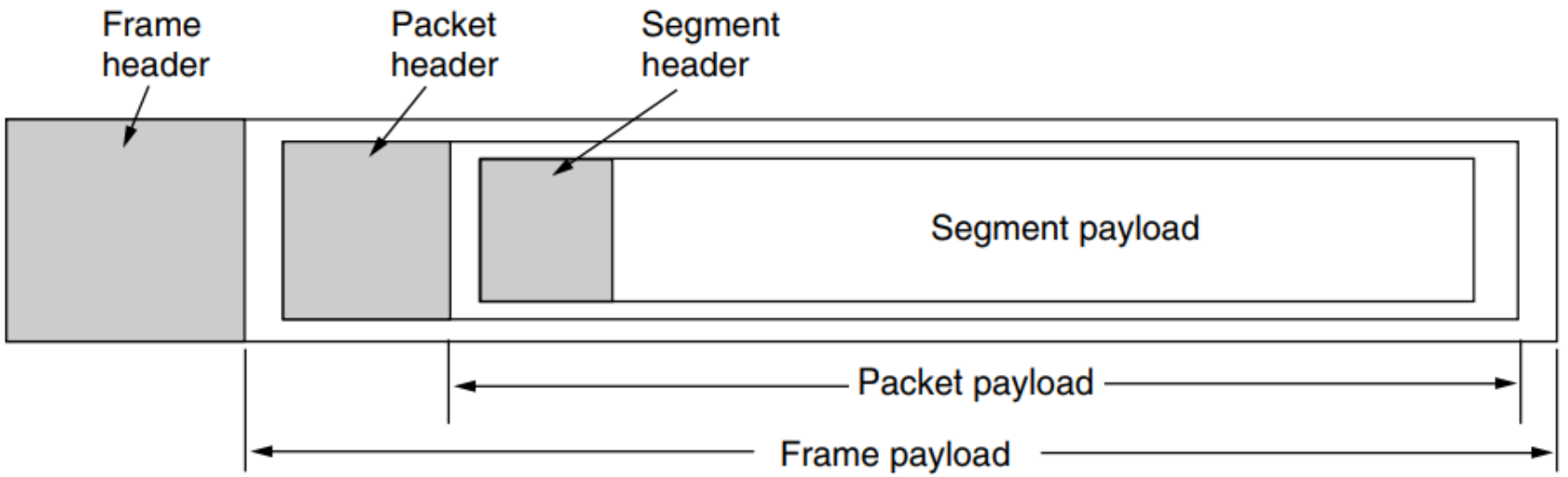
Transport Layer

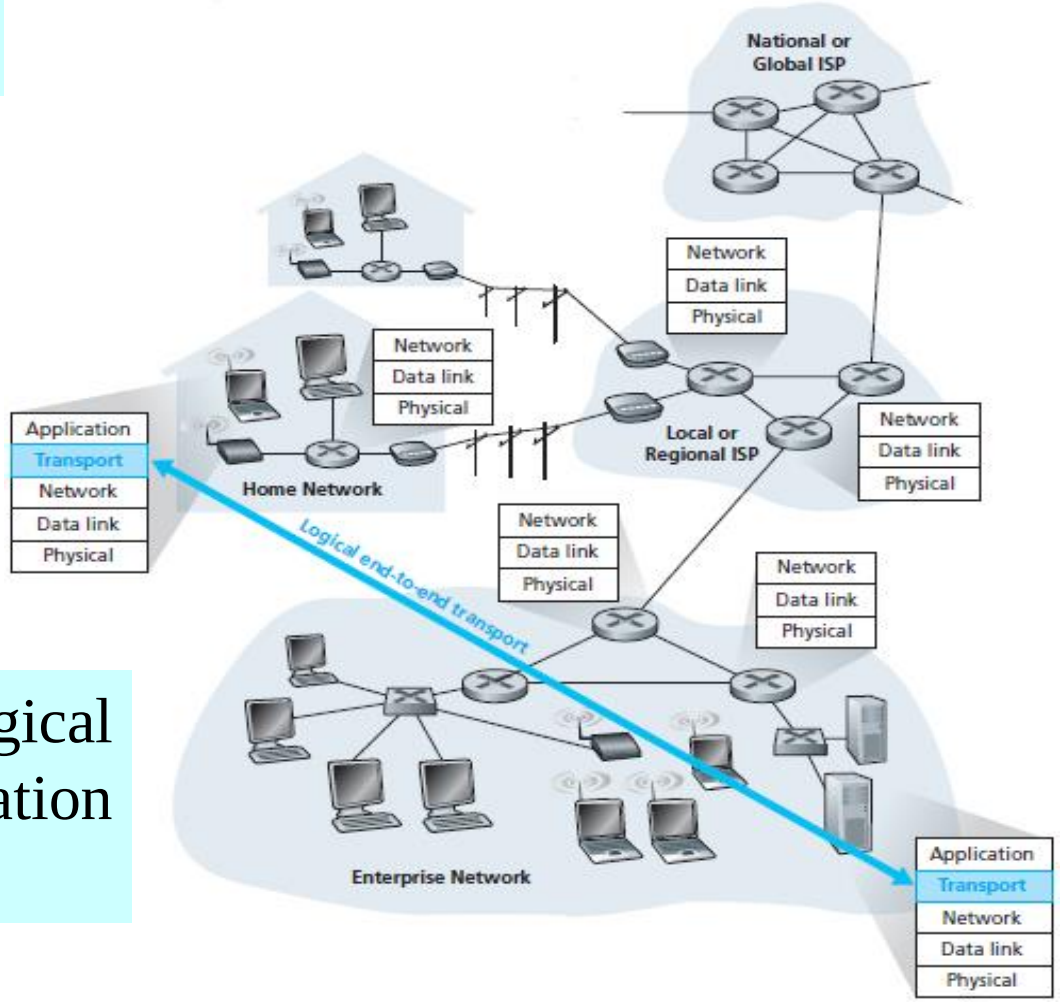
✓The task of transport layer is to provide reliable, cost-effective data transport from the source machine to the destination machine, independently of the physical network.

✓The hardware and/or software within the transport layer that does the work is called the **transport entity**.

✓We will use the term **segment** for messages sent from transport entity to transport entity. TCP, UDP and other Internet protocols use this term.



✓A transport-layer protocol provides logical communication between processes running on different hosts, a network-layer protocol provides logical communication between hosts. This distinction between these two layers is subtle but important. Let's examine this distinction with the aid of a household analogy.

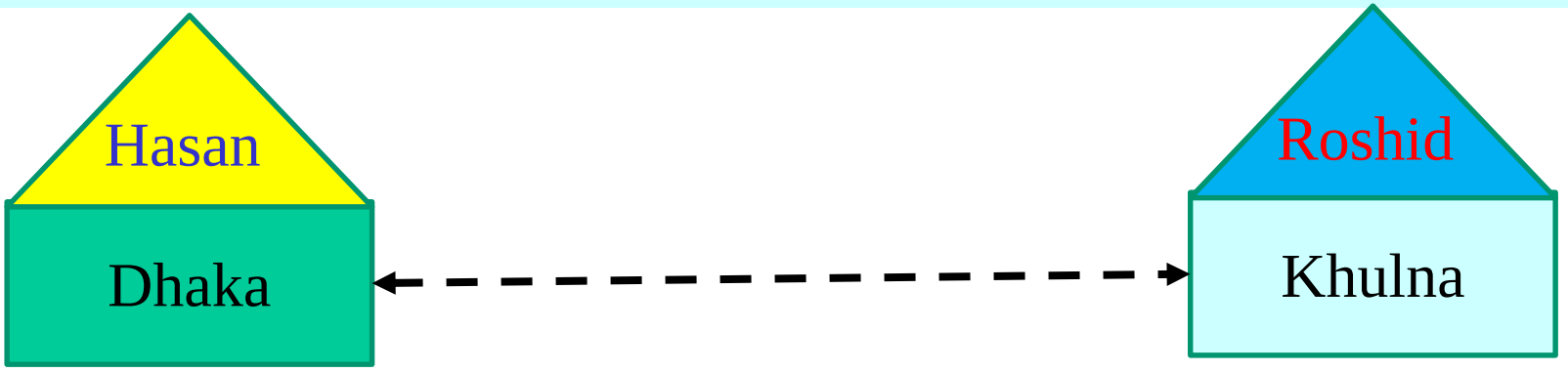


The transport layer provides logical rather than physical communication between application processes.

Household Analogy

✓Consider two houses, one on the Dhaka and the other on the Khulna, with each house being home to a **dozen kids**. The kids in the Dhaka household are cousins of the kids in the Khulna household.

