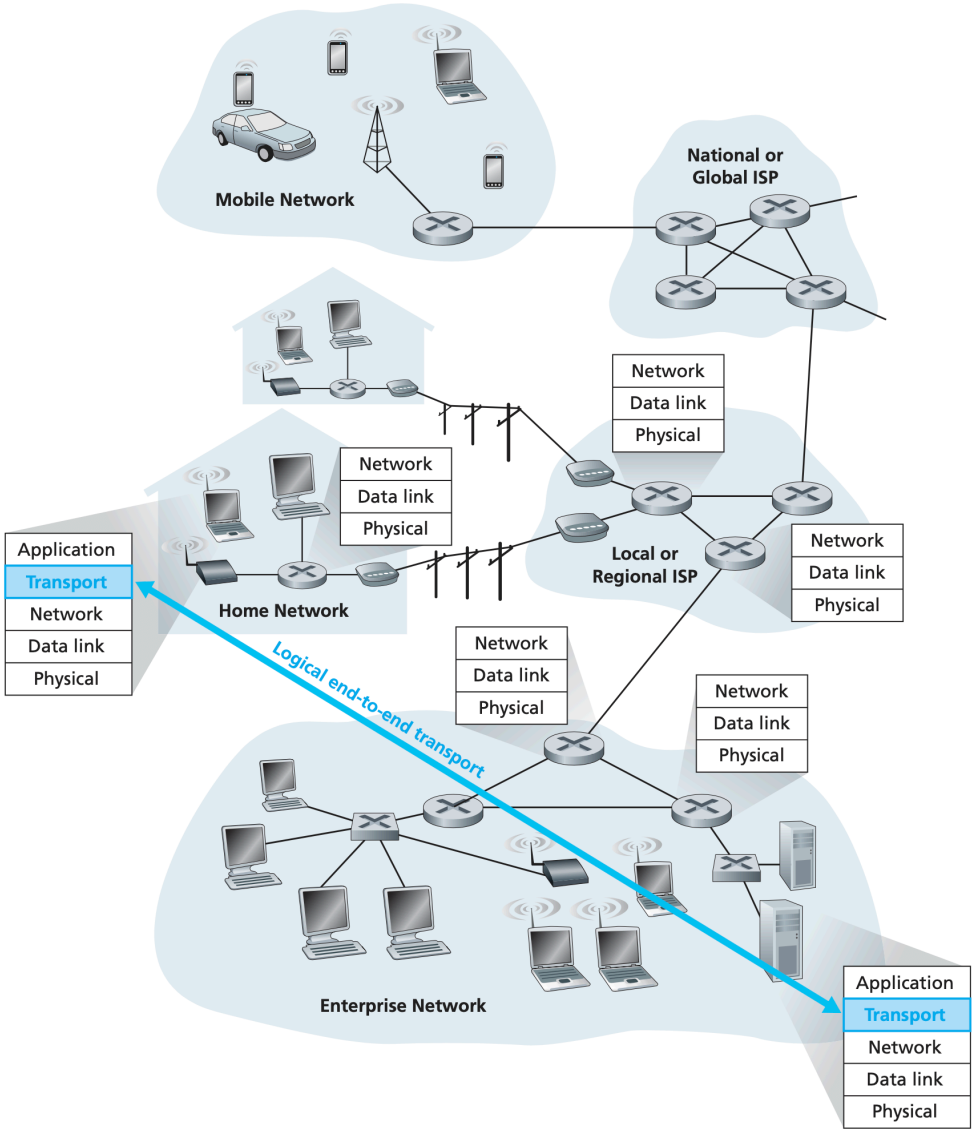
Transport Layer

Hailiang Zhao @ ZJU.CS.CCNT
http://hliangzhao.me

This slide is a reminder for chapter 3 of the book *Computer Networking: A Top-Down Approach.*

Introduction to Transport-layer Services

- The transport layer provides *logical* rather than physical communication between application process

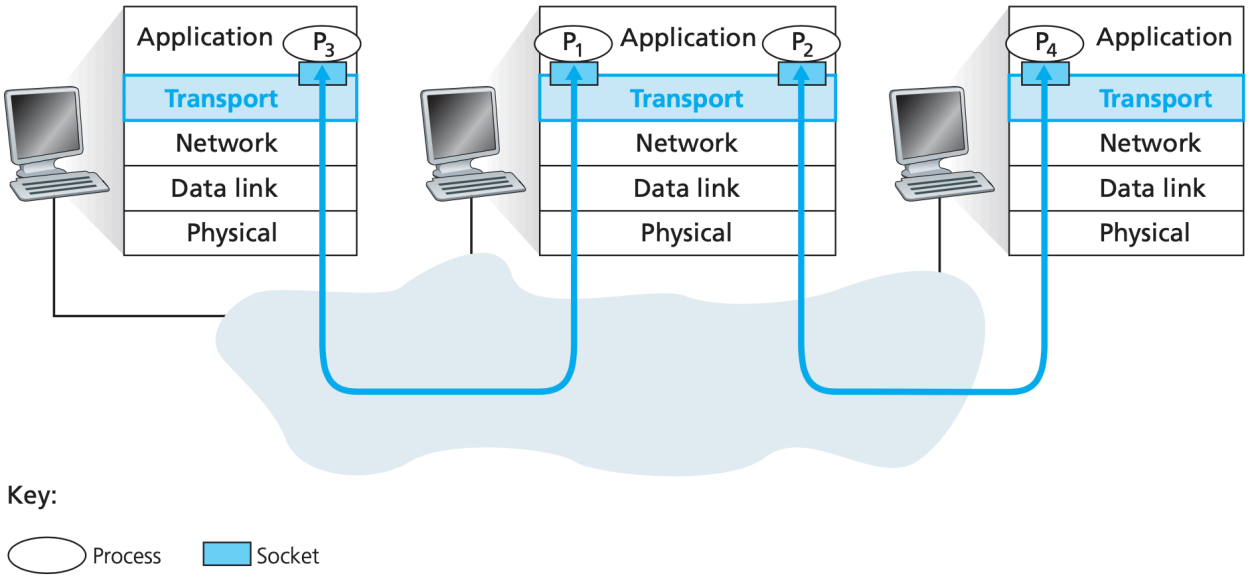


Introduction to Transport-layer Services

- The transport layer converts the application-layer *messages* it receives from a sending application process into transport-layer packets, known as transport-layer *segments*
  - breaking the application messages into smaller chunks and adding a transport-layer header to each chunk
- A transport-layer protocol provides logical communication between *processes* running on different end hosts
- A network-layer protocol provides logical communication between *hosts*
- The IP service model is **a best-effort delivery service** (unreliable and make no guarantees)
- UDP
  - process-to-process data delivery
  - error checking
- TCP
  - besides what UDP provides, also ...
  - flow control
  - congestion control

Multiplexing and Demultiplexing

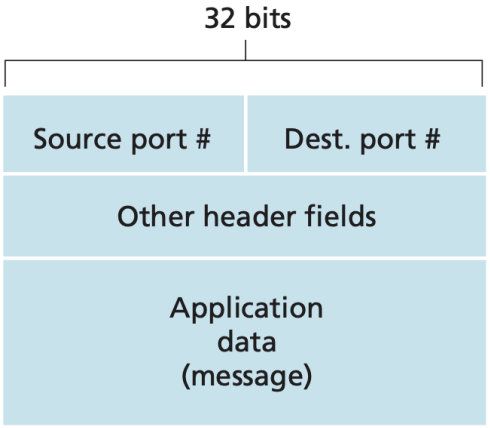
- The transport-layer multiplexing and demultiplexing is *extending the host-to-host delivery service provided by the network layer to a process-to-process delivery service for applications running on the hosts*
  - the job of delivering the data in a transport-layer segment to the correct socket is called **demultiplexing**
  - the job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information (that will later be used in demultiplexing) to create segments, and passing the segments to the network layer is called **multiplexing**
  - each socket has a unique identifier depend on whether it is a UDP or TCP socket



Demultiplexing: deliver to P1 or P2; Multiplexing: gather data from P1 and P2 and send them out

Multiplexing and Demultiplexing

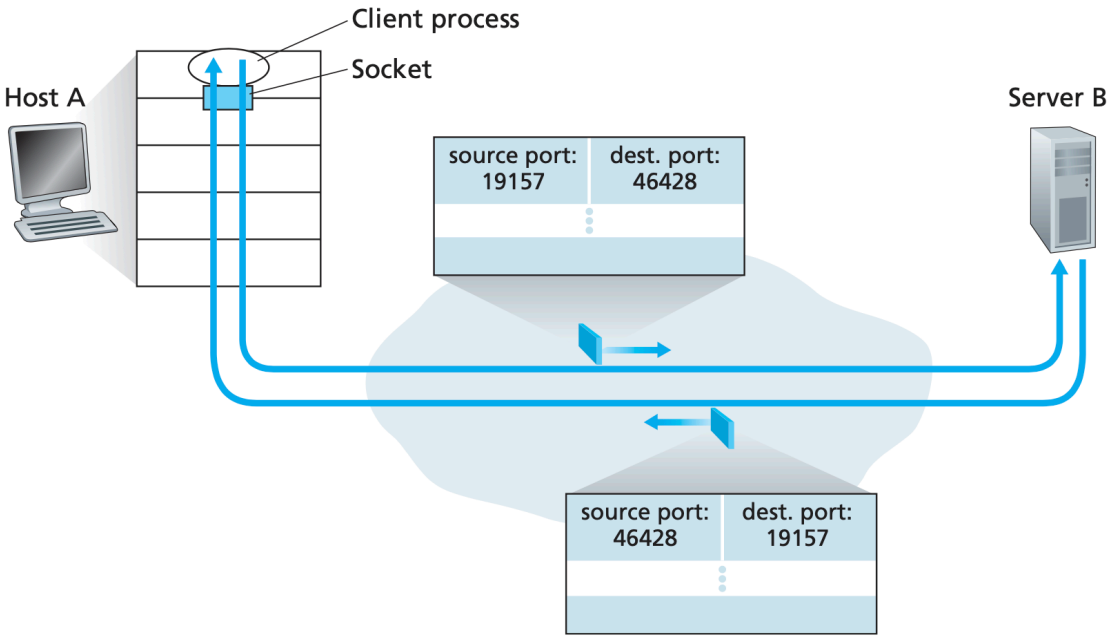
- The transport-layer multiplexing requires (1) that sockets have *unique* identifiers, and (2) that each segment have *special fields* that indicate the socket to which the segment is to be delivered
  - the special fields are are the **source port number field** and the **destination port number field**
  - 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
- How the transport-layer demultiplexing works (UDP does like this, TCP more complex)
  - each socket in the host is 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
  - each segment's data then passes through the socket into the attached process



Source and destination port-number fields in a transport-layer segment

Multiplexing and Demultiplexing

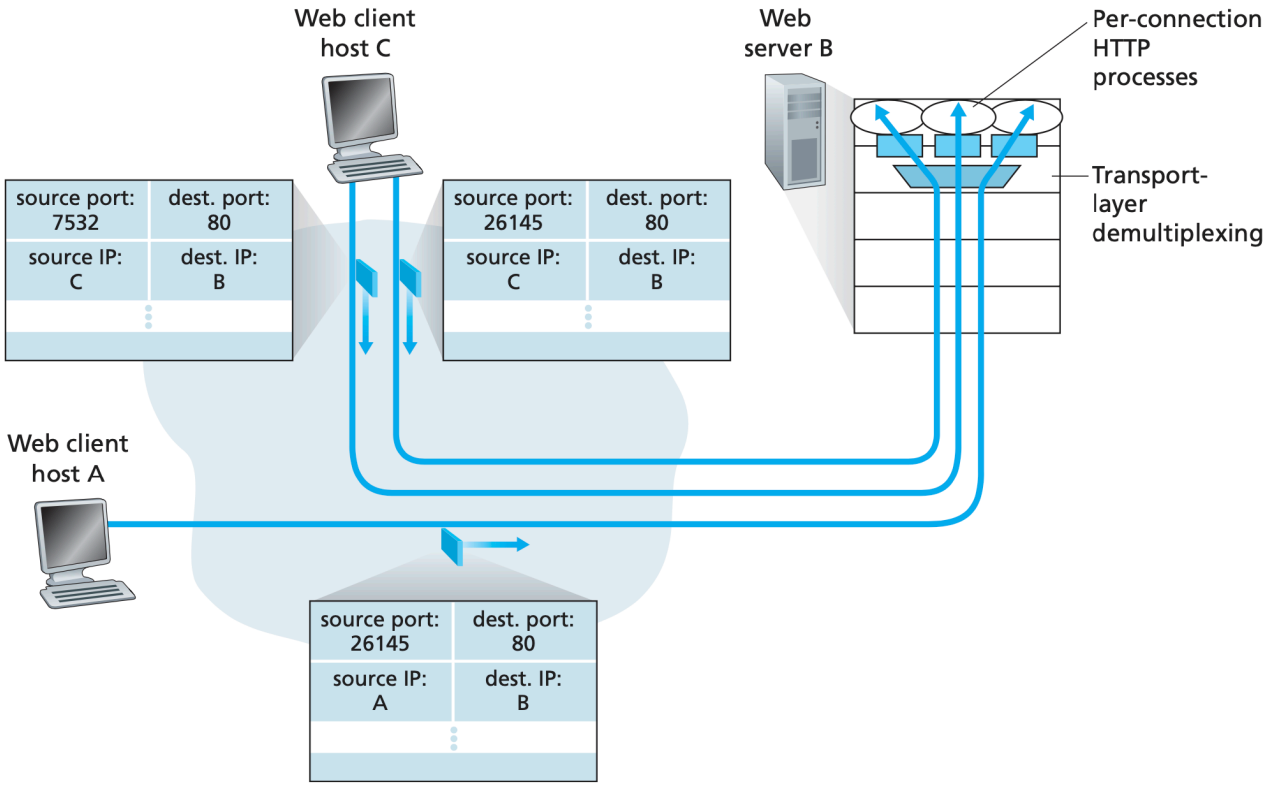
- Connectionless multiplexing and demultiplexing
  - a UDP socket is fully identified by a **two-tuple** consisting of *a destination IP address and a destination port number*. if two UDP segments have different source IP addresses and/or source port numbers, but have the same destination IP address and destination port number, then the two segments will be directed to the **same** destination process **via the same destination socket**
  - typically, the client side of the application lets the transport layer automatically (and transparently) assign the port number, whereas the server side of the application assigns a *specific* port number



