UNIT IV

The Transport Layer

PREVIOUS LAYERS

- THE PURPOSE OF THE PHYSICAL LAYER IS TO TRANSPORT A RAW BIT STREAM FROM ONE MACHINE TO ANOTHER.
- THE MAIN TASK OF THE DATA LINK LAYER IS TO TRANSFORM A RAW TRANSMISSION FACILITY INTO A LINE THAT APPEARS FREE OF UNDETECTED TRANSMISSION ERRORS TO THE NETWORK LAYER.

- THE NETWORK LAYER IS CONCERNED WITH GETTING PACKETS FROM THE SOURCE ALL THE WAY TO THE DESTINATION. GETTING TO THE DESTINATION MAY REQUIRE MAKING MANY HOPS AT INTERMEDIATE ROUTERS ALONG THE WAY. THUS, THE NETWORK LAYER IS THE LOWEST LAYER THAT DEALS WITH END-TO-END TRANSMISSION.

- THE TRANSPORT LAYER IS THE HEART OF WHOLE PROTOCOL HIERARCHY
- ITS TASK IS TO PROVIDE RELIABLE, COST-EFFECTIVE DATA TRANSPORT FROM SOURCE MACHINE TO DESTINATION MACHINE, INDEPENDENTLY OF THE PHYSICAL NETWORK OR NERWORKS CURRENTLY IN USE.

- WITHOUT THE TRANSPORT LAYER, THE WHOLE CONCEPT OF LAYERED PROTOCOLS WOULD MAKE LITTLE SENSE.
- IN THIS CHAPTER WE WILL STUDY THE TRANSPORT LAYER IN DETAIL, INCLUDING ITS SERVICES, DESIGN, PROTOCOLS, AND PERFORMANCE.

6.1. The Transport Service

6.2. Elements of Transport Protocols

6.3. A Simple Transport Protocol

6.4. The Internet Transport Protocols: UDP

6.5. The Internet Transport Protocols: TCP

6.6. Performance Issues

6.7. Summary

6.1. The Transport Service

- IN THE FOLLOWING SECTIONS WE WILL PROVIDE AN INTRODUCTION TO THE TRANSPORT SERVICE.
- WE LOOK AT WHAT KIND OF SERVICE IS PROVIDED TO THE APPLICATION LAYER.

6.1. The Transport Service

- TO MAKE THE ISSUE OF TRANSPORT SERVICE MORE CONCRETE, WE WILL EXAMINE TWO SETS OF TRANSPORT LAYER PRIMITIVES.
- FIRST COMES A SIMPLE ONE TO SHOW THE BASIC IDEAS.
- THEN COMES THE INTERFACE COMMONLY USED IN INTERNET

6.1. The Transport Service

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

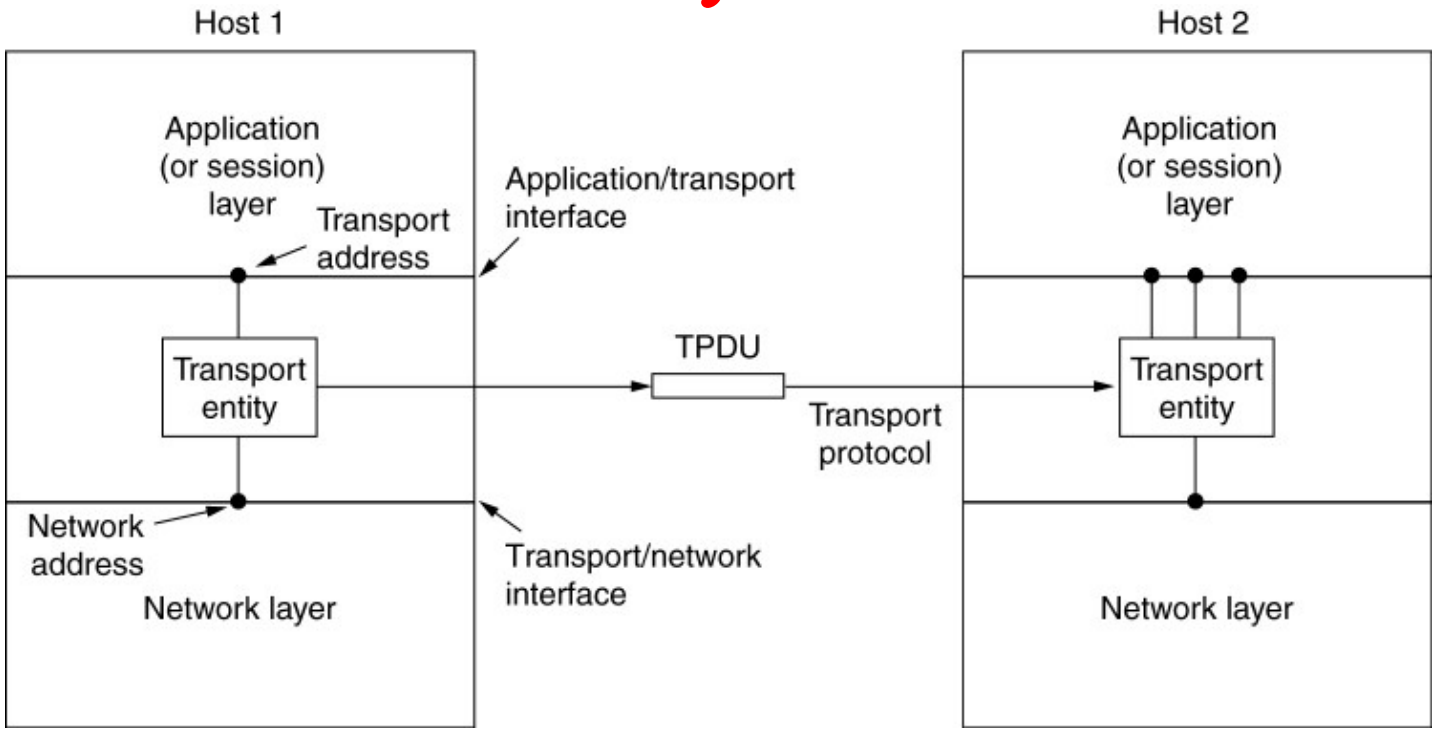
6.1.1. Services Provided to the Upper Layers

- THE ULTIMATE GOAL OF THE TRANSPORT LAYER IS TO PROVIDE EFFICIENT, RELIABLE, AND COST-EFFECTIVE SERVICE TO ITS USERS, NORMALLY PROCESSES IN THE APPLICATION LAYER.
- TO ACHIEVE THIS GOAL, TRANSPORT LAYER MAKES USE OF SERVICES PROVIDED BY THE NETWORK LAYER.

6.1.1. Services Provided to the Upper Layers

- THE HARDWARE AND/OR SOFTWARE WITHIN THE TRANSPORT LAYER THAT DOES THE WORK IS CALLED THE TRANSPORT ENTITY.
- THE TRANSPORT ENTITY CAN BE LOCATED IN THE OPERATING SYSTEM KERNEL, IN A SEPARATE USER PROCESS, IN A LABRARY PACKAGE BOUND INTO NETWORK APPLICATIONS, OR CONCEIVABLY ON NETWORK INTERFACE CARD.

6.1.1. Services Provided to the Upper Layers



The (logical) relationship of the network, transport, and application layers.

6.1.1. Services Provided to the Upper Layers

- Just as there are two types of network service, connection-oriented and connectionless, there are also two types of transport services.
- The connection-oriented transport service is similar to the connection-oriented network service in many ways.

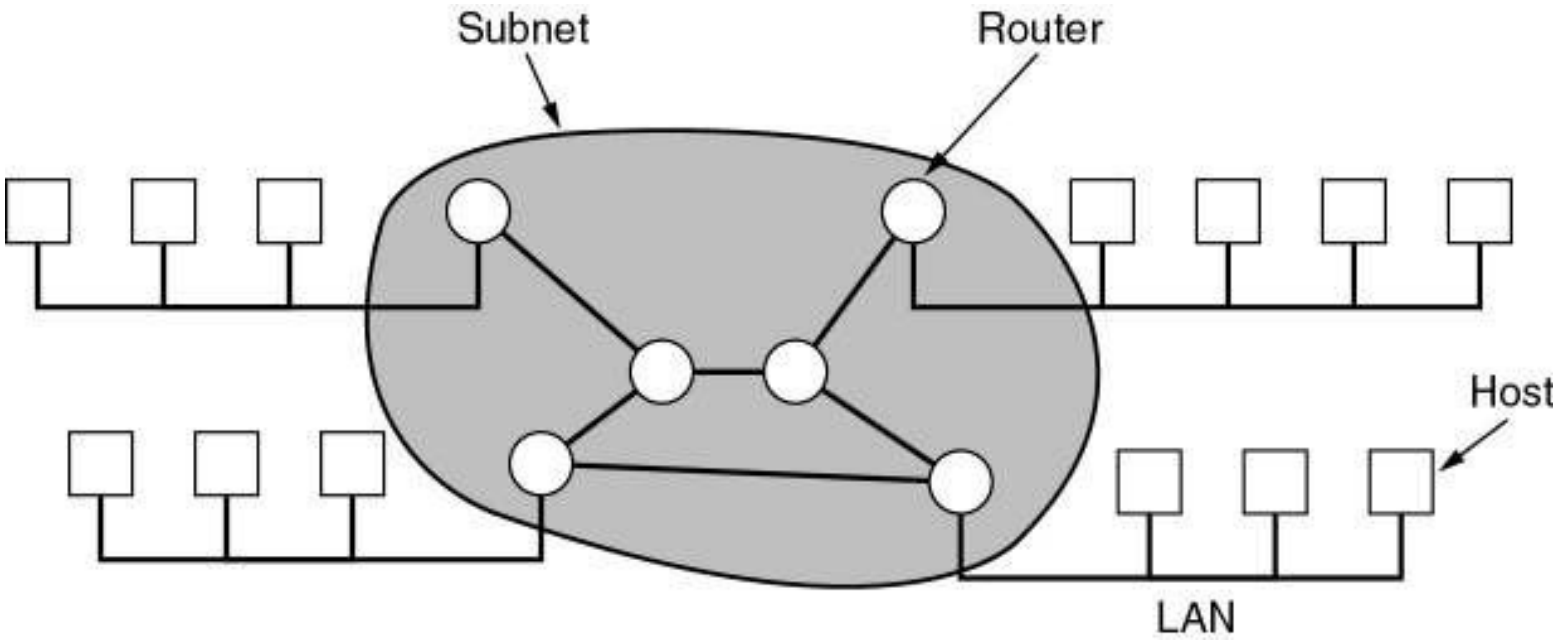
6.1.1. Services Provided to the Upper Layers

- In both cases, connections have three phases: establishment, data transfer, and release.
- Addressing and flow control are also similar in both layers.
- Furthermore, connectionless transport service is also very similar to the connectionless network service.

6.1.1. Services Provided to the Upper Layers

- QUESTION: If the transport layer service is so similar to the network layer service, why are there two distinct layers?
- The transport code runs entirely on the users' machines, but the network layer mostly runs on the routers, which are operated by the carrier (at least for a wide area network).

Wide Area Networks



Relation between hosts on LANs and the subnet.

6.1.1. Services Provided to the Upper Layers

- What happens if the network layer offers inadequate service? Suppose that it frequently loses packets. What happens if routers crash from time to time?
- Problems occur, that's what.
- The users have no real control over the network layer, so they cannot solve the problem of poor service by using better routers or putting more error handling in the data link layer.

6.1.1. Services Provided to the Upper Layers

- The only possibility is to put on top of the network layer another layer that improves the quality of the service.
- If in a connection-oriented subnet, a transport entity is informed halfway through a long transmission that its network connection has been abruptly terminated.

6.1.1. Services Provided to the Upper Layers

- It can set up a new network connection to the remote transport entity.
- Using this new network connection, it can send a query to its peer asking which data arrived and which did not, and then pick up from where it left off.

6.1.1. Services Provided to the Upper Layers

- In essence, the existence of the transport layer makes it possible for the transport service to be more reliable than the underlying network service. Lost packets and mangled data can be detected and compensated for by the transport layer.

6.1.1. Services Provided to the Upper Layers

- Furthermore, the transport service primitives can be implemented as calls to library procedures in order to make them independent of the network service primitives.
- The network service calls may vary considerably from network to network.

6.1.1. Services Provided to the Upper Layers

- By hiding the network service behind a set of transport service primitives, changing the network service merely requires replacing one set of library procedures by another one that does the same thing with a different underlying service.

6.1.1. Services Provided to the Upper Layers

- Thanks to the transport layer, application programmers can write code according to a standard set of primitives and have these programs work on a widely variety of network, without having to worry about dealing with different subnet interface and unreliable transmission.

6.1.1. Services Provided to the Upper Layers

- If all real networks were flawless and all had the same service primitives and were guaranteed never, ever to change, the transport layer might not be needed.
- However, in real world it fulfills the key function of isolating the upper layers from the technology, design, and imperfections of the subnet.

6.1.1. Services Provided to the Upper Layers

- For this reason, many people have traditionally made a distinction between layers 1 through 4 on the one hand and layer(s) above 4 on the other.
- The bottom four layers can be seen as the transport service provider, whereas the upper layer(s) are the transport service user.

6.1.1. Services Provided to the Upper Layers

- This distinction of provider versus user has a considerable impact on the design of the layers and puts the transport layer in a key position, since it forms the major boundary between the provider and user of the reliable data transmission service.

6.1.2. Transport Service Primitives

- To allow users to access the transport service, the transport layer must provide some operations to application programs, that is, a transport service interface.
- Each transport service has its own interface.

6.1.2. 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.

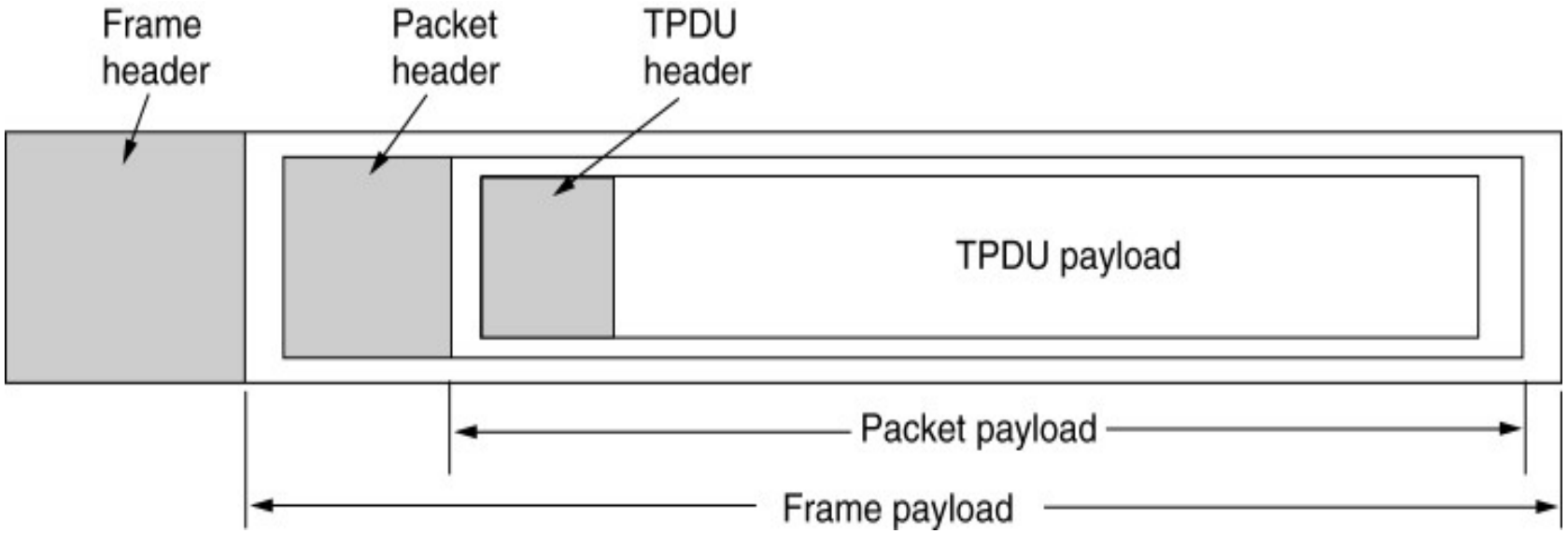
6.1.2. Transport Service Primitives

- This transport interface allows application programs to establish, use, and then release connections, which is sufficient for many applications.
- TPDU (Transport Protocol Data Unit) – for messages sent from transport entity to transport entity.

