Chapter 6

(Week 11)

The Transport Layer

ANDREW S. TANENBAUM
COMPUTER NETWORKS
FOURTH EDITION
PP. 481-524

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

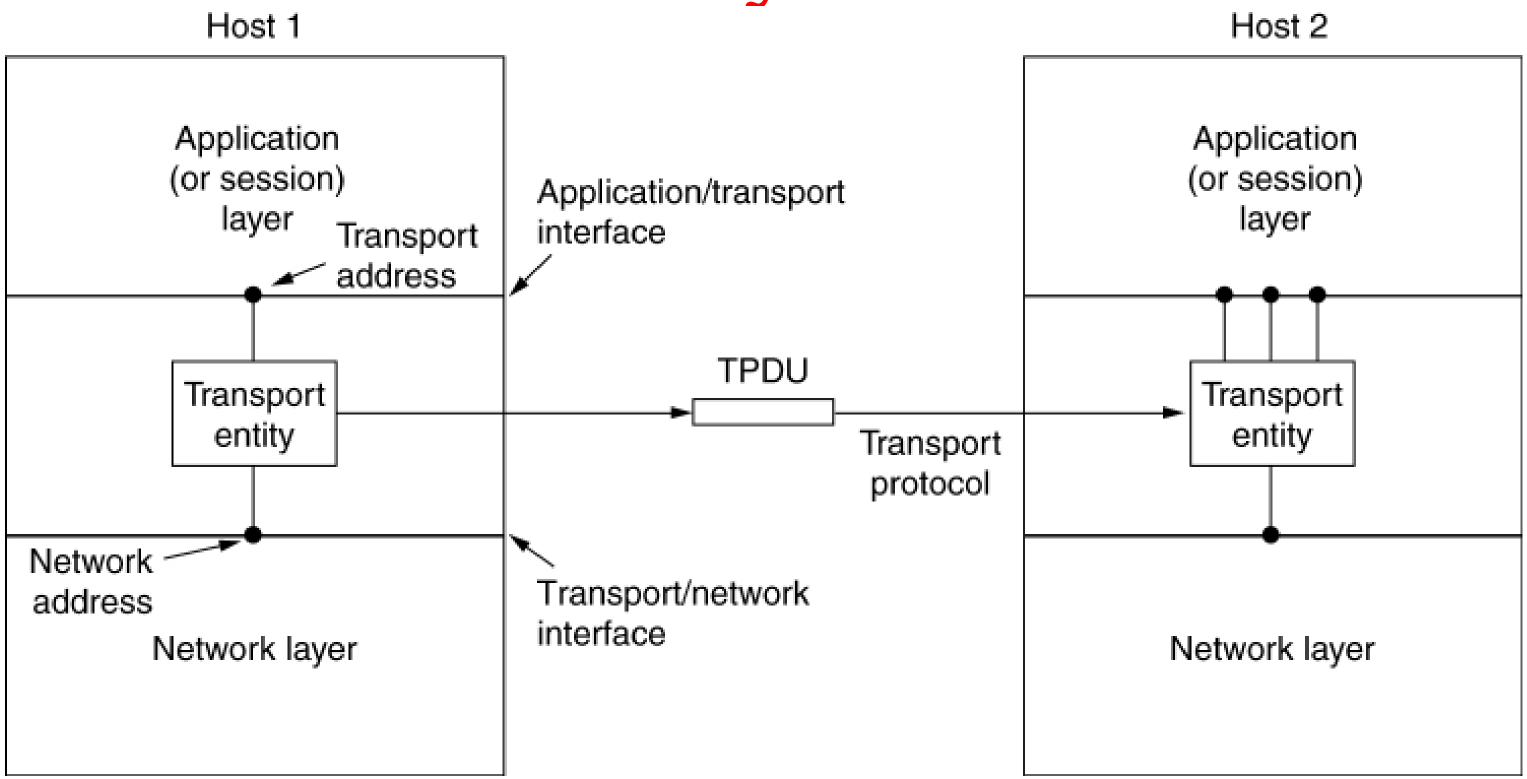
6.1.1. Services Provided to the Upper Layers

- 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.

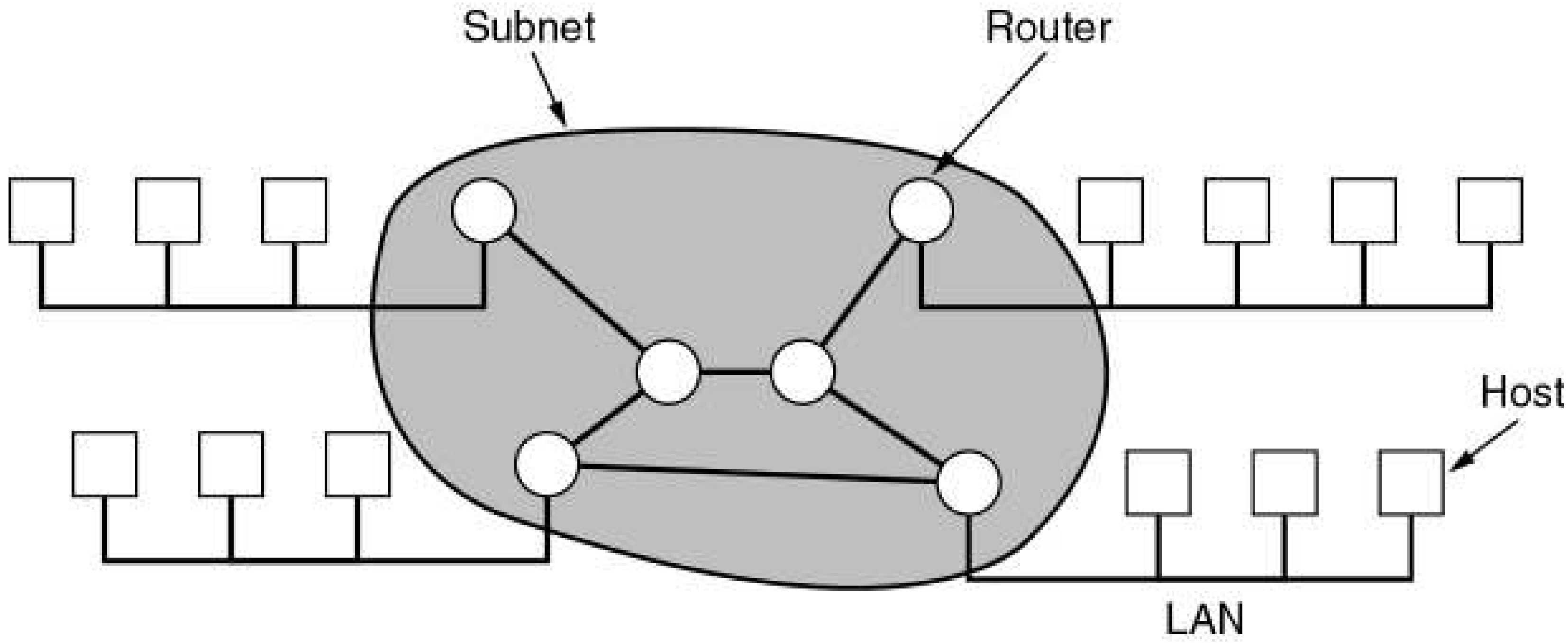
6.1.1. Services Provided to the Upper Layers

- 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.