The inversion of source and destination port numbers

Multiplexing and Demultiplexing

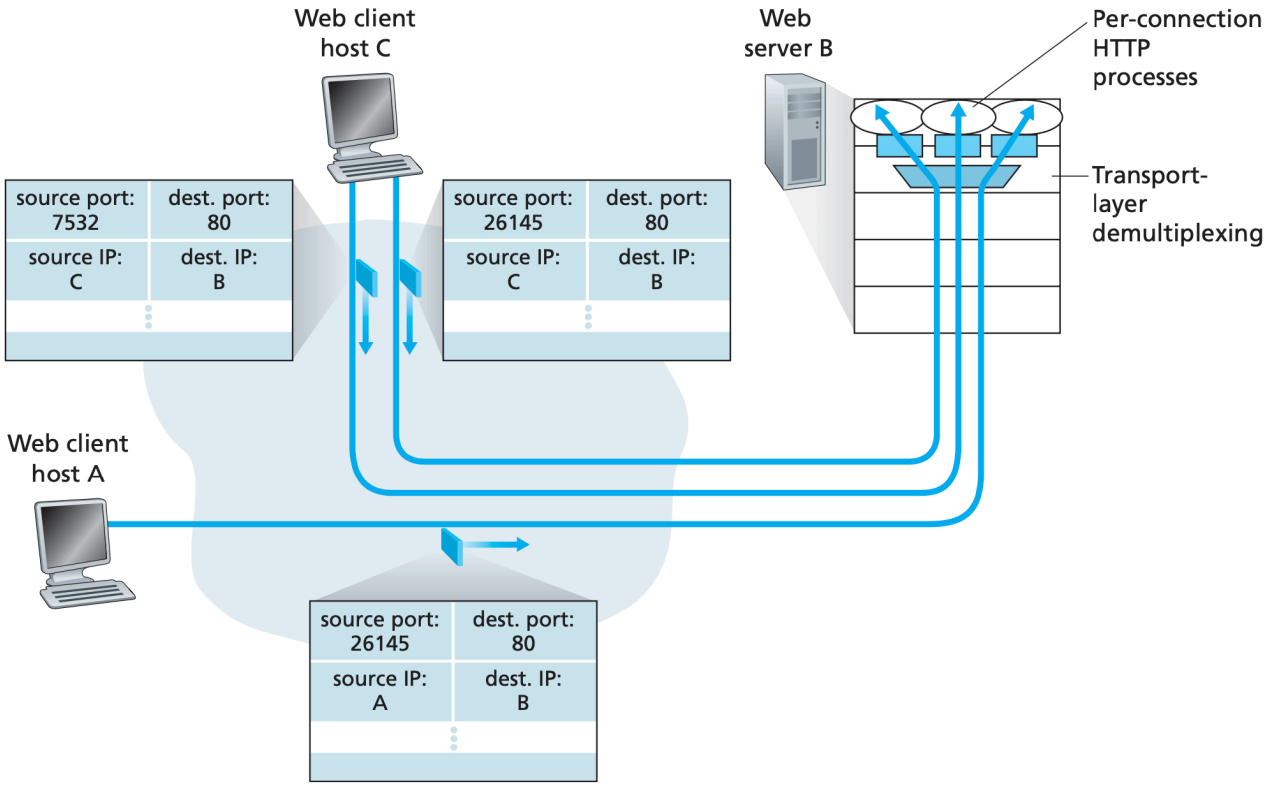
- Connection-oriented multiplexing and demultiplexing
  - a TCP socket is identified by a **four-tuple**: (source IP address, source port number, destination IP address, destination port number)
  - in contrast with UDP, two arriving TCP segments with different source IP addresses or source port numbers will (with the exception of a TCP segment carrying the original connection-establishment request) be directed to **two different sockets**



Two clients, using the same destination port number (80) to communicate with the same Web server application

Multiplexing and Demultiplexing

- Connection-oriented multiplexing and demultiplexing (cont'd)
  - the fig. below shows a Web server that spawns a **new** process for each connection
  - in fact, today's high-performing Web servers often use only **one** process, and create a new **thread** with a new connection socket for each new client connection
  - frequent creating and closing of sockets can severely impact the performance of a busy Web server



Two clients, using the same destination port number (80) to communicate with the same Web server application

UDP

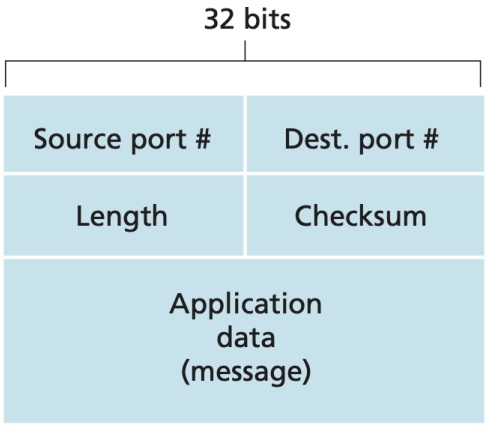
- Aside from the multiplexing/demultiplexing function and some light error checking, it adds nothing to IP
  - connectionless: no handshaking between sending and receiving transport-layer entities before sending a segment
- Why we use UDP?
  - real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss
  - no three-way handshake delay [sending before handshake = connection-oriented]
  - no cost for maintaining connection state
  - small packet header overhead

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

Popular Internet applications and their underlying transport protocols

UDP

- UDP segment structure
  - the application data occupies the **data field** of the UDP segment
  - the **length field** specifies the number of bytes in the UDP segment (header plus data)
  - the **checksum** is used by the receiving host to check whether errors have been introduced into the segment
- Checksum details:
  - 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
  - at the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the packet, then clearly the sum at the receiver will be *all ones*
  - why UDP needs checksum? Because there is no guarantee that all the links between source and destination provide error checking
  - if error found, just discard the segment

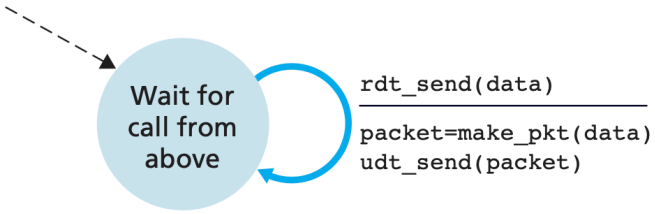


Principles of Reliable Data Transfer

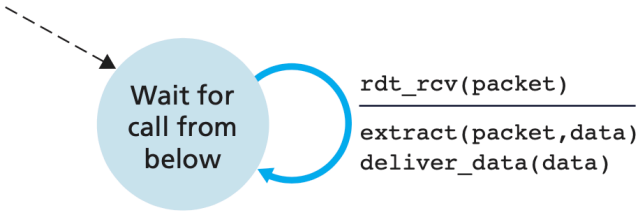
- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0
  - Reliable Data Transfer over a Channel with Bit Errors: rdt2.0
  - Reliable Data Transfer over a Lossy Channel with Bit Errors: rdt3.0
- Pipelined reliable data transfer protocols
- Go-Back-N (GBN)
- Selective Repeat (SR)

Principles of Reliable Data Transfer

- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0
    - the underlying channel is completely reliable
    - the sender and receiver FSMs shown below each have just one state
    - the event causing the transition is shown **above** the horizontal line labeling the transition, and the actions taken when the event occurs are shown **below** the horizontal line
    - the initial state of the FSM is indicated by the dashed arrow



a. rdt1.0: sending side



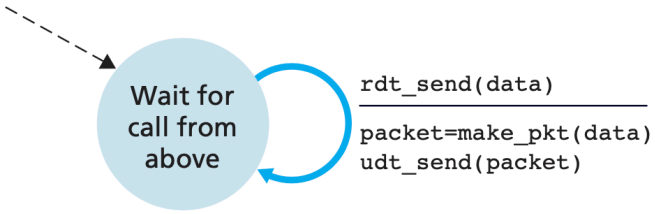
b. rdt1.0: receiving side

rdt1.0 – A protocol for a completely reliable channel

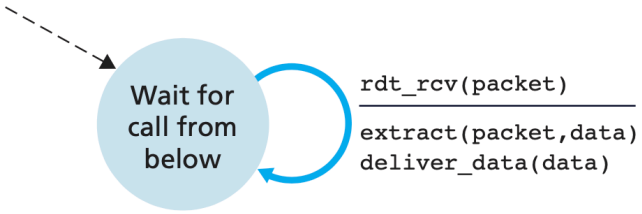
Principles of Reliable Data Transfer

- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0 (cont'd)

    - 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. The event would result from a procedure call by the upper-layer application.



a. rdt1.0: sending side



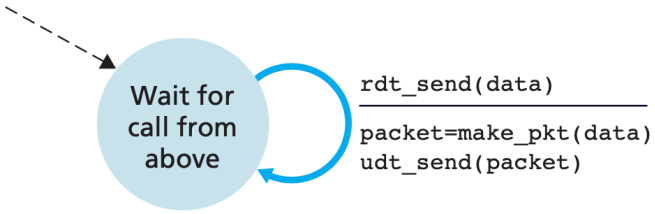
b. rdt1.0: receiving side

rdt1.0 – A protocol for a completely reliable channel

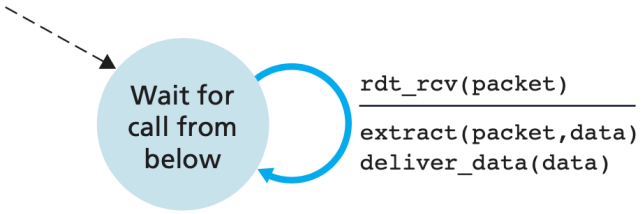
Principles of Reliable Data Transfer

- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0 (cont'd)

    - 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 event would result from a procedure call from the lower-layer protocol.



a. rdt1.0: sending side



b. rdt1.0: receiving side

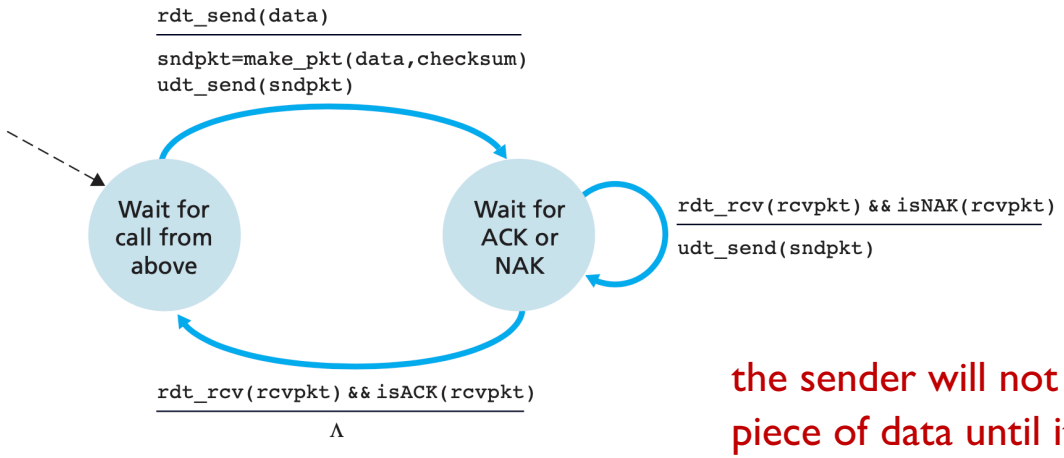
rdt1.0 – A protocol for a completely reliable channel

Principles of Reliable Data Transfer

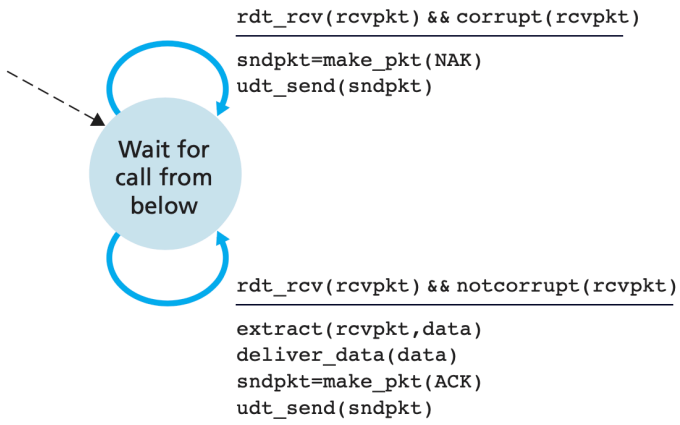
- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Channel with Bit Errors: rdt2.0
    - bit errors might occur in the physical components
    - all transmitted packets are received (although their bits may be corrupted) in the order in which they were sent
    - rdt2.0 should include a message-dictation protocol with positive acknowledgments ("OK") and negative acknowledgments ("Please repeat it")
    - reliable data transfer protocols based on such retransmission are known as **ARQ (Automatic Repeat reQuest) protocols**
      - error detection
      - receiver feedback (positive (ACK) and negative (NAK) acknowledgment)
      - retransmission

