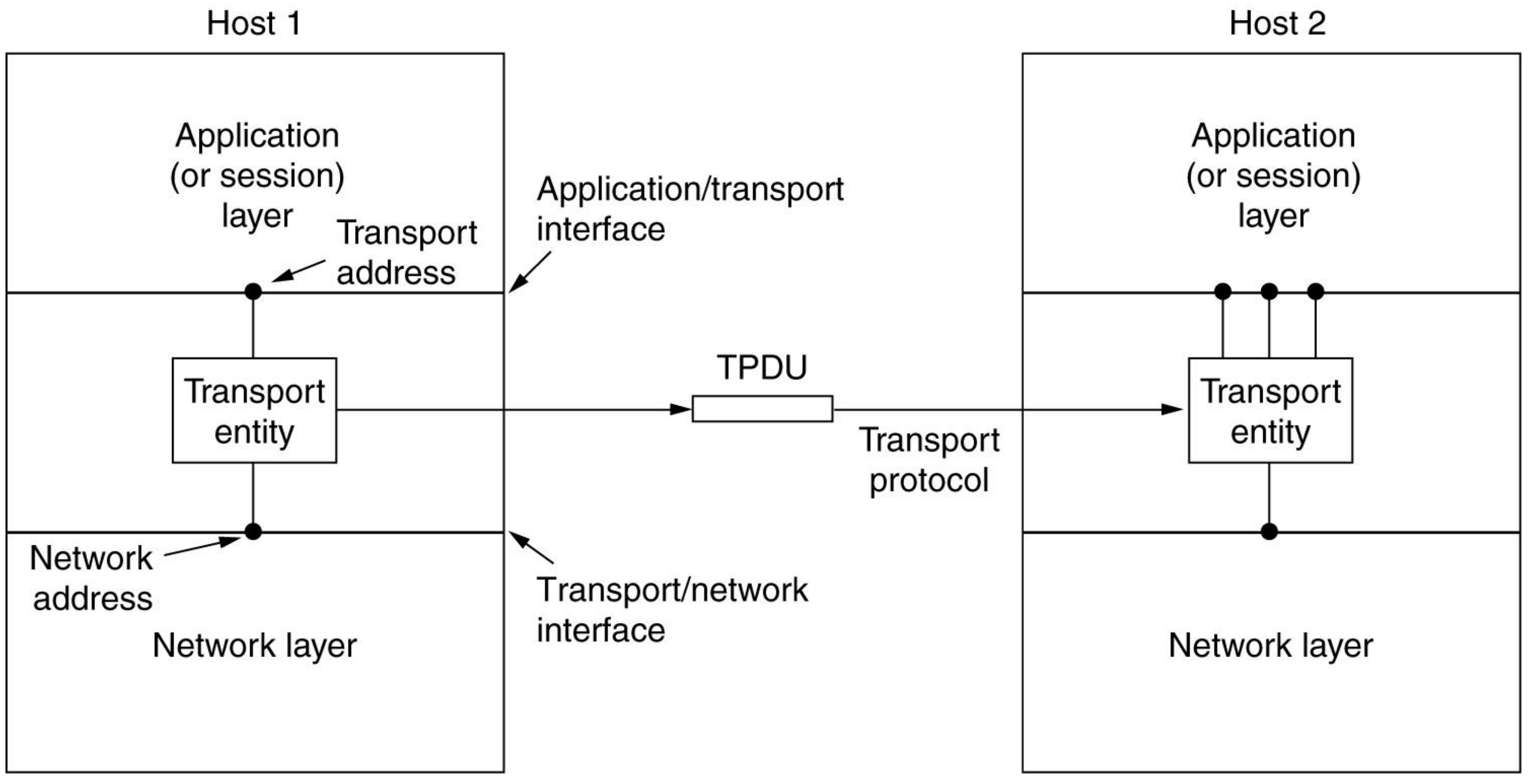
# Chapter 6

## The Transport Layer

# 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



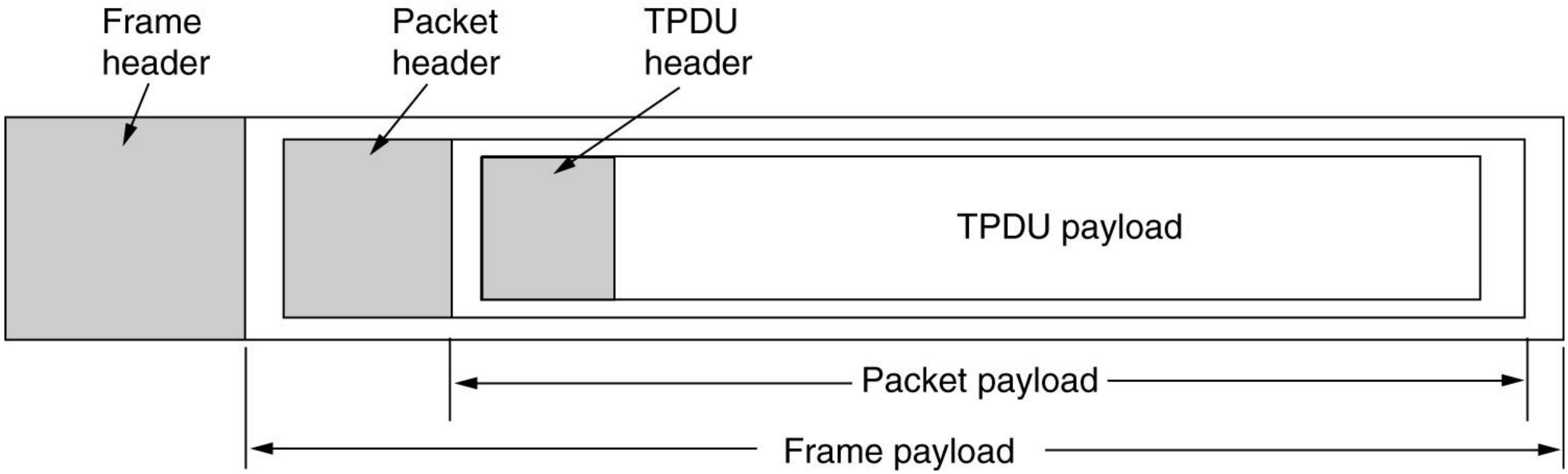

The network, transport, and application layers.

# 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 |

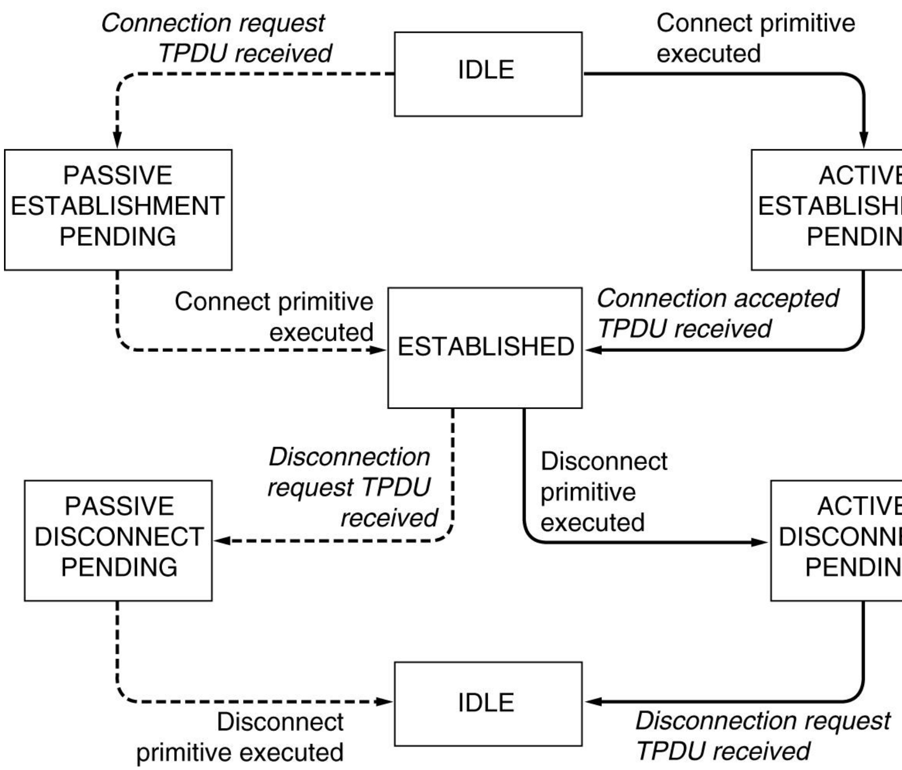
The primitives for a simple transport service.

# 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)

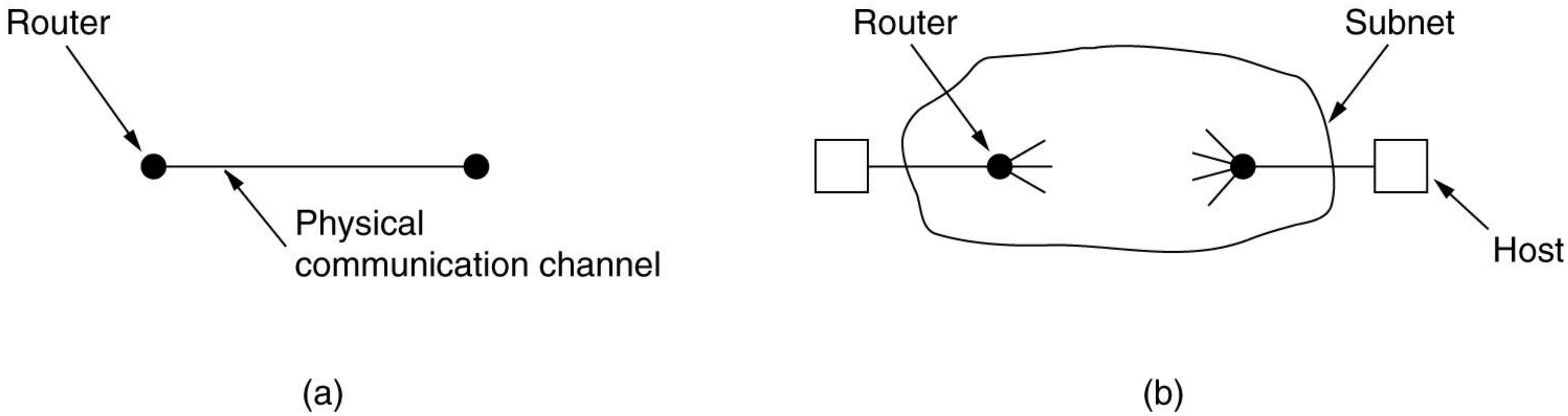
Client code using sockets.

```
#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

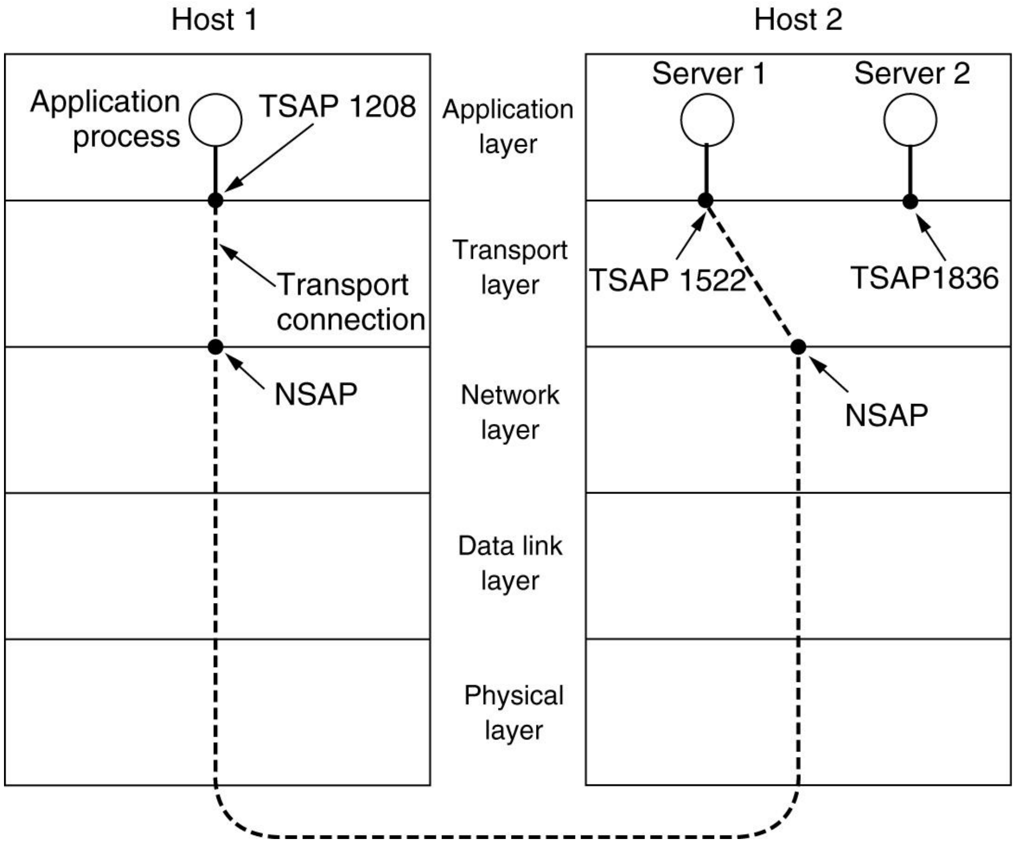
# Transport Protocol




(a) Environment of the data link layer.