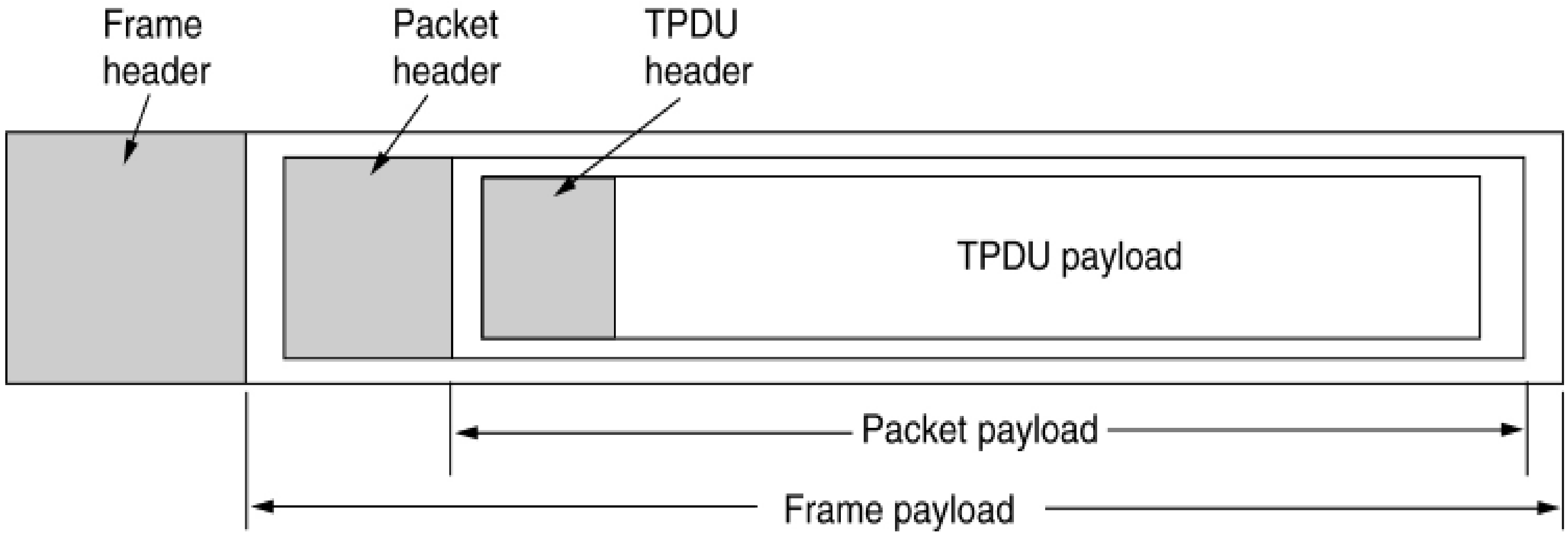
6.1.2. Transport Service Primitives

- 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

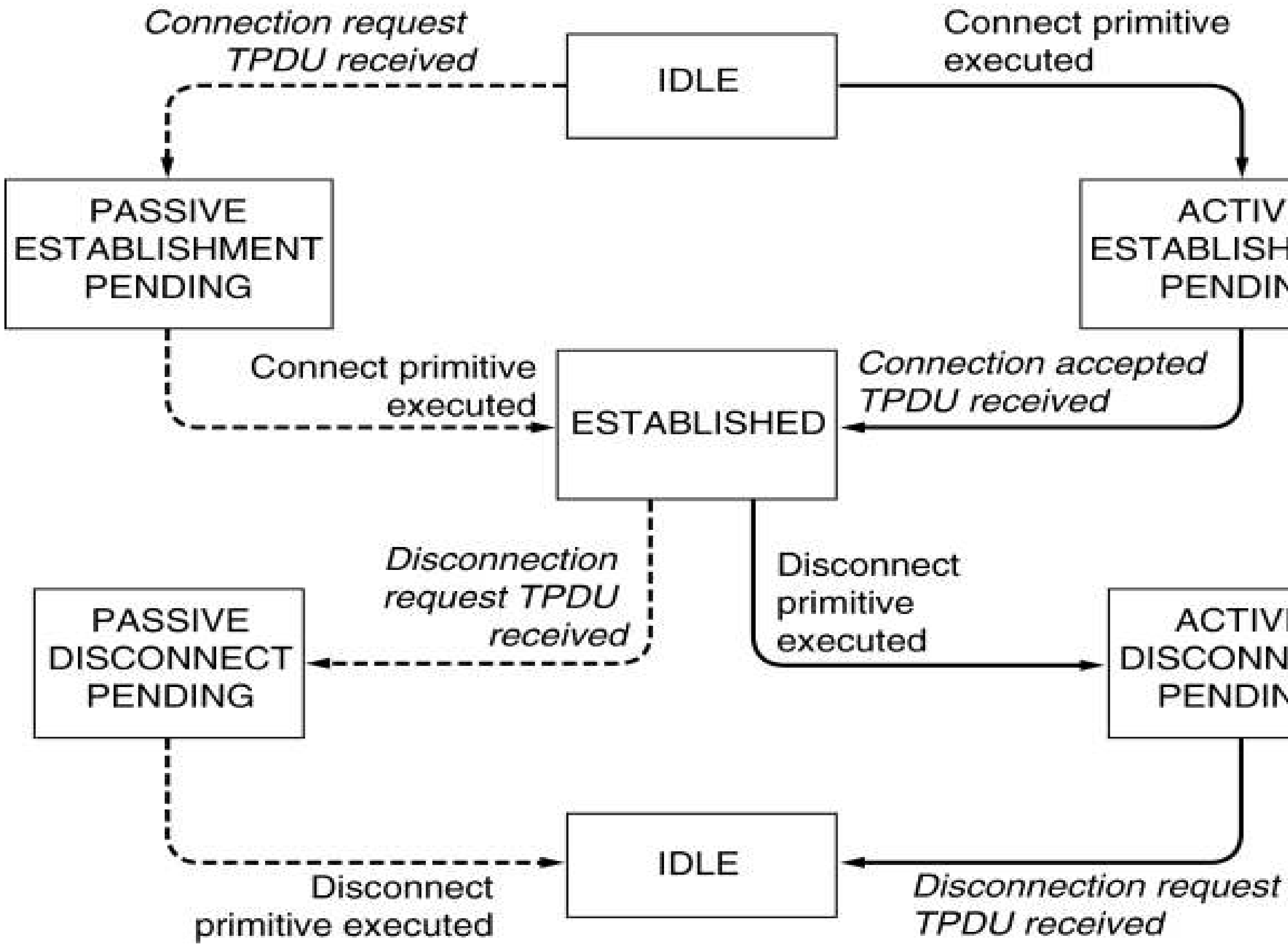
- A state diagram for a simple connection management scheme.

