Computer Networks

Malik Algazaeery

Chapter3-Transport Layer

Chapter-3 Agenda:

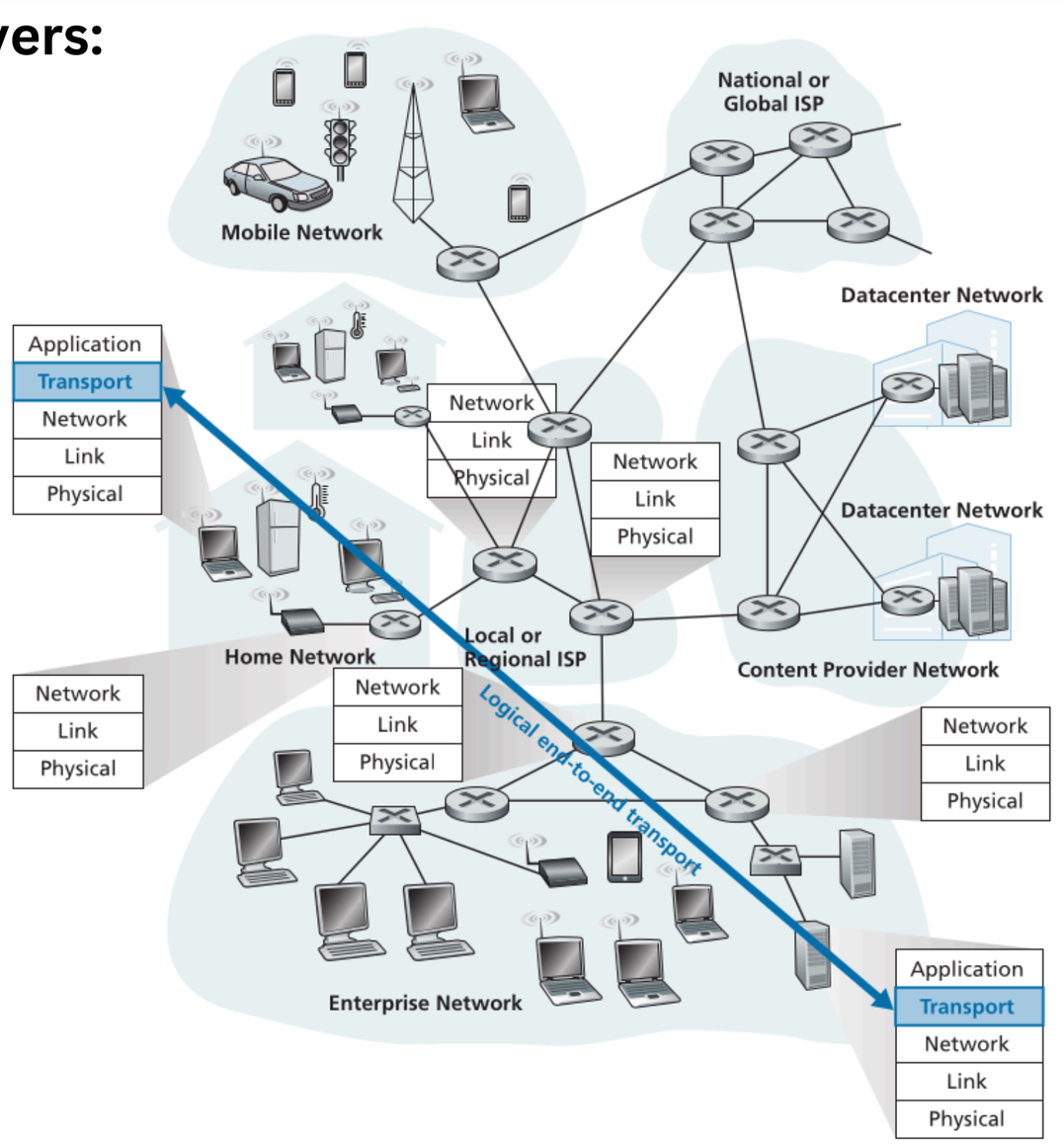
- Relationship Between Transport and Network Layers.
- Transport layer Multiplexing and Demultiplexing.
- Web Servers and TCP (Case study).
- UDP Segment Structure.
- UDP Checksum.
- Principles of Reliable Data Transfer, Building a Reliable Data Transfer Protocol.
- Pipelined Reliable Data Transfer Protocols:
  - Go-Back-N.
  - Selective Repeat.
- Connection-Oriented Transport: TCP.
- TCP Segment Structure.
- Round-Trip Time Estimation and Timeout.
- Reliable Data Transfer in TCP.
- Flow Control
- Principles of Congestion Control.
- Summary

Transport layer:

- Reside between the application and network layers.
- It has the critical role of providing comunication services directly to the application processes running on different hosts.
- Extends the network layer’s delivery service between two end systems to a delivery service between two application-layer processes running on the end systems.
- Make it possible for two entities to communicate reliably over a medium that may lose and corrupt data(Unreliable medium).
- Controls the transmission rate in order to avoid, or recover from, congestion within the network.
- A transport-layer protocol provides for logical communication between application processes running on different hosts(from an application’s perspective, it is as if the hosts running the processes were directly connected).
- Transport-layer protocols are implemented in the end systems but not in network routers.
- Breaks the application messages into smaller chunks and adds a transport-layer header to each chunk to create the transport-layer **segment**.
- The transport layer then passes the segment to the network layer at the sending end system, where the segment is encapsulated within a network-layer packet (a datagram) and sent to the destination.
- The Internet has two protocols TCP and UDP. Each of these protocols provides a different set of transport-layer services to the invoking application.

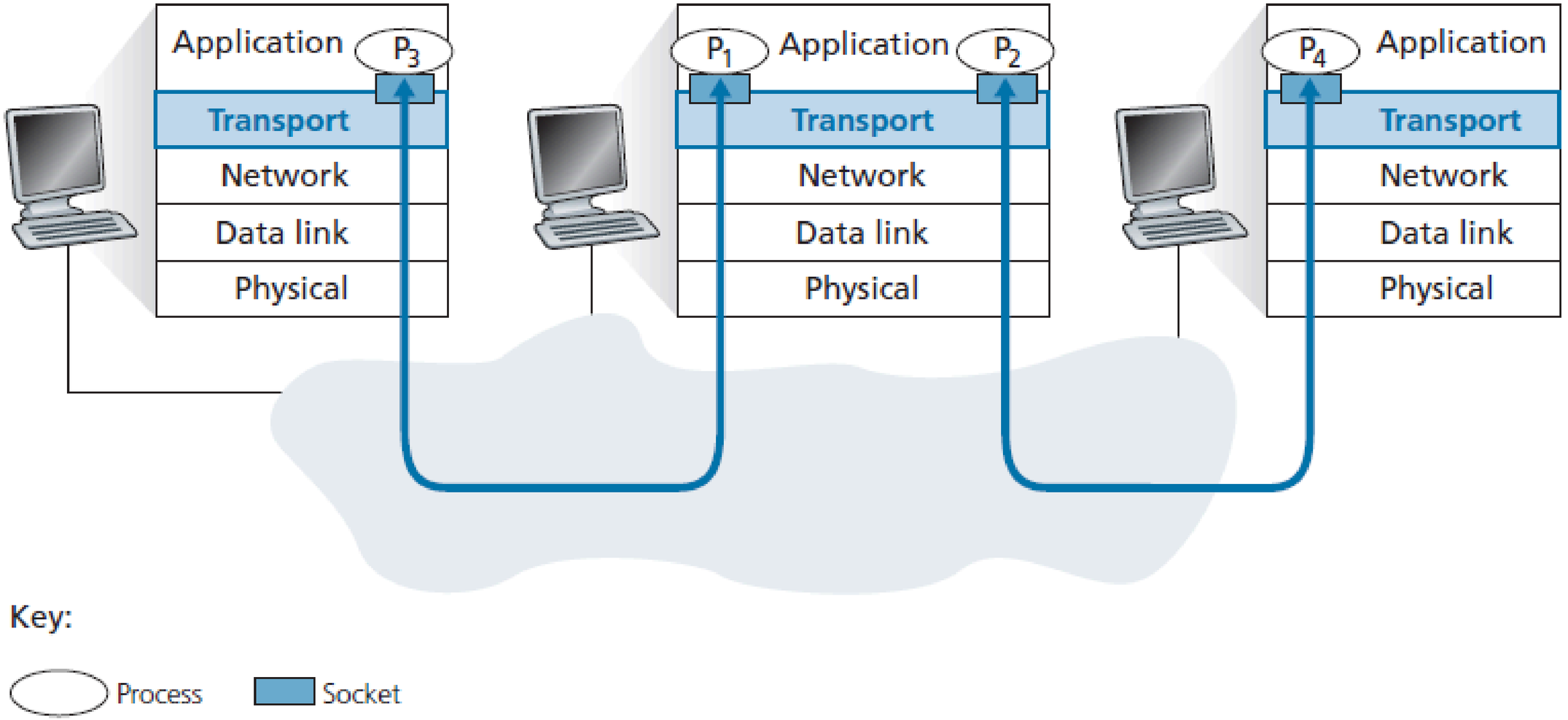
Relationship Between Transport and Network Layers:

- a transport-layer protocol provides logical communication between processes running on different hosts, a network-layer protocol provides logical-communication between hosts. This distinction is subtle but important.
- Transport-layer protocols live in the end systems. Within an end system, a transport protocol moves messages from application processes to the network edge (that is, the network layer) and vice versa, but it doesn’t have any say about how the messages are moved within the network core.
- The transport-layer protocol cannot provide delay or bandwidth guarantees for application messages sent between processes.
- A transport protocol can offer reliable data transfer service to an application even when the underlying network protocol is unreliable, that is, even when the network protocol loses, garbles, or duplicates packets.

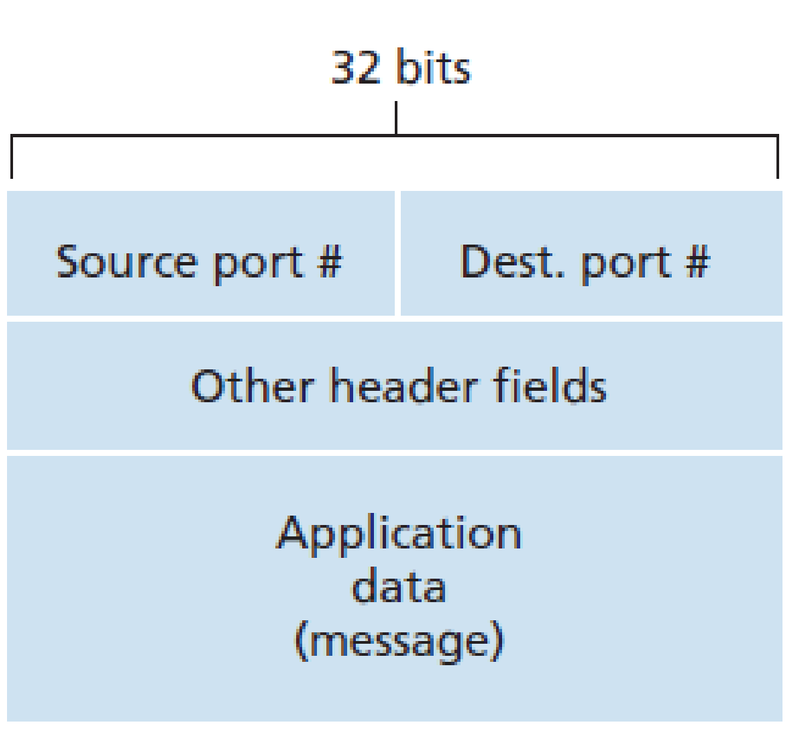


Multiplexing and Demultiplexing

- A process (as part of a network application) can have one or more sockets, doors through which data passes from the network to the process and through which data passes from the process to the network. Thus, the transport layer in the receiving host does not actually deliver data directly to a process, but instead to an intermediary socket. Because at any given time there can be more than one socket in the receiving host, each socket has a unique identifier.
- Each transport-layer segment has a set of fields in the segment. At the receiving end, the transport layer examines these fields to identify the receiving socket and then directs the segment to that socket.
- The job of delivering the data in a transport-layer segment to the correct socket is called demultiplexing.
- The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information to create segments, and passing the segments to the network layer is called multiplexing.


Application
P3
Transport
Network
Data link
Physical
P1
Application
P2
Transport
Network
Data link
Physical
P4
Application
Transport
Network
Data link
Physical
Key:
Process
Socket

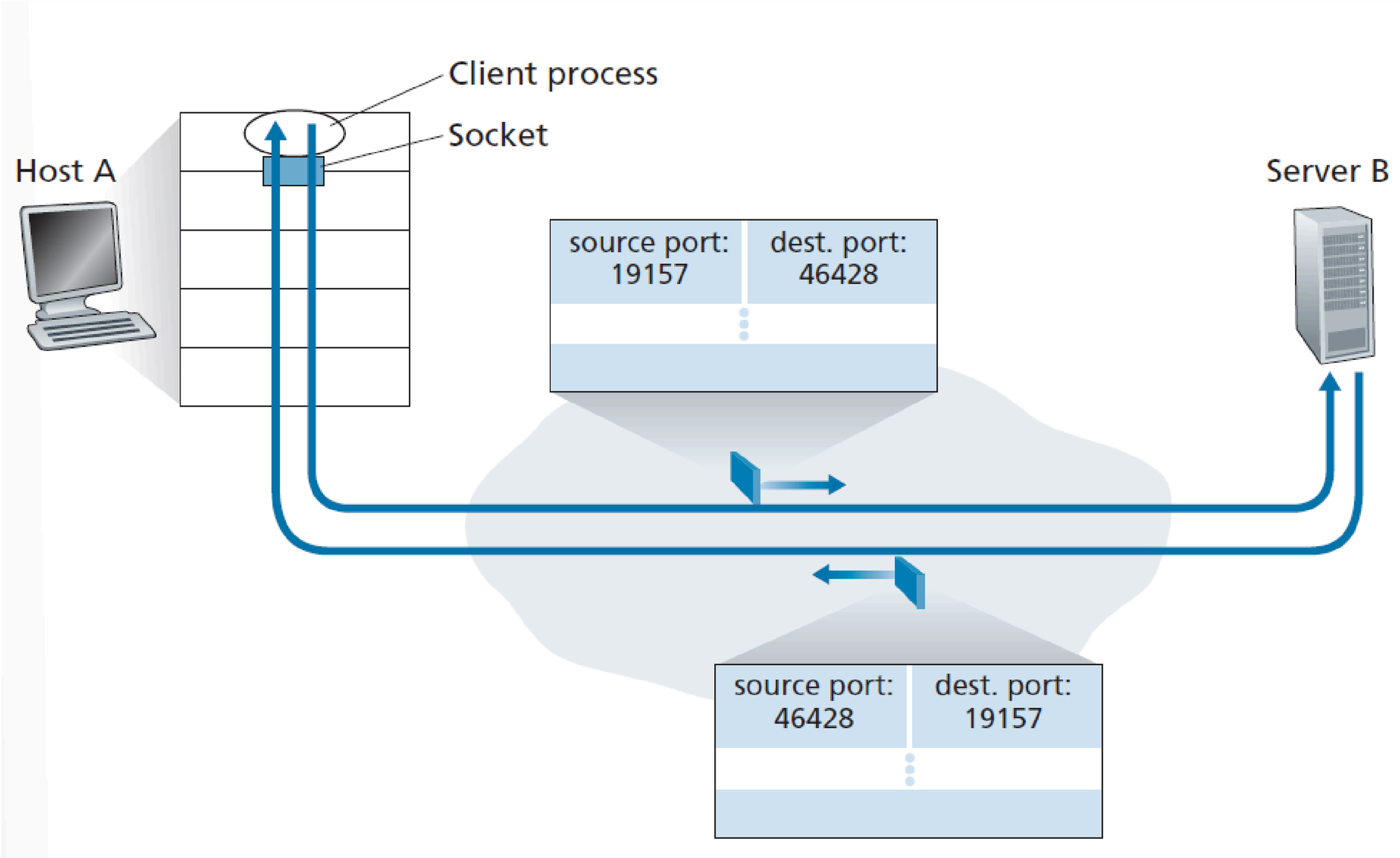
- Transport-layer multiplexing requires
  - That sockets have unique identifiers.
  - Each segment have special fields that indicate the socket to which the segment is to be delivered. These special fields, illustrated in the figure are the source port number field and the destination port number field.
- Each port number is a 16-bit number, ranging from 0 to 65535.
- The port numbers ranging from 0 to 1023 are called well-known port numbers and are restricted, which means that they are reserved for use by well-known application protocols such as HTTP (which uses port number 80) and FTP (which uses port number 21).
- When we develop a new application, we must assign the application a port number.
- Each socket in the host could be assigned a port number, and when a segment arrives at the host, the transport layer examines the destination port number in the segment and directs the segment to the corresponding socket.
- The segment's data then passes through the socket into the attached process.



