The Transport Layer

Chapter 6

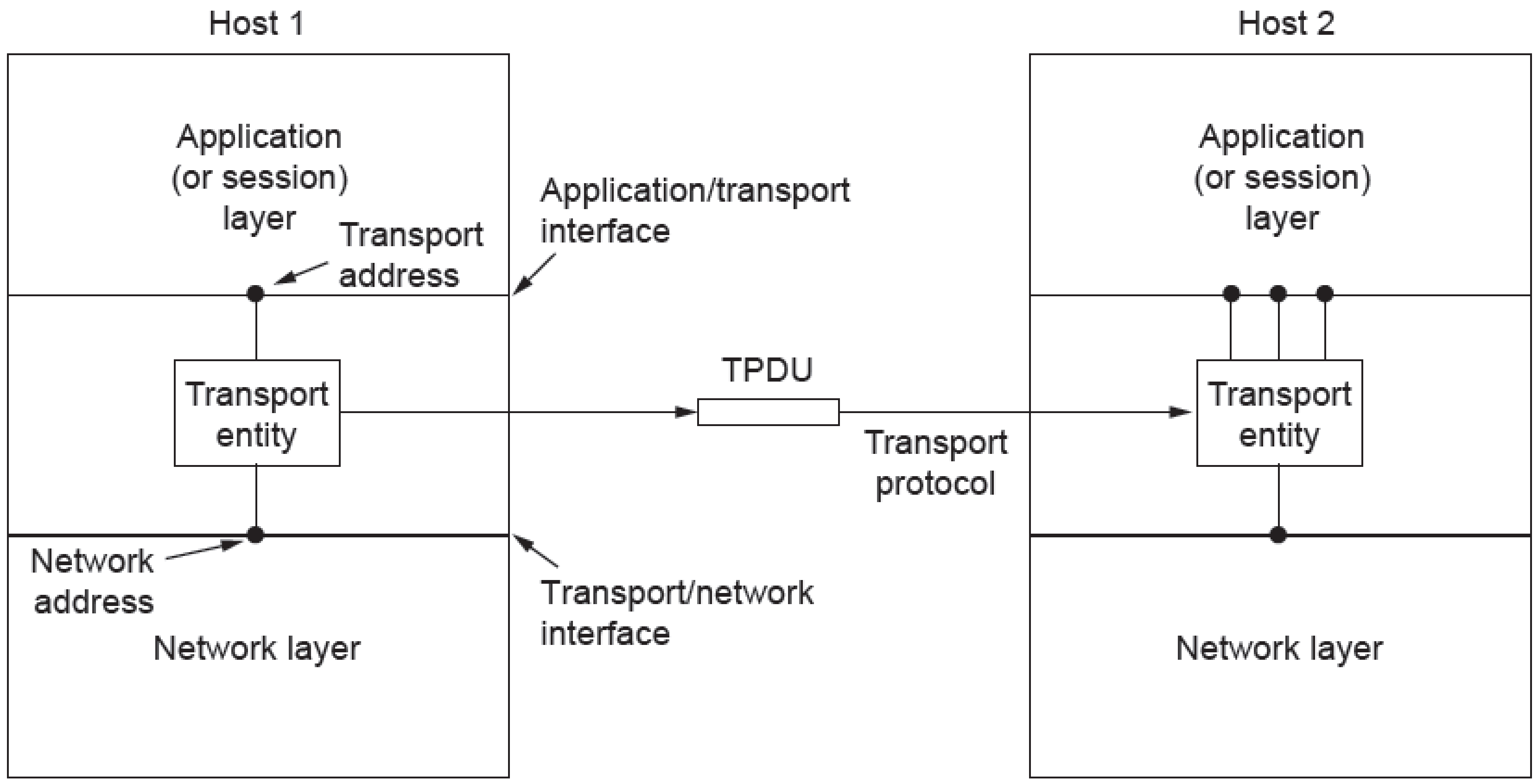
- Reliability
- Connections and congestion control
- TCP&UDP

  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.

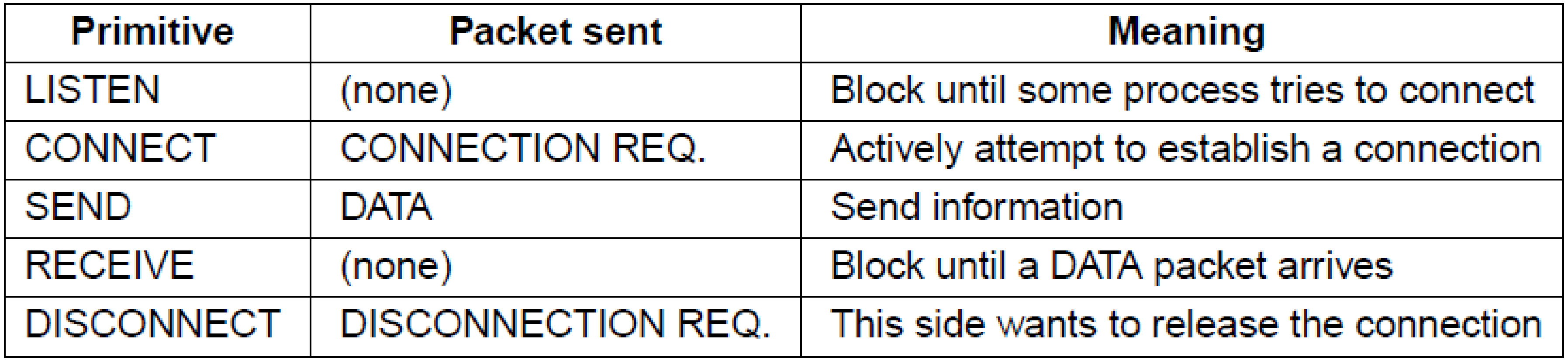
- The existence of the transport layer makes it possible for transport service to be more reliable than the underlying network service.
- The transport service primitives can be designed to be independent of the network service primitives which may vary from network to network.
  - It is possible for 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)



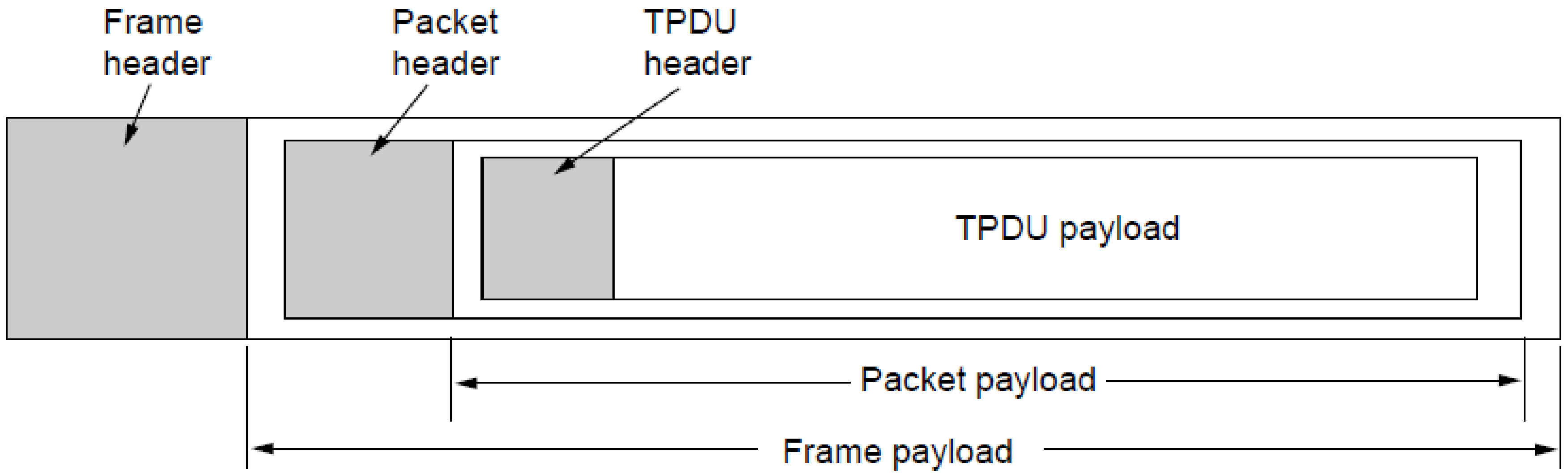
Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

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)

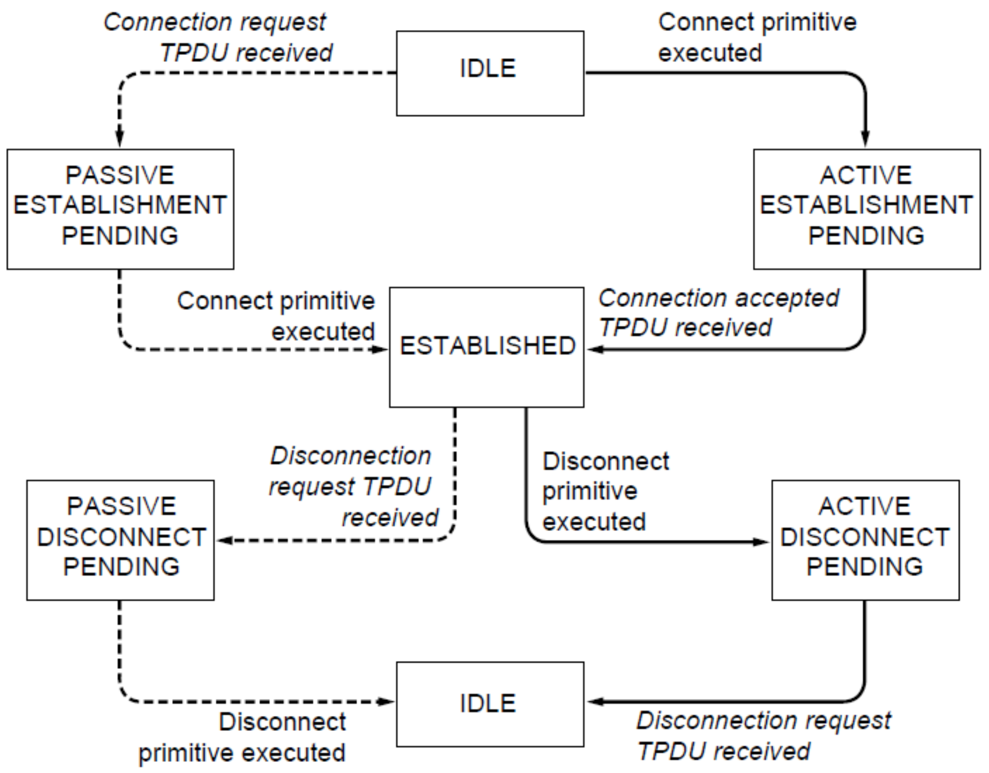


Nesting of TPDUs(segments), packets, and frames.

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.

Berkeley Sockets (1)



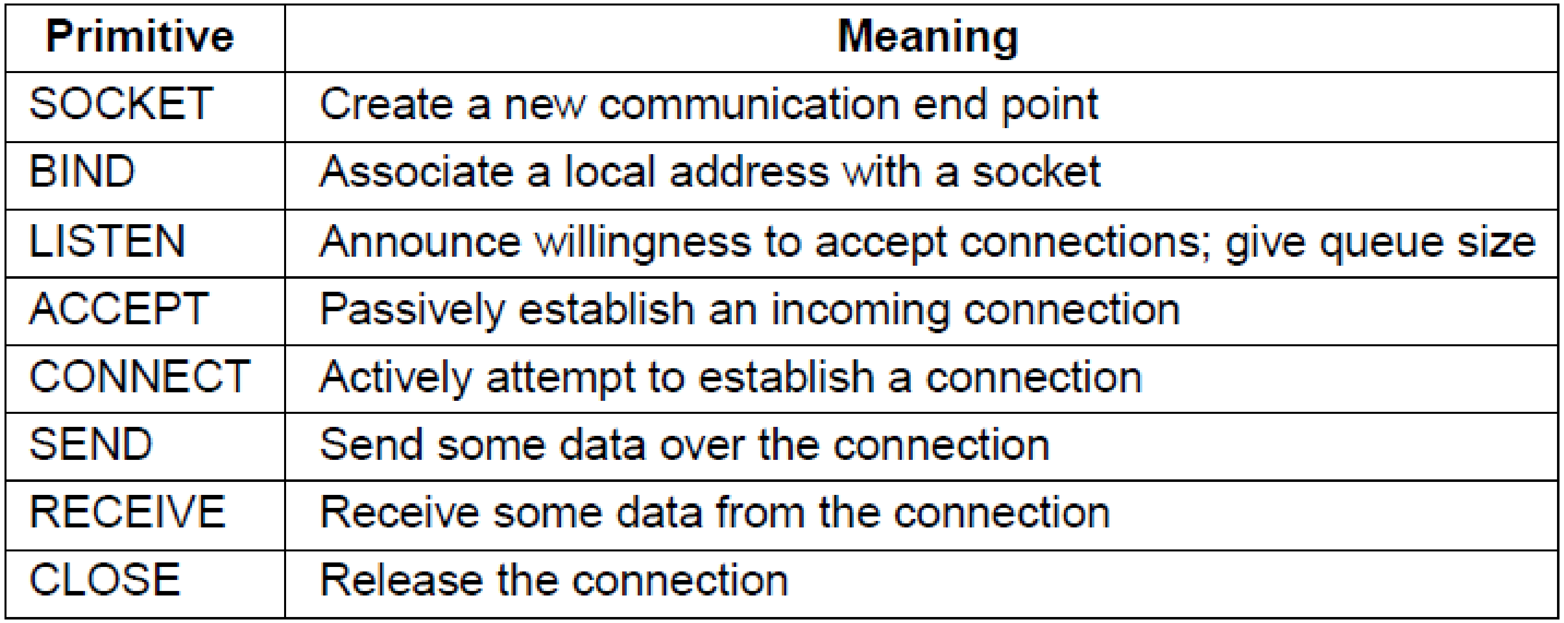
IDLE
Connection request TPDU received
Connect primitive executed
PASSIVE ESTABLISHMENT PENDING
ACTIVE ESTABLISHMENT PENDING
Connect primitive executed
ESTABLISHED
Connection accepted TPDU received
Disconnection request TPDU received
Disconnect primitive executed
PASSIVE DISCONNECT PENDING
ACTIVE DISCONNECT PENDING
IDLE
Disconnect primitive executed
Disconnection request TPDU received

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.

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

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)

. . .