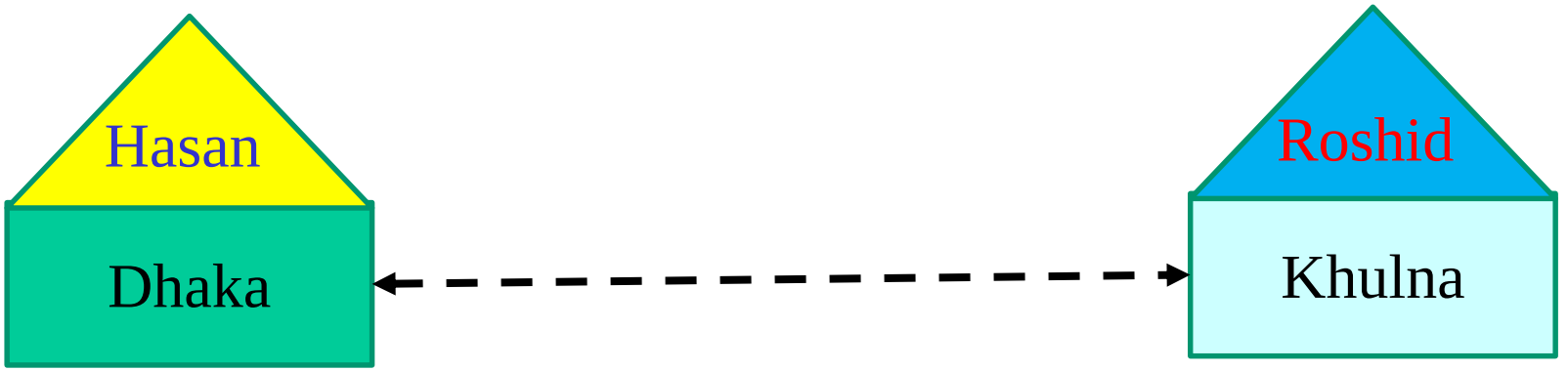
✓ The kids in the two households love to write to each other—each kid writes each cousin every week, with each letter delivered by the traditional postal service in a separate envelope. Thus, each household sends **144 letters** (12×12 i.e. all to all relation) to the other household every week.



✓In each of the households there is one kid—Roshid in the Khulna house and Hasan in the Dhaka house—responsible for mail collection and mail distribution.

✓Each week Roshid visits all his brothers and sisters, collects the mail, and gives the mail to a postal-service mail carrier, who makes daily visits to the house.

✓When letters arrive at the Khulna house, Roshid also has the job of distributing the mail to her brothers and sisters. Hasan has a similar job on the Dhaka.



✓In this example, the postal service provides logical communication between the two houses—the postal service moves mail from house to house, not from person to person.

✓On the other hand, Roshid and Hasan provide logical communication among the cousins — Roshid and Hasan pick up postal mail from, and deliver mail to, their brothers and sisters.

application messages = letters in envelopes
transport-layer protocol = Roshid and **Hasan**
processes = cousins (who order Hansan/Roshid to send the letter)
hosts (also called end systems) = houses
network-layer protocol = postal service (including mail carriers), who deals with several post offices hop-by-hop until reach the destination house.

✓On the sending side, the transport layer converts the application-layer messages it receives from a **sending application process** into **transport-layer packets**, known as **transport-layer segments** in Internet terminology.

✓This is done by (possibly) breaking the application messages into smaller chunks and adding a transport-layer header to each chunk to create the transport-layer segment.

✓The Internet has two main protocols in the transport layer, a **connectionless** protocol and a **connection-oriented** one. In the following sections we will study both of them. The connectionless protocol is **UDP (User Datagram Protocol)**. The connection-oriented protocol is **TCP (Transmission Control Protocol)**.

Connection Establishment

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

- ✓ The problem occurs when the network can lose, store, and duplicate packets. This behavior causes serious complications.

- The worst possible nightmare is as follows. A user establishes a connection with a bank, sends messages telling the bank to transfer a large amount of money to the account of a not-entirely-trustworthy person, and then releases the connection.

- Unfortunately, each packet in the scenario is duplicated and stored in the subnet. After the connection has been released, all the packets pop out of the subnet and arrive at the destination in order, asking the bank to establish a new connection, transfer money (again), and release the connection.

- The bank has no way of telling that these are duplicates. It must assume that this is a second, independent transaction, and transfers the money again.

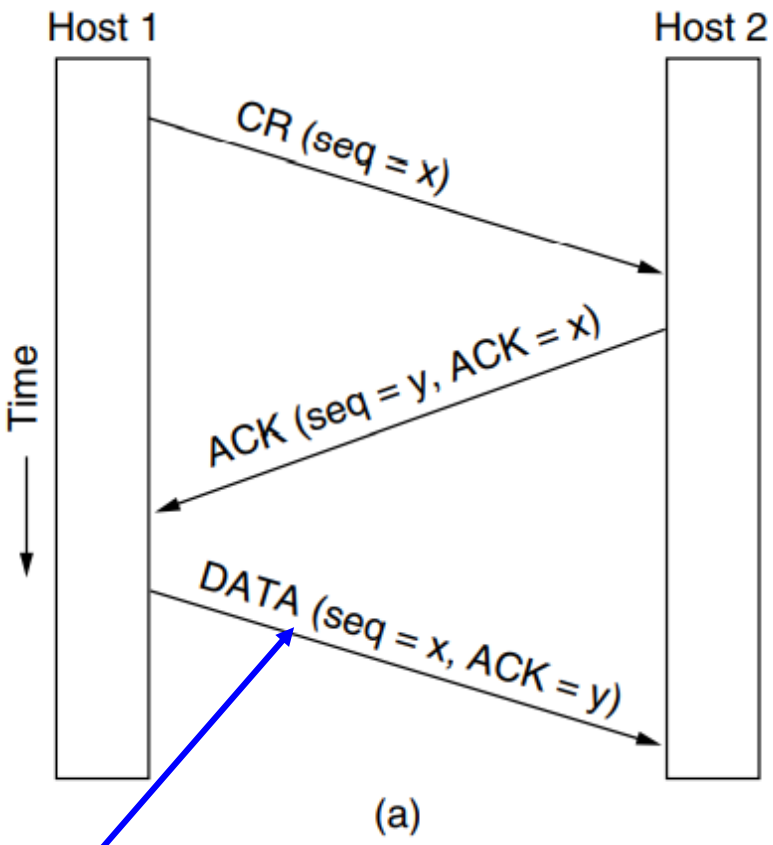
✓Tomlinson (1975) introduced the **three-way handshake.** The normal setup procedure when host 1 initiates is shown in fig. below. Host 1 chooses a sequence number, *x*, and sends a **CONNECTION REQUEST TPDU** containing it to host 2.

✓Host 2 replies with an **ACK TPDU** 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 TPDU** that it sends.

Seq = *x*

Seq = *y*

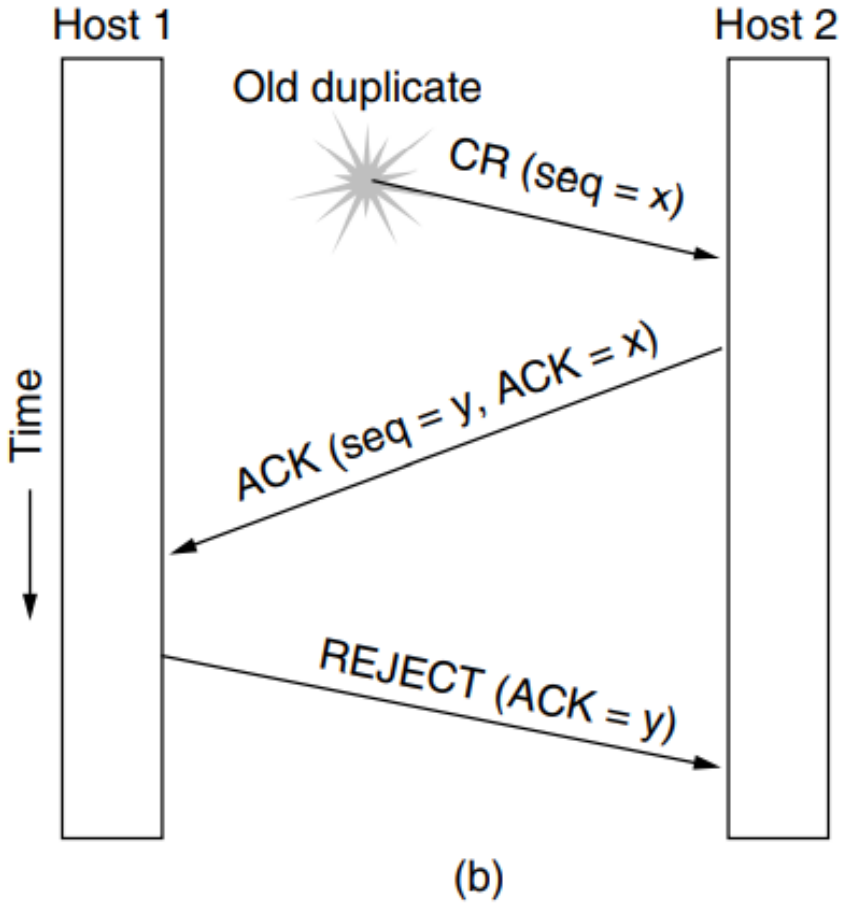


seq = *x* and ACK = *y* is sent only on first data segment but not on the subsequent segments.

✓Now let us see how the three-way handshake works in the presence of **delayed duplicate control TPDUs**.

✓In Fig. below, the first TPDU is a delayed duplicate CONNECTION REQUEST from an old connection. This TPDU arrives at host 2 without host 1's knowledge. Host 2 reacts to this TPDU by sending host 1 an ACK TPDU, in effect asking for verification that host 1 was indeed trying to set up a new connection.

