6

THE TRANSPORT LAYER

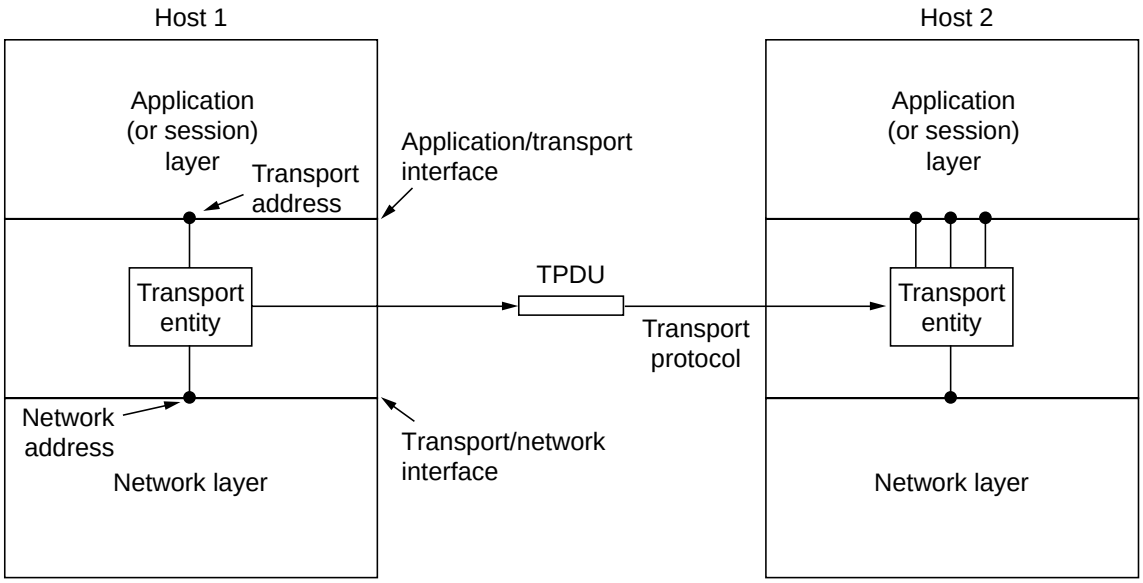

Fig. 6-1. The network, transport, and application layers.

Connection establishment delay
Connection establishment failure probability
Throughput
Transit delay
Residual error ratio
Protection
Priority
Resilience

Fig. 6-2. Typical transport layer quality of service parameters.

Primitive	TPDU 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 TPDU arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

Fig. 6-3. The primitives for a simple transport service.

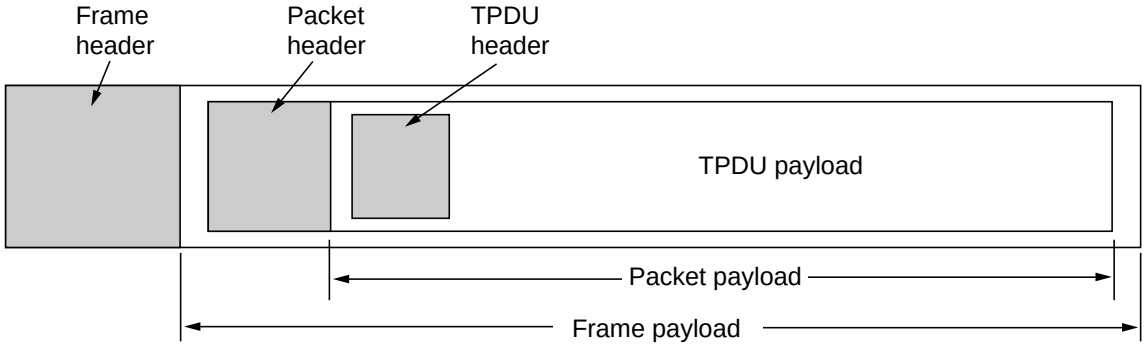

Fig. 6-4. Nesting of TPDUs, packets, and frames.

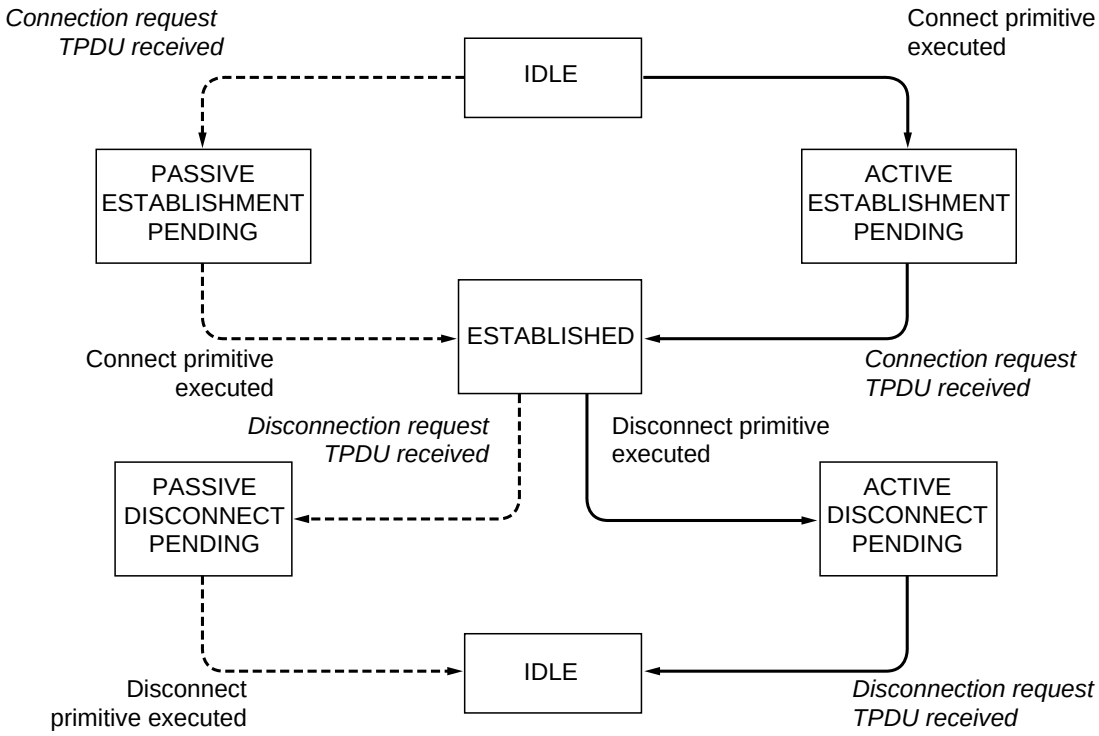

Fig. 6-5. A state diagram for a simple connection management scheme. Transitions labeled in italics are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.

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

Fig. 6-6. The socket primitives for TCP.

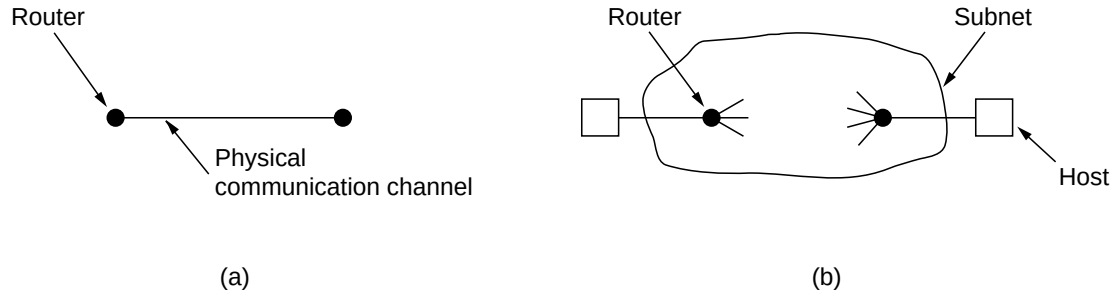

Fig. 6-7. (a) Environment of the data link layer. (b) Environment of the transport layer.

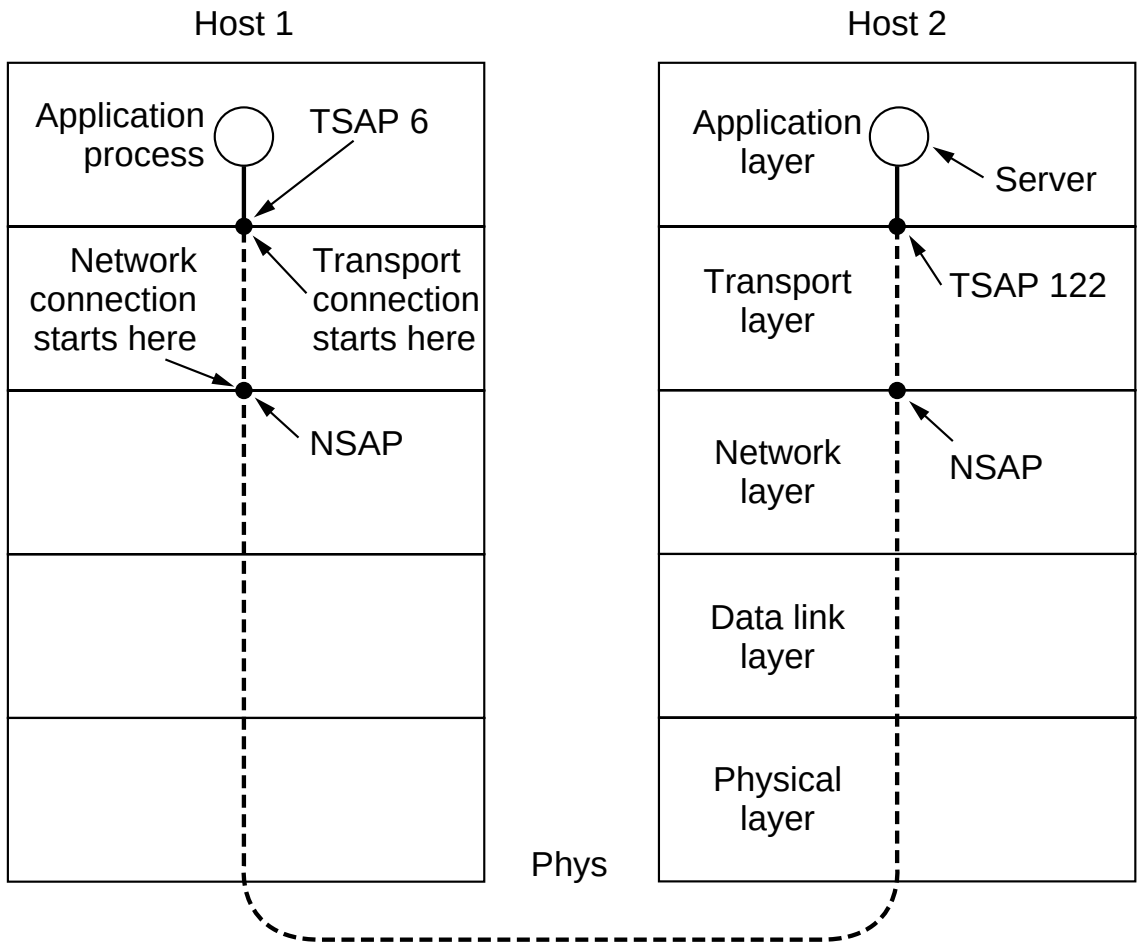

Fig. 6-8. TSAPs, NSAPs, and connections.

