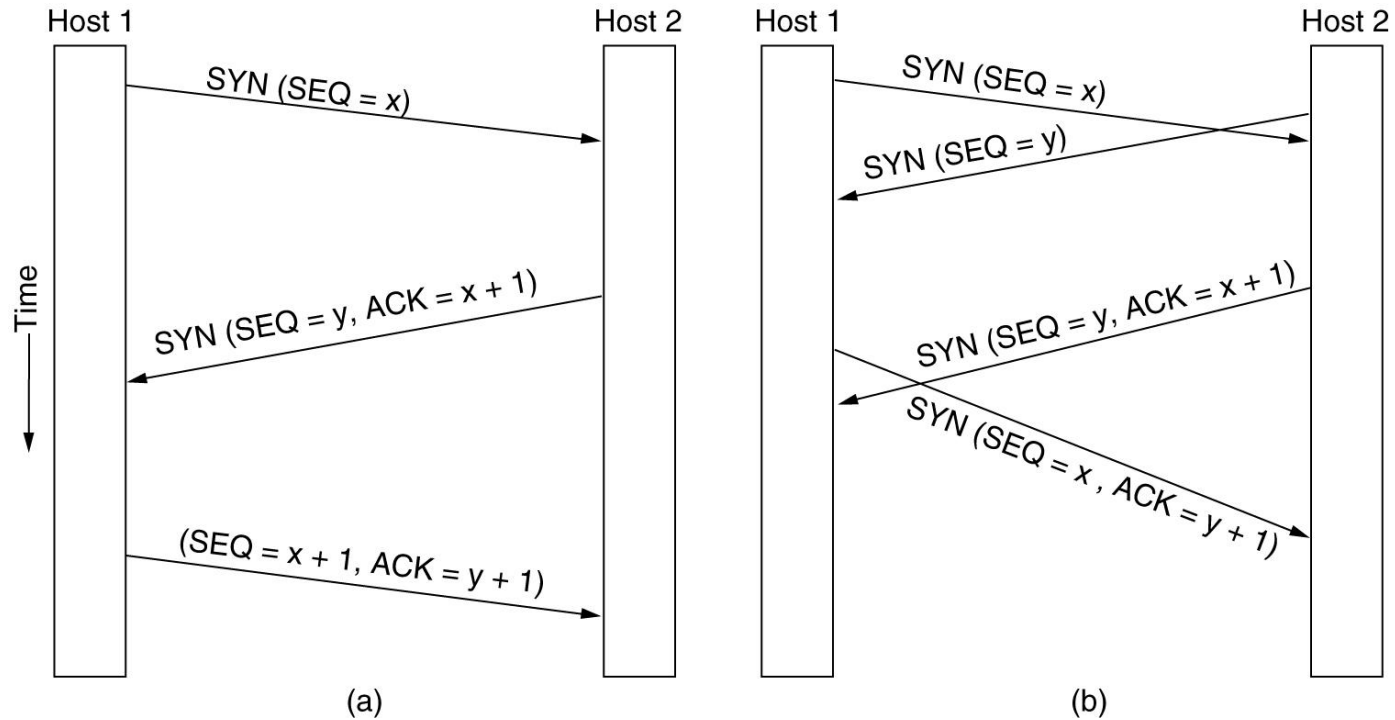
# Maximum segment Size

- All hosts are required to accept TCP segments of  $536+20 = 556$  bytes
- May negotiate max segment size using options.
- Another negotiable parameter – Window Scale
  - May shift to the left by up to 14 bits
  - Giving a max window size of  $2^{30}$  bytes

# Connection Establishment

- Uses three-way handshake
- Server passively listens
  - LISTEN and ACCEPT primitives
- Client executes CONNECT
  - Send a TCP Segment with SYN bit on and ACK bit off.
- Check to see if there is a process listening
  - If not send a RST
  - If yes, then give the segment to the process
  - If accepted – send an ACK message

# TCP Connection Establishment



(a) TCP connection establishment in the normal case.

(b) Call collision.

# Connection Release

- Think of the connection as a pair of simplex connections
  - Each is released independently
- Either party sends a segment with FIN bit set
- When FIN acked that direction is shut down