Connectionless Multiplexing and Demultiplexing

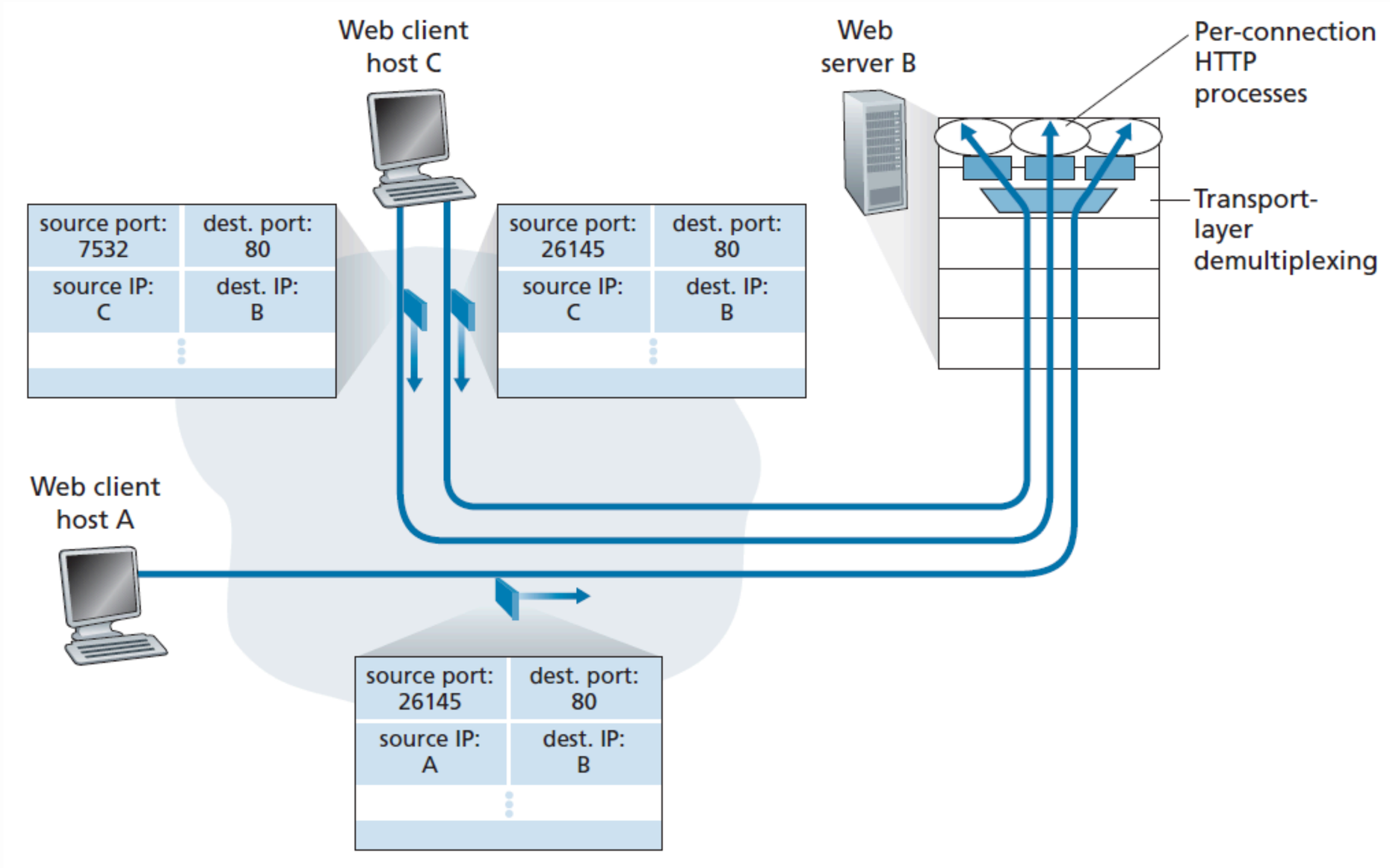
Suppose a process in Host A, with UDP port 19157, wants to send a chunk of application data to a process with UDP port 46428 in Host B:

- The transport layer in Host A creates a transport-layer segment that includes the application data, the source port number (19157), the destination port number (46428), and two other values (later).
- The transport layer then passes the resulting segment to the network layer.
- The network layer encapsulates the segment in an IP datagram and makes a best-effort attempt to deliver the segment to the receiving host.
- If the segment arrives at the receiving Host B, the transport layer at the receiving host examines the destination port number in the segment (46428) and delivers the segment to its socket identified by port 46428.
- As UDP segments arrive from the network, Host B directs (demultiplexes) each segment to the appropriate socket by examining the segment's destination port number.
- UDP socket is fully identified by a two-tuple consisting of a destination IP address and a destination port number. As a consequence, if two UDP segments have different source IP addresses and/or source port numbers, but have the same destination IP address and destination port number, then the two segments will be directed to the same destination process via the same destination socket.
- UDP socket is fully identified by a two-tuple consisting of a destination IP address and a destination port number. As a consequence.


Client process
Socket
Host A
Server B
source port: 19157
dest. port: 46428
source port: 46428
dest. port: 19157

Connection-Oriented Multiplexing and Demultiplexing

- TCP socket is identified by a four-tuple: (source IP address, source port number, destination IP address, destination port number). Thus, when a TCP segment arrives from the network to a host, the host uses all four values to direct (demultiplex) the segment to the appropriate socket.
- Two arriving TCP segments with different source IP addresses or source port numbers will (with the exception of a TCP segment carrying the original connection-establishment request) be directed to two different sockets.
- The TCP server application has a “welcoming socket,” that waits for connection establishment requests from TCP clients.
- The TCP client creates a socket and sends a connection establishment request segment.
- A connection-establishment request is nothing more than a TCP segment with destination port number and a special connection-establishment bit set in the TCP header. The segment also includes a source port number that was chosen by the client.
- When the host operating system of the computer running the server process receives the incoming connection-request segment, it locates the server process that is waiting to accept a connection. The server process then creates a new socket.
- The newly created connection socket is identified by thes four mentioned values; all subsequently arriving segments whose source port, source IP address, destination port, and destination IP address match these four values will be demultiplexed to this socket.


Web client
host C
Web
server B
Per-connection
HTTP
processes
Transport-
layer
demultiplexing
source port:
7532
dest. port:
80
source IP:
C
dest. IP:
B
source port:
26145
dest. port:
80
source IP:
C
dest. IP:
B
Web client
host A
source port:
26145
dest. port:
80
source IP:
A
dest. IP:
B

Web Servers and TCP (Case study)

- Consider a host running a Web server, such as an Apache Web server, on port 80. When clients (for example, browsers) send segments to the server, all segments will have destination port 80.
- In particular, both the initial connection-establishment segments and the segments carrying HTTP request messages will have destination port 80.
- As described, the server distinguishes the segments from the different clients using source IP addresses and source port numbers.
- The Web server spawns a new process for each connection. Each of these processes has its own connection socket through which HTTP requests arrive and HTTP responses are sent.
- If the client and server are using persistent HTTP, then throughout the duration of the persistent connection the client and server exchange HTTP messages via the same server socket.
- If the client and server use non-persistent HTTP, then a new TCP connection is created and closed for every request/response, and hence a new socket is created and later closed for every request/response. This frequent creating and closing of sockets can severely impact the performance of a busy Web server.

Connectionless Transport: UDP

- Aside from the multiplexing/demultiplexing function and some light error checking, UDP adds nothing to IP.
- In fact, if the application developer chooses UDP instead of TCP,then the application is almost directly talking with IP.
- UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer.
- The network layer encapsulates the transport-layer segment into an IP datagram and then makes a best-effort attempt to deliver the segment to the receiving host.
- If the segment arrives at the receiving host, UDP uses the destination port number to deliver the segment's data to the correct application process.
- With UDP there is no handshaking between sending and receiving transport-layer entities before sending a segment. For this reason, UDP is said to be connectionless.

why an application developer would ever choose to build an application over UDP rather than over TCP?

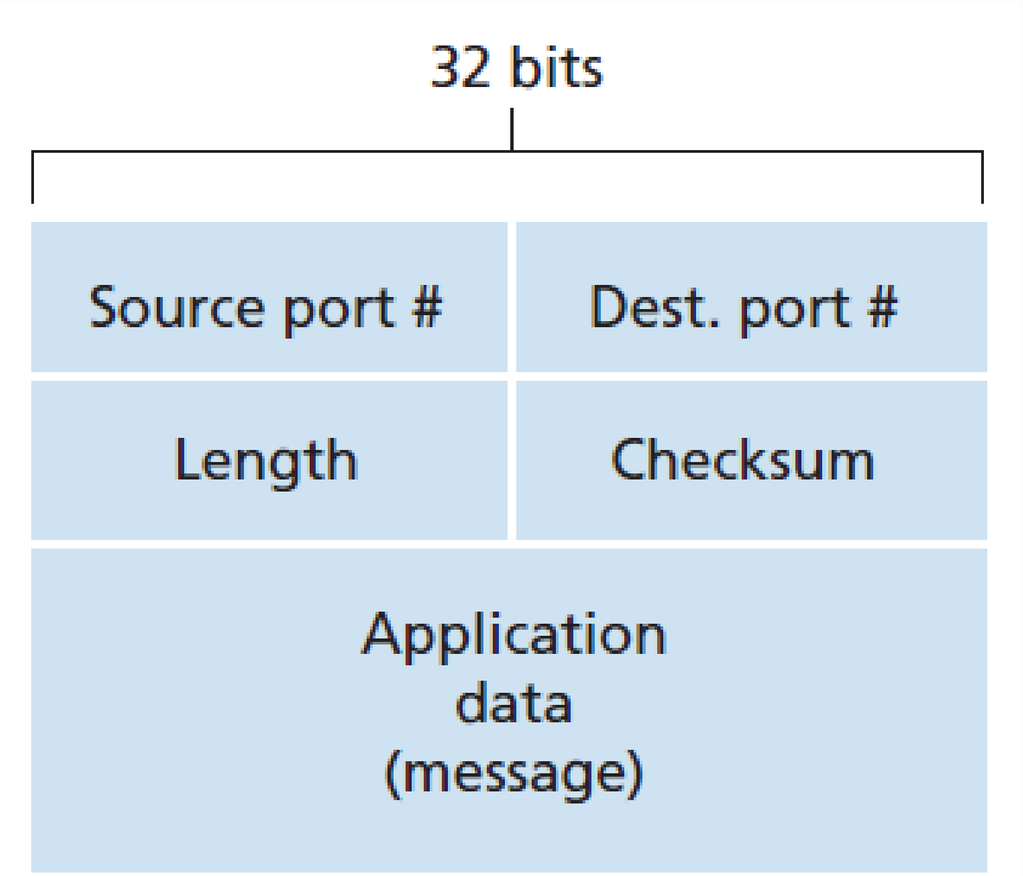
- Finer application-level control over what data is sent, and when:
  - UDP package the data inside a UDP segment and immediately pass the segment to the network layer. TCP, on the other hand, has a congestion-control mechanism that throttles the transport-layer TCP sender when one or more links between the source and destination hosts become excessively congested.
  - TCP will also continue to resend a segment until the receipt of the segment has been acknowledged by the destination, regardless of how long reliable delivery takes. Since real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss, TCP's service model is not particularly well matched to these applications' needs. These applications can use UDP.
- No connection establishment:
  - TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection.

- No connection state: TCP maintains connection state in the end systems. This includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters. UDP, on the other hand, doesnot maintain connection state and does not track any of these parameters.
- Small packet header overhead. The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead.

it is possible for an application to have reliable data transfer when using UDP. This can be done if reliability is built into the application itself (for example, by adding acknowledgment and retransmission mechanisms). For example early versions of HTTP ran over TCP but that more recent versions of HTTP like HTTP3 run over UDP, providing their own error control and congestion control (among other services) at the application layer.

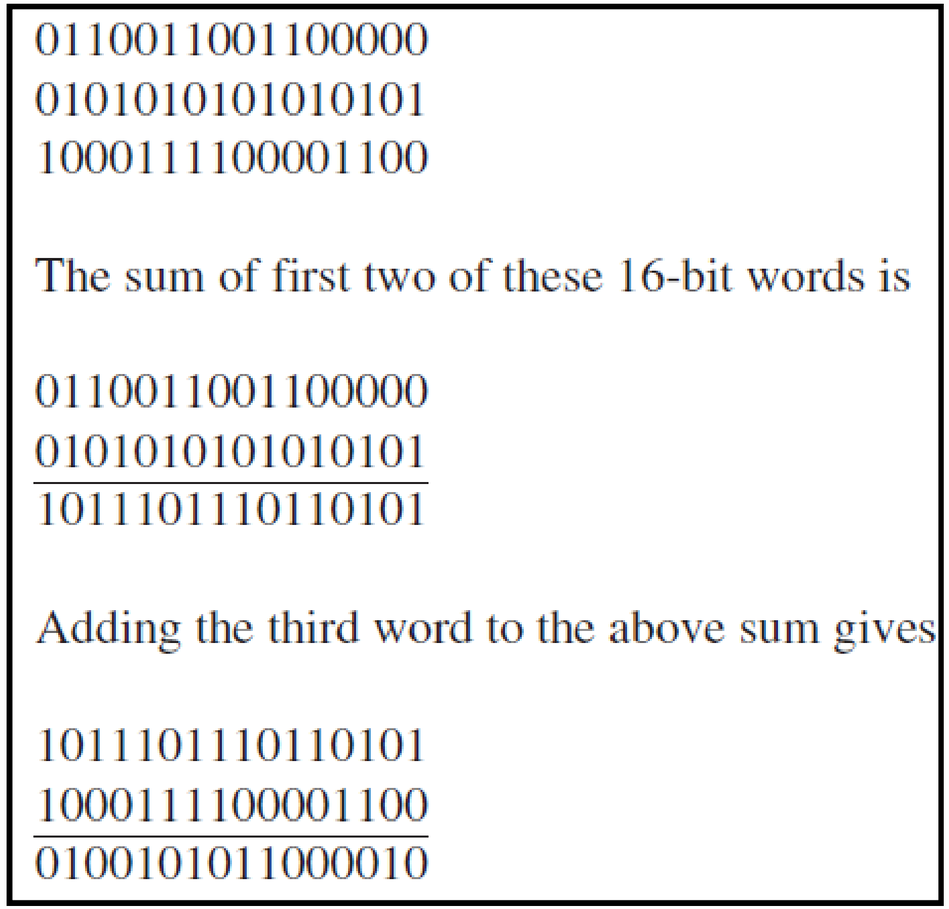
UDP Segment Structure

- The application data occupies the data field of the UDP segment. For example, for DNS, the data field contains either a query message or a response message. For a streaming audio application, audio samples fill the data field.
- The UDP header has only four fields, each consisting of two bytes. As discussed before, the port numbers allow the destination host to pass the application data to the correct process running on the destination end system (demultiplexing).
- The length field specifies the number of bytes in the UDP segment (header plus data). An explicit length value is needed since the size of the data field may differ from one UDP segment to the next.
- The checksum is used by the receiving host to check whether errors have been introduced into the segment.



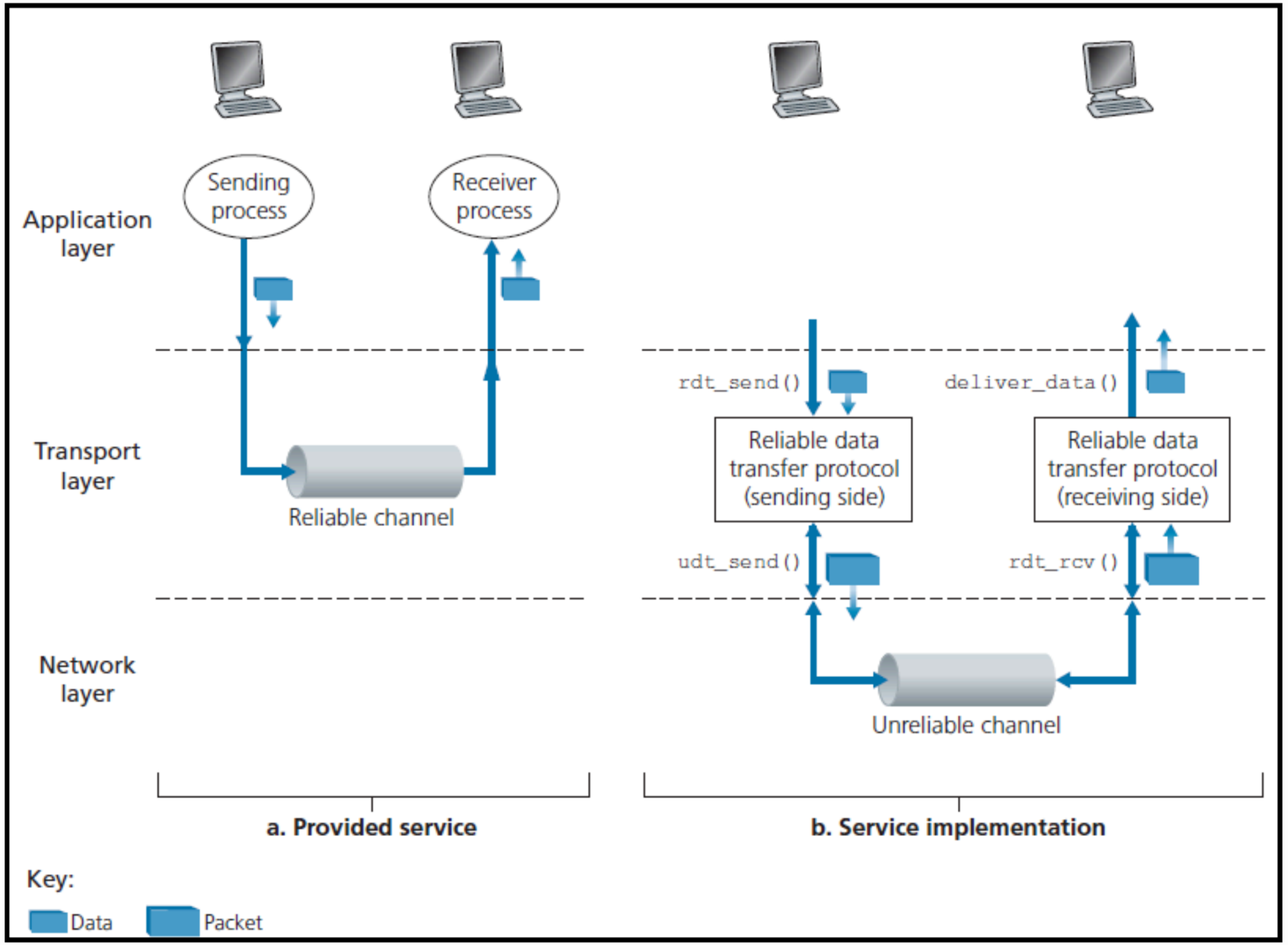
UDP Checksum

- The UDP checksum provides for error detection. That is, the checksum is used to determine whether bits within the UDP segment have been altered (for example, by noise in the links or while stored in a router) as it moved from source to destination.
- UDP at the sender side performs the 1s complement of the sum of all the 16-bit words in the segment, with any overflow encountered during the sum being wrapped around. This result is put in the checksum field of the UDP segment.
- last addition had overflow, which was wrapped around. The 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s. Thus, the 1s complement of the sum 0100101011000010 is 1011010100111101, which becomes the checksum.
- At the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the packet, then the sum at the receiver will be 1111111111111111. If one of the bits is a 0, then we know that errors have been introduced into the packet.