- 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

```
/* 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);
}
```

Client code using sockets.

Socket Programming Example: Internet File Server (2)

```
#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 */
 }
}
```

Client code using sockets.

6.2. Elements of Transport Protocols

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

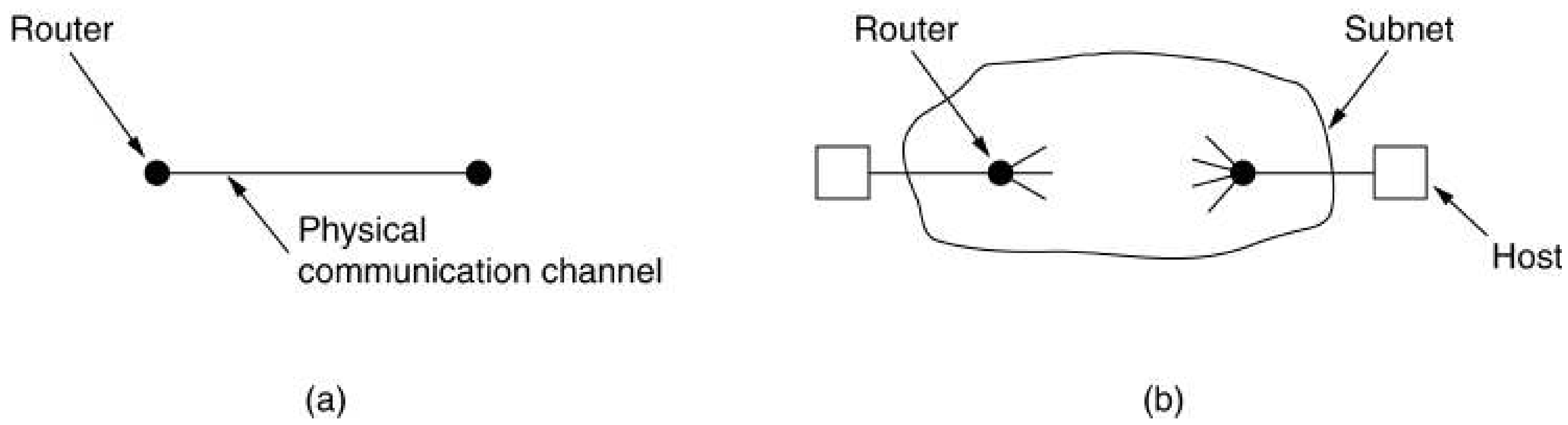
6.2. Elements of Transport Protocols

- 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

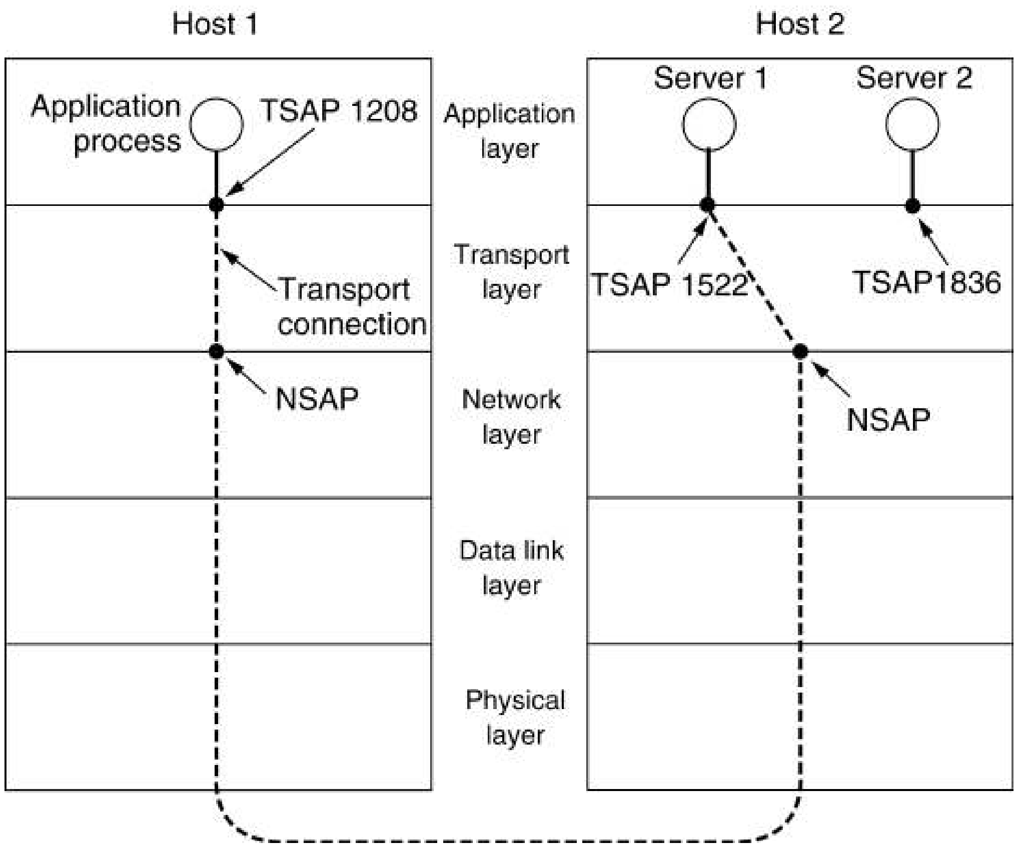
- When an application process wishes to set up a connection to a remote application process, it must specify which one to connect to.