# TCP Connection Management Modeling

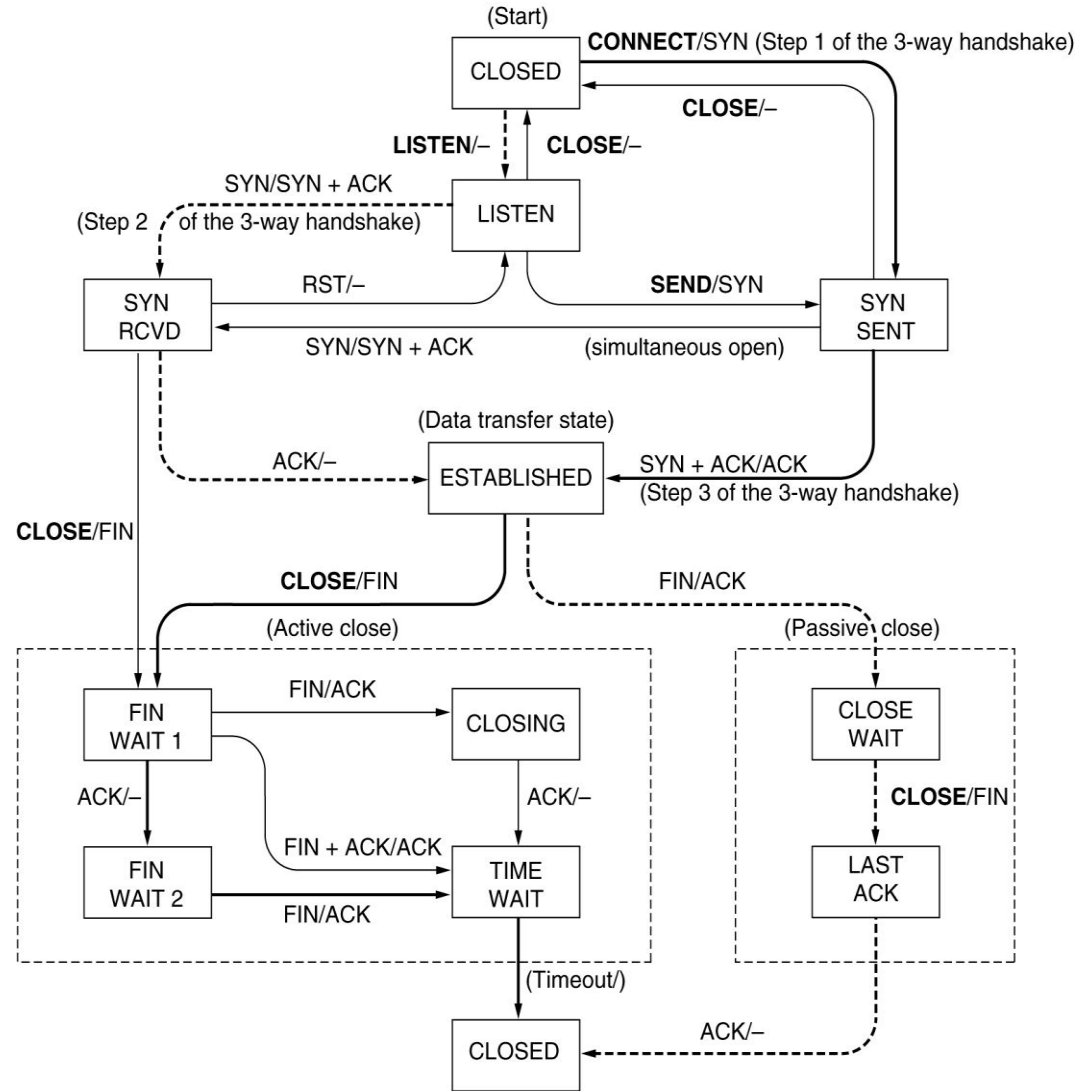
The

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

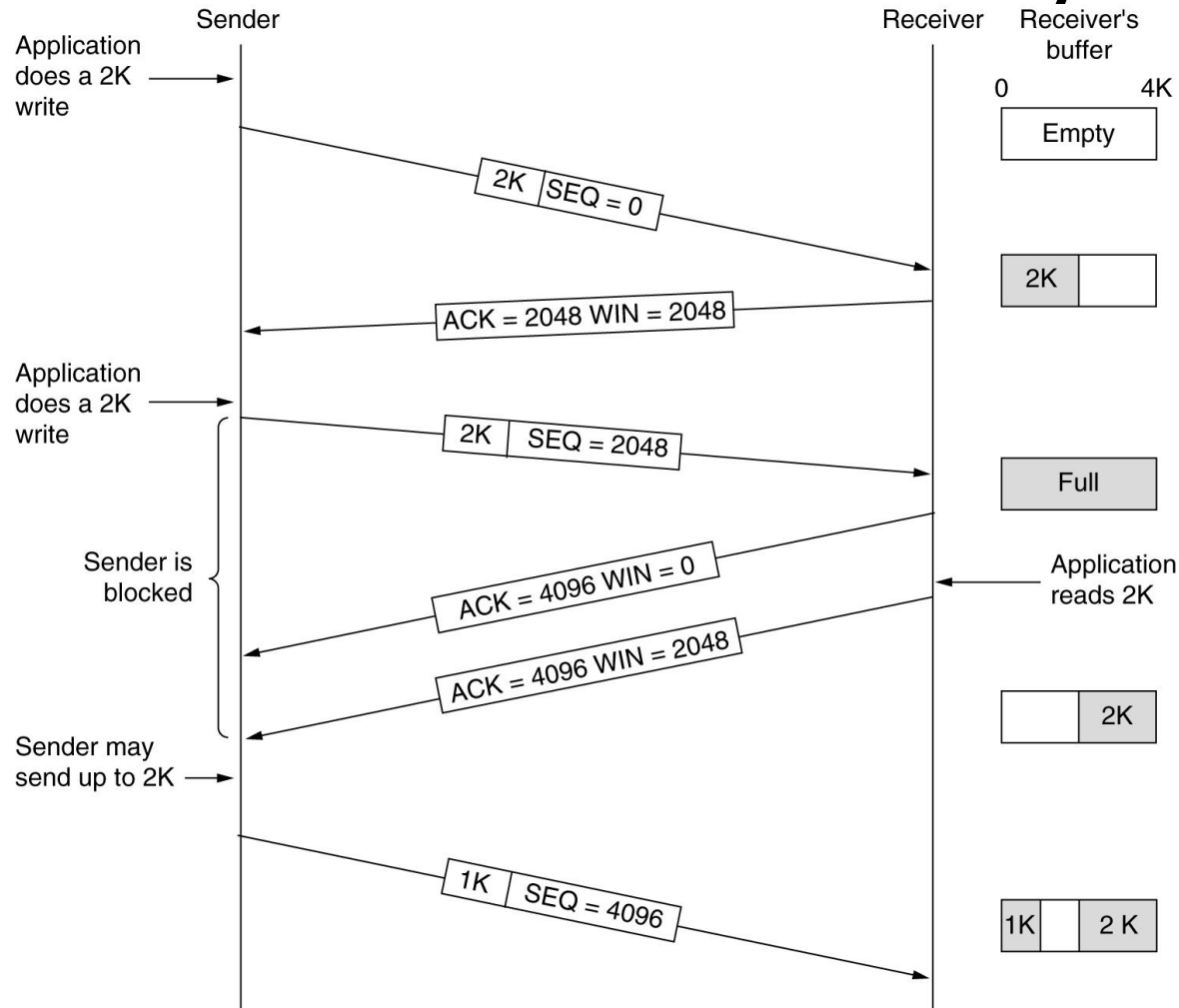
ment

# TCP Connection Management Modeling (2)

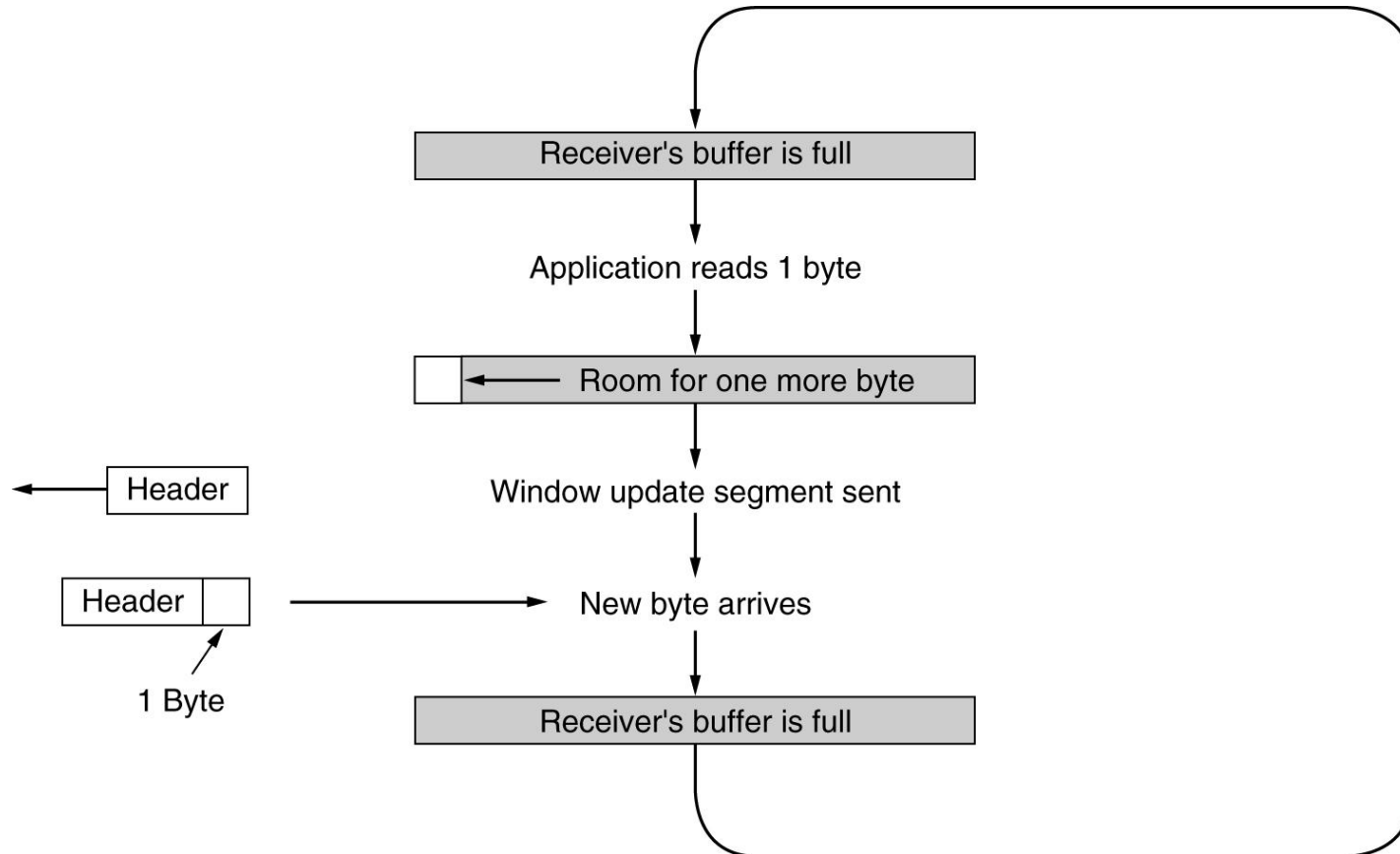
TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled by the event causing it and the action resulting from it, separated by a slash.



# TCP Transmission Policy



# TCP Transmission Policy



Silly window syndrome.

# Handling Silly Window Problem

- Delay ack and window updates for 500 ms.
- Nagle's Algorithm
  - When data comes in one byte at a time
    - Send the first byte and buffer the rest till the outstanding byte is acknowledged
    - Then send all the buffered characters in one TCP segment
  - Mouse movements have to be sent – Burst does not work well.
- Clark's Solution
  - Wait until decent amount of space available then advertise
    - Max segment size or half buffer
  - Sender not send tiny segments

# TCP Congestion Control

- Congestion – a function of total number of packets in the network, and where they are
- First step – detection
  - Is packet loss an indication of congestion??
  - All TCP algorithms assume timeouts are caused by congestion
- Initial steps
  - When connection is established – use suitable window size
    - Loss will not occur due to buffers at receiver
- Two issues
  - Network Capacity
  - Receiver Capacity

# TCP Congestion Control

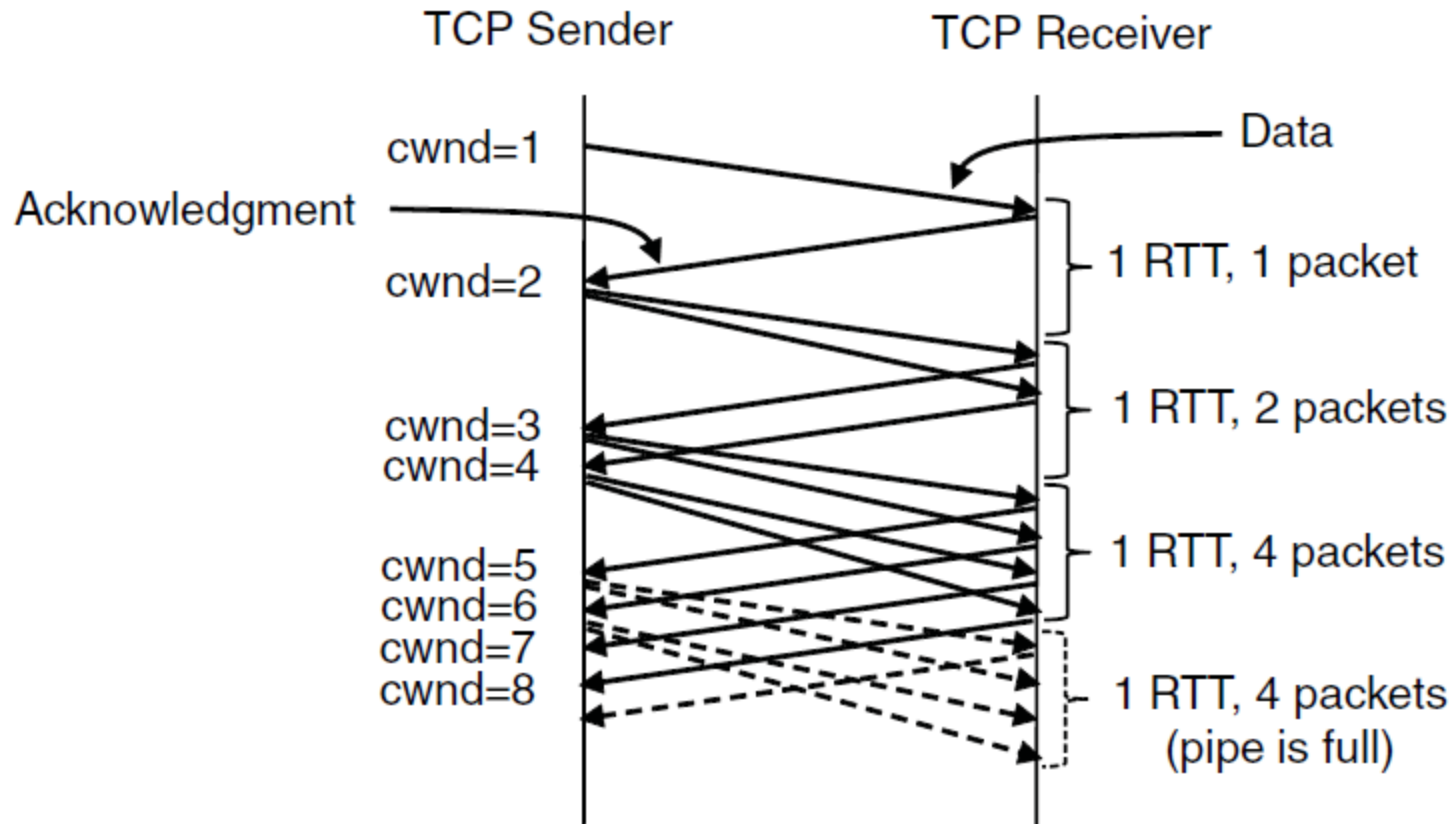
- Network Capacity and Receiver Capacity
- Maintain two windows
  - Receiver window
  - Congestion window
  - Use the min ( Receiver window and Congestion window)
- Initially
  - Sender sets congestion window to MSS (Max Seg Size)
  - If acked add one more MSS – 2 now
  - Repeat for each acked MSS
  - Congestion window grows exponentially
  - If timeout – go back to previous window size
  - SLOW START

