The Transport Layer

Chapter 6

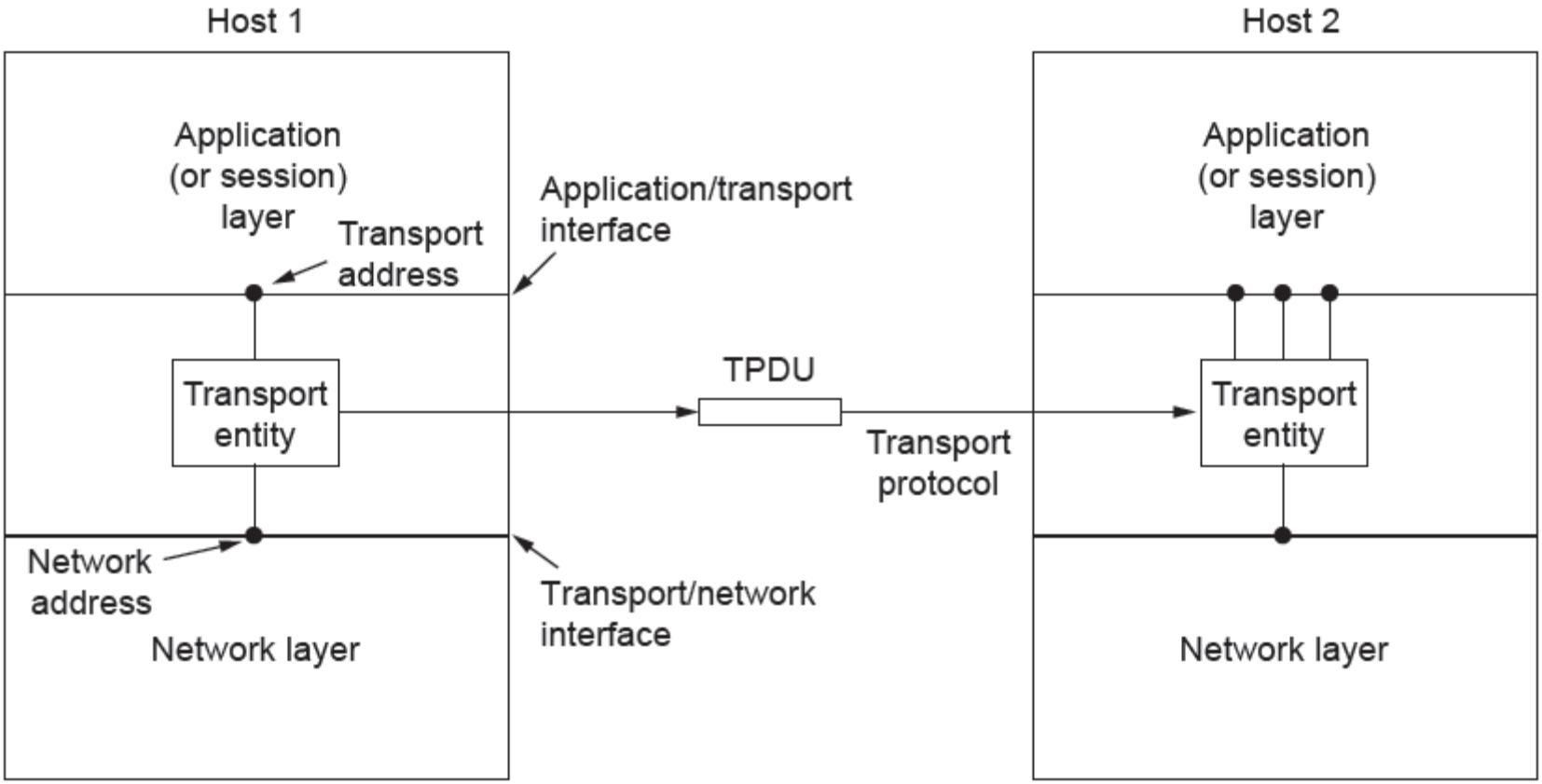
- Reliability
- Connections and congestion control
- TCP&UDP

  Performance

Transport Service

- Upper Layer Services
- Transport Service Primitives
- Berkeley Sockets
- Example of Socket Programming: Internet File Server

Services Provided to the Upper Layers



The network, transport, and application layers

Cont.

- The goal of the transport layer:
  - To provide efficient, reliable, and cost-effective data transmission service to its users, normally processes in the application layer.
- To achieve this goal:
  - The transport layer makes use of the services provided by the network.
- The **transport entity**:
  - The hardware and/or software within the transport layer that does the work.

Cont.

- Two types of transport service:
  - The *connection-oriented transport* service, similar to the connection-oriented network service: Both have connections in three phases.
    - establishment
    - data transfer
    - release

    Addressing and flow control are also similar.
  - The *connectionless transport service*, similar to the connectionless network service.

Cont.

- The existence of the transport layer makes it possible for transport service to be more reliable than the underlying network service.
- The transport service primitives can be designed to be independent of the network service primitives which may vary from network to network.
  - It is possible for application programs to be written using a standard set of primitives, and to have these programs work on a wide variety of networks, without having to deal with different subnet interfaces and levels of unreliability.
  - The transport layer fulfills the function of isolating the upper layers from the technology, design, and imperfections of the network.

Cont.

Service Provider versus *Service User*:

- The bottom four layers, layers 1 through 4, can be seen as the **transport service provider**.
- The upper layer(s) are the **transport service user.**
- The transport layer hence forms the major boundary between the provider and user of the reliable data transmission service.

Transport Service Primitives (1)

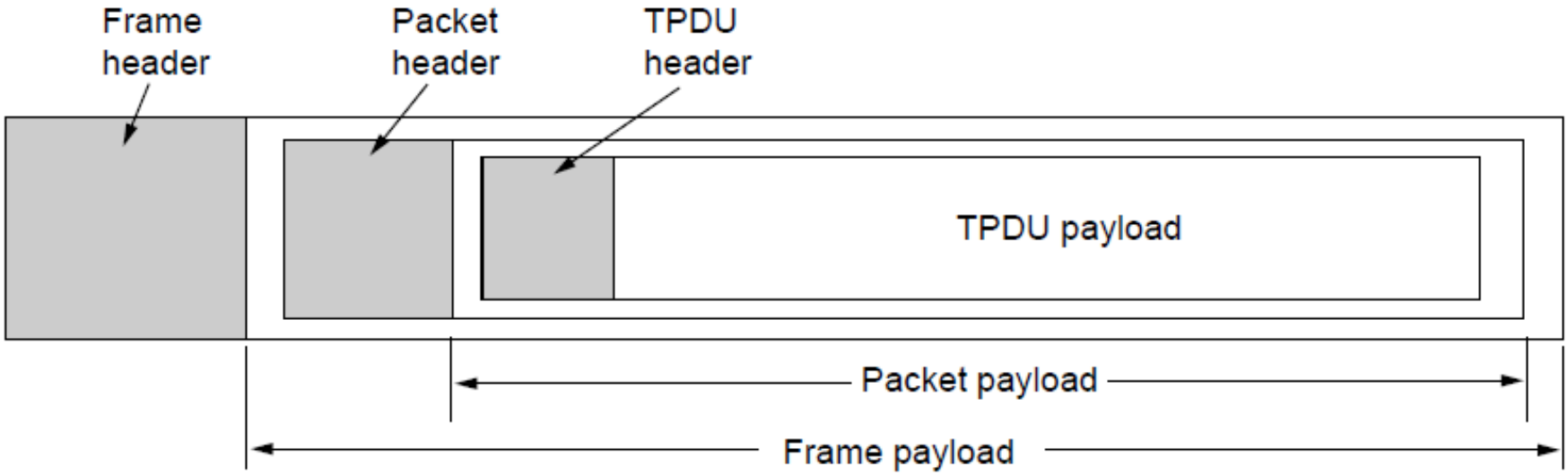
Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