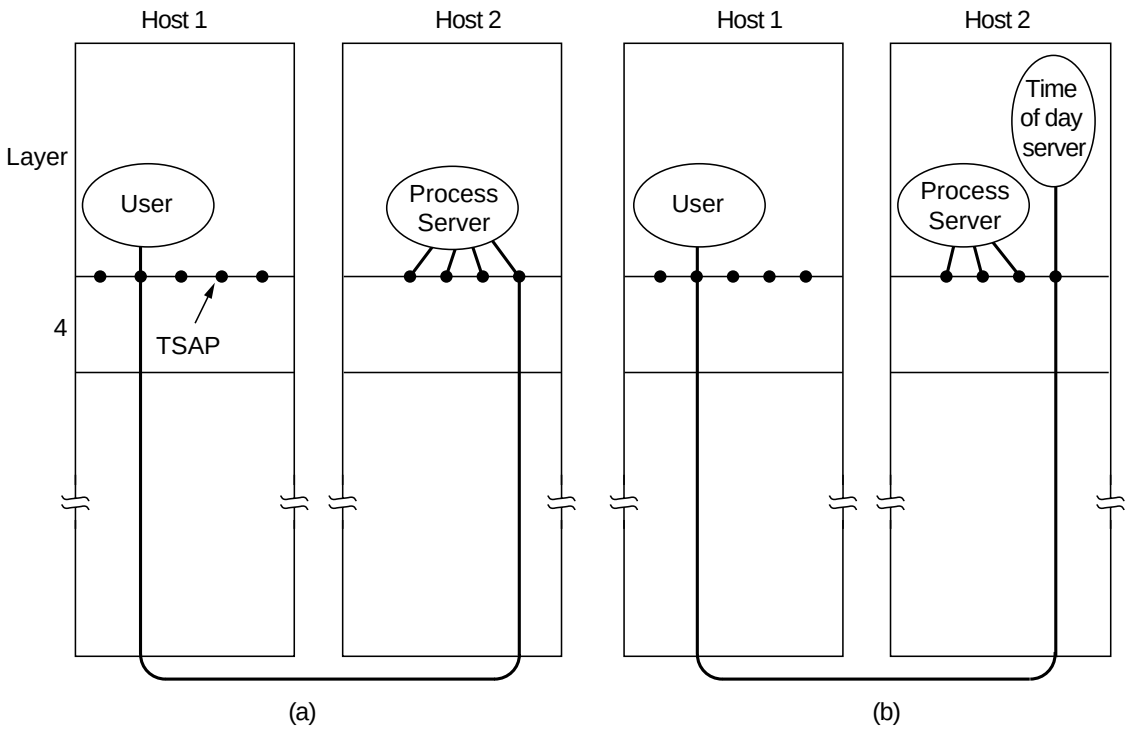

Fig. 6-9. How a user process in host 1 establishes a connection with a time-of-day server in host 2.

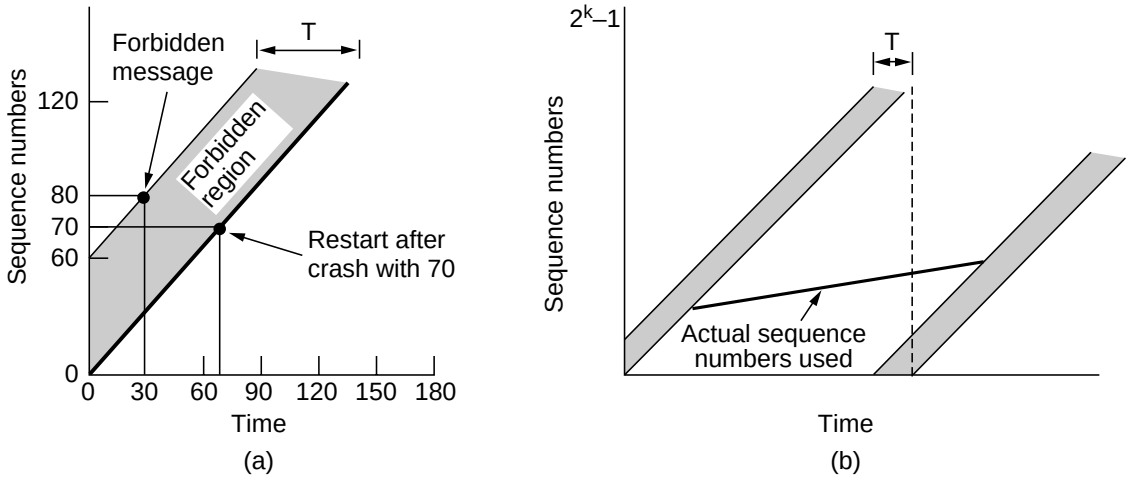

Fig. 6-10. (a) TPDUs may not enter the forbidden region. (b) The resynchronization problem.

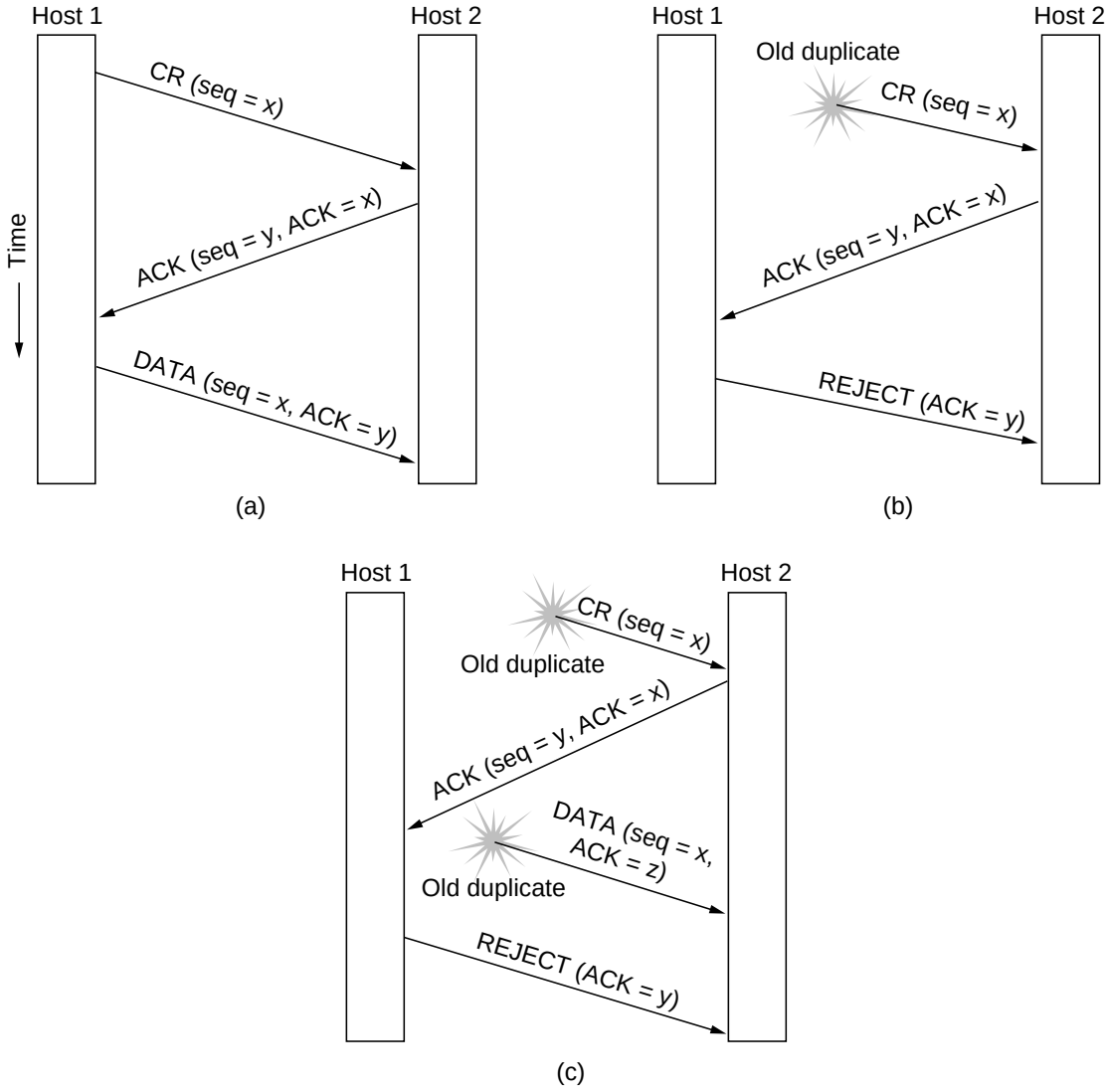

Fig. 6-11. Three protocol scenarios for establishing a connection using a three-way handshake. CR and ACC denote CONNECTION REQUEST and CONNECTION ACCEPTED, respectively. (a) Normal operation. (b) Old duplicate CONNECTION REQUEST appearing out of nowhere. (c) Duplicate CONNECTION REQUEST and duplicate ACK.

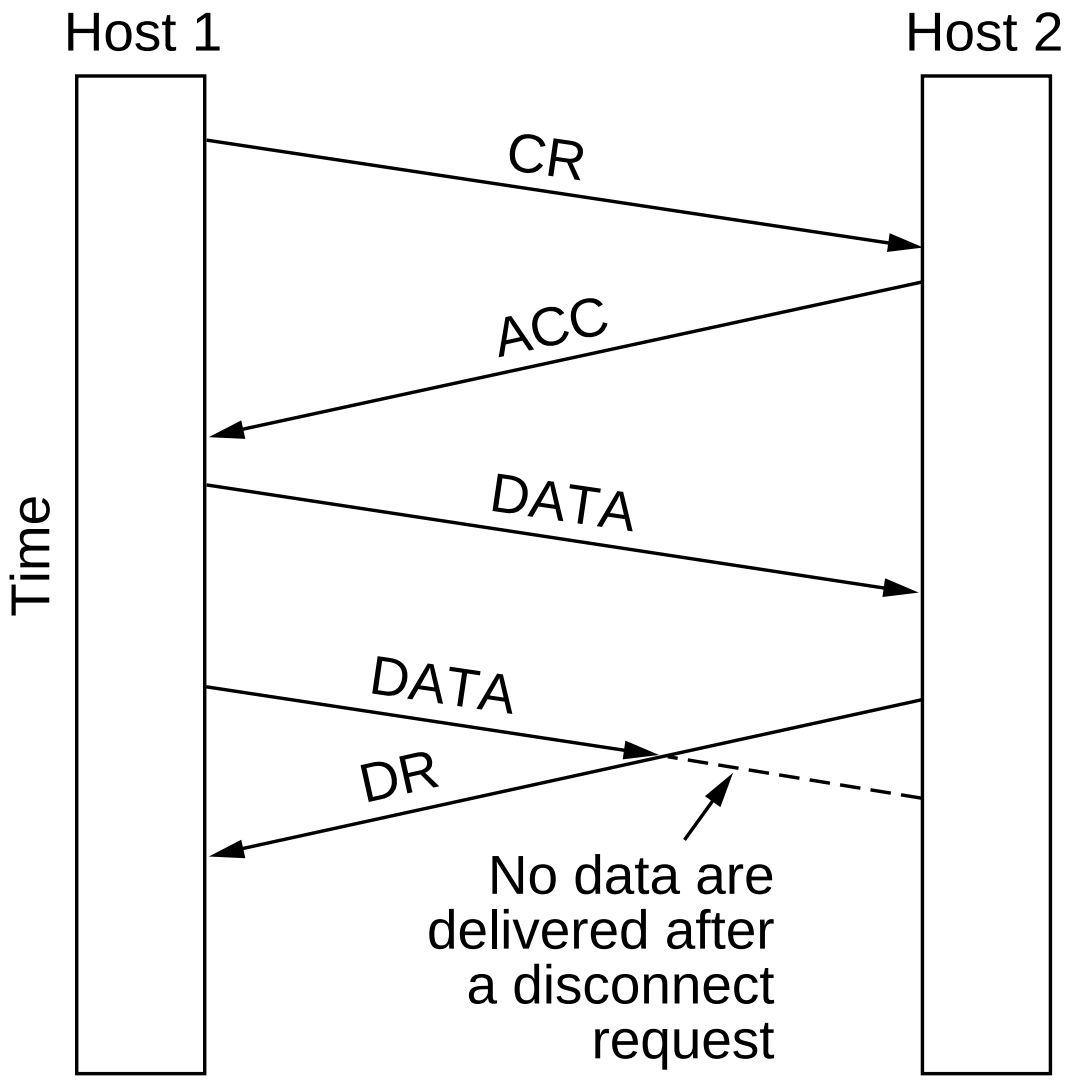

Fig. 6-12. Abrupt disconnection with loss of data.