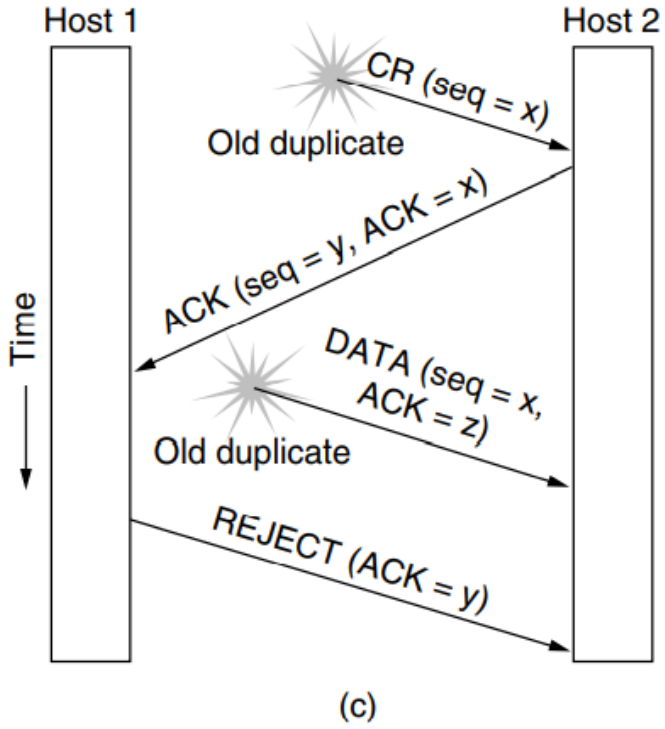
✓When host 1 rejects host 2's attempt to establish a connection, host 2 realizes that it was tricked by a delayed duplicate and abandons the connection. In this way, a delayed duplicate does no damage.



The worst case is when **both a delayed CONNECTION REQUEST and an ACK are floating around in the subnet**. This case is shown in Fig. below.

✓As in the previous example, host 2 gets a delayed **CONNECTION REQUEST** and replies to it using *y* as the initial sequence number.

✓When the second delayed TPDU arrives at host 2, the fact that z has been acknowledged rather than y tells host 2 that this, too, is an old duplicate.



(c)

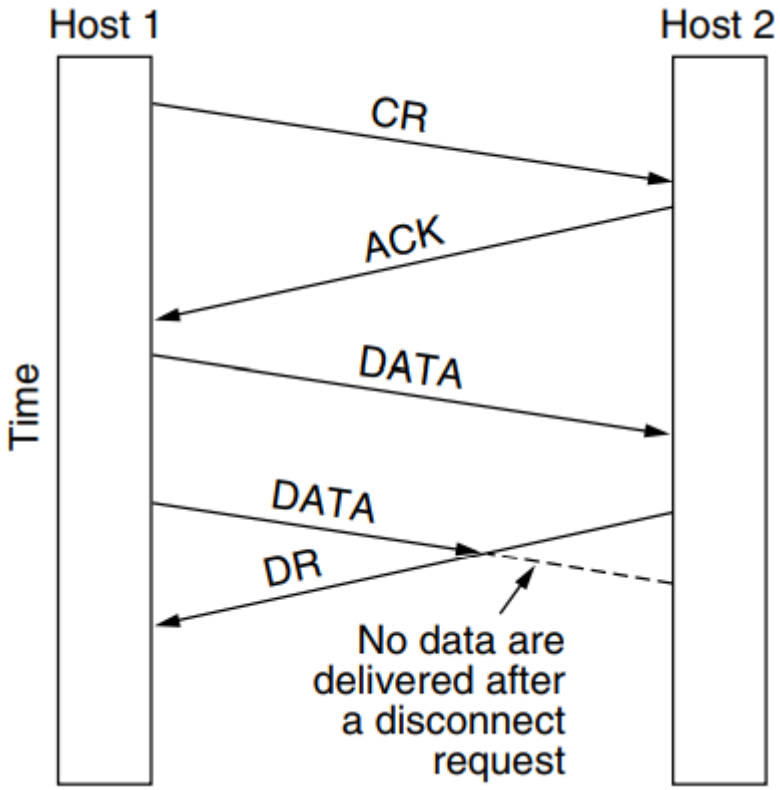
Connection Release

✓Releasing a connection is easier than establishing one. Nevertheless, there are more pitfalls than one might expect. There are two styles of terminating a connection: asymmetric release and symmetric release.

✓**Asymmetric release** is the way the telephone system works: when one party hangs up, the connection is broken. **Symmetric release** treats the connection as two separate unidirectional connections and requires each one to be released separately.

- ✓ Asymmetric release is abrupt and may result in data loss. Consider the scenario of Fig. below. After the connection is established, host 1 sends a TPDU that arrives properly at host 2. Then host 1 sends another TPDU. Unfortunately, host 2 issues a DISCONNECT before the second TPDU arrives. The result is that the connection is released and data are lost.

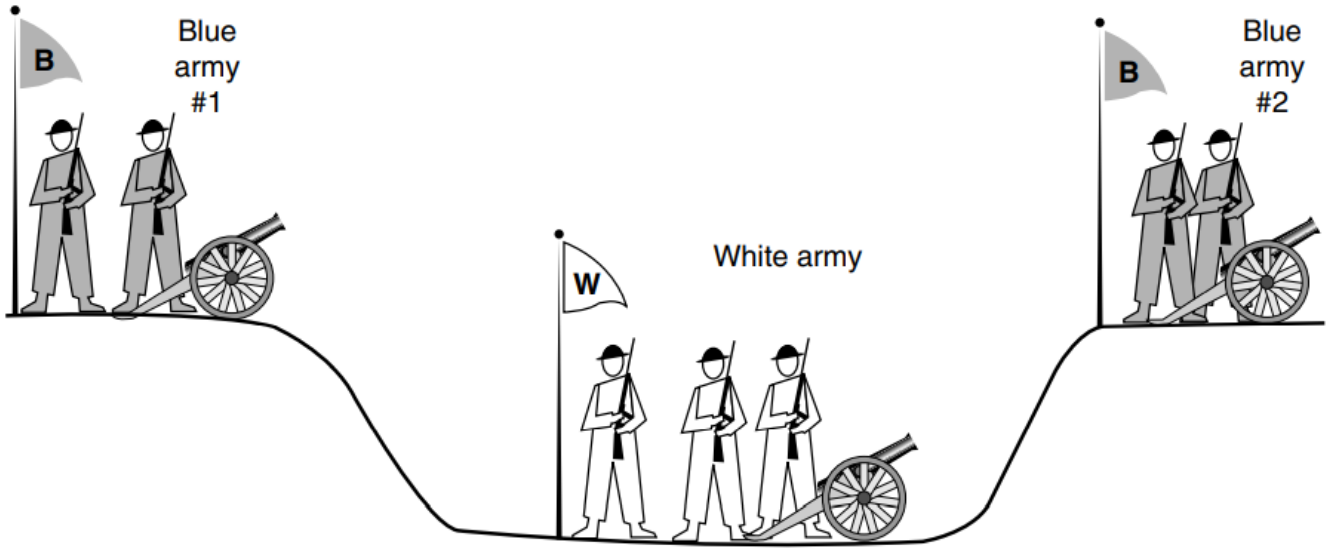
- ✓ Clearly, a more sophisticated release protocol is needed to avoid data loss.



✓One way is to use **symmetric** release, in which each direction is released independently of the other one. Here, a host can continue to receive data even after it has sent a DISCONNECT TPDU.

✓Symmetric release does the job when each process has a fixed amount of data to send and clearly knows when it has sent it.

✓One can envision a protocol in which host 1 says: I am done. Are you done too? If host 2 responds: I am done too. Goodbye, the connection can be safely released. Unfortunately, this protocol does not always work. There is a famous problem that illustrates this issue. It is called the **two-army problem.**



Suppose that the commander of blue army #1 sends a message reading: "I propose we attack at dawn on March 29. How about it?" Now suppose that the message arrives, the commander of blue army #2 agrees, and his reply gets safely back to blue army #1. Will the attack happen? Probably not, because commander #2 does not know if his reply got through. If it did not, blue army #1 will not attack, so it would be foolish for him to charge into battle

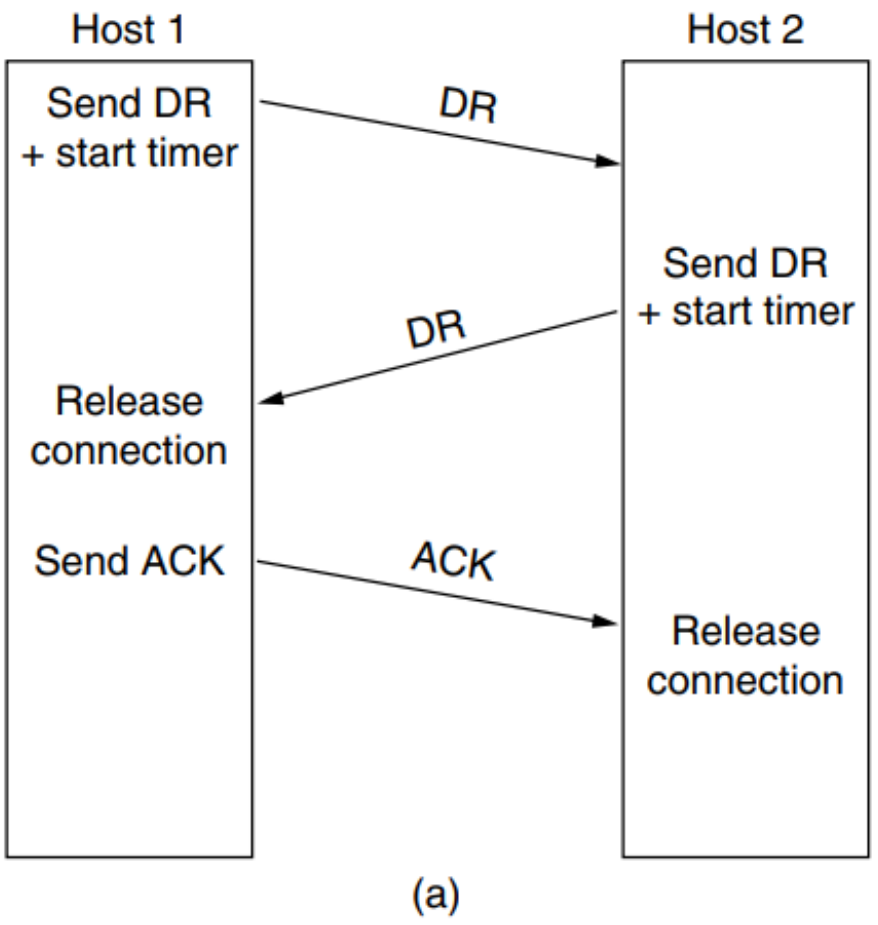
✓We will next consider using a three-way handshake for four scenarios of releasing connection.