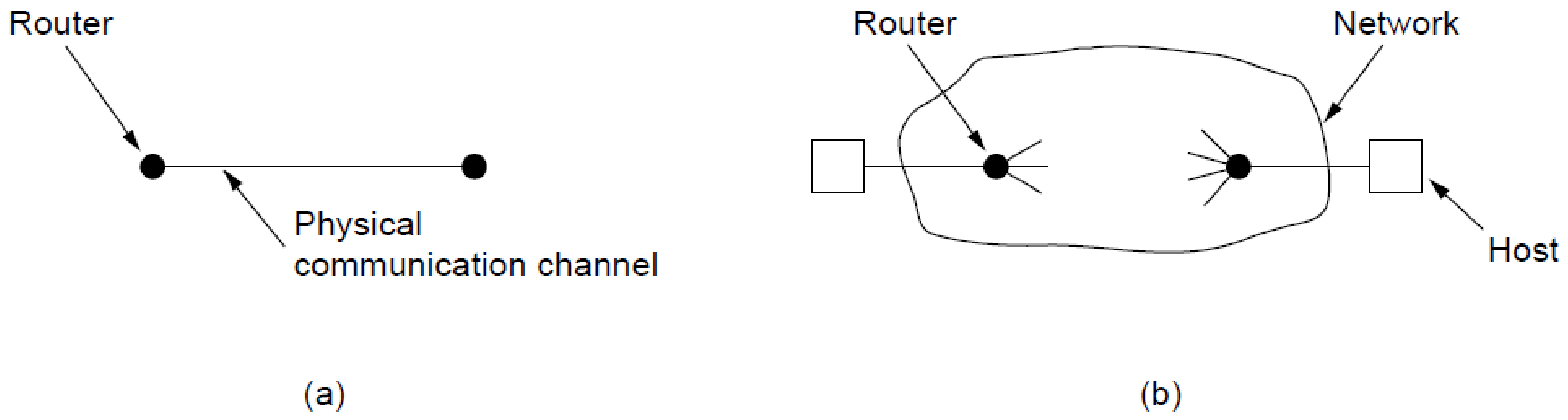
```
/* Socket is now set up and bound. Wait for connection and process it. */
while (1) {
     sa = accept(s, 0, 0);                      /* block for connection request */
     if (sa < 0) fatal("accept failed");

     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

Elements of Transport Protocols (1)

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

Elements of Transport Protocols (2)

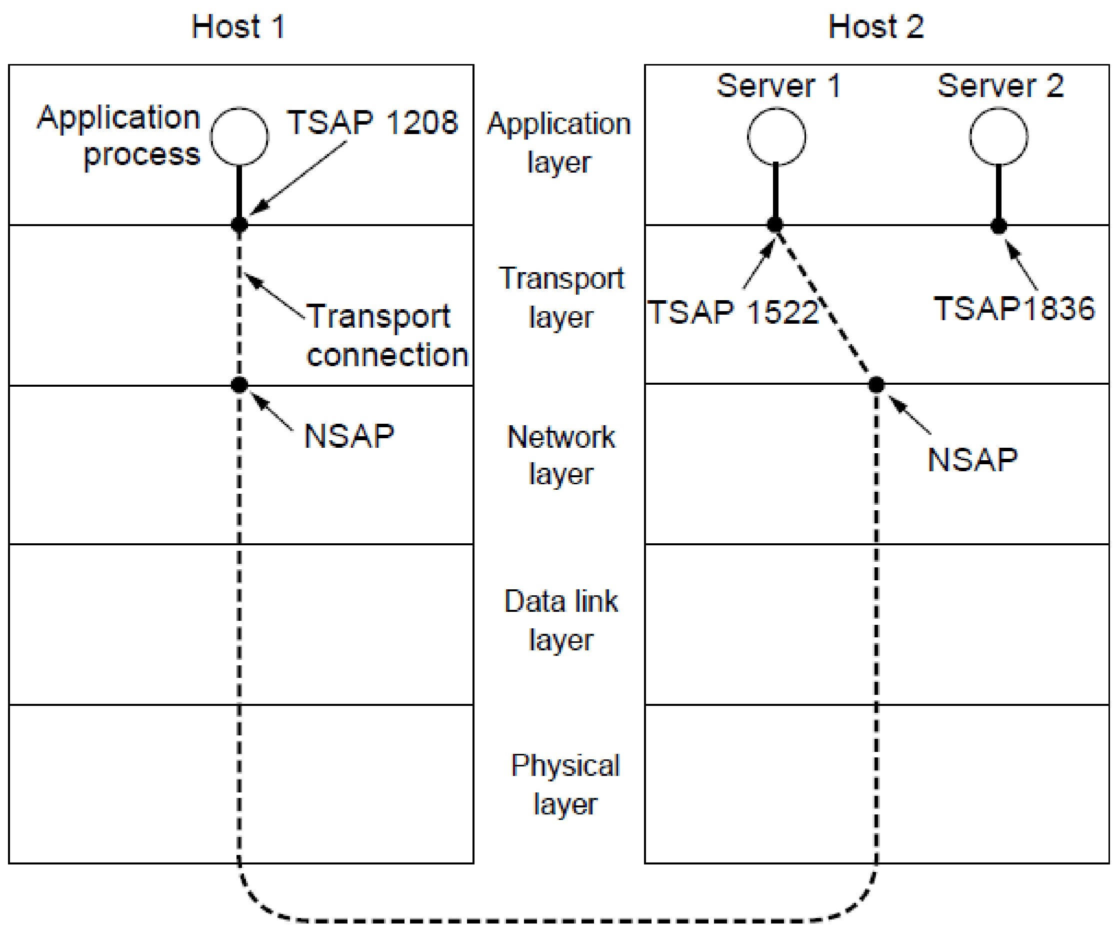


(a) Environment of the data link layer.
(b) Environment of the transport layer.

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.

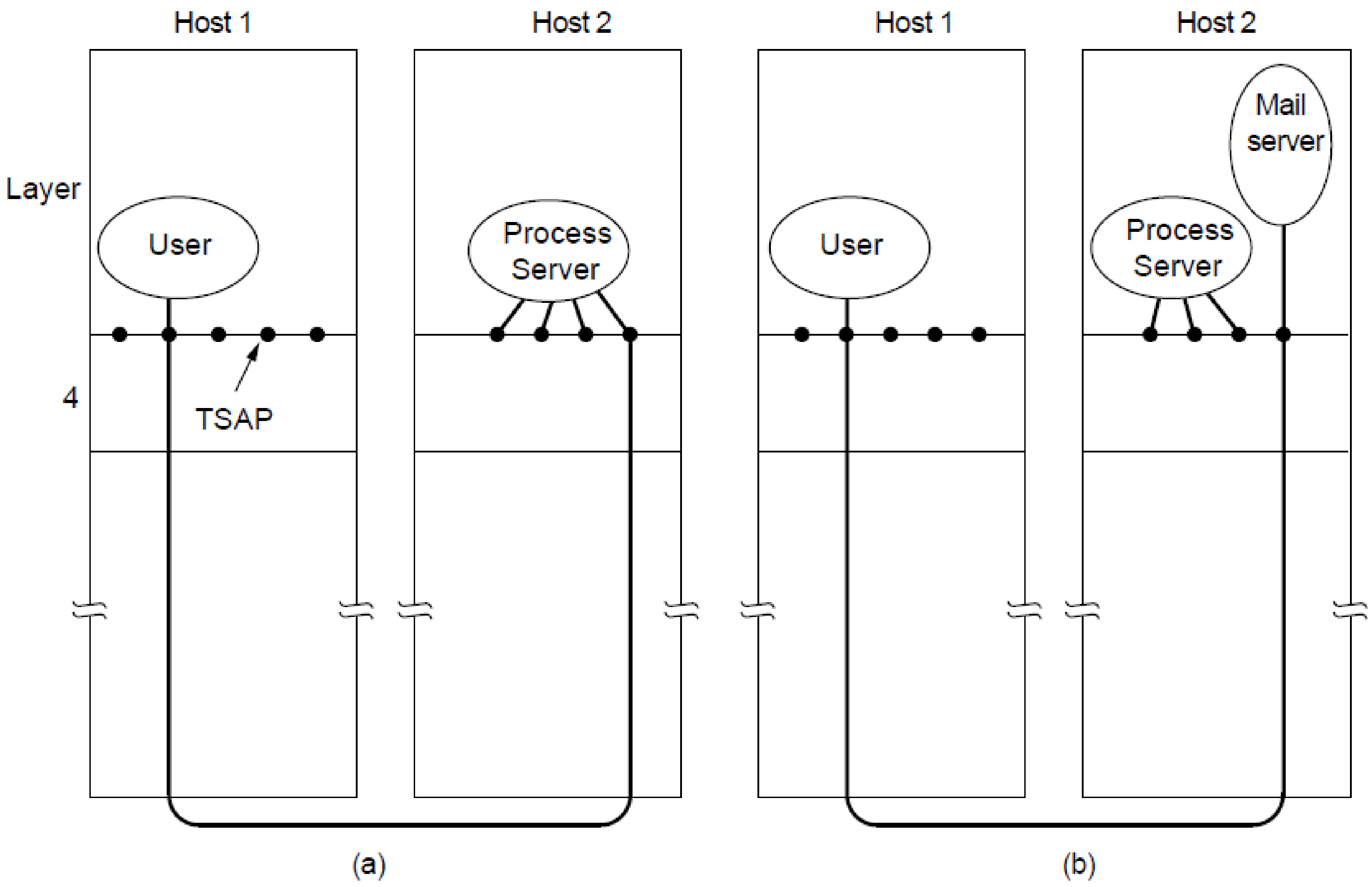
Cont.

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)



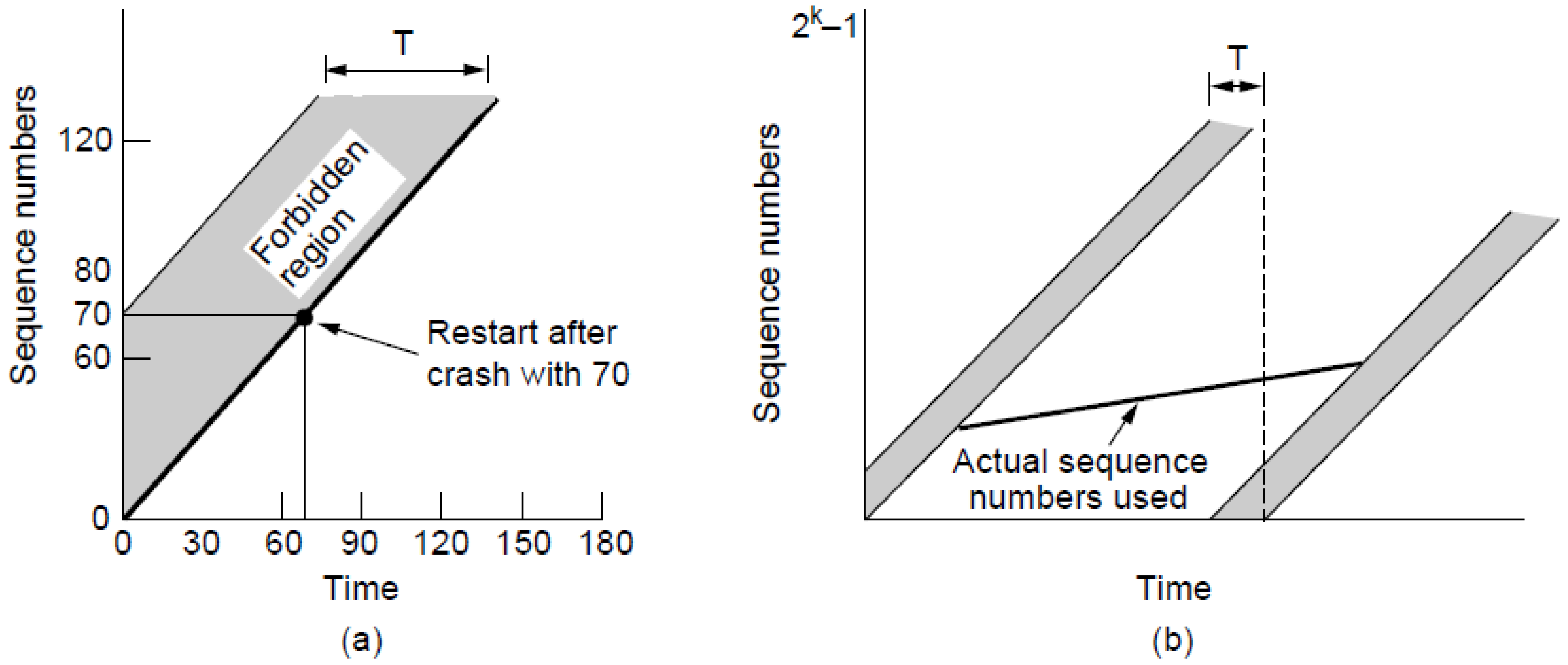
How a user process in host 1 establishes a connection with a mail server in host 2 via a process server.

Connection Establishment (1)

Techniques for restricting packet lifetime

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

Connection Establishment (2)



(a) TPDUs may not enter the forbidden region.

(b) The resynchronization problem.

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.

Cont

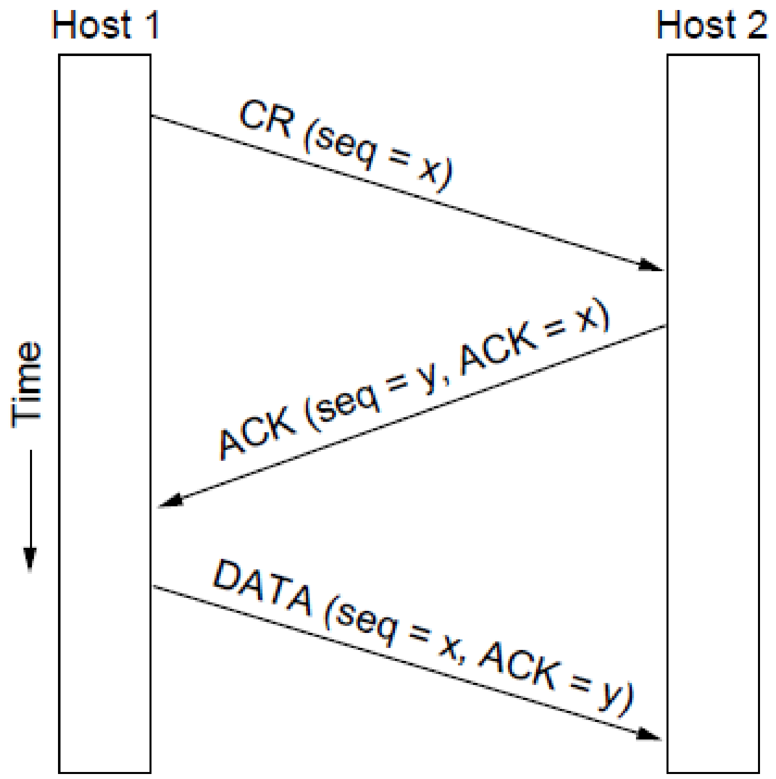
- The basic idea is to ensure that two identical numbered segments are never outstanding at the same time.
  - When a connection is set up, the low-order *k* bits of the clock are used as the initial sequence number.
  - The sequence space should be so large that by the time sequence numbers wrap around, 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)

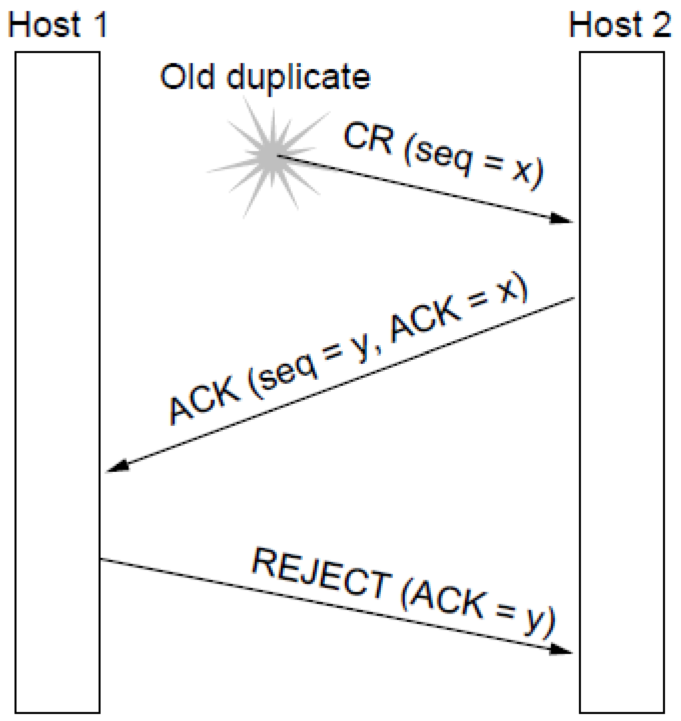


Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. Normal operation.

Cont.

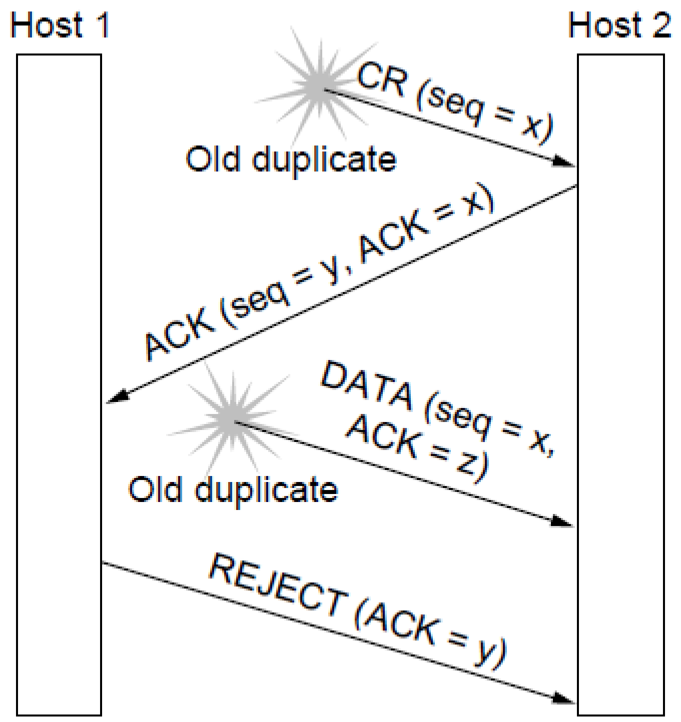
- The clock-based method, just mentioned, solves the delayed duplicate problem for data segments.
- To solve the delayed control segments, Tomlinson introduced the **three-way handshake**:
  - The normal setup procedure when host 1 initiates is shown in the figure above.
  - Host 1 chooses a sequence 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)



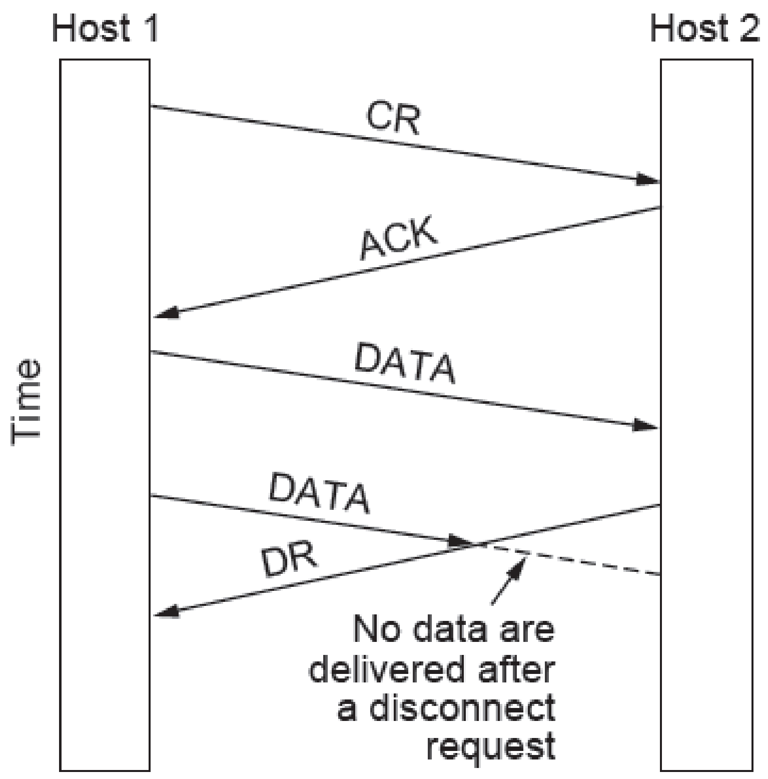
Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. Old duplicate CONNECTION REQUEST appearing out of nowhere.

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)

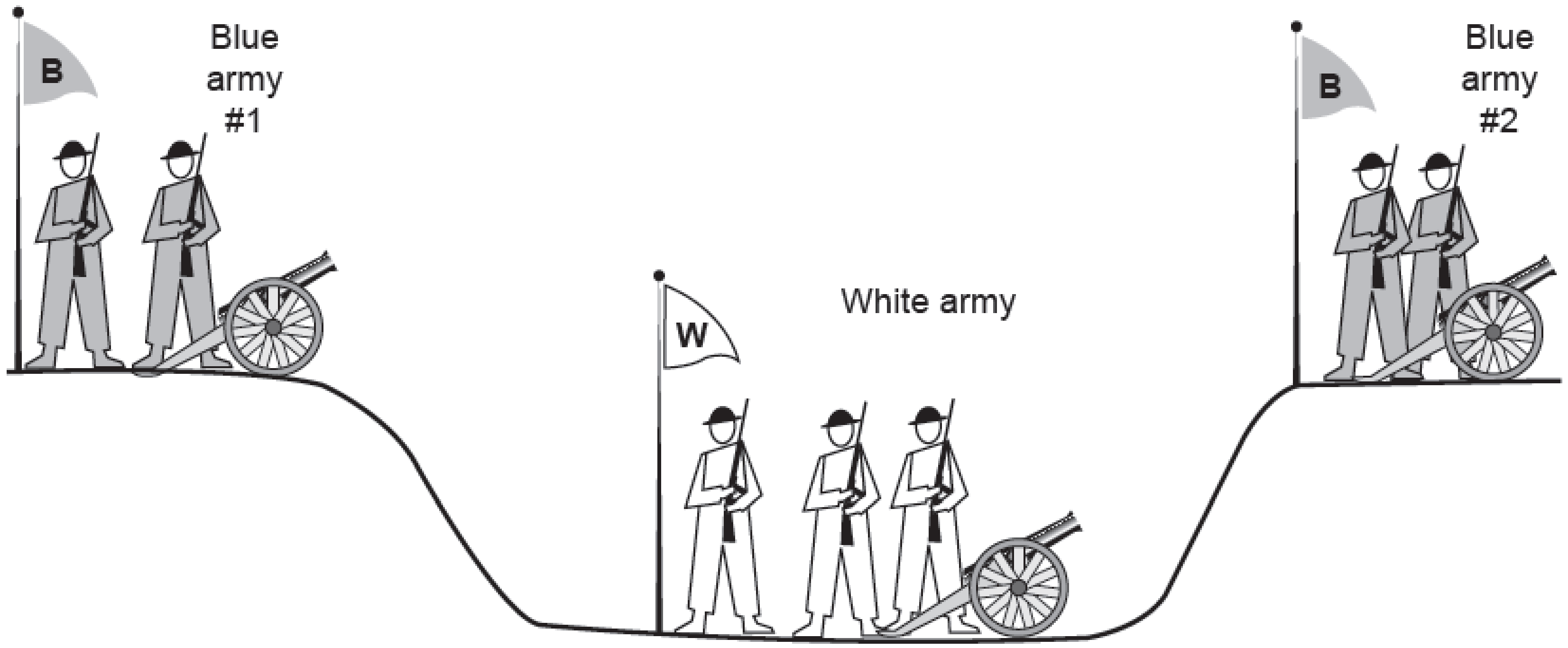


Abrupt disconnection with loss of data

Cont.

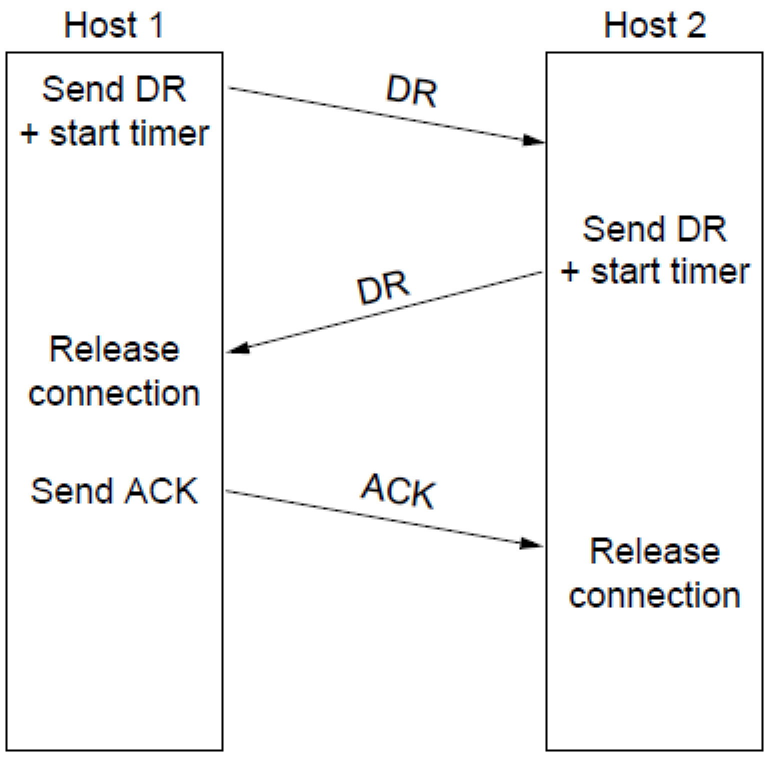
- Asymmetric release:
  - The way that the telephone system works: when one party hangs up, the connection is broken.
  - It may result in data loss, by the scenario of the previous figure.
- Symmetric release:
  - Each direction is released independently of the other one.
  - A two-army problem deals 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)



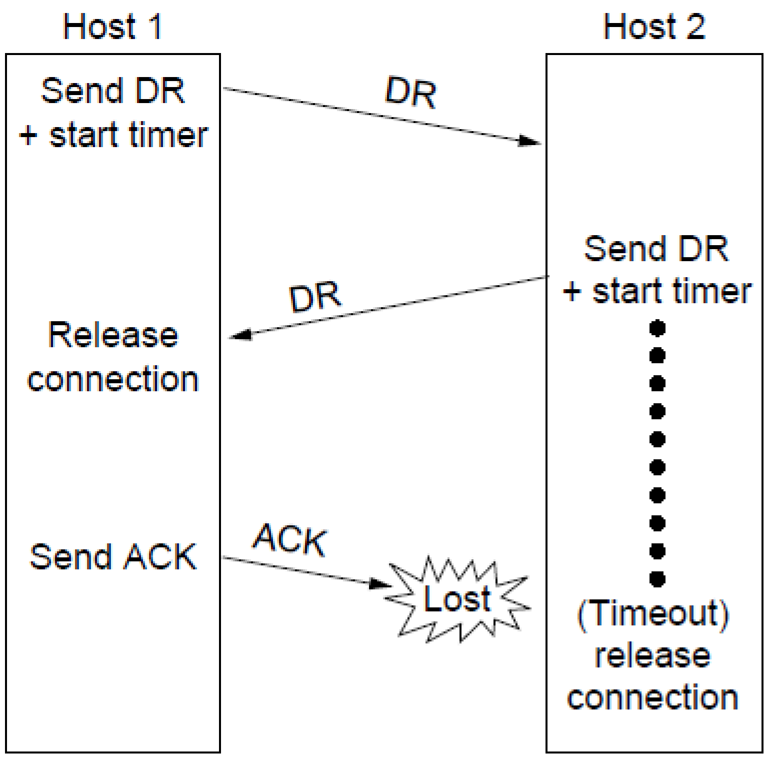
The two-army problem

Connection Release (3)



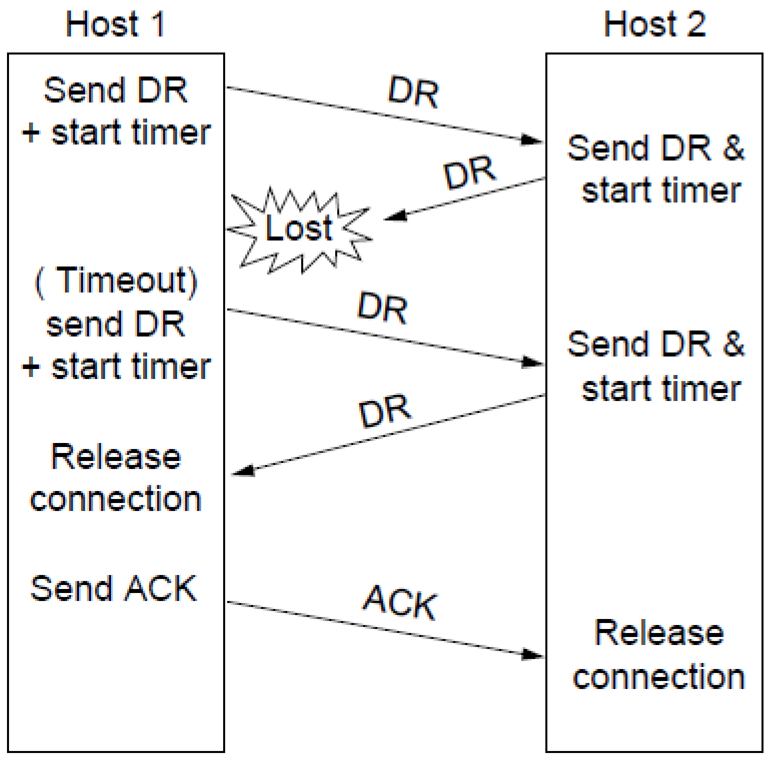
Four protocol scenarios for releasing a connection.
(a) Normal case of three-way handshake

Connection Release (4)



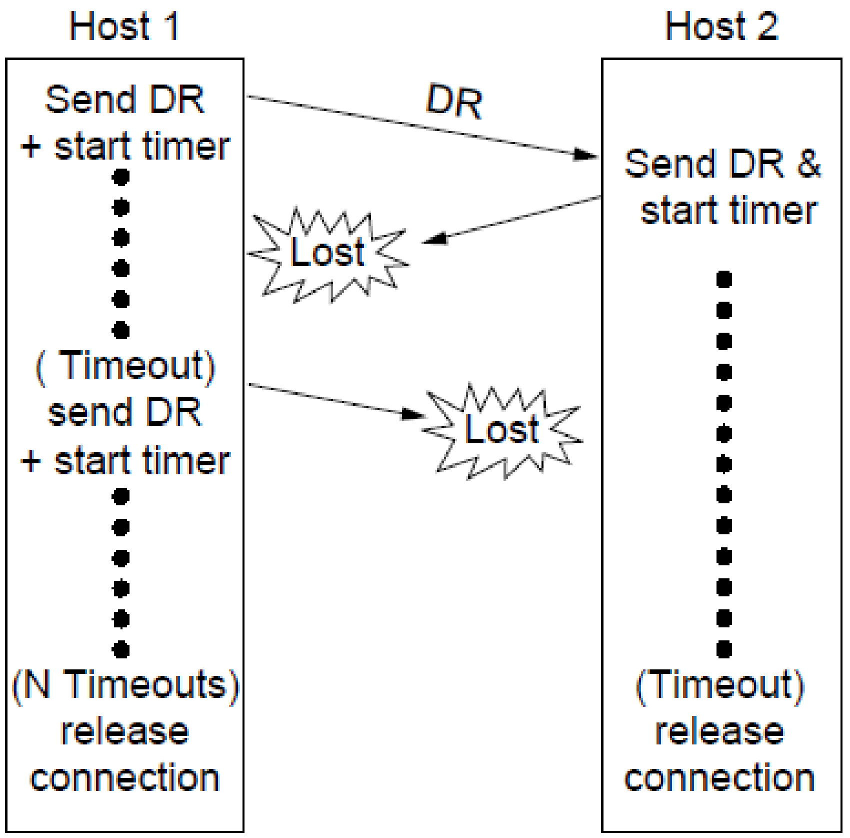
Four protocol scenarios for releasing a connection.
(b) Final ACK lost.

Connection Release (5)



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

Connection Release (6)

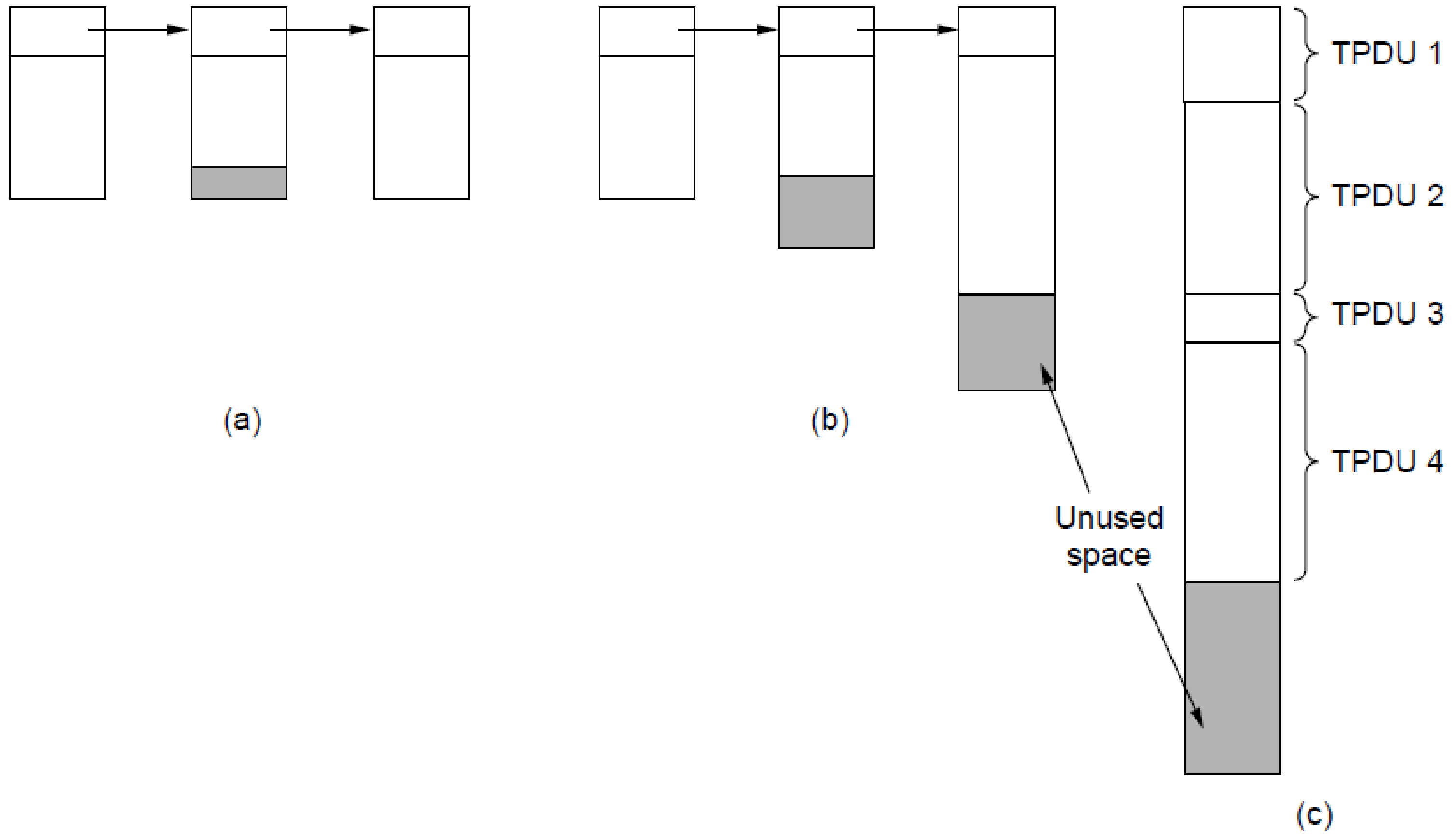


Four protocol scenarios for releasing a connection.
(d) Response lost and subsequent DRs lost.

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, …)

Error Control and Flow Control (1)



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

Cont.

- The basic similarity in the data link layer and the transport layer is that a sliding window or other scheme is needed on each connection to keep a fast transmitter from overrunning a slow receiver.
  - Window size ~ Bandwidth-delay product
- The main difference is that a router usually has a 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.

Error and flow control at layer II

1. A frame carries an error detecting code (e.g., a CRC or checksum) that is used to check if the information was correctly received.
2. A frame carries a sequence number to identify itself and is retransmitted by the sender until it receives an acknowledgement of successful receipt from the receiver, called ARQ(Automatic Repeat reQuest).
3. There is a maximum number of frames that the sender will allow to be outstanding at any time, pausing if the receiver is not acknowledging frames quickly enough.
   - Stop-and-wait
   - Go-back-N
   - Select-repeat
4. The sliding window protocol combines these features and is also used to support bidirectional data tranfer.

Cont.

- If the receiver knows that the sender buffers all segments until they are acknowledged, the receiver may or may not dedicate specific buffers to specific connections.
  - The receiver may maintain a single buffer pool shared by all connections.
  - When a segment comes in, an attempt is made to dynamically acquire a new buffer.
    - If one is available, the segment is accepted; otherwise, it is discarded.
- If the network service is unreliable, or if the receiver can not guarantee that every incoming segment will be accepted, the sender must buffer all segments sent.
  - In the latter case, the sender can not trust the network layer's acknowledgement, because *the acknowledgement means only that the segment arrived, not that it was accepted.*

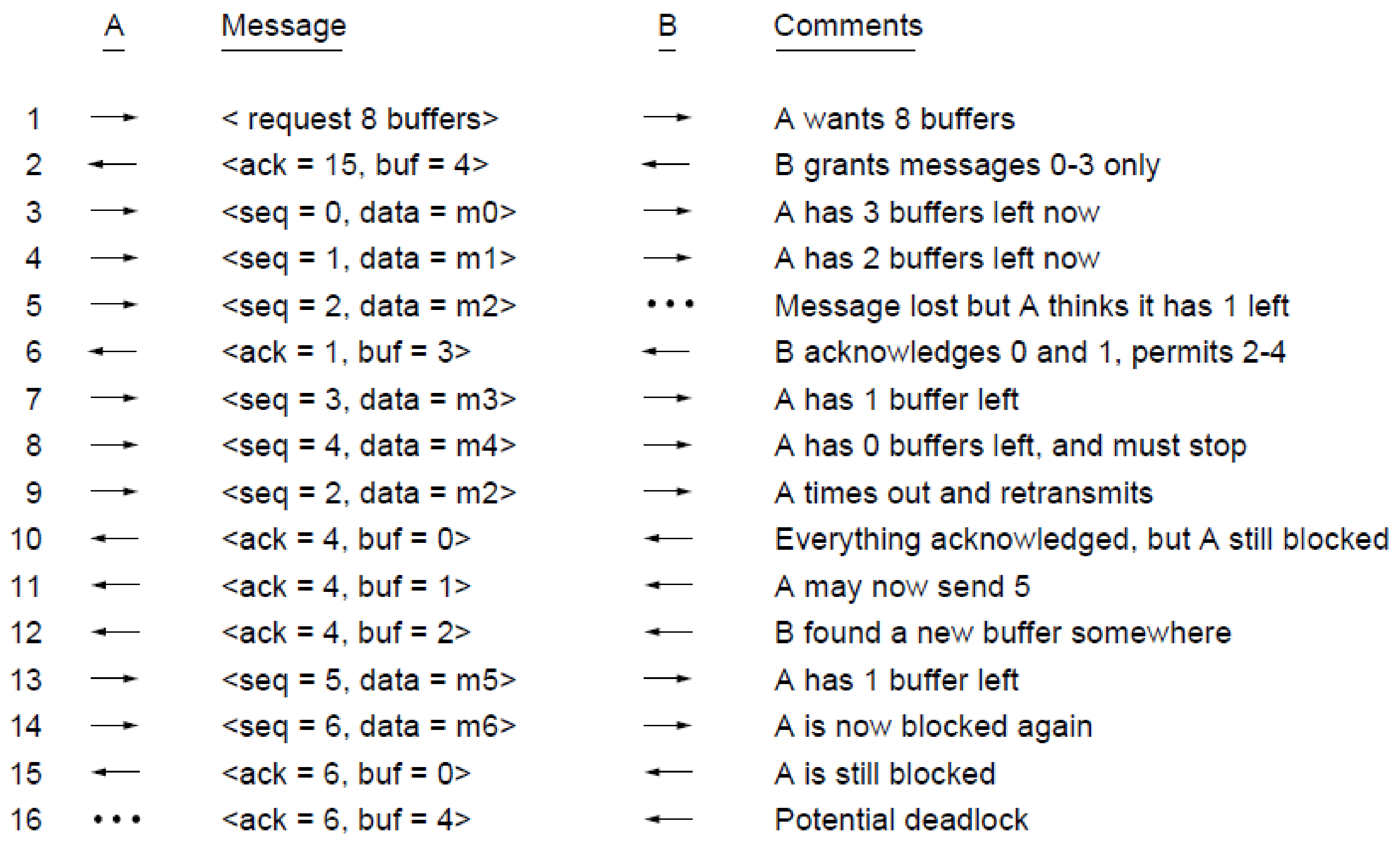
Cont.

- With reliable network service, other trade-offs become possible.