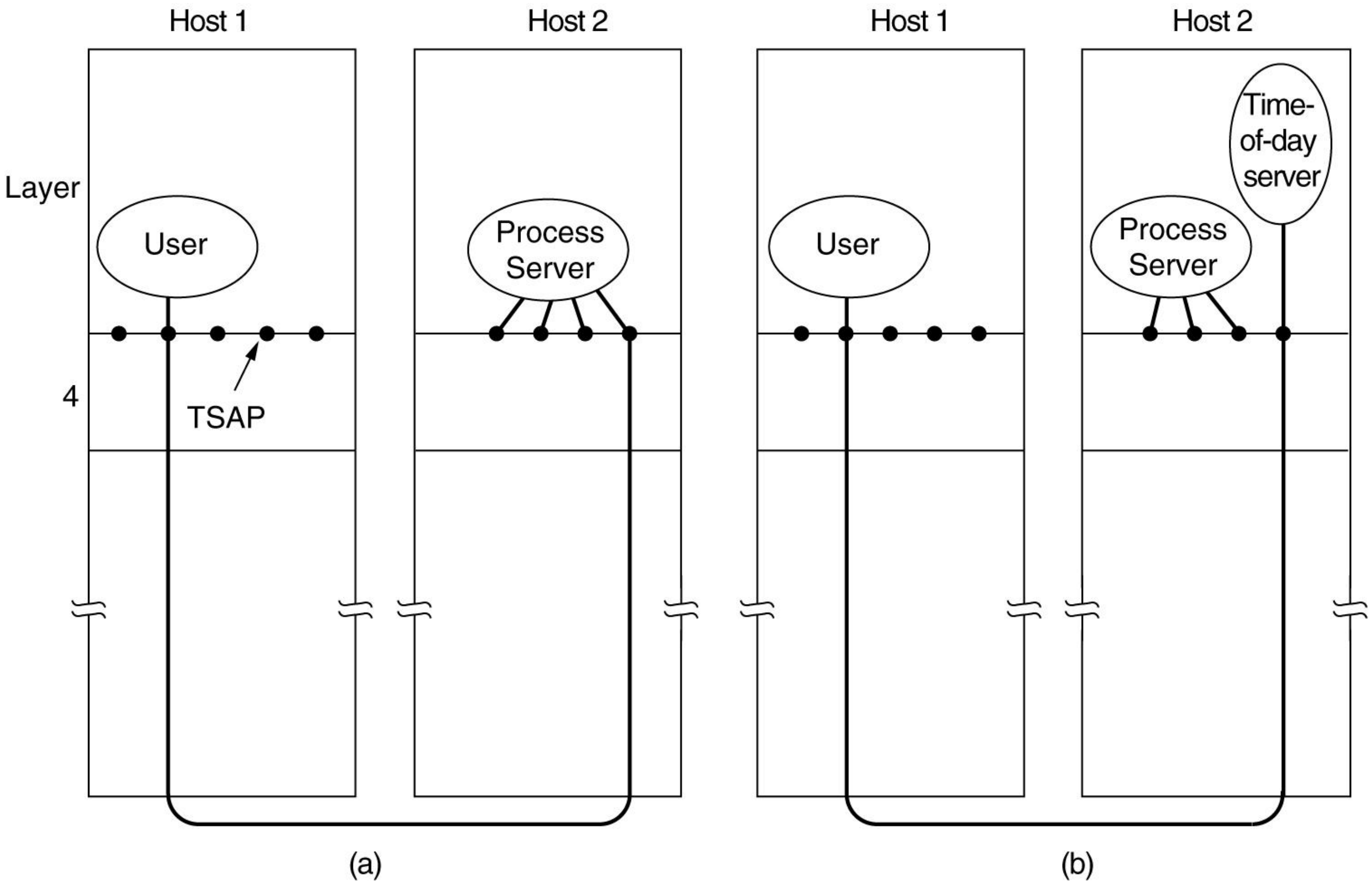
(b) Environment of the transport layer.

# Addressing



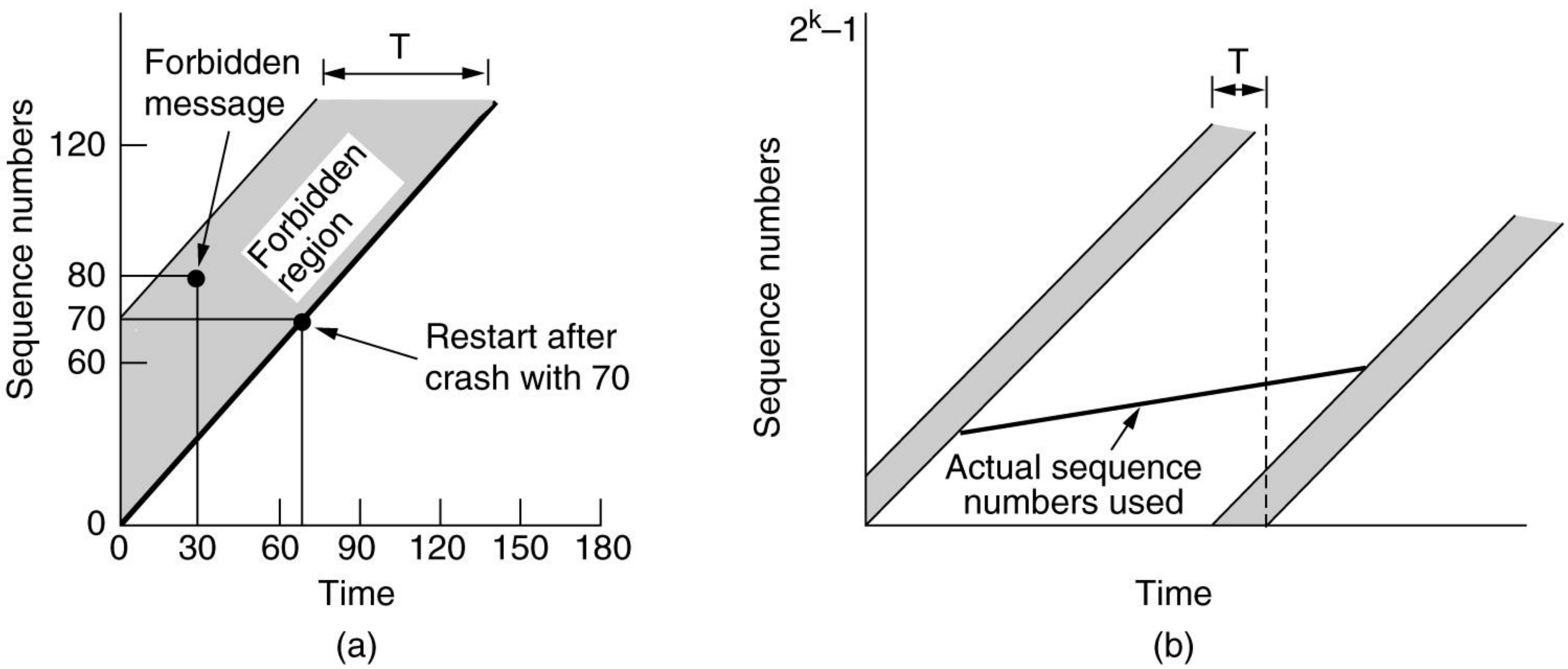

TSAPs, NSAPs and transport connections.

# Connection Establishment




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

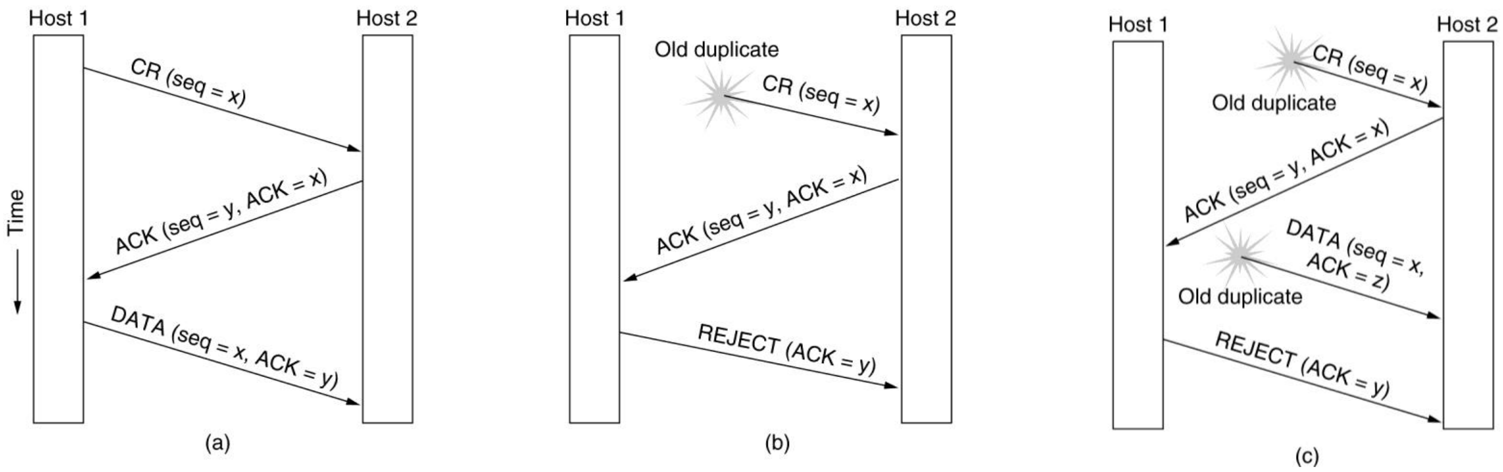
# Connection Establishment (2)




(a) TPDUs may not enter the forbidden region.

(b) The resynchronization problem.

# Connection Establishment (3)



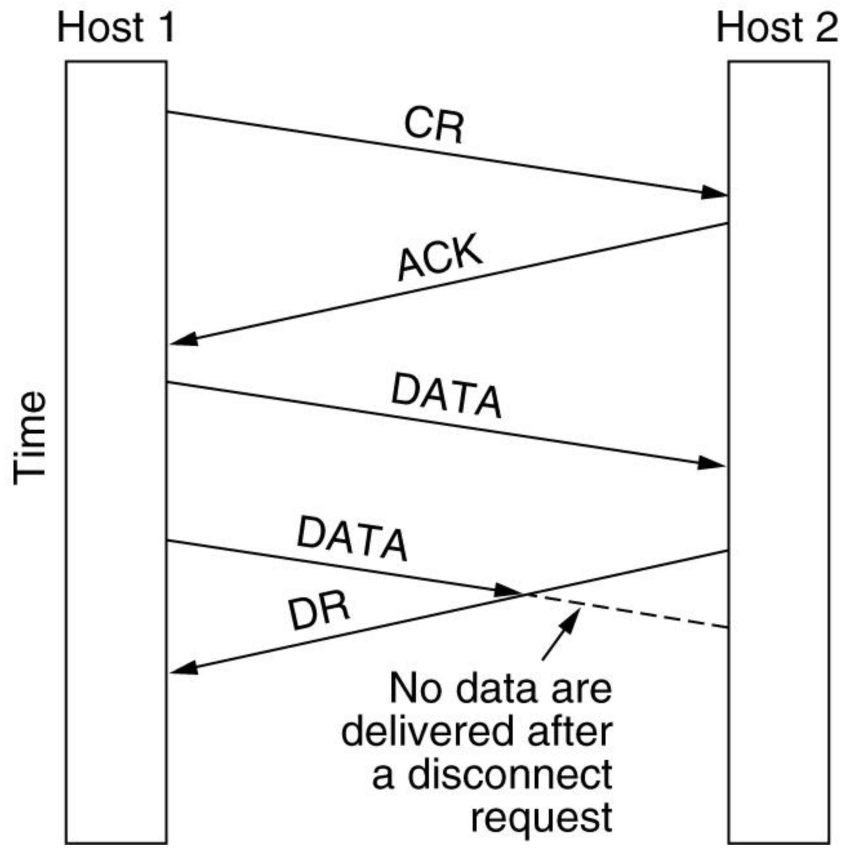

Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST.

(a) Normal operation,

(b) Old CONNECTION REQUEST appearing out of nowhere.

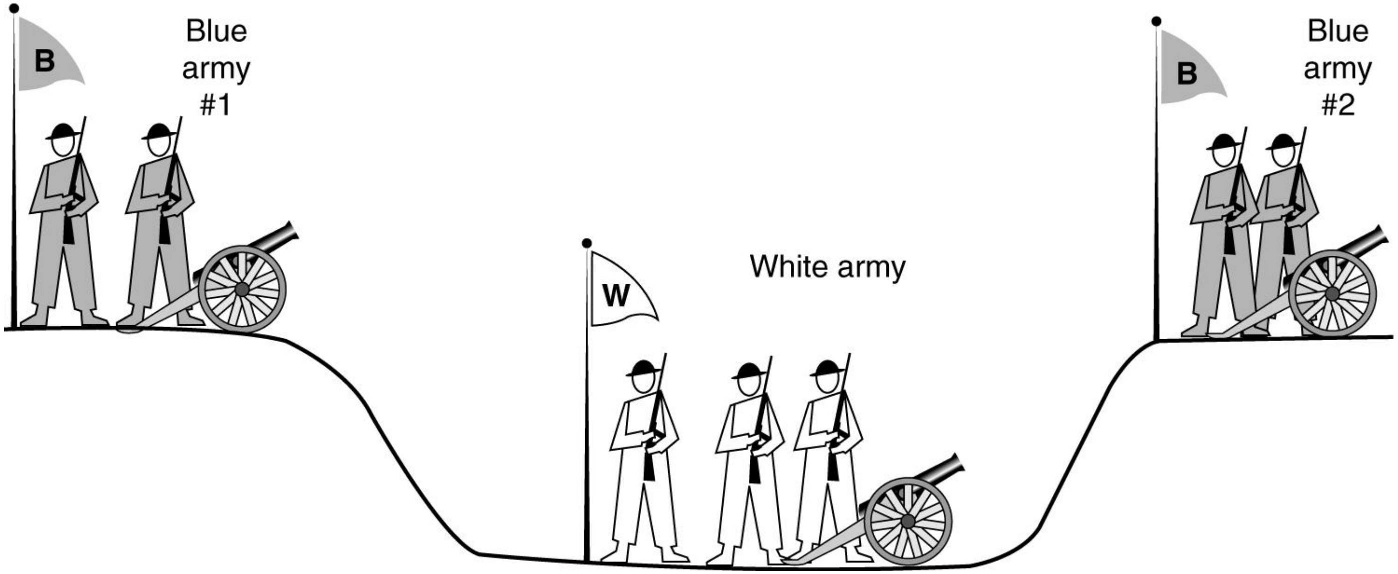
(c) Duplicate CONNECTION REQUEST and duplicate ACK.

# Connection Release



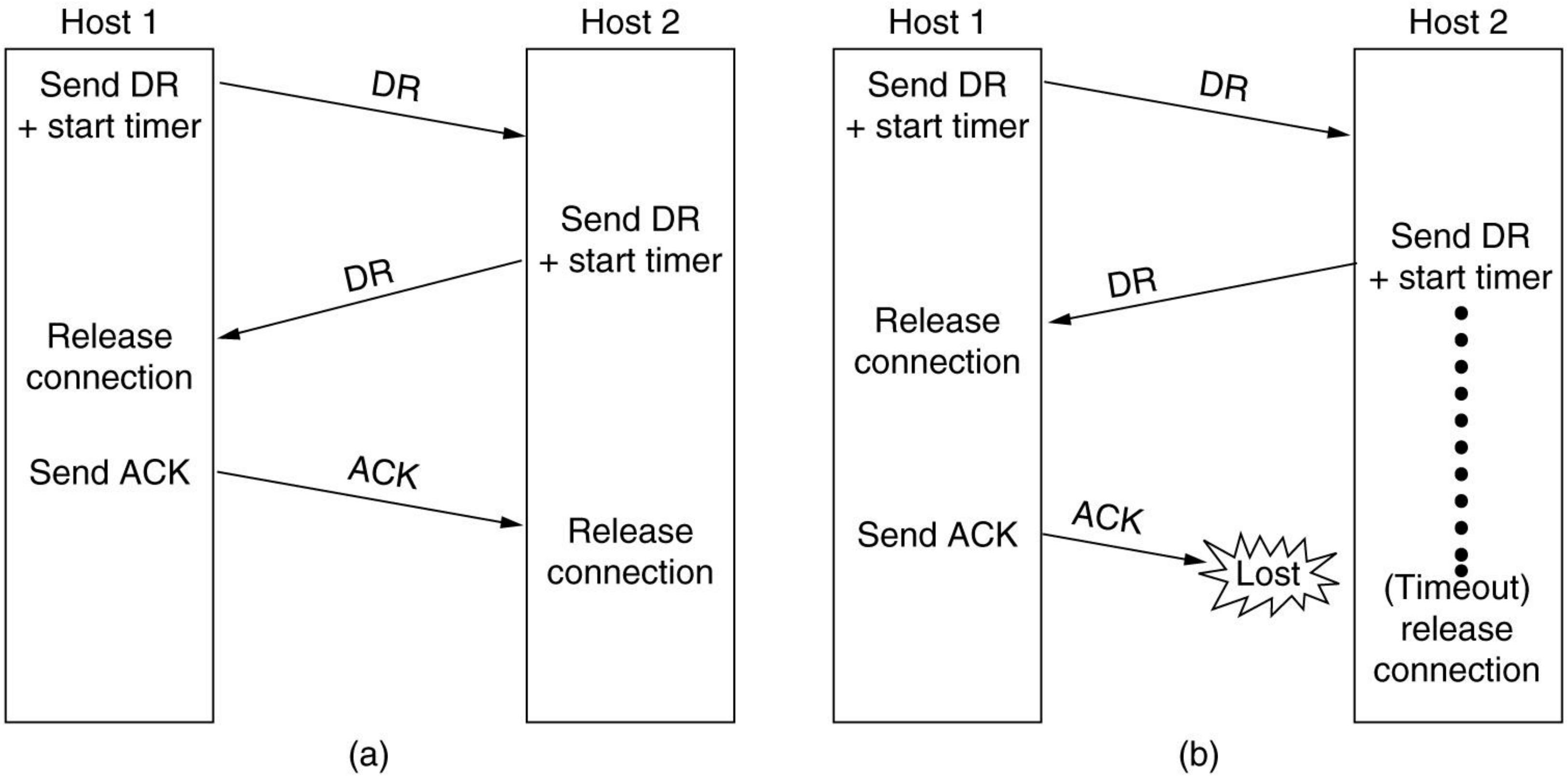

Abrupt disconnection with loss of data.