The primitives for a simple transport service

Cont.

- The transport service primitives allow transport users (e.g., applications programs) to access the transport service.
- Some important differences between the transport service and the network service:
  - The network service is intended to model the service offered by real networks which can lose packets, so it is generally unreliable. In contrast, the (connection-oriented) transport service is reliable. (Thus, it is the purpose of the transport layer--- to provide a reliable service on top of an unreliable network.)
  - The network service is used only by the transport entities. In contrast, many programs (programmers) see the transport primitives.

Transport Service Primitives (2)

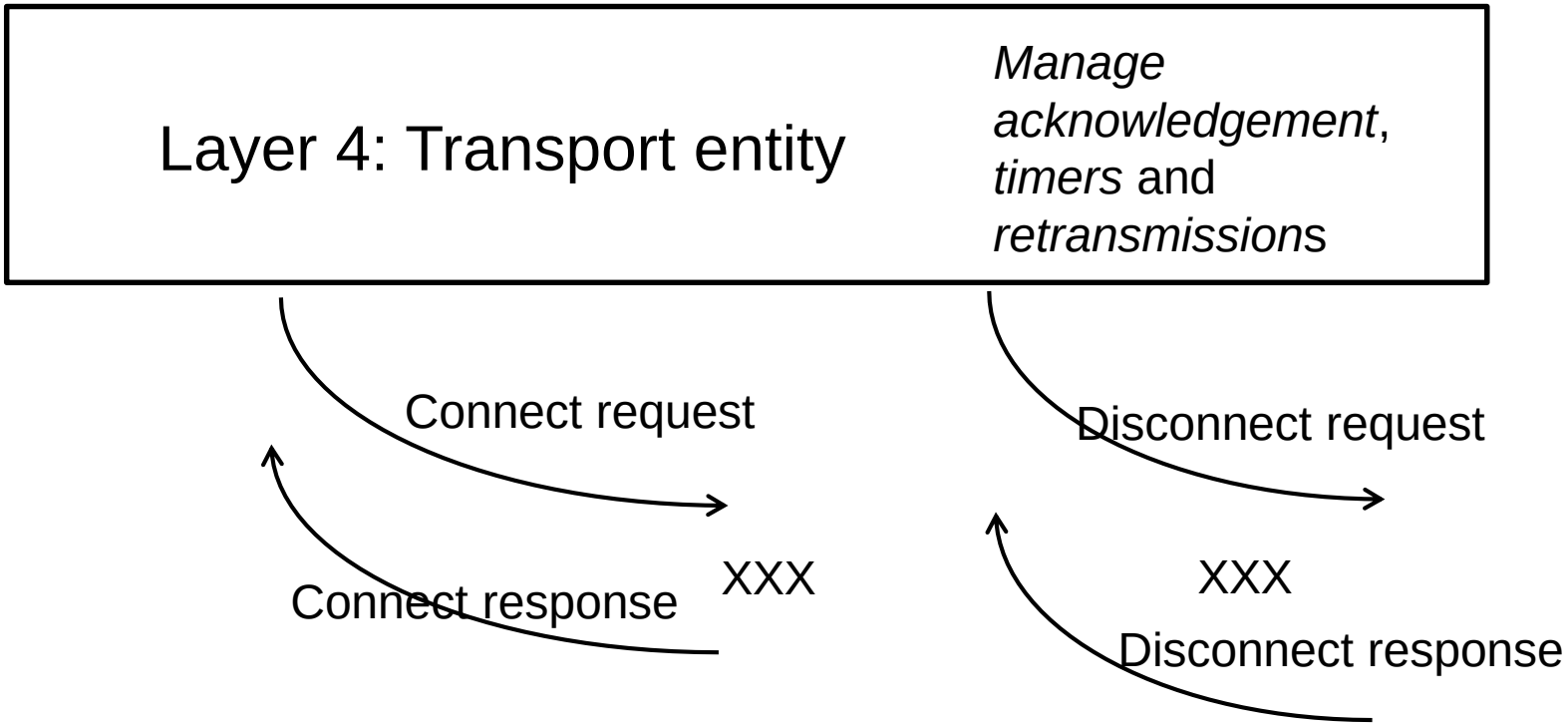


Nesting of TPDUs(segments), packets, and frames.

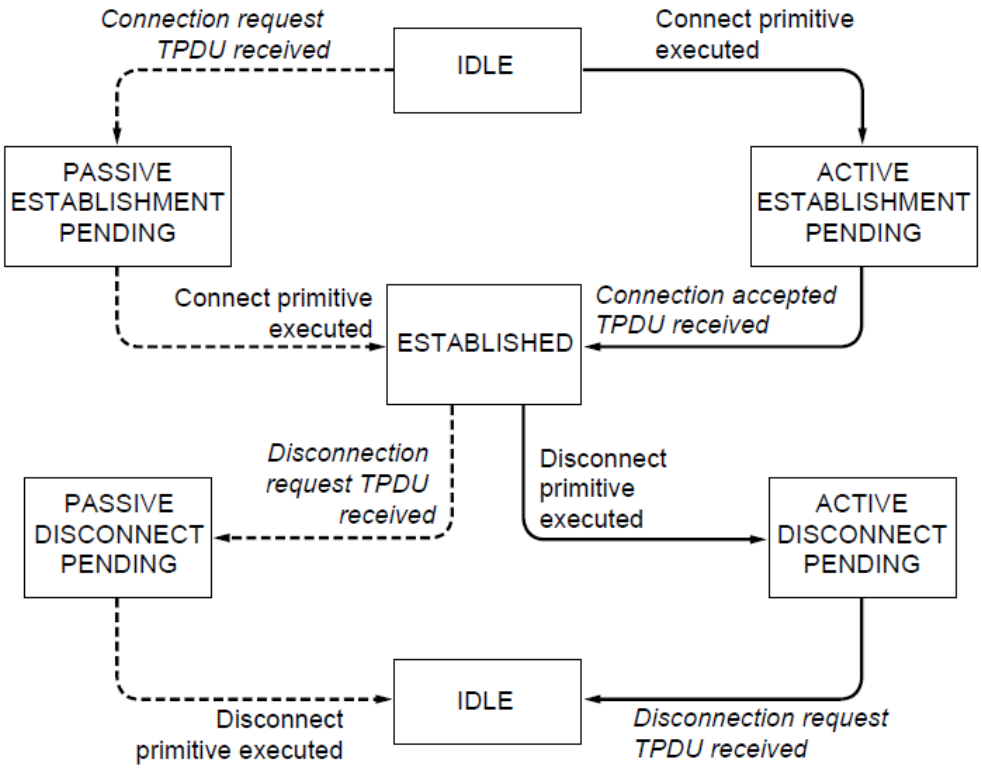
Cont.