# Internet Congestion Control

- Use a Threshold – initially 64 KB
- When a timeout occurs set threshold to half the current congestion window and reset congestion window to 1 MSS
- Use slow start till the threshold is reached
- Then successful transmissions grow congestion window linearly

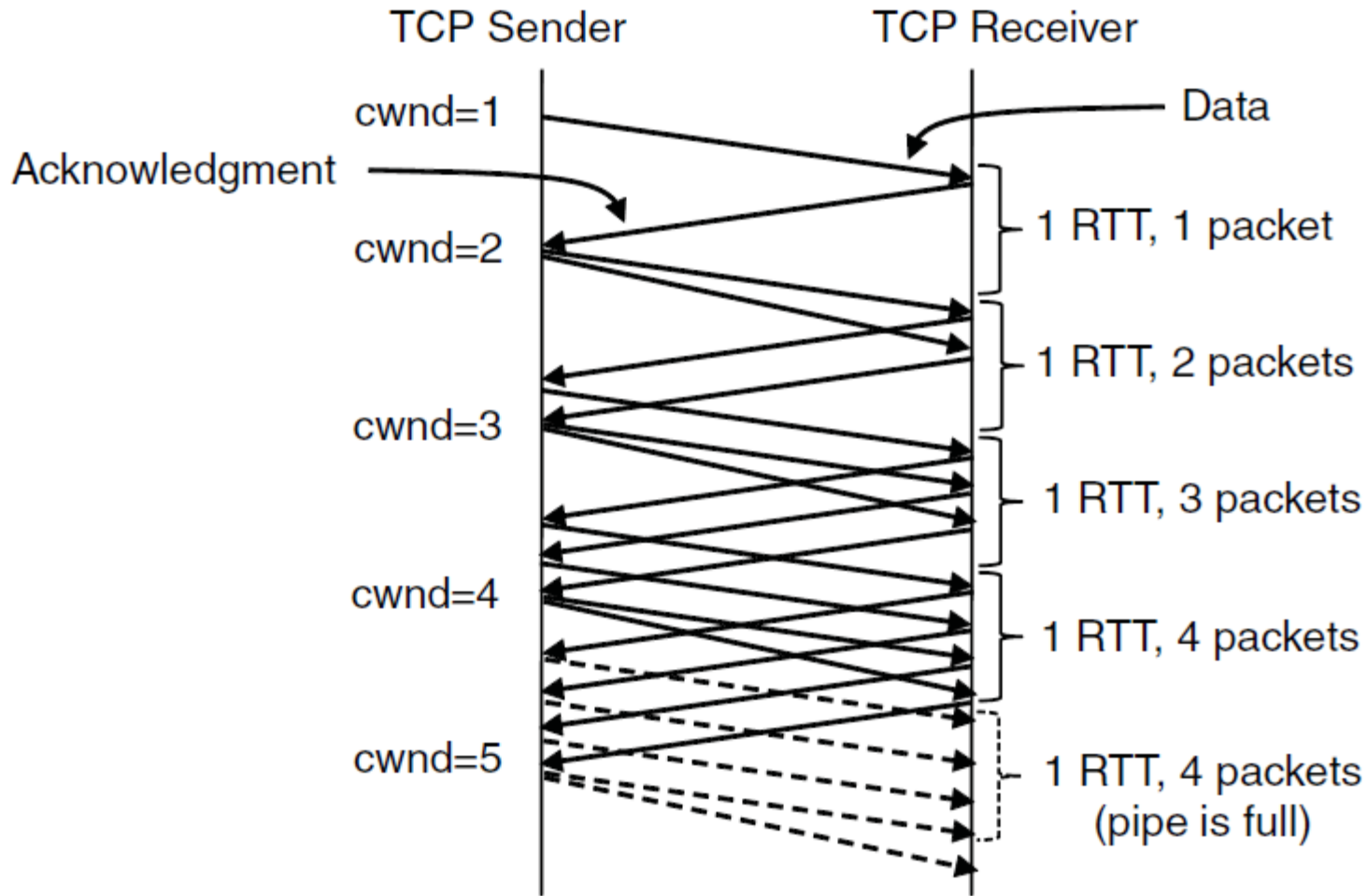
# TCP Congestion Control (1)