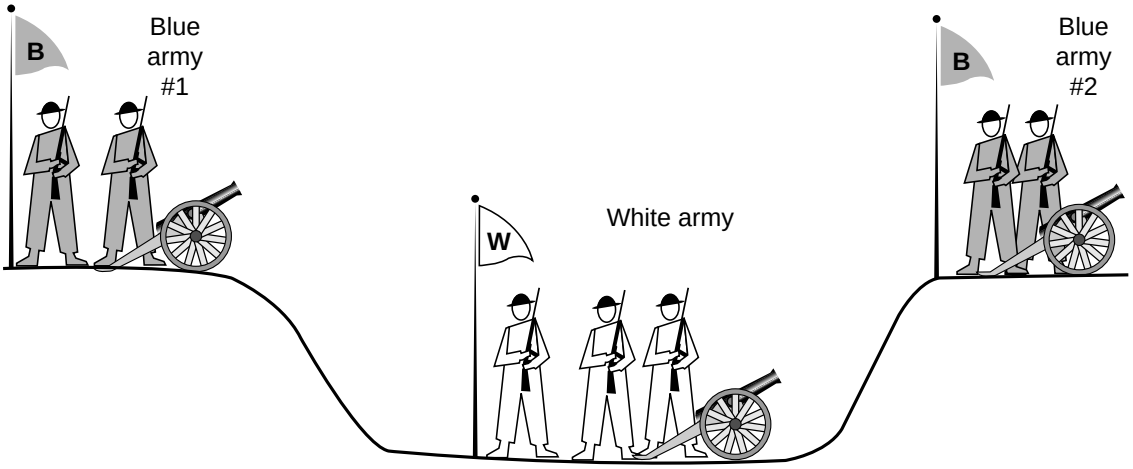

Fig. 6-13. The two-army problem.

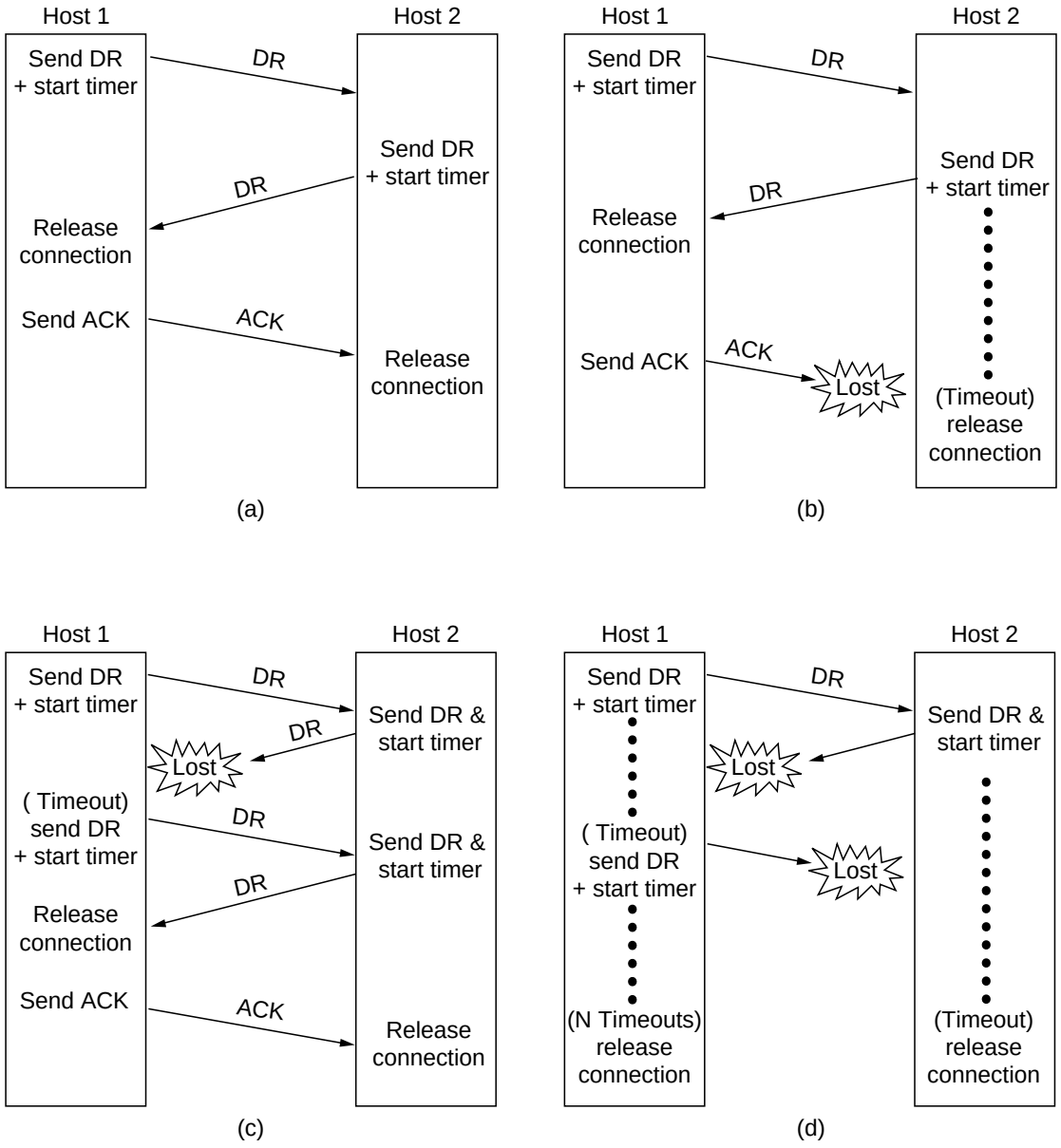

Fig. 6-14. Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final ACK lost. (c) Response lost. (d) Response lost and subsequent DRs lost.

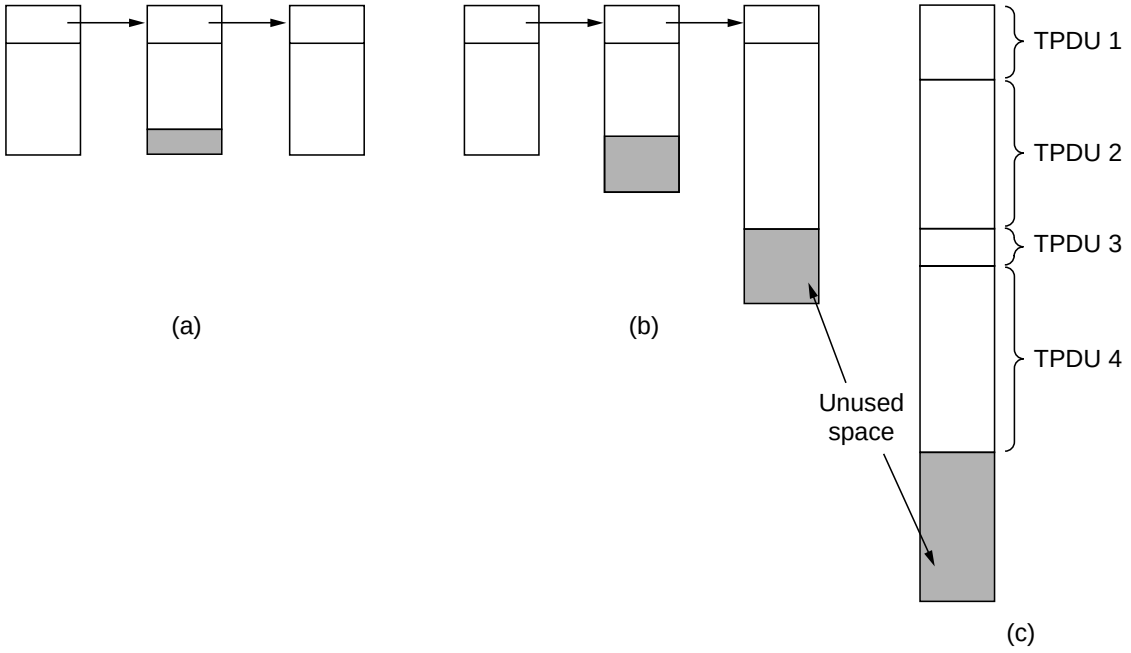

Fig. 6-15. (a) Chained fixed-size buffers. (b) Chained variable-size buffers. (c) One large circular buffer per connection.

	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 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

Fig. 6-16. Dynamic buffer allocation. The arrows show the direction of transmission. An ellipsis (...) indicates a lost TPDU.

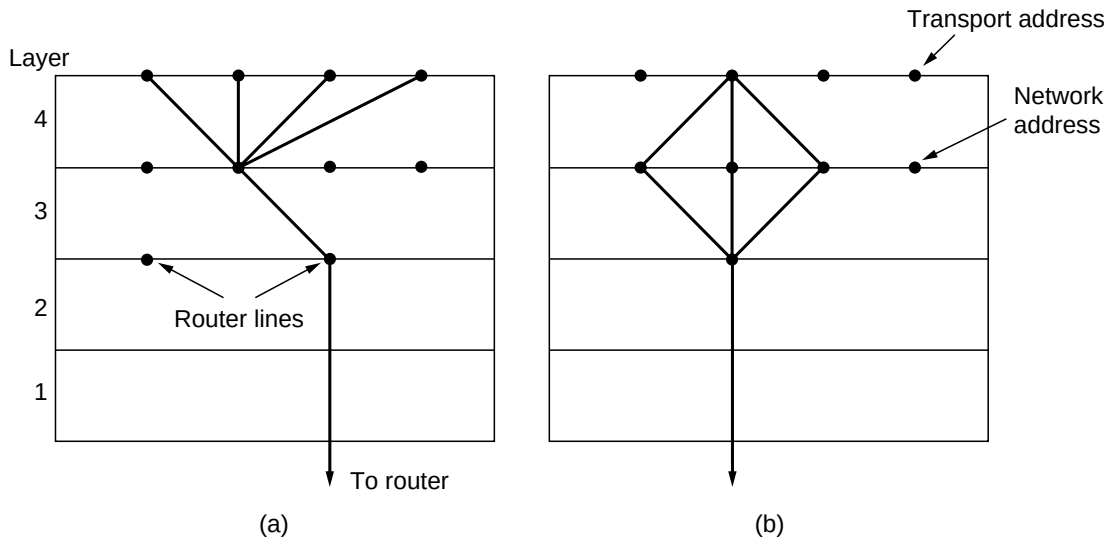

Fig. 6-17. (a) Upward multiplexing. (b) Downward multiplexing.

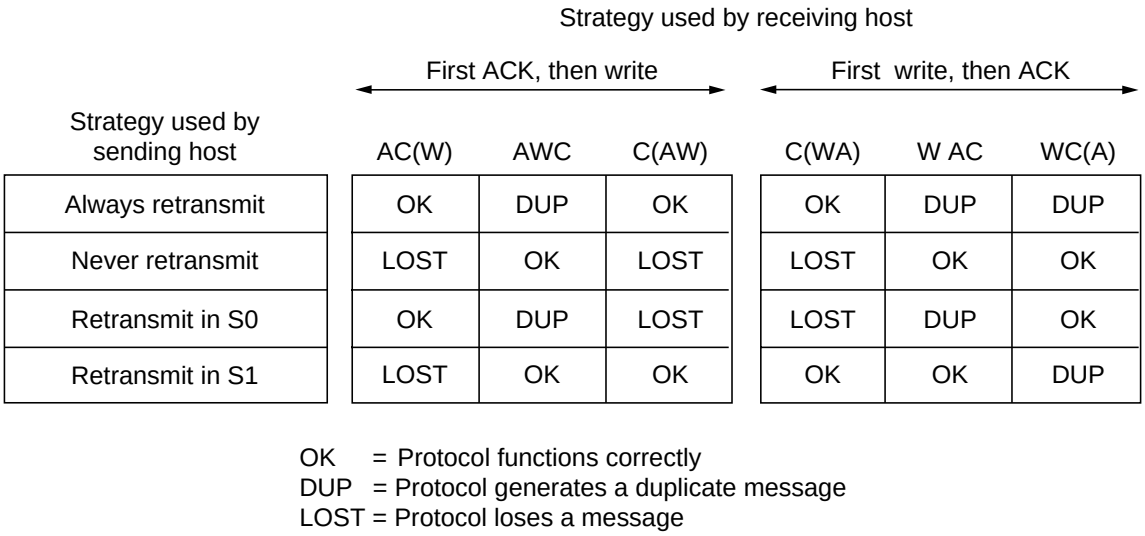

Fig. 6-18. Different combinations of client and server strategy.

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

Fig. 6-19. The network layer packets used in our example.

```
#define MAX_CONN 32               /* maximum 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];

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 = 1, 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 */
  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 */
}

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 */
       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);
}

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];

    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();
  }
}

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);
                }
        }
  }
}
```

Fig. 6-20. An example transport entity.

