Computer Networks

Chapter 3 – Transport Layer

Edition8-1

Topics

- Transport layer services:
  - Multiplexing, Demultiplexing
  - Reliable Data Transfer
  - Flow Control
  - Congestion Control
- Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control
  - TCP Cubic
- QUIC protocol: Transport layer's functions in application layer
  - http/3
- Our discussion of network edge is complete

Contents

3.1 Introduction and Transport-Layer Services

- Transport layer has critical role of providing communication services directly to application processes running on different hosts

- Transport layer extends network layer's

"delivery service between two end systems"

To

"delivery service between two application-layer processes running on end systems"

3.1 Introduction and Transport-Layer Services

- Network-layer protocol provides logical communication between hosts

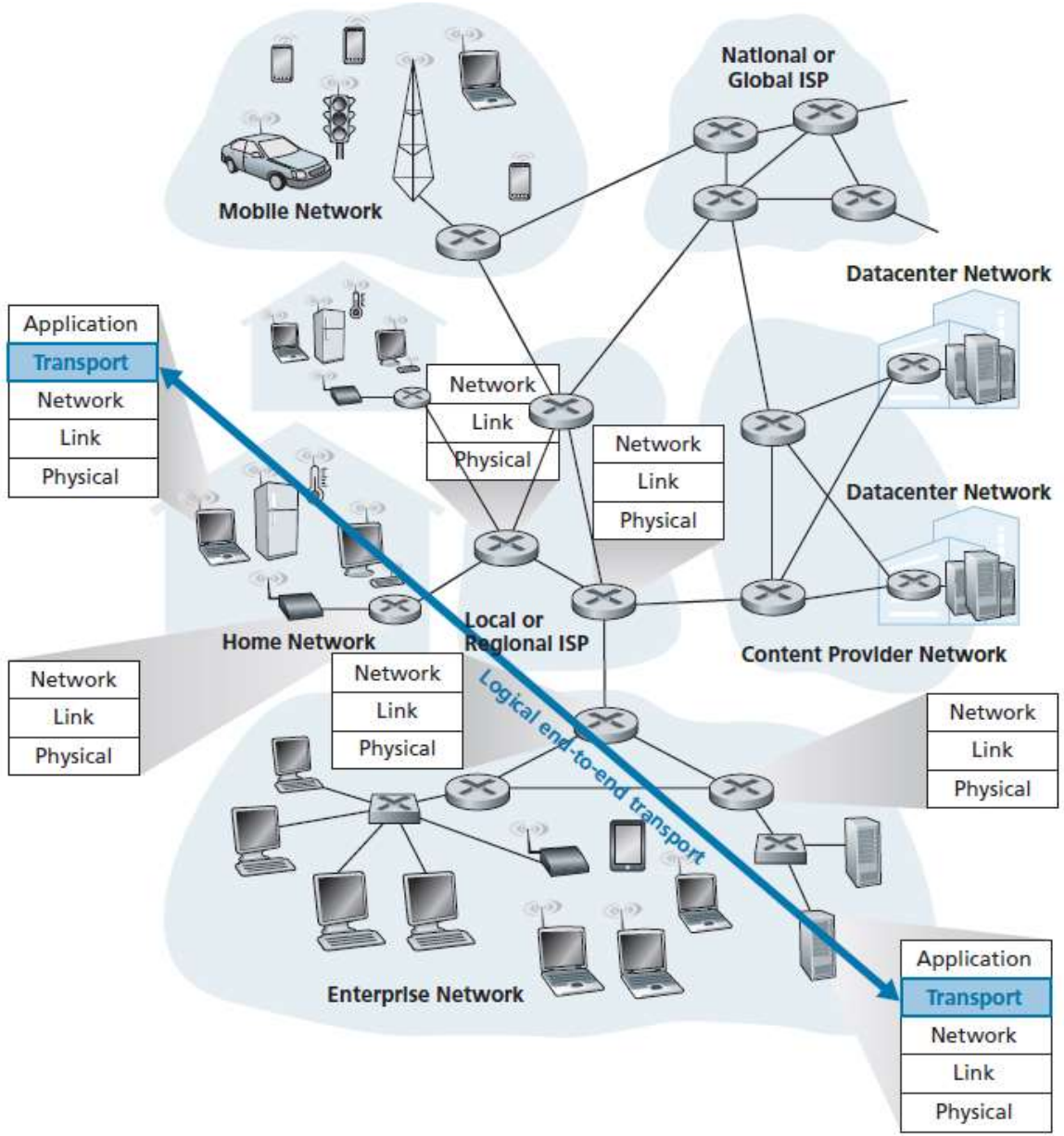
- Transport layer use logical communication provided by network layer to send messages to each other, free from worry of details of computer network used to carry these messages

So:

- A transport-layer protocol provides logical communication between application processes running on different hosts

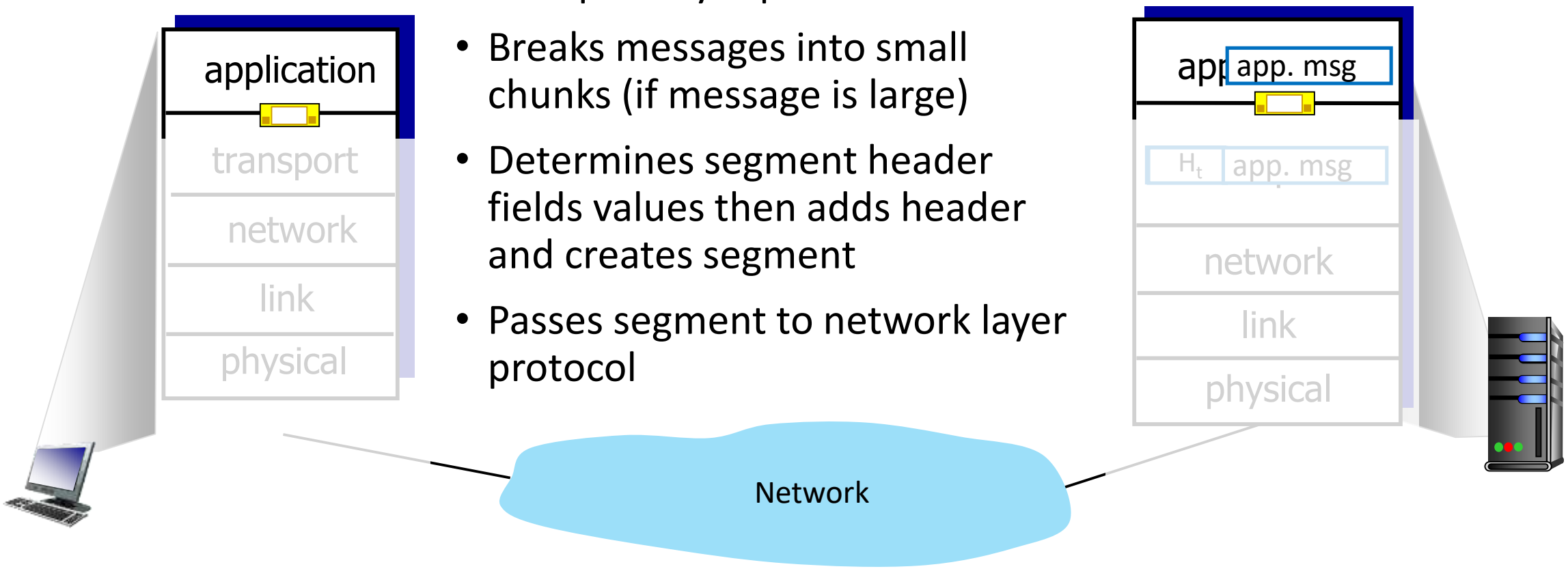
Figure 3.1

- In sending host: Transport layer breaks application-layer messages into small chunks and adding a header to each chunk to create a transport-layer packets (segments), then passes segment to network layer
- In receiving host: It examine header and passes data to receiving process



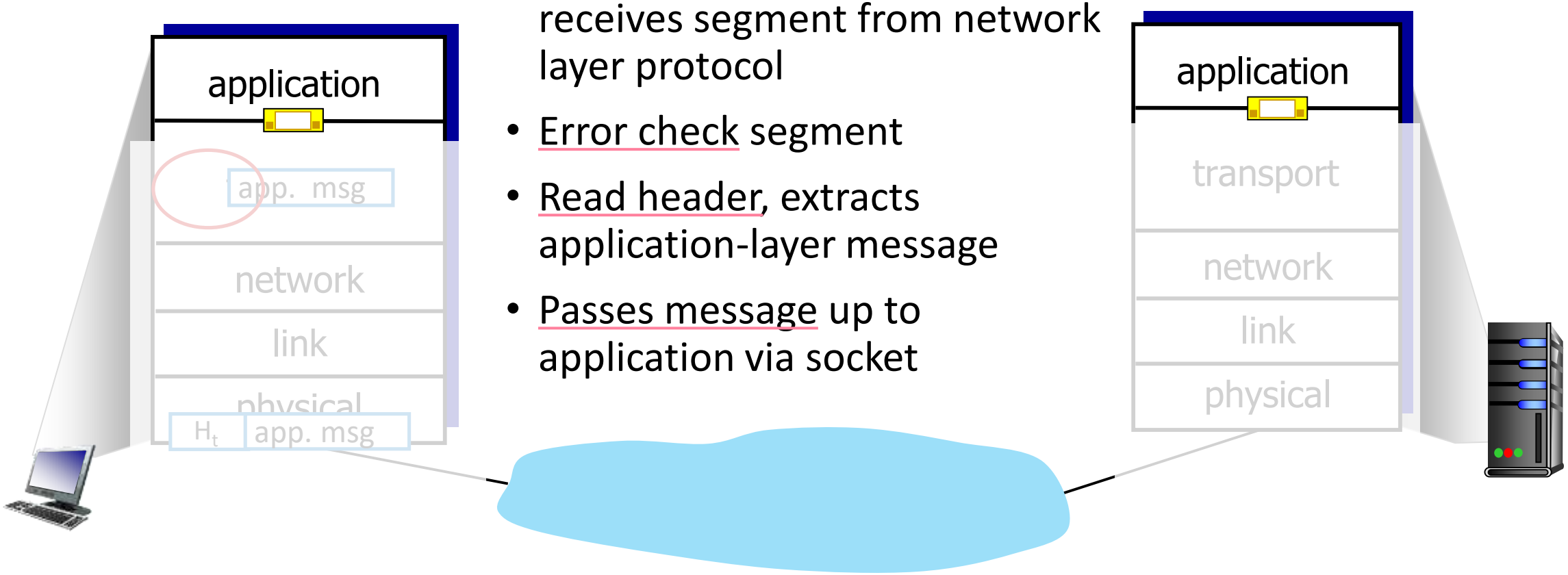
Transport Layer Actions: Sender

- App message is passed to transport layer protocol
- Breaks messages into small chunks (if message is large)
- Determines segment header fields values then adds header and creates segment
- Passes segment to network layer protocol



Transport Layer Actions: Receiver

- Transport layer protocol receives segment from network layer protocol
- Error check segment
- Read header, extracts application-layer message
- Passes message up to application via socket



3.1.1 Relationship Between Transport and Network Layers

- Transport-layer protocols work in end systems to serve Apps as a message passing mean
- Within an end system, a transport protocol moves messages from application processes to network layer and vice versa, but it doesn't have any say about how messages are moved within network core
- Network layers (in routers) move packet from source host to destination host
- Transport protocol services are constrained by service model of network-layer
  - If network-layer protocol cannot provide delay or bandwidth guarantees for transport layer, then transport-layer protocol cannot provide delay or bandwidth guarantees for application messages sent between processes
  - Certain services *can* be offered by a transport protocol even when underlying network protocol doesn't offer corresponding service at network layer
    - For example: reliable data transfer service to an application even when network protocol is unreliable

3.1.2 Overview of Transport Layer in Internet

- UDP (User Datagram Protocol):
  - provides an unreliable, connectionless service to application
  - no-frills extension of "best-effort" IP
- TCP (Transmission Control Protocol):
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- services not available:
  - delay guarantees
  - bandwidth guarantees
  - security

Contents

3.2 Multiplexing and Demultiplexing

- Multiplexing and demultiplexing extend host-to-host delivery service provided by network layer to a process-to-process delivery service for applications running on hosts
- One instance of TCP/UDP provides services for all applications processes running on a host
- Transport layer (at destination host) receives segments from network layer, and delivers data in segments to socket of appropriate application process
- Multiplexing: gathering data at source host from different sockets, creating segments, and passing segments to network layer
- Demultiplexing: delivering data in a transport-layer segment to correct socket of destination process

Socket identifier

- Transport-layer mux/demux requires:
  - sockets have unique identifiers in a host
  - each segment have special fields that indicate socket to which segment is to be delivered
- Special fields: source port number field and destination port number field. Port number is a 16-bit number (1 to 65535)
- Rang from 1 to 1023 are called well-known port numbers and are reserved for use by well-known application protocols such as:
  - HTTP (80), HTTPS(43), FTP (20, 21), SMTP (25), Time protocol (37), DNS (53), POP3(110), SNMP(161), DHCP-server(67), DHCP-client(68), …

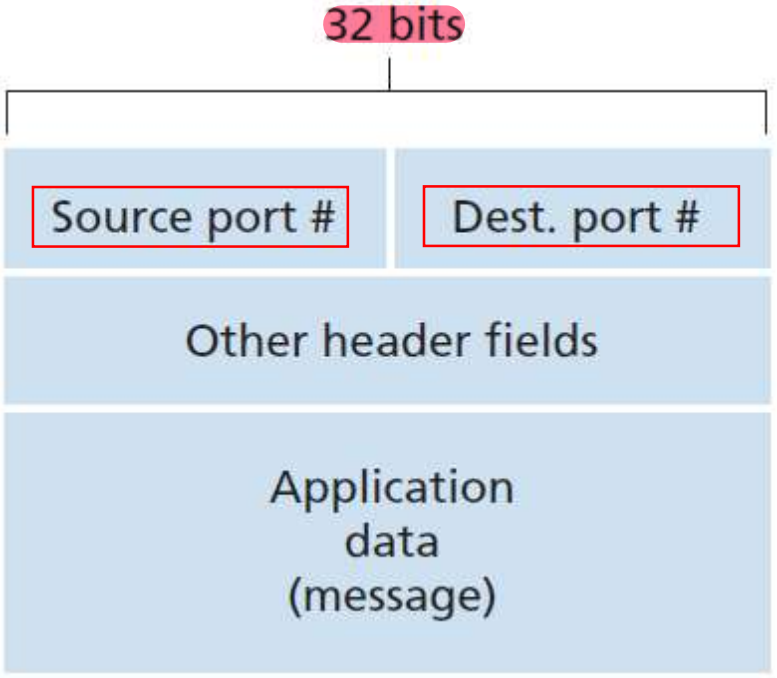


Figure 3.3 Source and destination port-number fields in a transport-layer segment

Connectionless Mux/Demux

- Python program running in a host can create a UDP socket with

```
cs = socket(AF_INET, SOCK_DGRAM)
```

- When a UDP socket is created in this manner, transport layer automatically assigns a port number (in range 1024 to 65535) to socket
- Alternatively, we can add a line into our Python program after we create socket to associate a specific port number (say, 19157) to this UDP socket via socket bind() method:

```
cs.bind(('', 19157))
```

Accept datagram from any IP address)

- Typically, client side of application lets transport layer automatically (and transparently) assign port number, whereas server side of application assigns a specific port number

Example: Inversion of source destination ports

- In UDPServer.py, server uses recvfrom() method to extract clientside (source) port number from segment it receives from client
- Server then sends a new segment to client, with extracted source port number serving as destination port number in this new segment

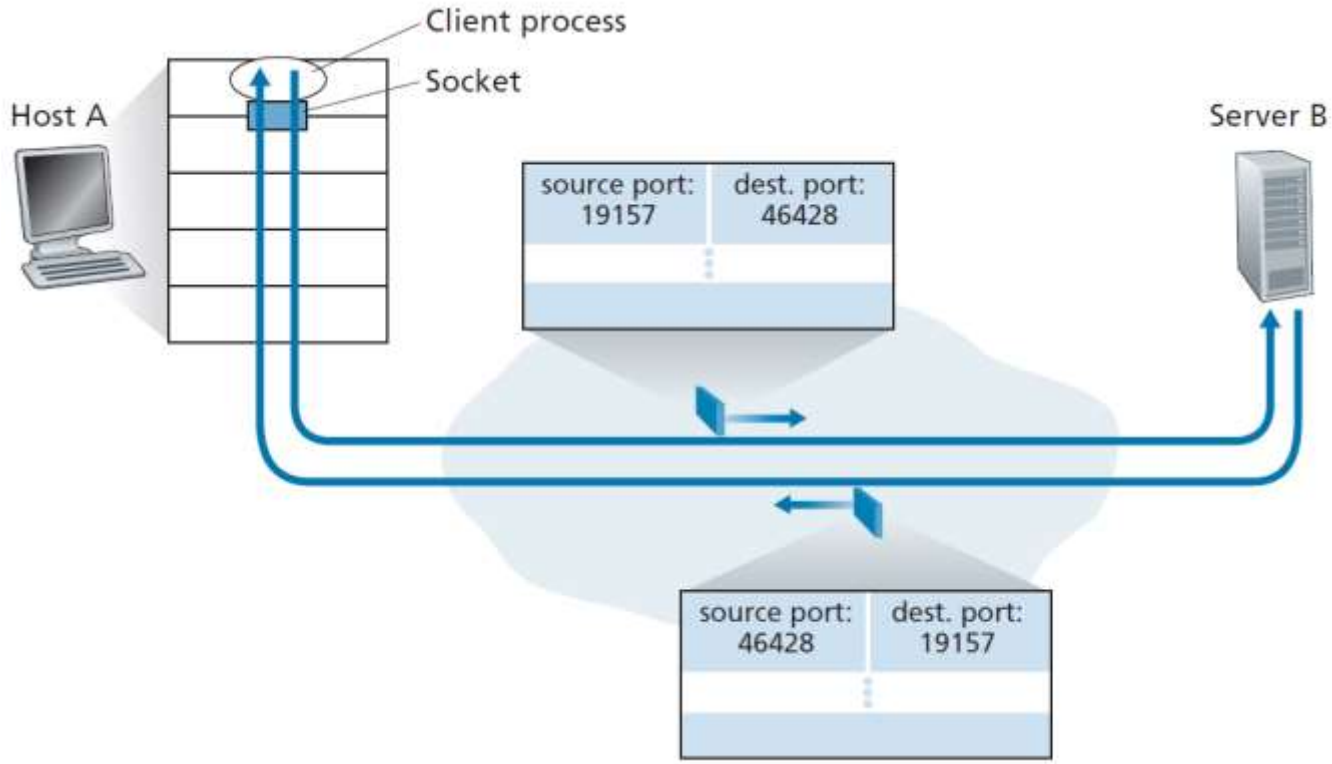
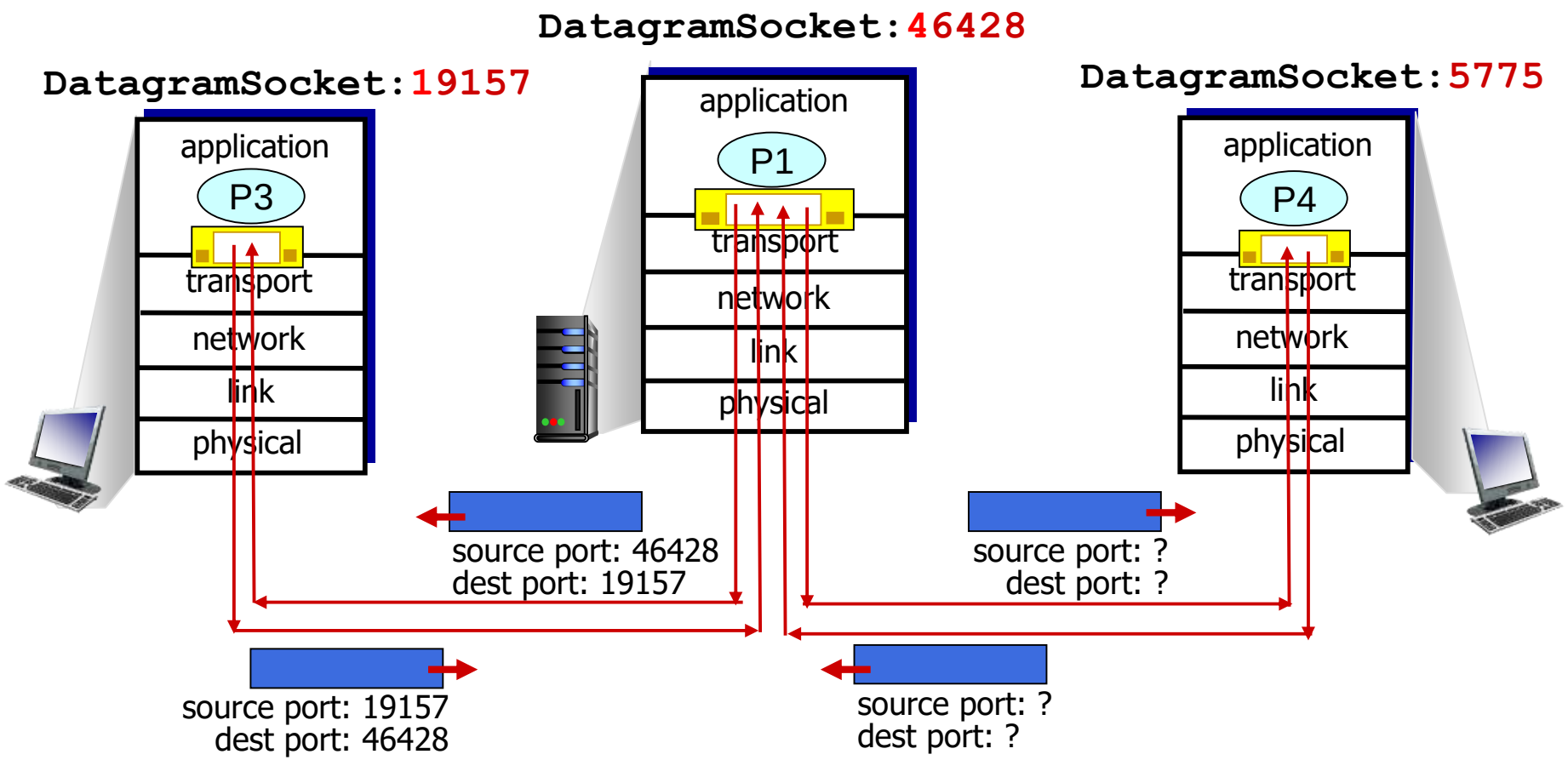


Figure 3.4 The inversion of source and destination port numbers

Connectionless demultiplexing: an example



Connection-Oriented Mux/Demux

- 2 types of TCP socket in server side:
  - Welcoming socket: Identified by 2-tuple
    - Server IP address
    - Server port number
  - Connection socket: identified by 4-tuple
    - Server IP address
    - Server process port number
    - Client IP address
    - Client process port number

Connection-Oriented Mux/Demux

- TCP receiver in server uses all four values (4-tuple) to direct segment to appropriate socket (server process)
- Server may support many simultaneous TCP connection sockets:
  - Each connection socket identified by its own 4-tuple
  - Each connection socket associated with a different connecting client
  - Two arriving TCP segments with different source IP addresses or source port numbers will be directed to two different sockets

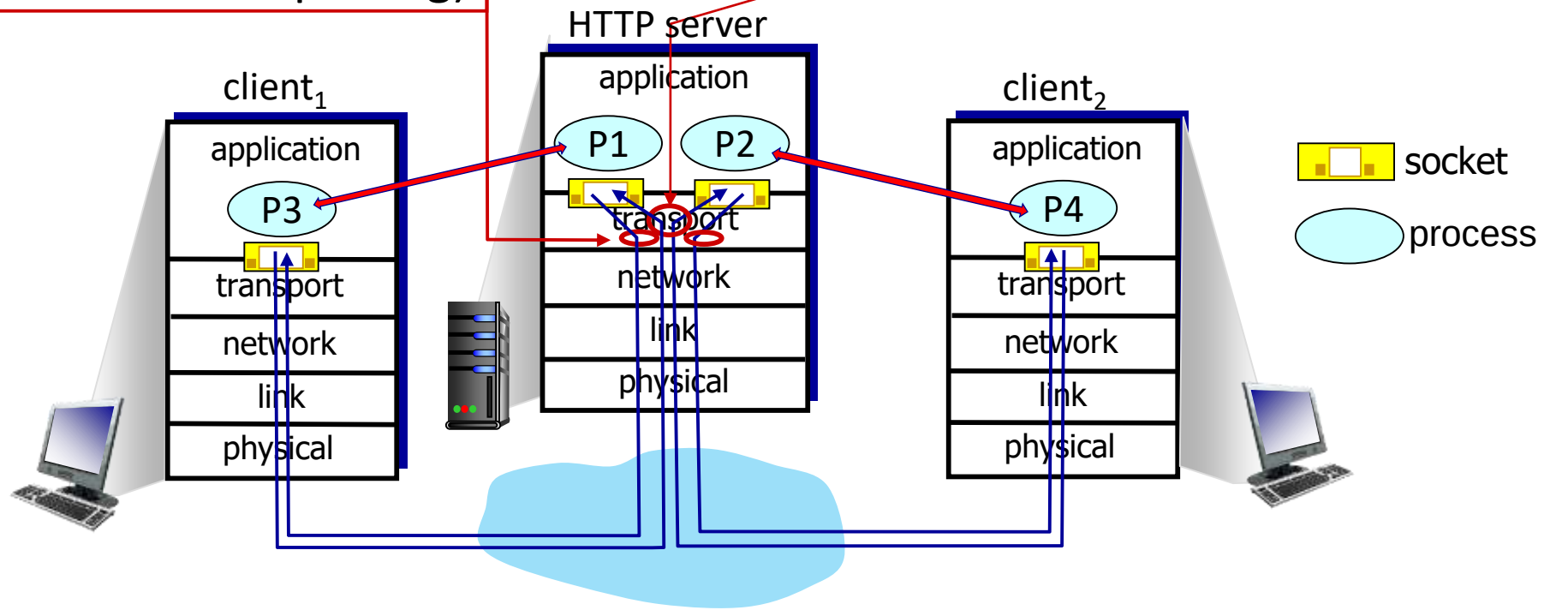
Example

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:

use header info to deliver received segments to correct socket



Example

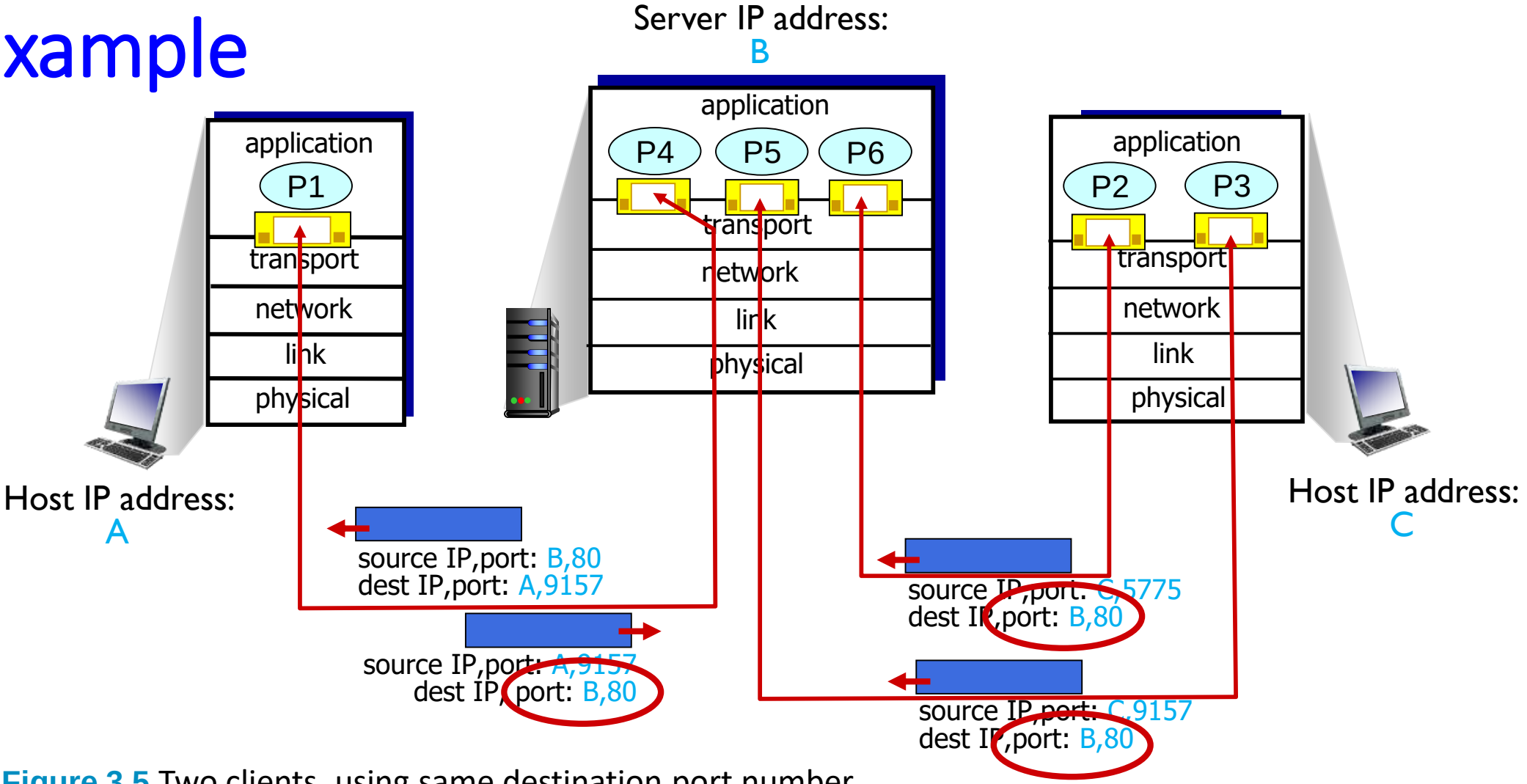


Figure 3.5 Two clients, using same destination port number (80) to communicate with same Web server application

Web Servers: Multiprocess vs Multithread

- Figure 3.5 shows a Web server that spawns a new process for each connection
- Each of processes has its own connection socket
- 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
- A thread can be viewed as a sub-process (or child)
- For such a server, at any given time there may be many connection sockets (with different identifiers) attached to same process

Multithread

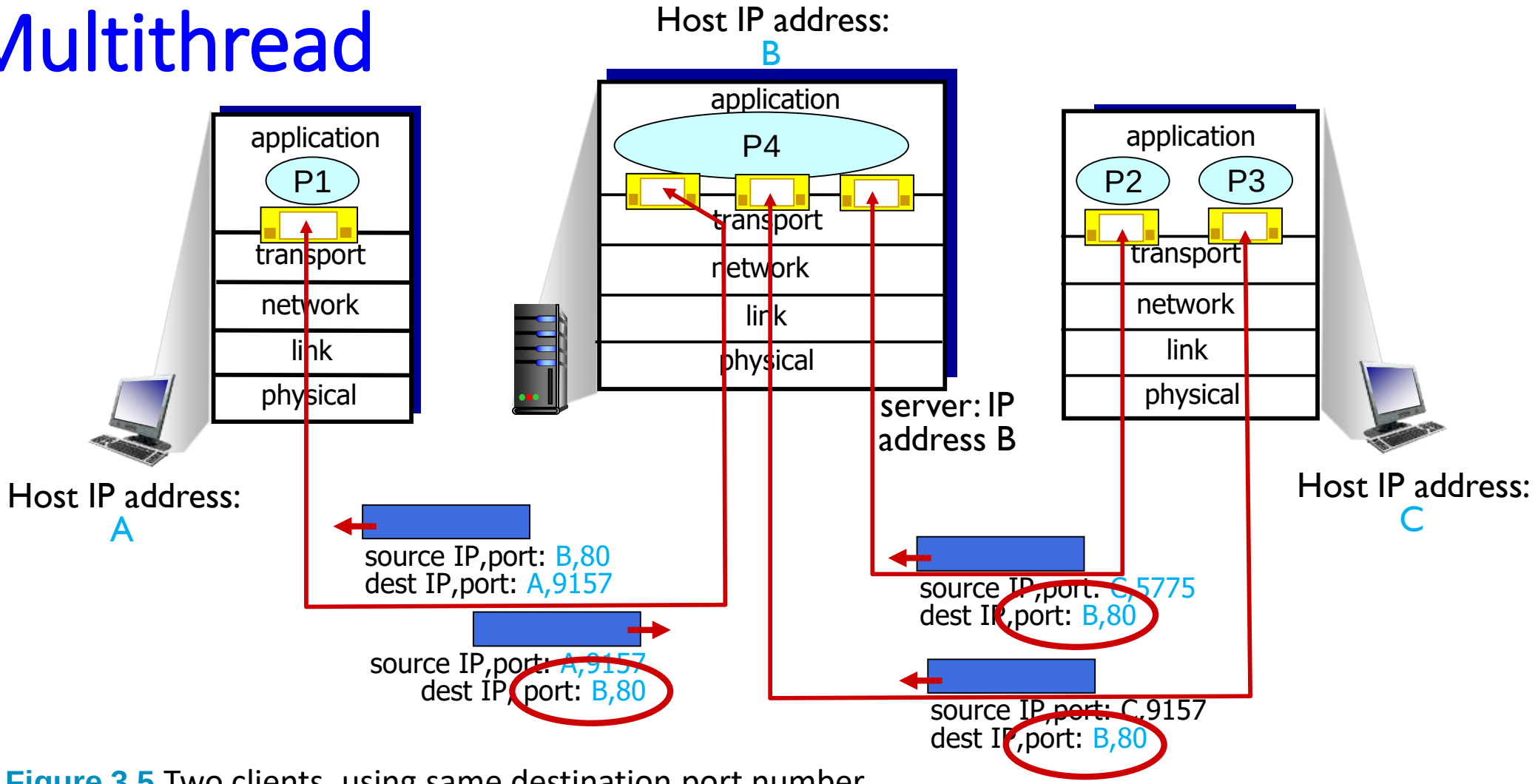


Figure 3.5 Two clients, using same destination port number (80) to communicate with same Web server application

Contents

3.3 Connectionless Transport: UDP [RFC768]

- No connection establishment
  - no connection state at sender, receiver
- Small header size
- No congestion control, security, reliable data transfer
  - congestion control is needed to prevent network from entering a congested state (high packet loss)
- UDP can send too many packets continuously
- Can function in face of congested network
- Segments may be lost or delivered out-of-order to App

Why some Apps prefer UDP

Finer application-level control over: **what** and **when** data is sent

- Under UDP, as soon as an application process passes data to UDP, UDP will package data inside a UDP segment and immediately pass segment to IP, App is almost directly talking with IP
- Real-time applications can not tolerate TCP's (congestion control, ACK).
- Applications can use UDP and implement, as part of application, additional functionality such as rdt, …

Why some Apps prefer UDP

No connection establishment

- UDP does not introduce any delay to establish a connection
- This is probably reason why DNS runs over UDP rather than TCP. DNS would be much slower if it ran over TCP
- HTTP uses TCP since reliability is critical for Web pages
- QUIC protocol (Quick UDP Internet Connection, [IETF QUIC 2020]), used in Google's Chrome browser, uses UDP as its underlying transport protocol and implements reliability in an application layer.
  - Web processes can communicate reliably without being subjected to transmission-rate constraints imposed by TCP's congestion-control mechanism

Why some Apps prefer UDP

No connection state

- TCP maintains connection state in end systems. This connection state includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters
- UDP does not maintain connection state and does not track any of these parameters
- Apps in UDP server can support many more active clients

Small packet header overhead

- TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead

Figure 3.6

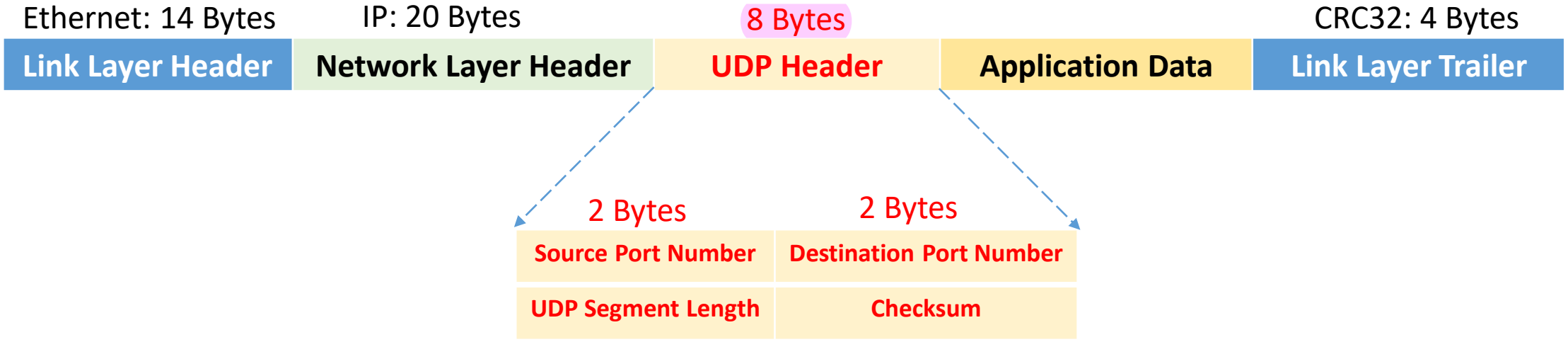
- Popular Internet applications and their underlying transport protocols

HTTP/3: add needed reliability and congestion control at application layer

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Secure remote terminal access	SSH	TCP
Web	HTTP, HTTP/3	TCP (for HTTP), UDP (for HTTP/3)
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	DASH	TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Name translation	DNS	Typically UDP

UDP in action

- Port numbers identify sending and receiving processes, range from 1 to 65,535
- UDP message length is at least 8 bytes (I.e., UDP data can be empty) and at most 2^{16} -1 = 65,535 bytes



3.3.1 UDP Segment Structure

- UDP header: four 2Byte fields
- Two port numbers for Max/Demux
- Length field specifies number of bytes in UDP segment (header plus data)
- Checksum field is used by receiving host to check whether errors have been introduced into segment
- Checksum is calculated over a few of fields in IP header and UDP segment

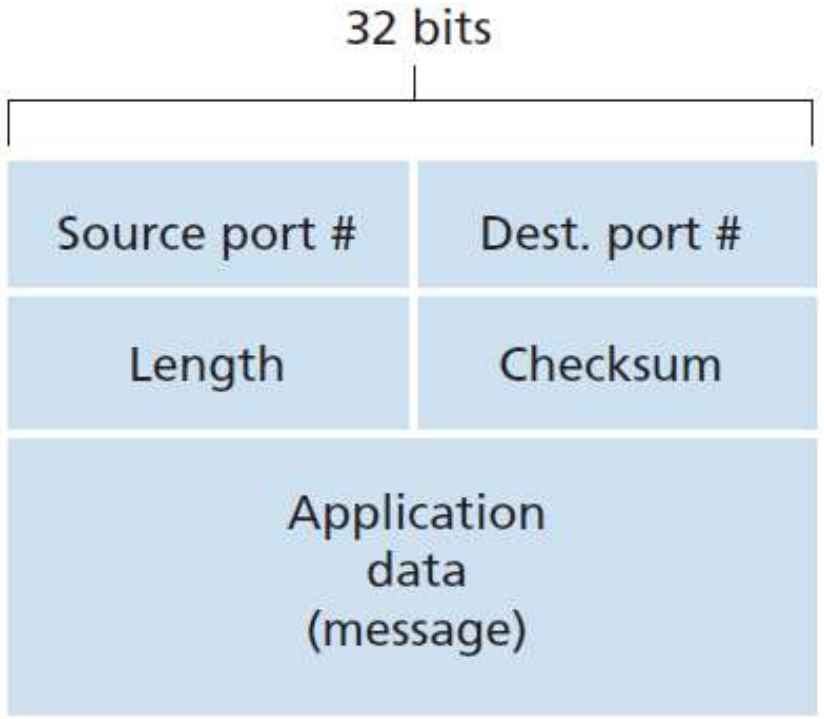


Figure 3.7 UDP segment structure

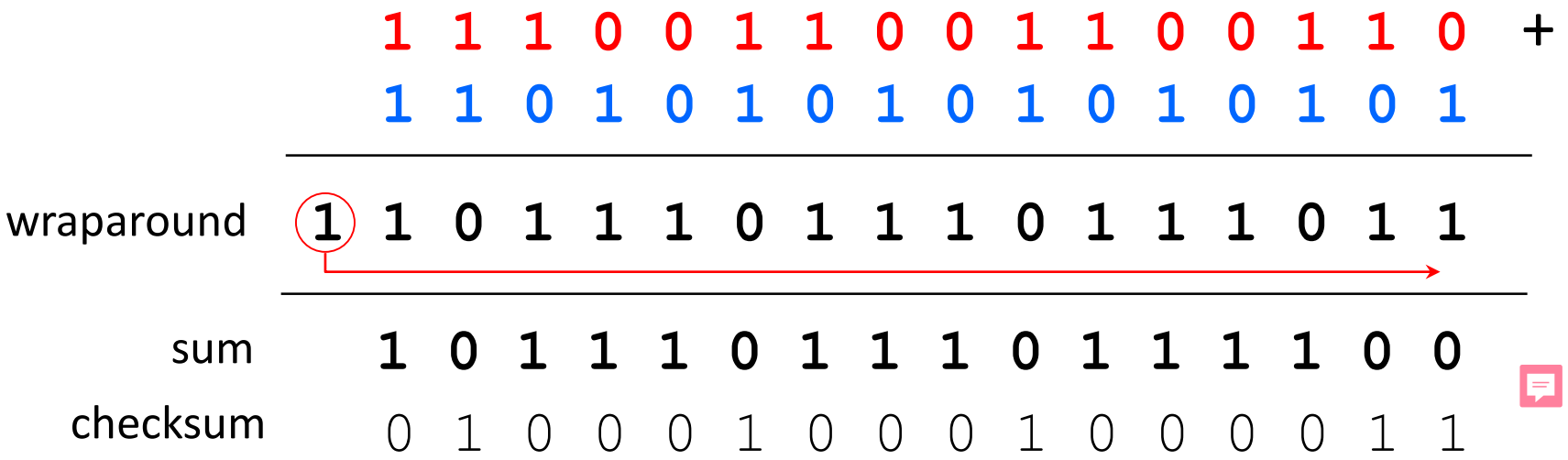
UDP checksum

- Checksum comes from adding (1's complete) 16 bit words of following items:
- UDP Header, UDP body and Pseudo IP header
- Pseudo IP header:
  - Source IP Address, 32bits
  - Destination IP Address, 32bits
  - Length field in IP header, 16bits
  - Upper Layer in IP header, 8bits
  - Time to Live in IP header, 8bits

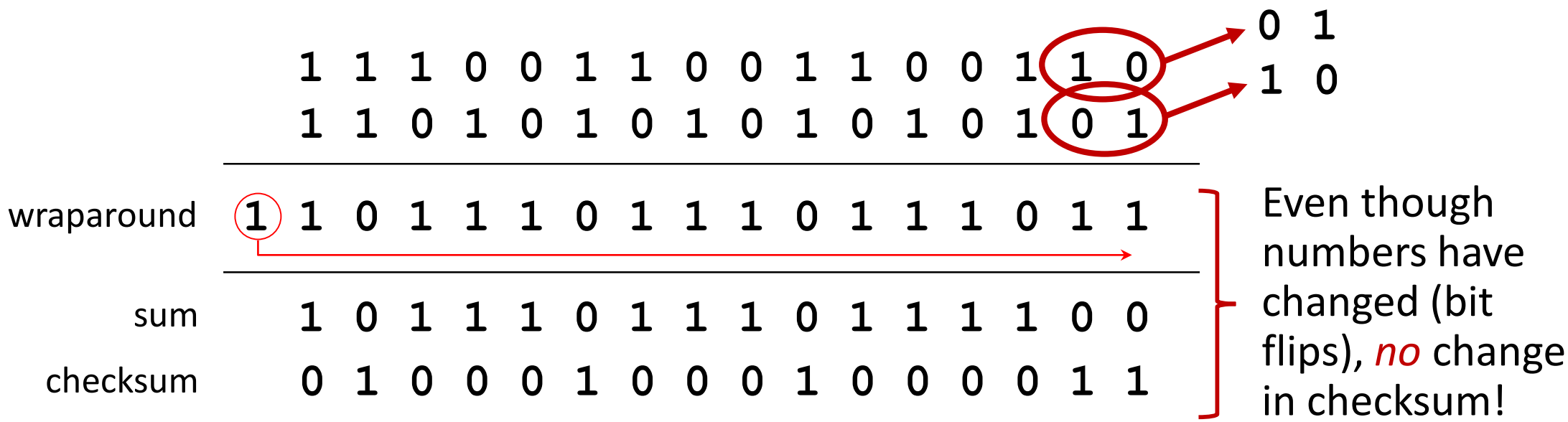
  TOTAL = 6*16 bits
- UDP Checksum: 6*16Pseudo IP header + 2*16UDP header + n*16 UDP Body