  - If the sender knows that the receiver always have buffer space, it need not retain copies of the segments it sends.
- On the buffer size at the receiver.
  - Fixed size buffers:
    - If most segments are nearly the same size, it is natural to organize the buffers as a pool of identical size buffers, with one segment per buffer.
    - If there is a wide variation in segment size: If the buffer is chosen equal to the largest possible segment, space will be wasted whenever a short segment arrives. If the buffer size is chosen less than the maximum segment size, multiple buffers will be needed for long segments, with the attendant complexity.
  - Variable-size buffers:
    - It gets better memory utilization, at the price of more complicated buffer management.
  - A single large circular buffer per connection:
    - It makes good use of memory, provided that all connections are heavily loaded but is poor if some connections are lightly loaded.

Cont.

- The optimal trade-off between source-buffering and destination-buffering depends on the type of traffic carried by the connection.
  - For low-bandwidth bursty traffic,
    - It is better not to dedicate any buffers, but rather to acquire them dynamically at both end.
    - It is better to buffer at the sender.
      - The sender must retain a copy of the segment until it is acknowledged.
  - For high-bandwidth smooth traffic, it is better to buffer at the receiver.
    - To allow the data to flow at maximum speed.

Error Control and Flow Control (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

Cont.

- Dynamic buffer allocation:
  - As connections are opened and closed, and as traffic pattern changes, the sender and receiver need to dynamically adjust their buffer allocations.
  - Dynamic buffer management means a variable-sized window:
    - Initially, the sender requests a certain number of buffers, based on its perceived need.
    - The receiver then grants as many of these as it can afford.
    - Every time the sender transmits a segment, it must decrement its allocation, stopping altogether when the allocation reaches zero.
    - The receiver then piggybacks both acknowledgements and buffer allocations onto the reverse traffic.

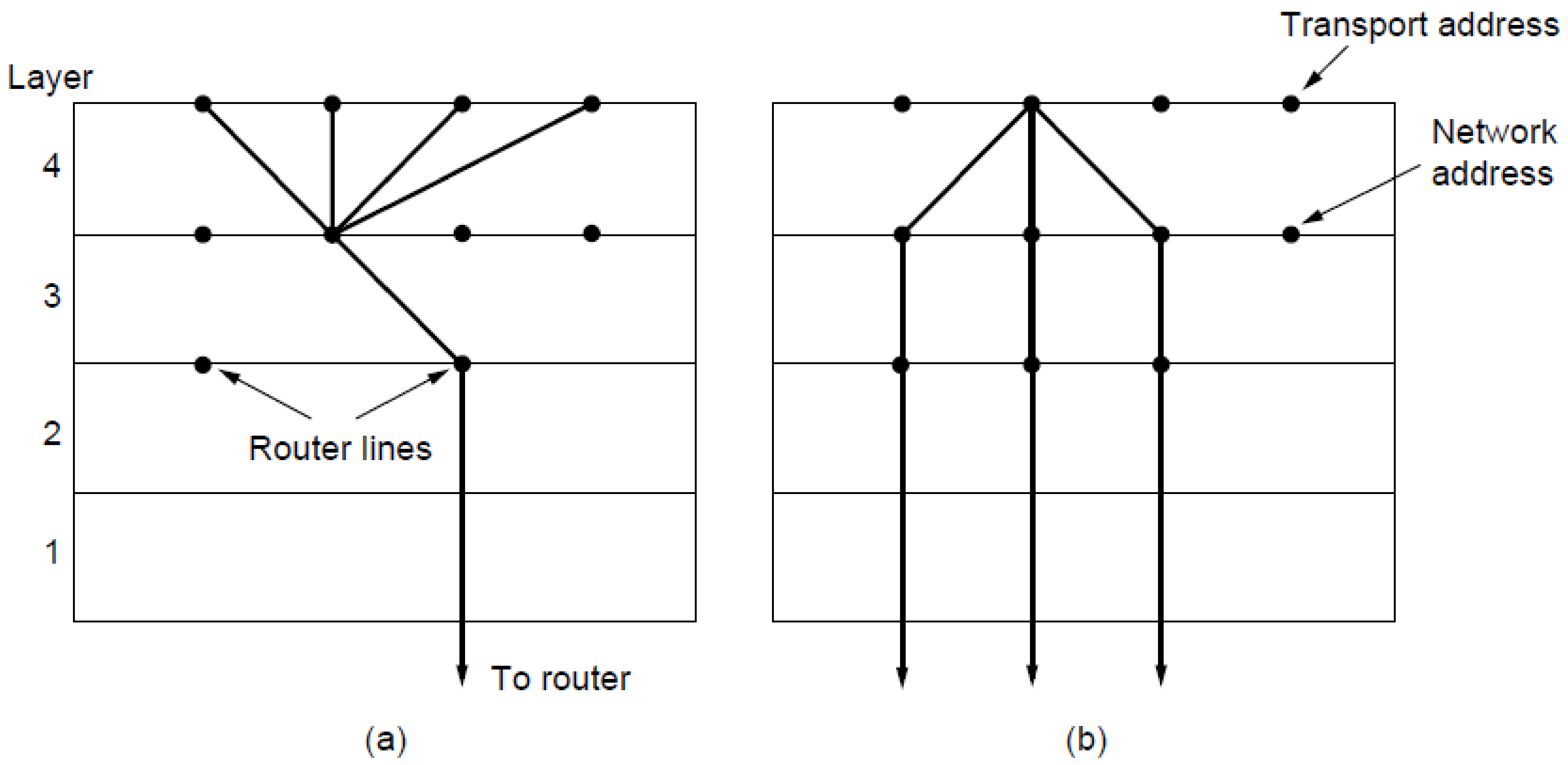
Cont.

- The previous figure shows an example of dynamic buffer allocation with 4-bit sequence numbers.
  - Assume that buffer allocation information travels in separate segments, and is not piggybacked onto reverse traffic.
  - Potential problems can arise in datagram networks if control segments can get lost. Check line 16 in the figure.
    - To prevent the deadlock situation, each host should periodically send control segments giving the acknowledgement and buffer status on each connection.

Cont.

- When buffer space no longer limits the maximum flow, another bottleneck will appear: *the carrying capacity of the subnet.*
  - The flow control mechanism must be applied at the sender to prevent it from having too many unacknowledged segments outstanding at once.
  - A dynamic sliding window implements both flow control and congestion control (by Belsnes):
    - If the network can handle *c* segments/sec and the cycle time (including transmission, propagation, queueing, processing at the receiver, and return of the acknowledgement) is *r*, then the sender window should be *cr*.
    - In order to adjust the window size periodically, the sender could monitor both parameters and then compute the desired window size.

Multiplexing



(a) Multiplexing. (b) Inverse multiplexing.

Cont.

- **Multiplexing** : *different transport connections use the same network connection* .
- **Inverse multiplexing**: The form is to have the transport layer open multiple network connections and distribute the traffic among them on a round-robin basis.
  - With *k* network connections open, the effective bandwidth is increased by a factor of *k*.
  - SCTP (Stream Control Transmission Protocol), an example of inverse multiplexing, can run a connection using multiple network interfaces.

Crash Recovery

	Strategy used by receiving host					
	First ACK, then write			First write, then ACK		
Strategy used by sending host	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK	= Protocol functions correctly
DUP	= Protocol generates a duplicate message
LOST	= Protocol loses a message

Different combinations of client and server strategy

Cont.

- If the transport entity is entirely within hosts, *recovery from network and router crashes* can be easily done as follows:
  - If the network layer provides datagram service, the transport entities expect lost segments all the time and know how to cope with them.
  - 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 segments it has received and which ones it has not received. The latter one can be retransmitted.

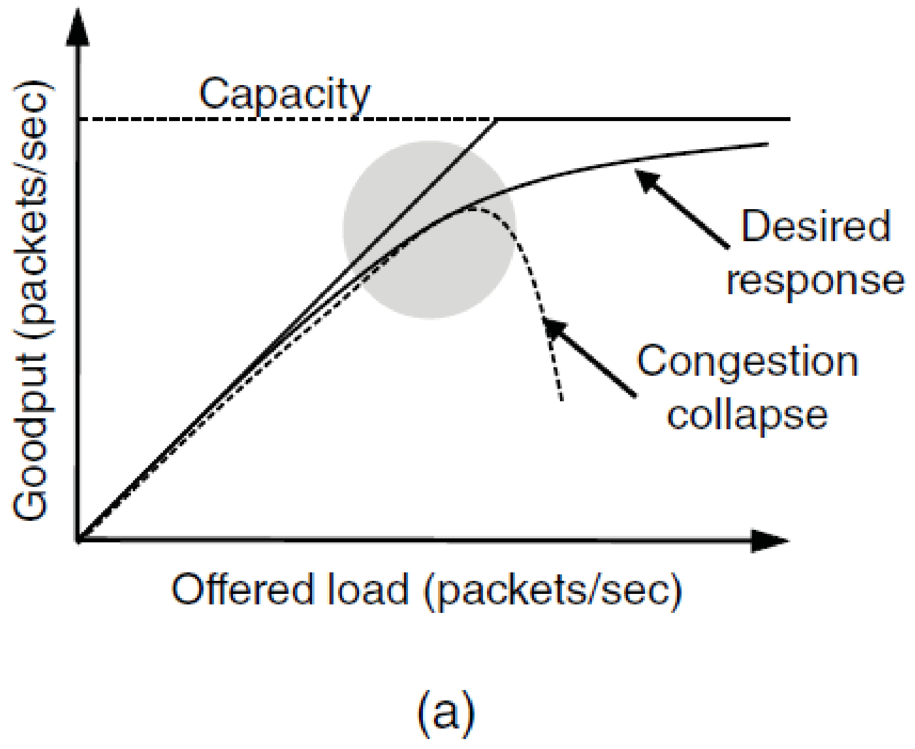
Cont.

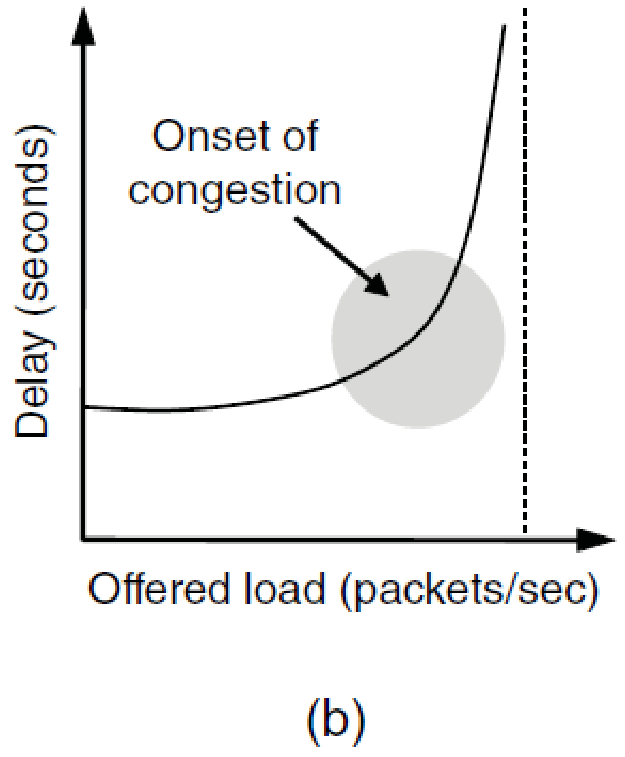
- A more troublesome problem is how to recover from host recovery.
  - Writing a segment onto the output stream (to the application process) and sending an acknowledgement are two distinct events that cannot be done simultaneously. A host may crash between two events.
  - The previous figure shows all eight combinations of client and server strategy and the valid event sequences for each one.
    - A, sending an acknowledgement; W, write to the output process; C, crashing.
    - Notice that for each strategy there is some sequence of events that causes the protocol to fail.
- Without simultaneous events, host crash and recovery cannot be made transparent to the higher layers.