- The transport entities need to manage *acknowledgement*, *timers* and *retransmissions*.
- Disconnection has two variants:
  - Asymmetric: Either transport user can issue a DISCONNECT primitive, which results in a DISCONNECT segment being sent to the remote transport entity. Upon arrival, the connection is released.
  - Symmetric: Each direction is closed separately, independent of the other one.
    - When one side does a DISCONNECT, that means it has no more data to send, but it is still willing to accept data from its partner.

TP service:
LISTEN( ), CONNECT( ), SEND( ), RECEIVE( ), DISCONNECT( );



Berkeley Sockets (1)



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.

Cont.

- In the above figure,
  - Each transition is triggered by some event, either a primitive executed by the local transport user or an incoming packet.
  - Assume that each segment is separately acknowledged, and a symmetric disconnection model is used.

Berkeley Sockets (2)

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
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

Cont.

- Used in Berkeley UNIX for TCP.
  - The first four primitives in the list are executed in that order by servers.
  - The SOCKET primitive creates a new end point and allocates table space for it within transport entity.
    - The parameters specify the addressing format to be used, the type of service desired, and the protocol.
    - A successful SOCKET call returns an ordinary file descriptor for use in succeeding calls.
  - To block waiting for an incoming connection, the server executes an ACCEPT primitive.
    - When a TPDU asking for a connection arrives, the transport entity creates a new socket with the same properties as the original one and returns a file descriptor for it.
    - The server can then fork off a process or thread to handle the connection on the new socket and go back to waiting for the next connection on the original socket.

Example of Socket Programming: An Internet File Server (4)

```
#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;           /* holds IP address */

      . . .
```

Server code

Example of Socket Programming: An Internet File Server (5)

. . .

```
/* 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");
```

. . .

Server code

Example of Socket Programming: An Internet File Server (6)

. . .

```
/* 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 */
  }
}
```

Server code

Cont.

- At the client side:
  - A socket must first be created using the SOCKET primitive, but BIND is not required.
  - The CONNECT primitive blocks the caller and actively starts the connection process.
  - When it completes (i.e., when the appropriate segment is received from the server), the client process is unblocked and the connection is established. Both sides can now use SEND and RECV to transmit and receive data over the full duplex connection.

Client flits.cs.vu.nl /usr/tom/filename >f

Example of Socket Programming: An Internet File Server (1)

```
/* 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 */
```

. . . Client code using sockets

Example of Socket Programming: An Internet File Server (2)

. . .

```
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");
```

. . .

Client code using sockets

Example of Socket Programming: An Internet File Server (3)

. . .