# Connection Release (2)



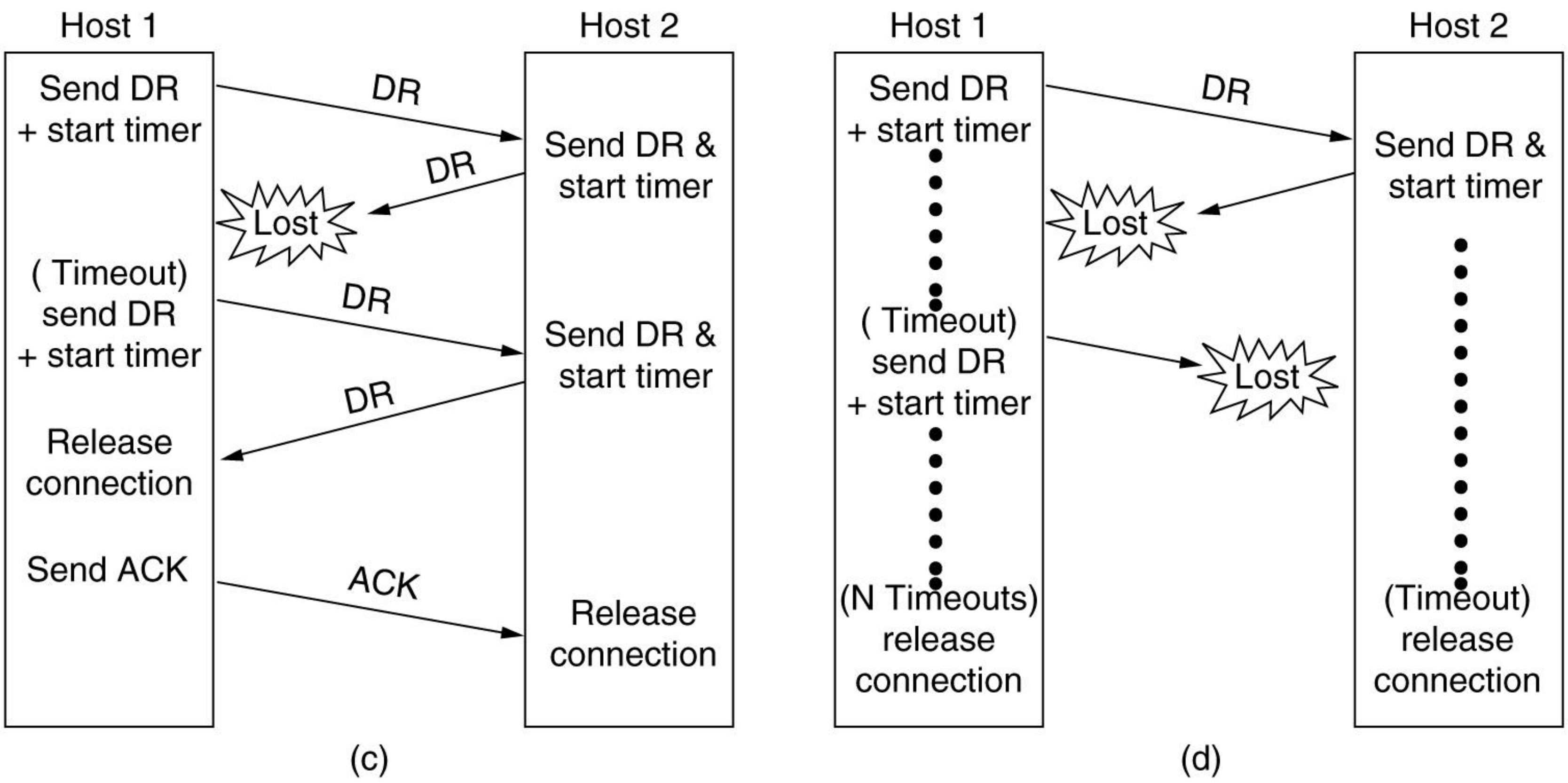

The two-army problem.

# Connection Release (3)



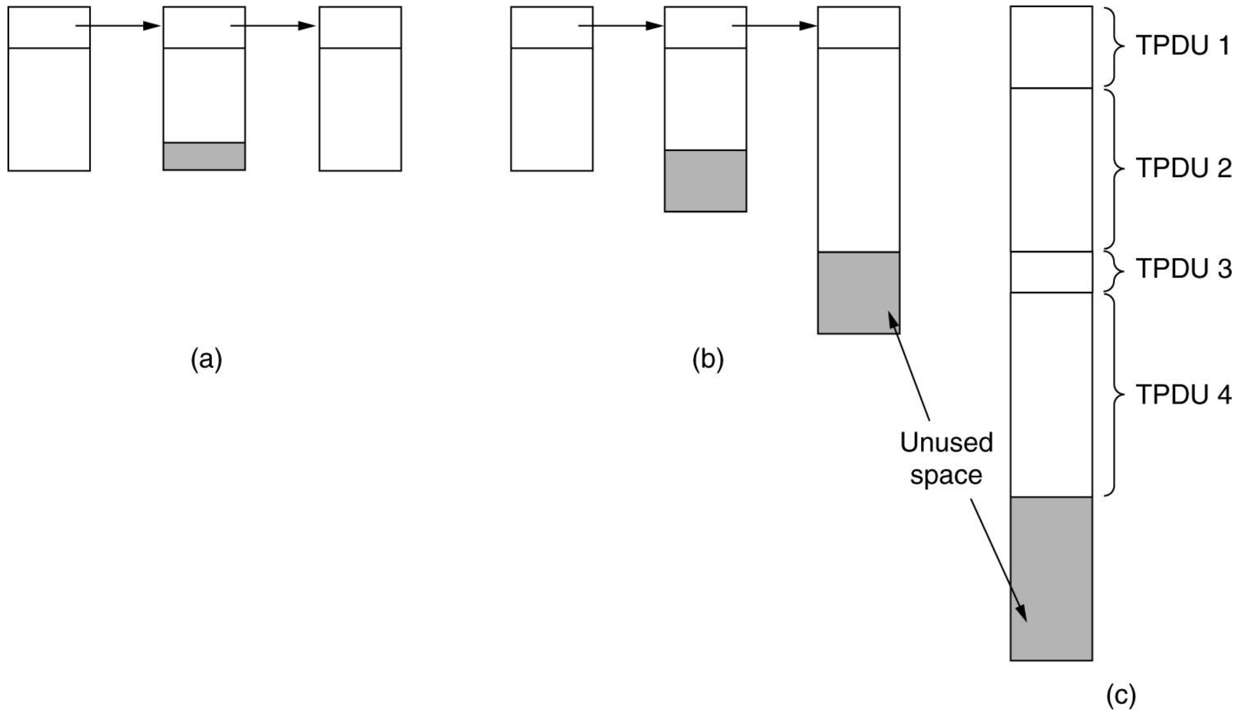

Four protocol scenarios for releasing a connection. (a) Normal case of a three-way handshake. (b) final ACK lost.

# Connection Release (4)




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

# Flow Control and Buffering



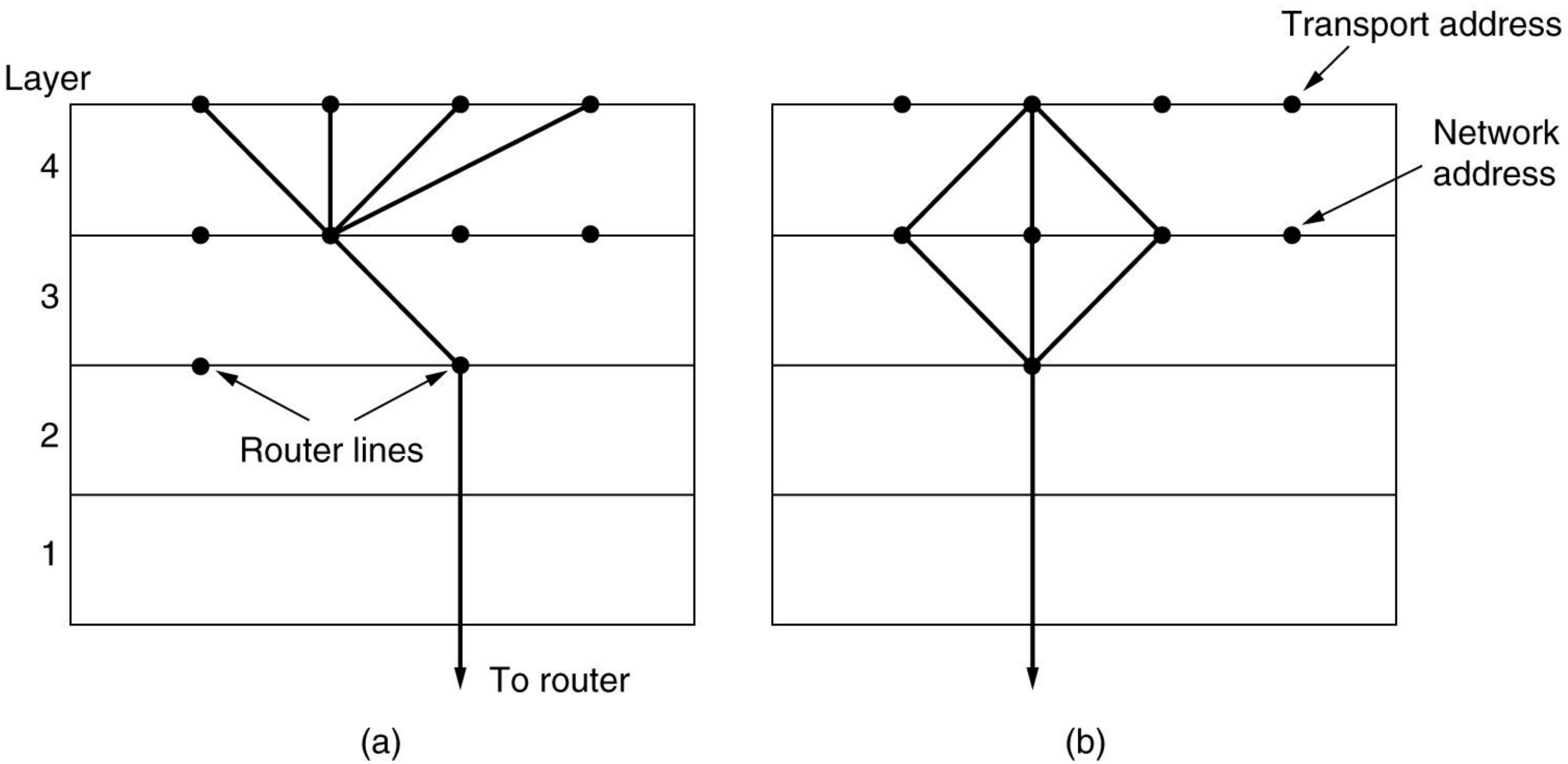

(a) Chained fixed-size buffers. (b) Chained variable-sized buffers.
(c) One large circular buffer per connection.

# 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.

# Multiplexing




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

# Crash Recovery

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

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.

# 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<br>P2: A1/Estab<br>$\overline{P2}$: A2/Idle | | ~/Estab | | | | |
| | CONNECT | P1: ~/Idle<br>$\overline{P1}$: A3/Wait | | | | | | |
| | DISCONNECT | | | | P4: A5/Idle<br>$\overline{P4}$: A6/Disc | | | |
| | SEND | | | | P5: A7/Estab<br>$\overline{P5}$: A8/Send | | | |
| | RECEIVE | | | | A9/Receiving | | | |
| Incoming packets | Call_req | P3: A1/Estab<br>$\overline{P3}$: 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 | | | | |

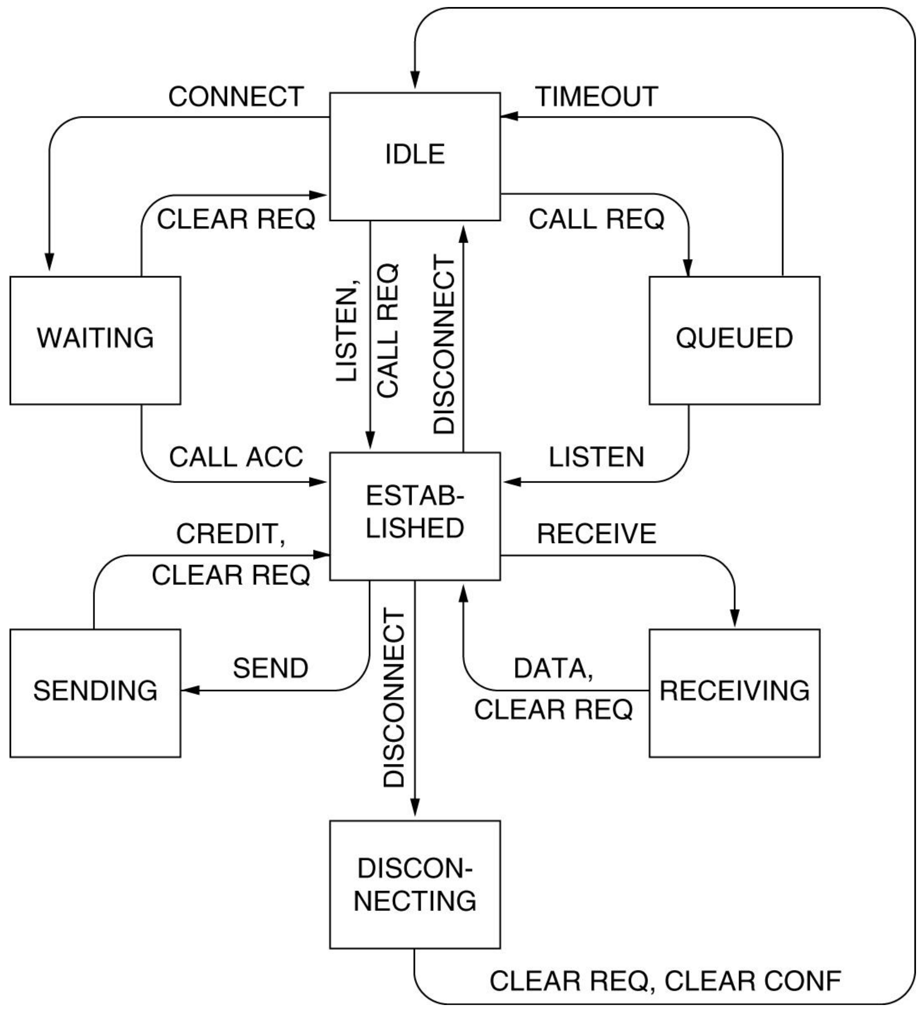
**Predicates**

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)

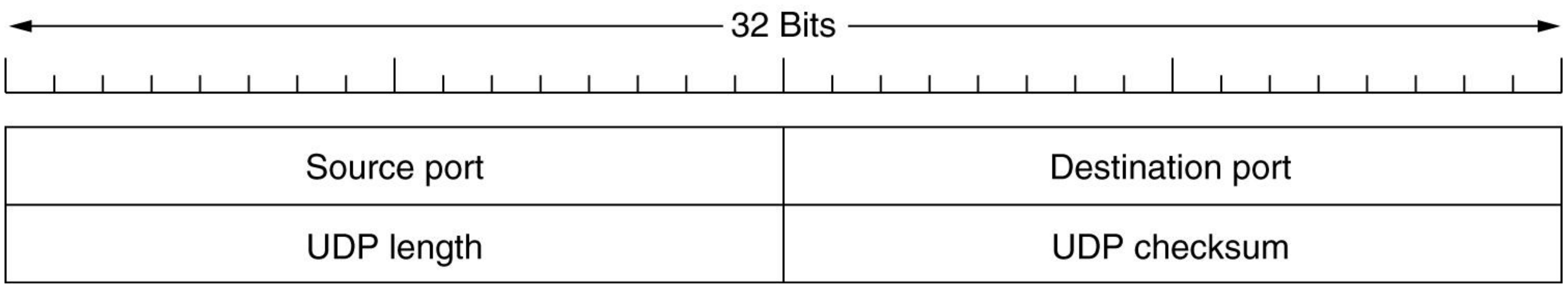



The example protocol in graphical form. Transitions that leave the connection state unchanged have been omitted for simplicity.

