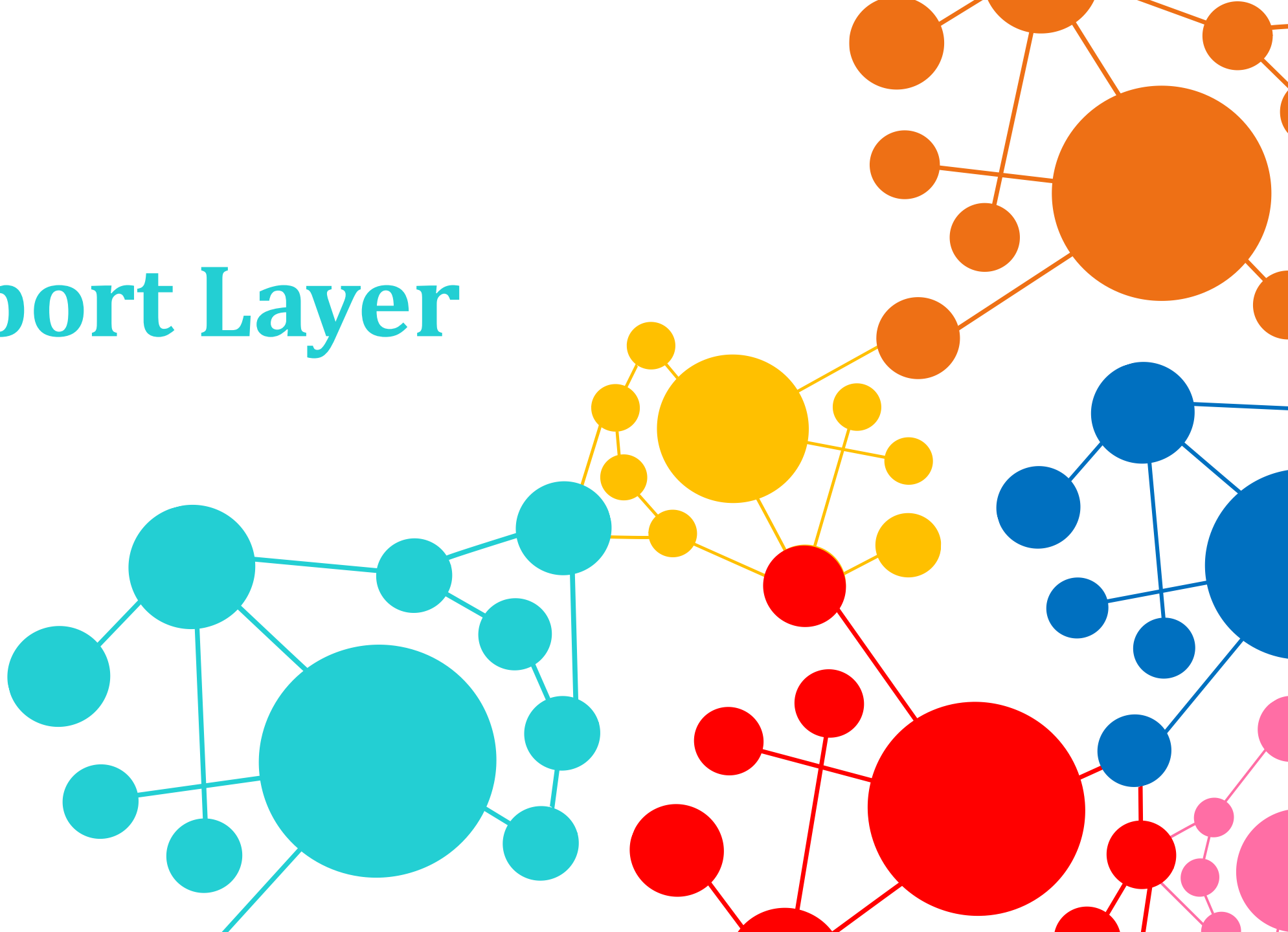
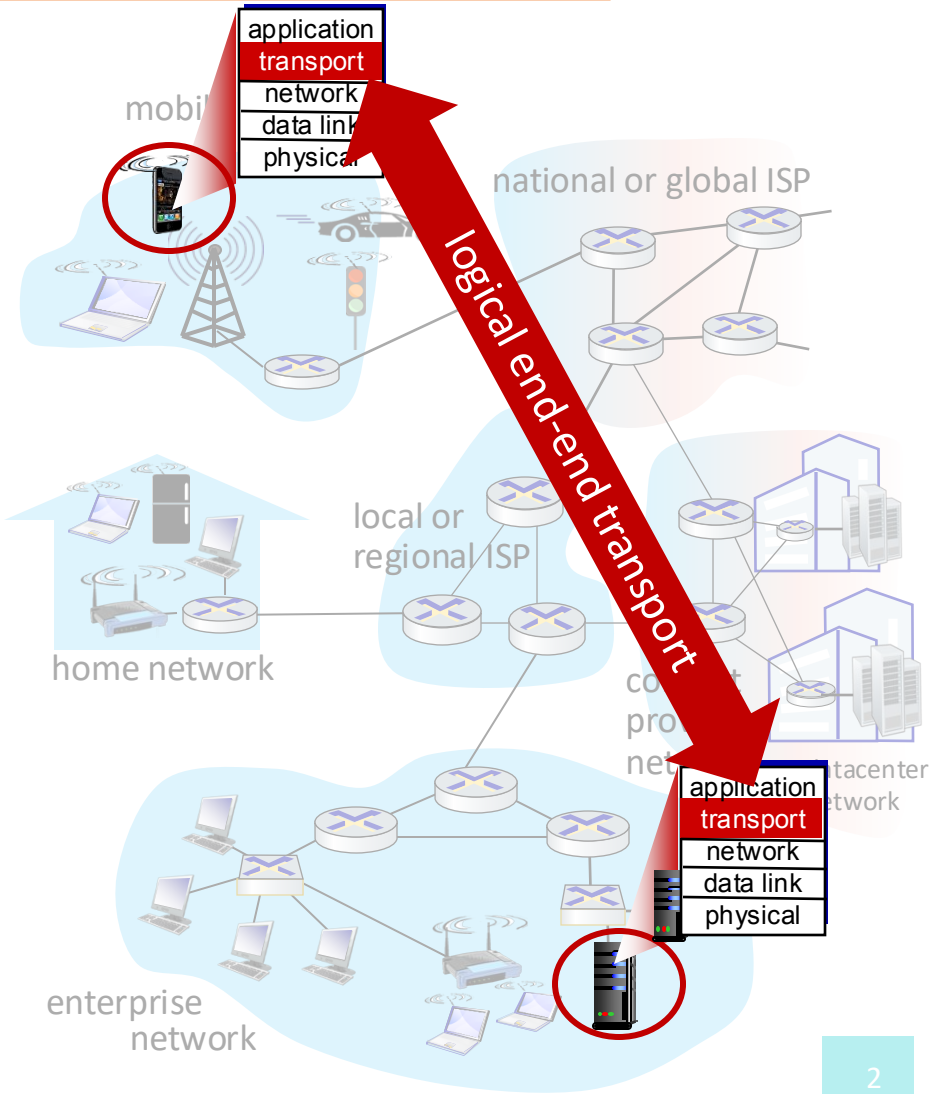
Transport Layer



Transport Layer

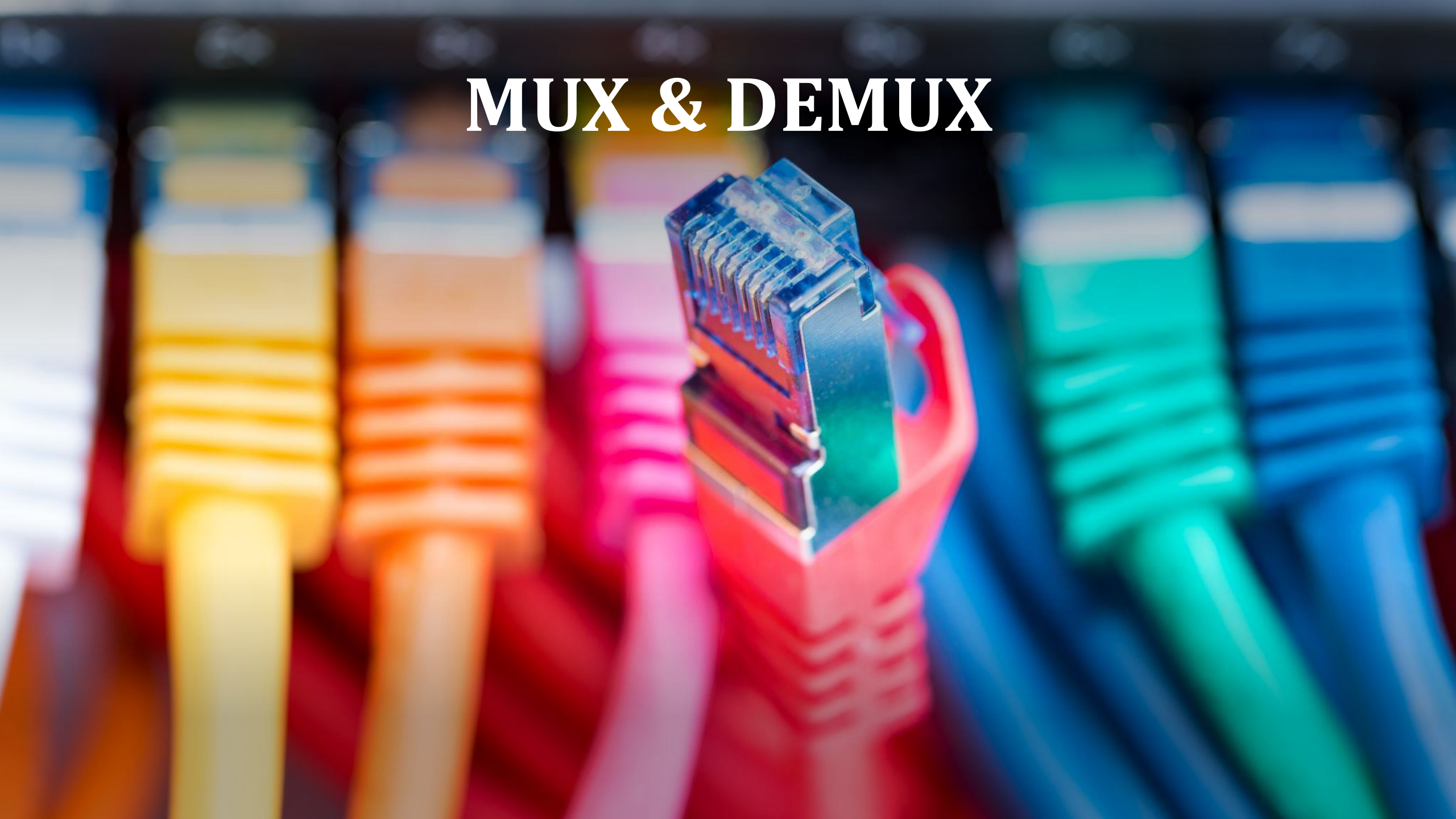
- Logical communication between processes
- Internet Transport-Layer Processes
  - **TCP**
    - Reliable Data Transfer
    - Connection Setup
    - Flow Control
    - Congestion Control
  - **UDP**
    - Best Effort IP
  - **Services not supported**
    - Delay Guarantee
    - Bandwidth Guarantees



Transport Layer

- Multiplexing and Demultiplexing
- Connectionless Transport :UDP
- Reliable Data Transfer
- Connection-oriented Transport: TCP
  - Segment Structure
  - Reliable Data Transfer
  - Flow Control
  - Connection Management
- Congestion Control
- TCP Congestion Control
- QUIC

MUX & DEMUX



Multiplexing & Demultiplexing

- Multiplex
  - Performed at **sender**
  - Handle data from multiple sockets and add transport header

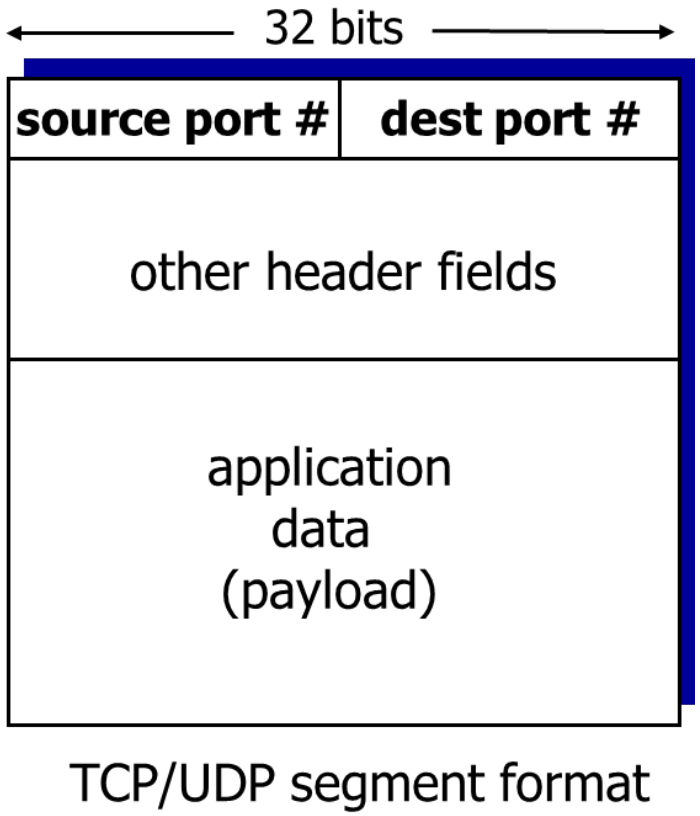
- Demultiplex
  - Performed at **receiver**
  - Use header information to deliver received segments to correct socket

Demultiplexing

- Host receives IP datagrams

- Each **datagram**
  - Source IP address
  - Destination IP address
  - Carries one Transport-Layer **Segment**

    Segment has
    - Source **port** number
    - Destination **port** number



TCP/UDP segment format

- **IP Address + Port number** used to direct segment to appropriate socket

Demultiplexing

- **Connectionless**
  - Information from datagrams used for Demux
    - Destination IP
    - Destination port number

- **Connection-Oriented**
  - Information from datagrams used for Demux
    - Source IP
    - Destination IP
    - Source port number
    - Destination port number

Connectionless Demux

- IP datagrams with **same destination port number,** but different source IP addresses and/or source port numbers will be directed to **same socket** at destination