✓In Fig. (a), we see the normal case in which one of the users sends a DR (**DISCONNECTION REQUEST**) TPDU to initiate the connection release.

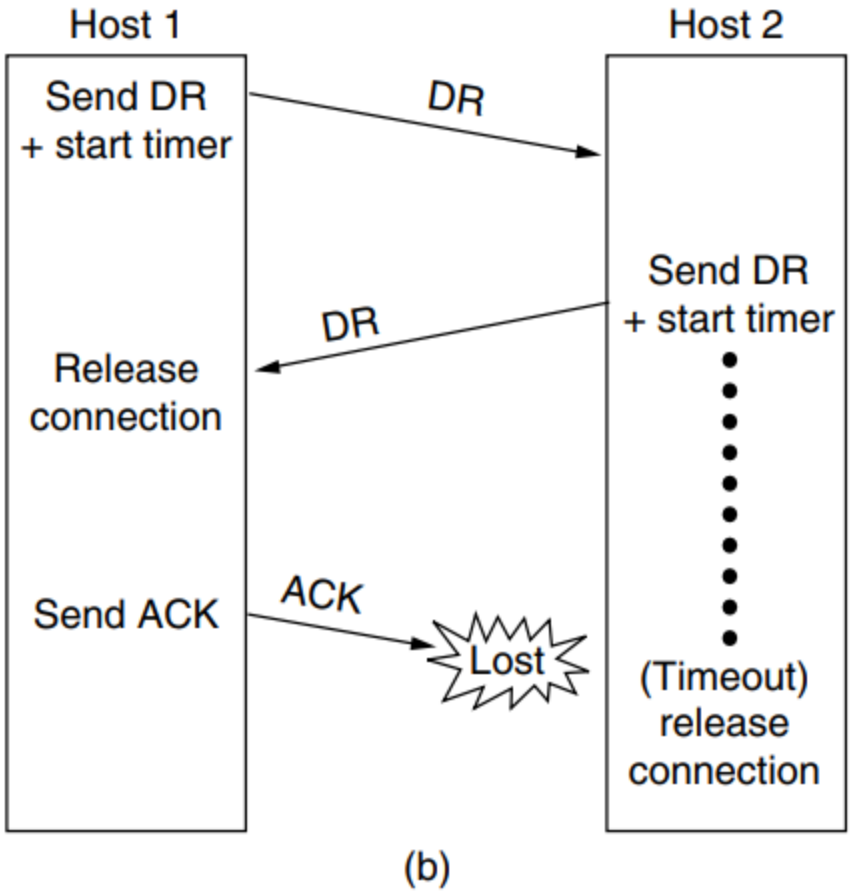
✓When DR arrives, the recipient sends back a DR TPDU, too, and **starts a timer**, just in case its DR is lost.

✓When this DR arrives, the original sender sends back an ACK TPDU and releases the connection.

✓Finally, when the ACK TPDU arrives, the receiver also releases the connection.

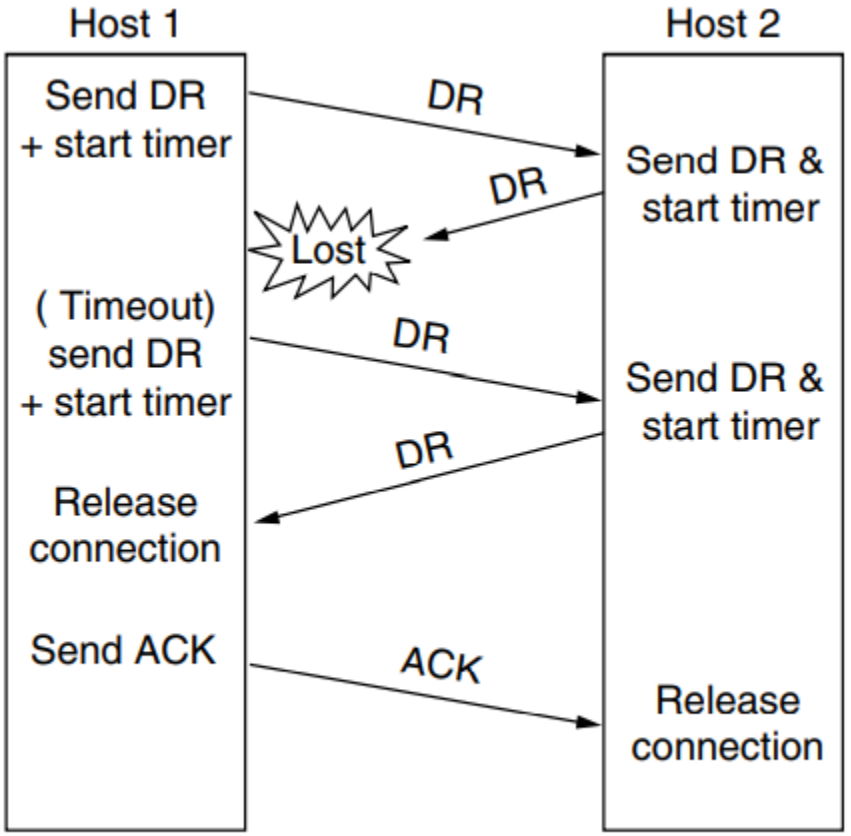


If the final ACK TPDU is lost, as shown in Fig. 6-14(b), the situation is saved by the timer. When the timer expires, the connection is released anyway.



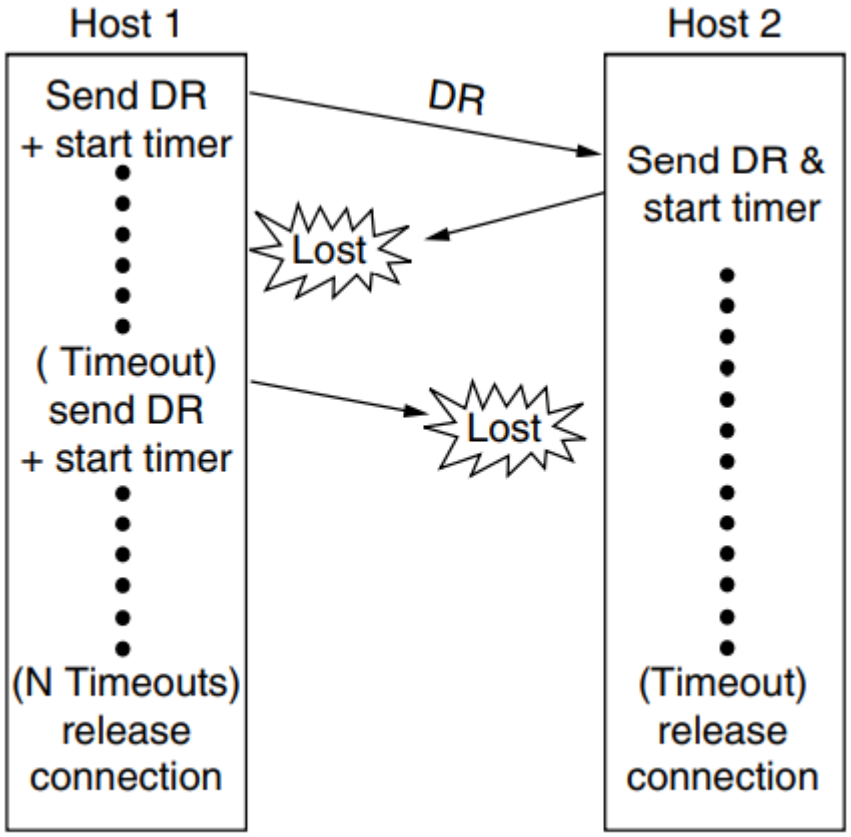
(b)

Now consider the case of the second DR being lost. The user initiating the disconnection will not receive the expected response, will time out, and will start all over again. In Fig. (c) we see how this works, assuming that the second time no TPDUs are lost and all TPDUs are delivered correctly and on time.