Example - add two 16-bit integers

- When adding numbers, a carryout from most significant bit needs to be added to the result



Internet checksum: weak protection



Why UDP/TCP provides checksum

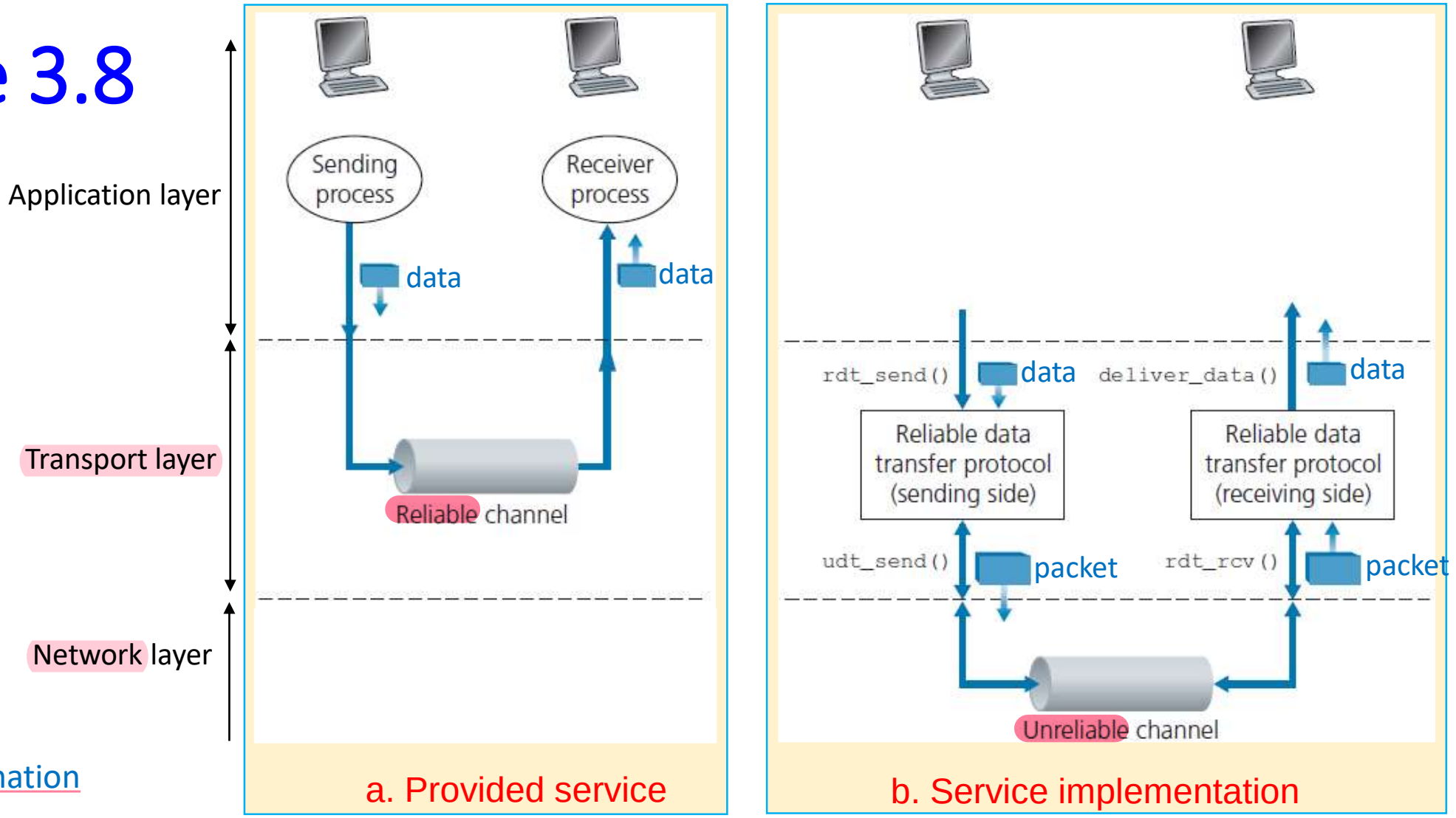
- Many link-layer protocols (including Ethernet) also provide error checking
- **Q**: Why UDP provides a checksum?
- **A**:
  - There is no guarantee that all links between source and destination provide error checking
  - Furthermore, it's possible that bit errors could be introduced when a segment is stored in a router's memory.
- UDP provides error detection at transport layer, on an end-end basis
- UDP does not do anything to recover from an error
  - Some implementations of UDP simply discard damaged segment
  - Others pass damaged segment to application with a warning

Contents

3.4 Principles of Reliable Data Transfer

- Implementing rdt occurs not only at transport layer, but also at link layer and application layer
- Figure 3.8: a. Service provided to upper-layer is a reliable channel. No transferred data bits are corrupted (flipped from 0 to 1, or vice versa) or lost, and all are delivered in order in which they were sent
- It is responsibility of a reliable data transfer protocol to implement this service, while layer below may be unreliable
- Example: TCP is an rdt protocol that is implemented on top of an unreliable lower layer (network layer)

Figure 3.8



Packets contain:

- control information

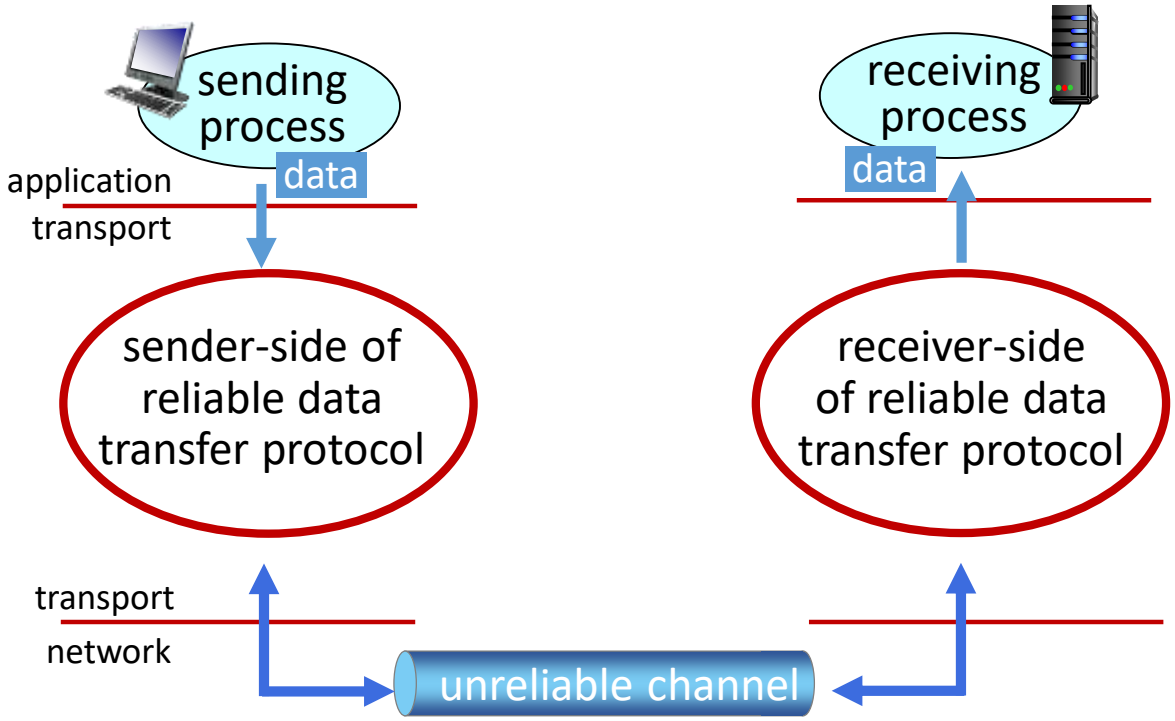
or

- control information + data

Reliable data transfer: a. Provided service and b. Service implementation

3.4.1 Building a Reliable Data Transfer Protocol

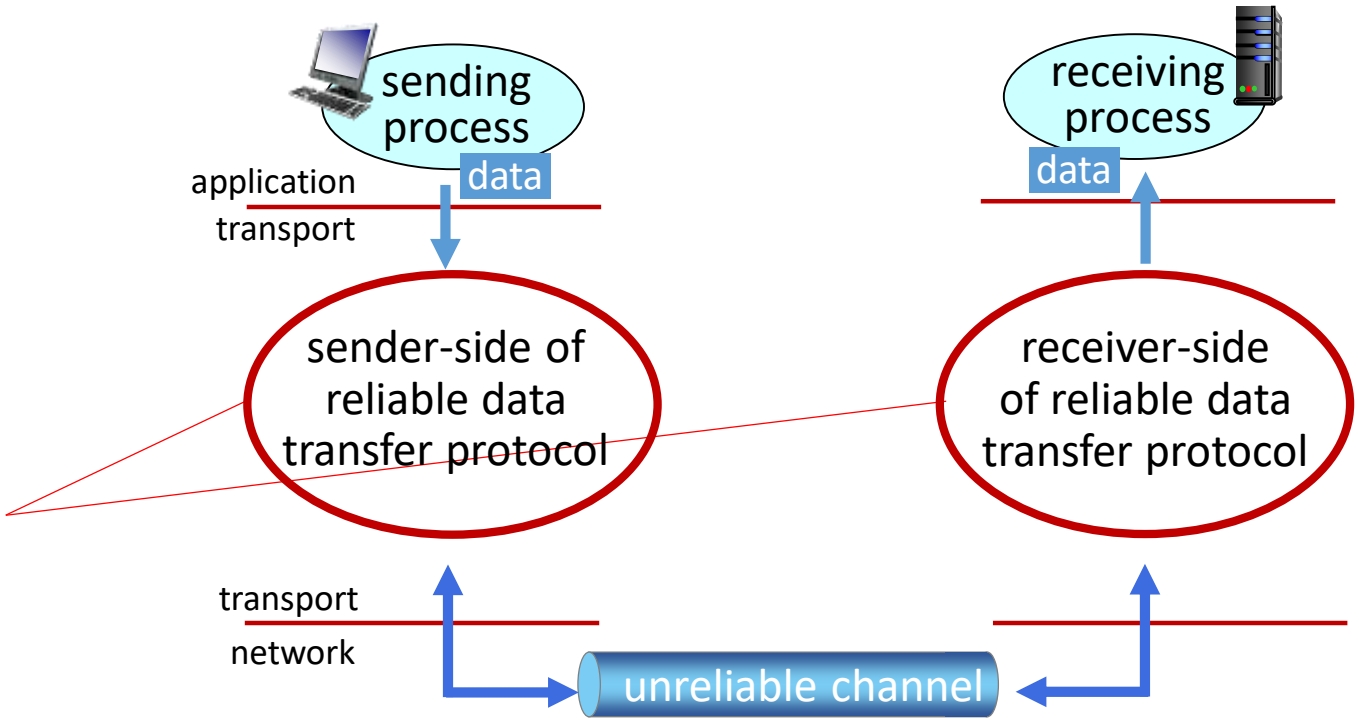
reliable service abstraction



reliable service implementation

3.4.1 Building a Reliable Data Transfer Protocol

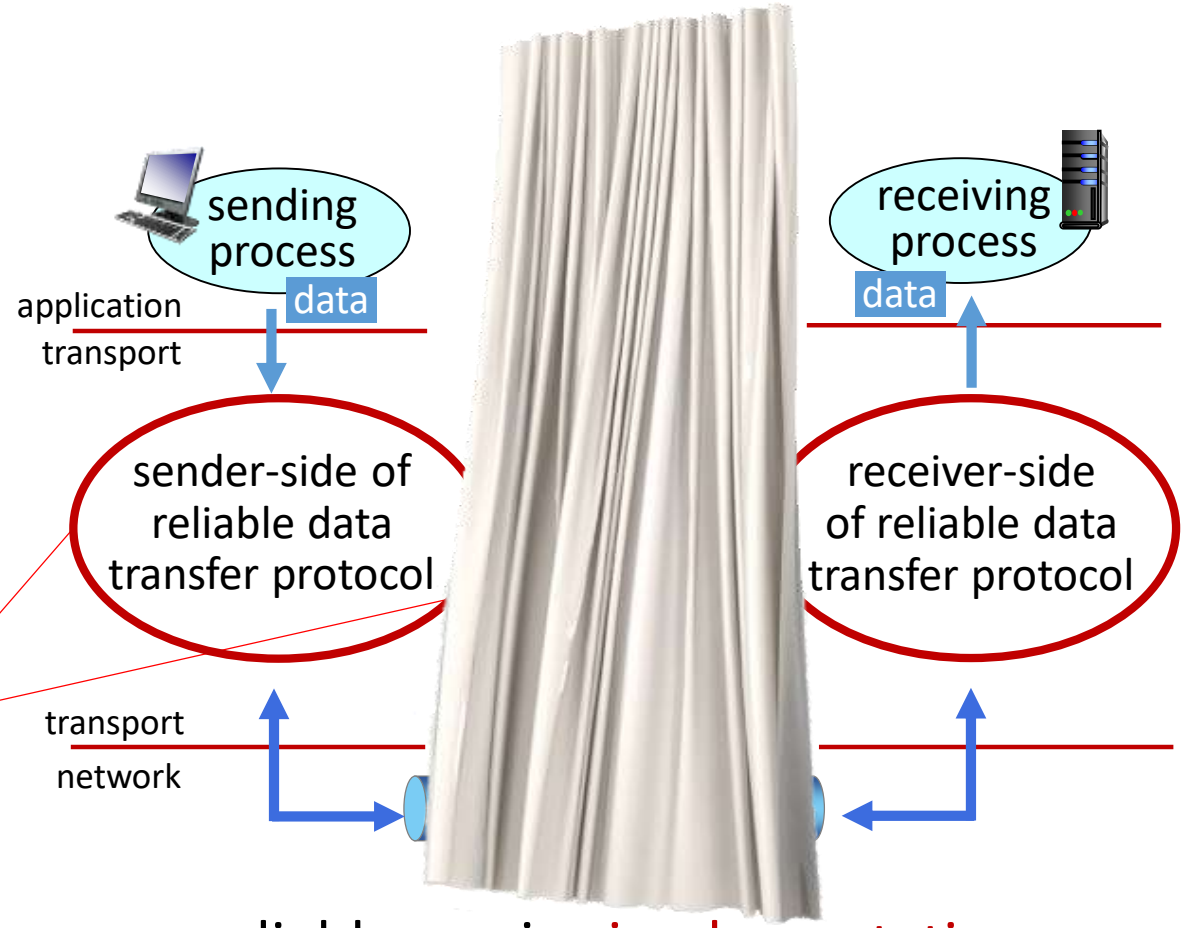
Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data)



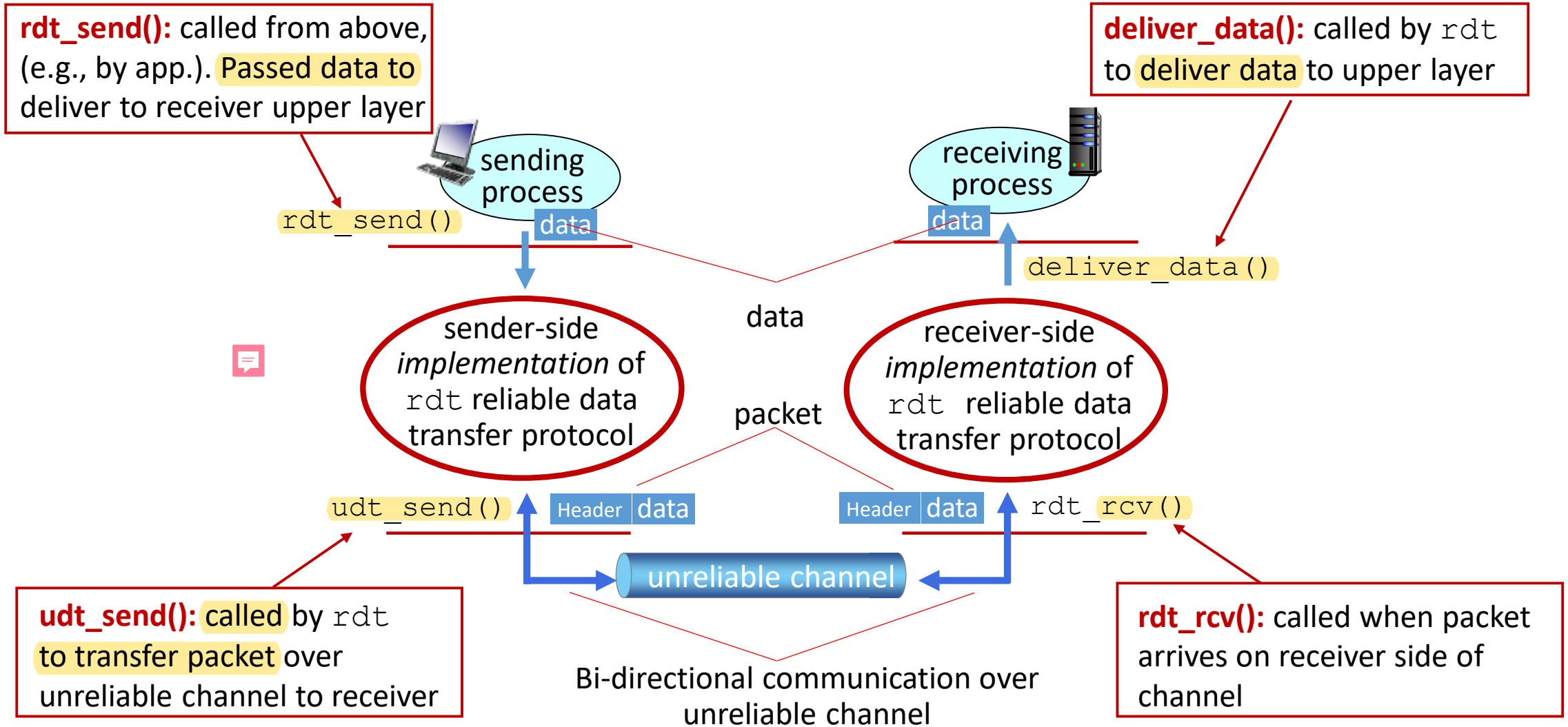
reliable service implementation

3.4.1 Building a Reliable Data Transfer Protocol

Sender, receiver do not know "state" of each other, e.g., was a message received, unless communicated via a message



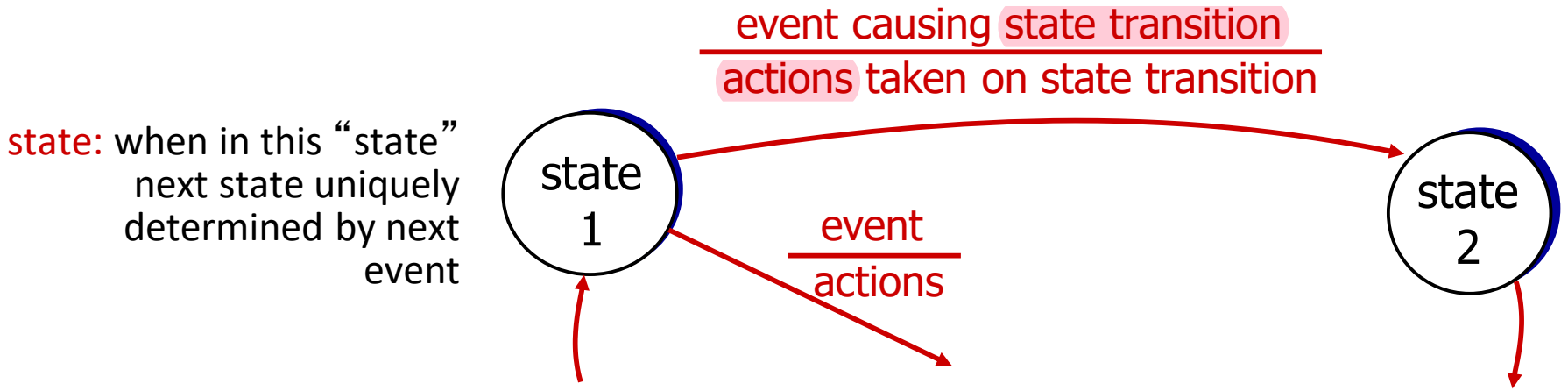
Reliable data transfer



Reliable data transfer: getting started

We will:

- incrementally develop sender, receiver sides of `rdt` protocol
- consider only unidirectional data transfer
  - but control info will flow in both directions
- use finite state machines (FSM) to specify sender, receiver



Incrementally develop **rdt** protocol

`rdt1.0`

- reliable transfer over a reliable channel

`rdt2.0`

- channel with bit errors using ACK/NAK packets

`rdt2.1`

- handling corrupted ACK/NAKs

`rdt2.2`

- a NAK-free protocol

`rdt3`

- channels with errors and loss

rdt1.0: reliable transfer over a reliable channel

- Underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

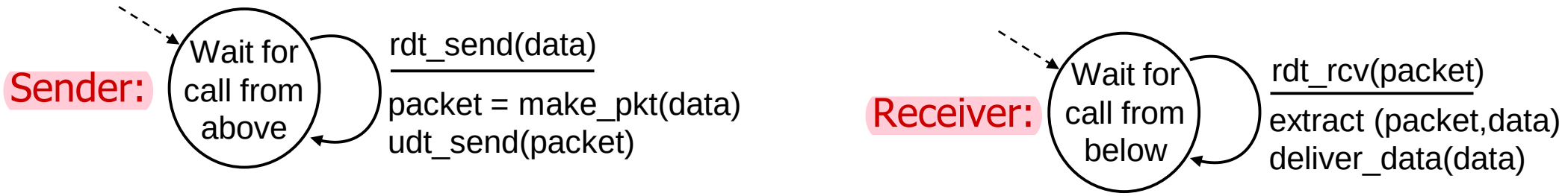


Figure 3.9 rdt1.0—A protocol for a completely reliable channel

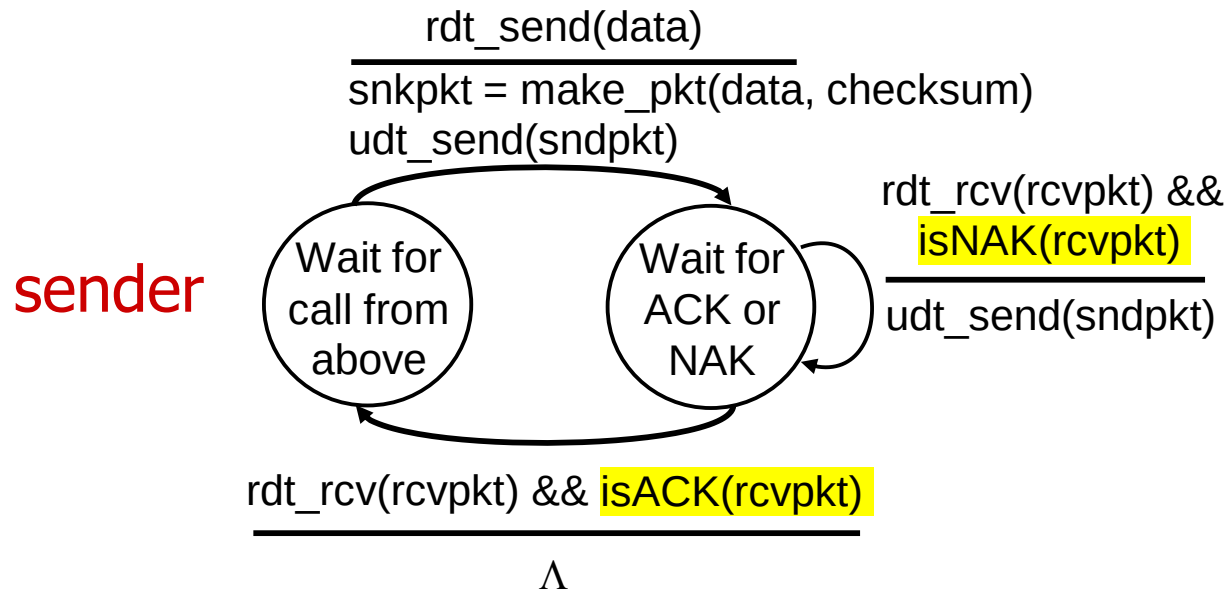
Reliable Data Transfer over a Channel with Bit Errors: rdt2.0

- Underlying channel may flip bits in packet
  - checksum to detect bit errors
- **Q**: how to recover from errors?
  - Acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
  - Negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK

stop and wait

sender sends one packet, then waits for receiver response

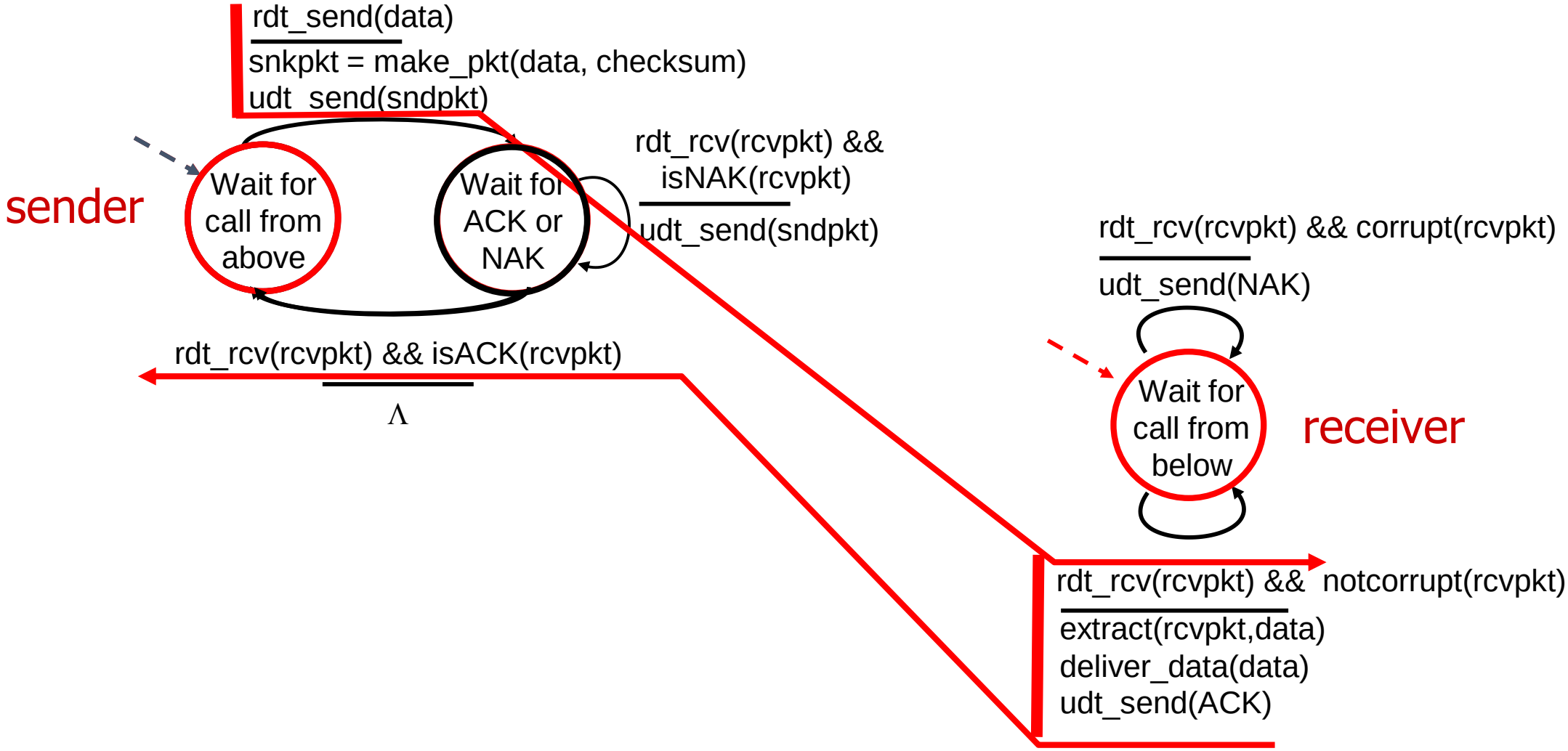
rdt2.0: FSM specification



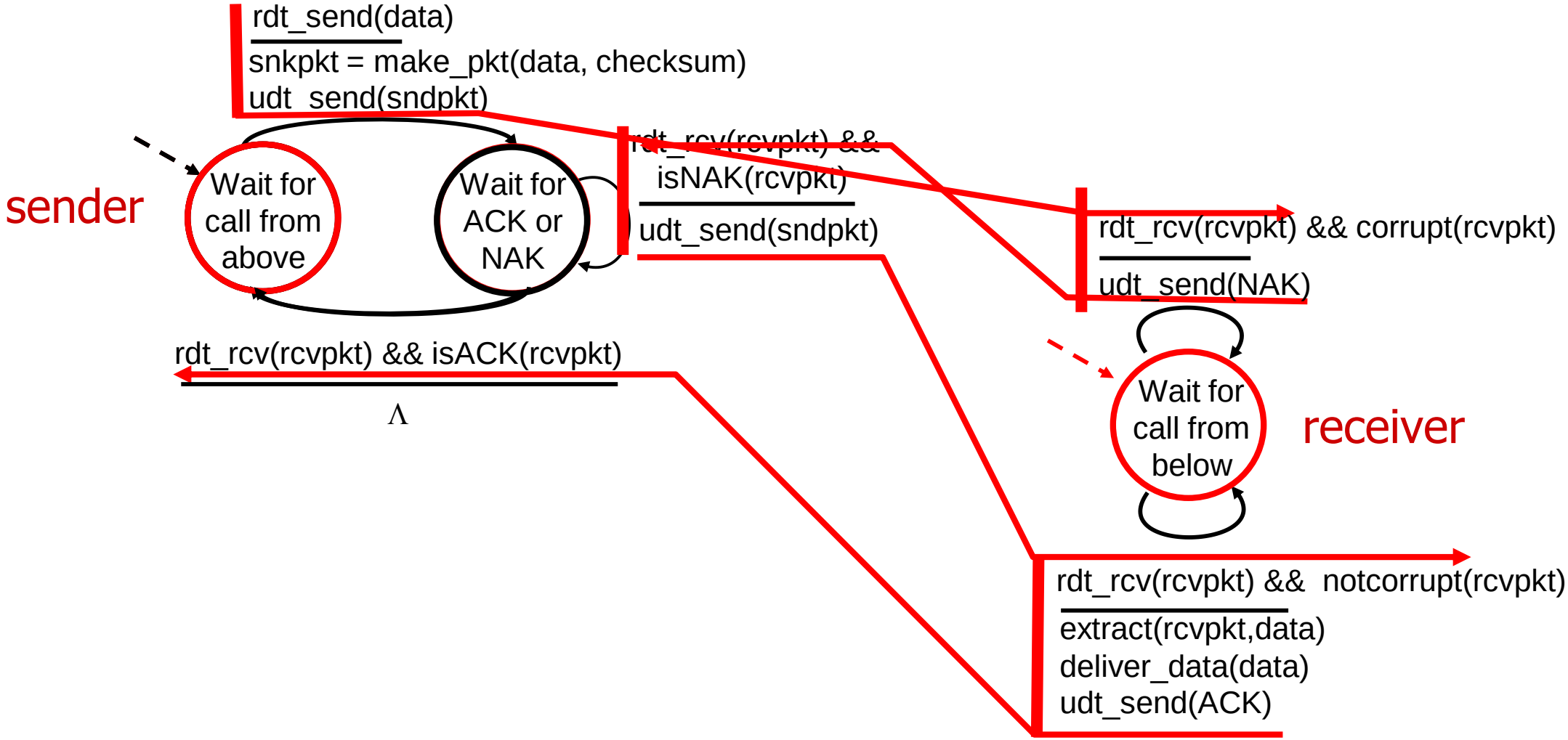
Note: "state" of receiver (did receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender

- that's why we need a protocol

rdt2.0: no error scenario



rdt2.0: corrupted packet scenario



rdt2.0: What happens if ACK/NAK corrupted?
rdt2.1: Adding sequence number to pkt

What happens if ACK/NAK corrupted?

- Sender doesn't know what happened at receiver
- Can't just retransmit: possible duplicate

Handling duplicates:

1. Sender retransmits current pkt if ACK/NAK corrupted
2. Sender adds sequence number to each pkt
3. Receiver discards (doesn't deliver up) duplicate pkt

rdt2.1: sender, handling corrupted ACK/NAKs

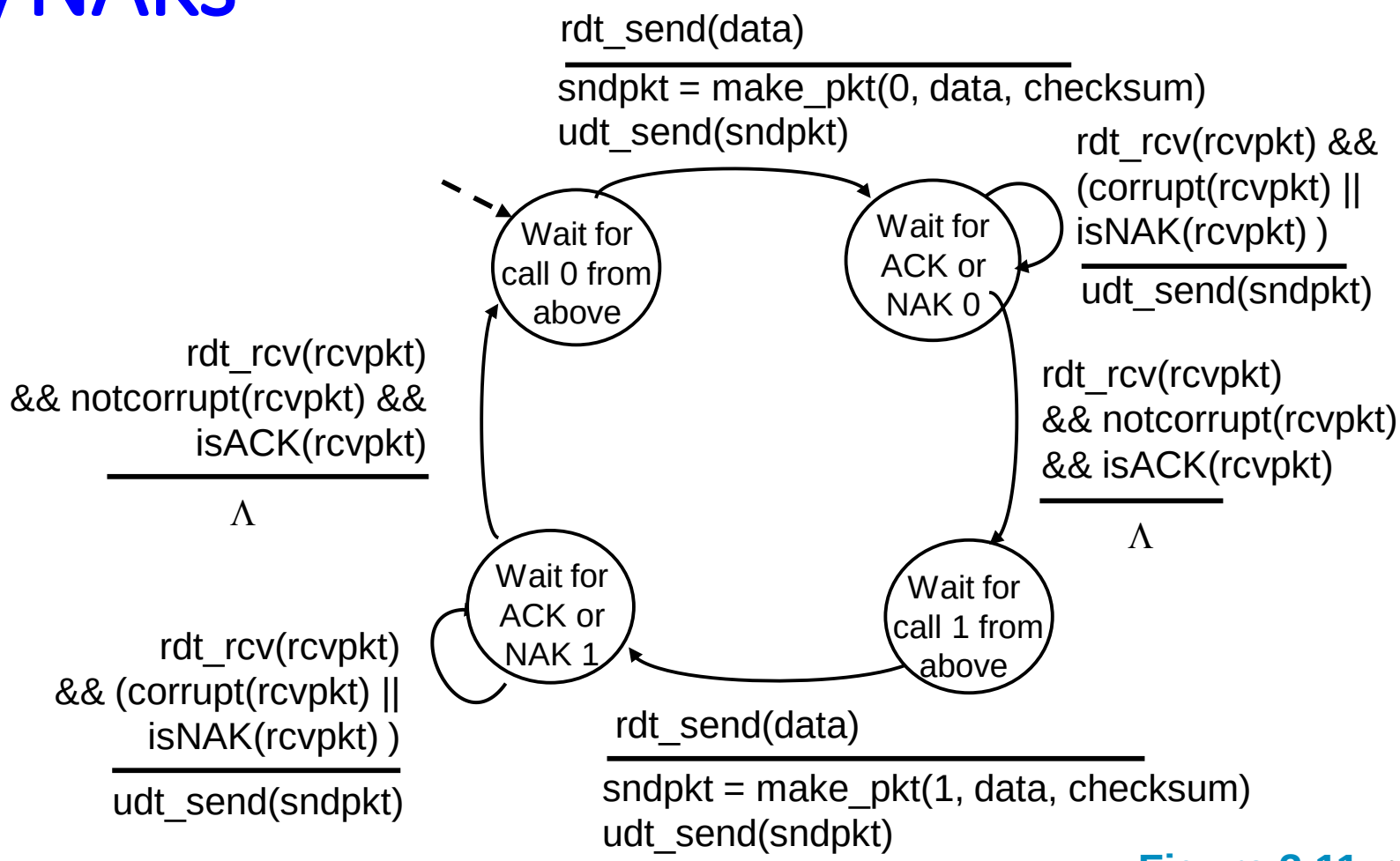


Figure 3.11 rdt2.1 sender

rdt2.1: receiver, handling corrupted ACK/NAKs

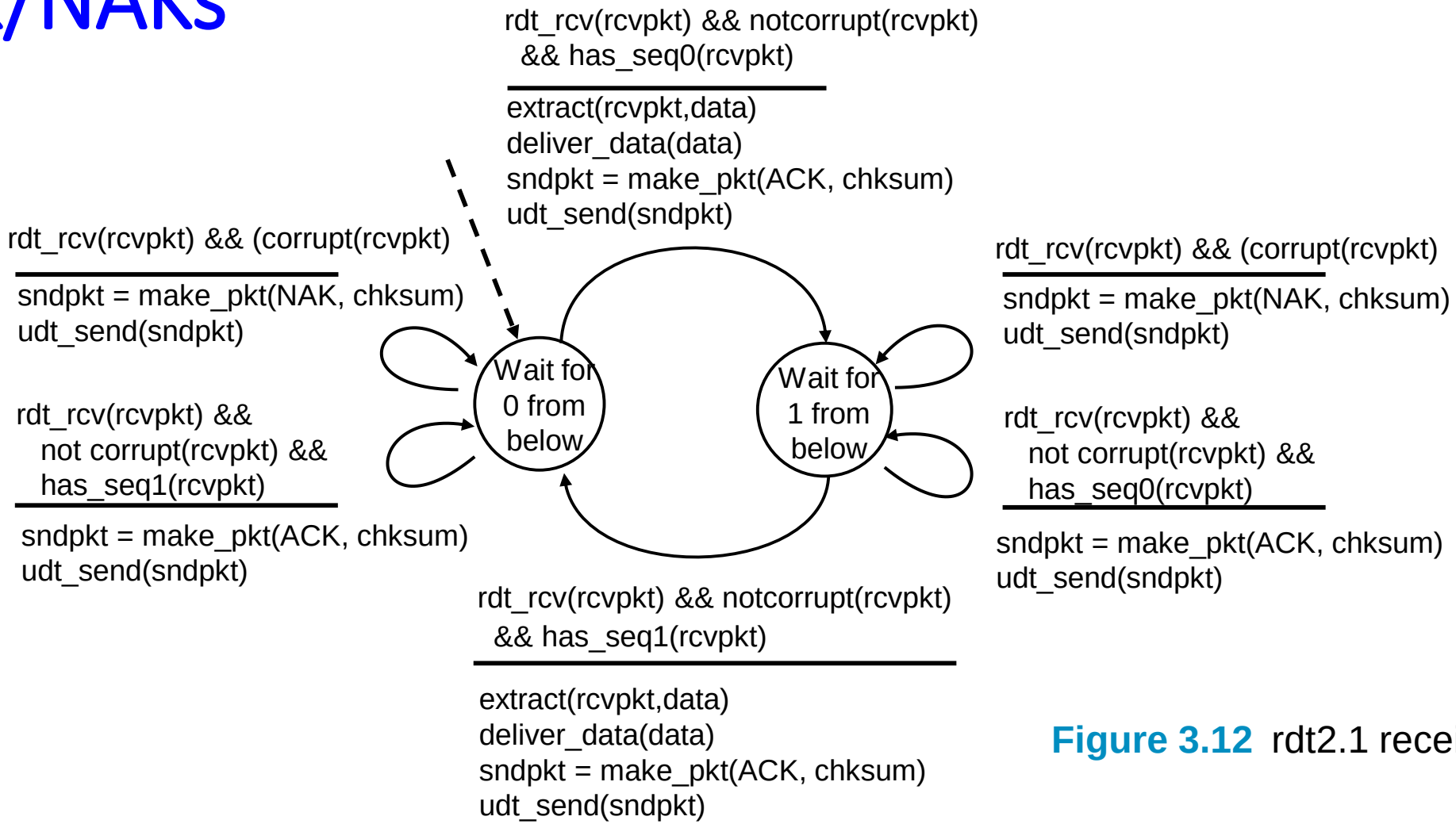


Figure 3.12 rdt2.1 receiver

rdt2.1: discussion

Sender:

- Seq # added to pkt
- Two seq. #s (0,1) will suffice. Why?
- Must check if received ACK/NAK corrupted
- Twice as many states
  - State must “remember” whether “expected” pkt should have seq # of 0 or 1

Receiver:

- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected pkt seq #
- Note: receiver can not know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- Same functionality as rdt2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt received OK
  - Receiver must explicitly include seq # of pkt being ACKed

- Duplicate ACK at sender results in same action as NAK: retransmit current pkt

rdt2.2: sender, receiver fragments

Figure 3.13 rdt2.2 sender

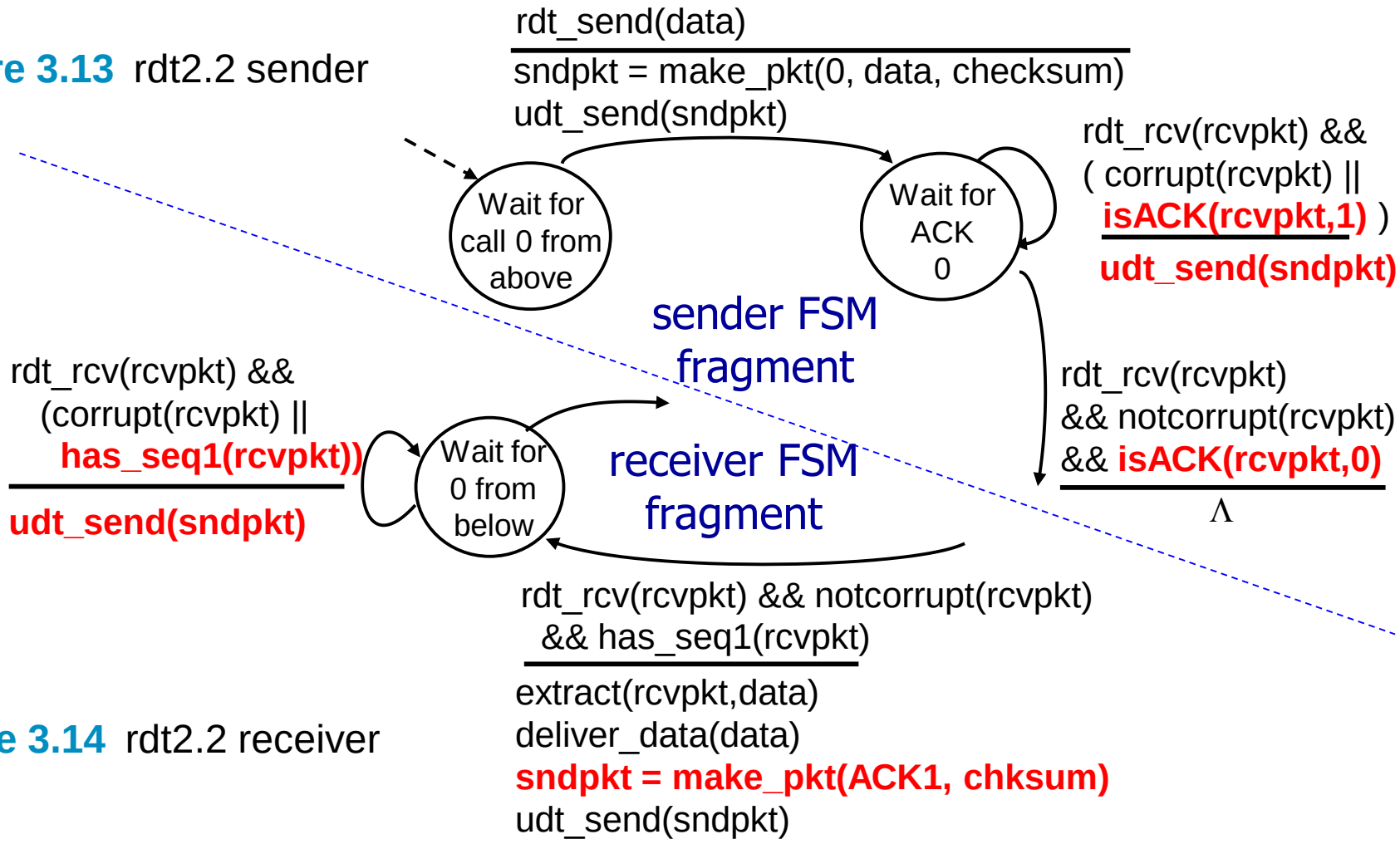


Figure 3.14 rdt2.2 receiver

rdt3.0: channels with errors and loss

New channel assumption*:* underlying channel can also lose packets (data, ACKs)