Principles of Reliable Data Transfer

- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt2.0 (cont'd)



a. rdt2.0: sending side



b. rdt2.0: receiving side

the sender will not send a new piece of data until it is sure that the receiver has correctly received the current packet (**stop-and-wait**)

rdt2.0 – A protocol for a channel with bit errors

Principles of Reliable Data Transfer

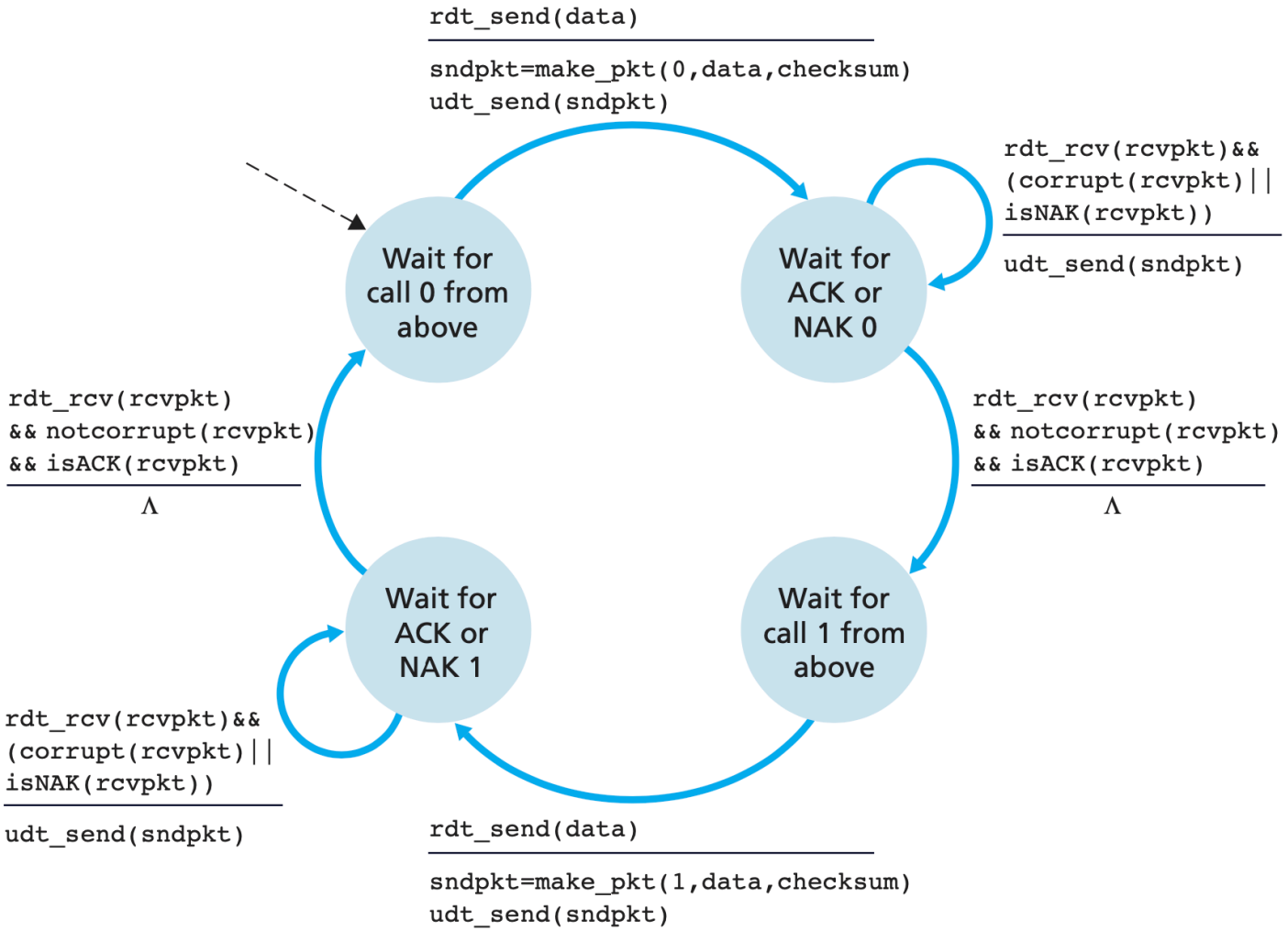
- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt2.0 (cont'd)
    - A fatal flaw: the ACK or NAK packet could be corrupted!
    - Minimally, we will need to add checksum bits to ACK/NAK packets in order to detect such errors
    - Three possibilities for handling corrupted ACKs or NAKs:
      - R: What did you say? S: What do *you* say? ...
      - add enough checksum bits to allow the sender not only to detect, but also to recover from, bit errors. This solves the immediate problem for a channel that can corrupt packets but not lose them (rdt2.0's assumption)
      - resend the current data packet when it receives a garbled ACK or NAK packet (however introduces duplicated packets. Which is the latest one?)
    - A simple but powerful solution: add a new field to the data packet and have the sender number its data packets by putting a **sequence** number into this field → rdt2.1

Principles of Reliable Data Transfer

- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt2.1
    - Since we are currently assuming a channel that does not lose packets, ACK and NAK packets do not themselves need to indicate the sequence number of the packet they are acknowledging. The sender knows that a received ACK or NAK packet (whether garbled or not) was generated in response to *its most recently transmitted data packet*
    - For this simple case of a **stop-and-wait protocol**, a **1-bit sequence number** will suffice, since it will allow the receiver to know whether the sender is resending the previously transmitted packet (the sequence number of the received packet has the same sequence number as the most recently received packet) or a new packet

Principles of Reliable Data Transfer

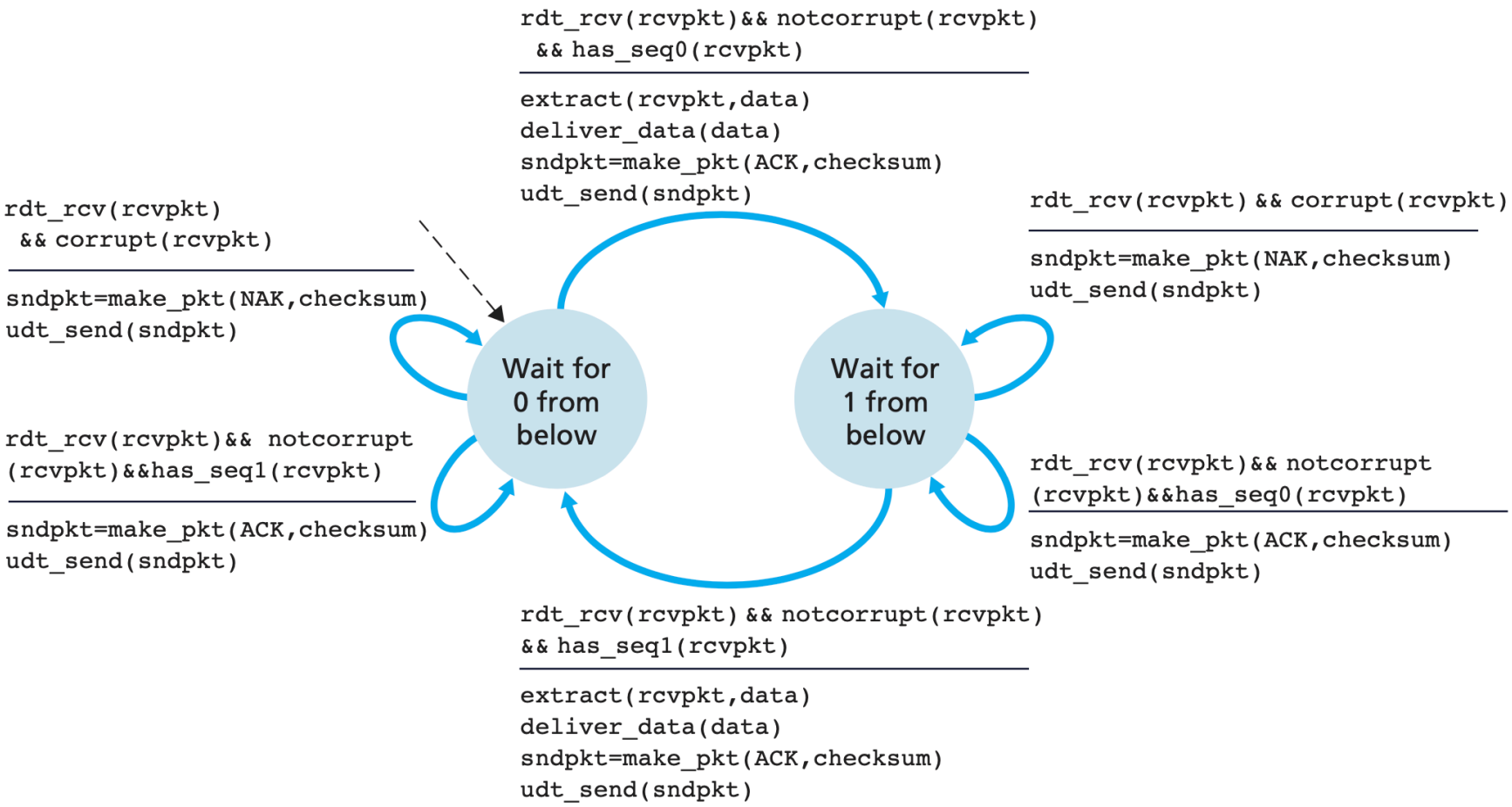
- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt2.1 (cont'd)



rdt2.1 - sender

Principles of Reliable Data Transfer

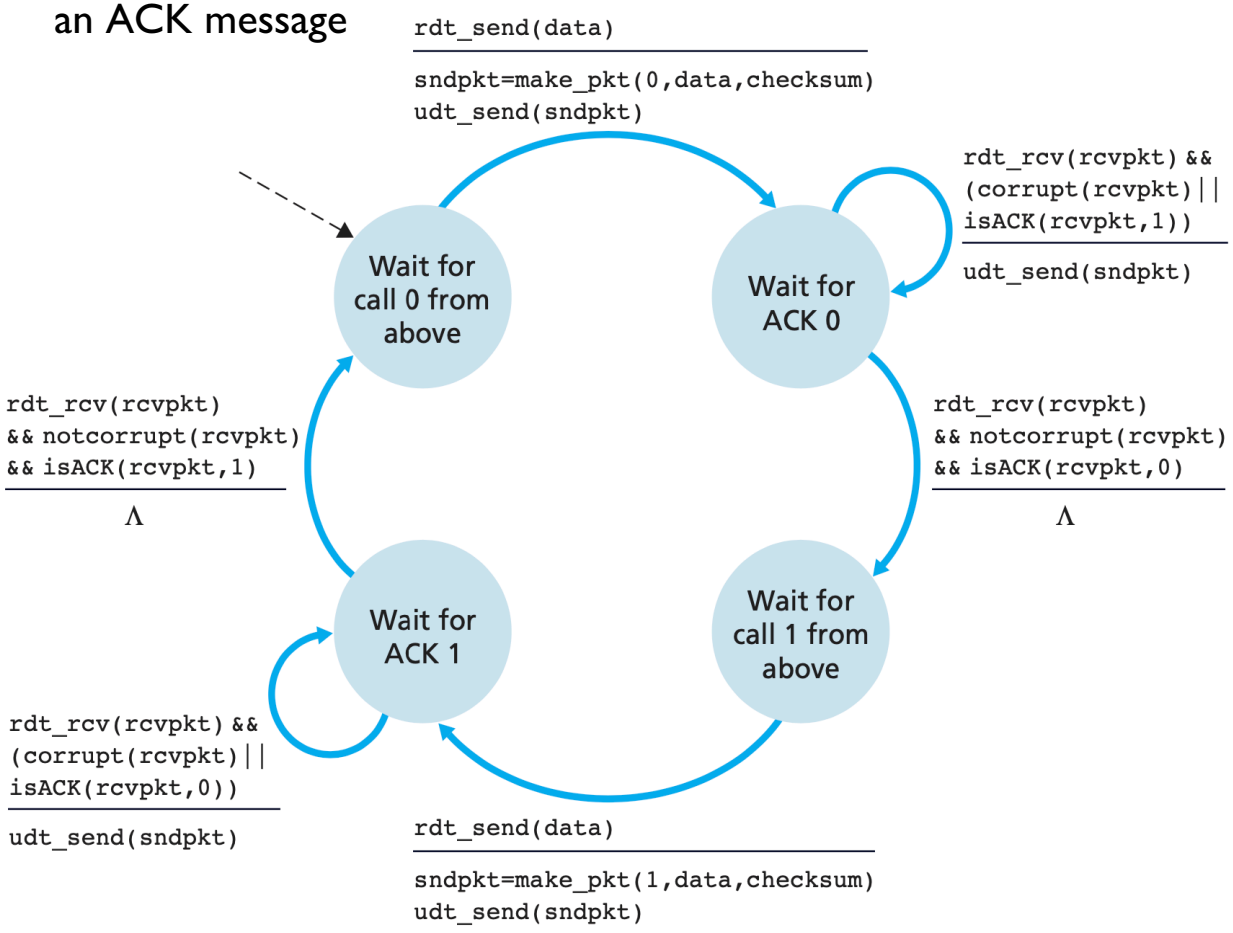
- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt2.1 (cont'd)



rdt2.1 - receiver

Principles of Reliable Data Transfer

- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt2.2
    - rdt2.1 uses both ACK and NAK. What if we only use ACK? → The receiver must now include the sequence number of the packet being acknowledged by an ACK message



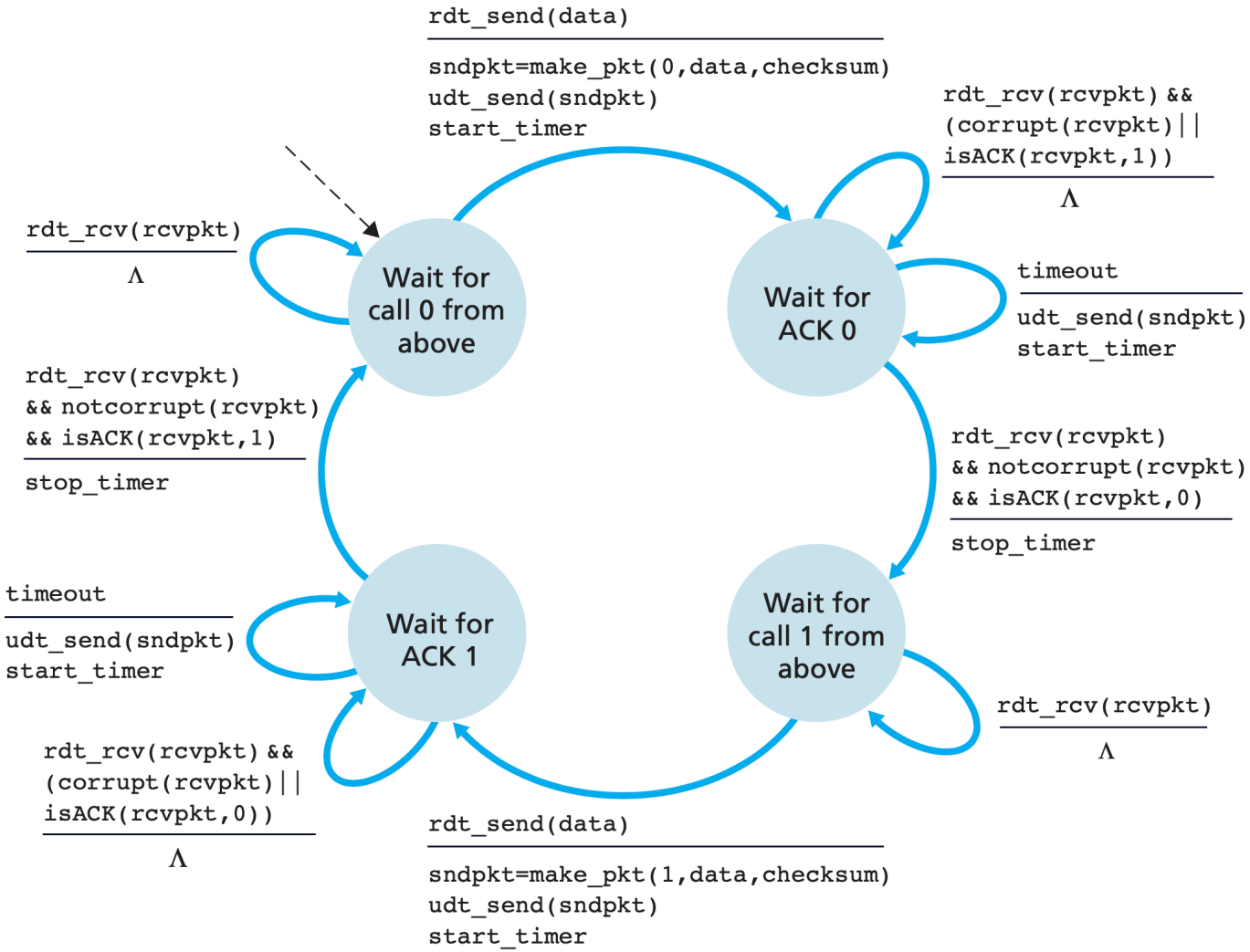
rdt2.2 - sender

Principles of Reliable Data Transfer

- Build a reliable data transfer protocol
  - 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
    - NEW PROBLEMS: how to detect packet loss and what to do when packet loss occurs (techs used by rdt2.2 will help)
    - How to detect packet loss? —— 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
    - But, how long? The sender must clearly wait at least as long as a round-trip delay between the sender and receiver (which may include buffering at intermediate routers) plus whatever amount of time is needed to process a packet at the receiver [however worst-case maximum delay is hard to estimate]
    - Retransmit with a countdown timer: (1) start the timer each time a packet (either a first-time packet or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate actions), and (3) stop the timer
    - In rdt3.0, we only need to change the sender