# The Internet Transport Protocols: UDP

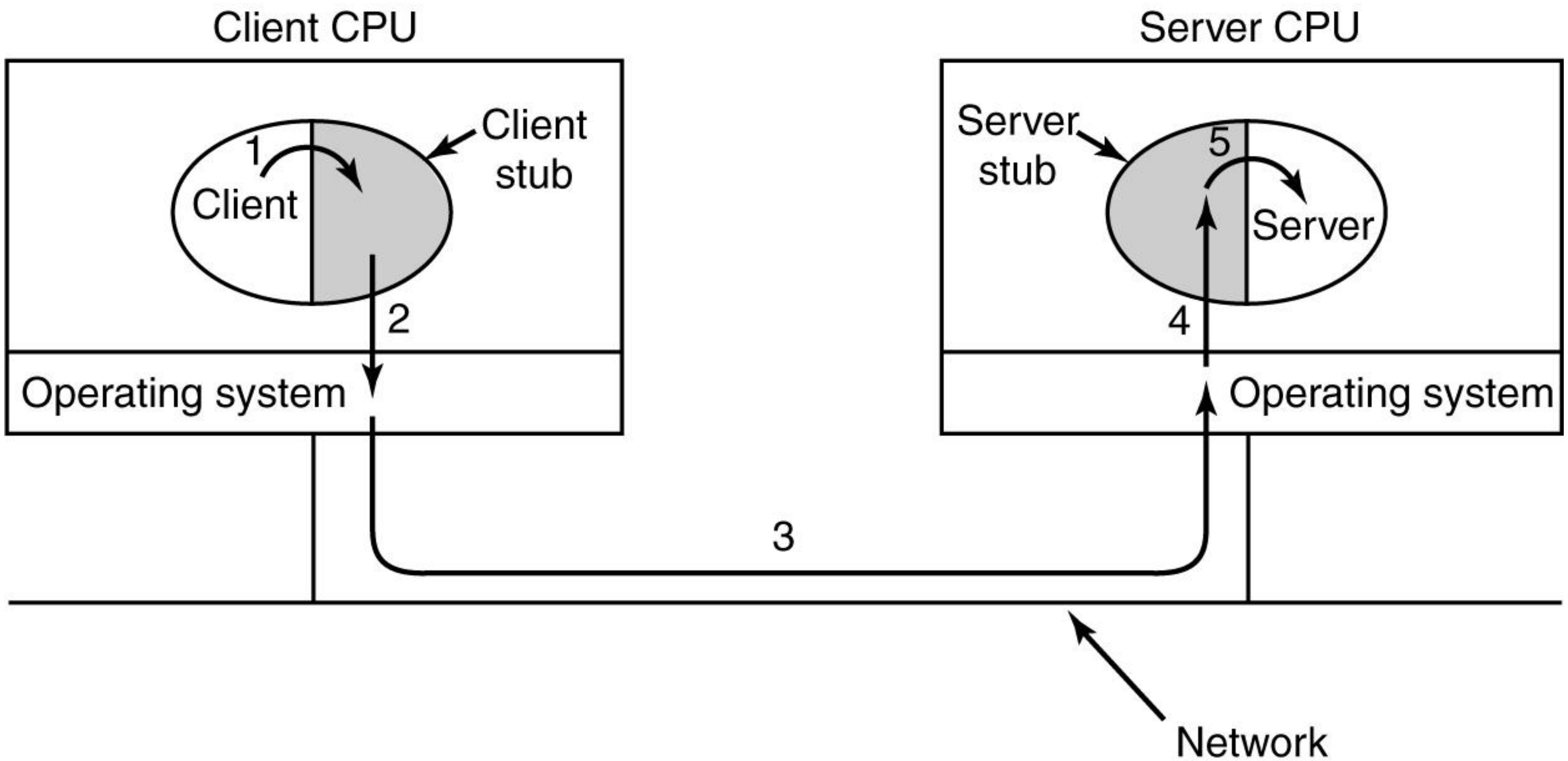
- Introduction to UDP
- Remote Procedure Call
- The Real-Time Transport Protocol

# Introduction to UDP



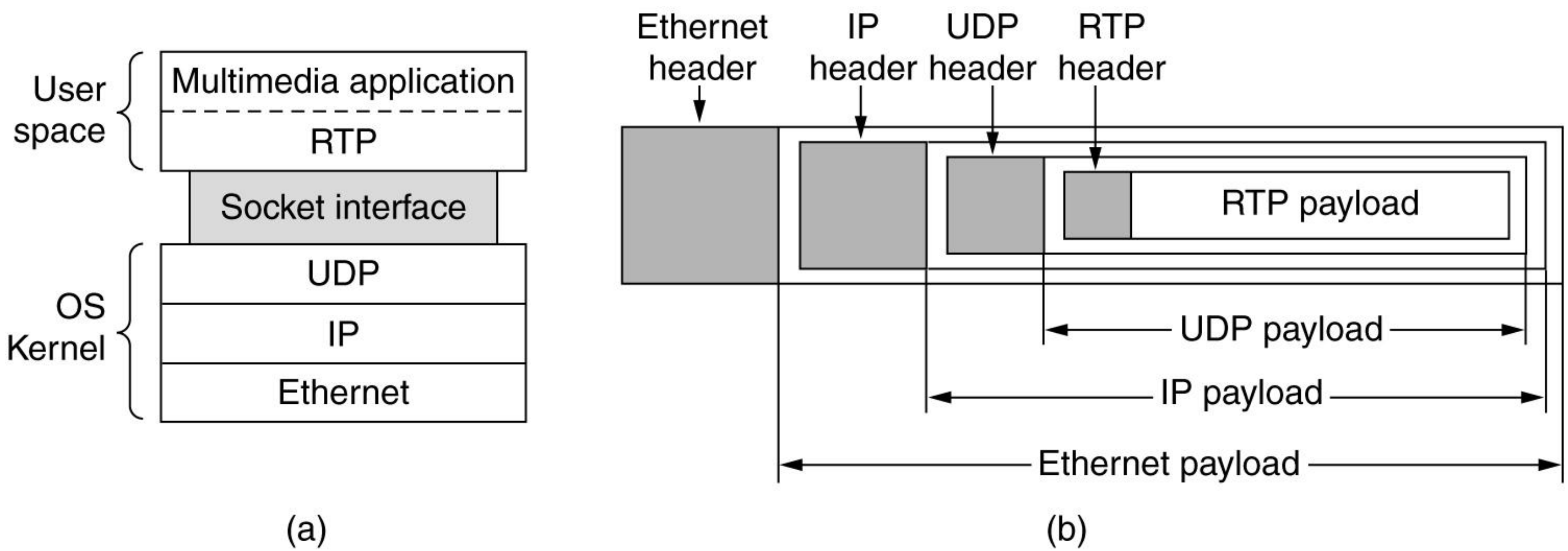

The UDP header.

# Remote Procedure Call



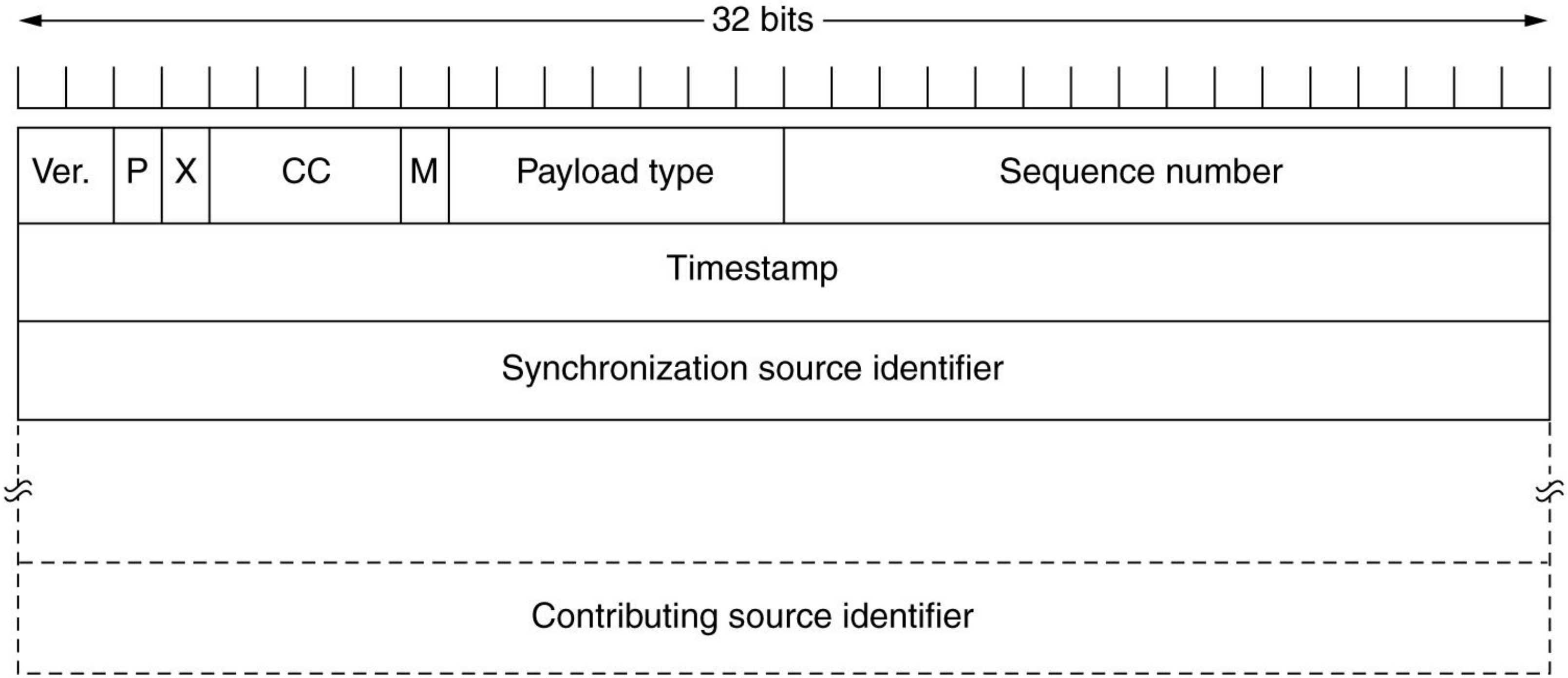

Steps in making a remote procedure call. The stubs are shaded.

# 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

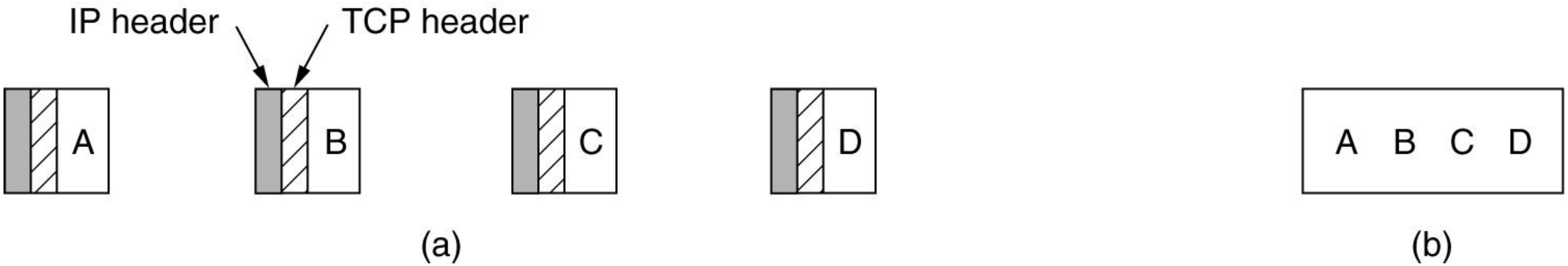
- Introduction to TCP
- The TCP Service Model
- The TCP Protocol
- The TCP Segment Header
- TCP Connection Establishment
- TCP Connection Release
- TCP Connection Management Modeling
- TCP Transmission Policy
- TCP Congestion Control
- TCP Timer Management
- Wireless TCP and UDP
- Transactional TCP