- checksum, sequence #s, ACKs, retransmissions will be of help … but not quite enough

rdt3.0: channels with errors and loss

- Approach: sender waits "reasonable" amount of time for ACK
- Retransmits if no ACK received in this time
- If pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

timeout

rdt3.0 sender

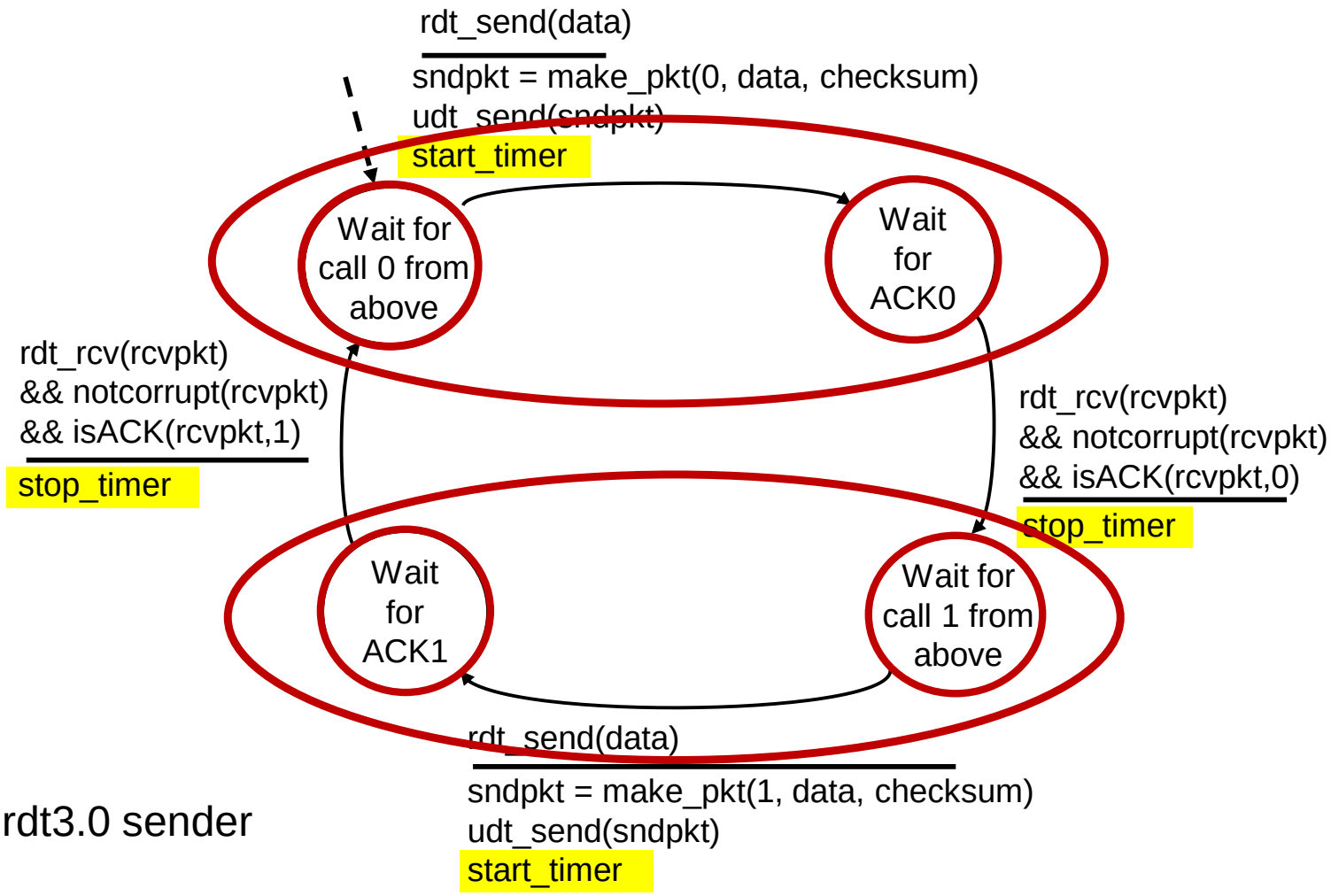
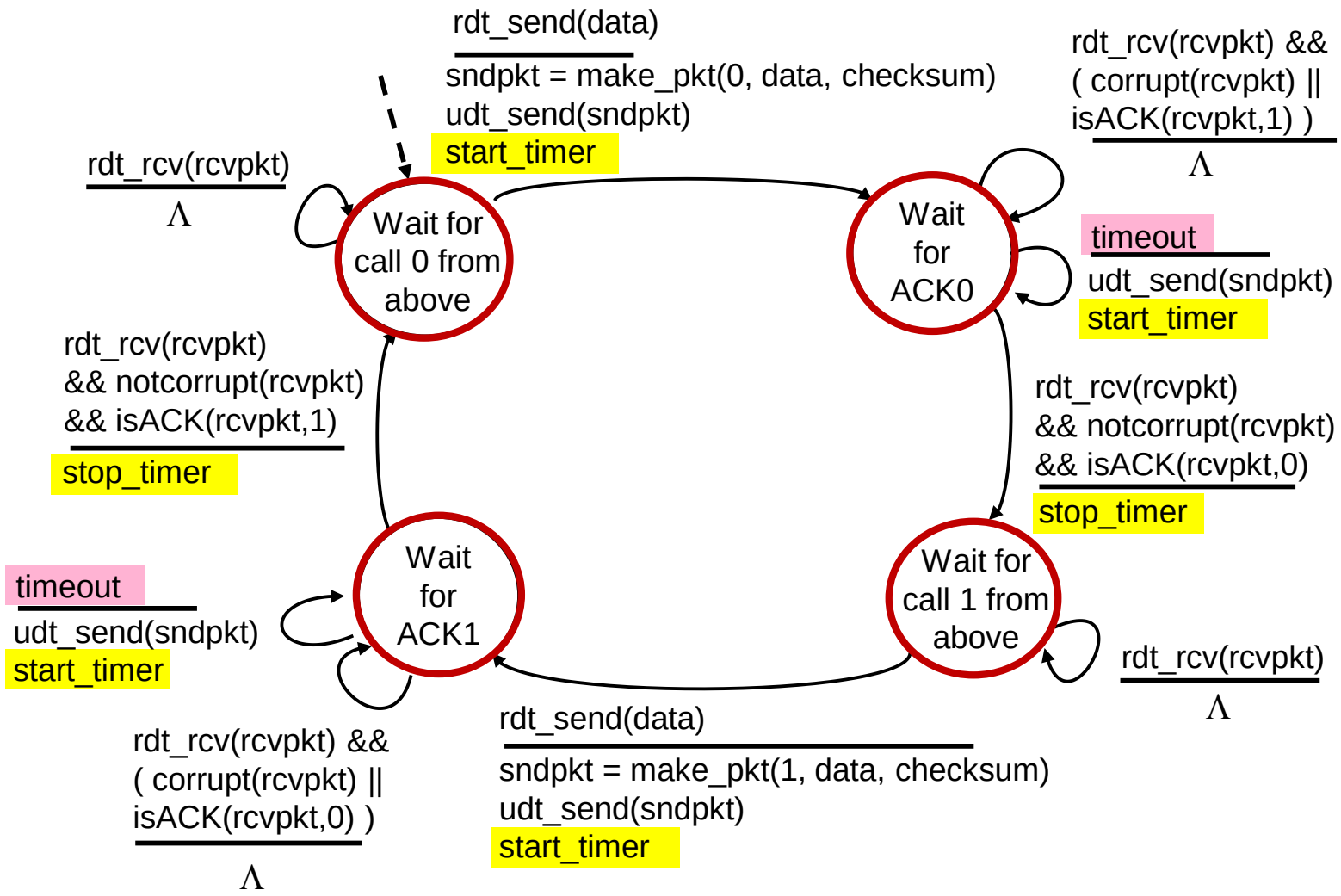


Figure 3.15 rdt3.0 sender

rdt3.0 in action



rdt3.0 in action

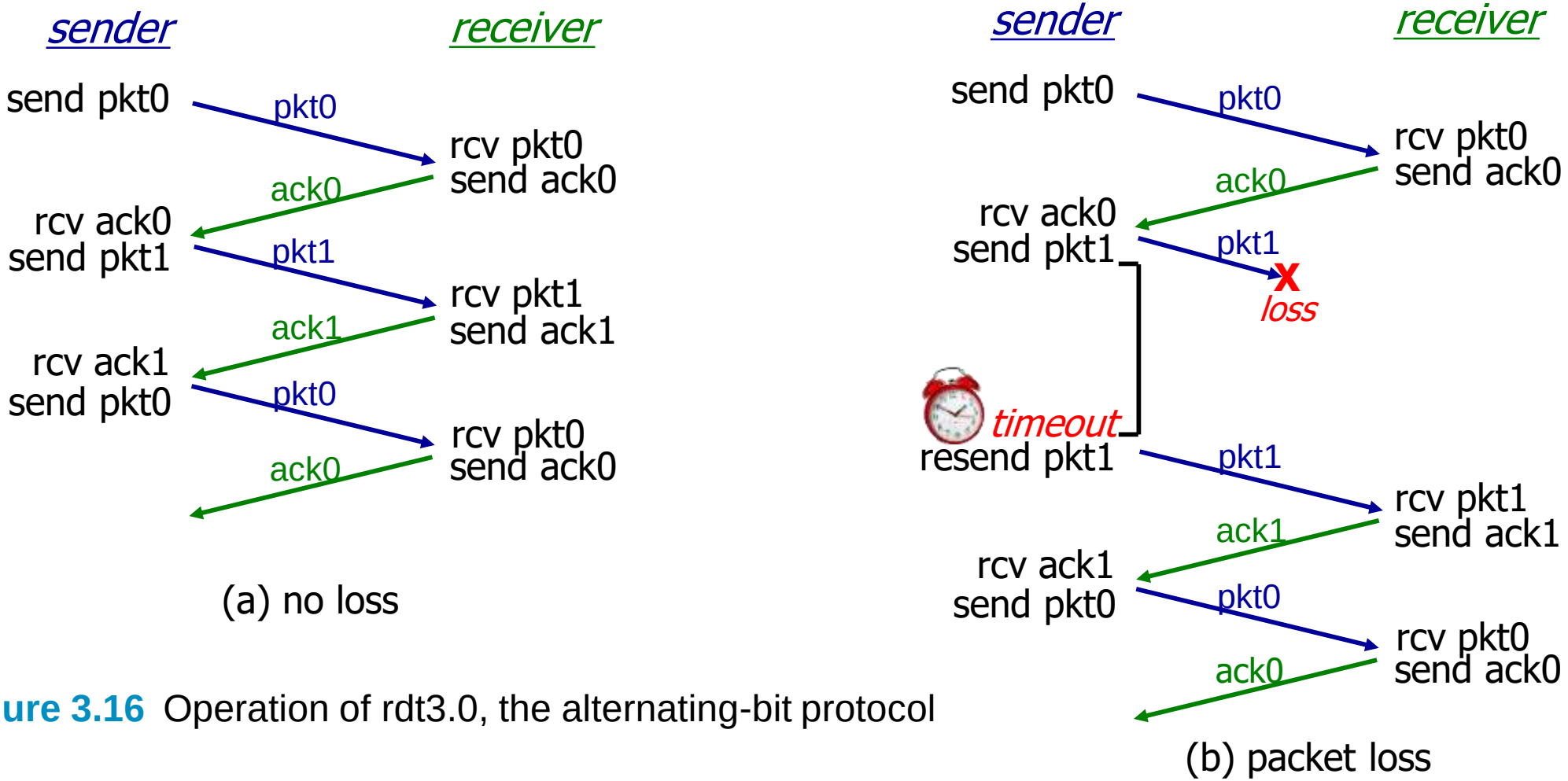
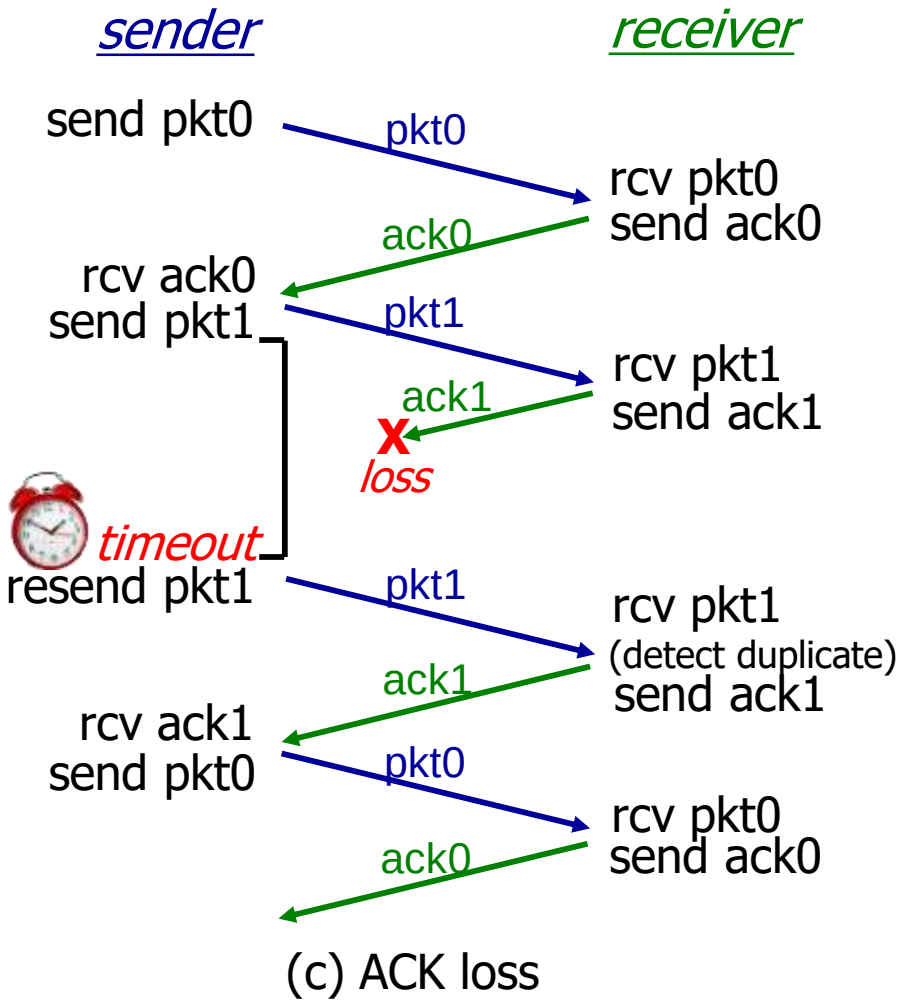
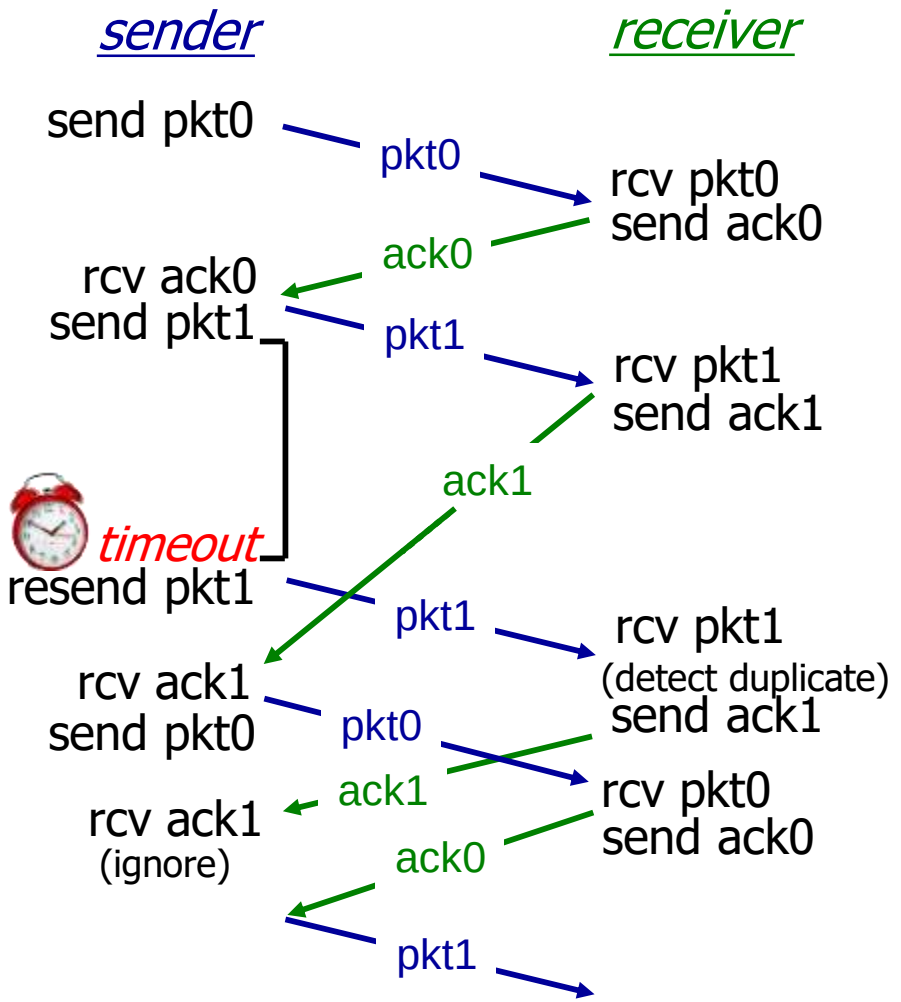


Figure 3.16 Operation of rdt3.0, the alternating-bit protocol

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

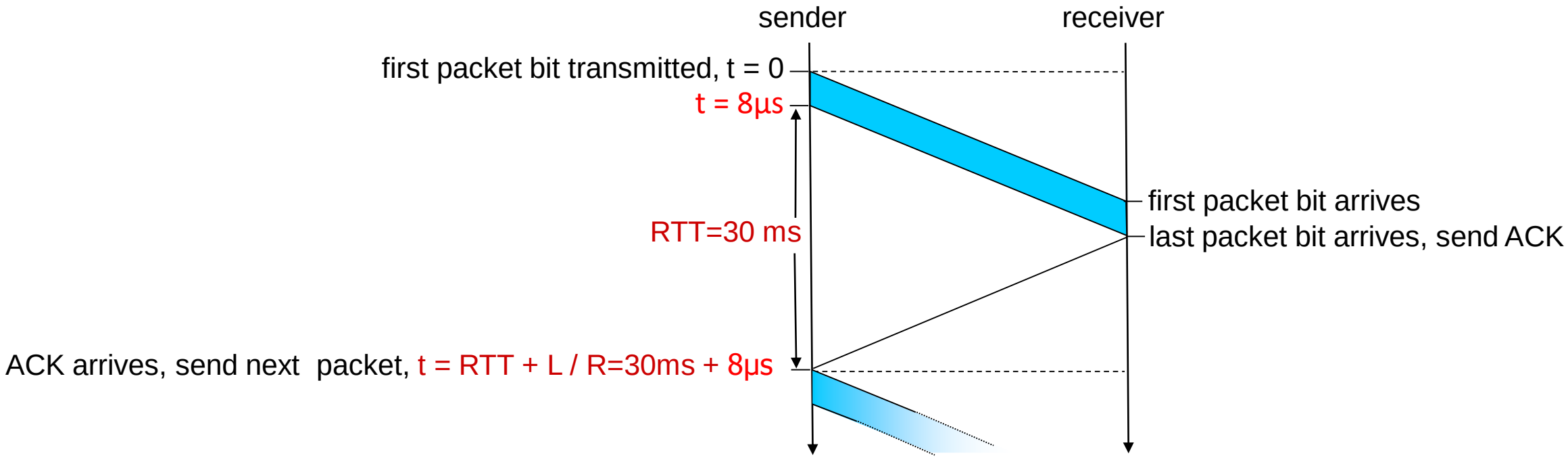
Performance of rdt3.0 (stop-and-wait)

- U_{sender}: utilization – fraction of time sender busy sending

- Example: 8000 bit packet, 1 Gbps link, RTT= 30 ms
  - Time to transmit packet into channel (packet transmission time):

$$T_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8\mu s$$

rdt3.0: stop-and-wait operation



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{0.008}{30.008} = 0.00027$$

rdt 3.0 limits performance of underlying network

Figure 3.18 Stop-and-wait and pipelined sending

3.4.2 Pipelined Reliable Data Transfer Protocols

Pipelining: Sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

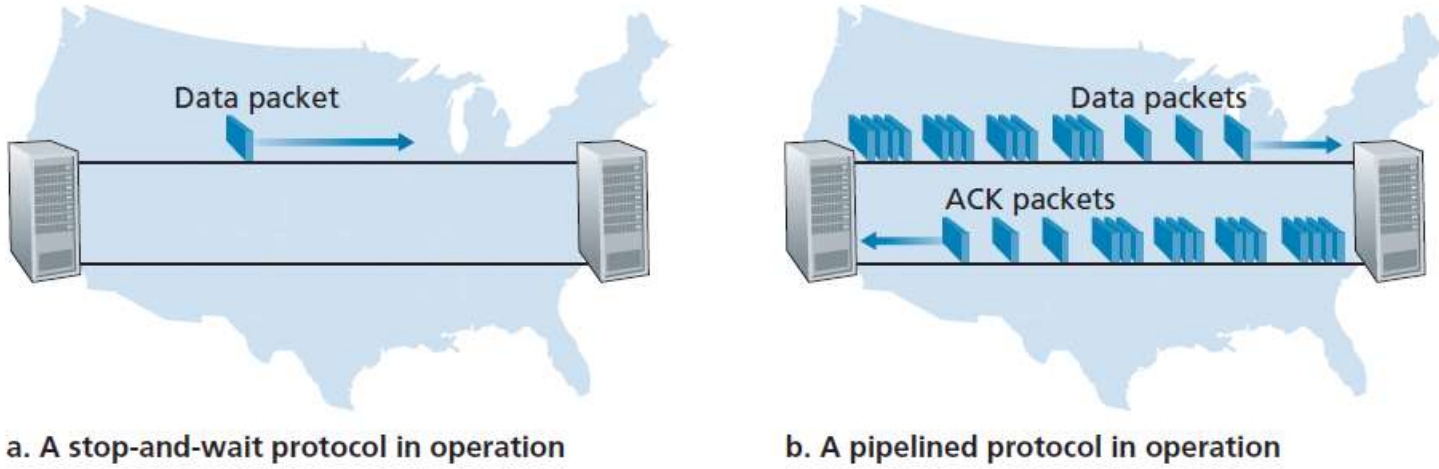


Figure 3.17 Stop-and-wait versus pipelined protocol

Pipelining: increased utilization

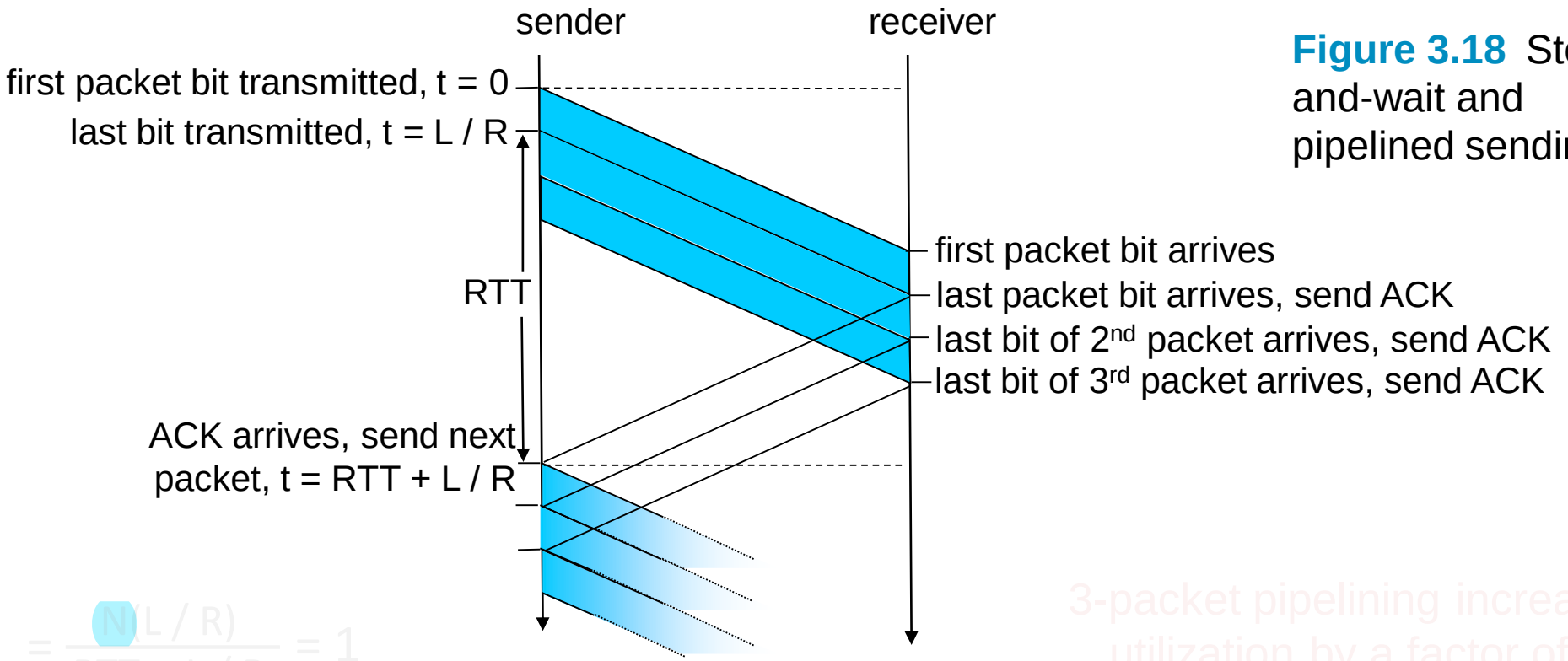


Figure 3.18 Stop-and-wait and pipelined sending

3.4.3 Go-Back-N (GBN): sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header
- cumulative ACK: ACK(n): ACKs all packets up to, including seq # n
  - on receiving ACK(n): move window forward to begin at n+1
- timer for oldest in-flight packet
- timeout(n): retransmit packet n and all higher seq # packets in window

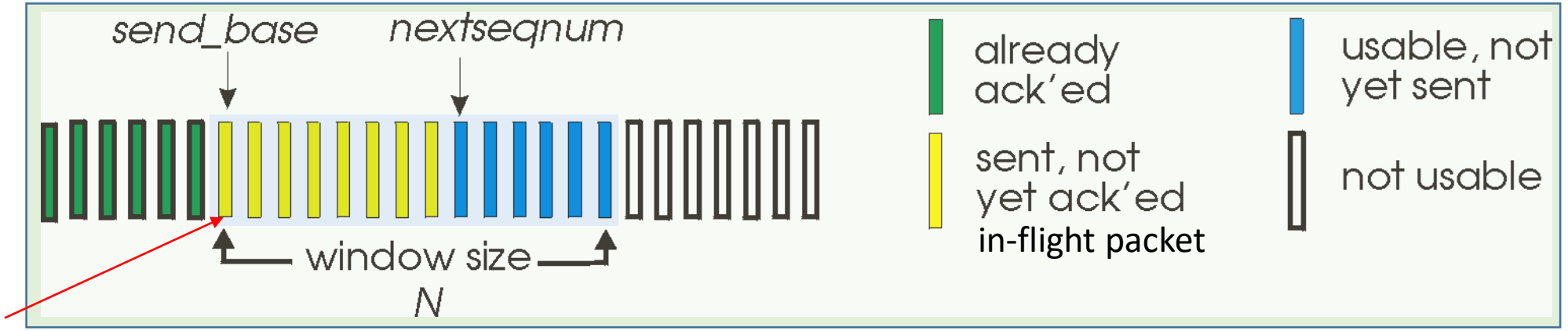
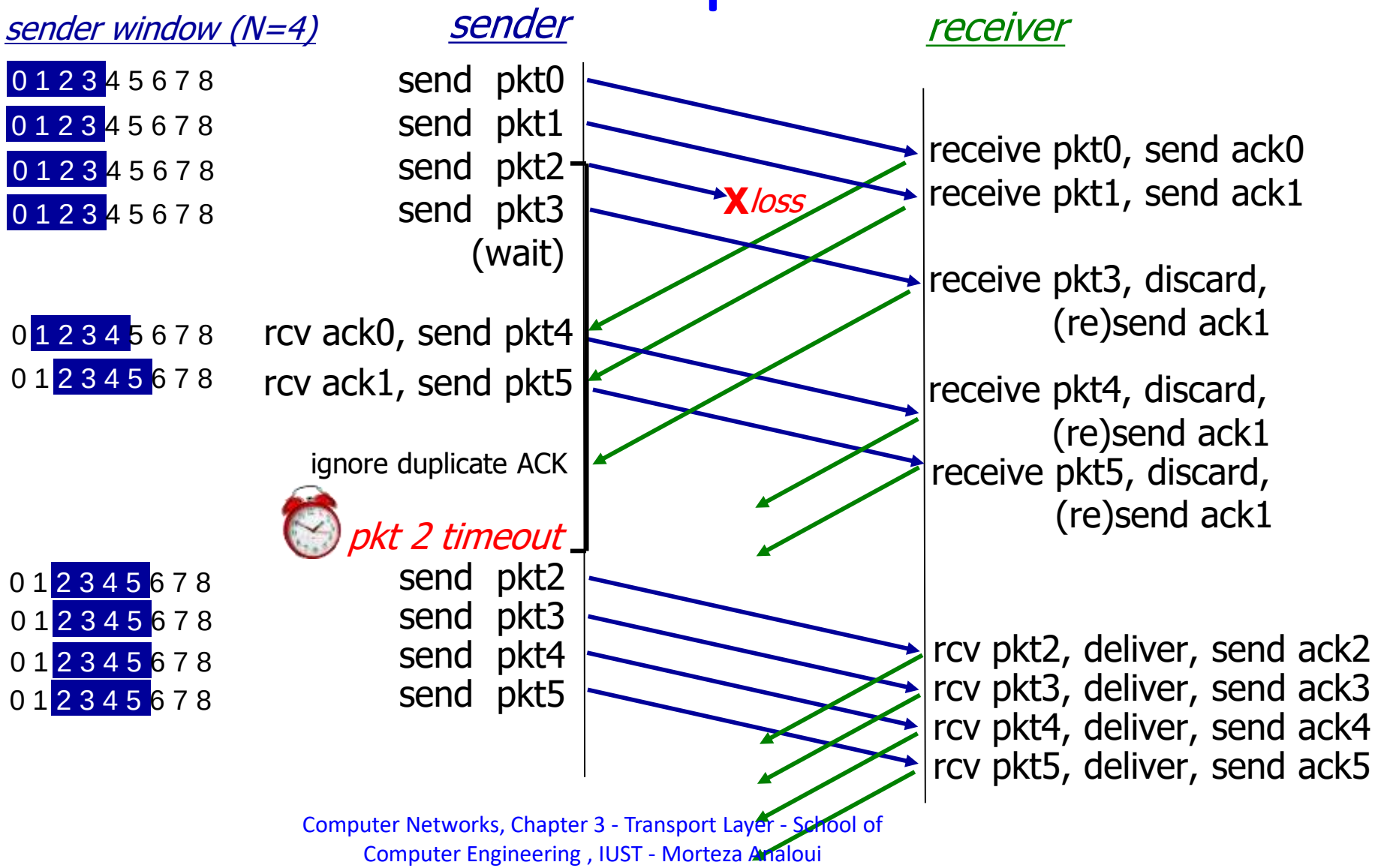


Figure 3.19 Sender's view of sequence numbers in Go-Back-N

Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest in-order seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base` (highest ACKed)
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Figure 3.22 Go-Back-N in operation



3.4.4 Selective Repeat (SR)

- receiver individually acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - N consecutive seq #s
  - limits seq #s of sent, unACKed packets

Selective repeat: sender, receiver windows

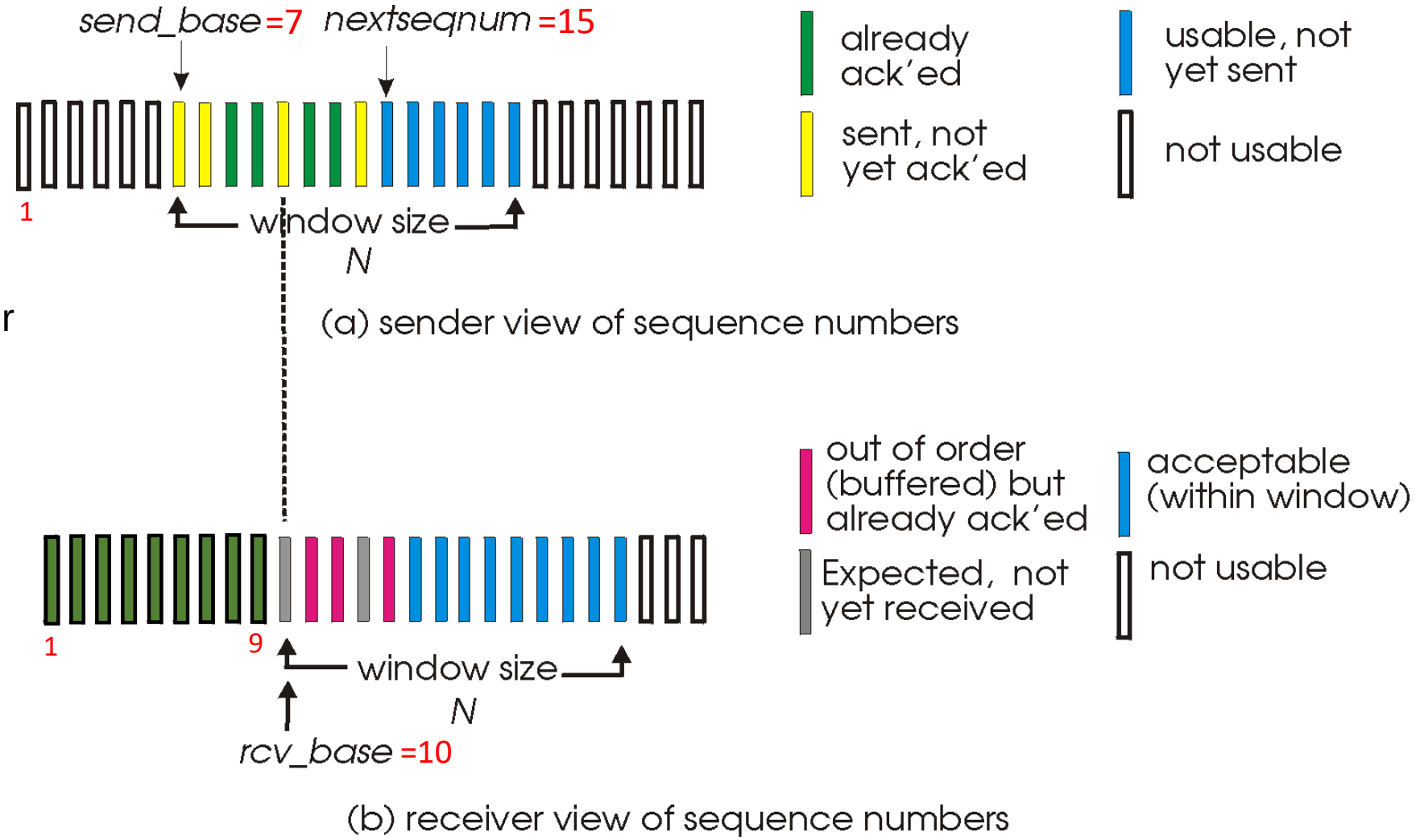


Figure 3.23 SR sender and receiver views of sequence-number space

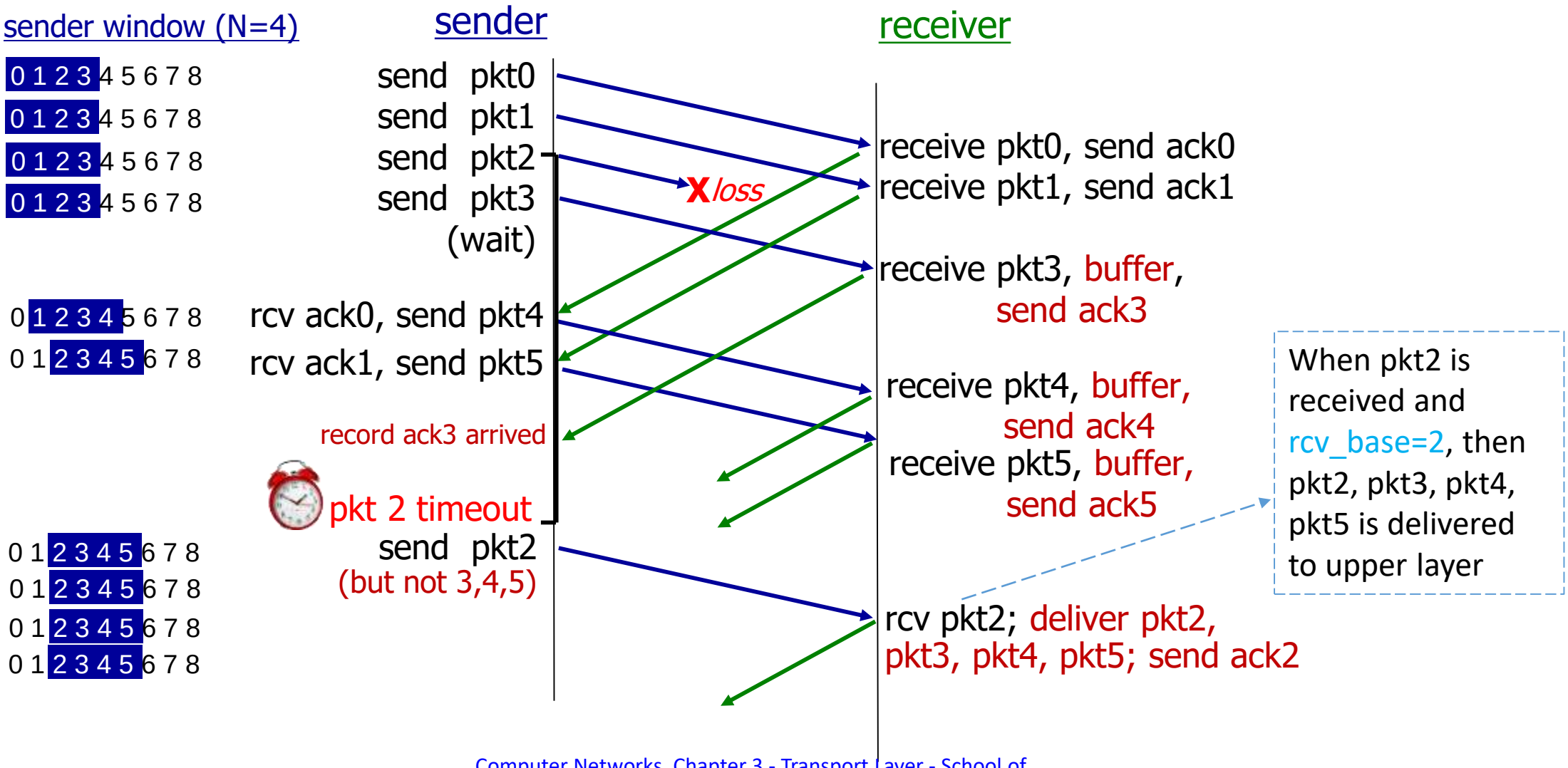
SR sender events and actions

1. Data received from above:
    1. SR sender checks next available sequence number for packet. If sequence number is within sender's window, data is packetized and sent; otherwise it is either buffered or returned to upper layer for later transmission, as in GBN
2. Timeout: 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 operation of multiple logical timers
3. ACK received:
    1. If an ACK is received in [send_base, send_base+N], SR sender marks that packet as having been received, provided it is in window.
    2. If packet's sequence number is equal to send_base, window base is moved forward to unacknowledged packet with smallest sequence number
    3. If window moves and there are untransmitted packets with sequence numbers that now fall within window, these packets are transmitted

SR receiver events and actions

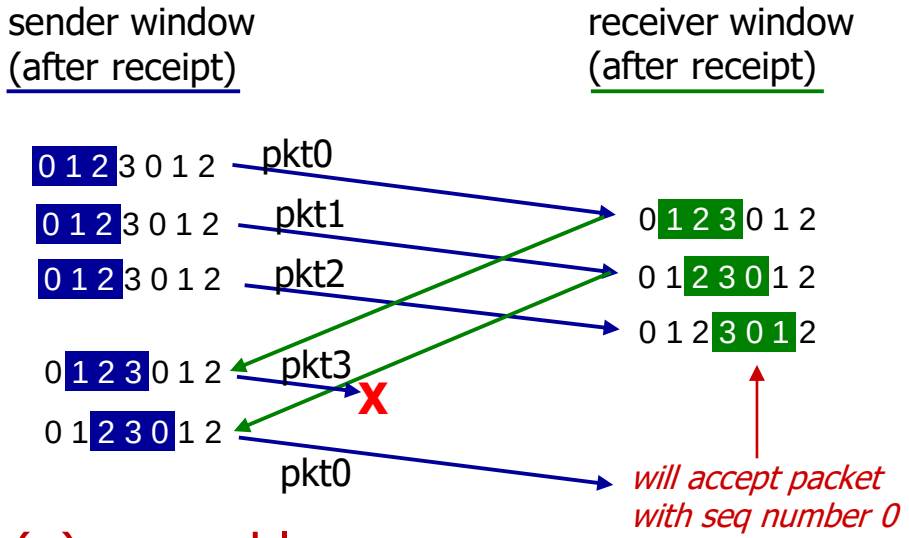
1. Packet with sequence number in [rcv_base, rcv_base+N-1] is correctly received:
    1. A selective ACK packet is returned to sender. If packet was not previously received, it is buffered.
    2. If this packet has a sequence number equal to base of receive window (rcv_base), then this packet, and any previously buffered and consecutively numbered packets are delivered to upper layer.
    3. Receive window is then moved forward by number of packets delivered to upper layer.
2. Packet with sequence number in [rcv_base-N, rcv_base-1] is correctly received.
    1. ACK must be generated, even though this is a packet that receiver has previously acknowledged. Receiver reacknowledges already received packets with certain sequence numbers *below* current window base
3. Otherwise. Ignore packet

Selective Repeat in action



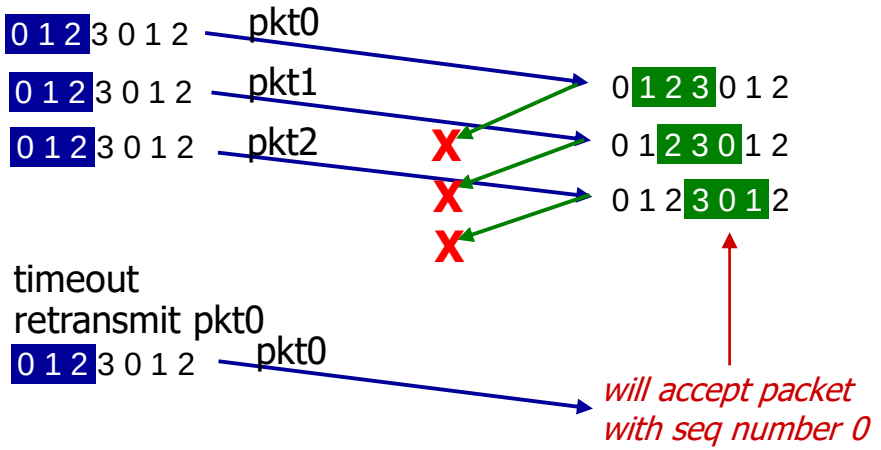
Selective repeat: a dilemma!