- 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

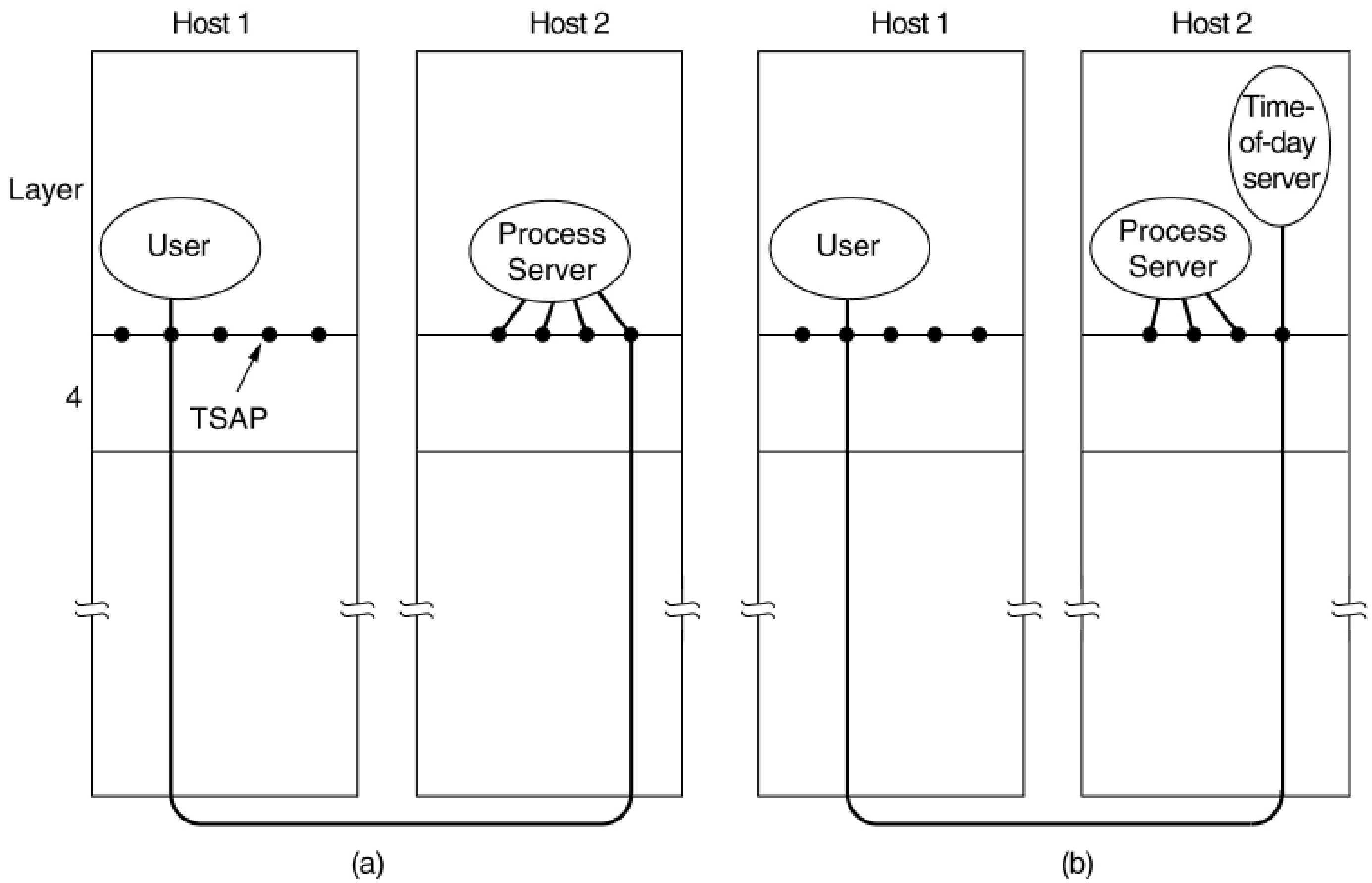
- The crux of the problem is the existence of delayed duplicates.
- It can be attacked in various ways.
- One way is to use throw-away transport address
- Another possibility is to give each connection a connection identifier, etc.

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.

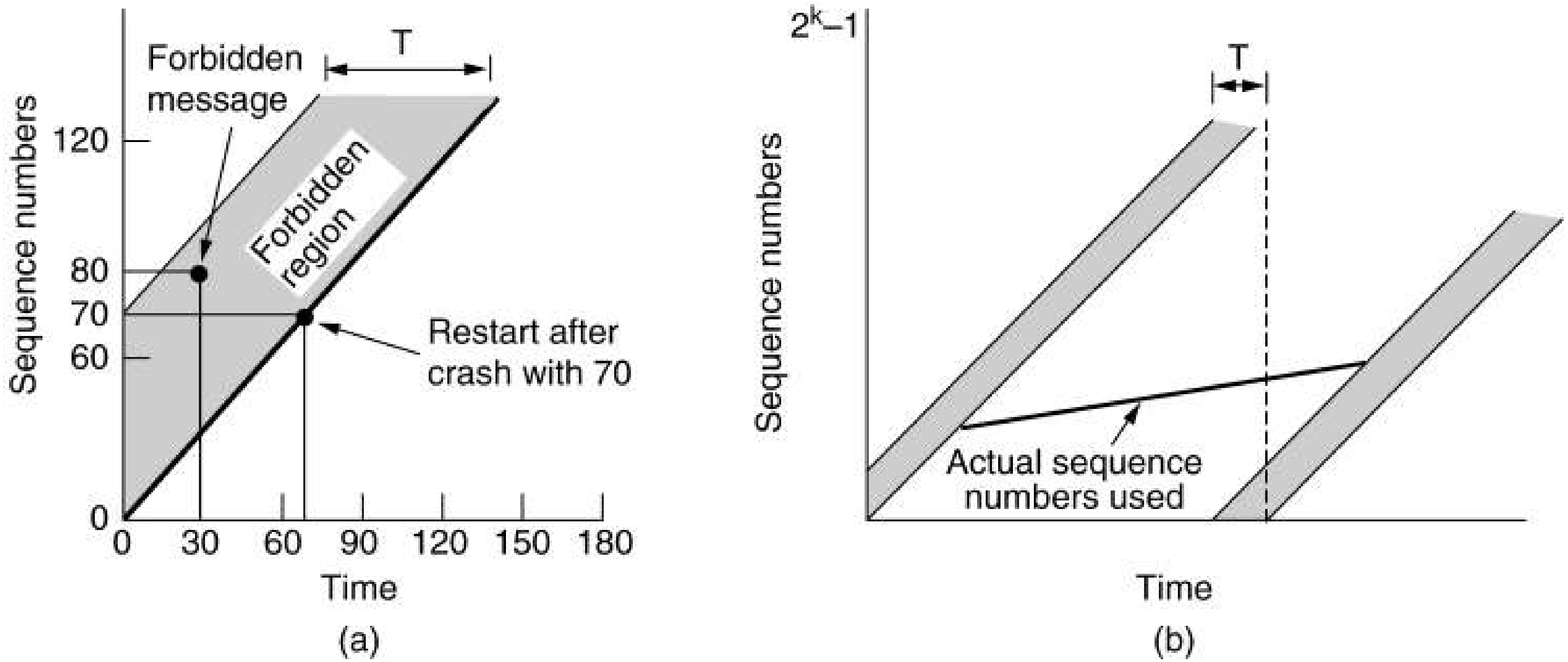
- The basic idea is to ensure that two identically numbered TPDUs are never outstanding at the same time.