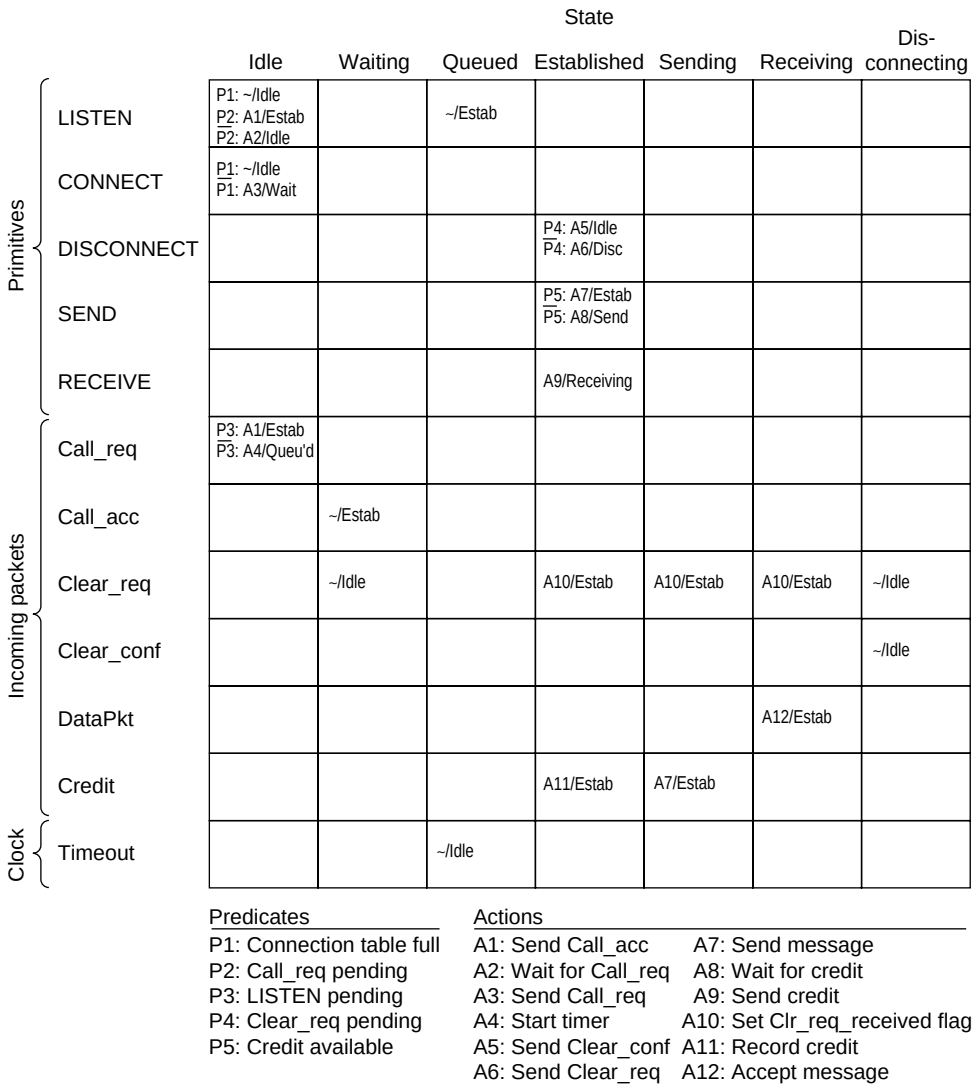

Fig. 6-21. The example protocol as a finite state machine. Each entry has an optional predicate, an optional action, and the new state. The tilde indicates that no major action is taken. An overbar above a predicate indicates the negation of the predicate. Blank entries correspond to impossible or invalid events.

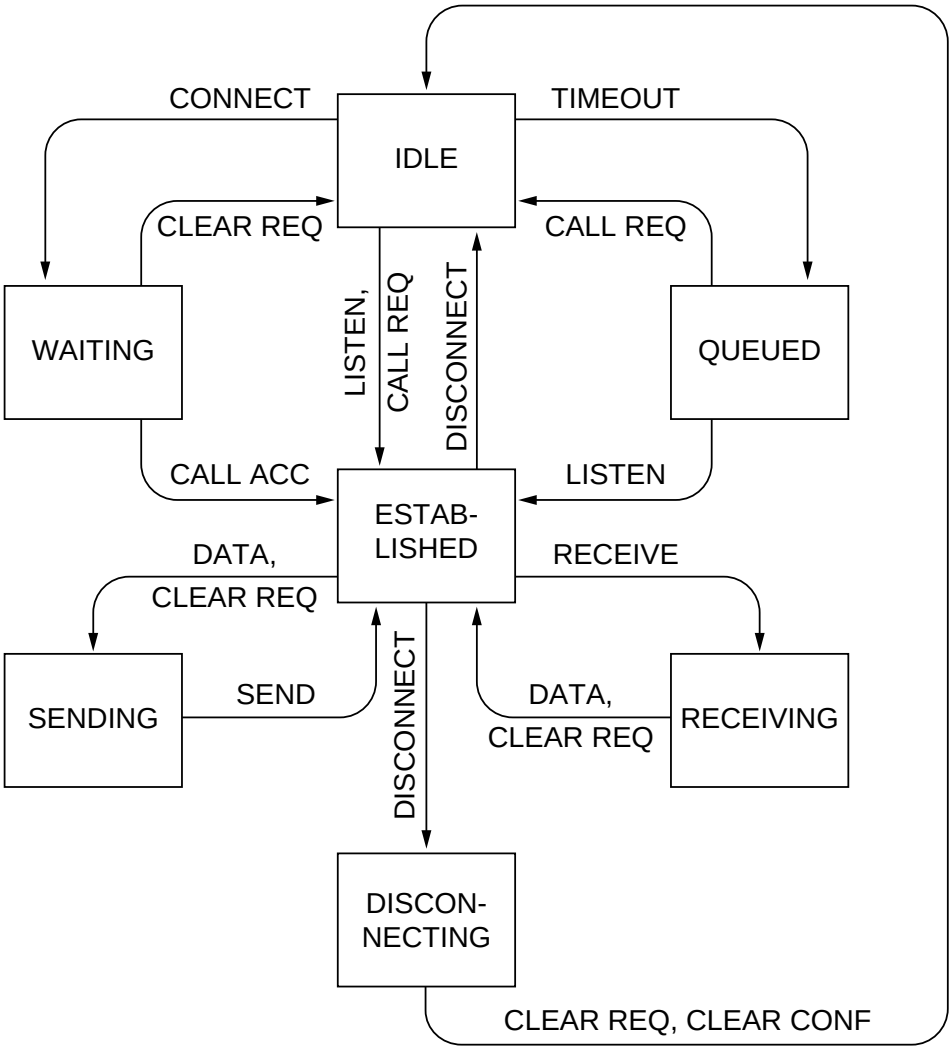

Fig. 6-22. The example protocol in graphical form. Transitions that leave the connection state unchanged have been omitted for simplicity.

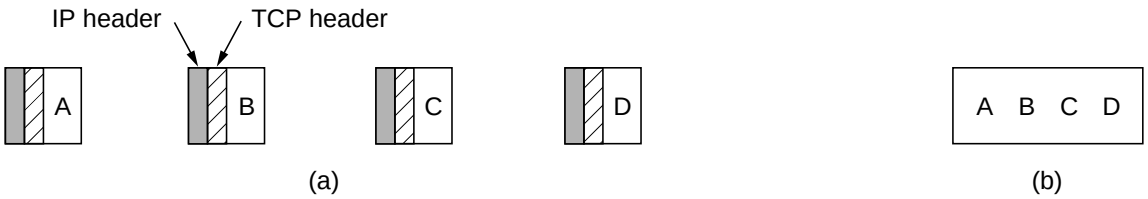

Fig. 6-23. (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

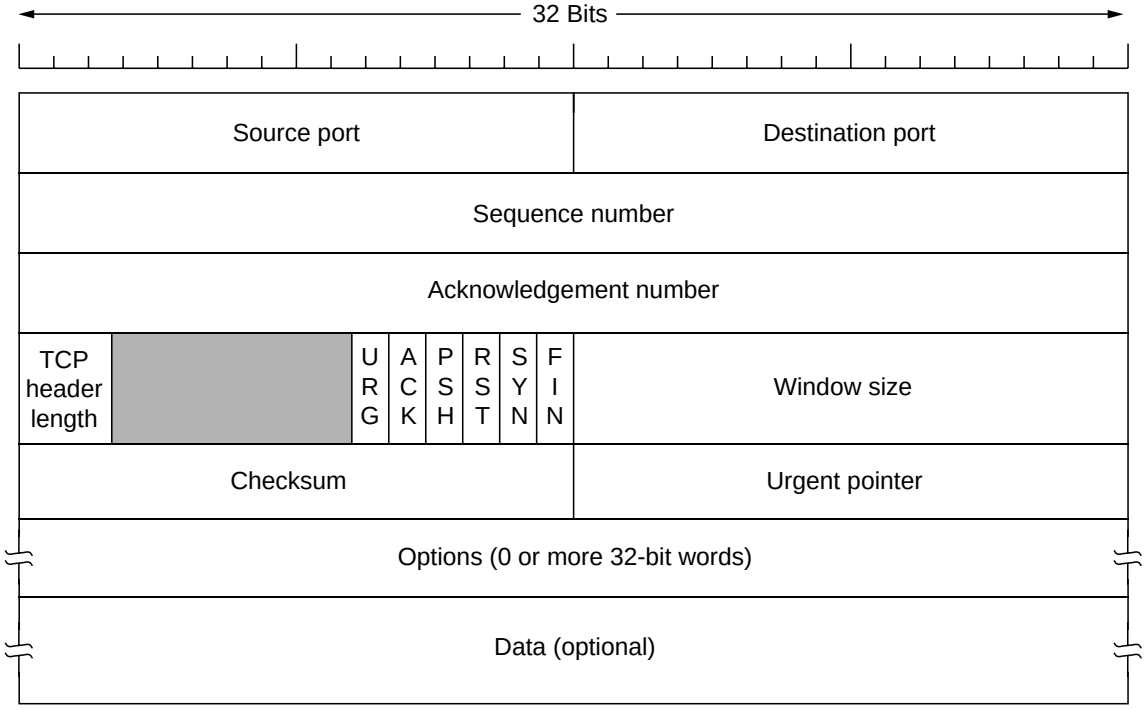

Fig. 6-24. The TCP header.

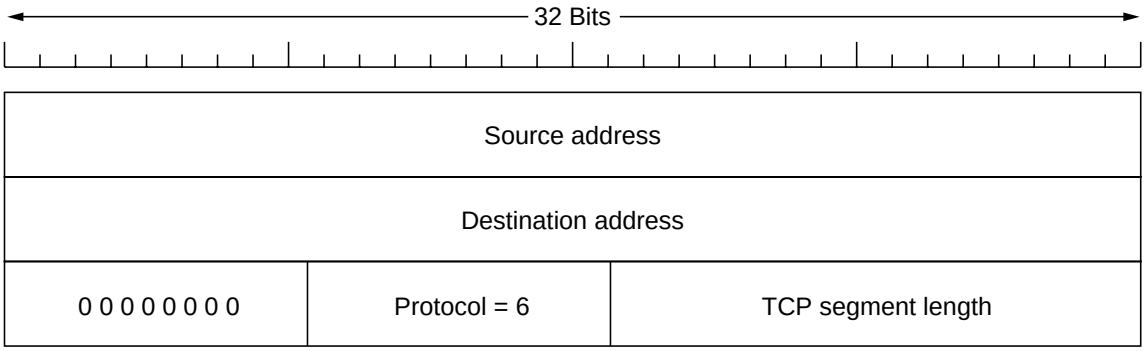

Fig. 6-25. The pseudoheader included in the TCP checksum.