# 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.

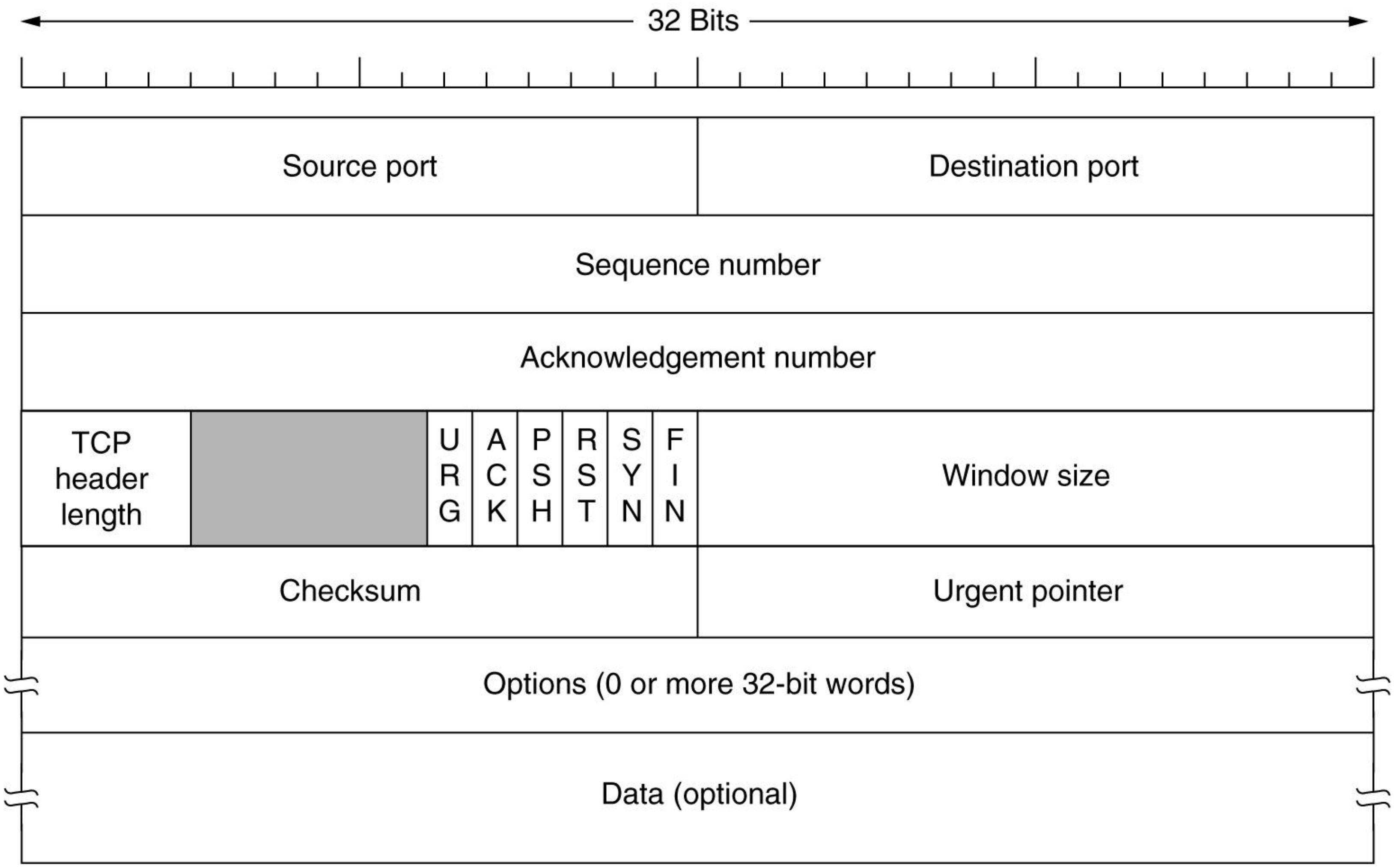
# The TCP Service Model (2)




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

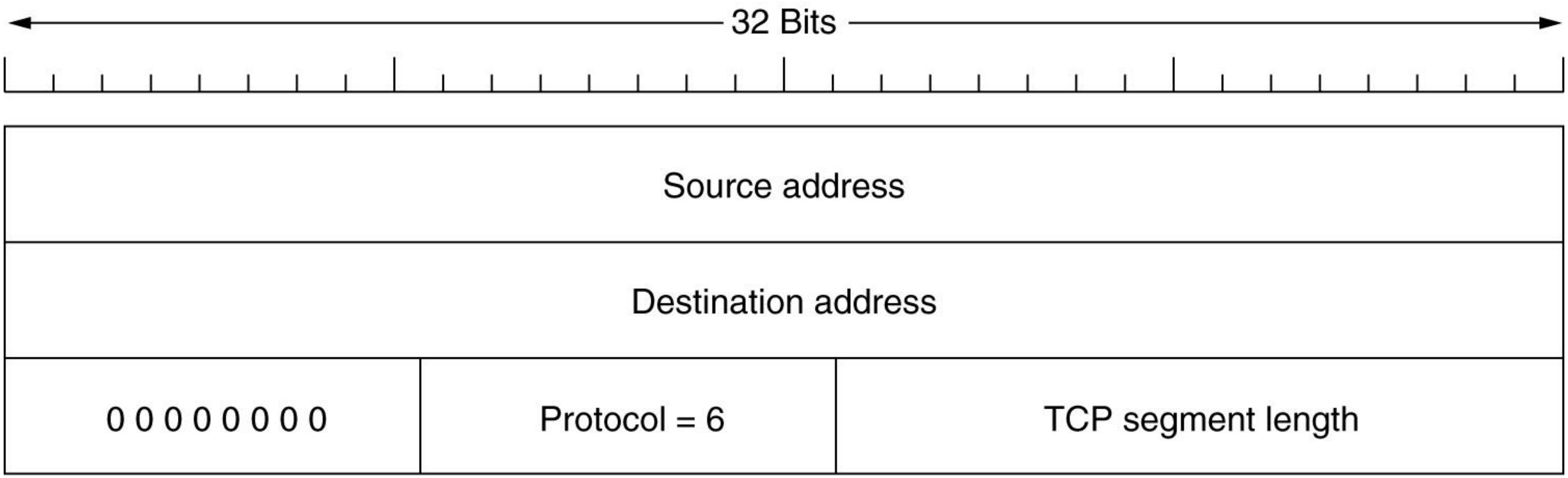
(b) The 2048 bytes of data delivered to the application in a single READ CALL.

# The TCP Segment Header



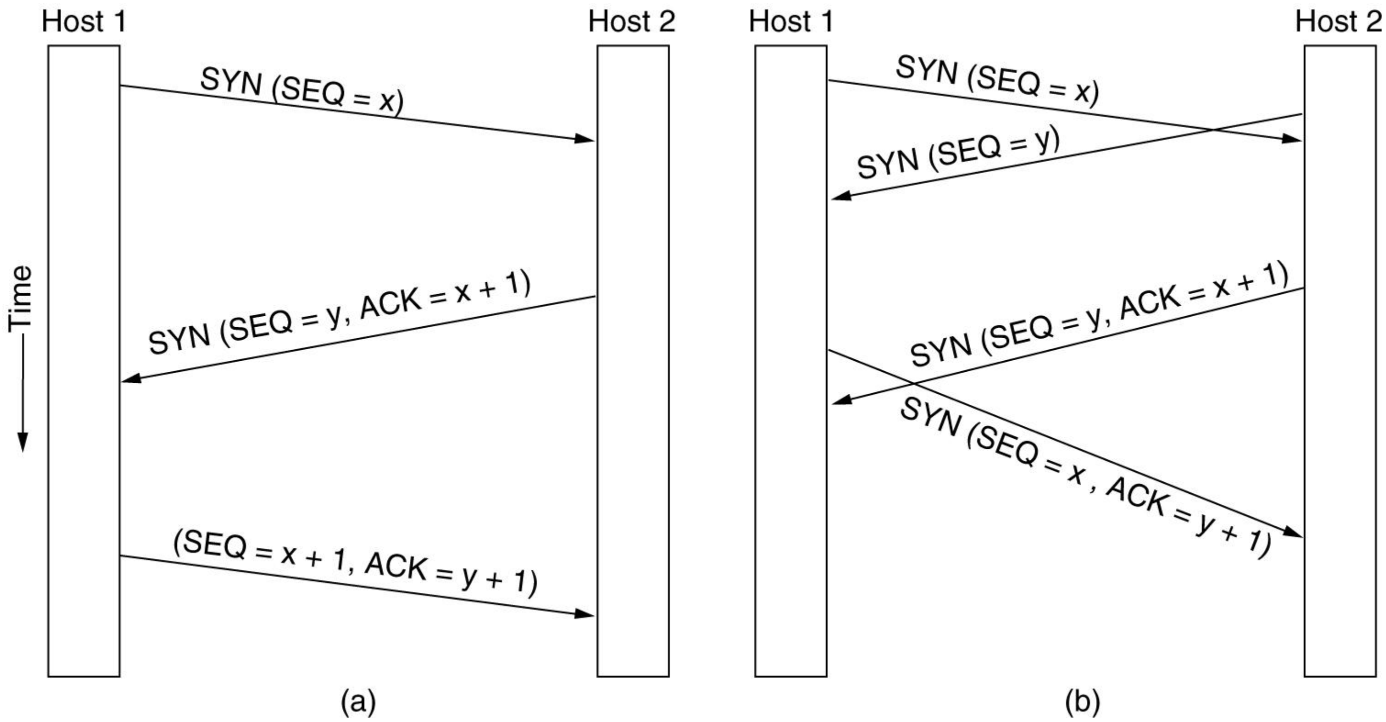

TCP Header.

# The TCP Segment Header (2)




The pseudoheader included in the TCP checksum.

# TCP Connection Establishment




(a) TCP connection establishment in the normal case.

(b) Call collision.

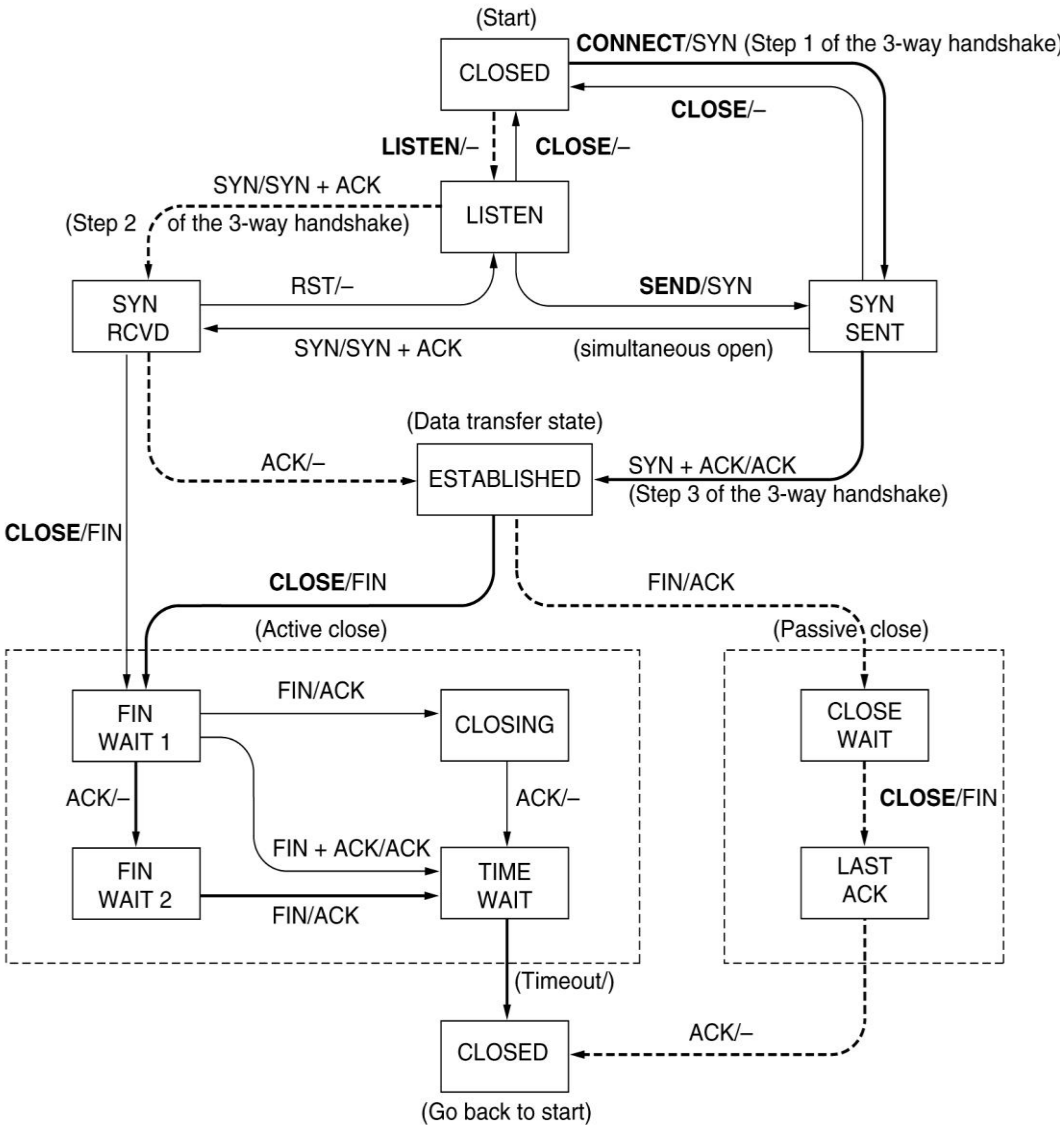
# TCP Connection Management Modeling

| 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 |

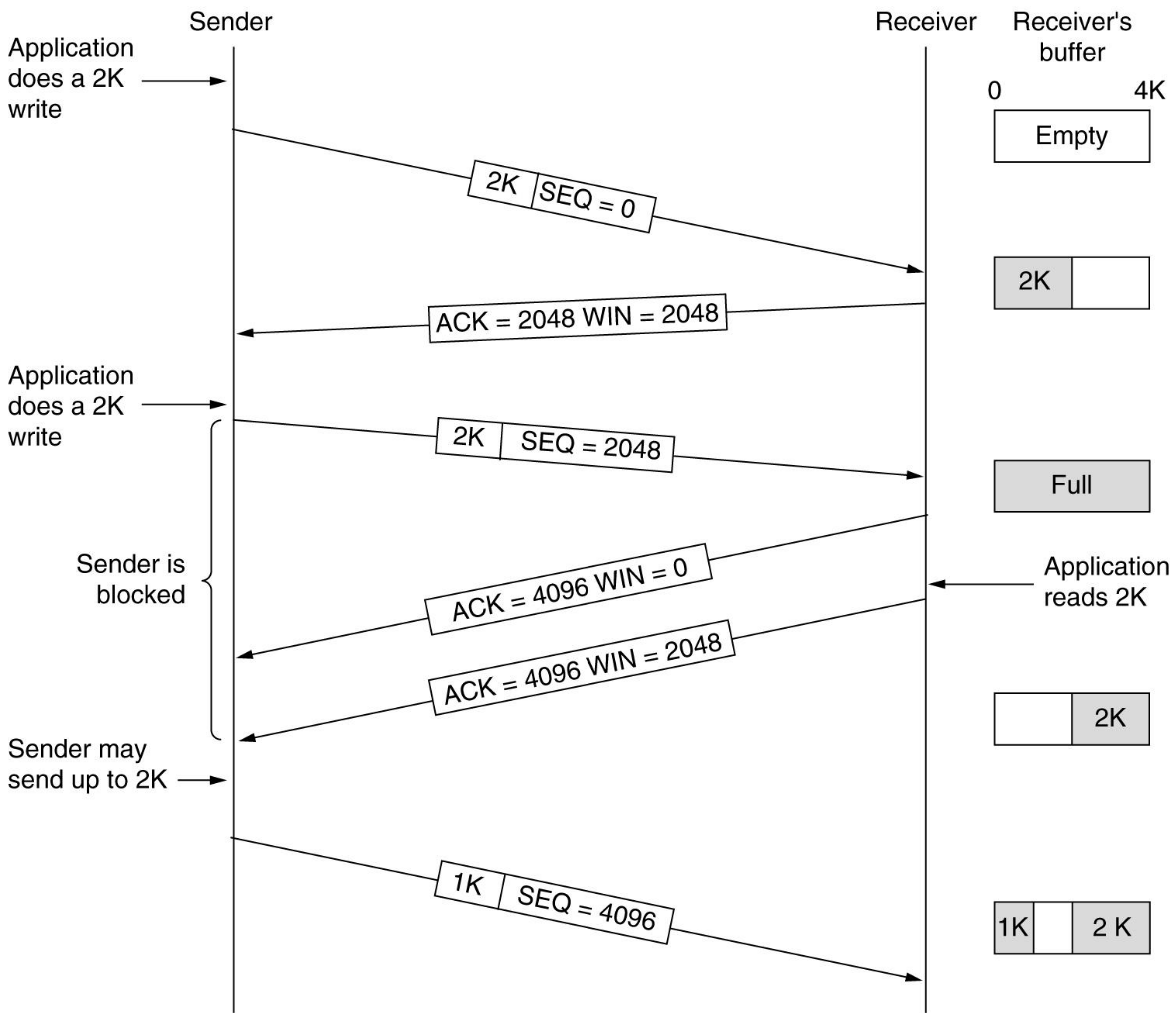
The states used in the TCP connection management finite state machine.