- example:
  - seq #s: 0, 1, 2, 3 (base 4 counting)
  - window size=3



(a) no problem

Figure 3.27 SR receiver dilemma with too-large windows: A new packet or a retransmission?



Selective repeat: a dilemma!

- example:
  - seq #s: 0, 1, 2, 3 (base 4 counting)
  - window size=3

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

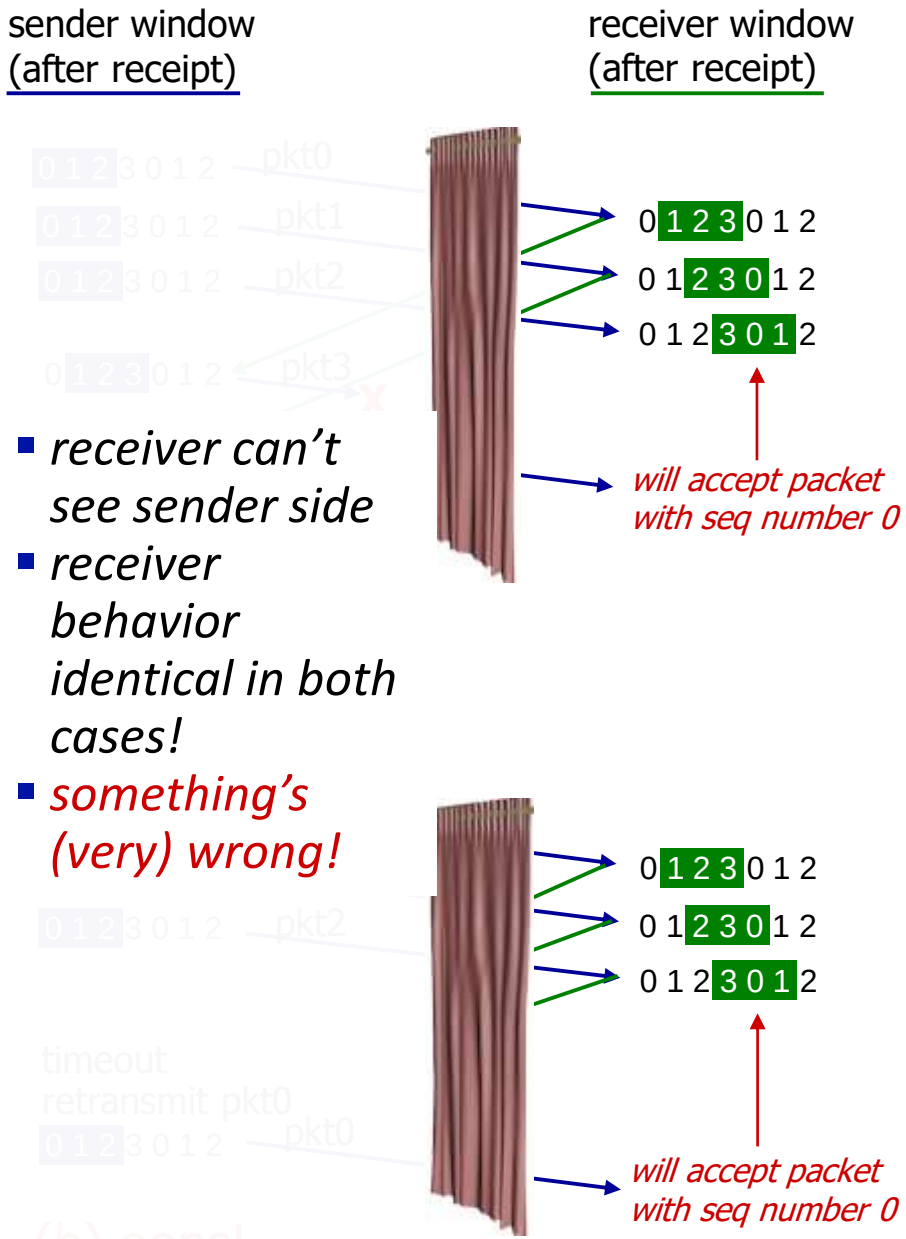


Table 3.1 Summary of reliable data transfer mechanisms and their use

Mechanism	Use, Comments
Checksum	• Used to detect bit errors in a transmitted packet
Timer	• Used to timeout/retransmit a packet, possibly because packet (or its ACK) was lost • Timeouts can occur when a packet is delayed but not lost, or when a packet has been received by receiver but receiver-to-sender ACK has been lost
Sequence number	• Used for sequential numbering of packets of data flowing from sender to receiver • Gaps in sequence numbers of received packets allow receiver to detect a lost packet • Packets with duplicate sequence numbers allow receiver to detect duplicate copies of a packet

Table 3.1 Summary of reliable data transfer mechanisms and their use

Mechanism	Use, Comments
Acknowledgment	• Used by receiver to tell sender that a packet or set of packets has been received correctly • Acknowledgments will typically carry sequence number of packet or packets being acknowledged • Acknowledgments may be individual or cumulative, depending on the protocol
Negative acknowledgment	• Used by receiver to tell sender that a packet has not been received correctly • Negative acknowledgments will typically carry sequence number of packet that was not received correctly
Window, pipelining	• Sender is 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 window size may be set on basis of receiver's ability to receive and buffer messages, or level of congestion in network, or both

Example

- Suppose p is drop probability of a packet in a packet switch
- **Q:** What is average hops (transmissions) for a packet to move from server to client (N)?
- Three possibilities:
  - a packet experience 3 hops when neither PS1 and PS2 drop packet
  - a packet experience 2 hops when S1 forwards packet and S2 drops packet
  - a packet experience 1 hop when S1 drops packet
- **A:** $N = 3 \times (1-p)^2 + 2 \times (1-p)p + 1 \times p$

Example

- p : drop probability of a packet in a packet switch.
- **Q:** How many packets server sends in average to deliver a packet to client (M)?

- **A:** Delivering a packet to client after M packet-transmission by server means:

$$1 = M(1-p)^2$$

- Then: $M = \frac{1}{(1-p)^2}$

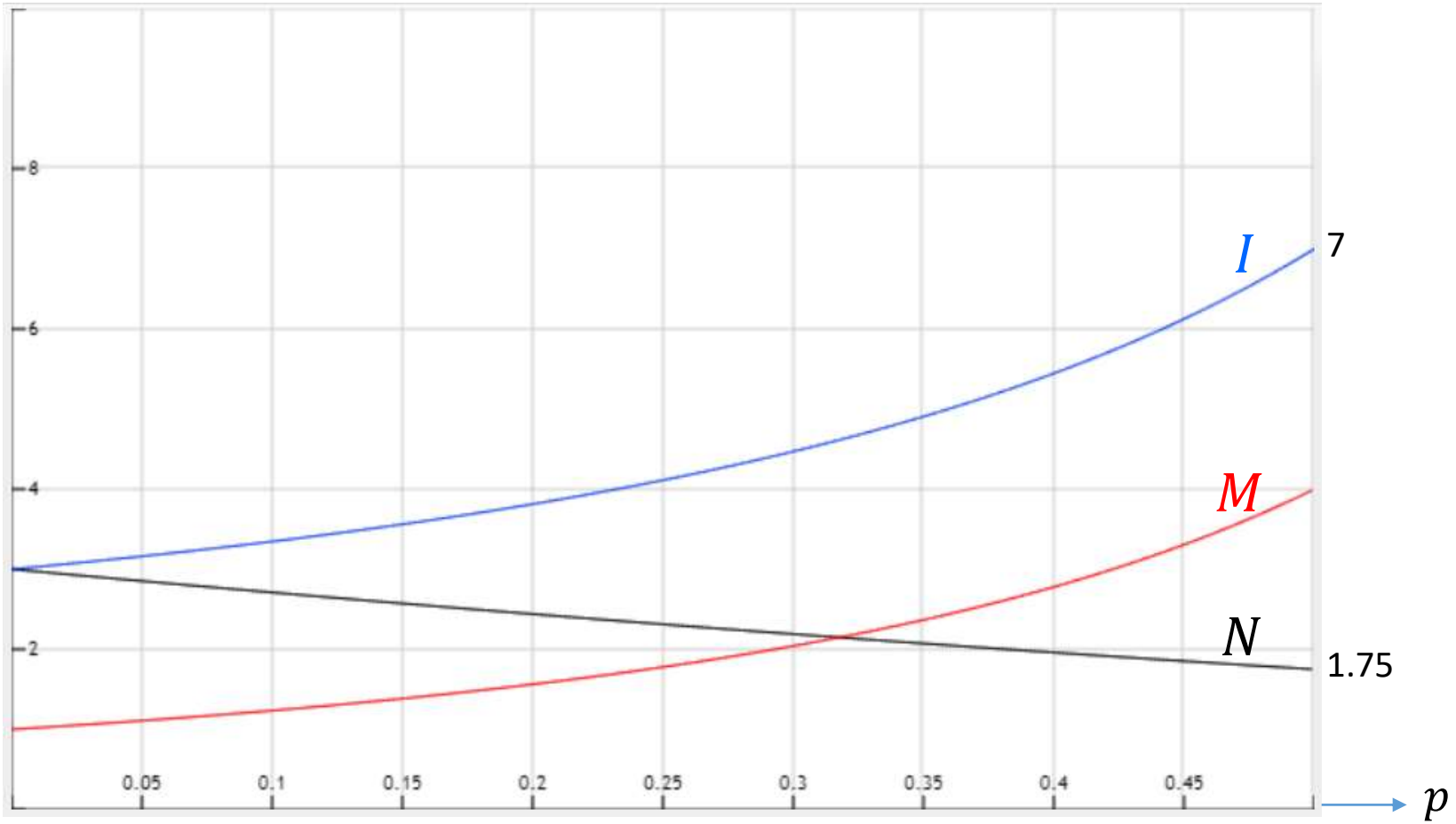
Example

- p : drop probability of a packet in a packet switch.
- **Q:** When a packet is received by client, how many hopes (transmission) happens in average (I)?
- **A:** In average, a packet experience N hops (transmission) to be delivered to client and server sends M packets to deliver a packet to client
- So: $I = N \times M$

$$N = 3 \times (1-p)^2 + 2 \times (1-p)p + 1 \times p$$

$$M = \frac{1}{(1-p)^2}$$

Example



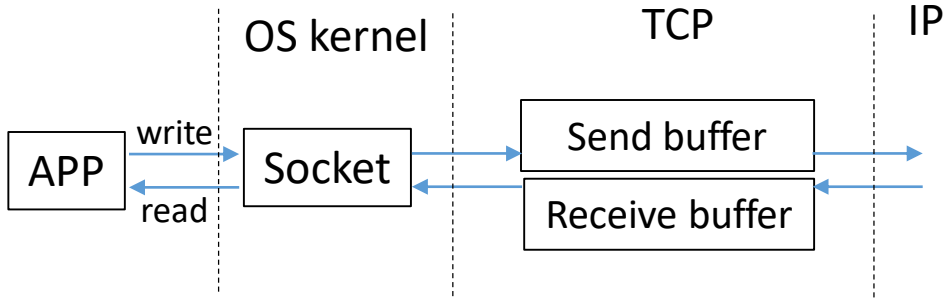
Contents

3.5 Connection-Oriented Transport: TCP

- TCP (Transmission Control Protocol) defined in RFC 793, RFC 1122, RFC 2018, RFC 5681, and RFC 7323
  - Connection-oriented: Handshaking (exchange of control messages) initializes sender, receiver state (parameters) before data exchange
  - Reliable, in-order byte steam (no “message boundaries“), cumulative ACKs
  - Pipelining: Sending window (TCP congestion and flow control set window size)
  - Point-to-point: One server connection socket connected to one client socket
  - Full duplex data: Bi-directional data flow in one connection
  - Flow controlled: Sender will not overwhelm receiver
  - Congestion control: Sender will not overwhelm network

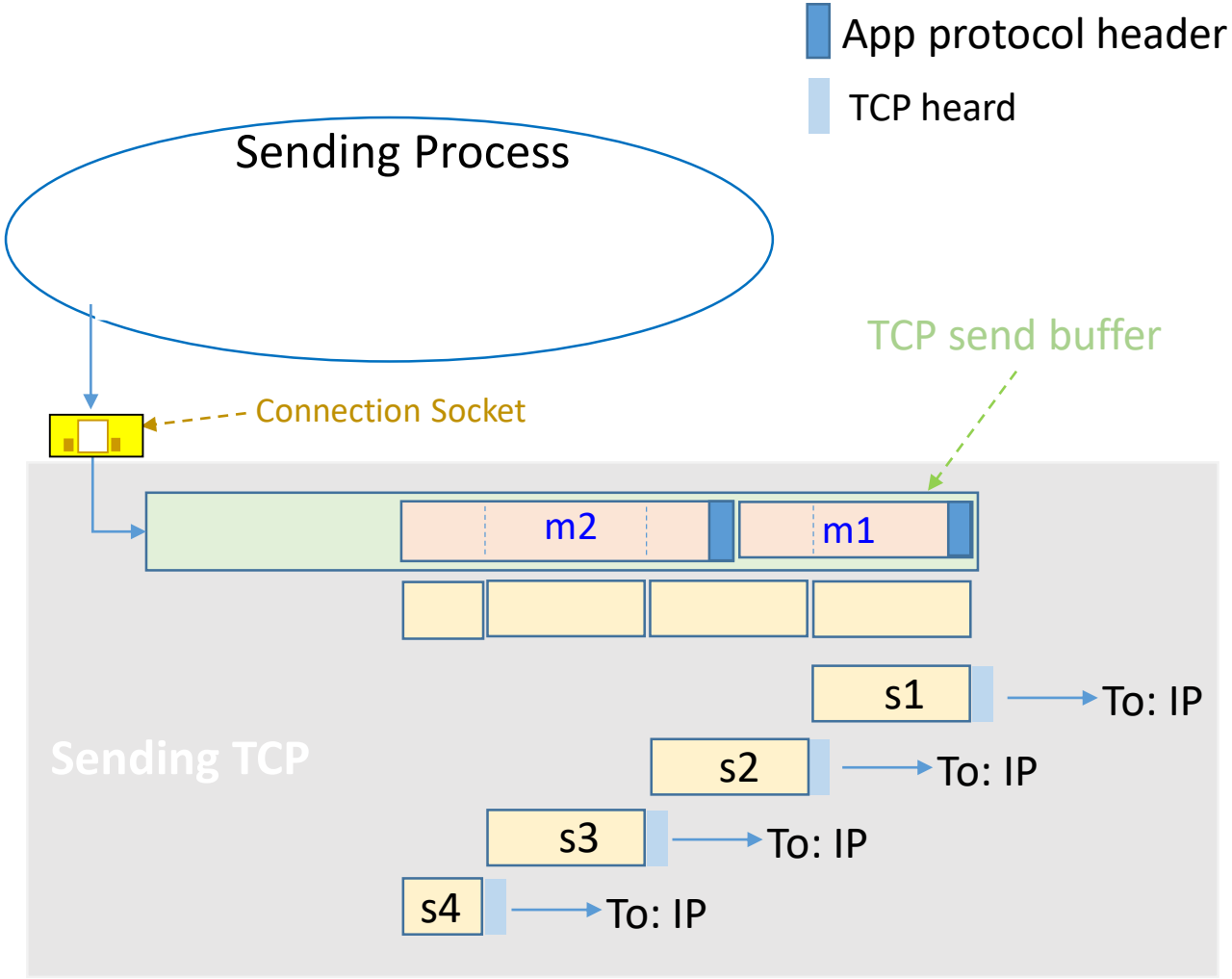
TCP send segments at its own convenience

- Sender process passes a stream of data (objects, files) through socket, then, data is in hands of TCP
- TCP directs this data to connection's send buffer, (one of buffers that is set aside during connection setup for TCP sender by OS)
- TCP will grab chunks of data from send buffer and pass data to IP
- App has no control on when and which data be sent by TCP



No control by sending process

- Byte steam (no “message boundaries“)
- From time to time, TCP, at its own convenience, will grab chunks of data, from TCP send buffer, put its own header on it, and pass data to network layer (IP)
- TCP will remove data from buffer whenever its ACK be in hand of TCP



Maximum Segment Size, Maximum Transmission Unit

- Maximum amount of data that can be grabbed and placed in a segment is limited by maximum segment size (MSS)
- MSS is maximum amount of application-layer data in a segment

- MSS is set by:
  - Largest link-layer frame (maximum transmission unit, MTU),
  - TCP segment including TCP/IP header length (typically 40 bytes) will fit into a single link-layer frame
- Both Ethernet and PPP link-layer protocols have an MTU of 1,500 bytes, thus MSS = 1500 – 40 = 1460Bytes = 365*4

TCP connection setup

- TCP connection socket setup: **B**uffers (receive and send) allocation, **V**ariables setting, and **C**onnection **S**ocket assignment to processes in both hosts
- When TCP receives a segment, segment's data is placed in TCP connection's receive buffer. App reads stream of data from this buffer
- Each side of connection has its own send buffer and its own receive buffer
- In network elements (routers, switches, and repeaters) no buffers or variables are allocated to TCP connections

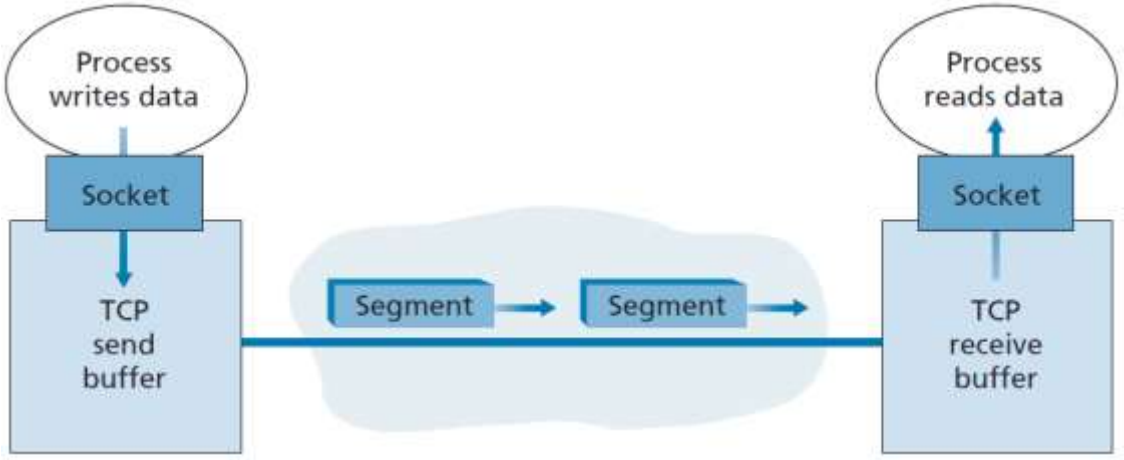
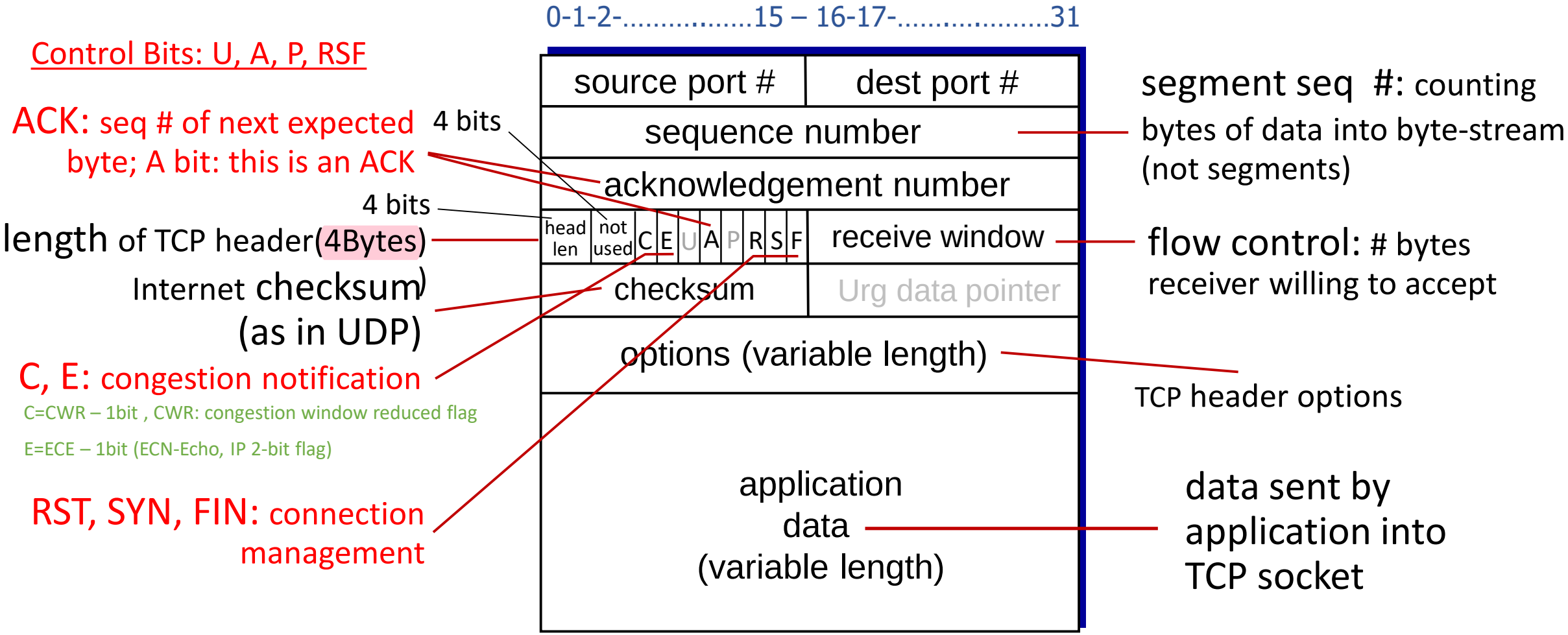


Figure 3.28 TCP send and receive buffers

3.5.2 TCP Segment Structure



TCP segment size and its headers

- Total length of TCP segment must be a multiple of four bytes
- Header length is 4 bits long (1111_2=15_{10}), maximum header length is 15x4=60bytes, mandatory is 20 Bytes, 40 bytes are left for optional items
- TCP header options and their structure:

Type (1 Byte)	Length (1Byte)	Header value
2	4	Maximum Segment Size (2Byte)
3	3	Receive Window Size scale Factor (1 Byte)
8	10	Time Stamp Value (4 Byte), Time Stamp Echo (4Byte)
…	…	…

- If total optional header length is not a multiple of four, it is padded with NOP (no operation-1Byte) options

TCP-header Options

1. Maximum Segment Size (MSS) - 2 bytes
    1. Example: Maximum Ethernet Frame size = 18(Ethernet heard)+20 (IP header)+20 (TCP header)+1460 (data, MSS)
    2. Used during TCP connection setup (SYN and SYN/ACK phase of 3-way-handshake)
2. Receive Window Size scale Factor – 1 Byte
    1. Window scale factor is sent during connection setup in SYN packet
    2. Value of Window scale factor can go up to 2^{255} but maximum allowed is 2^{14}
    3. Reason why maximum value is 14 is because $2^{16} * 2^{14} = 2^{32}$.
        1. If it increases beyond this then it will surpass Sequence Number 2^{32}
3. Time Stamps: Sender local time in Time Stamp value and echo a timestamp value sent by remote TCP (to improve RTT Measurement, protect against wrapped sequence numbers)
4. …

PUSH and URG flags

- PSH=1 indicates that TCP receiver should pass data to upper layer immediately
- URG=1 indicates that there is data in this segment that sending-side has marked as “urgent”
- Location of last byte of this urgent data is indicated by 16-bit urgent data pointer field
- TCP must inform upper layer when urgent data exists and pass it a pointer to end of urgent data
- In practice, PSH, URG, and urgent data pointer are not used

TCP checksum

- Checksum comes from adding (1's complete) 16 bit words of following items:
- TCP Header, TCP body and Pseudo IP header
- Pseudo IP header:
  - Source IP Address, 32bits
  - Destination IP Address, 32bits
  - Length, 16bits
  - Upper Layer, 8bits
  - Time to Live, 8bits

  TOTAL = 6*16 bits

- TCP Checksum=6*16Pseudo IP header+ m*16TCP header + n*16 TCP Body

m>=10

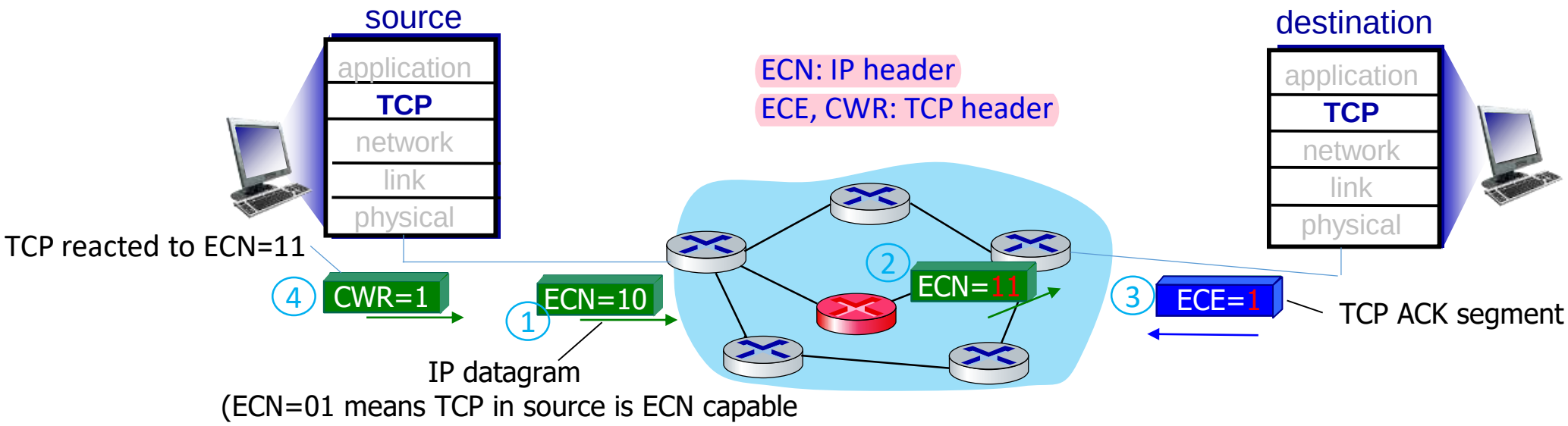
Explicit Congestion Notification (ECN)

- TCP detects congestion when a packet experiences:
  - Retransmit Timeout, or
  - 3 duplicate ACKs
- ECE flag: ECN-Echo
  - Destination receives a packet with ECN=11, set ECE=1 (echoes congestion information to sender transport layer)
- CWR flag: Congestion Window Reduced
  - Sender receives a packet with ECE=1, starts congestion control and set CWR=1 to indicate that sender has reacted to congestion
  - TCP sender's reaction is same as its congestion control response to a packet
- ECN flags
  - A router in early state of congestion, sets ECN=11 in packets forwarded to destination
  - ECN flags: last 2 bits of TOS part in IP header

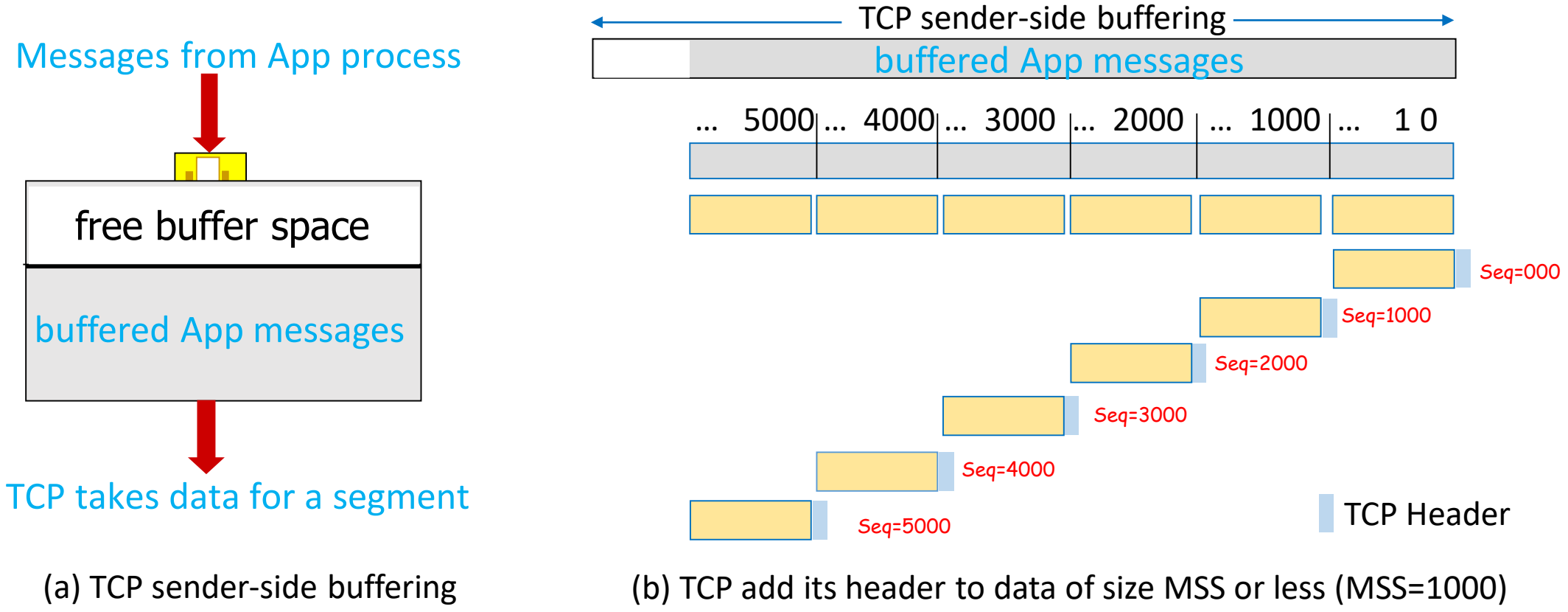
Explicit congestion notification (ECN)

TCP often implement network-assisted congestion control:

- ECN: 2bits in IP header (TOS field) marked by network router to indicate congestion
- Congestion indication carried to destination
- Destination sets ECE bit on ACK segment to notify sender of congestion
- Involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



Sequence Numbers Numbers



(a) TCP sender-side buffering

(b) TCP add its header to data of size MSS or less (MSS=1000)

Figure 3.30 TCP views data as an unstructured, but ordered, stream of bytes

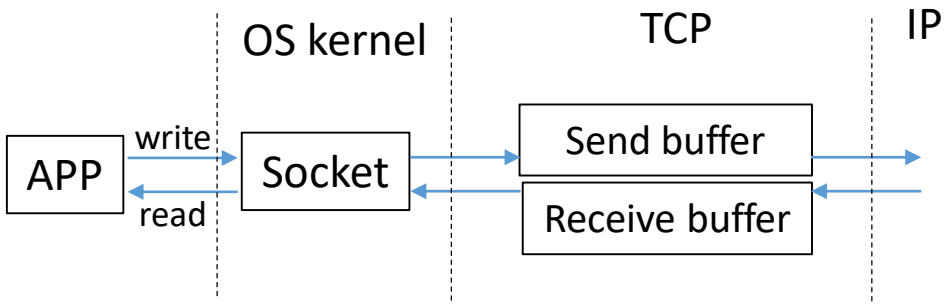
Initial sequence numbers

- In Figure 3.30, we assumed that initial sequence number is zero
- In truth, both sides of a TCP connection randomly choose an initial sequence number
- This is done to minimize possibility that a segment that is still present in network from an already-terminated connection between two hosts is mistaken for a valid segment in a current connection between these same two hosts (which also happen to be using same port numbers as terminated connection)

Acknowledgment Numbers

Acknowledgements

- Sequence number of next byte expected from other side
- Cumulative ACK



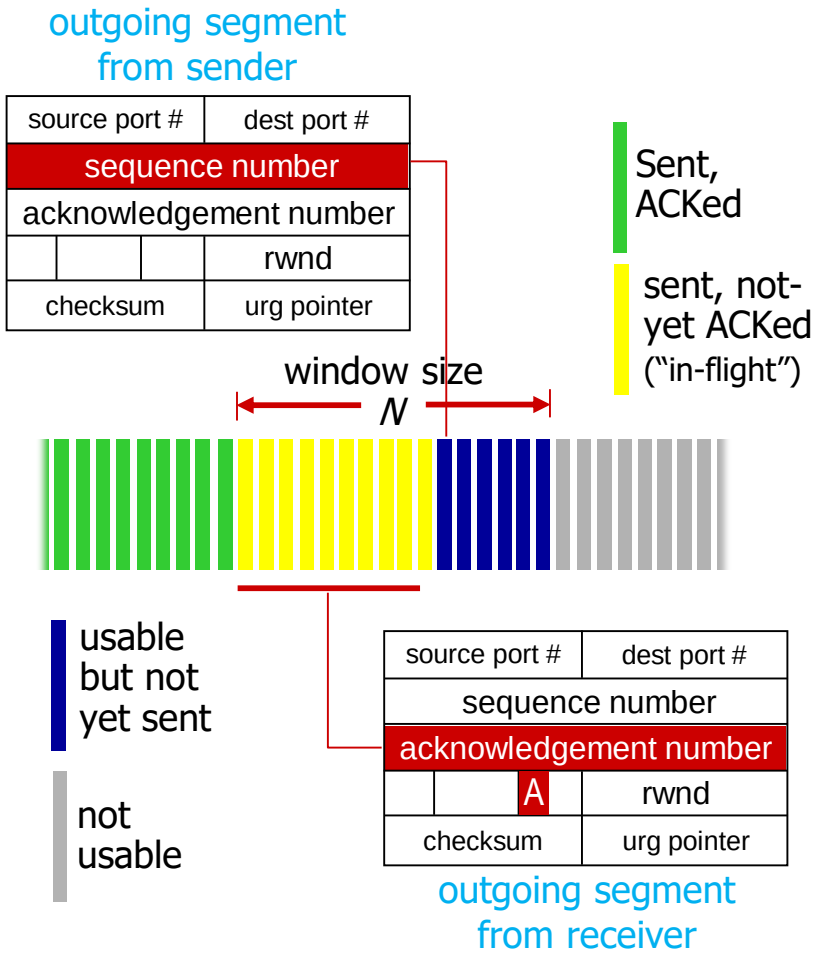
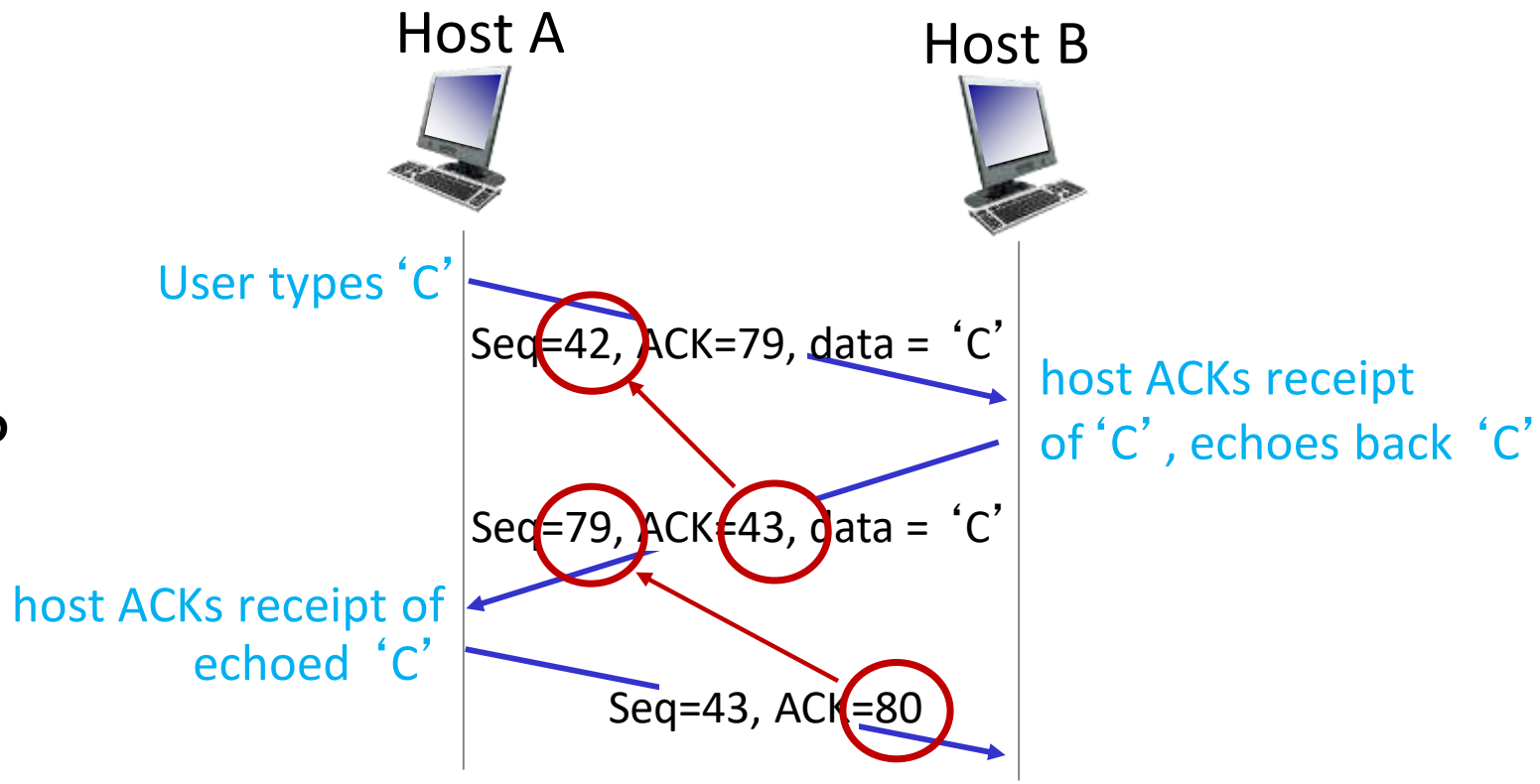
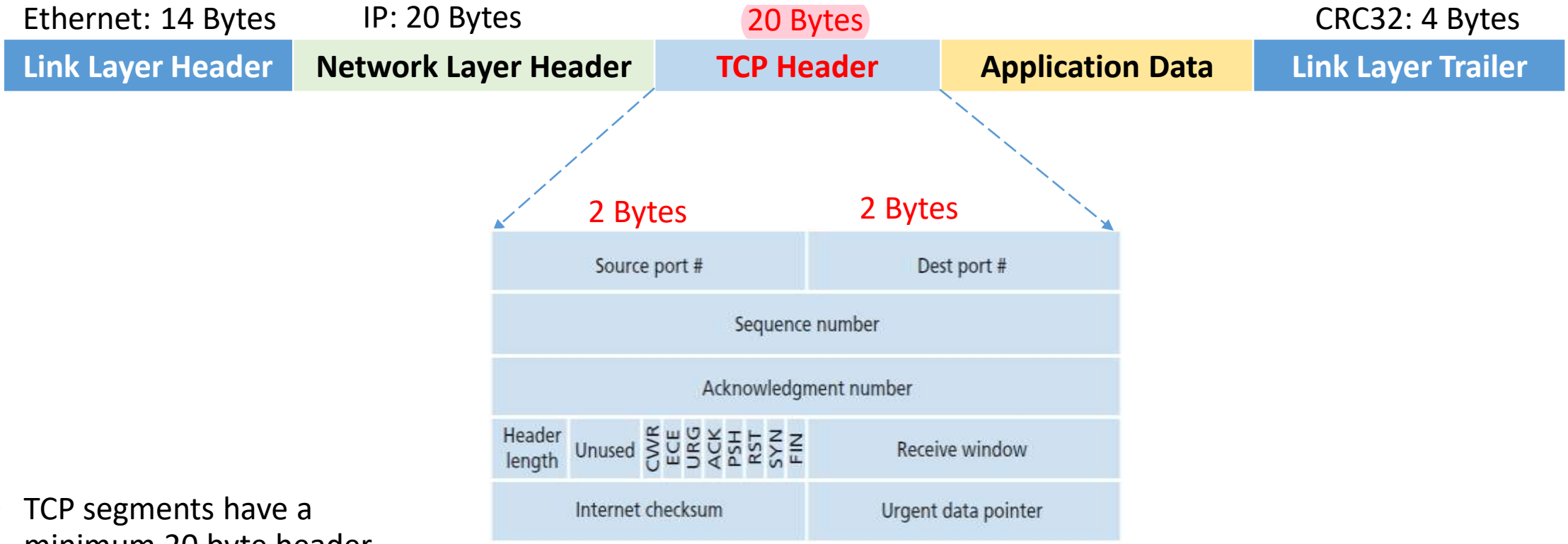


Figure 3.31

- Sequence and acknowledgment numbers for a simple Telnet application over TCP



TCP in action



- TCP segments have a minimum 20 byte header with ≥ 0 bytes of data

3.5.3 Round-Trip Time Estimation and Timeout

- TCP uses a Retransmit TimeOut mechanism (RTO) to recover from lost segments. Clearly, RTO> RTT
- **Q**: How TCP sets length of timeout intervals RTO? How much larger than RTT?
- **Q**: How should RTT be estimated in first place?
- **A**: [RFC 6298]
  - 1- TCP measures RTT (`SampleRTT`) whenever receives an ACK, at time `t`
  - 2- TCP, then updates Exponentially Weighted Moving Average (EWMA) of `SampleRTTs`

EWMA of `SampleRTTs` up to time `t=` $EstimatedRTT(t) = (1-\alpha) * EstimatedRTT(t-1) + \alpha * SampleRTT(t)$

- Recommended value of α is $\alpha = 0.125$ [RFC 6298]

Variability of RTT (Deviation RTT): DevRTT

- 3- Calculate variability of RTT. [RFC 6298] defines RTT variation, DevRTT:

$$DevRTT(t) = (1-\beta) * DevRTT(t-1) + \beta * |SampleRTT(t) - EstimatedRTT(t)|$$

- DevRTT is an EWMA of difference between SampleRTT and EstimatedRTT
- If SampleRTT values have little fluctuation, then DevRTT will be small, if there is a lot of fluctuation, DevRTT will be large
- Recommended value of β is 0.25

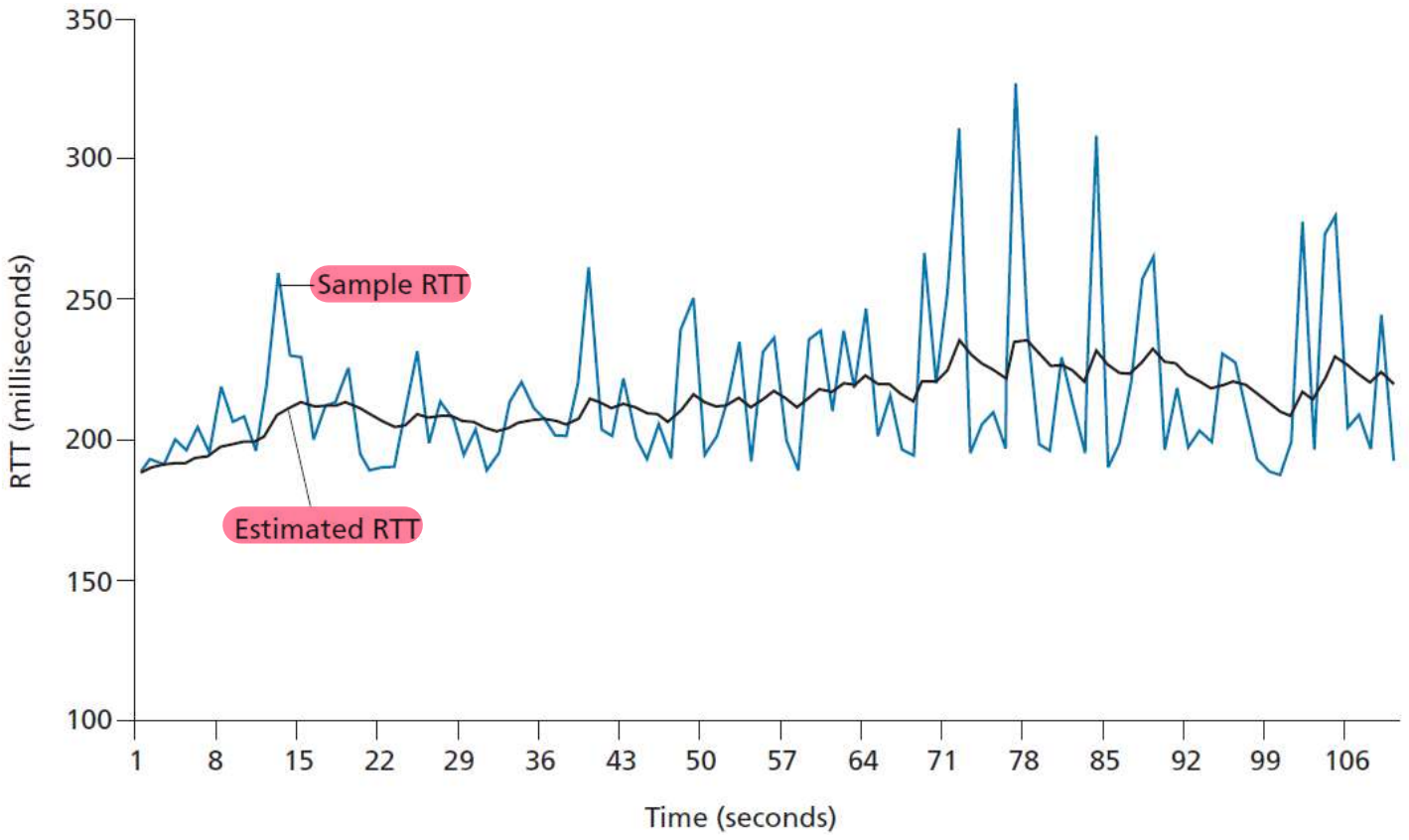
Exponential weighted moving average

- `SampleRTT` values will fluctuate from segment to segment due to congestion in routers and to varying load on end systems
- `EstimatedRTT` is a weighted average of `SampleRTT`. This puts more weight on recent samples than on old samples

- In statistics, such an average is called an exponential weighted moving average (EWMA)
- "exponential" means, weight of a given `SampleRTT` decays exponentially fast as updates proceed

Figure 3.32 RTT samples and RTT estimates

- `SampleRTT` values and `EstimatedRTT` for a value of α=1/8 for a TCP connection between gaia.cs.umass.edu and fantasia.eurecom.fr
- Variations in `SampleRTT` are smoothed out in computation of `EstimatedRTT`



Setting and Managing Retransmission Timeout Interval (RTO)

- TCP's method for determining retransmission timeout interval (RTO):

```
RTO=TimeoutInterval=EstimatedRTT+4*DevRTT
```

- An initial Timeout Interval value of 1 second is recommended [RFC 6298]
- When a timeout occurs, value of RTO is doubled to avoid a premature timeout occurring for a subsequent segment that will soon be acknowledged
  - As soon as an ACK is received, `EstimatedRTT` is updated and RTO is again computed using formula above

3.5.4 Reliable Data Transfer

- TCP creates a reliable data transfer service on top of IP's unreliable best effort service
- In TCP receiver: A process reads data out of TCP receive-buffer, so that
  - uncorrupted
  - without gaps
  - without duplication
  - in sequence
- Received byte stream is exactly same byte stream that was sent by TCP in sending host

TCP Sender (simplified)

- There are three major events related to data transmission and retransmission in TCP sender:
  - 1- data read from TCP send buffer
  - 2- RTO timer timeout
  - 3- ACK receipt

1- data read from TCP send buffer

- Encapsulates data in a segment, and passes segment to IP
- If RTO timer is already not running for some other segment, TCP starts timer when segment is passed to IP. (It is helpful to think of RTO timer as being associated with oldest unacknowledged segment)

TCP Sender (simplified)

2- RTO timer timeout

- TCP responds to RTO timeout event by retransmitting segment that caused timeout. TCP then restarts RTO timer

3- ACK receipt

- TCP compares ACK value y with its SendBase. (SendBase is sequence number of oldest unacknowledged byte)
- TCP uses cumulative acknowledgments:
  - ACK number = y: It acknowledges receipt of all bytes before byte number y
  - If y>SendBase, then ACK is acknowledging one or more previously unacknowledged segments
    - Sender updates its SendBase variable; it also restarts RTO timer if there are any not-yet-acknowledged segments

How many RTO timers?