(c)

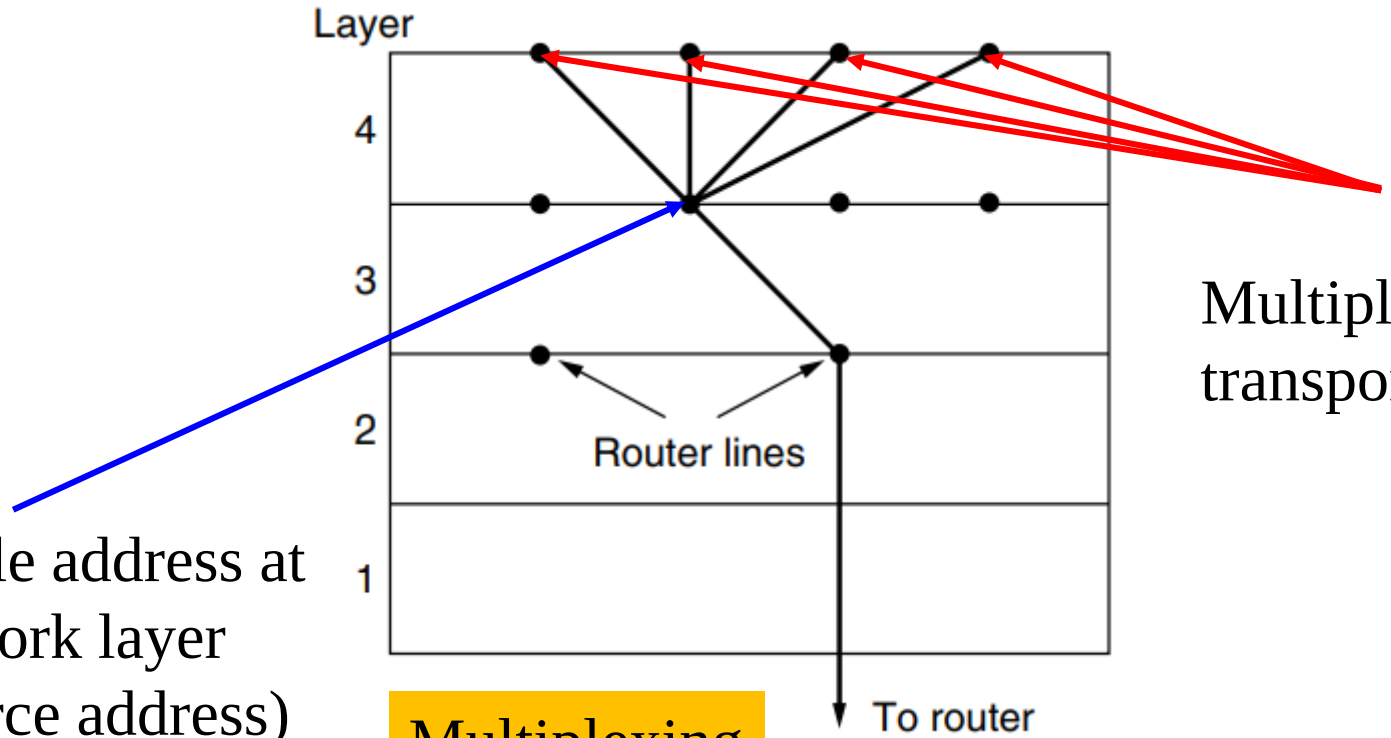
Our last scenario, Fig. (d), is the same as Fig. (c) except that now we assume all the repeated attempts to retransmit the DR also fail due to lost TPDUs. After N retries, the sender just gives up and releases the connection. Meanwhile, the receiver times out and also exits.



(d)

Multiplexing

If only one network address is available on a host, all transport connections on that machine have to use it. When a segment comes in (during reception), some way is needed to tell which process to give it to. This situation, called **multiplexing**, is shown in Fig. below.

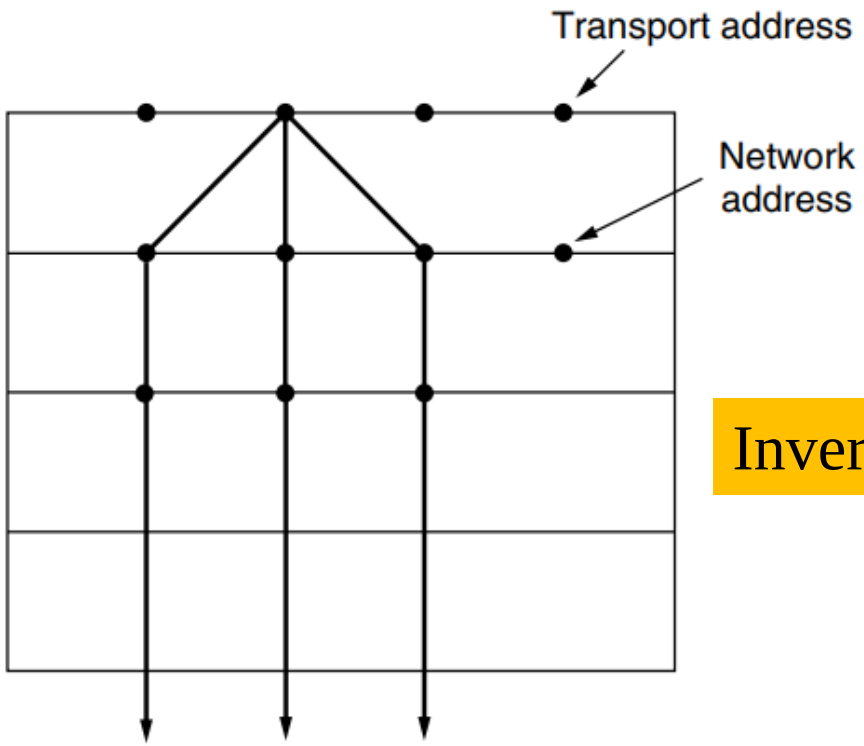


Multiple process at transport layer.

Single address at network layer (source address)

Multiplexing

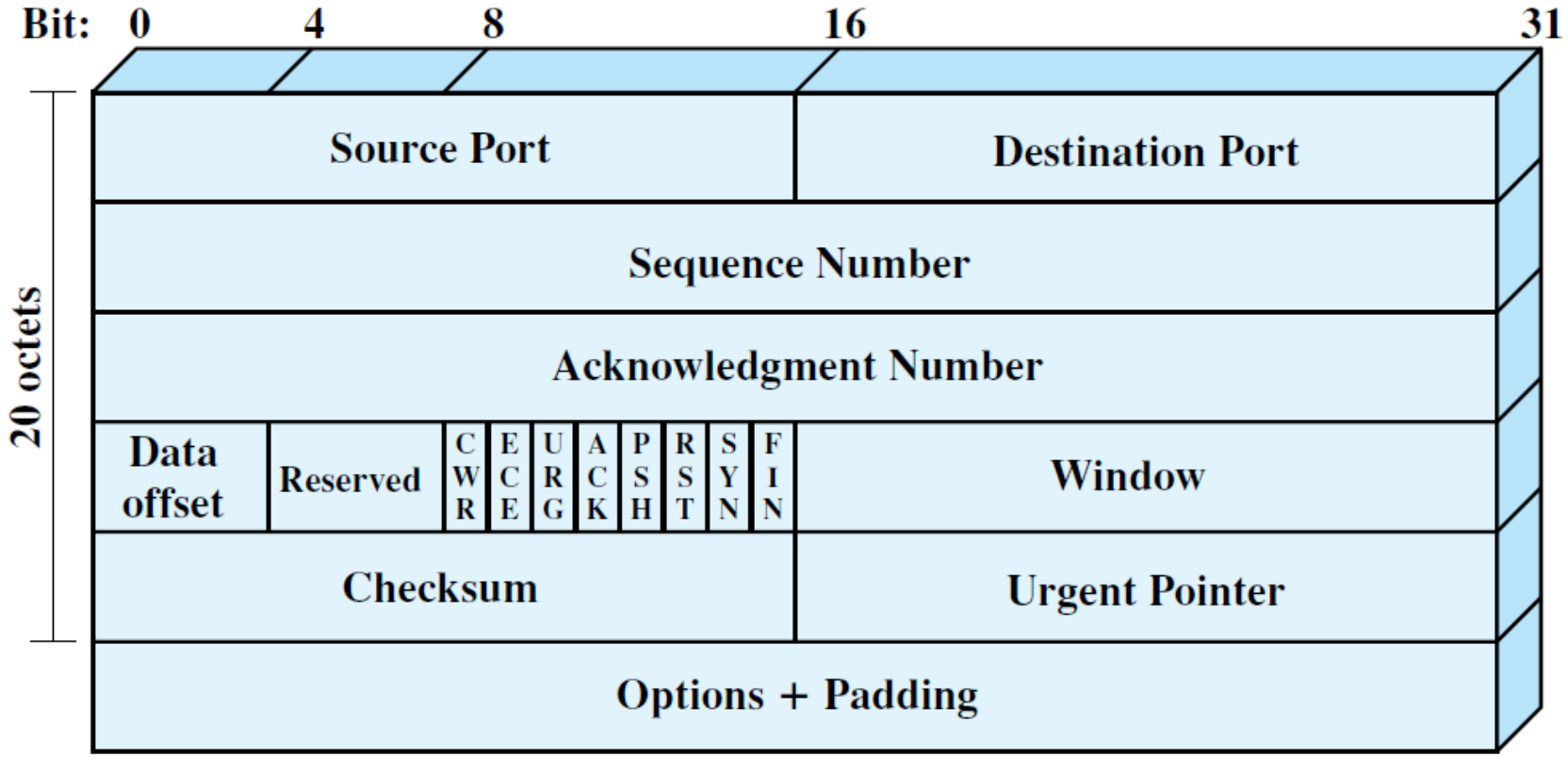
- ✓ If a user needs more bandwidth or more reliability than one of the network paths can provide, a way out is to have a connection that distributes the traffic among multiple network paths on a round-robin basis, as indicated in Fig. below.
- ✓ This mode of operation is called **inverse multiplexing**. With *k* network connections open, the effective bandwidth might be increased by a factor of *k*.



