CSMC 417

Computer Networks

Prof. Ashok K Agrawala

Set 7

Spring 2003

Chapter 6

The Transport Layer

Spring 2003

Transport Layer

- Goal: provide error free end-to-end delivery of data
  - provide in-order delivery over unreliable network layer
- Issues:
  - checking packet integrity
  - re-transmission of lost of corrupt packets
  - connection establishment and management
  - addresses
    - need to define a host plus process
    - typical abstraction is <host, port>
  - byte vs. packet transport service
    - byte service
      - bytes are in order, but packet boundaries are lost
      - used by TCP
    - packet service
      - preserve packet boundaries

Transport Service

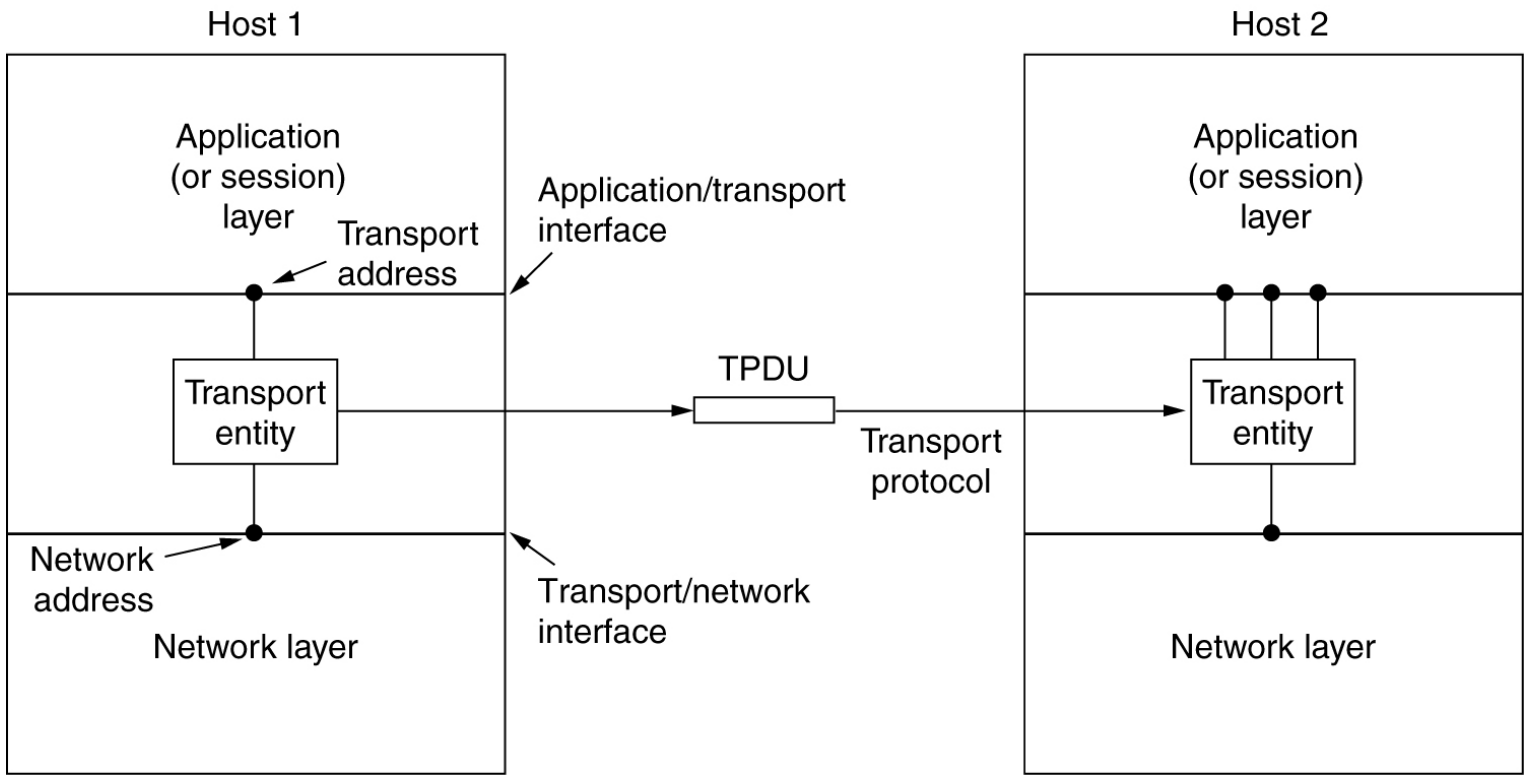
- Efficient
- Reliable
- Cost effective
- Service to its users - processes in the application layer

- Transport Entities can be
  - in Operating System Kernel
  - In a separate user process
  - in a library package bound into applications
  - on a network card
  - On Interface machines

The Transport Service

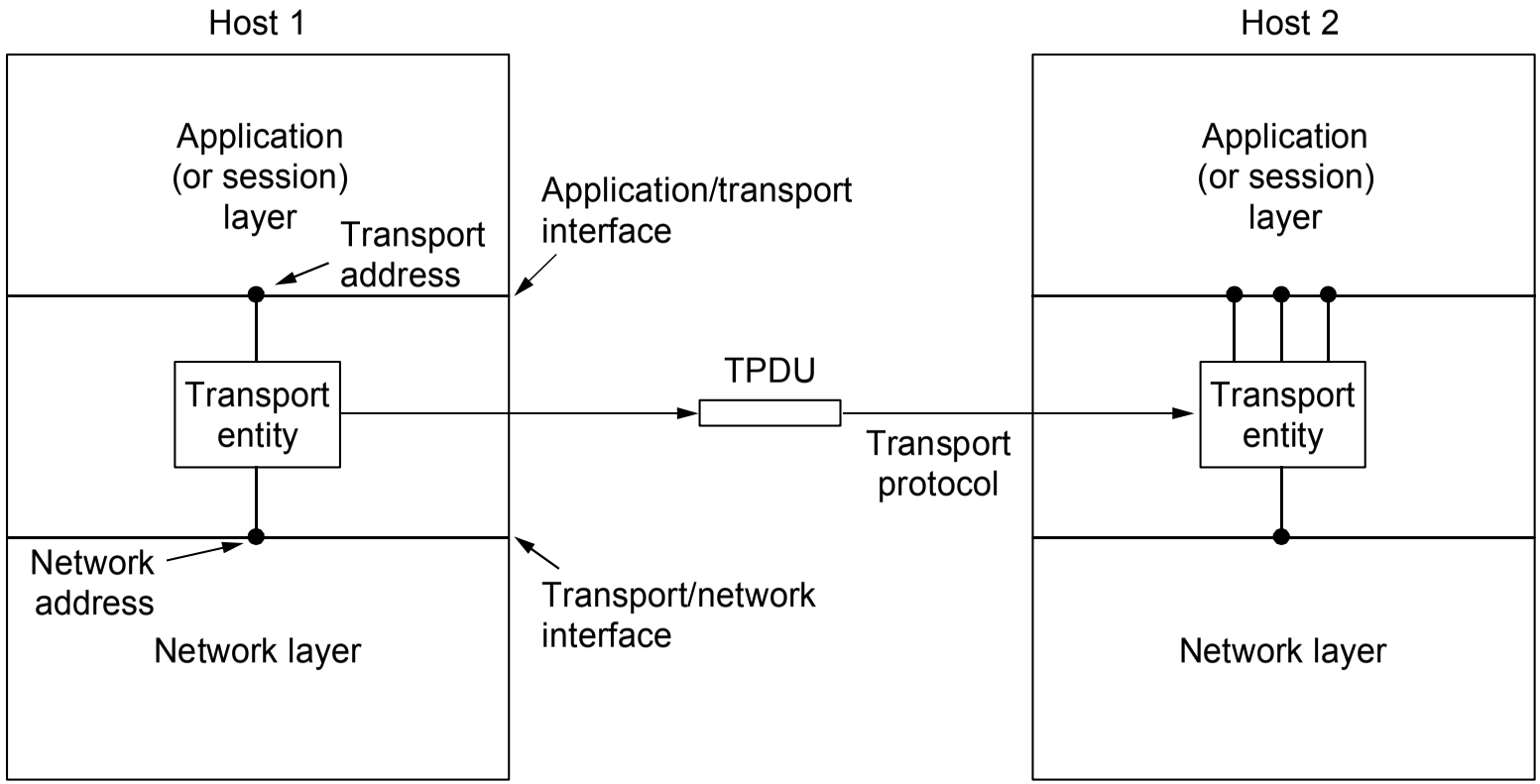
- Services Provided to the Upper Layers
- Transport Service Primitives
- Berkeley Sockets
- An Example of Socket Programming:
  - An Internet File Server

Services Provided to the Upper Layers



The network, transport, and application layers.

Layer Structure



Types of Service

- Connection Oriented
  - Establishment
  - Data transfer
  - Release
- Connection-less

- Transport Layer - part of the host not the subnet

Quality of Service

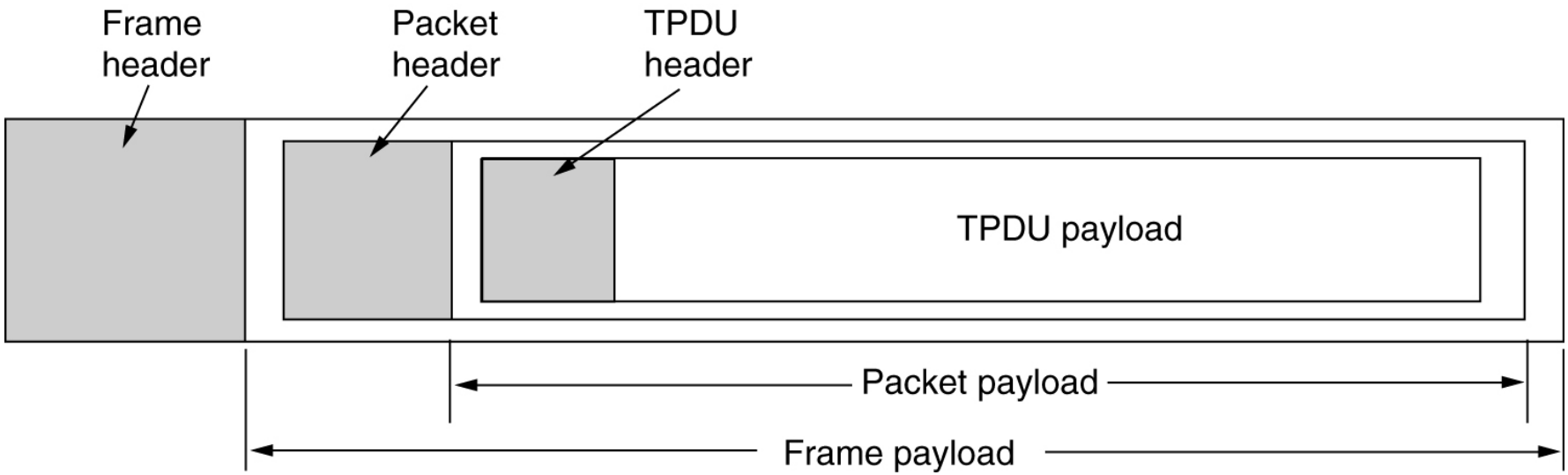
- Parameters
  - Connection Establishment Delay
  - connection Establishment Failure Probability
  - Throughput
  - Transit Delay
  - Residual Error Ratio
  - Protection
  - Priority
  - Resilience
    - Probability of the transport layer terminating a connection due to internal problems

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

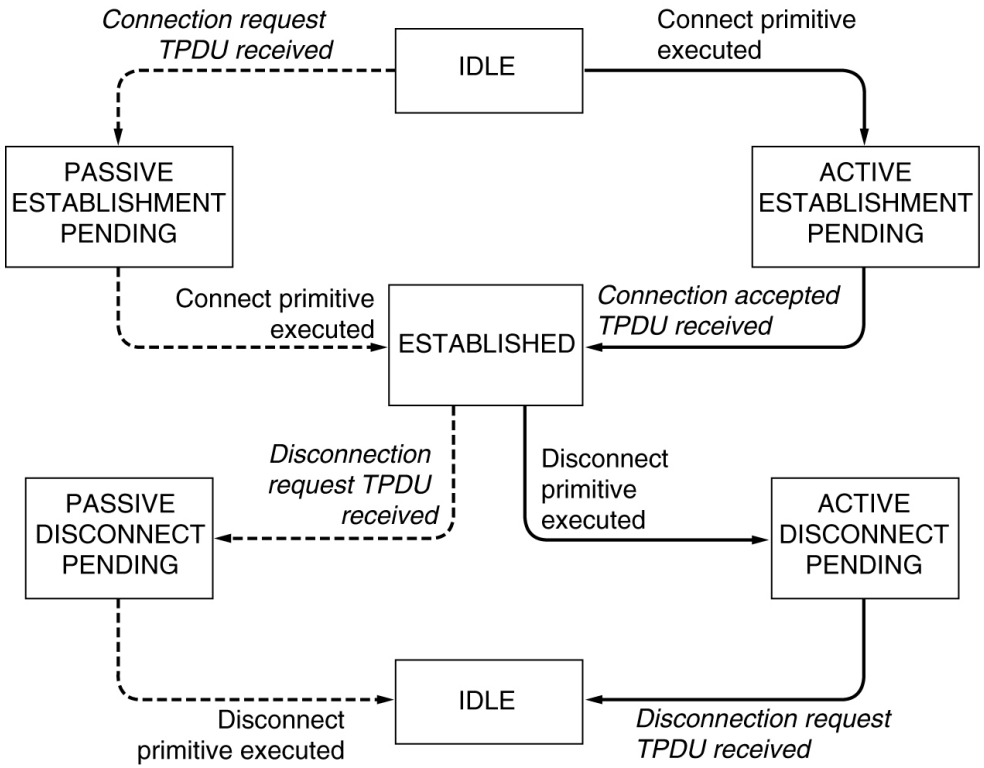
The primitives for a simple transport service.

Transport Service Primitives (2)



The nesting of TPDUs, 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

Primitive	Meaning
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 socket primitives for TCP.

Socket Programming Example: Internet File Server

Client code using sockets.

```
/* 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);
}
```

Socket Programming Example: Internet File Server (2)

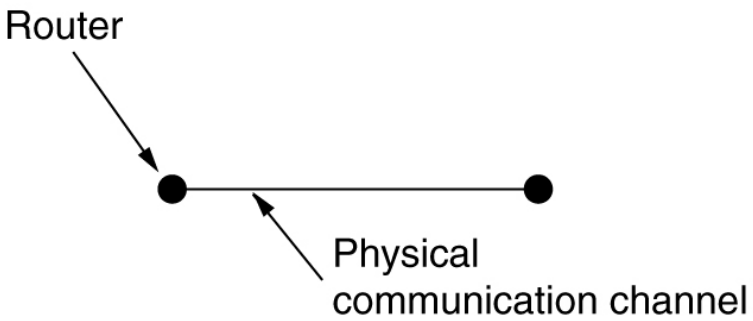
Client code using sockets.

```
#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 */
  }
}
```

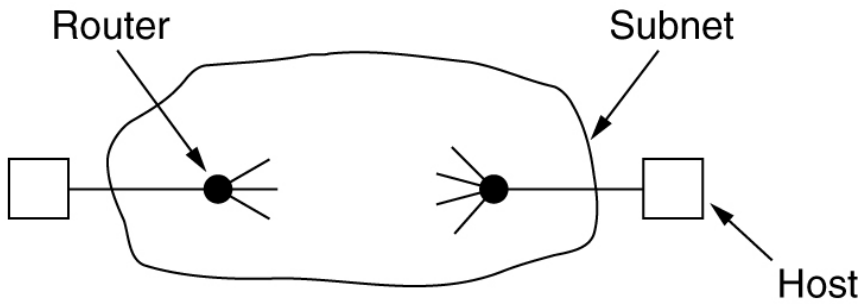
Elements of Transport Protocols

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

Transport Protocol



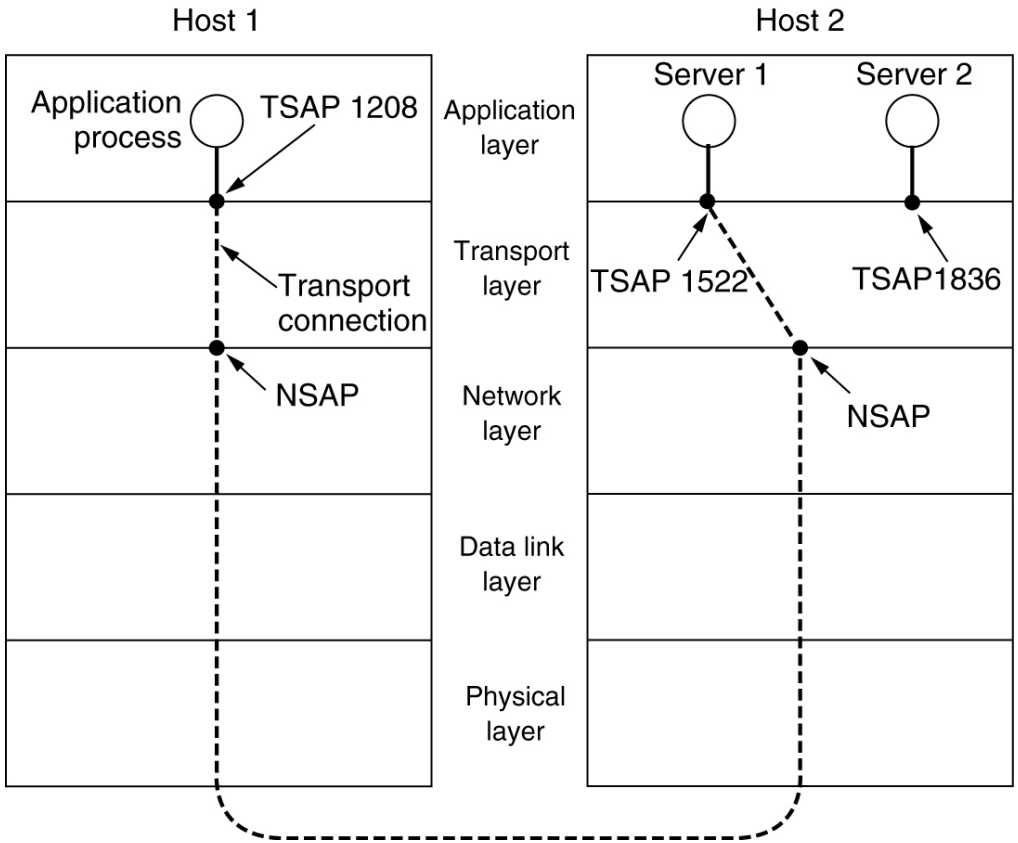
(a)



(b)

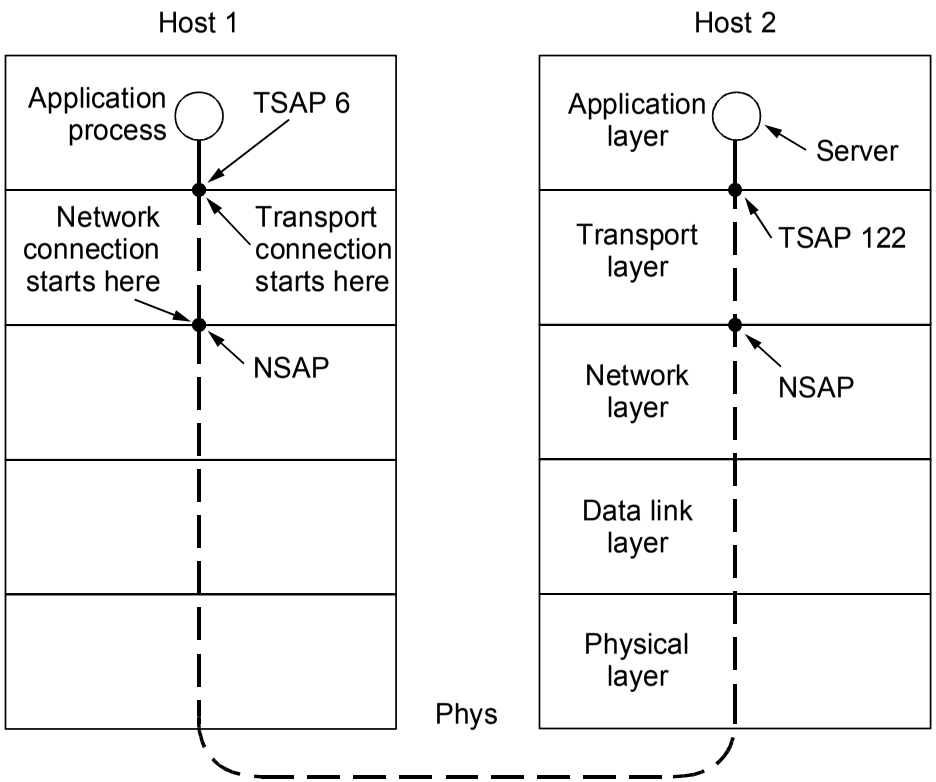
(a) Environment of the data link layer.
(b) Environment of the transport layer.

Addressing



TSAPs, NSAPs and transport connections.

Addressing



App process attaches to TSAP 122 on host 2

App process on host one wants to connect to the app process on host 2 at TSAP 122 and issues a connection request

Transport entity establishes a connection with local NSAP and establishes a network connection to NSAP on host 2

Transport entities talk to each other to establish a connection

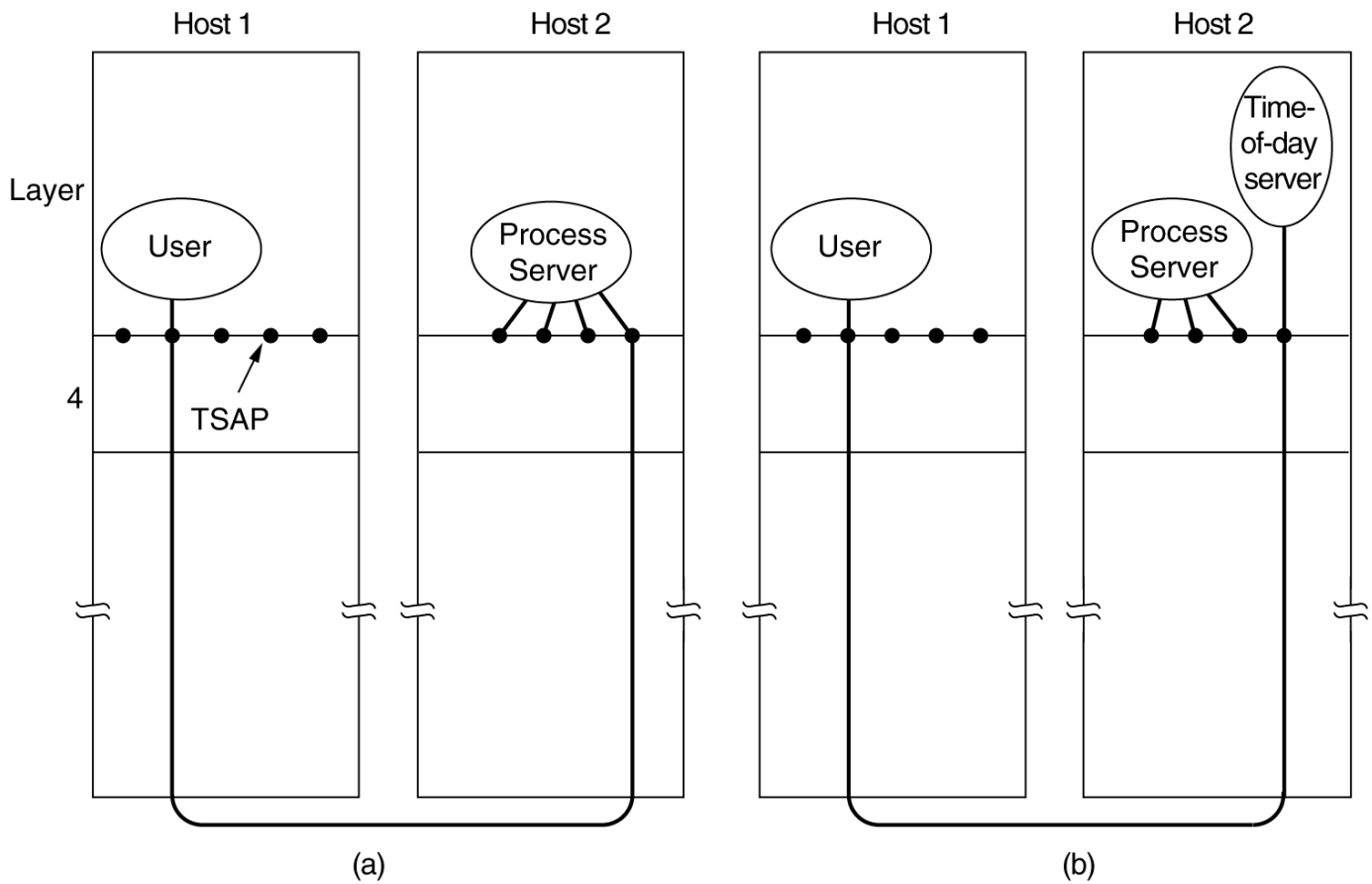
On host 2 the app process confirms the acceptance of the connection for it to be established

From: *Computer Networks*, 3rd Ed. by Andrew S. Tanenbaum, (c)1996 Prentice Hall.