6.1.2. Transport Service Primitives

- Thus TPDUs (exchanged by the transport layer) are contained in packets (exchanged by the network layer).
- In turn, packets are contained in frames (exchanged by the data link layer).
- When a frame arrives, the data link layer processes the frame header and passes the contents of the frame payload field up to the network entity.
- The network entity processes the packet header and passes the contents of the packet payload up to the transport entity.

6.1.2. Transport Service Primitives (2)



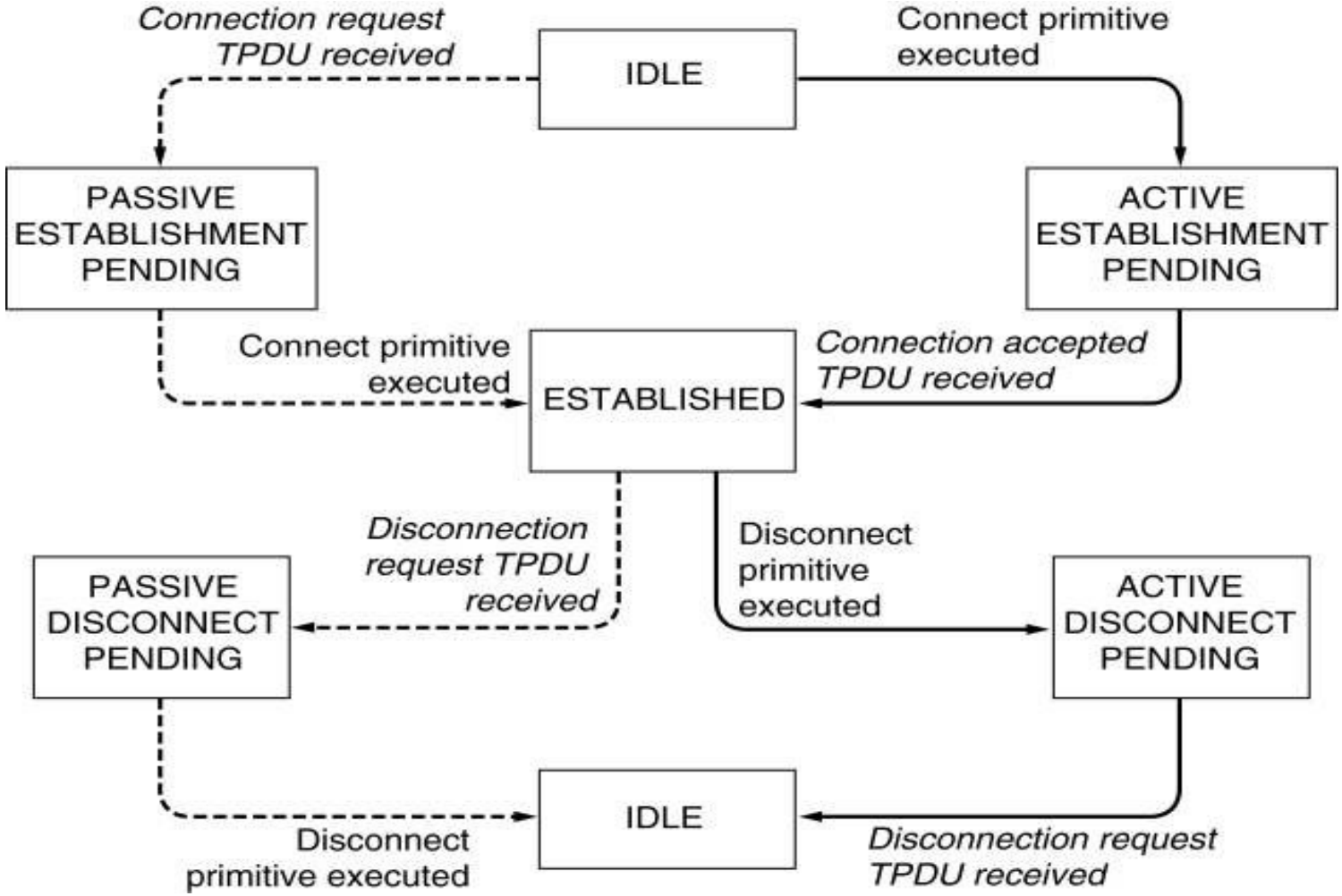
The nesting of TPDUs, packets, and frames.

To see how these primitives might be used, consider an application with a server and a number of remote clients.

6.1.2. Transport Service Primitives

- 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.

6.1.2.Transport Service Primitives (3)



6.1.3. Berkeley Sockets

- Let us now briefly inspect another set of transport primitives, the socket primitives used in Berkeley UNIX for TCP.
- These primitives are widely used for Internet programming.
- The first four primitives in the list are executed in that order by servers, others are executed by client.

6.1.3. 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.