S



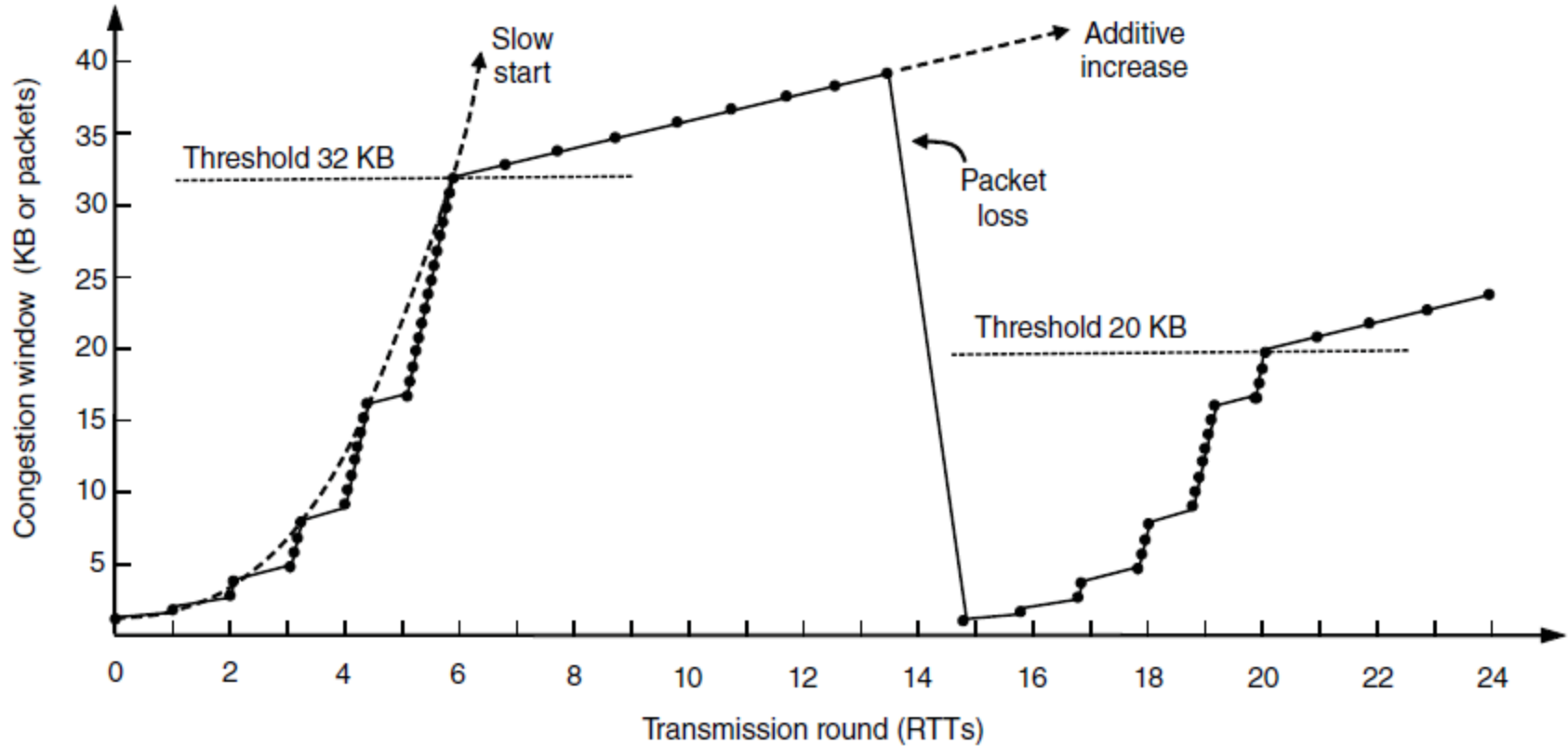
1

# TCP Congestion Control (2)



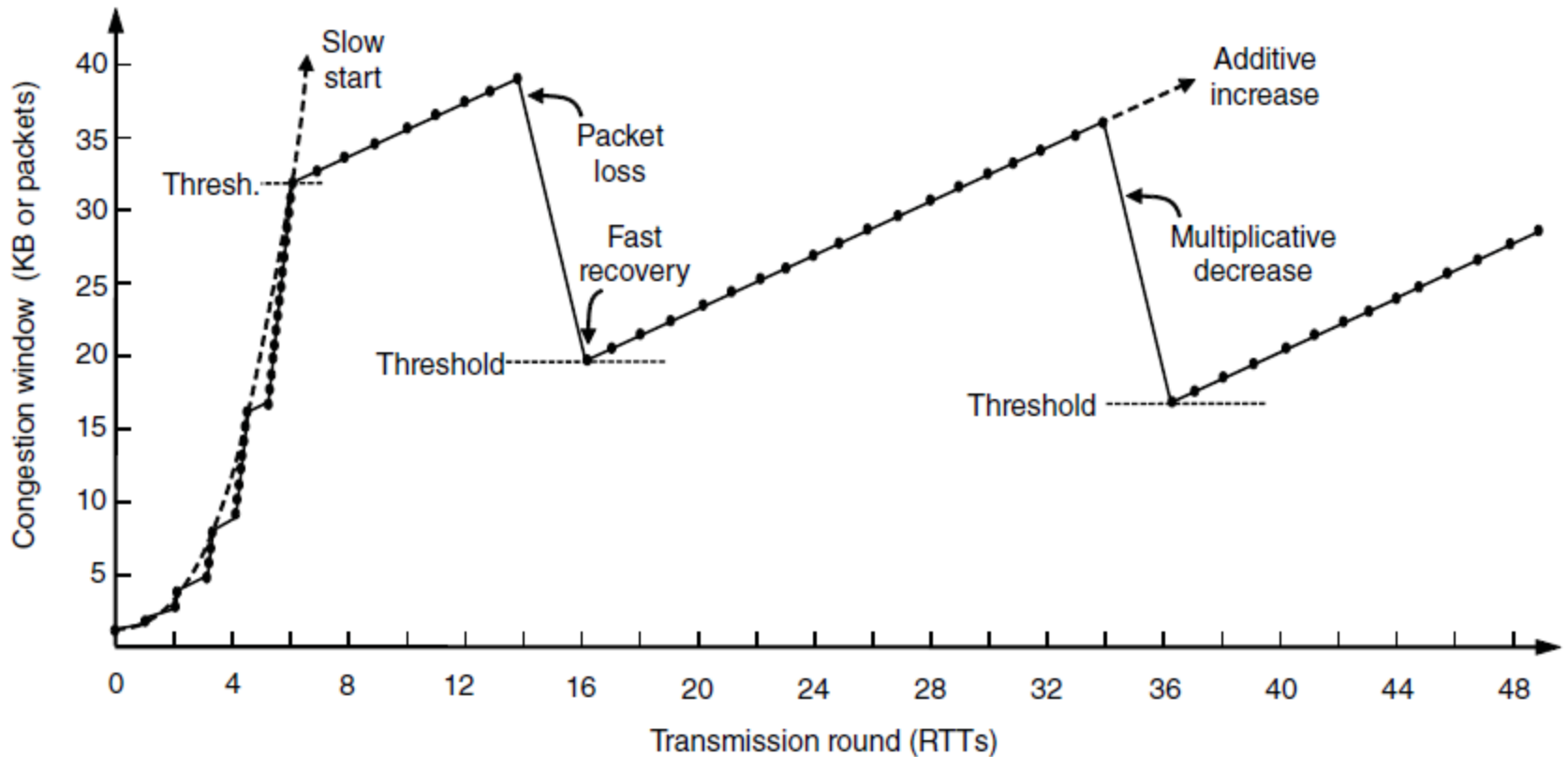
Additive increase from an initial congestion window of 1 segment.

# TCP Congestion Control (3)



Slow start followed by additive increase in TCP Tahoe.

# TCP Congestion Control (4)



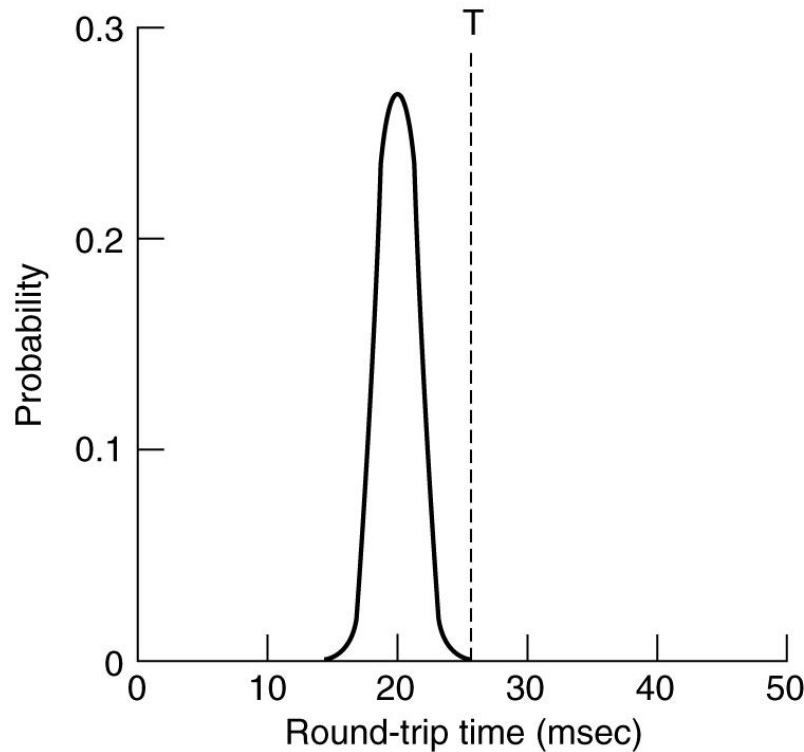
Fast recovery and the sawtooth pattern of TCP Reno.

# Timer Management

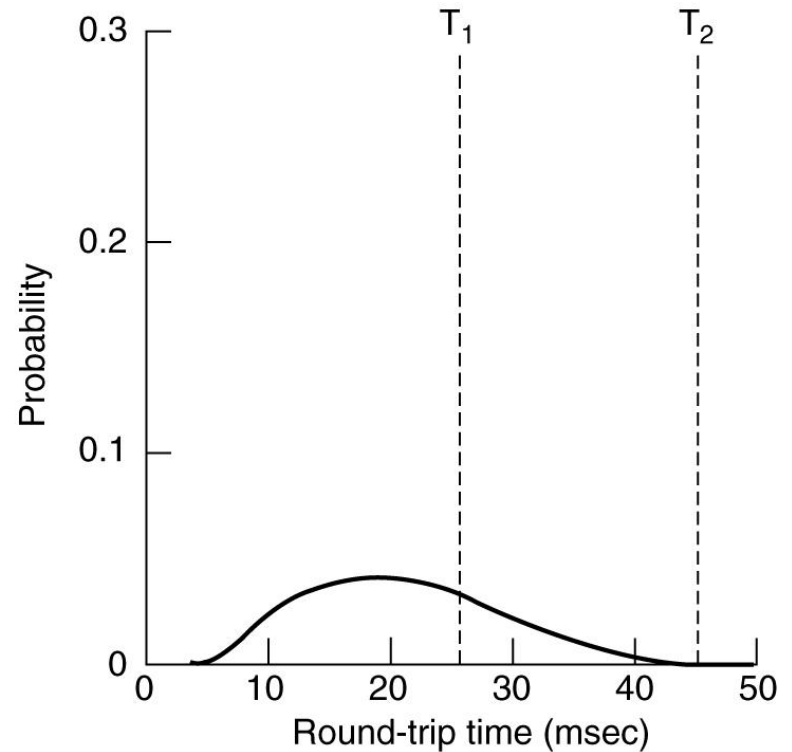
- TCP uses multiple timers
  - Most important is the *retransmission timer*
  - What value to set it at??
- Round Trip time
  - Highly variable
  - Varies over time
  - Have to track it
  - Estimate it
  - M is a new measurement
  - $\alpha = 7/8$
  - Use  $\beta RTT$  for retransmission timer
  - Initial values of  $\beta$  were 2 – make is proportional to standard deviation of M

$$RTT = \alpha RTT + (1 - \alpha)M$$

# TCP Timer Management



(a)



(b)

- (a) Probability density of ACK arrival times in the data link layer.
- (b) Probability density of ACK arrival times for TCP.

# Timer Management

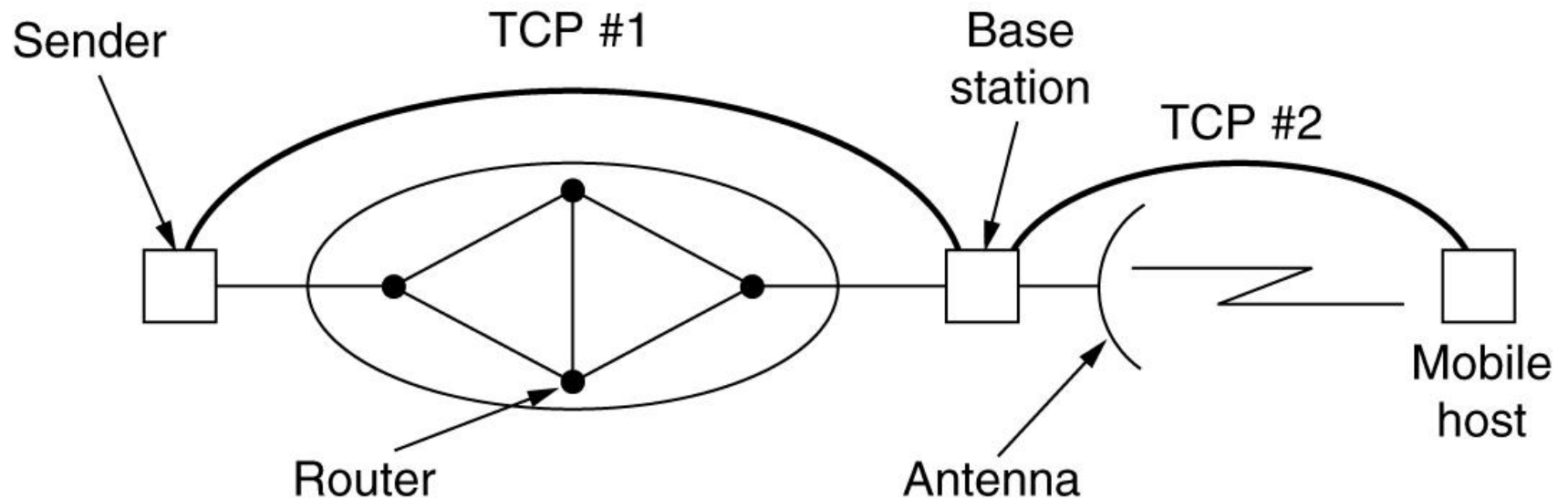
- Jacobson Approach
- Mean deviation estimate
- $D = \gamma D + (1-\gamma) |RTT - M|$
- Timeout = RTT + 4 D
  