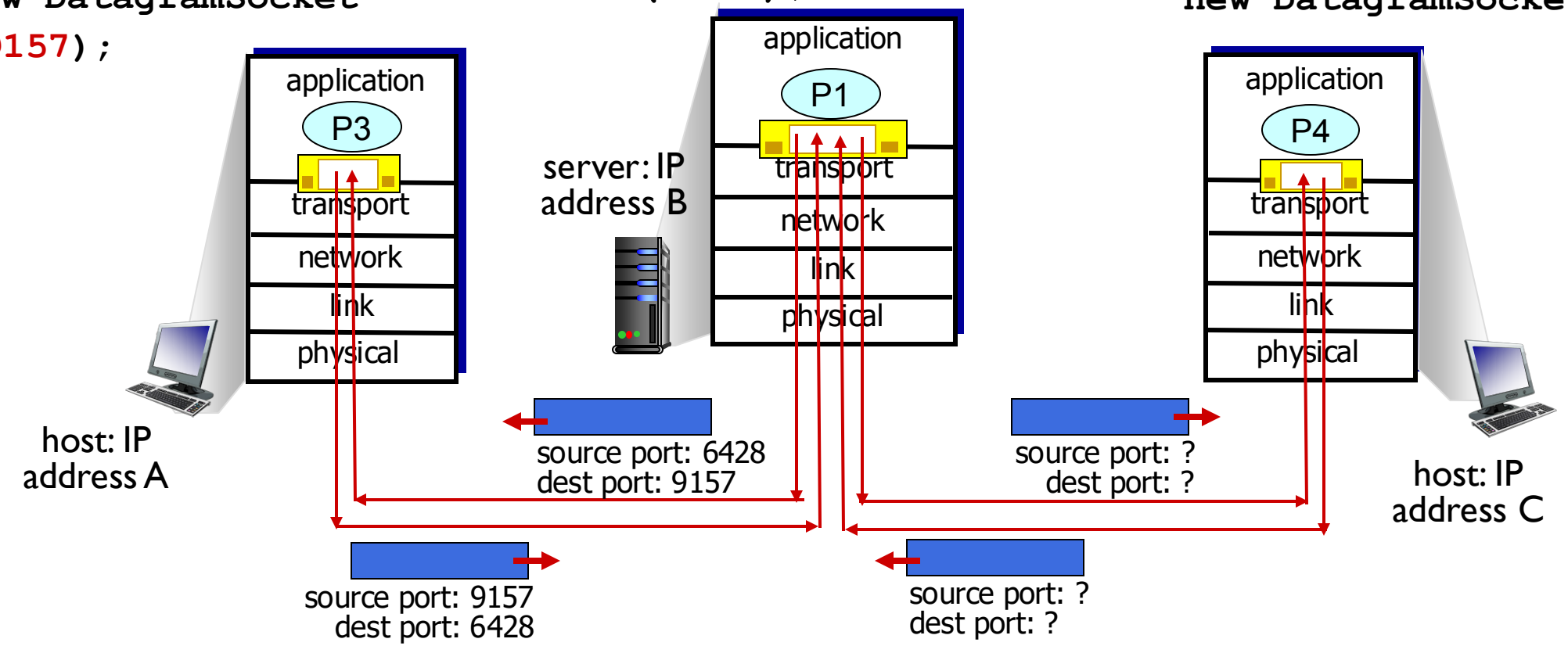
- Socket has host-local port number

Connectionless Demux

```
DatagramSocket mySocket2 =
 new DatagramSocket
 (9157);
```

```
DatagramSocket
  serverSocket = new
  DatagramSocket
  (6428);
```

```
DatagramSocket mySocket1 =
 new DatagramSocket (5775);
```



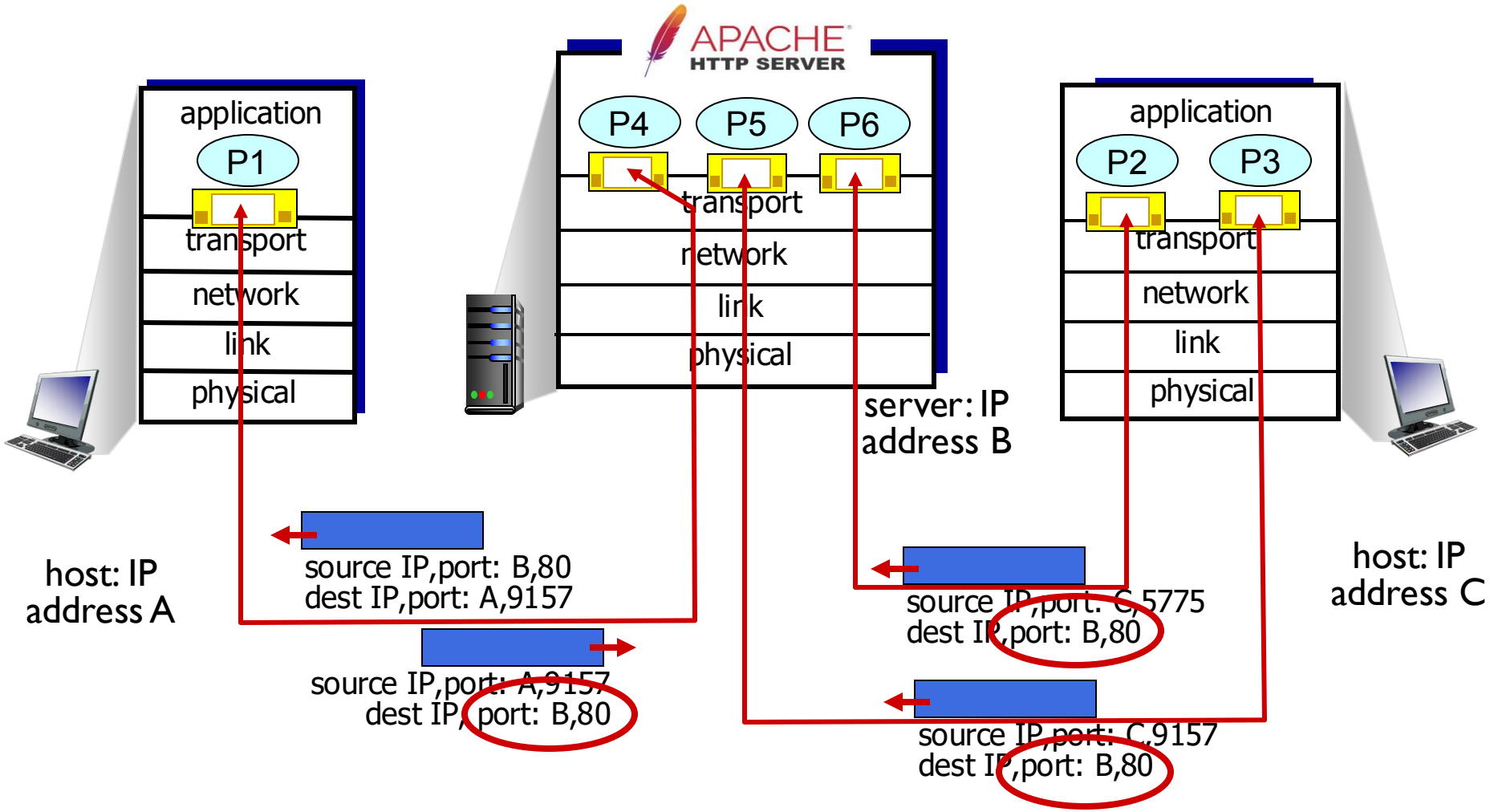
Connection-Oriented Demux

- TCP **socket** identified by 4-tuple
  - Source IP address
  - Source port number
  - Destination IP address
  - Destination port number

  Receiver uses all four values to de-mux and direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets
  - Each socket identified by its own 4-tuple
  - Web servers have different sockets for each connecting client
    - Non-persistent HTTP will have different socket for each request

Connection-Oriented Demux



Connectionless Transport

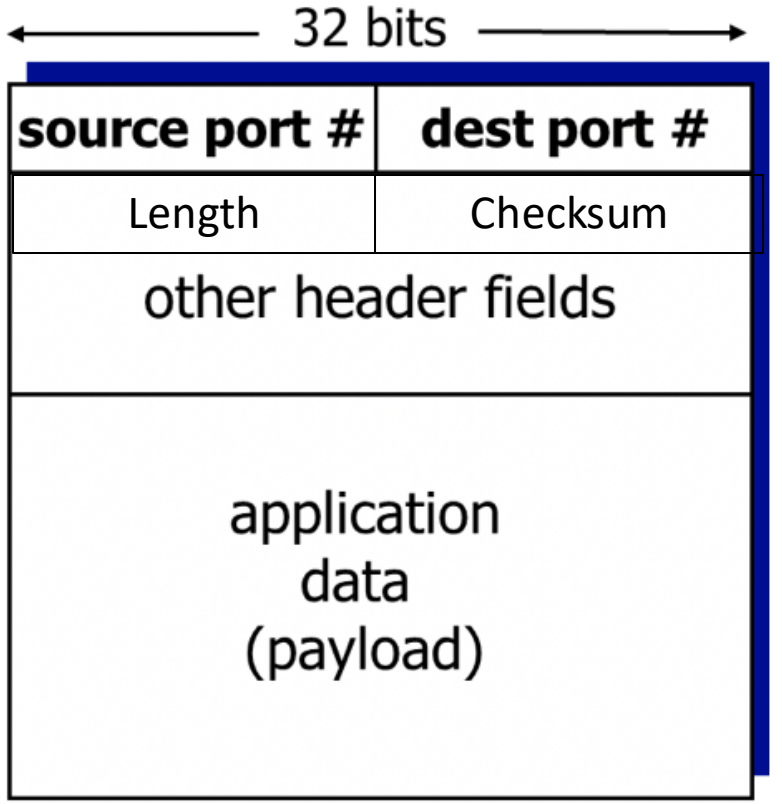
UDP

- **UDP**: User Datagram Protocol
  - Best effort
    - Loss
    - Out of order delivery
  - Connectionless
    - No handshaking between UDP sender and receiver
    - Each UDP segment handled independently of others

- Reliable Transfer over UDP is possible
  - Add reliability at application layer
  - Application-specific error recovery

UDP

- UDP Usage
  - Why?
    - No connection establishment
    - Simple: No connection state at sender and receiver
    - Small header size
    - No congestion control: Can be used as fast as desired
  - Where?
    - Streaming multimedia
      Loss tolerant & rate sensitive traffic
    - DNS
    - SNMP



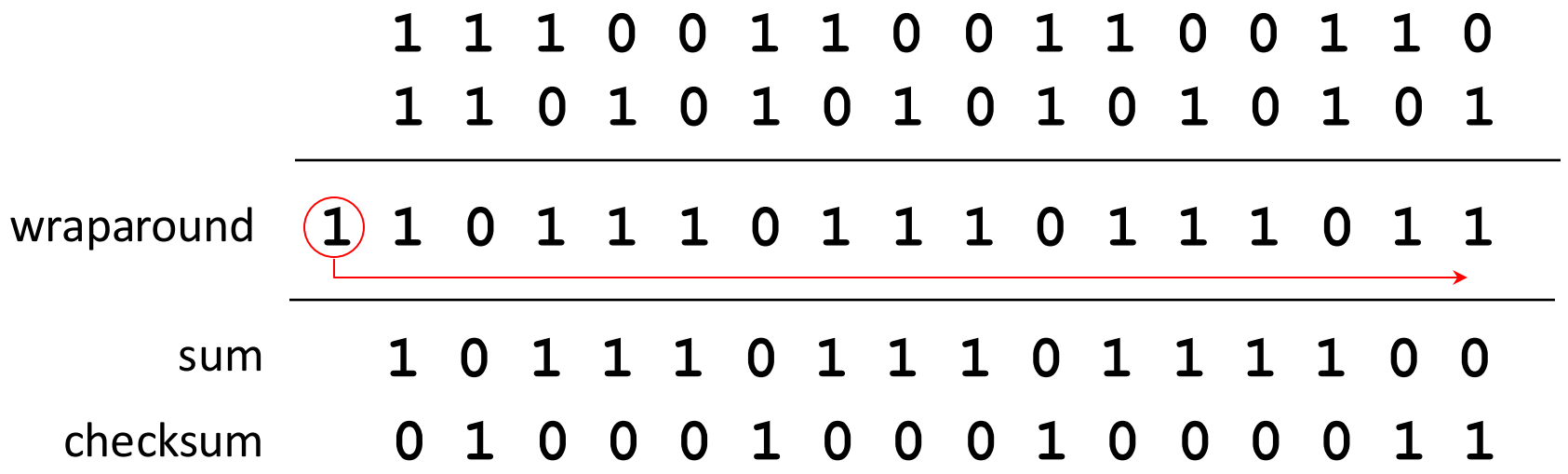
UDP Header Format

UDP

- Checksum
  - Sender
    - Treat segment contents, including header fields, as sequence of 16-bit integers
    - Checksum: Addition (one's compliment sum) of segment contents
    - Sender puts checksum value into UDP Checksum field
  - Receiver
    - Compute checksum of received segment
    - Check if computed checksum equals checksum field value:
      - No → Error detected!
      - Yes → No errors **detected!**

UDP: Checksum

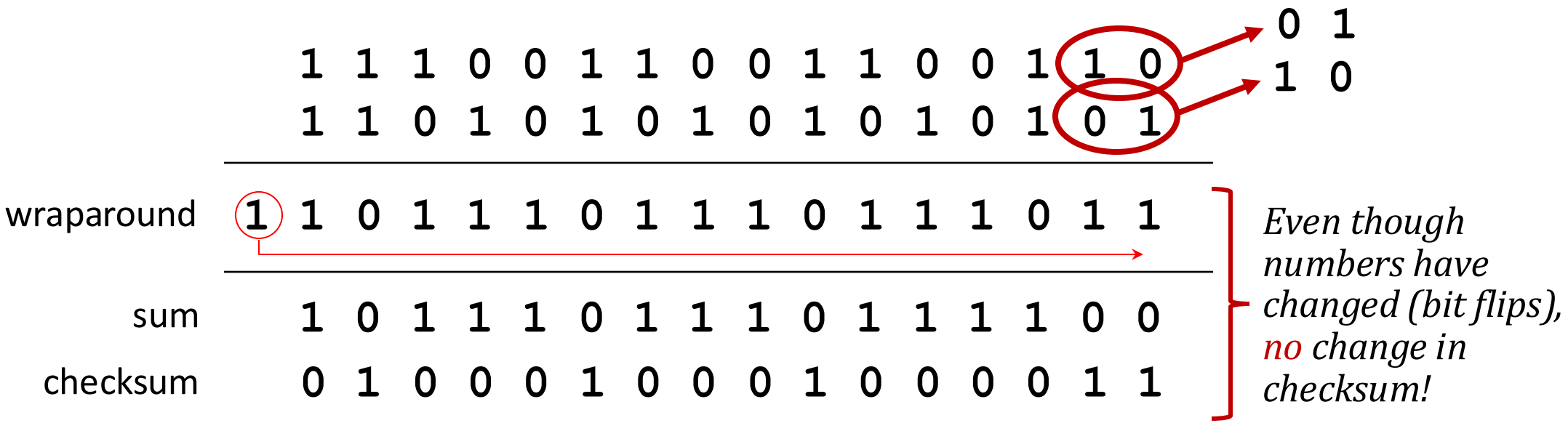
Example: Add two 16-bit integers



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

UDP: Checksum

Weak Detection!