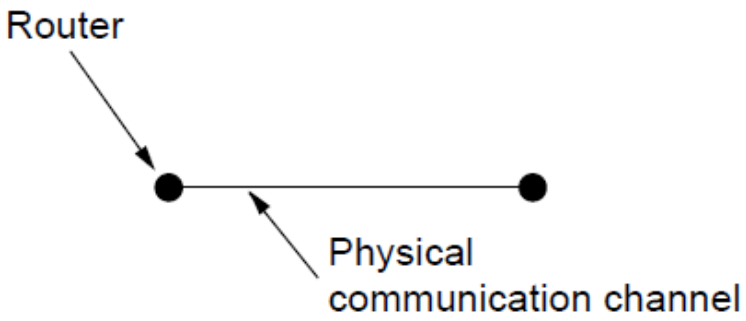
```
 c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
 if (c < 0) fatal("connect failed");

 /* 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

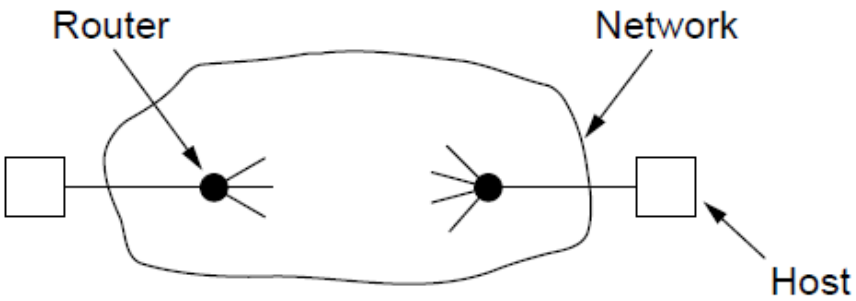
Elements of Transport Protocols (1)

- Addressing
- Connection establishment
- Connection release
- Error control and flow control
- Multiplexing
- Crash recovery

Elements of Transport Protocols (2)



(a)



(b)

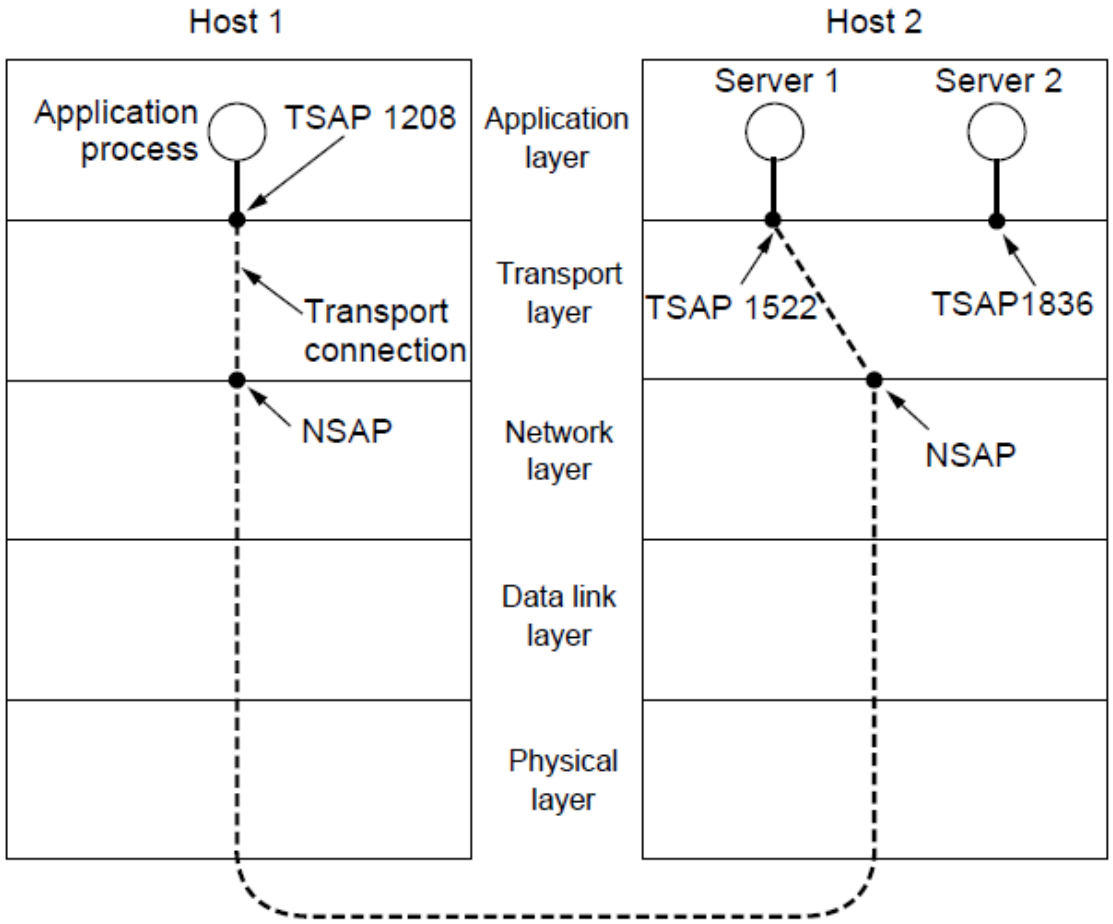
(a) Environment of the data link layer.

(b) Environment of the transport layer.

Cont.

- Transport protocols and data link protocols both have to deal with error control, sequencing, flow control, etc.
  - Differences between the transport and the data link protocols:
    - The environments in which the two protocol operate, as shown in the previous figure.
    - The potential existence of storage capacity in the subnet.
    - The difference between the data link and transport layers is one of amount rather than of kind.
      - Buffering and flow control are needed in both layers.

Addressing (1)



TSAPs, NSAPs, and transport connections

Cont

- Transport addresses are defined so that processes can listen to the addresses for connection requests.
  - In the Internet, these end points (i.e., transport addresses) are (*IP address, local port*) pairs.
    - Some terminology:
      - TSAP: Transport Service Access Point
      - NSAP: Network Service Access Point

      IP addresses are examples of NSAPS.
  - The figure above illustrates the relationship between the NSAP, TSAP, and transport connection.

Cont.

- The scheme of *stable* TSAP addresses can be used for a small number of key services that never change.
  - The schemes for addressing user processes that only exist for a short time and do not have a TSAP address that is known in advance:
    1. The **initial connection protocol**, as shown in the previous figure.
       - Instead of every conceivable server listening at a well-known TSAP, each machine that wishes to offer service to remote users has a special **process server** that acts as a proxy for less-heavily used servers. It listens to a set of ports at the same time, waiting for a connection request.

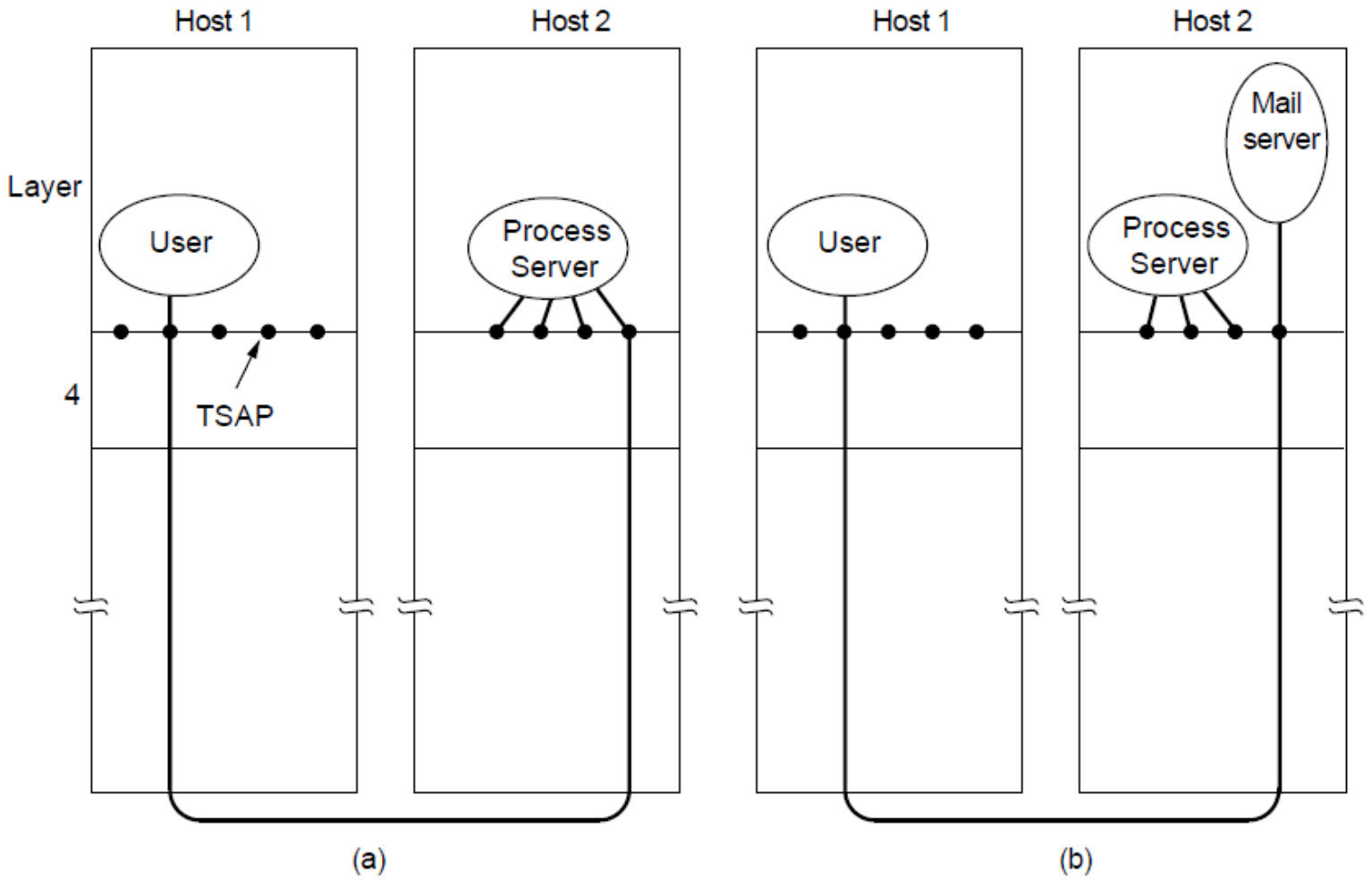
Cont.

2. **Name server** (or **directory server**)← *Portmapper*

   - To find the TSAP address corresponding to a given service name, a user sets up a connection to the name server which listens to a well-known TSAP).
   - The user then sends a message specifying the service name, and the server sends back the TSAP address.
   - Then the user releases the connection with the name server and establishes a new one with the desired service.

   In this model, when a new service is created, it must register itself with the name server, giving both its service name and the address of its TSAP.

Addressing (2)



How a user process in host 1 establishes a connection with a mail server in host 2 via a process server.

Connection Establishment (1)

Techniques for restricting packet lifetime

- Restricted network design.
- Putting a hop counter in each packet.
- Timestamping each packet.

Cont.

- A hard issue is to solve the problem on the existence of delayed duplicates.
  - If it can be ensured that no packet lives longer than some known time, the problem becomes somewhat more manageable.
  - Packet lifetime can be restricted to a known maximum using one of the following techniques:

1 Restricted subnet design.
   - It prevents packet from looping, combined with some way of bounding congestion delay over the longest possible path.

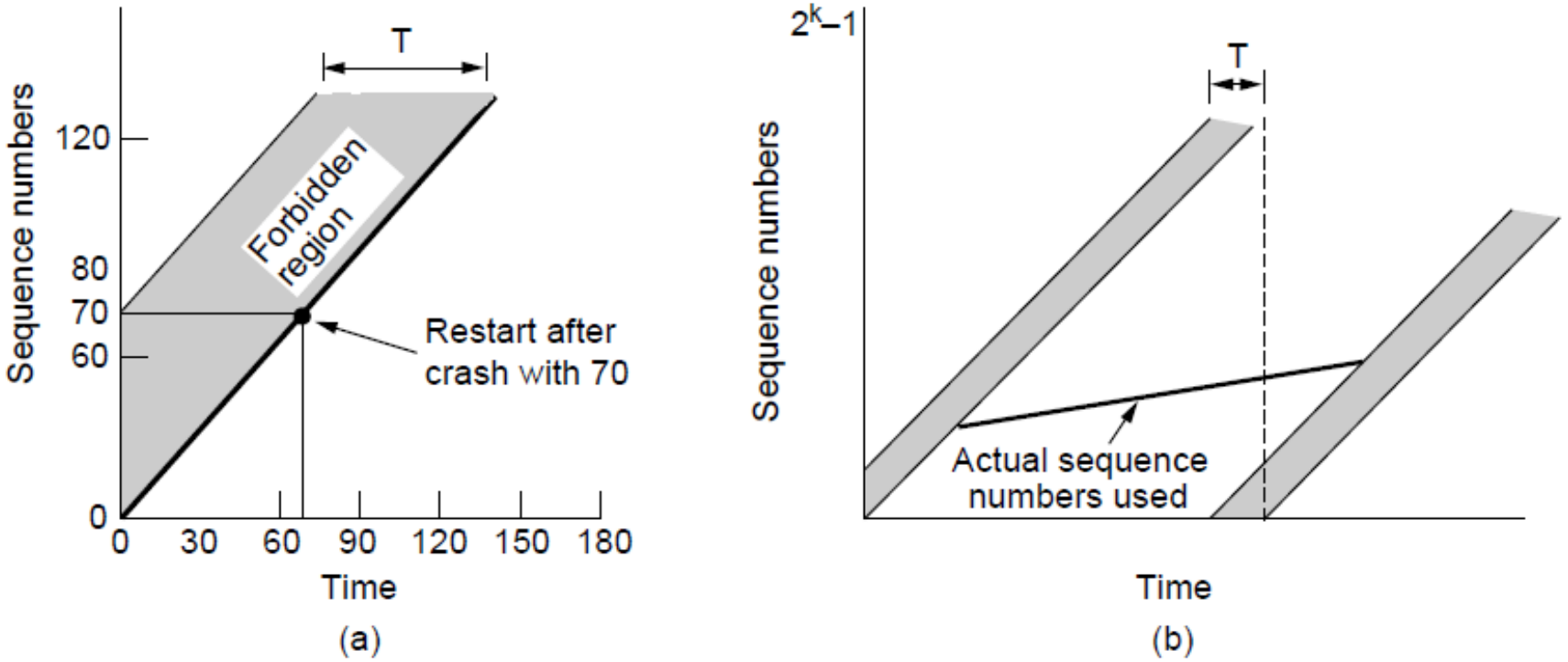
2 Putting a hop counter in each packet.

3 Timestamping each packet
   - It requires the router clocks to be synchronized.