  - Recovery from a layer *N* crash can only be done by layer *N*+1, and then only if the higher layer retains enough status information.

Congestion Control

- Desirable bandwidth allocation
- Regulating the sending rate

Desirable Bandwidth Allocation (1)





(a) Goodput and (b) delay as a function of offered load

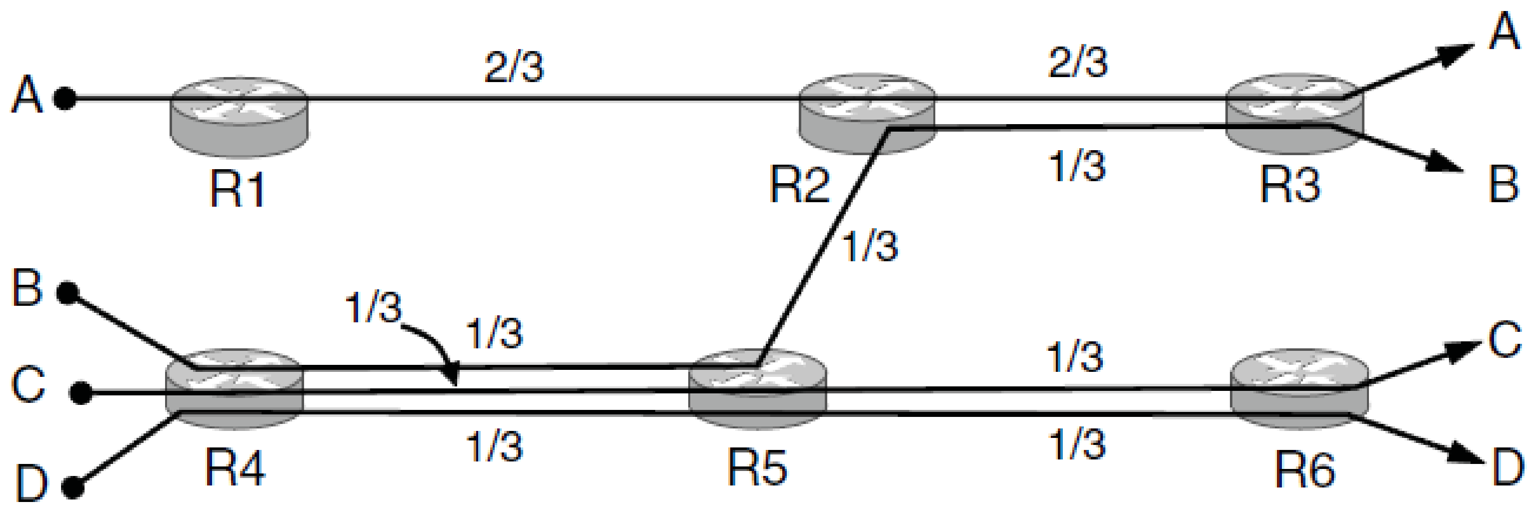
Desirable bandwidth allocation

- Efficiency and Power
  - For both goodput and delay, performance begins to degrade at the onset of congestion
  - Bandwidth should be allocated up until the delay starts to climb rapidly. This point can be identified by Kleinrock's ***power***:

    Power=load/delay

  - Power will initially rise with offered load, but will reach a maximum and fall as delay grows rapidly.
- Max-min fairness
  - Divide bandwidth between different transport senders.
  - Fairness versus efficiency
  - *An allocation is max-min fair if the bandwidth given to one flow cannot be increased without decreasing the bandwidth given to another flow with an allocation that is no larger.*

Desirable Bandwidth Allocation (2)



$$\max \min\{x_A, x_B, \cdots, x_D\}$$

subject to the constraint of link flow capacicity :

$$x_B + x_C + x_D \le f_{45}$$

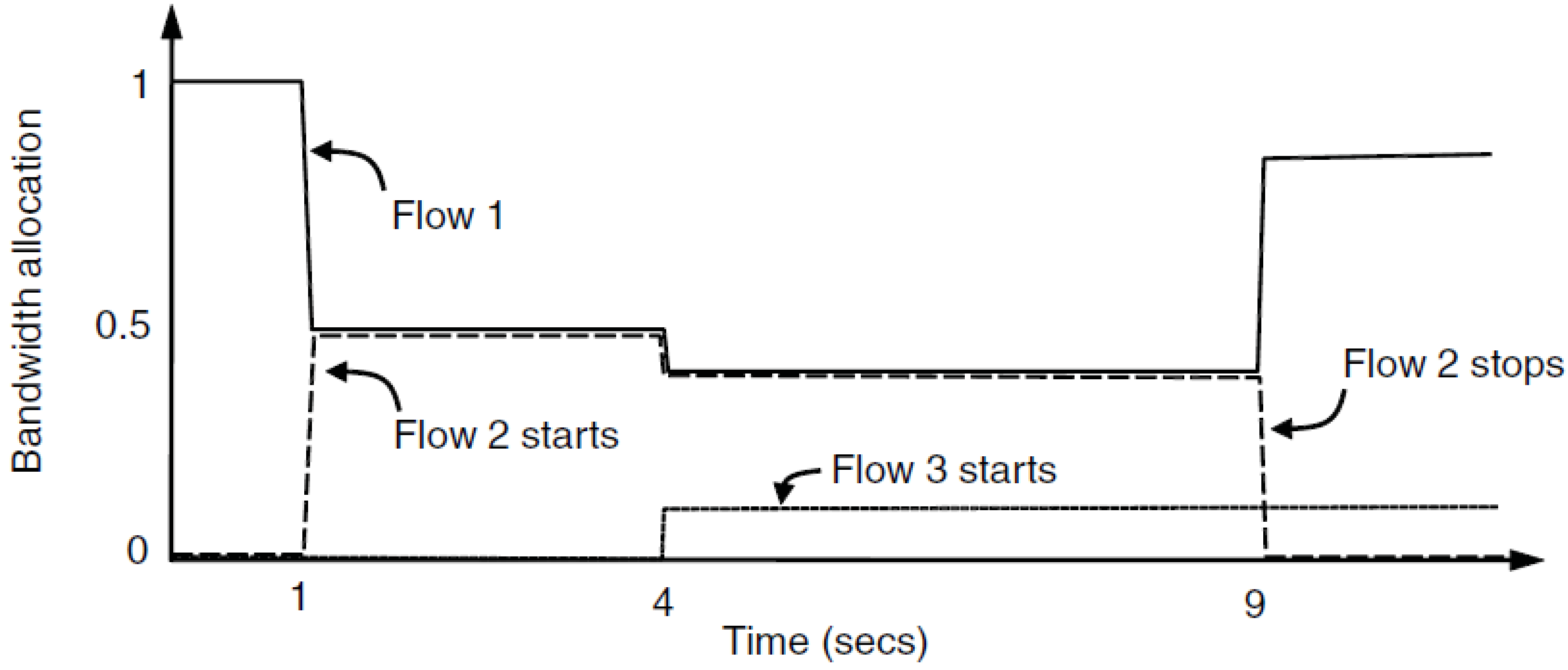
$$x_C + x_D \le f_{56}$$

$$x_A + x_B \le f_{23}$$

$$x_A \le f_{12}$$

Max-min bandwidth allocation for four flows

Desirable Bandwidth Allocation (3)

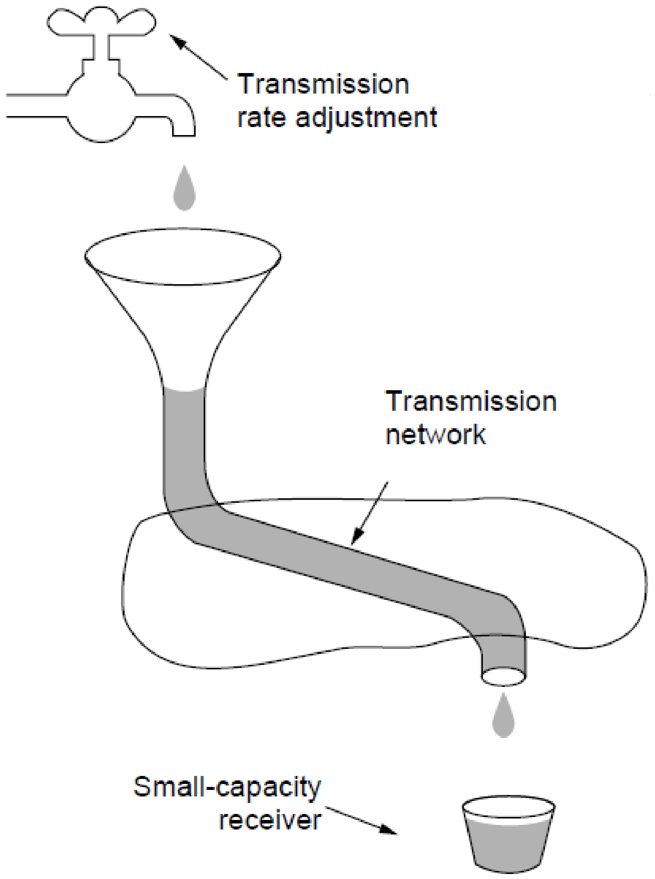


Changing bandwidth allocation over time

Regulating the sending rate

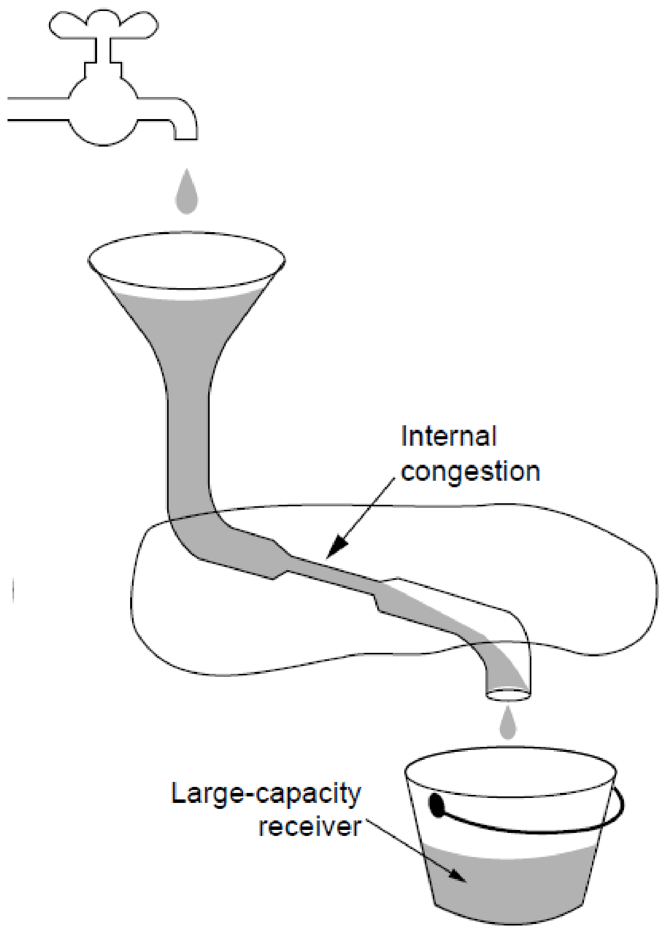
- Two factors limit the sending rate:
  - Flow control
  - Congestion
- Depend on the form of the feedback returned by the network:
  - Explicit or implicit
  - Precise or imprecise
- Control law for rate change(increase or decrease)
  - **AIMD** (Additive Increase Multiplicative Decrease) is an appropriate control law for approaching the *efficient* and *fair* operating point.

Regulating the Sending Rate (1)



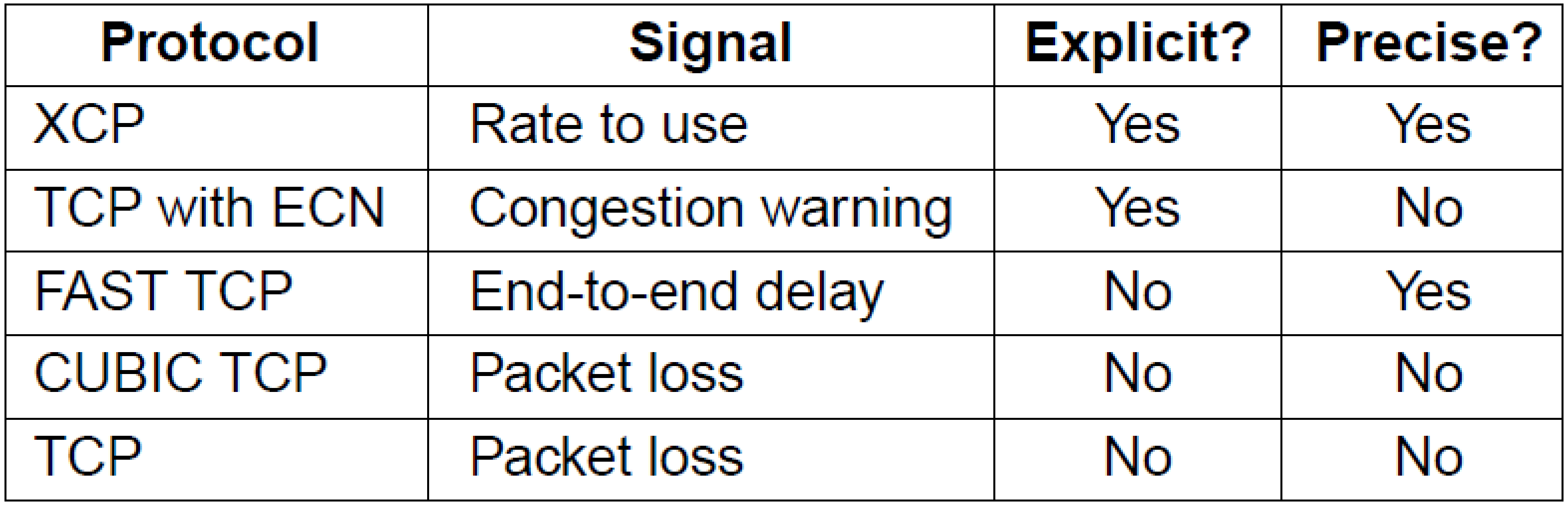
A fast network feeding a low-capacity receiver

Regulating the Sending Rate (2)



A slow network feeding a high-capacity receiver

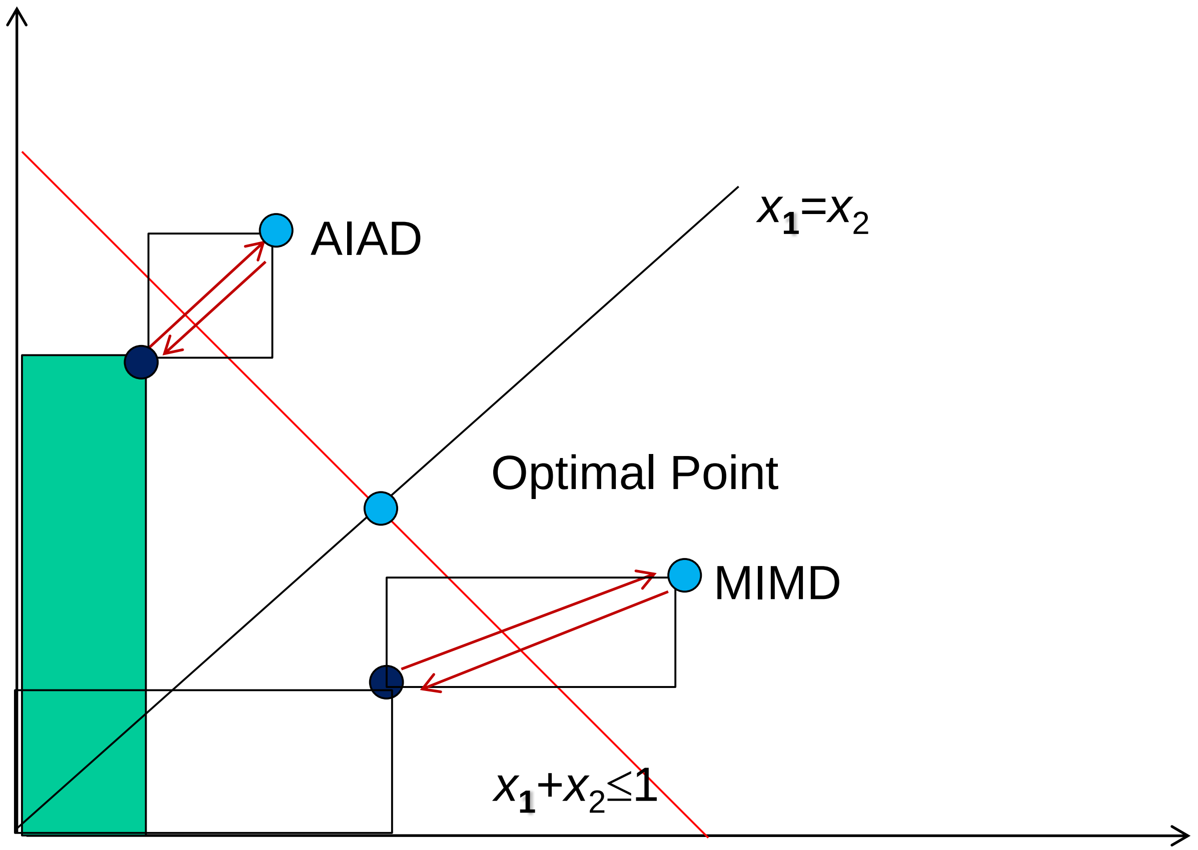
Regulating the Sending Rate (3)



Protocol	Signal	Explicit?	Precise?
XCP	Rate to use	Yes	Yes
TCP with ECN	Congestion warning	Yes	No
FAST TCP	End-to-end delay	No	Yes
CUBIC TCP	Packet loss	No	No
TCP	Packet loss	No	No

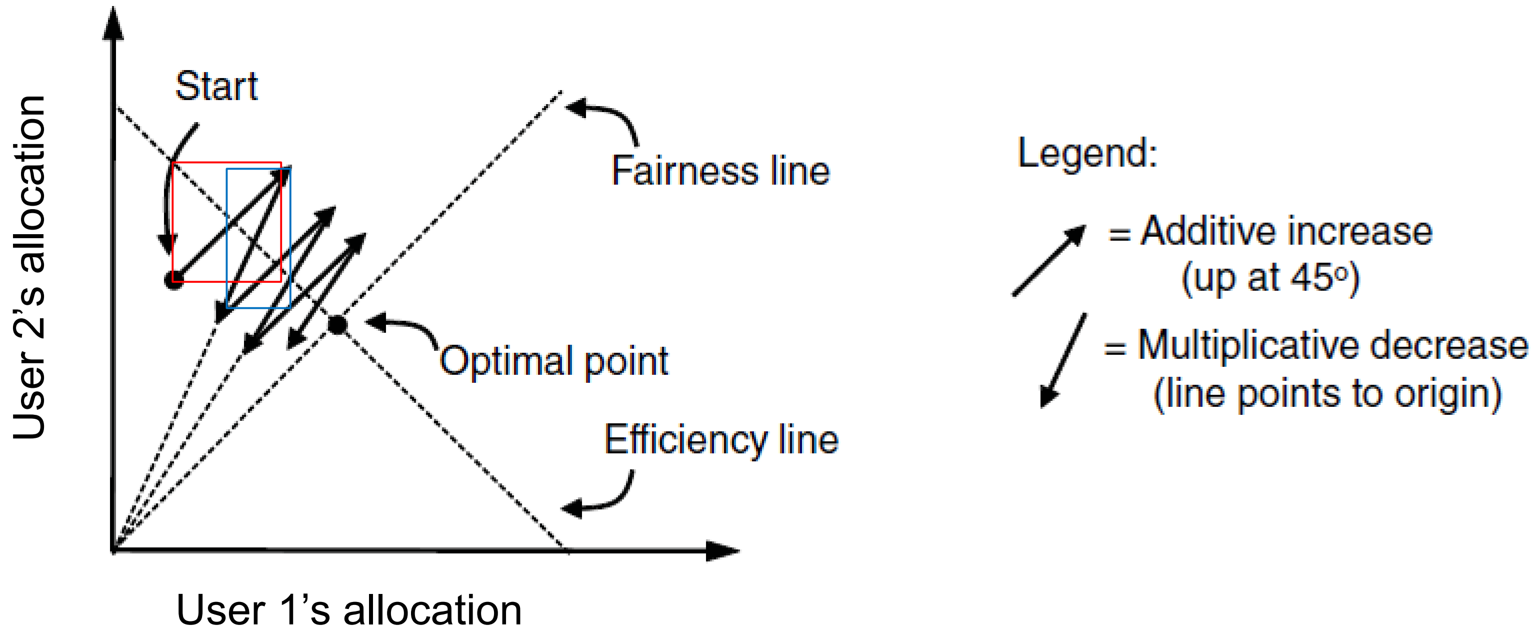
Some congestion control protocols

Control laws



Both AIAD and MIMD are close to efficient but not necessarily fair

Regulating the Sending Rate (4)



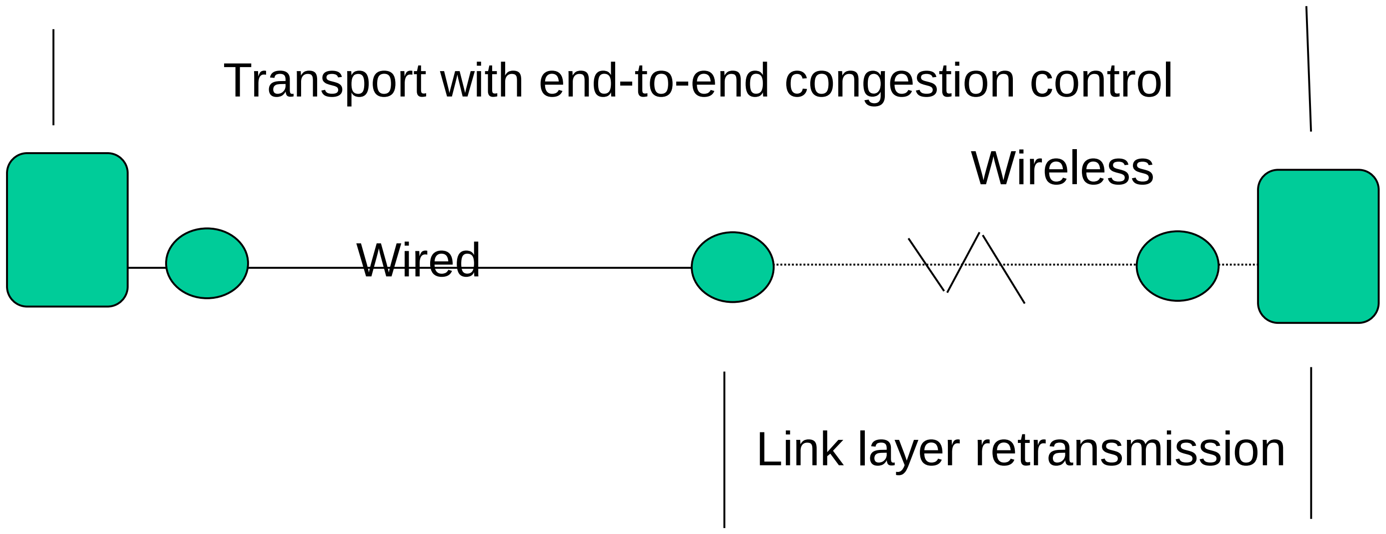
Additive Increase Multiplicative Decrease (AIMD) control law.

AIMD Converges to the optimal point VS MIAD diverges from the point.

Wireless Issue

- Wireless network lose packets all the time due to transmission errors, while packet loss is often used as a congestion signal, including TCP
  - The AIMD control law requires very small levels of packet loss.
  - One solution: Mask the wireless losses by using retransmission.
- The capacity of a wireless link changes over time.
  - Causes: mobility, interfering links, …

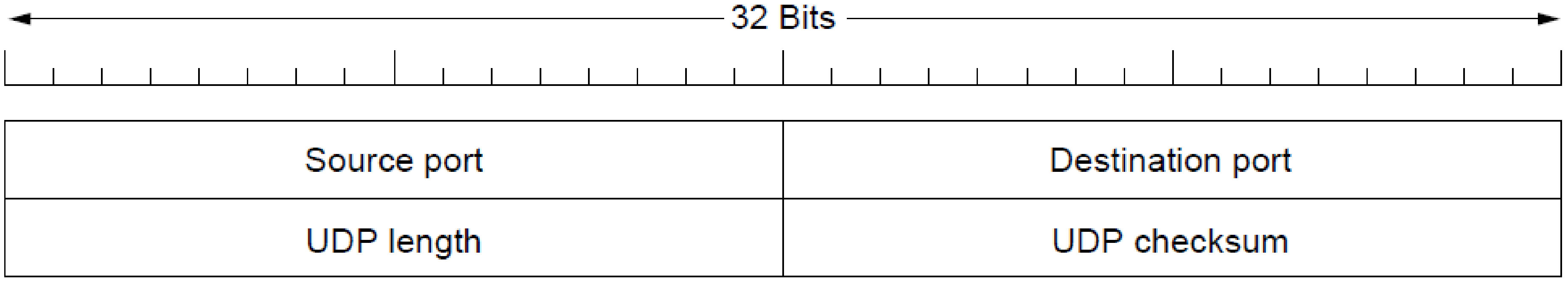
Wireless Issues


Transport with end-to-end congestion control
Wireless
Wired
Link layer retransmission

The Internet Transport Protocols: UDP

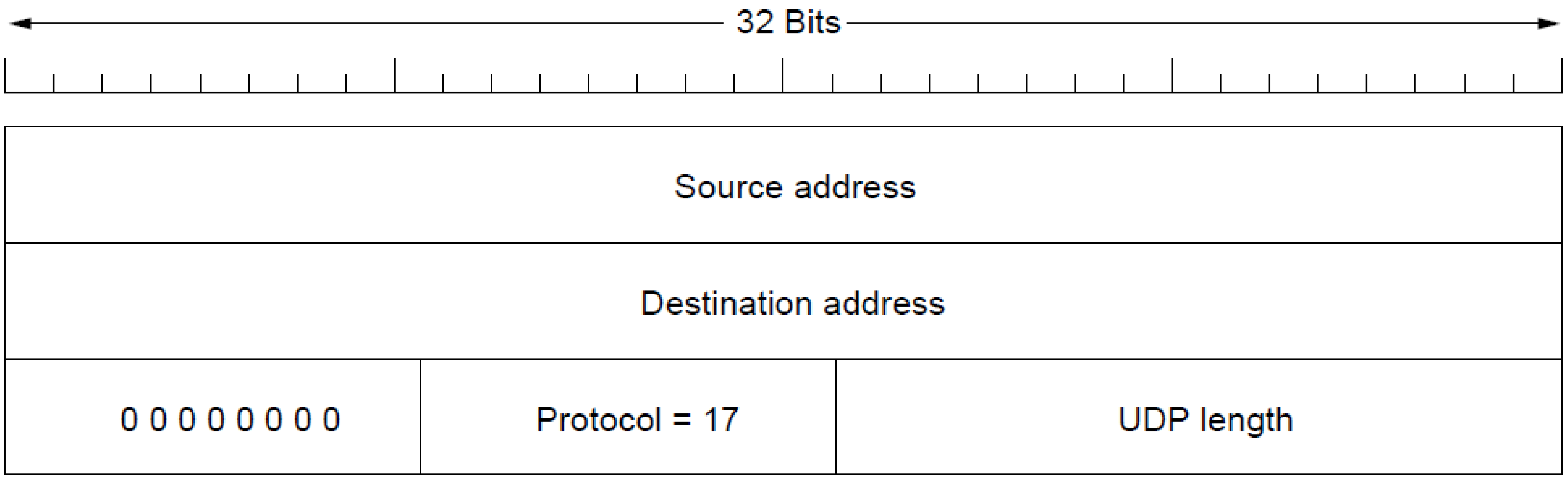
- Introduction to UDP
- Remote Procedure Call
- Real-Time Transport

Introduction to UDP (1)



The UDP header.

Introduction to UDP (2)



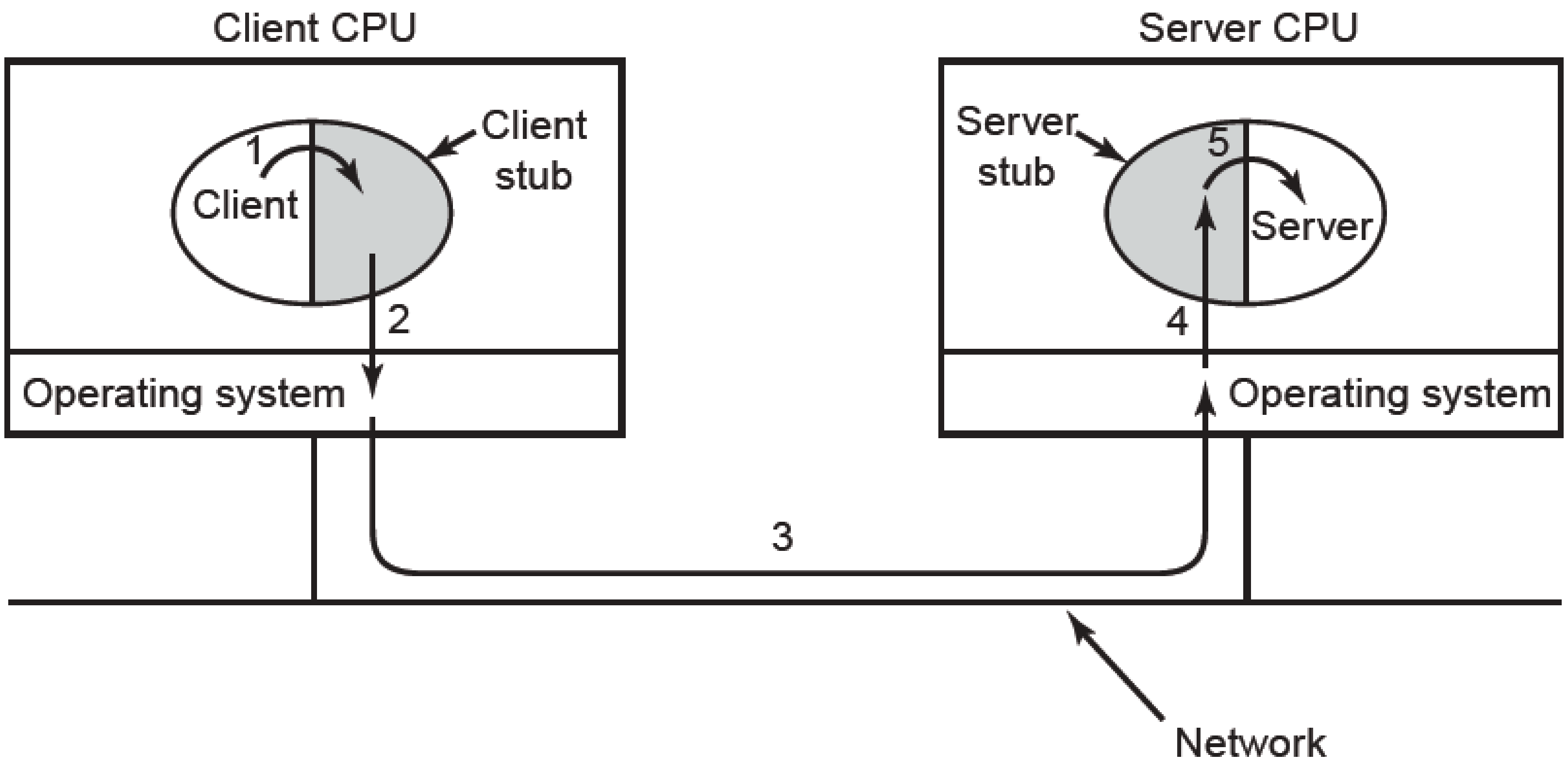
The IPv4 pseudoheader included in the UDP checksum.

UDP

- UDP segment:
  - 8-byte headers followed by the payload
    - Two ports
    - UDP length: The maximum length is 65515 bytes
    - Checksum

- UDP does not do flow control, congestion control, or retransmission upon receipt of a bad frame.
- It provides an interface to the IP protocol with the added feature of demultiplexing multiple processes.
  - RPC, RTP, RTCP,..

Remote Procedure Call



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

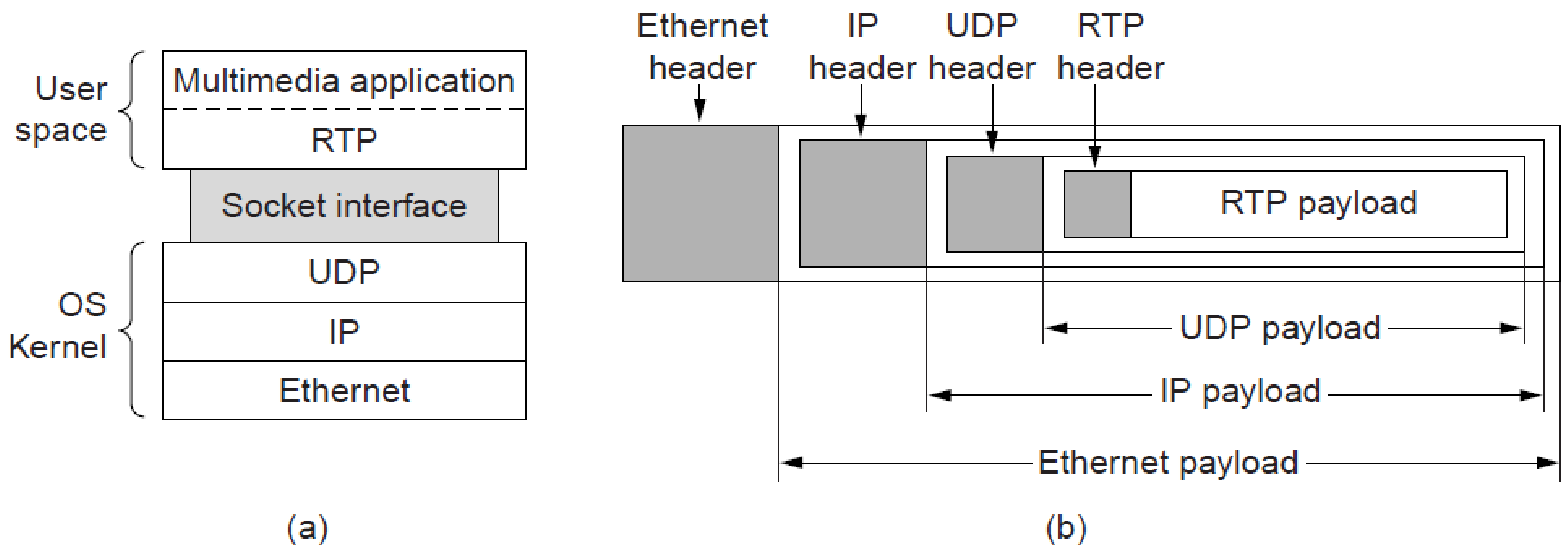
Real-Time Transport Protocol

- To multiplex several real-time data streams onto a single stream of UDP packets.
- No acknowledgements and no mechanism to request retransmissions.
  - ....
  - Payload type
  - Sequence number
  - Timestamp
  - Synchronization source identifier
  - Contributing source identifier

RTCP (Real-time Transport Control Protocol)

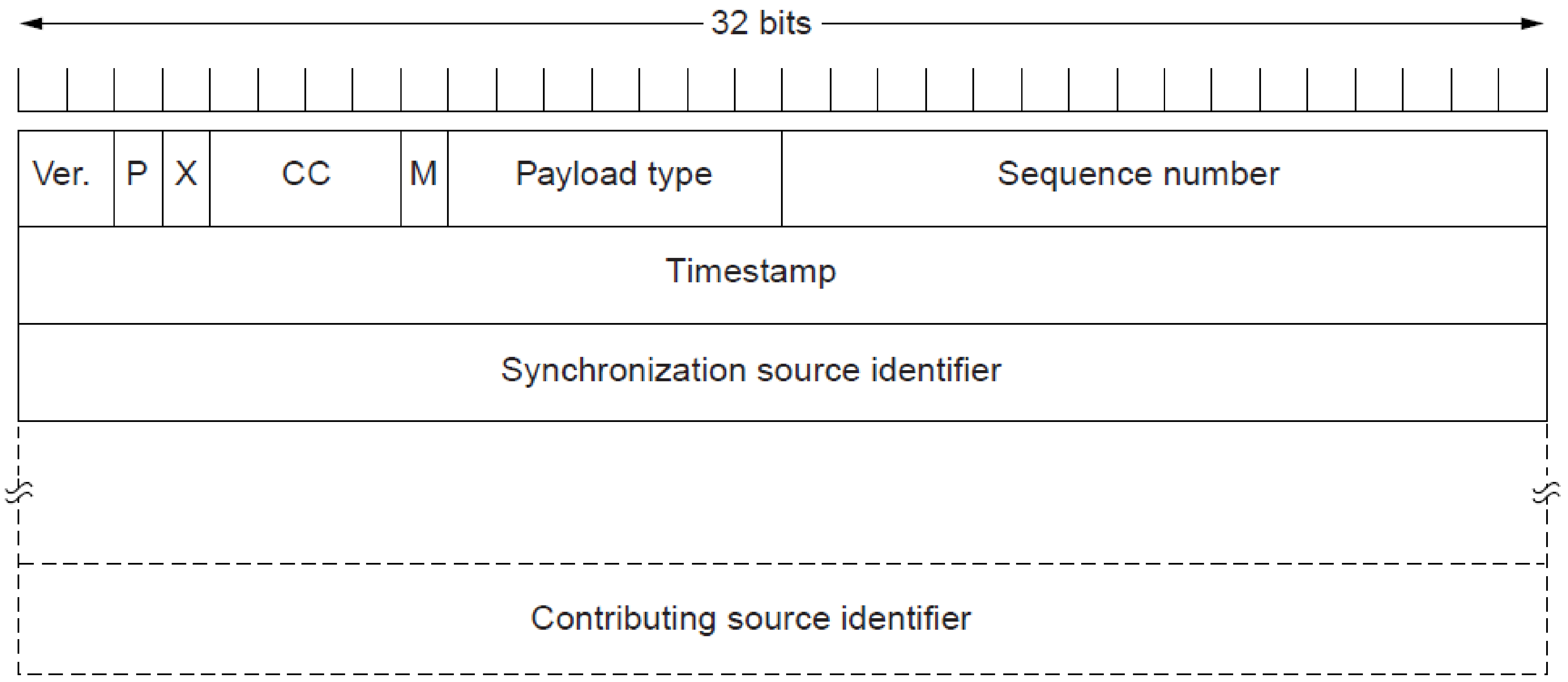
- It handles
  - Feedback
    - Delay, variation in delay or jitter, bandwidth, congestion,…..
  - synchronization,
  - the user interface.
    - Name the various sources.

Real-Time Transport (1)



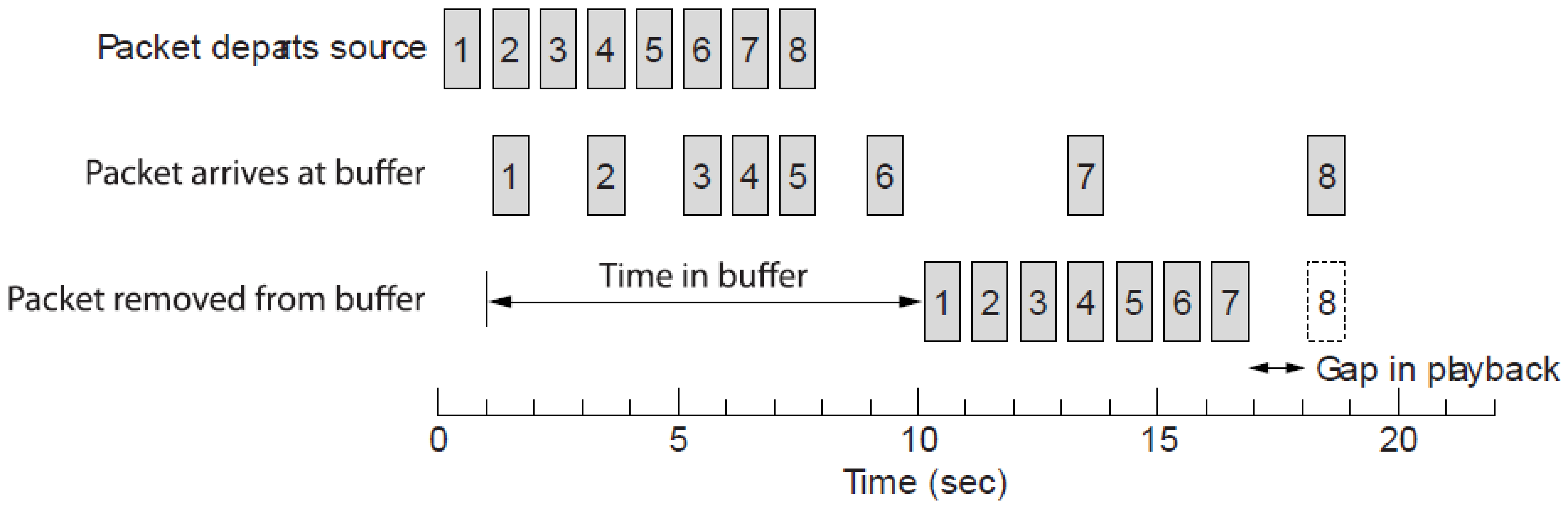
(a) The position of RTP in the protocol stack. (b) Packet nesting.

Real-Time Transport (2)


32 bits
Ver.
P
X
CC
M
Payload type
Sequence number
Timestamp
Synchronization source identifier
Contributing source identifier

The RTP header

Real-Time Transport (3)

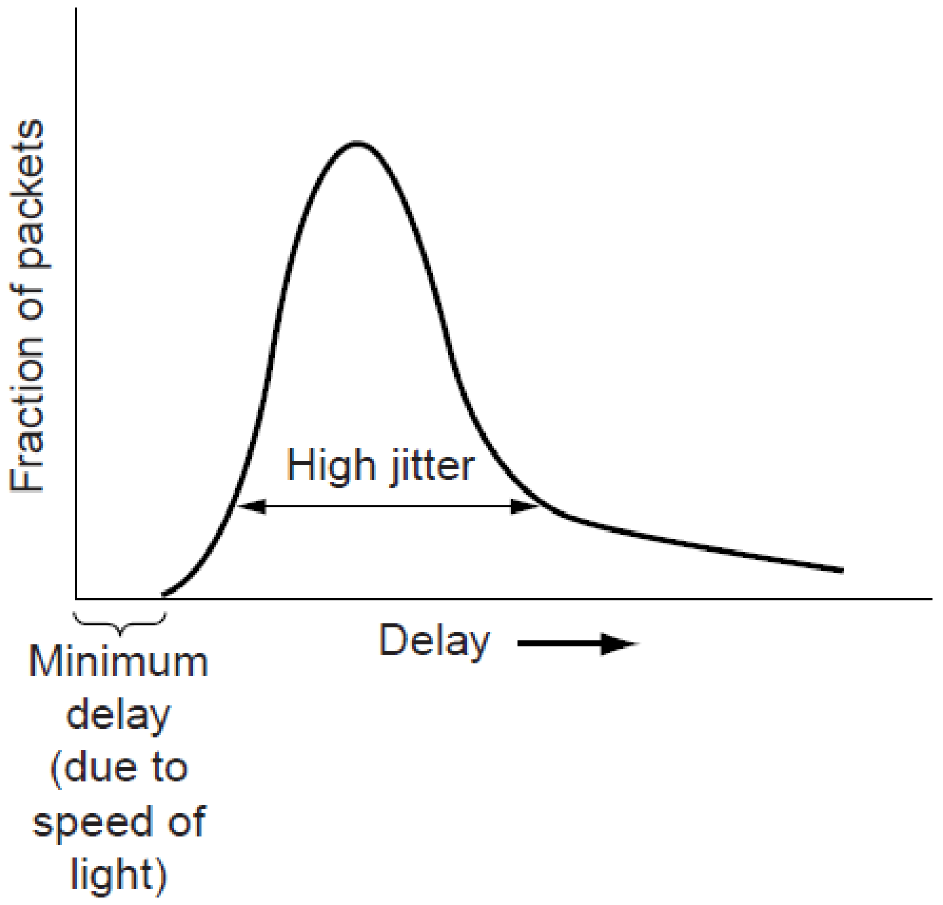


Smoothing the output stream by buffering packets

Playout with buffering and jitter control

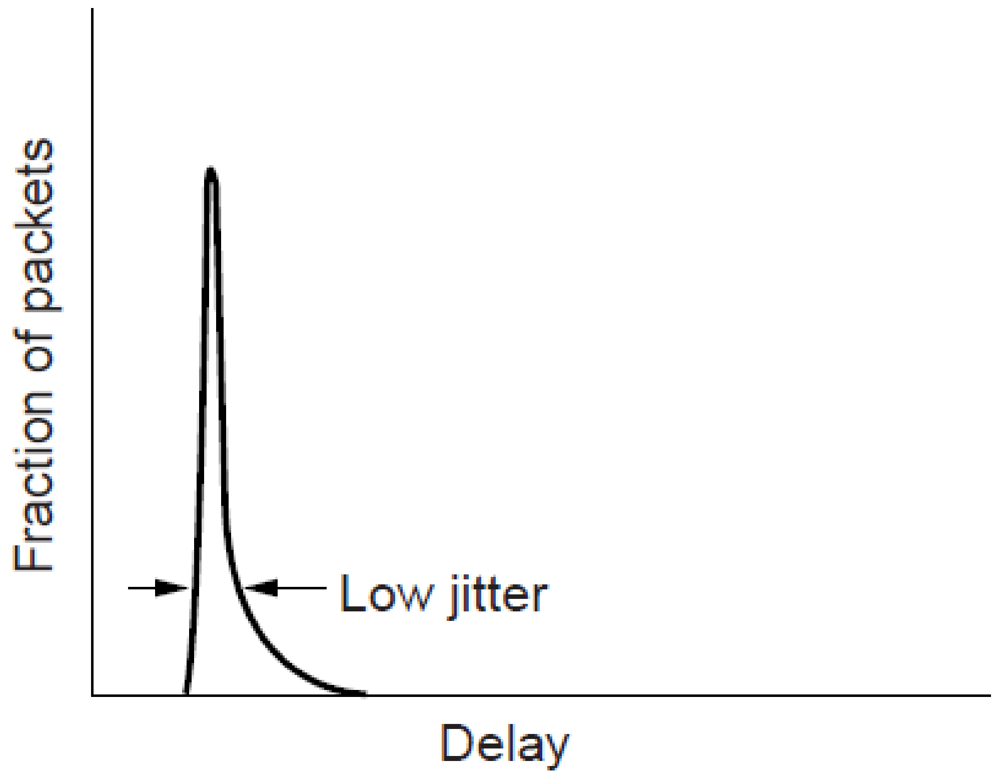
- The variation in delay is called **jitter**.
- The solution to jitter problem is to buffer packets at the receiver before they are played out.
- The RTP timestamps tell when the media should be played.
- The **playback point** is a key for smooth playout.

Real-Time Transport (3)