Addressing

- How to know a remote TSAP address??
- Unix uses initial connection protocol
  - each machine that offers a service has a special process server which acts as a proxy
  - It listens to a set of ports at the same time waiting for a TCP connection request
  - Users begin by doing a connect request for that TSAP address (TCP port)
    - If no server is waiting for them they get a connection to the process server
  - On receiving a request it spawns off a server
- Name Server

Connection Establishment



How a user process in host 1 establishes a connection with a time-of-day server in host 2.

Connection Establishment

- Basic Idea
  - Send a connect request TPDU and wait for response
- Network may lose, delay, and duplicate packets

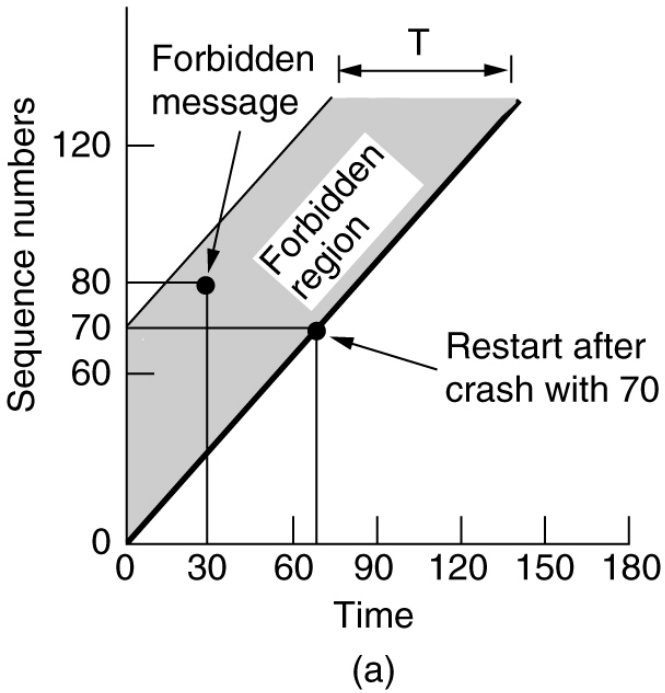
- Connection Identifiers
  - Each connection has a new identifier
  - Each entity has to maintain state information
    - What if a machine crashes??
- Limit packet lifetime
  - Through subnet design
  - Hop counter
  - time stamping each packet

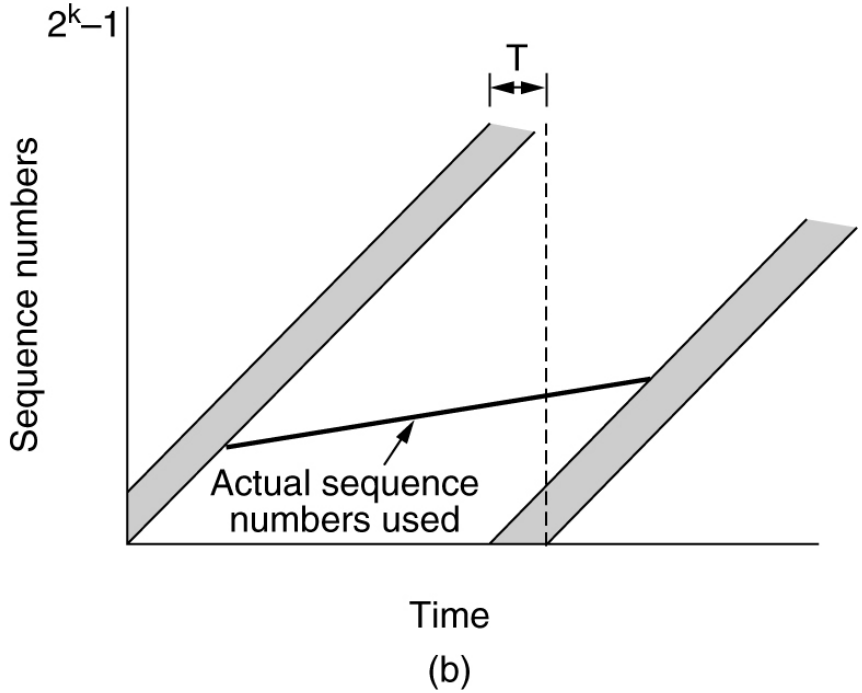
  - All acks must be dead also

Packet Life Time

- T a time when all traces of the packet and its acknowledgements have disappeared
  - T is a multiple of the packet life time
- Establishing connections properly requires a guarantee of packet life times.
- When a connection is set up the low order k bits of the clock are used at the initial sequence number.

Connection Establishment (2)





(a) TPDUs may not enter the forbidden region.

(b) The resynchronization problem.

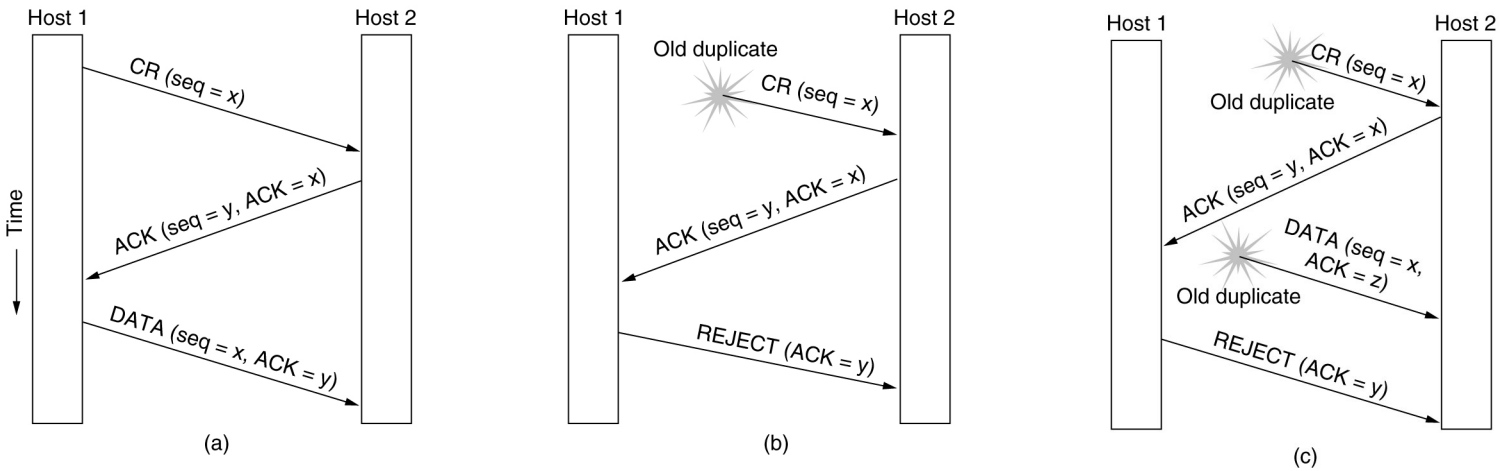
Duplicate Packets

- Issue: packets can be lost or duplicated
  - need to detect duplicates
  - need to re-send lost packets
    - but how do we know they are not just delayed?
- Solution 1
  - use a sequence number
    - each new packet uses a new sequence number
    - can detect arrival of stale packets
  - problem: when node crashes, sequence number resets
- Solution 2
  - use a clock for the sequence number
    - clocks don't reset on reboot, so we never lose sequence #
  - use a max lifetime for a packet
    - permits clocks to roll over
  - can get into forbidden region

Three-way Handshake

- Use different sequence number spaces for each direction
- Three messages used
  - Connection Request
    - send initial sequence number from caller to callee
  - Connection Request Acknowledgment
    - send ACK of initial sequence number from caller to callee
    - send initial sequence number from callee to caller
  - First Data TPDU
    - send ACK of initial sequence number from callee to caller
- Each Side Selects an initial number
  - it knows that the number is not currently valid
    - uses time of day
    - limits number of connects per unit time, but not data!

Connection Establishment (3)



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

(a) Normal operation,

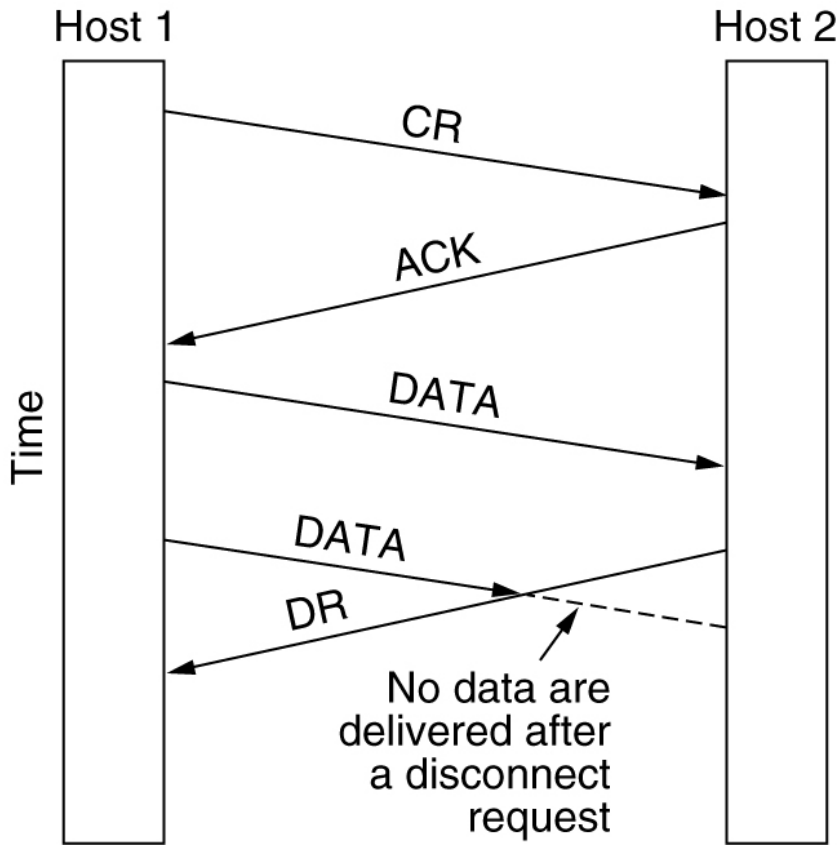
(b) Old CONNECTION REQUEST appearing out of nowhere.

(c) Duplicate CONNECTION REQUEST and duplicate ACK.

Closing a Connection

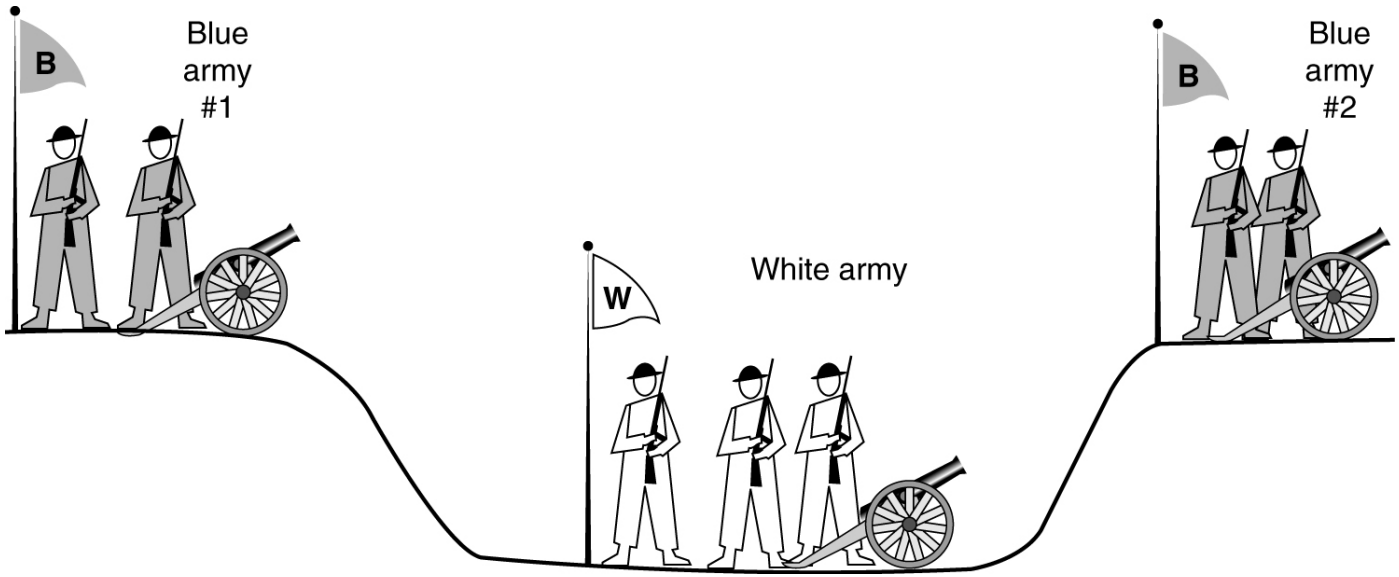
- To prevent data loss,
  - both sides must agree they are done
- Problem: how to agree
  - possible that "I am done" messages will get lost
  - possible that "I ACK you are done" messages will get lost
- Solution:
  - initiator sends Disconnect Request, start DR timer
  - when initiated party receives DR, send DR and start DR timer
  - when initiator gets DR back, send ACK and release connection
  - when initiated gets ACK, release connection
  - if initiator times out, send new DR
  - if initiated times out, release connection

Connection Release



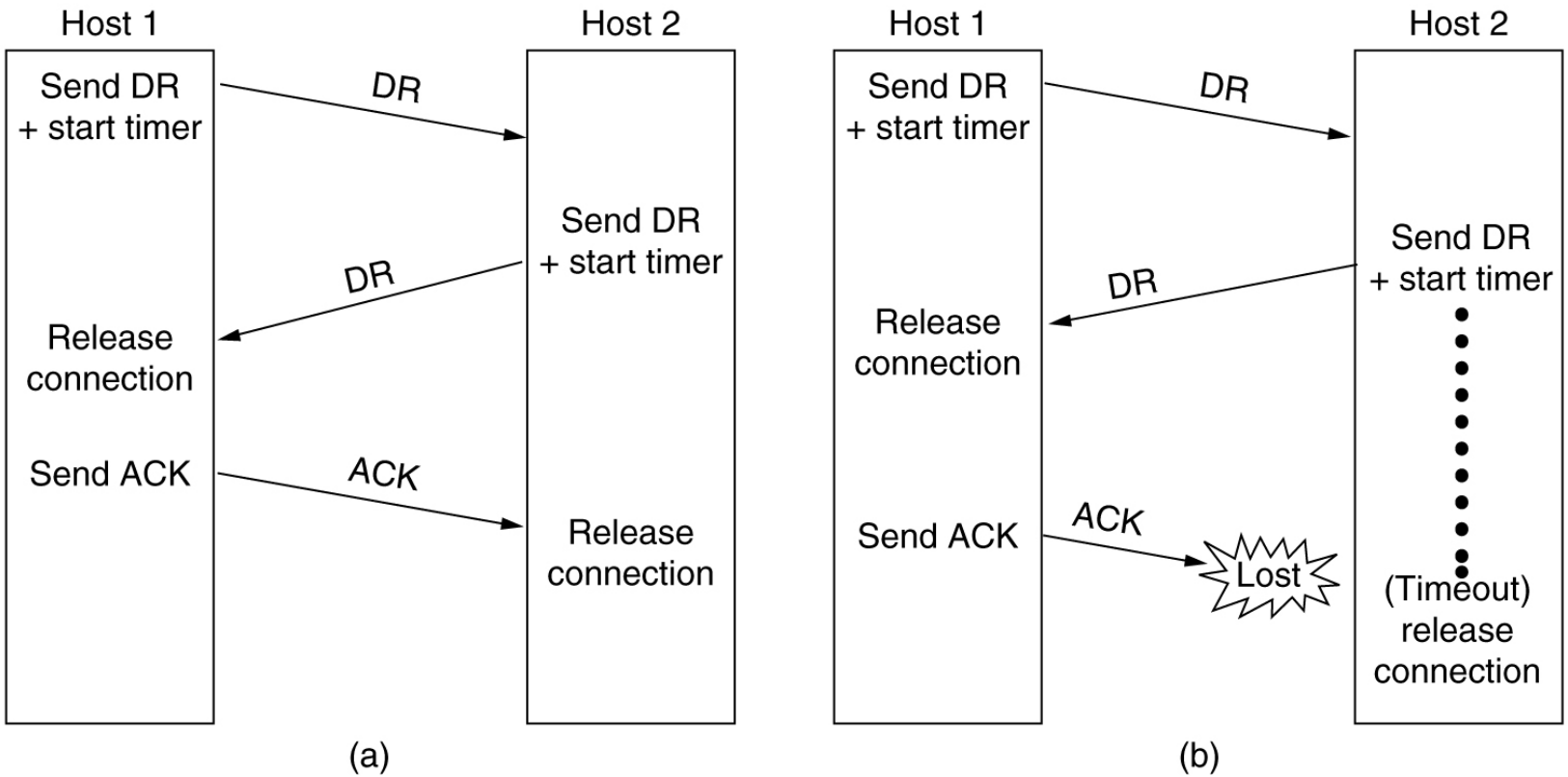
Abrupt disconnection with loss of data.

Connection Release (2)



The two-army problem.

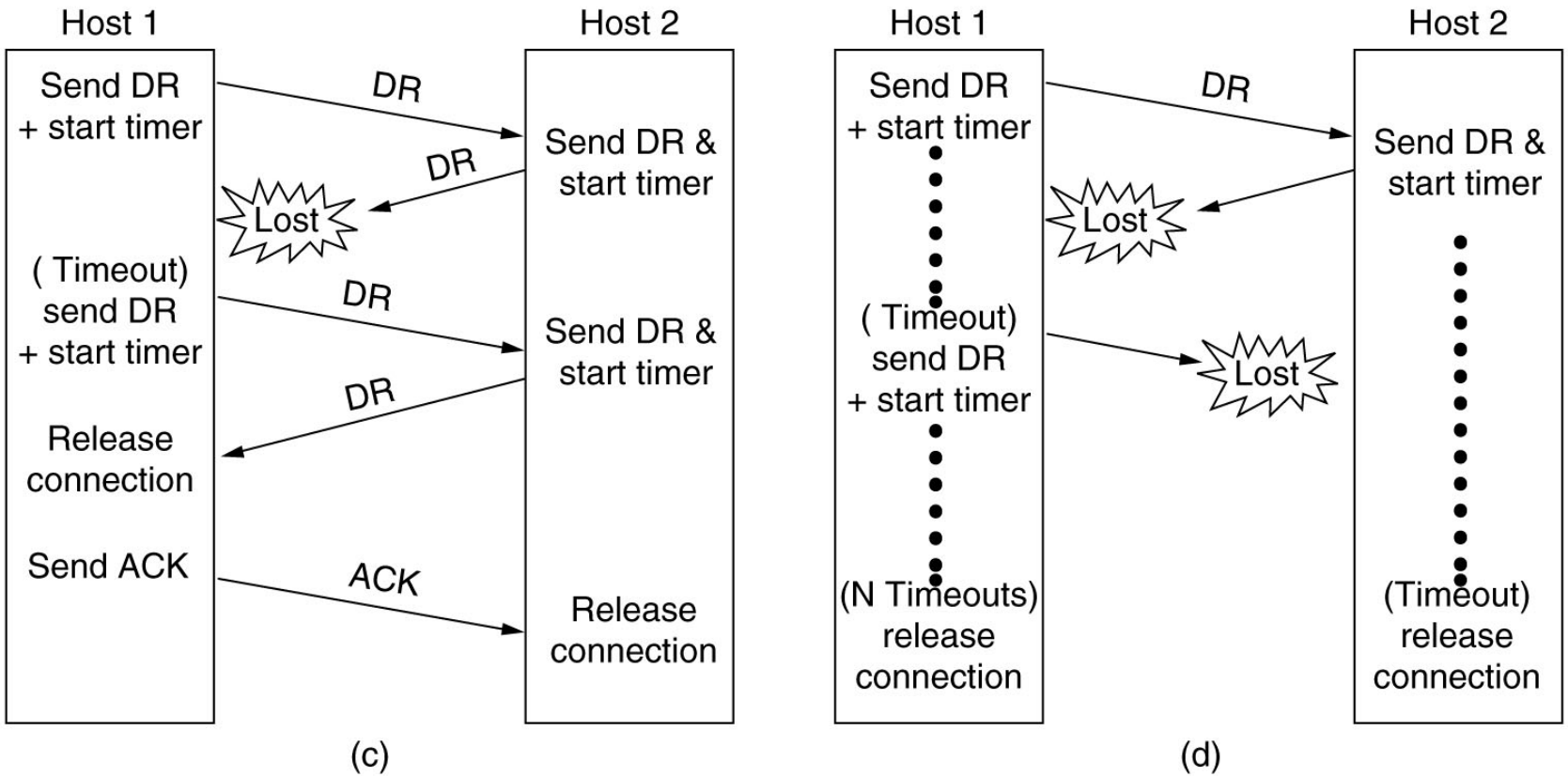
Connection Release (3)



Four protocol scenarios for releasing a connection. (a) Normal case of a three-way handshake. (b) final ACK lost.