6.1.4. 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 */
  }
}
```

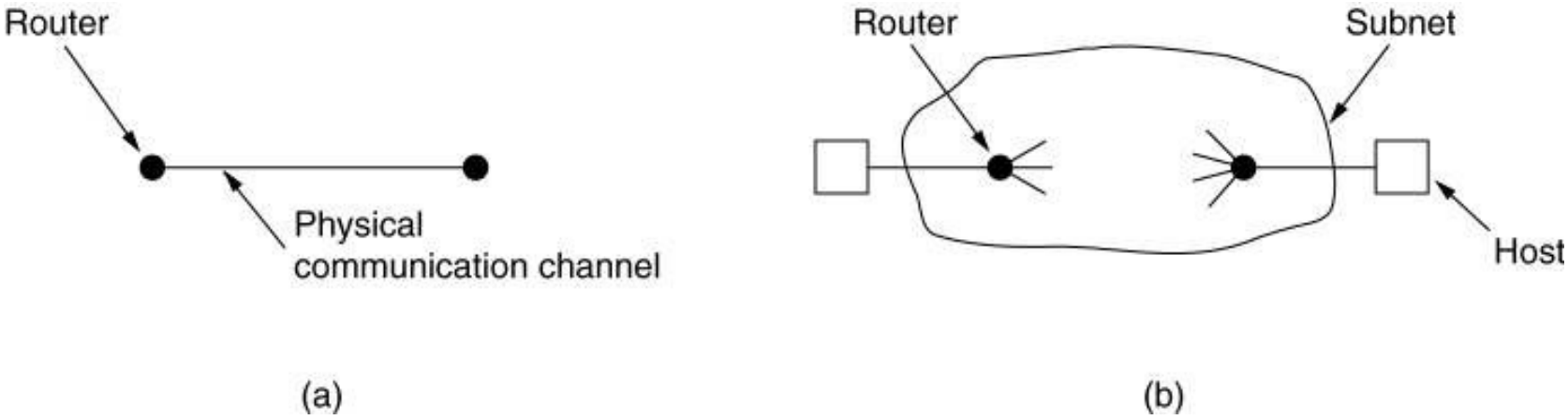
6.2. Elements of Transport Protocols

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

6.2. Elements of Transport Protocols

- The transport service is implemented by a transport protocol used between the two transport entities.
- Transport protocols as the data link protocols have to deal with error control, sequencing, and flow control.
- The differences between these protocols are due to major dissimilarities between the environments in which the two protocols operate.

Transport Protocol



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

6.2. Elements of Transport Protocols

. Differences betwen data link layer and transport layer:

1) At data link layer, two routers communicate directly via a physical channel, whereas at the transport layer, this physical channel is replaced by the entire subnet.

6.2. Elements of Transport Protocols

2) In the data link layer, it is not necessary for a router to specify which router it wants to talk to – each outgoing line uniquely specifies a particular router. In the transport layer, explicit addressing of destinations is required.

6.2. Elements of Transport Protocols

3) In the data link layer, the process of establishing a connection over the wire is smple: the other end is always there (unless it has crashed, in which case it is not there). In the transport layer, initial connection esteblishment is more complicated.

6.2. Elements of Transport Protocols

4) In the data link layer, when a router sends a frame, it may arrive or lost, but it cannot bounce around for a while, etc. In the transport layer, if the subnet uses datagrams and adaptive routing inside, there is a nonnegligible probability that a packet may be stored for a number of seconds and then delivered later.

6.2. Elements of Transport Protocols

5) A final difference between the data link layer and transport layers is following: Buffering and flow control are needed in both layers, but the presence of a large and dynamically varying number of connections in the transport layer may require a different approch than used in data link layer.

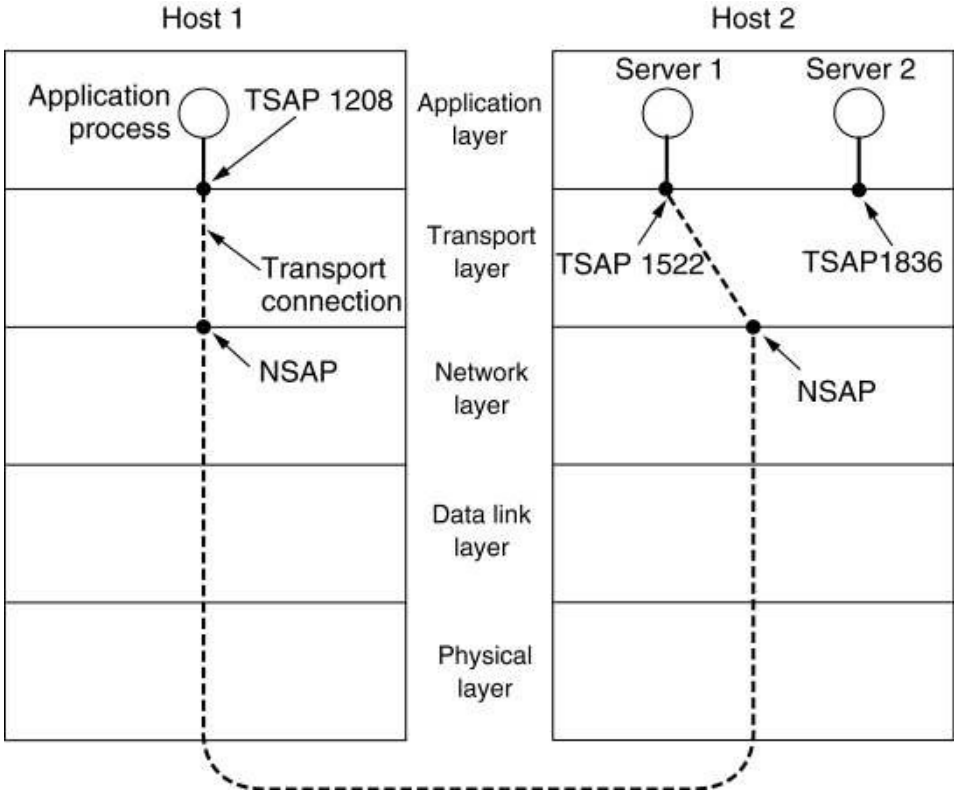
6.2.1. Addressing

- When an application process wishes to set up a connection to a remote application process, it must specify which one to connect to.
- The method normally used is to define transport addressing to which processes can listen for connection requests.

6.2.1. Addressing

- In Internet, these end points are called ports.
- In ATM networks, they are called AAL-SAPs.
- We will use generic term TSAP, (Transport Service Access Point).
- The analogous end points in the network layer are then called NSAPs.
- IP addresses are examples of NSAPs.

6.2.1. Addressing



The relationship between TSAPs, NSAPs and transport connections.

6.2.1. Addressing

- Application processes, both clients and servers, can attach themselves to a TSAP to establish a connection to a remote TSAP.
- These connections run through NSAPs on each host.
- The purpose of having TSAPs is that in some networks, each computer has a single NSAP, so some way is needed to distinguish multiple transport end points that share that NSAP.

6.2.1. Addressing

. A possible scenario for a transport connection is as follows.

1. A time of day server process on host 2 attaches itself to TSAP 1522 to wait for an incoming call.

2. An application process on host 1 wants to find out the time-of-day, so it issues a CONNECT request specifying TSAP 1208 as the source and TSAP 1522 as destination

6.2.1. Addressing

. This action ultimately results in a transport connection being established between the application process on host 1 and server 1 on host 2.

3) The application process then sends over a request for the time.

4) The time server process responds with the current time.

5) The transport connection is then released.

6.2.2. Connection Establishment

- Establishing a connection sound easy, but it actually surprisingly tricky.
- At first glance, it would seem sufficient for one transport entity to just send a CONNECTION REQUEST TPDU to the destination and wait for a CONNECTION ACCEPTED reply.

6.2.2. Connection Establishment

- The problem occurs when the network can lose, store, and duplicate packets.
- This behavior causes serious complications.
- Imagine a subnet that is so congested that acknowledgements hardly ever get back in time and each packet times out is retransmitted two or more times.

6.2.2. Connection Establishment

- Suppose that the subnet uses datagrams inside and that every packet follows a different route.
- Some of the packets might get stuck in a traffic jam inside the subnet and take a long time to arrive, that is, they are stored in the subnet and pop out much later.
- Bank problem, etc.

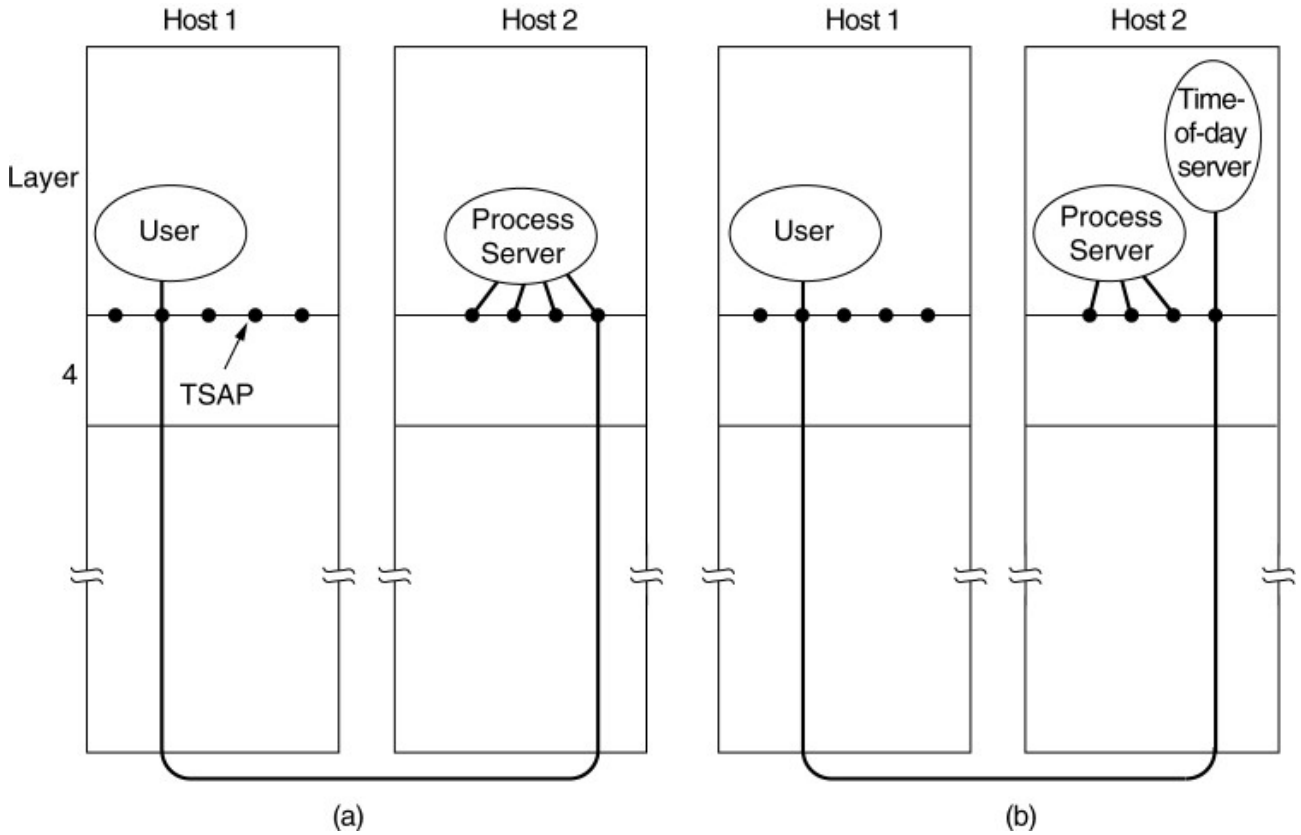
6.2.2. Connection Establishment

- The crux of the problem is the existence of delayed duplicates.
- It can be attacked in various ways.
- One way is to use throw-away transport address
- Another possibility is to give each connection a connection identifier, etc.

6.2.2. Connection Establishment

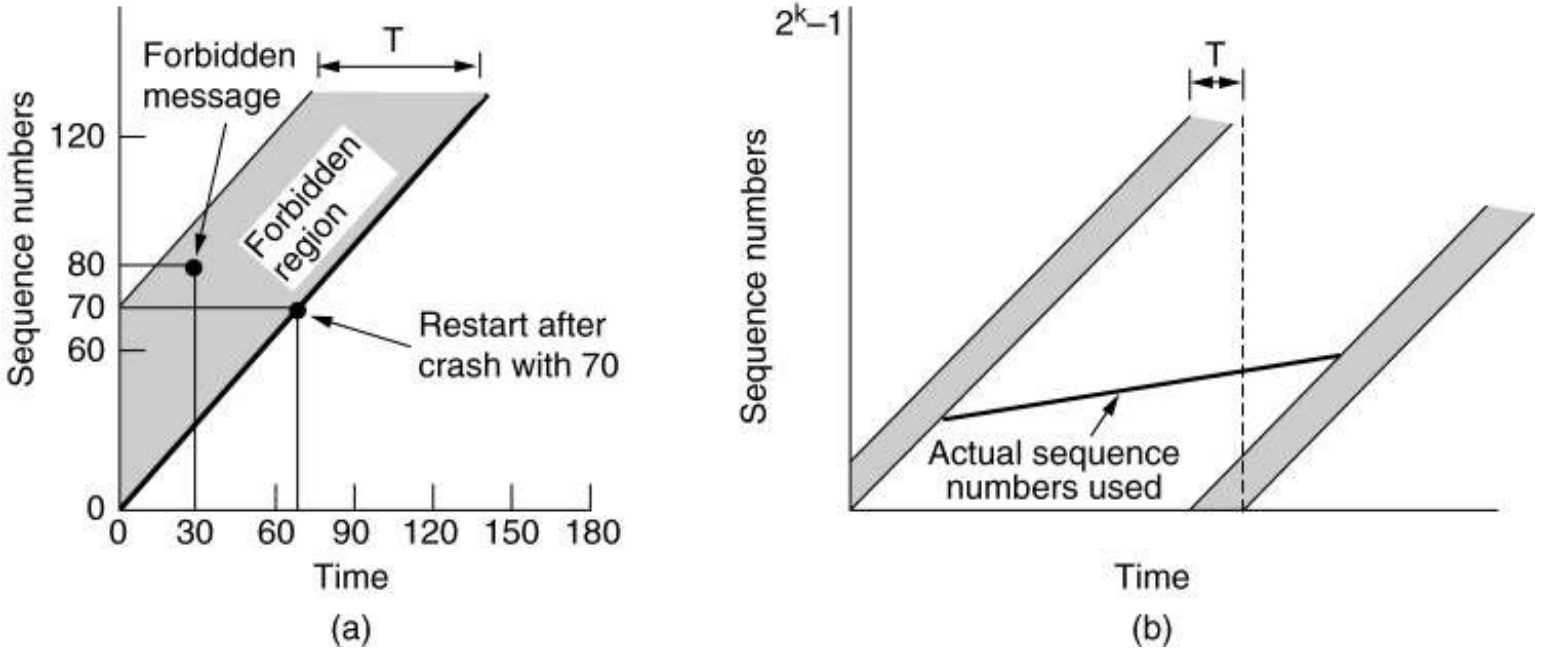
- To get around the problem of a machine losing all memory of where it was after a crash, Tomlinson proposed equipping each host with a time-of-day clock.
- The basic idea is to ensure that two identically numbered TPDUs are never outstanding at the same time.

6.2.2. Connection Establishment



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

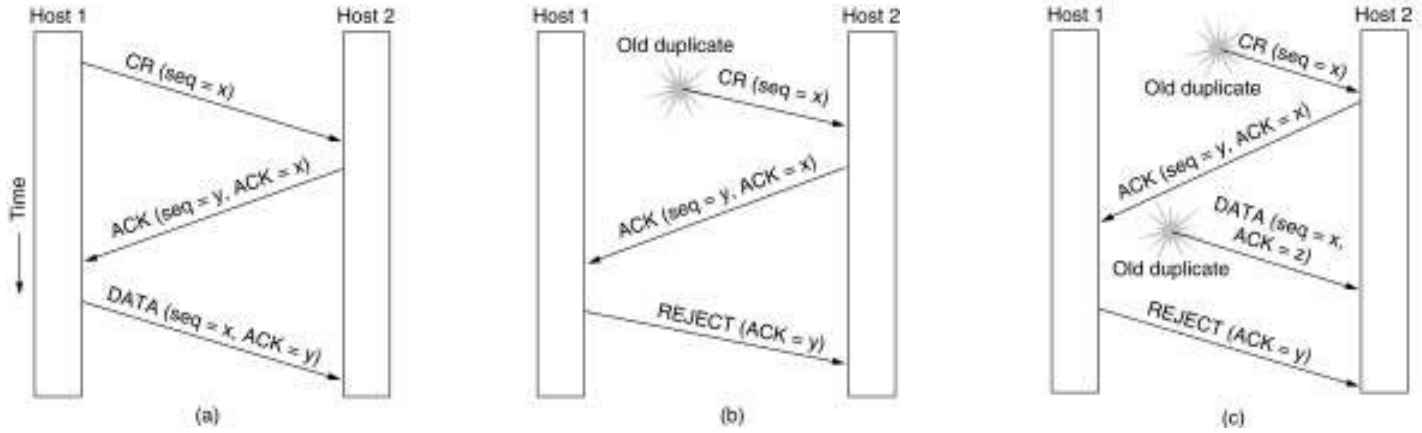
Connection Establishment (58)



(a) TPDUs may not enter the forbidden region.

(b) The resynchronization problem.

Connection Establishment (59)

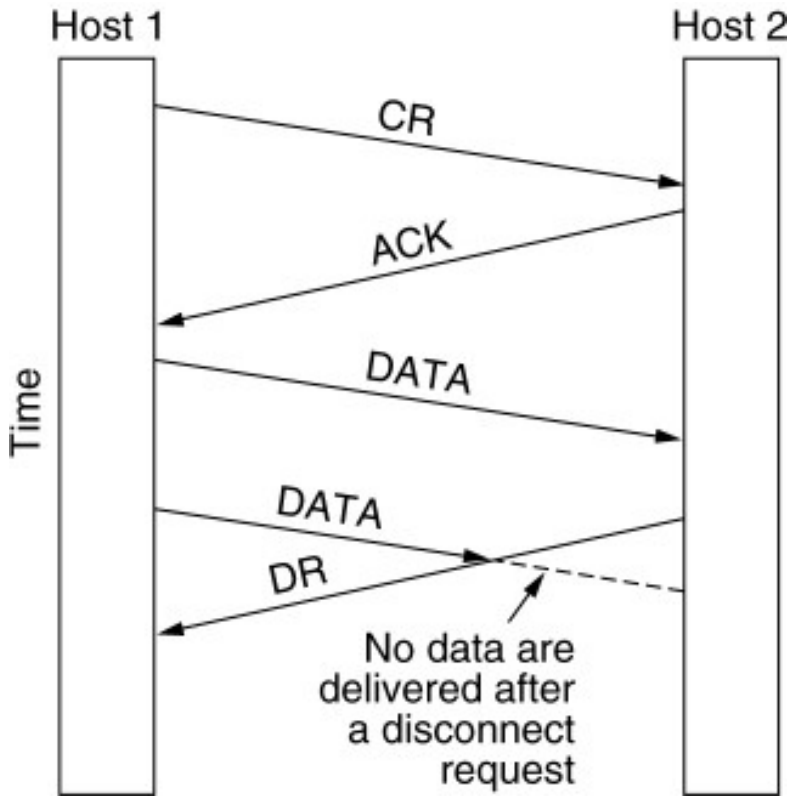


Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST.
(a) Normal operation,
(b) Old CONNECTION REQUEST appearing out of nowhere.
(c) Duplicate CONNECTION REQUEST and duplicate ACK.

6.2.3. Connection Release

- Releasing a connection is easier than establishing one.
- There are two styles of terminating a connection: asymmetric release and symmetric release.
- Asymmetric release is abrupt and may result in data loss

6.2.3. Connection Release

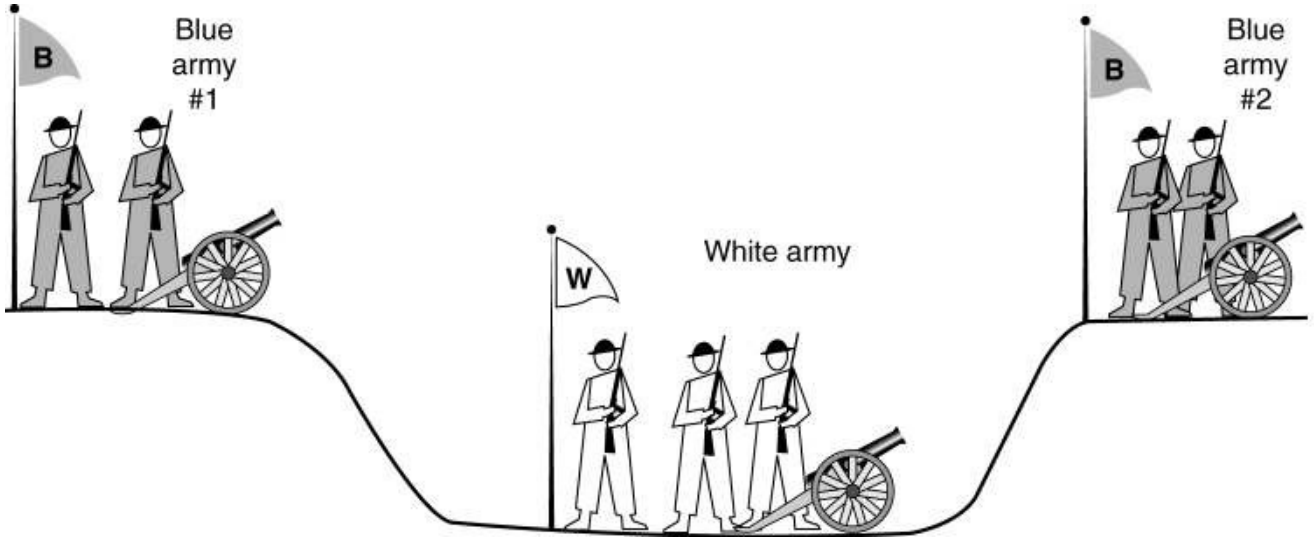


Abrupt disconnection with loss of data.

6.2.3. Connection Release

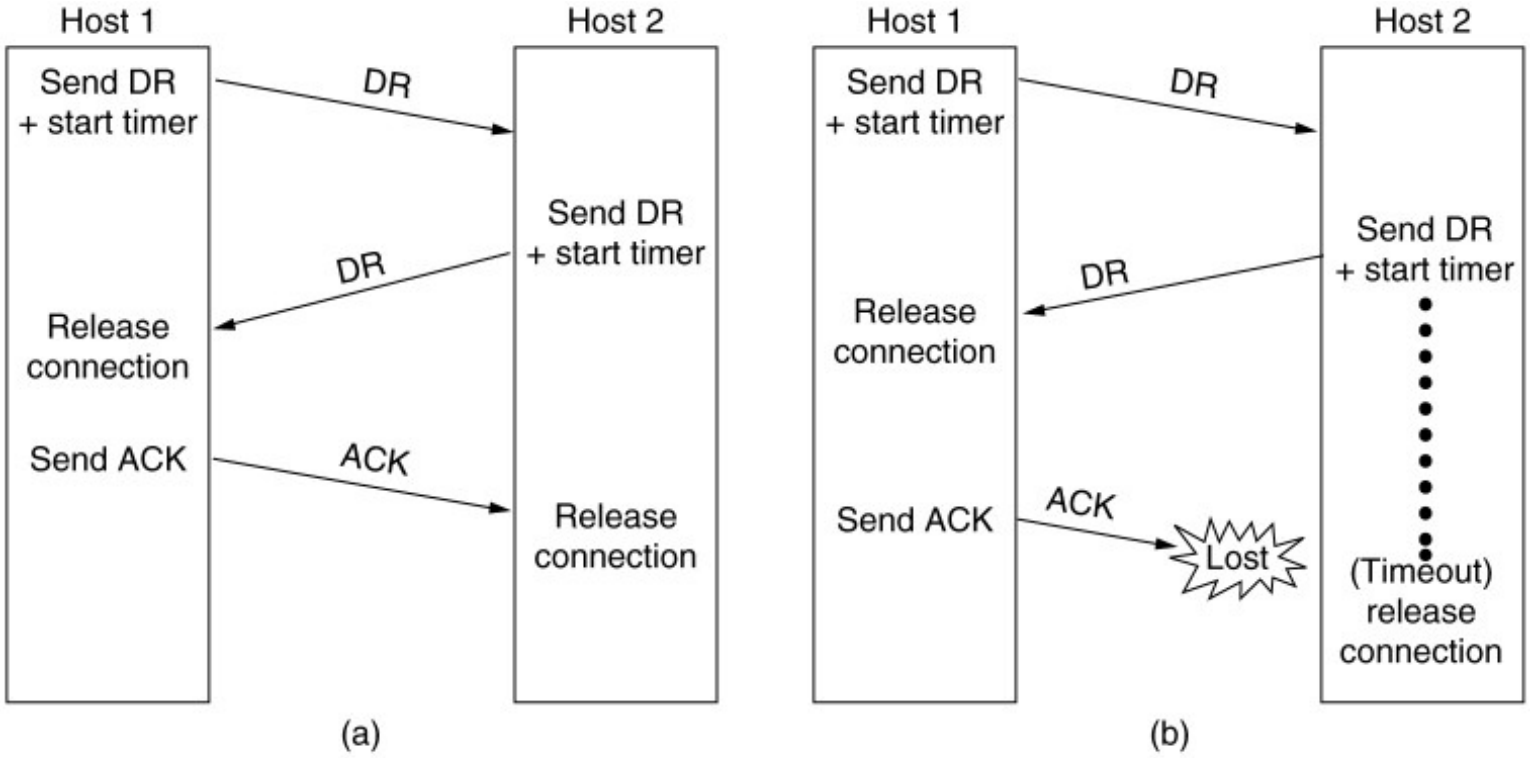
- One way to avoid data loss is to use symmetric release, in which each direction is released independently of the other one.
- One can envision a protocol in which host 1 says: I am done. Are you done too? If host 2 responds: I am done too. Goodbye, the connection can be safely released.