High jitter

Real-Time Transport (4)



Low jitter

The Internet Transport Protocols: TCP (1)

- Introduction to TCP
- The TCP service model
- The TCP protocol
- The TCP segment header
- TCP connection establishment
- TCP connection release

The Internet Transport Protocols: TCP (2)

- TCP connection management modeling
- TCP sliding window
- TCP timer management
- TCP congestion control
- TCP futures

Introduction to TCP

- To provide a reliable end-to-end byte stream over an unreliable network.
- To dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.
- Defined in RFC 793 in Sept 1981.
- TCP transport entity:
  - A library procedure, a user process, or part of the kernel.

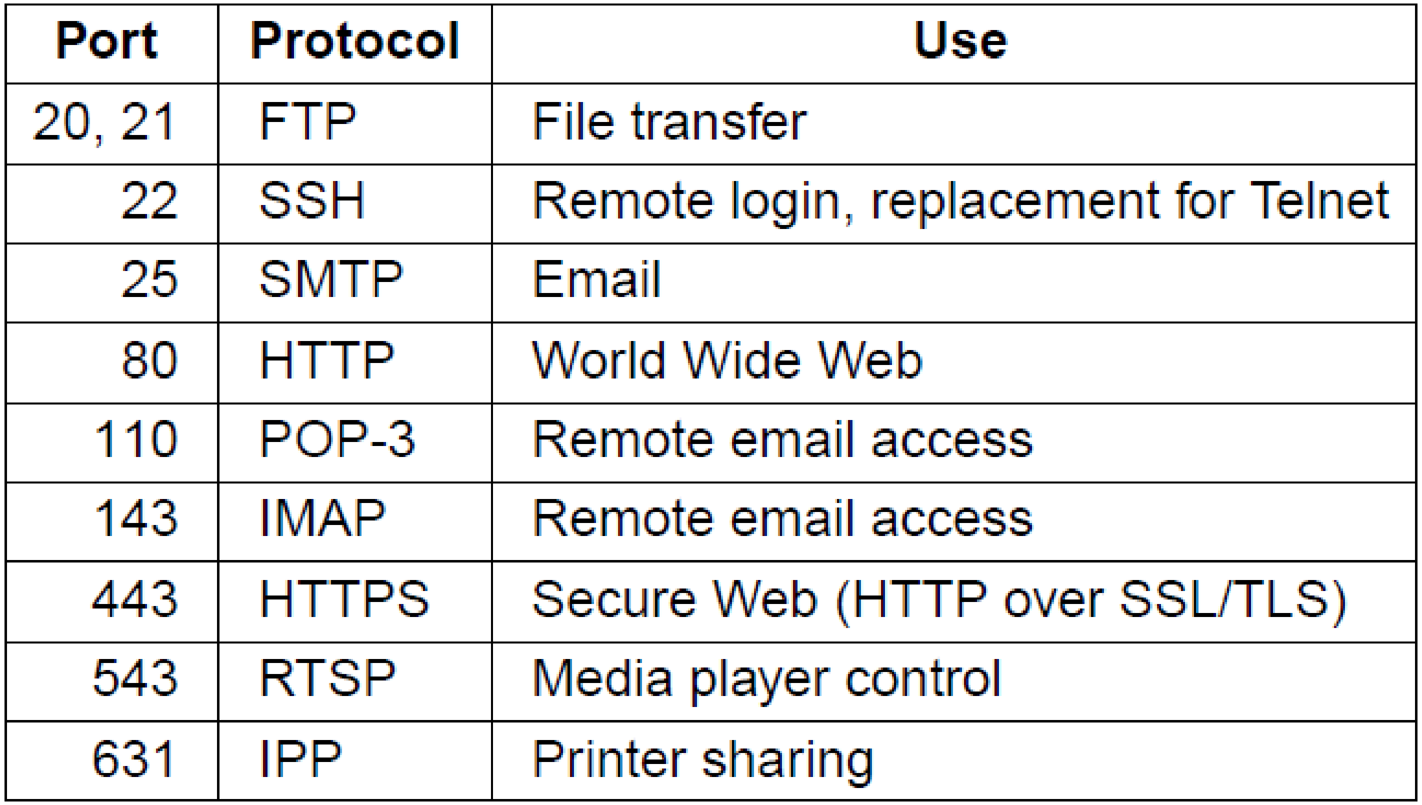
The TCP service model

- **Socket**
  - A socket number consists of the IP address of the host and a 16-bit **port** number.
- A socket may be used for multiple connections at the same time.
  - Connections are identified by the socket identifier (*socket1*, *socket2*).
- Port numbers below 1024 are called **well-known ports**.
  - Usually, a single daemon, called **inetd** (**Internet daemon**), in UNIX attaches itself to multiple ports and waits for the first incoming connection.
- All TCP connections are *full-duplex* and **point-to-point**.
  - No support for multicasting and broadcasting
- A TCP connection is a byte stream, not a message stream.

TCP service model

- When an application passes data to TCP, TCP may send it immediately or buffer it at its discretion.
    - To force data out, TCP has the notion of a PUSH flag that is carried on packets.
- One feature of TCP service remains in the protocolbut is rarely used: **Urgent data**
    - When an interactive user hits the DEL or CTRL-C key to break off a remote computation that has already begun, the sending application puts some control information in the data stream and gives it to TCP along with the **URGENT** flag.
    - – This event causes TCP to stop accumulating data and transmit everything it has for that connection immediately.

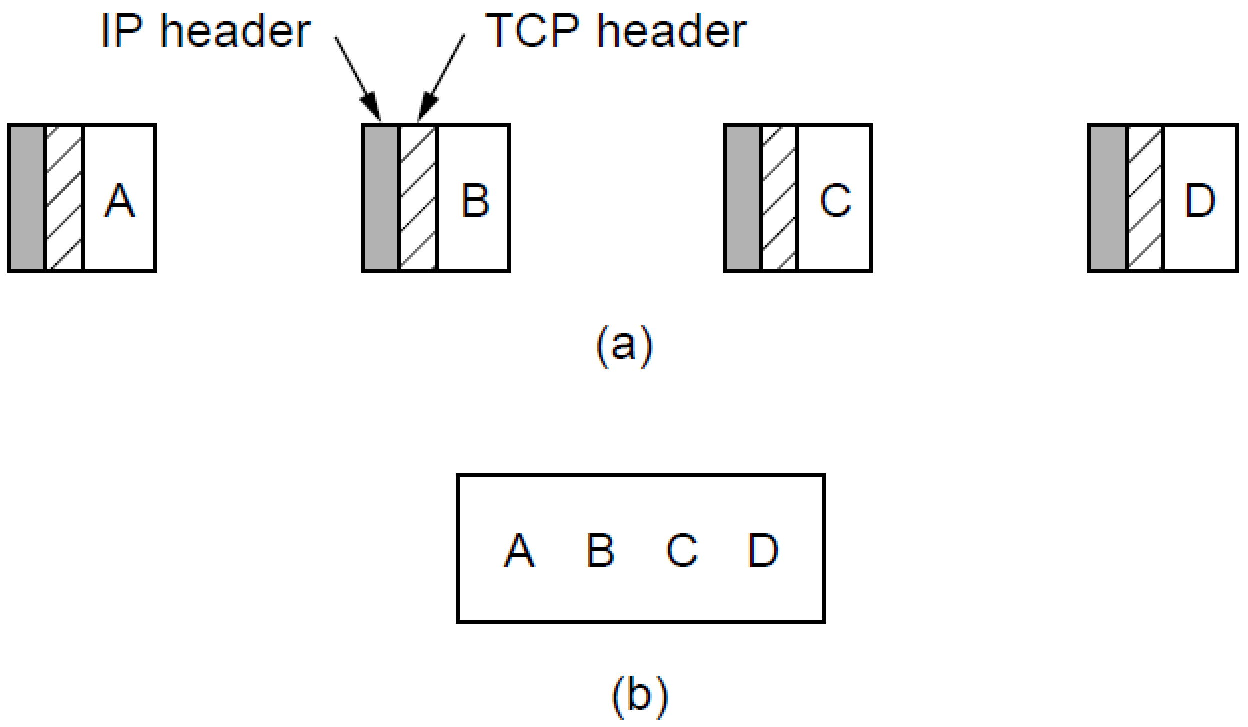
The TCP Service Model (1)



Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Some assigned ports

The TCP Service Model (2)



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

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

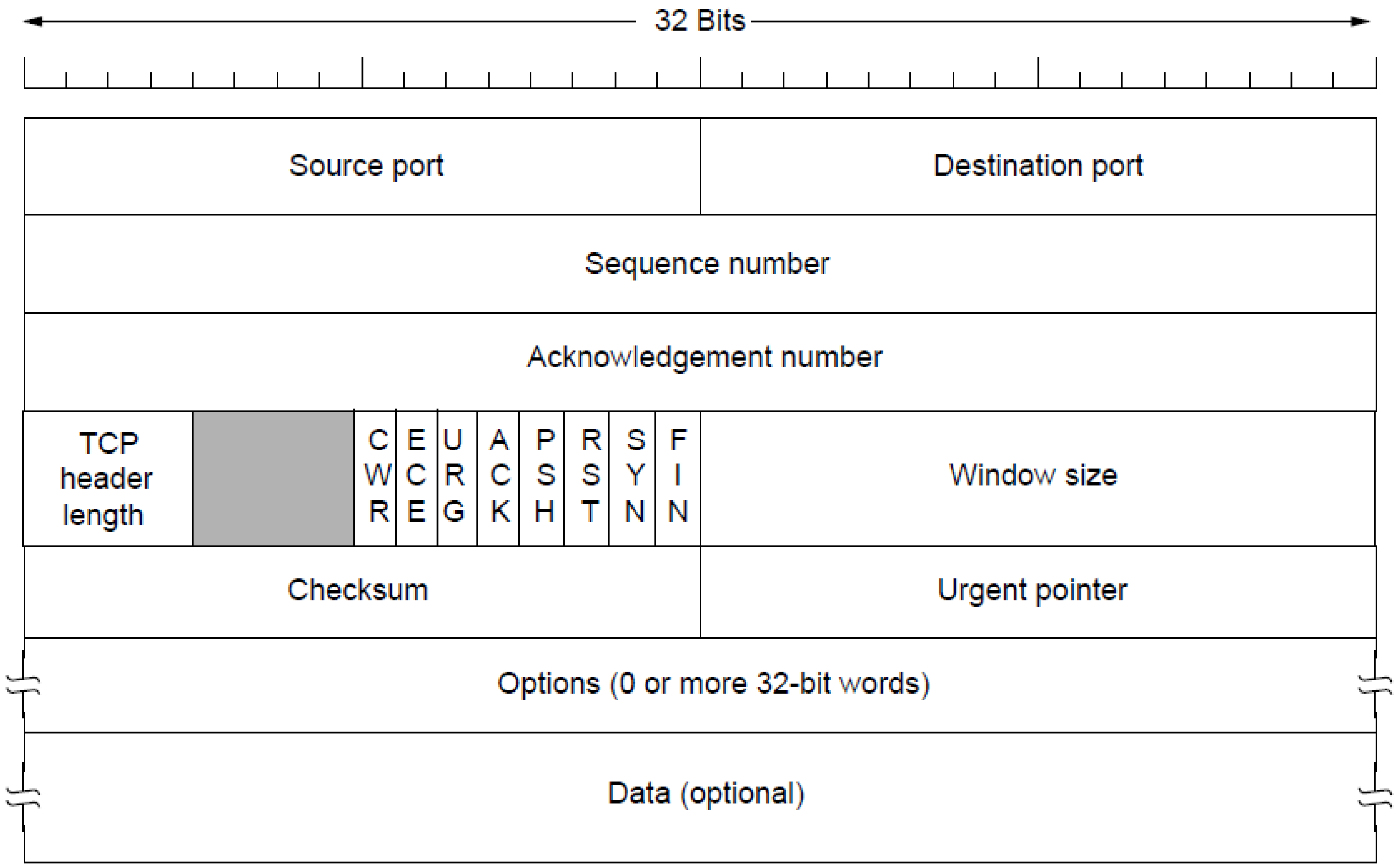
The TCP protocol

- Each byte on a TCP connection has its own 32-bit sequence number.
  - Separate 32-bit sequence numbers are carried on packets for the sliding window position in one direction and for acknowledgements in the reverse direction.
- 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.
  - The TCP software decides how big segments should be. However, *two limits* restrict the segment size.
    - Each segment, including the TCP header, must fit in the 65515 IP payload.
    - Each network has a **maximum transfer unit** or MTU and each segment must fit in the MTU.

The TCP protocol

- Modern TCP implementations perform path MTU discovery, by using ICMP error messages.
- The basic protocol used by TCP entities is the sliding window protocol with a dynamic window size.
    - When a sender transmits a segment, it also starts a timer
    - When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exists, otherwise without data) bearing an acknowledgement number equal to the next sequence number it expects to receive and the remaining window size.
    - If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again.

The TCP Segment Header



The TCP header.

TCP segment header

- Every TCP segment begins with a fixed-format 20-byte header.
  - It may be followed by header option, After the options, if any, up to 65535-20-20=65495 data bytes may follow.
  - Segments without any data are commonly used for acknowledgements and control messages.
- The *Source port* and *Destination port* fields identify the local end points of the connection.
  - A TCP port plus its host's IP address forms a 48-bit unique end point.
  - The source and destination end points (socket numbers) together identify a connection.
  - A **five tuple** connection identifier: the protocol(TCP), source IP and source port, and destination IP and destination port.
- The *Sequence number* and *Acknowledgement number* fields
  - The acknowledgement number specifies *the next byte* expected, not the last byte correctly received. --- a **cumulative acknowledgement**.

TCP segment header

- Two The *TCP header length* (of 4 bits) tells how many 32-bit words are contained in the TCP header.
  - It is needed because the *Options* field is of variable length.
  - It indicates the start of the data within the segment.
- A 3-bit field, which is not used. (6→3)
- Nine 1-bit flags.
  - NS: ECN-nonce - concealment protection (?)
  - CWR: *Congestion window reduced*, from the TCP sender to the TCP receiver.
  - ECE: *ECN-Echo* to a TCP sender
  - *URG* is set to 1 if *Urgent pointer* is in use. The *Urgent pointer* is used to indicate a byte offset from the current sequence number at which urgent data are to be found.
  - The *ACK* bit is set to 1 to indicate that the *Acknowledgement number* is valid. If the *ACK* is 0, the segment does not contain an acknowledgement.

TCP segment header

- The *PSH* bit indicates PUSHed data.
- The *RST* bit is used to reset a connection confused due to a host crash or some other reason. It is also used to reject an invalid segment or refuse an attempt to open a connection.