Cont.

- Let *T* be some small multiple of the true maximum packet lifetime. If we wait *T* after a packet has been sent, we can be sure neither it nor its acknowledgement will suddenly appear. With packet lifetimes bounded, it is possible to devise a way to establish connections safely:
- 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 clocks at different hosts need not be synchronized.
  - Each clock is assumed to take the form of a binary counter that increments itself at uniform intervals.
  - The number of bits in the counter must equal or exceed the number of bits in the sequence numbers.
  - The clock is assumed to continue running even if the host goes down.

Connection Establishment (2)



(a) TPDUs may not enter the forbidden region.

(b) The resynchronization problem.

Cont

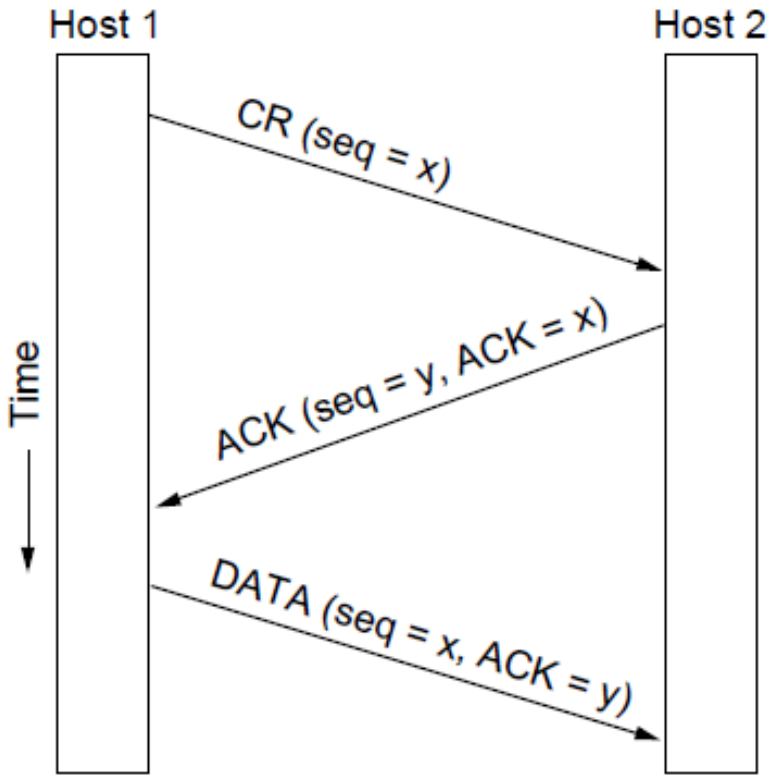
- The basic idea is to ensure that two identical numbered segments are never outstanding at the same time.
  - When a connection is set up, the low-order *k* bits of the clock are used as the initial sequence number.
  - The sequence space should be so large that by the time sequence numbers wrap around, old segments with the same sequence number are long gone.
  - The linear relation between time and initial sequence numbers is shown in the previous figure.
- When a host crashes and comes up again, its transport entity does not know where it was in the sequence space.
  - One solution is to require transport entities to be idle for T sec after a recovery to let all old segments die off---The strategy is unattractve.
  - To avoid requiring *T* sec of dead time after a crash, a new restriction on the use of sequence number is introduced

    The period T and the rate of packet per second determine the size of the sequence numbers.

Cont.

- The **forbidden region** shows the illegal combinations of time and sequence number.