Principles of Reliable Data Transfer

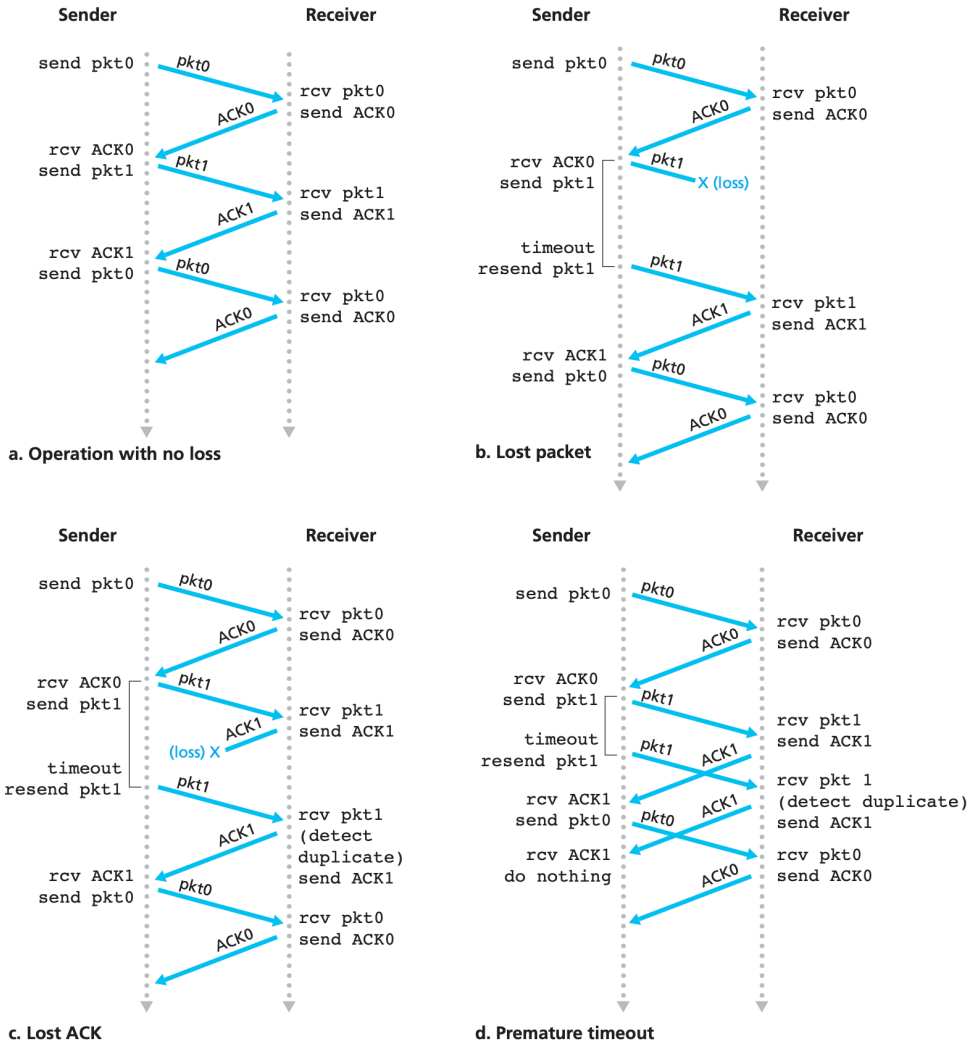
- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt3.0 (cont'd)



rdt3.0 - sender

Principles of Reliable Data Transfer

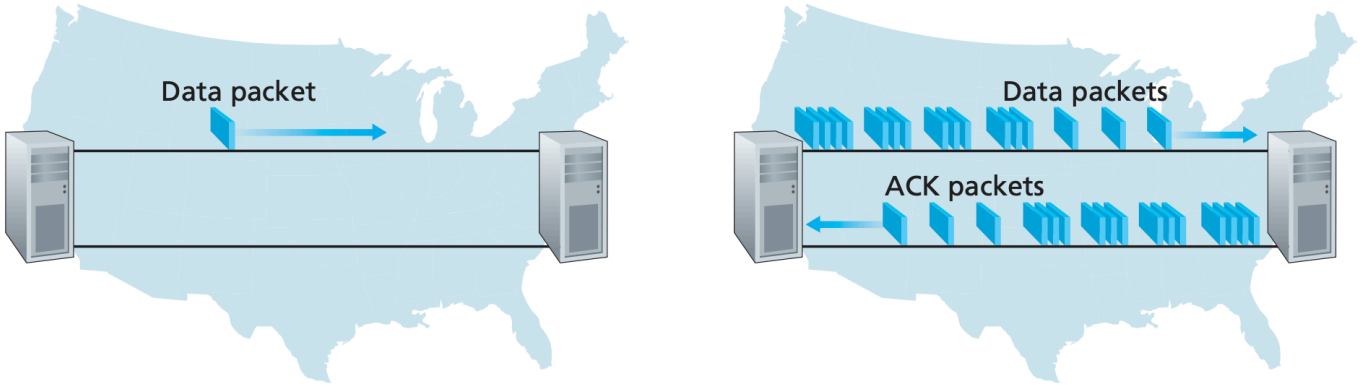
- Build a reliable data transfer protocol
  - Reliable Data Transfer over a Perfectly Reliable Channel: rdt3.0 (cont'd)



Operation of rdt3.0 – the alternating-bit protocol

Principles of Reliable Data Transfer

- Pipelined reliable data transfer protocols
  - rtd3.0 is a stop-and-wait protocol, that is, the sender will not send a new piece of data until it is sure that the receiver has correctly received the current packet
  - It surely has a significant impact on the utility of transmitter
  - Case study:
    - speed-of-light round-trip propagation delay (RTT): ~30ms
    - transmission rate R: 1Gbps, packet size L: 8000bits
    - transmission time is L/R = 8us
    - last bit of the packet emerging at the receiver at t = RTT/2 + L/R = 15.008ms
    - extremely idealy, the ACK emerges back at the sender at t = RTT + L/R = 30.008ms
    - sender utilization is only 8us/30.008ms = 0.00027! (an effective throughput of only 267kbps—even though a 1Gbps link was available)



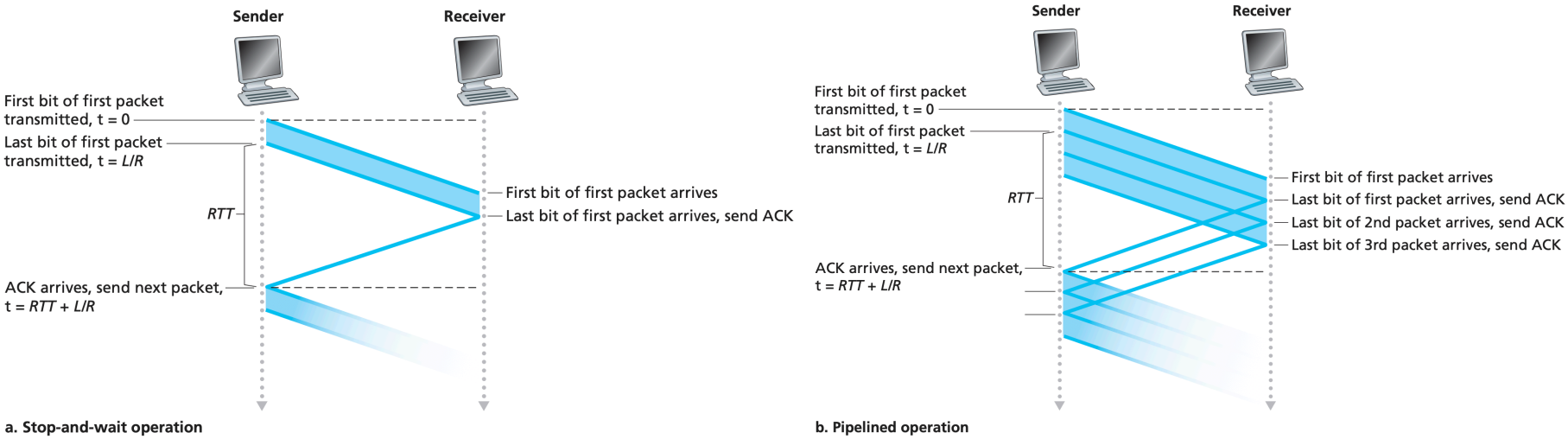
a. A stop-and-wait protocol in operation

b. A pipelined protocol in operation

Stop-and-wait versus pipelined protocol

Principles of Reliable Data Transfer

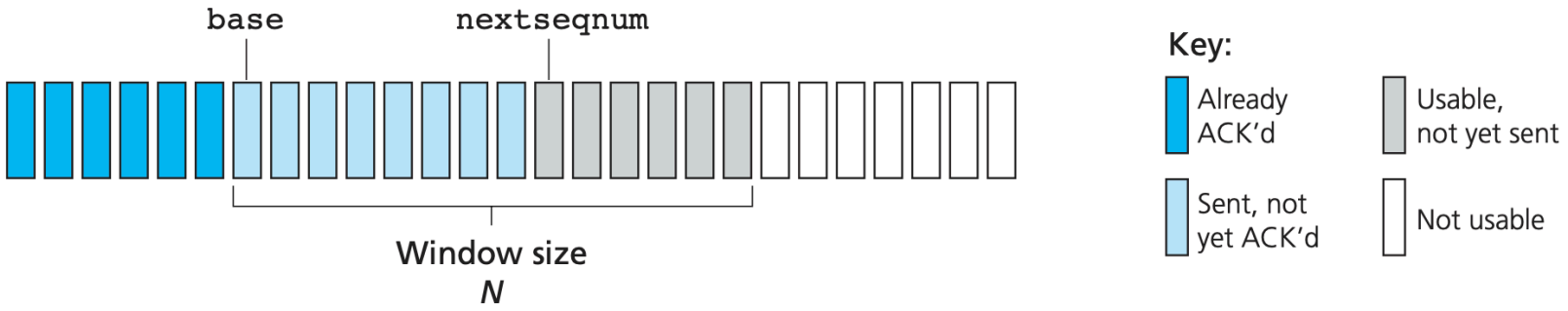
- Pipelined reliable data transfer protocols (cont'd)
  - We need **pipelining** —— Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments
    - The range of sequence numbers must be increased
    - The sender and receiver sides of the protocols may have to buffer more than one packet
    - The range of sequence numbers needed and the buffering requirements will depend on the manner in which a data transfer protocol responds to lost, corrupted, and overly delayed packets
      - Go-Back-N
      - selective repeat



Stop-and-wait versus pipelined sending

Principles of Reliable Data Transfer

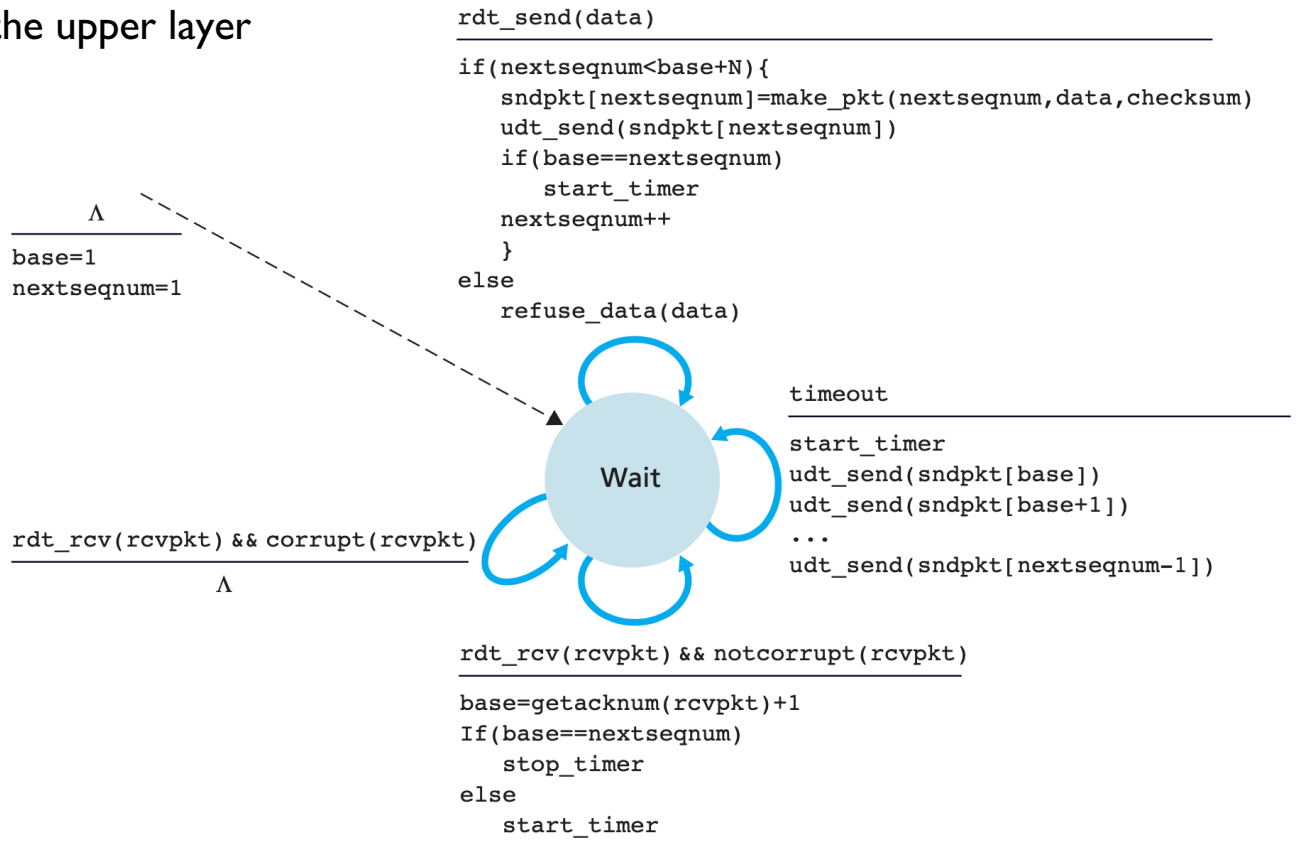
- Go-Back-N (GBN)
  - N is often referred to as the window size and the GBN protocol itself as a sliding-window protocol
  - N is not an unlimited number because of flow & congestion control
  - A packet's sequence number is carried in a fixed-length field in the packet header (k bits for $[0,2^k - 1]$)
  - `base`: the sequence number of the oldest unacknowledged packet
  - `nextseqnum`: the smallest unused sequence number