- The *SYN* bit is used to establish connections. It is used to denote *CONNECTION REQUEST* and *CONNECTION ACCEPTED*, with the *ACK* bit used to distinguish those two possibilities.
- The *FIN* bit is used to release a connection.

• The *Window* field tells how many bytes may be sent starting at the byte acknowledged.

- A *Window* field of 0 says that the bytes up to and including *Acknowledgement number –1* have been received, but the receiver would like no more data for the moment.
- Permission to send can be granted later by sending a segment with the same *Acknowledgement number* and a nonzero *Window* field.

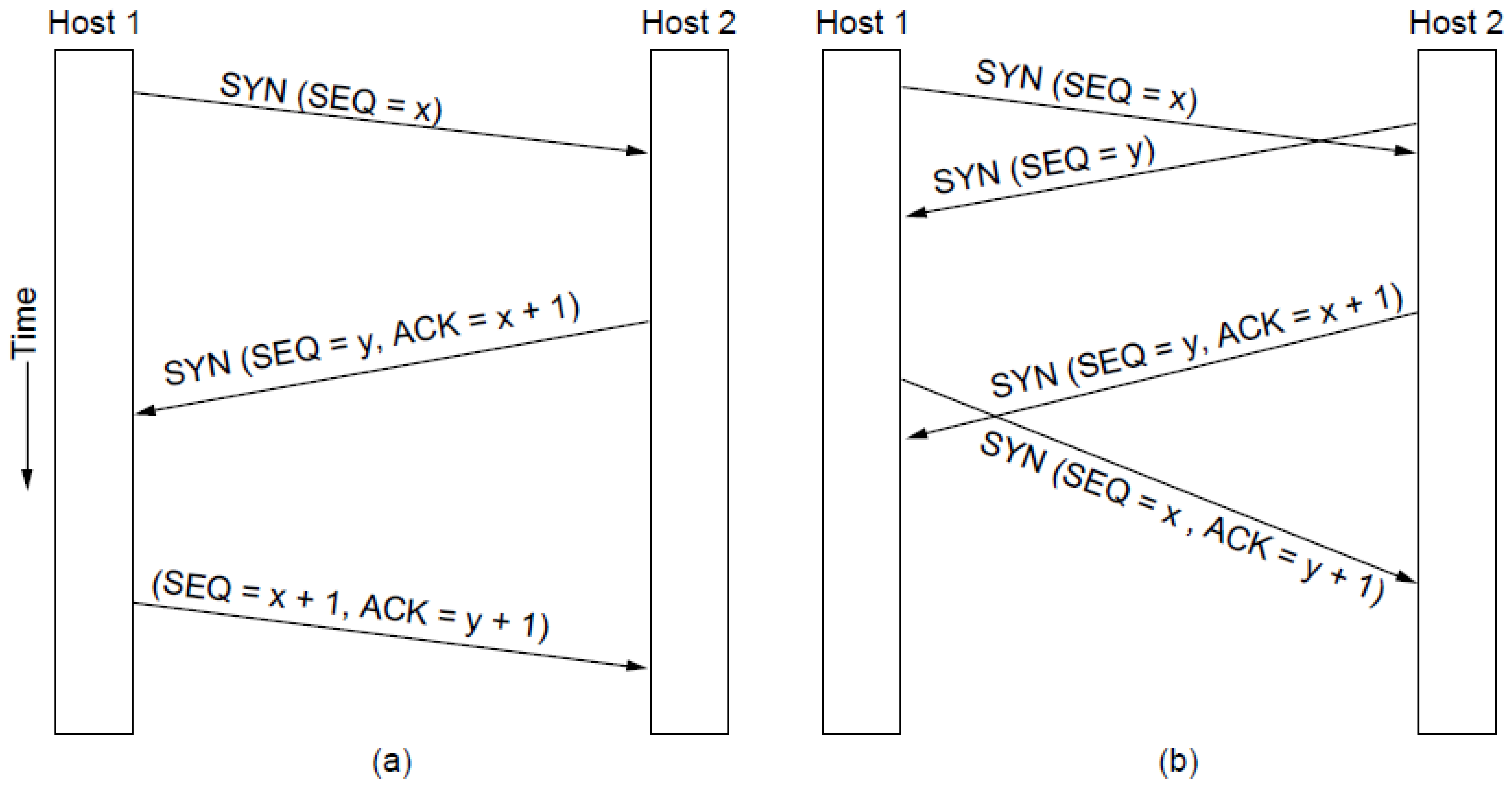
TCP segments

- The *Checksum* checksums the header, the data, and the conceptual pseudoheader shown in next figure.
  - The pseudoheader contains the 32-bit IP address of the source and destination machines, the protocol number for TCP (6), and the byte count in the TCP segment (including the header).
- The *Options* field was designed to provide a way to add extra facilities not covered by the regular header.
  - The most important option is the one that allows each host to specify the MSS(Maximum Segment Size) it is willing to accept.
  - If a host does not use the option, it defaults to a 536-byte payload. All Internet hosts are required to accept TCP segment of 536+20=556 bytes.

TCP segment header

- For lines with high bandwidth, high delay, or both, the 64-KB window is often a problem, limited by the 16-bit *Window size*.
  - A *Window scale* option was proposed, allowing the sender and receiver to negotiate a window scale factor. This number allows both sides to shift the *Window size* field up to 14 bits to the left, thus allowing windows of up to 2^{30} bytes.
- The timestamp option, used
  - To compute round-trip time
  - As a logical extension of the 32-bit sequence number
  - PAWS(Protection Against Wrapped Sequence numbers) scheme
- SACK (Selective ACKnowledgement) option, proposed to use the **selective repeat** instead of **go back n** protocol.

TCP Connection Establishment


Host 1
Host 2
Time
SYN (SEQ = x)
SYN (SEQ = y, ACK = x + 1)
(SEQ = x + 1, ACK = y + 1)
(a)
Host 1
Host 2
SYN (SEQ = x)
SYN (SEQ = y)
SYN (SEQ = y, ACK = x + 1)
SYN (SEQ = x , ACK = y + 1)
(b)

(a) TCP connection establishment in the normal case.

(b) Simultaneous connection establishment on both sides.

TCP Connection establishment

- Connections are established in TCP using the three-way handshake. To establish a connection:
  - One side (the server) passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives.
  - The other side (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.

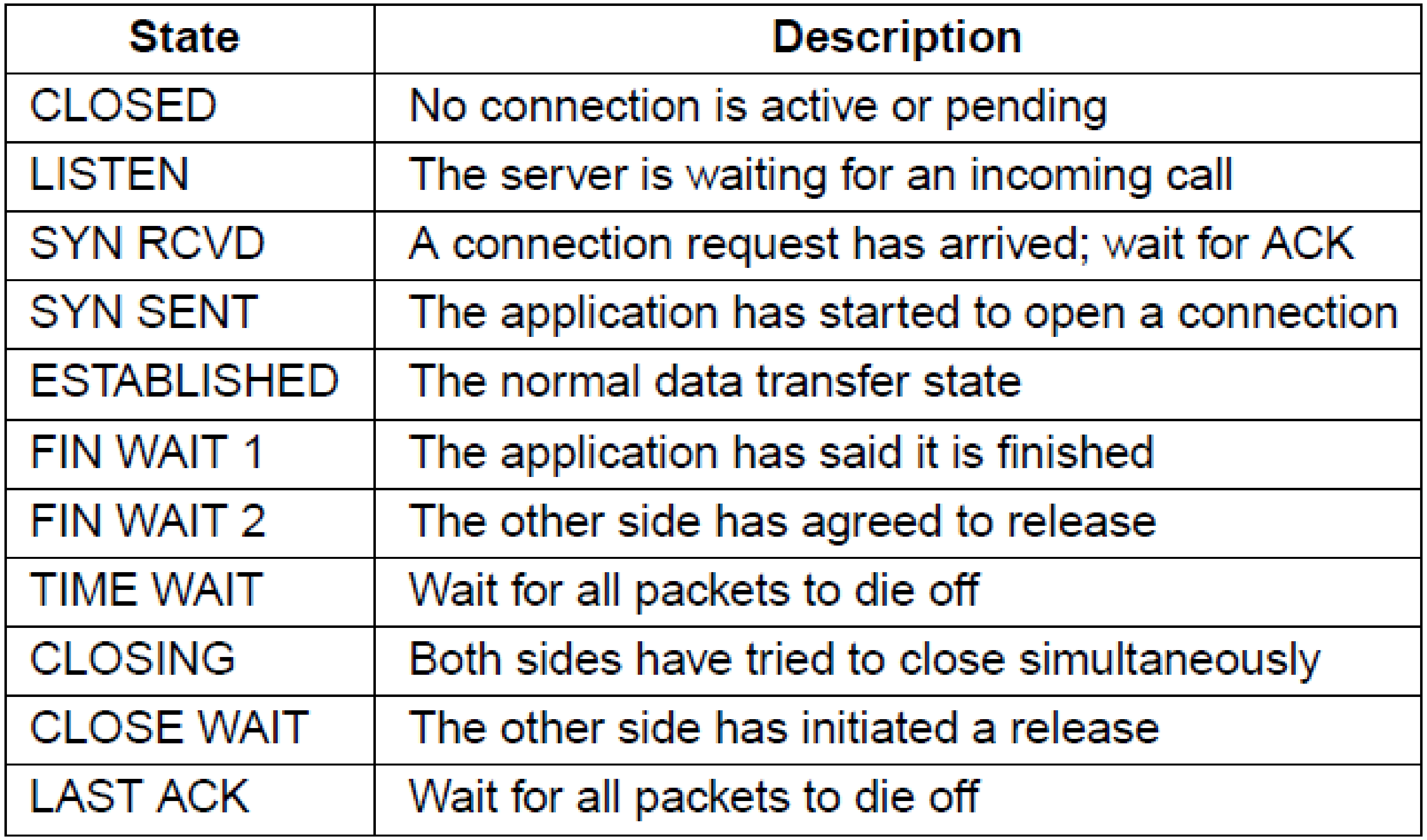
Connection Establishment

- When this segment arrives at the destination, the TCP entity there checks 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 giving the TCP segment. It can then either accept or reject the connection.
- In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the result is that just one connection is established because connections are identified by their end points.
- A clock-based scheme is used for the initial sequence number, with a clock tick every 4 μsec.
  - When a host is subject to **SYN flood ?**
    - » Use **SYN cookies**

TCP connection release

- TCP connections are full duplex. To release TCP connections, it is considered 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.
  - When the *FIN* is acknowledged, that direction is shut down. Data may continue to flow indefinitely in the other direction.
  - When both directions have been shut down, the connection is released. Normally, four TCP segments are needed to release a connection. However,…
  - To avoid the two-army problem, timers is used. If a response to a *FIN* is not forthcoming within *two maximum packet lifetimes*, the sender of the *FIN* releases the connection.

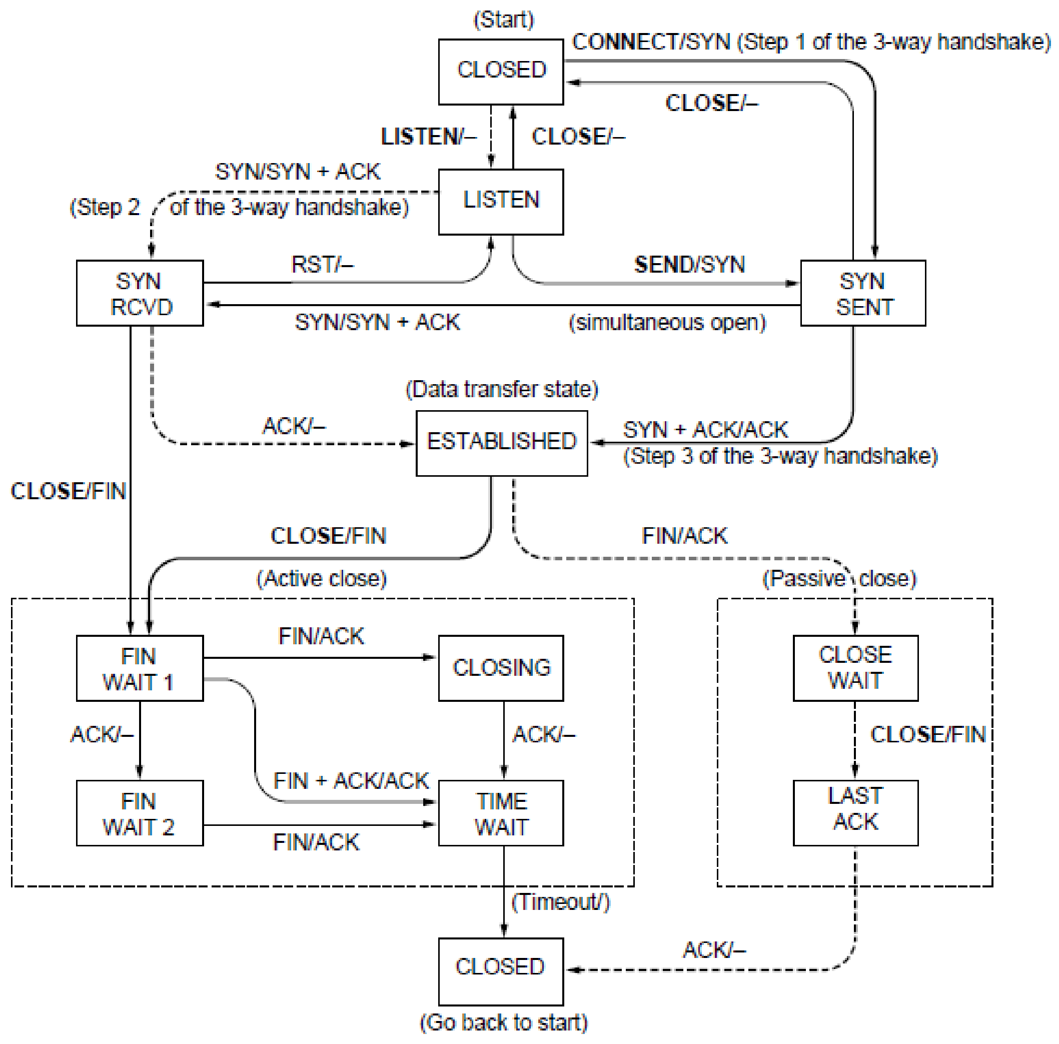
TCP Connection Management Modeling (1)



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
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

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

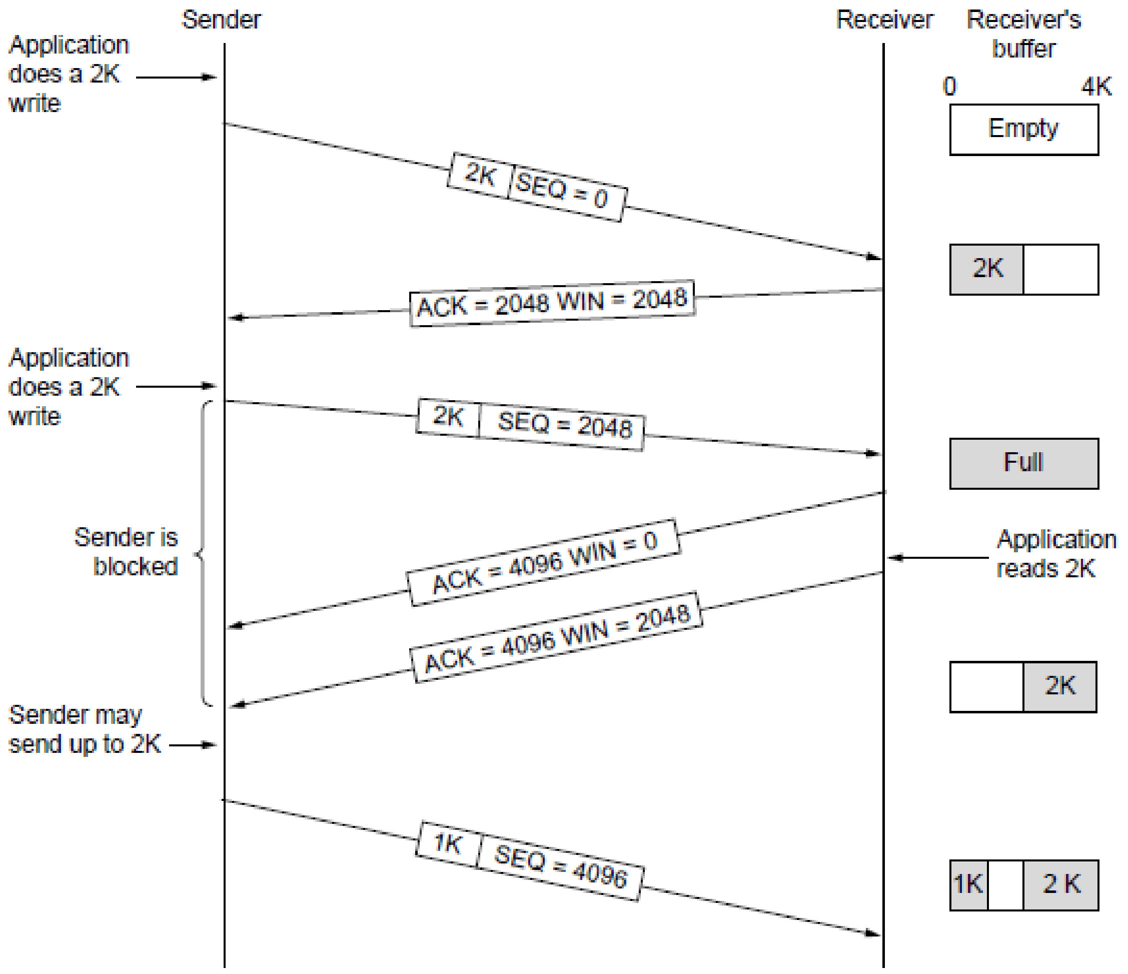
TCP Connection Management Modeling (2)



TCP connection management finite state machine.

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

TCP Sliding Window (1)



Window management in TCP

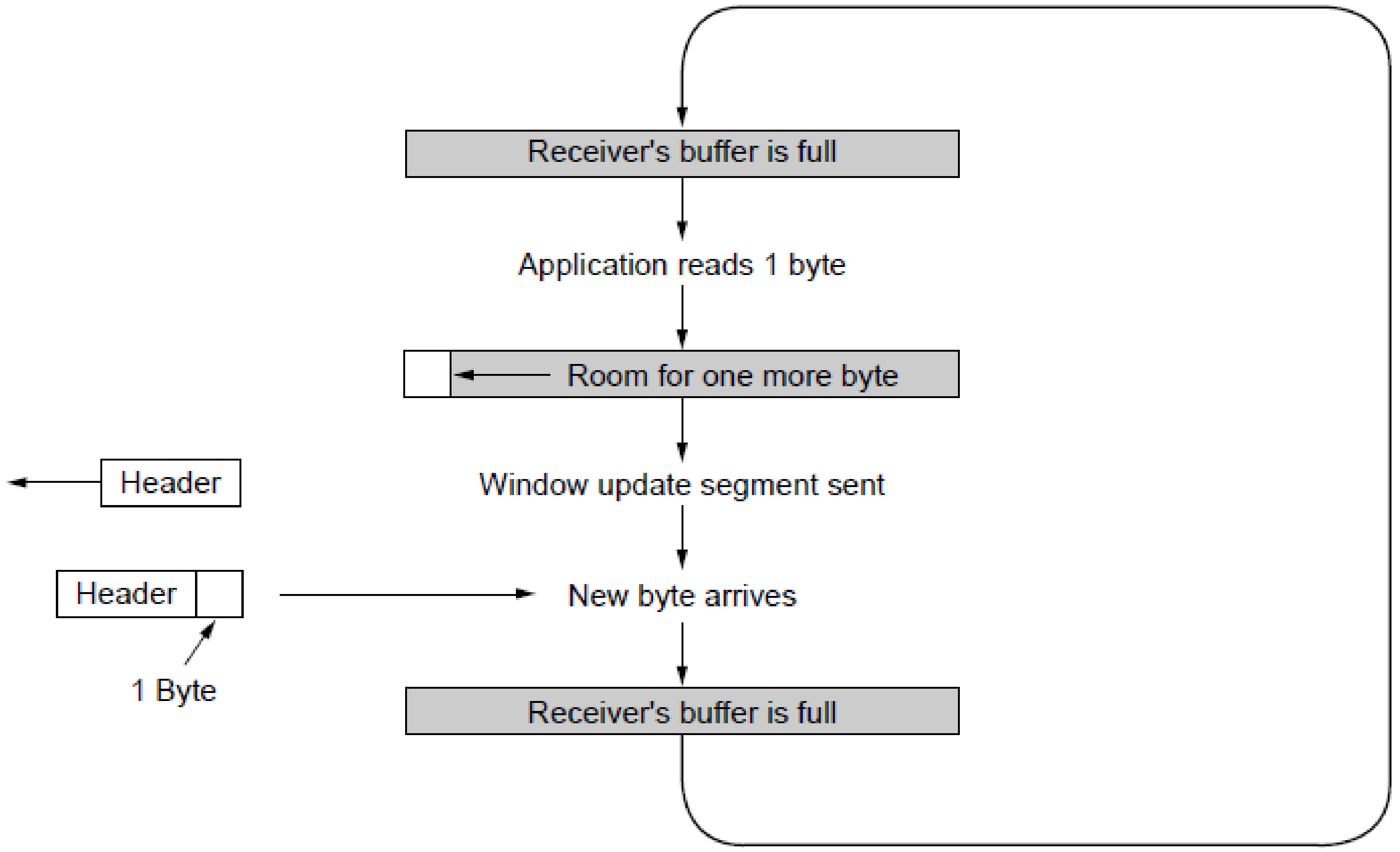
TCP sliding window

- Window management in TCP decouples the issues of acknowledgement of the correct receipt of segments and receiver buffer allocation.
  - When the window is 0, the sender may not normally send segments, with two exceptions.
    - First, urgent data may be sent, for example, to allow the user to kill the process running on the remote machine.
    - Second, the sender may send a 1-byte segment to make the receiver reannounce the next byte expected and window size.
      - **Window probe**: The TCP standard explicitly provides this option to prevent deadlock if a window announcement ever gets lost.
  - Senders are not required to transmit data as soon as they come in from the application. Neither are receivers required to send acknowledgements as soon as possible.

TCP sliding window

- **Delayed acknowledgement**: One approach to reduce the load (of acknowledgement and window advertisement) placed on the network by the receiver is to delay acknowledgements and window updates for 500 msec in the hope of acquiring some data on which to hitch a free ride.
- A way to improve the efficiency at the sender is the **Nagle's algorithm**:
  - When data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged.
  - Then send all the buffered data in one TCP segment and start buffering again until the next segment is acknowledged.
  - It allows a new segment to be sent if enough data have trickled in to fill *a maximum segment* or *half the window* .

TCP Sliding Window (2)



Silly window syndrome

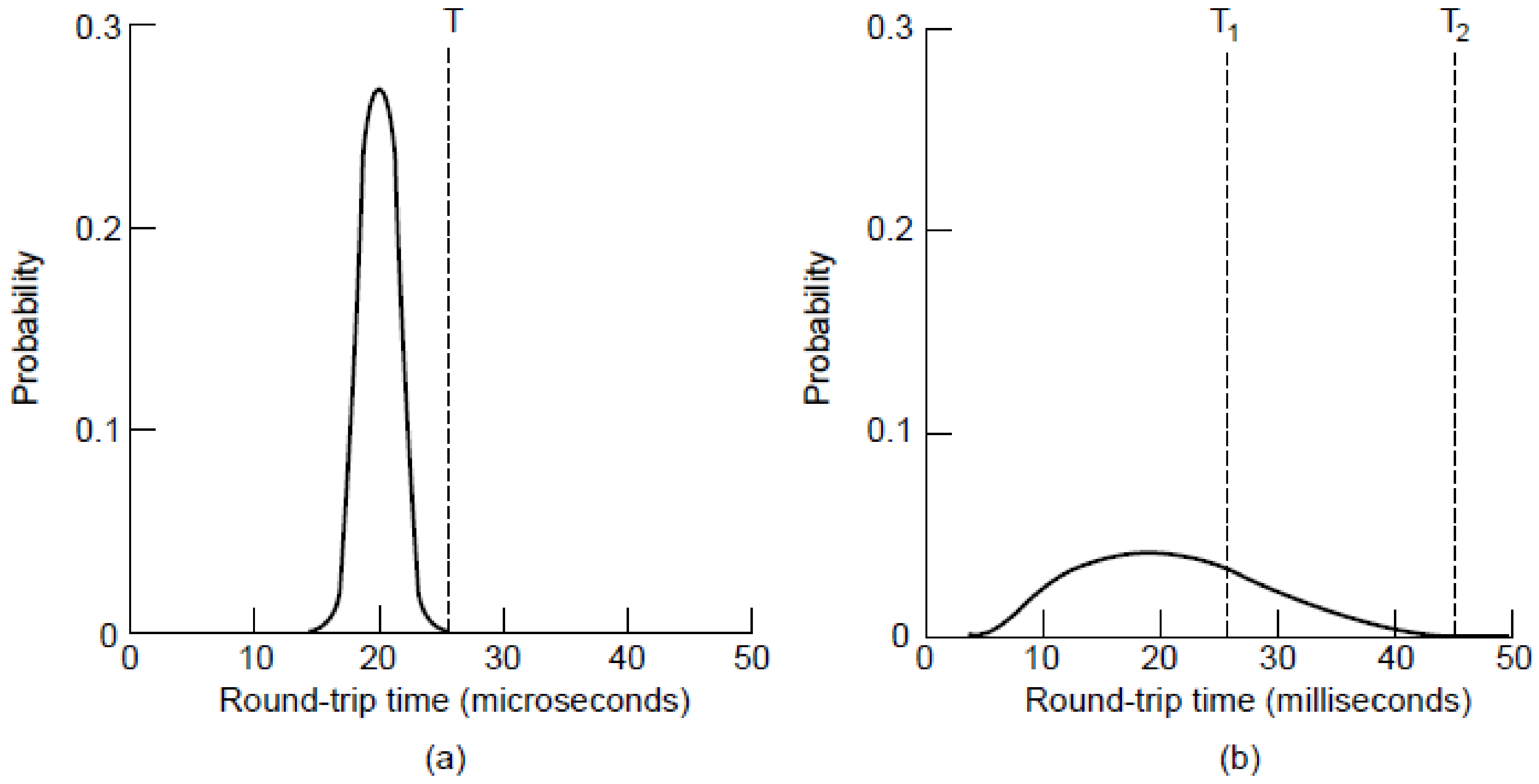
TCP sliding window

- Another problem with TCP is the **silly window syndrome** (Clark, 1982):
  - The problem occurs when data are passed to the sending TCP entity in large blocks, but an interactive application on the receiving side read data 1 byte at a time.
    - The previous figure illustrates the problem.
  - Clark's solution is to prevent <u>the receiver</u> from sending a window update for 1 byte.
    - The receiver should not send a window update until it can handle *the maximum segment size* it advertised when the connection was established, or until *its buffer is half empty*, whichever is smaller.
  - Nagle's algorithm and Clark's solution to the silly window syndrome are complementary. The goal is for <u>the sender</u> not to send small segments and <u>the receiver</u> not to ask for them.

TCP Timer Management

- TCP uses multiple timers: *Retransmission timer*, *Persistence timer*, *Keepalive timer*, and the one used in the *TIMED WAIT* state.
- **Retransmission timer (RTO)**:
  - 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 is started again.
  - *How long should the timeout interval be?*

TCP Timer Management



(a) Probability density of acknowledgment arrival times in data link layer. (b) … for TCP

TCP Timer Management

- **EWMA**(Exponentially Weighted Moving Average):
- TCP uses a highly dynamic algorithm that constantly adjusts the timeout interval, based on continuous measurements of network performance. (due to Jacobson)
  - For each connection, TCP maintains a variable, *RTT*, that is the best current estimate of the round-trip time to the destination.
  - When a segment is sent, a timer is started, both to see how long (*M*) the acknowledgement takes and to trigger a retransmission if it takes too long.
  - *If the acknowledgement gets back before the timer expires*, the *RTT* is updated according to the formula,

$$RTT = \alpha RTT + (1 - \alpha)M$$

where α (=7/8 typically) is a smoothing factor.

TCP Timer Management

- Initial implementation of TCP uses 2x*RTT* as a retransmission timeout(RTO) interval.
- To account for delay variation, an estimate of round-trip time variation (*standard deviation*) is taken into account. (By Jacobson)
  - The *mean deviation(D)*, as an estimator of the standard deviation,

    $$D = \beta D + (1 - \beta)\,|RTT - M|$$

    where *β=3/4 typically.*
  - Most TCP implementations now set the timeout interval to

$$\text{RTO} = RTT + 4{*}\text{D} \quad ; \ (\text{RTO} \geq 1 \text{ sec, initial value=?})$$

- What to do when a segment times out and is sent again:
  - **Karn's algorithm**: *Do not update RTT on any segments that has been retransmitted. Instead, the timeout is doubled on each failure until the segments get through first time.*

TCP Timer Management

- The **persistence timer**:
  - It is designed to prevent the following deadlock:
    - The receiver sends an acknowledgement with a window size of 0, telling the sender to wait.
    - Later, the receiver updates the window, but the packet with update is lost.
  - When the persistence timer goes off, the sender transmits a probe to the receiver.
    - The response to the probe gives the window size.
    - If it is still zero, the persistence timer is set again and the cycle repeats.

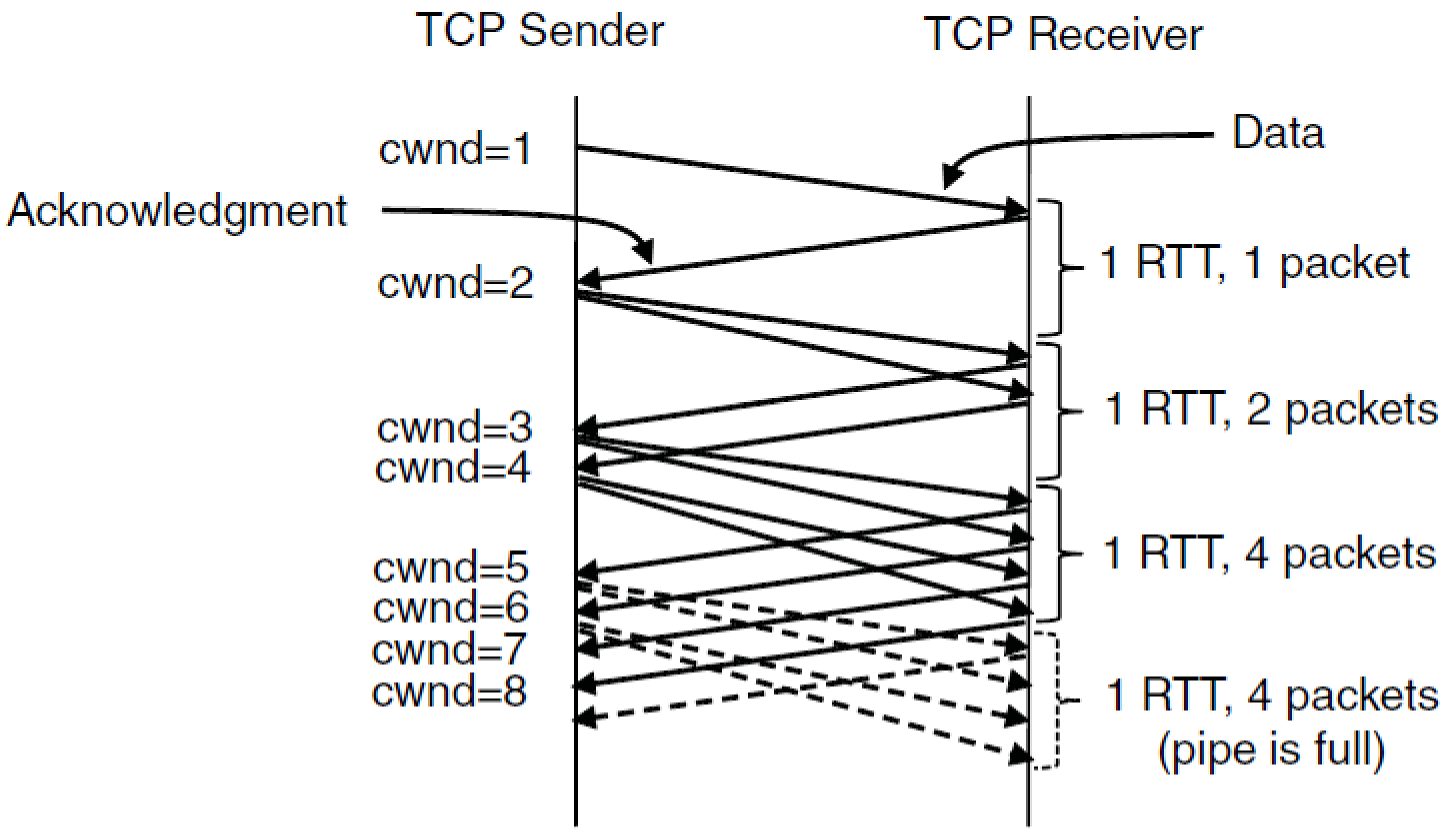
TCP Timer Management

- The **keepalive timer**:
  - When a connection has been idle for a long time, the keepalive timer may go off to cause one side to check if the other is still there.
  - If it fails to respond, the connection is terminated.
- The timer used in the TIMED WAIT state while closing a TCP connection:
  - It runs for *twice* the maximum packet lifetime to make sure that when a connection is closed, all packets created by it have died off.

TCP Congestion Control

- TCP maintains a **congestion window**.
  - The corresponding rate is the window size divided by the round-trip time of the connection.
  - TCP adjusts the window size according to the AIMD control law.
- The Internet deals with separately the two potential problems: network capacity and receiver capacity.
  - Each sender maintains two windows: the flow control window the receiver has granted and a second window, the **congestion window**.
    - The number of bytes that may be sent is the minimum of the two windows.
      - Thus, the effective window is the minimum of what the sender thinks is all right and what the receiver thinks is all right.
- Ack clock, used by TCP to smooth traffic and avoid unnecessary queues at routers.
- The speed to reach desired operating point (i.e., rate).

TCP Congestion Control (1)



Slow start from an initial congestion window of 1 segment

TCP Congestion Control

- **Slow Start:**
  - When a connection is established, the sender initializes the congestion window to a small initial value of at most four segments. (A change from the size of flow control window used previously.)
  - It then sends the initial window.
  - If this segment is acknowledged before the timer goes off, it adds one segment's worth of bytes to the congestion window.