# 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.

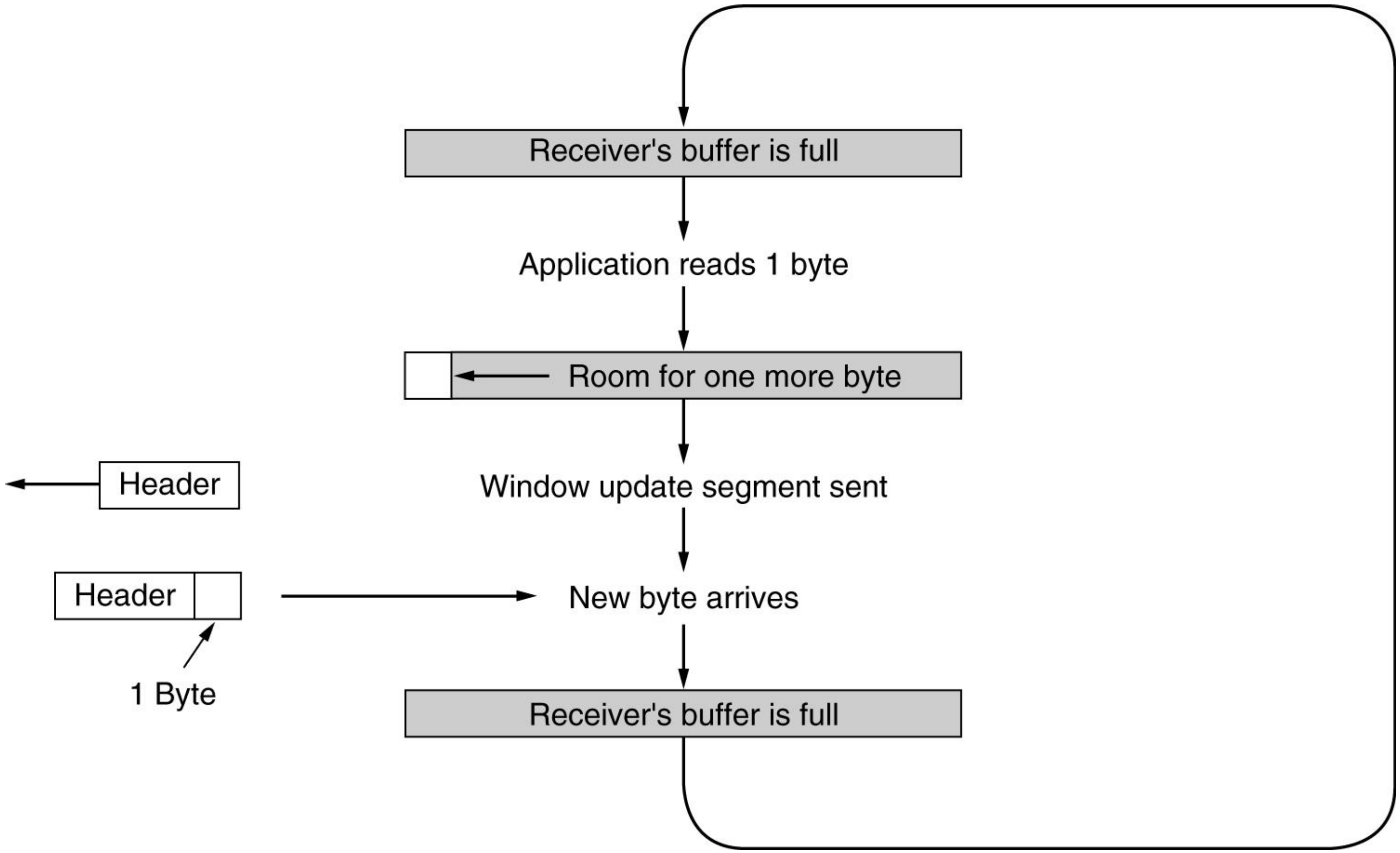


# TCP Transmission Policy



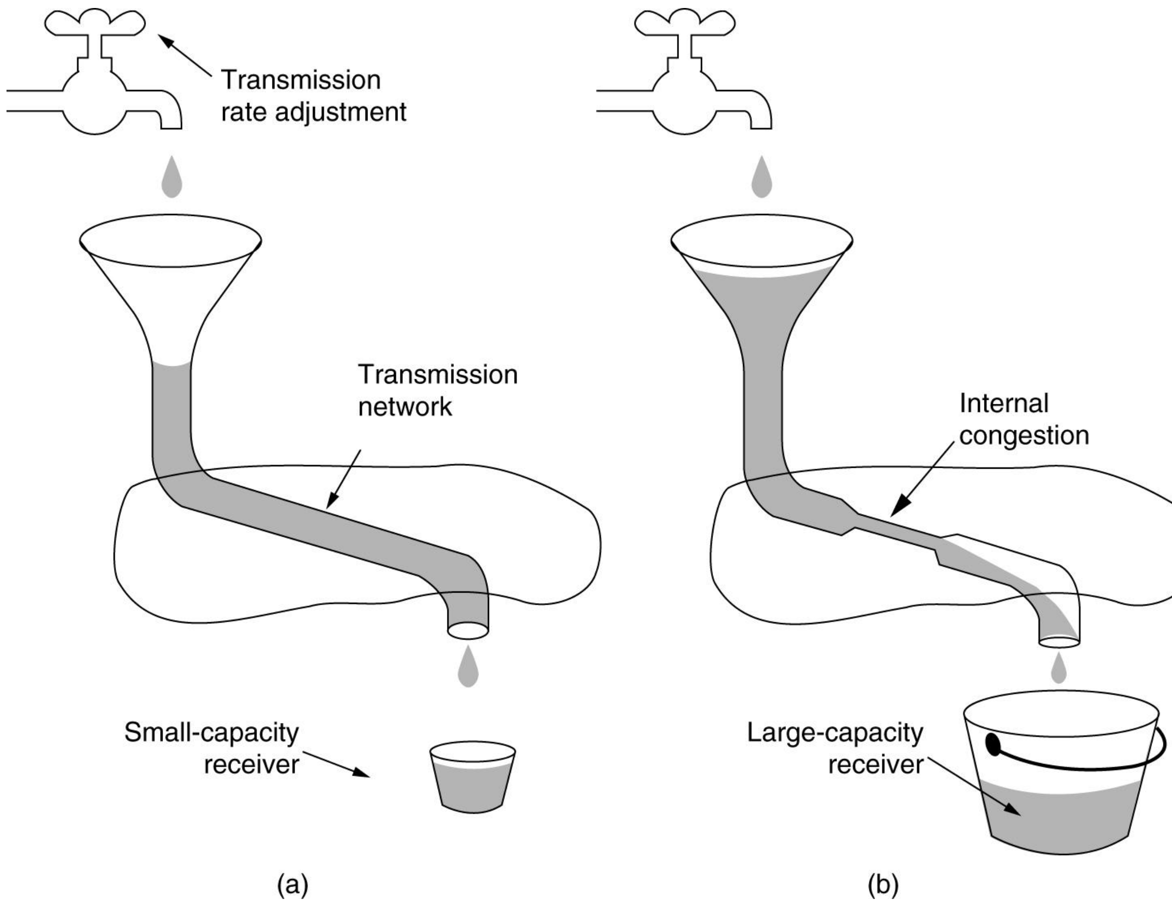

Window management in TCP.

# TCP Transmission Policy (2)




Silly window syndrome.

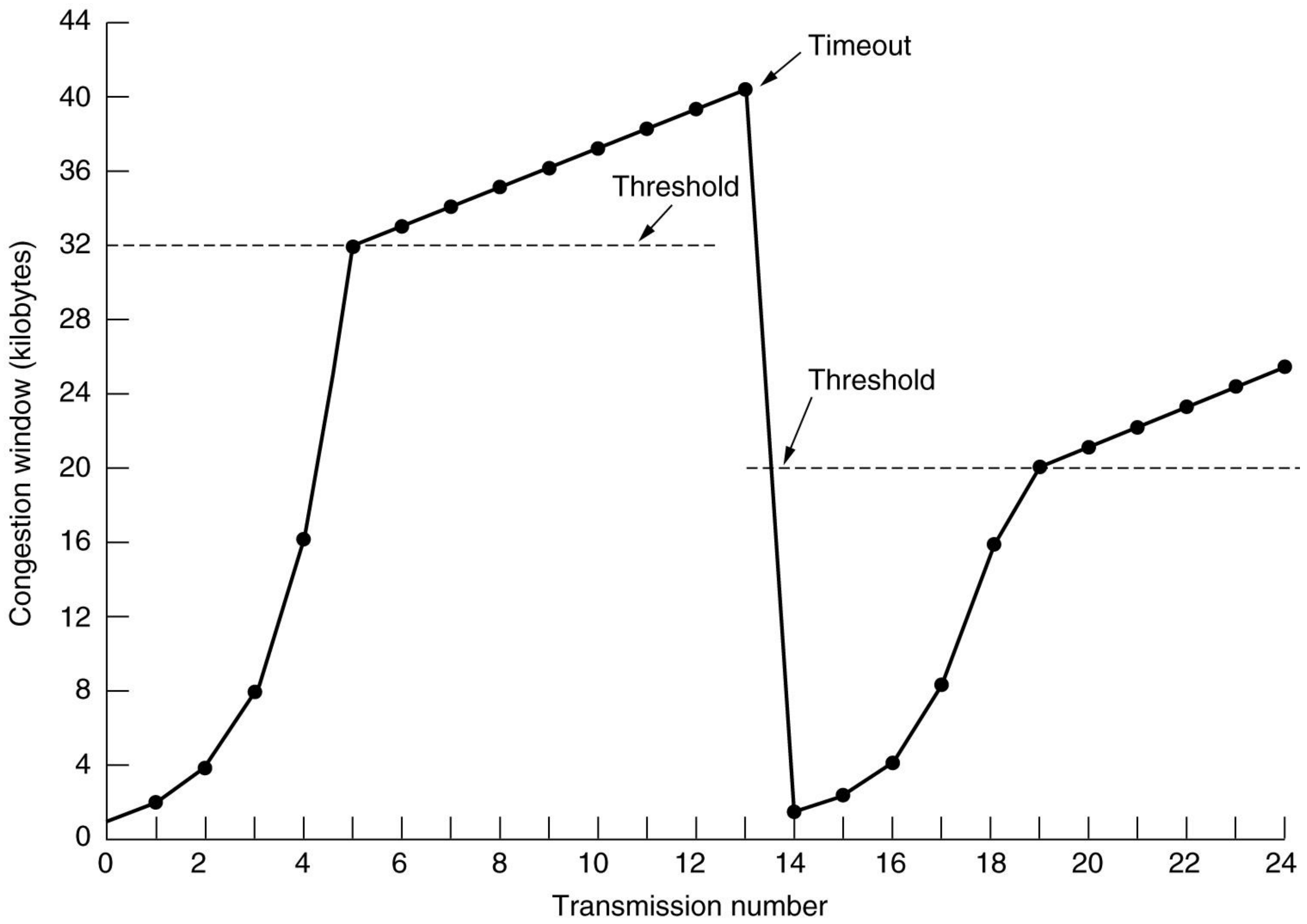
# TCP Congestion Control




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

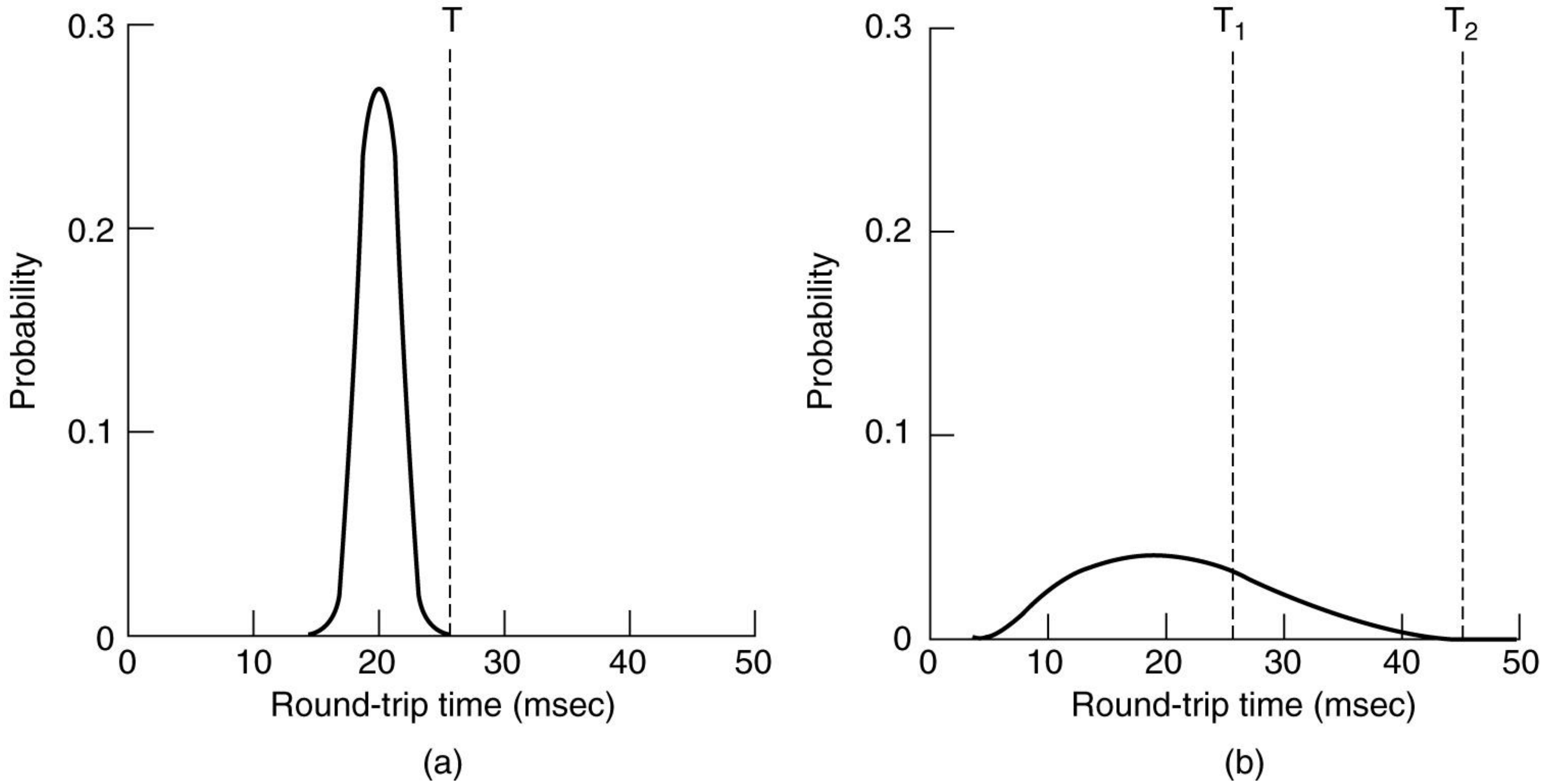
(b) A slow network feeding a high-capacity receiver.