(b)

Inverse multiplexing

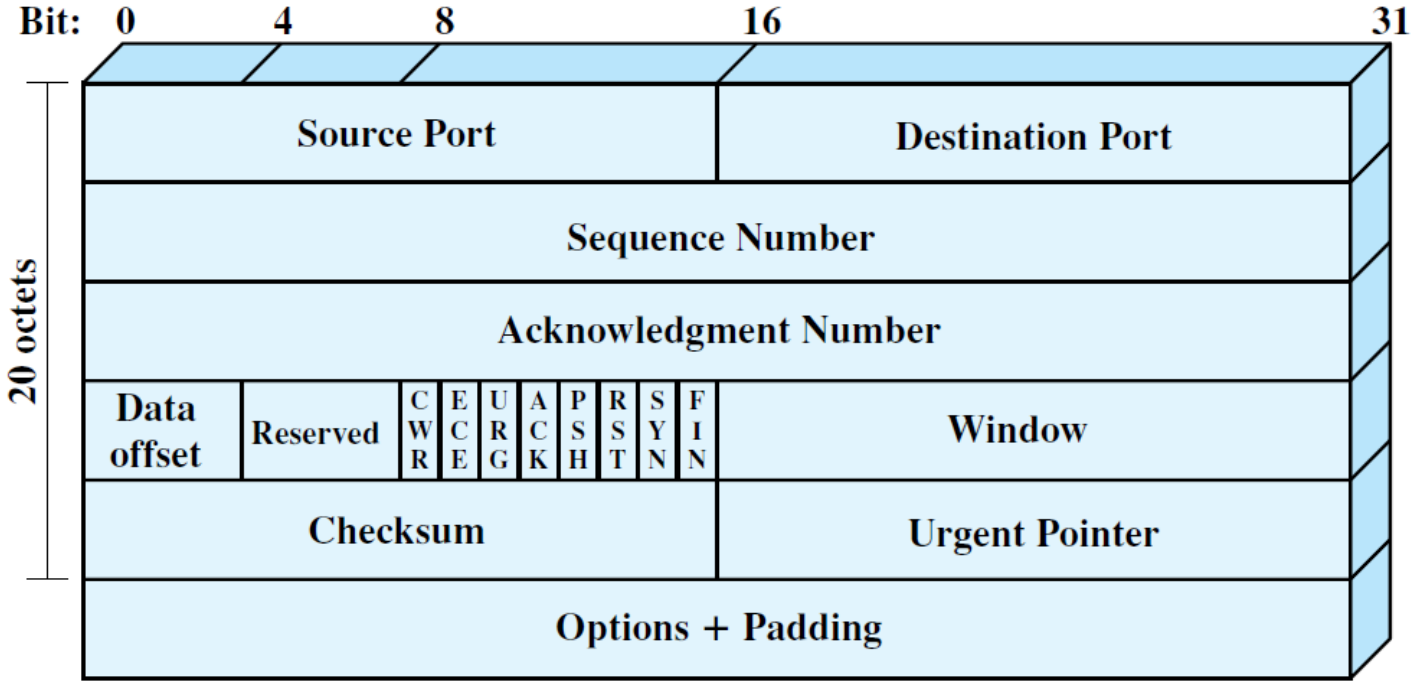
TCP Header


Bit: 0
4
8
16
31
20 octets
Source Port
Destination Port
Sequence Number
Acknowledgment Number
Data offset
Reserved
CWR
ECE
URG
ACK
PSH
RST
SYN
FIN
Window
Checksum
Urgent Pointer
Options + Padding

❖**Source Port (16 bits): Source TCP user. Example values are Telnet 23;**A complete list is maintained at **http://www.iana.org/assignments/port-numbers.**

❖**Destination Port (16 bits): Destination TCP user.**

❖**Sequence Number (32 bits):** Sequence number of the first data octet in this segment except when the SYN flag is set. If SYN is set, this field contains the initial sequence number (ISN) and the first data octet in this segment has sequence number ISN + 1.



❖**Acknowledgment Number (32 bits): Contains the sequence number of the**

next data octet that the TCP entity expects to receive.

❖ **Data Offset (4 bits): Number of 32-bit words in the header.**

❖ **Reserved (4 bits): Reserved for future use.**

❖ **Flags (6 bits):** For each flag, if set to 1, the meaning is

CWR: congestion window reduced.

ECE: ECN-Echo; the CWR and ECE bits, defined in RFC 3168, are used for the explicit congestion notification function; a discussion of this function is beyond our scope.

URG: urgent pointer field significant.

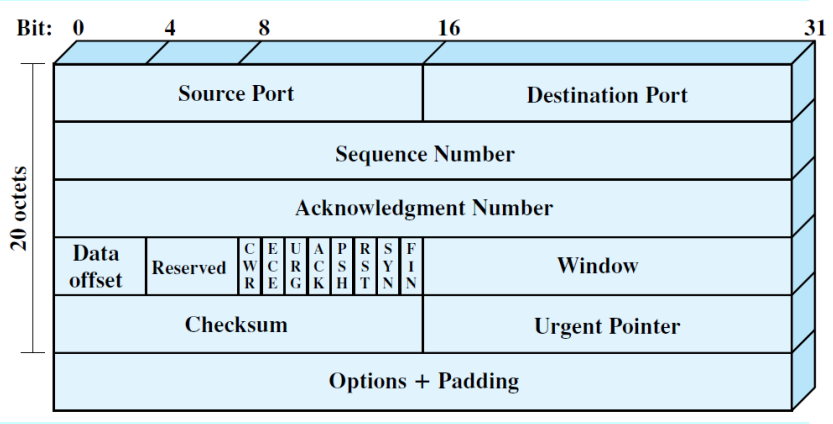
ACK: acknowledgment field significant.

PSH: push function.

RST: reset the connection.

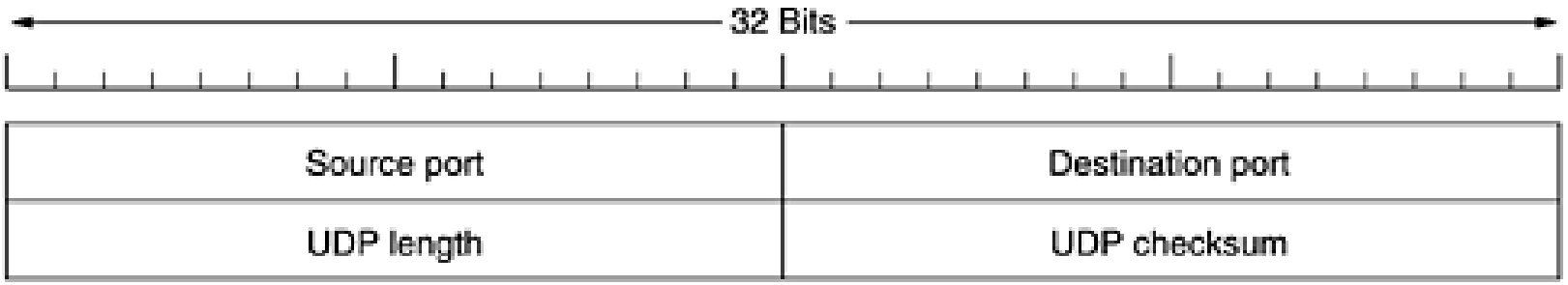
SYN: synchronize the sequence numbers

FIN: no more data from sender.



The **Sequence Number and Acknowledgment Number** are bound to octets rather than to entire segments. For example, if a segment contains sequence number 1001 and includes 600 octets of data, the sequence number refers to the first octet in the data field; the next segment in logical order will have sequence number 1601.

UDP transmits segments consisting of an 8-byte header followed by the payload. The header is shown in Fig. 6-23. The two ports serve to identify the end points within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port.



Congestion Control

Regulating the Sending Rate

✓When the load offered to any network is more than it can handle, congestion builds up.

✓When a connection is established, a suitable **window** size (amount of data can be sent without acknowledgement, initially size of the TCP packet) has to be chosen. The receiver can specify a **window** based on its buffer size.

✓If the sender sticks to the receiver's window size, problems will not occur due to buffer overflow at the receiving end, but they may still occur due to internal congestion within the network.

- ✓ In Fig.(a)-(b), we see this problem illustrated hydraulically. In Fig. (a), we see a thick pipe leading to a small-capacity receiver. As long as the sender does not send more water than the bucket can contain, no water will be lost.

- ✓ In Fig. (b) the limiting factor is not the bucket capacity, but the internal carrying capacity of the network. If too much water comes in too fast, it will back up and some will be lost (in this case by overflowing the funnel).