6.2.2. Connection Establishment



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

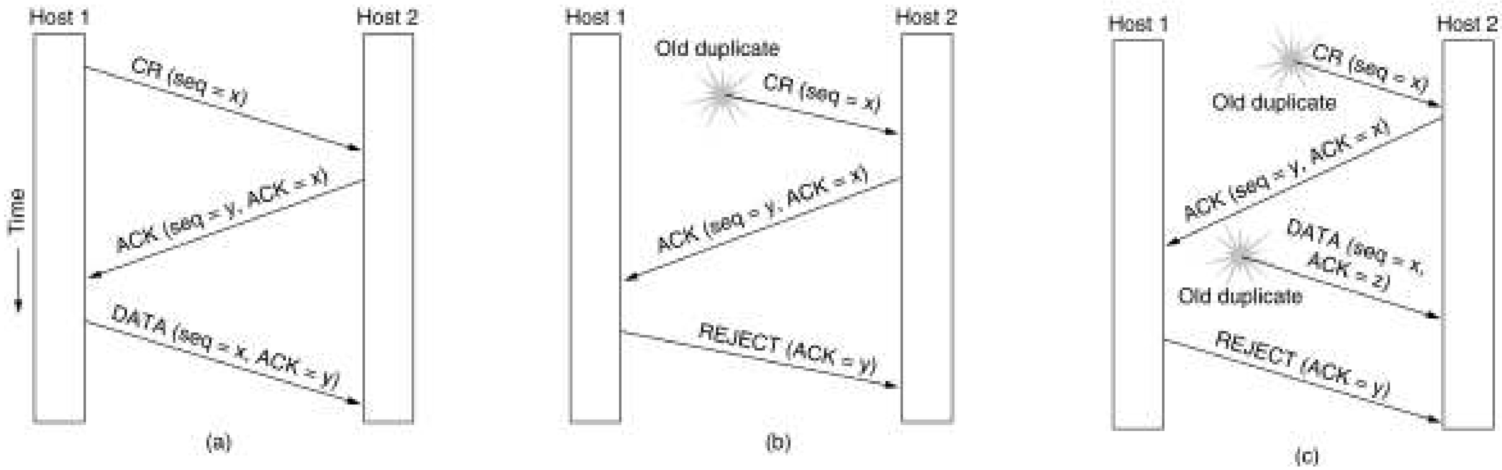
Connection Establishment (2)



(a) TPDUs may not enter the forbidden region.

(b) The resynchronization problem.

Connection Establishment (3)

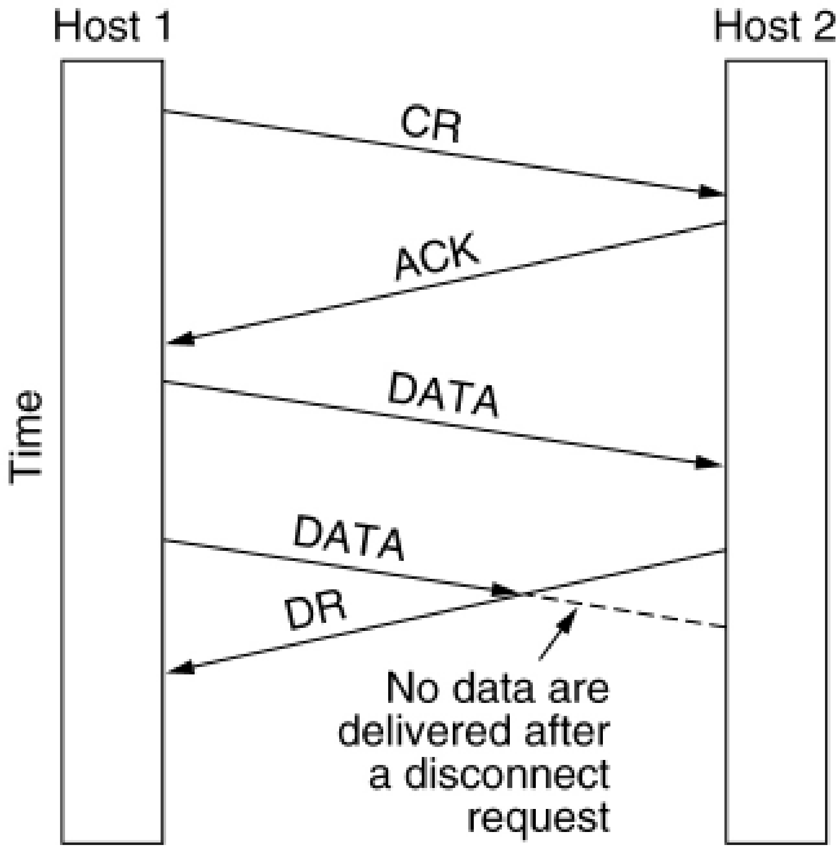


Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST.
(a) Normal operation,
(b) Old CONNECTION REQUEST appearing out of nowhere.
(c) Duplicate CONNECTION REQUEST and duplicate ACK.