# TCP Congestion Control (2)




An example of the Internet congestion algorithm.

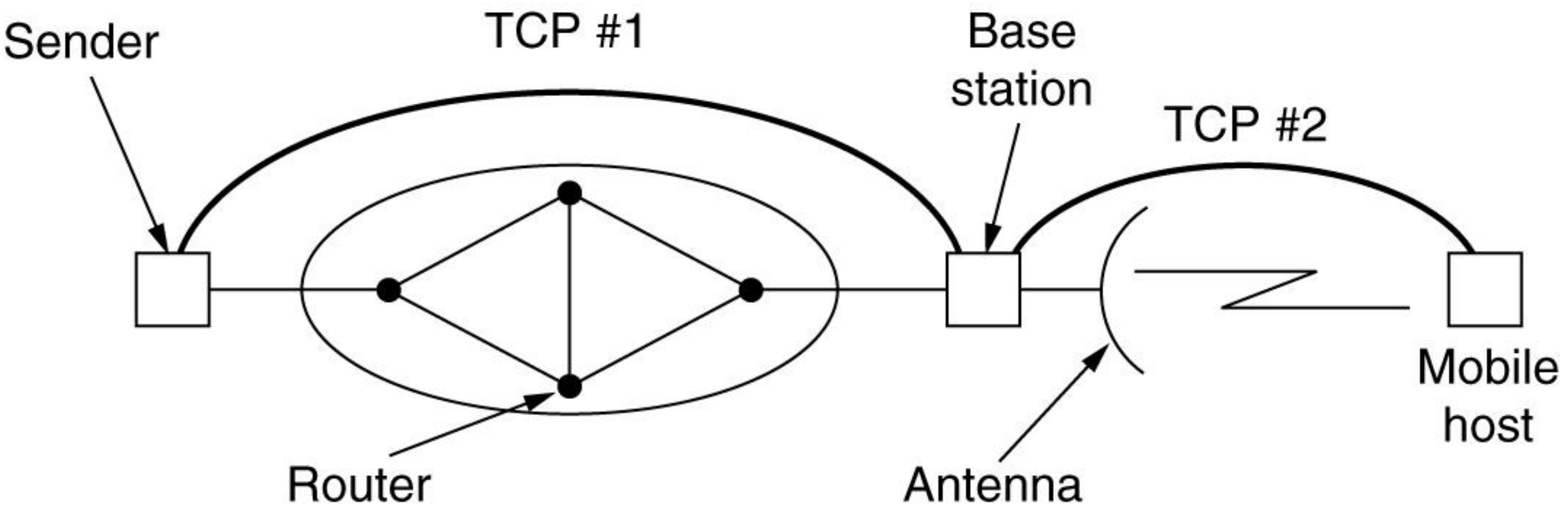
# TCP Timer Management




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

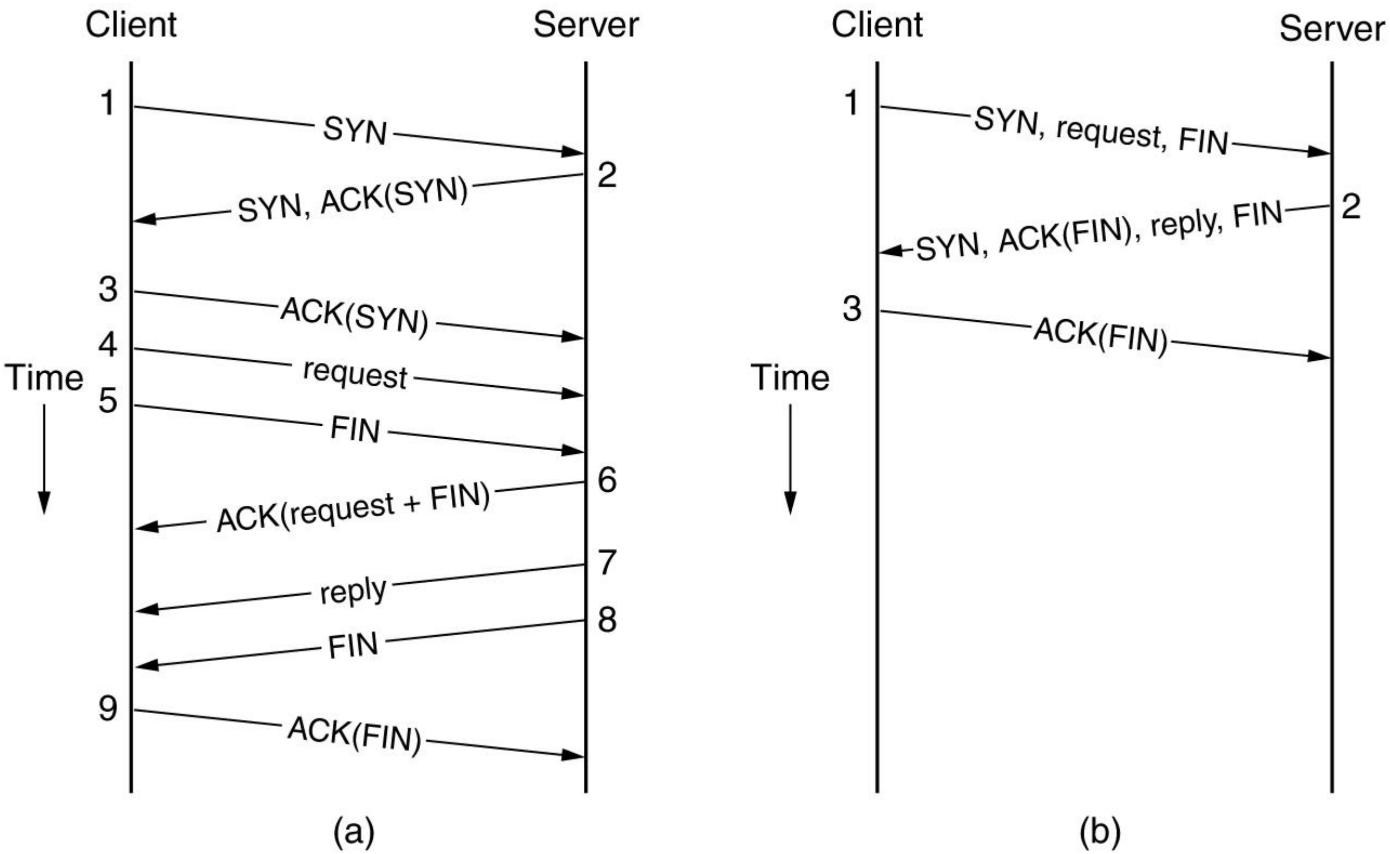
(b) Probability density of ACK arrival times for TCP.

# Wireless TCP and UDP




Splitting a TCP connection into two connections.

# Transitional TCP

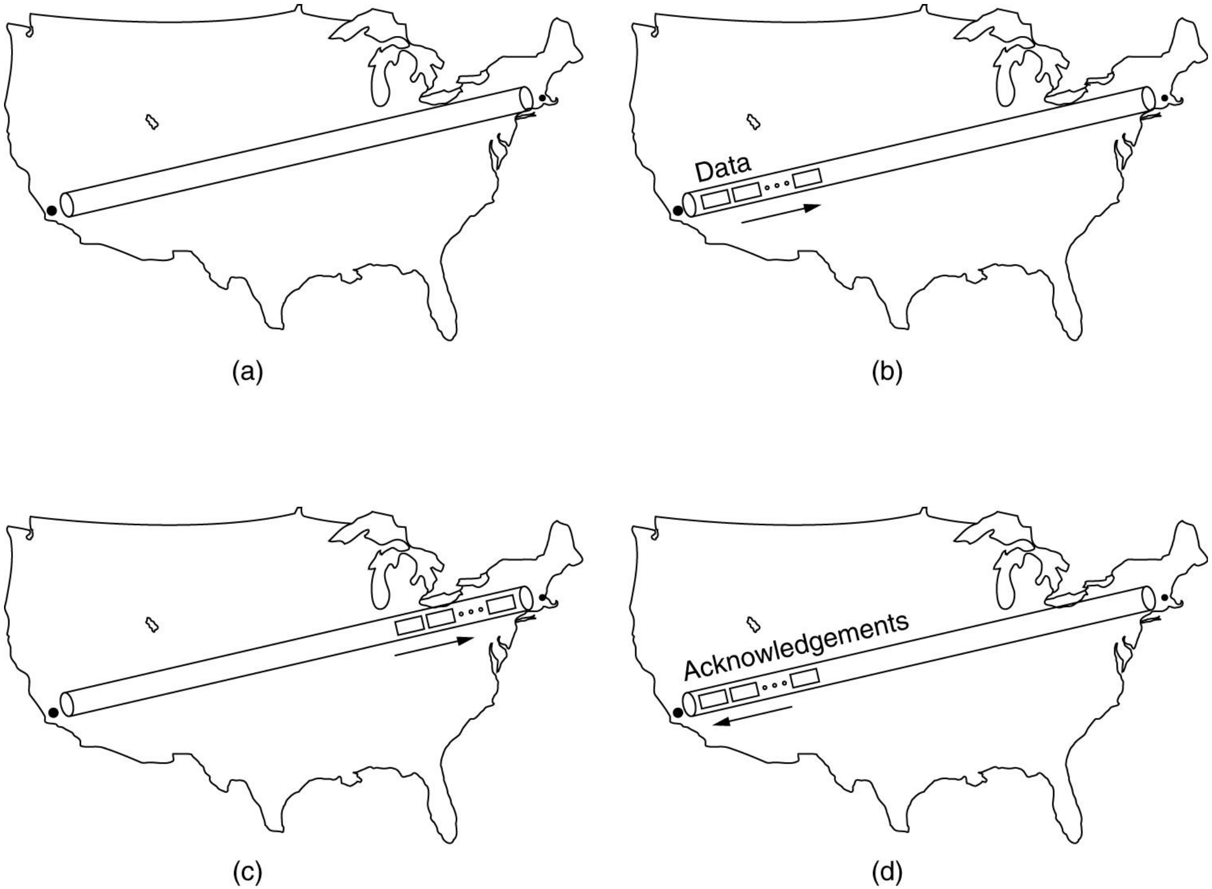



(a) RPC using normal TPC.
(b) RPC using T/TCP.

# 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.