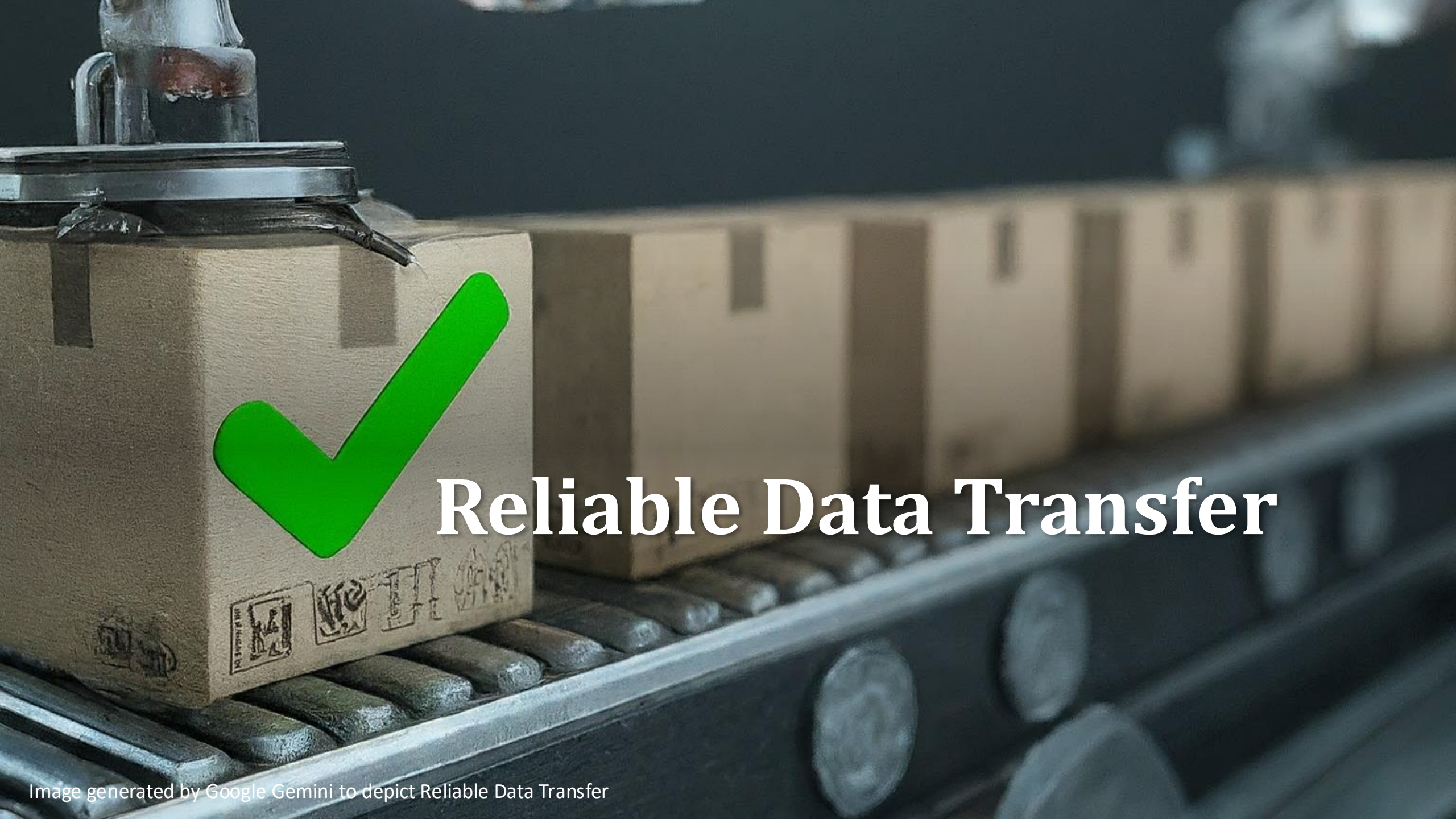
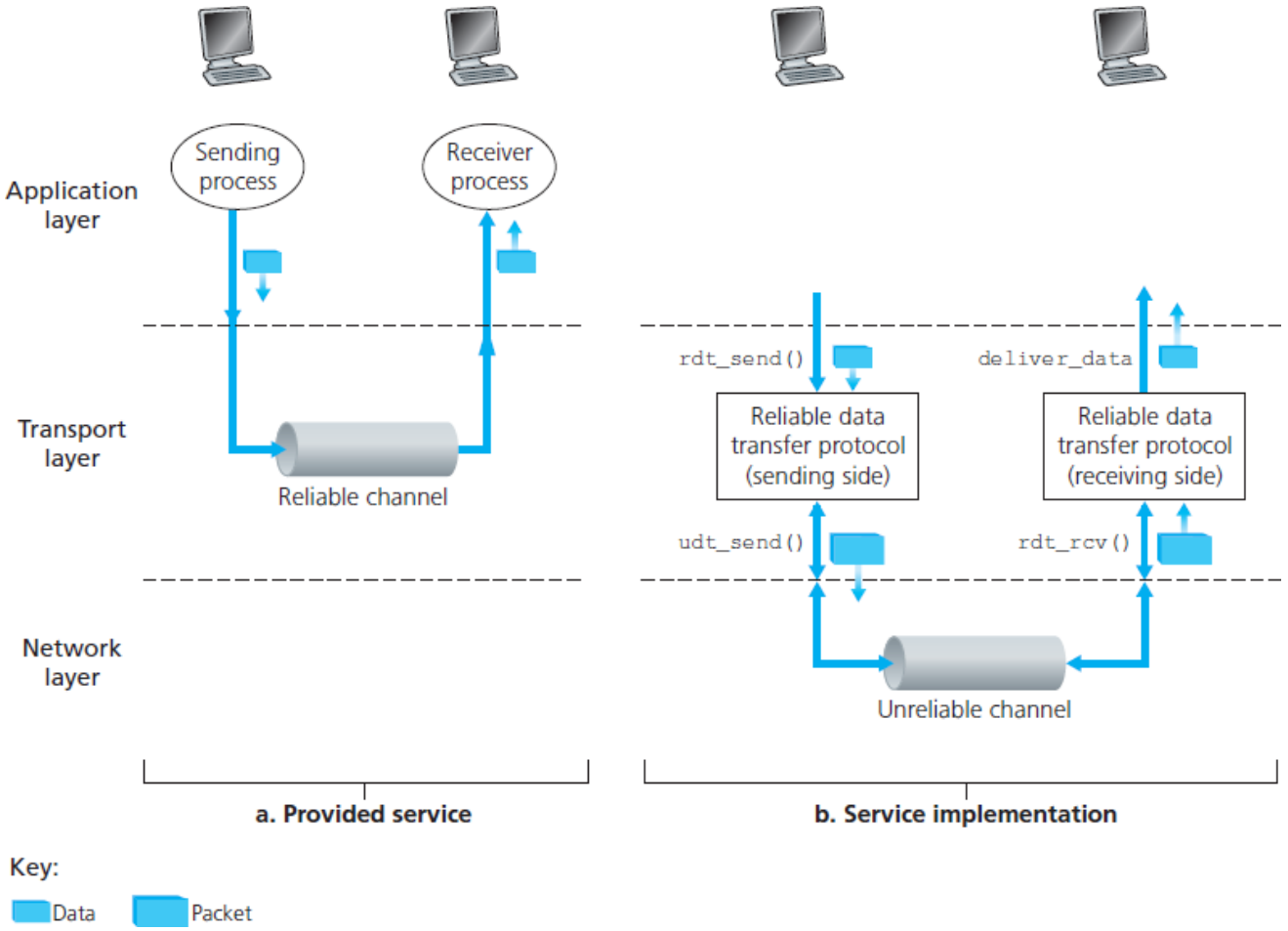


Image generated by Google Gemini to depict Reliable Data Transfer

Reliable Data Transfer

- Reliable transfer of information important for many applications
- Characteristics of unreliable channel will determine the complexity of data transfer protocol.

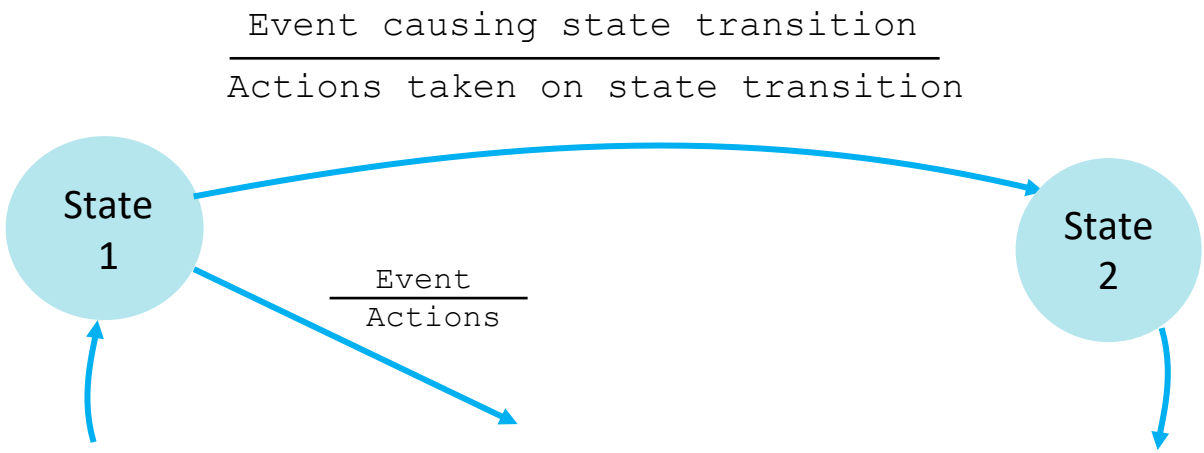


Reliable Data Transfer

To understand the needs of reliable data transfer, we will examine:

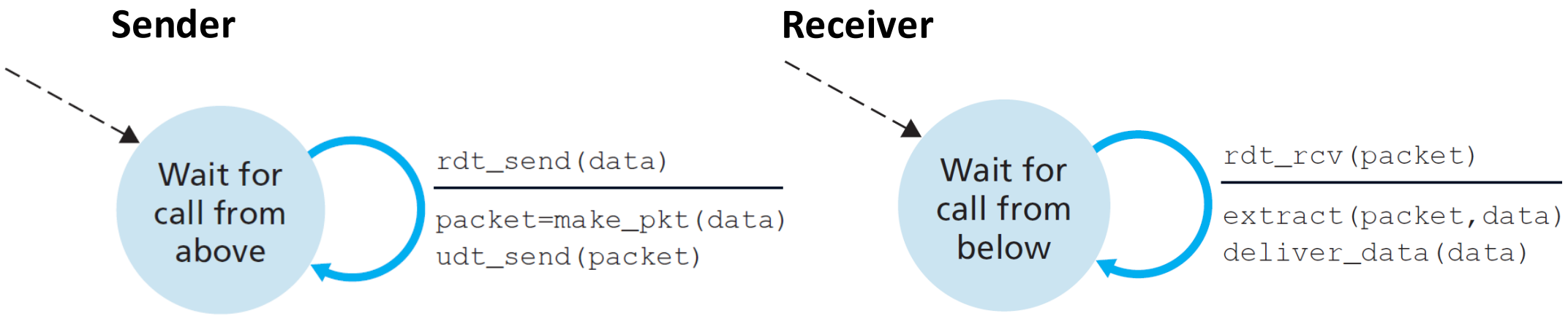
- Incrementally develop sender & receiver sides of what we need for reliable data transfer
- Consider only unidirectional data transfer but control info will flow on both directions
- Use finite state machines (FSM) to specify sender & receiver

State: When in this **state** next state uniquely determined by next event



RDT1.0: Reliable Data Transfer over a Reliable Channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
- Separate FSMs for sender and receiver
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel



RDT2.0: Reliable Data Transfer over a Channel with Bit Errors

- Underlying channel unreliable
  - Channel may flip bits in packet
  - Checksum to detect bit errors

- **The question**: How to recover from errors?