- Unfortunately, this protocol does not always work.

Connection Release (63)



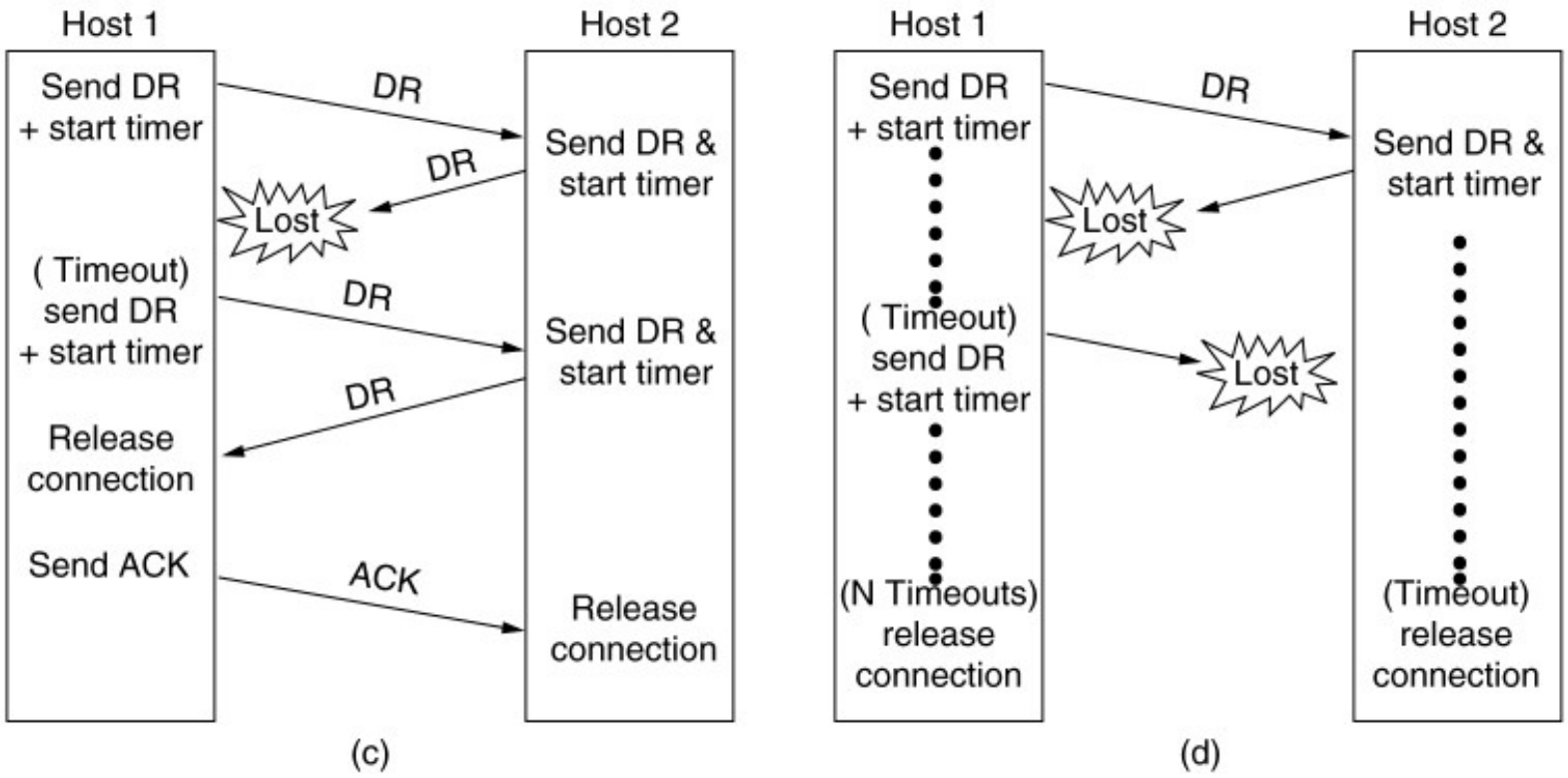
The two-army problem.

Connection Release (64)



Connection Release (65)

Four protocol scenarios for releasing a connection. (a) Normalcase of a three-way handshake. (b) Final ACK lost.


Host 1
Host 2
Send DR
+ start timer
DR
Send DR &
start timer
DR
Lost
(Timeout)
send DR
+ start timer
DR
Send DR &
start timer
DR
Release
connection
Send ACK
ACK
Release
connection
(c)
Host 1
Host 2
Send DR
+ start timer
DR
Send DR &
start timer
Lost
(Timeout)
send DR
+ start timer
Lost
(N Timeouts)
release
connection
(Timeout)
release
connection
(d)

6.2.4. Flow Control and Buffering

- Having examined connection establishment and release in some detail, let us now look at how connections are managed while they are in use.
- Flow control: In some ways the flow control problem in the transport layer is the same as in the data link layer, but in other ways it is different.

6.2.4. Flow Control and Buffering

- The main difference is that a router usually has relatively few lines, whereas a host may have numerous connections.
- This difference makes it impractical to implement the data link buffering strategy in the transport layer.

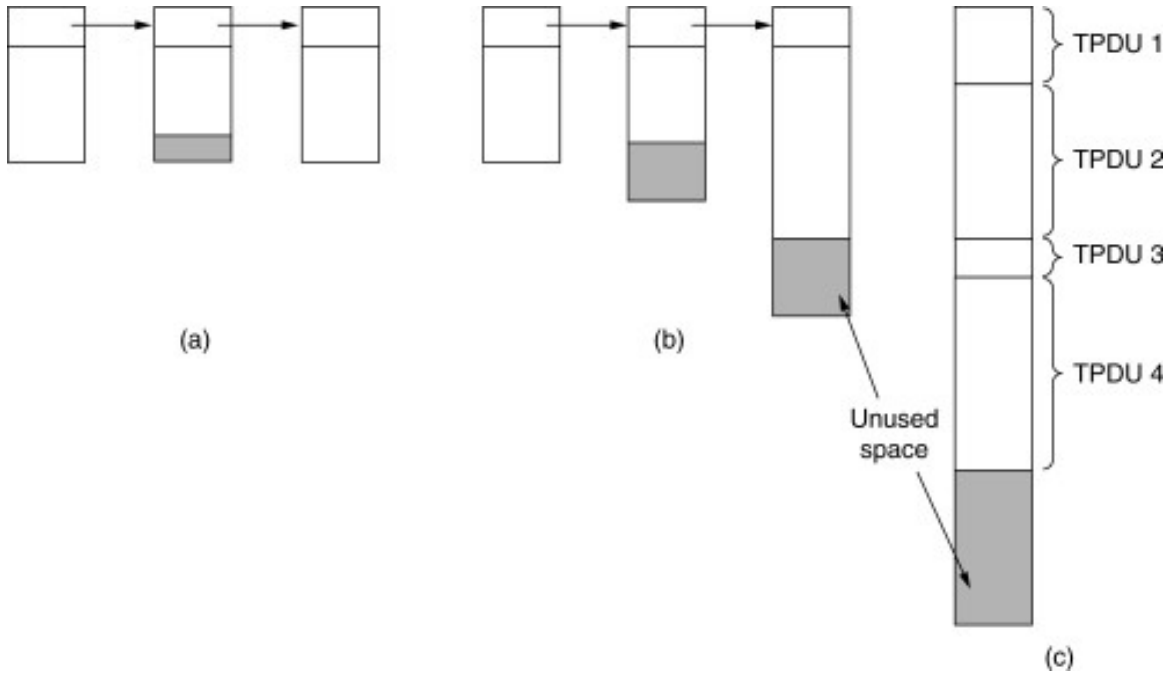
6.2.4. Flow Control and Buffering

- If the network service is unreliable, the sender must buffer all TPDUs sent.
- However, with reliable network service, other trade-off become possible.
- If the sender knows that the receiver always has buffer size, it need not retain copies of the TPDUs it sends.

6.2.4. Flow Control and Buffering

- However, if the receiver cannot guarantee that every incoming TPDU will be accepted, the sender will have to buffer anyway.
- Even if the receiver has agreed to do the buffering, there still remains the question of the buffer size.

6.2.4. Flow Control and Buffering



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

6.2.4. Flow Control and Buffering

- For low-bandwidth bursty traffic, it is better to buffer at the sender, and for high bandwidth smooth traffic, it is better to buffer at the receiver.
- Dynamic buffer management: the sender requests a certain number of buffers, based on its perceived needs. The receiver then grants as many of these as it can afford.

Flow Control and Buffering (2)

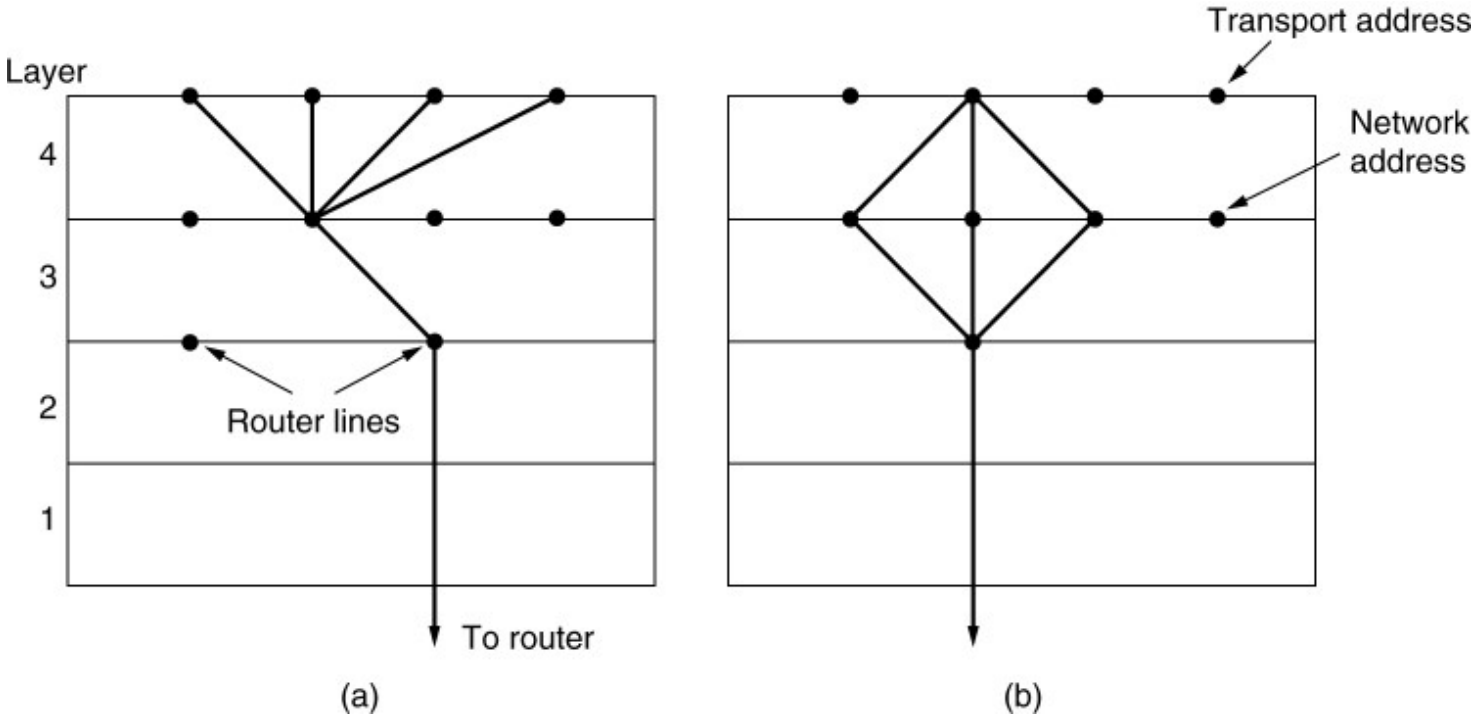
	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	•••	Message lost but A thinks it has 1 left
6	←	<ack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

6.2.5. Multiplexing

- Multiplexing several conversations onto connections, virtual circuits, and physical links plays a role in several layers of the network architecture.
- In the transport layer the need for multiplexing can arise in a number of ways.
- For example, if only one network address is available on a host, all transport connections on that machine have to use it (upward).

6.2.5. Multiplexing



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

6.2.5. Multiplexing

- Suppose that a subnet uses virtual circuits internally and imposes a maximum data rate on each one.
- If a user needs more bandwidth than one virtual circuit can provide, a way out is to open multiple network connections and distribute the traffic among them on a round-robin basis (downward).

6.2.6. Crash Recovery

- If hosts and routers are subject to crashes, recovery from these crashes becomes an issue.
- If the transport entity is entirely within the hosts, recovery from network and router crashes is straightforward.
- If the network layer provides datagram service, the transport entities expect lost TPDUs all the time and know how to cope with them.

6.2.6. Crash Recovery

- If the network layer provides connection-oriented service, then loss of a virtual circuit is handled by establishing a new one and then probing the remote transport entity to ask it which TPDUs it has received and which ones it has not received. The latter ones can be retransmitted.
- A more troublesome problem is how to recover from host crashes.

6.2.6. 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.

6.3. A Simple Transport Protocol

- Even when the network layer is completely reliable, the transport layer has plenty of work to do.
- It must handle all the service primitives, manage connections and timers.

6.3. A Simple Transport Protocol

- To make the ideas discussed so far more concrete, in this section we will study an example transport layer in detail.
- The abstract service primitives we will use are the connection-oriented primitives .

6.3. A Simple Transport Protocol

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

6.3.1. The Example Service Primitives

- The first problem is how to express these transport primitives concretely

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.

6.3.1. The Example Service Primitives

- LISTEN:
- When a process wants to be able to accept incoming calls, it calls *listen*, specifying a particular TSAP to listen to.
- The process then blocks until some remote process attempts to establish a connection to its TSAP.
- Note that this model is highly asymmetric.

6.3.1. The Example Service Primitives

- CONNECT:
- There is a library procedure *connect*
- It can be called with appropriate parameters necessary to establish a connection
- The parameters are local and remote TSAPs
- During call, caller is blocked while transport entity tries to set up the connection.
- If connection succeeds, caller is unblocked and can start transmitting data.

6.3.1. The Example Service Primitives

- DISCONNECT:
- To release a connection, we will use a procedure *disconnect.*
- When both sides have disconnected, the connection is released.
- In other words, we are using a symmetric disconnection model.

6.3.1. The Example Service Primitives

- SEND and RECEIVE
- Sending is active but receiving is passive
- An active call *send* that transmits data and a passive call *receive* that blocks until a TPDU arrives.

6.3.1. The Example Service Primitives

- Our concrete service definition therefore consists of five primitives:
- CONNECT, LISTEN, DISCONNECT, SEND, and RECEIVE
- Each primitive corresponds exactly to a library procedure that executes the primitive.

6.3.1. The Example Service Primitives

- The parameters for the service primitives and library procedures are as following:
- connum=LISTEN(local)
- connum=CONNECT(local, remote)
- status=SEND(connum, buffer, bytes)
- status=RECEIVE(connum, buffer, bytes)
- status=DISCONNECT(connum)

6.3.2. 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.

6.3.2. 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.