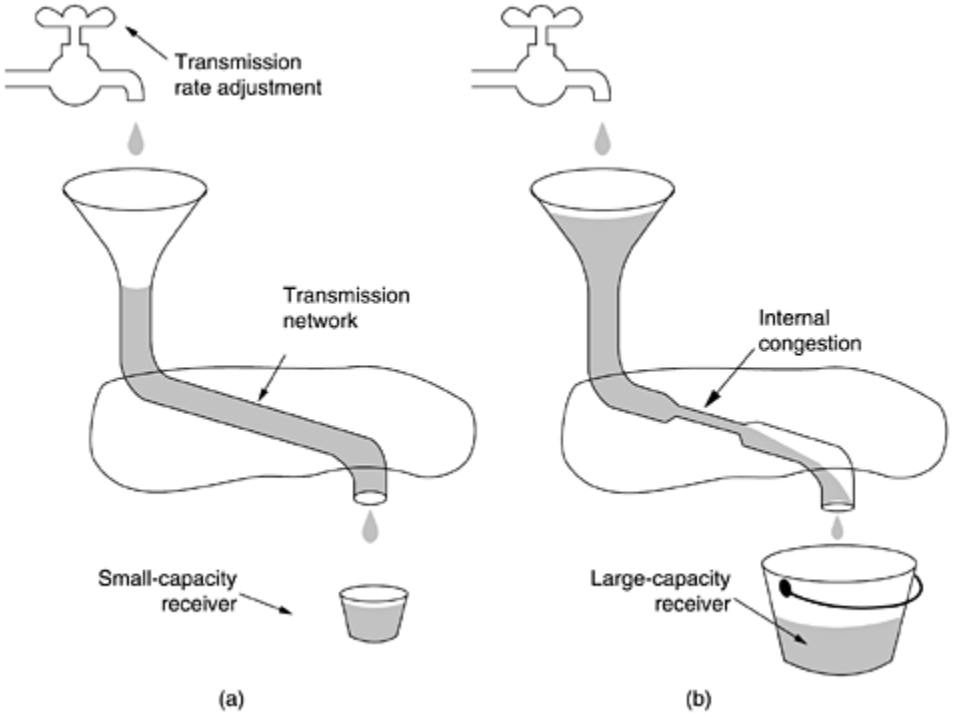


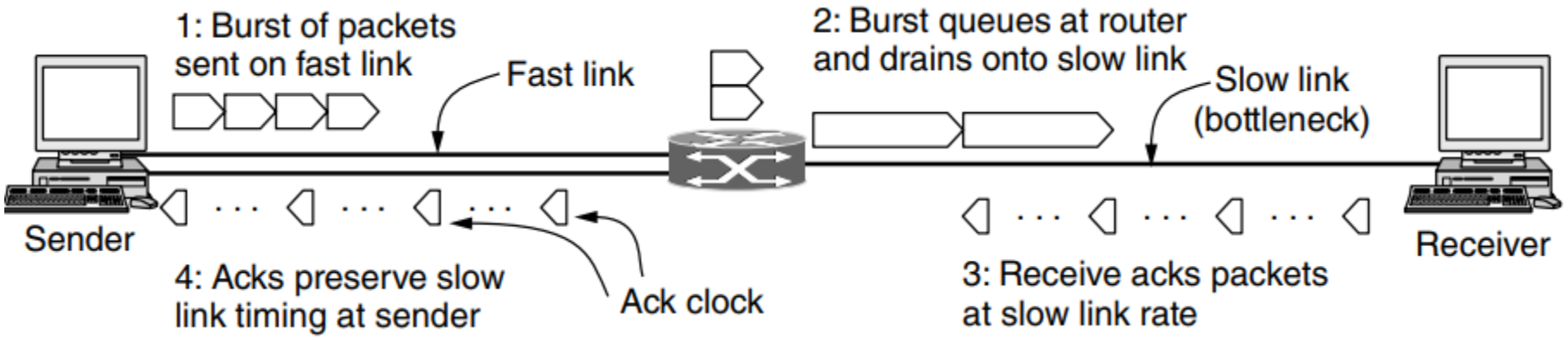
Figure (a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.

✓The Internet solution is to realize that two potential problems exist: network capacity and receiver capacity and to deal with each of them separately. To do so, each sender maintains two windows: **the window the receiver has granted** and **a second window, the congestion window**.

TCP Congestion Control

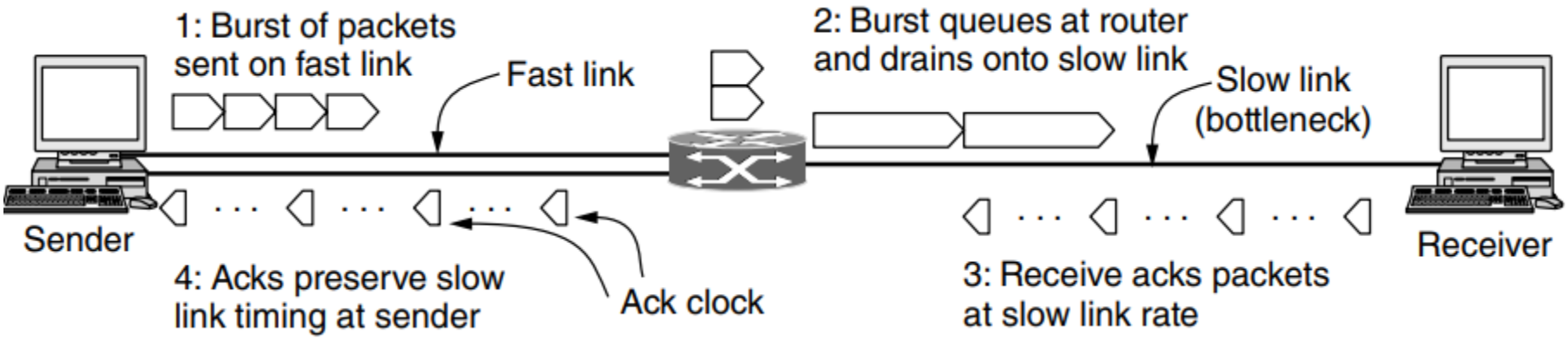
- ✓ Fig. below shows what happens when a sender on a fast network (the 1-Gbps link) sends a small burst of four packets to a receiver on a slow network (the 1- Mbps link) that is the bottleneck or slowest part of the path.

- ✓ Initially the four packets travel over the link as quickly as they can be sent by the sender. At the router, they are queued while being sent because it takes longer to send a packet over the slow link than to receive the next packet over the fast link.



- ✓ Eventually the packets get to the receiver, where they are acknowledged. The times for the acknowledgements reflect the times at which the packets arrived at the receiver after crossing the slow link.

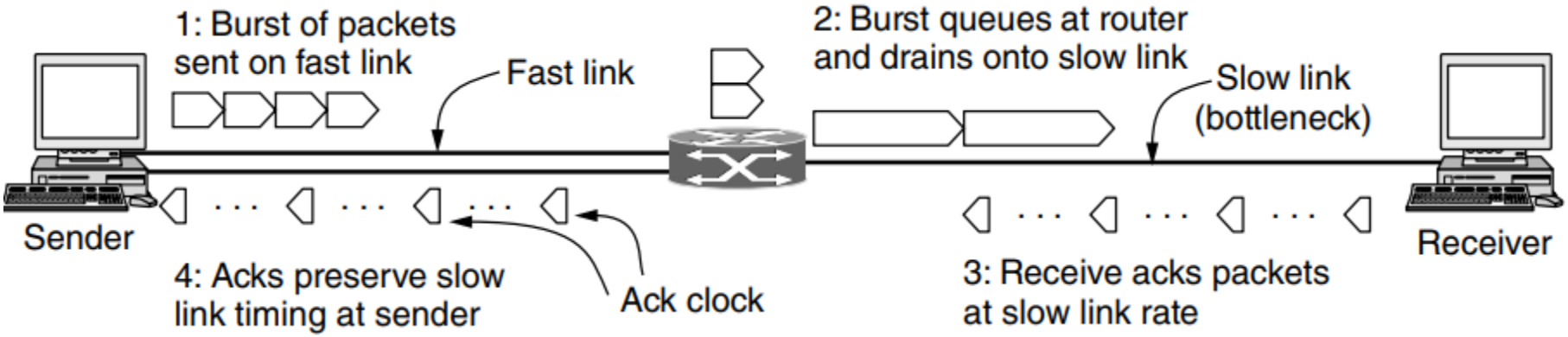
- ✓ They are spread out compared to the original packets on the fast link. As these acknowledgements travel over the network and back to the sender they preserve this timing.



A burst of packets from a sender and the returning ack clock

- ✓ The key observation is this: the acknowledgements return to the sender at about the rate that packets can be sent over the slowest link in the path. This is precisely the rate that the sender wants to use.
- ✓ If it injects new packets into the network at this rate, they will be sent as fast as the slow link permits, but they will not queue up and congest any router along the path. This timing is known as an **ack clock**. It is an essential part of TCP.

- ✓ By using an ack clock, TCP smoothes out traffic and avoids unnecessary queues at routers.



A burst of packets from a sender and the returning ack clock

Congestion Window

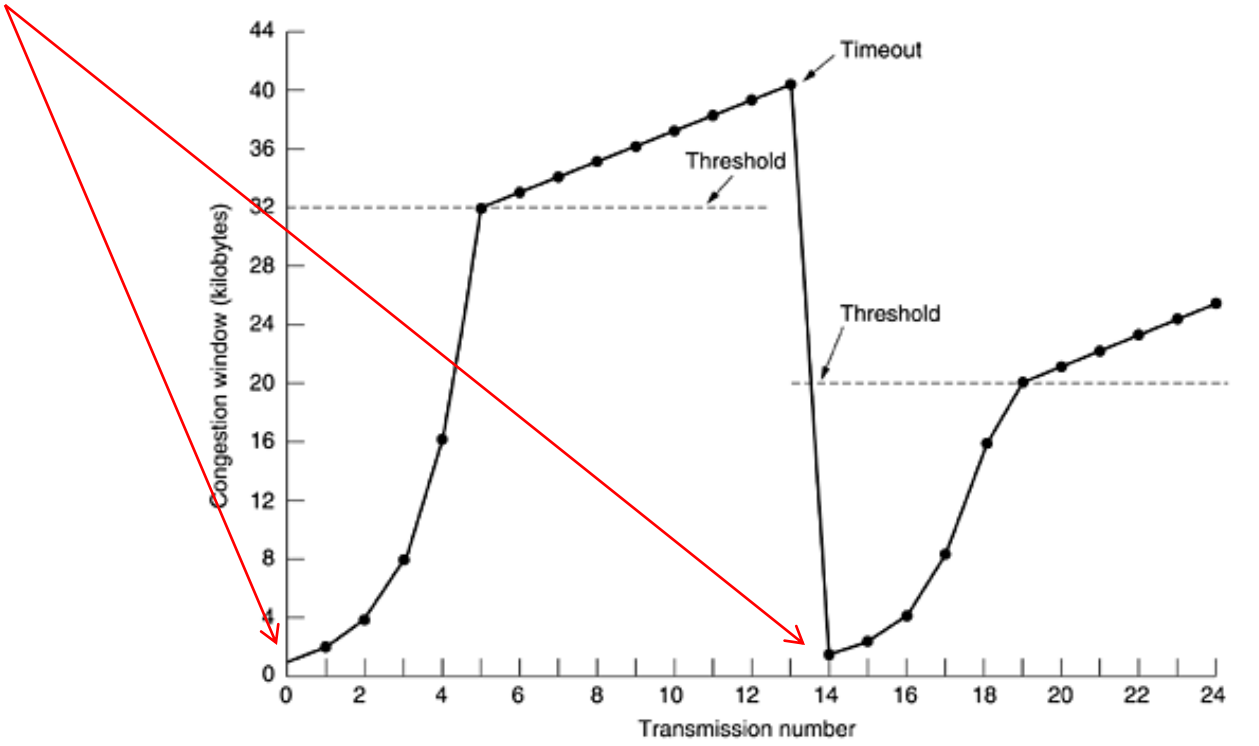
TCP maintains a new state variable for each connection, called Congestion Window, which is used by the source to limit how much data it is allowed to have in transit at a given time. The unit of Congestion Window is number of packets.