  - When the congestion window is *n* segments, if all n are acknowledged on time, the congestion window is increased by the byte count corresponding to *n* segments.
    - In effect, each burst successfully acknowledged doubles the congestion window.

TCP Congestion Control

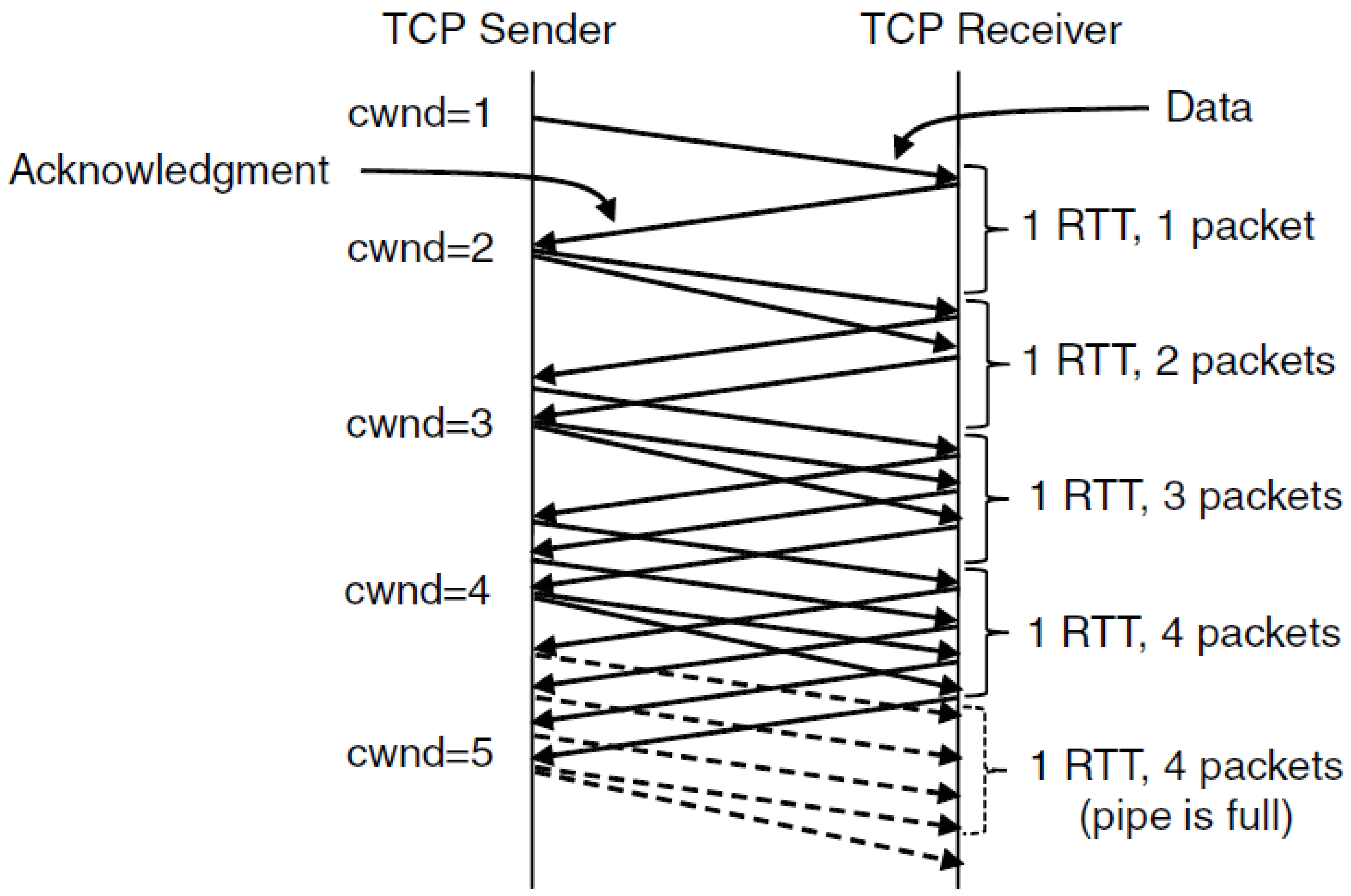
- The congestion window keeps growing exponentially until either a timeout occurs or the receiver's window is reached.

  This algorithm is called **slow start**.
- To keep slow start under control, the sender keeps a threshold (the third parameter) called the **show start threshold**, in addition to <u>the receiver window</u> and <u>the congestion window</u>.
  - *When a timeout occurs, the threshold is set to half of the current congestion window, and the entire process is restarted.*
  - Slow start is then used to determine what the network can handle, except that *exponential growth stops when the threshold is hit. From that point on, successful transmissions grow the congestion window linearly (by one maximum segment for each burst instead of one per segment).* ←Additive increase

TCP congestion control

- Additive increase:
  - *cwnd*: congestion window
  - MSS: maximum segment size
  - Increas cwnd by (MSS×MSS)/cwnd for each of the cwnd/MSS packets that may be acknowledged.
  - $$\frac{\left(\frac{cwnd}{MSS}+1\right)MSS}{\frac{cwnd}{MSS}} = MSS + \frac{MSS \times MSS}{cwnd}$$
    - If no more timeouts occur, the congestion will continue to grow up to the size of the receiver's window.
      - At that point, it stop growing and remain constant as long as there are no more timeouts and the receiver's window does not change size.
- **Duplicate acknowledgement**
  - TCP assumes that three duplicate acknowledgements imply that a packet has been lost.

TCP Congestion Control (2)

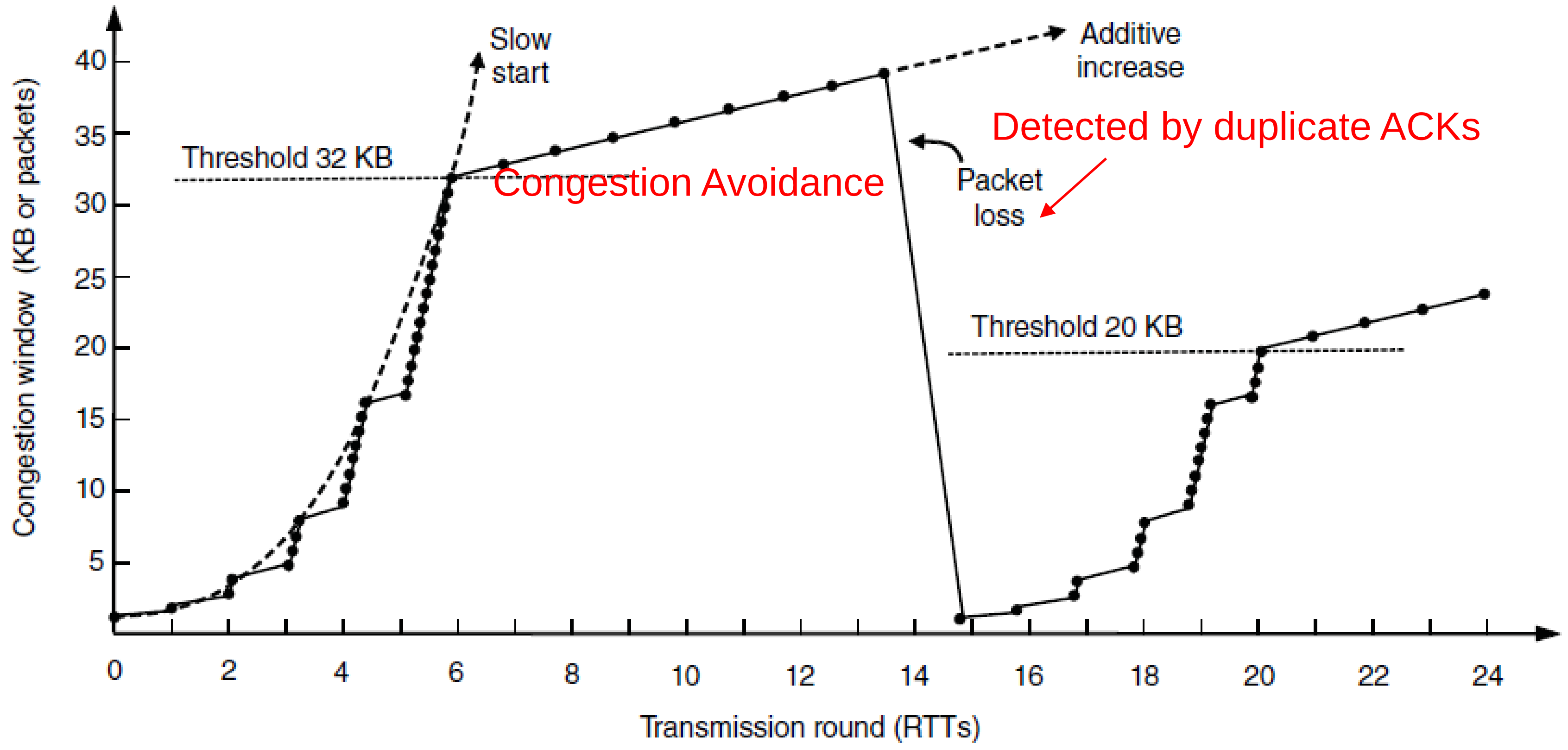


Additive increase from an initial congestion window of 1 segment.

TCP congestion control

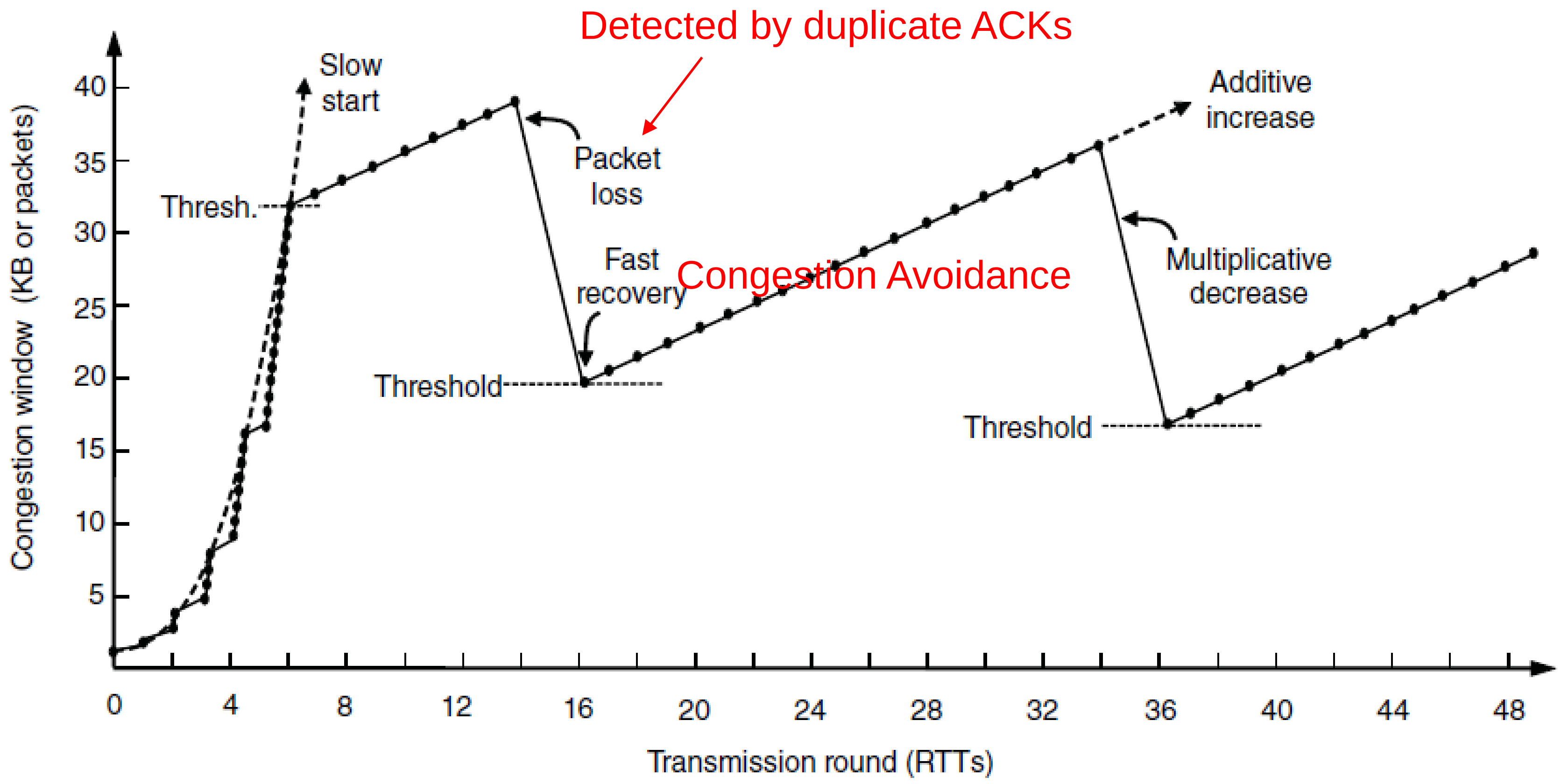
- Fast retransmission
  - ✓ After three duplicate acknowledgements
- Fast recovery
  - ✓ A temporary mode aims to maintain the ack clock running with a congestion window that is the new threshold….

TCP Congestion Control (3)



Slow start followed by additive increase in TCP Tahoe.

TCP Congestion Control (4)



Fast recovery and the sawtooth pattern of TCP Reno.

TCP

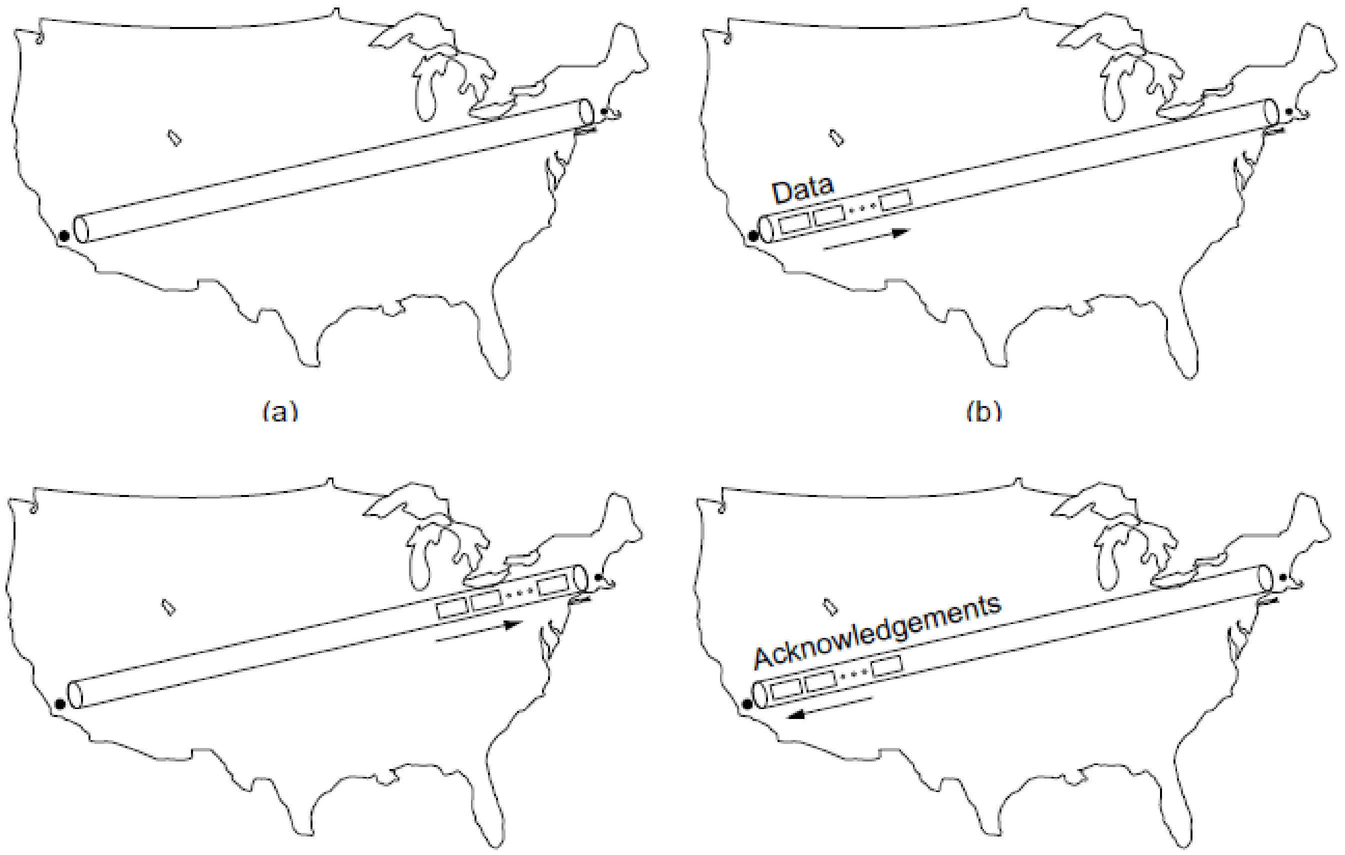
Two changes affect TCP implementations:

- SACK (Selective ACKnowledgements)
  - ✓ List up to three ranges of bytes that have been received.
- ECE (ECN Echo)
  - ✓ The TCP senders reacts in exactly the same ways it does to packet loss via duplicate ACKs.

Performance Issues

- Performance problems in computer networks
- Network performance measurement
- System design for better performance
- Fast TPDU processing
- Protocols for high-speed networks

Performance Problems in Computer Networks



The state of transmitting one megabit from San Diego to Boston.
(a) At *t = 0.* (b) *After 500 μ sec.*
(c) *After 20 msec.* (d) *After 40 msec.*

Network Performance Measurement (1)

Steps to performance improvement

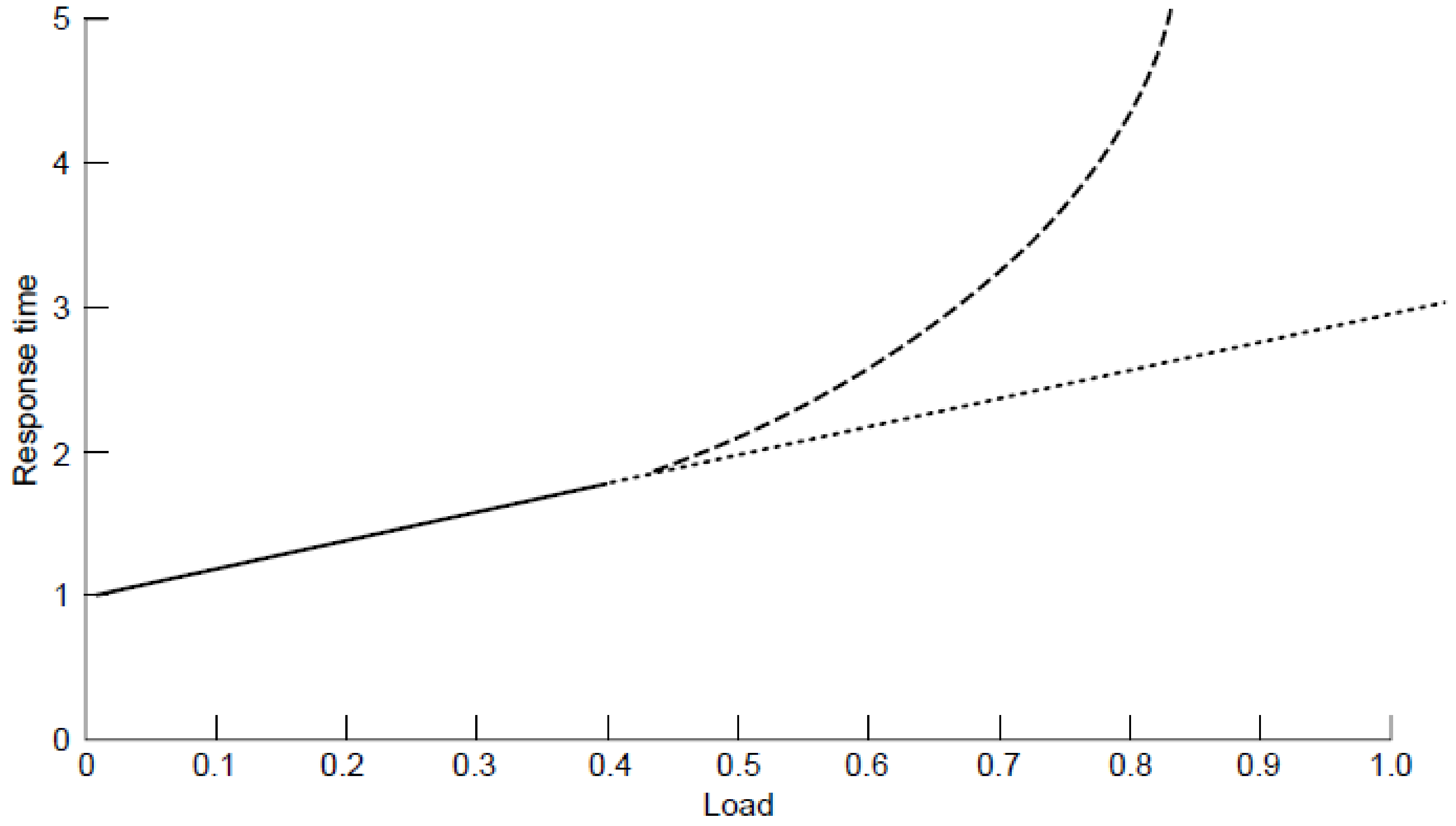
1. Measure relevant network parameters, performance.
2. Try to understand what is going on.
3. Change one parameter.

Network Performance Measurement (2)

Issues in measuring performance

- Sufficient sample size
- Representative samples
- Clock accuracy
- Measuring typical representative load
- Beware of caching
- Understand what you are measuring
- Extrapolate with care

Network Performance Measurement (3)



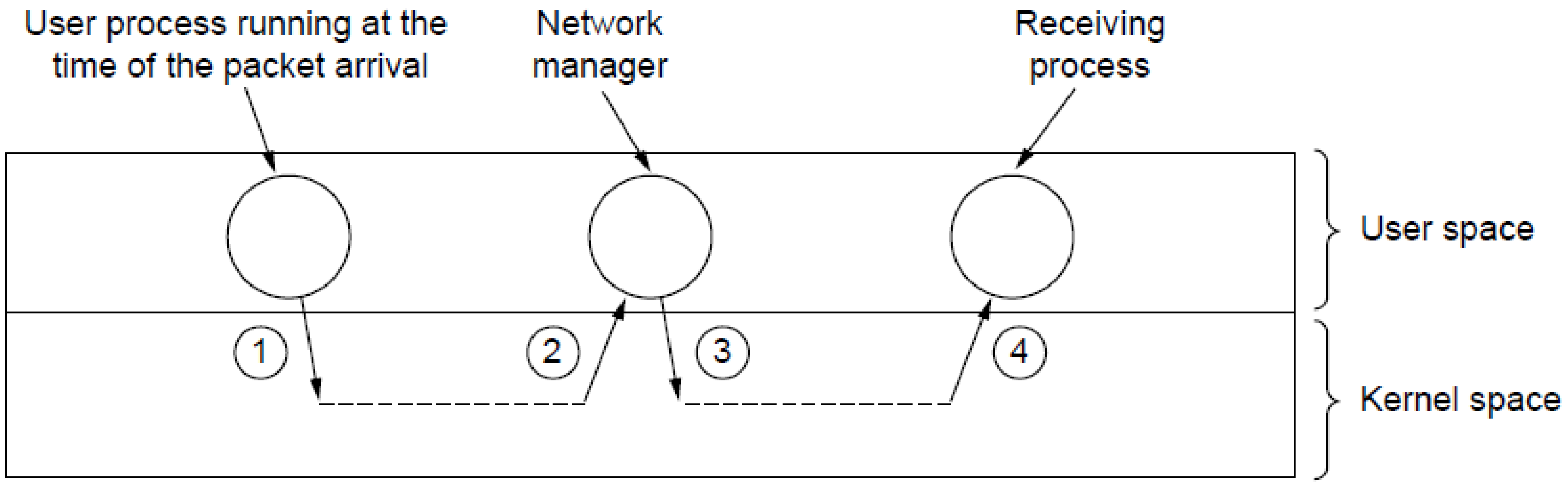
Response as a function of load.

System Design for Better Performance (1)

Rules of thumb

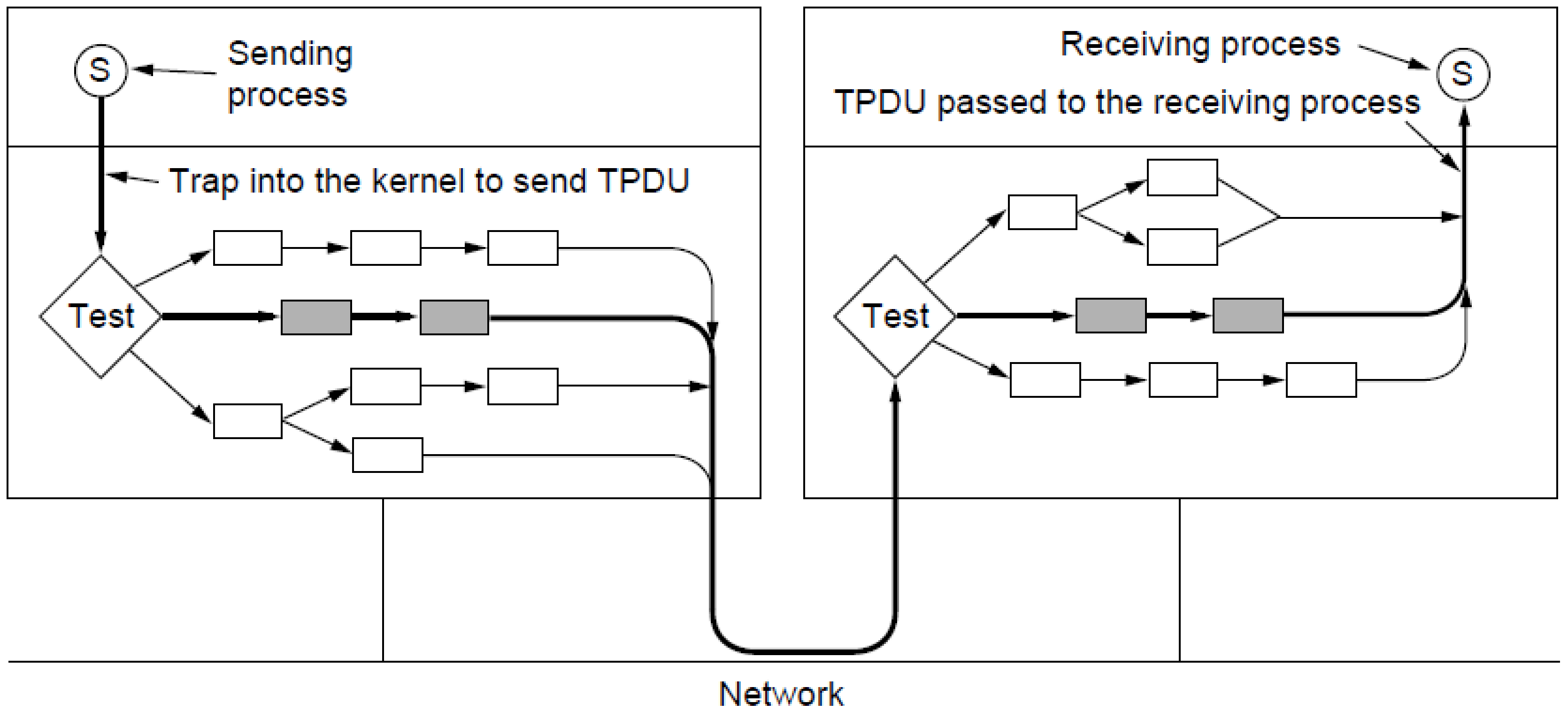
1. CPU speed more important than network speed
2. Reduce packet count to reduce software overhead
3. Minimize data touching
4. Minimize context switches
5. Minimize copying
6. You can buy more bandwidth but not lower delay
7. Avoiding congestion is better than recovering from it
8. Avoid timeouts

System Design for Better Performance (2)



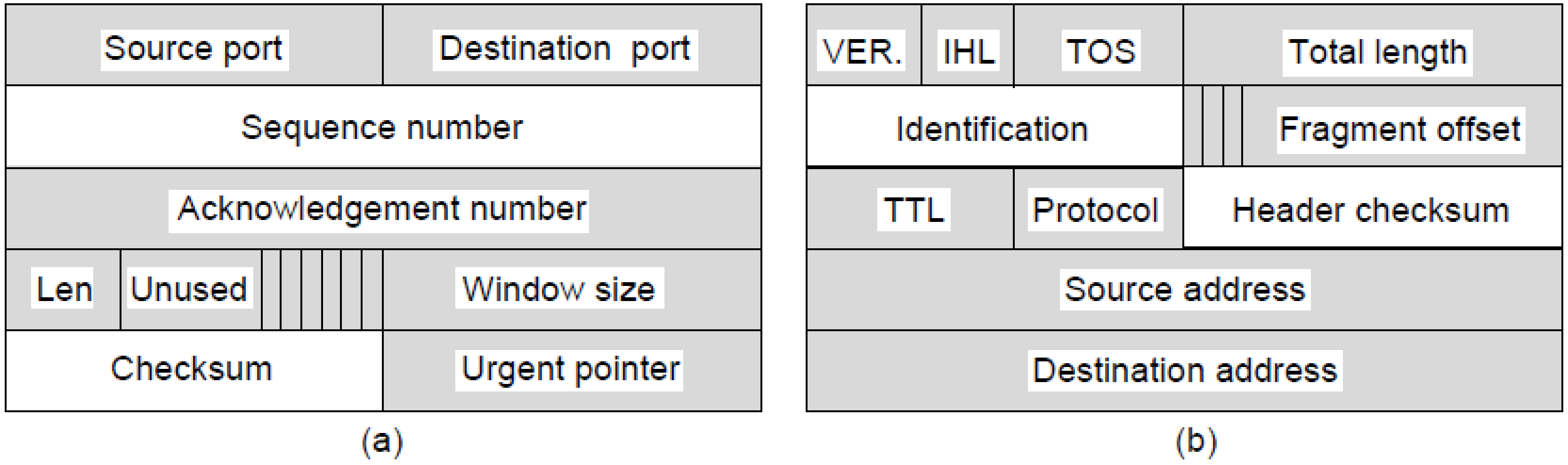
Four context switches to handle one packet with a user-space network manager.

Fast TPDU Processing (1)


S
Sending process
Trap into the kernel to send TPDU
Test
Receiving process
S
TPDU passed to the receiving process
Test
Network

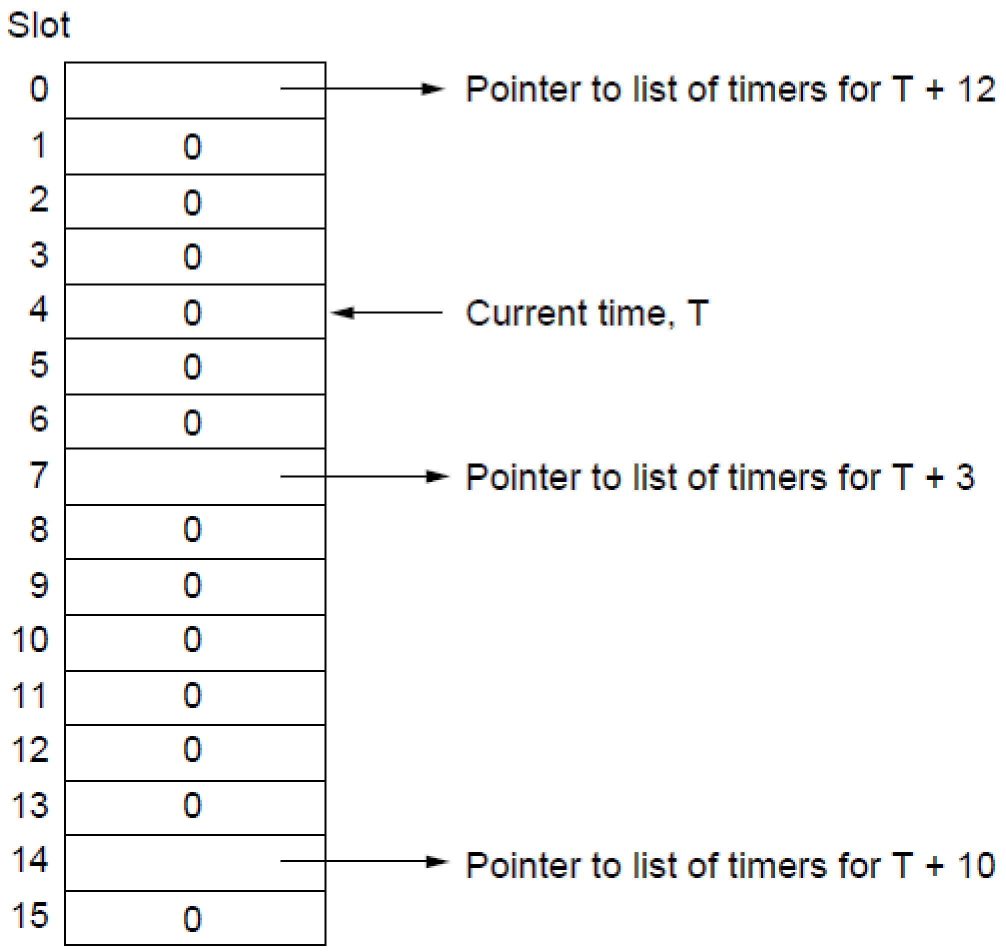
The fast path from sender to receiver is shown with a heavy line. The processing steps on this path are shaded.

Fast TPDU Processing (2)



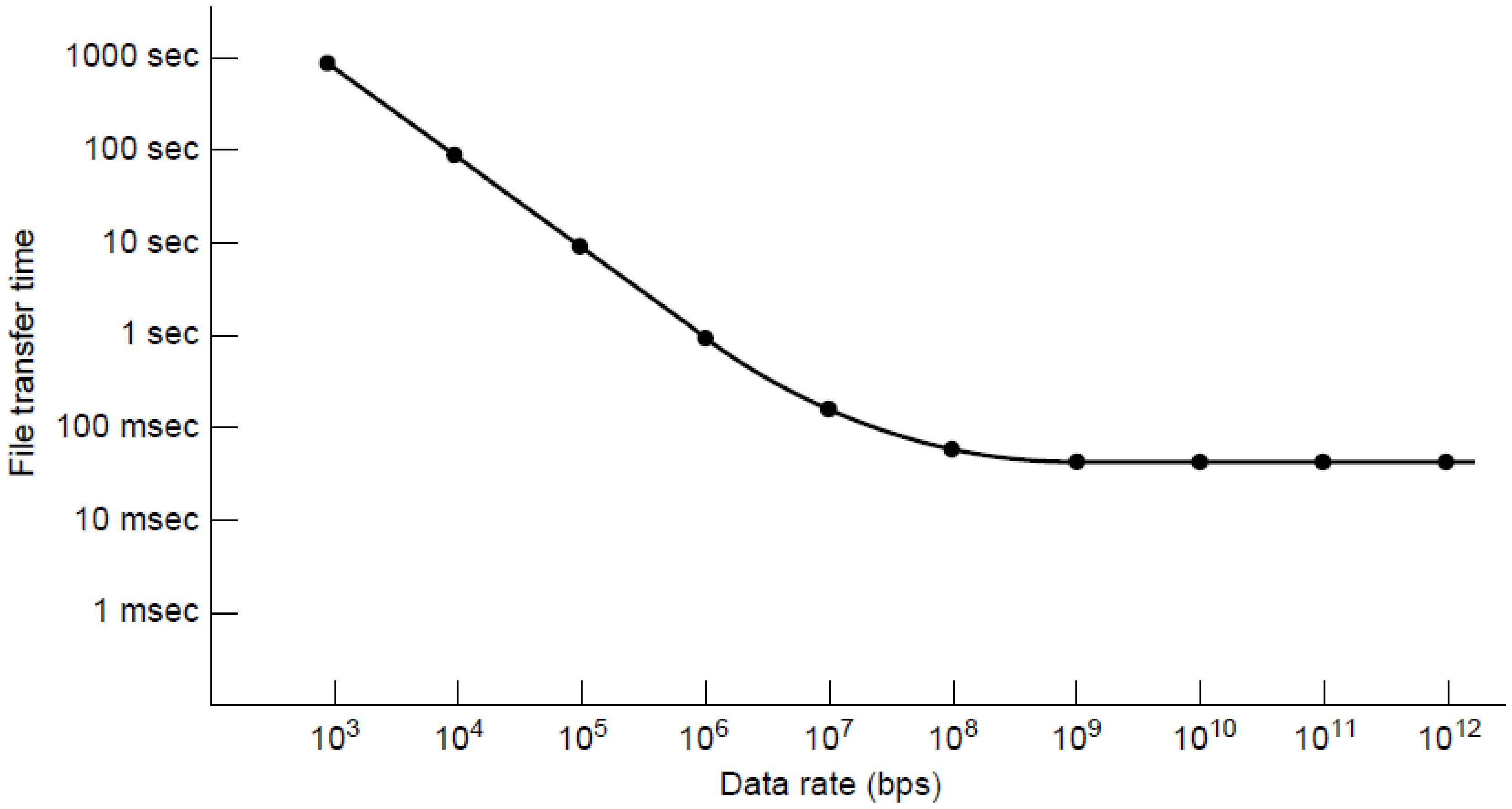
(a) TCP header. (b) IP header. In both cases, the shaded fields are taken from the prototype without change.

Protocols for High-Speed Networks (1)


Slot
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
0
0
0
0
0
0
0
0
0
0
0
0
Pointer to list of timers for T + 12
Current time, T
Pointer to list of timers for T + 3
Pointer to list of timers for T + 10

A timing wheel

Protocols for High-Speed Networks (2)

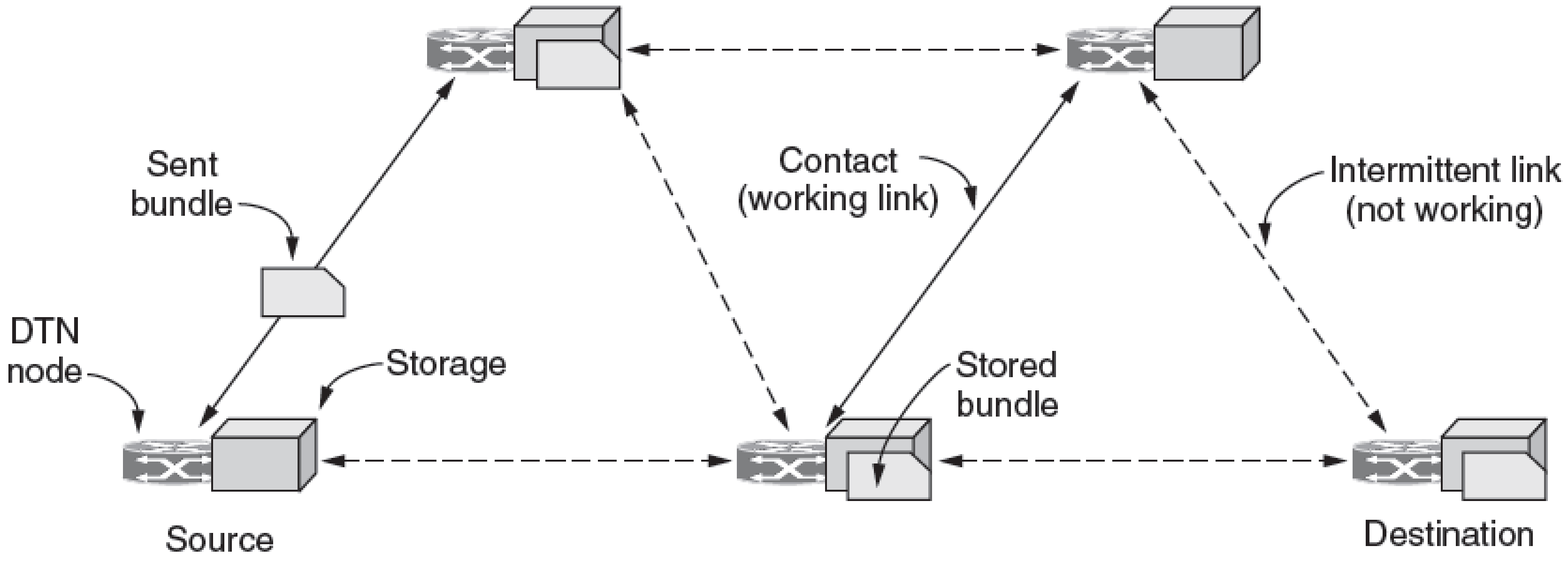


Time to transfer and acknowledge a 1-megabit file over a 4000-km line

Delay Tolerant Networking

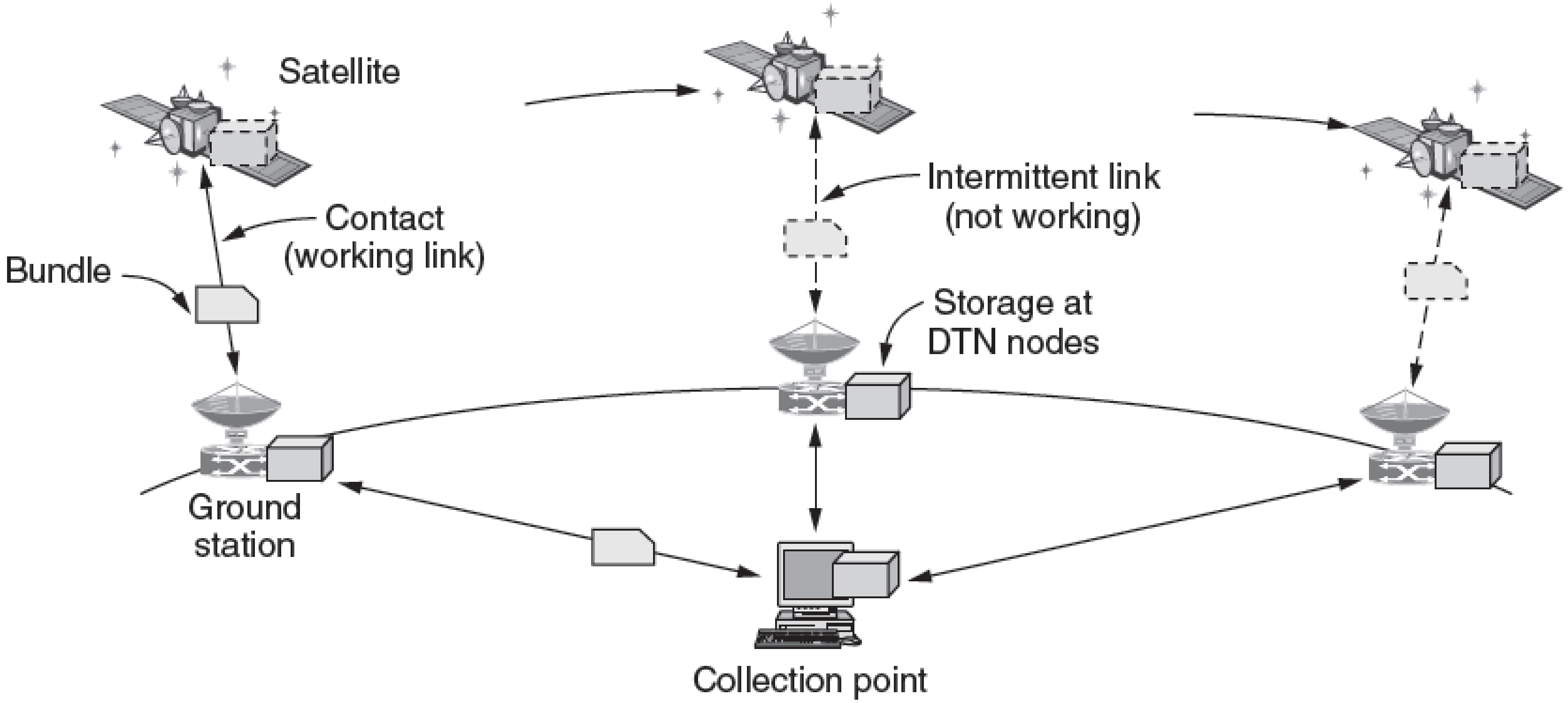
- DTN Architecture
- The Bundle Protocol

DTN Architecture (1)



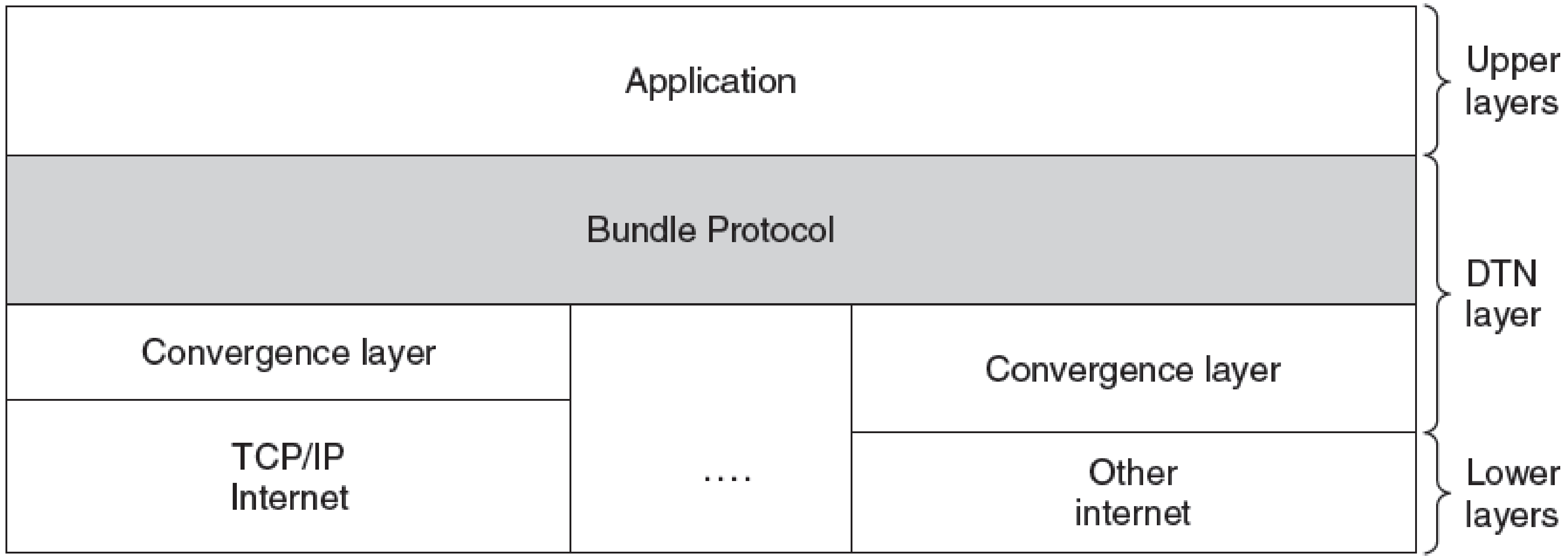
Delay-tolerant networking architecture

DTN Architecture (2)



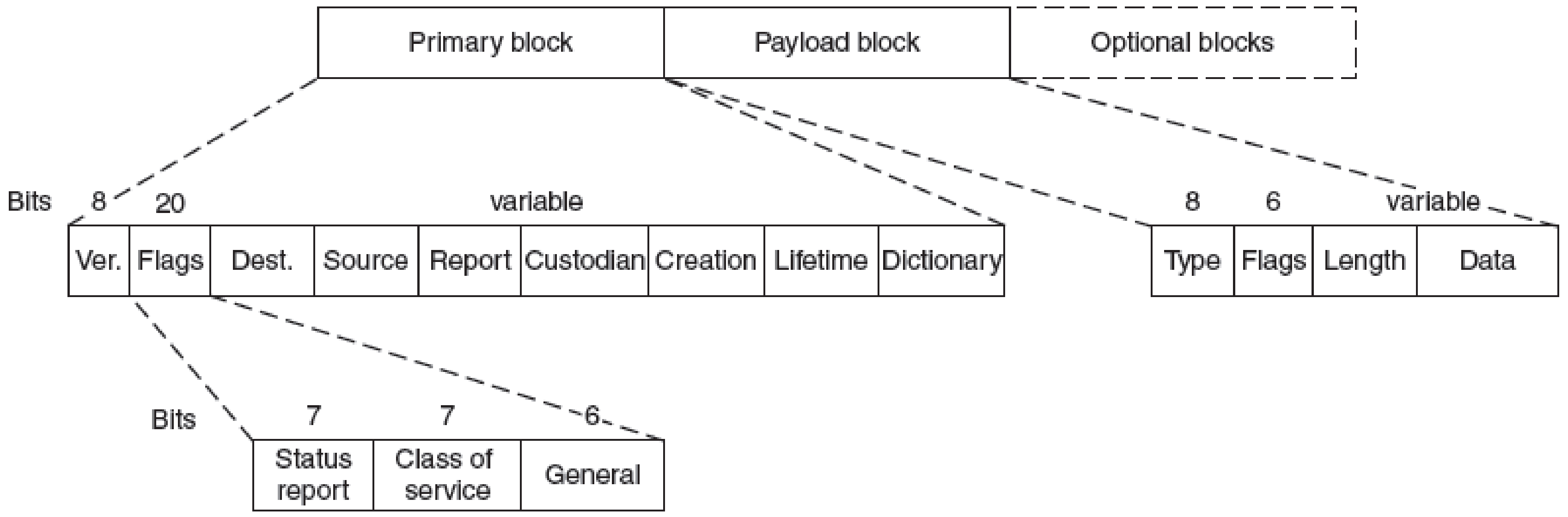
Use of a DTN in space.

The Bundle Protocol (1)



Delay-tolerant networking protocol stack.

The Bundle Protocol (2)



Bundle protocol message format.

End

Chapter 6