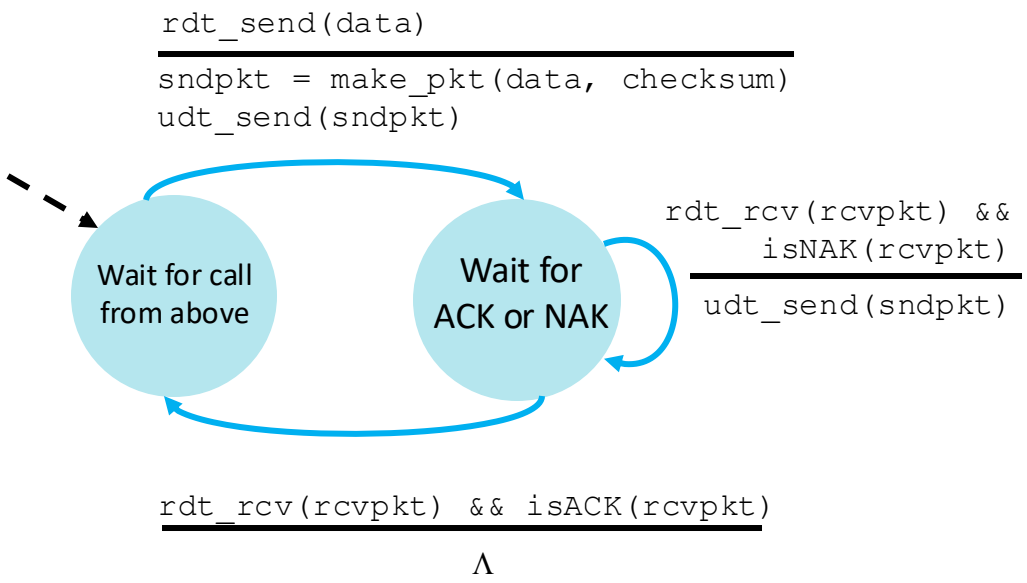
  - **Tip:** How do humans recover from error during conversation?
    - Error Detection
    - Feedback

RDT2.0: Reliable Data Transfer over Channel with Bit Errors

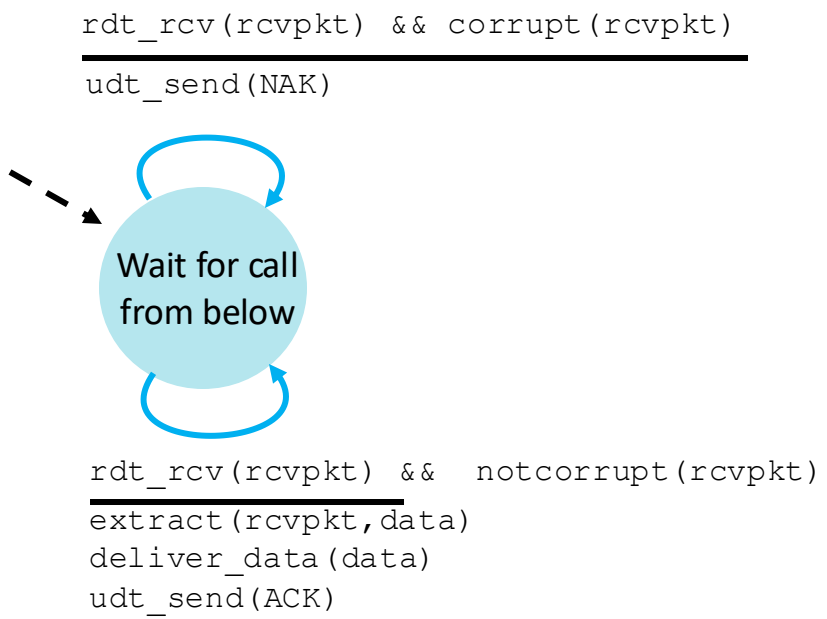
- Reliable Data Transfer Mechanisms
  - Detect
    - **Checksum:** Bit errors
  - Feedback
    - **Acknowledgements (ACKs):** Receiver explicitly tells sender that packet received OK
    - **Negative acknowledgements (NAKs):** Receiver explicitly tells sender that packet had errors
  - Correction
    - Sender retransmits packet on receipt of NAK

RDT2.0: FSM Specification

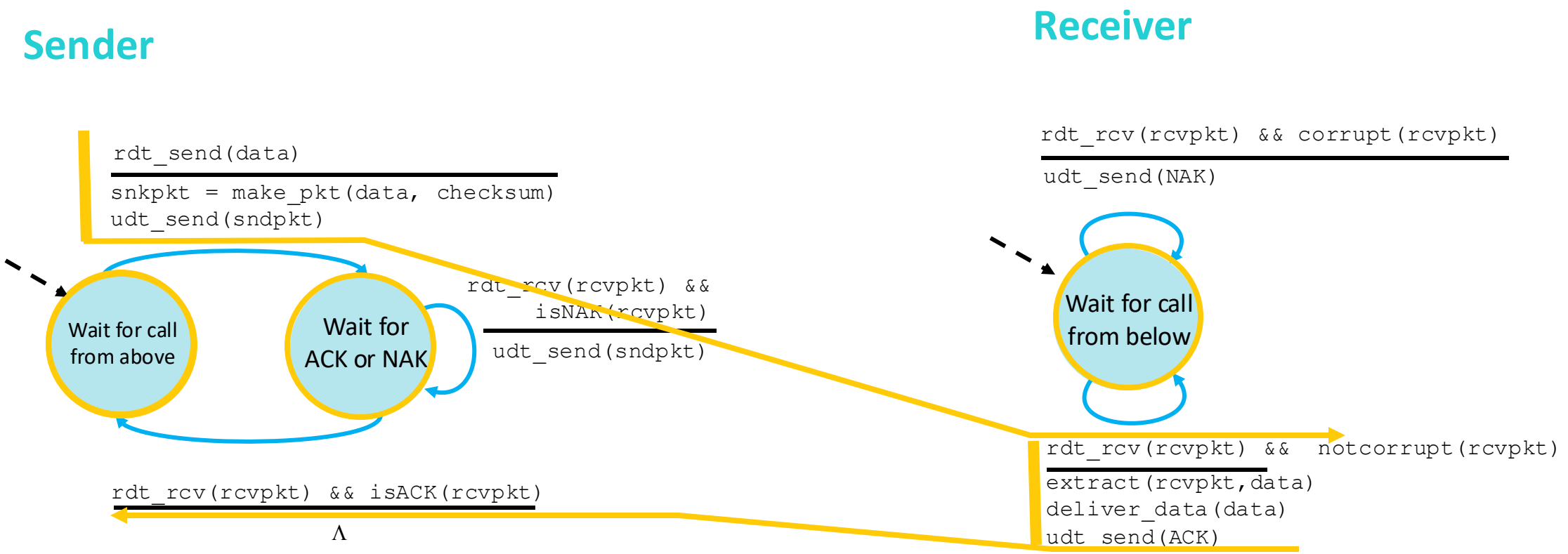
Sender



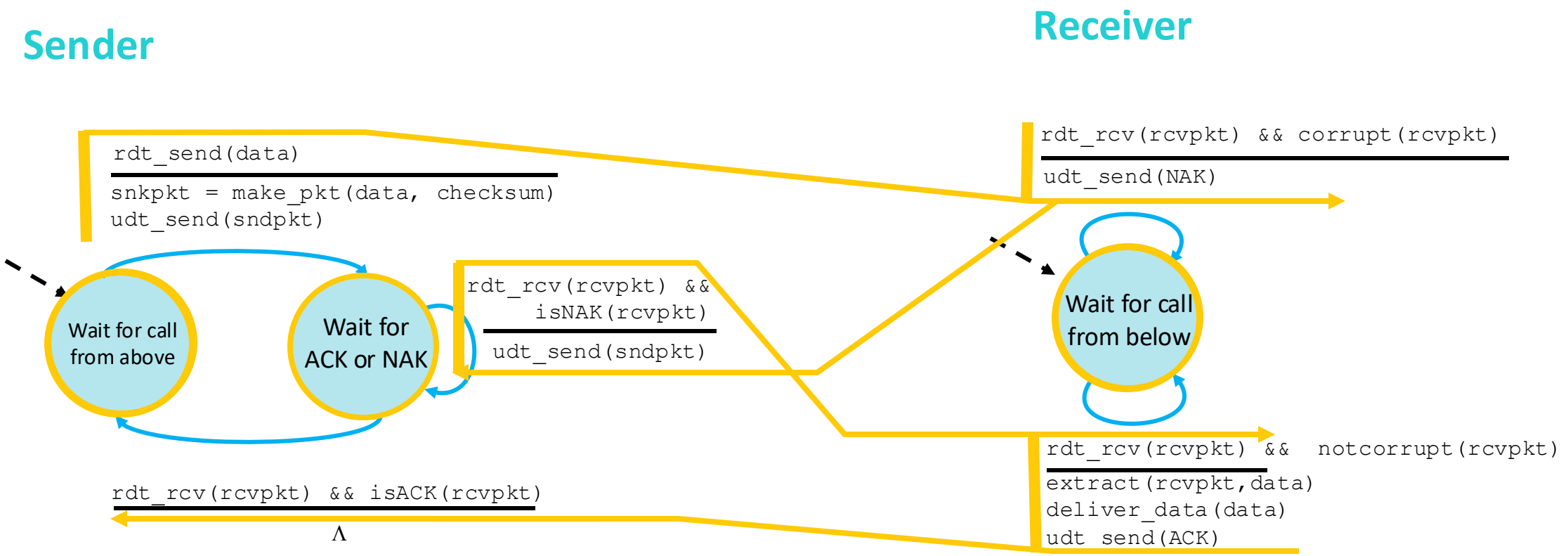
Receiver



RDT2.0: Operation with No Errors



RDT2.0: Error Scenario



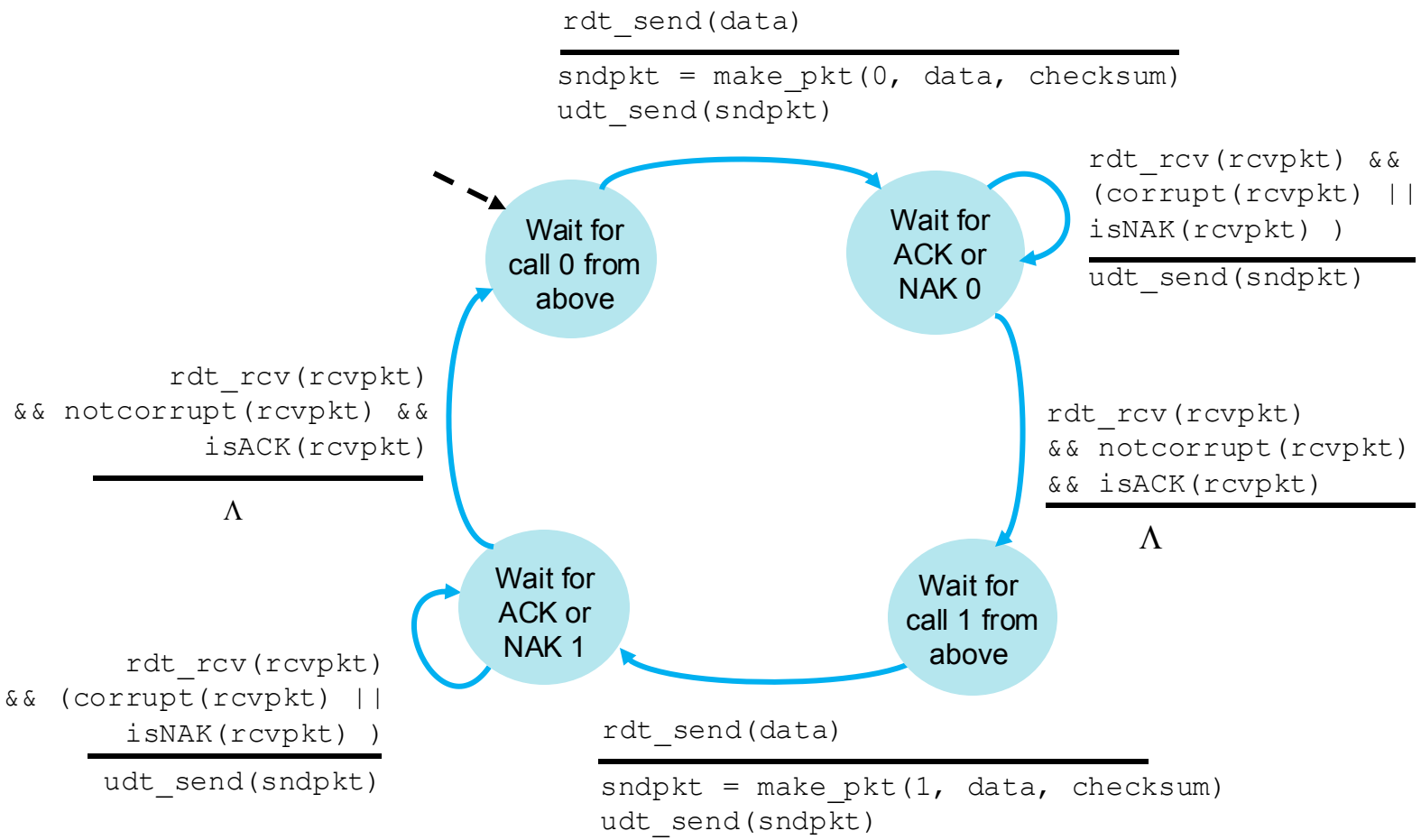
RDT2.0: Fatal Flaw!

- **What happens if ACK/NAK corrupted?**
  - Sender does not know what happened at receiver!
  - Can not just retransmit: Possible duplicate

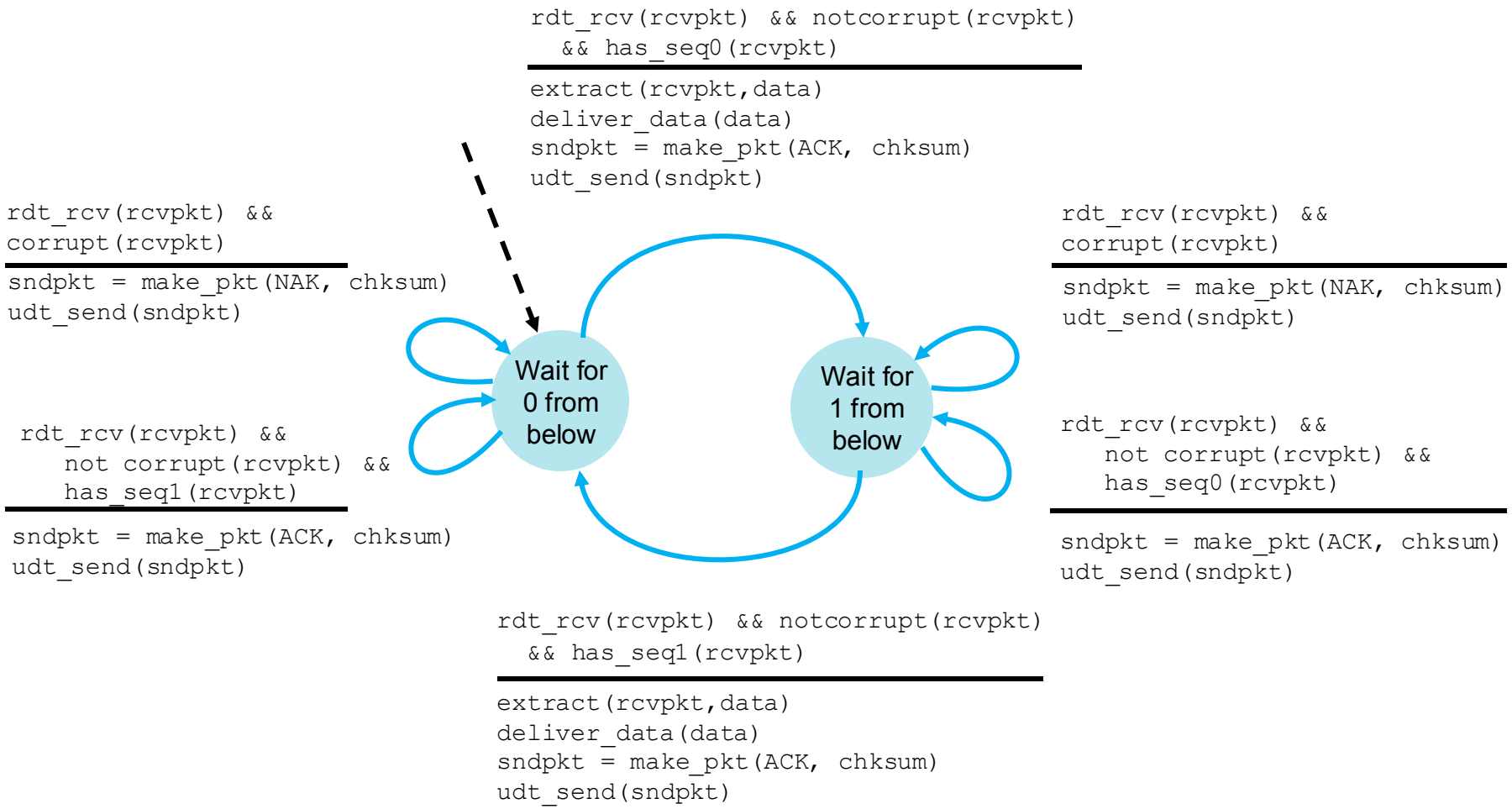
  - **Stop and wait**
    - Sender sends one packet, then waits for receiver response