- **Q**: Is there an individual RTO timer is associated with each transmitted but not yet acknowledged segment?
- **A**: This is great in theory, but timer management can require considerable overhead
- Recommended TCP RTO timer management procedures [RFC 6298] use only a single RTO timer, even if there are multiple transmitted but not yet acknowledged segments (RTO timer is associated with oldest unacknowledged segment)

A Few Interesting Scenarios

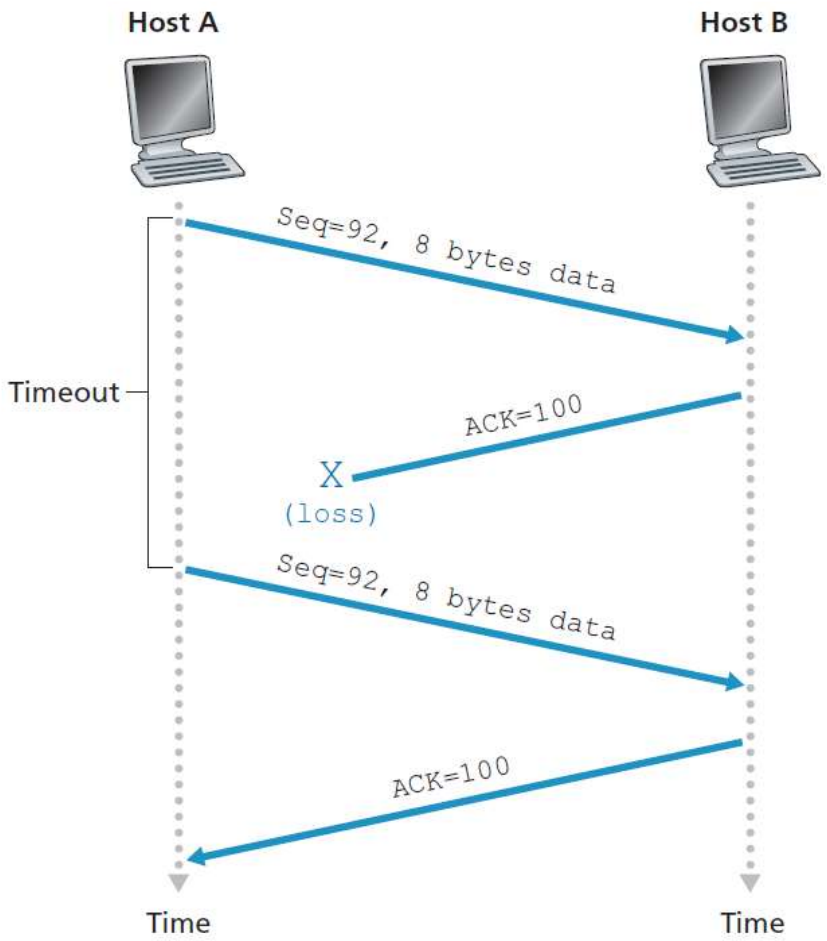


Figure 3.34 Retransmission due to a lost acknowledgment

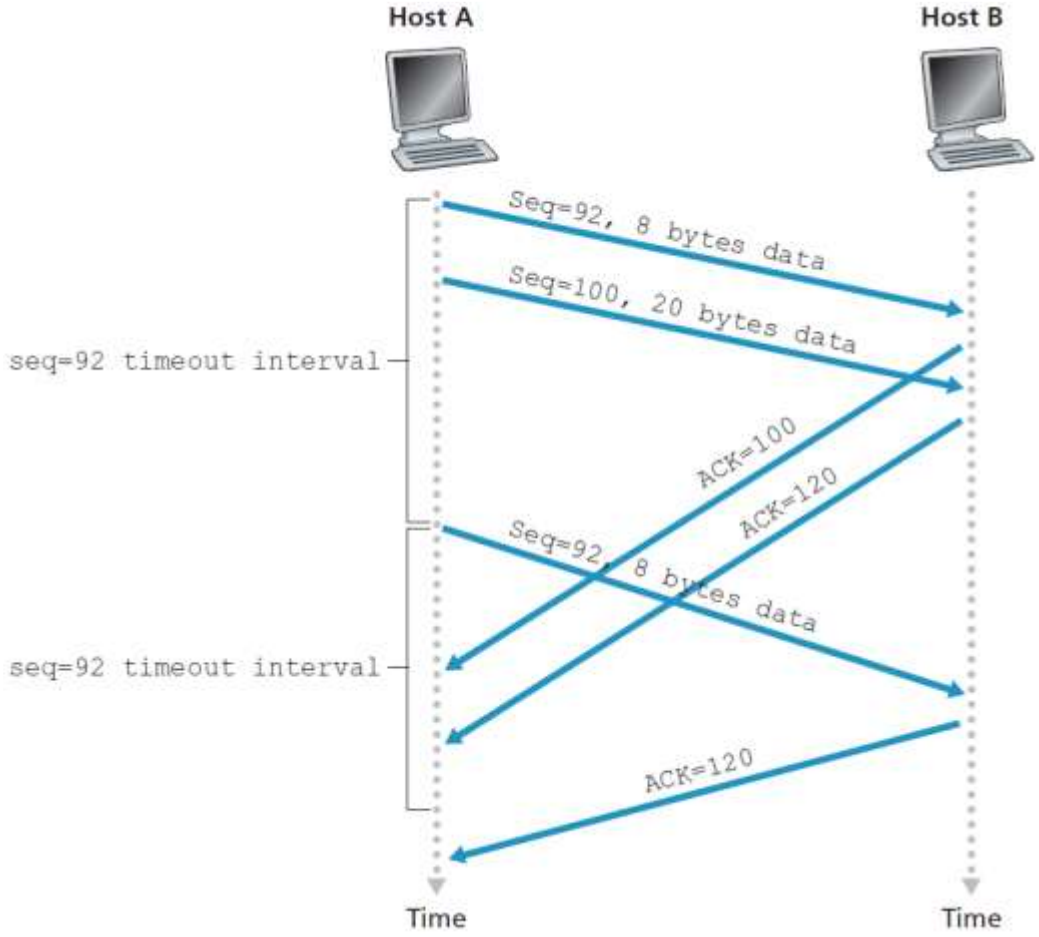
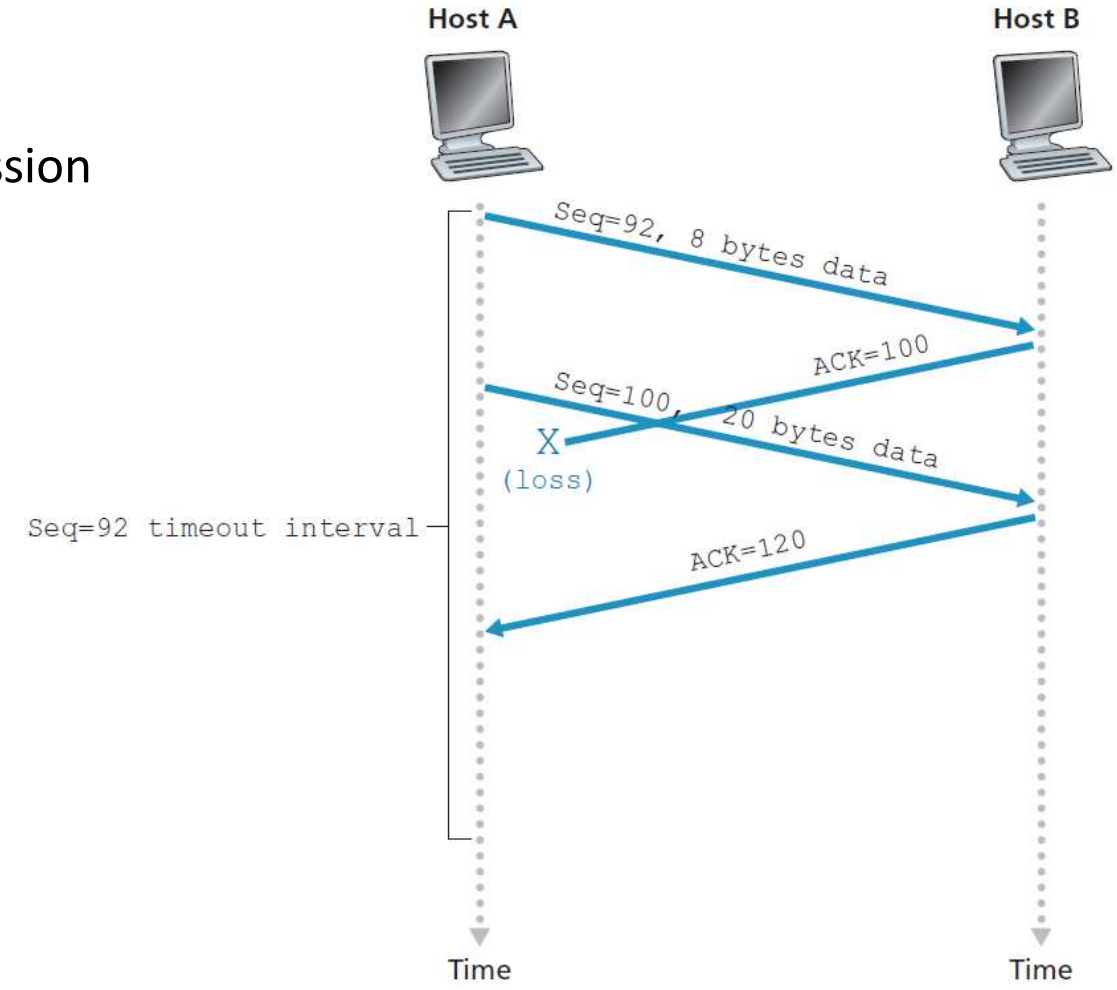


Figure 3.35 Segment 100 not retransmitted

A Few Interesting Scenarios

- **Figure 3.36** A cumulative acknowledgment avoids retransmission of the first segment



Doubling the Timeout Interval

- On RTO timeout event: TCP retransmits not-yet-acknowledged segment with smallest sequence number
- Each time TCP retransmits, it sets next RTO interval to twice previous value, rather than deriving it from last `EstimatedRTT` and `DevRTT`
  - Timer expiration is most likely caused by congestion in network, so doubling RTO helps reducing congestion
- Example: Suppose RTO=0.75sec (for oldest not yet acknowledged segment)
- If timer expires, TCP retransmit this segment and set new RTO=1.5sec. If timer expires again 1.5 sec later, TCP will again retransmit this segment, now setting RTO=3.0 sec

TCP ACK Generation – Receiver Action

Table 3.2 TCP ACK Generation Recommendation [RFC 5681]

Event	TCP Receiver Action
Arrival of in-order segment with expected sequence number and All data up to expected sequence number already acknowledged	Delayed ACK. Wait up to 500msec for arrival of another in-order segment. If next in-order segment does not arrive in this interval, send an ACK
Arrival of in-order segment with expected sequence number and 1 in-order segment waiting for ACK transmission	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment with higher-than-expected sequence number (Gap detected)	Immediately send duplicate ACK, indicating sequence number of next expected byte (lower end of gap)
Arrival of segment that partially or completely fills in gap in received data	Immediately send ACK, provided that segment starts at lower end of gap

When receiver sends a duplicate ACK

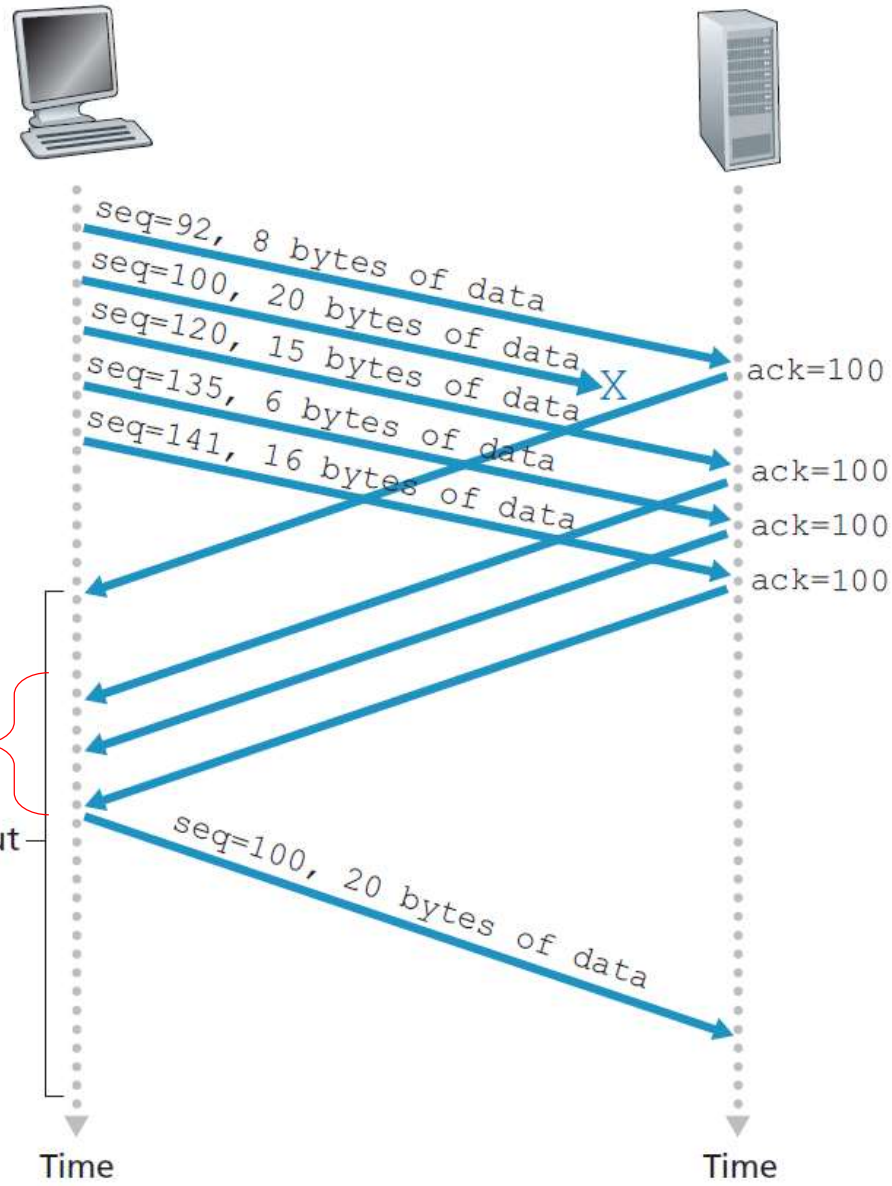
- A duplicate ACK is an ACK that reacknowledges a segment for which sender has already received an earlier acknowledgment

- When a TCP receiver detects a gap in data stream (missing segment)
  - This gap could be result of lost or reordered segments within network
  - TCP receiver reacknowledges (generates a duplicate ACK for) last in-order byte of data it has received

Fast retransmission

- Sender often sends a large number of segments back to back, if one segment is lost, there will likely be many back-to-back duplicate ACKs
- If TCP sender receives three duplicate ACKs for same segment, it takes this as an indication that segment following segment that has been ACKed three times has been lost
- Then TCP sender does not wait for timeout. It retransmits missing segment. This is called Fast Retransmit [RFC 5681]

Figure 3.37



3 duplicate ACKs indicates 3 segments received after a missing segment – lost segment likely

First duplicate ACK for segment 100

Second duplicate ACK for segment 100

Third duplicate ACK for segment 100

Figure 3.37 Fast retransmit: retransmitting missing segment before segment's timer expires

Does TCP use Go-Back-N or Selective Repeat?

- Consider sender sends a window of segments $1, 2, \ldots, N$

1. When all of segments arrive in order without error at receiver, and ACK for packet n gets lost ($n \leq N$), but ACKs for remaining $N-1$ segments arrive at sender before timeouts
   - TCP-sender: Retransmitting at most segment n. TCP would not even retransmit segment n if acknowledgment for segment $n+1$ arrived before timeout for segment n (cumulative ACK)
     - GBN: Retransmitting segments $n, n+1, n+2, \ldots, N$
2. When multiple sent segments are lost from one window of segments, a TCP sender can only learn about a single lost packet
   - RFC 2018 (1996) proposed "Selective Acknowledgment (SAK) mechanism"

Selective Acknowledgment (SACK) mechanism

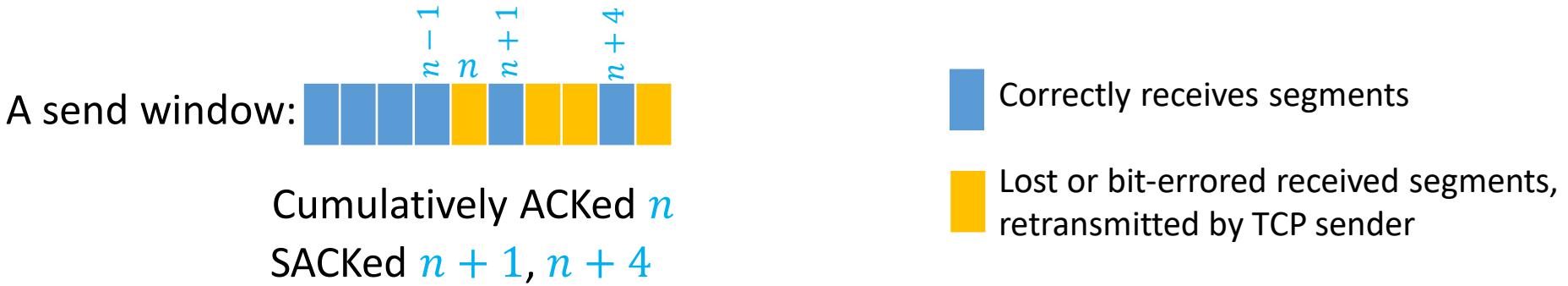
- Sack-Permitted Option: A two-byte control data is sent in optional header of a SYN segment (see Figure 3.39) by a TCP that likes to receive SACK segments once connection has opened

- SACK segment:
- Receiving TCP sends back a SACK segment to TCP sender informing sender of received data. TCP sender can then retransmit only missing data segment
  - SACK segment contains received non-contiguous blocks of data that are queued at TCP receiver

Selective repeat of non cumulatively ACKed Segments

Sender skips retransmission of:

- SACKed segments (received out-of-order segments)
- Cumulatively ACKed segments (received in-order segment)



- TCP's error-recovery mechanism is probably best categorized as a hybrid of GBN and SR protocols

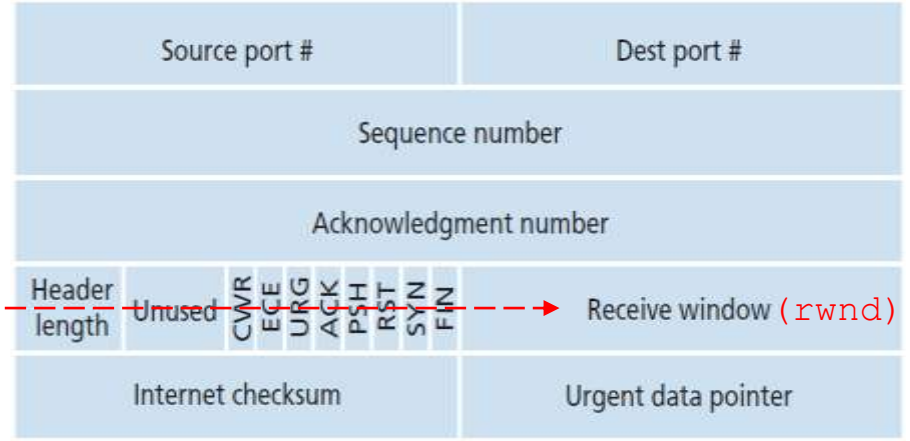
3.5.5 Flow Control

- In a TCP connection, if receiving App is relatively slow at reading data, sending App can overflow receive buffer by sending too much data too quickly

- TCP's flow-control service**:** Eliminating possibility of sender overflowing receiver's buffer. TCP sender throttled due to **low free space in receiver's buffer**
- Note: Also, TCP sender can be throttled due to **congestion within IP network**; this form of sender control is referred to as congestion control
- Actions taken by flow and congestion control are similar (throttling of sender), but reasons are different
- For simplicity, we suppose throughout section 3.5.5 that TCP implementation is such that TCP receiver discards out of order segments

TCP receiver buffer

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via **socket** options (typical default is 4096 bytes)
- Many OSs auto adjust **`RcvBuffer`**
  - Sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
  - Guarantees receive buffer will not overflow



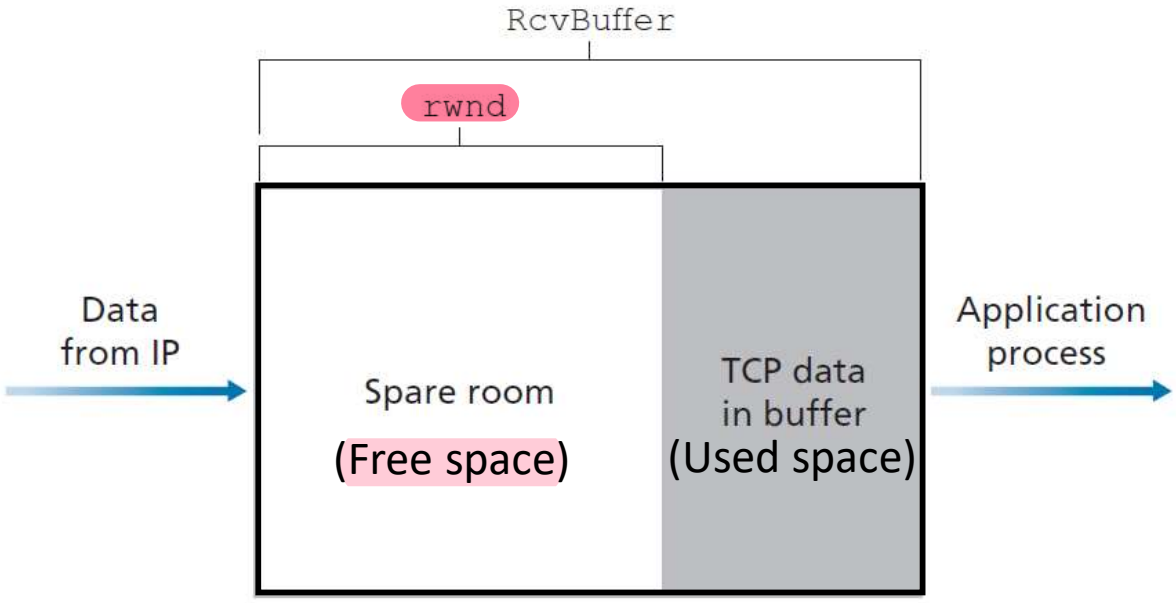


Figure 3.38 The receive window (rwnd) and receive buffer (RcvBuffer)

TCP receiver buffer

- TCP sender maintain a variable called `rwnd`=receive window (free buffer space available at receiver)
- TCP is full-duplex, sender at each side of connection maintains a distinct `rwnd` variable
- Initially, `rwnd = RcvBuffer`
- Suppose in a particular time `rwnd = 0`. TCP continues sending segments with one data byte. These segments will be acknowledged by receiver with updated `rwnd`. Eventually buffer will begin to empty and acknowledgments will contain a nonzero `rwnd` value

3.5.6 TCP Connection Management

1. Client TCP sends a special TCP segment, contains no application-layer data and flags SYN=1, ACK=0, (SYN segment), and a randomly chooses sequence number (client initial sequence number, `client_isn`)
   - Randomizing `client_isn` is to avoid certain security attacks
2. Server creates a connection socket, allocates **TCP buffers and variables**, and sends a connection-granted segment to client TCP containing no application-layer data and SYN=1, ACK=1, ACK#=`client_isn+1`, and a randomly chosen `server_isn` (SYNACK segment)
3. Client **allocates buffers and variables**. Client then sends a segment to ACK server's connection-granted segment by putting ACK#=`server_isn+1` in acknowledgment field of TCP segment header, and flags SYN=0, ACK=1
   - This segment may carry client-to server data (APP data or request) in payload

Figure 3.39 three-way |

- After step 2, in each of segments, flag SYN=0

- Connection variables at client and server:
  - **client_isn**
  - **server-isn**
  - **rcvBuffer size**

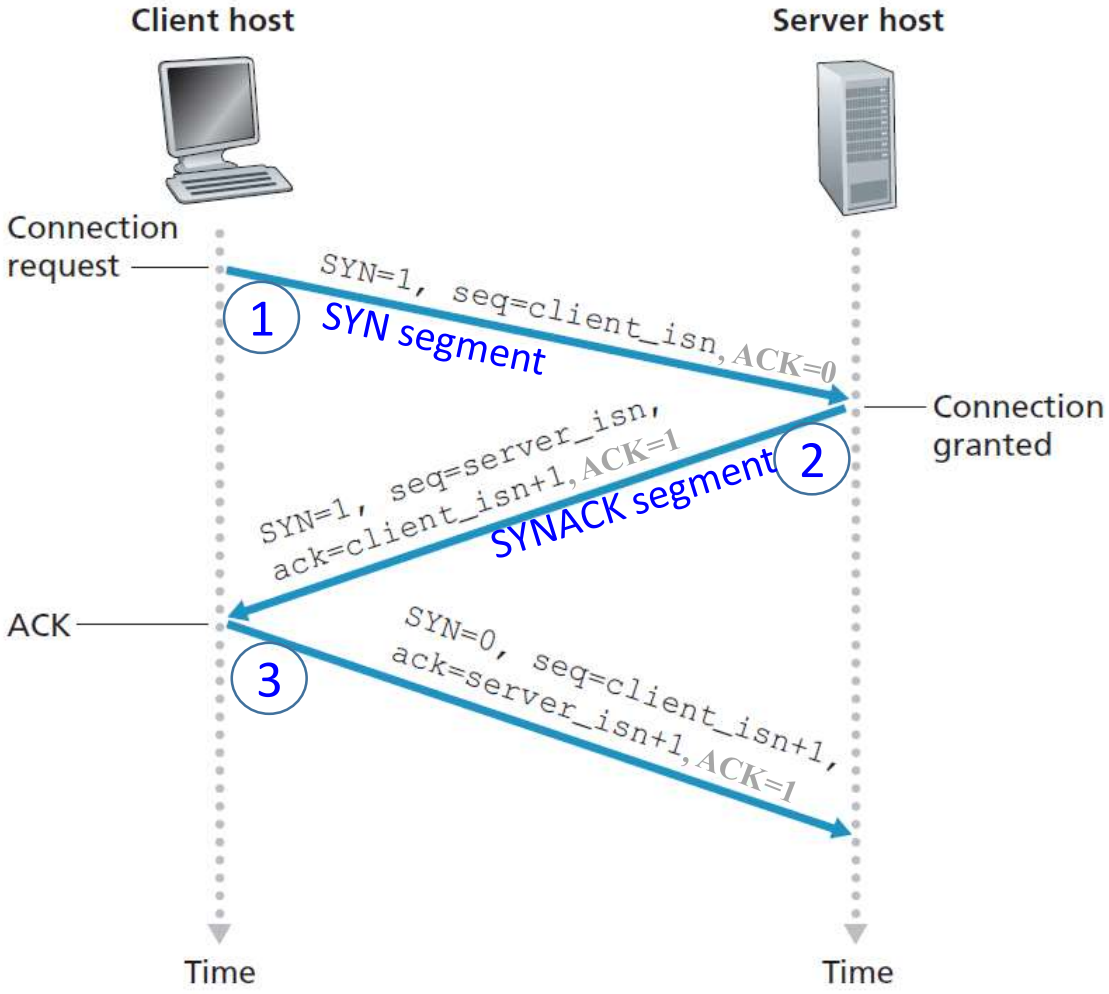
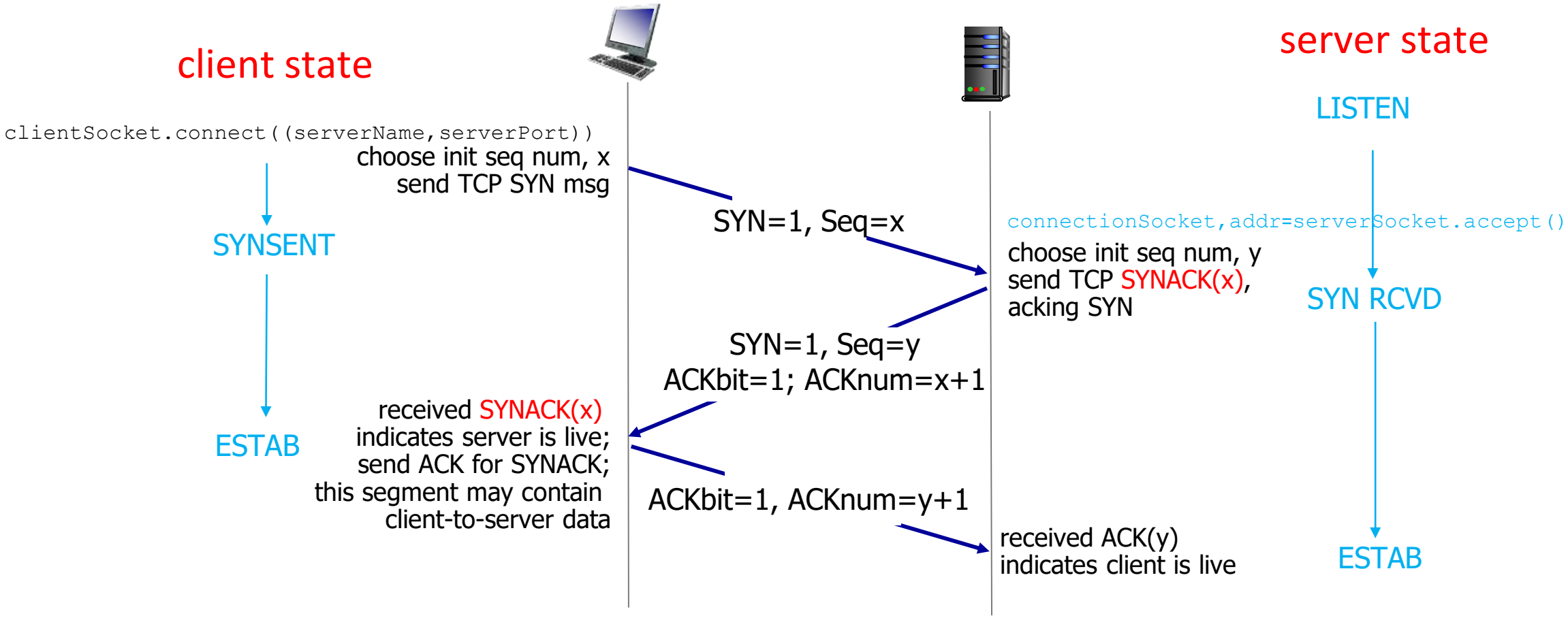


Figure 3.39 TCP three-way handshake: segment exchange

Figure 3.39 three-way handshake

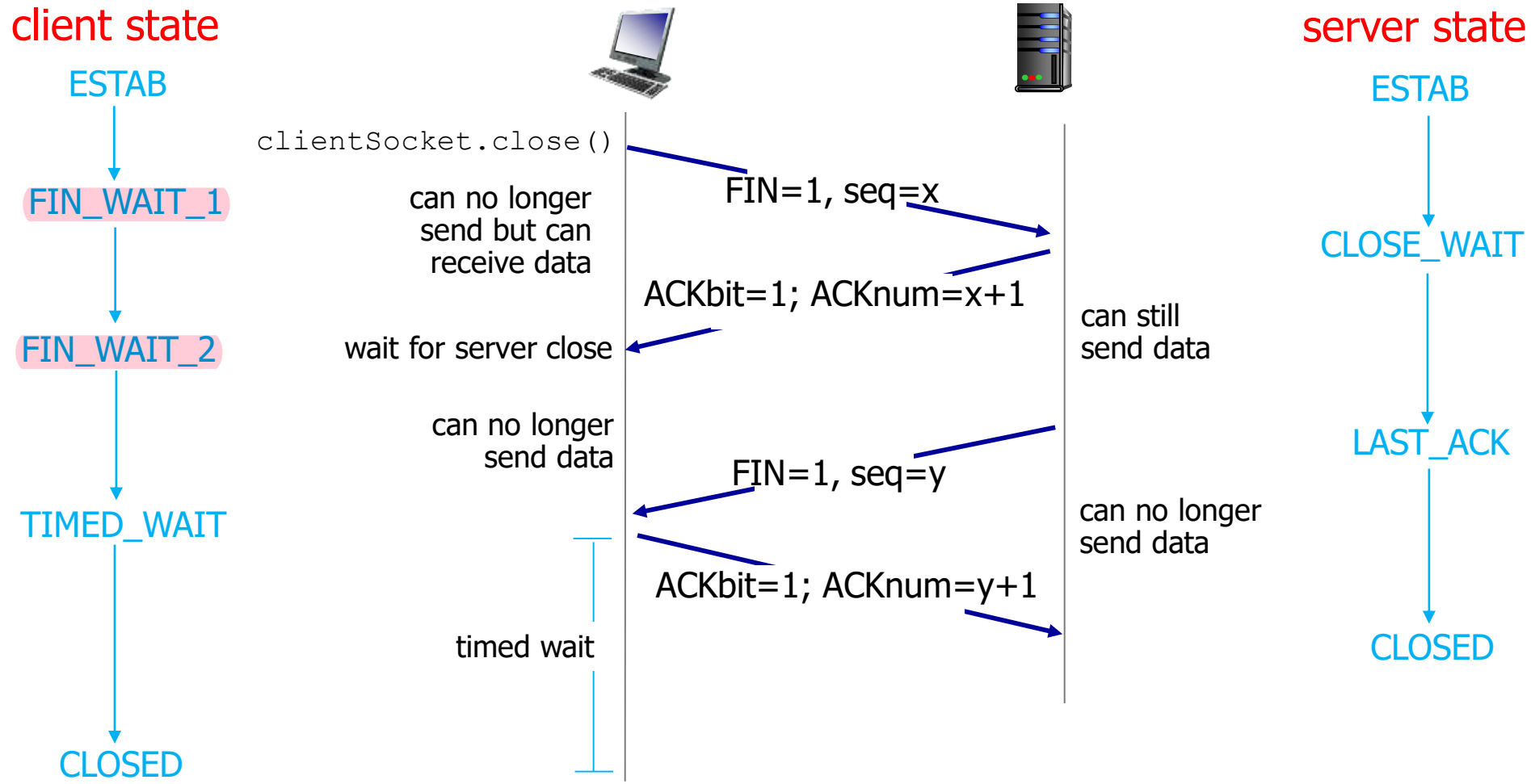


Closing a TCP connection

- Either of two processes participating in a TCP connection can end connection
- When a connection ends, “resources” (buffers an variables) in hosts are deallocated

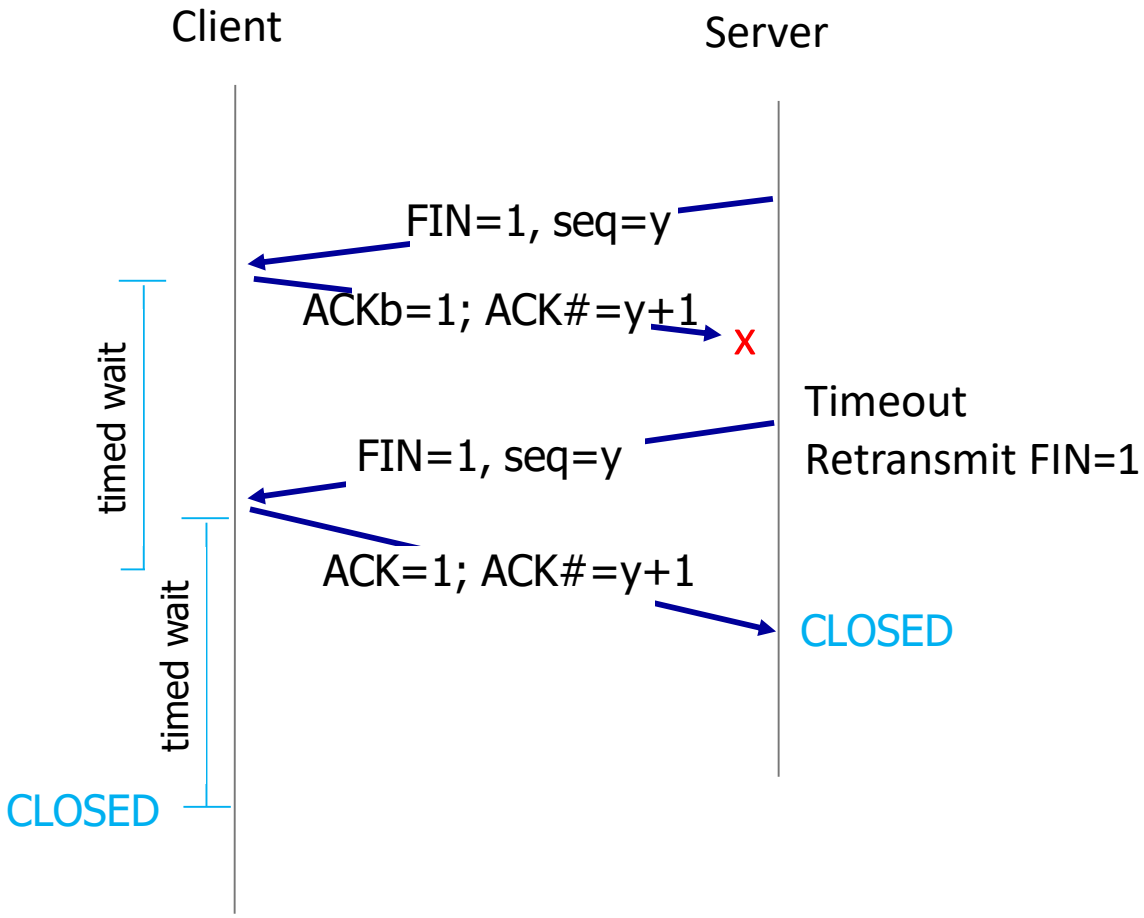
- Example: client decides to close connection, Figure 3.40
  1. Client application process issues a close command, a segment with flag FIN=1 and no payload
  2. When server receives this segment, sends an acknowledgment
  3. Server then sends its own shutdown segment, which has flag FIN=1
  4. Finally, client acknowledges server’s shutdown segment. At this point, all resources in two hosts are now deallocated

Figure 3.40 Closing a TCP connection



Timed wait

- Timed wait lets TCP client resend final acknowledgment in case its ACK is lost
- Timed wait is implementation-dependent, but typical values are 30 seconds, 1 minute, and 2 minutes



Contents

3.6 Principles of Congestion Control

- Why congestion occurs?
- What is cost of congestion?
  - In a path from sender to receiver links and routers that are located after a congested link are not fully utilized
  - Poor performance received by end systems (long delay, packet loss)
- How control or avoid congestion?