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

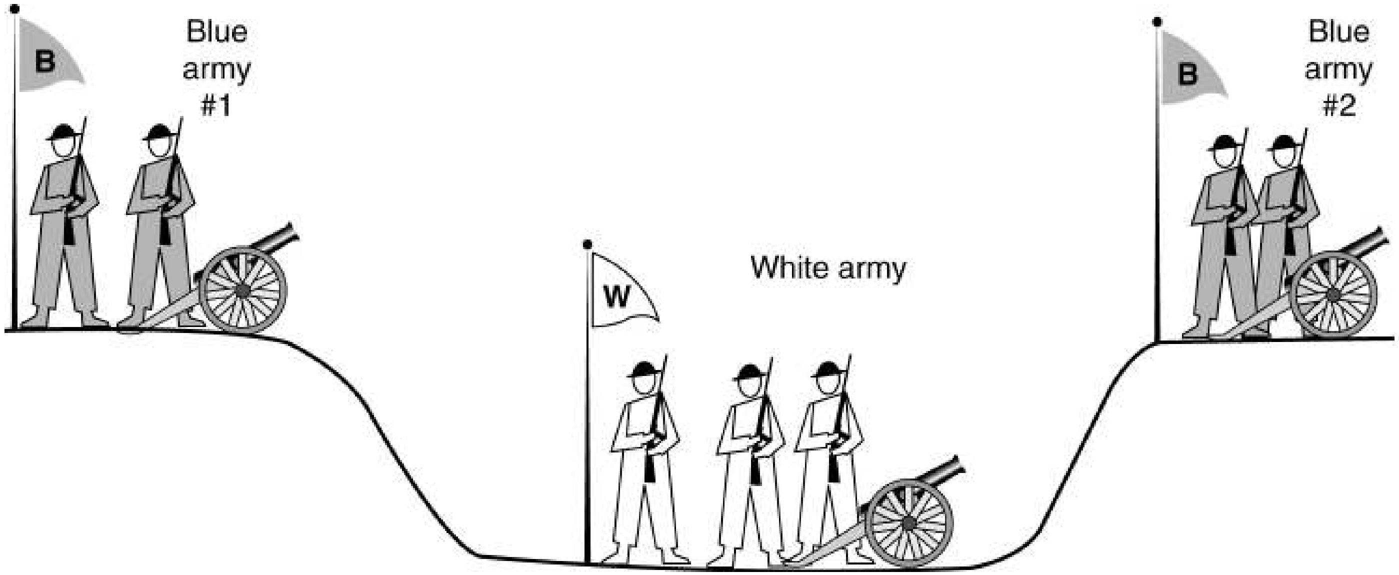


Abrupt disconnection with loss of data.

6.2.3. Connection Release

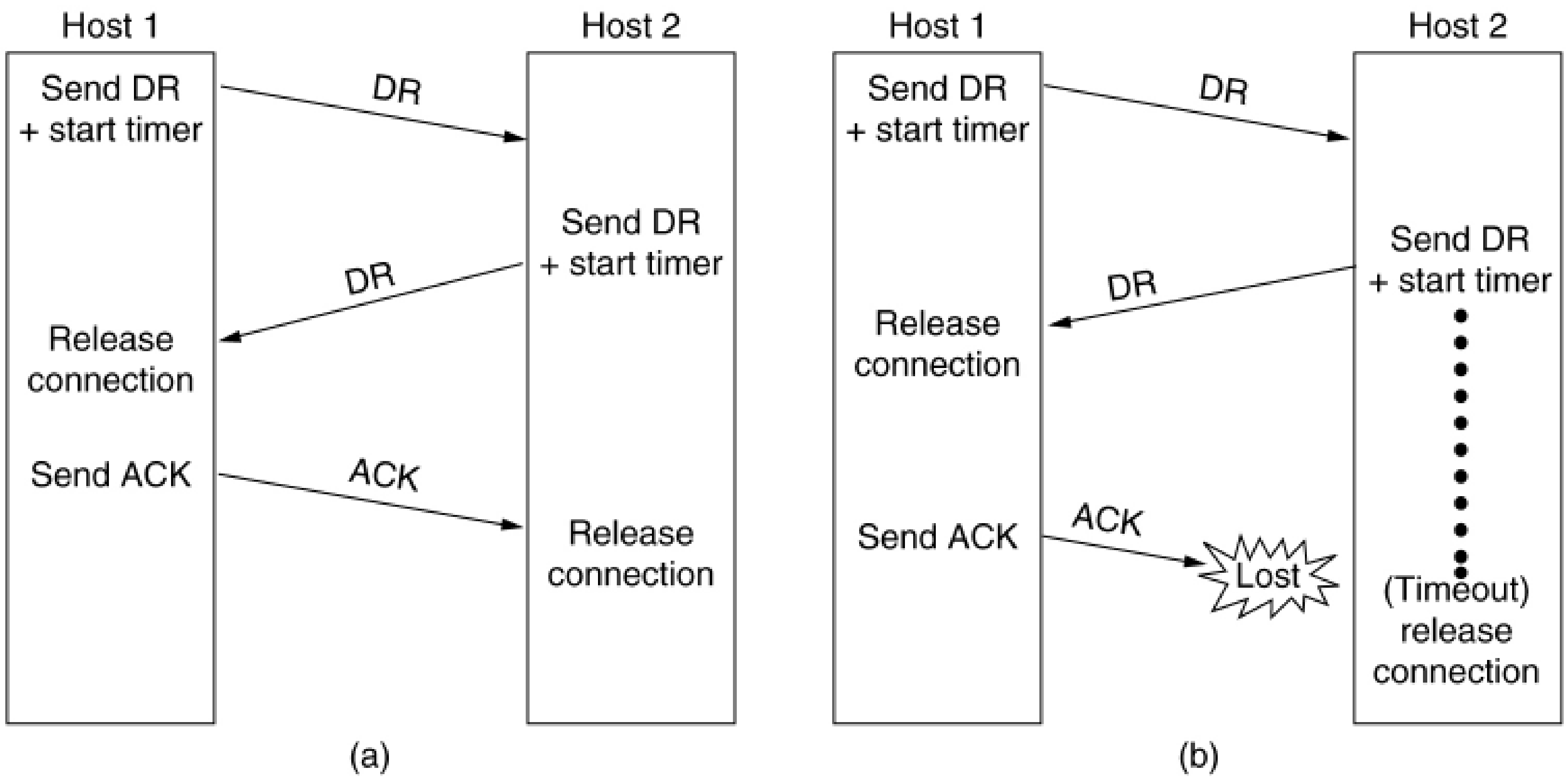
- One way to avoid data loss is to use symmetric release, in which each direction is released independently of the other one.
- One can envision a protocol in which host 1 says: I am done. Are you done too? If host 2 responds: I am done too. Goodbye, the connection can be safely released.
- Unfortunately,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

Connection Release (2)