Spring 2003

Connection Release (4)

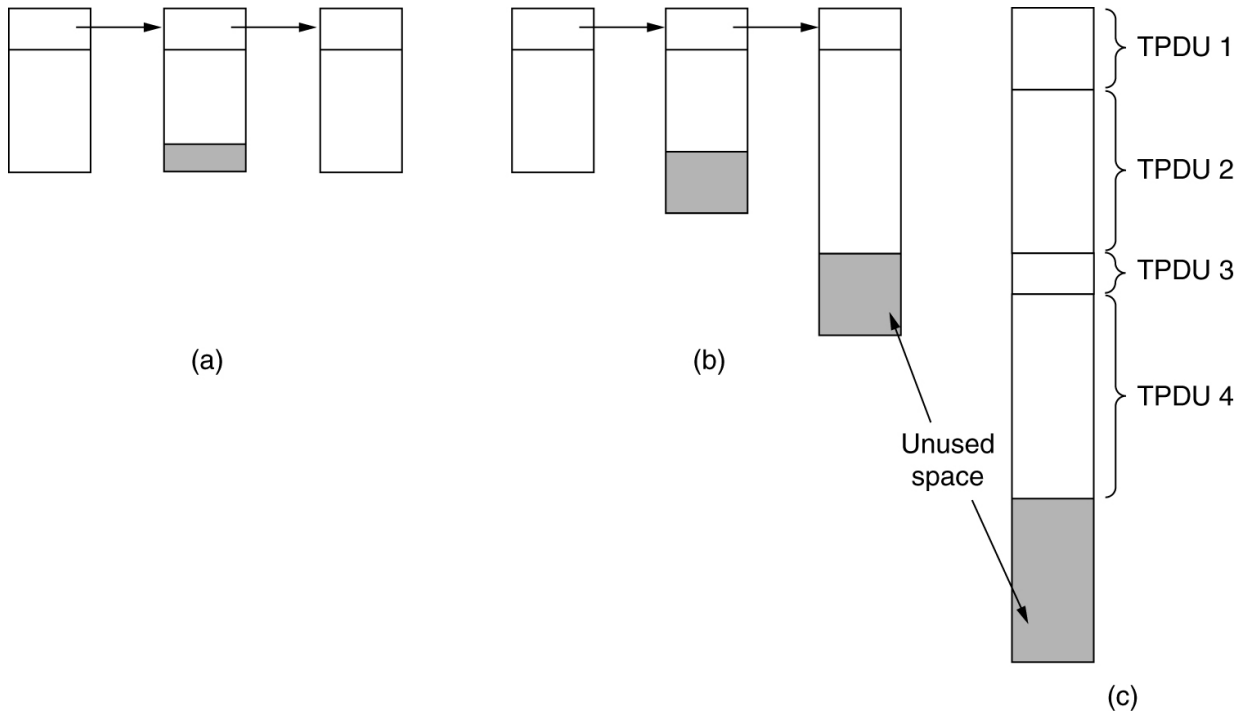


(c) Response lost. (d) Response lost and subsequent DRs lost.

Lingering Half-Duplex Connections

- If a party (or a link) dies
  - can be left with dead connections
- Solution: use keep-alive packets
  - every n seconds, send a packet
  - if no packet is received after n * m seconds, cleanup

Flow Control and Buffering



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

Buffer Management

- Unreliable Network
  - sender must buffer all un-acked packets
  - receiver can buffer if space is available
    - if not, drop packet and wait to re-transmission
- Buffer Size
  - does one size fit all?
    - are TPDUs of uniform size?
  - might use a fixed size buffer smaller than max TPDU
    - requires support for multiple buffers per TPDU
- Possible to decouple buffer allocation from window
  - ACKs contain both buffer credits and ACKSs
- Buffer Copies
  - possible for each layer to copy the buffer, but this is slow
  - handoff pointers to data, but requires coordination between layers

Flow Control and Buffering (2)

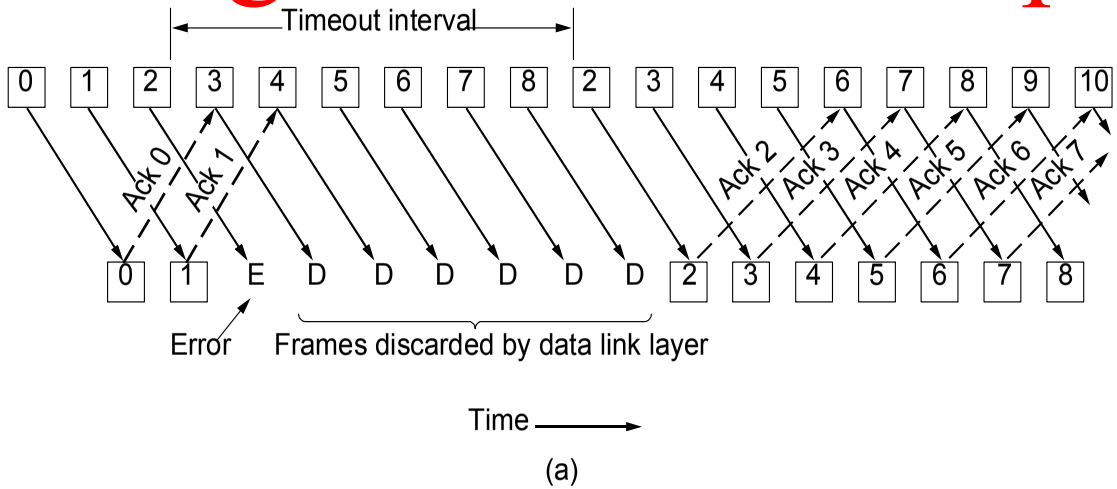
	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	• • •	Message lost but A thinks it has 1 left
6	←	<ack = 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

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

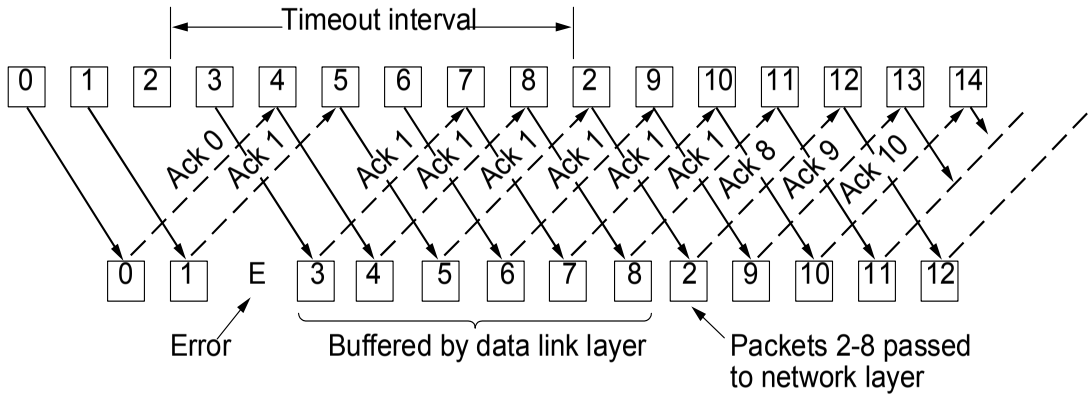
Sliding Window Protocol

- Need to
  - have multiple outstanding packets
  - limit total number of outstanding packets
  - permit re-transmissions to occur
- Sliding Window
  - permit at most N outstanding packets
  - when packet is ACK'd advance window to first non-ACK'd pkt
- Retransmission
  - Go-back N
    - when a packet is lost, restart from that packet
    - provides in-order delivery, but wastes bandwidth
  - Selective Retransmission
    - use timeout to re-sent lost packet
    - use NACK as a hint that something was lost

Sliding Window Example



(a)



(b)

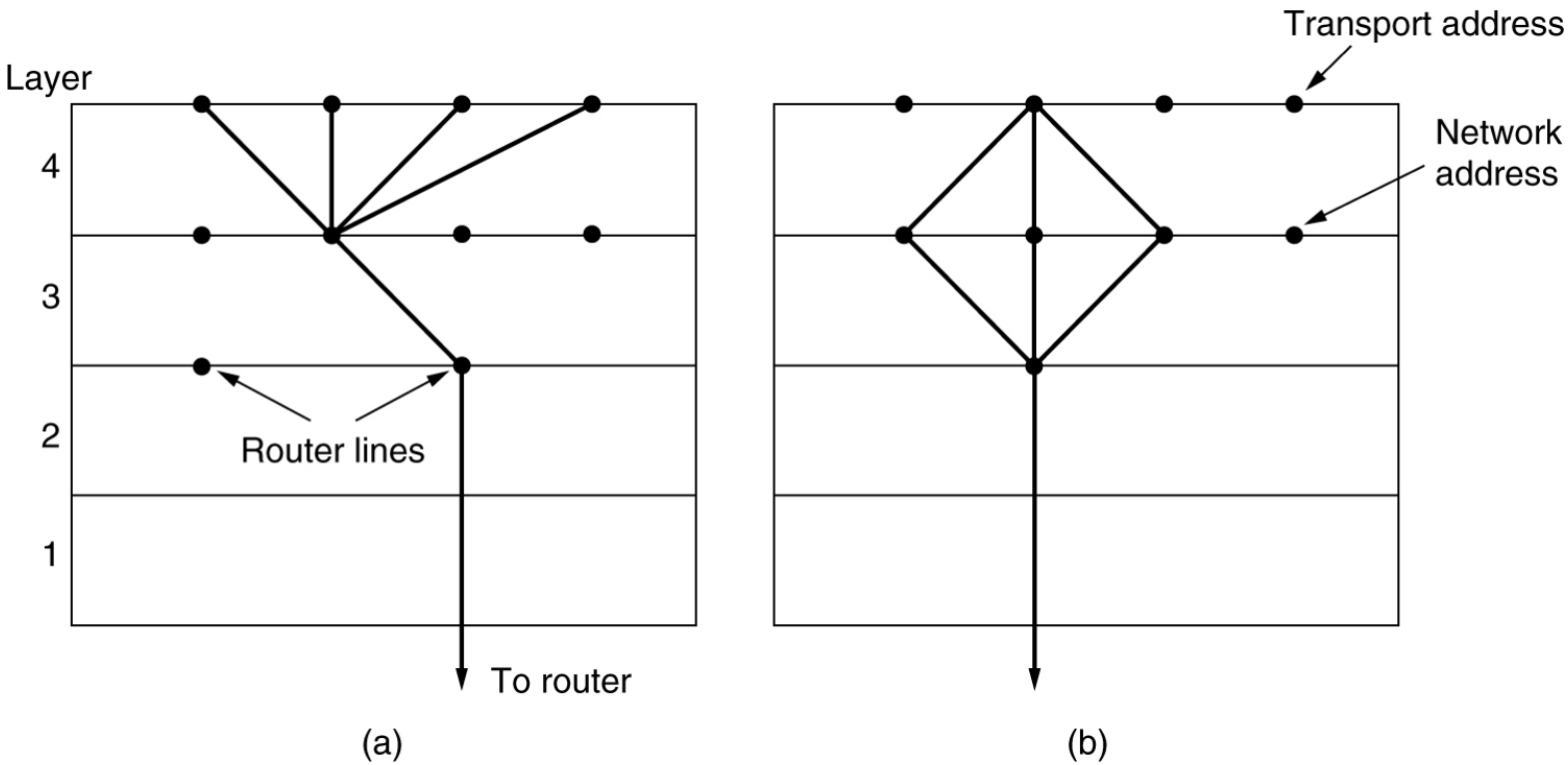
From: *Computer Networks*, 3rd Ed. by Andrew S. Tanenbaum, (c)1996 Prentice Hall.

Spring 2003

Multiplexing in the Transport Layer

- Upward multiplexing
  - putting multiple transport connections onto one network connection
  - used to accommodate pricing strategies that charge for connections
- Downward multiplexing
  - using several network connections per transport connection
  - permits use of multiple copies of network resources
    - if the network layer uses sliding windows
      - a high latency network may under utilize the link
      - multiple connections each get a window
    - per connection buffer allocation
      - get more buffers
    - round-robin scheduling
      - get a larger share of link bandwidth

Multiplexing



(a) Upward multiplexing. (b) Downward multiplexing.

Crash Recovery

- Router or Link Crashes
  - Data in transit can be lost.
  - End nodes have sufficient state to recover lost data.
  - Transport protocol can hide network failures from the application.
- Host Crashes
  - Transport level state will be lost at one end.
  - Does the transport layer have sufficient info to recover?, No!.
    - Information must flow down to network and up to transport user
      - ACKs go down, and data goes up.
      - It is not possible to make these two operations atomic.
    - lack of stable storage causes this problem
  - Result, higher up layer must deal with host crashes

Crash Recovery

Strategy used by sending host	Strategy used by receiving host: First ACK, then write			Strategy used by receiving host: 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

Different combinations of client and server strategy.

A Simple Transport Protocol

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

The Example Transport Entity

Network packet	Meaning
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 network layer packets used in our example.

The Example Transport Entity (2)

- Each connection is in one of seven states:
- Idle – Connection not established yet.
- Waiting – CONNECT has been executed, CALL REQUEST sent.
- Queued – A CALL REQUEST has arrived; no LISTEN yet.
- Established – The connection has been established.
- Sending – The user is waiting for permission to send a packet.
- Receiving – A RECEIVE has been done.
- DISCONNECTING – a DISCONNECT has been done locally.

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 (9)

```
switch (ptype) {
  case CALL_REQ:                          /* remote user wants to establish connection */
    cptr->local_address = data[0];  cptr->remote_address = data[1];
    if (cptr->local_address == listen_address) {
          listen_conn = cid;  cptr->state = ESTABLISHED;  wakeup();
    } else {
          cptr->state = QUEUED;  cptr->timer = TIMEOUT;
    }
    cptr->clr_req_received = 0;   cptr->credits = 0;
    break;
  case CALL_ACC:                          /* remote user has accepted our CALL_REQ */
    cptr->state = ESTABLISHED;
    wakeup();
    break;
  case CLEAR_REQ:                         /* remote user wants to disconnect or reject call */
    cptr->clr_req_received = 1;
    if (cptr->state == DISCONN) cptr->state = IDLE; /* clear collision */
    if (cptr->state ==  WAITING || cptr->state == RECEIVING || cptr->state == SENDING) wakeup();
    break;
  case CLEAR_CONF:                        /* remote user agrees to disconnect */
    cptr->state = IDLE;
    break;
  case CREDIT:                            /* remote user is waiting for data */
    cptr->credits += data[1];
    if (cptr->state == SENDING) wakeup();
    break;
  case DATA_PKT:                          /* remote user has sent data */
    for (i = 0; i < count; i++) cptr->user_buf_addr[cptr->byte_count + i] = data[i];
    cptr->byte_count += count;
    if (m == 0 ) wakeup();
}
}
```

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 $\overline{P2}$: A1/Estab $\overline{P2}$: A2/Idle		~/Estab				
	CONNECT	$\overline{P1}$: ~/Idle P1: A3/Wait						
	DISCONNECT				P4: A5/Idle $\overline{P4}$: A6/Disc			
	SEND				$\overline{P5}$: A7/Estab P5: A8/Send			
	RECEIVE				A9/Receiving			
Incoming packets	Call_req	P3: A1/Estab $\overline{P3}$: A4/Queu'd						
	Call_acc		~/Estab					
	Clear_req		~/Idle		A10/Estab	A10/Estab	A10/Estab	~/Idle
	Clear_conf							~/Idle
	DataPkt						A12/Estab	
	Credit				A11/Estab	A7/Estab		
Clock	Timeout			~/Idle				

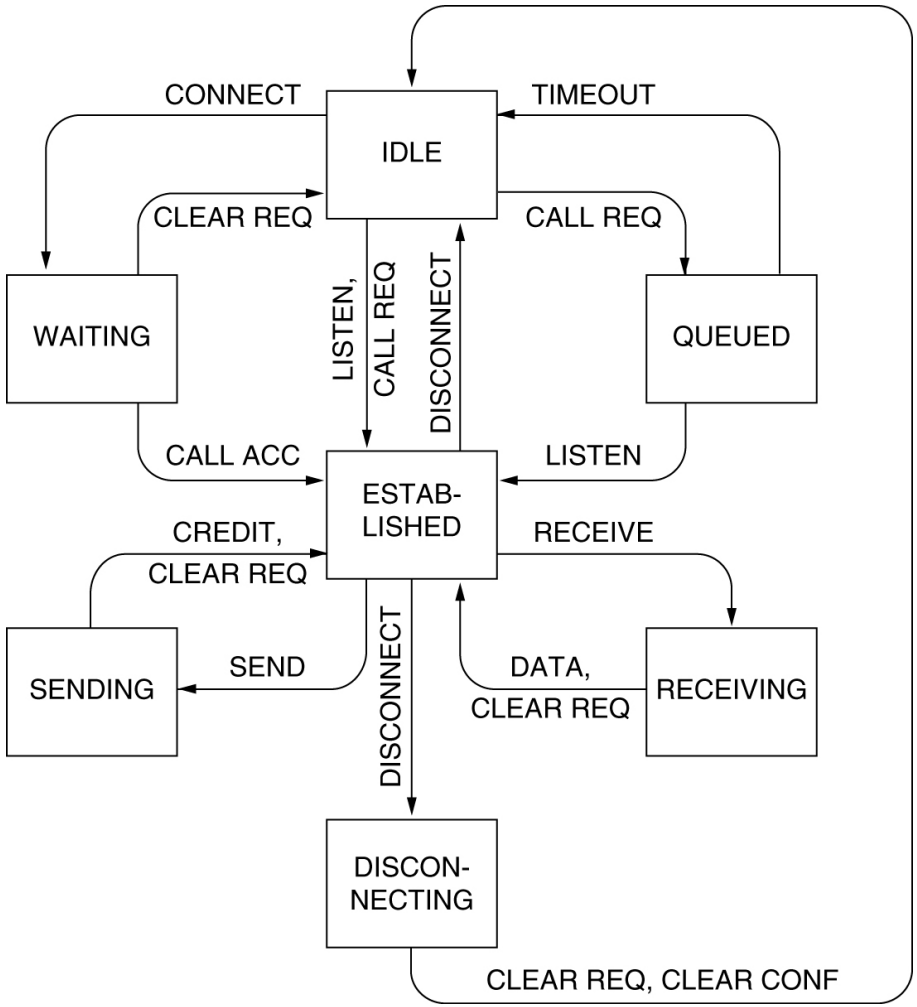
Predicates

P1: Connection table full
P2: Call_req pending
P3: LISTEN pending
P4: Clear_req pending
P5: Credit available

Actions

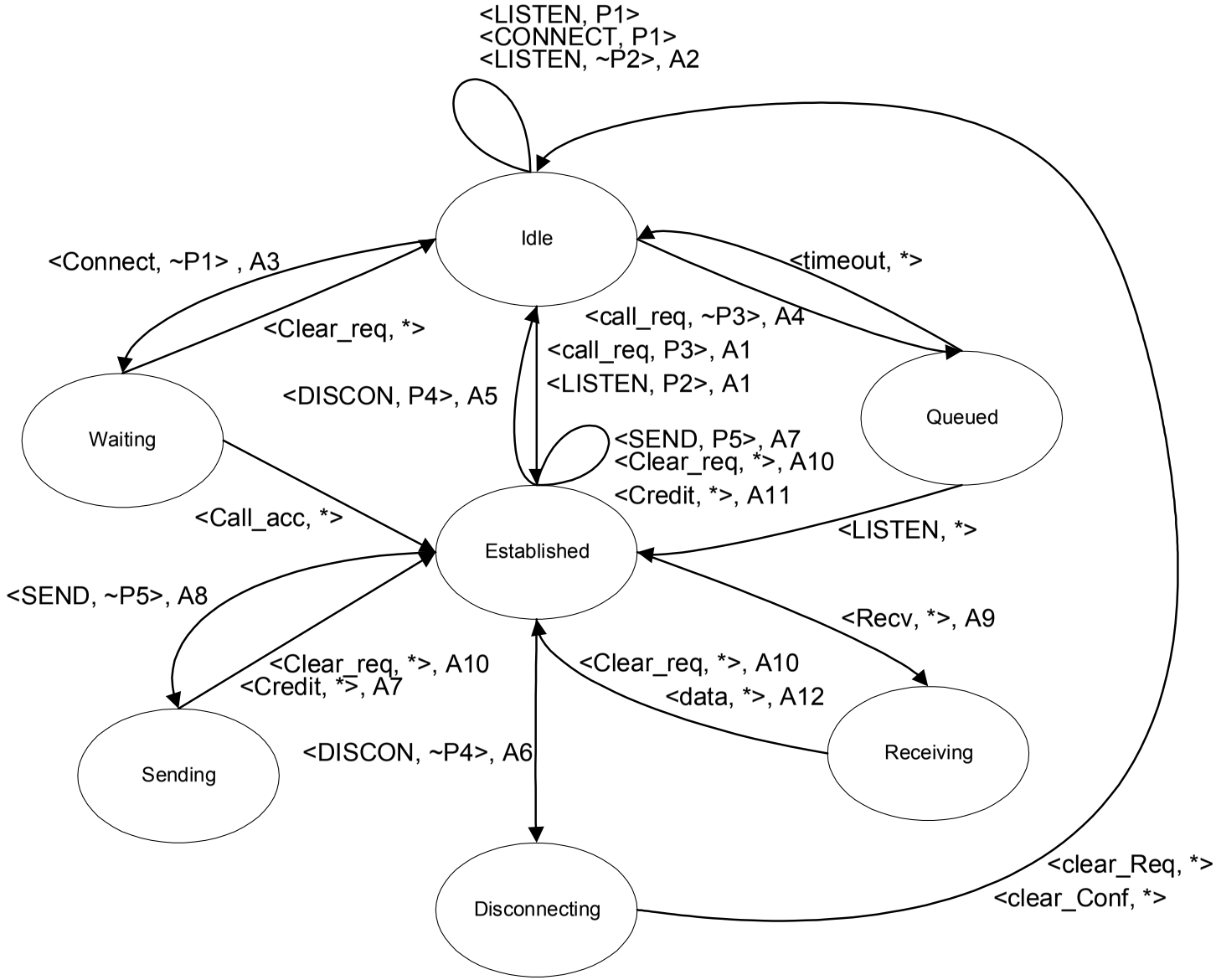
A1: Send Call_acc
A2: Wait for Call_req
A3: Send Call_req
A4: Start timer
A5: Send Clear_conf
A6: Send Clear_req
A7: Send message
A8: Wait for credit
A9: Send credit
A10: Set Clr_req_received flag
A11: Record credit
A12: Accept message

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.