- 3 increasingly complex scenarios in next slides

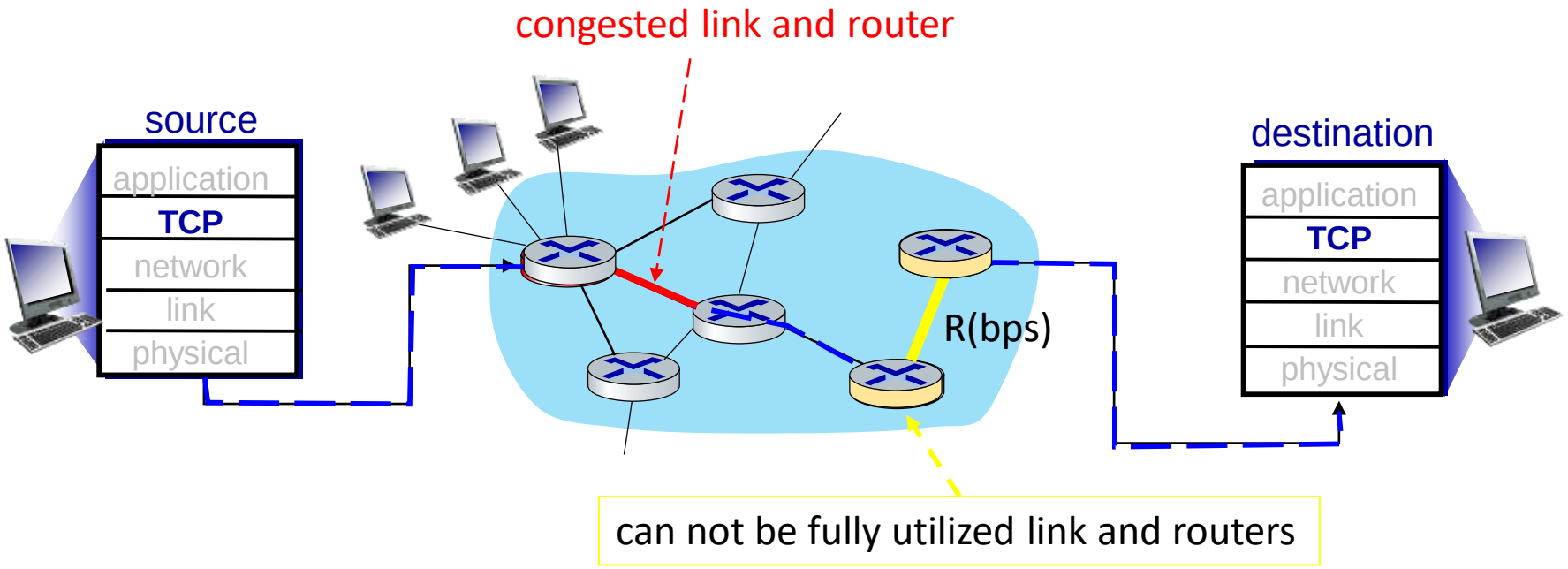
3.6.1 The Causes and the Costs of Congestion

Congestion cause:

- "Too many sources sending too much data too fast for network to handle"
- Congested link: Traffic (bps) passing a link reaches near its capacity R

Congestion costs:

1. Long delays (queueing in router buffers)
2. Packet loss (buffer overflow at routers)
3. Underutilized resources



Congestion Control

Congestion control:

- Different from flow control
- Congestion control approaches:
  - To avoid congestion
  - React to congestion
- It is high on “top-ten” list of fundamentally important problems in networking

Scenario 1 – no retransmission

APPs in Host A and Host B are sending data (λ_{in} bps or segments per second) to C and D respectively

λ_{in} original data (APP data is sent into socket only once), λ_{out} = App throughput

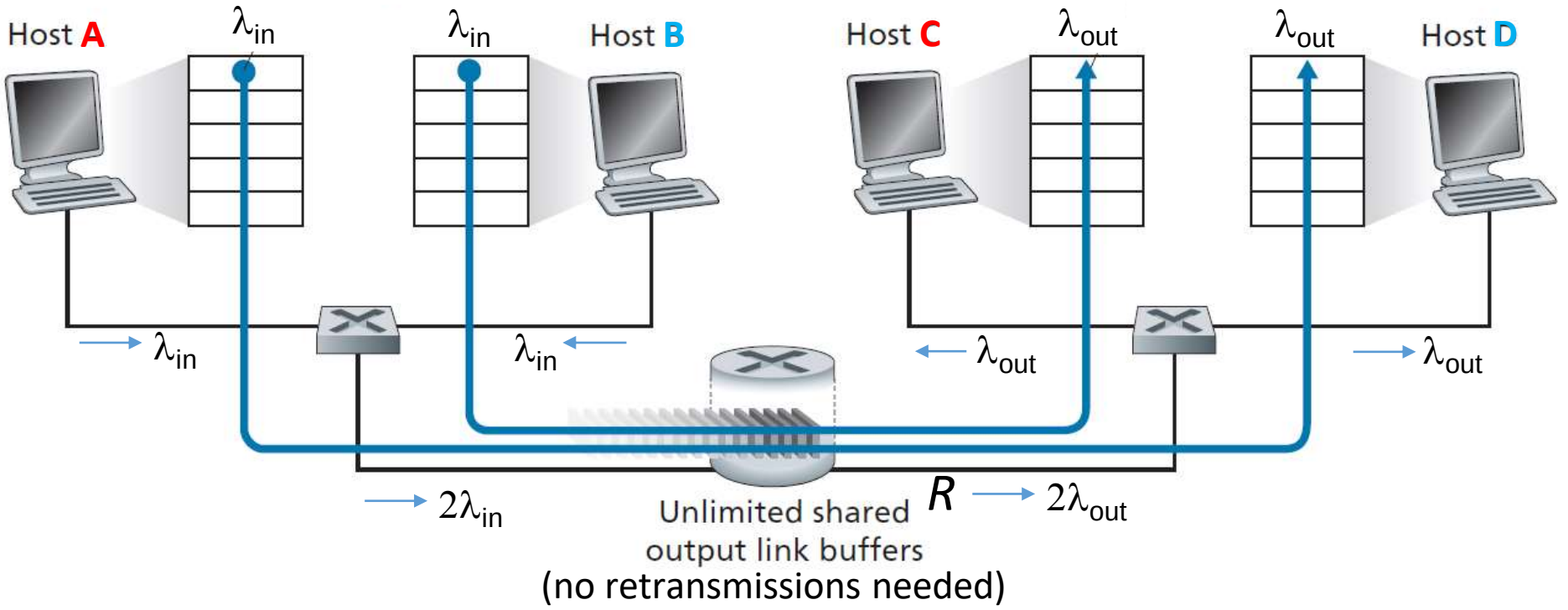


Figure 3.43 Congestion scenario 1: Two connections sharing a single hop with infinite buffers

Per-connection throughput and delay

- One cost of a congested network: large queuing delays are experienced as packet-arrival rate nears link capacity

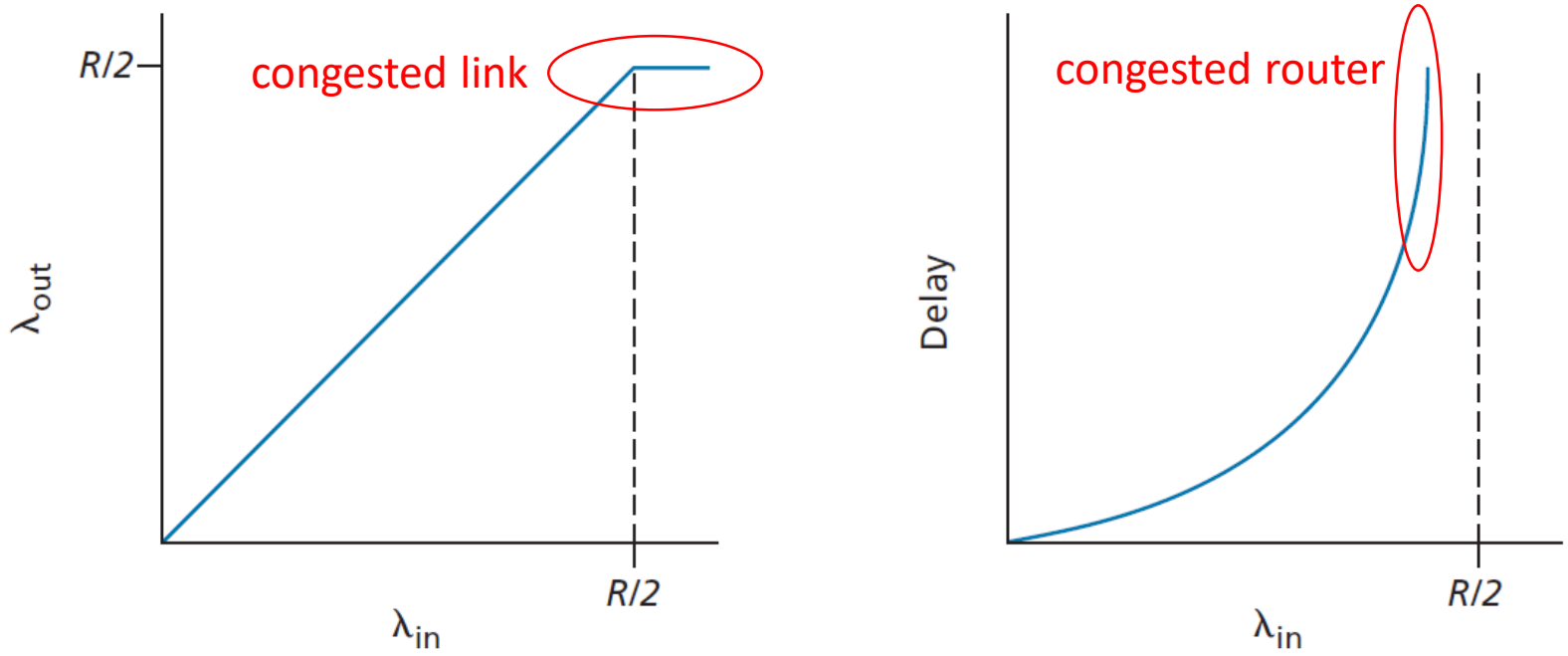
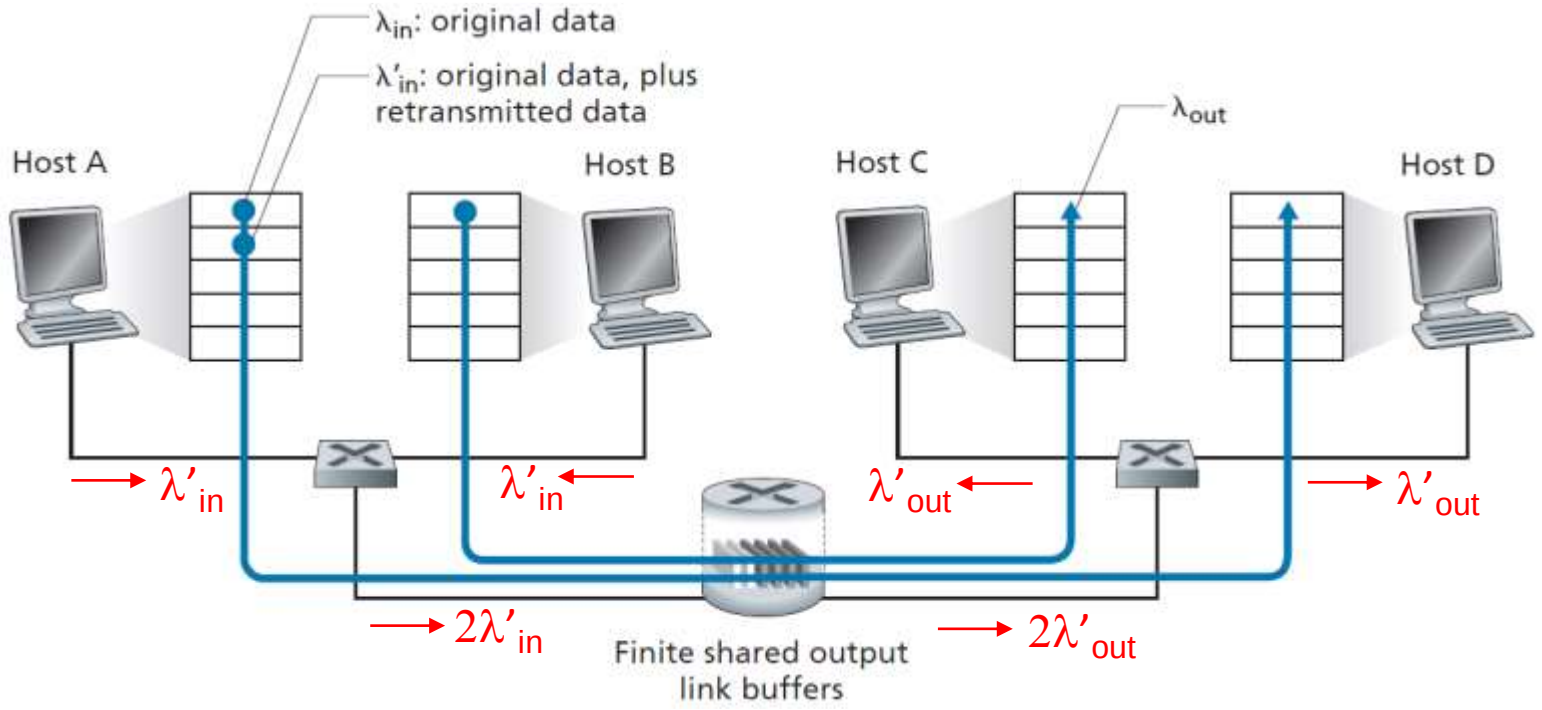


Figure 3.44 Congestion scenario 1: Throughput and delay as a function of host sending rate

Scenario 2 - retransmission

Figure 3.45 Scenario 2: Two hosts (with retransmissions) and a router with finite buffers

- Sender retransmits lost and timed-out packet
- Sender application-layer output = Receiver application-layer input: $\lambda_{in} = \lambda_{out}$
- Transport-layer sending rate, including retransmissions: λ'_{in}
- λ'_{in} : offered load to network
- λ'_{out}: Network throughput
- λ_{out}: APP throughput



Needed and Un-needed retransmissions

1. Packets can be lost, dropped at router due to full buffers, requiring retransmissions (needed retransmission)
2. Sender can timeout prematurely, sending a copy of packet, both of which are delivered (un-needed retransmission)

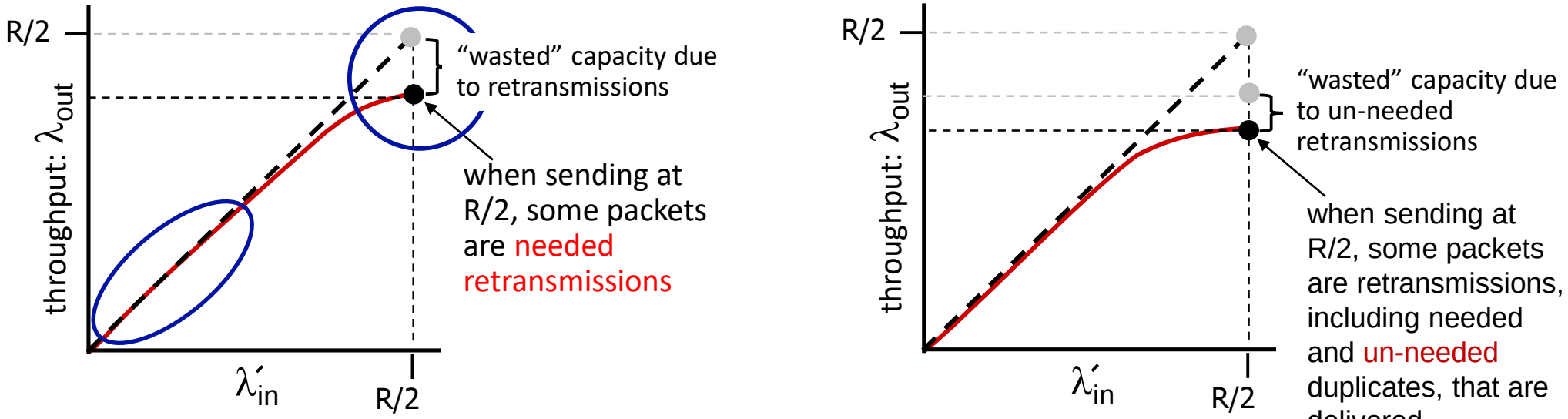
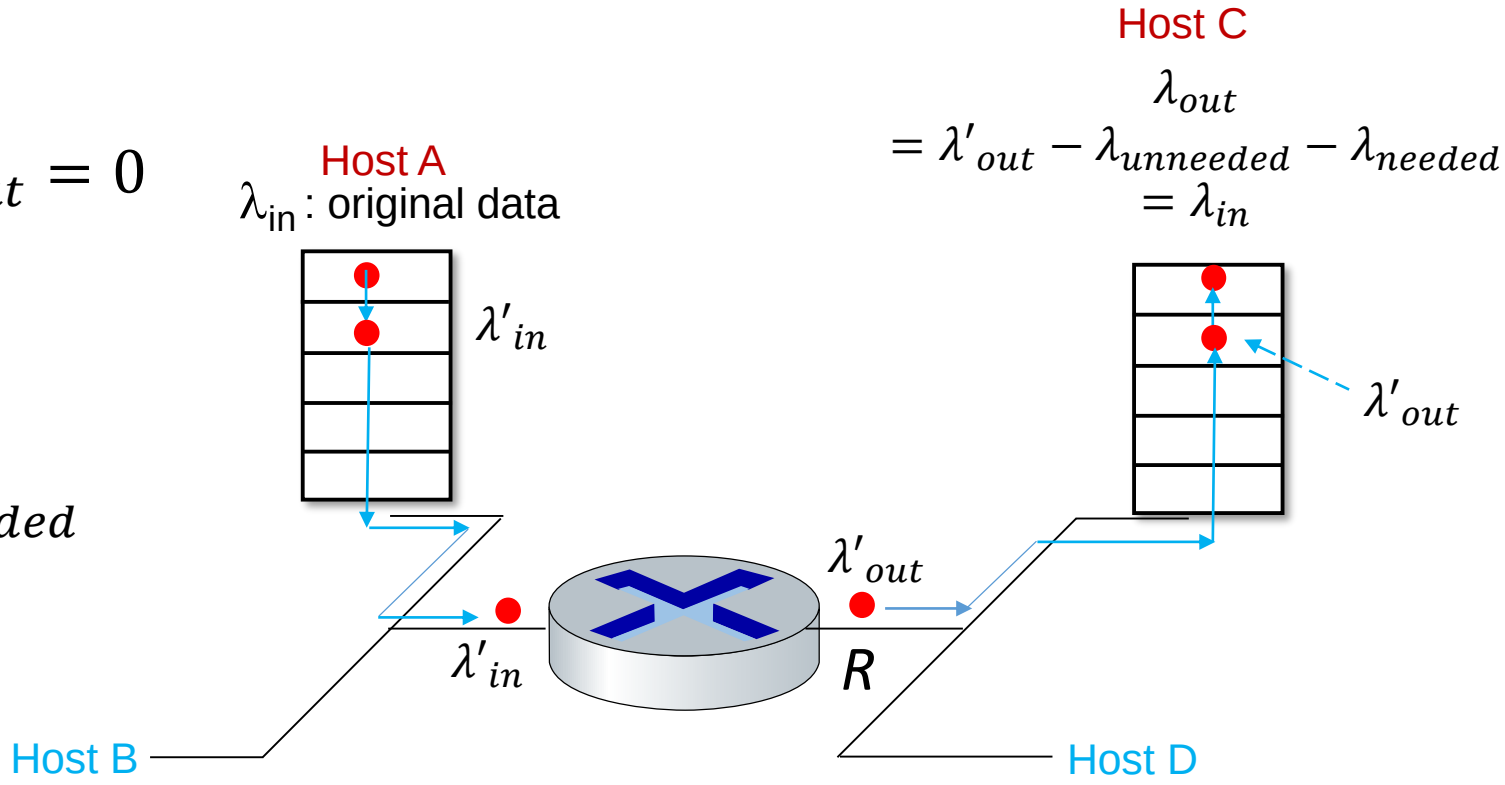


Figure 3.46 Scenario 2 performance with finite buffers

Max APP throughput

- If: $\lambda_{dropped} = \lambda'_{in}$, Then: $\lambda'_{out} = 0$

- If: $\lambda_{dropped} < \lambda'_{in}$
- Then:
- $\lambda'_{in} = \lambda_{in} + \lambda_{unneded} + \lambda_{needed}$
- $\lambda'_{out} = \lambda'_{in}$
- $\lambda_{out} = \lambda_{in}$



- Max APP throughput: $\lambda_{out}^{max} = \frac{R}{2} - \lambda_{unneded} - \lambda_{neded}$

Costs of congestion

- Cost1: large queuing delays are experienced as packet-arrival rate nears link capacity (mild congestion)
- Cost2: large queuing delay causes unneeded retransmissions, link carries multiple copies of a packet
  - decreasing maximum achievable throughput
- Cost3: sender must perform needed retransmissions in order to compensate for dropped (lost) packets due to router's buffer overflow (heavier congestion)
  - more work (retransmission) for given receiver throughput

Scenario 3: Multiple flow in a router

- **Q:** What happens as red λ_{in} and λ'_{in} increase and fill up Router1 buffer?
- **A:** All arriving blue pkts at Router1 are dropped ($\lambda'_{in} \rightarrow \lambda_{dropped}$) and (blue throughput) $\lambda_{out} \rightarrow 0$

Another "cost" of congestion:

- When packet dropped, any upstream transmission capacity and buffering used for that packet is wasted

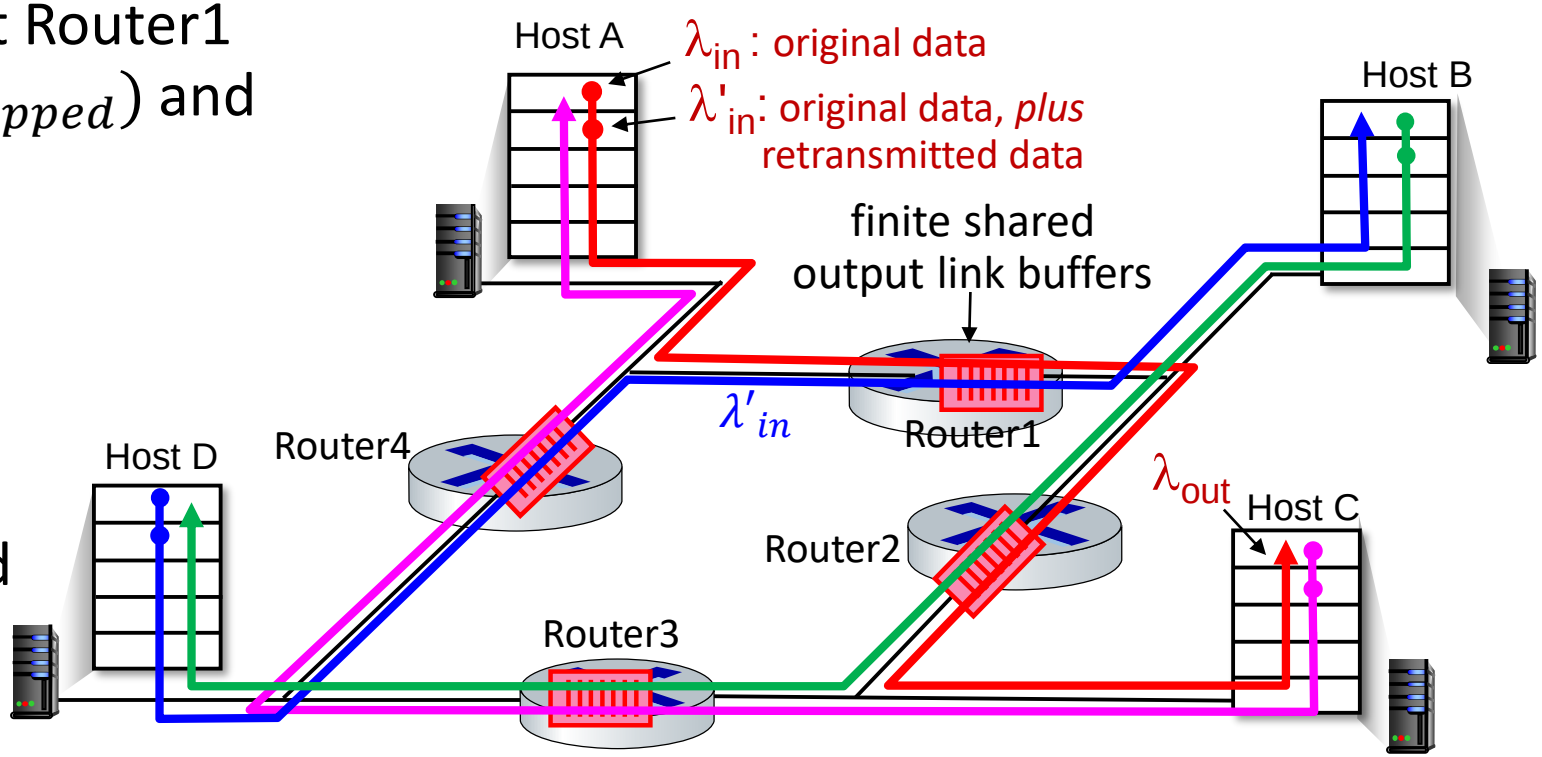
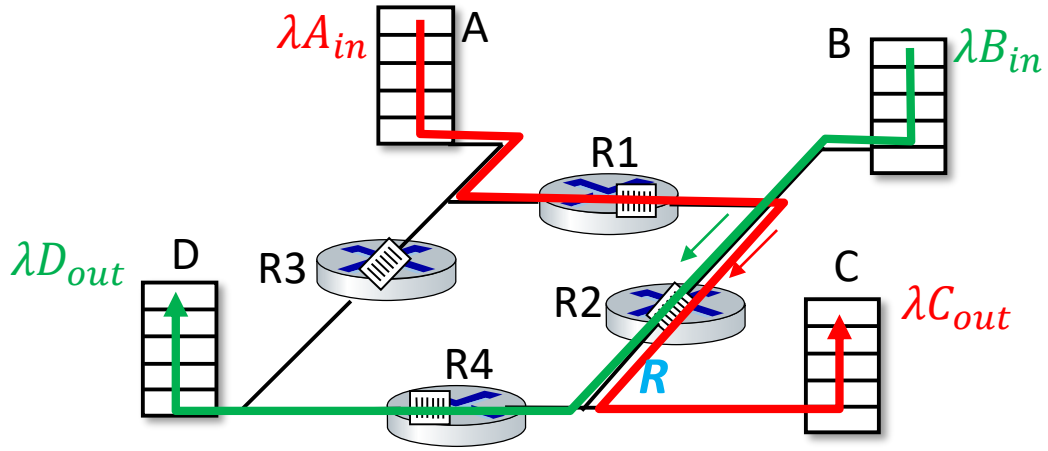


Figure 3.47 Four senders, routers with finite buffers, and multi-hop paths

Competing flows (traffics)

- Suppose ***R*** is bottleneck in following network. Max "A-to-C traffic + B-to-D traffics" arriving to Router2 can be ***R***
- A-to-C and B-to-D traffics must compete at Router2 for limited amount of buffer space
- As λB_{in} from B-to-D gets larger and larger, A-to-C traffic that successfully gets through Router2 (not lost due to buffer overflow) becomes smaller and smaller



Congestion collapse

- In limit, as λB_{in} from B-to-D approaches to ***R***, an empty buffer at R2 is immediately filled by a B-to-D packet, and throughput of A-to-C connection at R2 goes to zero
- This implies that A-to-C throughput goes to zero in limit of heavy traffic from B-to-D
- These considerations give rise to offered load versus throughput tradeoff shown in Figure 3.48.

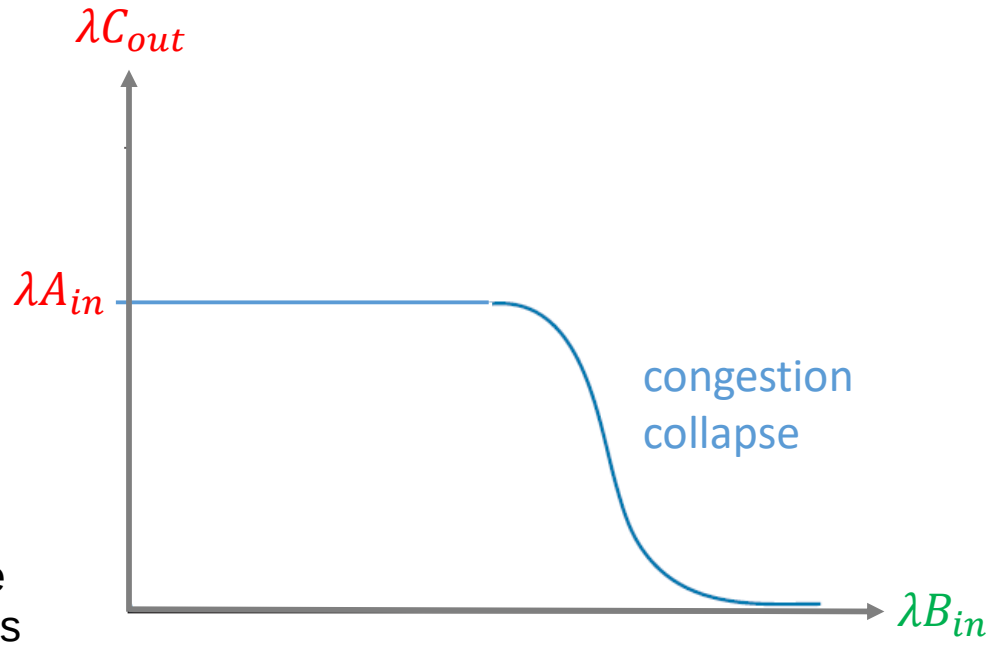


Figure 3.48 Scenario 3 performance with finite buffers and multi-hop paths

3.6.2 Approaches to Congestion Control

1- End-to-end congestion control

- Presence of congestion inferred by hosts based only on observed network behavior (packet loss, latency)
- No assistance from IP network (Routers)

- Internet-default:
  - IP layers in Routers are not required to provide congestion feedback to sending host. TCP adopts an end-to-end approach towards congestion control
  - TCP segment loss (RTO timeout or three duplicate ACK) is taken as an indication of network congestion, and TCP decreases its sending rate
- We'll see (3.7.2) IP may assist TCP to implement congestion control (network-assisted). ECN bits
- We'll also see a recent proposal for TCP congestion control that uses increasing round-trip segment delay as an indicator of increased network congestion

3.6.2 Approaches to Congestion Control

2- Network-assisted congestion control

- IP layers (routers) provide direct feedback to sending/receiving hosts with flows passing through congested router
- May indicate congestion level or explicitly set sending rate
  - TCP ECN, ATM, DECbit protocols
- Congestion fed back: either directly from congested routers or by receiver (takes a RTT)

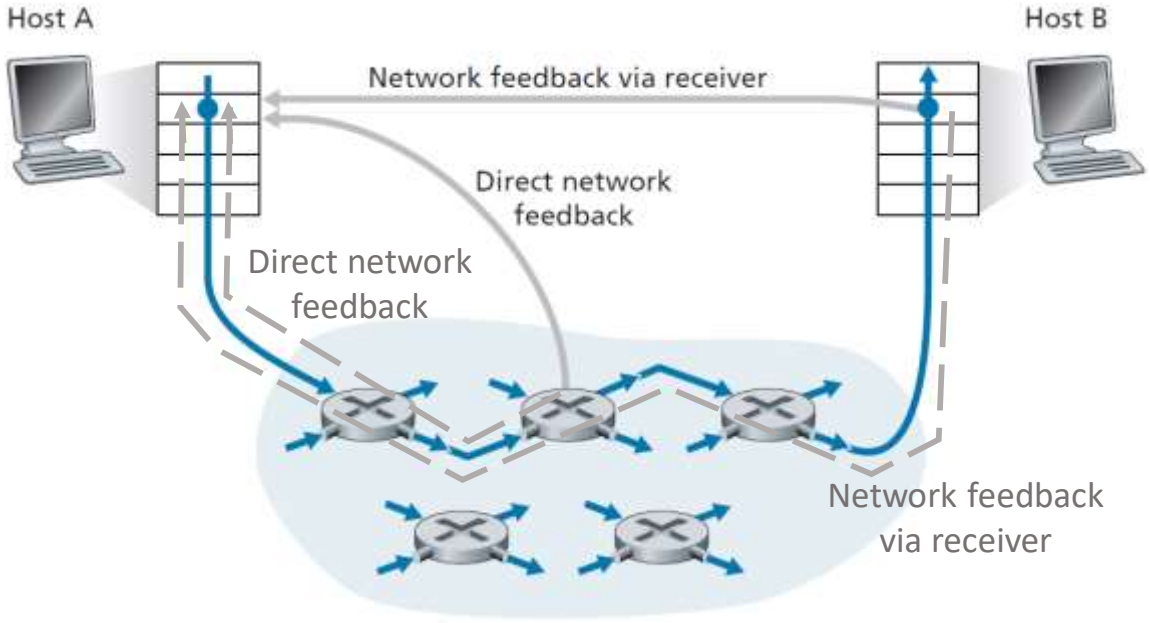


Figure 3.49 Two feedback pathways for network-indicated congestion information

Contents

3.7.1 Classic TCP Congestion Control

- Flow control: TCP sender limits sending rat to receiver empty buffer `rwnd` (Section 3.5)
- Congestion-control (operating at sender): TCP uses `cwnd` (congestion window) variable
- `cwnd` imposes a constraint on rate at which a TCP sender can send traffic into network
- Number of unacknowledged data at a sender ≤ min {`cwnd`, `rwnd`}

`LastByteSent - LastByteAcked ≤ min {cwnd, rwnd}`

Timeout/3DupACKs: "Loss event"

- In order to focus on congestion control assume:
  1. TCP receive buffer is large so that `rwnd≥cwnd`
  2. Sender always has data to send
- So, `cwnd` limits amount of unacknowledged data at sender (indirectly limits sender's send rate)
- **Suppose:** Occurrence of either a timeout or receipt of three duplicate ACKs means "Loss event" at a TCP sender!
- When there is congestion, one (or more) router buffers along path overflows, a datagram is dropped. It results in a "Loss event" at sender, indicates congestion on sender-to-receiver path

Normal arrival of ACK – "ACK event"

- TCP takes normal arrival of ACKs as an indication all is well
- TCP increases `cwnd` on arrival of first acknowledged of a packet ("ACK event")
- In a high latency connection, ACKs arrive at a slow rate, then congestion window will be increased at a slow rate
- In a low latency connection, ACKs arrive at a high rate, then congestion window will be increased more quickly

- Because TCP uses ACKs to trigger (or clock) its increase in congestion window size, TCP is said to be self-clocking

TCP's strategy

- TCP sender controls sending rates such that:
  - don't congest network and at same time make use of all available bandwidth

Q: Are TCP senders explicitly coordinated?

A: No, there is a distributed approach in which TCP senders can set their sending rates based only on local information

- TCP following principles:
  - In a "Loss event", TCP sender decreases `cwnd`
  - In an "ACK event" , TCP sender increases `cwnd`
  - TCP's strategy: increase rate in response to arriving ACKs until a loss event occurs, then decrease transmission rate

TCP's strategy: Bandwidth probing

- Bandwidth Probing:
  - TCP sender increases its transmission rate to probe for rate at which congestion begins
  - TCP backs off from that rate, and then to begins probing again to see if congestion onset rate has changed
- **No explicit** signaling of congestion state by IP network
- "ACK event" and "Loss event" serve as **implicit** signals
- TCP sender acts on local information asynchronously from other TCP senders on same host and other hosts

TCP congestion-control algorithm [RFC 5681]

- Algorithm has **three major components**:
  1. Slow Start (SS)
  2. Congestion Avoidance (CA)
  3. Fast Recovery (FR)
- SS and CA are mandatory components, FR is recommended, but not required

Figure 3.51

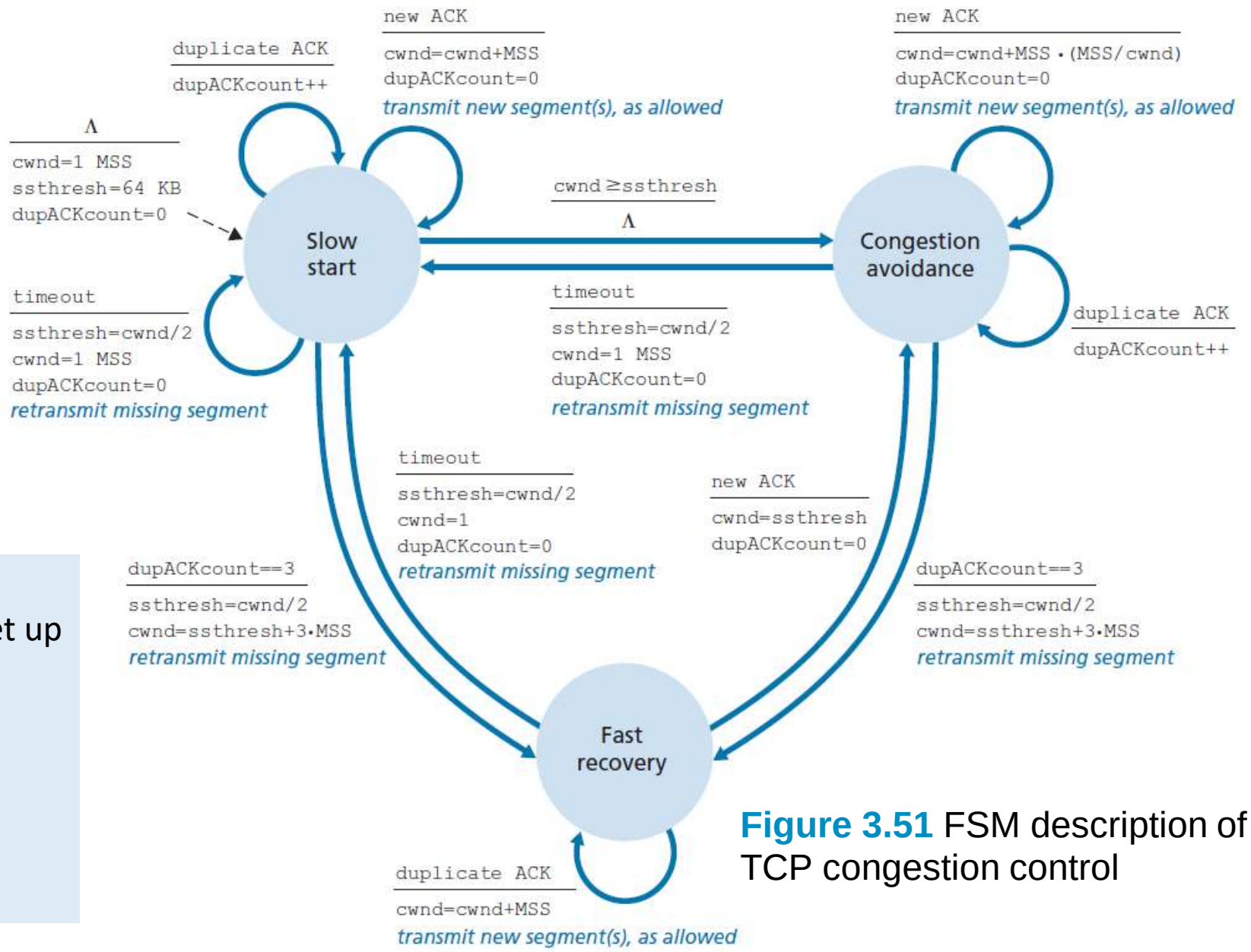


Figure 3.51 FSM description of TCP congestion control

Parameter:

- MSS: set at connection set up

State variables:

- cwnd
- ssthresh
- Timer
- dupACKcount

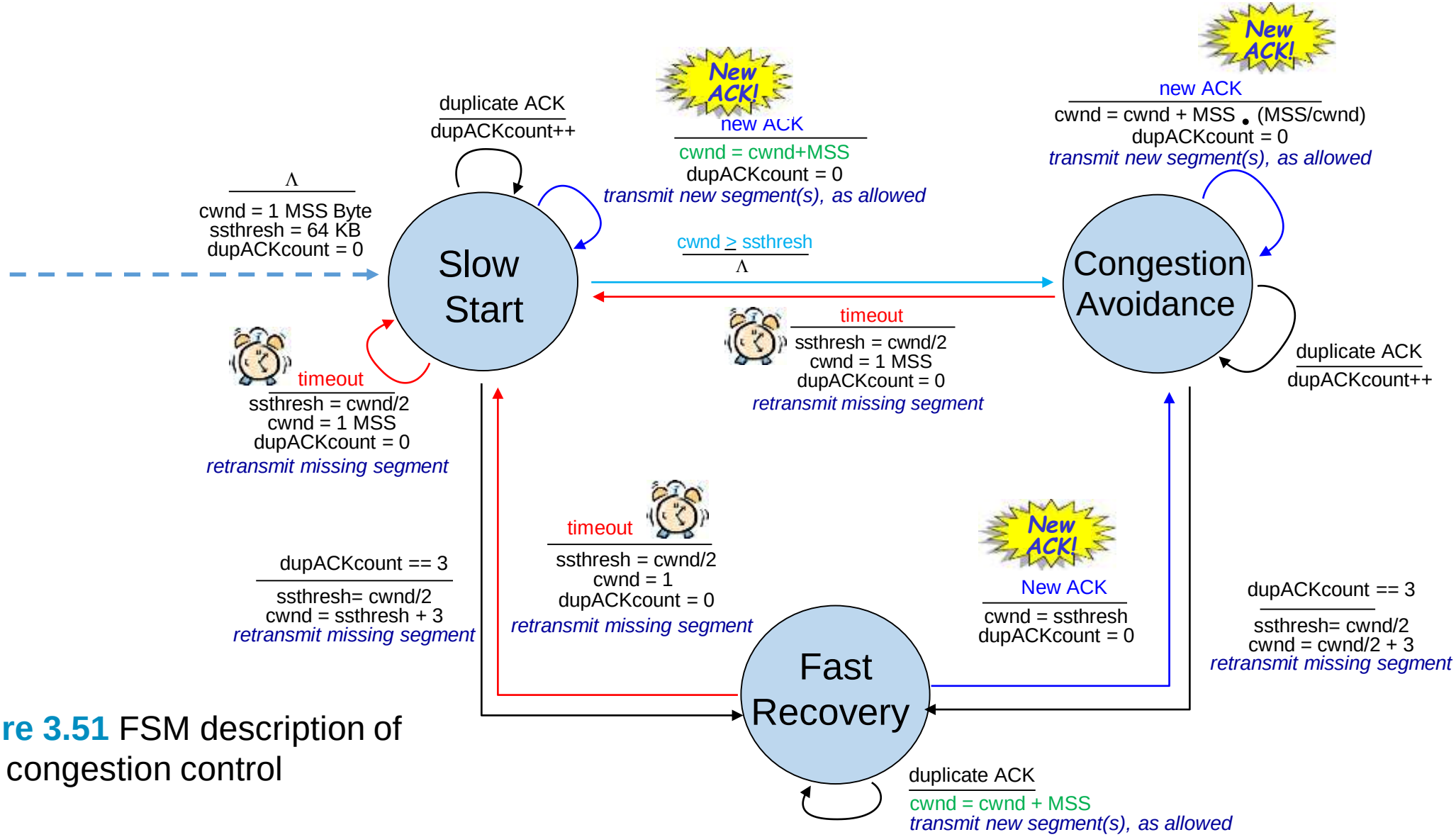


Figure 3.51 FSM description of TCP congestion control

1- Slow Start

- When a TCP connection begins, cwnd=1MSS [RFC 3390]
- TCP sender would like to **find amount of available bandwidth quickly**, thus, cwnd increases by 1MSS every time a transmitted segment is first acknowledged ("ACK event")
- Figure 3.50, sender increases cwnd by 1MSS for each of ACK, result is doubling cwnd (sending rate) every RTT
- Send rate **starts slow** (cwnd=1) and grows exponentially