Principles of Reliable Data Transfer

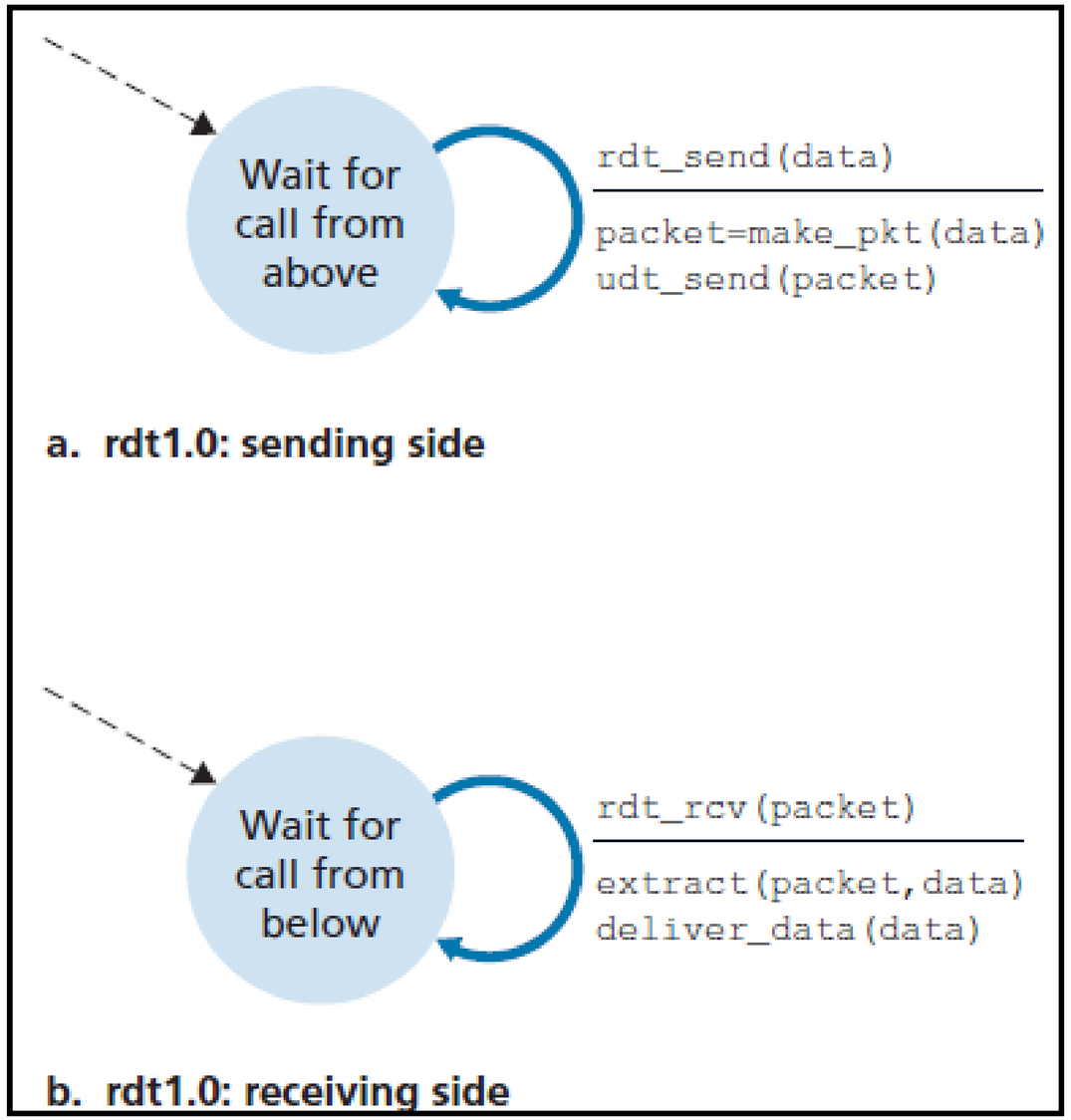
- The service abstraction provided to the upper-layer entities is that of a reliable channel through which data can be transferred.
- With a reliable channel, no transferred data bits are corrupted or lost, and all are delivered in the order in which they were sent. This is precisely the service model offered by TCP to the Internet applications that invoke it.
- The layer below the reliable data transfer protocol may be unreliable. For example, TCP is a reliable data transfer protocol that is implemented on top of an unreliable (IP) end-to-end network layer.



Building a Reliable Data Transfer Protocol

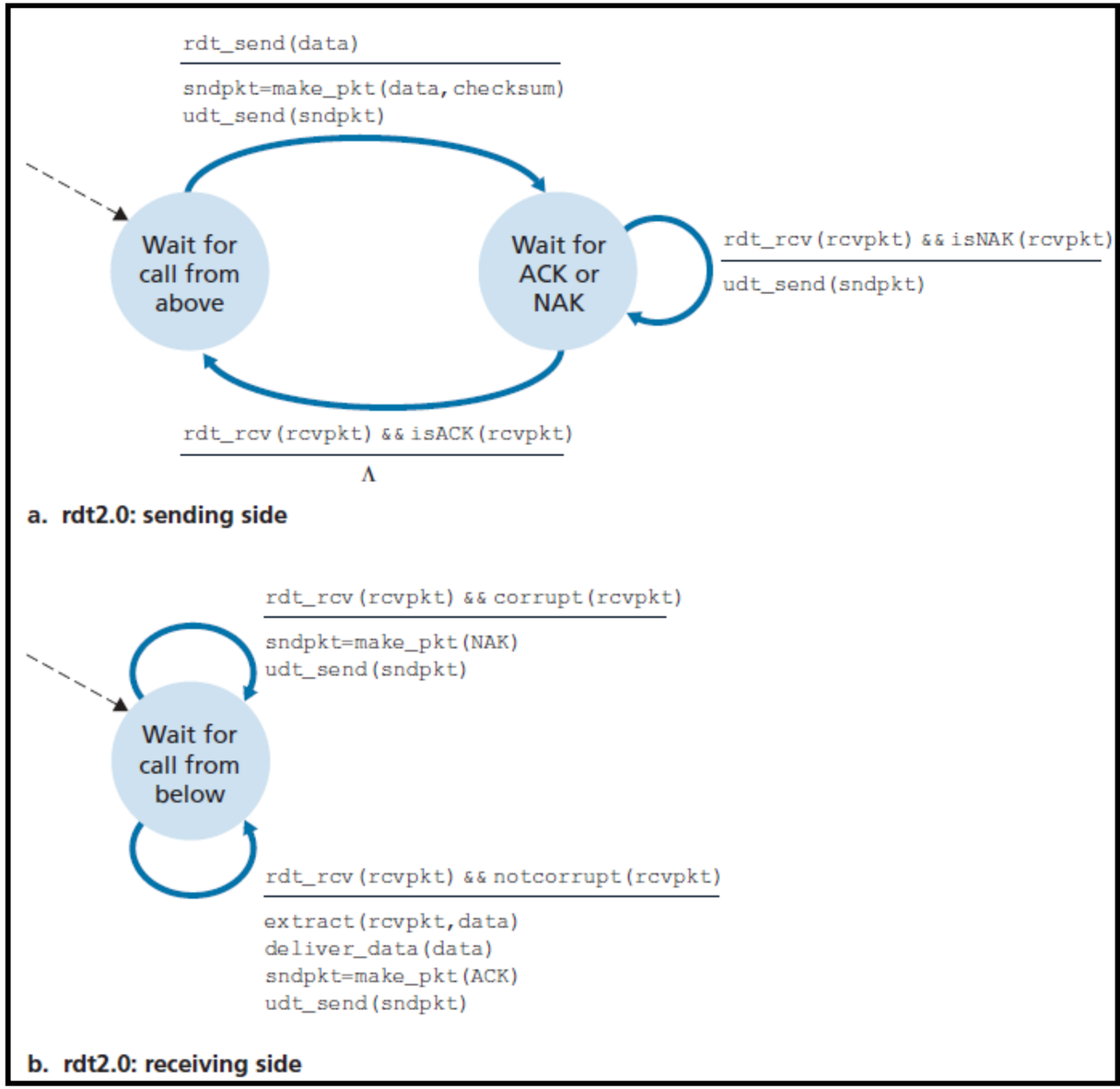
Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0

- We first consider the simplest case, in which the underlying channel is completely reliable. The protocol itself, which we'll call rdt1.0, is trivial.
- The sender and receiver finite-state machines (FSM) as shown in the figure have just one state each.
- The sending side of rdt simply accepts data from the upper layer via the rdt_send(data) event, creates a packet containing the data (via the action make_pkt(data)) and sends the packet into the channel. In practice, the rdt_send(data) event would result from a procedure call (for example, to rdt_send()) by the upper-layer application.
- On the receiving side, rdt receives a packet from the underlying channel via the rdt_rcv(packet) event, removes the data from the packet (via the action extract (packet, data)) and passes the data up to the upper layer (via the action deliver_data(data)). In practice, the rdt_rcv(packet) event would result from a procedure call (for example, to rdt_rcv()) from the lowerlayer protocol.



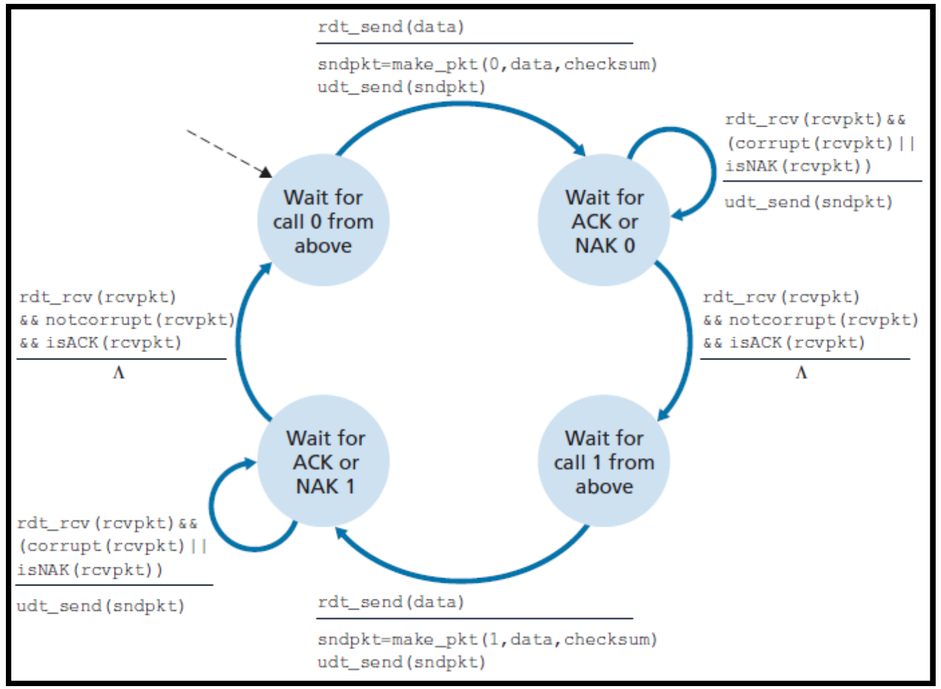
Reliable Data Transfer over a Channel with Bit Errors: rdt2.0

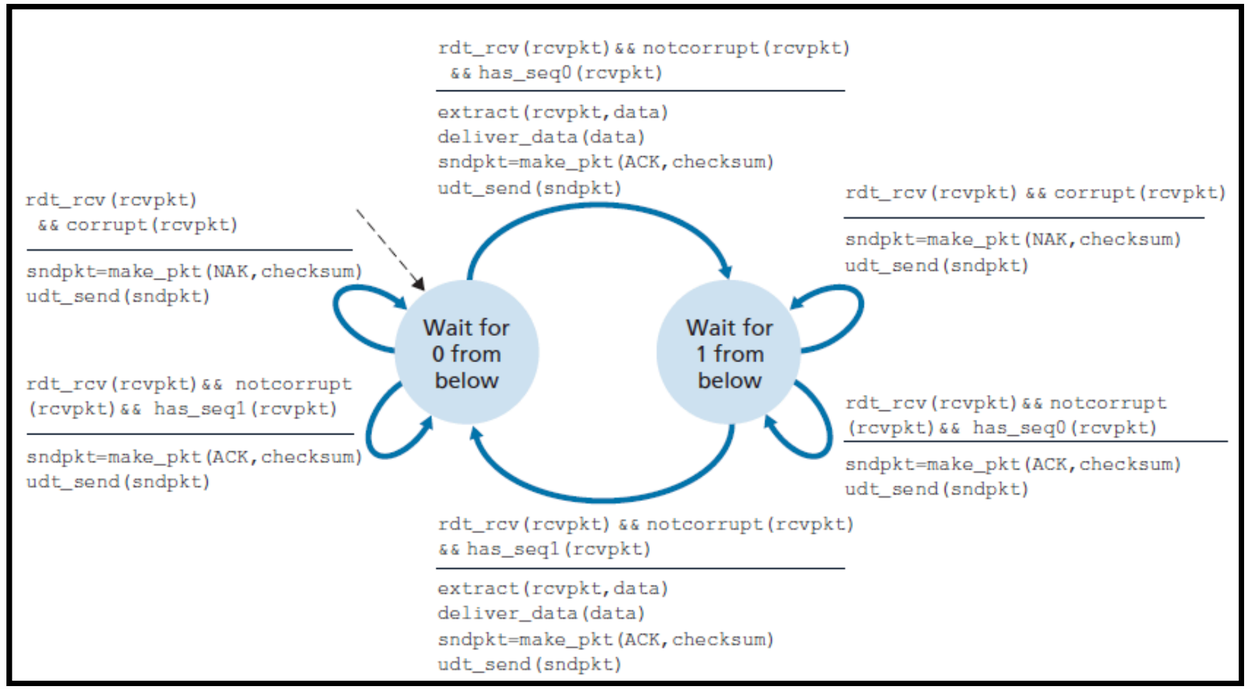
- The FSM representation of rdt2.0, a data transfer protocol employing error detection, positive acknowledgments, and negative acknowledgments.
- The send side of rdt2.0 has two states. In the leftmost state, the send-side protocol is waiting for data to be passed down from the upper layer. When the rdt_send(data) event occurs, the sender will create a packet (sndpkt) containing the data to be sent, and then send the packet via the udt_send(sndpkt) operation.
- In the rightmost state, the sender protocol is waiting for an ACK or a NAK packet from the receiver. If an ACK packet is received, the sender knows that the most recently transmitted packet has been received correctly and thus the protocol returns to the state of waiting for data from the upper layer. If a NAK is received, the protocol retransmits the lastpacket and waits for an ACK or NAK to be returned by the receiver.