Sender's view of sequence numbers in Go-Back-N

Principles of Reliable Data Transfer

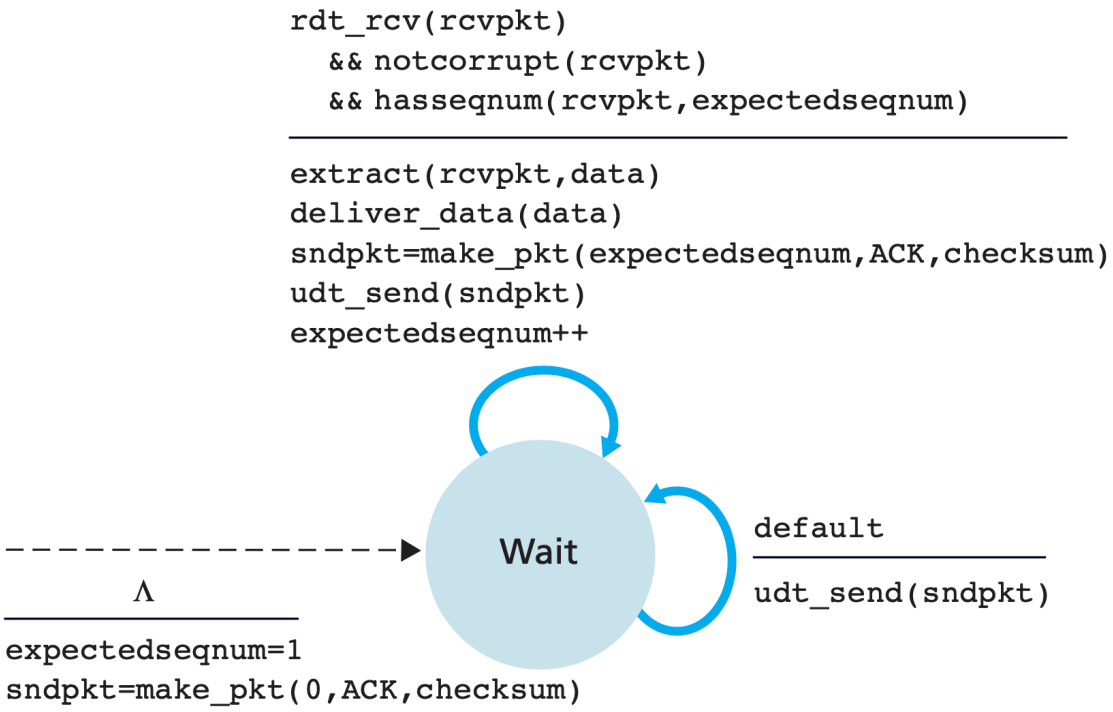
- Go-Back-N (cont'd)
  - If the window is not full , create and sent a packet
  - Cumulative acknowledgment
  - If a timeout occurs, the sender resends all packets that have been previously sent but that have not yet been acknowledged
  - If a packet with sequence number n is received correctly and is in order, the receiver sends an ACK for packet n and delivers the data portion of the packet to the upper layer



Extended FSM description of GBN sender

Principles of Reliable Data Transfer

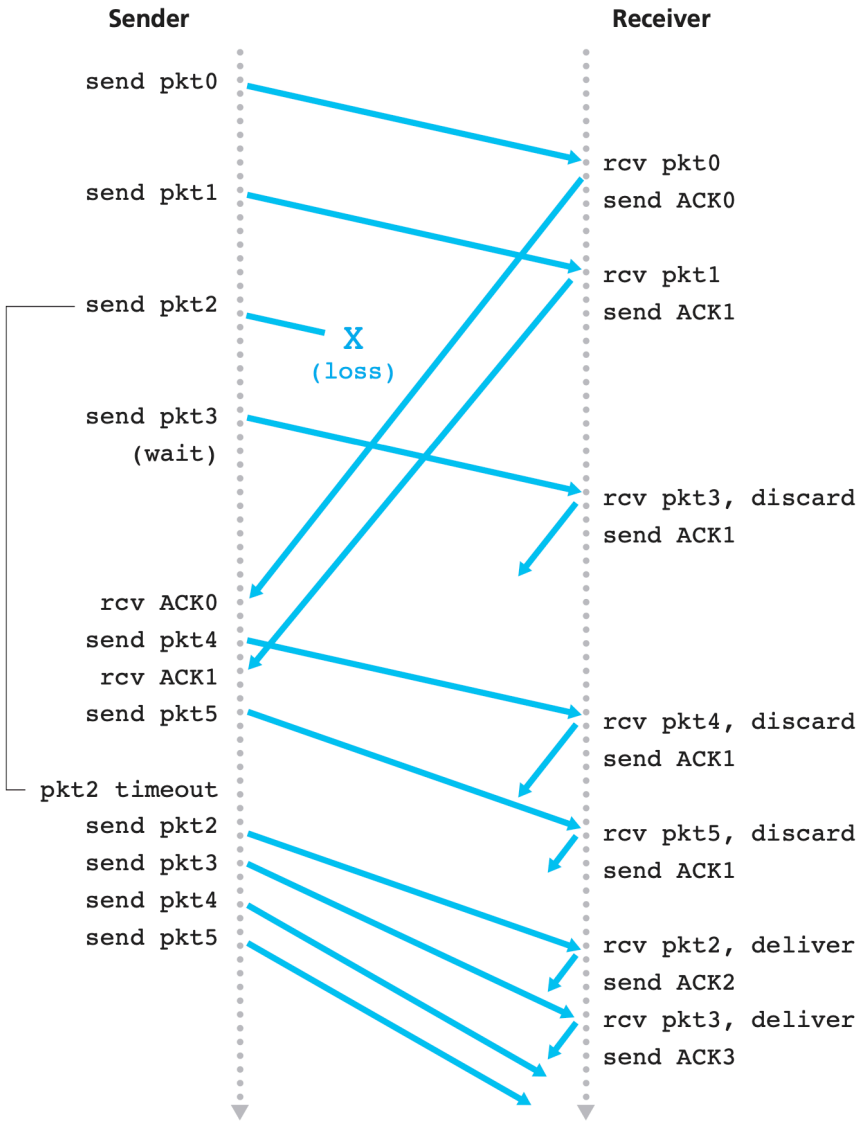
- Go-Back-N (cont'd)
  - While the sender must maintain the upper and lower bounds of its window and the position of `nextseqnum` within this window, the only piece of information the receiver need maintain is the sequence number of the next in-order packet



Extended FSM description of GBN receiver

Principles of Reliable Data Transfer

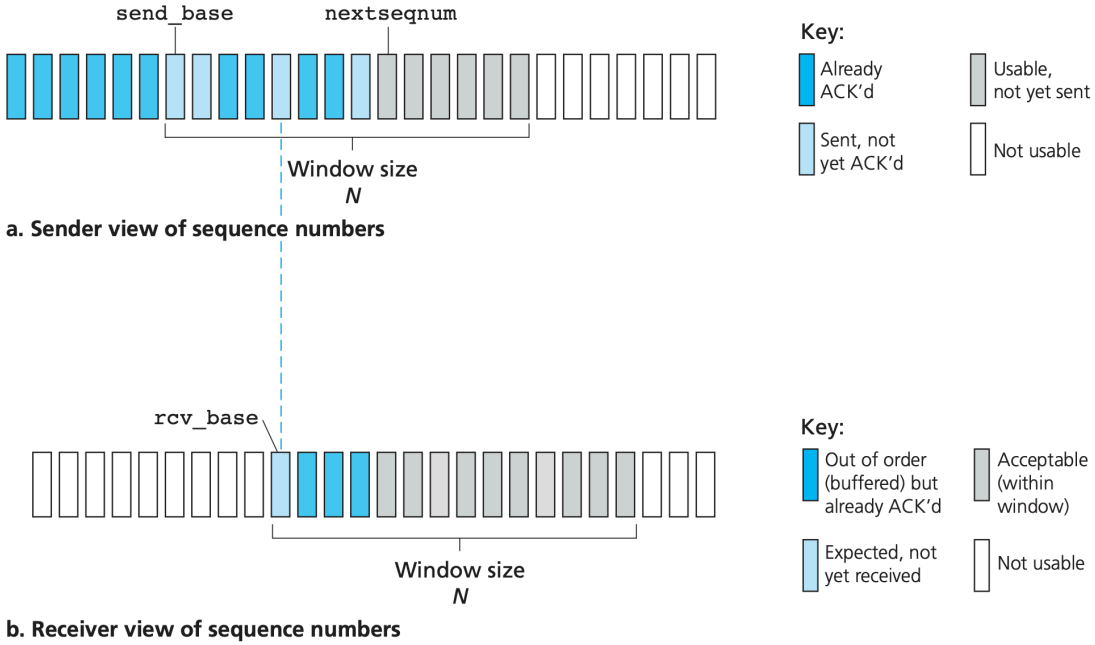
- Go-Back-N (cont'd)



Go-Back-N in operation

Principles of Reliable Data Transfer

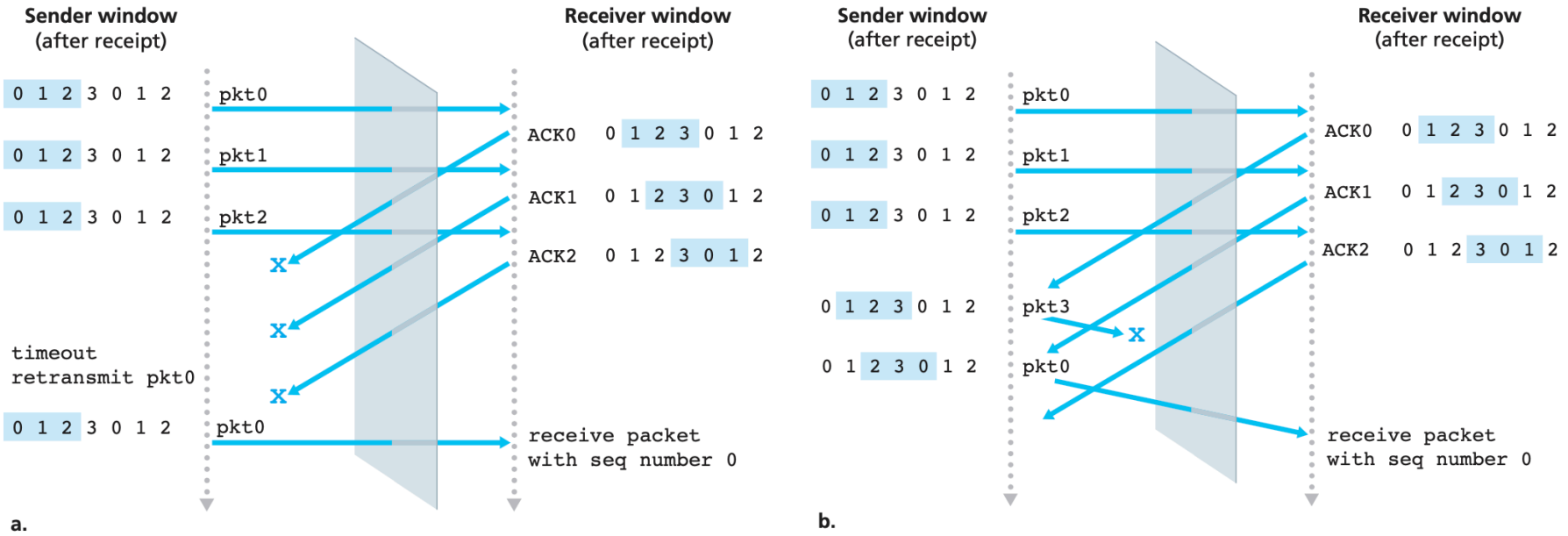
- Selective repeat
  - selective-repeat protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver
  - the receiver has to individually acknowledge correctly received packets
  - 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



Selective-repeat (SR) sender and receiver views of sequence-number space

Principles of Reliable Data Transfer

- Selective repeat (cont'd)
  - There is no way of distinguishing the retransmission of the first packet from an original transmission of the fifth packet
  - **A proper window size must be less than or equal to half the size of the sequence number space for SR protocols**



SR receiver dilemma with too-large windows: A new packet or a retransmission?

Principles of Reliable Data Transfer

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. We'll see shortly that the window size may be set on the basis of the receiver's ability to receive and buffer messages, or the level of congestion in the network, or both.