  - **Handling duplicates**
    - Sender retransmits current packet if ACK/NAK corrupted
    - Sender adds **sequence number** to each packet
    - Receiver discards (does not deliver up) duplicate packet

RDT2.1: Sender (Garbled ACK/NAKs)



RDT2.1: Receiver (Garbled ACK/NAKs)



RDT2.1: Discussions

- **Sender**
  - Sequence number added to packet
  - Two sequence numbers (0,1) will suffice. **Why?**
  - Must check if received ACK/NAK corrupted
  - Twice as many states
    - State must **remember** whether **expected** packet should have sequence number of 0 or 1
- **Receiver**
  - Must check if received packet is duplicate
    - State indicates whether 0 or 1 is expected packet sequence number
    - Note: Receiver cannot know if its last ACK/NAK received OK at sender

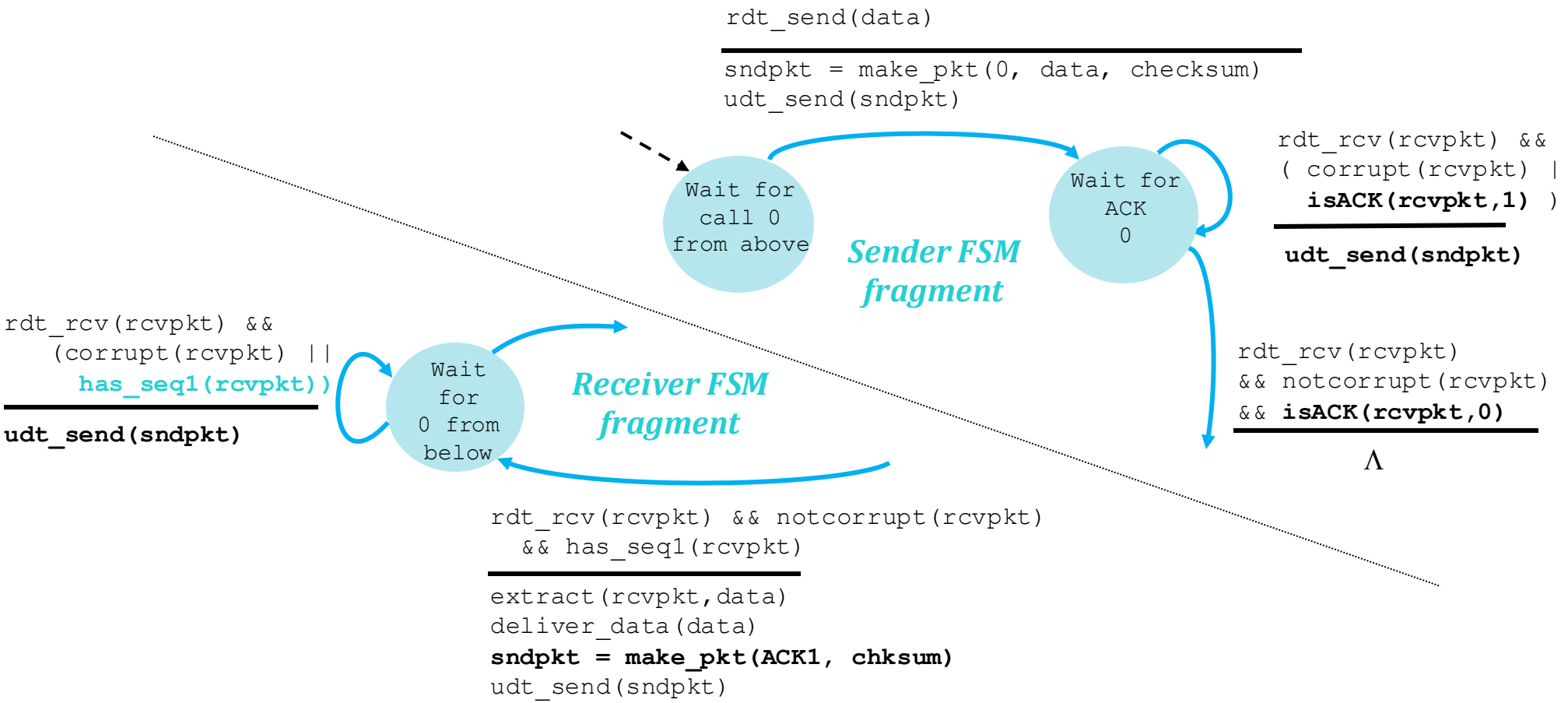
RDT2.2: A NAK-Free Protocol

- Same functionality as RDT2.1 using **ACKs only**

- Instead of NAK, receiver sends ACK for last packet received OK
  - Receiver must **explicitly** include sequence number of packet being ACKed

- Duplicate ACK at sender results in same action as NAK: **Retransmit Current Packet**

RDT2.2: Sender & Receiver



RDT3.0: Channels with Errors & Loss

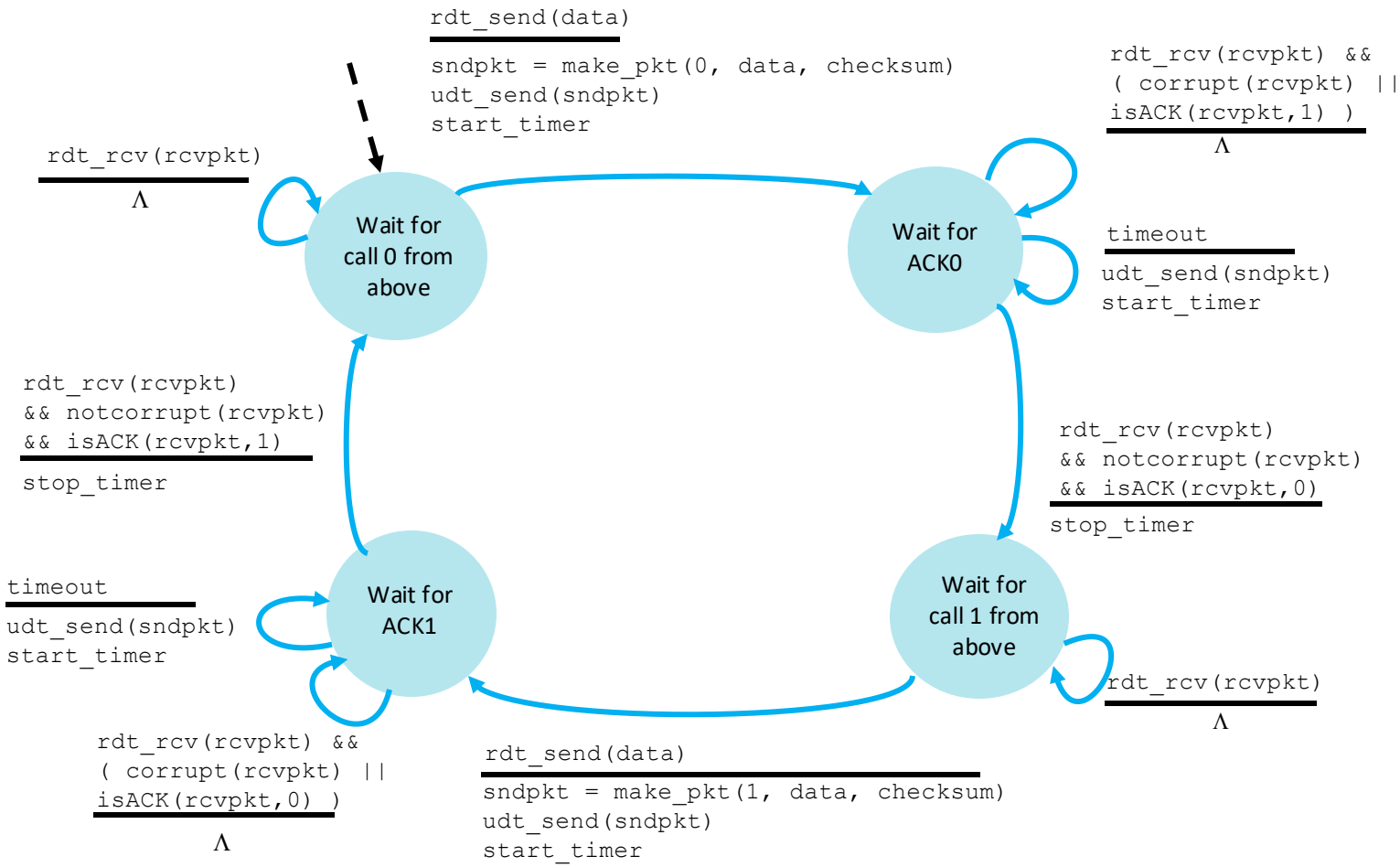
- **New assumption:** Underlying channel can also lose packets (data, ACKs)
  - Checksum, sequence number, ACKs, retransmissions will be of help … but not enough

- **Approach:** Sender waits **reasonable** amount of time for ACK
  - Retransmits if no ACK received in this time

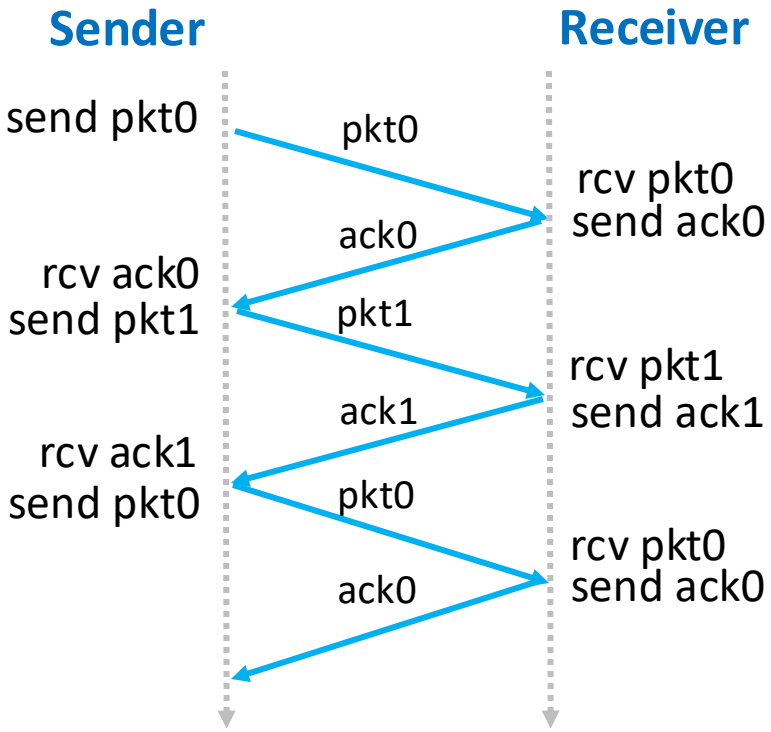
  - If packet (or ACK) just delayed (not lost):
    - Retransmission will be duplicate, but sequence numbers already handle this
    - Receiver must specify sequence number of packets being ACKed