Her we consider two congestion control algorithms graphically: TCP Tahoe and TCP Reno.

Internet congestion control algorithm

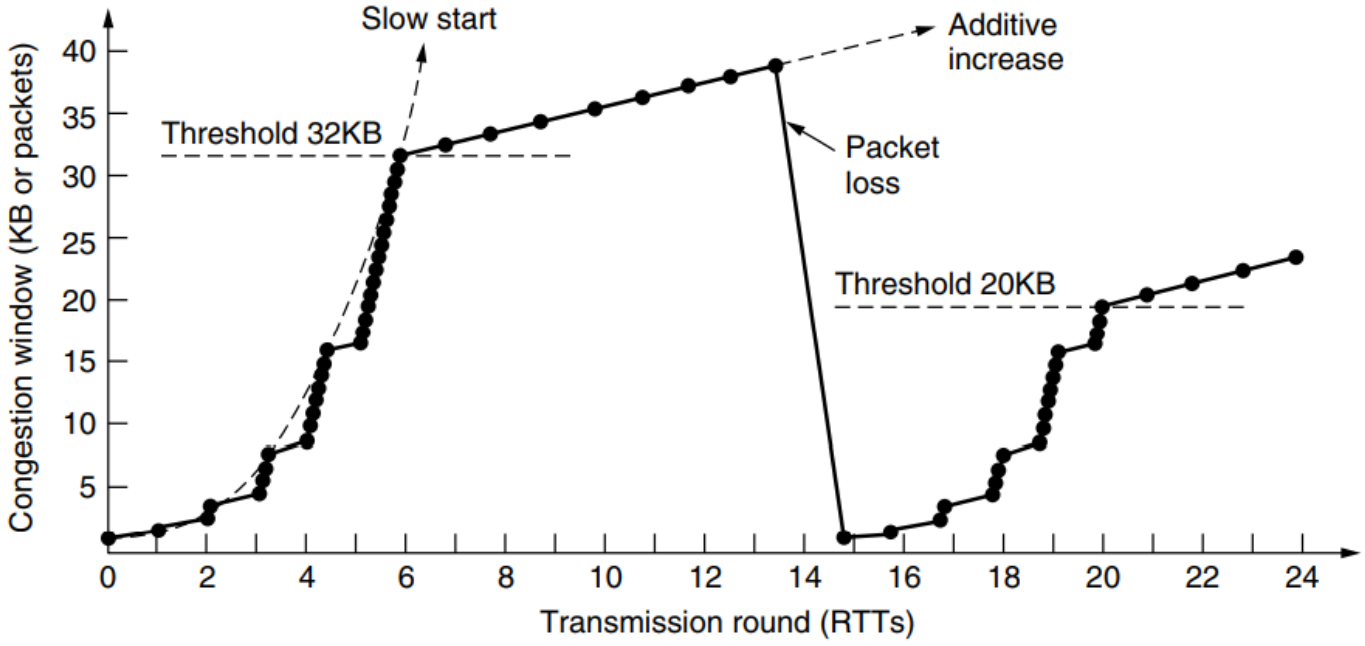
✓As an illustration of how the congestion algorithm works, see Fig. below. The maximum segment size (the size of a packet is called maximum segment size in TCP terminology) here is 1024 bytes or 1KB. Congestion Window is not allowed to fall below the size of a single packet.

✓After timeout the algorithm starts with size of the congestion window = size of a packet = 1KB.



✓The threshold value of congestion window is the maximum allowed value of the congestion window till which the window grows exponentially. The exponential growth means congestion window W, is *increased* by 1/W each time a regular ACK is received

✓Initially, the threshold congestion window was 64 KB, but a timeout occurred, so the threshold is set to 32 KB and the congestion window to 1 KB for transmission 0 here. The congestion window then grows exponentially until it hits the threshold (32 KB). Starting then, it grows linearly.



Slow start followed by additive increase in TCP Tahoe

✓If no more timeouts occur, the congestion window will continue to grow up to the size of the receiver's window. At that point, it will stop growing and remain constant as long as there are no more timeouts and the receiver's window does not change size.

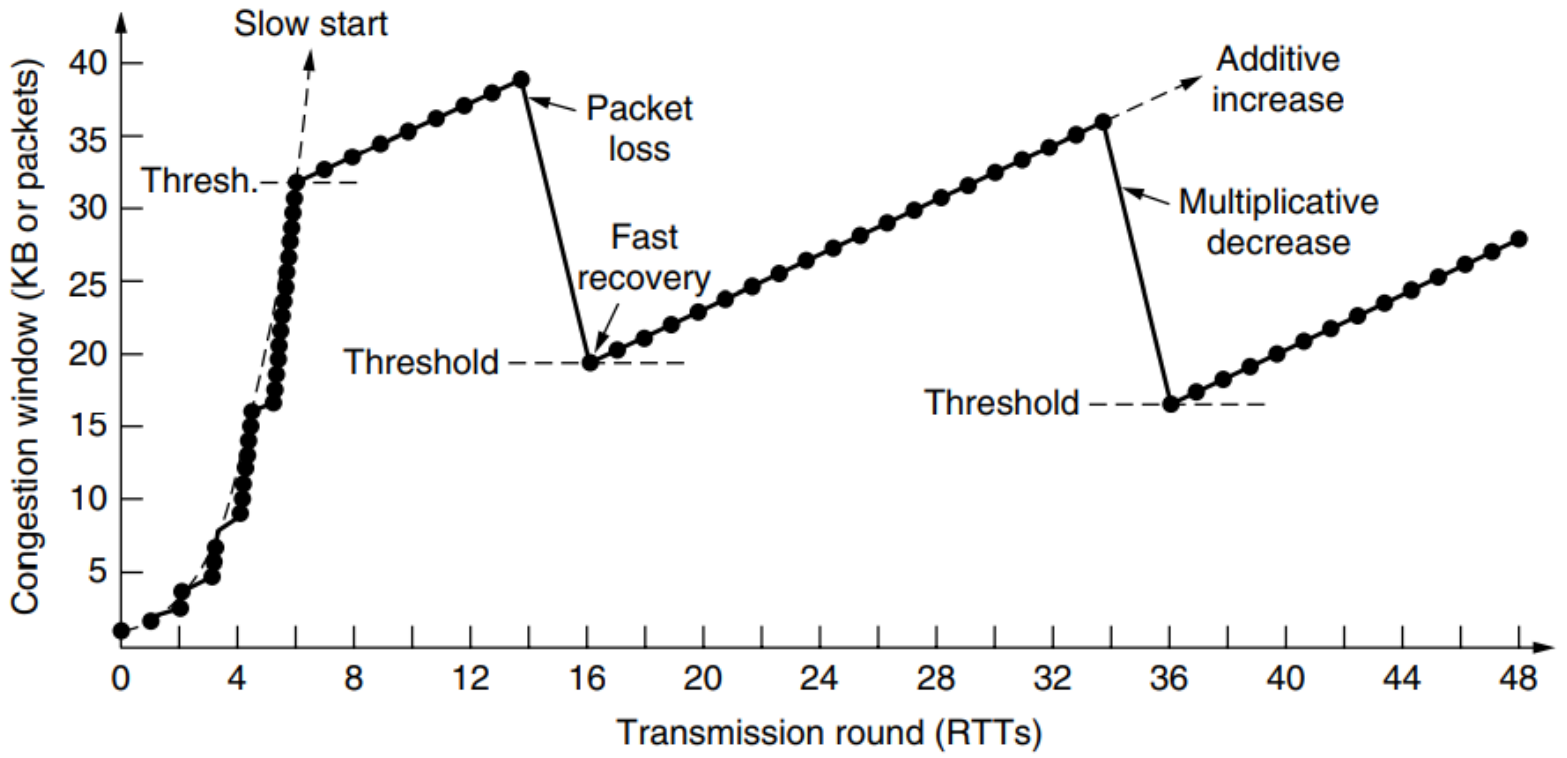
✓If the loss is detected by *the triple dup ACKs*, the lost packet is retransmitted, the threshold is set to half the current window and slow start is initiated all over again.

Triple duplicate ACK

- ❖Packet *n* is lost, but packets *n*+1, *n*+2, etc. arrive
- ❖Receiver sends duplicate acknowledgments and the sender retransmits packet *n* quickly

Do a multiplicative decrease and keep going

In **TCP Reno**, instead of repeated slow starts, the congestion window of a running connection follows a sawtooth pattern of additive increase (by one segment every RTT) and multiplicative decrease (by half in one RTT).



. Fast recovery and the sawtooth pattern of TCP Reno.

- ✓ After an initial slow start, the congestion window climbs linearly until a packet loss is detected by duplicate acknowledgements.
- ✓ The lost packet is retransmitted and fast recovery (**fast recovery** aims to maintain the **ack clock** running with a congestion window that is the new threshold, or half the value of the congestion window at the time of the fast retransmission) is used.
- ✓ Under **fast recovery** the congestion window is resumed from the new slow start threshold, rather than from 1.