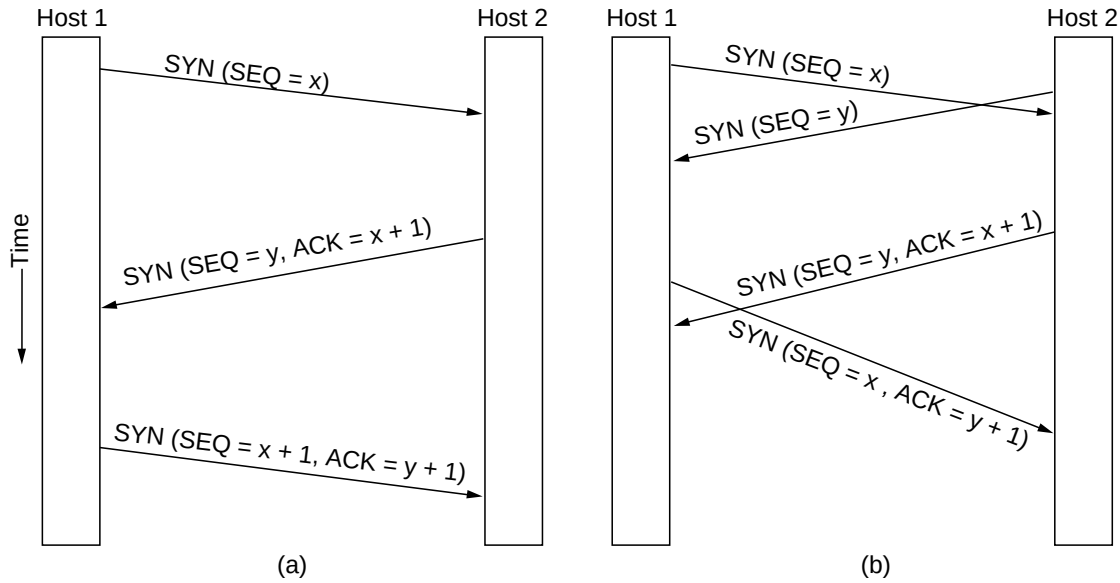

Fig. 6-26. (a) TCP connection establishment in the normal case. (b) Call collision.

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

Fig. 6-27. The states used in the TCP connection management finite state machine.

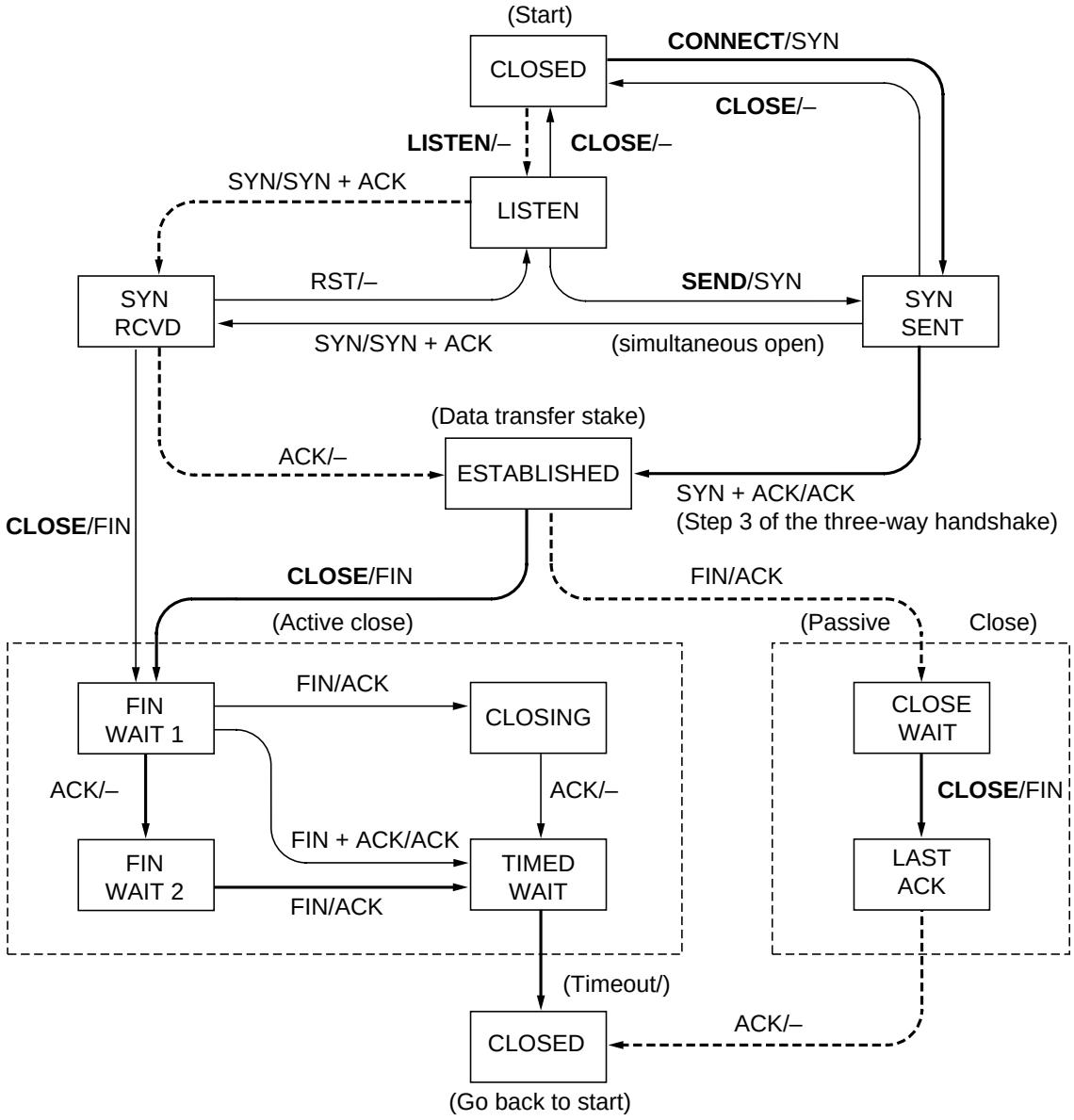

Fig. 6-28. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events.

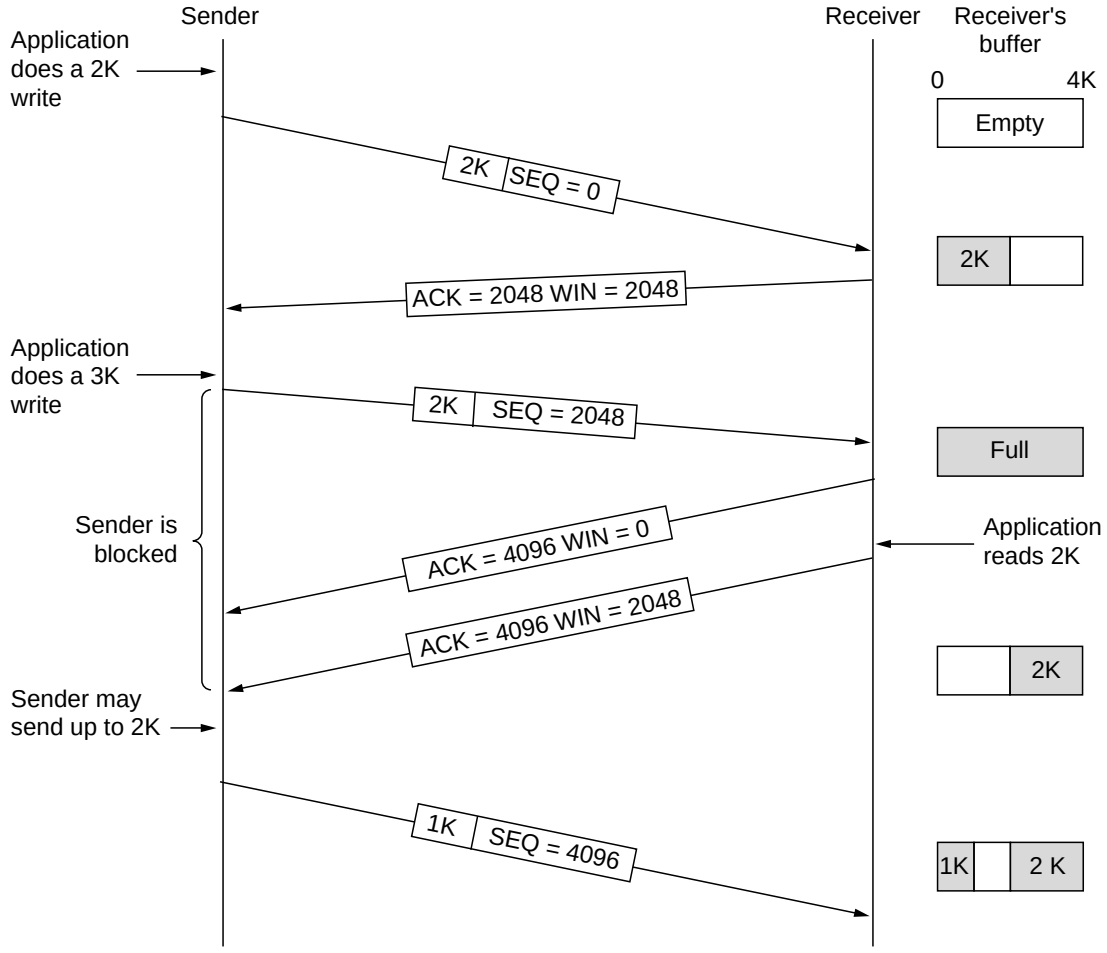

Fig. 6-29. Window management in TCP.

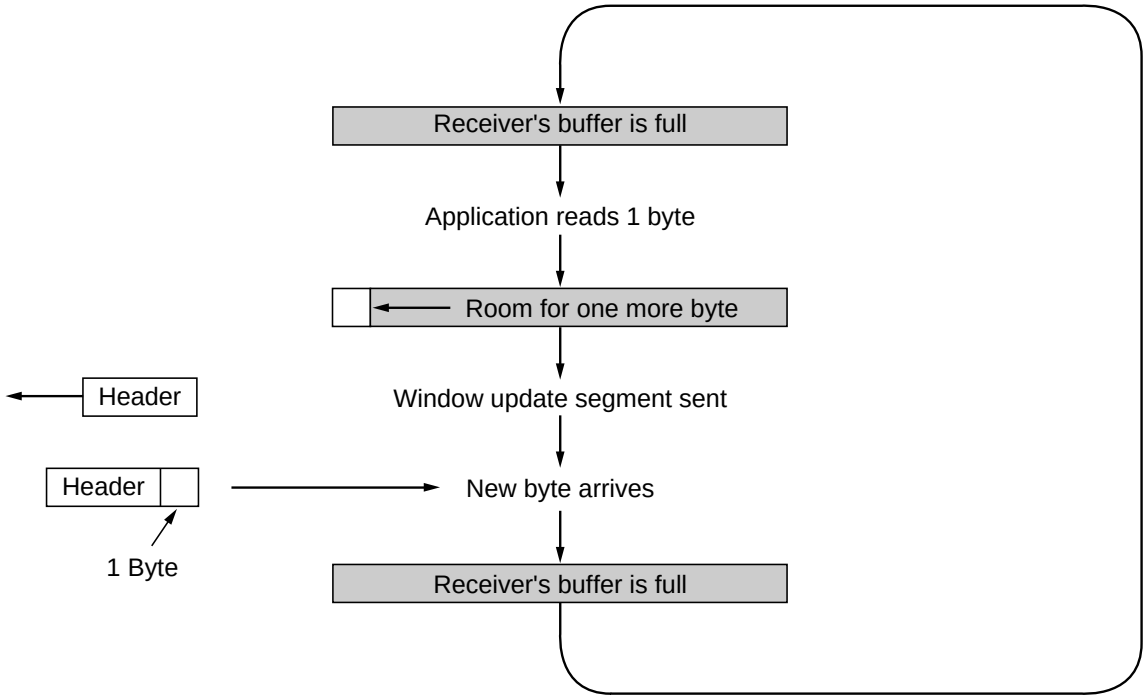

Fig. 6-30. Silly window syndrome.