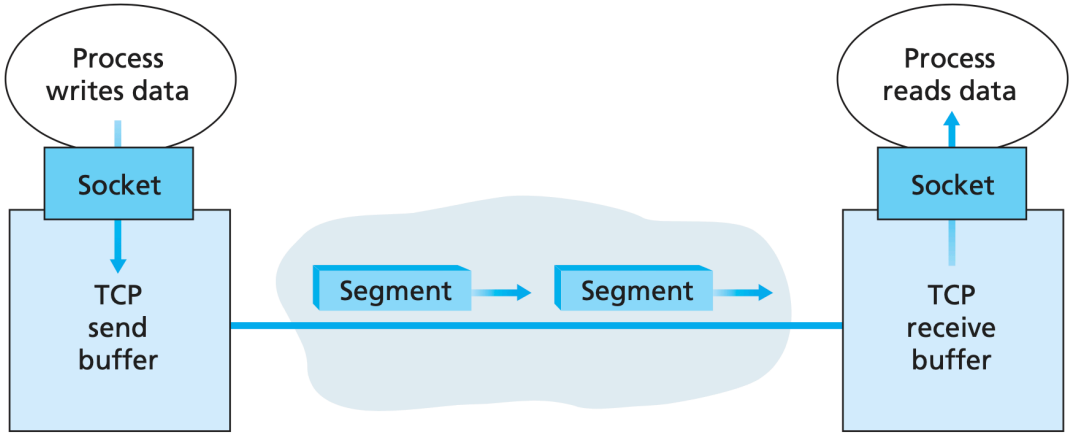
Summary of reliable data transfer mechanisms and their use

TCP

- The TCP connection
  - connection-oriented
  - full-duplex
  - three-way handshake: the client first sends a special TCP segment; the server responds with a second special TCP segment; and finally the client responds again with a third special segment

```
clientSocket.connect((serverName,serverPort))
```

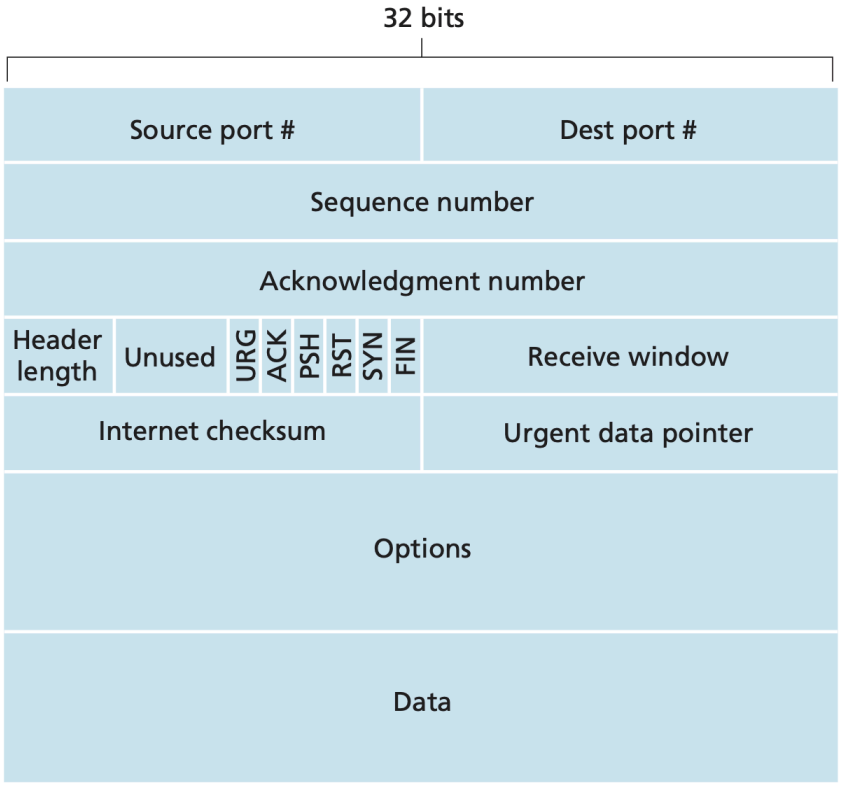
  - TCP connection consists of buffers, variables, and a socket connection from src. host to des. host
  - both Ethernet and PPP link-layer protocols have an MSS of 1,500 bytes



TCP send and receive buffers

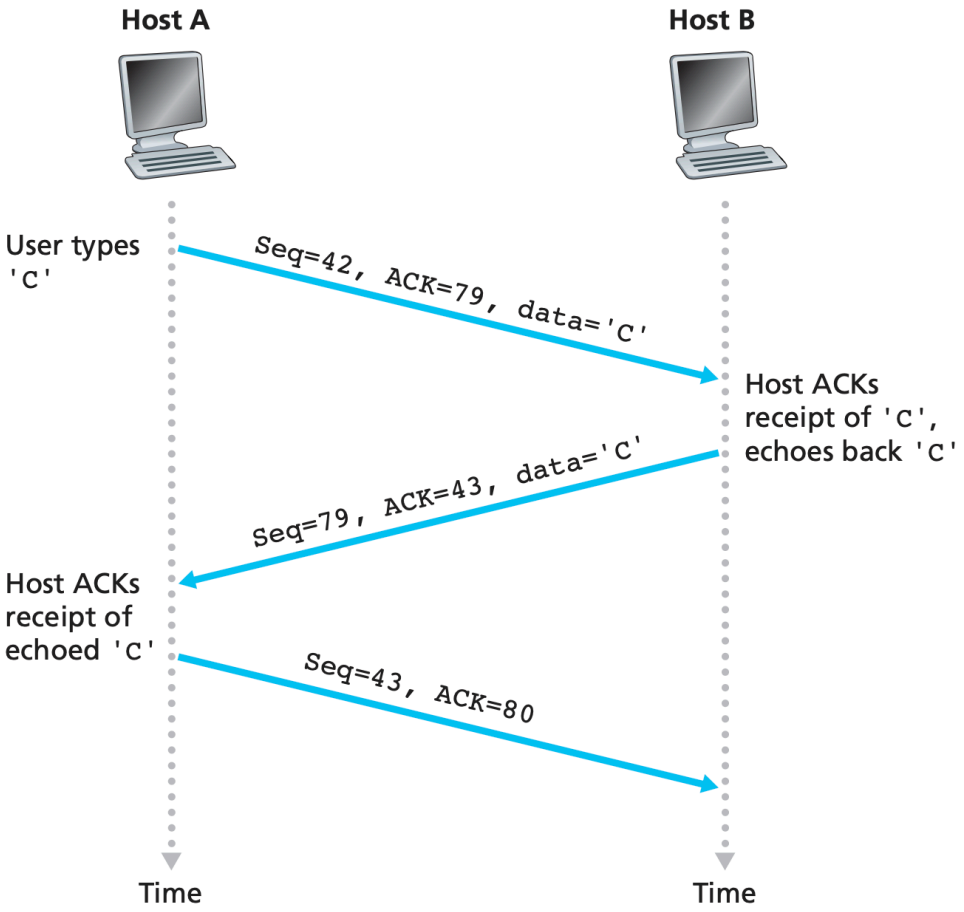
TCP

- TCP segment structure
  - **sequence number**: the byte-stream number of the *first* byte in the segment
  - **acknowledgment number**: the acknowledgment number that Host A puts in its segment is the sequence number of the next byte Host A is expecting from Host B
  - in truth, both sides of a TCP connection randomly choose an initial sequence number



TCP

- Telnet: A Case Study for Sequence and Acknowledgment Numbers
  - Telnet is a app-layer protocol runs over TCP



Sequence and acknowledgment numbers for a simple Telnet application over TCP (we suppose the starting sequence numbers are 42 and 79 for the client and server, respectively)

TCP

- Round-Trip Time Estimation and Timeout
  - Estimating the Round-Trip Time (exponential weighted moving average):

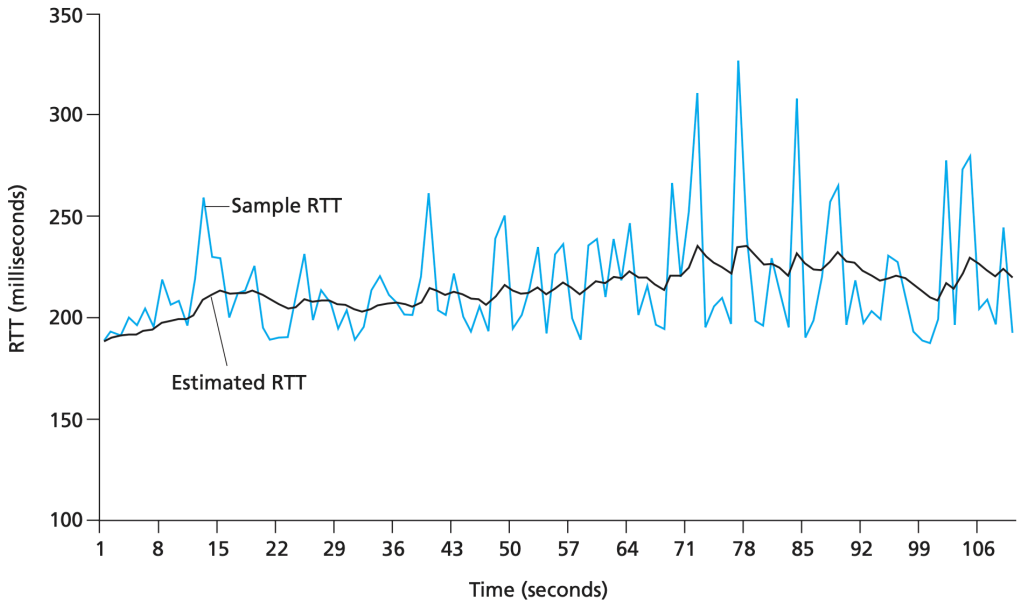
$$\texttt{EstimatedRTT} = (1 - \alpha) \cdot \texttt{EstimatedRTT} + \alpha \cdot \texttt{SampleRTT}$$

  - RTT variation:

$$\texttt{DevRTT} = (1 - \beta) \cdot \texttt{DevRTT} + \beta \cdot |\texttt{SampleRTT} - \texttt{EstimatedRTT}|$$

  - Retransmission Timeout Interval:

$$\texttt{TimeoutInterval} = \texttt{EstimatedRTT} + 4 \cdot \texttt{DevRTT}$$



RTT samples and RTT estimates

TCP

- Reliable Data Transfer – A Simplified Version
  - timer management can require considerable overhead
  - we take single-timer recommendation

```
/* Assume sender is not constrained by TCP flow or congestion control, that data from above is less
than MSS in size, and that data transfer is in one direction only. */

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

TCP

- Reliable Data Transfer – A Simplified Version (cont'd)
  - TCP looks a lot like a GBN-style protocol. But there are some striking differences between TCP and Go-Back-N —— Many TCP implementations will buffer correctly received but out-of-order segments
  - A proposed modification to TCP, the so-called selective acknowledgment [RFC 2018], allows a TCP receiver to acknowledge out-of-order segments selectively rather than just cumulatively acknowledging the last correctly received, in-order segment
  - Thus, TCP's error-recovery mechanism is probably best categorized as a hybrid of GBN and SR protocols

TCP

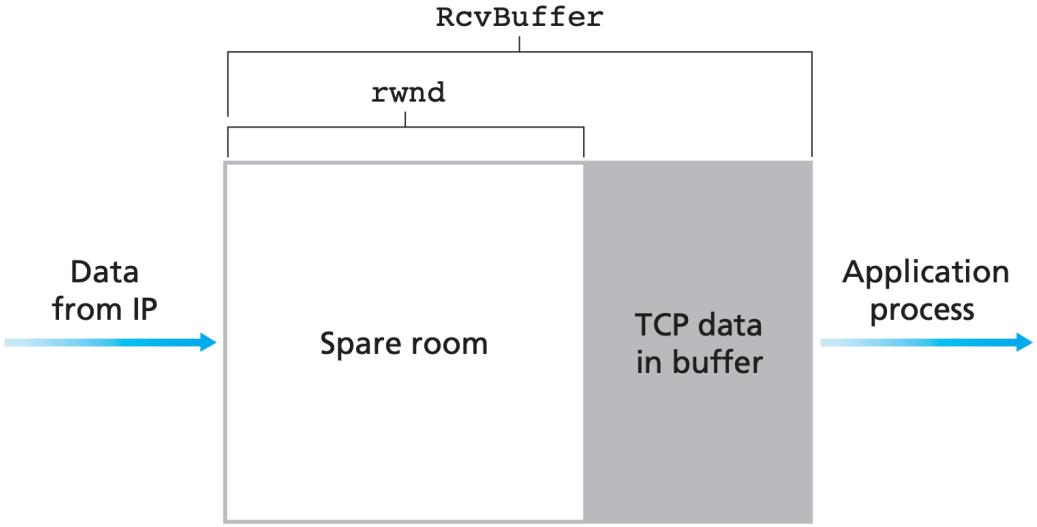
- Flow Control
  - To eliminate the possibility of the sender overflowing the receiver's buffer
  - A speed-matching service — matching the rate at which the sender is sending against the rate at which the receiving application is reading
  - We introduce flow control by assuming that *the TCP receiver discards out-of-order segments*

$$\texttt{LastByteRcvd} - \texttt{LastByteRead} \leq \texttt{RcvBuffer}$$

$$\texttt{rwnd} = \texttt{RcvBuffer} - [\texttt{LastByteRcvd} - \texttt{LastByteRead}]$$

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{rwnd}$$

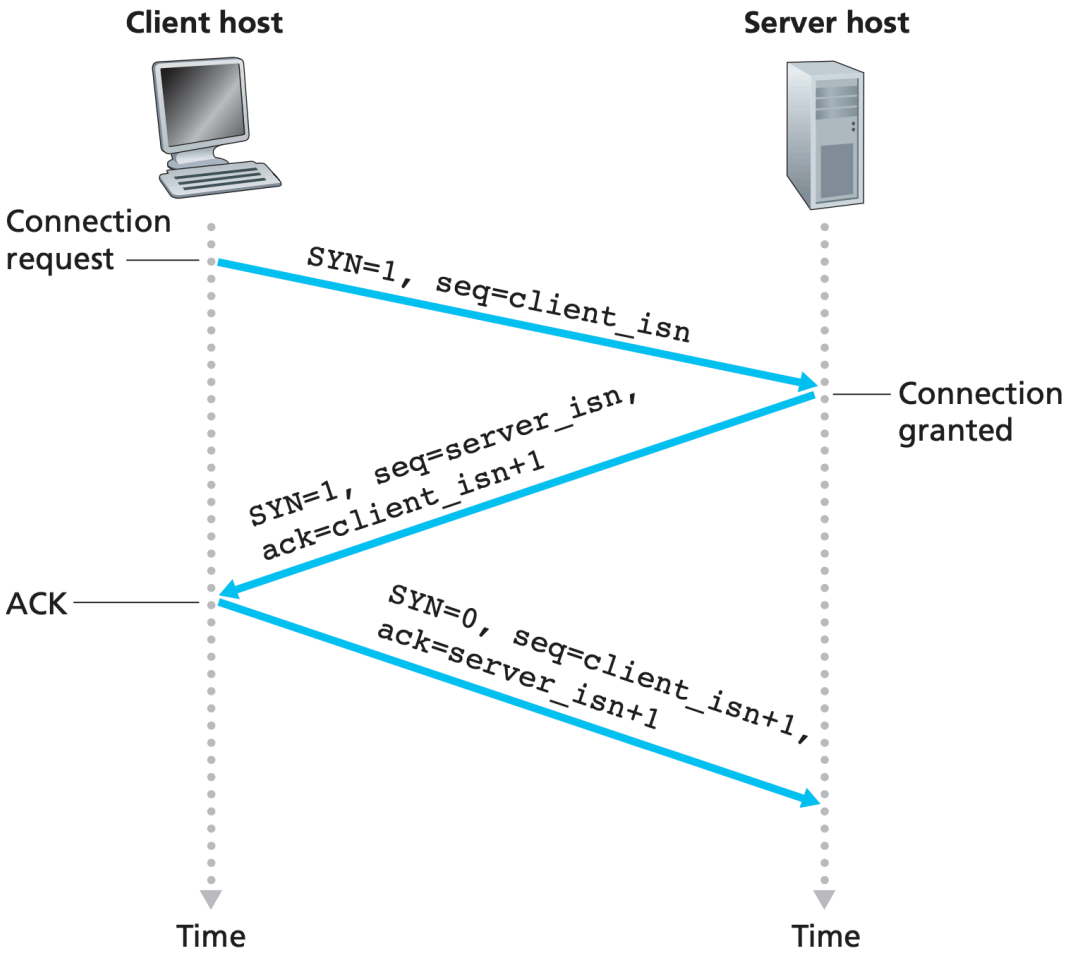
  - The sender is required to send segment with one data byte when the receiver's window size is zero (for info. update)



The receive window (`rwnd`) and the receive buffer (`RcvBuffer`)

TCP

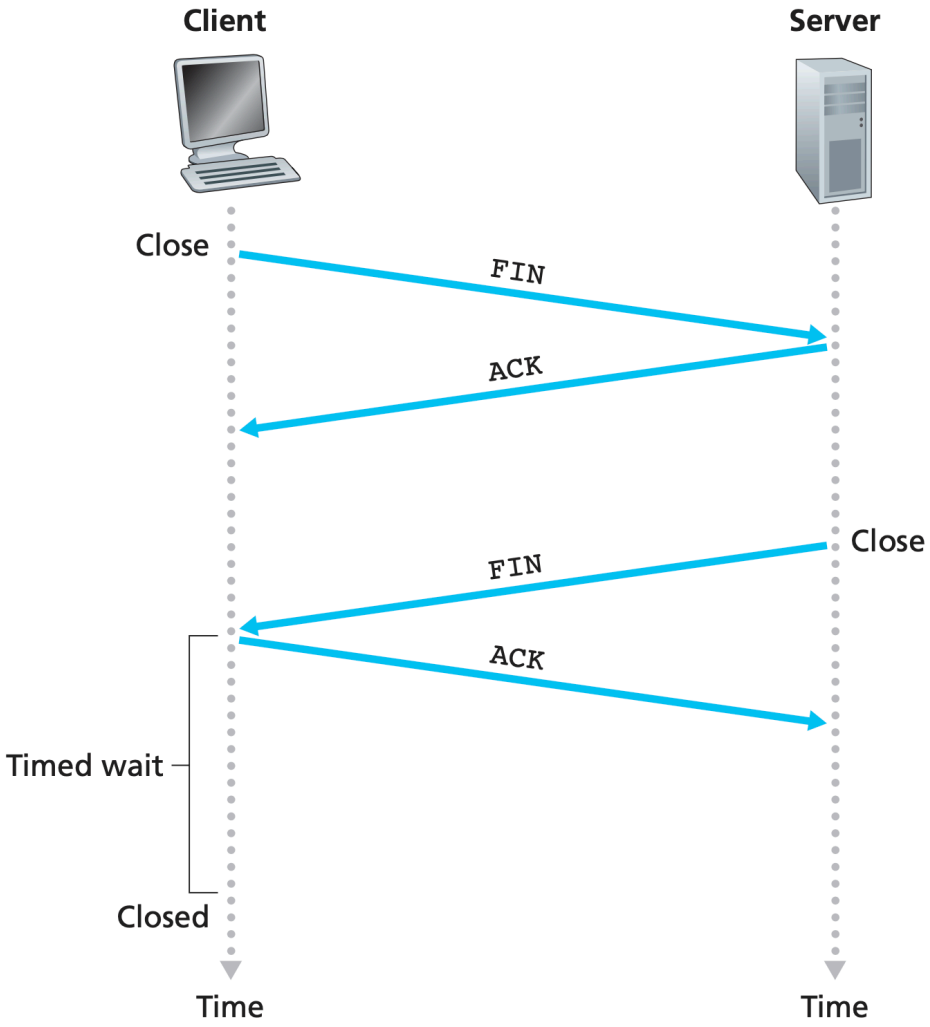
- TCP Connection Management
  - **three-way handshake**: pay attention to the SYN bit
  - SYN-flood attack & SYN cookies



TCP three-way handshake: segment exchange

TCP

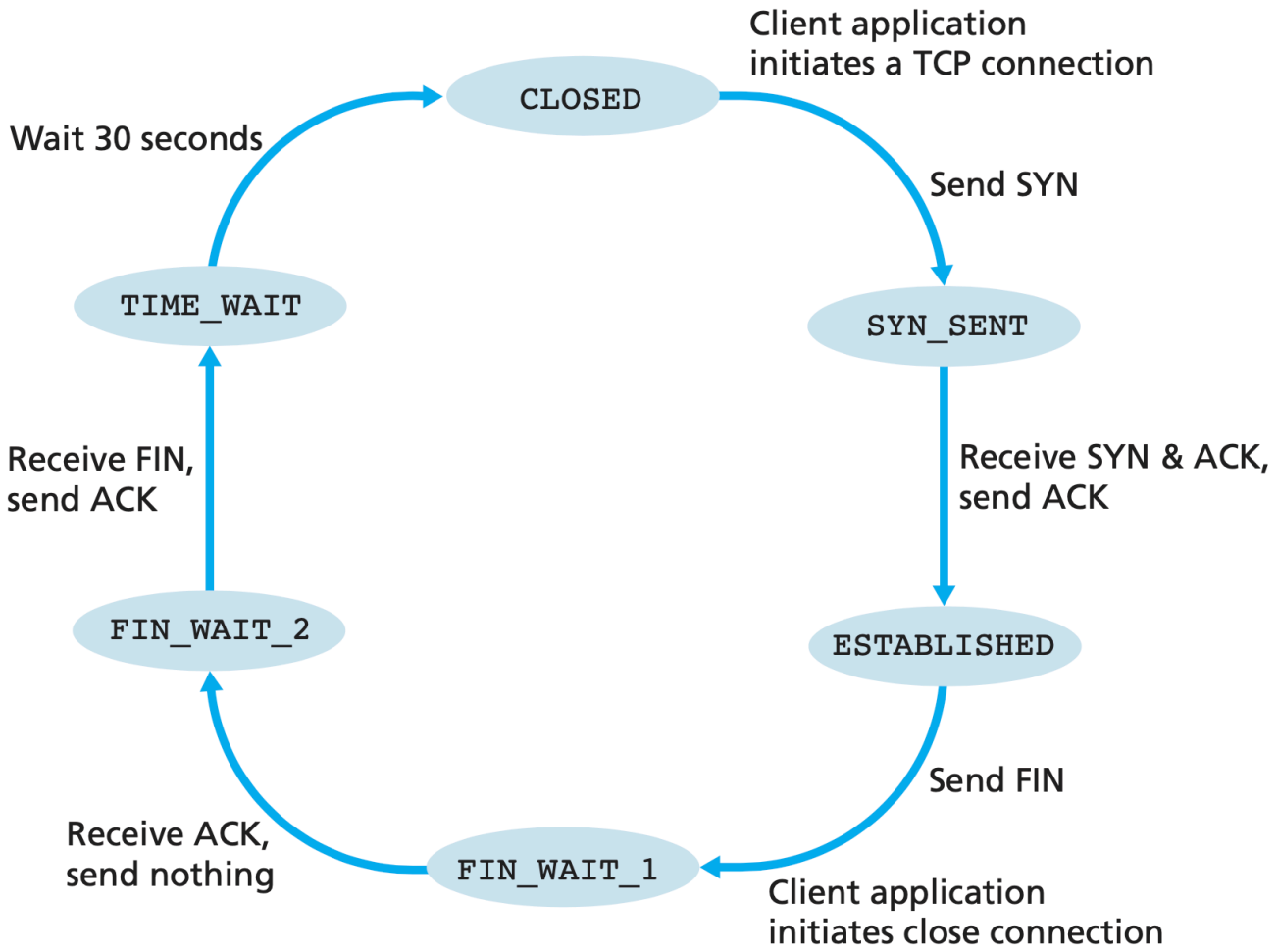
- TCP Connection Management (cont'd)
  - **closing a TCP connection**: pay attention to the FIN bit



Closing a TCP connection

TCP

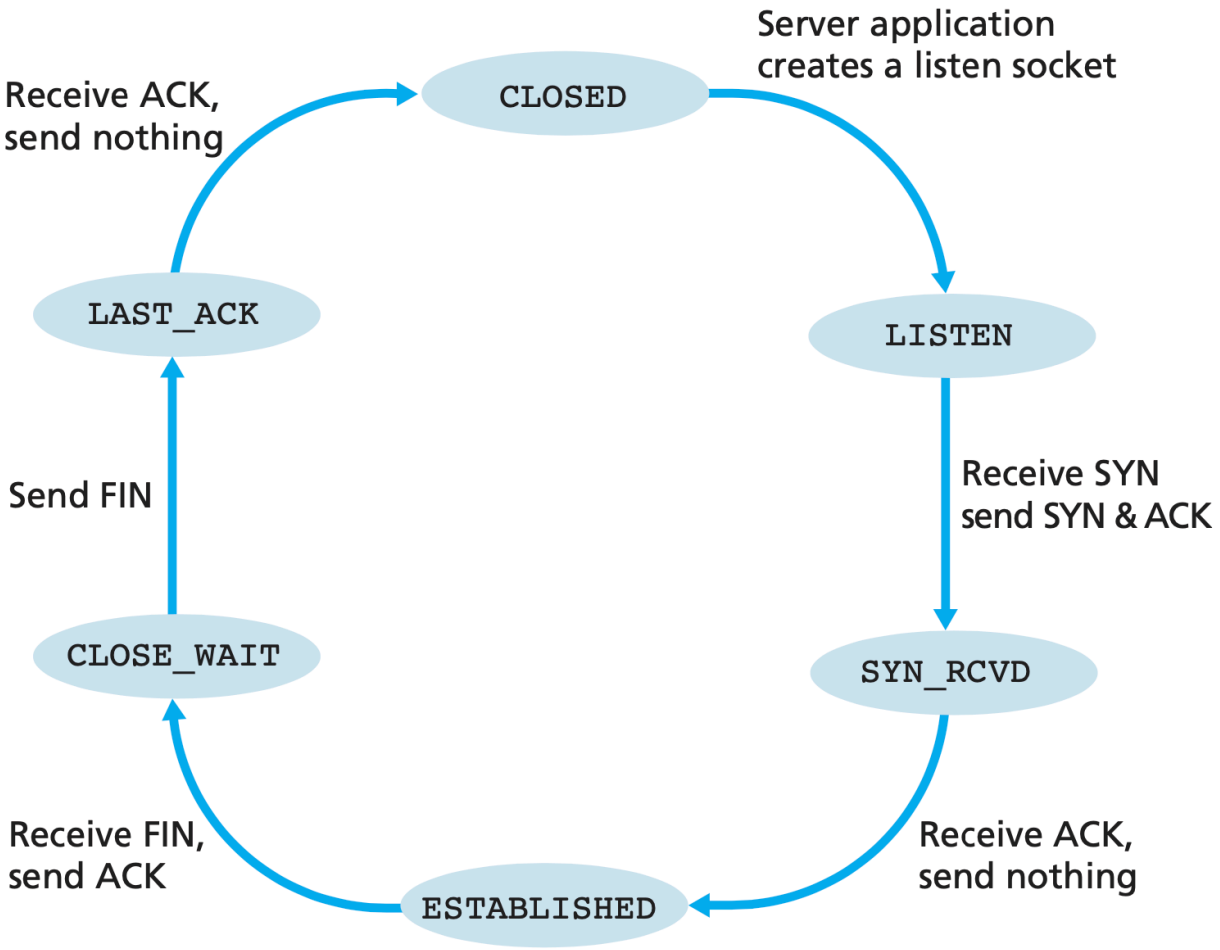
- TCP Connection Management (cont'd)



A typical sequence of TCP states visited by a client TCP

TCP

- TCP Connection Management (cont'd)



A typical sequence of TCP states visited by a server-side TCP

TCP

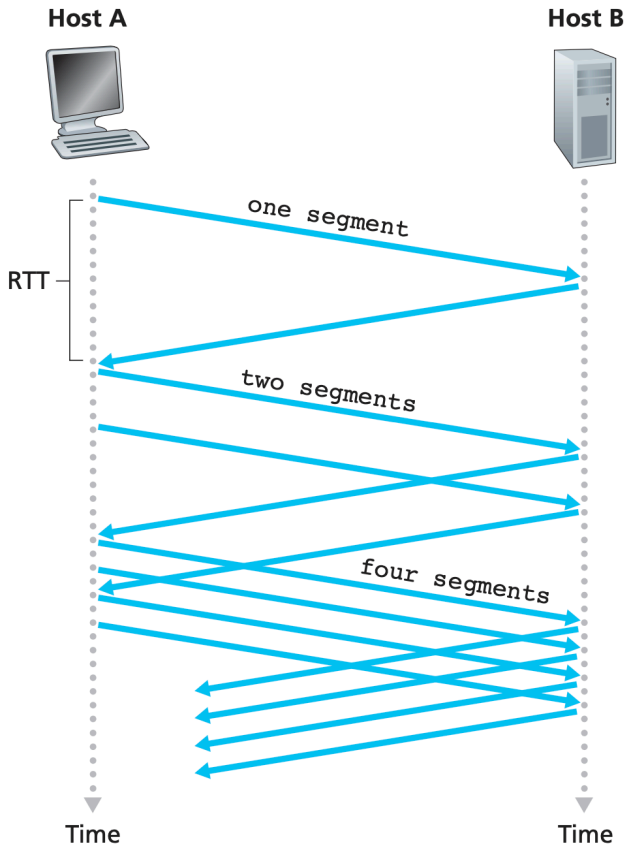
- TCP Congestion Control
  - The TCP congestion-control mechanism operating at the sender keeps track of an additional variable, the **congestion window**

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \min\{\texttt{cwnd}, \texttt{rwnd}\}$$

  - Given the mechanism of adjusting the value of `cwnd` to control the sending rate, the critical question remains: How should a TCP sender determine the rate at which it should send?