Protocol State Machines



Spring 2003

Predicates And State Transitions

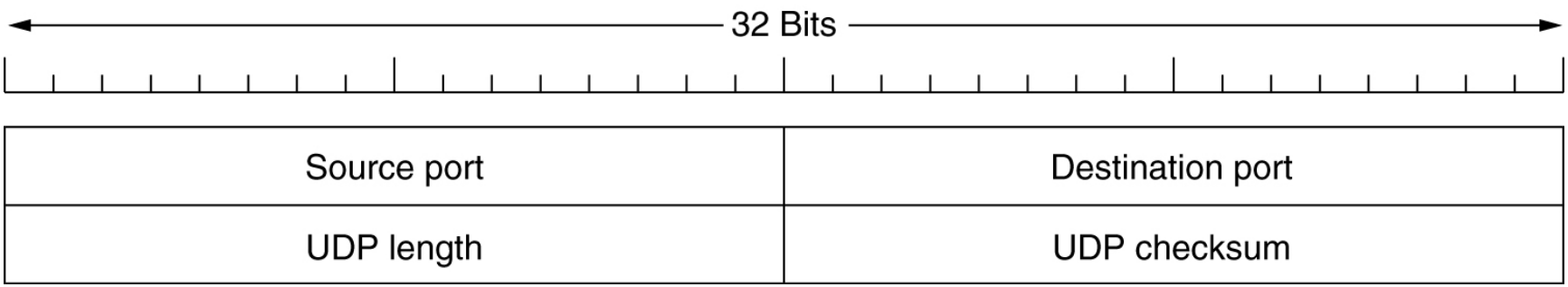
Pred	Meaning
P1	Connection table full
P2	Call_req pending
P3	LISTEN Pending
P4	Clear_req Pending
P5	Credit Available

Act	Meaning
A1	Send Call_acc
A2	Wait for Call_req
A3	Send Call_req
A4	Start Timer
A5	Send Clear_conf
A6	Send Clear_req
A7	Send message
A8	Wait for credit
A9	Send Credit
A10	Set Clr_req_recv flag
A11	Record credit
A12	Accept message

The Internet Transport Protocols: UDP

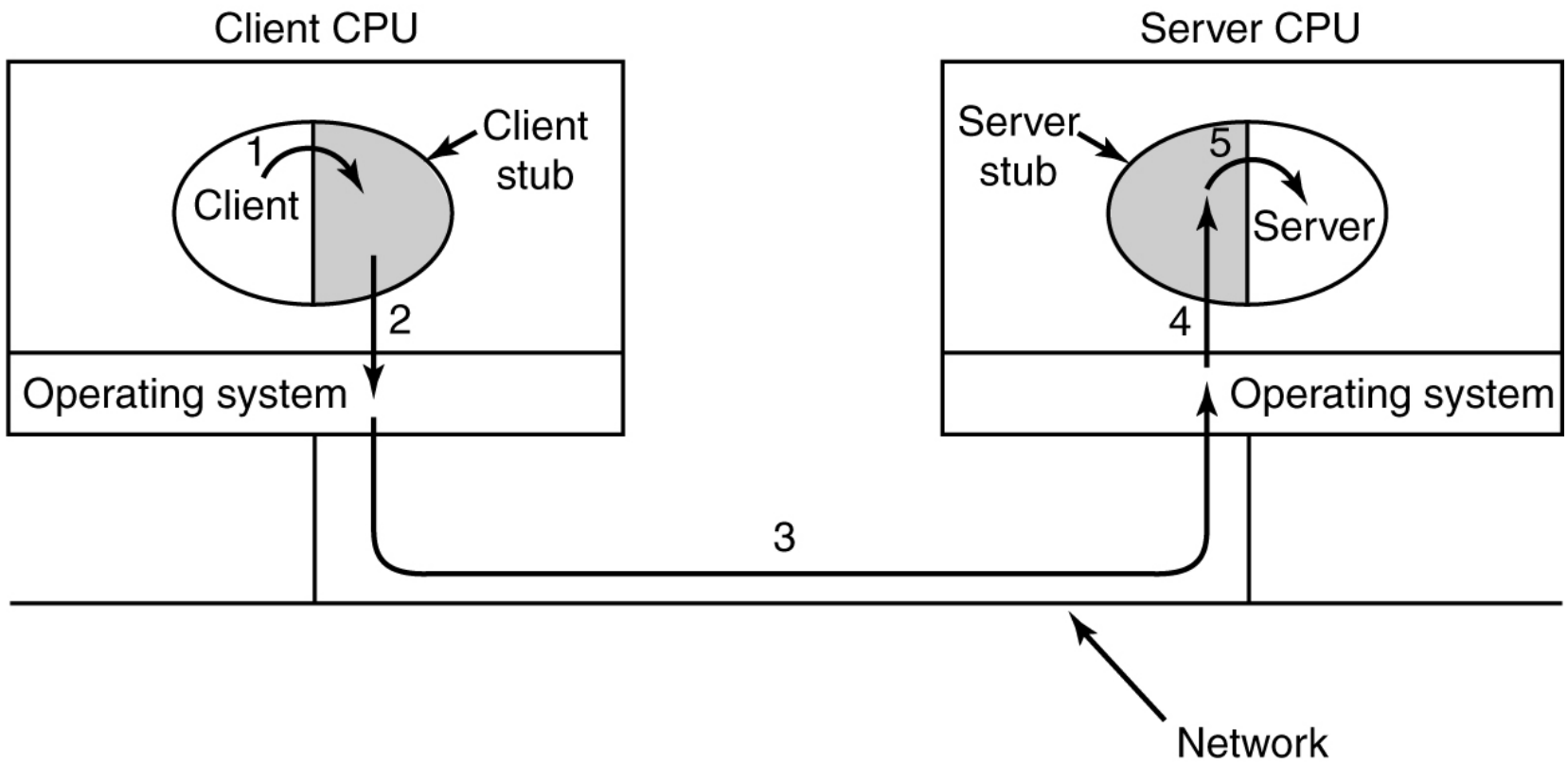
- Introduction to UDP
- Remote Procedure Call
- The Real-Time Transport Protocol

Introduction to UDP



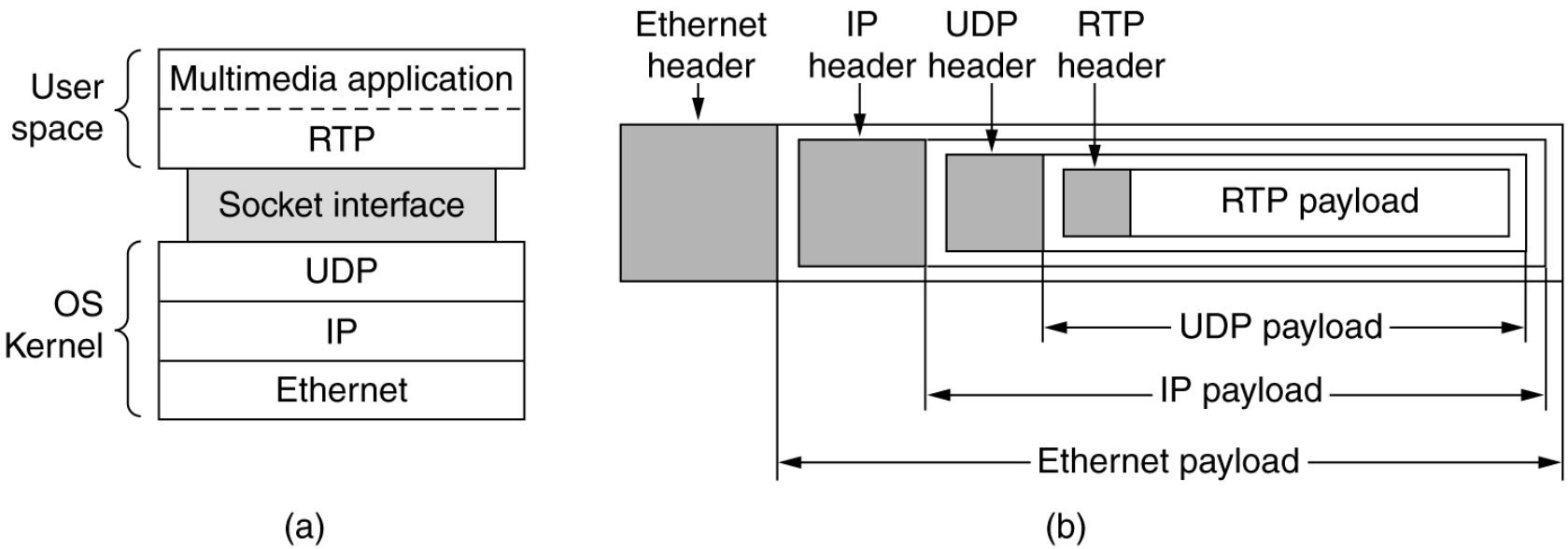
The UDP header.

Remote Procedure Call



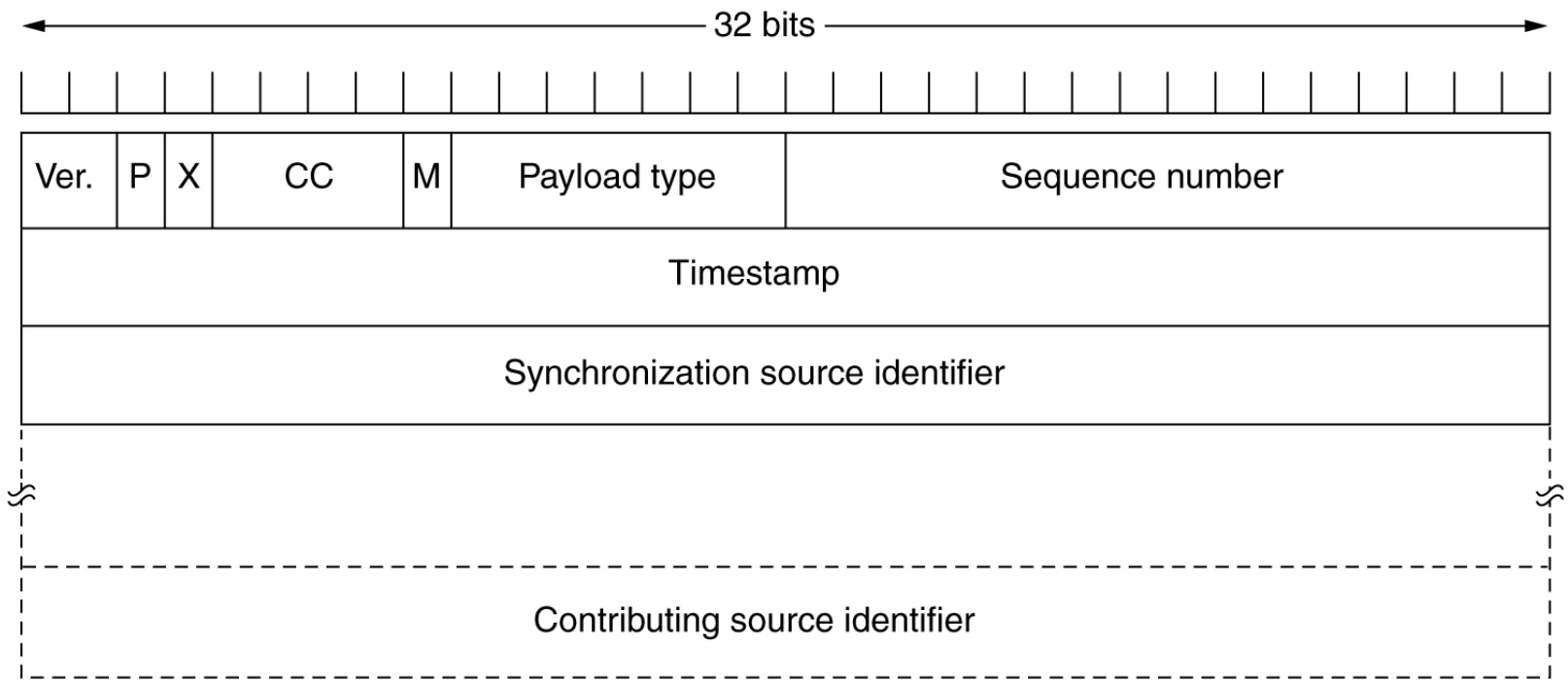
Steps in making a remote procedure call. The stubs are shaded.

The Real-Time Transport Protocol



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

The Real-Time Transport Protocol (2)



The RTP header.

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

Basic Concepts

- Treat data as a byte stream
  - Every byte has a sequence number (32-bit)
- Segment
    - Fixed 20 Byte header
    - Optional part
    - Data
  - Segment Size
    - Smaller of 65,535, and
    - Maximum Transfer Unit (MTU)
      - Network wide fixed
      - Generally a few thousand bytes
  - Boundary Routers may fragment a segment
    - Each new segment gets its own IP header

TCP Basic Concepts

- Sliding Window Protocol
- Cumulative Acknowledgement
- Timers for timeout

- Sender may retransmit
  - Bits and pieces may arrive at the receiver

TCP Protocol

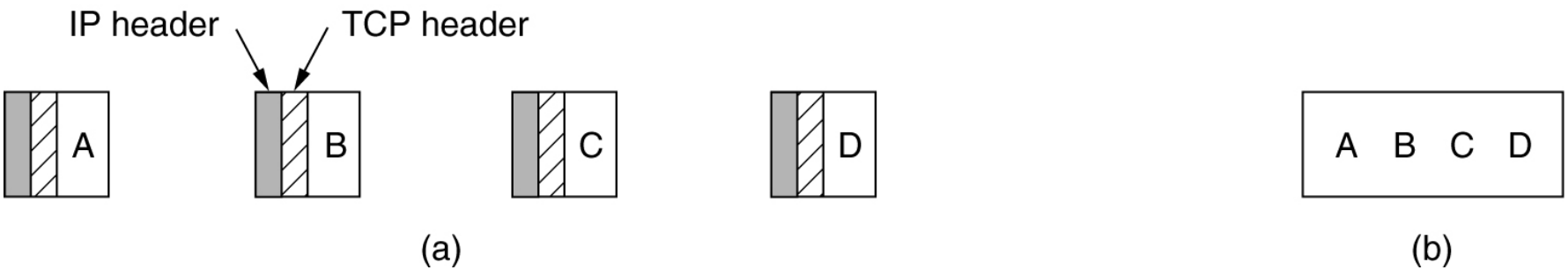
- TSAPs
  - Use <host, port> combination
    - 32 + 16 = 48 bits
  - Well known ports provide services
    - first 256 ports
    - SMTP 25, Telnet 23, Ftp 21, HTTP 80
- Provides a byte stream
  - this is not a message stream
  - a message (single call to send) may be split, merged, etc.
- Urgent Data field
  - provides cut through delivery within a transport connection
  - used to send breaks or other high priority info

The TCP Service Model

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

Some assigned ports.

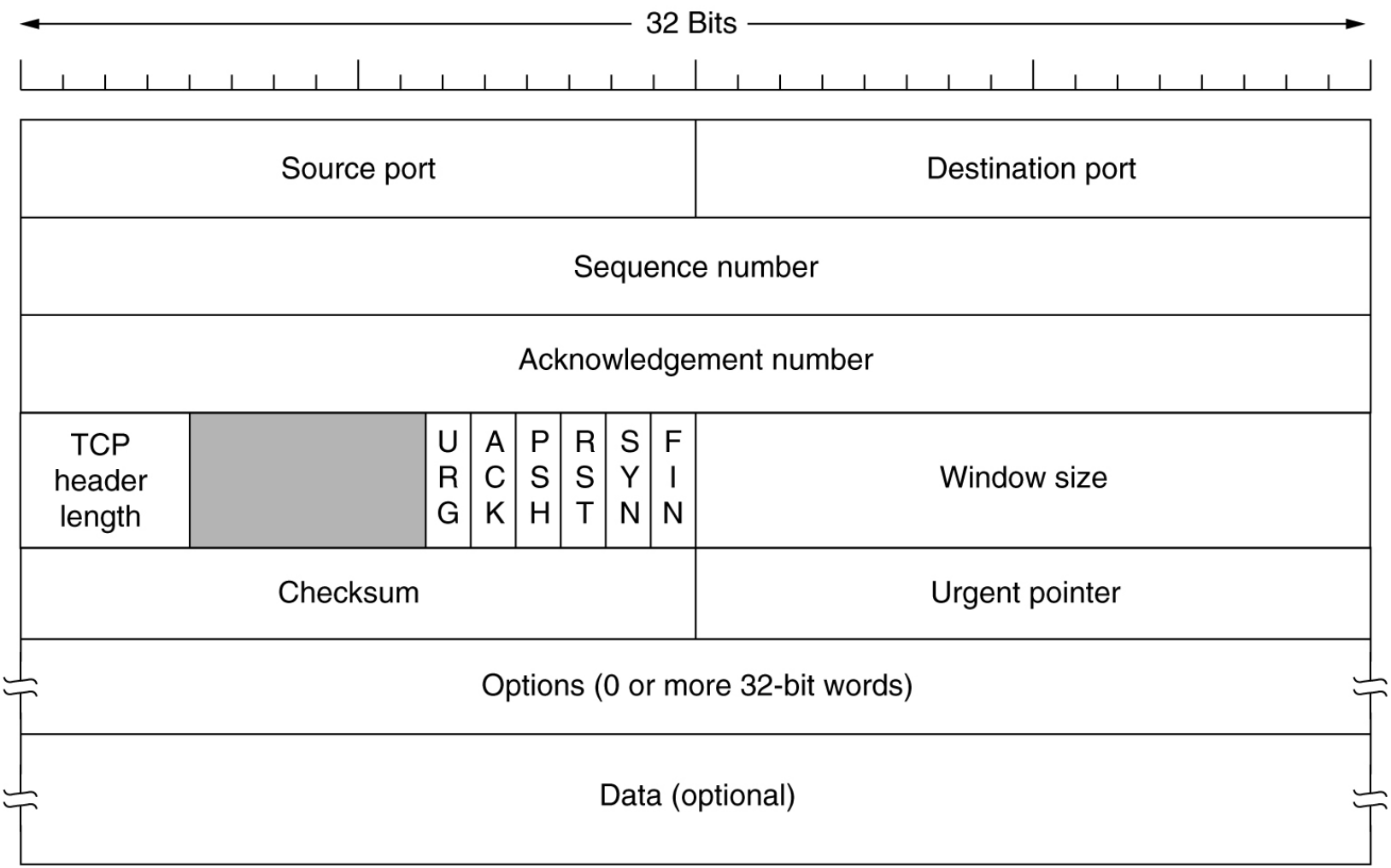
The TCP Service Model (2)



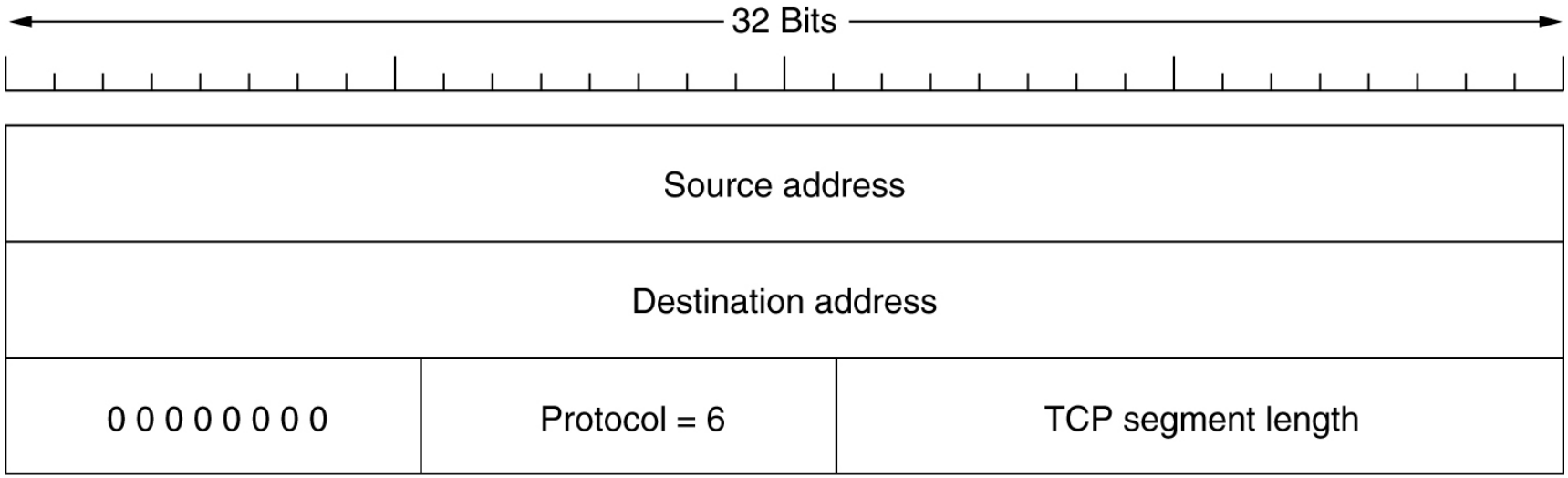
(a) Four 512-byte segments sent as separate IP datagrams.

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

The TCP Segment Header



The TCP Segment Header (2)



The pseudoheader included in the TCP checksum.

TCP Header

- Source Port and Destination Port
- Sequence Number
  - Sequence Number of the first byte of the segment
- Acknowledgement
  - Next sequence number expected
- TCP Header length
  - Number of 32 bit words contained in the header

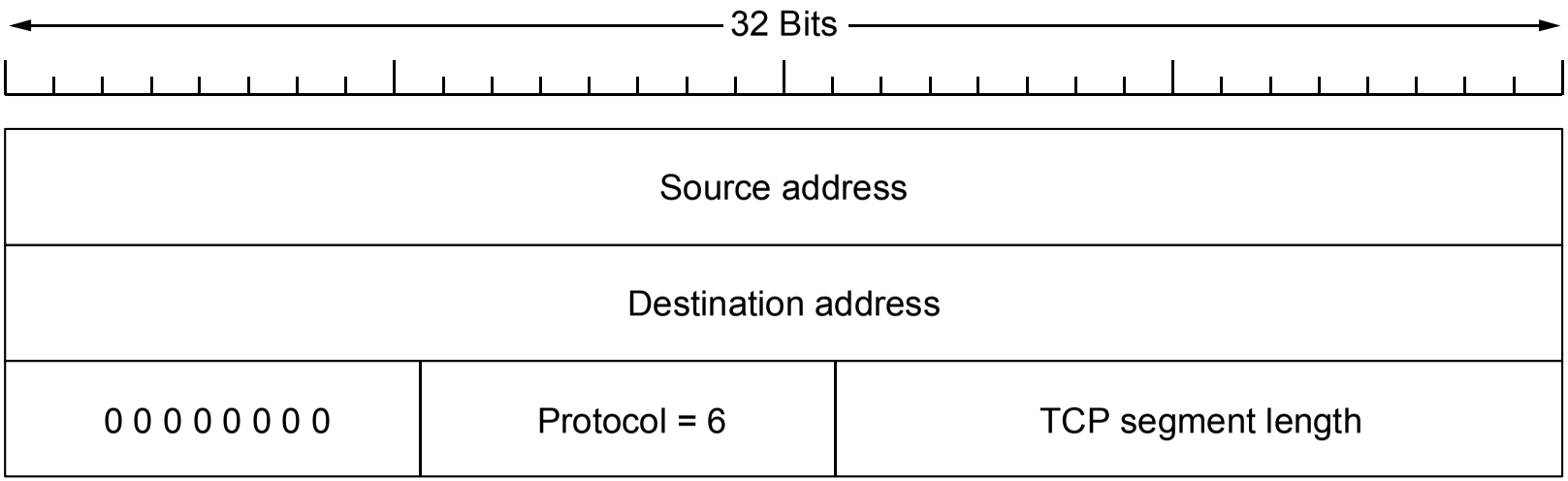
Header Flags

- URG - Urgent - 1 when urgent pointer is in use
- ACK - Acknowledgement - 1 when Ack No is valid
- PSH - Push - 1 -> send segment immediately
- RST - Reset - 1 to reset or refuse a connection
- SYN - Synch - used for connection establishment
- FIN - Finish - No more data - Connection Release

- Urgent Pointer - offset from the current sequence number where urgent data is

Checksum

- Add all the 16 bit words in 1's complement and take 1's complement of the sum
- Include Pseudo-header

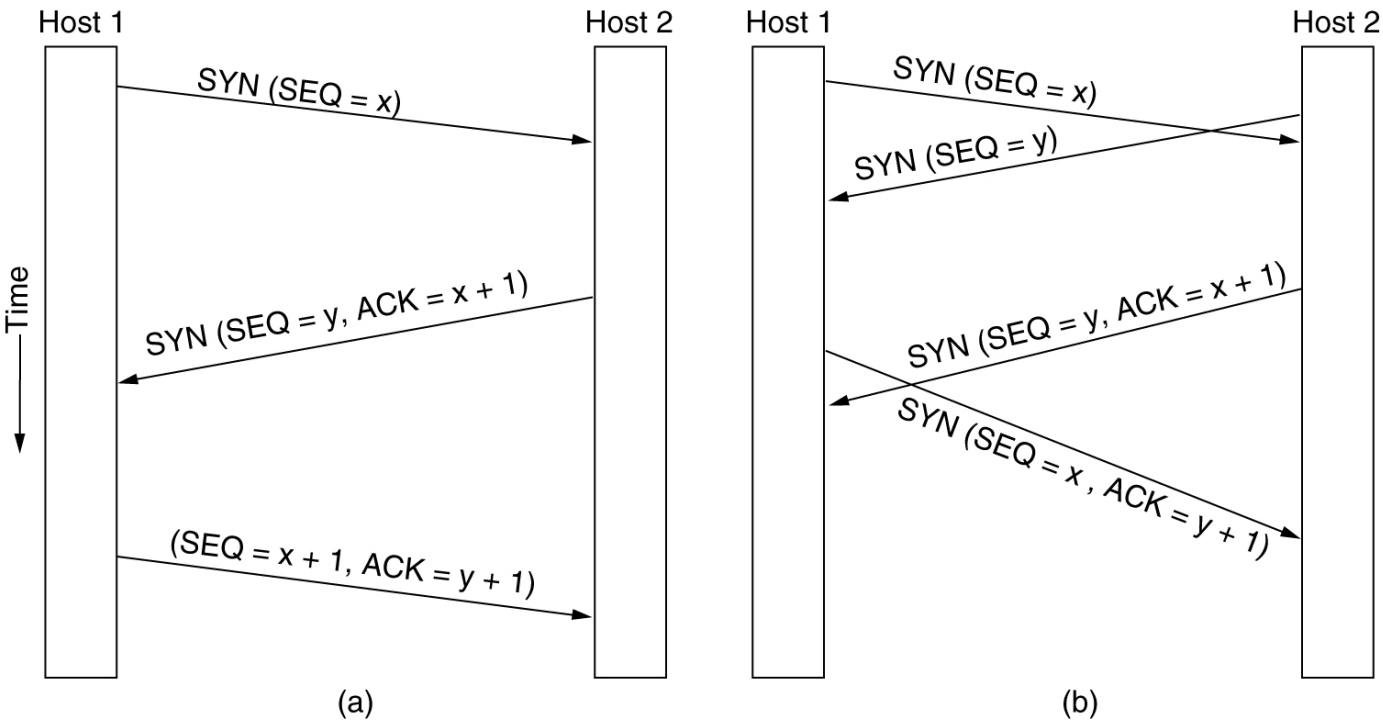


From: *Computer Networks*, 3rd Ed. by Andrew S. Tanenbaum, (c)1996 Prentice Hall.