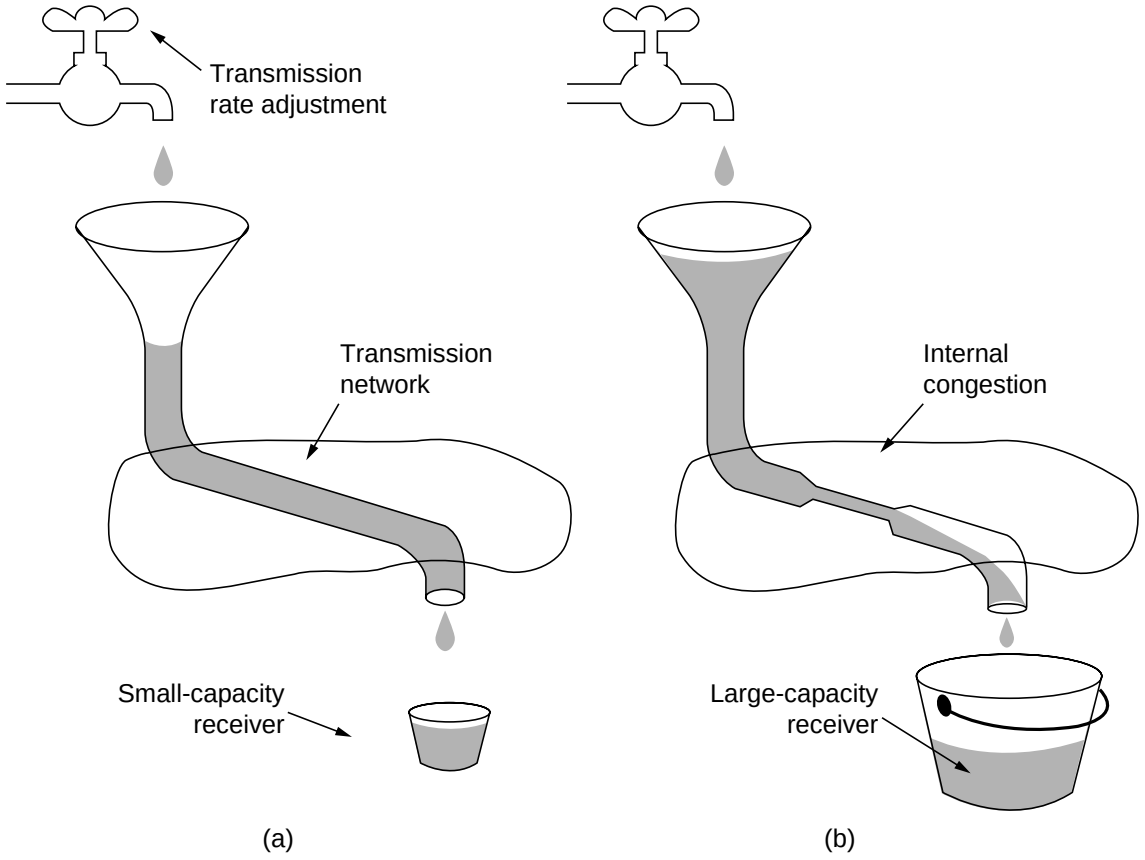

Fig. 6-31. (a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.

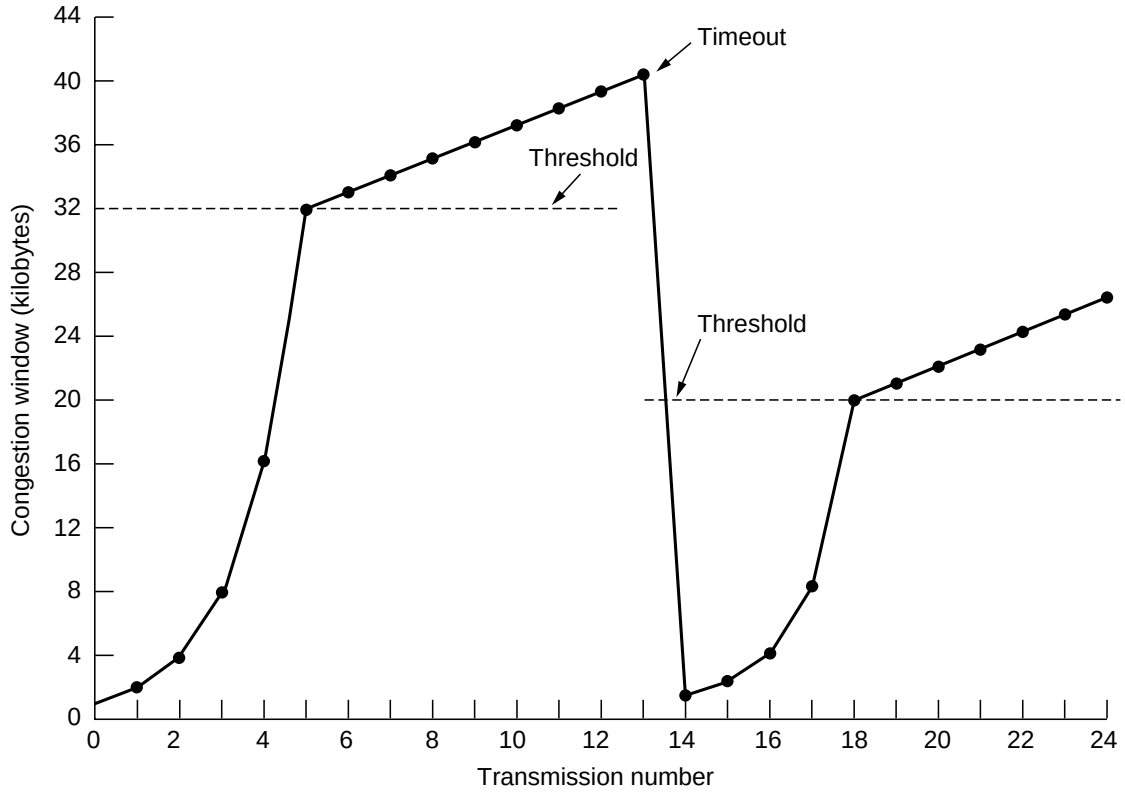

Fig. 6-32. An example of the Internet congestion algorithm.

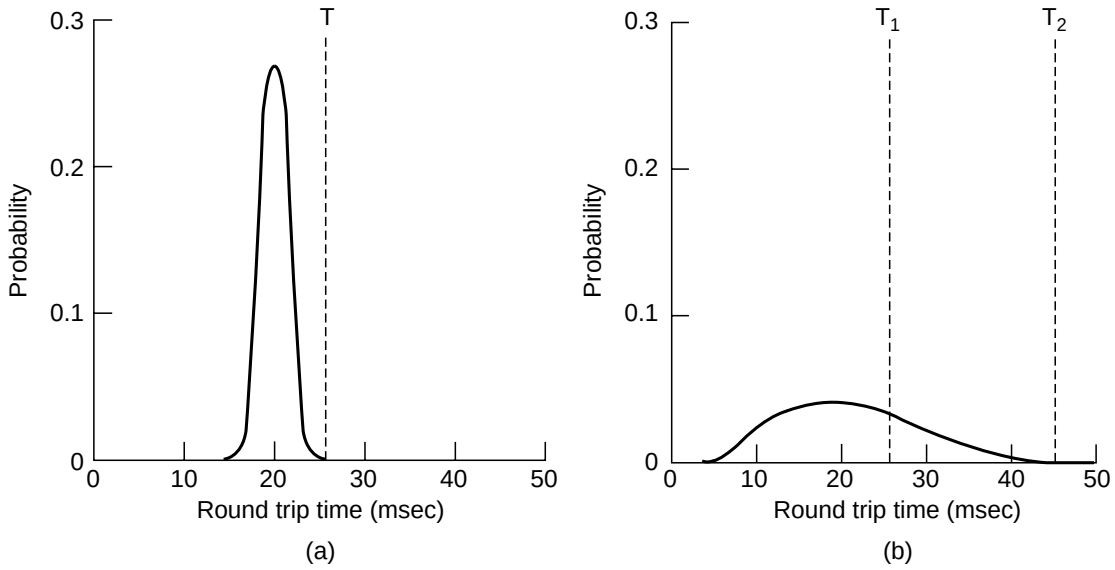

Fig. 6-33. (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

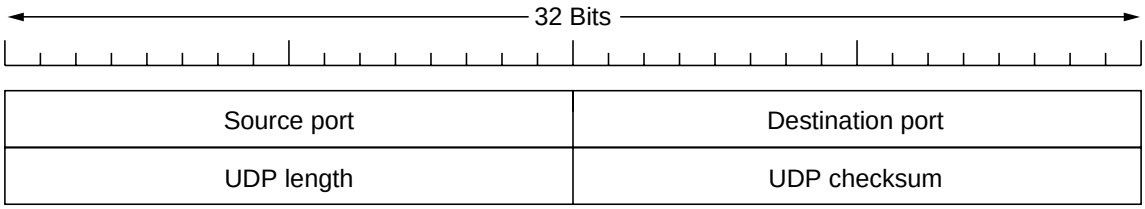

Fig. 6-34. The UDP header.

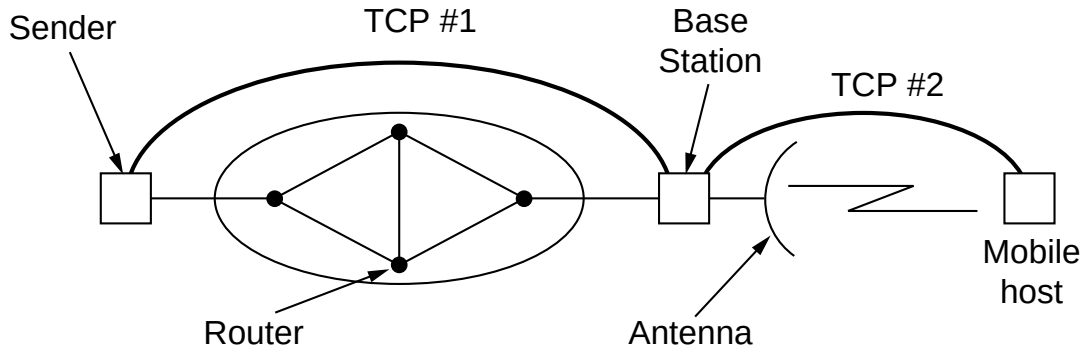

Fig. 6-35. Splitting a TCP connection into two connections.

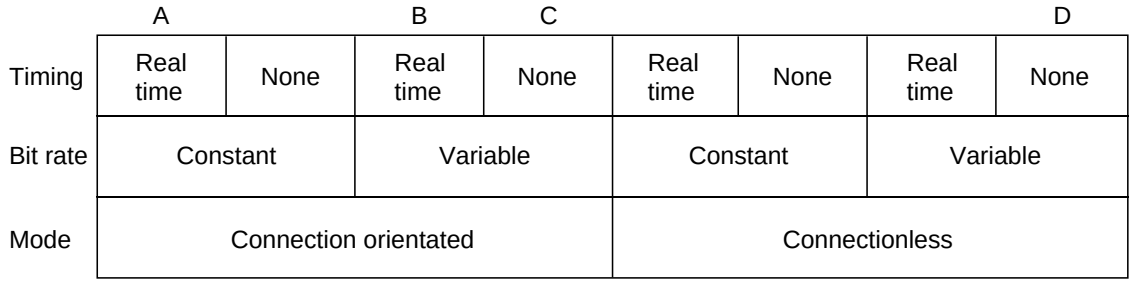

Fig. 6-36. Original service classes supported by AAL (now obsolete).

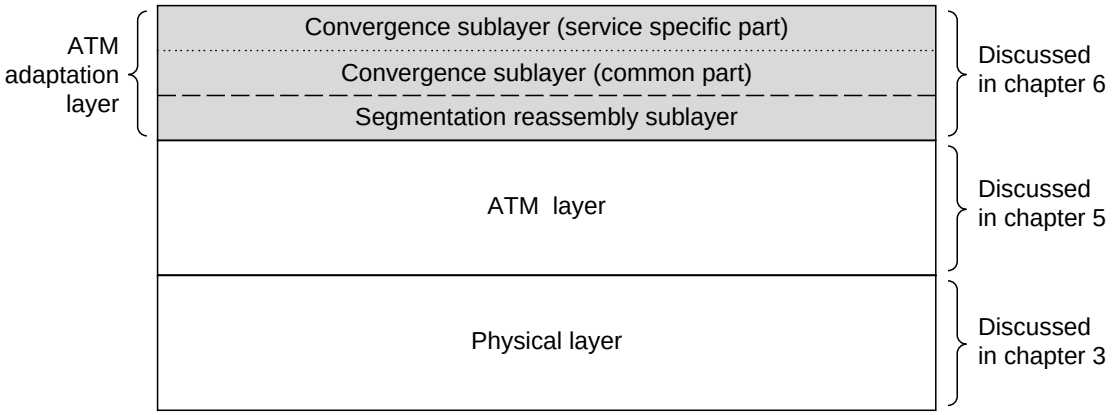

Fig. 6-37. The ATM model showing the ATM adaptation layer and its sublayers.