- What to do on retransmissions
  - Do not know if the ack is for the first or second
- Karn Algorithm
  - Do not update RTT on any segments that have been retransmitted

# Persistence Timer

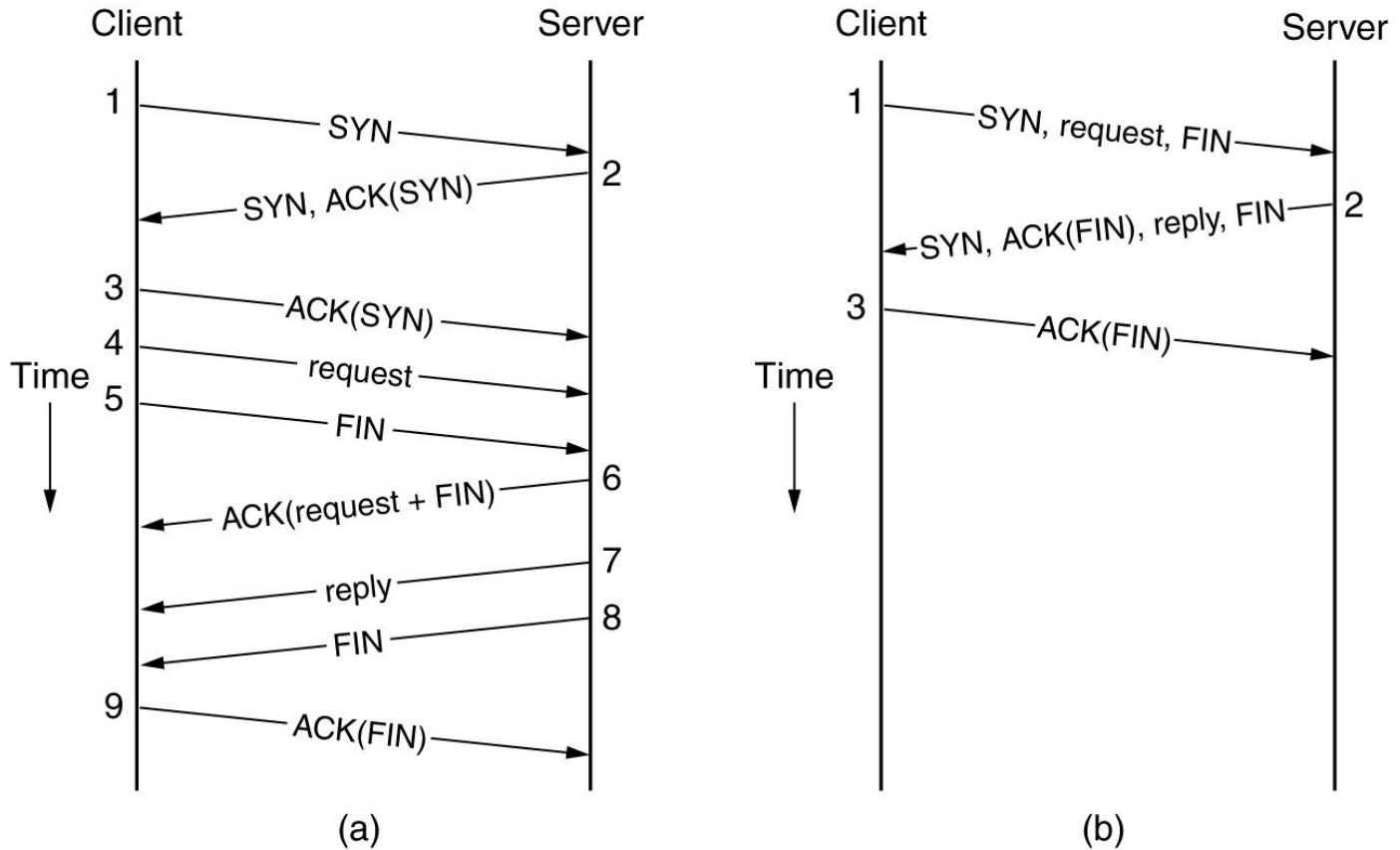
- Receiver sends a window of 0
- Later sends a window size but that packet is lost
  - Both wait
- Persistence Timer
  - When it goes off – sender sends a probe request to receiver to get a window size
  - If still zero – continue to wait and reset persistence timer
- Keepalive Timer
  - When a connection is idle for a long time – check if the other side is still there

# Wireless TCP and UDP

Splitting a TCP connection into two connections.



# Transactional TCP



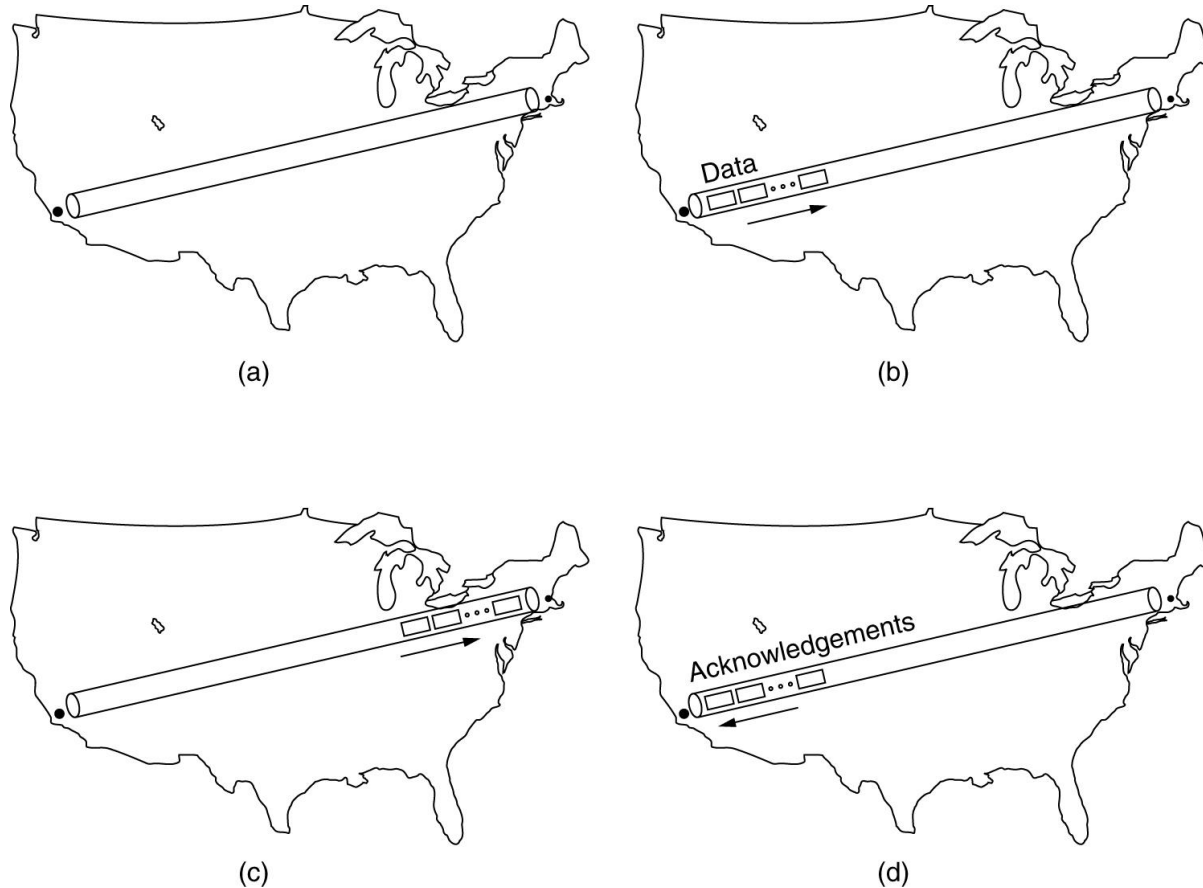
(a) RPC using normal TCP.

(b) RPC using T/TCP.

# Performance Issues

- Performance Problems in Computer Networks
- Network Performance Measurement
- System Design for Better Performance
- Fast TPDU Processing
- Protocols for Gigabit Networks

# Performance Problems in Computer Networks



The state of transmitting one megabit from San Diego to Boston  
(a) At  $t = 0$ , (b) After  $500 \mu\text{sec}$ , (c) After  $20 \text{ msec}$ , (d) after  $40 \text{ msec}$ .

# Network Performance Measurement

The basic loop for improving network performance.