  - Requires countdown timer

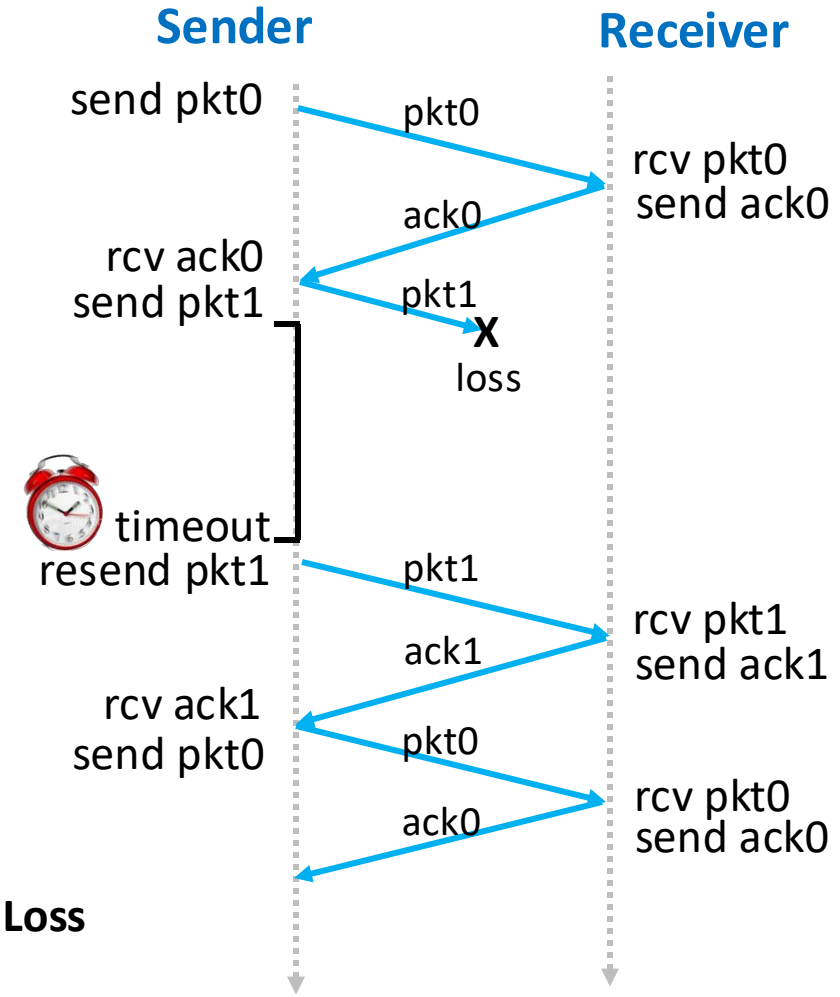
RDT3.0 Sender



RDT3.0 In Action

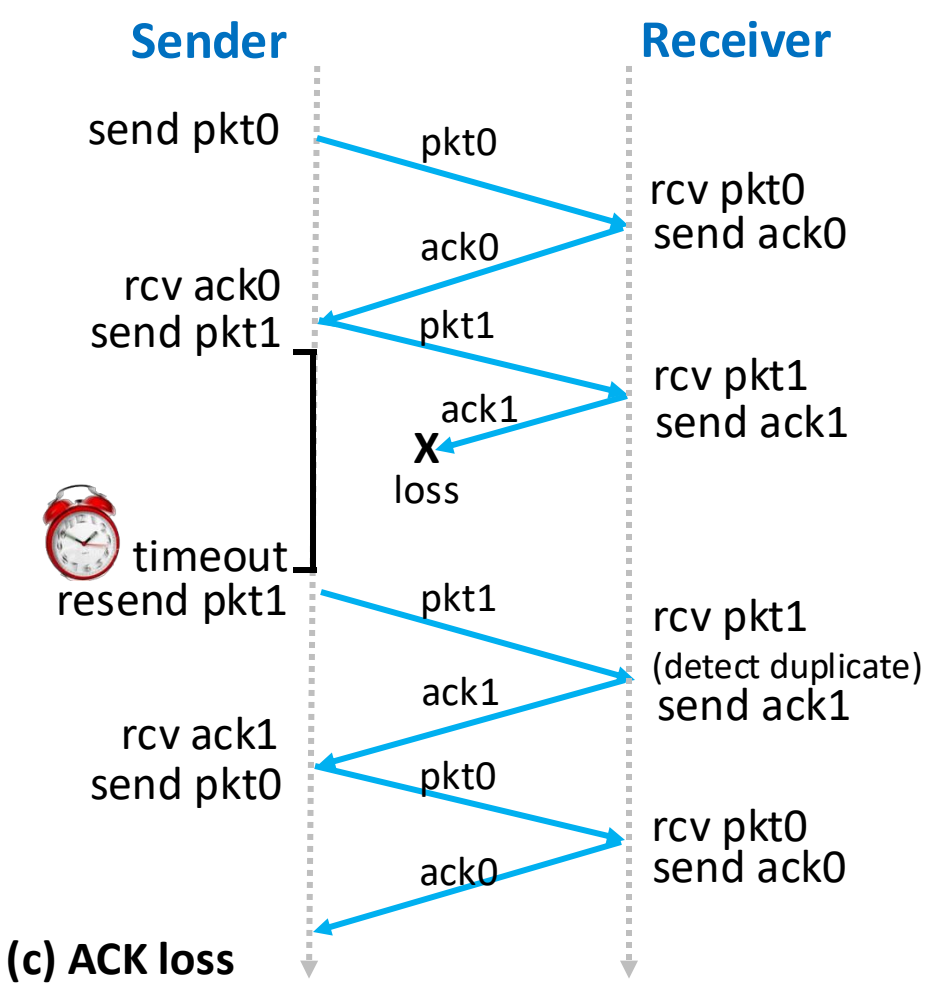


(a) No Loss

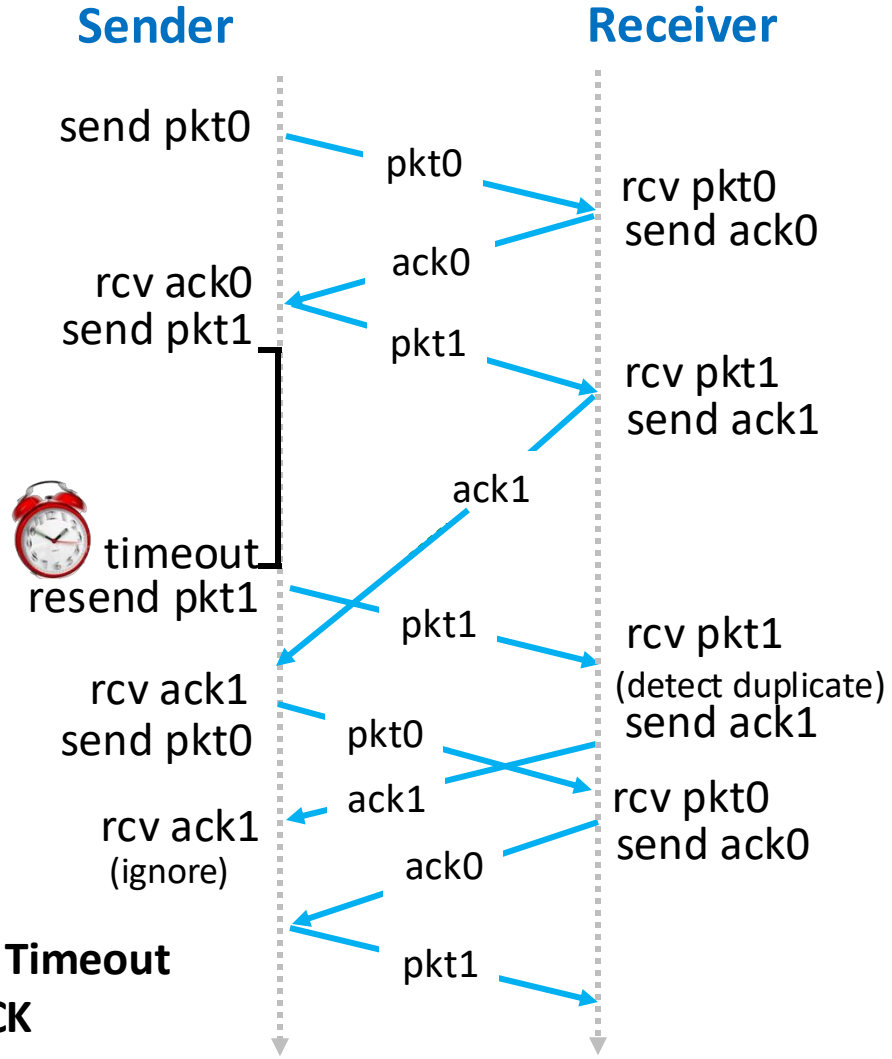


(b) Packet Loss

RDT3.0 In Action



(c) ACK loss



(d) Premature Timeout
Delayed ACK

RDT3.0 Performance

- RDT3.0 is correct but low performance
- **Example:** 1 Gbps link, 15ms propagation delay, 8000 bit packet:

$$d_{trans} = \frac{L}{R} = \frac{8000 \text{ bits / packet}}{10^9 \text{ bits / sec}} = 8 \text{ microseconds}$$

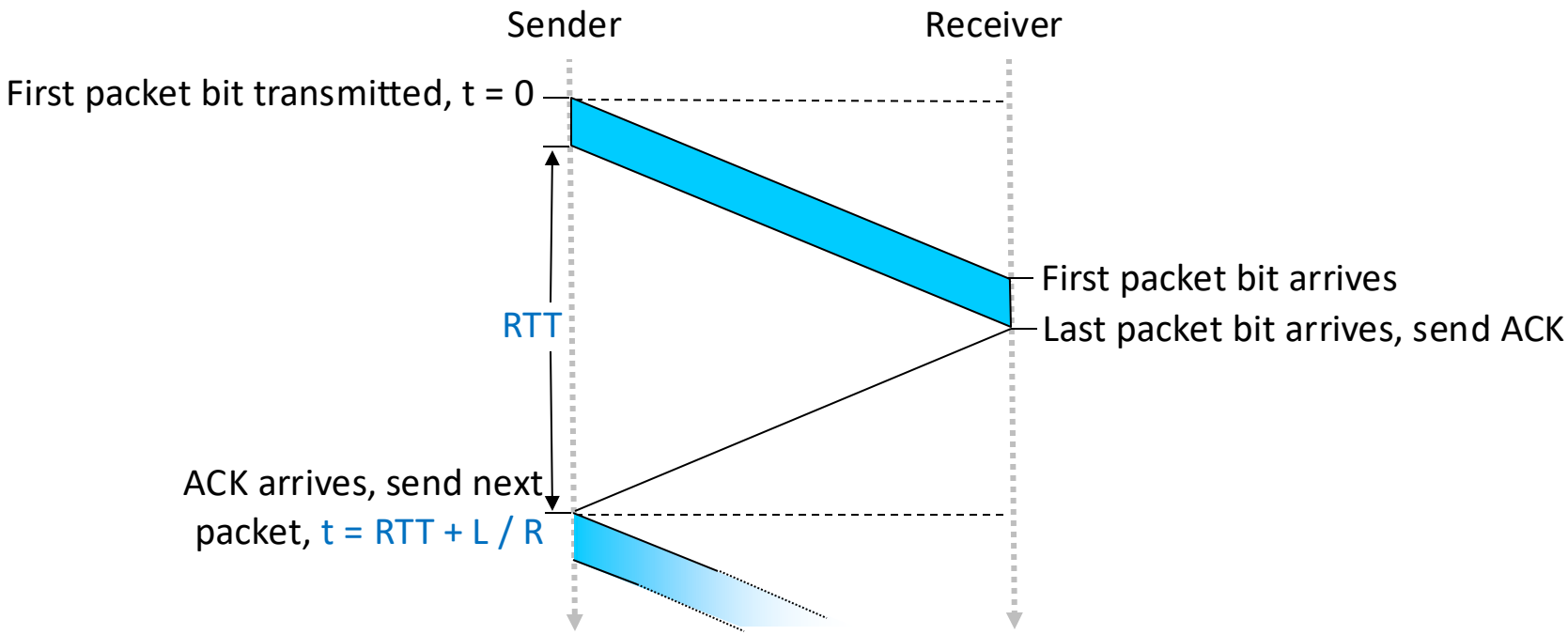
RDT3.0 Performance

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

$$U_{sender} = \frac{\frac{L}{R}}{RTT + \frac{L}{R}} = \frac{.008}{30.008} = 0.00027$$

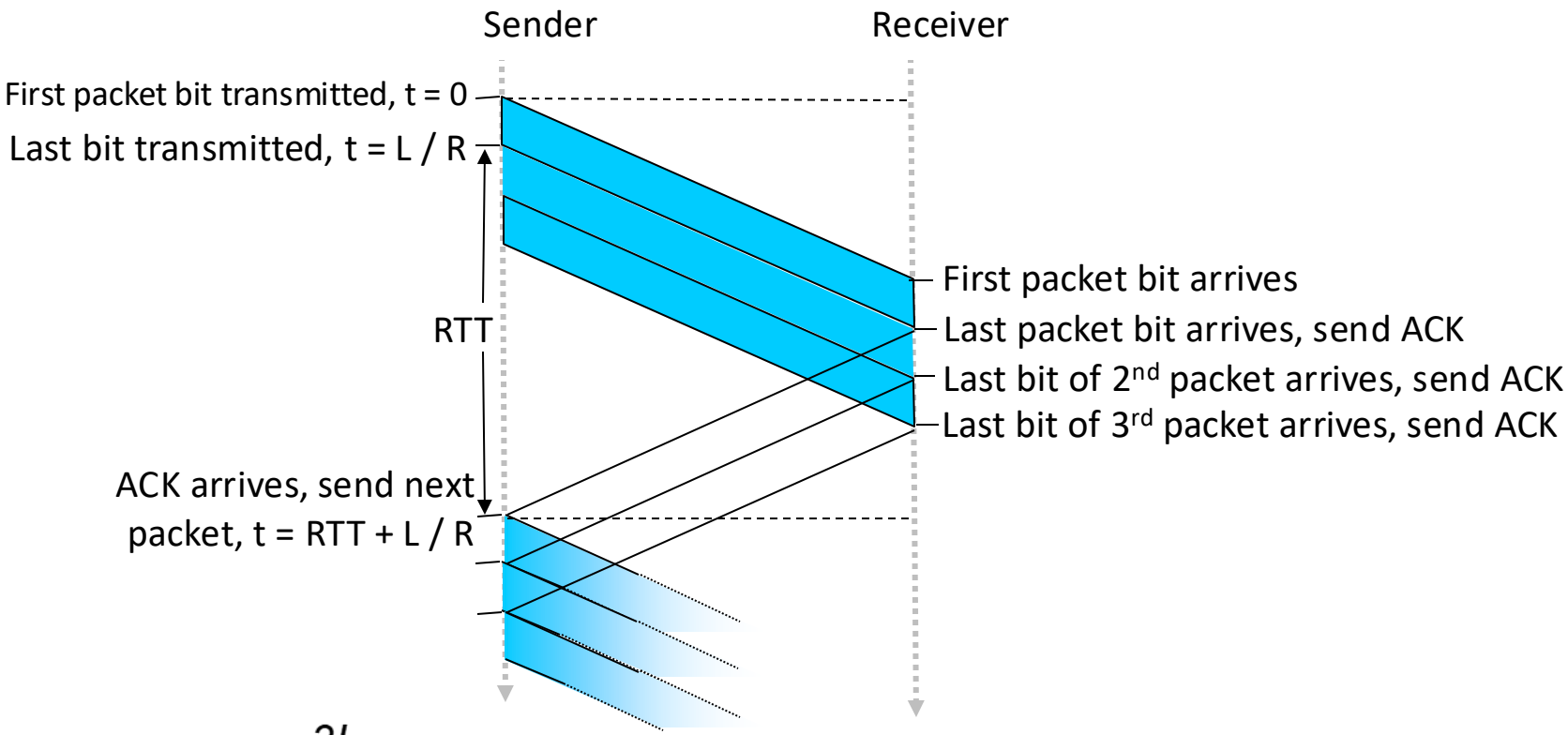
- If RTT = 30 msec, 1KB packet every 30 msec: 33kB/sec throughput over 1 Gbps link
- Network protocol limits use of physical resources!