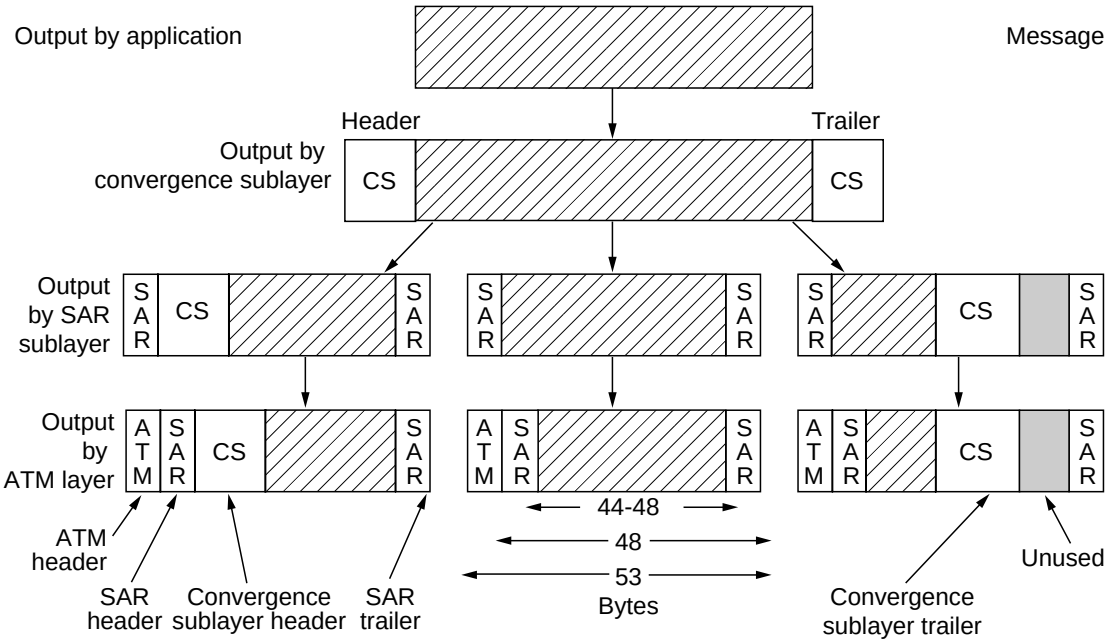

Fig. 6-38. The headers and trailers that can be added to a message in an ATM network.

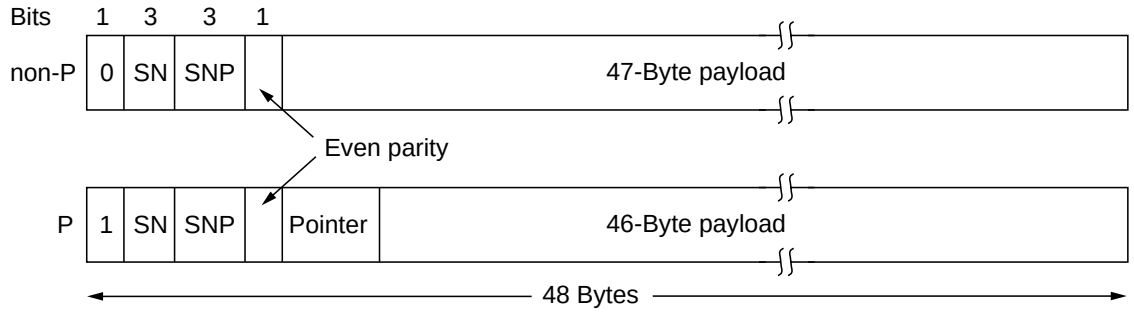

Fig. 6-39. The AAL 1 cell format.

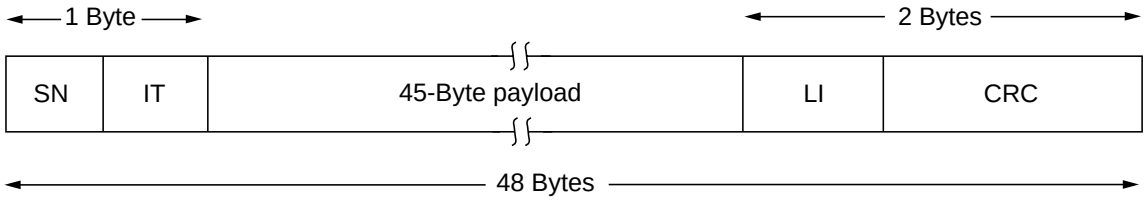

Fig. 6-40. The AAL 2 cell format.

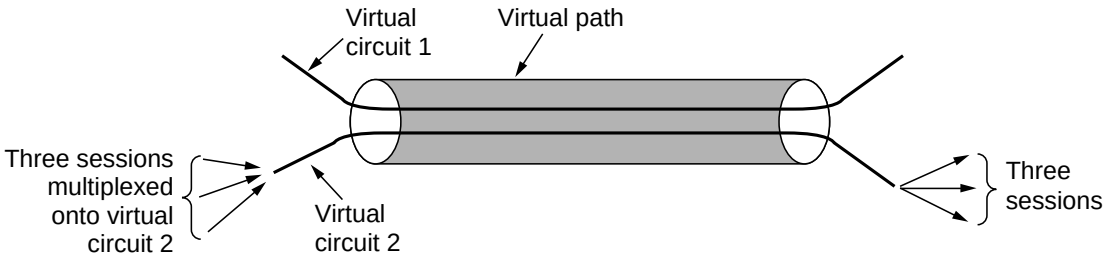

Fig. 6-41. Multiplexing of several sessions onto one virtual circuit.

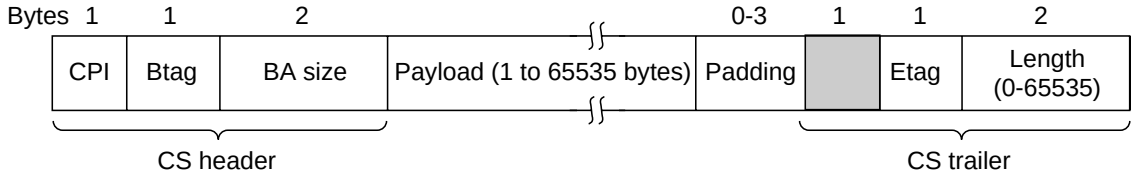

Fig. 6-42. AAL 3/4 convergence sublayer message format.

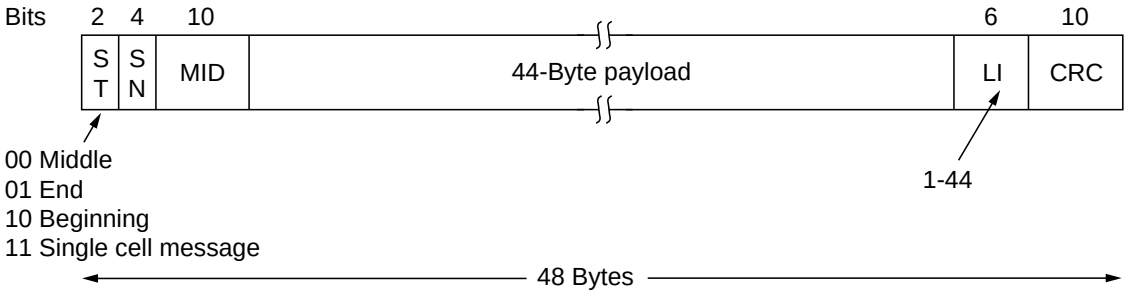

Fig. 6-43. The AAL 3/4 cell format.

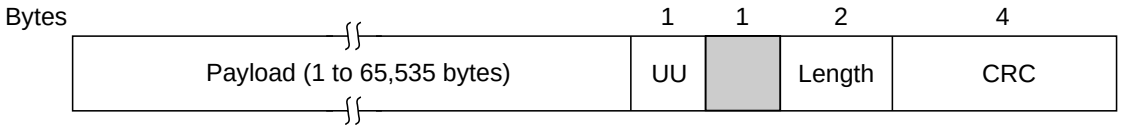

Fig. 6-44. AAL 5 convergence sublayer message format.

Item	AAL 1	AAL 2	AAL 3/4	AAL 5
Service class	A	B	C/D	C/D
Multiplexing	No	No	Yes	No
Message delimiting	None	None	Btag/Etag	Bit in PTI
Advance buffer allocation	No	No	Yes	No
User bytes available	0	0	0	1
CS padding	0	0	32-Bit word	0–47 bytes
CS protocol overhead (bytes)	0	0	8	8
CS checksum	None	None	None	32 Bits
SAR payload bytes	46–47	45	44	48
SAR protocol overhead (bytes)	1–2	3	4	0
SAR checksum	None	None	10 Bits	None

Fig. 6-45. Some differences between the various AAL protocols.

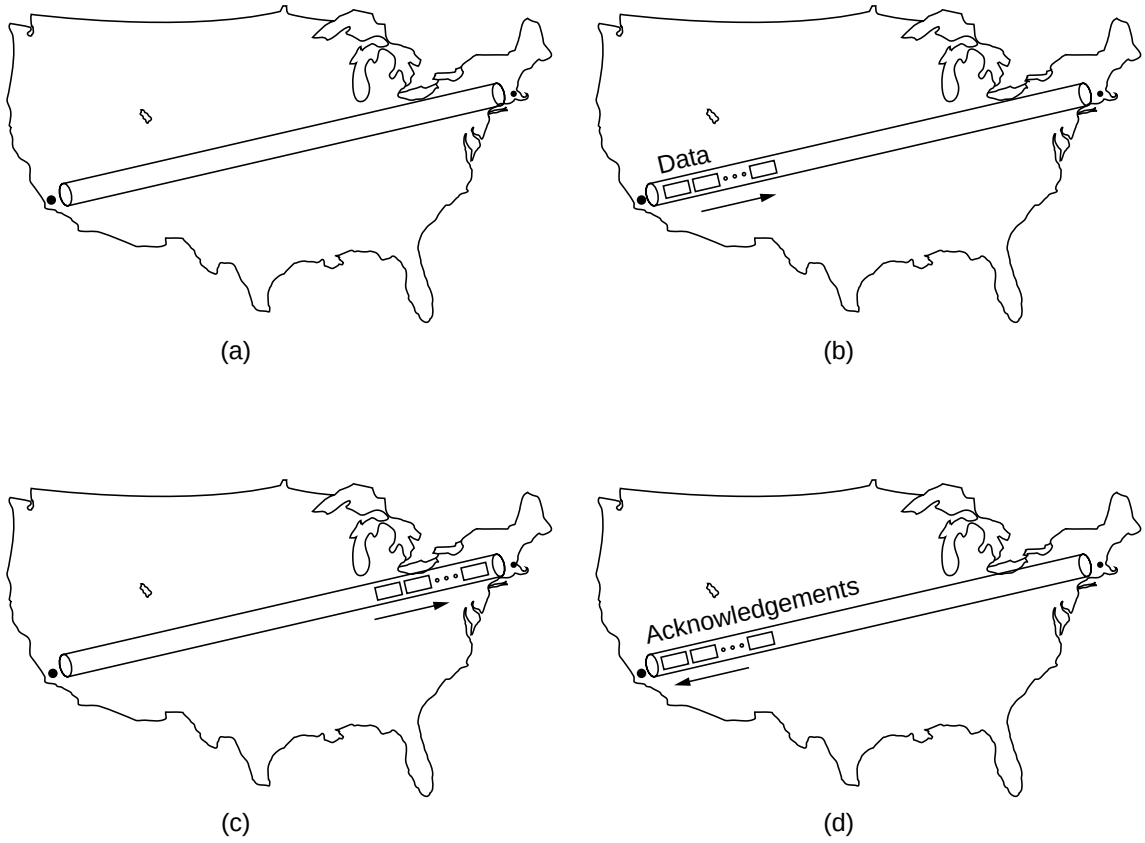

Fig. 6-46. The state of transmitting one megabit from San Diego to Boston. (a) At $t = 0$. (b) After 500 μsec. (c) After 20 msec. (d) After 40 msec.