1. Measure relevant network parameters, performance.
2. Try to understand what is going on.
3. Change one parameter.

# Network Performance Measurement (2)

## Issues in measuring performance

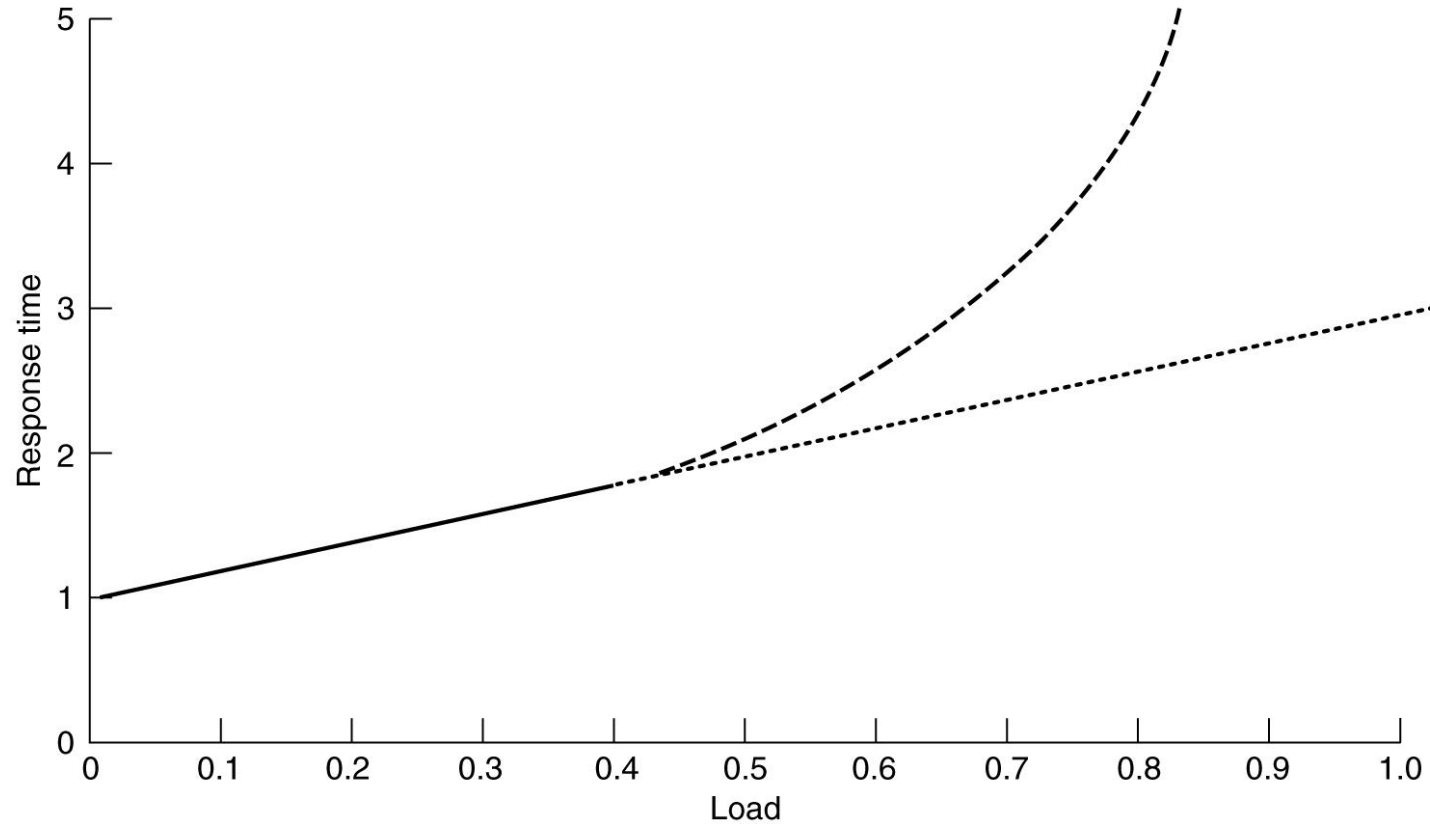
- Sufficient sample size
- Representative samples
- Clock accuracy
- Measuring typical representative load
- Beware of caching
- Understand what you are measuring
- Extrapolate with care

# System Design for Better Performance (1)

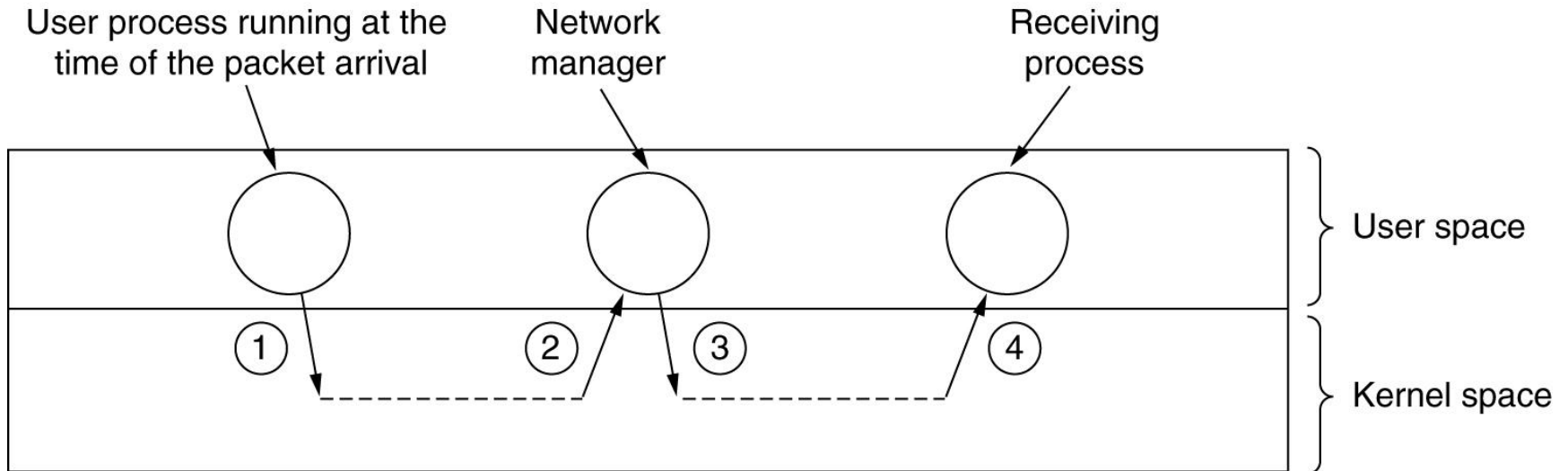
## Rules of thumb

1. CPU speed more important than network speed
2. Reduce packet count to reduce software overhead
3. Minimize data touching
4. Minimize context switches
5. Minimize copying
6. You can buy more bandwidth but not lower delay
7. Avoiding congestion is better than recovering from it
8. Avoid timeouts

# System Design for Better Performance (2)

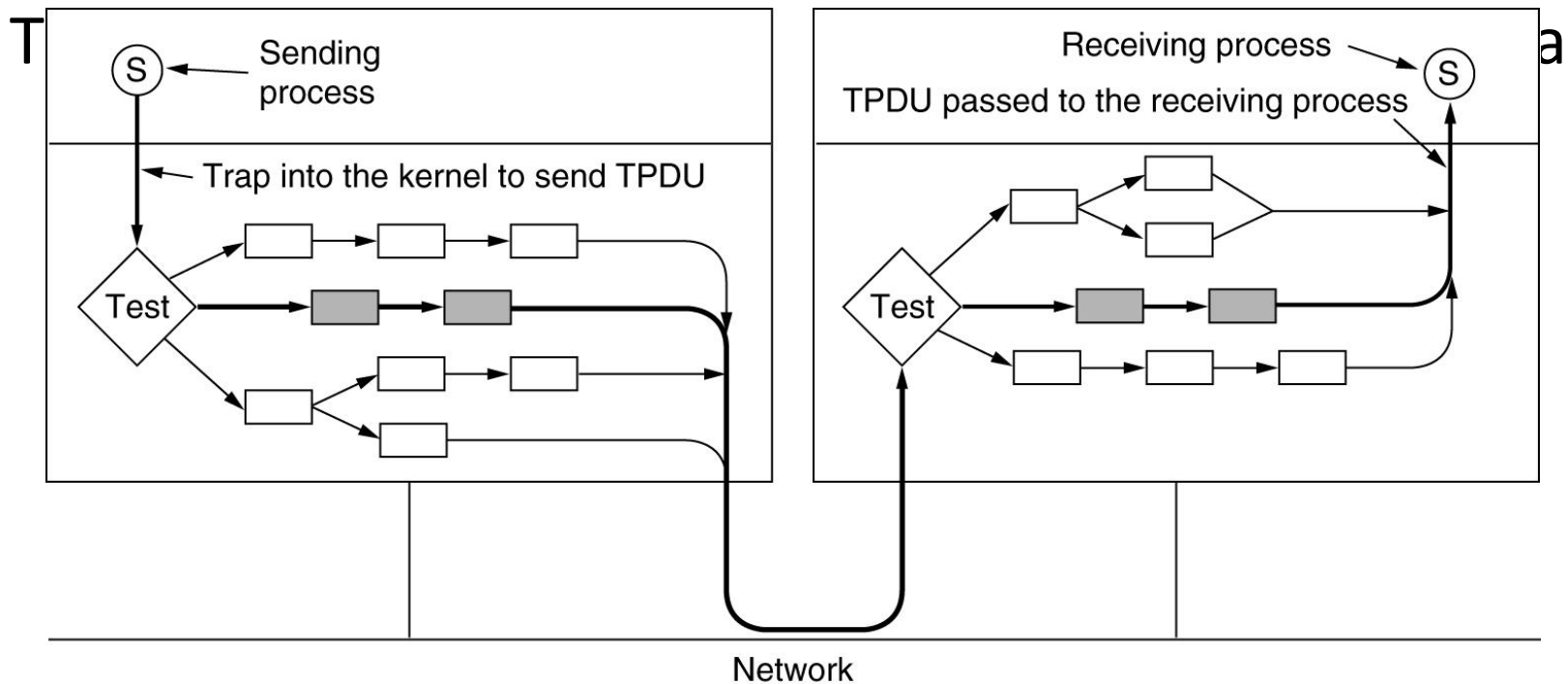


# System Design for Better Performance (3)



Four context switches to handle one packet  
with a user-space network manager.