    - A lost segment implies congestion, and hence, the TCP sender's rate should be *decreased* when a segment is lost
    - An acknowledged segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be *increased* when an ACK arrives for a previously unacknowledged segment
    - Bandwidth probing: The TCP sender thus increases its transmission rate to probe for the rate that at which congestion onset begins, backs off from that rate, and then to begins probing again to see if the congestion onset rate has changed
  - TCP congestion-control algorithm
    - slow start
    - congestion avoidance
    - fast recovery

TCP

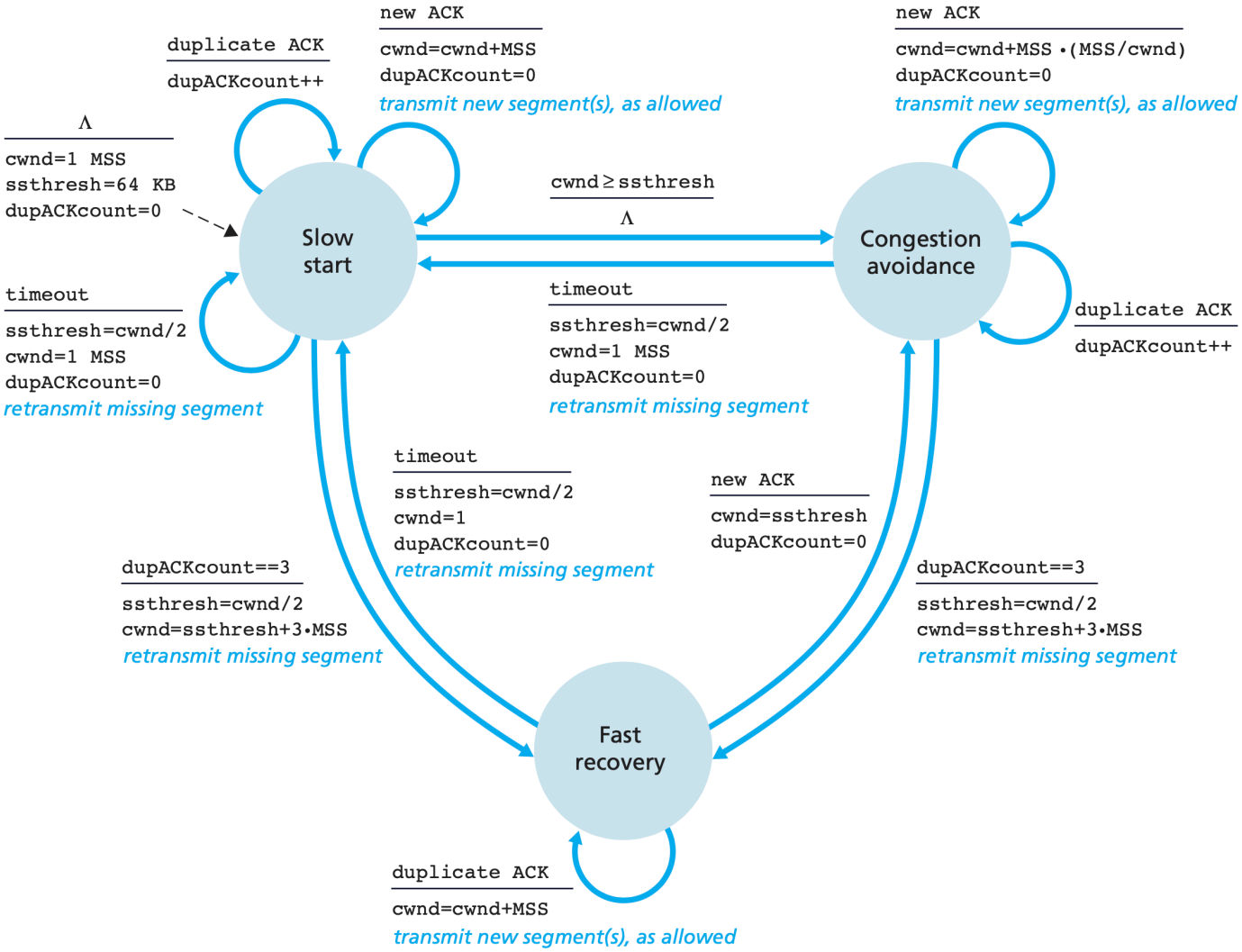
- TCP Congestion Control (cont'd)
  - slow start
    - the value of `cwnd` begins at 1 MSS and increases by 1 MSS every time a transmitted segment is first acknowledged (starts slow but grows exponentially)



TCP slow start

TCP

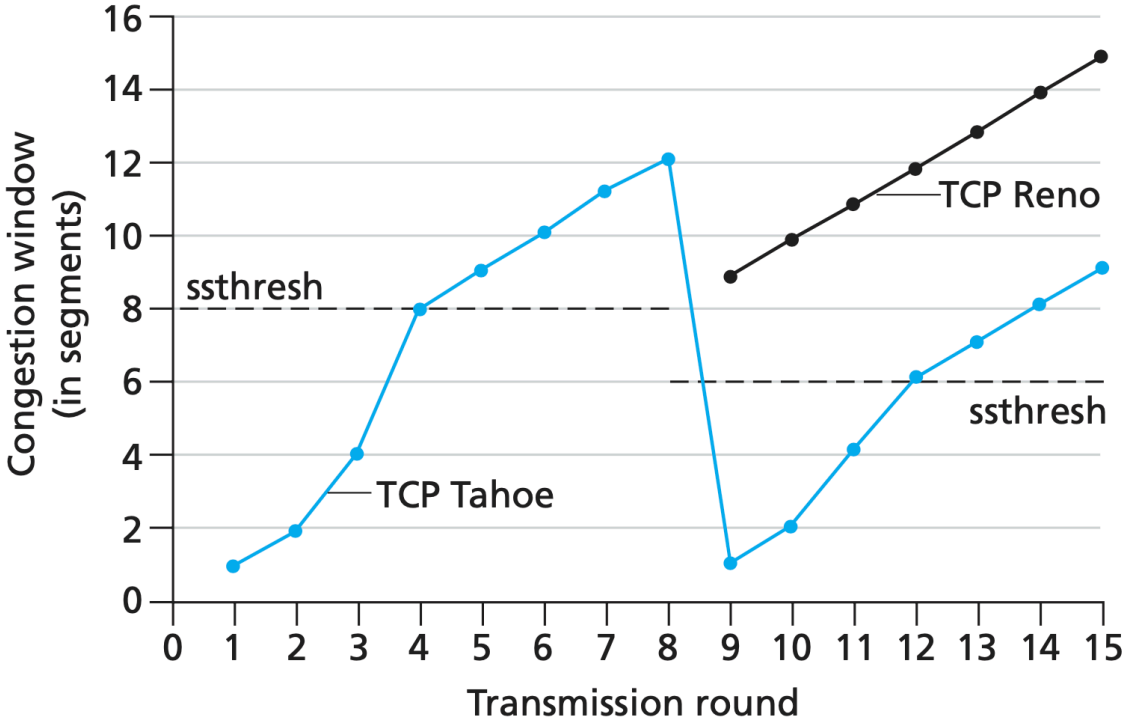
- TCP Congestion Control (cont'd)
  - slow start, congestion avoidance, and fast recovery



FSM description of TCP congestion control

TCP

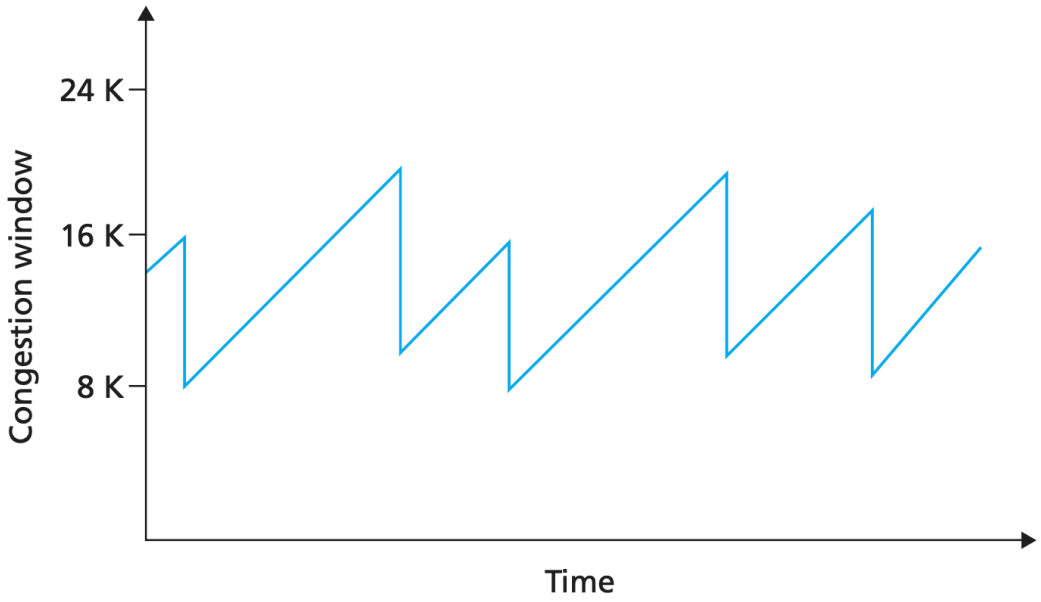
- TCP Congestion Control (cont'd)
  - TCP Tahoe → TCP Reno (incorporated fast recovery)



Evolution of TCP's congestion window (Tahoe and Reno)

TCP

- TCP Congestion Control (cont'd)
  - Ignoring the initial slow-start period when a connection begins and assuming that losses are indicated by triple duplicate ACKs rather than timeouts, TCP's congestion control consists of linear (additive) increase in `cwnd` of 1 MSS per RTT and then a halving (multiplicative decrease) of `cwnd` on a triple duplicate-ACK event. For this reason, TCP congestion control is often referred to as an **additive-increase, multiplicative- decrease (AIMD)** form of congestion control
  - Theoretical analyses showed that TCP's congestion-control algorithm serves as a distributed asynchronous-optimization algorithm that results in several important aspects of user and network performance being simultaneously optimized



Additive-increase, multiplicative-decrease congestion control

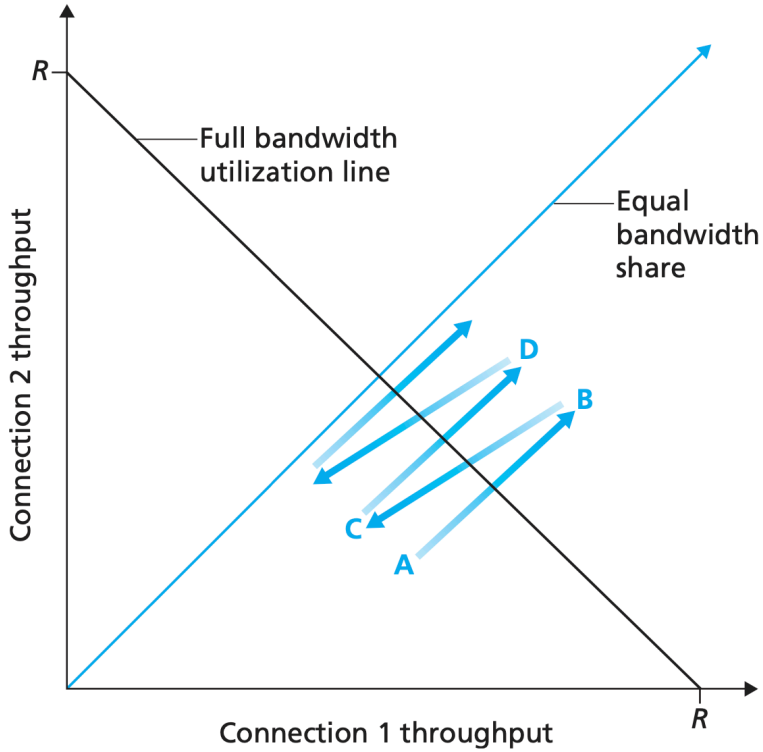
TCP

- TCP Congestion Control (cont'd)
  - Macroscopic Description of TCP Throughput:

$$\text{average throughput of a connection} = \frac{0.75 \cdot W}{RTT}$$

  - TCP Over High-Bandwidth Paths:

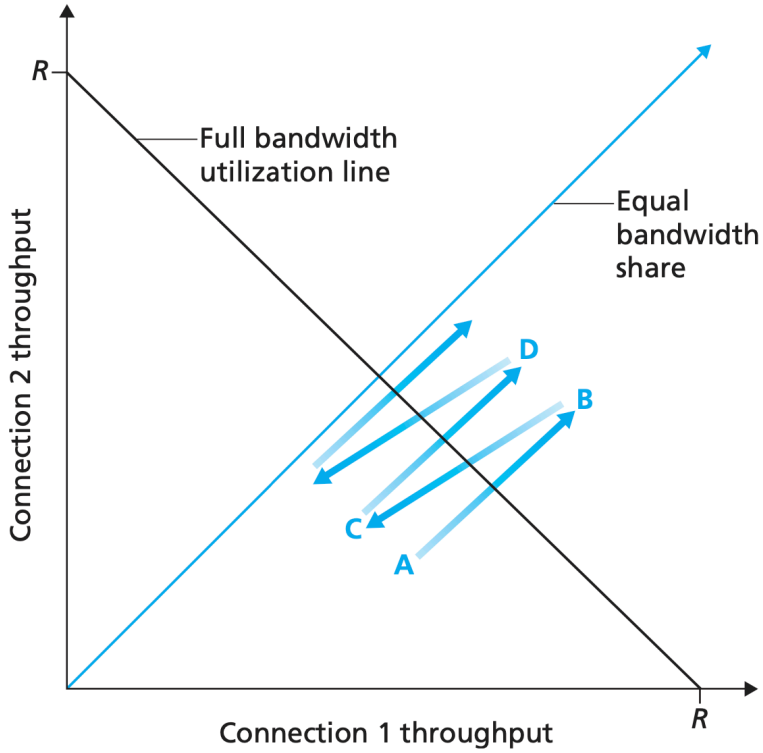
$$\text{average throughput of a connection} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$



Throughput realized by TCP connections 1 and 2

TCP

- TCP Congestion Control (cont'd)
  - Fairness:
    - Because TCP congestion control will decrease its transmission rate in the face of increasing congestion (loss), while UDP sources need not, it is possible for UDP sources to crowd out TCP traffic
    - When an application uses multiple parallel connections, it gets a larger fraction of the bandwidth in a congested link



Throughput realized by TCP connections 1 and 2