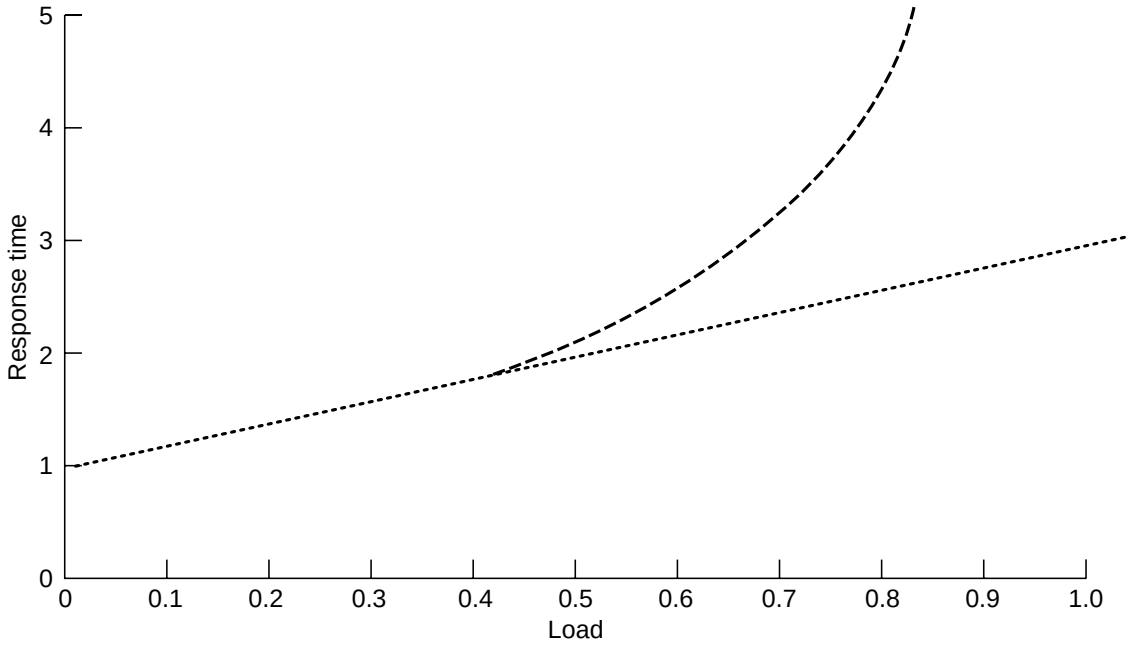

Fig. 6-47. Response as a function of load.

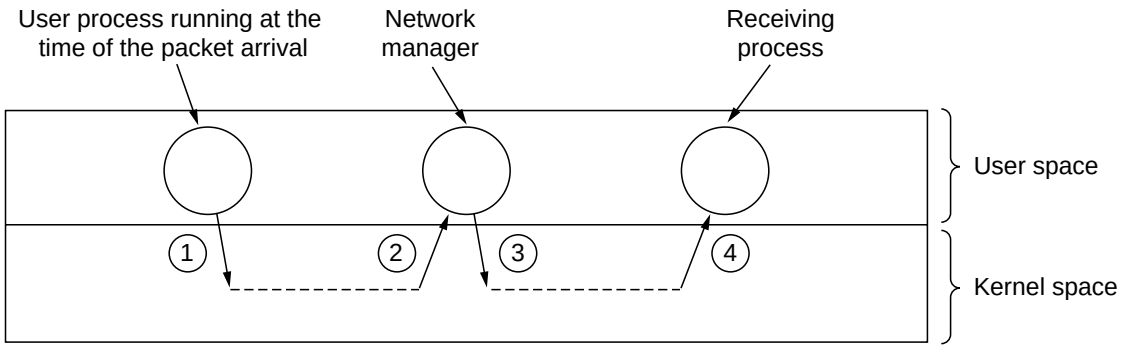

Fig. 6-48. Four context switches to handle one packet with a user-space network manager.

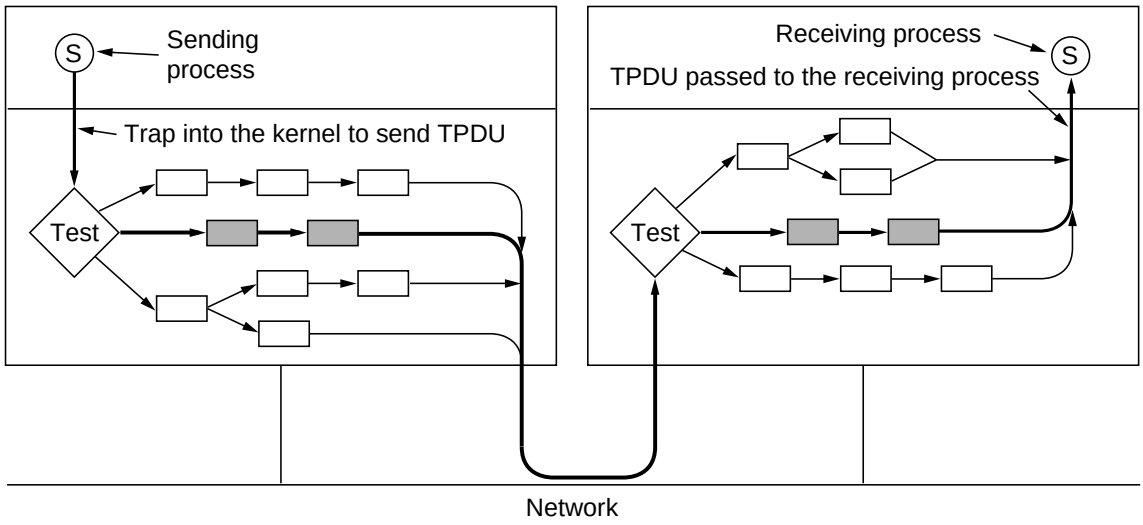

Fig. 6-49. The fast path from sender to receiver is shown with a heavy line. The processing steps on this path are shaded.

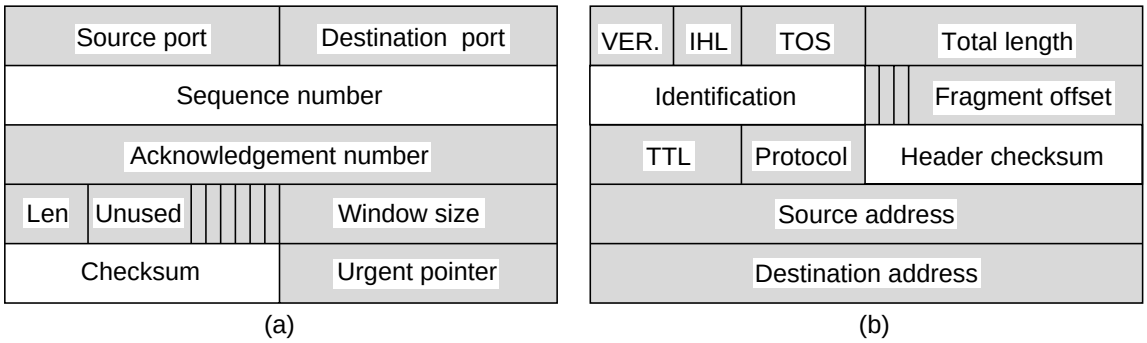

Fig. 6-50. (a) TCP header. (b) IP header. In both cases, the shaded fields are taken from the prototype without change.

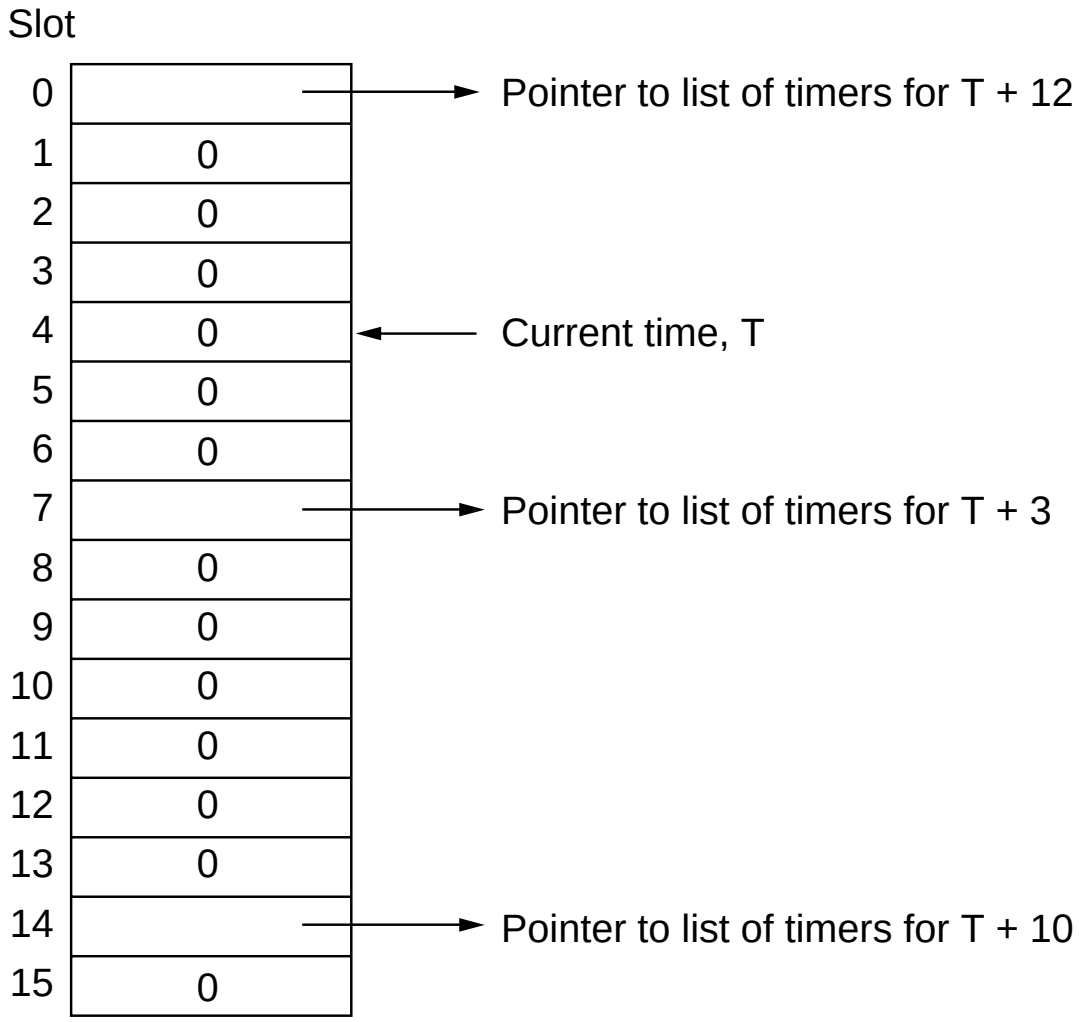

Fig. 6-51. A timing wheel.

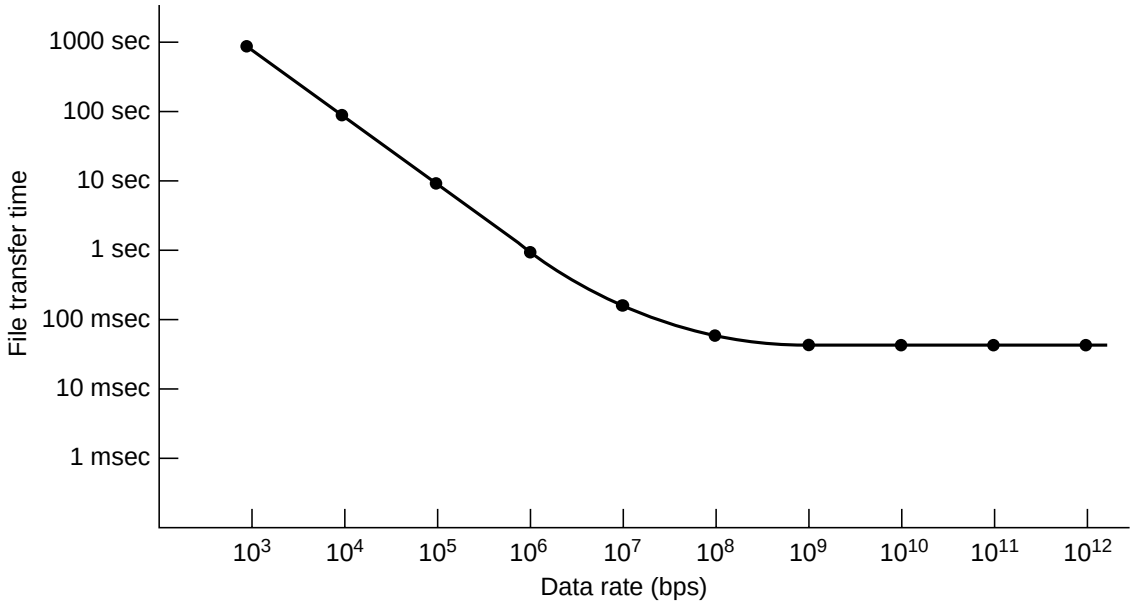

Fig. 6-52. Time to transfer and acknowledge a 1-megabit file over a 4000-km line.