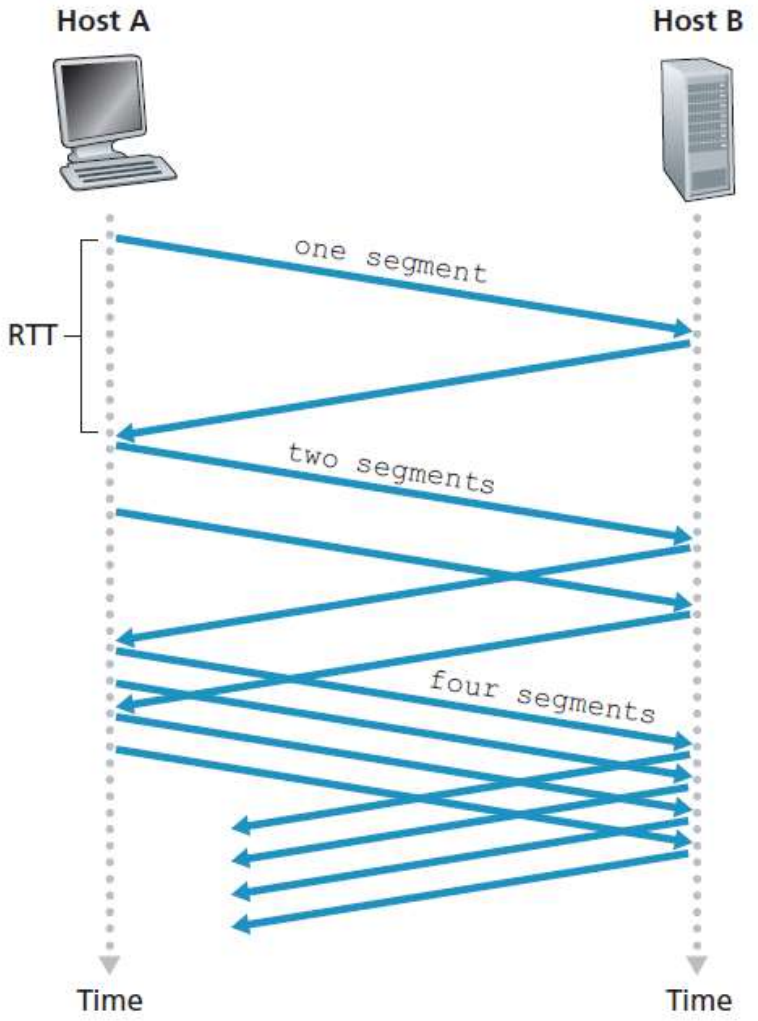


Figure 3.50 TCP slow start

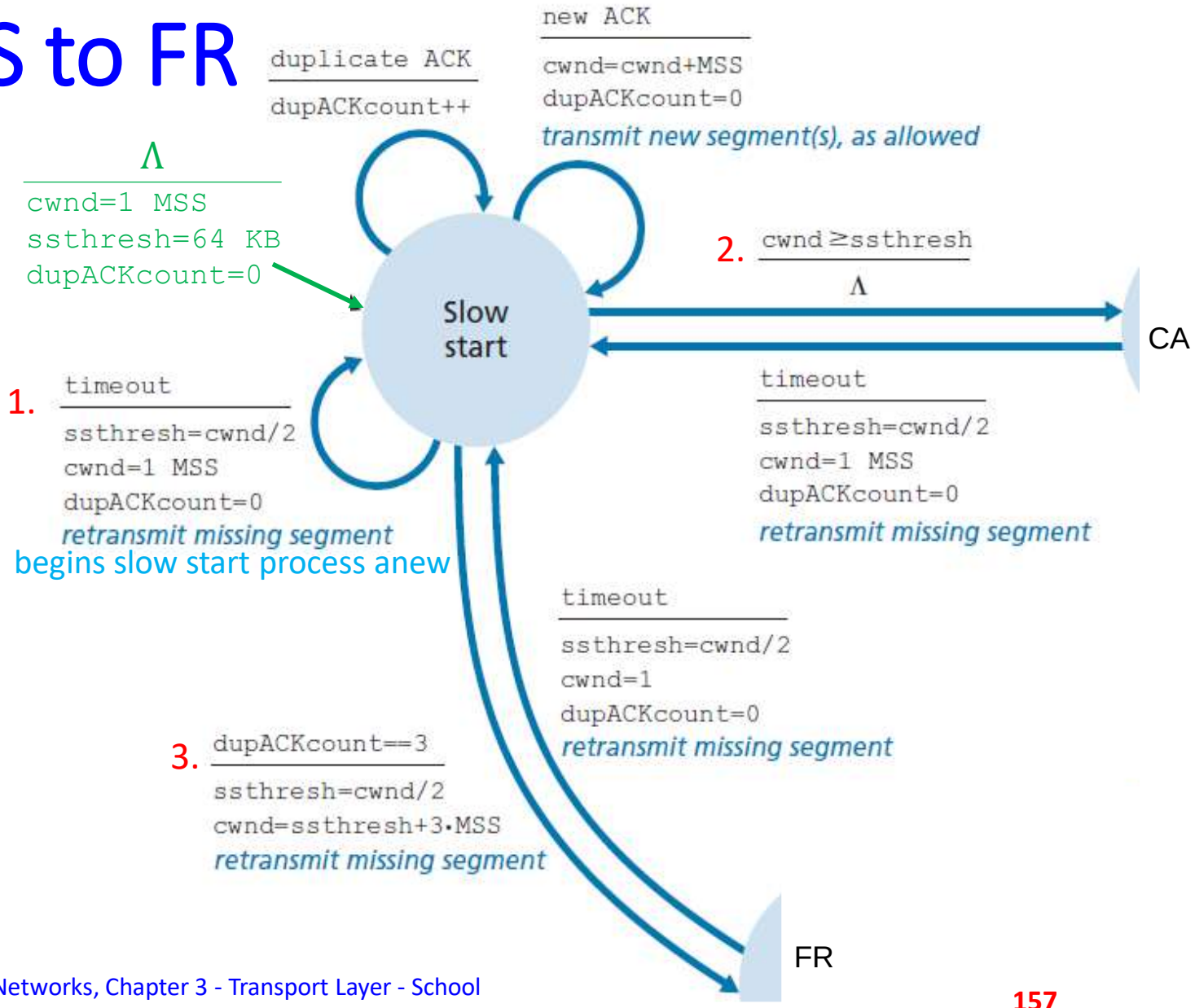
Transition from SS to SS, CA or FR

1. If there is a "Loss event", indicated by a RTO timeout, TCP sender sets `cwnd`=1 and begins SS process anew. It also sets value of a second state variable, ssthresh (slow start threshold) to `cwnd`/2 (half of value of `cwnd` when congestion was detected)
2. If there is a "ACK event" and `cwnd` reaches to ssthresh, slow start ends and TCP transitions into CA mode
3. If there is a "Loss event", indicated by 3 duplicate ACKs, TCP transitions into FR (Section 3.5.4)

See Figure 3.51 (FSM)

dupACKcount=3: SS to FR

- When triple duplicate ACKs were received (dupACKcount=3), then
- `ssthresh=cwnd/2`
- `cwnd=cwnd/2+3•MSS`
- (adding in 3 MSS for good measure to account for triple duplicate ACKs received)
- FR state is then entered

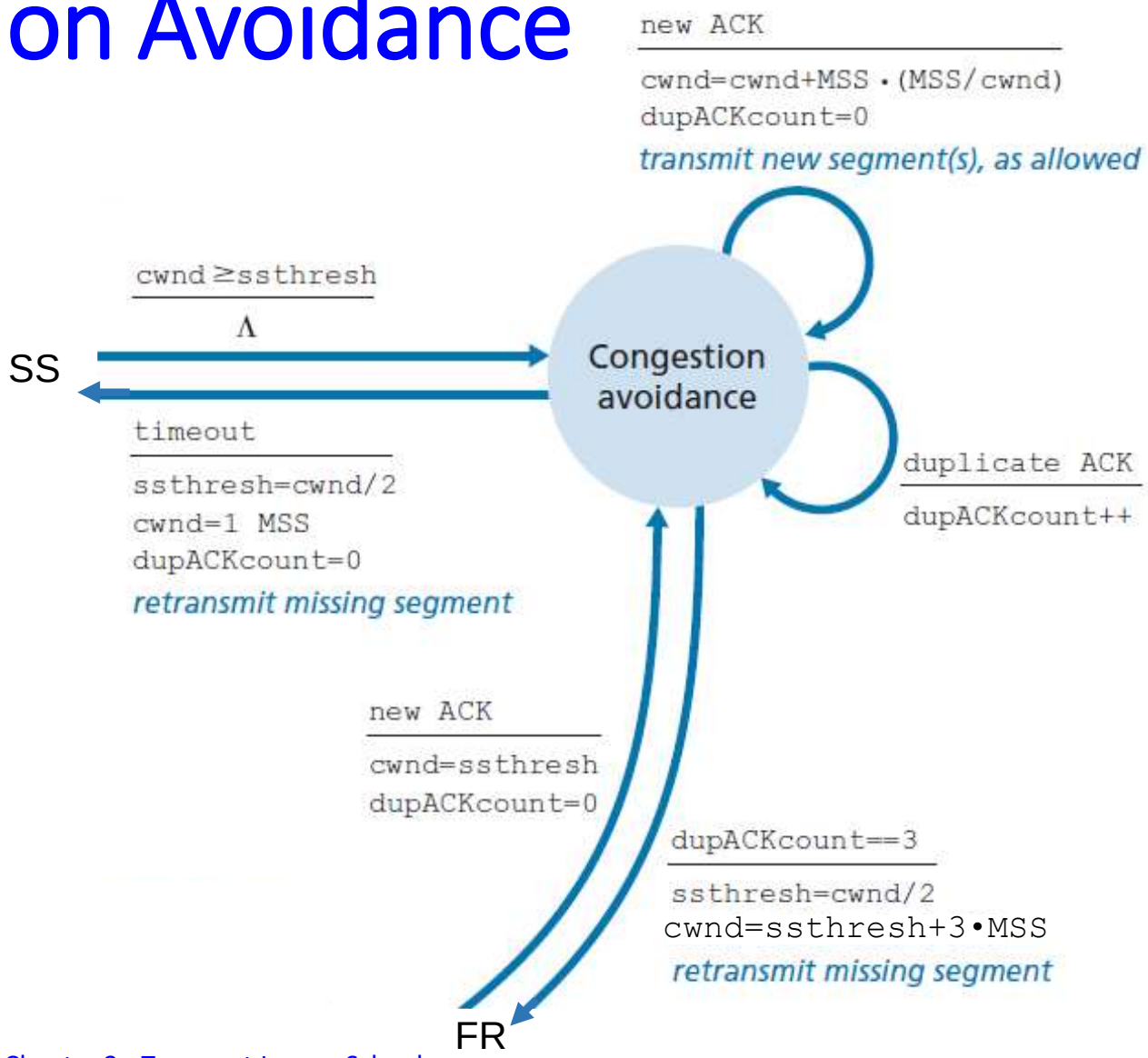


2- Congestion Avoidance

- On entry to CA state: `cwnd` is half its value when congestion was last encountered
- Then, `cwnd` increased by just a single MSS every RTT [RFC 5681]
- **Q**: When should CA's linear increase (of 1 MSS per RTT) end?
- **A**: timeout or triple duplicated ACKs ("Los event")
- Example:, MSS=1,460Bytes and cwnd=14600Bytes (10MSS)
  - 10 segments are being sent within an RTT
  - Each arriving ACK (assuming one ACK per segment) increases cwnd by 1/10 MSS, and thus, cwnd increases by one MSS after ACKs of all 10 segments have been received

Figure 3.51 - Congestion Avoidance

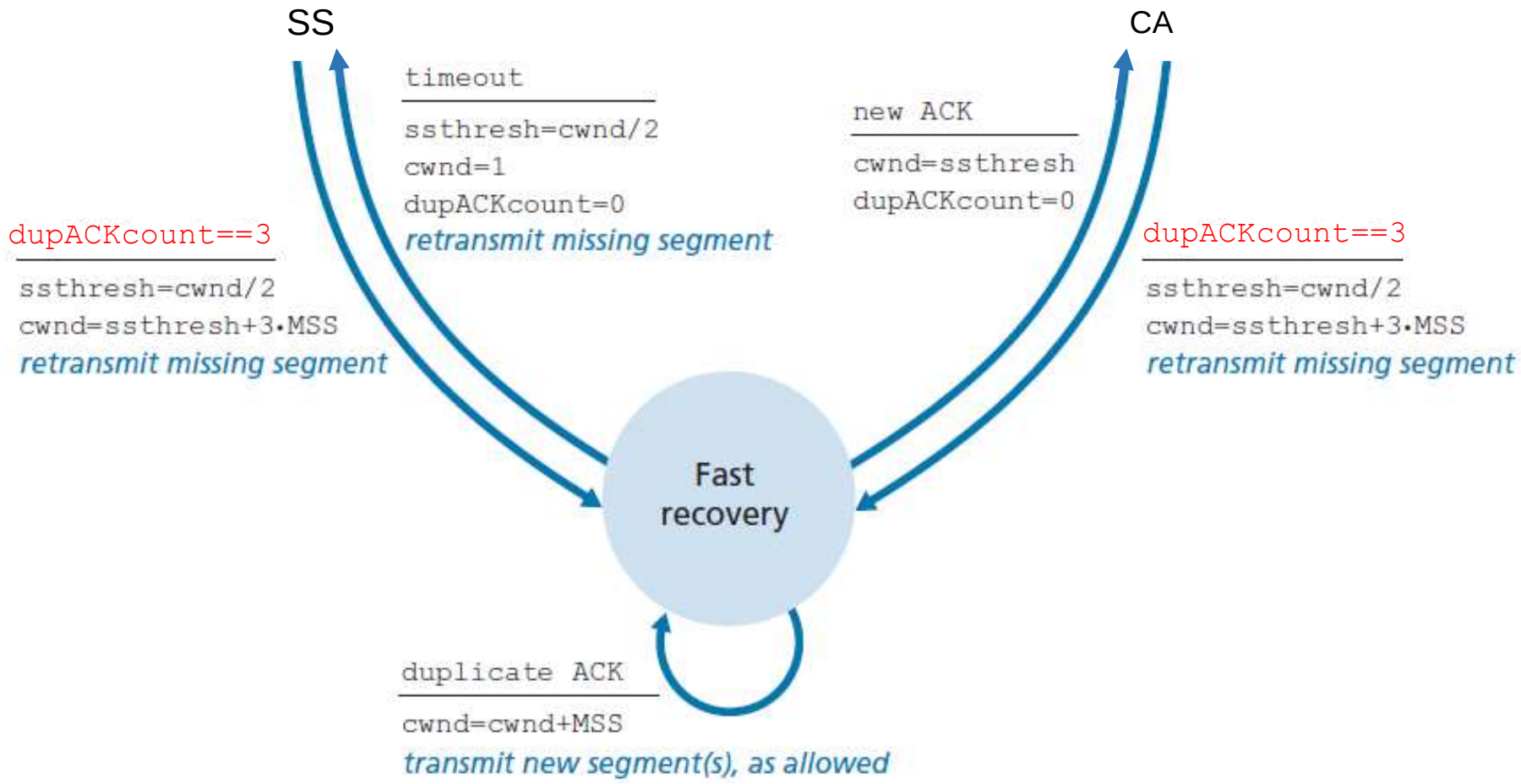
- When, in SS, rwnd≥ssthresh, state transits from SS to CA
- When, in FR, a new ACK received, state transits from FR to CA
- In CA: cwnd increased by just a single MSS every RTT [RFC 5681]



3- Fast Recovery

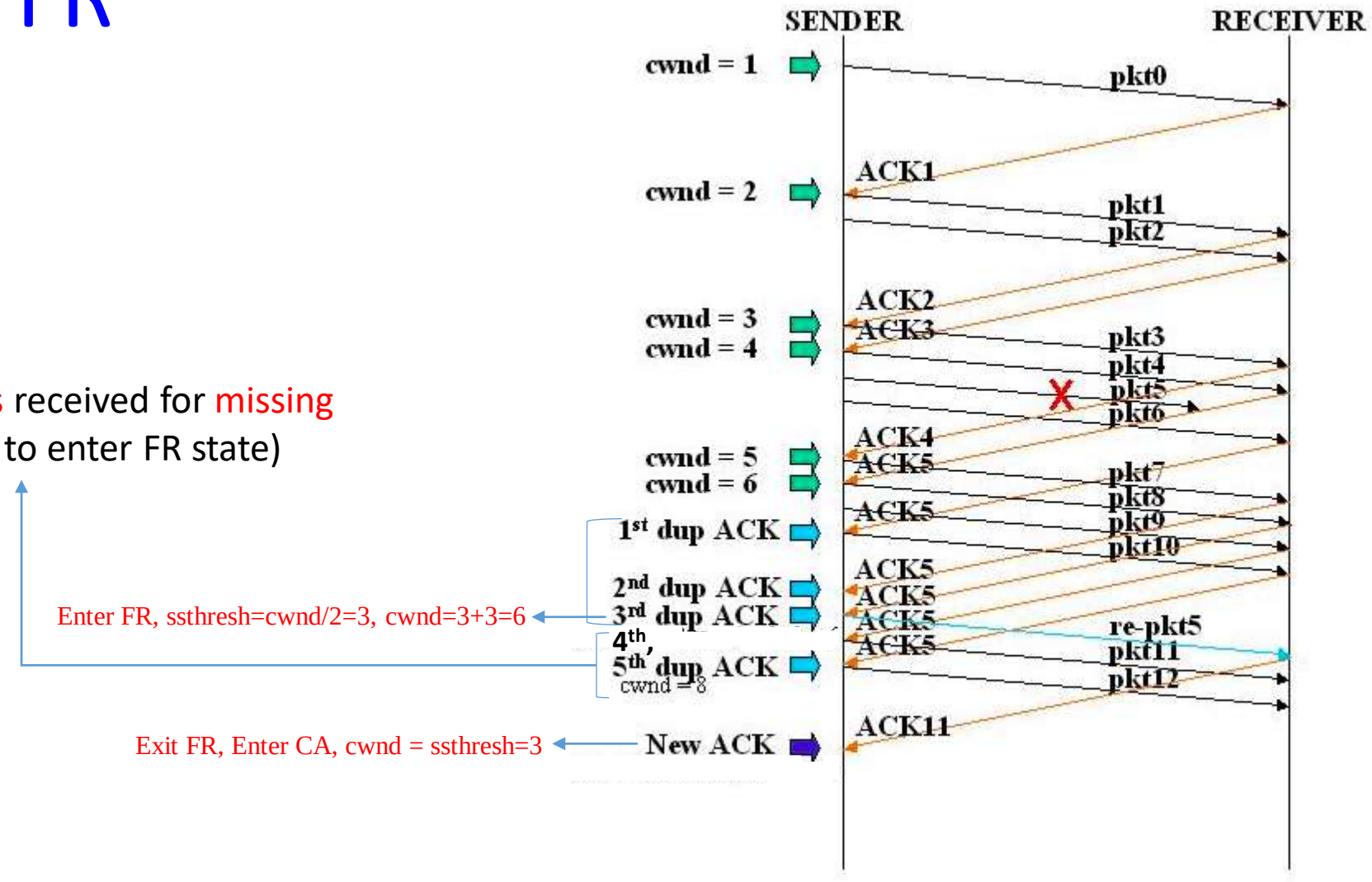
- `cwnd` is increased by 1 MSS for every duplicate ACK received for missing segment that caused TCP to enter FR state
- Eventually, when an ACK arrives for missing segment, TCP enters congestion-avoidance state after deflating `cwnd`
- If a timeout event occurs, fast recovery transitions to slow-start state after performing same actions as in slow start and congestion avoidance: value of `cwnd` is set to 1 MSS, and value of `ssthresh` is set to half value of `cwnd` when loss event occurred
- Fast recovery is a recommended, but not required, component of TCP [RFC 5681]
- Early version of TCP, **TCP Tahoe**, unconditionally cut its congestion window to 1 MSS and entered slow-start phase after either a timeout-indicated or triple-duplicate-ACK-indicated loss event. **TCP Reno**, incorporated fast recovery

Figure 3.51 - FR



Example: FR

In FR, 2 duplicate ACKs received for missing pkt5 (pkt5 caused TCP to enter FR state)



cwnd-time graph

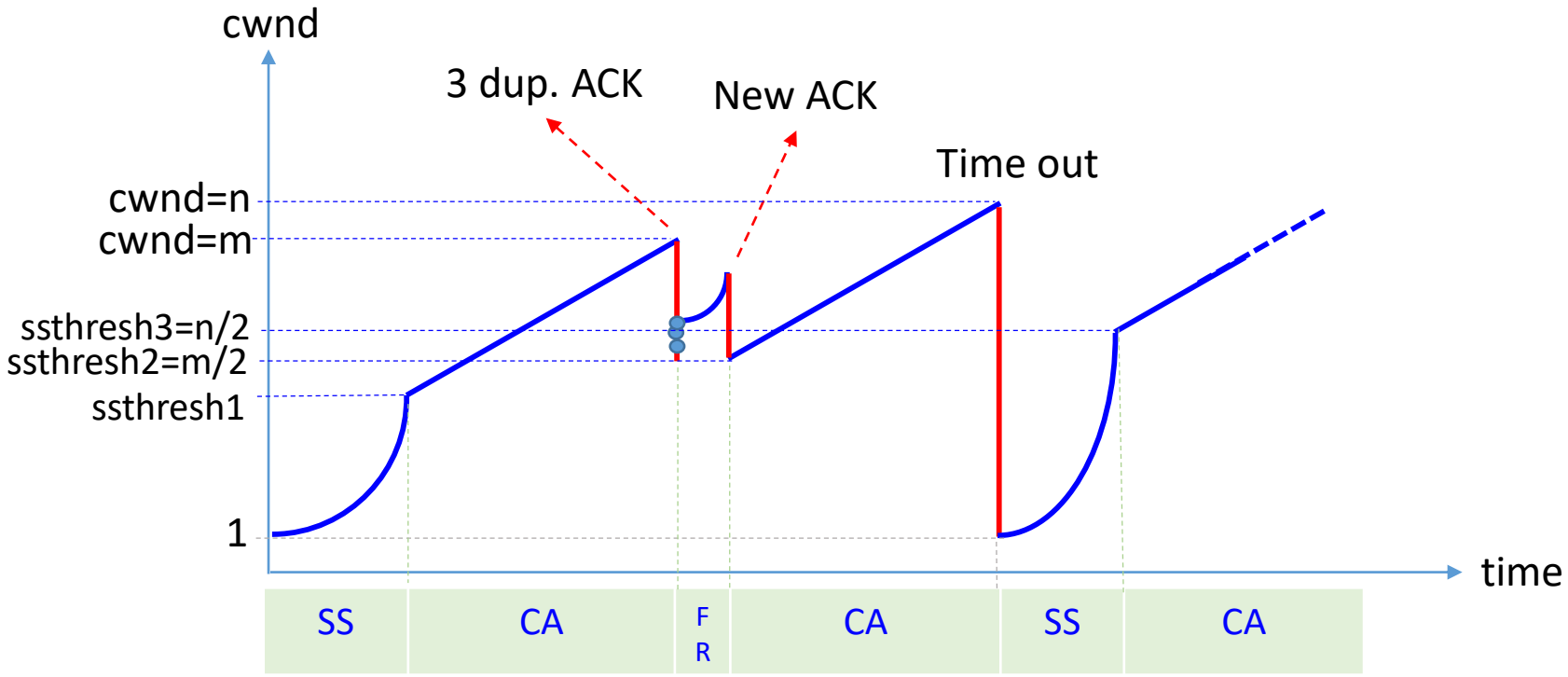
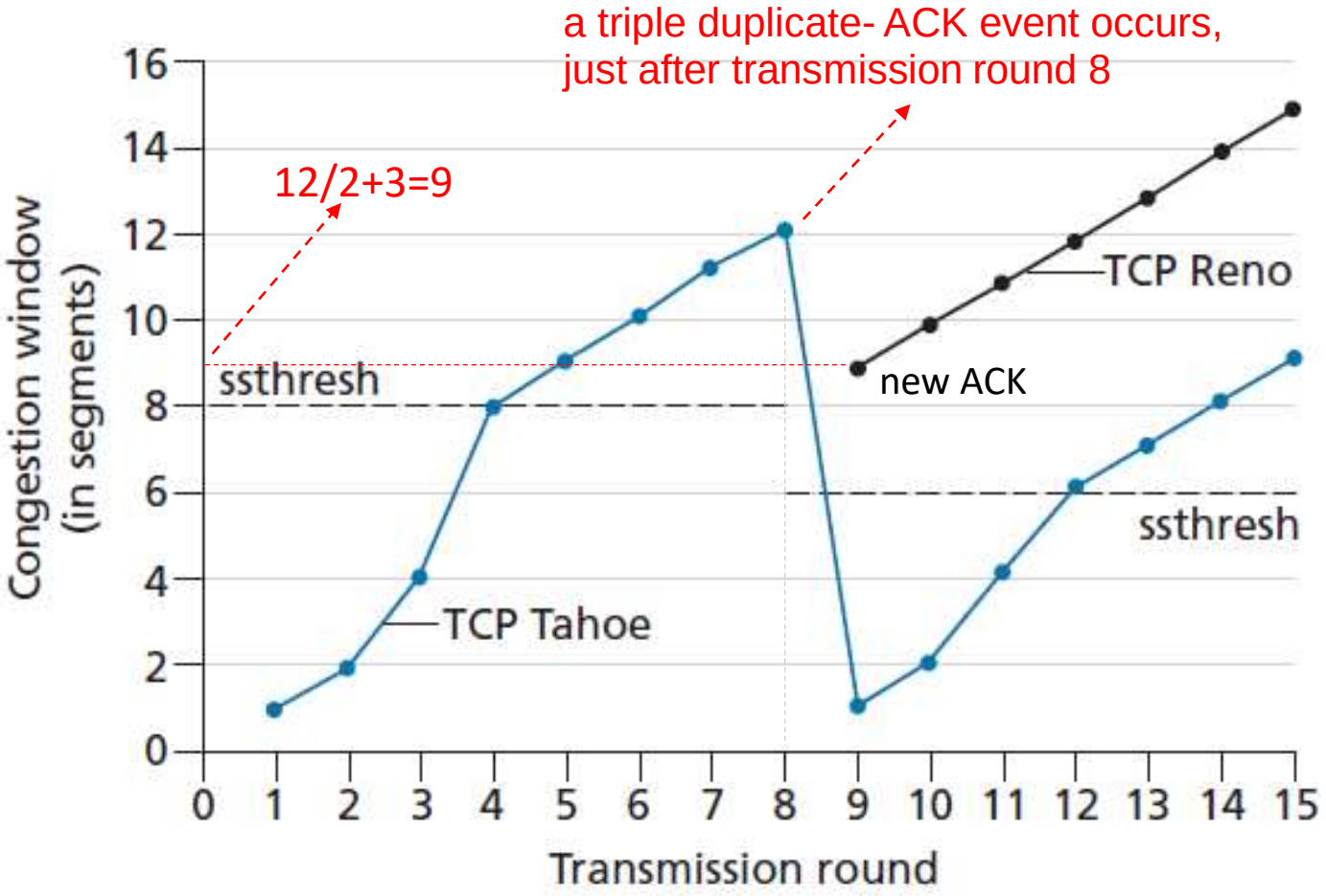


Figure 3.52 – TCP Reno

- Evolution of TCP's congestion window (Tahoe and Reno)
- FR is not implemented
- Reno: After 3 duplicate ACK, `cwnd=cnwd/2`, and CA state maintained
- **TCP Reno** implements SS, CA and FR, but not Selective ACK
- **TCP New Reno**: Slight modification on **Reno** in FR state



CA: Additive-Increase and Multiplicative-Decrease (AIMD)

- TCP's congestion control (CA) consists of
  - Linear (additive) increase in `cwnd` of 1 MSS per RTT (additive-increase)
  - Halving (multiplicative decrease) of `cwnd` on a triple duplicate-ACK event (multiplicative-decrease)
- CA is an AIMD congestion control: Linearly increasing `cwnd` until a triple duplicate-ACK event occurs, then decreasing `cwnd` by a factor of two but then again begins increasing it linearly
  - Probing to see if there is additional available bandwidth

AIMD

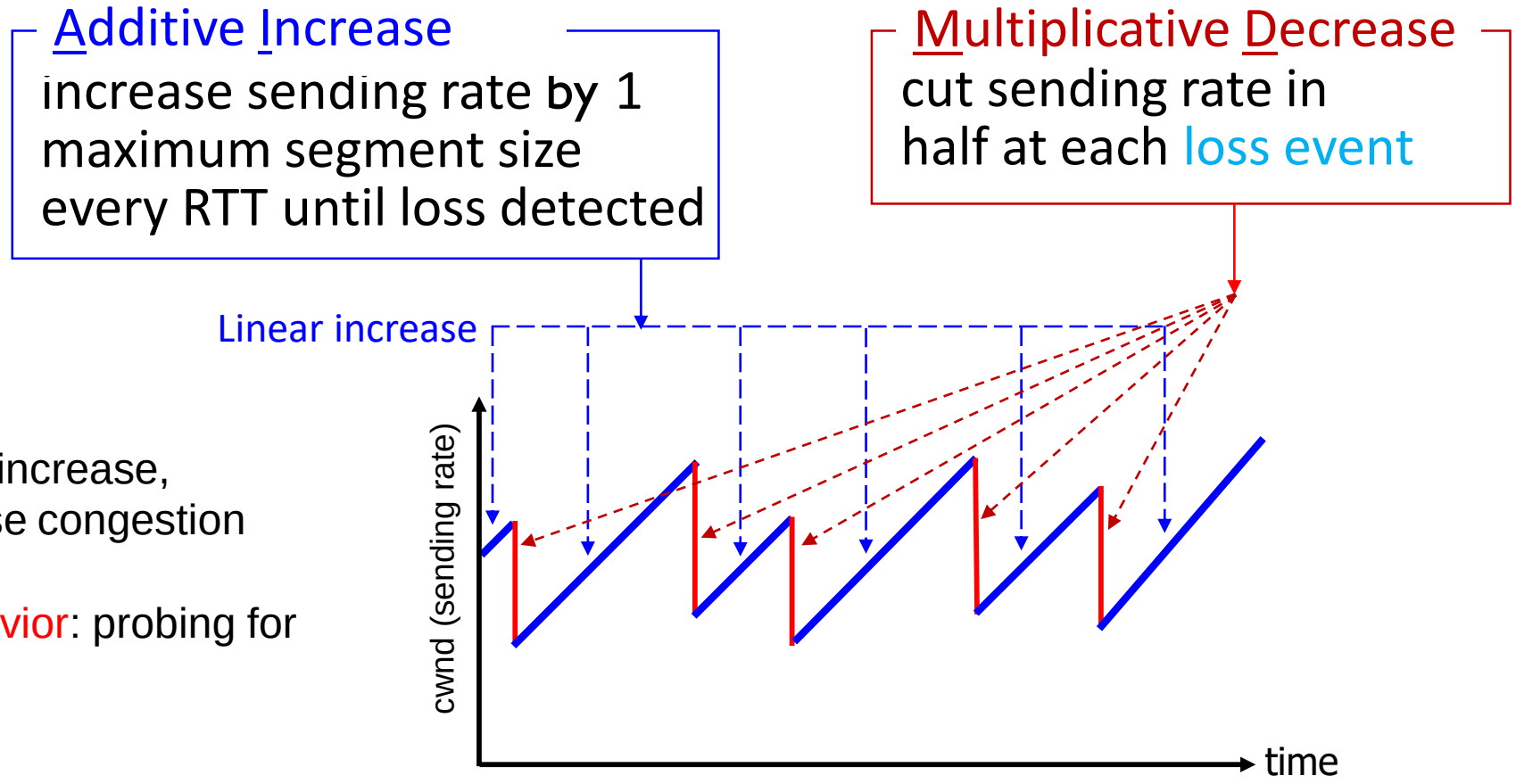


Figure 3.53 Additive-increase, multiplicative-decrease congestion control.
AIMD saw tooth behavior: probing for bandwidth

Macroscopic Description of TCP Throughput

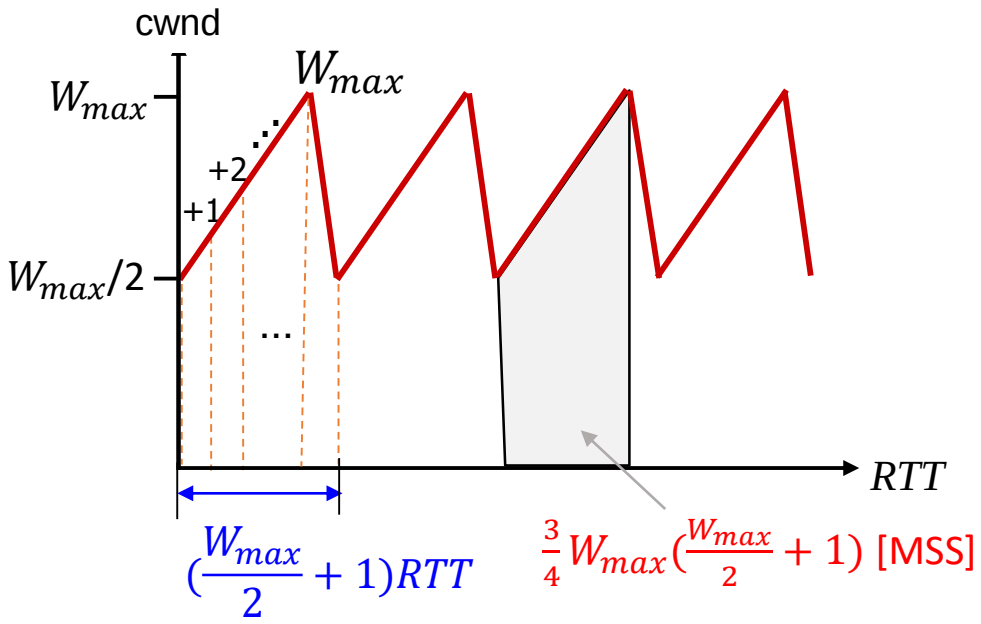
- Given saw-toothed behavior of TCP (ignoring slow-start phases that occur after timeout events)

Data send during one period: $\frac{W_{max}}{2} + (\frac{W_{max}}{2} + 1) + (\frac{W_{max}}{2} + 2) + \cdots + W_{max} = \frac{3}{4} W_{max}(\frac{W_{max}}{2} + 1)$

$$Average\ throughput = \frac{\frac{3}{4} W_{max}(\frac{W_{max}}{2} + 1)}{(\frac{W_{max}}{2} + 1)RTT} = \frac{3}{4}\frac{W_{max}}{RTT}$$

$$average\ packet\ loss = L = \frac{1}{\frac{3}{4} W_{max}(\frac{W_{max}}{2} + 1)} \approx \frac{8/3}{{W_{max}}^2}$$

$$average\ throughput = \frac{3}{4}\frac{W_{max}}{RTT} = \frac{\sqrt{3/2}}{RTT\sqrt{L}} = \frac{1.22}{RTT\sqrt{L}}$$



TCP's congestion control algorithm: Optimum

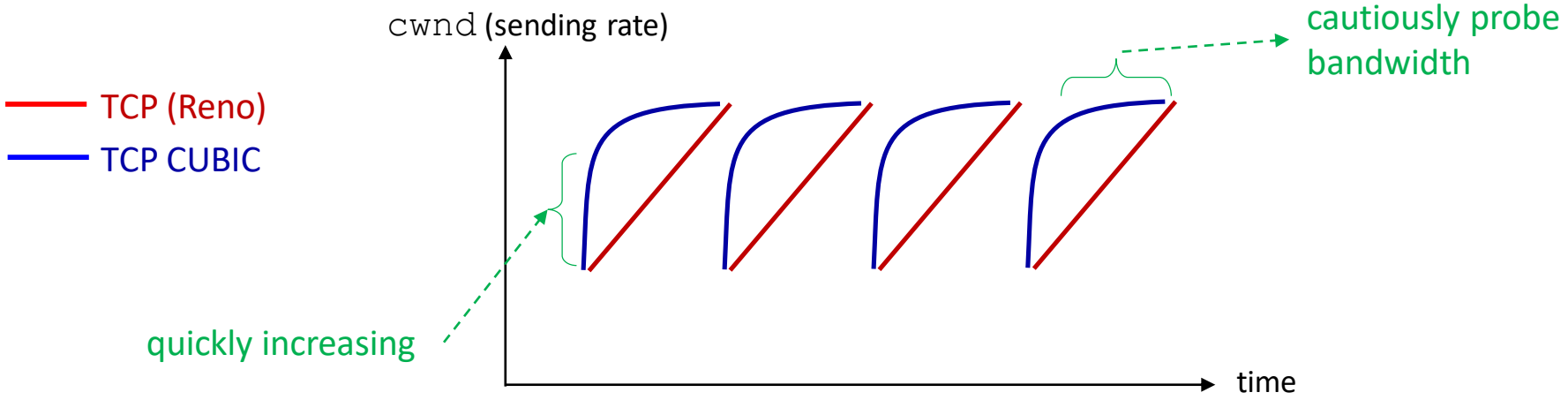
- TCP's AIMD algorithm was developed based on a tremendous amount of engineering insight and experimentation with congestion control in operational networks
- Theoretical analyses showed:
  - TCP's congestion-control algorithm serves as a **distributed asynchronous-optimization algorithm** that results in several important aspects of user and network performance being simultaneously optimized [Kelly 1998]
  - A rich theory of congestion control has since been developed [Srikant 2012]

TCP Cubic – nonlinear increase

- TCP CUBIC [Ha 2008, RFC 8312], **optimized for high bandwidth, high delay connections**

If: State of congested link where packet loss occurred hasn't changed much

Then: More quickly increasing sending rate to get close to pre-loss sending rate and then probe cautiously for bandwidth



TCP Cubic - parameters

- `cwnd` is increased on ACK , and SS and FR phases remain the same
- CUBIC only changes **congestion avoidance phase**, as follows:
- W_{max} : Sending rate at which congestion loss was detected
- K : time duration in which CUBIC's window size will again will reach to W_{max} (how quickly `cwnd` would reach W_{max})
- Tunable CUBIC parameters β and C determine value of K

TCP Cubic - algorithm

- CUBIC increases `cwnd` as a function of **cube** of distance between current time t and K (that is $(t-K)^3$)
- For $t \to K$, $(t-K)^3$ increases are small when t is close to K (good if congestion level of link causing loss hasn't changed much), then increases rapidly as t exceeds K (more quickly find a new operating point if congestion level of link that caused loss has changed significantly)

cubic function before reaching to 0 is concave and after that turns into a convex profile

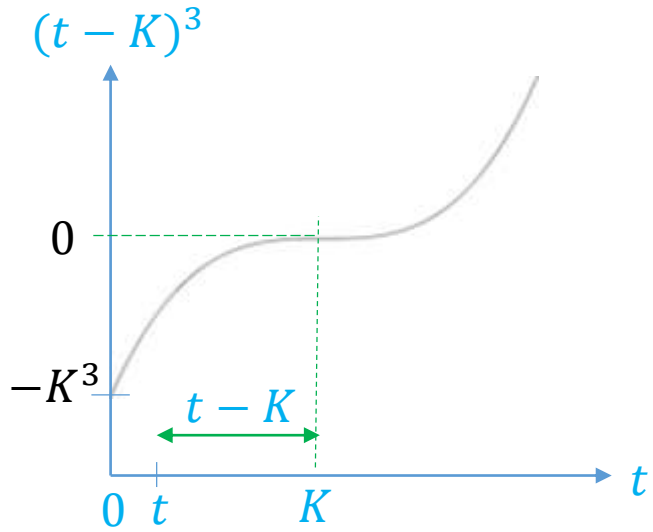
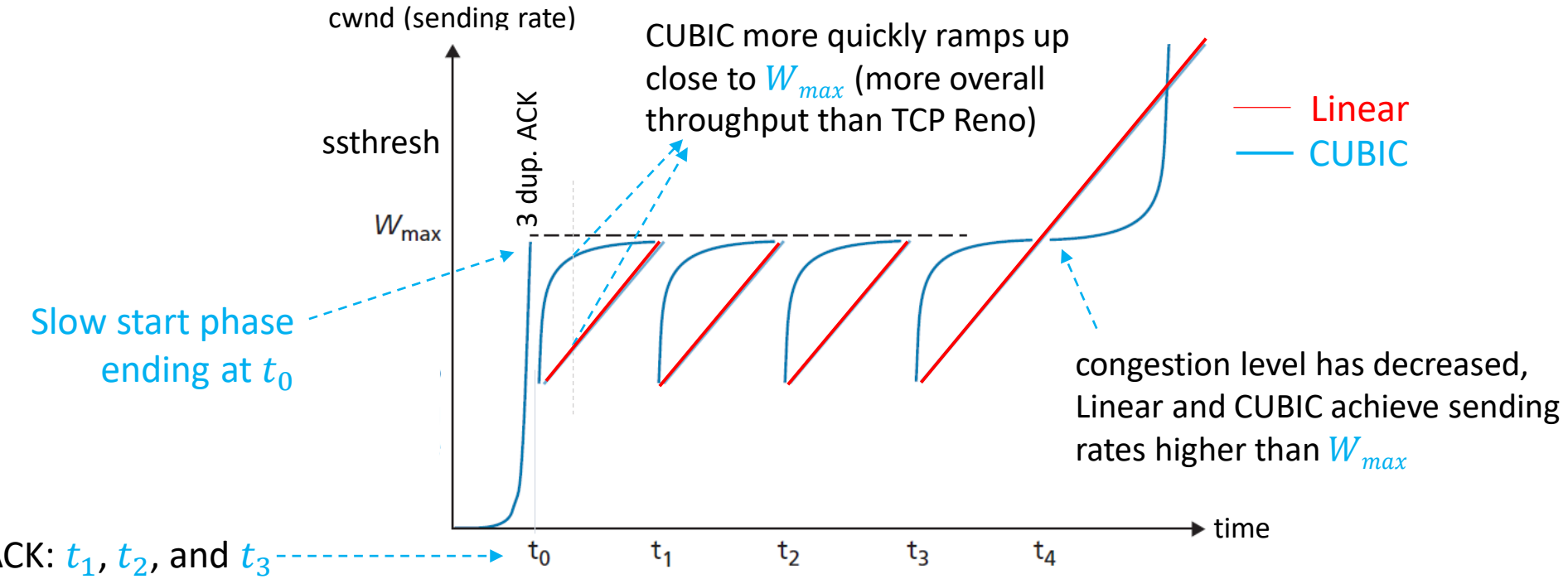


Figure 3.54 TCP CA sending rates: Linear vs CUBIC

- TCP CUBIC default in Linux, most popular TCP for popular Web servers, windows10, MacOS



3 dup ACK: t_1, t_2, and t_3

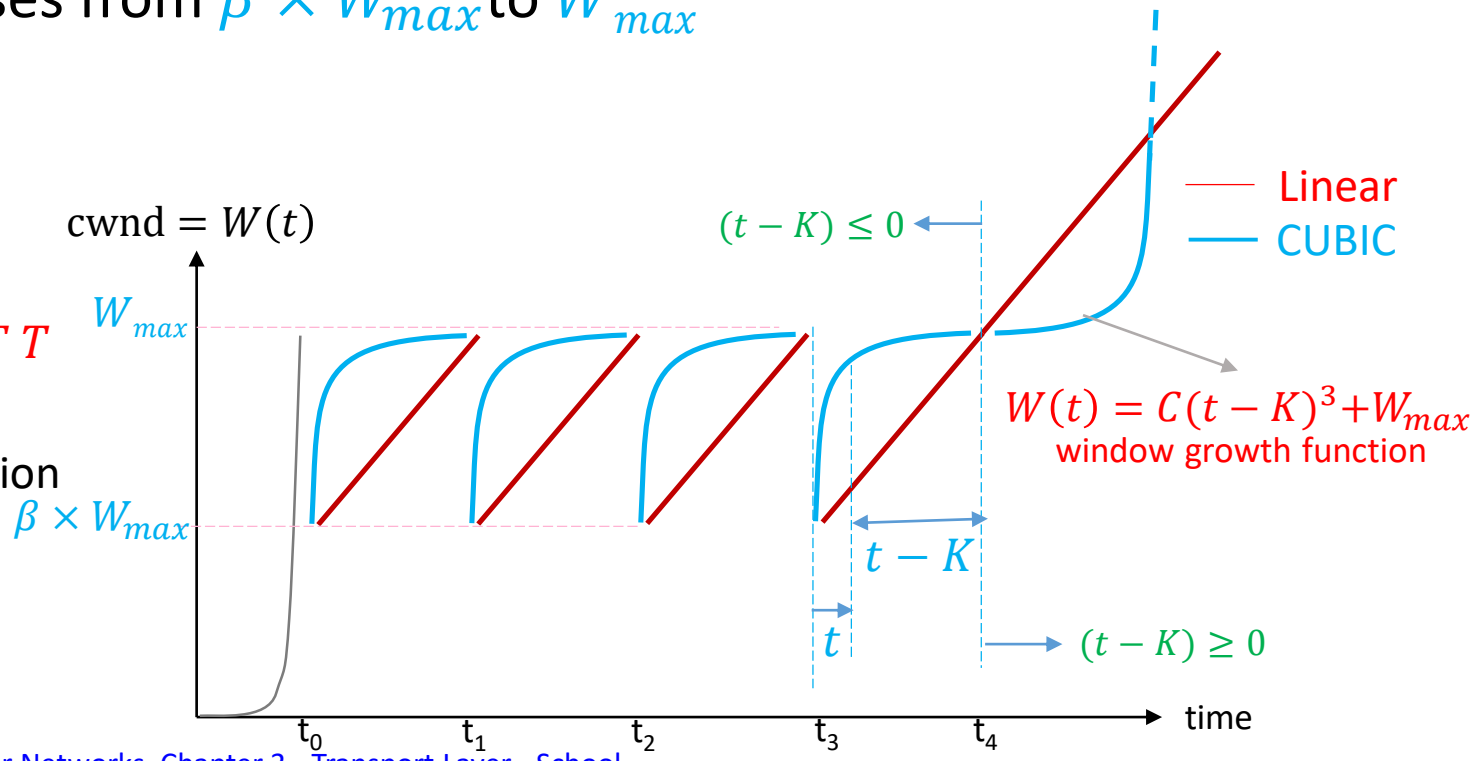
$K = t_1 - t_0 = t_2 - t_1 = \cdots$

Multiplicative decrease: $\beta \times W_{max}$

- In "3 dup ACK" TCP CUBIC registers W_{max} to be window size where loss event occurred and performs a multiplicative decrease of `cwnd` by a factor of β
- K: time duration `cwnd` increases from $\beta \times W_{max}$ to W_{max}
- CUBIC calculates parameter K:

$$K = \sqrt[3]{\frac{(1-\beta) \times W_{max}}{C}}$$

- Receiving an ACK during CA: $t = t + RTT$ and $\text{cwnd} = W(t) = C(t-K)^3 + W_{max}$
- t: elapsed time from last window reduction