# Fast TPDU Processing



# Fast TPDU Processing (2)

Source port		Destination port	
Sequence number			
Acknowledgement number			
Len	Unused	Window size	
Checksum		Urgent pointer	

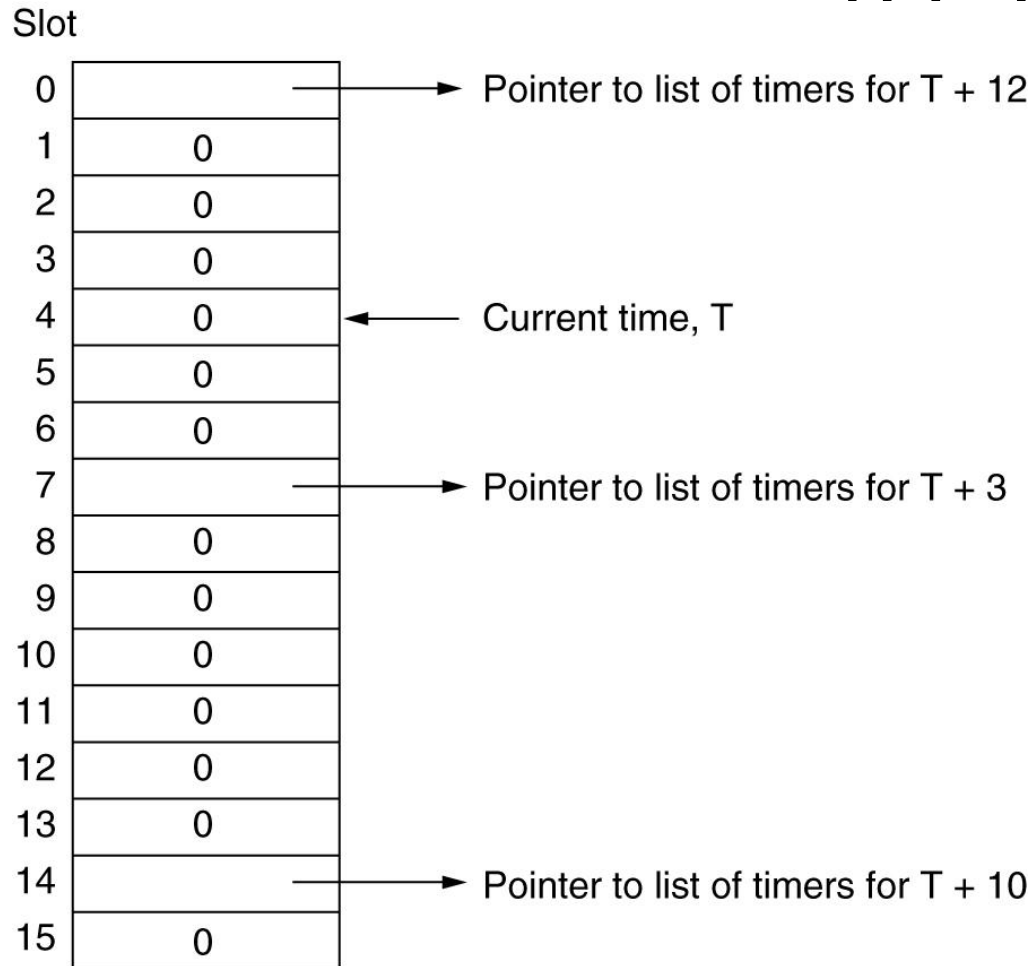
(a)

VER.	IHL	TOS	Total length	
Identification			Fragment offset	
TTL		Protocol	Header checksum	
Source address				
Destination address				

(b)

(a) TCP header. (b) IP header. In both cases, the shaded fields are taken from the prototype without change.

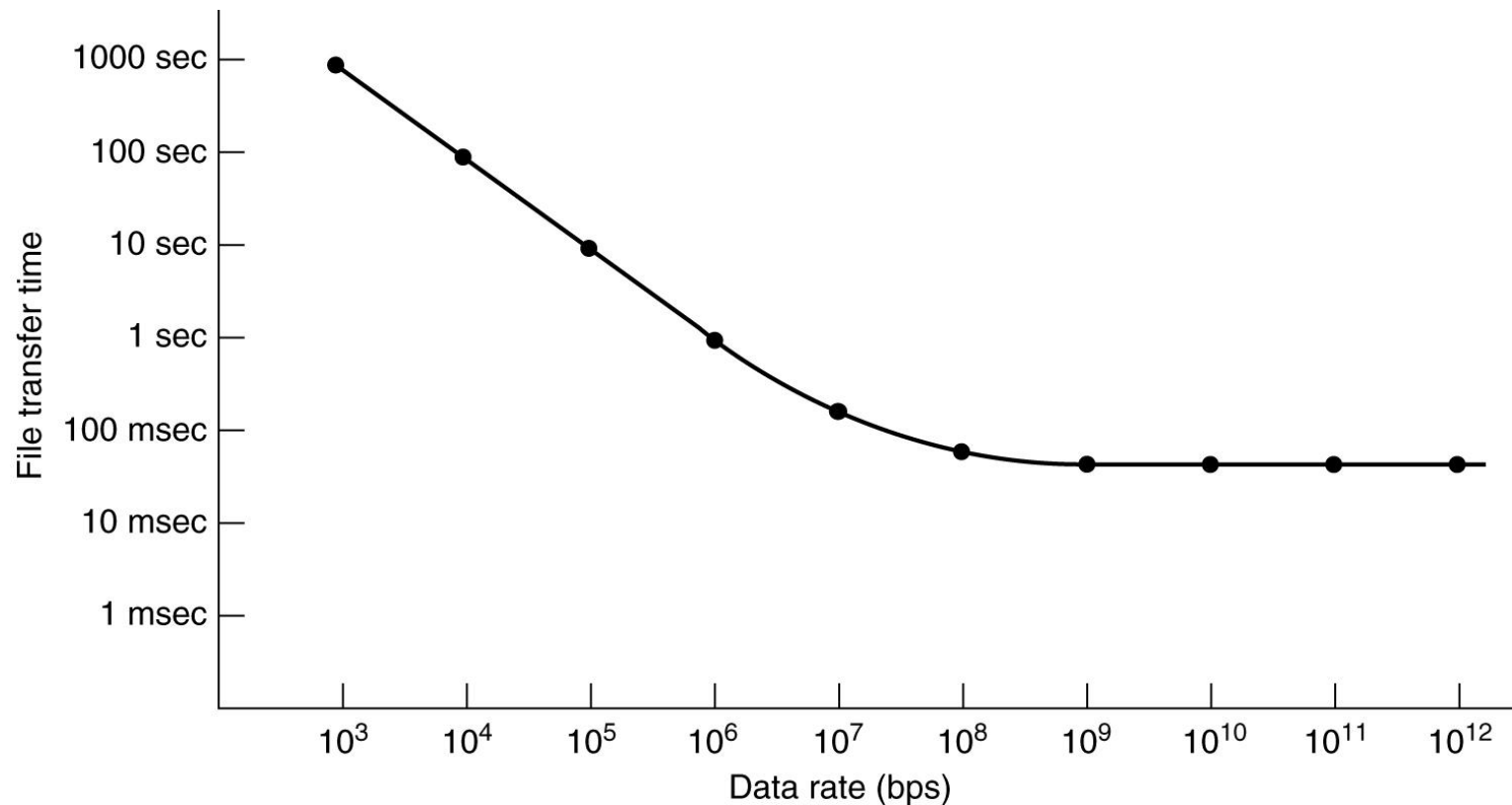
# Fast TPDU Processing (3)



A timing wheel.

# Protocols for Gigabit Networks

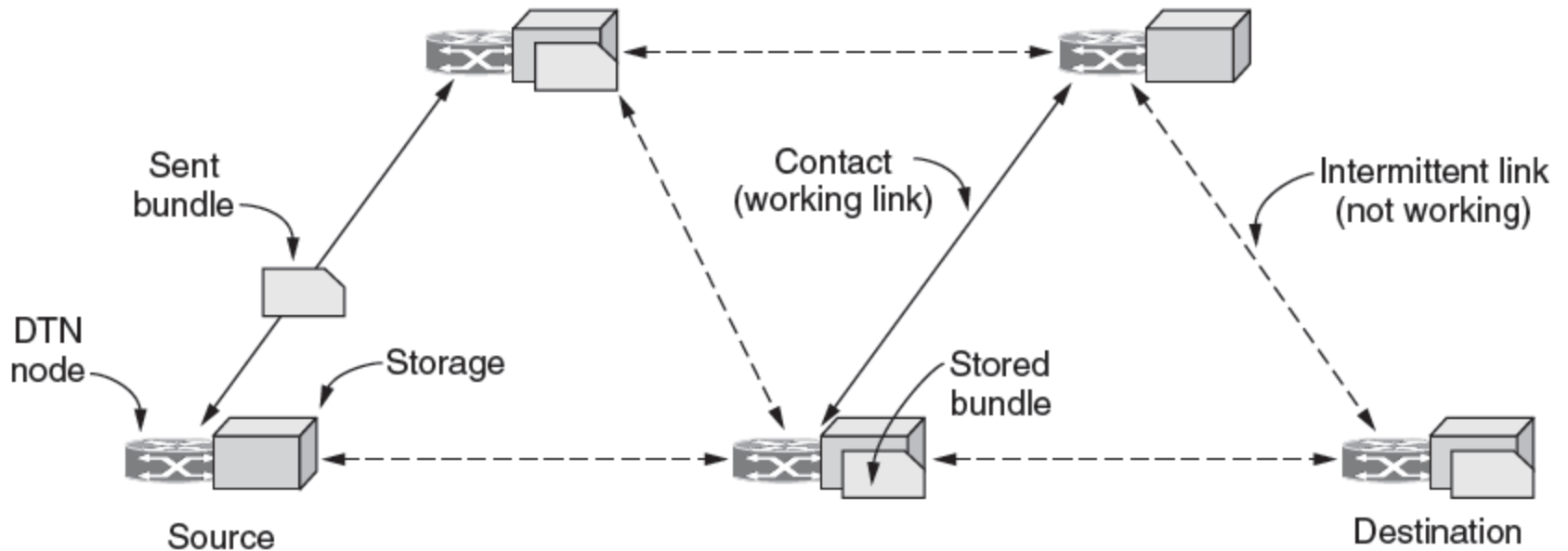
Time to transfer and acknowledge a 1-megabit file over a 4000-km line.