TCP Connection Management

- Three-way Handshake
- Initial Sequence Numbers
  - Use a 4 micro-second clock
  - hosts must wait T (120 seconds) before a reboot
- Connection Closure
  - Each side uses a FIN and FIN_ACK message
  - A FIN times out after 2 T (240 seconds)
  - Keep alives used to timeout half dead connections

TCP Connection Establishment



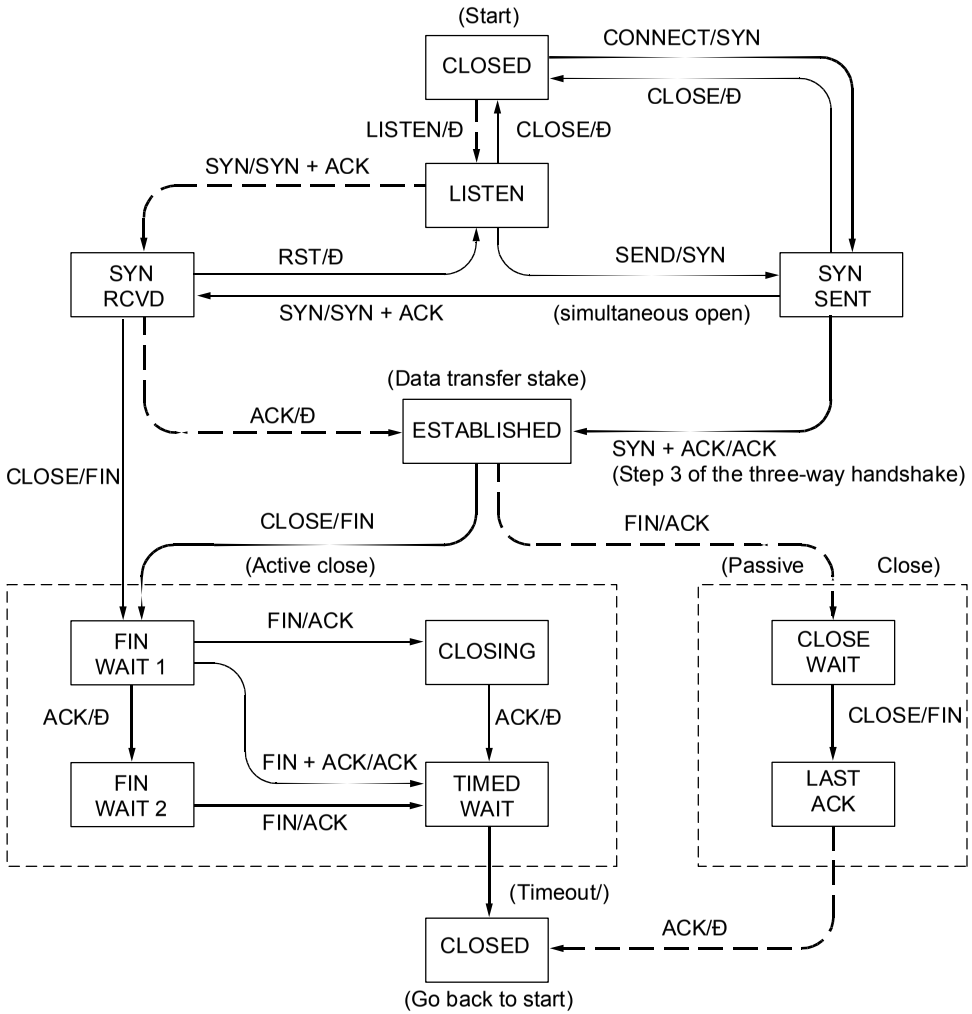
(a) TCP connection establishment in the normal case.

(b) Call collision.

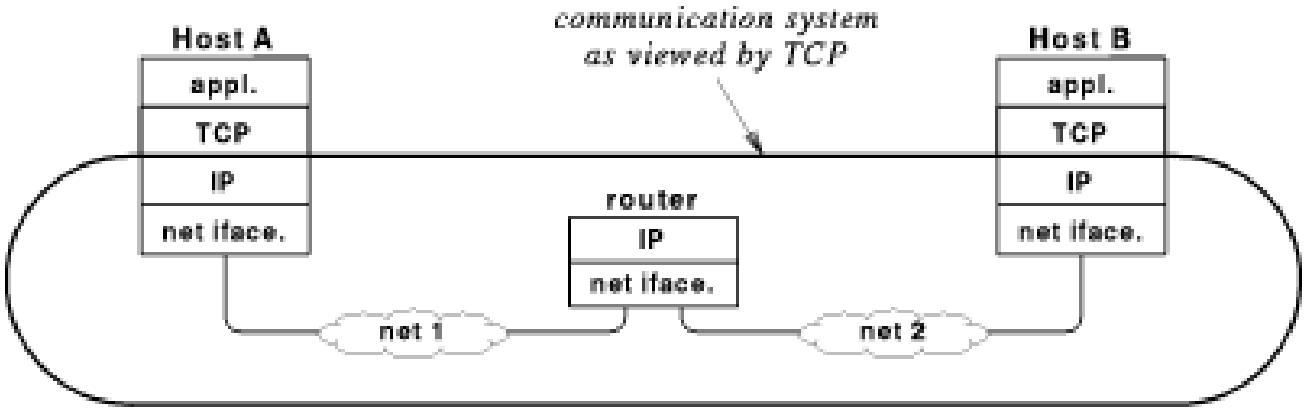
TCP States

State	Description
Closed	No Connection is active or pending
Listen	The server is waiting for an incoming call
Syn Rcvd	Connection request arrived; wait for ack
Syn Sent	The application has started to open a connection
Established	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

TCP States


(Start)
CLOSED
CONNECT/SYN
CLOSE/Ð
LISTEN/Ð
CLOSE/Ð
SYN/SYN + ACK
LISTEN
SYN
RCVD
RST/Ð
SEND/SYN
SYN
SENT
SYN/SYN + ACK
(simultaneous open)
(Data transfer stake)
ACK/Ð
ESTABLISHED
SYN + ACK/ACK
(Step 3 of the three-way handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
(Active close)
(Passive
Close)
FIN
WAIT 1
FIN/ACK
CLOSING
CLOSE
WAIT
ACK/Ð
ACK/Ð
CLOSE/FIN
FIN + ACK/ACK
FIN
WAIT 2
TIMED
WAIT
LAST
ACK
FIN/ACK
(Timeout/)
ACK/Ð
CLOSED
(Go back to start)

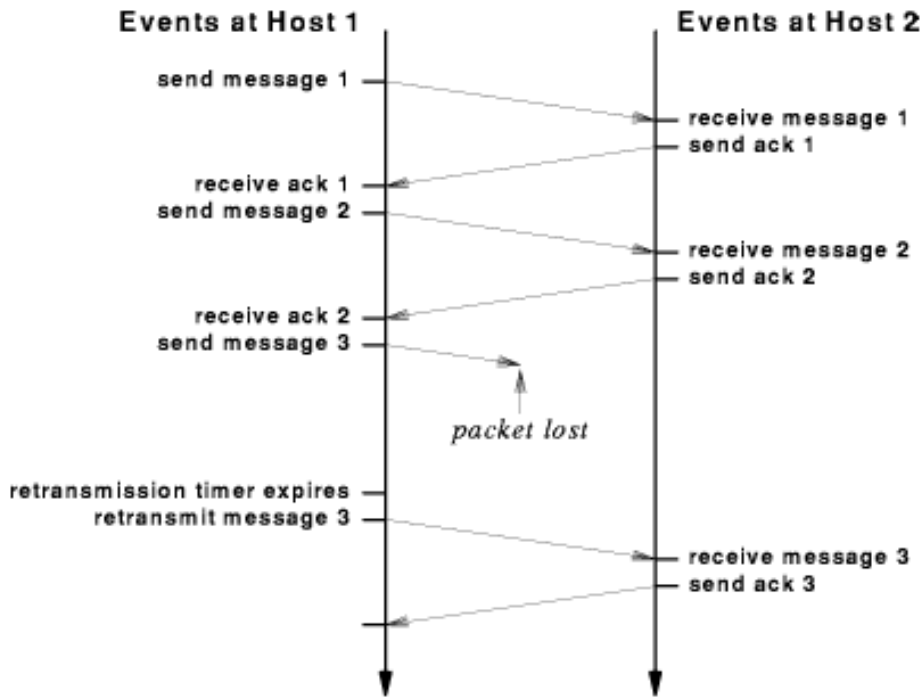
TCP As End-to-end Protocol



From "Computer Networks and Internets" by Comer

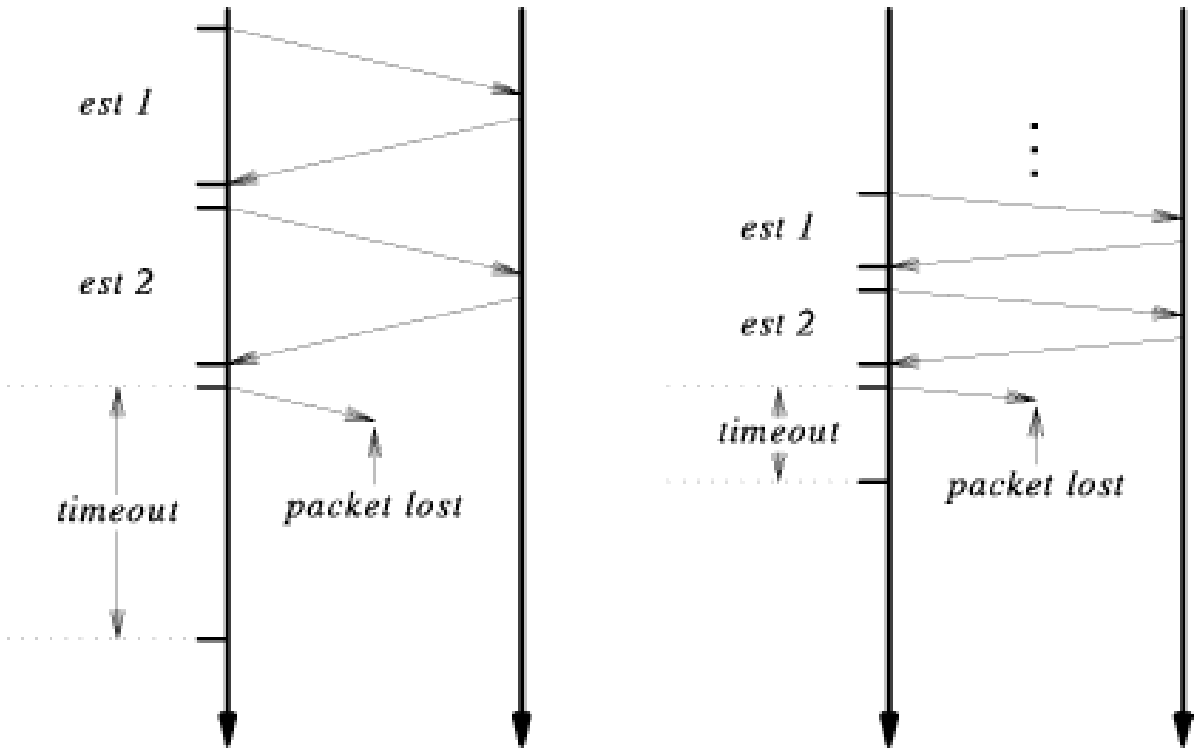
Achieving Reliability

Retransmission



From "Computer Networks and Internets" by Comer

Adaptive Retransmission

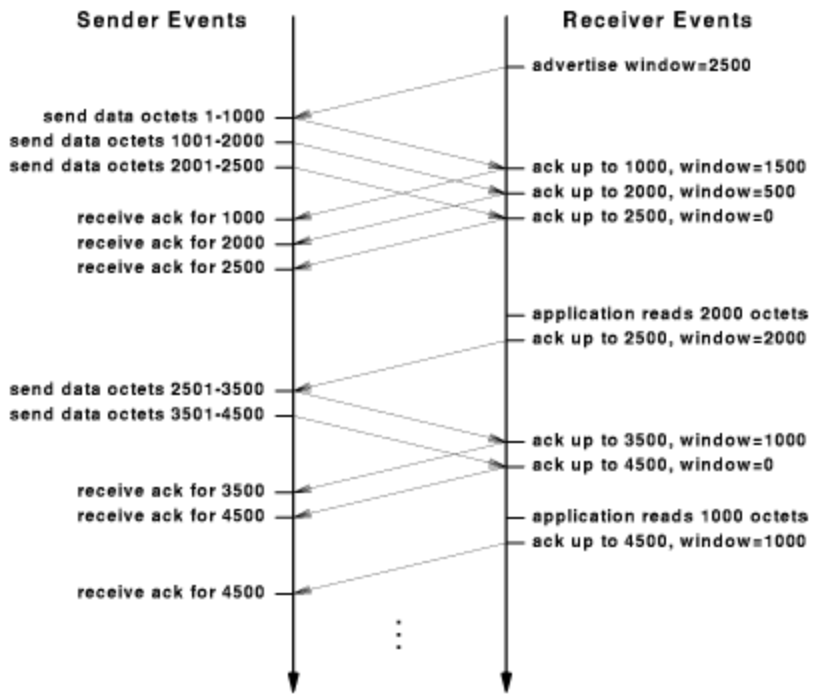


Adjust RTT and Timeout estimates

From "Computer Networks and Internets" by Comer

TCP Data Flow

Max Seg Size = 1000



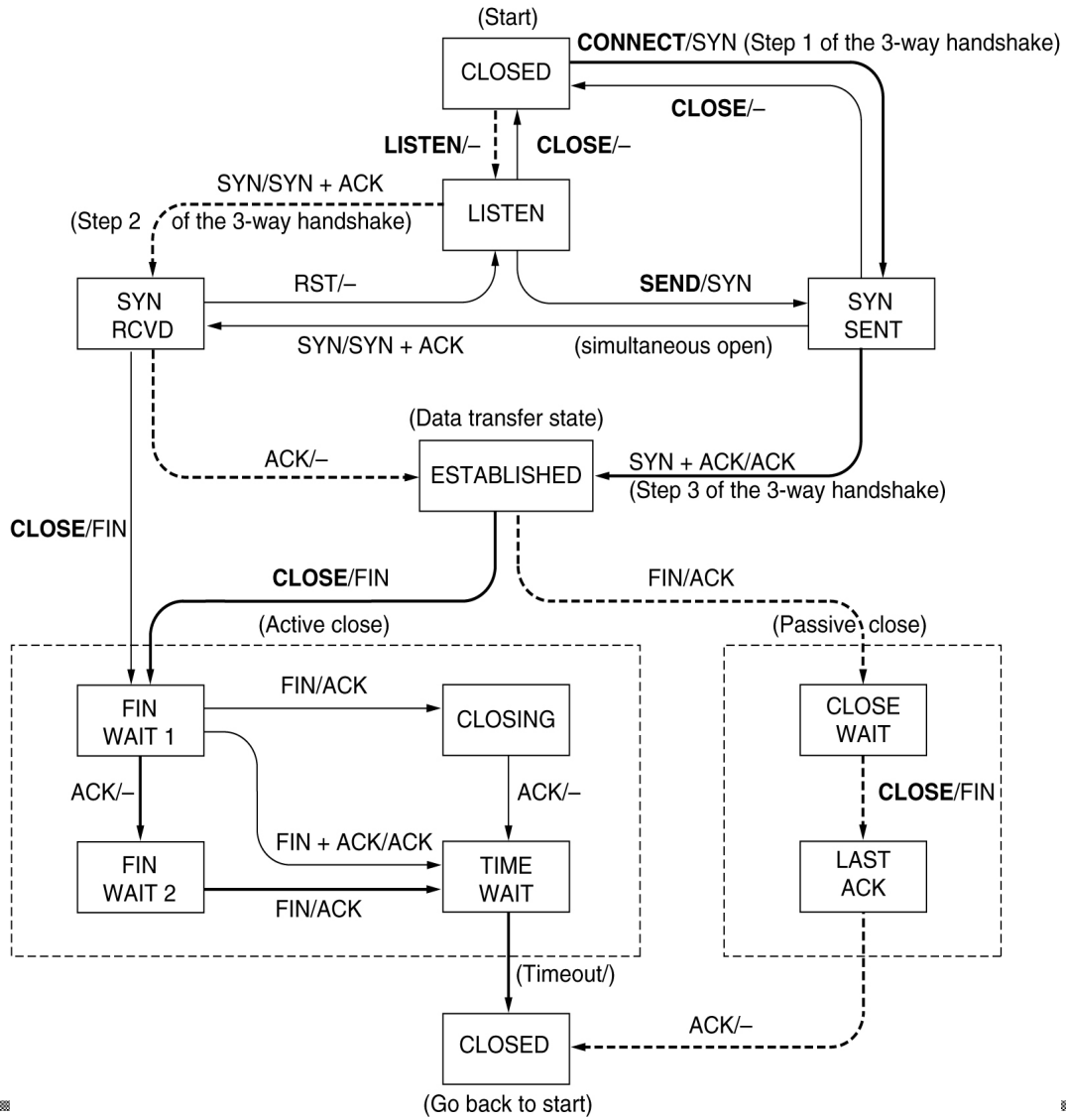
TCP Connection Management Modeling

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

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

TCP Connection Management Modeling (2)

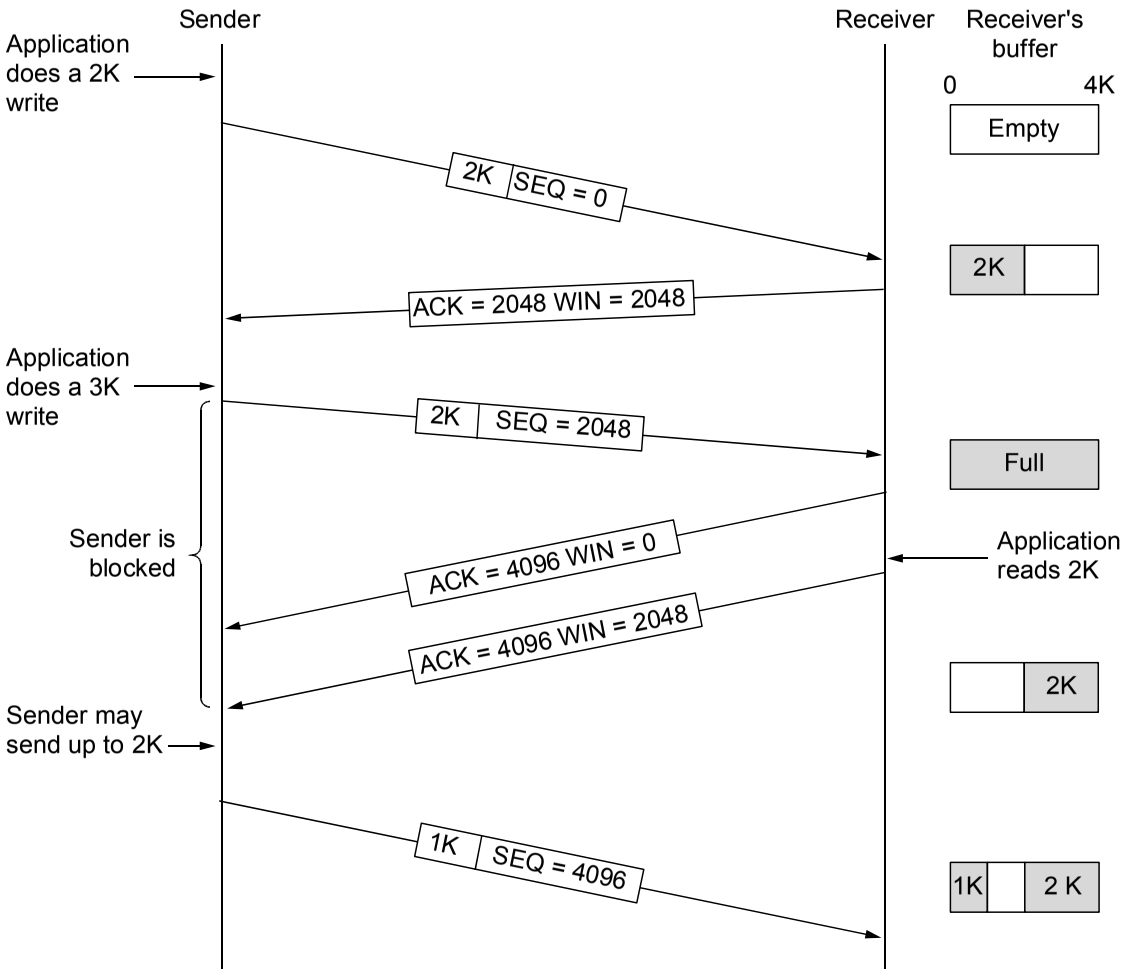
TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled by the event causing it and the action resulting from it, separated by a slash.