RDT3.0 Stop and Wait Operation



$$U_{sender} = \frac{\frac{L}{R}}{RTT + \frac{L}{R}} = \frac{.008}{30.008} = 0.00027$$

Pipelining: Increased Utilization

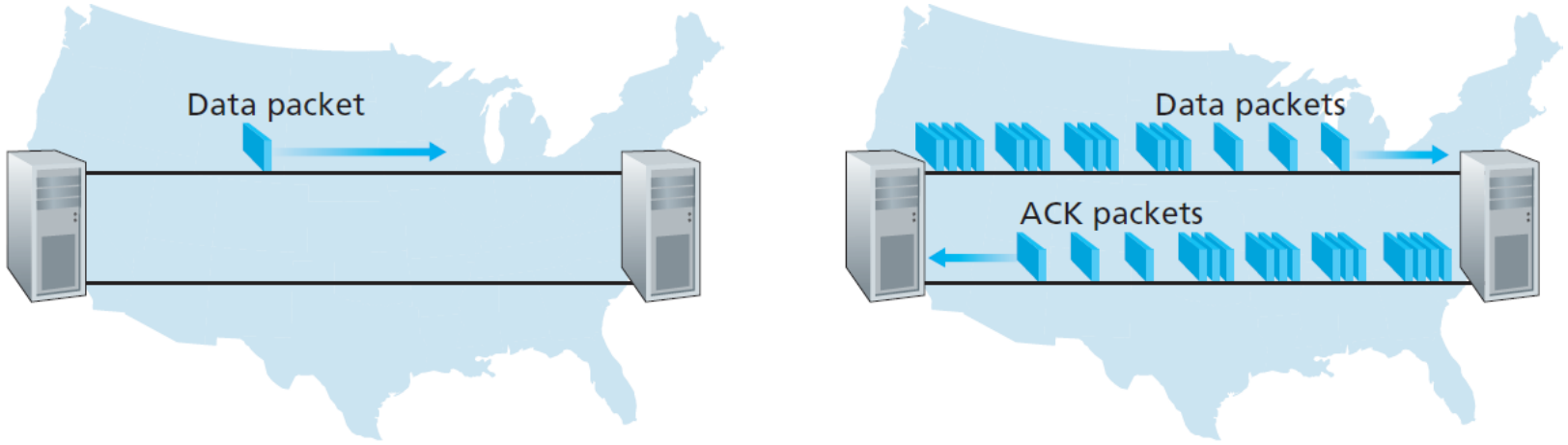


$$U_{sender} = \frac{\frac{3L}{R}}{RTT + \frac{L}{R}} = \frac{.0024}{30.008} = 0.00081$$

3-packet pipelining increases utilization by a factor of 3!

Pipelined 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



a. A stop-and-wait protocol in operation

b. A pipelined protocol in operation

- Two generic forms of pipelined protocols: **Go-Back-N, Selective Repeat**

Pipelined Protocols: Overview

Go-back-N

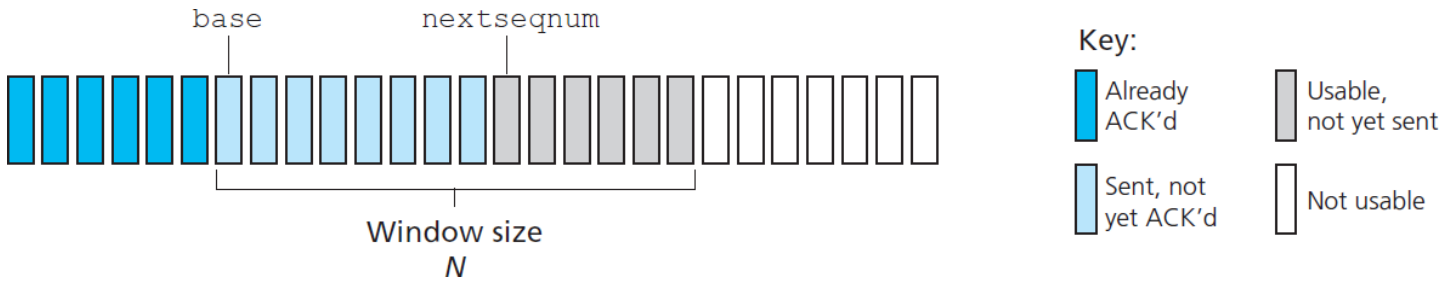
- Sender can have up to N un-acked packets in pipeline
- Receiver only sends **cumulative ack**
  - Does not ack packet if there's a gap
- Sender has timer for oldest un-acked packet
  - When timer expires, retransmit **all** un-acked packets

Selective Repeat

- Sender can have up to N un-acked packets in pipeline
- Receiver sends **individual ack** for each packet
- Sender maintains timer for each un-acked packet
  - When timer expires, retransmit only that un-acked packet

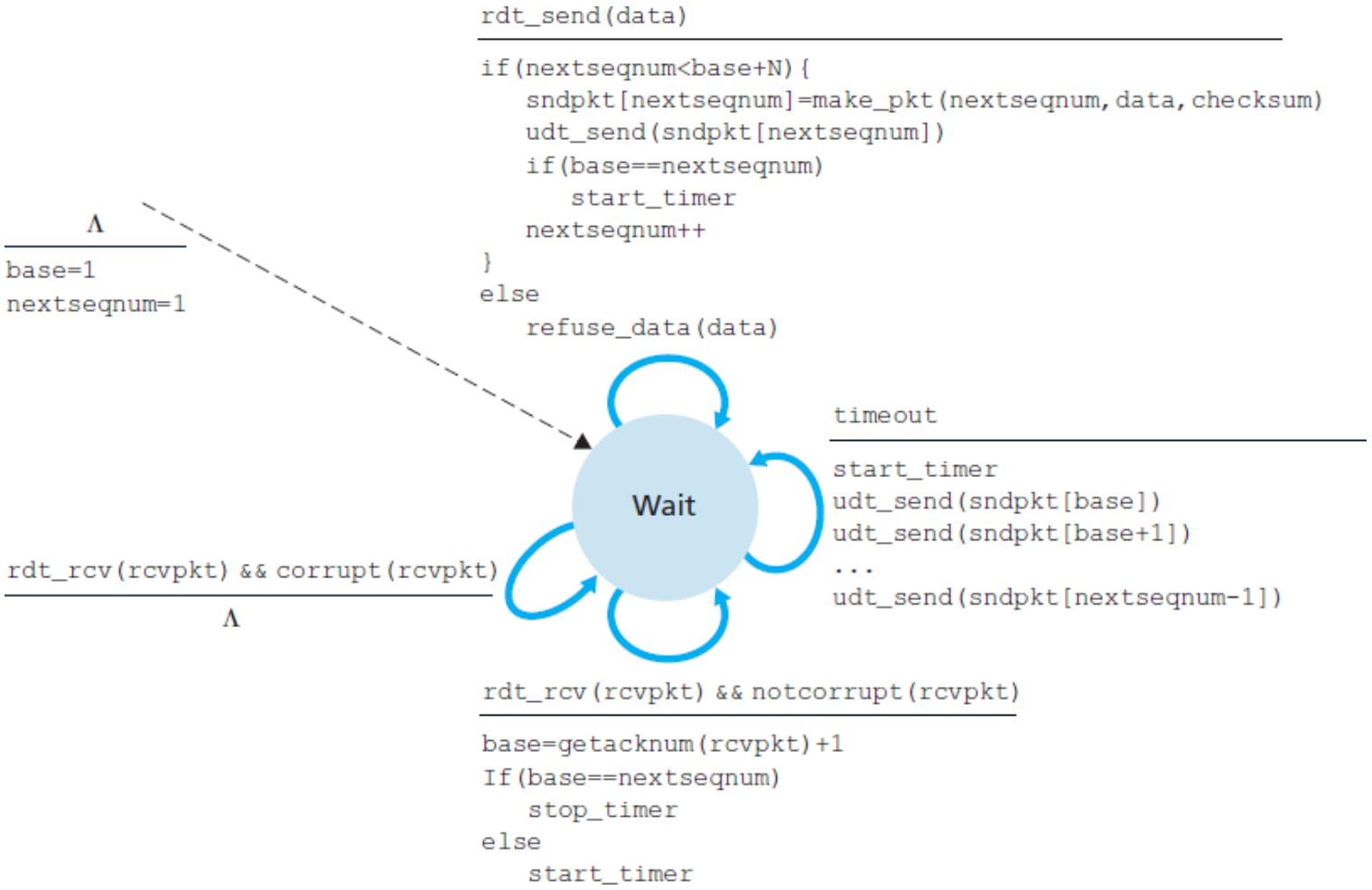
Go-Back N: Sender

- k-bit sequence number in packet header
- **Window** of up to N, consecutive un-acked packets allowed

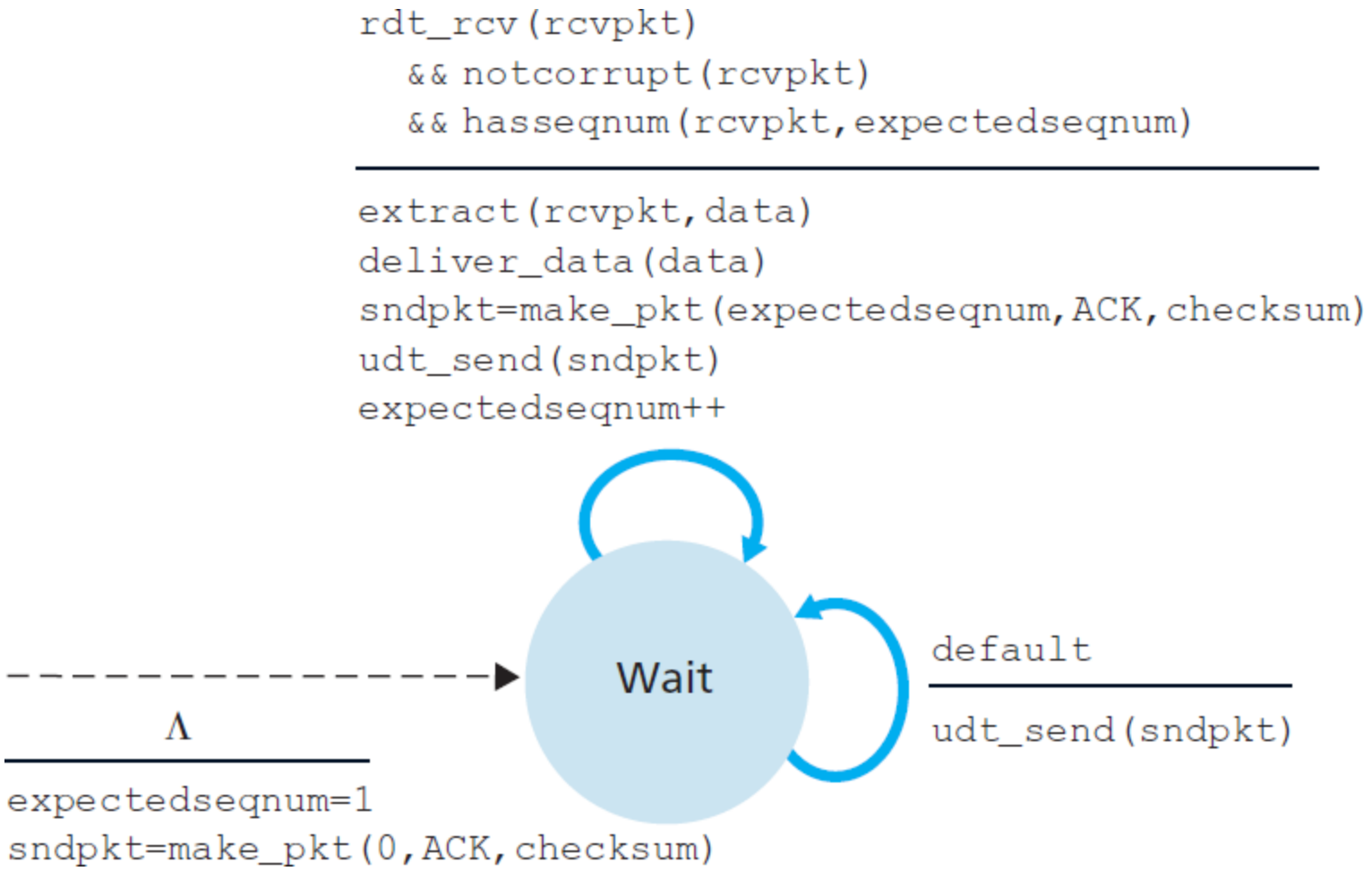


- ACK(n): ACKs all packets up to, including sequence number n – **Cumulative ACK**
  - May receive duplicate ACKs (see receiver)
- Timer for oldest in-flight packet
- **Timeout(n):** Retransmit packet **n** and all higher sequence number packets in window

GBN: Sender Extended FSM



GBN: Receiver Extended FSM

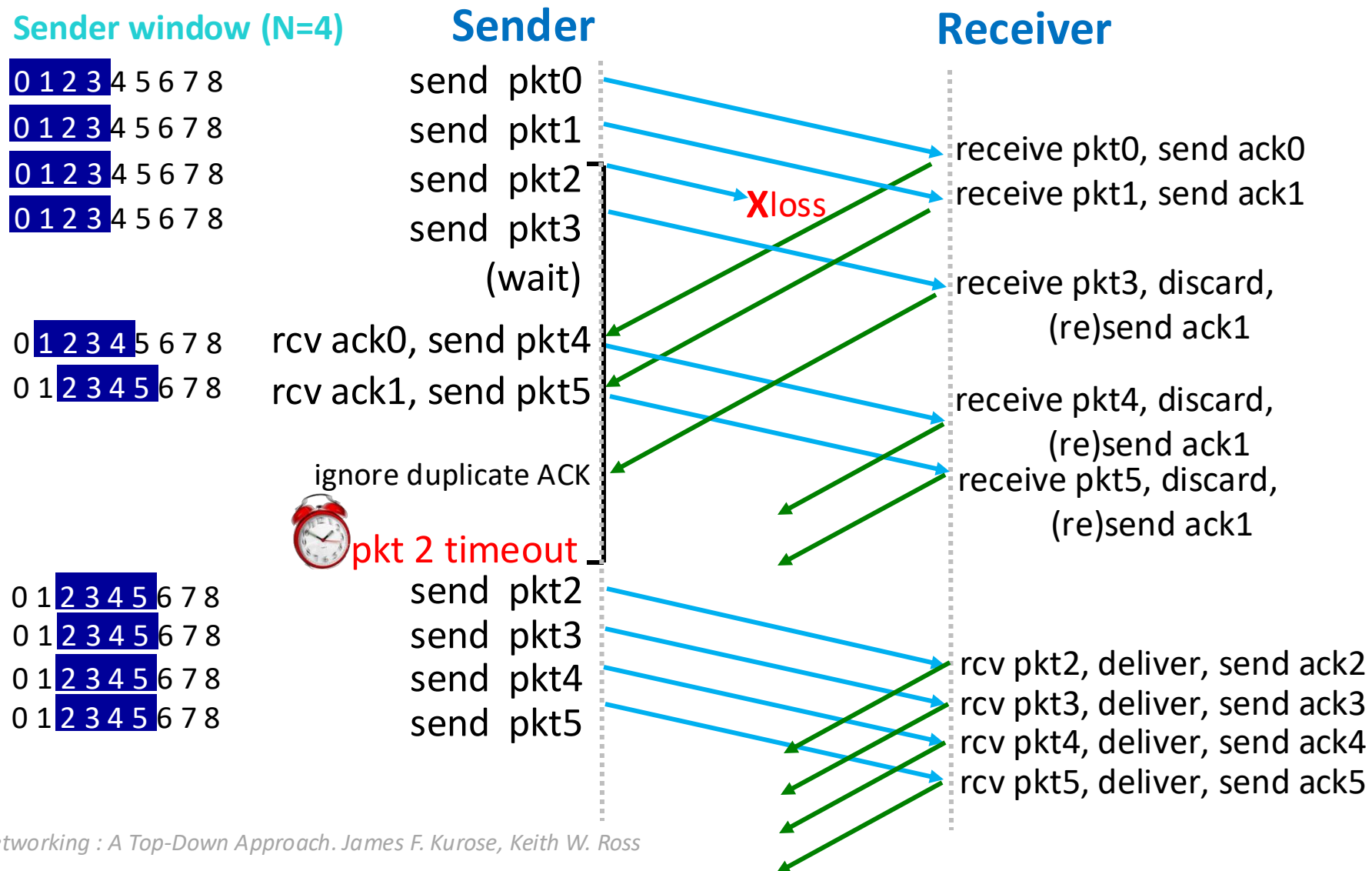


GBN: Receiver Extended FSM

- ACK-only: Always send ACK for correctly-received packet with highest **in-order** sequence number
  - May generate duplicate ACKs
  - Need only remember `expectedseqnum`