```
#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 Transport Layer

THE TRANSPORT LAYER'S TASK IS

TO PROVIDE RELIABLE, COST-EFFECTIVE DATA TRANSPORT

FROM SOURCE MACHINE TO DESTINATION MACHINE, INDEPENDENTLY OF THE PHYSICAL NETWORK OR NERWORKS CURRENTLY IN USE.

6.1. The Transport Service

6.2. Elements of Transport Protocols

6.3. A Simple Transport Protocol

6.4. The Internet Transport Protocols: UDP

6.5. The Internet Transport Protocols: TCP

6.6. Performance Issue

6.4. The Internet Transport Protocols: UDP

- The Internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one.
- UDP is the connectionless protocol.
- TCP is the connection-oriented protocol.

6.4. The Internet Transport Protocols: UDP (User Datagram Protocol)

- UDP is basically just IP with a short header added
- UDP is a connectionless protocol that is mainly a wrapper for IP packets with the additional feature of multiplexing and demultiplexing multiple processes using a single IP address.
- UDP can be used for client-server interactions, for example, using RPC (Remote Procedure Call).
- UDP can also be used for building real-time protocols such as RTP (Real-Time Transport

Protocol).

6.4. The Internet Transport Protocols: UDP

- Introduction to UDP
- Remote Procedure Call (RPC)
- The Real-Time Transport Protocol (RTP)

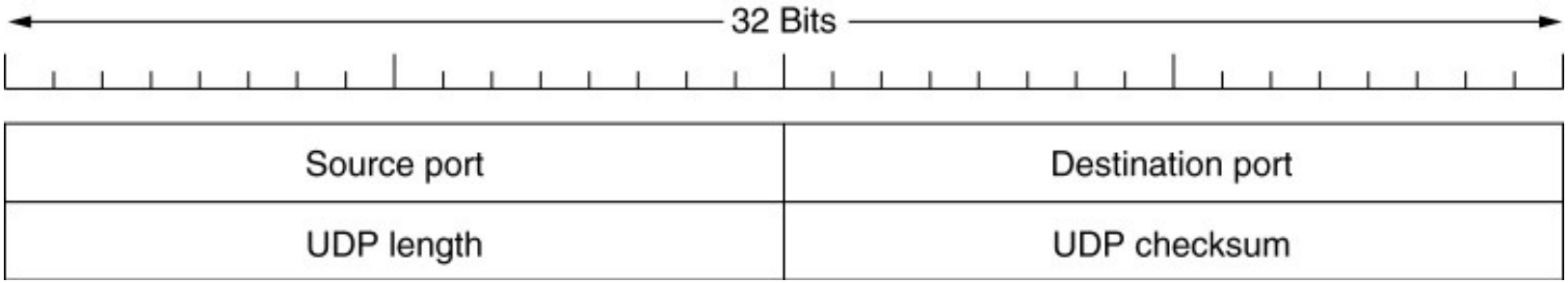
6.4.1. Introduction to UDP

- The Internet protocol suite supports a connectionless transport protocol, UDP (User Datagram Protocol).
- UDP provides a way for applications to send encapsulated IP datagrams and send them without having to establish a connection.

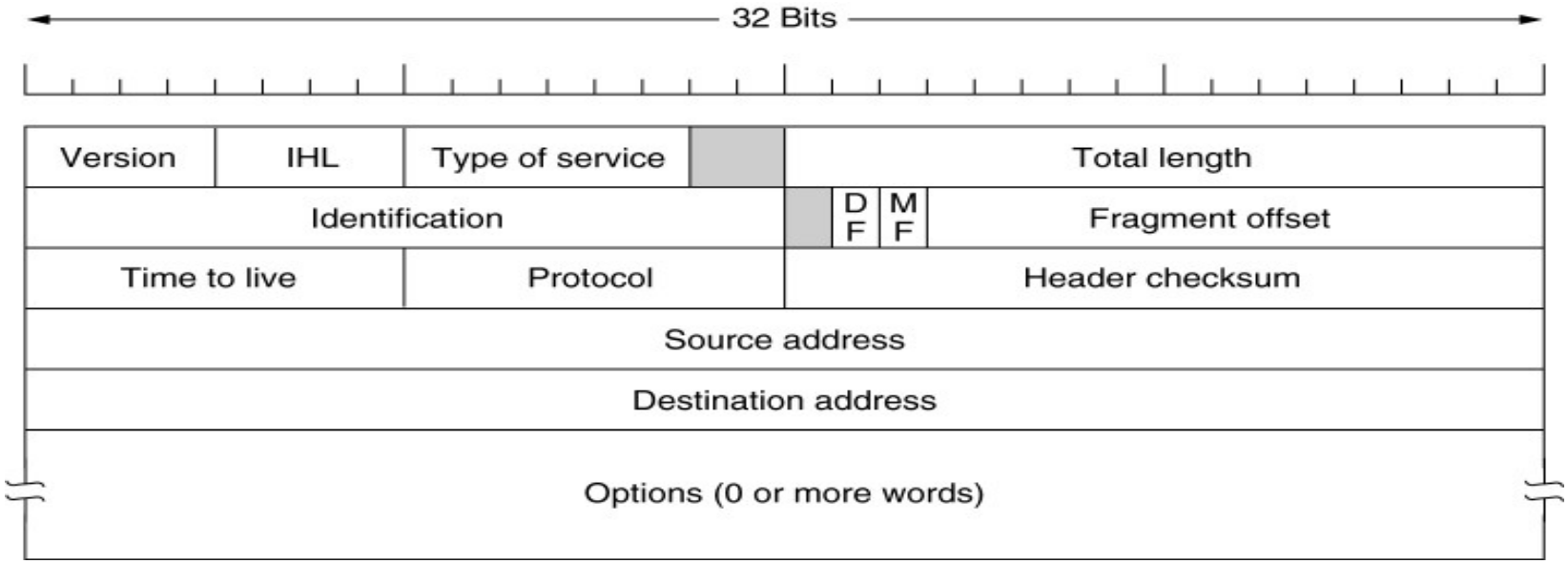
6.4.1. Introduction to UDP

- UDP transmits segments consisting of an 8-byte header followed by the payload.
- The two ports serve to identify the end points within the source and destination machines.
- When a UDP packet arrives, its payload is handed to the process attached to the destination port.

6.4.1. Introduction to UDP



The UDP header.



The IPv4 (Internet Protocol) header.

6.4.1. Introduction to UDP

- UDP does not do flow control, error control, or retransmission upon receipt of a bad segment.
- What UDP does do is provide an interface to the IP protocol with the added feature of demultiplexing multiple processes using the ports.

6.4.1. Introduction to UDP

- One area where UDP is especially useful is in client-server situations.
- Often, the client sends a short request to the server and expects a short reply back.
- If either the request or reply is lost, the client can just time out and try again.
- An application that uses UDP this way is DNS (Domain Name System)

6.4.1. Introduction to UDP

- In brief, a program that needs to look up the IP address of some host name, for example, www.cs.berkeley.edu, can send a UDP packet containing the host name to a DNS server.
- The server replies with a UDP packet containing the host's IP address.
- No setup is needed in advance and no release is needed afterward.
- Just two messages go over the network.

6.4.2. Remote Procedure Call

- In a certain sense, sending a message to a remote host and getting a reply back is a lot like making a function call in programming language.
- In both cases you start with one or more parameters and you get back a result.

6.4.2. Remote Procedure Call

- This observation has led people to try to arrange request-reply interaction on networks to be cast in the form of procedure calls.
- Such an arrangement makes network applications much easier to program and more familiar to deal with.

6.4.2. Remote Procedure Call

- For example, just imagine a procedure named *get_IP_address(host_name)* that works by sending a UDP packet to a DNS server and waiting for the reply, timing out and trying again if one is not forthcoming quickly enough.
- In this way, all the details of networking can be hidden from the programmer.

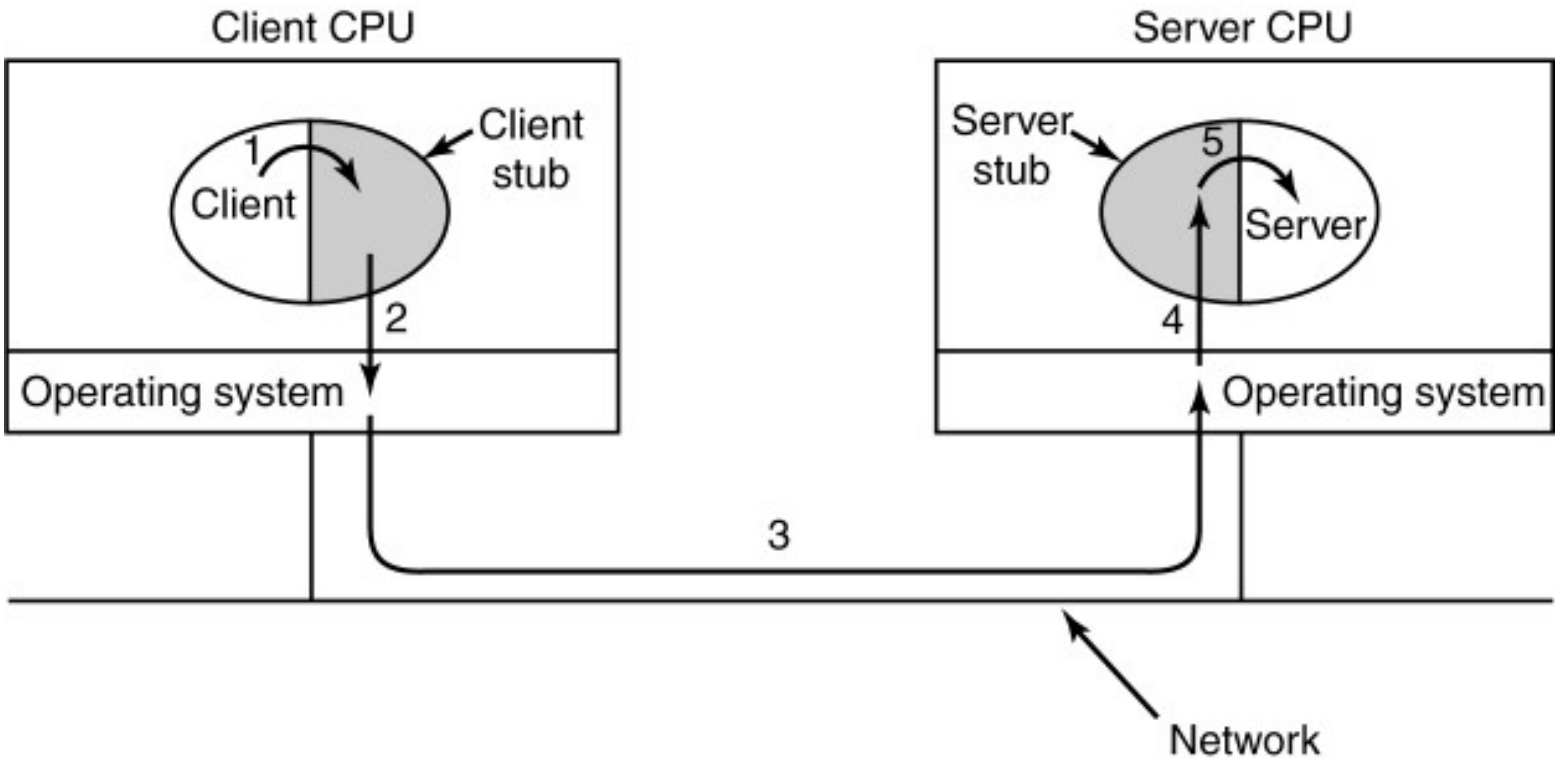
6.4.2. Remote Procedure Call

- Birrell and Nelson: programs can call procedures located on remote hosts.
- Information can be transported from the caller to the called in the parameters and can come back in the procedure result. No message passing is visible to the programmer.

6.4.2. Remote Procedure Call

- This technique is known as RPC (Remote Procedure Call) and has become the basis for many networking applications.
- Traditionally, the calling procedure is known as the client and called procedure is known as the server.

6.4.2. Remote Procedure Call



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

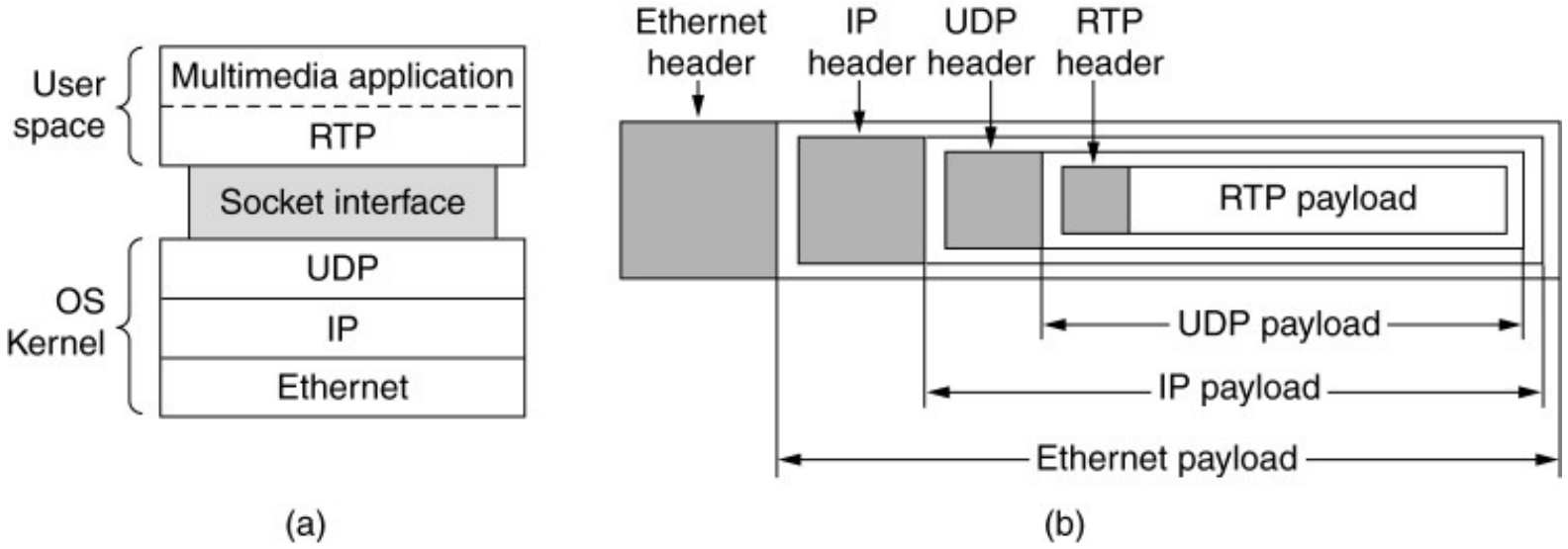
6.4.3. The Real-Time Transport Protocol

- Client-server RPC is one area in which UDP is widely used.
- Another one is real-time multimedia application.
- In particular, as Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on-demand,…

6.4.3. The Real-Time Transport Protocol

- People discovered that each application was reinventing more or less the same real-time transport protocol.
- It gradually became clear that having a generic real-time transport protocol for multiple application would be a good idea.
- Thus was RTP (Real-Time Transport Protocol) born.