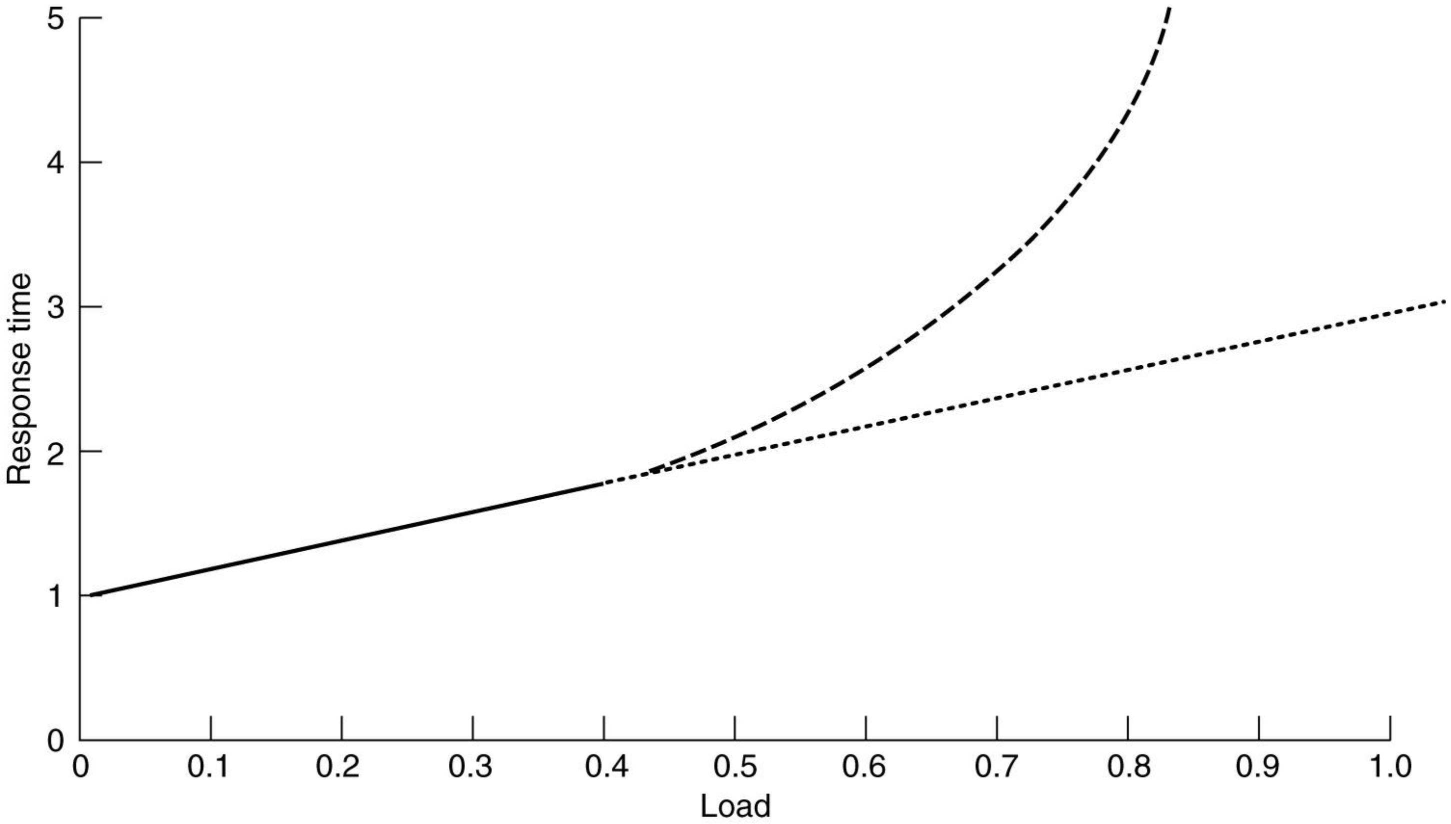
1. Measure relevant network parameters, performance.
2. Try to understand what is going on.
3. Change one parameter.

# System Design for Better Performance

Rules:

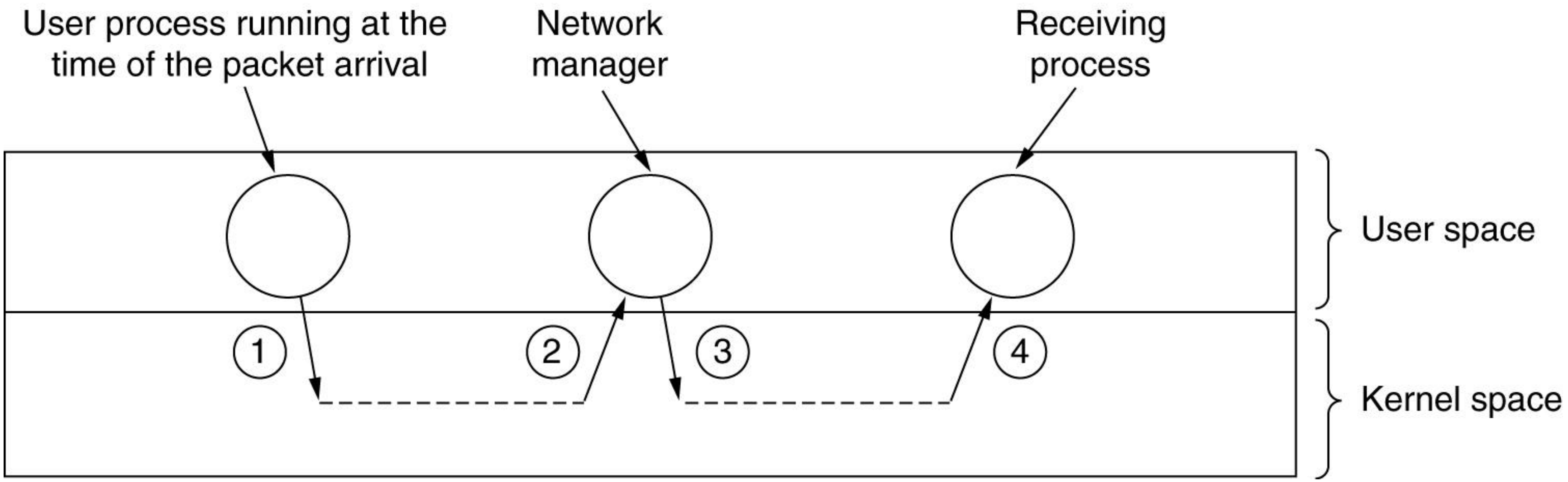
1. CPU speed is more important than network speed.
2. Reduce packet count to reduce software overhead.
3. Minimize context switches.
4. Minimize copying.
5. You can buy more bandwidth but not lower delay.
6. Avoiding congestion is better than recovering from it.
7. Avoid timeouts.

# System Design for Better Performance (2)



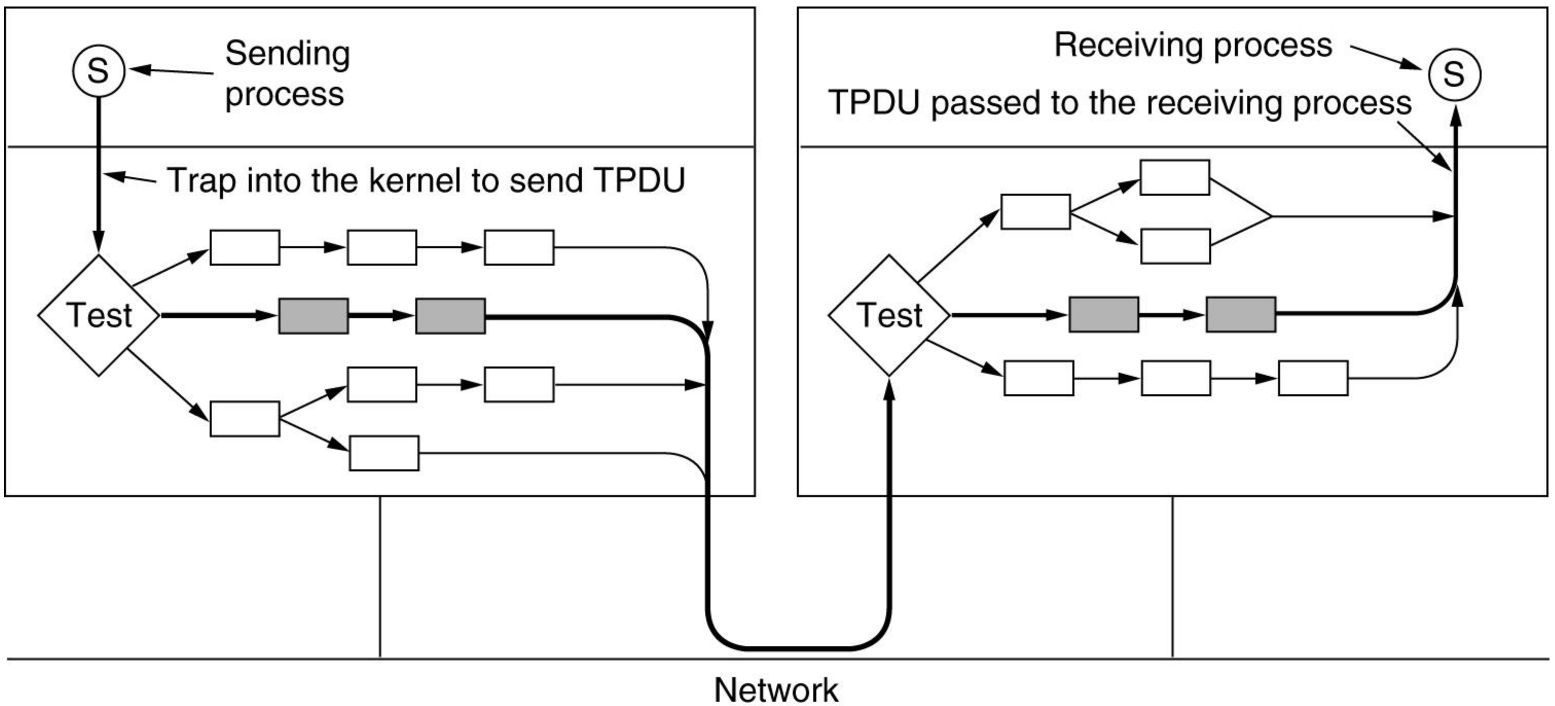

Response as a function of load.

# System Design for Better Performance (3)



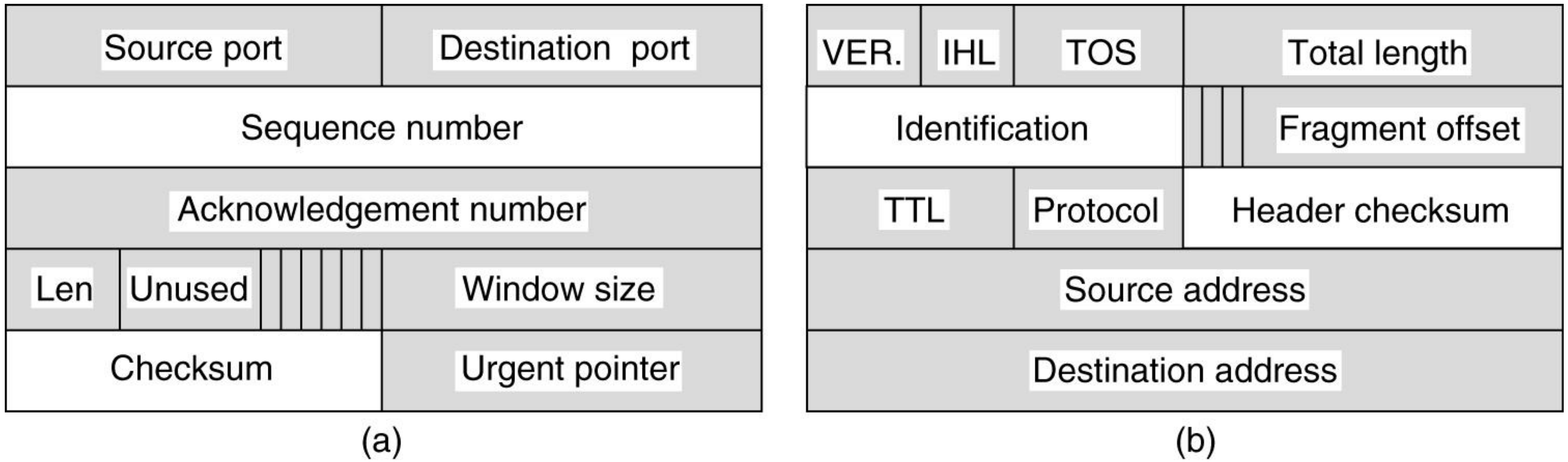

Four context switches to handle one packet with a user-space network manager.

# Fast TPDU Processing



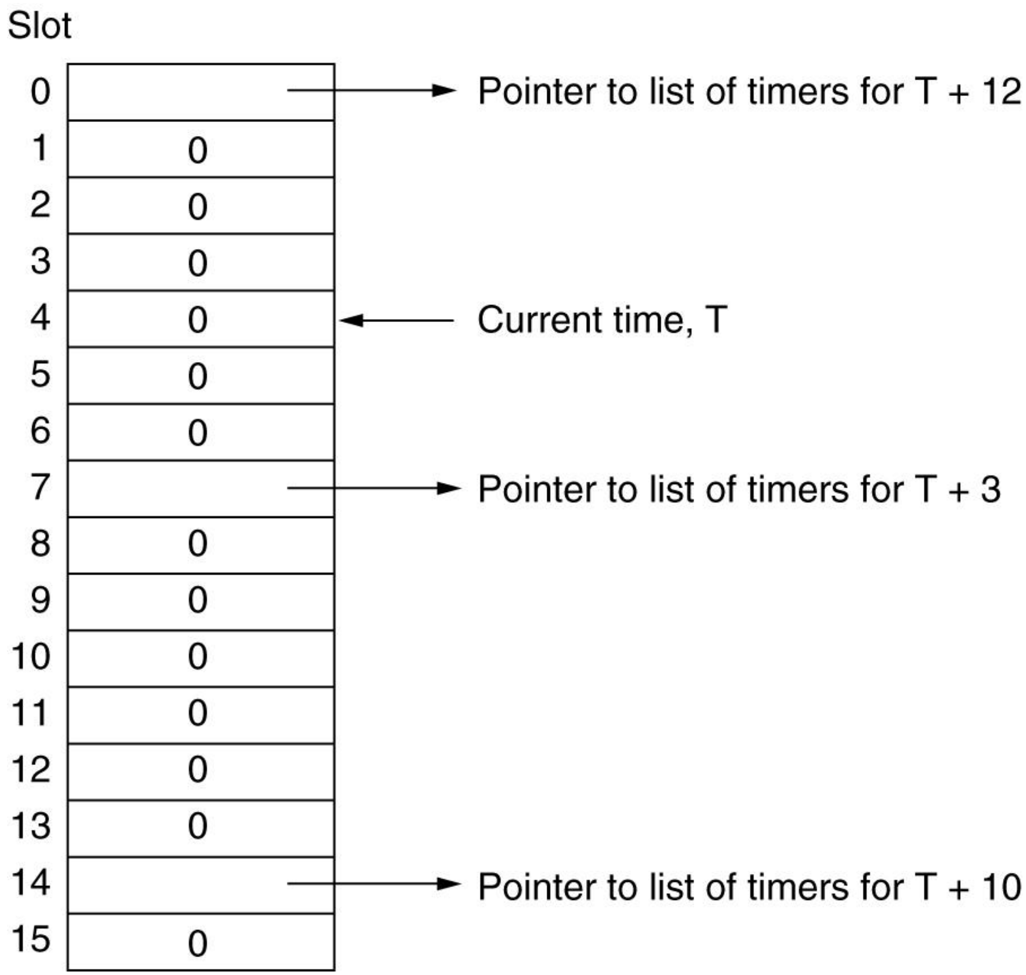

The fast path from sender to receiver is shown with a heavy line.
The processing steps on this path are shaded.

# Fast TPDU Processing (2)



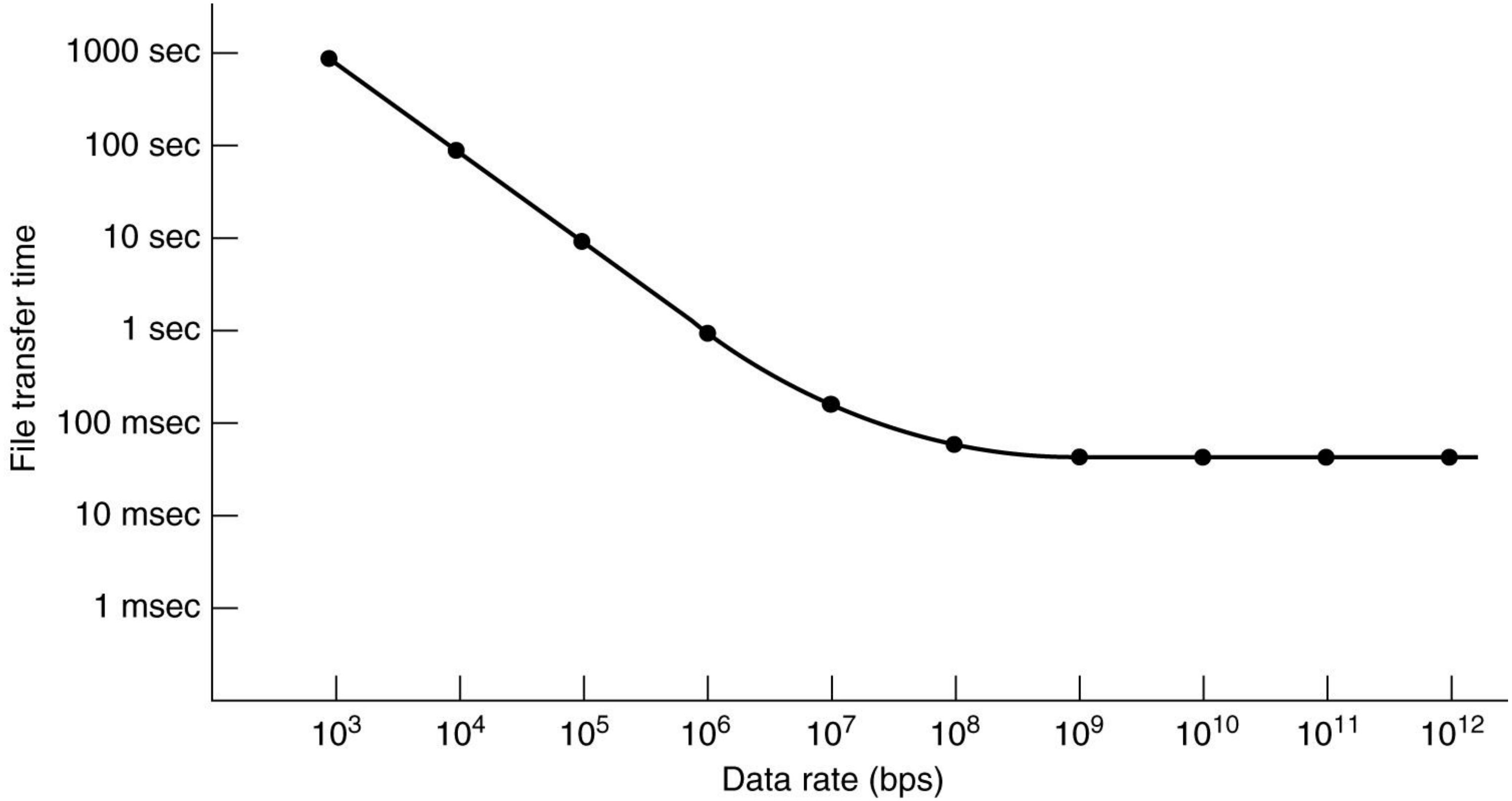

(a) TCP header. (b) IP header. In both cases, the shaded fields are taken from the prototype without change.

# 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.