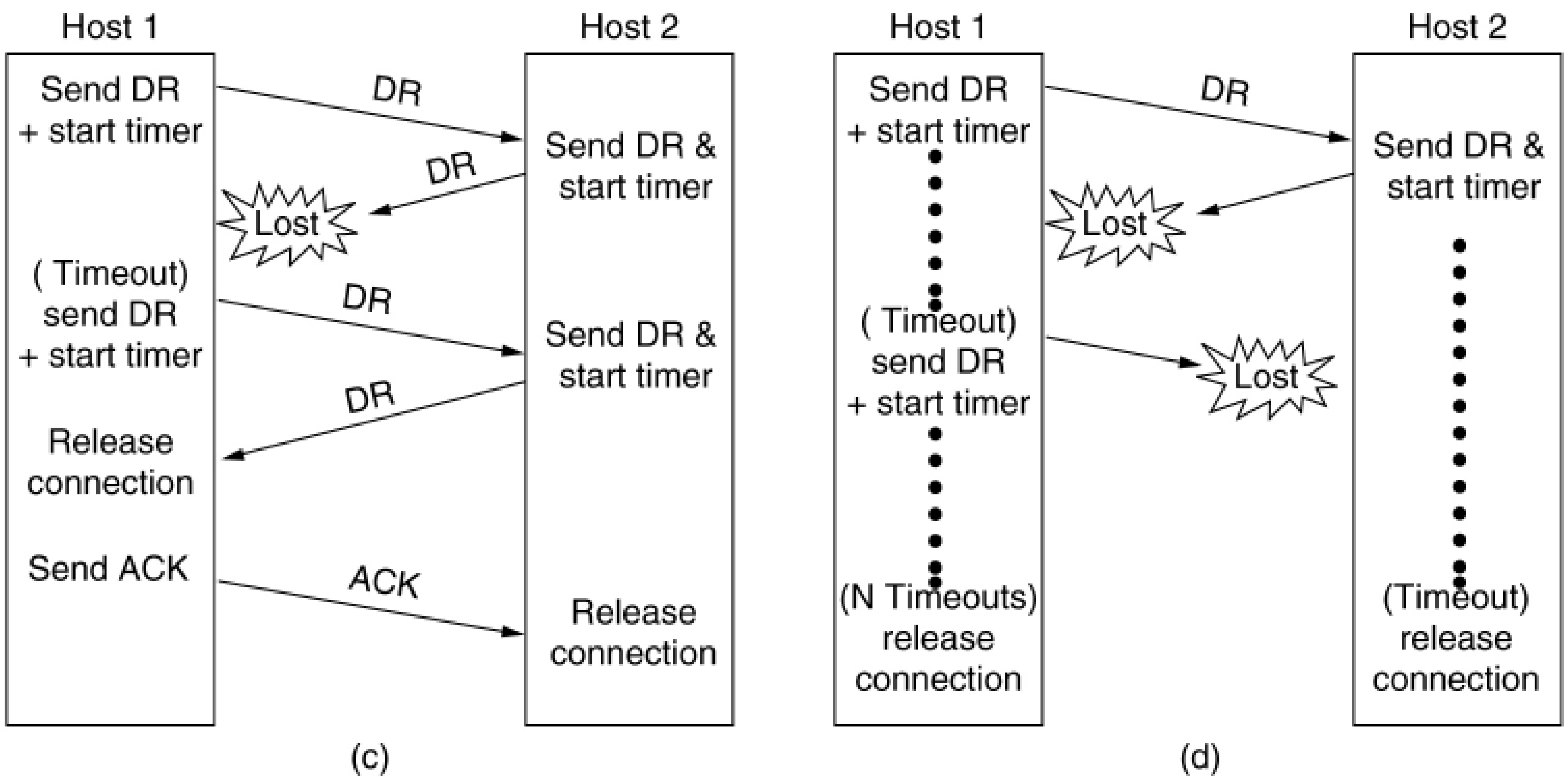
The two-army problem.

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.

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

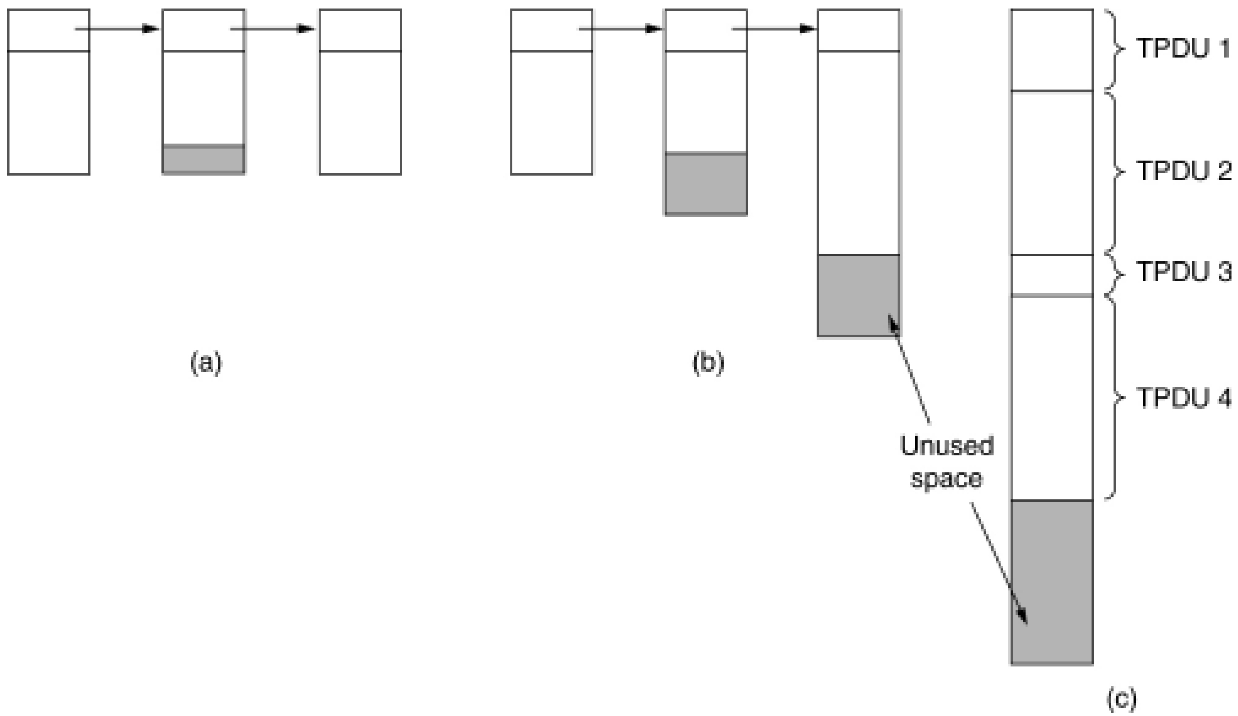
- If the network service is unreliable, the sender must buffer all TPDUs sent.
- However, with reliable network service, other trade-off become possible.
- If the sender knows that the receiver always has buffer size, it need not retain copies of the TPDUs it sends.