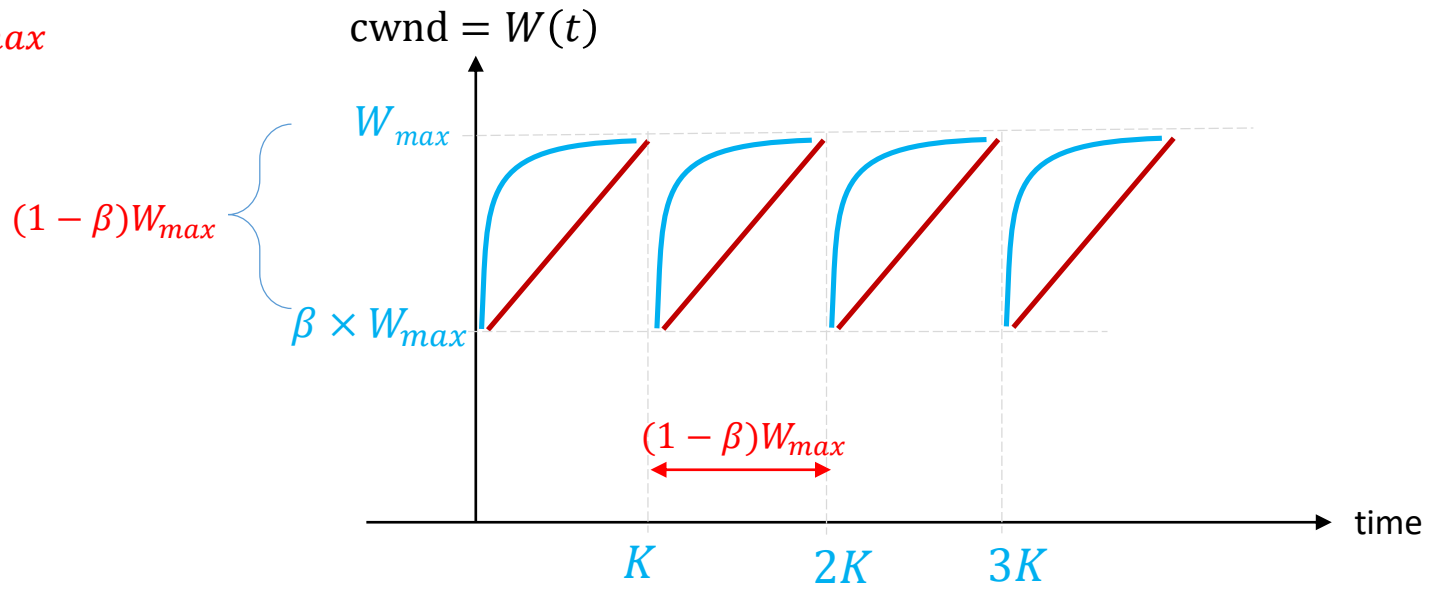


Period of CUBIC vs Period of RENO

- For "**Period of CUBIC = Period of RENO**", then:

$$K = \sqrt[3]{\frac{(1-\beta) \times W_{max}}{C}} = (1-\beta)W_{max}$$

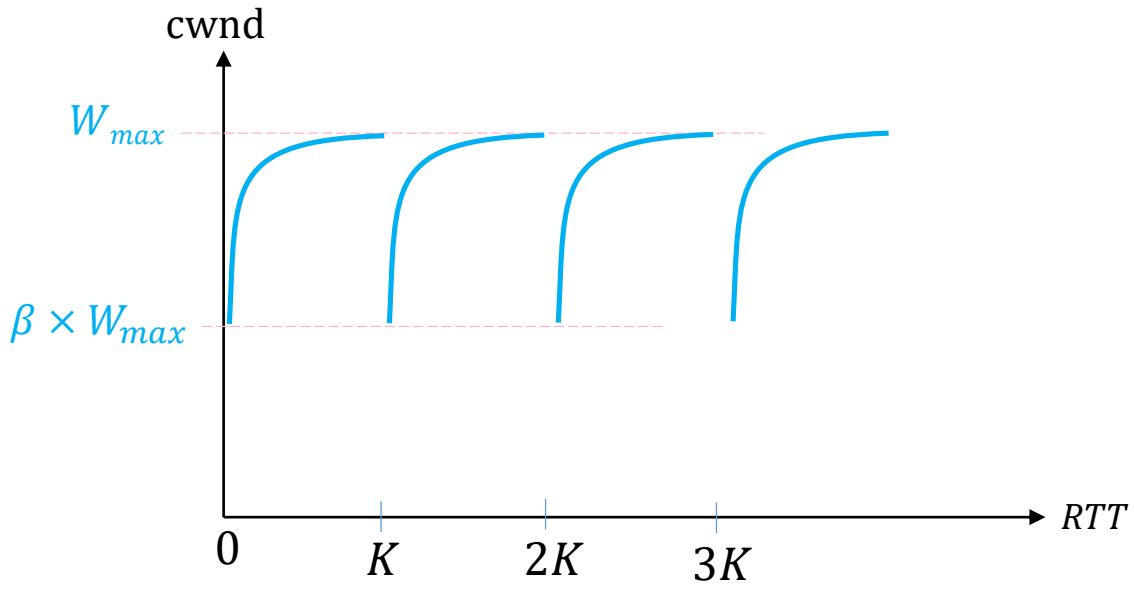
$$C = \left[\frac{1}{(1-\beta)W_{max}}\right]^2$$



- Example: $W_{max} = 16, \beta = 0.5$
- $C = 8^{-2}, K = 8$

TCP CUBIC: CA throughput

- Average Throughput = $\frac{\int_0^K C(t-K)^3+W_{max}dt}{K\times RTT} = \frac{K\times W_{max}-CK^4/4}{K\times RTT} = (W_{max}-\frac{CK^3}{4})/RTT$
- Average Throughput$= (W_{max}-\frac{(1-\beta)W_{max}}{4})/RTT = \frac{(3+\beta)W_{max}}{4RTT}$
- If: $\beta = 0.5$, Average Throughput= $\frac{3.5}{4}\frac{W_{max}}{RTT}$



3.7.2 Network-Assisted Explicit Congestion Notification and Delayed-based Congestion Control

- In 2001, extensions to both IP and TCP [RFC 3168] have been proposed, and then implemented and deployed
- It allows network to explicitly signal congestion to a TCP sender and receiver
- In addition, a number of variations of TCP congestion control protocols have been proposed that infer congestion using measured packet delay. We'll take a look at both network-assisted and delay-based congestion control in this section

Explicit Congestion Notification

- TCP and IP are involved
- ECN: two bits in Type of Service of IP header (Section 4.3)
  - ECN=10: sending host informs routers that sender and receiver are ECN-capable, (taking action in response to ECN-indicated network congestion)
  - ECN=11: router indicates experiencing congestion (before full buffer causes packets to be dropped)
  - ECN=11 is carried in marked IP datagram to destination host, which then informs sending host

Explicit Congestion Notification Echo Congestion Window Reduced

- TCP receiver sets ECE=1 (Explicit Congestion Notification Echo) in a receiver-to-sender TCP ACK segment
- TCP sender halve congestion window (`cwnd`), as it would react to a 3 duplicate ACKs using fast retransmit, and sets CWR=1 (Congestion Window Reduced) bit in header of next transmitted TCP sender-to-receiver segment

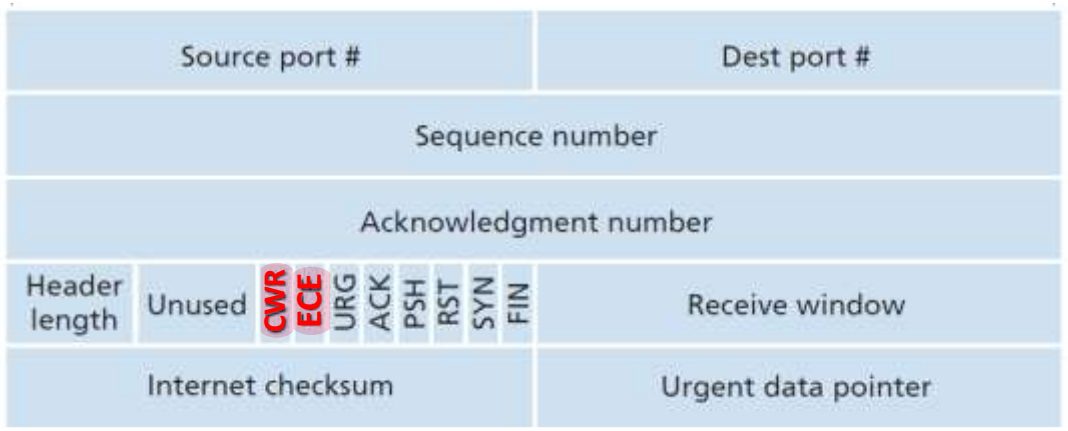


Figure 3.29 TCP header

TCP&IP Network-Assisted congestion control

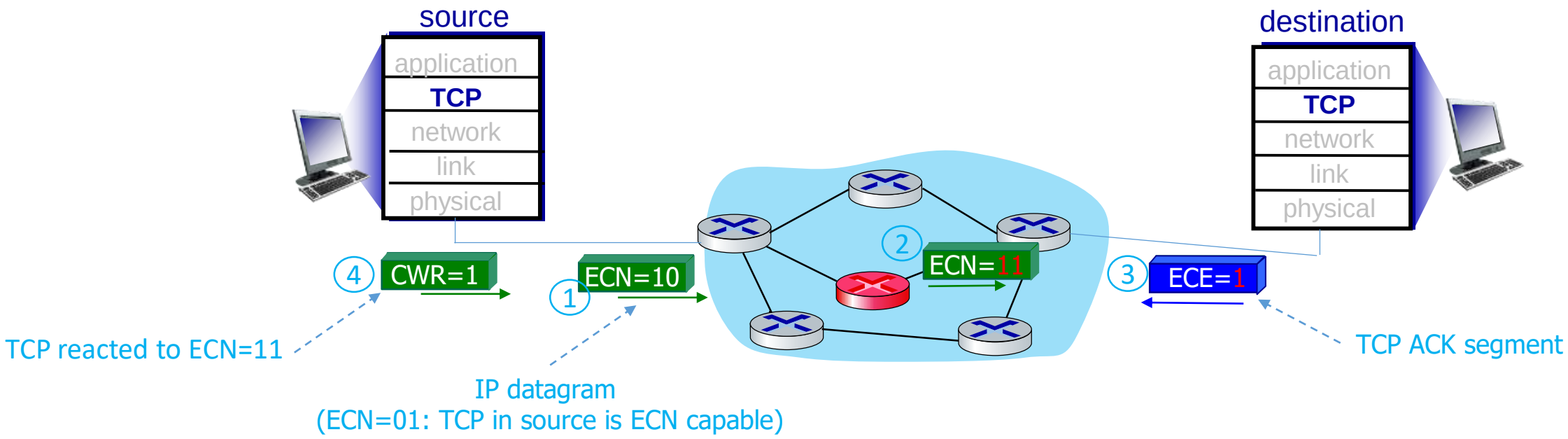


Figure 3.55 Explicit Congestion Notification: network-assisted congestion control

Other transport-layer protocols use ECN

- Datagram Congestion Control Protocol (DCCP) [RFC 4340] provides a low-overhead, congestion-controlled UDP-like unreliable service that utilizes ECN
- Data Center TCP (DCTCP) [RFC 8257] and Data Center Quantized Congestion Notification (DCQCN) designed specifically for data center networks, also makes use of ECN
- Recent Internet measurements show increasing deployment of ECN capabilities in popular servers as well as in routers along paths to those servers

Delay-based Congestion Control-TCP Vegas [1995]

- Detect congestion before packet loss occurs
- TCP sender measures RTT for all ACKed packets
- In A TCP session **minimum of RTTs** is measured: RTT_{min}
- This occurs when path is **uncongested** and packets experience minimal queueing delay
- If for measured RTT_{min} window be `cwnd`, then:

  Average of Uncongested (Maximum) Throughput=`cwnd`/ RTT_{min}

Delay-based Congestion Control-TCP Vegas

Vegas Algorithm:

- Sender measure RTT and keep latest RTT_{min}, and compute Average of Max Throughput
- If **sender-measured throughput** is close to this value, TCP sending rate can be **increased** (since path is not yet congested). If **throughput** is significantly less than uncongested throughput rate, path is congested and Vegas TCP sender will **decrease** its sending rate

Target is:

- "Keeping the pipe full": bottleneck link is kept busy transmitting
- "but no fuller": queues are not allowed to build up while pipe is kept full

Delay-based Congestion Control-BBR

- Bottleneck Bandwidth and Round-trip propagation time (BBR) congestion control protocol [2017] builds on ideas in TCP Vegas, and incorporates mechanisms that allows it compete fairly (Section 3.7.3) with TCP non-BBR senders
- BBR measures connection's **maximum delivery rate** (bottleneck bandwidth) using **derivatives of sent data respect to time** on every ACKed packet and **minimum round trip time**
- Google began using BBR for all TCP traffic on its private B4 network that interconnects Google data centers, replacing CUBIC
- BBR deployed on Google and YouTube Web servers
- Other delay-based TCP congestion control protocols include TIMELY for data center networks [2015], and Compound TCP (CTPC) [2006] and FAST [2006] for high-speed and long distance networks

3.7.3 Fairness

- Consider K TCP connections, each with a different end-to-end path, but all passing through a bottleneck link with transmission rate R bps
  - Bottleneck link: for each connection, all other links along connection's path are not congested and have abundant transmission capacity as compared with transmission capacity of bottleneck link
- Suppose each of K TCP connection is transferring a large file and there is no UDP traffic passing through bottleneck link
- A congestion control mechanism is said to be fair if average transmission rate of each connection is approximately R/K; each connection gets an equal share of link bandwidth

Is TCP's AIMD algorithm fair

- TCP connections may **start at different times** and thus may have **different window** sizes at a given point in time?
- Consider simple case of two TCP connections sharing a single link with transmission rate R
- Assume two connections have same MSS and RTT (if they have same `cwnd` size, then same throughput)
- Assume they have a large amount of data to send, and that no other TCP connections or UDP datagrams traverse this shared link
- Also, ignore slow-start phase of TCP and assume TCP connections are operating in CA mode (AIMD) at all times

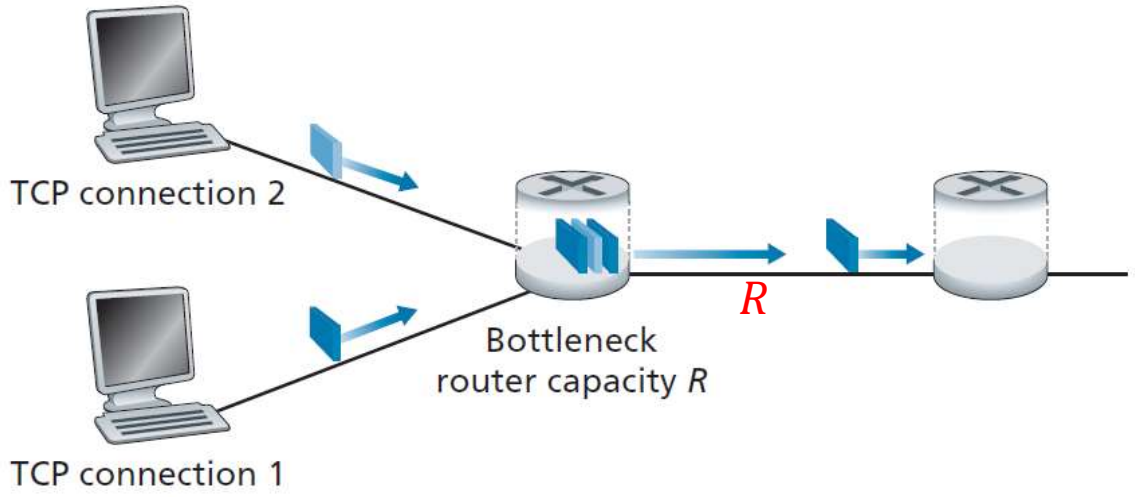
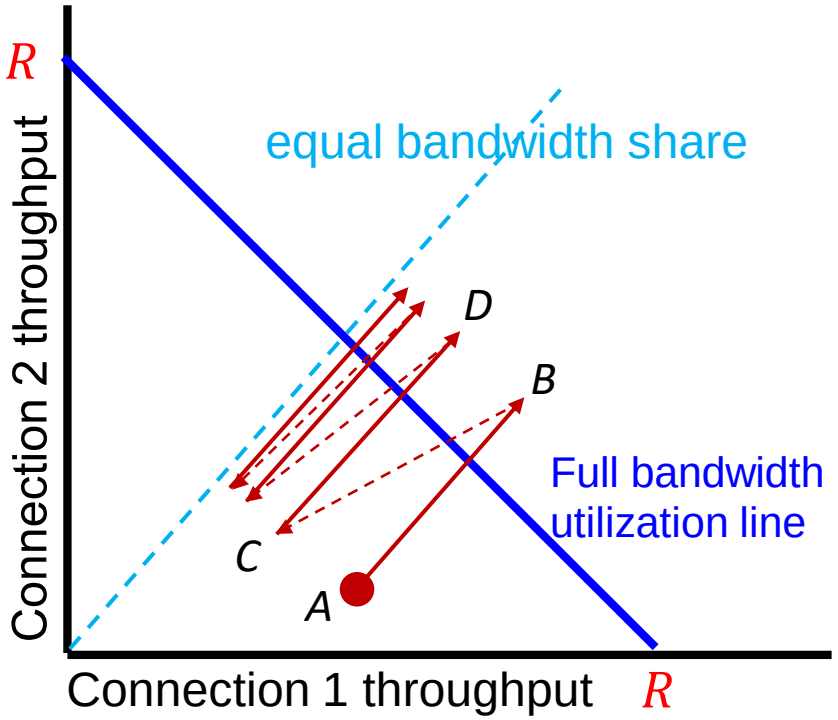


Figure 3.56 Two TCP connections sharing a single bottleneck link

Figure 3.57- AIMD

- Point A: both increase their window by 1 MSS per RTT. Joint throughput proceeds along a 45-degree line (increase for both 1 packet per RTT)
- Eventually, joint throughput will be greater than R, and packet loss will occur
- Suppose connections 1 and 2 experience packet loss, then **decrease their windows by a factor of two** Throughputs at point C, halfway along a vector starting at B and ending at origin
- Two connections again increase their throughputs along a 45-degree line starting from C



Fairness in practice

- We assumed:
  - only TCP connections traverse bottleneck link
  - connections have same RTT value
  - only a single TCP connection is associated with a host-destination pair
- In practice, these conditions are typically not met, client-server applications can obtain very **unequal portions of link bandwidth**
- When multiple connections share a common bottleneck, sessions with a smaller RTT are able to grab available bandwidth at that link more quickly as it becomes free (smaller RTT: expand their congestion windows faster) and thus will enjoy higher throughput than those connections with larger RTTs

Fairness and UDP

- Some multimedia applications do not run over TCP:
  - to avoid connection setup time
  - can pump their audio and video into network at a constant rate (they do not want their transmission rate throttled)
- From perspective of TCP, multimedia applications running over UDP are not being fair
- A number of congestion-control mechanisms have been proposed for Internet that prevent UDP traffic from bringing Internet's throughput to a grinding halt [RFC 4340]

Fairness and Parallel TCP Connections

- Web browsers (http1.1) often use multiple parallel TCP connections to transfer multiple objects within a Web page, (number of multiple connections is configurable in most browsers)
- When an application uses multiple parallel connections, it gets a larger fraction of bandwidth in a congested link
- Example: consider a link of rate R supporting 9 ongoing client-server applications, with each of applications using one TCP connection. If a new application comes along and also uses one TCP connection, then each application gets approximately the same transmission rate of $R/10$.
- But if this new application instead uses 11 parallel TCP connections, then new application gets an unfair allocation of more than $R/2$

Contents

3.8 Evolution of Transport-Layer Functionality

- There are several newer versions of TCP that have been developed, implemented, deployed, and are in significant use today

- These include TCP CUBIC, DCTCP, CTCP, BBR, and more
- Measurements indicate CUBIC and CTCP are more widely deployed on Web servers than classic TCP Reno
- BBR is being deployed in Google's internal B4 network, as well as on many of Google's public-facing servers

3.8 Evolution of Transport-Layer Functionality

- There are versions of TCP specifically designed for use, over high-bandwidth paths with large RTTs (Long, fat pipes), over wireless links for paths with packet re-ordering, and for short paths strictly within data centers, …

Scenario	Challenges
Long, fat pipes	Many packets “in flight”; loss shuts down pipeline
Wireless links	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive (short paths)
Background traffic flows	Low priority, “background” TCP flows
Priorities among TCP	Competing for bandwidth at a bottleneck link based on priorities
Segments over paths	Different source-destination paths in parallel

Evolution of Transport-Layer Functionality

- There are also variations of TCP that deal with packet acknowledgment and TCP session establishment/closure differently than we studied in Section 3.5.6

- Indeed, it's probably not even correct anymore to refer to "the" TCP protocol; perhaps the only common features of these protocols is that they use:
  - TCP segment format that we studied in Figure 3.29 (TCP segment structure)
  - They should compete "fairly" amongst themselves in face of network congestion

QUIC: Quick UDP Internet Connections

- QUIC protocol [2017]
- QUIC is a new **application-layer protocol** designed for improving performance of transport-layer services for secure HTTP
- QUIC has been widely deployed, is still in process of being standardized as an Internet RFC
- Google has deployed QUIC on many of its public-facing Web servers, YouTube, Chrome, and Android's Google Search app
- With more than 7% of Internet traffic today now being QUIC
- QUIC uses many of approaches for **reliable data transfer**, **congestion control**, and **connection management** that we've studied in this chapter

Figure 3.58

- QUIC is an application-layer protocol, using UDP as its underlying transport-layer protocol, and is designed to interface above specifically to a simplified but evolved version of HTTP/2
- In near future, HTTP/3 will natively incorporate QUIC
- Some of QUIC's major features include:
  - Connection-Oriented and Secure
  - Streams
  - Reliable, TCP-friendly congestion-controlled data transfer

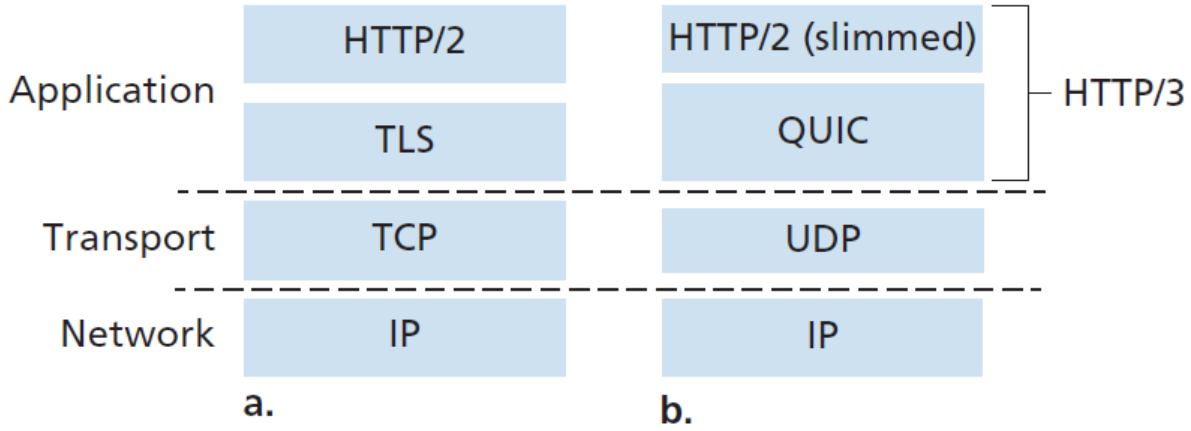


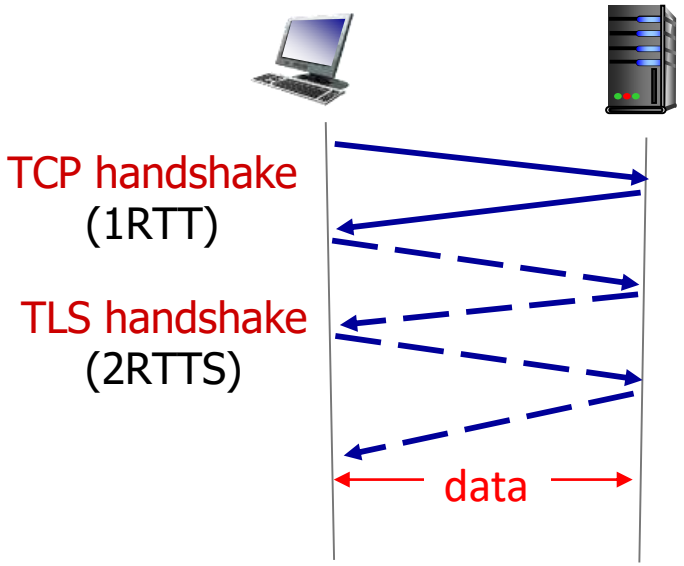
Figure 3.58 (a) traditional secure HTTP protocol stack, (b) secure QUIC-based HTTP/3 protocol stack

QUIC: Connection-Oriented and Secure

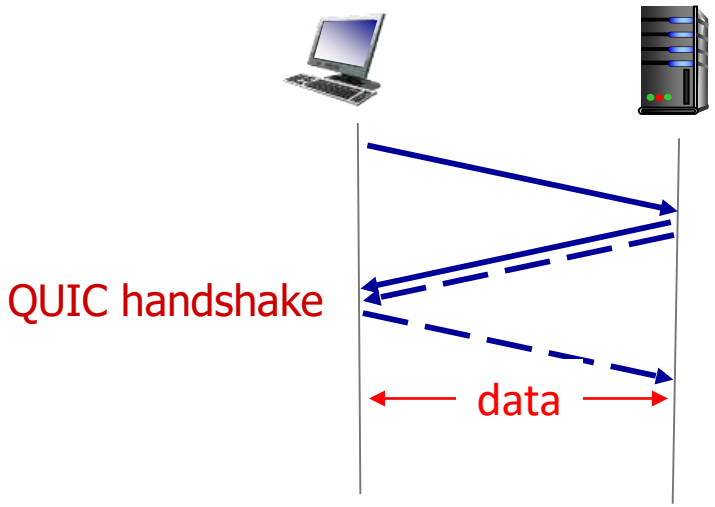
- Handshake between endpoints to set up QUIC connection state
- Connection state: Source and destination connection ID
- All QUIC packets are encrypted
- QUIC **combines** handshakes needed to establish connection state with those needed for authentication and encryption (TLS/SSL)

Connection-Oriented and Secure

- Two serial handshakes
  - TCP: reliability, congestion control state
  - TLS: authentication, crypto state



- One combined handshake
  - QUIC: reliability, congestion control, authentication, crypto state



QUIC: Streams

- QUIC allows:
  - Several different application-level "streams" to be multiplexed through **a single QUIC connection**
  - Once a QUIC connection is established, new streams can be quickly added
- A stream is an abstraction for: reliable, in-order bi-directional delivery of data between two QUIC endpoints

- In HTTP/3, there is a different stream for each object in a Web page

QUIC: Streams

- A QUIC connection has a **64 bit** connection ID (randomly selected by connection initiator)
- Each stream within a QUIC connection has a **62 bit** stream ID (unique for all streams on a connection)
  - Client-initiated streams have even-numbered stream IDs, and server-initiated streams have odd-numbered stream IDs
- Connection and Stream IDs are contained in a QUIC packet header (along with other header information)
- STREAM frames encapsulate data sent by an application
- Offset fields in STREAM frames provide ordered byte-stream delivery
- One or more QUIC packets can be encapsulated in a single UDP datagram

QUIC and SCTPS

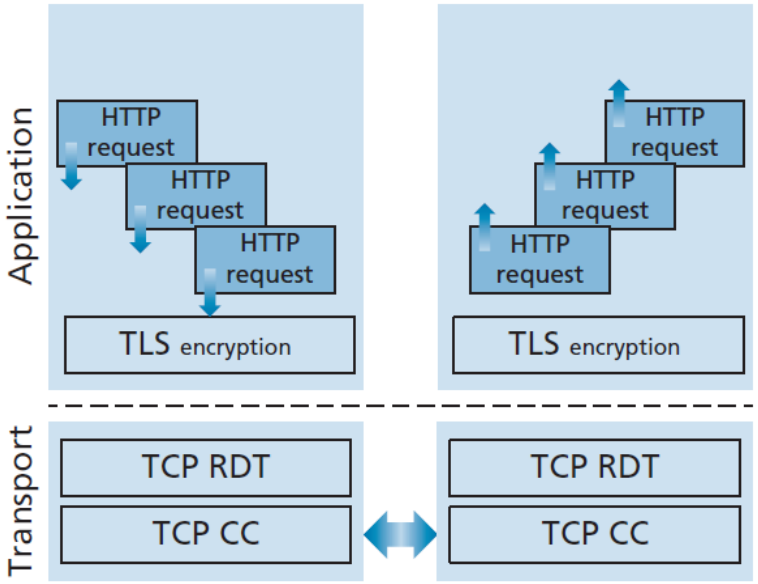
- Stream Control Transmission Protocol (SCTP) [RFC 4960, RFC 3286] is an earlier protocol that is:
  - reliable
  - message-oriented protocol
  - multiplexing multiple application- level “streams” through a single SCTP connection
- SCTP is used in control plane protocols in 4G/5G cellular wireless networks

QUIC: Reliable data transfer, Fig. 3.59

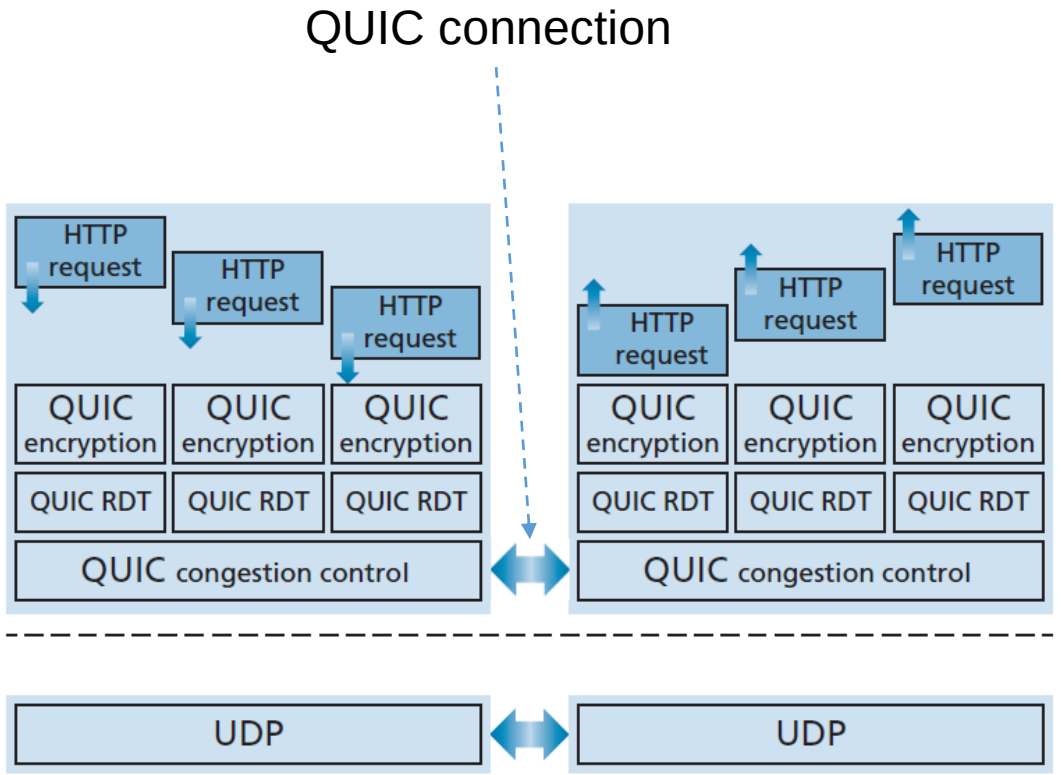
- QUIC provides reliable data transfer to each QUIC stream separately, using ACK mechanisms similar to TCP [RFC 5681]

- HTTP/1.1 sending multiple HTTP requests, all over a single TCP connection
  - TCP provides reliable, in-order byte delivery
  - Multiple HTTP requests must be delivered in order at destination HTTP server
  - If bytes from one HTTP request are lost, remaining HTTP requests can not be delivered until those lost bytes are retransmitted and correctly received by TCP at HTTP server (HOL blocking problem)
- QUIC provides a reliable in-order delivery on a per-STEAM basis
  - A lost UDP segment only impacts streams whose data was carried in that segment
  - HTTP messages in other STREAMs continue to be received and delivered to application

Figure 3.59

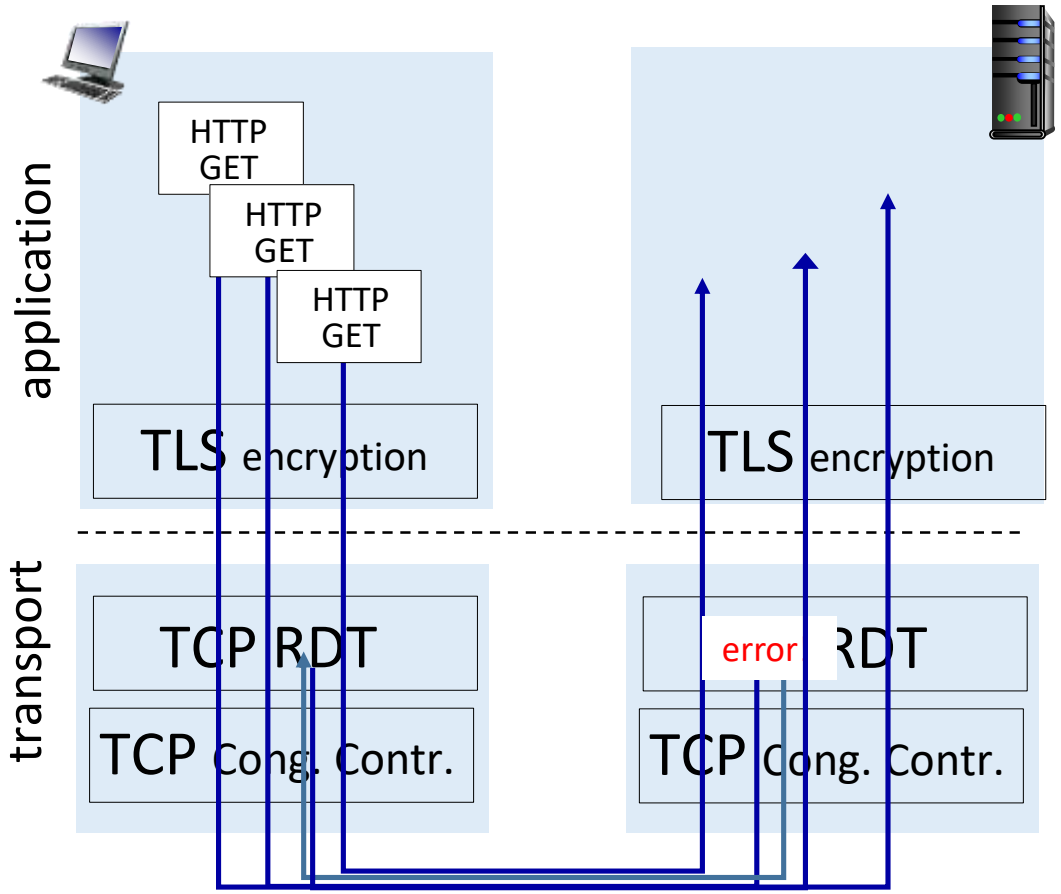


(a) HTTP/1.1: a single-connection client and server using application-level TLS encryption over TCP's reliable data transfer (RDT) and congestion control (CC)



(b) HTTP/3: 3-STREAM client and server using QUIC's encryption, and reliable data transfer over a congestion controlled QUIC connection over a UDP instance

Figure 3.59



(a) HTTP 1.1

QUIC's TCP-friendly congestion-control

- QUIC's congestion control is based on TCP New Reno [RFC 6582], a slight modification to TCP Reno protocol that we studied in Section 3.7.1
- QUIC's Draft specification [https://datatracker.ietf.org/doc/draft-ietf-quic-transport/] notes:
  - "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well known TCP ones"
- We've studied TCP's congestion control in Section 3.7.1

Connection migration

- QUIC uses a connection ID and can survive changes to endpoint addresses (IP address and port)
- During a QUIC connection, an endpoint can migrating to a new network using a new IP address and even new port number
- An endpoint MUST NOT initiate connection migration before handshake is confirmed

QUIC summary

- QUIC is an application-layer protocol providing
- Reliable
- Flow control
  - A QUIC receiver controls maximum amount of data sender can send on a STREAM as well as across all streams at any time
- Congestion-controlled data transfer between two endpoints
- Ordered byte stream for each STREAM
- Connection migration capable
- Changes can be made to QUIC at "application-update timescales," that is, much faster than TCP or UDP update timescales

3.9 Summary

- Services a transport-layer protocol can provide to APPs
  - Very simple services (like UDP's): multiplexing/ demultiplexing function for communicating processes
  - Variety of guarantees, such as reliable delivery of data, delay guarantees, bandwidth guarantees, security, …
- Transport protocol services are constrained by service model of underlying network-layer protocol. If network-layer protocol cannot provide delay or bandwidth guarantees to transport-layer segments, then transport layer protocol cannot provide delay or bandwidth guarantees for messages sent between processes

3.9 Summary

- Transport-layer protocol can provide reliable data transfer even if underlying network layer is unreliable.
- Reliable data transfer can be provided by link-, network-, transport-, or application-layer protocols
- TCP is complex, involving connection management, flow control, and round-trip time estimation, as well as reliable data transfer
- TCP is actually more complex than our description
  - we intentionally did not discuss a variety of TCP patches, fixes, and improvements that are widely implemented in various versions of TCP
  - If a client on one host wants to send data reliably to a server on another host, it simply opens a TCP socket to server and pumps data into that socket
  - client-server APPs is blissfully unaware of TCP's complexity

3.9 Summary

- Congestion control is imperative for well-being of network
- TCP implements an end-to-end congestion-control mechanism that additively increases its transmission rate or multiplicatively decreases
- TCP congestion control has evolved over years, it remains an area of intensive research and will likely continue to evolve in upcoming years
- TCP connection establishment and termination
- Recent developments of many transport layer's functions in application layer (QUIC)
- Our discussion of network edge is complete
- Next, we study network core (next two chapters, we study network layer, and then link layer in Chapter 6)

Appendix

Well-known PORTs and Apps

- Server process waits on an open port for contact or data from a client
- Ports for well-known applications: Web:80, FTP:21, DNS: , SMTP:25, Microsoft Windows SQL:1434 (UDP), ….
- If we determine open ports on a host, we know Apps running on host.
- This is very useful for system administrators
- But attackers try to find running Apps on hosts, and then to attack
- Example: SQL server listen on UDP port:1434 and is subject to a buffer overflow, allowing attacker to execute arbitrary code on vulnerable host

PORT SCANNING - Network Mapper (nmap)

- There are a number of public domain programs, called port scanners
- nmap pert scanner is a freely available at http://nmap.org and included in most Linux distributions
- For TCP, nmap sequentially scans ports, looking for ports that are accepting TCP connections
- For UDP, nmap again sequentially scans ports, looking for UDP ports that respond to transmitted UDP segments
- In both cases, nmap returns a list of open, closed, or unreachable ports
- A host running nmap can attempt to scan any target host *anywhere* in Internet.

Port scanning example

Exploring TCP port = 6789, on a target host:

- nmap sends a TCP SYN segment with destination port 6789 to host. There are three possible outcomes:

1. nmap receives a TCP SYNACK segment from target host. This means an application is running with TCP port 6789 on target post
2. nmap receives a TCP RST segment from target host. This means target host is not running an application with TCP port 6789. But attacker at least knows that segments destined to host at port 6789 are not blocked by any firewall on path between source and target hosts
3. nmap receives nothing. This likely means that SYN segment was blocked by an intervening firewall and never reached the target host

Flag RST (reset) in TCP header

- What happens a host receives a TCP segment whose port numbers or source IP address do not match with any of ongoing sockets
- For example, suppose a host receives a TCP SYN packet with destination port 80, but host is not running a Web server on port 80 Then host sends back a RST=1 segment
- It is telling "I don't have a socket for that segment. Please do not resend segment."

- When a host receives a UDP packet whose destination port number doesn't match with an ongoing UDP socket, host sends a special ICMP datagram

SYN FLOOD ATTACK

- In TCP's three-way handshake, if server allocates and initializes connection variables and buffers in response to a received SYN:
- Attacker(s) send a large number of TCP SYN segments, without completing third handshake step. Server's connection resources become exhausted as they are allocated for half-open connections; legitimate clients are then denied service
- It is a Denial of Service (DoS) attack known as SYN flood attack

- Defense strategy: Server allocates resources just after do so

SYN cookies

Effective defense known as SYN cookies [RFC 4987] are now deployed in most major operating systems. SYN cookies work as follows:

- When server receives a SYN segment, it does not know if segment is coming from a legitimate user or is part of a SYN flood attack
- Server creates an initial TCP sequence number that is hash function of source and destination IP addresses and port numbers of SYN segment, as well as a secret number only known to server. This initial sequence number is so-called "cookie." Server then sends client a SYNACK packet with this initial sequence number
- Importantly, server does not remember cookie or any other state information corresponding to this SYN

SYN cookies

- When server receives ACK for SYNACK from a legitimate client , verifies that ACK corresponds to some SYN sent earlier
- Value in acknowledgment field is equal to server's initial sequence number+1
- Server then run hash function using source and destination IP address and port numbers in ACK (which are same as in original SYN) and secret number. If result of function plus one is same as acknowledgment (cookie) value in client's ACK, server concludes that ACK corresponds to an earlier SYN segment and is hence valid. Server then creates a fully open connection along with a socket
- If client does not return an ACK for SYNACK, then original SYN has done no harm at server, since server hasn't yet allocated any resources in response to original bogus SYN

TCP SPLITTING: OPTIMIZING THE PERFORMANCE OF CLOUD SERVICES

- For cloud services such as search, e-mail, and social networks, it is desirable giving users illusion that services are running within their own end systems
- Users are often located far away (large RTT) from data centers of cloud services
- There can be significant packet loss in access networks, leading to TCP retransmissions and even larger delays
- Potentially leading to poor response time performance due to TCP slow start
- Example: Consider delay in receiving a response for a search query:
  - one RTT for TCP connection plus three RTTs for three congestion windows of data plus the processing time in the data center
  - Four RTTs lead to a noticeable delay in returning search results

TCP SPLITTING

- One way to mitigate this problem and improve user-perceived performance is to
  1. deploy front-end servers closer to the users, and
  2. utilize **TCP splitting** by breaking the TCP connection at the front-end server

- TCP splitting:
  - client establishes a TCP connection to nearby front-end,
  - front-end maintains a persistent TCP connection to data center with a very large TCP congestion window

TCP SPLITTING

- Response time = $4RTT_{FE} + RTT_{BE}$ + processing time,
  - RTT_{FE} = round-trip time between client and front-end server, RTT_{FE} is negligibly small when front-end server is close to client
  - RTT_{BE} = round-trip time between front-end server and data center (back-end server)
- So, Response time $\approx RTT_{BE}$ + processing time
- Note: $RTT_{BE} \approx$ RTT (client- data center roundtrip time)
- TCP splitting can reduce response time roughly from 4RTT to RTT
- TCP splitting also helps reduce TCP retransmission delays caused by losses in access networks
- Google and Akamai have made extensive use of their CDN servers in access networks (Section 2.6) to perform TCP splitting for cloud services they support