TCP Flow Control

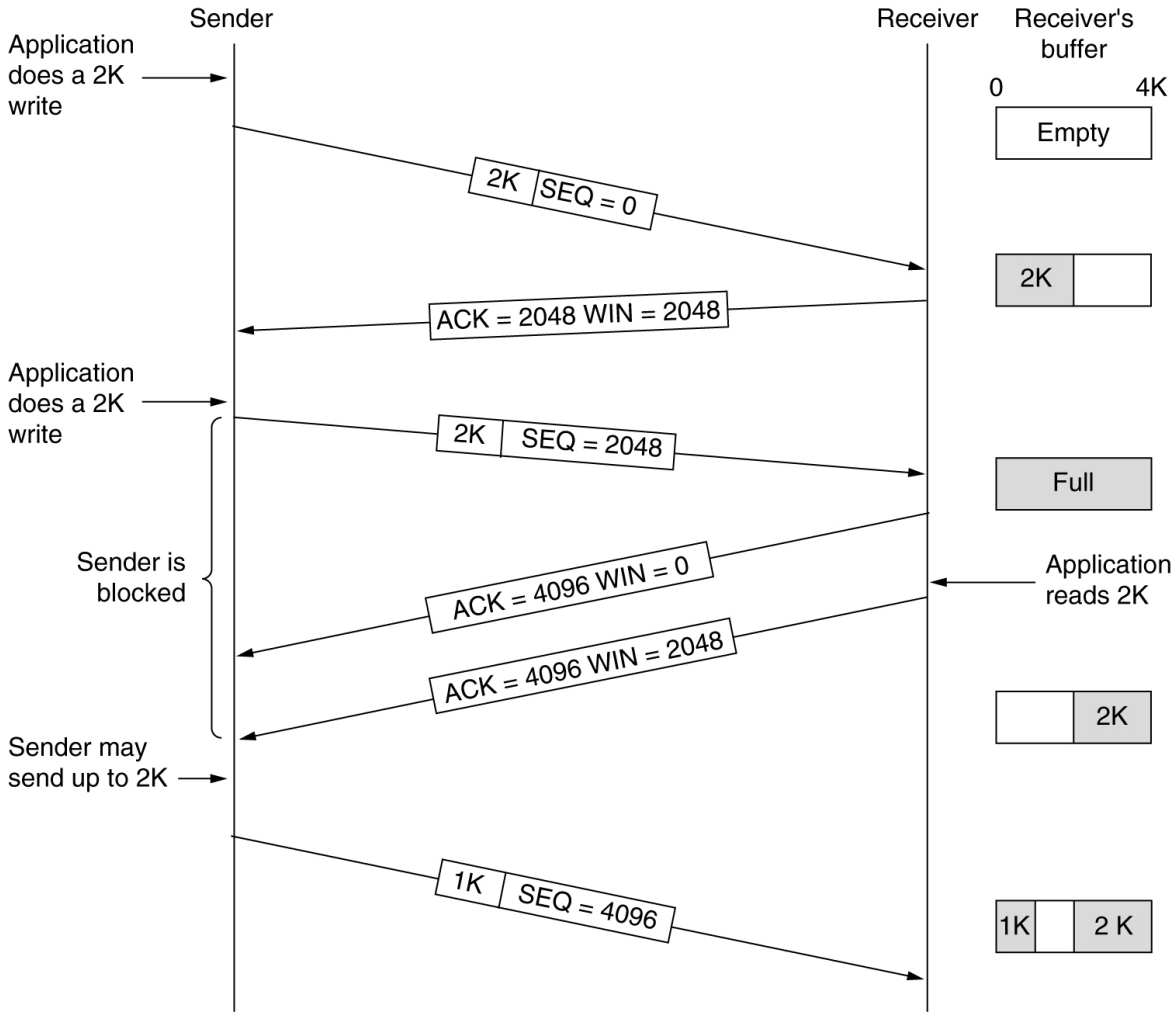
- Use Variable Sized Sliding Window
  - ACK indicates start of window
  - Window size indicates current size of window
- Receiver can send a window of 0
  - indicates that it want to pause connection
  - urgent data need not follow this request
- Window size of 16 bits is too small
  - 64K Bytes
  - only a small fraction of the in-flight bytes when
    - bandwidth is high
    - delay is high
  - solution: window shift option:
    - bit shift window up to 16 bits
    - permits up to 2^{32} byte windows
    - reduces window granularity

Window Management



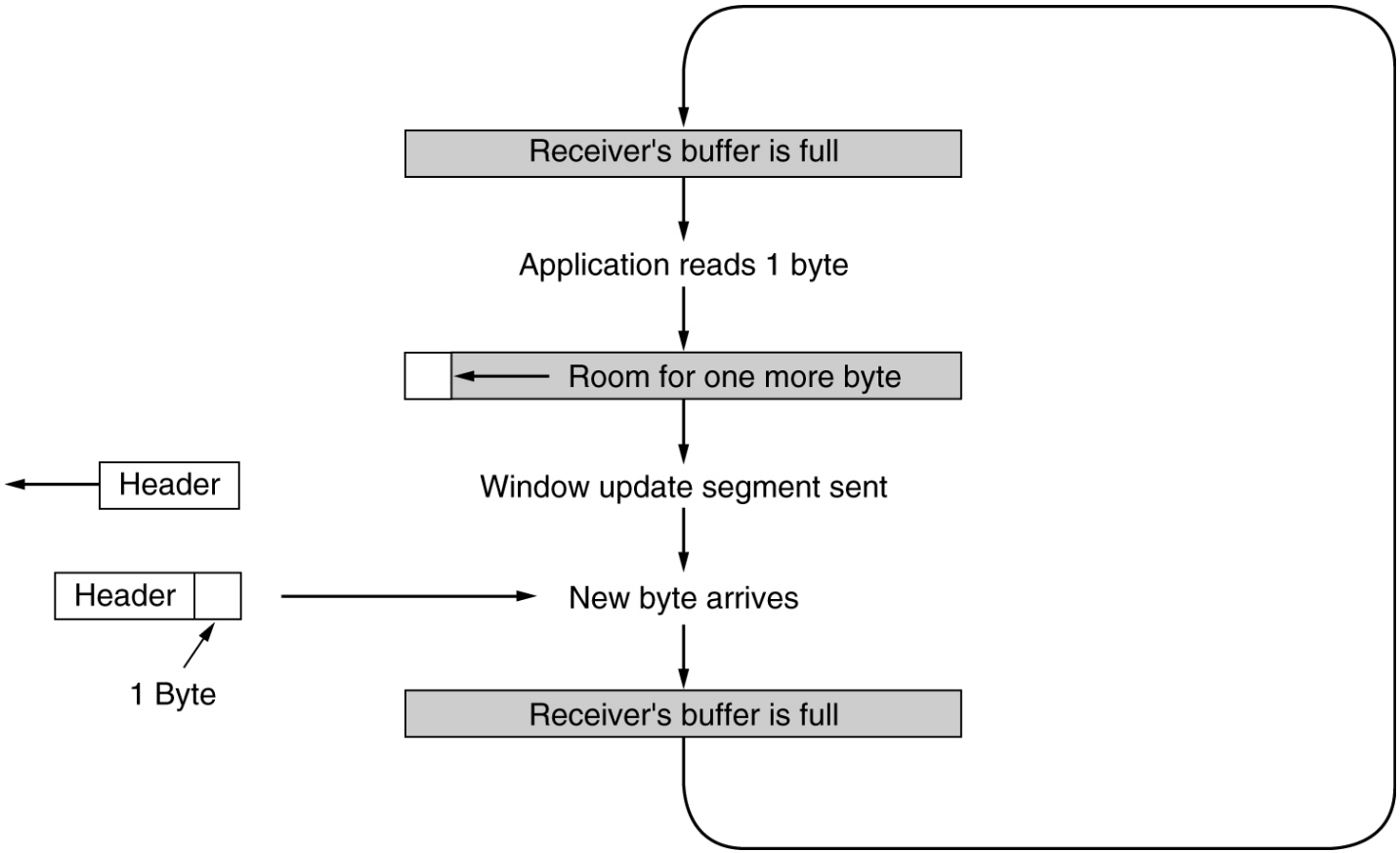
From: *Computer Networks*, 3rd Ed. by Andrew S. Tanenbaum, (c)1996 Prentice Hall.

TCP Transmission Policy



Window management in TCP.

TCP Transmission Policy (2)



Silly window syndrome.

Flow Control Taxonomy

- First generation
  - ignores network state
  - only match receiver rate
  - Examples
    - On-Off, Stop-and-Wait, and Static-Window
- Second generation
  - responsive to state
  - three choices
    - State measurement
      - explicit or implicit
    - Control
      - flow control window size or rate
    - Point of control
      - endpoint or within network

Explicit vs. Implicit

- Explicit
  - Network tells source its current rate
  - Better control
  - More overhead
- Implicit
  - Endpoint figures out rate by looking at network behavior
  - Less overhead
- Ideally, want overhead of implicit with effectiveness of explicit

Flow control window

- Error control window
  - Largest number of packet outstanding (sent but not acked)
- If endpoint has sent all packets in window, it must wait => slows down its rate
- Thus, window provides both error control and flow control
- This is called *transmission window*
- Coupling can be a problem
  - Few buffers are receiver => slow rate!

Window vs. Rate

- In adaptive rate, we directly control rate
- Needs a timer per connection
- Plusses for window
  - no need for fine-grained timer
  - self-limiting
- Plusses for rate
  - better control (finer grain)
  - no coupling of flow control and error control
- Rate control must be careful to avoid overhead and sending too much

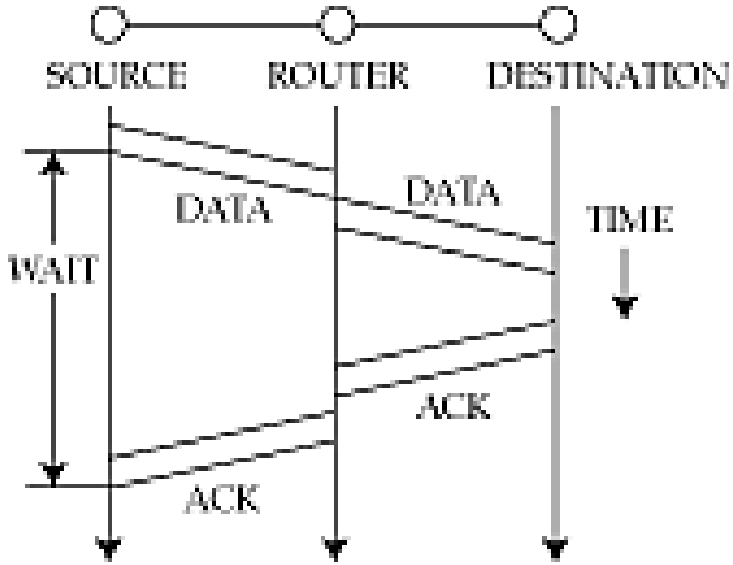
Hop-by-hop vs. end-to-end

- Hop-by-hop
  - first generation flow control at each link
    - next server = sink
  - easy to implement
- End-to-end
  - sender matches all the servers on its path
- Plusses for hop-by-hop
  - simpler
  - distributes overflow
  - better control
- Plusses for end-to-end
  - cheaper

On-off

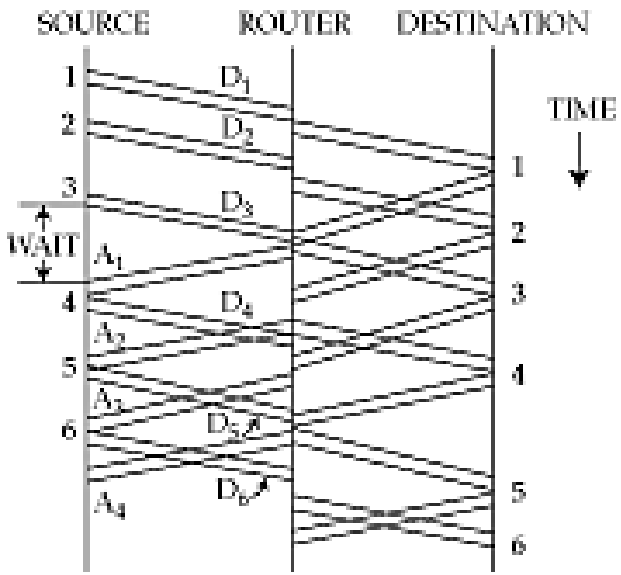
- Receiver gives ON and OFF signals
- If ON, send at full speed
- If OFF, stop
- OK when RTT is small
- What if OFF is lost?
- Bursty
- Used in serial lines or LANs

Stop and Wait



- Send a packet
- Wait for ack before sending next packet

Static window



- Stop and wait can send at most one pkt per RTT
- Here, we allow multiple packets per RTT (= transmission window)

What should window size be?

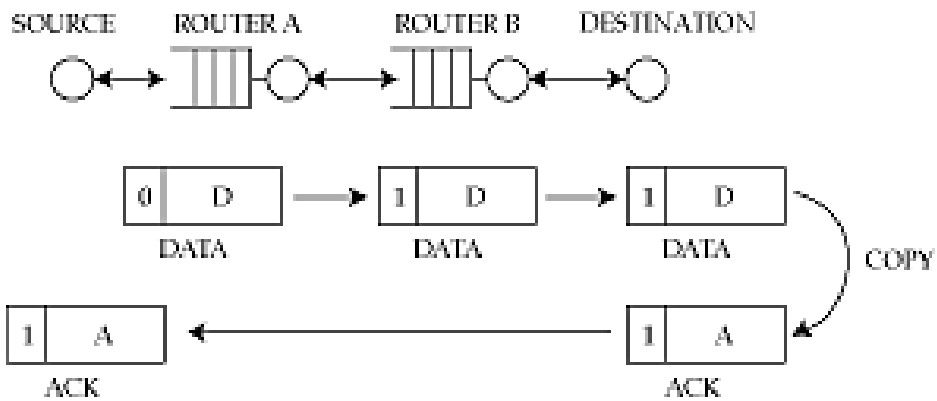
- Let bottleneck service rate along path = b pkts/sec
- Let round trip time = R sec
- Let flow control window = w packet
- Sending rate is w packets in R seconds = w/R
- To use bottleneck w/R > b => w > bR
- This is the bandwidth delay product or optimal window size

Static window

- Works well if b and R are fixed
- But, bottleneck rate changes with time!
- Static choice of w can lead to problems
  - too small
  - too large
- So, need to adapt window
- Always try to get to the current optimal value

DECbit flow control

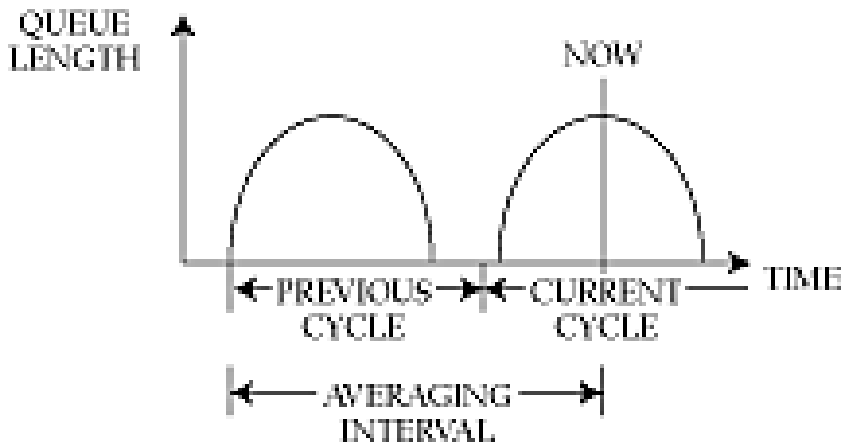
- Intuition
  - every packet has a bit in header
  - intermediate routers set bit if queue has built up => source window is too large
  - sink copies bit to ack
  - if bit is set, source reduces window size
  - in steady state, oscillate around optimal size



DECbit

- When do bits get set?
- How does a source interpret them?

DECbit details: router actions



- Measure demand and mean queue length of each source
- Computed over queue regeneration cycles
- Balance between sensitivity and stability

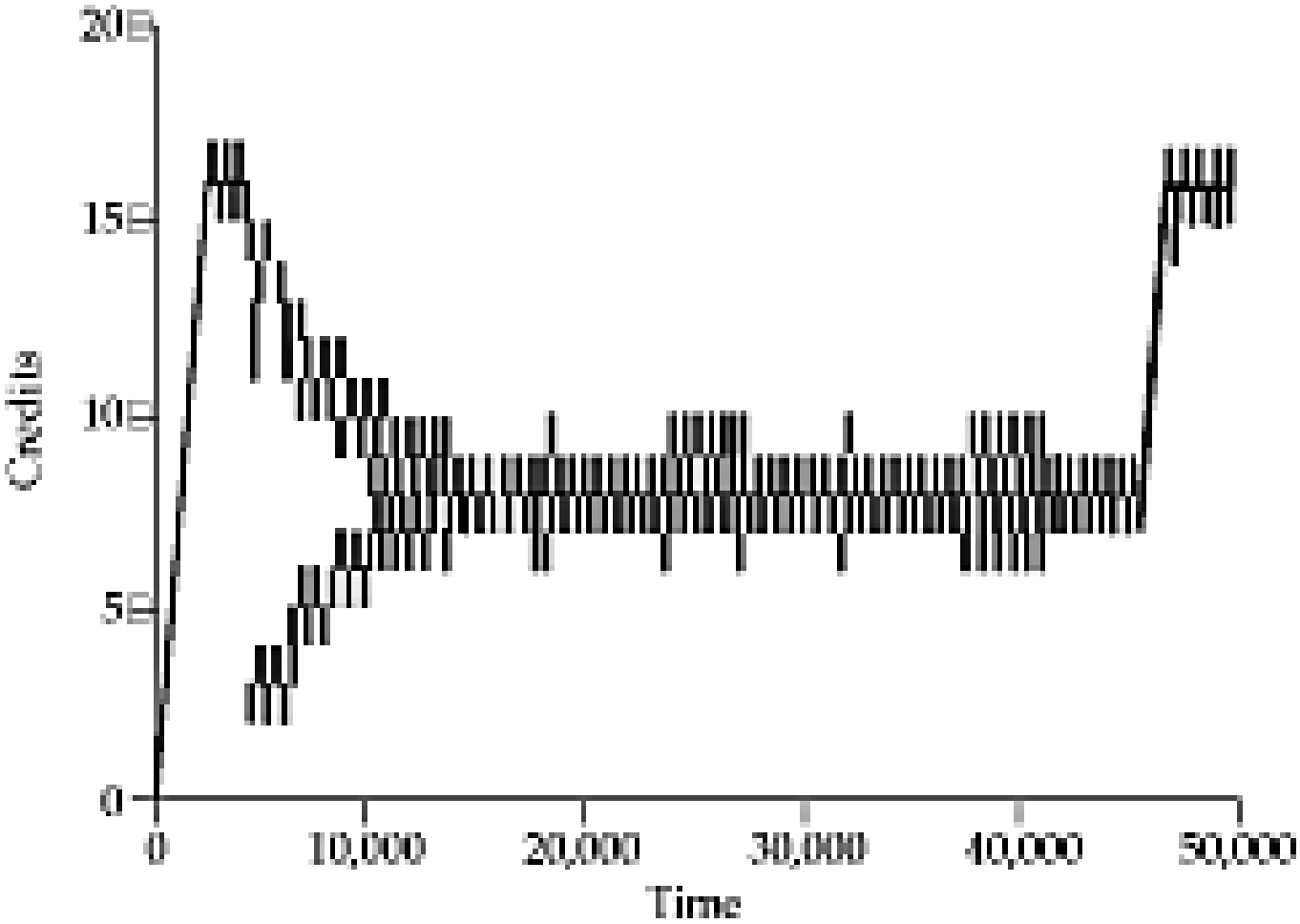
Router actions

- If mean queue length > 1.0
  - set bits on sources whose demand exceeds fair share
- If it exceeds 2.0
  - set bits on everyone
  - panic!

Source actions

- Keep track of bits
- Can't take control actions too fast!
- Wait for past change to take effect
- Measure bits over past + present window size
- If more than 50% set, then decrease window, else increase
- Additive increase, multiplicative decrease

Sample trace



Two Sources

Evaluation

- Works with FIFO
  - but requires per-connection state (demand)
- Software
- But
  - assumes cooperation!
  - conservative window increase policy

TCP Flow Control

- Implicit
- Dynamic window
- End-to-end
- Very similar to DECbit, but
  - no support from routers
  - increase if no loss (usually detected using timeout)
  - window decrease on a timeout
  - additive increase multiplicative decrease

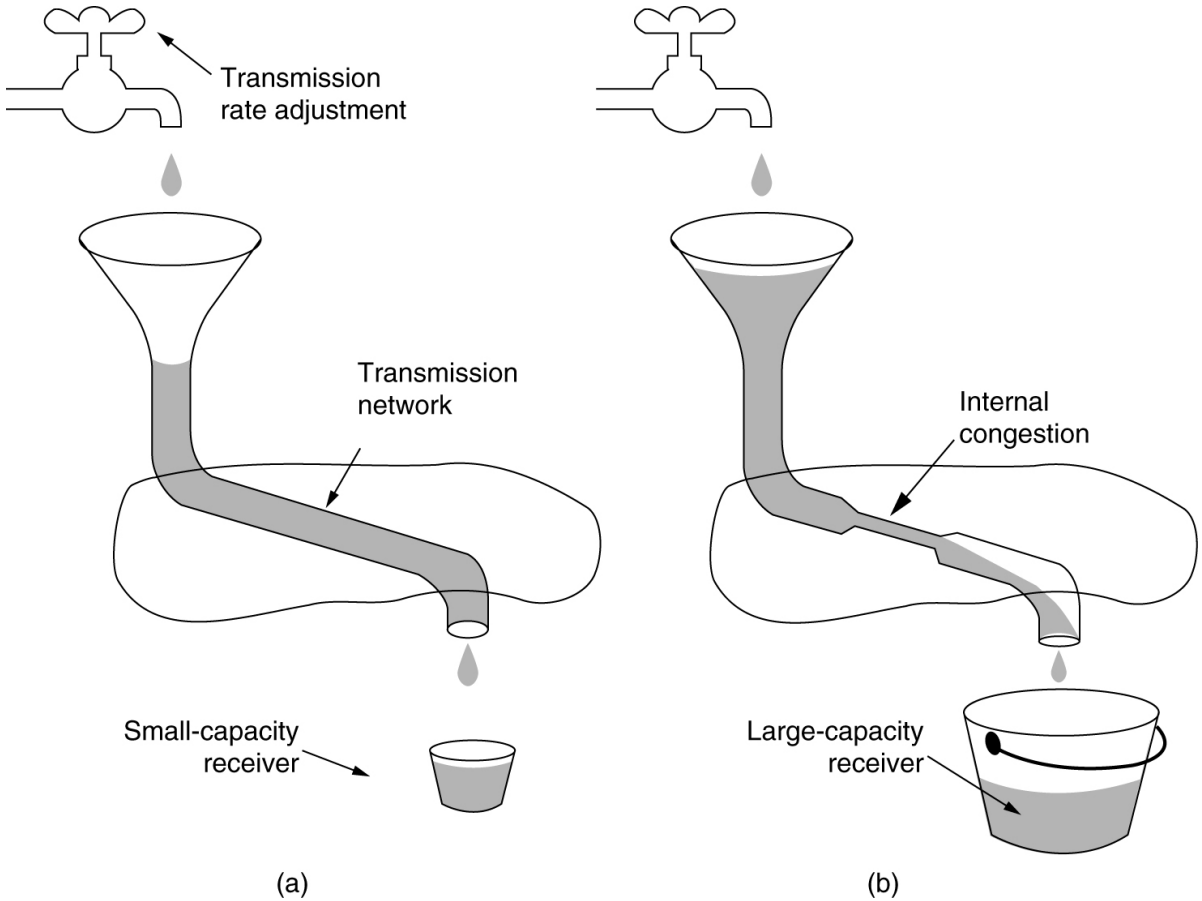
TCP Congestion Control

- Detecting Congestion
  - In general it is difficult
  - But, consider why a packet might be dropped
    - link error - but links are very reliable now
    - buffer overflow --> congestion
  - Use re-transmission timeouts as an estimate of congestion
- Dealing with Congestion
  - add a second window (congestion window)
    - limit transmissions to min(recv window, congestion window)
  - start with congestion window = max segment window
    - initial max segment is one kilo-byte
    - on a ACK without a timeout
      - if window < threshold, increment by one max segment
      - otherwise increment by initial max segment
  - on timeout
    - cut threshold in half
    - set window size to initial max segment

TCP details

- Window starts at 1
- Increases exponentially for a while, then linearly
- Exponentially => doubles every RTT
- Linearly => increases by 1 every RTT
- During exponential phase, every ack results in window increase by 1
- During linear phase, window increases by 1 when # acks = window size
- Exponential phase is called slow start
- Linear phase is called congestion avoidance