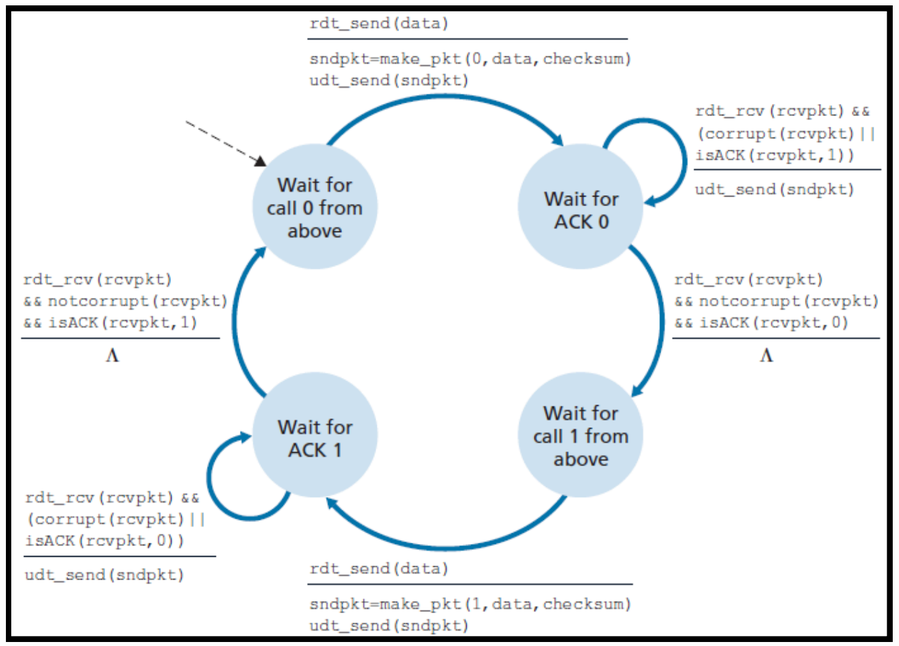
- It is important to note that when the sender is in the wait-for-ACK-or-NAK state, it cannot get more data from the upper layer; that is, the rdt_send() event can not occur; that will happen only after the sender receives an ACK and leaves this state. Thus, the sender will not send a new piece of data until it is sure that the receiver has correctly received the current packet. Because of this behavior, protocols such as rdt2.0 are known as stop-and-wait protocols.
- The receiver-side FSM for rdt2.0 still has a single state. On packet arrival, the receiver replies with either an ACK or a NAK, depending on whether or not the received packet is corrupted.
- Protocol rdt2.0 has a fatal flaw. In particular, we haven’t accounted for the possibility that the ACK or NAK packet could be corrupted!
- A possible approach is for the sender simply to resend the current data packet when it receives a garbled ACK or NAK packet. This approach, however, introduces duplicate packets into the sender-to receiver channel.
- The fundamental difficulty with duplicate packets is that the receiver doesn’t know whether the ACK or NAK it last sent was received correctly at the sender. Thus, it cannot know whether an arriving packet contains new data or is a retransmission!
- Simple solution to this new problem is to add a new field to the data packet and have the sender number its data packets by putting a sequence number into this field. The receiver then need only check this sequence number to determine whether or not the received packet is a retransmission.

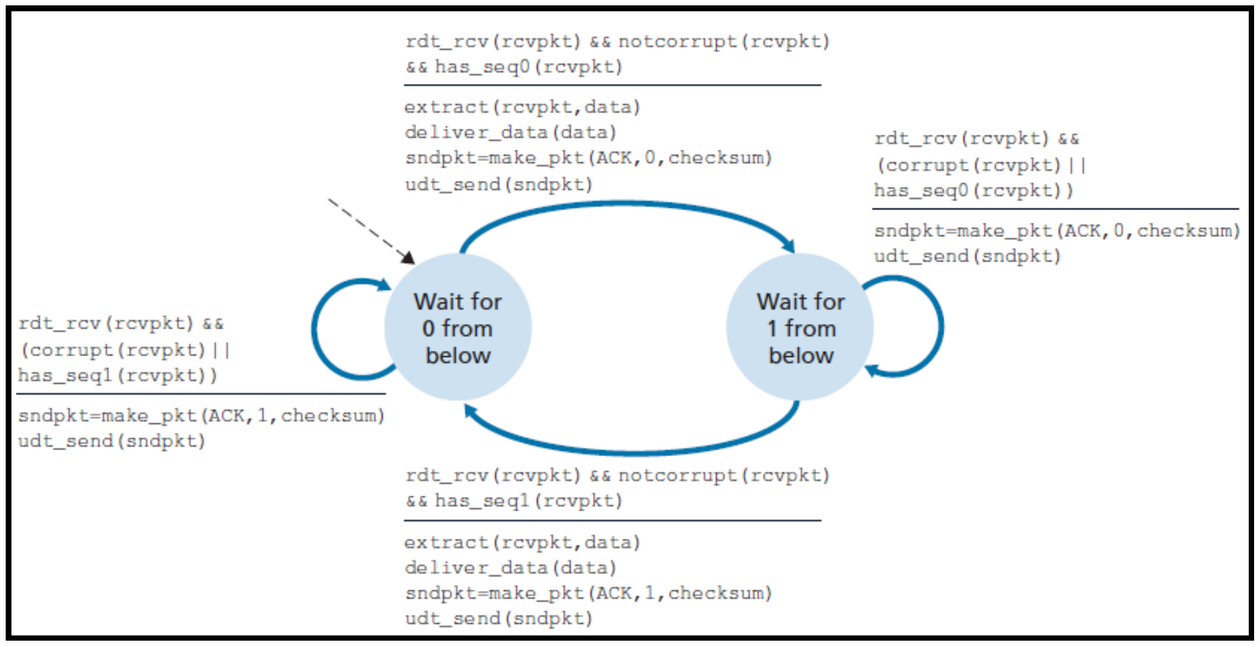
The figures show the FSM description for rdt2.1, our fixed version of rdt2.0. The rdt2.1 sender and receiver FSMs each now have twice as many states as before. This is because the protocol state must now reflect whether the packet currently being sent (by the sender) or expected (at the receiver) should have a sequence number of 0 or 1.





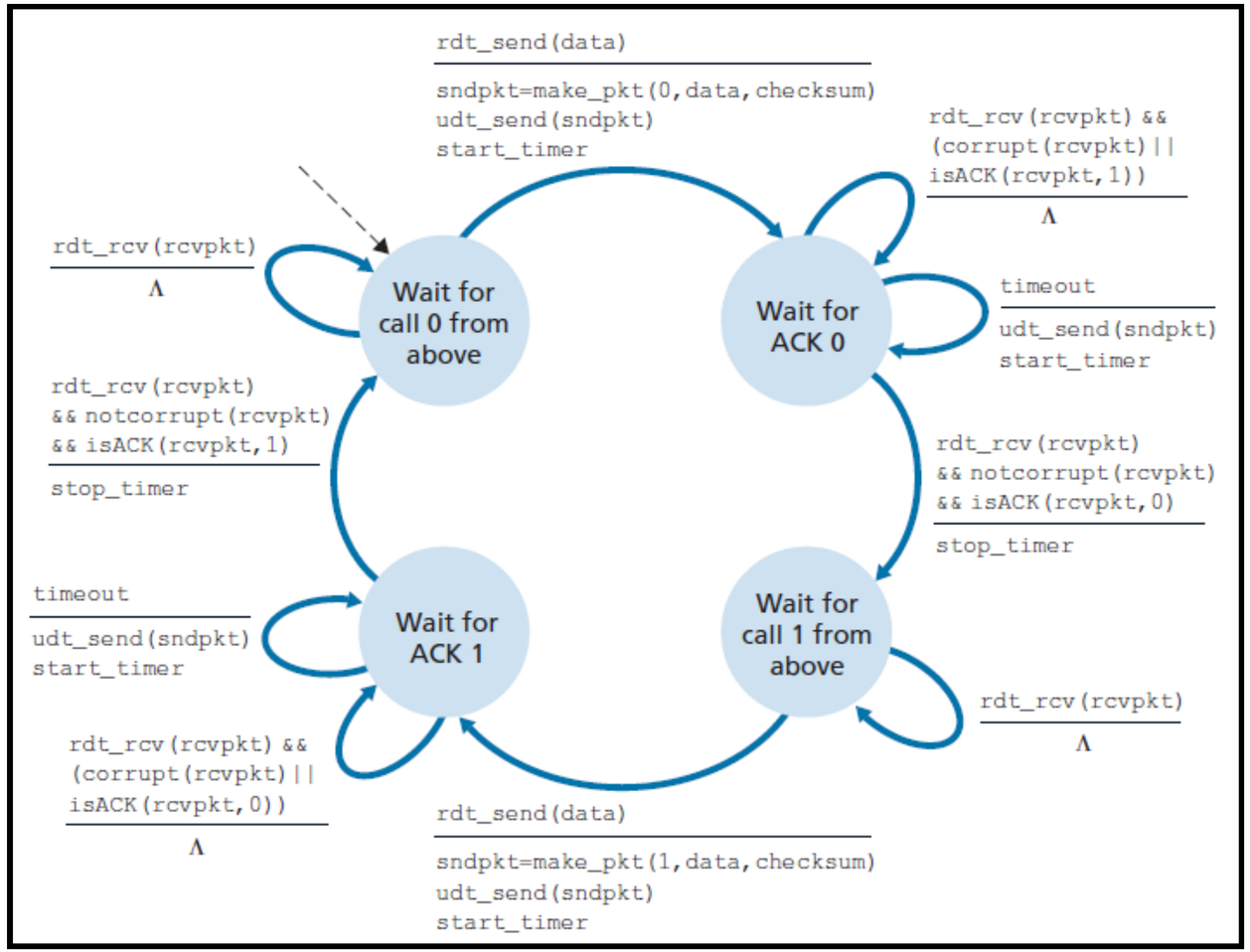
- We can accomplish the same effect as a NAK if, instead of sending a NAK, we send an ACK for the last correctly received packet. A sender that receives two ACKs for the same packet knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice. Our NAK-free reliable data transfer protocol for a channel with bit errors is rdt2.2.
- One subtle change between rtdt2.1 and rdt2.2 is that the receiver must now include the sequence number of the packet being acknowledged by an ACK message (this is done by including the ACK, 0 or ACK, 1 argument in make_pkt() in the receiver FSM), and the sender must now check the sequence number of the packet being acknowledged by a received ACK message (this is done by including the 0 or 1 argument in isACK() in the sender FSM).