  - Before sending any segment on any connection, the transport entity must read the clock and check to see that it is not in the forbidden region.
    - The maximum data rate on any connection is one segments per clock tick.
    - The transport entity must wait until the clock ticks before opening a new connection after a crash restart.
    - If the sequence number curve is about to enter the forbidden region (from underneath), either delay the segments for *T* sec or resynchronize the sequence number.
  - For a clock rate of C and a sequence number space of size S, we must have S/C>T so that the sequence numbers cannot wrap around quickly.

Connection Establishment (3)

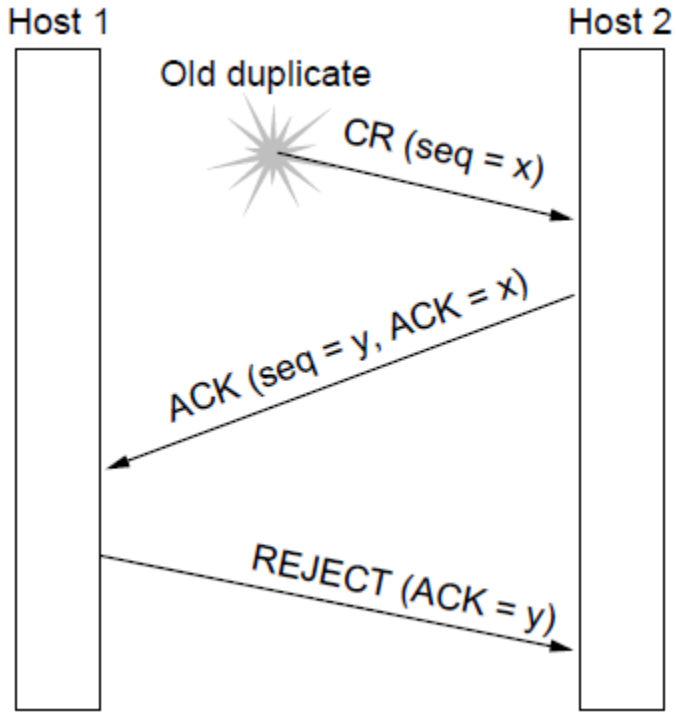


Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. Normal operation.

Cont.

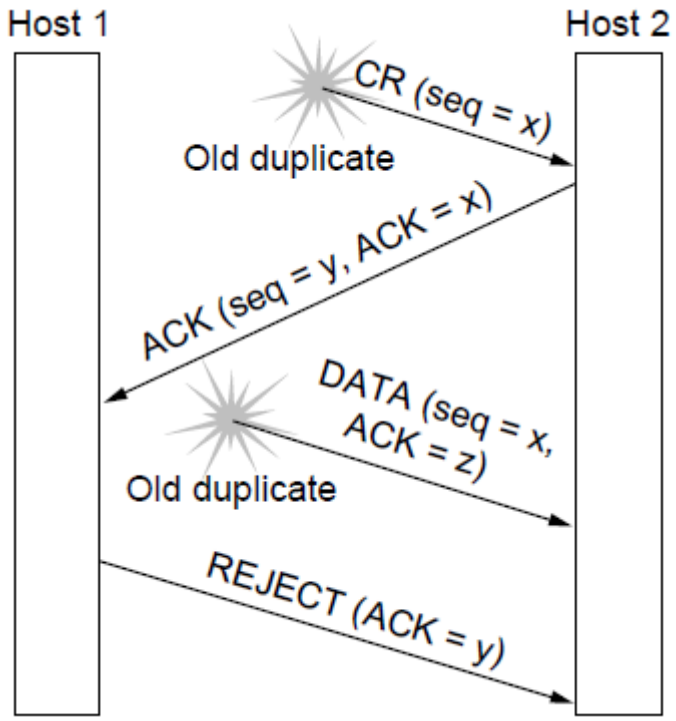
- The clock-based method, just mentioned, solves the delayed duplicate problem for data segments.
- To solve the delayed control segments, Tomlinson introduced the **three-way handshake**:
  - The normal setup procedure when host 1 initiates is shown in the figure above.
  - Host 1 chooses a sequence number *x* and send a CONNECTION REQUEST segment containing it to host 2.
  - Host 2 replies with a CONNECTION ACCEPTED segment acknowledging *x* and announcing its own initial sequence number *y*.
  - Finally, host 1 acknowledges host 2's choice of an initial sequence number in the first data segment that it sends.

Connection Establishment (4)



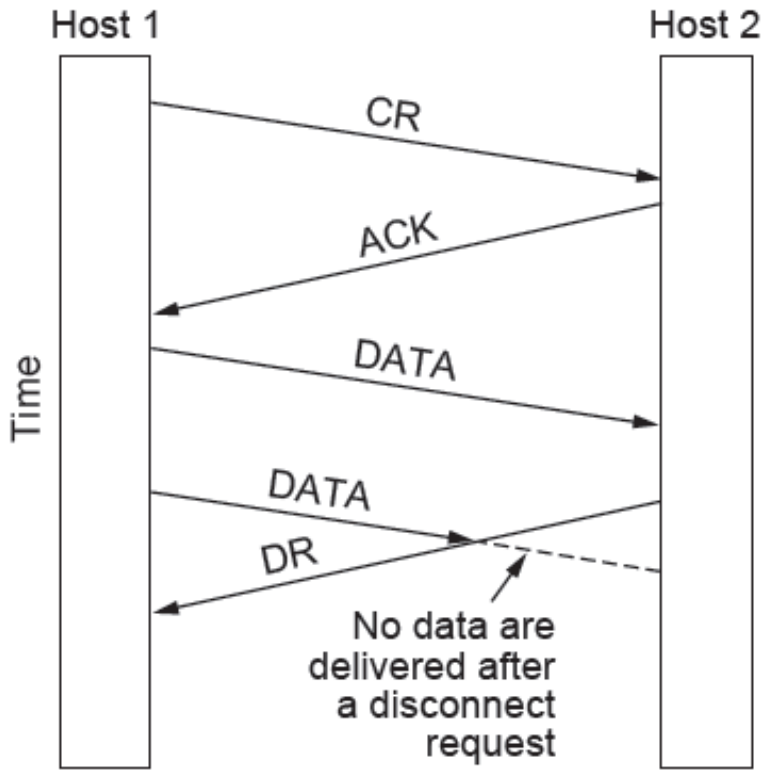
Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. Old duplicate CONNECTION REQUEST appearing out of nowhere.

Connection Establishment (5)



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

Connection Release (1)

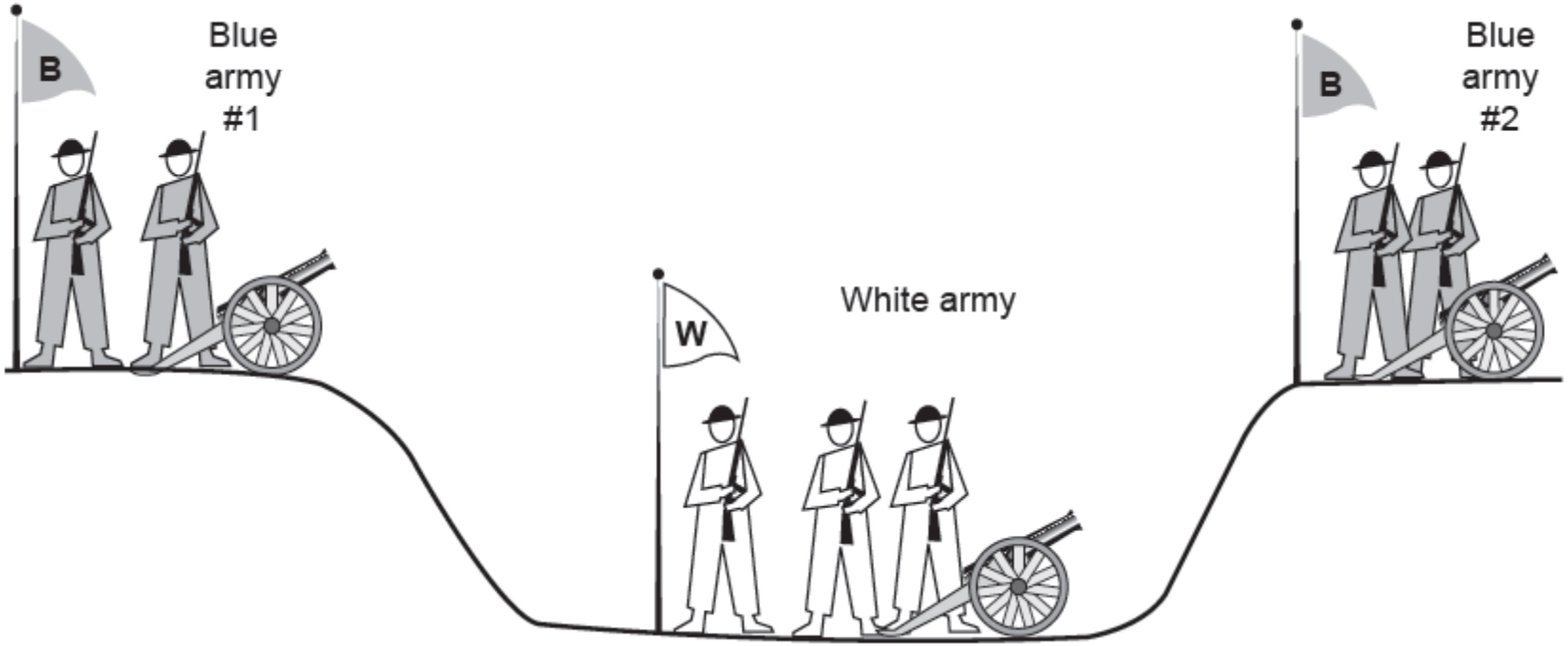


Abrupt disconnection with loss of data

Cont.