6.4.3. The Real-Time Transport Protocol



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

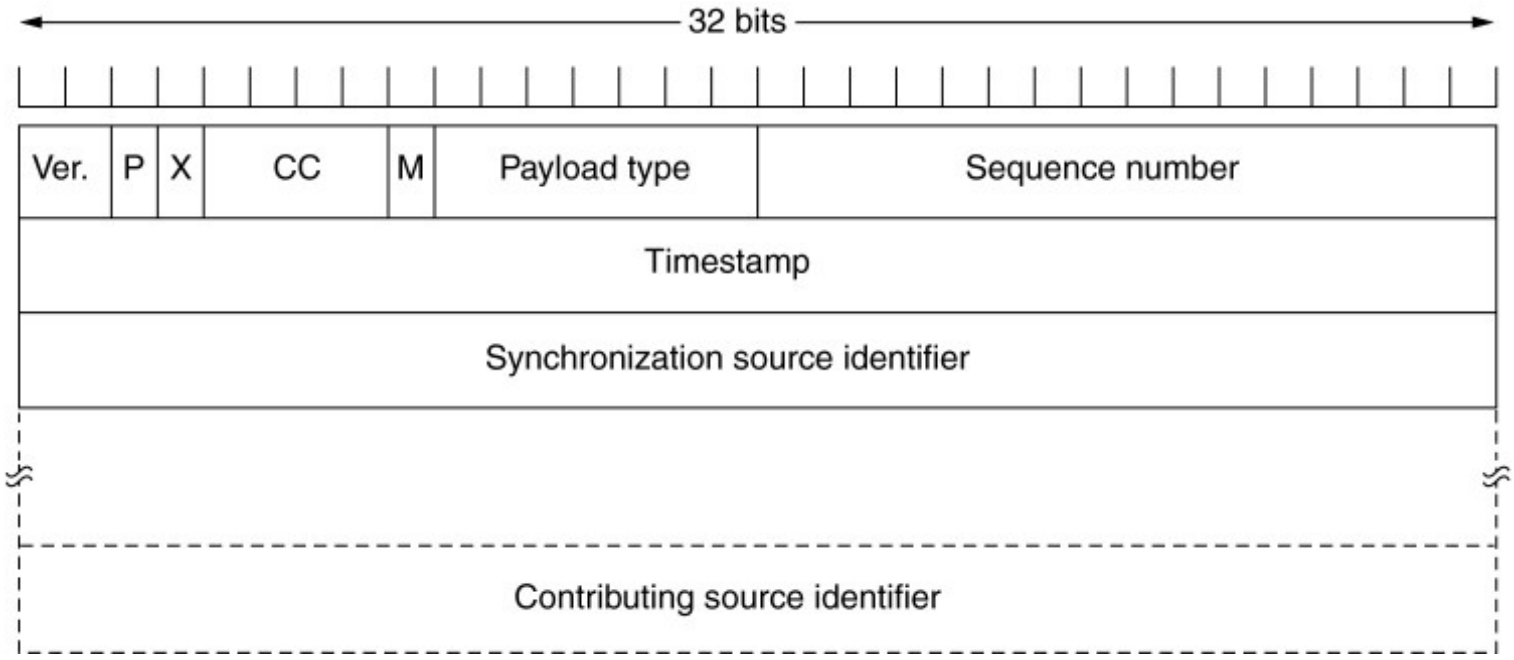
6.4.3. The Real-Time Transport Protocol

- The basic function of RTP is to multiplex several real-time data streams onto a single stream of UDP packets.
- The UDP stream can be sent to a single destination (unicasting) or to multiple destinations (multicasting).

6.4.3. The Real-Time Transport Protocol

- Each packet sent in an RTP stream is given a number one higher than its predecessor.
- RTP has no flow control, no error control, no acknowledgements, and no mechanism to request retransmission.

4.3. The Real-Time Transport Protocol



The RTP header.

6.4.3. The Real-Time Transport Protocol

- RTCP (Real-Time Transport Control Protocol)
- It handles feedback, synchronization, and the user interface but does not transport any data.
- The first function can be used to provide feedback on delay, jitter, bandwidth, congestion, and other network properties to the sources.

6.4.3. The Real-Time Transport Protocol

- RTCP also handles interstream synchronization. The problem is that different streams may use different clocks, with different granularities and different drift rates. RTCP can be used to keep them in sync.

6.4.3. The Real-Time Transport Protocol

- RTCP provides a way for naming the various sources (e.g., in ASCII text). This information can be displayed on the receiver's screen to indicate who is talking at the moment.

6.5. The Internet Transport Protocols: TCP

- UDP is a simple protocol and it has some niche uses, such as client-server interactions and multimedia.
- But for most Internet applications, reliable, sequenced delivery is needed.
- UDP cannot provide this, so another protocol is required.
- It is called TCP and is the main workhorse of the Internet.

6.5. The Internet Transport Protocols: TCP

- The main Internet transport protocol is TCP.
- It provides a reliable bidirectional byte stream.
- It uses a 20-byte header on all segments.
- Segments can be fragmented by routers within the Internet, so hosts must be prepared to do reassembly.

6.5. The Internet Transport Protocols: TCP

- A great deal of work has gone into optimizing TCP performance, using algorithms from Nagle, Clark, Jacobson, Karn, and others.
- Wireless links add a variety of complications to TCP.
- Transactional TCP is an extension to TCP that handles client-server interactions with a reduced number of packets.

6.5. 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

6.5.1. Introduction to TCP

- TCP - Transmission Control Protocol
- TCP was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork.

6.5.1. Introduction to TCP

- An Internetwork differs from a single network because different parts may have widely different topologies, bandwidths, delays, packet sizes, and other parameters.
- TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.

6.5.1. Introduction to TCP

- Each machine supporting TCP has a TCP transport entity.
- A TCP transport entity is a library procedure (a user process) or part of the kernel.
- In both cases, it manages TCP streams and interfaces to the IP layer.

6.5.1. Introduction to TCP

- A TCP transport entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64 KB (in practice, often 1460 data bytes in order to fit in a single Ethernet frame with the IP and TCP headers), and sends each piece as a separate IP datagram.

6.5.1 Introduction to TCP

- When datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which reconstructs the original byte streams.
- "TCP" or "TCP transport entity" is a piece of software.
- TCP protocol is a set of rules.

6.5.1. Introduction to TCP

- IP layer gives no guarantee that datagrams will be delivered properly, so it up to TCP to time out and retransmit them as need be.
- Daragrams that do arrive may well do so in wrong order; it is also up to TCP to reassemble them into messages in proper sequence.
- In short, TCP must furnish the reliability that most users want and that IP does not provide.

6.5.2. The TCP Service Model

- TCP service is obtained by both the sender and receiver creating end points, called sockets.
- Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called port.
- A port is the TCP name for a TSAP.

6.5.2. The TCP Service Model

- For TCP service to be obtained, a connection must be explicitly established between a socket on the sending machine and a socket on the receiving machine.
- A socket may be used for multiple connections at the same time.
- In other words, two or more connections may terminate at the same socket
- socket1, socket2, … at both ends

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 calls for TCP.

The TCP Service Model

Some assigned ports.

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

Port numbers below 1024 are called well-known ports and are reserved for standard services.

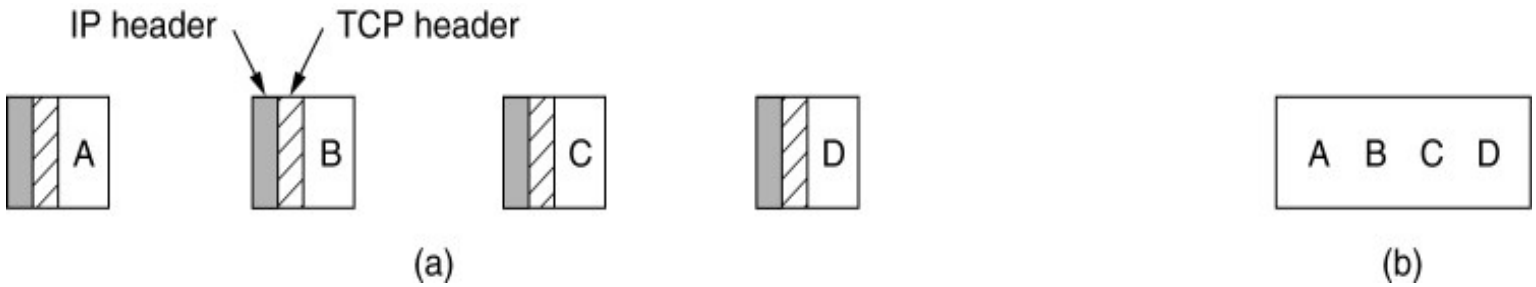
6.5.2. The TCP Service Model

- All TCP connections are full duplex and point-to-point.
- Full duplex means that traffic can go in both directions at the same time.
- Point-to-Point means that each connection has exactly two end points.
- TCP does not support multicasting or broadcasting.

6.5.2. The TCP Service Model

- A TCP connection is a byte stream, not a message stream.
- Message boundaries are not preserved end to end.
- For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as four 512-byte chunks, two 1024-byte chunks, one 2048-byte chunk, or some other way.

The TCP Service Model



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

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

6.5.2. The TCP Service Model

- urgent data:
- This feature of the TCP service causes TCP to stop accumulating data and transmit everything it has for that connection immediately.

6.5.3. The TCP Protocol

- A key feature of TCP is that every byte on a TCP connection has its own 32-bit sequence number.
- Separate 32-bit sequence numbers are used for acknowledgements and for window mechanism.

6.5.3. The TCP Protocol

- The sending and receiving TCP entities exchange data in the form of segments.
- A TCP SEGMENT consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes.
- Two limits restrict the segment size.
- First, each segment, including the TCP header, must fit in the 65,515-byte IP payload.

6.5.3. The TCP Protocol

- Second, each network has a maximum transfer unit, or MTU, and each segment must fit in the MTU.
- In practice, the MTU is generally 1500 bytes (the Ethernet payload size) and thus defines the upper bound on segment size.
- The basic protocol used by TCP entities is the sliding window protocol.

6.5.4. The TCP Segment Header

- Every segment begins with a fixed-format, 20-byte header.
- The fixed header may be followed by header options.

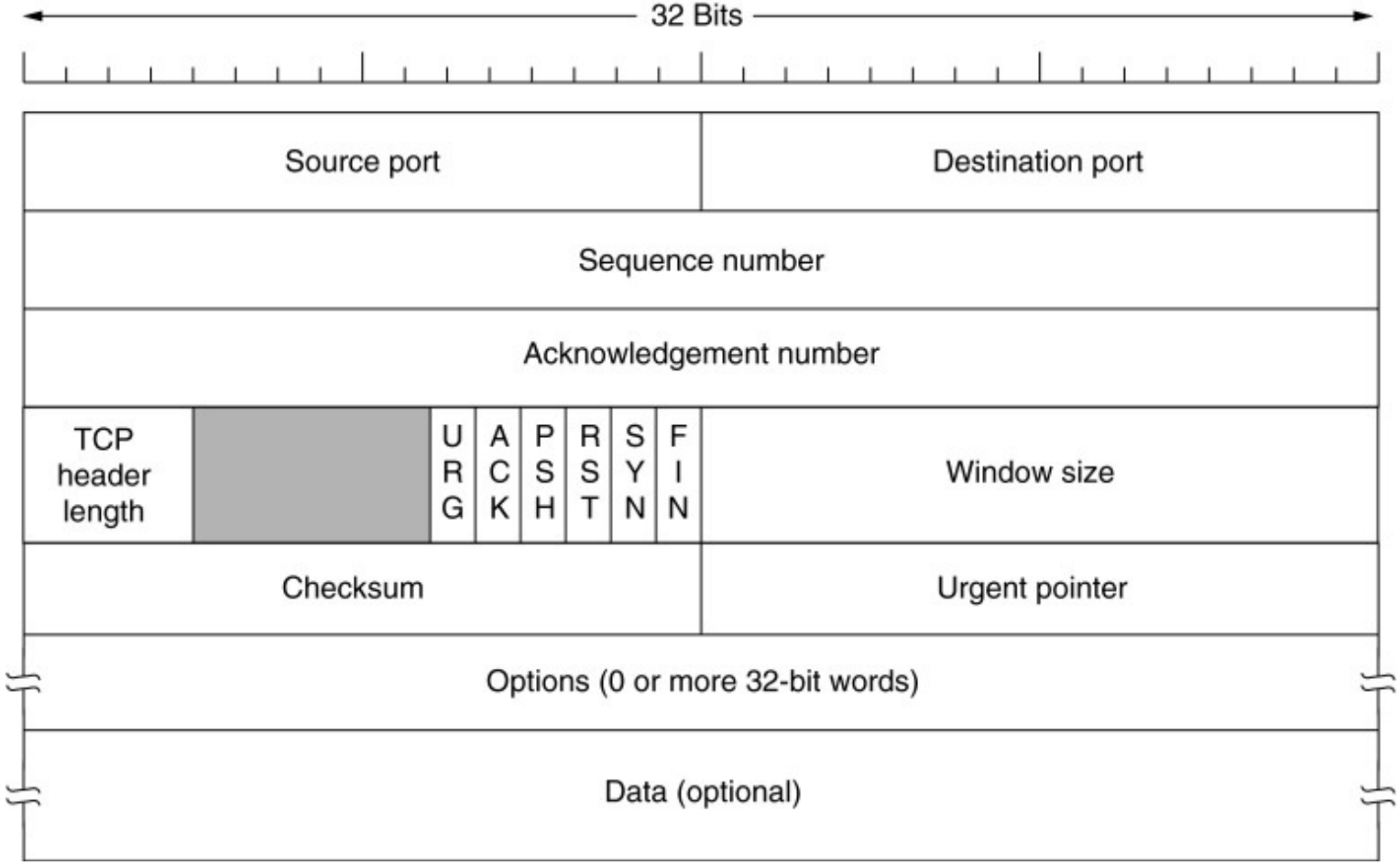
6.5.4. The TCP Segment Header

- After the options, if any, up to 65,535-20-20=65,495 data bytes may follow, where the first 20 refer to the IP header and the second to the TCP header.
- Segments without any data are legal and are commonly used for acknowledgements and control messages.