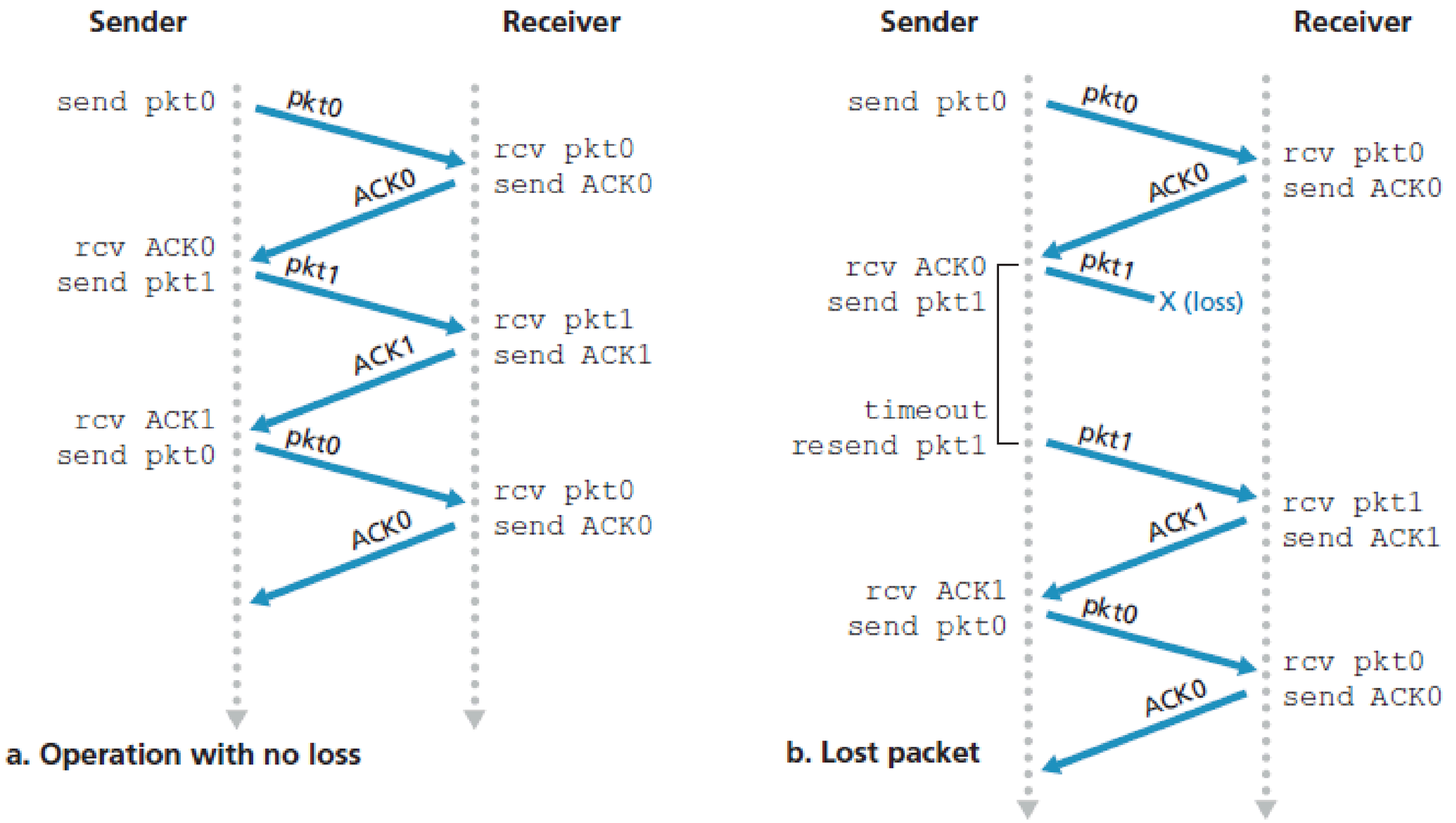


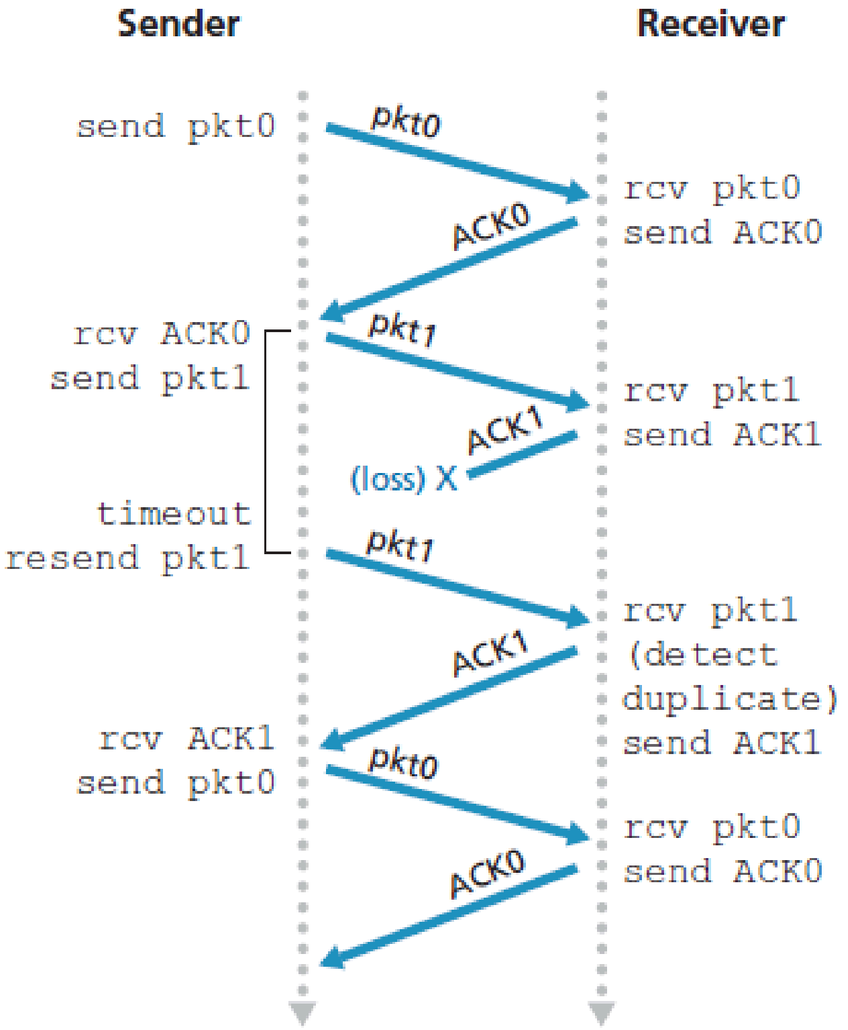


Reliable Data Transfer over a Lossy Channel with Bit Errors: rdt3.0

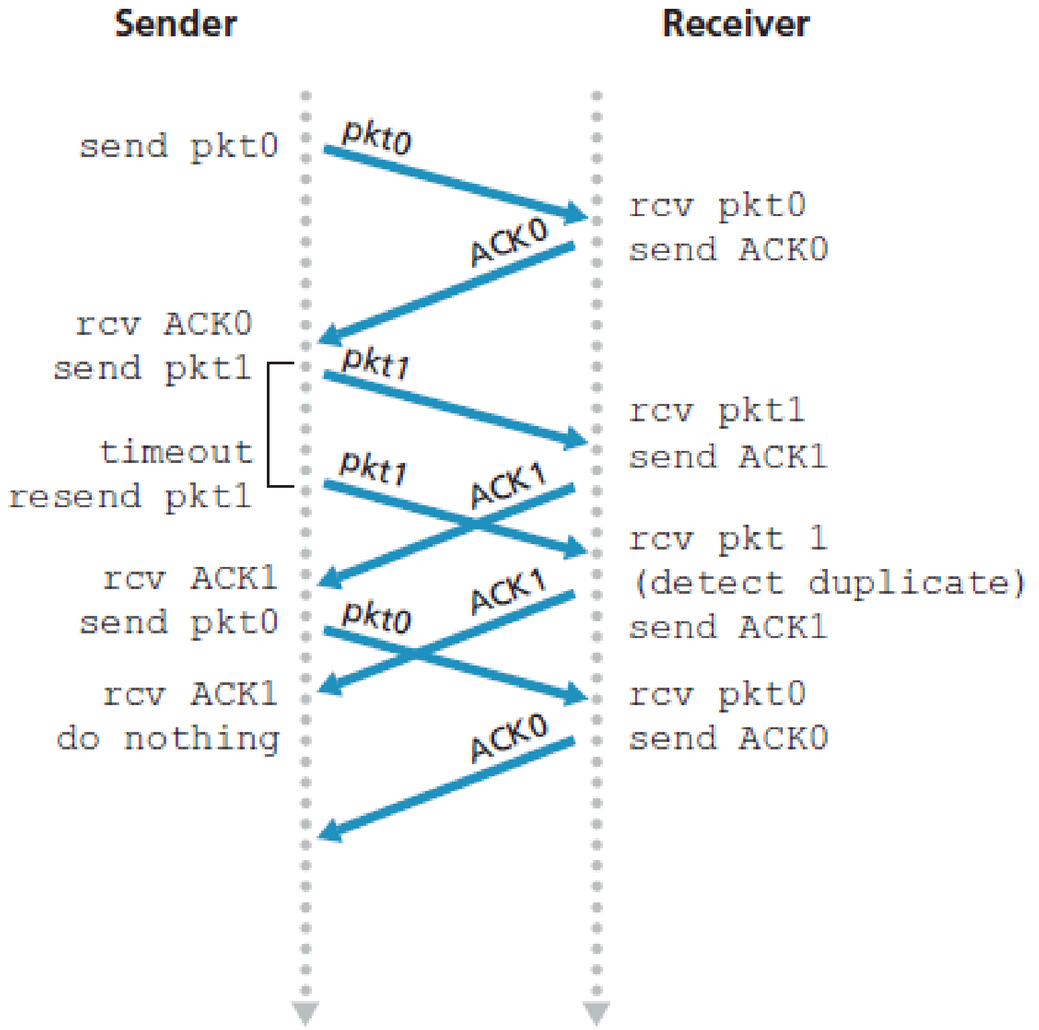
- in addition to corrupting bits, the underlying channel can lose packets as well. Two additional concerns must now be addressed by the protocol: how to detect packet loss and what to do when packet loss occurs.
- Suppose that the sender transmits a data packet and either that packet, or the receiver's ACK of that packet, gets lost. In either case, no reply is forthcoming at the sender from the receiver. If the sender is willing to wait long enough so that it is certain that a packet has been lost, it can simply retransmit the data packet.




Sender
Receiver
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
rcv ACK0
send pkt1
pkt1
rcv pkt1
send ACK1
ACK1
rcv ACK1
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
a. Operation with no loss
Sender
Receiver
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
rcv ACK0
send pkt1
pkt1
X (loss)
timeout
resend pkt1
pkt1
rcv pkt1
send ACK1
ACK1
rcv ACK1
send pkt0
pkt0
rcv pkt0
send ACK0
ACK0
b. Lost packet



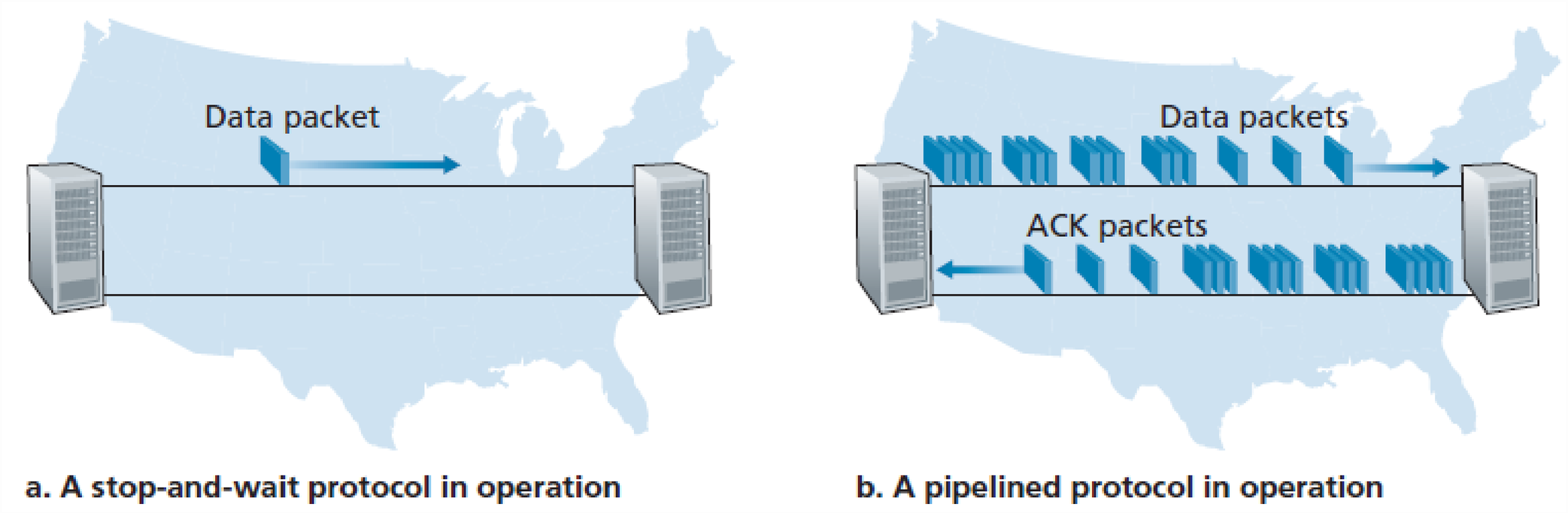
c. Lost ACK



d. Premature timeout

Pipelined Reliable Data Transfer Protocols

- Protocol rdt3.0 is a functionally correct protocol, but it is unlikely that anyone would be happy with its performance.
- At the heart of rdt3.0's performance problem is the fact that it is a stop-and-wait protocol.
- Suppose that transmission rate, R, of 1 Gbps. With a packet size, L, of 1,000 bytes, the time needed to transmit the packet into the 1 Gbps link is: d(Trans)=L/R= 8000/(10^9)=8 microseconds.



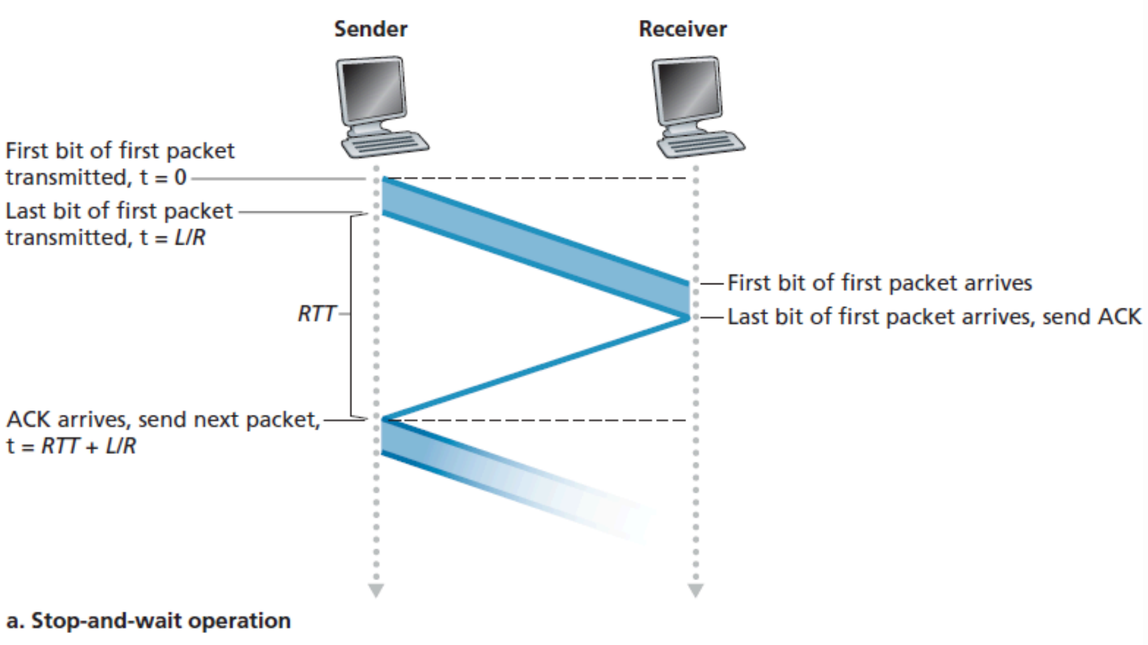
a. A stop-and-wait protocol in operation

b. A pipelined protocol in operation

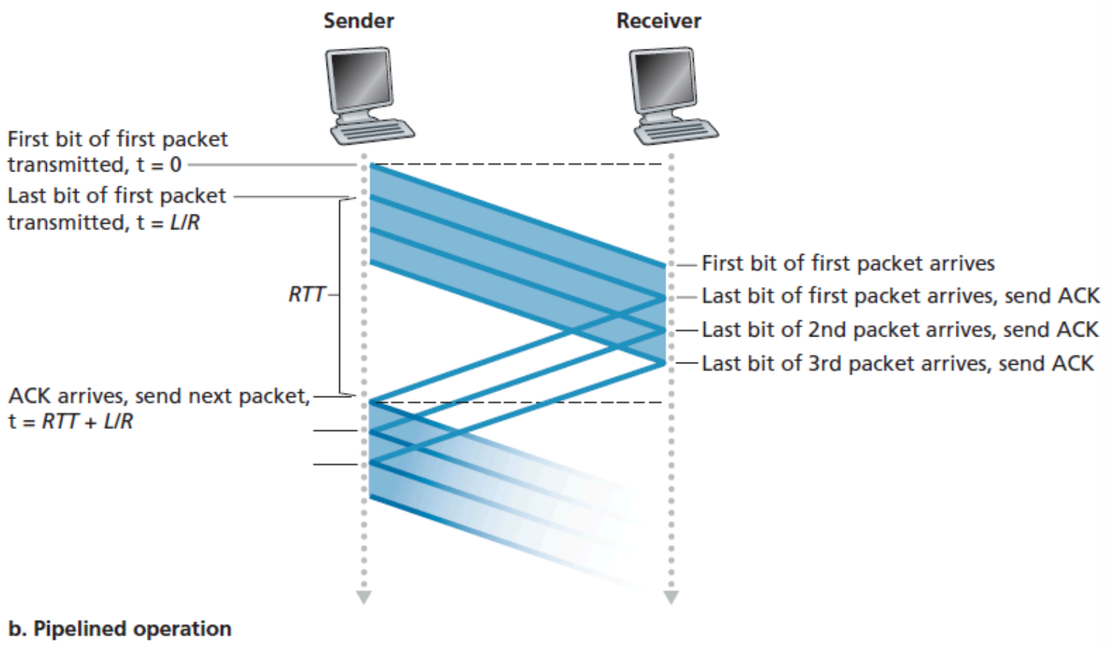
Suppose RTT= 30 mse

Thus, in 30.008 msec, the sender was sending for only 0.008 msec. If we define the utilization of the sender (or the channel) as the fraction of time the sender is busy sending bits into the channel, the analysis in the Figure shows that the stop-and-wait protocol has a rather dismal sender utilization, U(sender), of:

$$U(\text{sender}) = (L/R)/(RTT+L/R) = 0.008/30.008 = 0.00027$$



a. Stop-and-wait operation



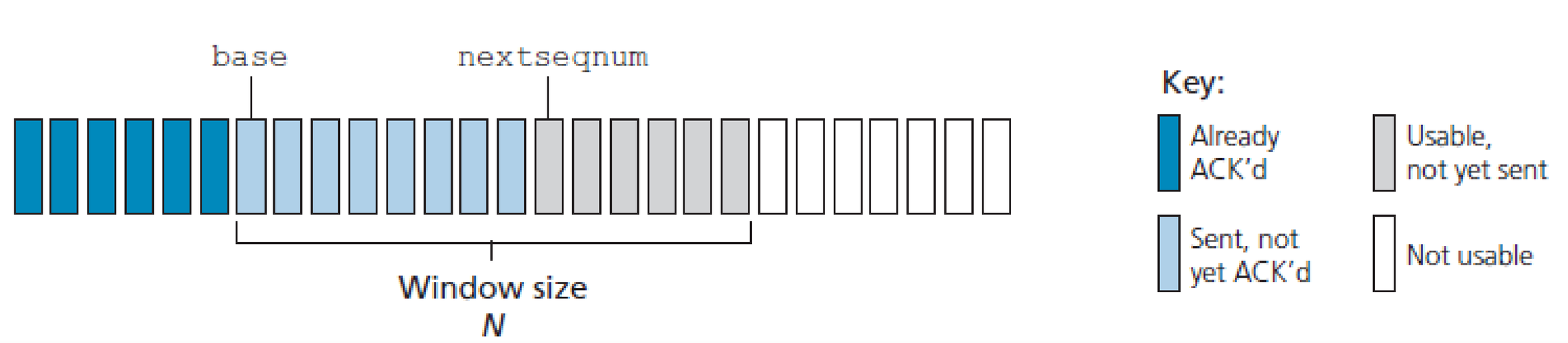
b. Pipelined operation

- The sender was busy only 2.7 hundredths of one percent of the time! Viewed another way, the sender was able to send only 1,000 bytes in 30.008 milliseconds, an effective throughput of only 267 kbps—even though a 1 Gbps link was available!
- This is a graphic example of how network protocols can limit the capabilities provided by the underlying network hardware.
- The solution to this particular performance problem is simple: Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, as illustrated in the last Figure.
- This Figure shows that if the sender is allowed to transmit three packets before having to wait for acknowledgments, the utilization of the sender is essentially tripled. Since the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as pipelining.
- Two basic approaches toward pipelined error recovery can be identified: Go-Back-N and selective repeat.

Go-Back-N (GBN)

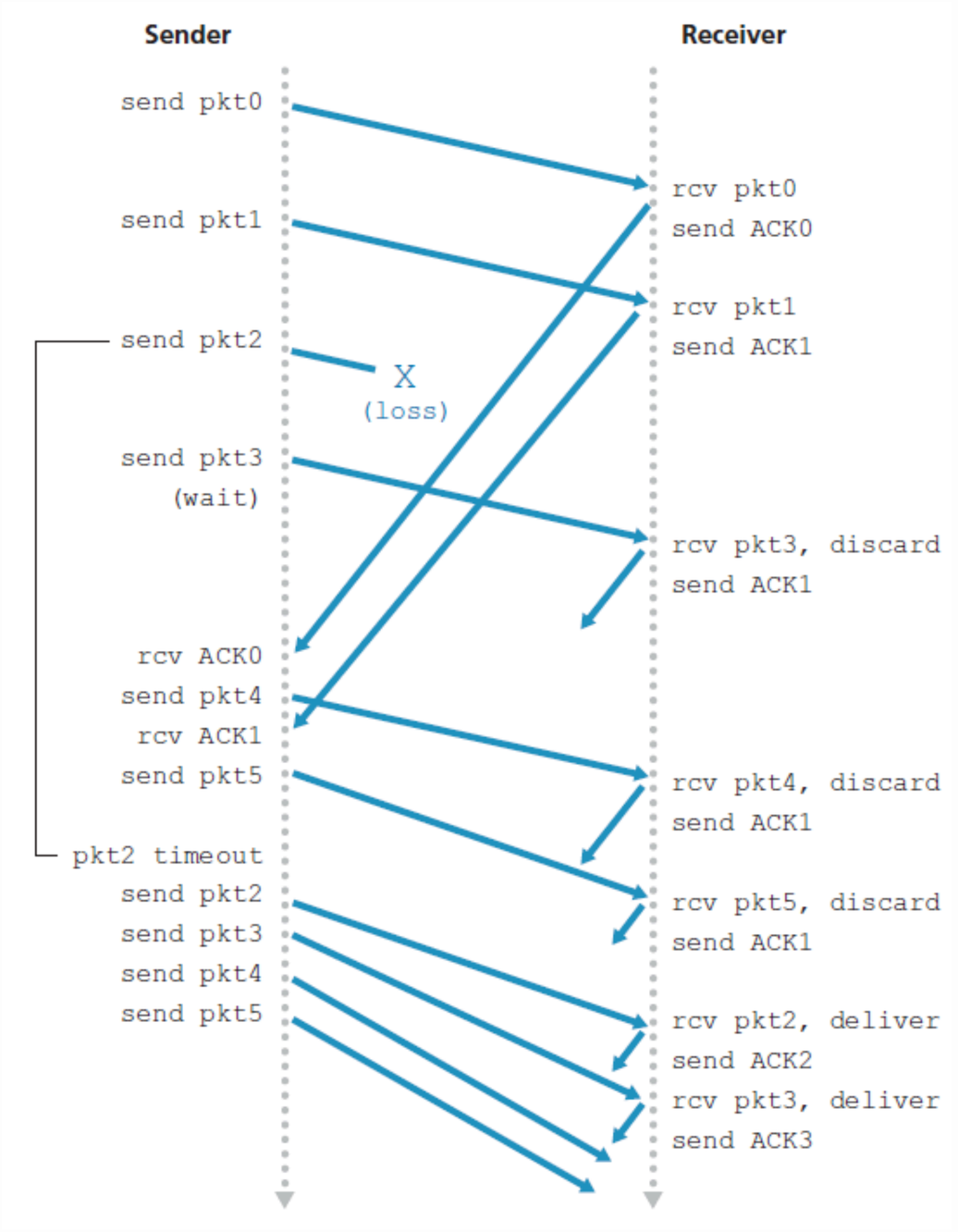
- In GBN-protocol, the sender is allowed to transmit multiple packets (when available) without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, N, of unacknowledged packets in the pipeline.
- The figure shows the sender's view of the range of sequence numbers in a GBN protocol. We define:

    - Base: as the sequence number of the oldest unacknowledged packet.
    - nextseqnum: as the smallest unused sequence number (that is, the sequence number of the next packet to be sent).