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)

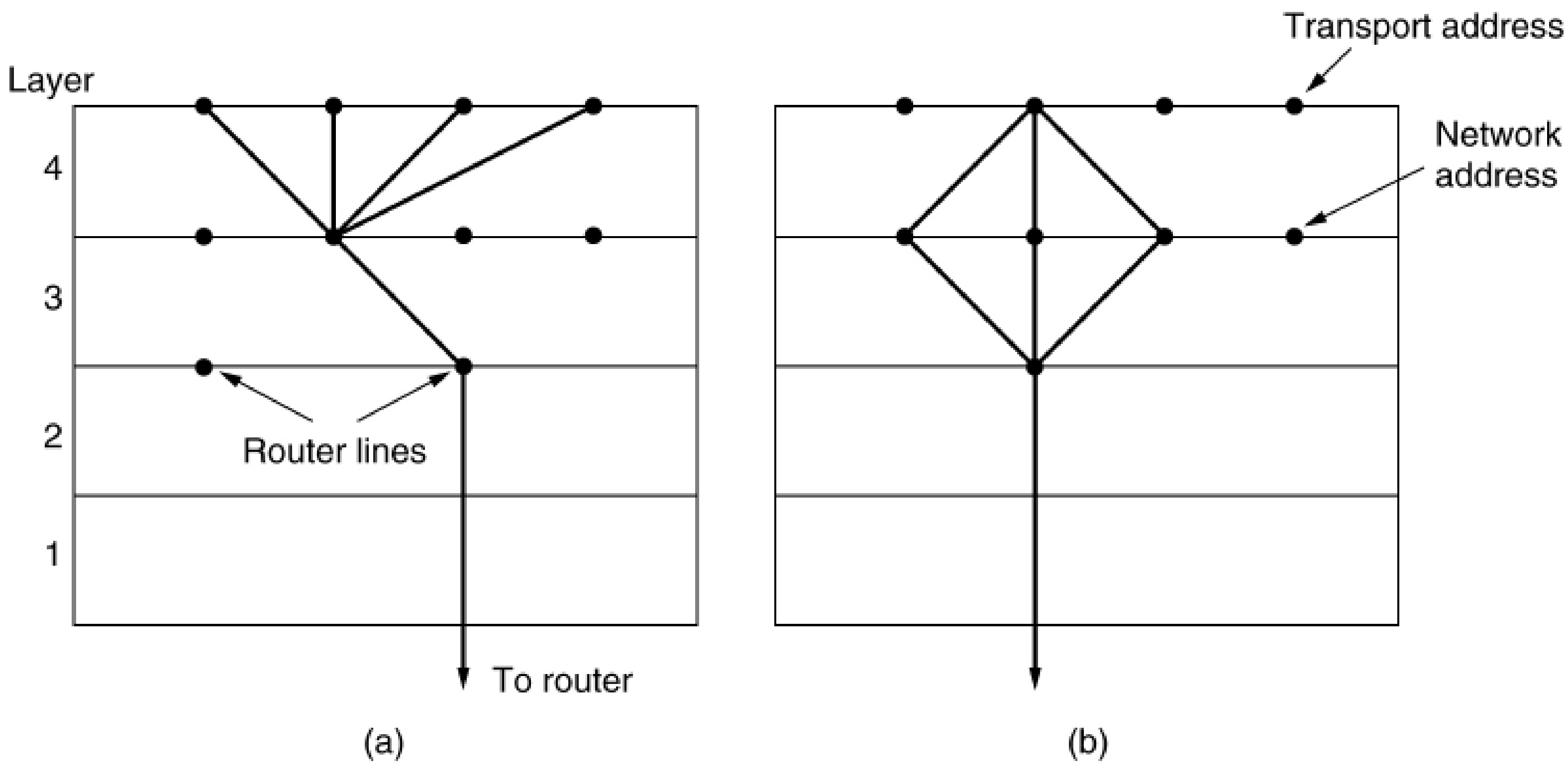
	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	• • •	Message lost but A thinks it has 1 left
6	←	<ack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

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);
              }
        }
  }
}
```

6.3.3. The Example as a Finite State Machine

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

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

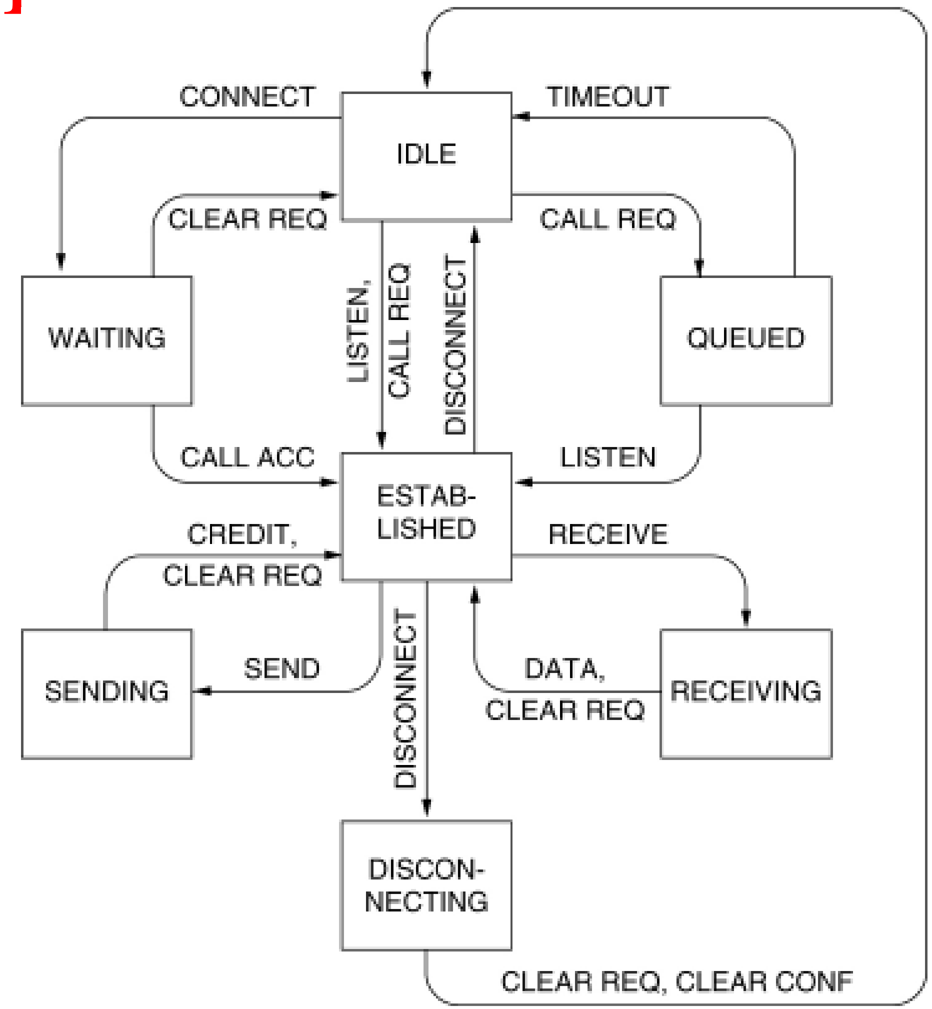
Predicates

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

Actions

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

The Example as a Finite State Machine (2)



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