TCP Congestion Control

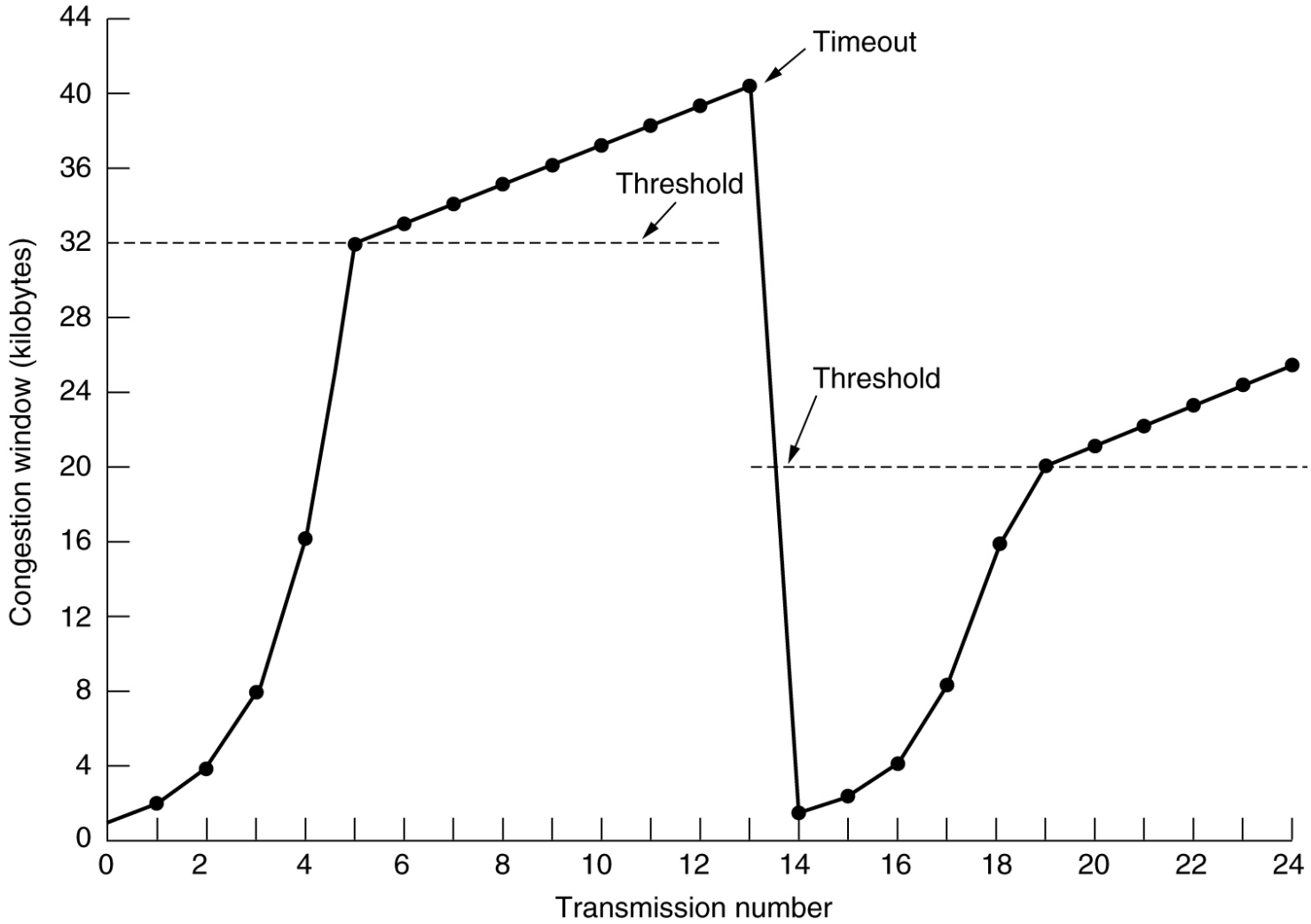


(a) A fast network feeding a low capacity receiver.

(b) A slow network feeding a high-capacity receiver.

Spring 2003

TCP Congestion Control (2)



An example of the Internet congestion algorithm.

More TCP details

- On a loss, current window size is stored in a variable called slow start threshold or *ssthresh*
- Switch from exponential to linear (slow start to congestion avoidance) when window size reaches threshold
- Loss detected either with timeout or fast retransmit (duplicate cumulative acks)
- Two versions of TCP
  - Tahoe: in both cases, drop window to 1, ssthresh to half the current window size, enter slow start
  - Reno: on timeout, drop window to 1, and on fast retransmit drop window to half previous size (also, increase window on subsequent acks)
    - After fast retransmit send a pkt for every duplicate ack - called fast recovery

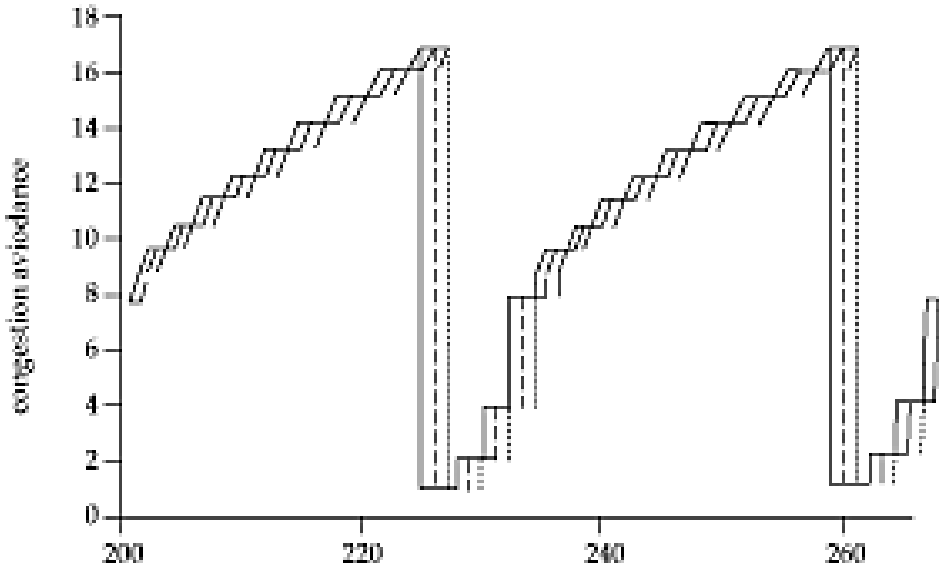
TCP vs. DECbit

- Both use dynamic window flow control and additive-increase multiplicative decrease
- TCP uses implicit measurement of congestion
  - probe a black box
- Operates at the cliff
- Source does not filter information

Evaluation

- Effective over a wide range of bandwidths
- A lot of operational experience
- Weaknesses
  - loss => overload? (wireless)
  - overload detected only on a loss
    - in steady state, source induces loss
  - needs at least bR/3 buffers per connection

Sample trace

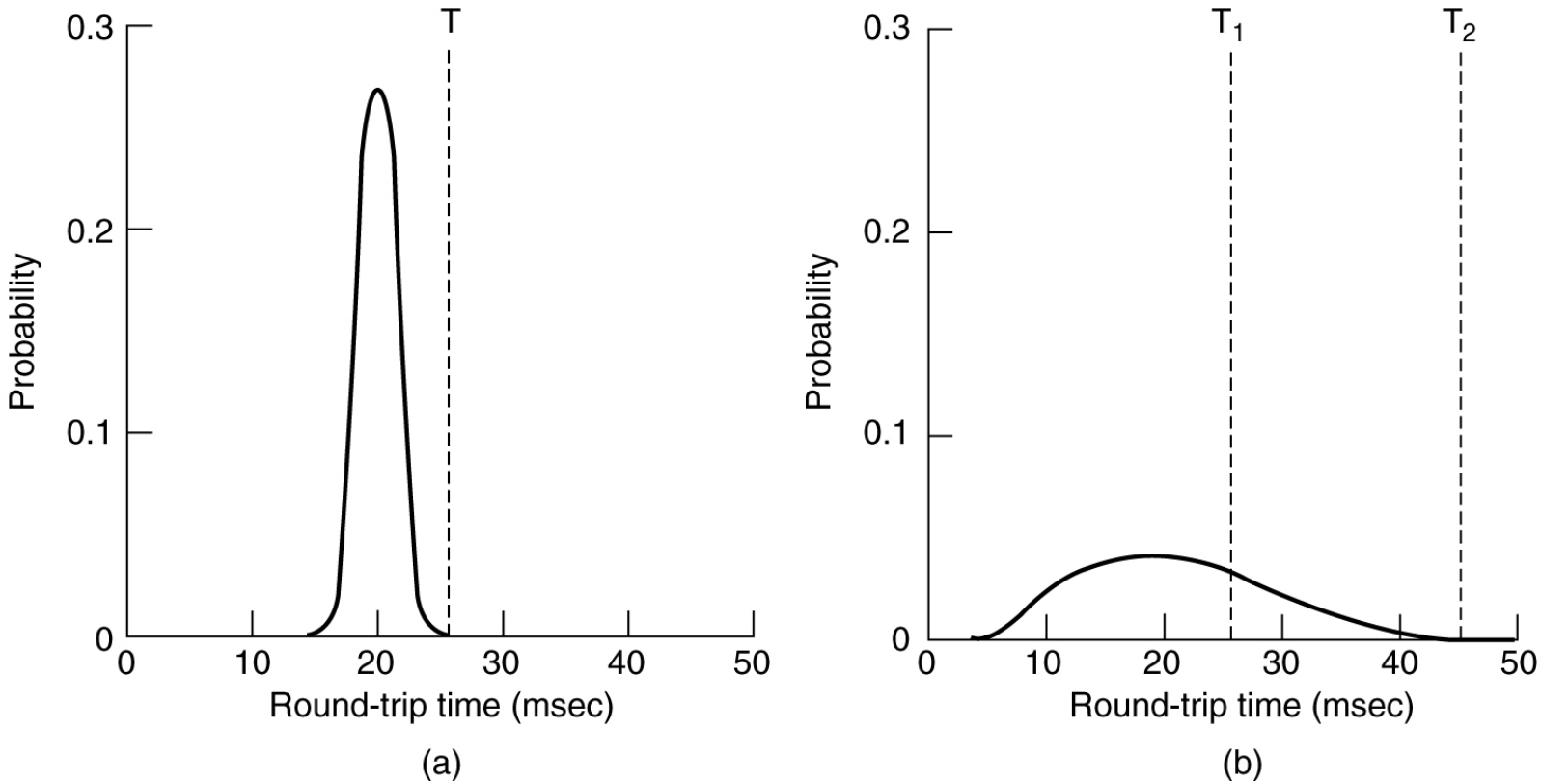


Three Source - TCP Tahoe

TCP Vegas

- Expected throughput = transmission_window_size/propagation_delay
- Numerator: known
- Denominator: measure smallest RTT
- Also know actual throughput
- Difference = how much to reduce/increase rate
- Algorithm
  - send a special packet
  - on ack, compute expected and actual throughput
  - (expected - actual)* RTT packets in bottleneck buffer
  - adjust sending rate if this is too large
- Works better than TCP Reno

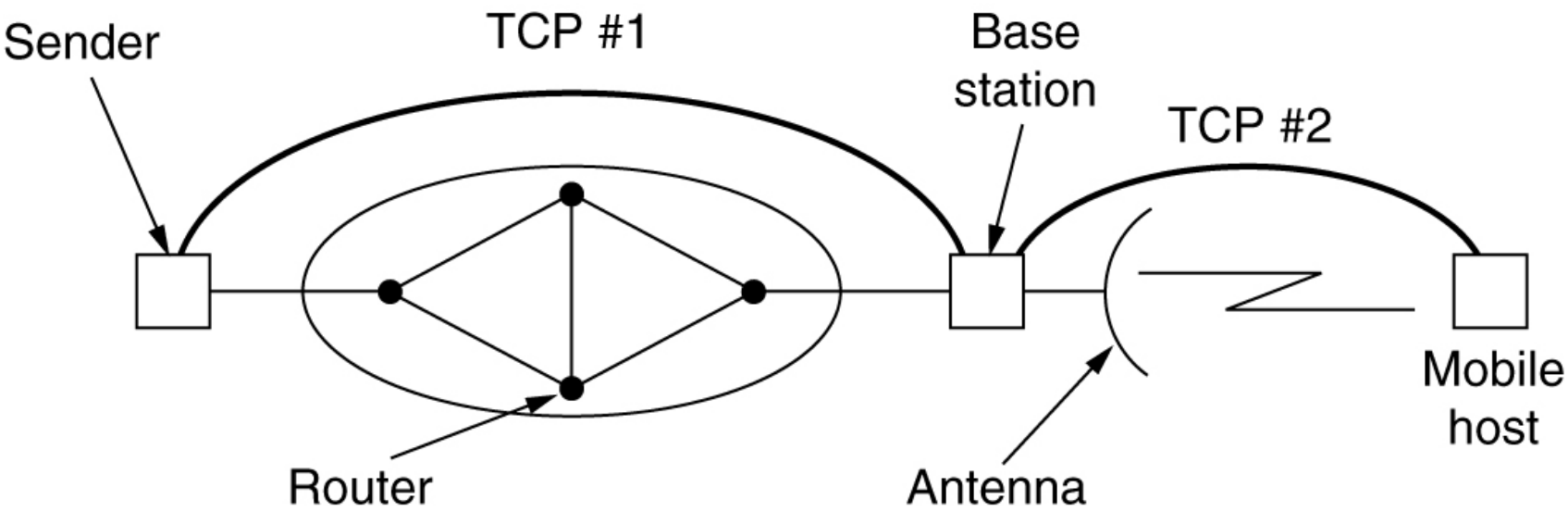
TCP Timer Management



(a) Probability density of ACK arrival times in the data link layer.

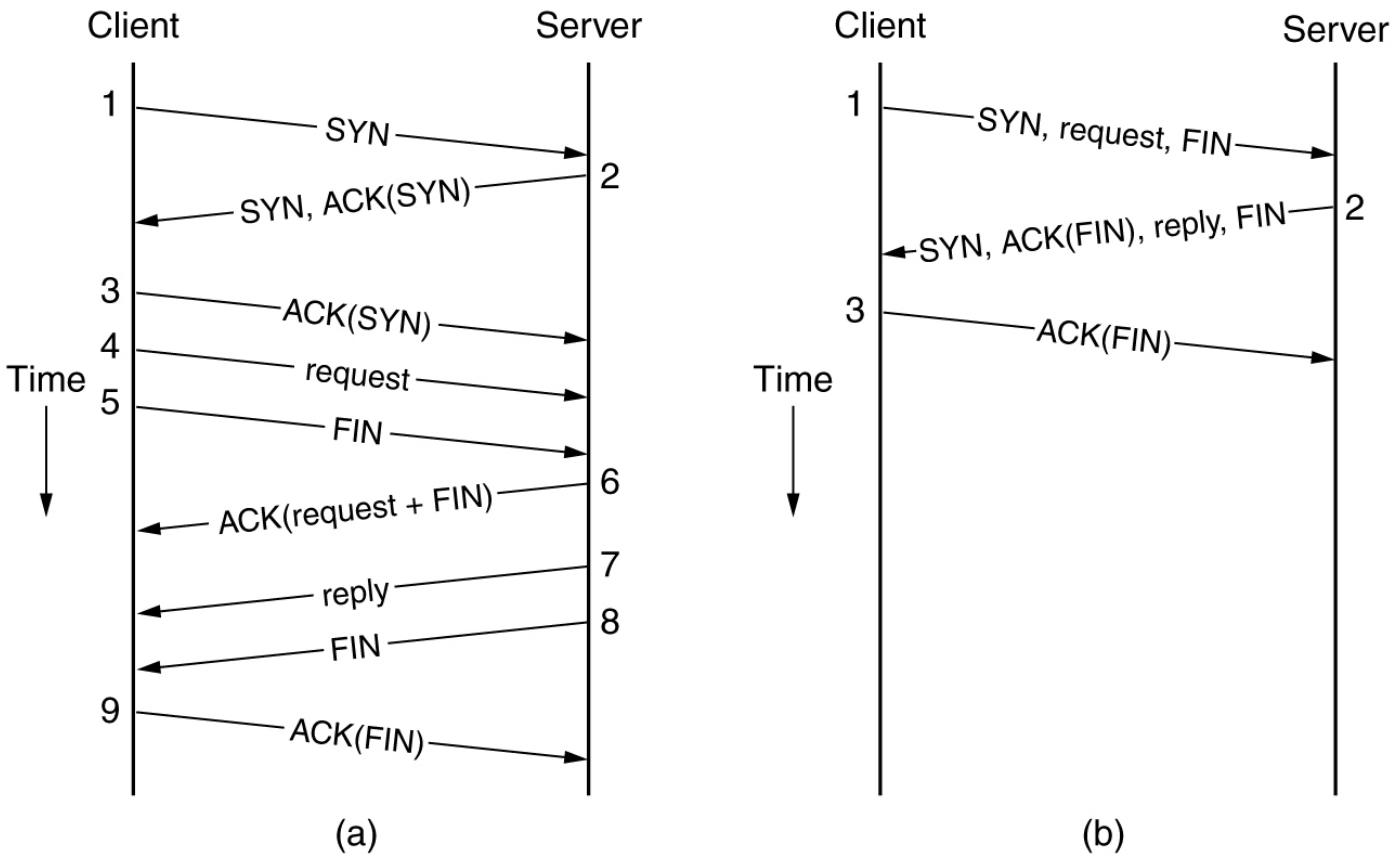
(b) Probability density of ACK arrival times for TCP.

Wireless TCP and UDP



Splitting a TCP connection into two connections.

Transitional TCP



(a) RPC using normal TPC.

(b) RPC using T/TCP.

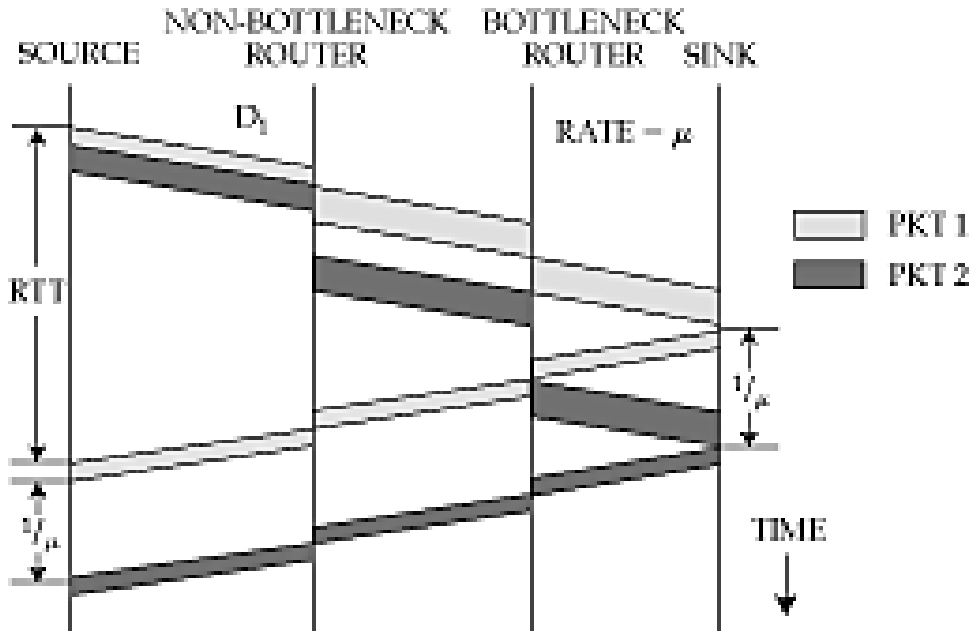
NETBLT

- First rate-based flow control scheme
- Separates error control (window) and flow control (no coupling)
- So, losses and retransmissions do not affect the flow rate
- Application data sent as a series of buffers, each at a particular rate
- Rate = (burst size + burst rate) so granularity of control = burst
- Initially, no adjustment of rates
- Later, if received rate < sending rate, multiplicatively decrease rate
- Change rate only once per buffer => slow

Packet pair

- Improves basic ideas in NETBLT
  - better measurement of bottleneck
  - control based on prediction
  - finer granularity
- Assume all bottlenecks serve packets in round robin order
- Then, spacing between packets at receiver (= ack spacing) = 1/(rate of slowest server)
- If all data sent as paired packets, no distinction between data and probes
- Implicitly determine service rates if servers are round-robin-like

Packet pair


SOURCE
NON-BOTTLENECK ROUTER
BOTTLENECK ROUTER
SINK
D_1
RATE = μ
PKT 1
PKT 2
RTT
$1/\mu$
$1/\mu$
TIME

Packet-pair details

- Acks give time series of service rates in the past
- We can use this to predict the next rate
- Exponential average, with fuzzy rules to change the averaging factor
- Predicted rate feeds into flow control equation

Packet-pair flow control

Let X = # packets in bottleneck buffer

S = # outstanding packets

R = RTT

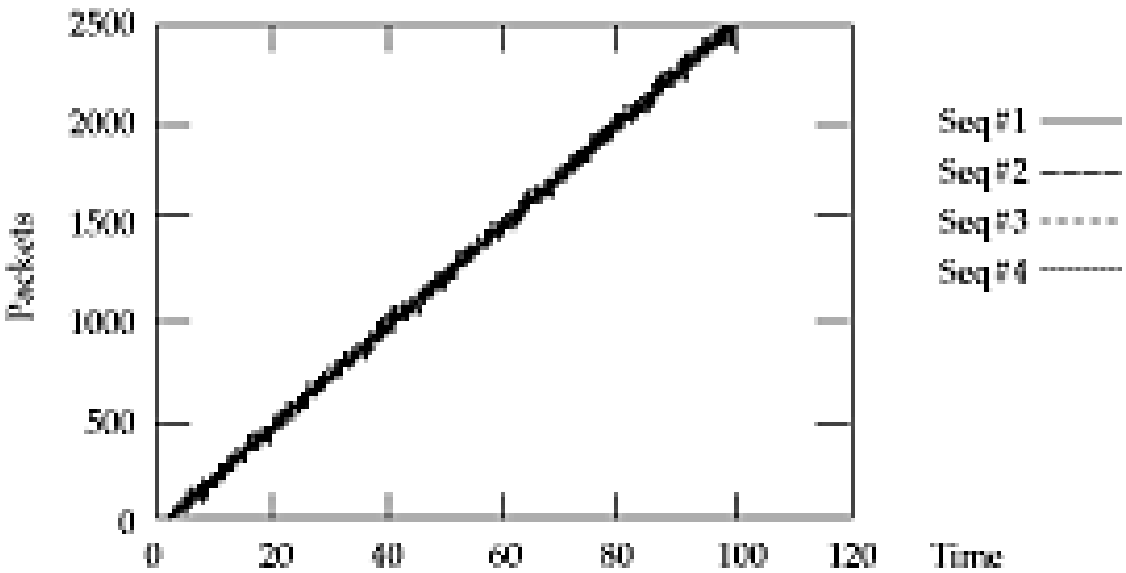
b = bottleneck rate

Then, X = S - Rb (assuming no losses)

Let l = source rate

l(k+1) = b(k+1) + (setpoint -X)/R

Sample trace



TCP Timer Management

- Problem: How to pick timeout value?
  - need to estimate round-trip latency
  - need low variance in round trip latency
- Solution: dynamic estimates of RTT
  - RTT = a RTT + (1 - a) M
    - M time of an ACK
    - a = 7/8
  - Need to pick retransmission time
    - old policy, use Timeout = RTT b , with b = 2
    - estimate standard deviation of RTT using mean deviation
      - D = a D + (1 - a) | RTT - M |
      - Timeout = RTT + 4 * D
  - How to update RTT on retransmission's
    - double Timeout on a retransmission