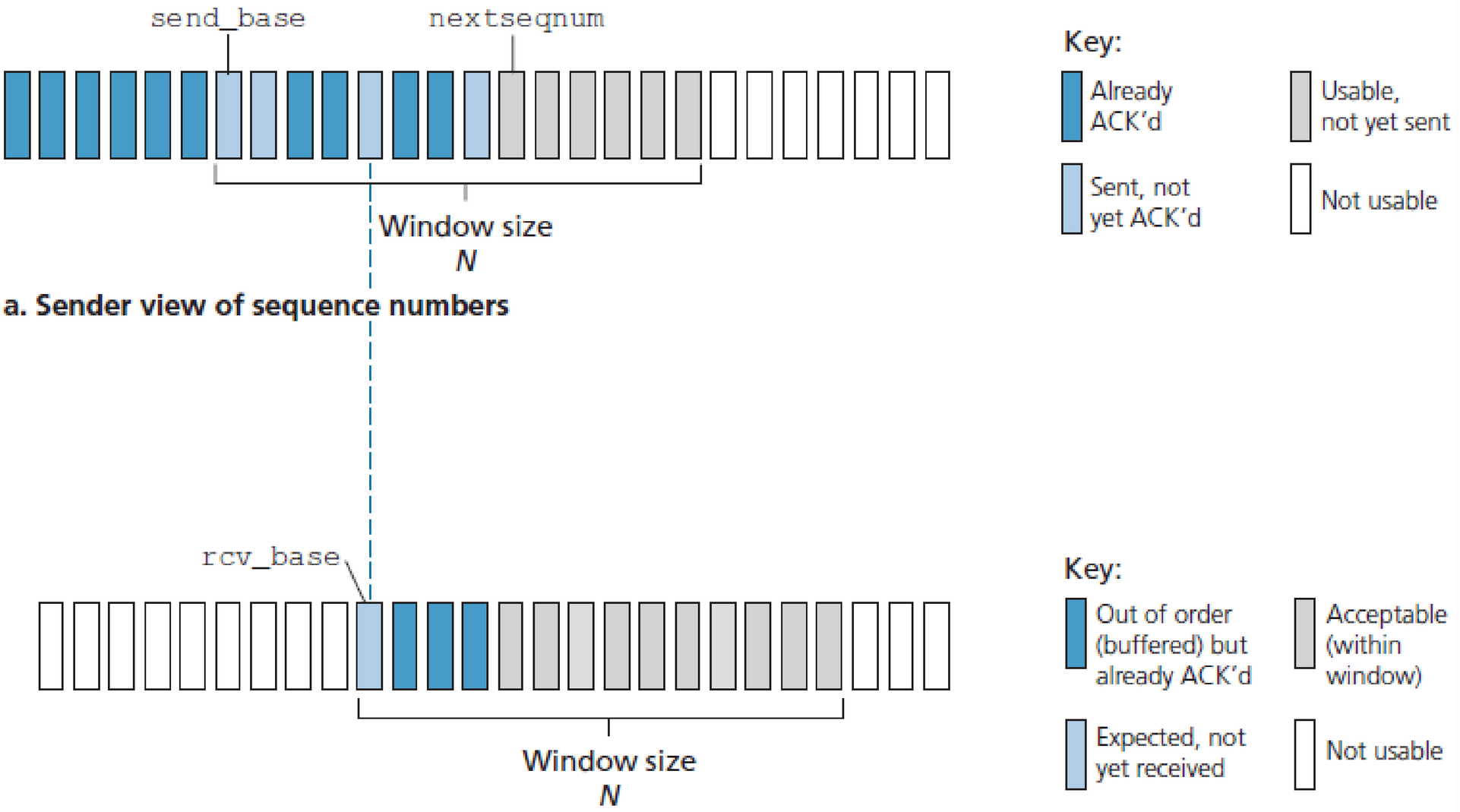
- Four intervals in the range of sequence numbers can be identified:
  - Sequence numbers in the interval [0,base-1]: correspond to packets that have already been transmitted and acknowledged.
  - Interval [base,nextseqnum-1]: corresponds to packets that have been sent but not yet acknowledged.
  - Interval [nextseqnum,base+N-1]: can be used for packets that can be sent immediately, should data arrive from the upper layer.
  - Sequence numbers greater than or equal to base+N cannot be used until an unacknowledged packet currently in the pipeline (specifically, the packet with sequence number base) has been acknowledged.
- The range of permissible sequence numbers for transmitted but not yet acknowledged packets can be viewed as a window of size N over the range of sequence numbers. As the protocol operates, this window slides forward over the sequence number space. For this reason, N is often referred to as the window size and the GBN protocol itself as a sliding-window protocol.
- When rdt_send() is called from above, the sender first checks to see if the window is full, that is, whether there are N outstanding, unacknowledged packets. If the window is not full, a packet is created and sent, and variables are appropriately updated. If the window is full, the sender simply returns the data back to the upper layer.
- An acknowledgment for a packet with sequence number n will be taken to be a cumulative acknowledgment, indicating that all packets with a sequence number up to and including n have been correctly received at the receiver.
- The protocol's name, "Go-Back-N," is derived from the sender's behavior in the presence of lost or overly delayed packets. As in the stop-and-wait protocol, a timer will again be used to recover from lost data or acknowledgment packets. If a timeout occurs, the sender resends all packets that have been previously sent but that have not yet been acknowledged.
- If an ACK is received but there are still additional transmitted but not yet acknowledged packets, the timer is restarted. If there are no outstanding, unacknowledged packets, the timer is stopped.
- The receiver's actions in GBN are simple. If a packet with sequence number n is received correctly and is in order (that is, the data last delivered to the upper layer came from a packet with sequence number n - 1), the receiver sends an ACK for packet n and delivers the data portion of the packet to the upper layer. In all other cases, the receiver discards the packet and resends an ACK for the most recently received in-order packet.

- The figure shows the operation of the GBN protocol for the case of a window size of four packets.
- Because of this window size limitation, the sender sends packets 0 through 3 but then must wait for one or more of these packets to be acknowledged before proceeding.
- As each successive ACK (for example, ACK0 and ACK1) is received, the window slides forward and the sender can transmit one new packet (pkt4 and pkt5, respectively).
- On the receiver side, packet 2 is lost and thus packets 3, 4, and 5 are found to be out of order and are discarded.



Selective Repeat (SR)

- There are scenarios in which GBN itself suffers from performance problems. In particular, when the window size and bandwidth-delay product are both large, many packets can be in the pipeline. A single packet error can thus cause GBN to retransmit a large number of packets, many unnecessarily.
- Selective-repeat protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error at the receiver. This individual, as-needed, retransmission will require that the receiver individually acknowledge correctly received packets. A window size of N will again be used to limit the number of outstanding, unacknowledged packets in the pipeline. However, unlike GBN, the sender will have already received ACKs for some of the packets in the window.
- The SR receiver will acknowledge a correctly received packet whether or not it is in order. Out-of-order packets are buffered until any missing packets (that is, packets with lower sequence numbers) are received, at which point a batch of packets can be delivered in order to the upper layer.


send_base
nextseqnum
Window size
N
Key:
Already ACK'd
Usable, not yet sent
Sent, not yet ACK'd
Not usable
a. Sender view of sequence numbers
rcv_base
Window size
N
Key:
Out of order (buffered) but already ACK'd
Acceptable (within window)
Expected, not yet received
Not usable
b. Receiver view of sequence numbers

1. *Data received from above*. When data is received from above, the SR sender checks the next available sequence number for the packet. If the sequence number is within the sender's window, the data is packetized and sent; otherwise it is either buffered or returned to the upper layer for later transmission, as in GBN.
2. *Timeout*. Timers are again used to protect against lost packets. However, each packet must now have its own logical timer, since only a single packet will be transmitted on timeout. A single hardware timer can be used to mimic the operation of multiple logical timers
3. *ACK received*. If an ACK is received, the SR sender marks that packet as having been received, provided it is in the window. If the packet's sequence number is equal to `send_base`, the window base is moved forward to the unacknowledged packet with the smallest sequence number. If the window moves and there are untransmitted packets with sequence numbers that now fall within the window, these packets are transmitted.

SR sender events and actions

1. *Packet with sequence number in* `[rcv_base, rcv_base+N-1]` *is correctly received.* In this case, the received packet falls within the receiver's window and a selective ACK packet is returned to the sender. If the packet was not previously received, it is buffered. If this packet has a sequence number equal to the base of the receive window (`rcv_base` in Last figure) , then this packet, and any previously buffered and consecutively numbered (beginning with `rcv_base`) packets are delivered to the upper layer. The receive window is then moved forward by the number of packets delivered to the upper layer. As an example, consider Figure 3.26. When a packet with a sequence number of `rcv_base=2` is received, it and packets 3, 4, and 5 can be delivered to the upper layer.
2. *Packet with sequence number in* `[rcv_base-N, rcv_base-1]` *is correctly received.* In this case, an ACK must be generated, even though this is a packet that the receiver has previously acknowledged.
3. *Otherwise.* Ignore the packet.

SR receiver events and actions

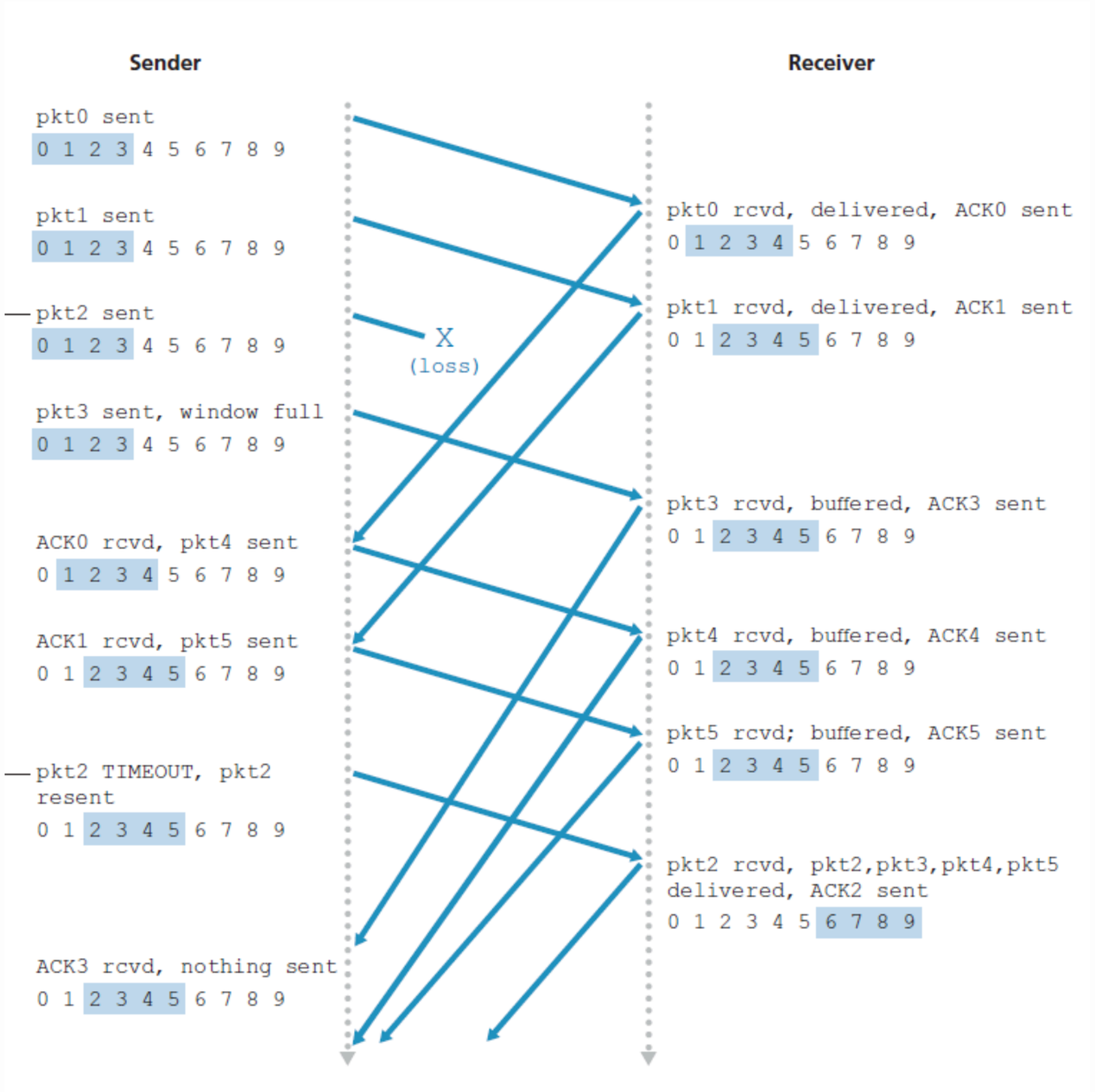
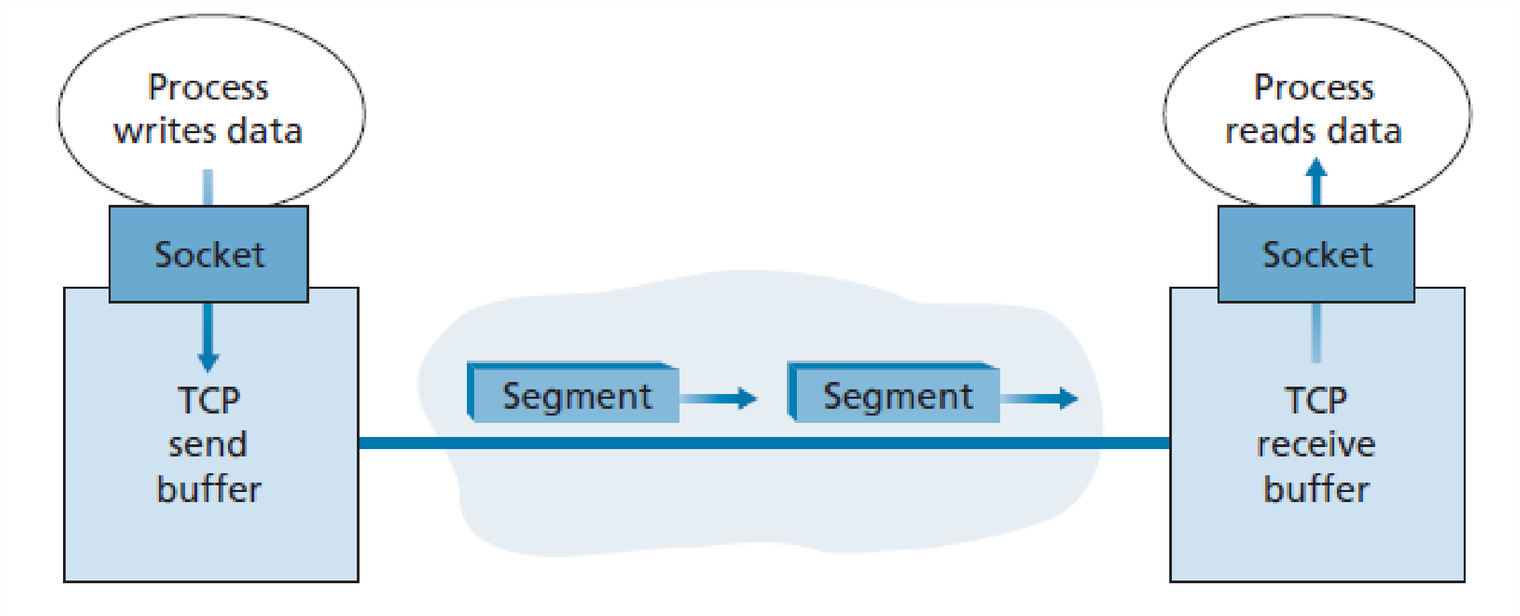

Sender
Receiver
pkt0 sent
0 1 2 3 4 5 6 7 8 9
pkt1 sent
0 1 2 3 4 5 6 7 8 9
pkt2 sent
0 1 2 3 4 5 6 7 8 9
X
(loss)
pkt3 sent, window full
0 1 2 3 4 5 6 7 8 9
ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9
ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9
pkt2 TIMEOUT, pkt2
resent
0 1 2 3 4 5 6 7 8 9
ACK3 rcvd, nothing sent
0 1 2 3 4 5 6 7 8 9
pkt0 rcvd, delivered, ACK0 sent
0 1 2 3 4 5 6 7 8 9
pkt1 rcvd, delivered, ACK1 sent
0 1 2 3 4 5 6 7 8 9
pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9
pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9
pkt5 rcvd; buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9
pkt2 rcvd, pkt2,pkt3,pkt4,pkt5
delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9

Figure 3.26

Mechanism	Use, Comments
Checksum	Used to detect bit errors in a transmitted packet.
Timer	Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within the channel. Because timeouts can occur when a packet is delayed but not lost (premature timeout), or when a packet has been received by the receiver but the receiver-to-sender ACK has been lost, duplicate copies of a packet may be received by a receiver.
Sequence number	Used for sequential numbering of packets of data flowing from sender to receiver. Gaps in the sequence numbers of received packets allow the receiver to detect a lost packet. Packets with duplicate sequence numbers allow the receiver to detect duplicate copies of a packet.
Acknowledgment	Used by the receiver to tell the sender that a packet or set of packets has been received correctly. Acknowledgments will typically carry the sequence number of the packet or packets being acknowledged. Acknowledgments may be individual or cumulative, depending on the protocol.
Negative acknowledgment	Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgments will typically carry the sequence number of the packet that was not received correctly.
Window, pipelining	The sender may be restricted to sending only packets with sequence numbers that fall within a given range. By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation.