6.5.4. The TCP Segment Header

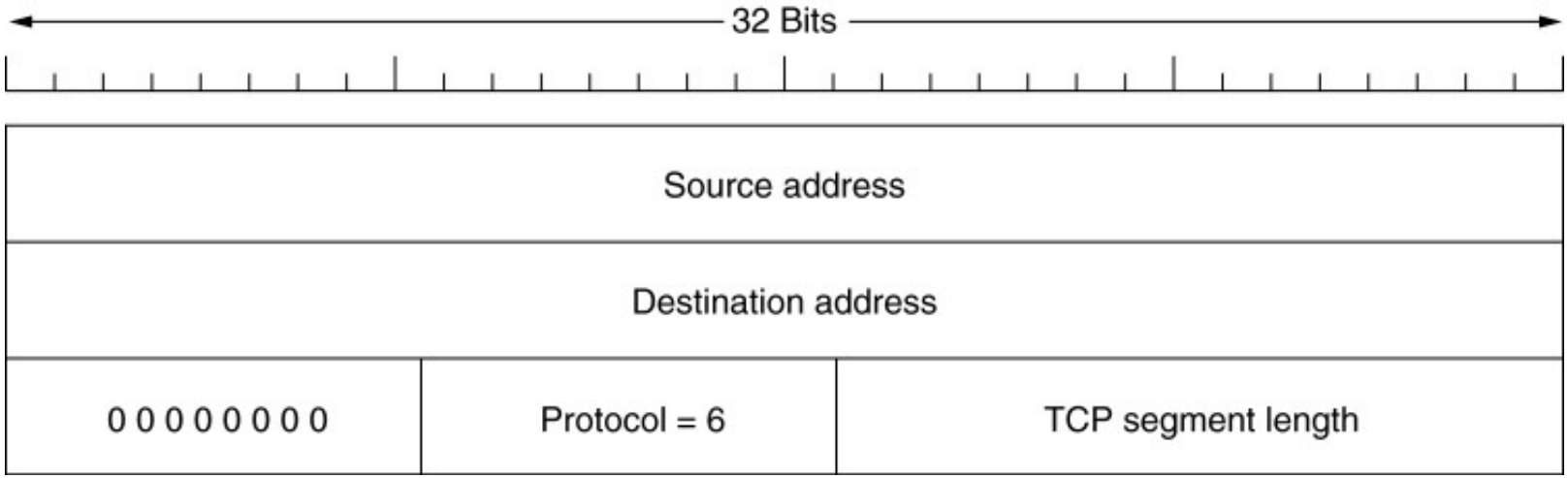
- Source port and destination port fields identify the local end points of the connection.
- The source and destination end points together identify the connection.

6.5.4.The TCP Segment Header



TCP Header.

The TCP Segment Header (2)



The pseudoheader included in the TCP checksum.

6.5.5. TCP Connection Establishment

- Connections are established in TCP by means of the three-way handshake.
- To establish a connection:
- The server passively waits for an incoming connection by executing the LISTEN AND ACCEPT primitives, either specifying a specific source or nobody in particular.

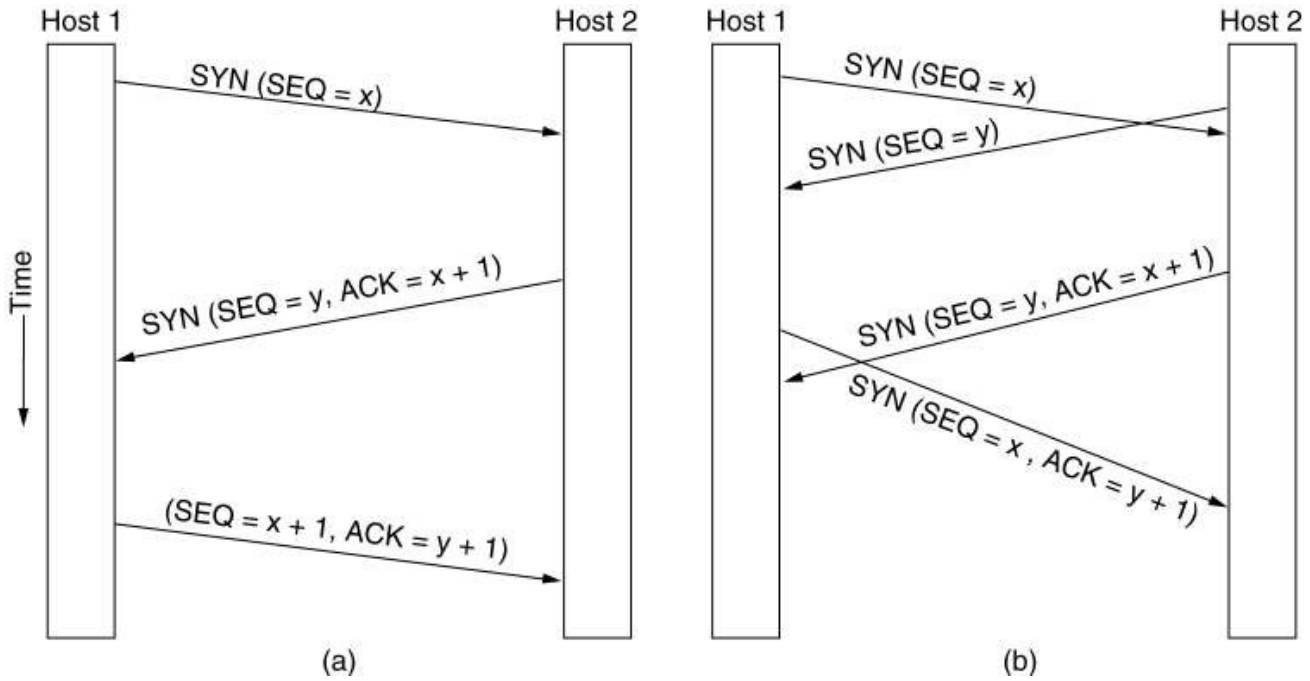
6.5.5. TCP Connection Establishment

- The client executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password). The CONNECT primitive sends a TCP segment with the SYN bit ON and ACK bit OFF and waits for a response.

6.5.5. TCP Connection Establishment

- When this segment arrives at the destination, the TCP entity there check to see if there is a process that has done a LISTEN on the port given in the Destination port field. If not, it sends a reply with the RST bit on to reject the connection.
- If some process is listening to the port, that process is given the incoming TCP segment. It can then either accept or reject connection. If it accepts, an acknowledgement segment is sent back.

6.5.5. TCP Connection Establishment



a) TCP connection establishment in the normal case.b)Call collision.

6.5.6. TCP Connection Release

- Although TCP connections are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections.
- Each simplex connection is released independently of its sibling.
- To release a connection, either party can send a TCP segment with the FIN bit set, which means that it has no more data to transmit.

6.5.6. TCP Connection Release

- When FIN is acknowledged, that direction is shut down for new data.
- Data may continue to flow indefinitely in the other direction, however.
- When both directions have been shut down, the connection is released.

6.5.6. TCP Connection Release

- Normally, four TCP segments are needed to release a connection, one FIN and one ACK for each direction.
- However, it is possible for the first ACK and the second FIN to be contained in the same segment, reducing the total count to three.

6.5.7 TCP Connection Management Modeling

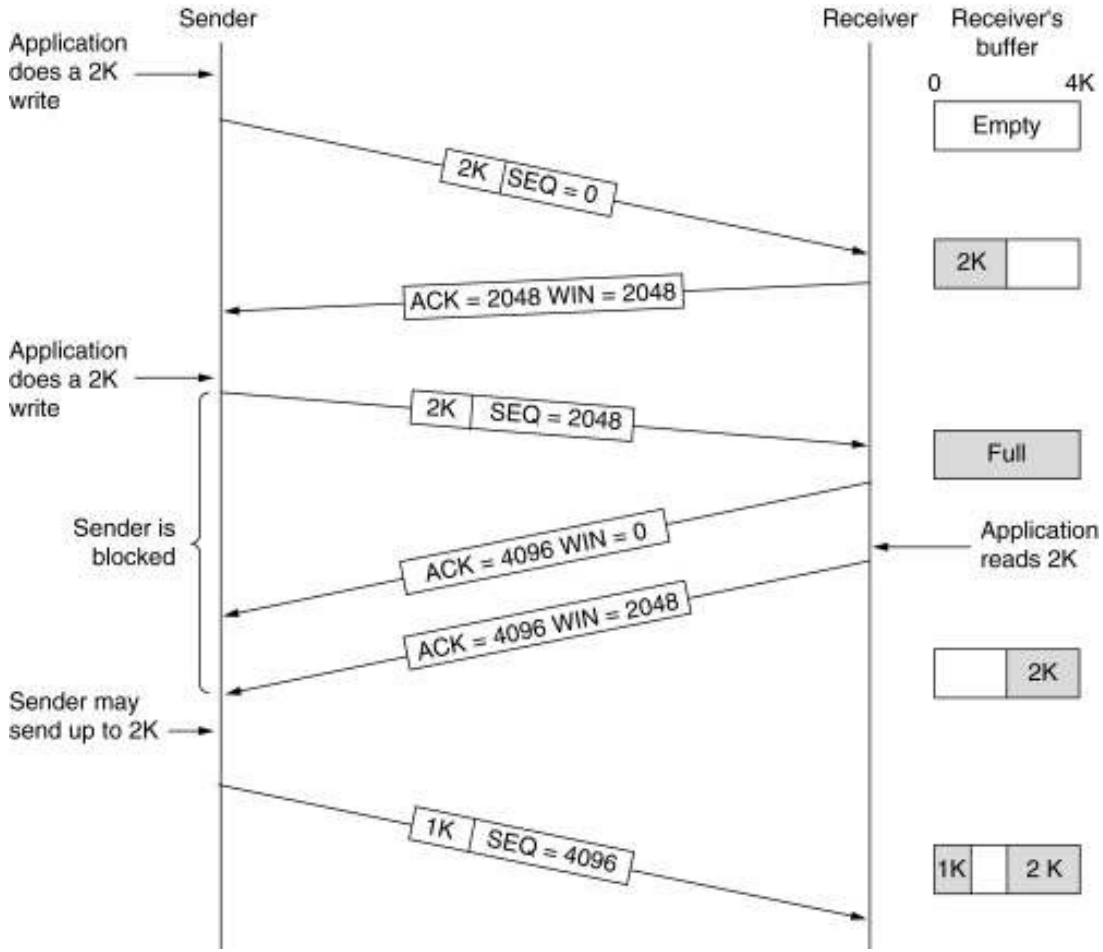
State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said 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)

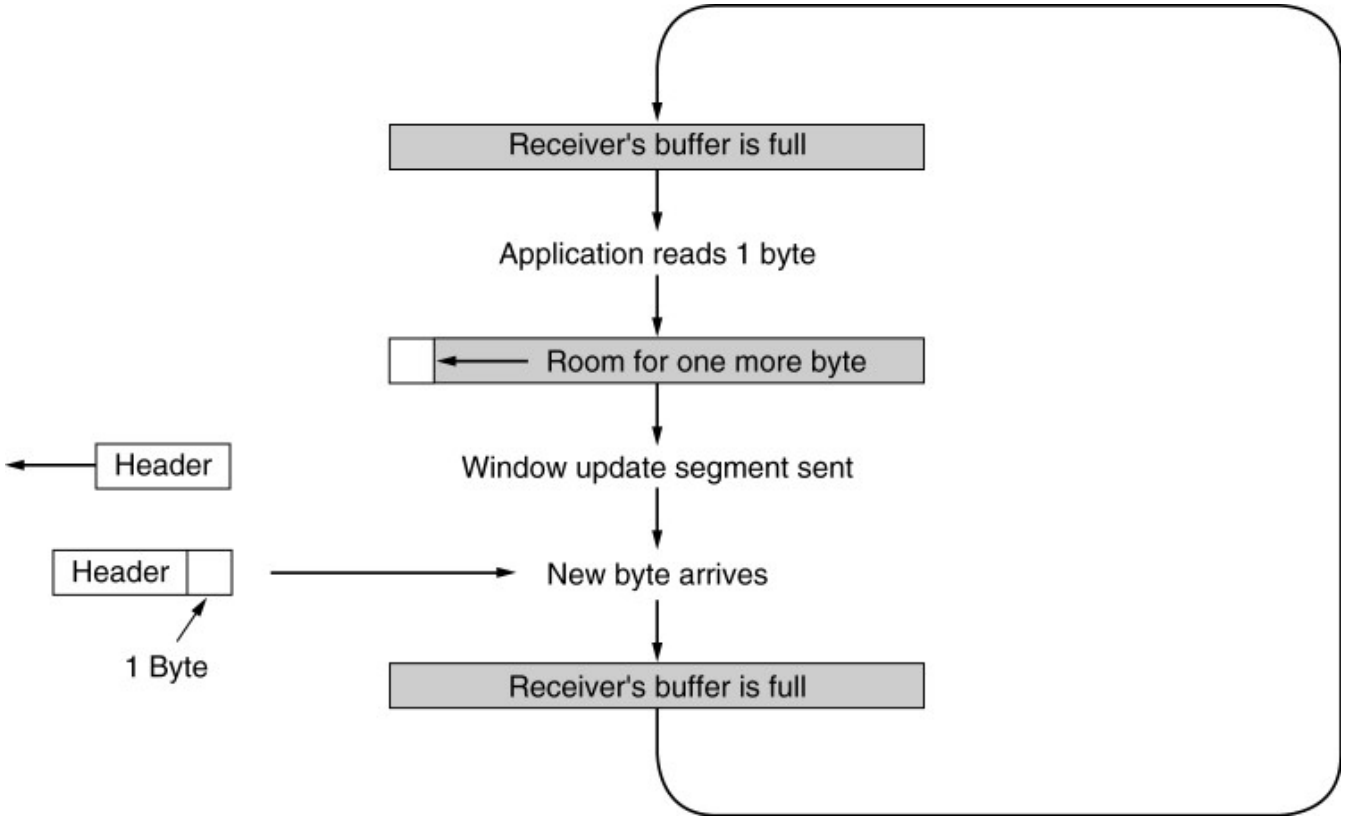
TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled by the event causing it and the action resulting from it, separated by a slash.

6.5.8. TCP Transmission Policy



Window management in TCP.

TCP Transmission Policy (2)



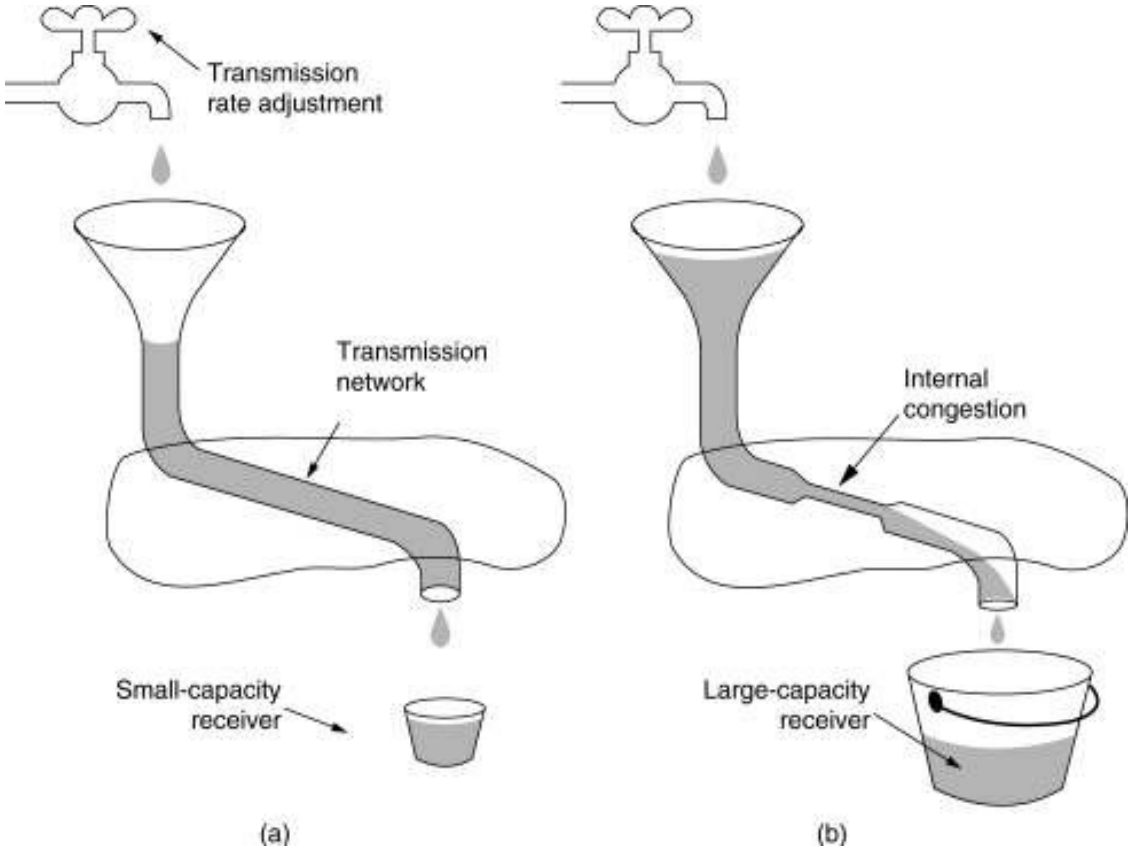
Silly window syndrome.