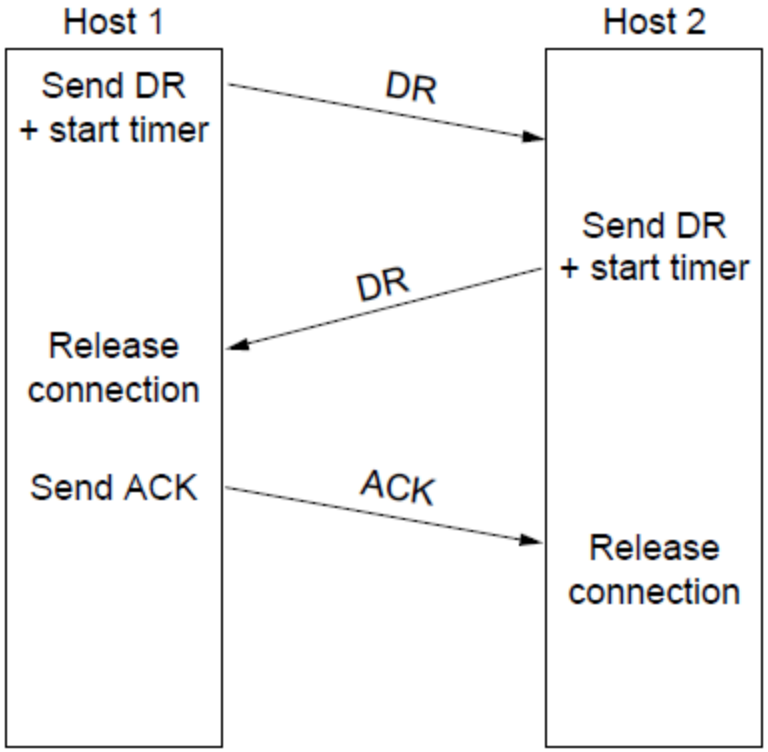
- Asymmetric release:
  - The way that the telephone system works: when one party hangs up, the connection is broken.
  - It may result in data loss, by the scenario of the previous figure.
- Symmetric release:
  - Each direction is released independently of the other one.
  - A two-army problem deals with the issue, in next figure.
    - If either side is prepared to disconnect (``attack") until it is convinced that the other side is prepared to disconnect too, the disconnection will never happen.

Connection Release (2)



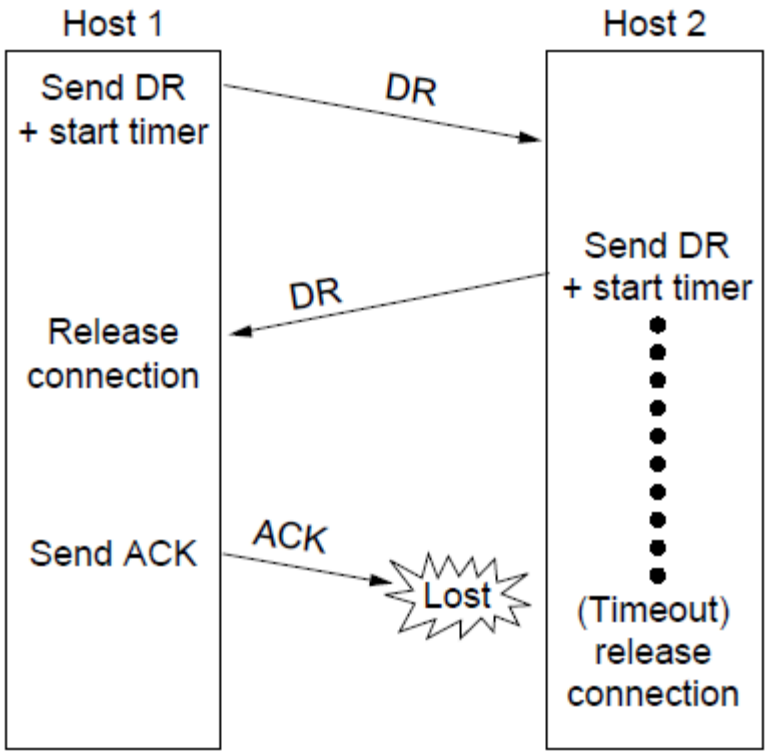
The two-army problem

Connection Release (3)



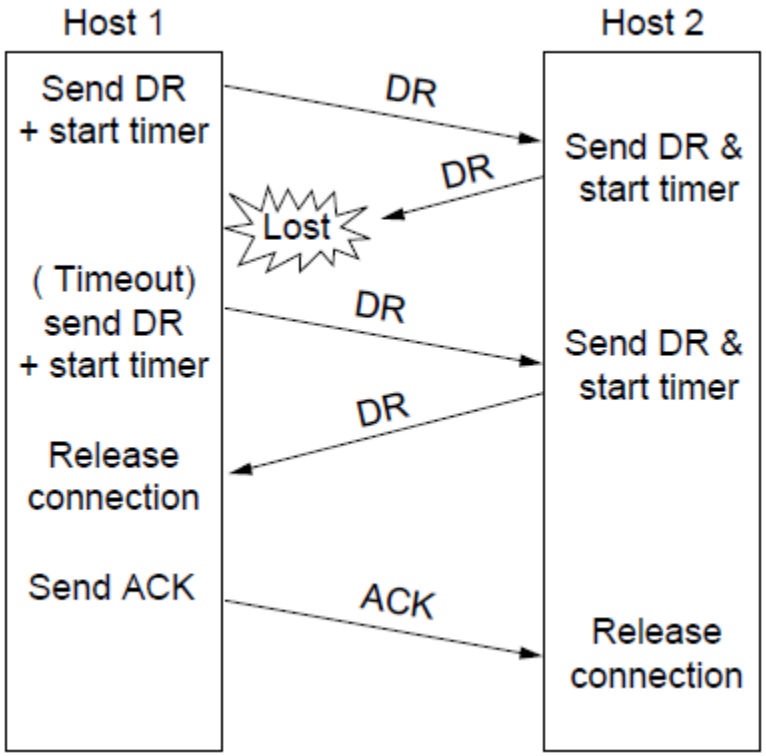
Four protocol scenarios for releasing a connection.
(a) Normal case of three-way handshake

Connection Release (4)



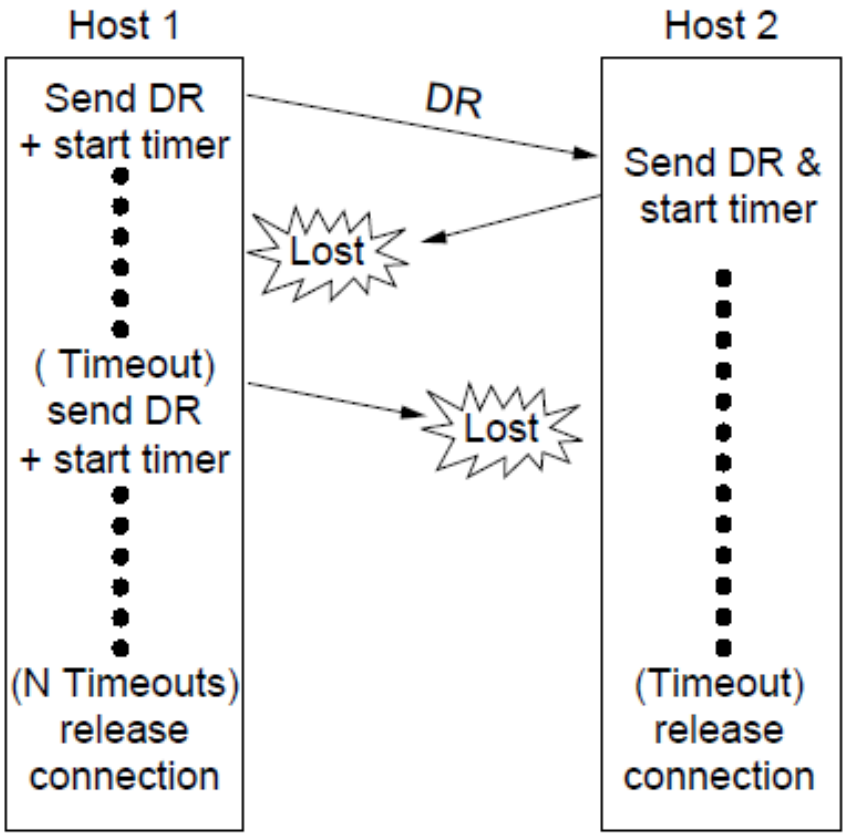
Four protocol scenarios for releasing a connection.
(b) Final ACK lost.

Connection Release (5)



Four protocol scenarios for releasing a connection.
(c) Response lost

Connection Release (6)



Four protocol scenarios for releasing a connection.
(d) Response lost and subsequent DRs lost.

Cont.

- The previous figures show four protocol scenarios of releasing using a three-way handshake.
  - In theory, the protocol can fail if the initial DR and N retransmission are all lost. The sender will give up and release the connection while the other side is still fully active. This situation results in a half-open connection.
- One way to kill off half-open connections is to have a rule that if no segments have arrived for a certain number of seconds, the connection is automatically disconnected.
  - This rule requires that each transport entity has a timer that is stopped and then restarted whenever a segment is sent.
  - If this timer expires, a dummy segment is transmitted, just to keep the other side from disconnecting. (but, …)