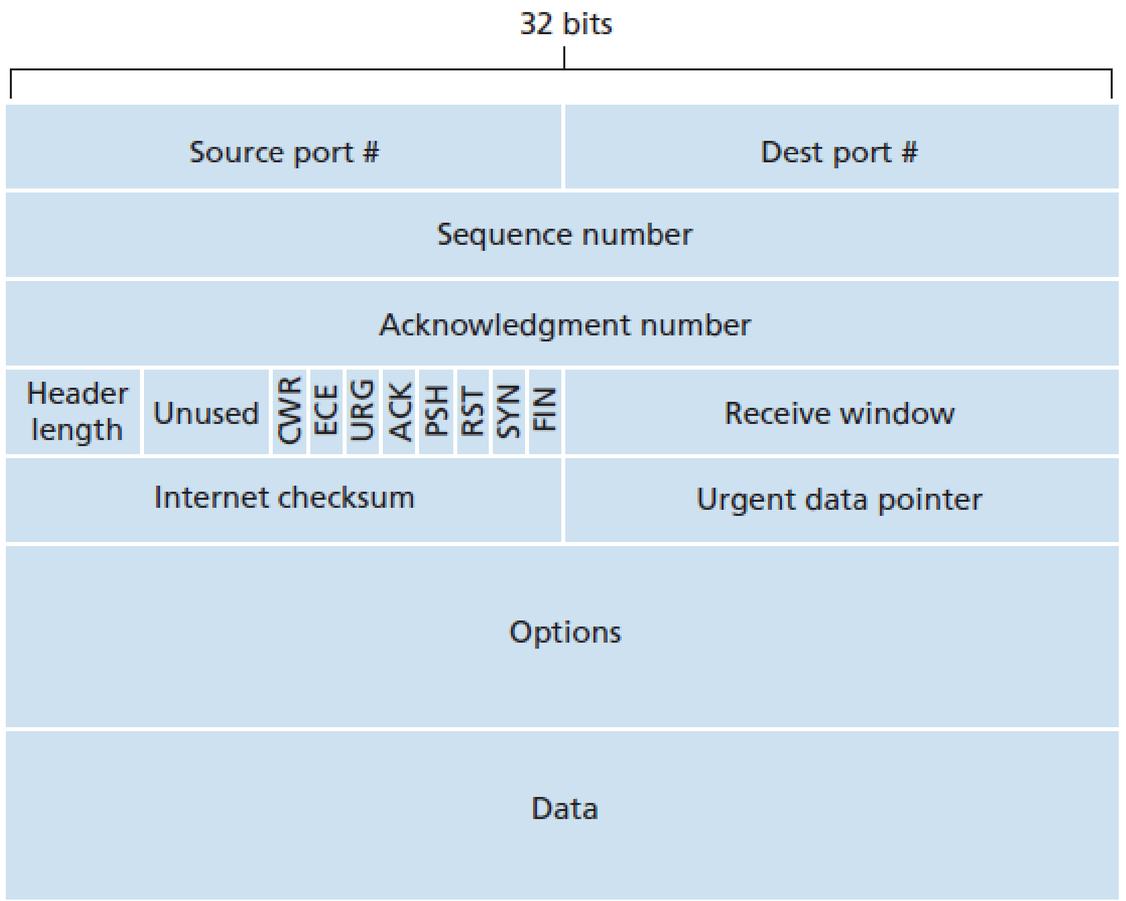
Connection-Oriented Transport: TCP

- TCP is said to be connection-oriented because before one application process can begin to send data to another, the two processes must first “handshake” with each other—that is, they must send some preliminary segments to each other to establish the parameters of the ensuing data transfer. As part of TCP connection establishment, both sides of the connection will initialize many TCP state variables associated with the TCP connection.



TCP Segment Structure

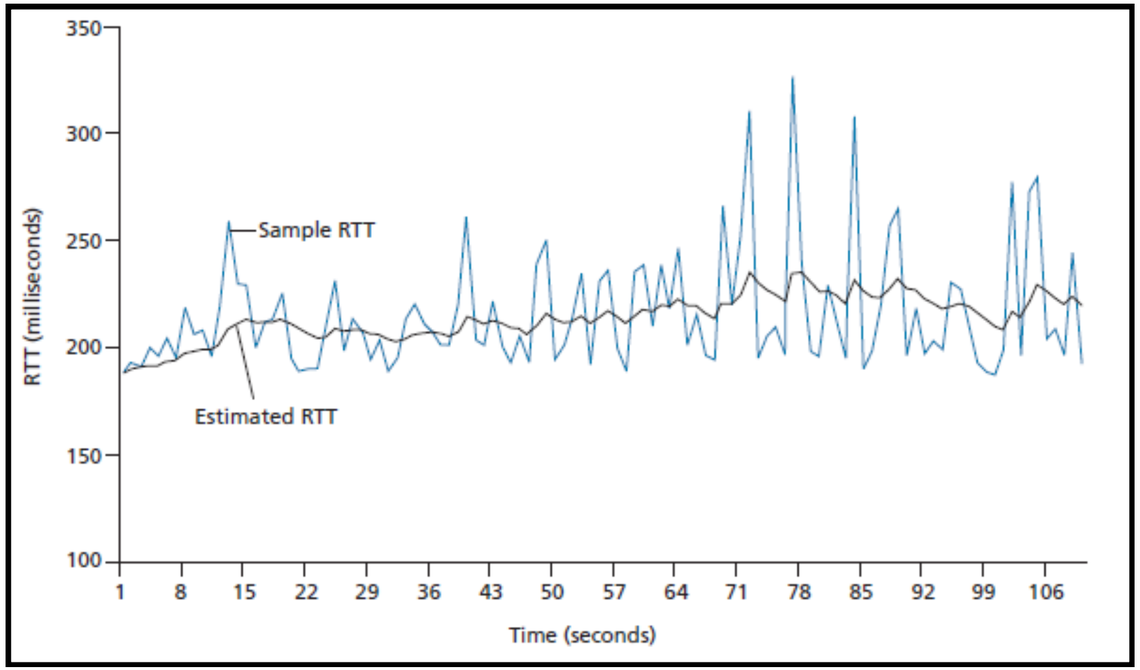
- The TCP segment consists of header fields and a data field. The data field contains a chunk of application data.
- When TCP sends a large file, such as an image as part of a Web page, it typically breaks the file into chunks of size MSS (Maximum segment size)
- Interactive applications, however, often transmit data chunks that are smaller than the MSS
- The TCP header is typically 20 bytes (12 bytes more than the UDP header).
- As with UDP, the header includes source and destination port numbers, which are used for multiplexing/demultiplexing data from/to upper-layer applications.
- Also, as with UDP, the header includes a checksum field.



- The 32-bit sequence number field and the 32-bit acknowledgment number field are used by the TCP sender and receiver in implementing a reliable data transfer service.
- The 16-bit receive window field is used for flow control. It is used to indicate the number of bytes that a receiver is willing to accept.
- The 4-bit header length field specifies the length of the TCP header in 32-bit words. The TCP header can be of variable length due to the TCP options field. (Typically, the options field is empty, so that the length of the typical TCP header is 20 bytes.)
- The optional and variable-length options field is used when a sender and receiver negotiate the maximum segment size (MSS)
- The flag field contains 8 bits:
    - The ACK bit is used to indicate that the value carried in the acknowledgment field is valid; that is, the segment contains an acknowledgment for a segment that has been successfully received.
    - The RST, SYN, and FIN bits are used for connection setup and teardown.
    - The CWR and ECE bits are used in explicit congestion notification.
    - Setting the PSH bit indicates that the receiver should pass the data to the upper layer immediately
    - Finally, the URG bit is used to indicate that there is data in this segment that the sending-side upperlayer entity has marked as “urgent.” The location of the last byte of this urgent data is indicated by the 16-bit urgent data pointer field.
- (In practice, the PSH, URG, and the urgent data pointer are not used.

Round-Trip Time Estimation and Timeout

- The timeout should be larger than the connection's round-trip time (RTT), that is, the time from when a segment is sent until it is acknowledged. Otherwise, unnecessary retransmissions would be sent.
- The sample RTT, denoted SampleRTT, for a segment is the amount of time between when the segment is sent (that is, passed to IP) and when an acknowledgment for the segment is received.
- TCP maintains an average, called EstimatedRTT, of the SampleRTT values. Upon obtaining a new SampleRTT, TCP updates EstimatedRTT according to the following formula:
  - EstimatedRTT = (1 – α) . EstimatedRTT + α . SampleRTT
  - The recommended value of α is α = 0.125
  - EstimatedRTT = 0.875 . EstimatedRTT + 0.125 . SampleRTT



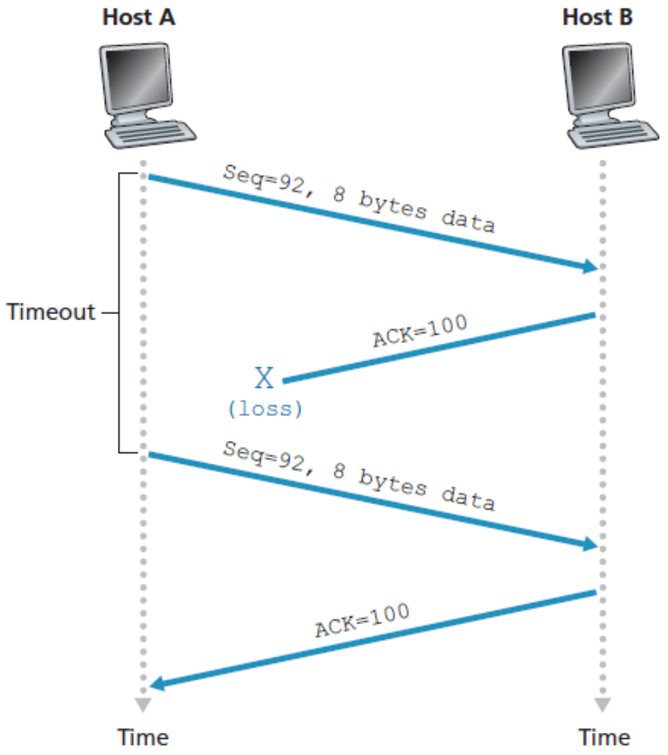
Reliable Data Transfer in TCP

- Recall that the Internet's network-layer service (IP service) is unreliable. IP does not guarantee datagram delivery, does not guarantee in-order delivery of datagrams, and does not guarantee the integrity of the data in the datagrams.
- TCP creates a reliable data transfer service on top of IP's unreliable besteffort service.
- The recommended TCP timer management procedures use only a single retransmission timer, even if there are multiple transmitted but not yet acknowledged segments.
- There are three major events related to data transmission and retransmission in the TCP sender:
  - data received from application above
  - timer timeout
  - ACK receipt

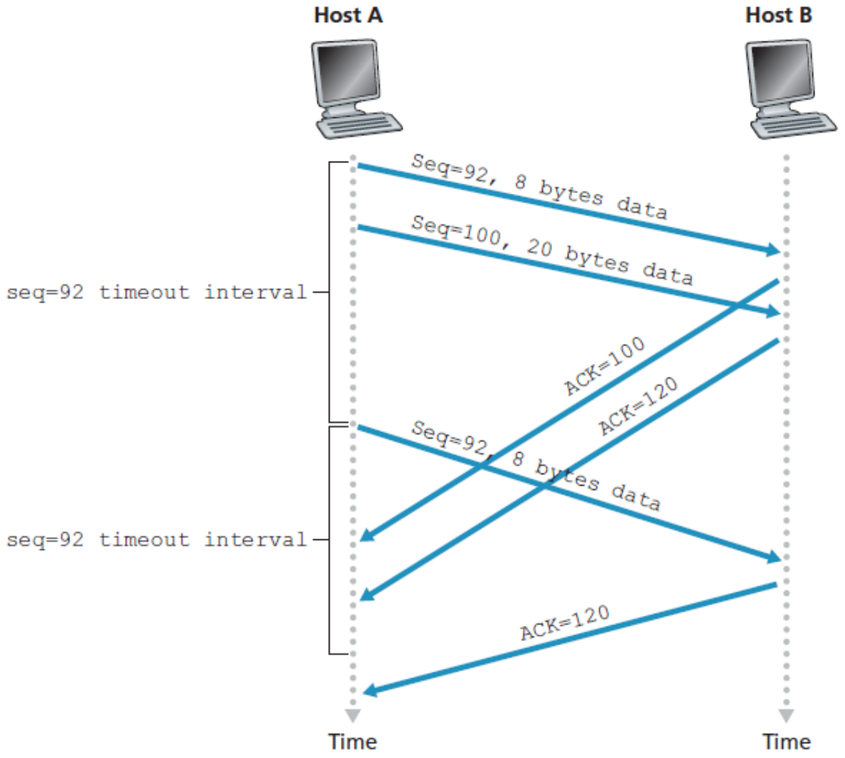
```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum=NextSeqNum+length(data)
            break;

        event: timer timeout
            retransmit not-yet-acknowledged segment with
                smallest sequence number
            start timer
            break;

        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase=y
                if (there are currently any not-yet-acknowledged segments)
                    start timer
                }
            break;

    } /* end of loop forever */
```

Simplified TCP sender

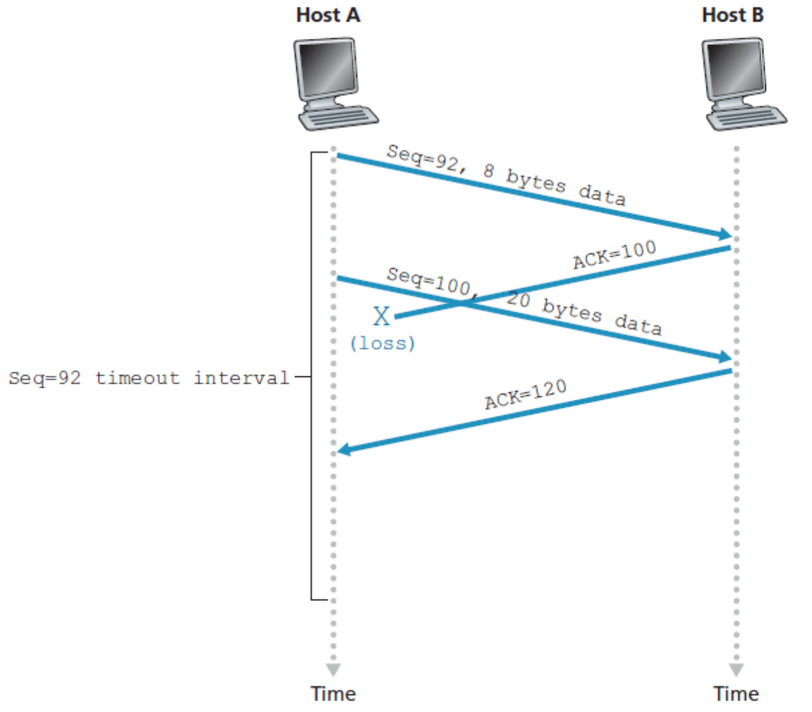
A Few Interesting Scenarios



Retransmission due to a lost acknowledgment



Segment 100 not retransmitted



A cumulative acknowledgment avoids retransmission of the first segment

Go-Back-N or Selective Repeat?