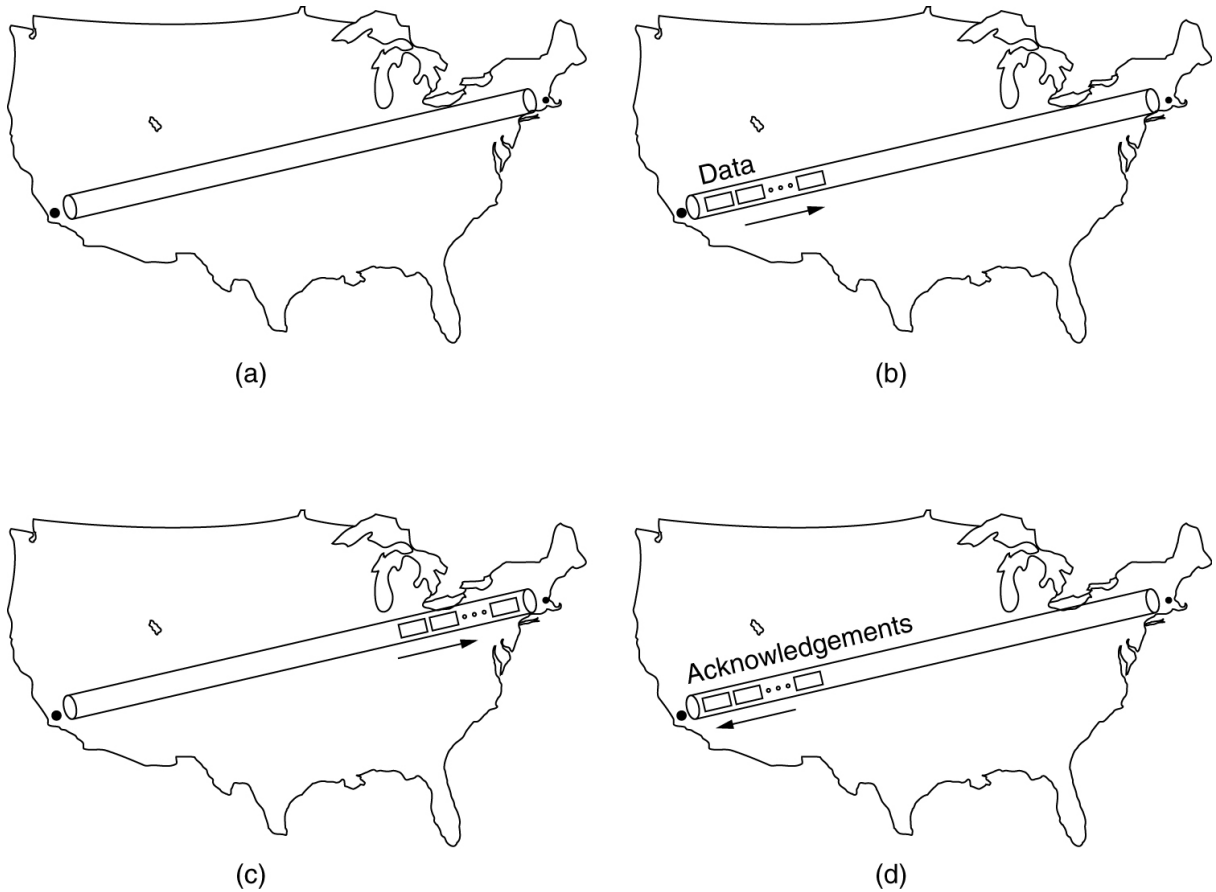
Other TCP Timers

- Persistence Timer
  - Prevents deadlock due to dropped window packets
    - This is a problem if the window is set to 0
- Keepalive Timer
  - Prevents half dead connections
  - may consume bandwidth
  - may kill live connections when net hiccups
- TIMED Wait
  - prevents re-use of a connection before max packet life is over
  - set to twice max packet lifetime

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 μsec, (c) After 20 msec, (d) after 40 msec.

Network Performance Measurement

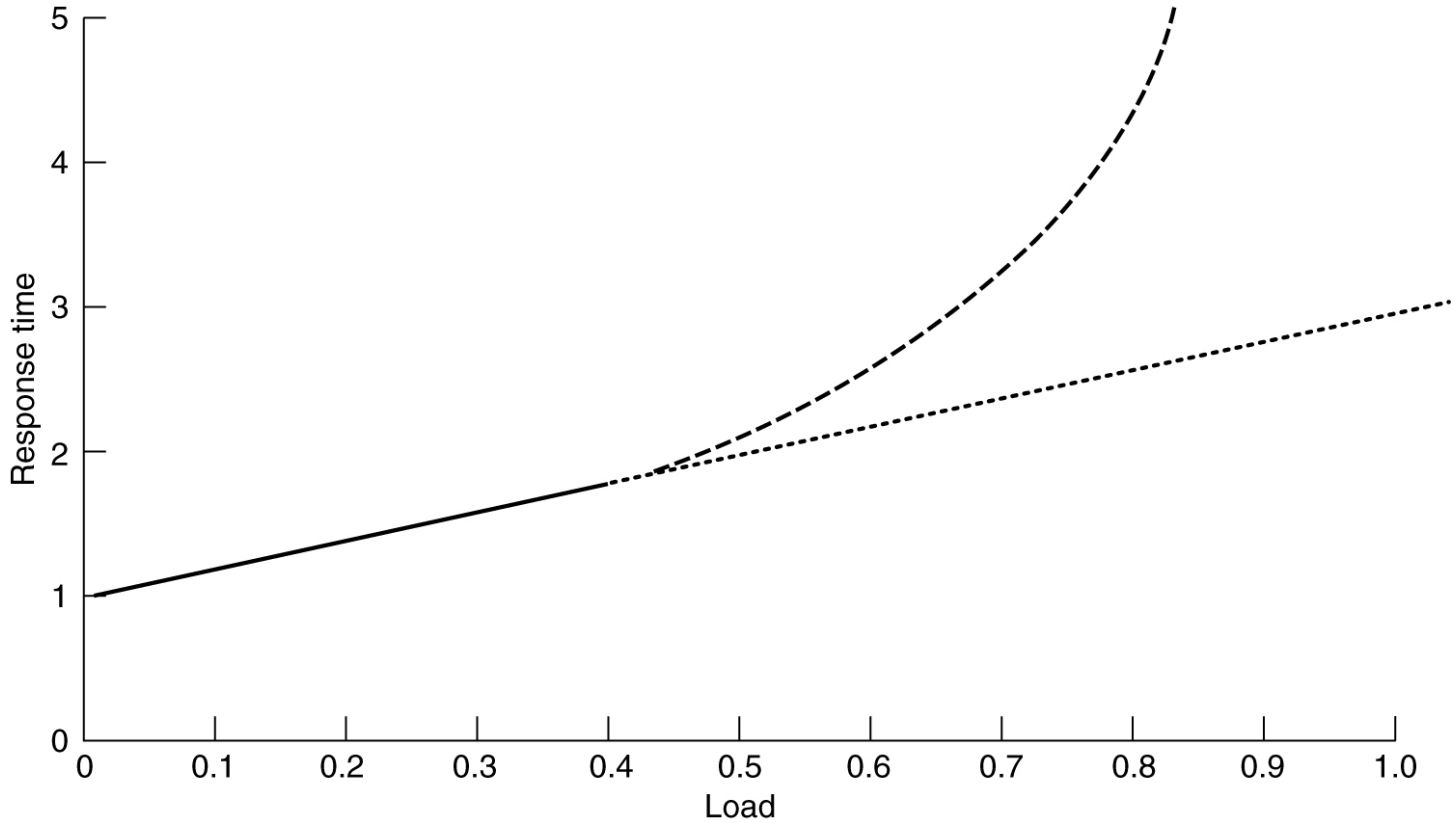
- The basic loop for improving network performance.
- Measure relevant network parameters, performance.
- Try to understand what is going on.
- Change one parameter.

System Design for Better Performance

Rules:

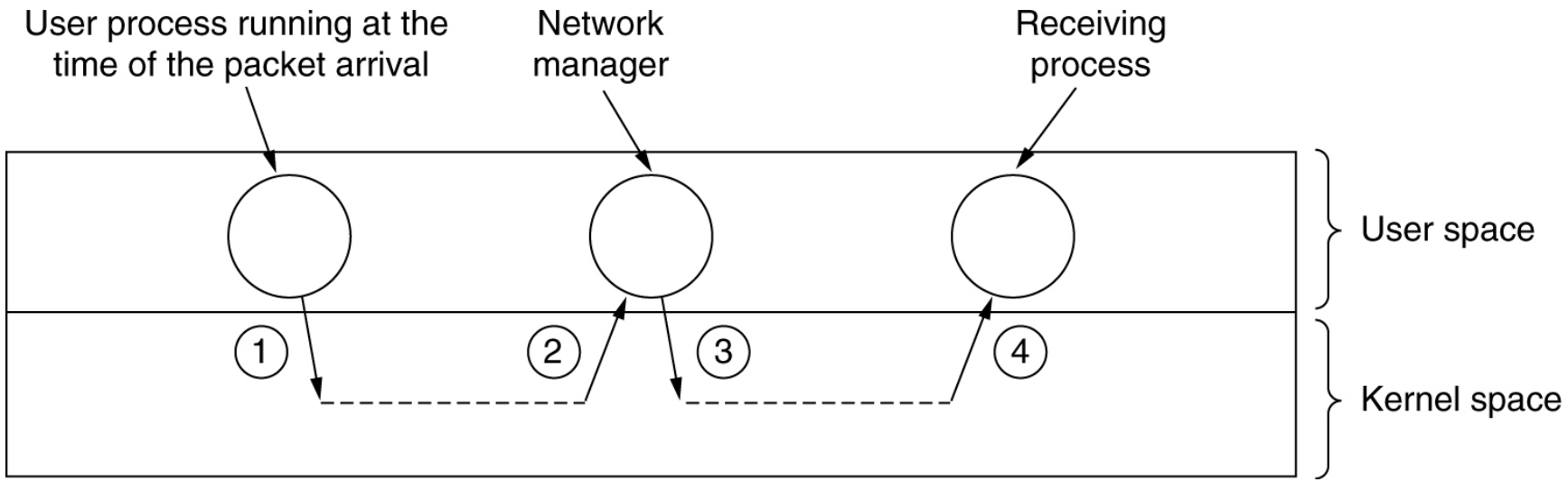
- CPU speed is more important than network speed.
- Reduce packet count to reduce software overhead.
- Minimize context switches.
- Minimize copying.
- You can buy more bandwidth but not lower delay.
- Avoiding congestion is better than recovering from it.
- Avoid timeouts.

System Design for Better Performance (2)



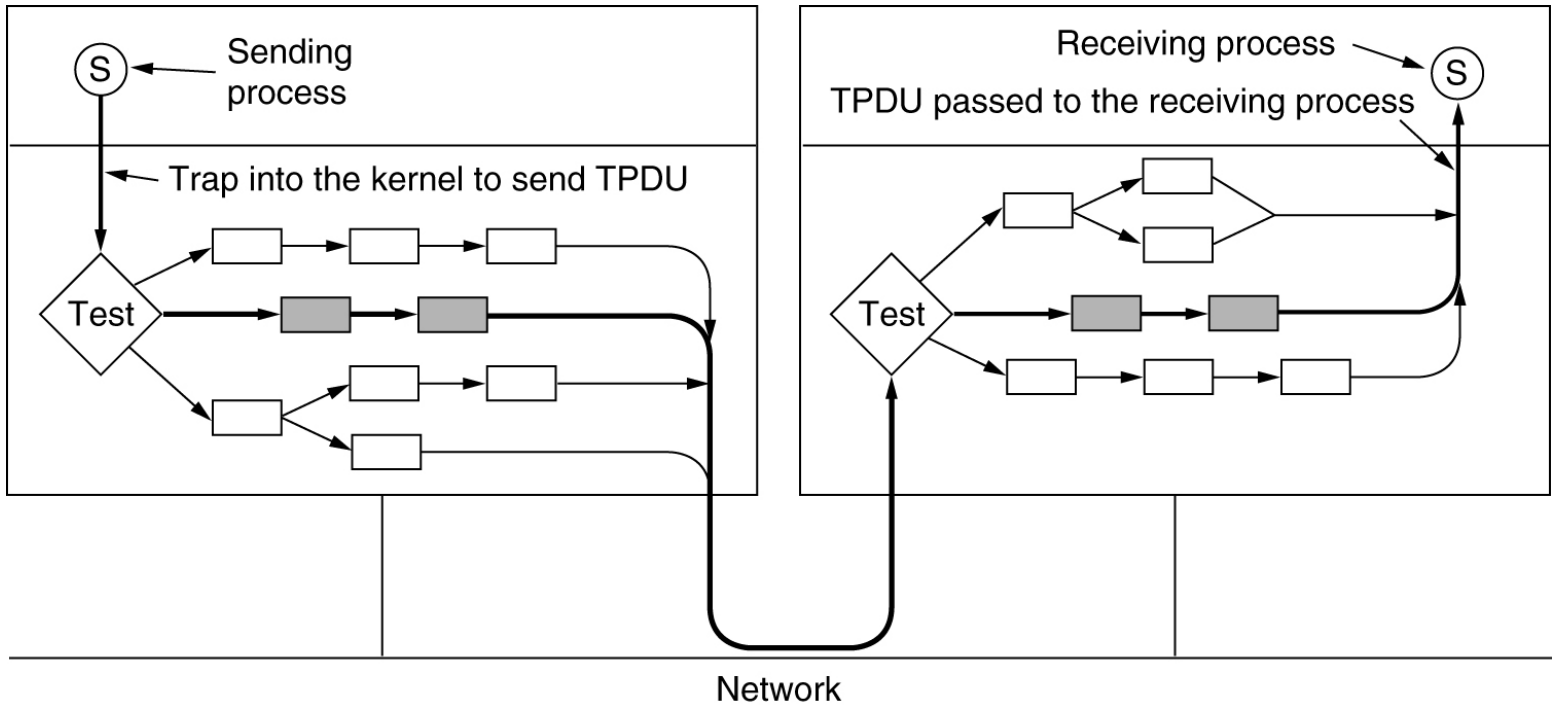
Response as a function of load.

System Design for Better Performance (3)



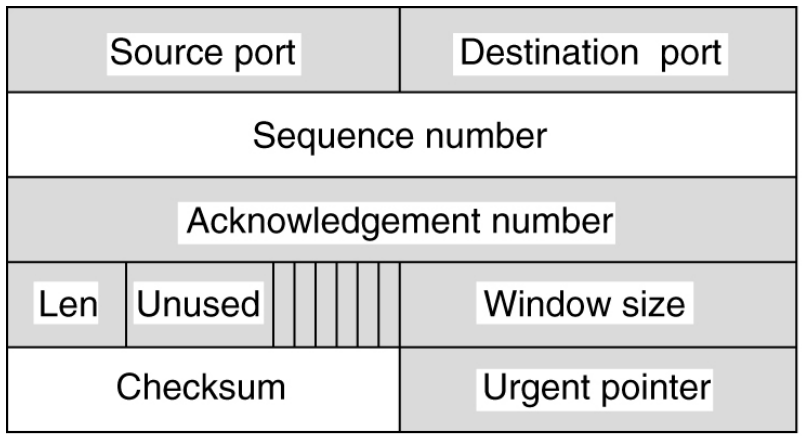
Four context switches to handle one packet with a user-space network manager.

Fast TPDU Processing

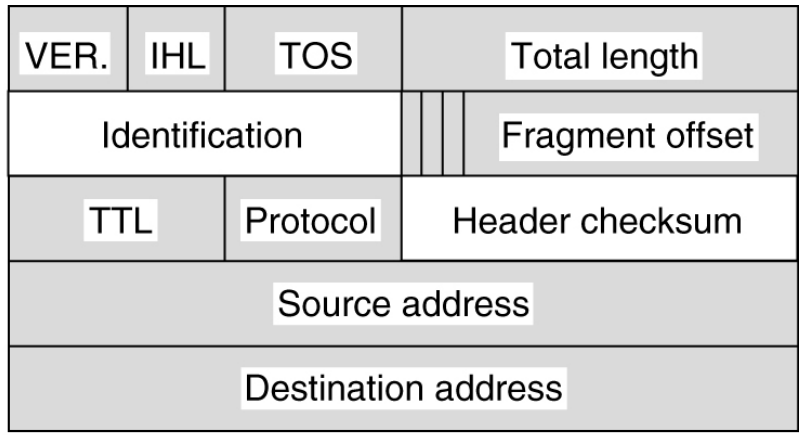


The fast path from sender to receiver is shown with a heavy line. The processing steps on this path are shaded.

Fast TPDU Processing (2)



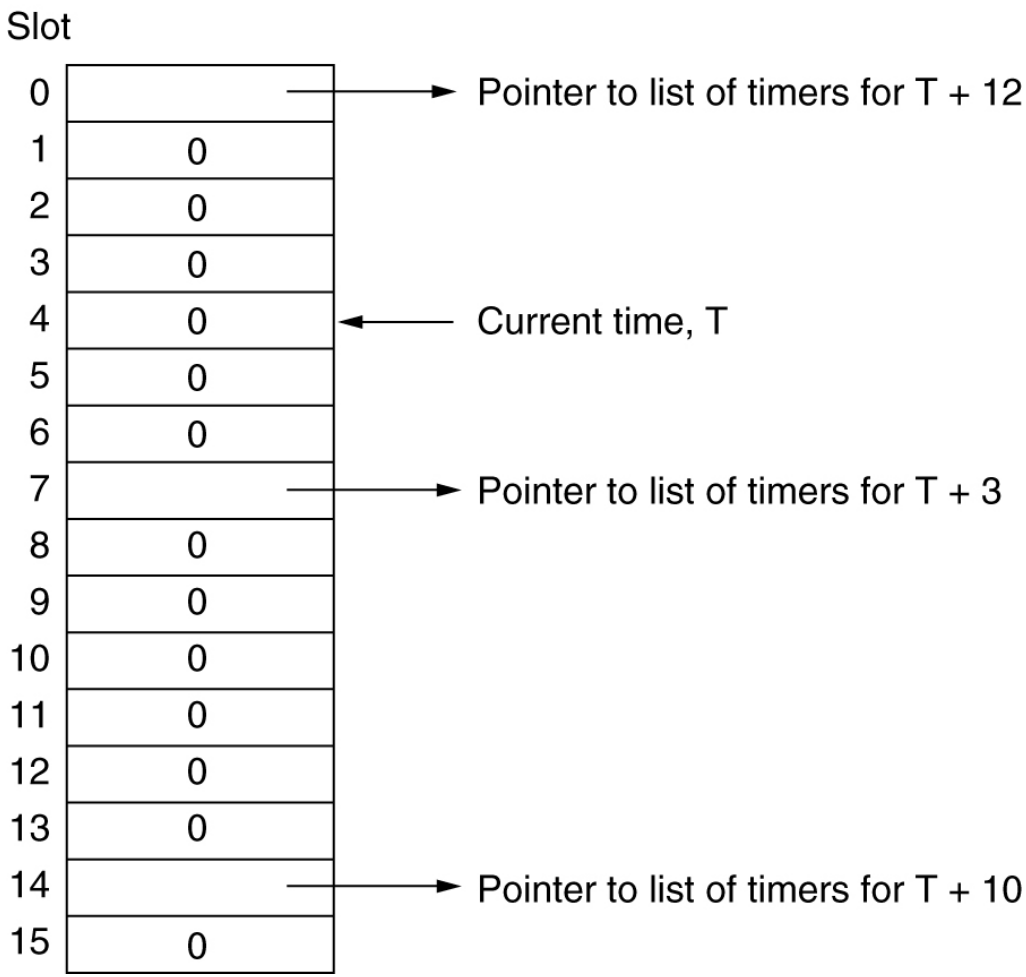
(a)



(b)

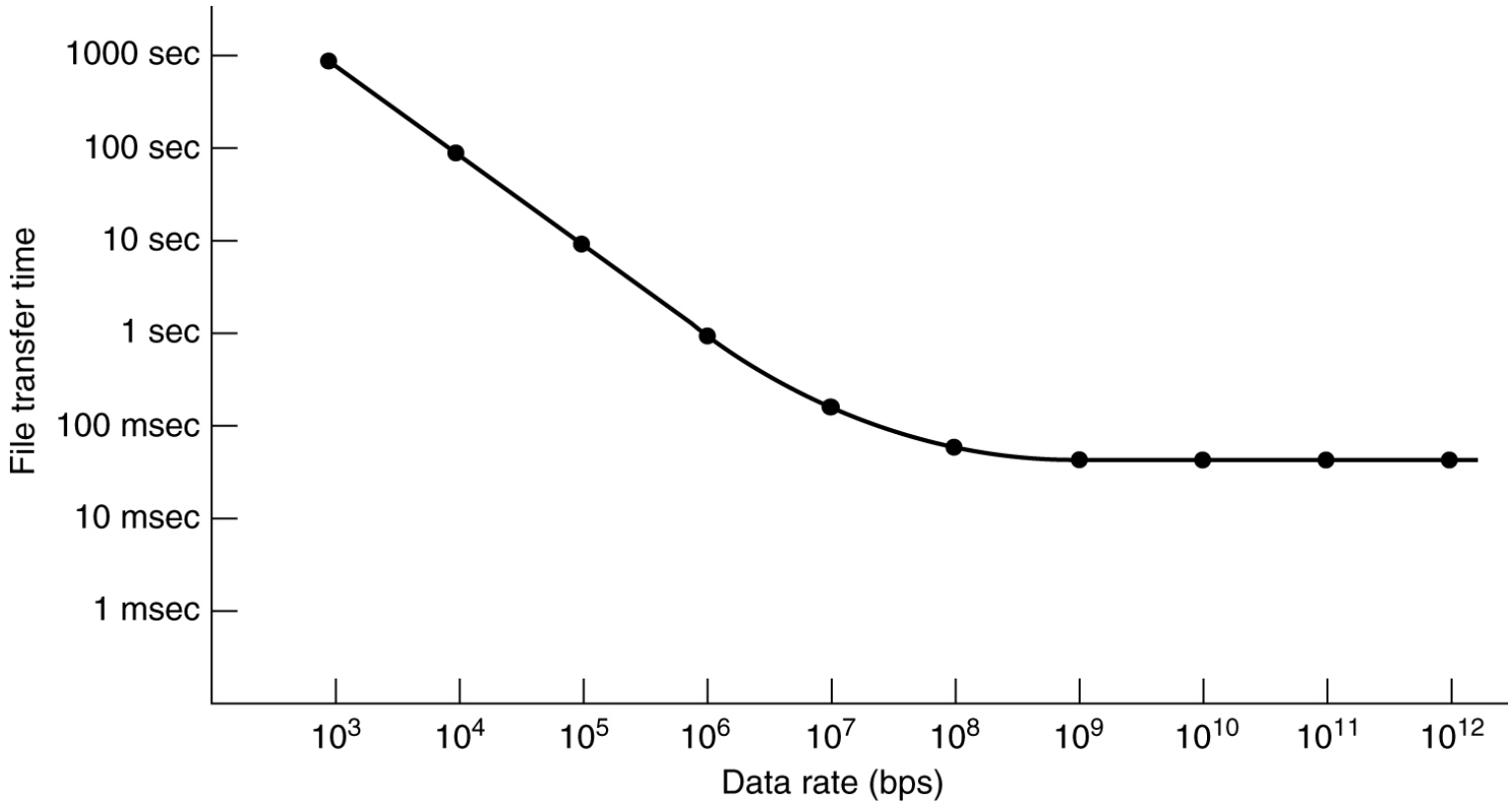
(a) TCP header. (b) IP header. In both cases, the shaded fields are taken from the prototype without change.

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.