# Delay Tolerant Networking

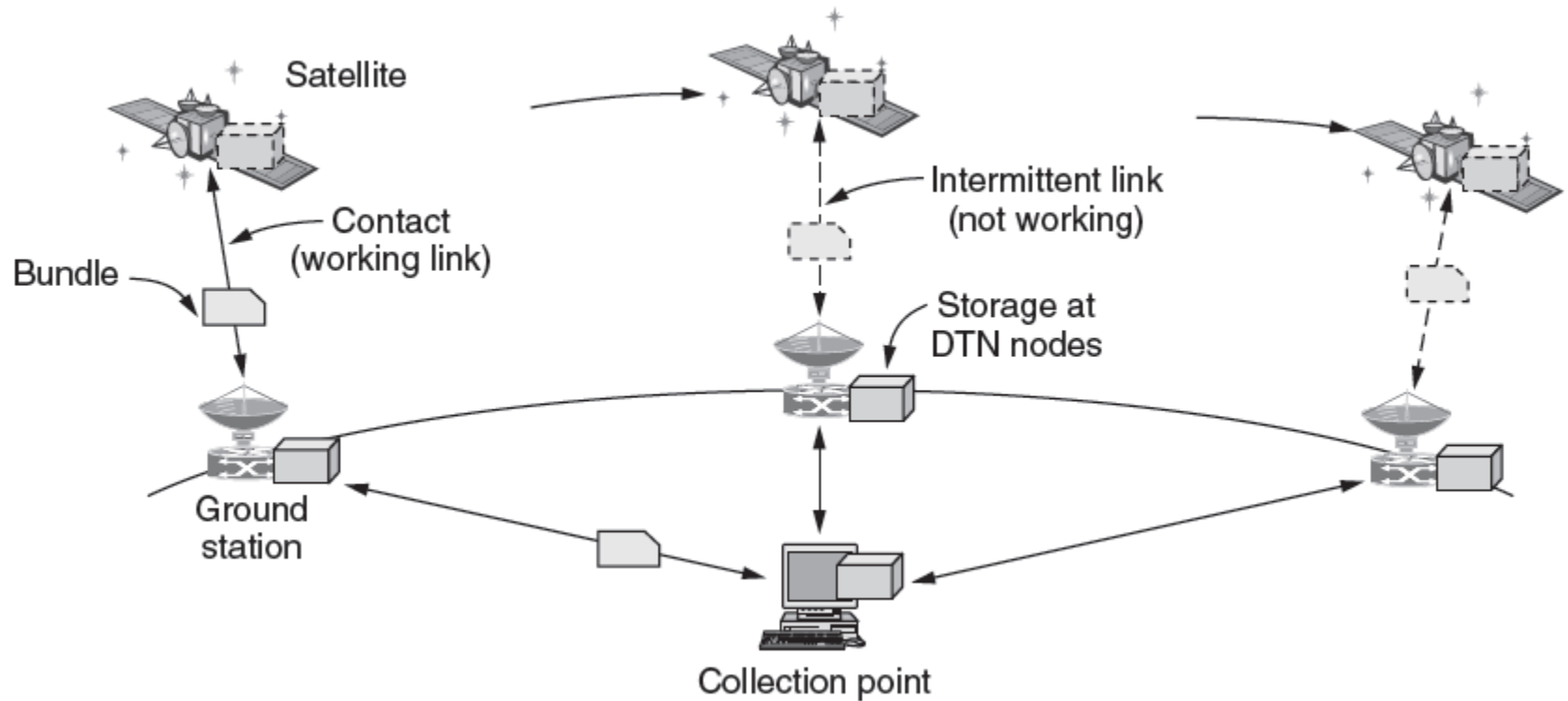
- DTN Architecture
- The Bundle Protocol

# DTN Architecture (1)



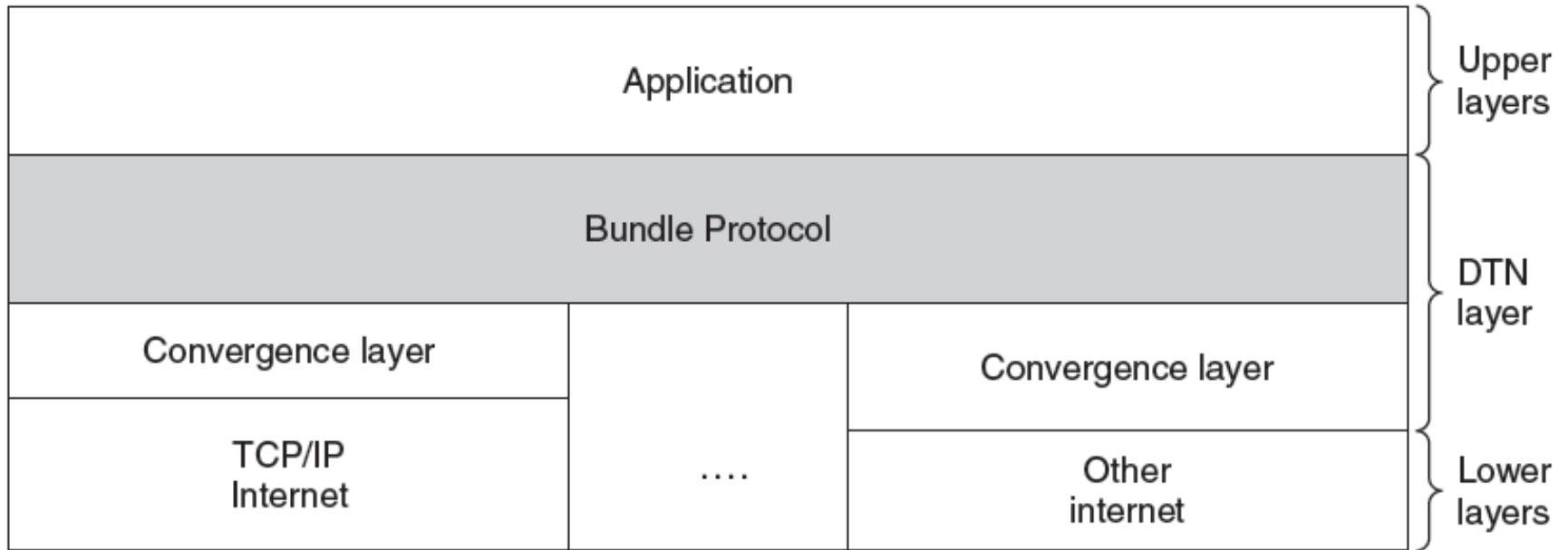
Delay-tolerant networking architecture

# DTN Architecture (2)



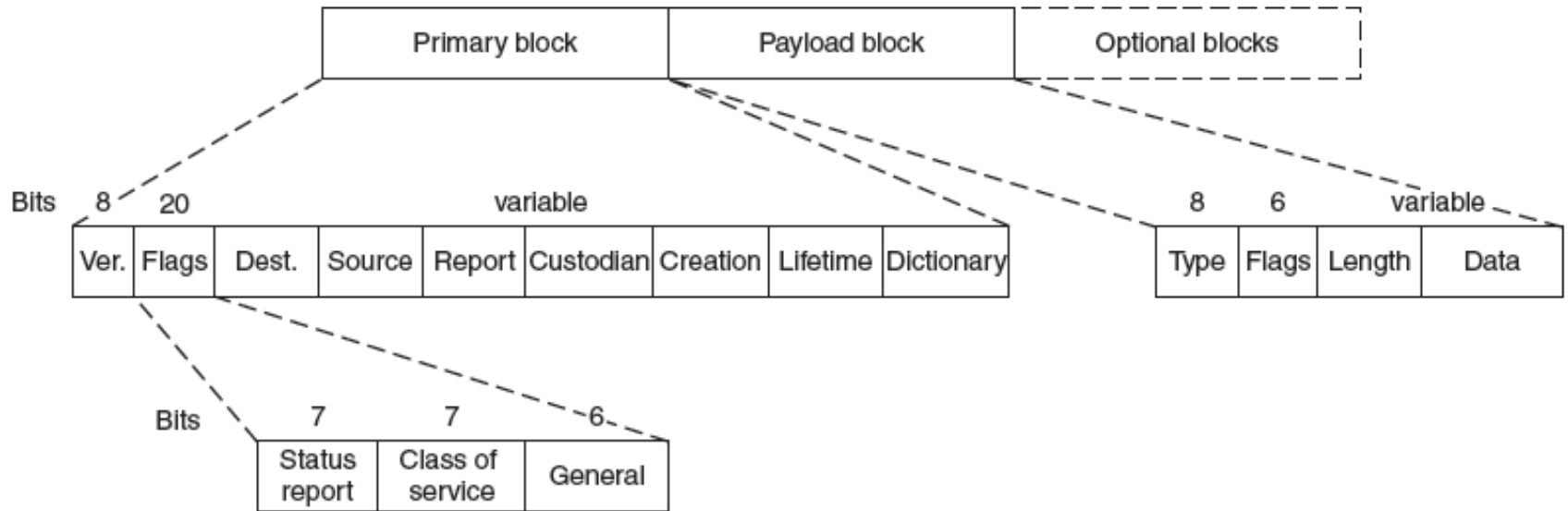
Use of a DTN in space.

# The Bundle Protocol (1)



Delay-tolerant networking protocol stack.

# The Bundle Protocol (2)



Bundle protocol message format.