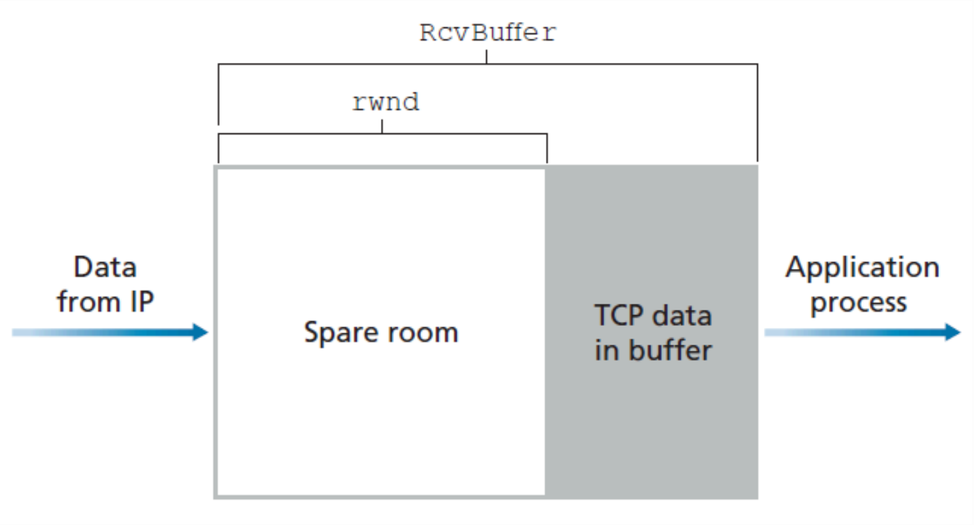
Is TCP a GBN or an SR protocol?

- TCP acknowledgments are cumulative and correctly received but out-of-order segments are not individually ACKed by the receiver.
- The TCP sender need only maintain the smallest sequence number of a transmitted but unacknowledged byte (SendBase) and the sequence number of the next byte to be sent (NextSeqNum). In this sense, TCP looks a lot like a GBN-style protocol.
- There are some striking differences between TCP and Go-Back-N. Many TCP implementations will buffer correctly received but out-of-order segments.
- Consider what happens when the sender sends a sequence of segments 1, 2, . . . , N, and all of the segments arrive in order without error at the receiver. Further suppose that the acknowledgment for packet n < N gets lost, but the remaining N - 1 acknowledgments arrive at the sender before their respective timeouts. In this example, GBN would retransmit not only packet n, but also all of the subsequent packets n + 1, n + 2, . . . , N. TCP, on the other hand, would retransmit at most one segment, namely, segment n. Moreover, TCP would not even retransmit segment n if the acknowledgment for segment n + 1 arrived before the timeout for segment n.

Flow Control

- hosts on each side of a TCP connection set aside a receive buffer for the connection.
- When the TCP connection receives bytes that are correct and in sequence, it places the data in the receive buffer.
- The associated application process will read data from this buffer, but not necessarily at the instant the data arrives.
- If the application is relatively slow at reading the data, the sender can very easily overflow the connection’s receive buffer by sending too much data too quickly.
- TCP provides a flow-control service to its applications to eliminate the possibility of the sender overflowing the receiver’s buffer. Flow control is thus a speedmatching service—matching the rate at which the sender is sending against the rate at which the receiving application is reading.
- TCP provides flow control by having the sender maintain a variable called the receive window. Informally, the receive window is used to give the sender an idea of how much free buffer space is available at the receiver.

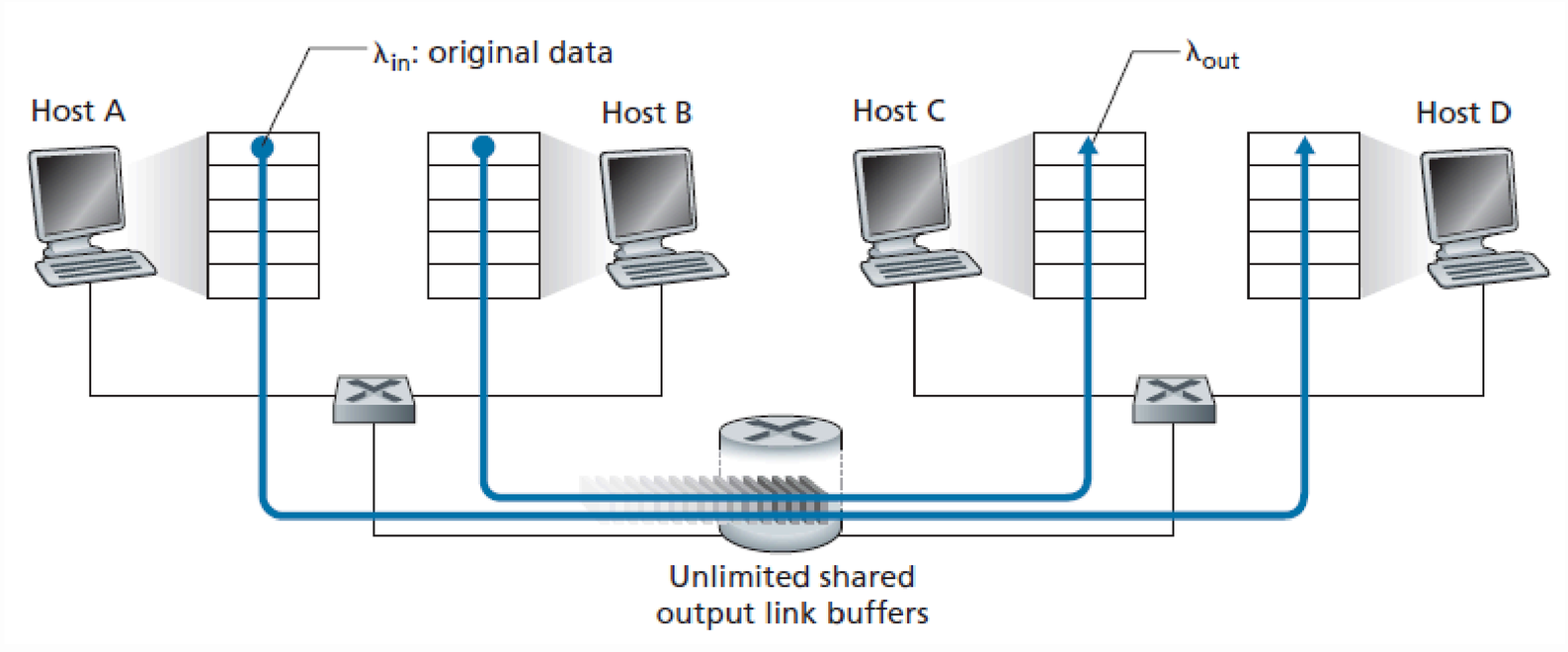
- Suppose that Host A is sending a large file to Host B over a TCP connection. Host B allocates a receive buffer to this connection; denote its size by RcvBuffer. From time to time, the application process in Host B reads from the buffer. Define the following variables:
  - LastByteRead: the number of the last byte in the data stream read from the buffer by the application process in B
  - LastByteRcvd: the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B
- Because TCP is not permitted to overflow the allocated buffer, we must have:
  - LastByteRcvd – LastByteRead =< RcvBuffer
- The receive window, denoted rwnd is set to the amount of spare room in the buffer:
  - rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]
- Because the spare room changes with time, rwnd is dynamic. The variable rwnd is illustrated in the Figure.
- Host B tells Host A how much spare room it has in the connection buffer by placing its current value of rwnd in the receive window field of every segment it sends to A. Initially, Host B sets
  - rwnd = RcvBuffer.



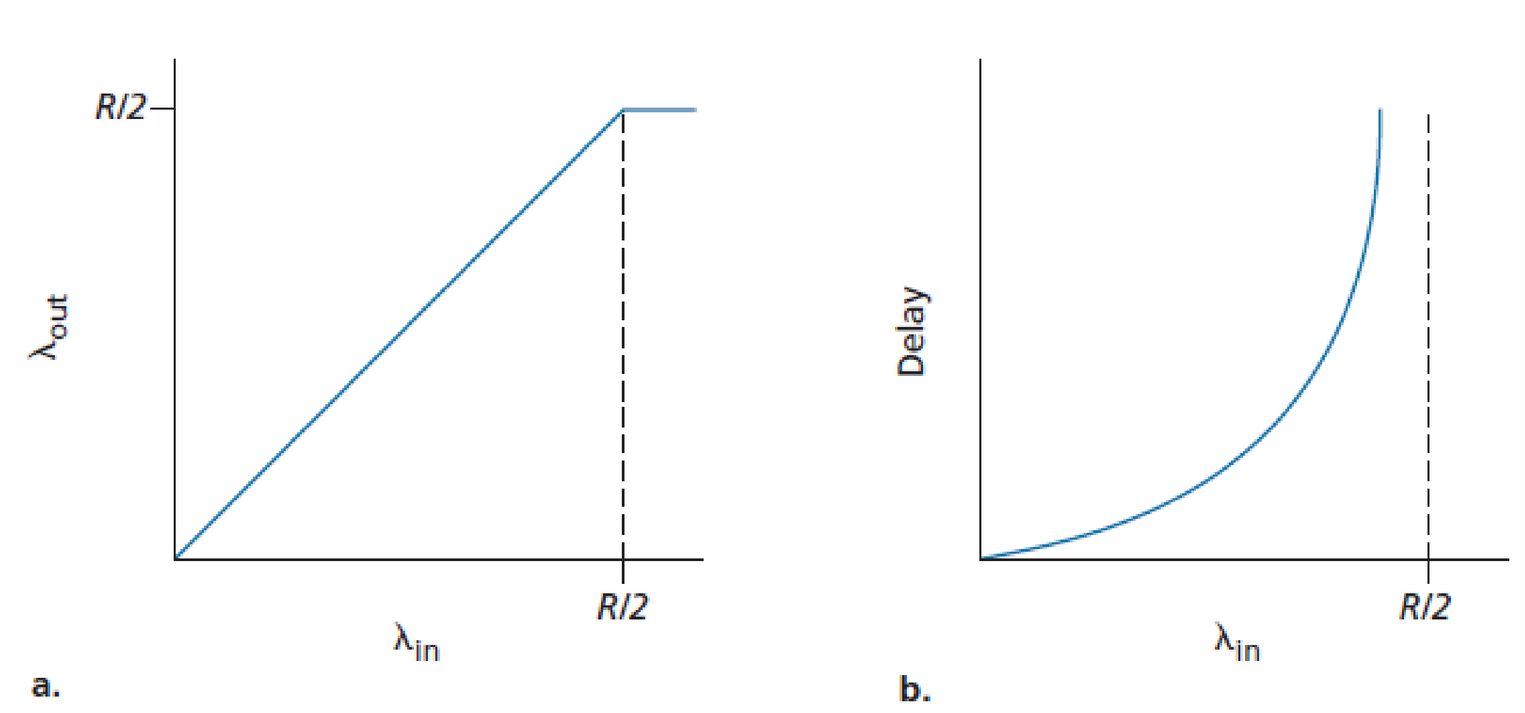
- Host A in turn keeps track of two variables, LastByteSent and Last-ByteAcked, which have obvious meanings. Note that the difference between these two variables, LastByteSent – LastByteAcked, is the amount of unacknowledged data that A has sent into the connection. By keeping the amount of unacknowledged data less than the value of rwnd, Host A is assured that it is not overflowing the receive buffer at Host B. Thus, Host A makes sure throughout the connection’s life that:
    - LastByteSent – LastByteAcked =< rwnd

Principles of Congestion Control

- To treat the cause of network congestion, mechanisms are needed to throttle senders in the face of network congestion.
- Scenario: Two Senders, a Router with Infinite Buffers:
- Two hosts (A and B) each have a connection that shares a single hop between source and destination.
- The application in Host A is sending data into the connection at an average rate of λ_{in} bytes/sec.
- Host B operates in a similar manner, and we assume that it too is sending at a rate bytes/sec.
- Packets from Hosts A and B pass through a router and over a shared outgoing link of capacity R.
- In this scenario, we assume that the router has an infinite amount of buffer space.



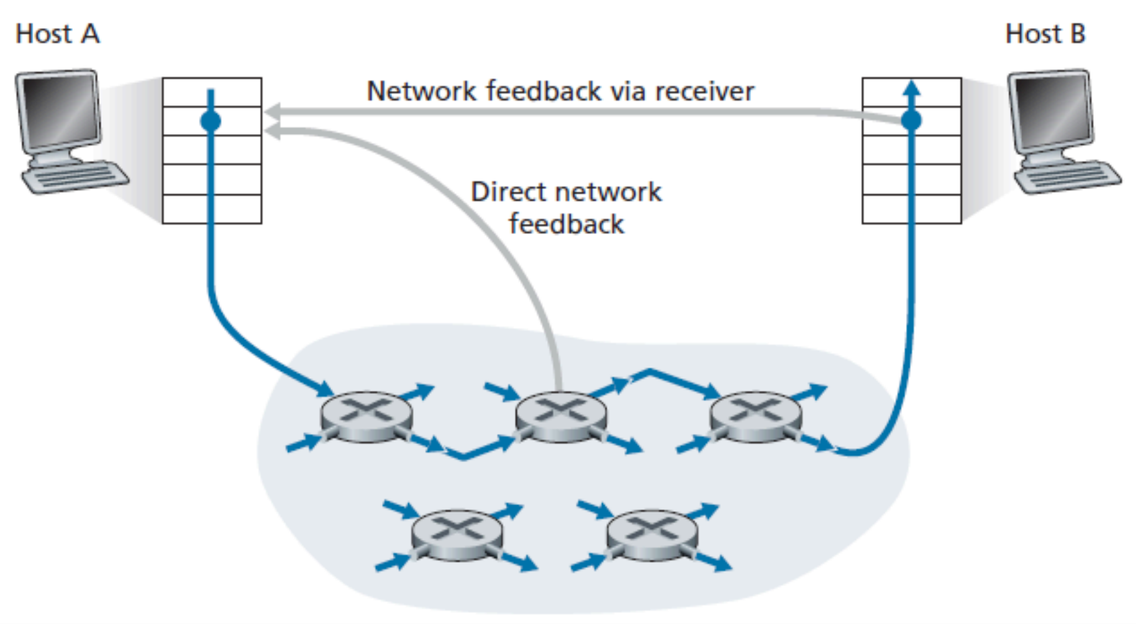
- For a sending rate between 0 and R/2, the throughput at the receiver equals the sender's sending rate. Everything sent by the sender is received at the receiver with a finite delay.
- When the sending rate is above R/2, however, the throughput is only R/2. This upper limit on throughput is a consequence of the sharing of link capacity between two connections. The link simply cannot deliver packets to a receiver at a steady-state rate that exceeds R/2. No matter how high Hosts A and B set their sending rates, they will each never see a throughput higher than R/2.
- As the sending rate approaches R/2, the average delay becomes larger and larger. When the sending rate exceeds R/2, the average number of queued packets in the router is unbounded, and the average delay between source and destination becomes infinite.



Approaches to Congestion Control

- We identify the two broad approaches to congestion control that are taken in practice.
- At the highest level, we can distinguish among congestion-control approaches by whether the network layer provides explicit assistance to the transport layer for congestion-control purposes:
  - End-to-end congestion control: In an end-to-end approach to congestion control, the network layer provides no explicit support to the transport layer for congestion-control purposes. The presence of network congestion must be inferred by the end systems based only on observed network behavior (for example, packet loss and delay). TCP takes this end-to-end approach toward congestion control, since the IP layer is not required to provide feedback to hosts regarding network congestion. TCP segment loss (as indicated by a timeout or the receipt of three duplicate acknowledgments) is taken as an indication of network congestion, and TCP decreases its window size accordingly.
  - Network-assisted congestion control: With network-assisted congestion control, routers provide explicit feedback to the sender and/or receiver regarding the congestion state of the network. This feedback may be as simple as a single bit indicating congestion at a link.

- For network-assisted congestion control, congestion information is typically fed back from the network to the sender in one of two ways, as shown in the Figure:
  - Direct feedback may be sent from a network router to the sender. This form of notification typically takes the form of a choke packet (essentially saying, “I’m congested!”).
  - The second and more common form of notification occurs when a router marks/updates a field in a packet flowing from sender to receiver to indicate congestion. Upon receipt of a marked packet, the receiver then notifies the sender of the congestion indication. This latter form of notification takes a full round-trip time.



Summary

- The transport-layer protocol can be very simple and offer a no-frills service to applications, providing only a multiplexing/demultiplexing function for communicating processes like in UDP.
- A transport-layer protocol can provide a variety of guarantees to applications, such as reliable delivery of data, delay guarantees, and bandwidth guarantees. Nevertheless, the services that a transport protocol can provide are often constrained by the service model of the underlying network-layer protocol.
- A transport-layer protocol can provide reliable data transfer even if the underlying network layer is unreliable.
- We took a close look at TCP, the Internet's connection-oriented and reliable transport-layer protocol. We learned that TCP is complex, involving connection management, flow control, and round-trip time estimation, as well as reliable data transfer.
- We learned that congestion control is imperative for the well-being of the network. Without congestion control, a network can easily become gridlocked.
- In Chapter 1, we said that a computer network can be partitioned into the "network edge" and the "network core." The network edge covers everything that happens in the end systems. Having now covered the application layer and the transport layer, our discussion of the network edge is complete. It is time to explore the network core! This journey begins in the next chapter.