6.5.9. TCP Congestion Control

- When the load offered to any network is more than it can handle, congestion builds up.
- The Internet is no exception.
- The law of conservation of packets:
- To refrain from injecting a new packet into the network until an old one leaves (i.e., is delivered).
- TCP attempts to achieve this goal by

dynamically manipulating the window size.

6.5.9. TCP Congestion Control



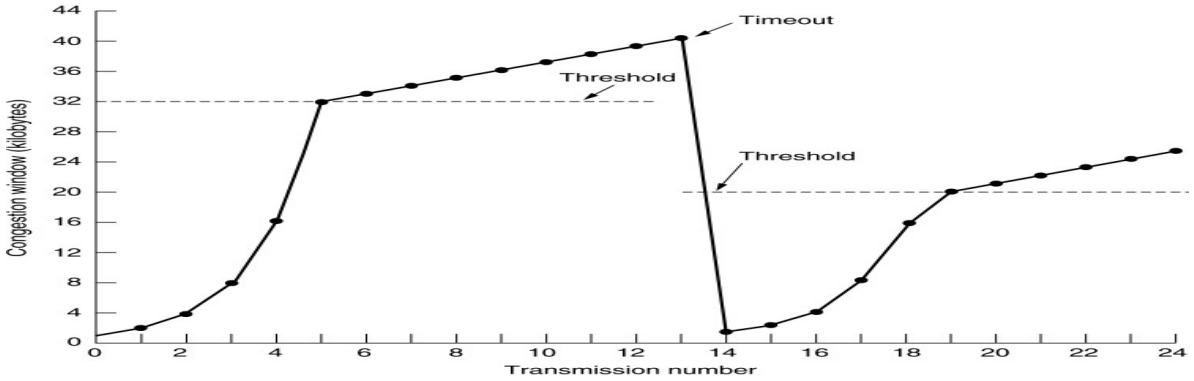
(a) A fast network feeding a low capacity receiver.

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

6.5.9. TCP Congestion Control

- Most transmission timeouts on the Internet are due to congestion.
- The Internet solution is to realize that two potential problems exist – network capacity and receiver capacity – and to deal with each of them separately.
- To do so, each sender maintains two windows: the window the receiver has granted and a second window, the congestion window.

6.5.9. TCP Congestion Control

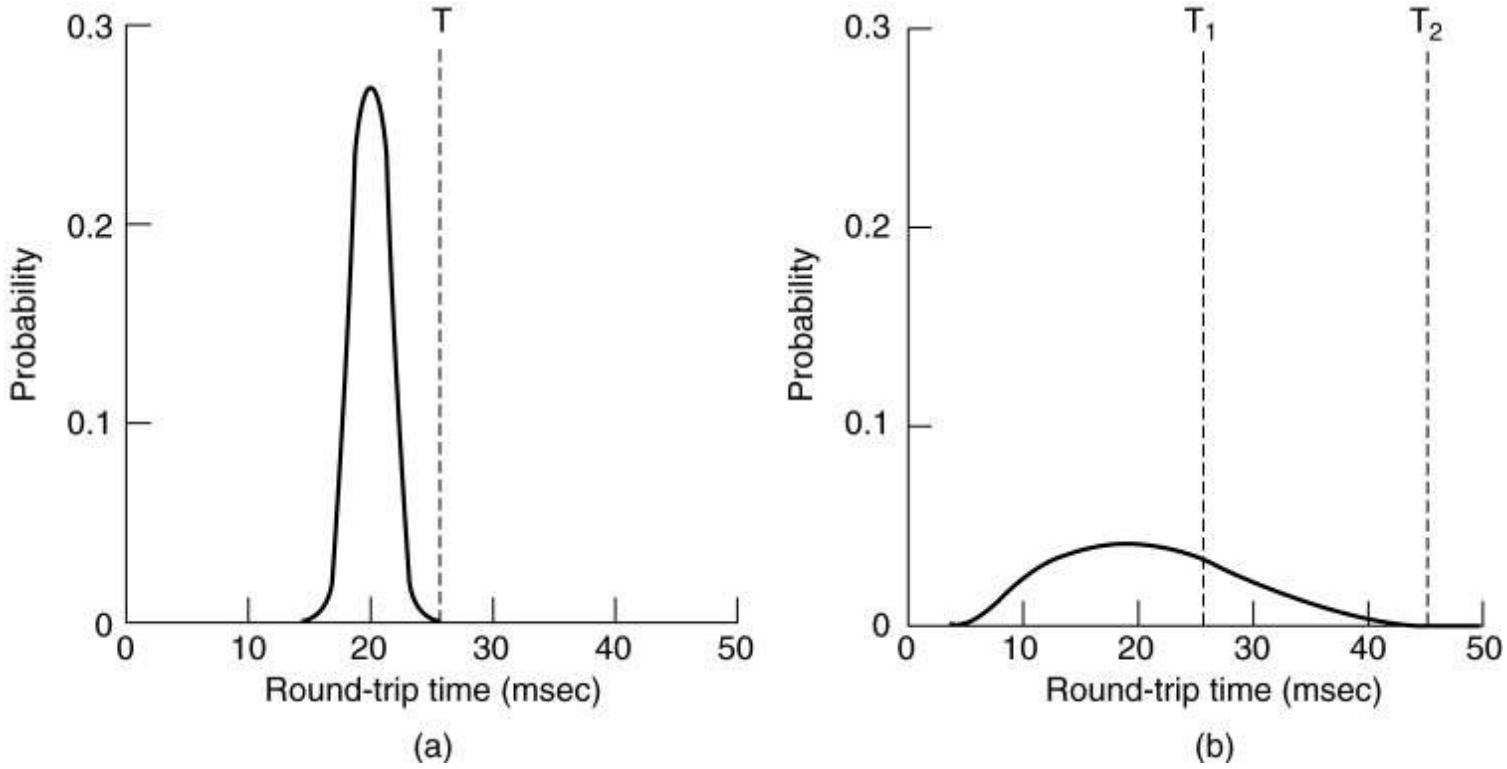


An example of the Internet congestion algorithm

6.5.10. TCP Timer Management

- TCP uses multiple timers to do its work.
- The most important of these is the retransmission timer.
- When a segment is sent, a retransmission timer is started.
- If the segment is acknowledged before the timer expires, the timer is stopped.
- If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted (and the timer started again).

6.5.10. TCP Timer Management



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

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

6.5.10. TCP Timer Management

- How long should the timeout interval be?
- The solution is to use a highly dynamic algorithm that constantly adjusts the timeout interval, based on continuous measurements of network performance.
- The algorithm generally used by TCP is due to Jacobson.
- For each connection, TCP maintains a variable, RTT, that is the best current estimate of round-trip time to destination in question.

6.5.11. Wireless TCP and UDP

- Most TCP implementations have been carefully optimized based on assumptions that are true for wired networks but that fail for wireless networks.
- The principal problem is the congestion control algorithm.
- Nearly all TCP implementation nowadays assume that timeouts are caused by congestion, not by lost packets.

6.5.11. Wireless TCP and UDP

- Consequently, when a timer goes off, TCP slow down and sends less vigorously (e.g., Jacobson's slow start algorithm)
- The idea behind this approach is to reduce the network load and thus alleviate the congestion.

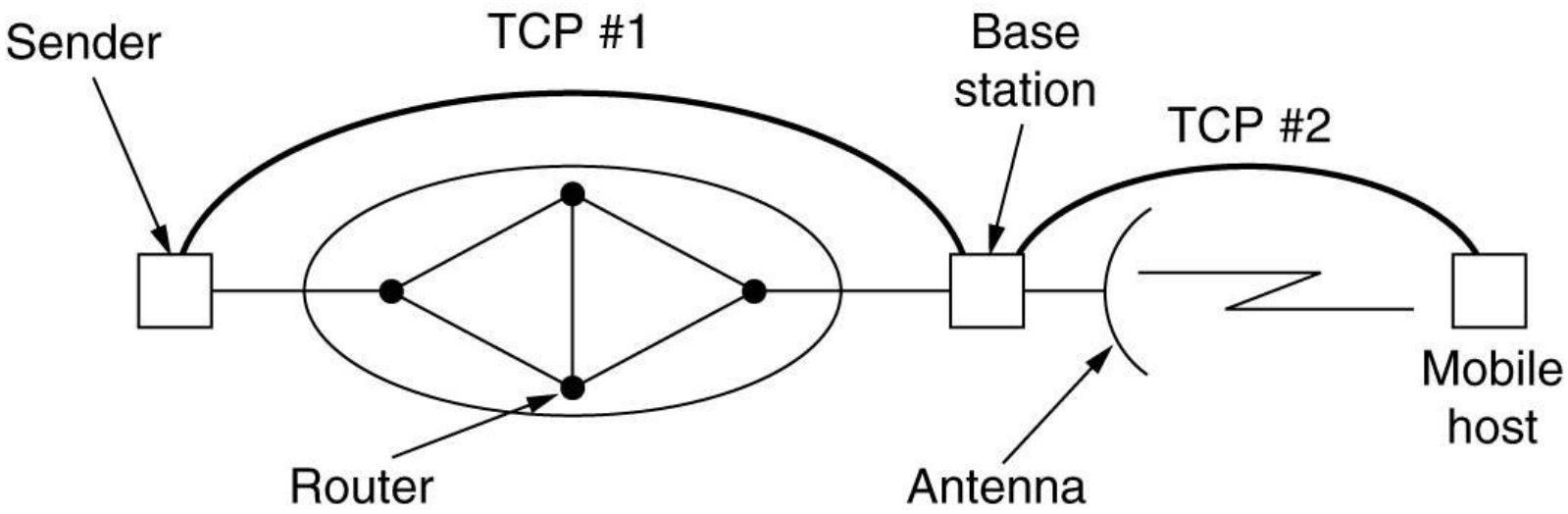
6.5.11. Wireless TCP and UDP

- Unfortunately, wireless transmission links are highly unreliable.
- They lose packets all the time.
- The proper approach to dealing with lost packets is to send them again, and as quickly as possible.
- Slowing down just makes matters worse.

6.5.11. Wireless TCP and UDP

- Indirect TCP – is to split the TCP connection into two separate connections.
- The first connection goes from the sender to the base station.
- The second one goes from the base station to the receiver.
- The base station simply copies packets between the connections in both directions.

6.5.11. Wireless TCP and UDP

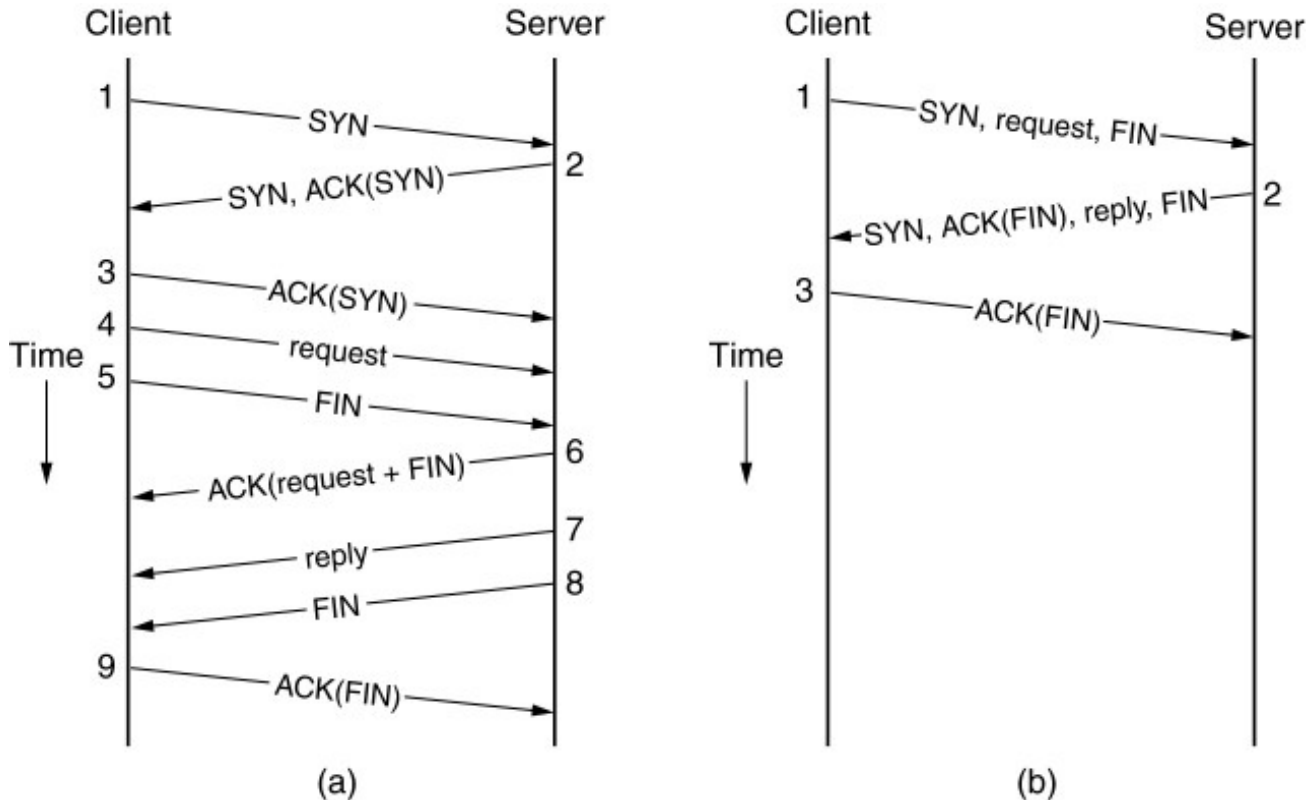


Splitting a TCP connection into two connections.

6.5.12. Transitional TCP

- The problem is the efficiency.
- The normal sequence of packets for doing an RPC over TCP consists of nine packets in the best case.
- The question quickly arises of whether there is some way to combine the efficiency of RPC using UDP with the reliability of TCP.
- ALMOST. It can be done with an experimental TCP variant called T/TCP.

6.5.12. Transitional TCP



(a) RPC using normal TCP.

(b) RPC using T/TCP.

Transitional TCP

- 1: SYN, request, FIN
  - I want to establish a connection, here is the data, and I am done.
- 2: SYN, ACK(FIN), reply, FIN:
  - I acknowledge your FIN, here is the answer, and I am done.
- 3: ACK(FIN)
  - The client then acknowledges the server's FIN and the protocol terminates in three messages.