- Out-of-order packet:
  - Discard (don't buffer): **No receiver buffering!**
  - Re-ACK packet with highest in-order sequence number

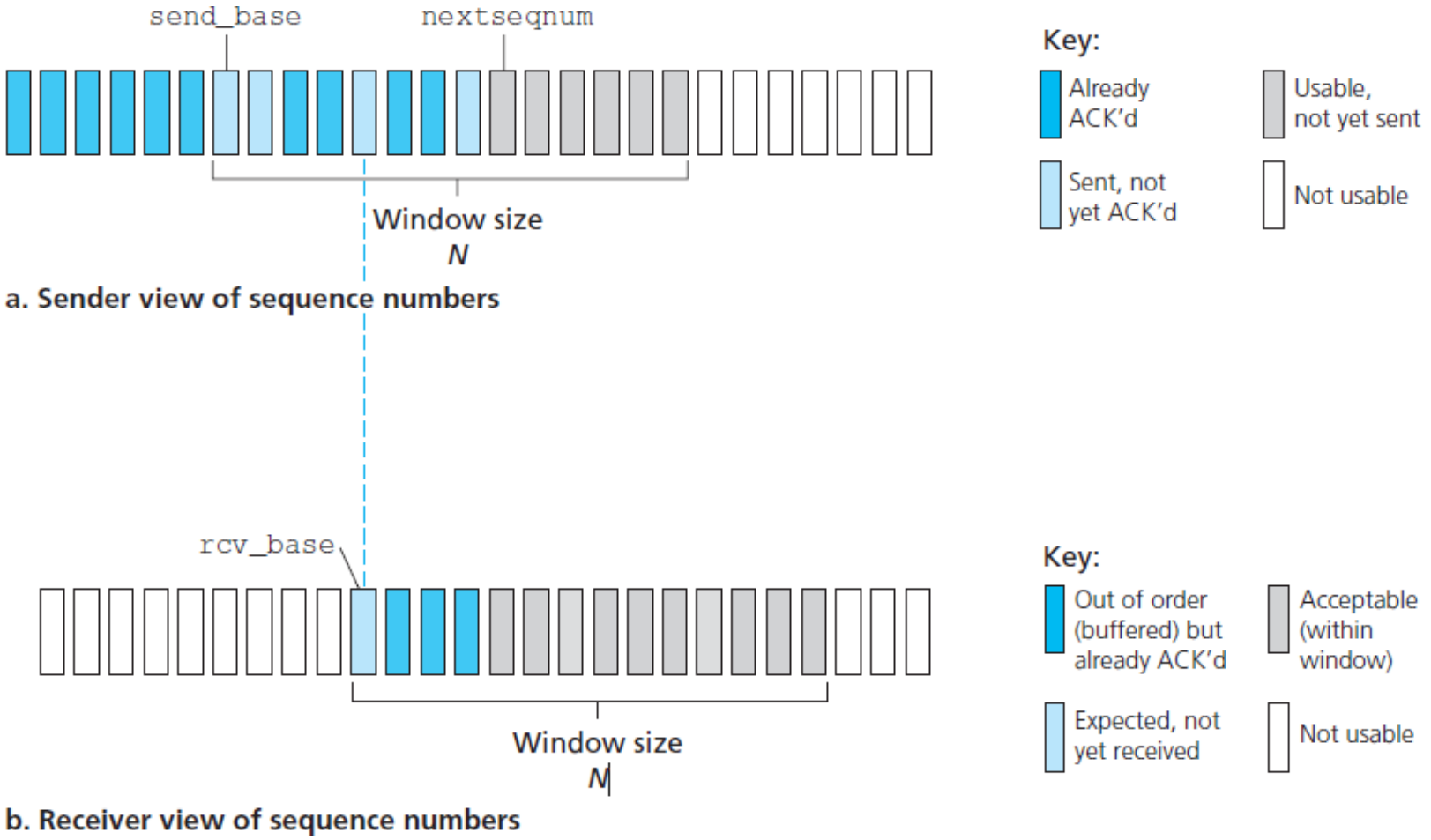
GBN In Action



Selective Repeat

- Receiver **individually** acknowledges all correctly received packets
  - Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK not received
  - Sender timer for each un-ACKed packet
- Sender window
  - **N** consecutive sequence numbers
  - Limits sequence numbers of sent, unACKed packets

Selective Repeat: Sender, Receiver Windows



Selective Repeat: Sender

Data from above

- If next available sequence number in window, send packet

Timeout(n)

- Resend packet **n**, restart timer

ACK(n) in **`[sendbase, sendbase+N]`:**

- Mark packet **n** as received
- If **n** smallest un-ACKed packet, advance window base to next un-ACKed sequence number

Selective Repeat: Receiver

Packet n in `[rcvbase, rcvbase+N-1]`

- Send ACK(n)
- Out-of-order: Buffer
- In-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

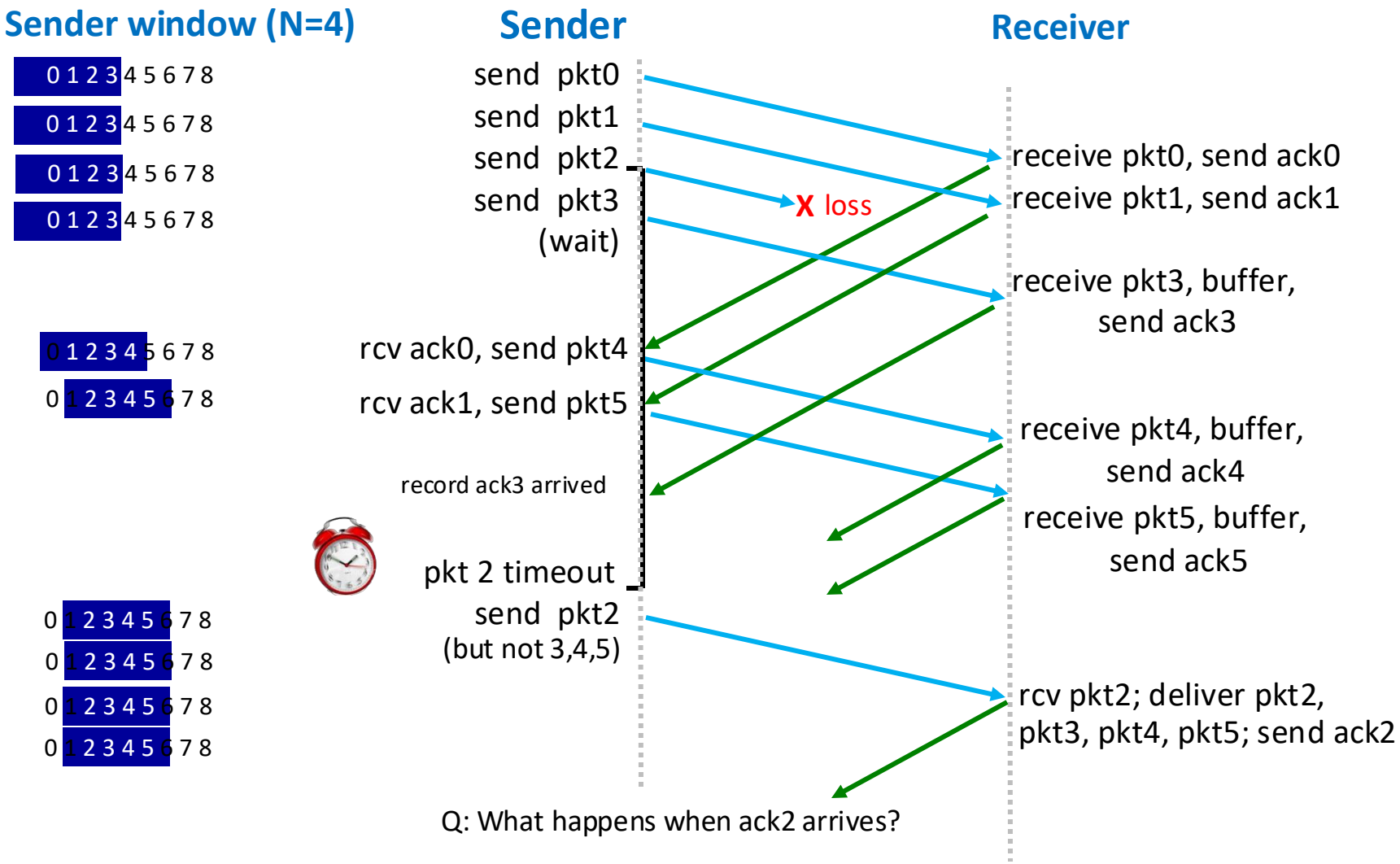
Packet n in `[rcvbase-N,rcvbase-1]`

- ACK(n)

Otherwise

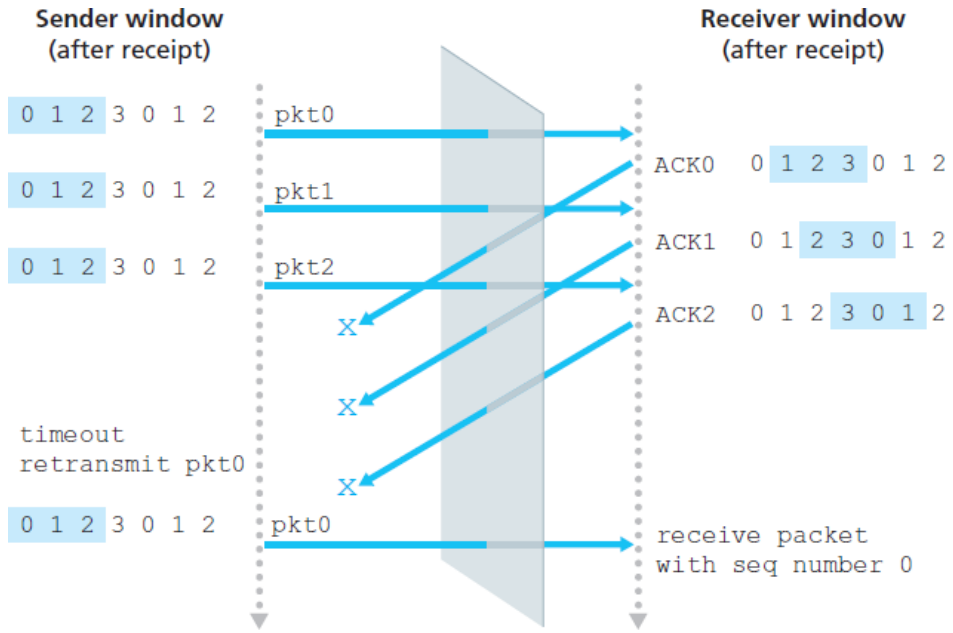
- Ignore

Selective Repeat In Action



Selective Repeat: Dilemma

- Example:
  - Sequence numbers: 0, 1, 2, 3
  - Window size=3
  - Receiver sees no difference in two scenarios!
  - Duplicate data accepted as new in (b)

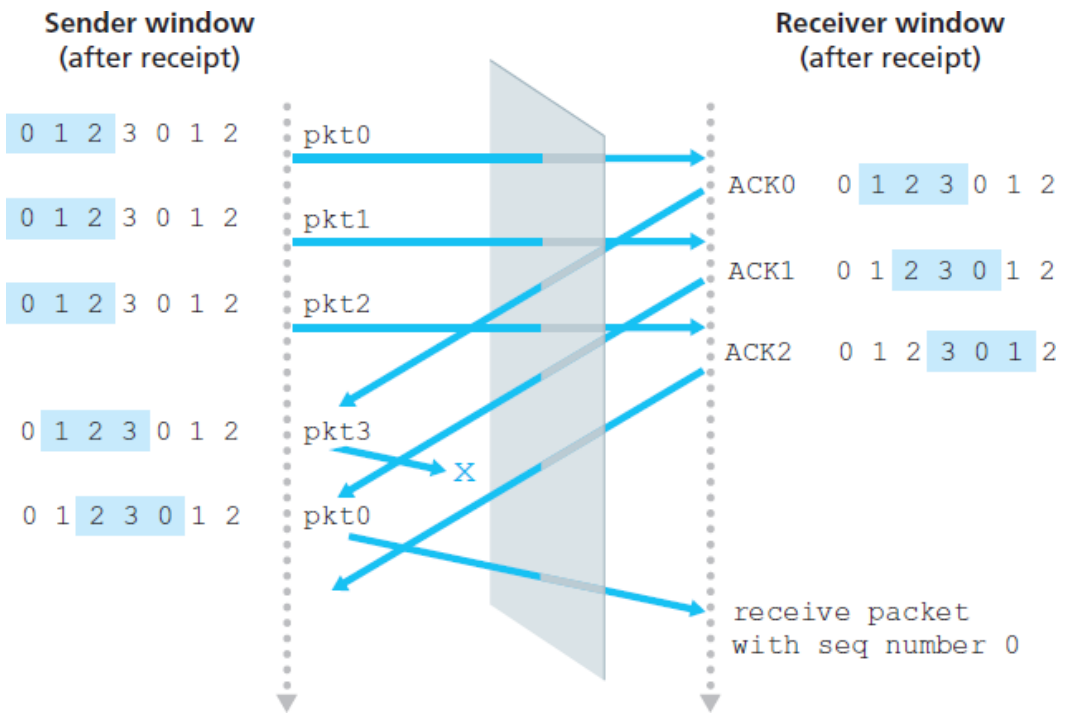


(a)

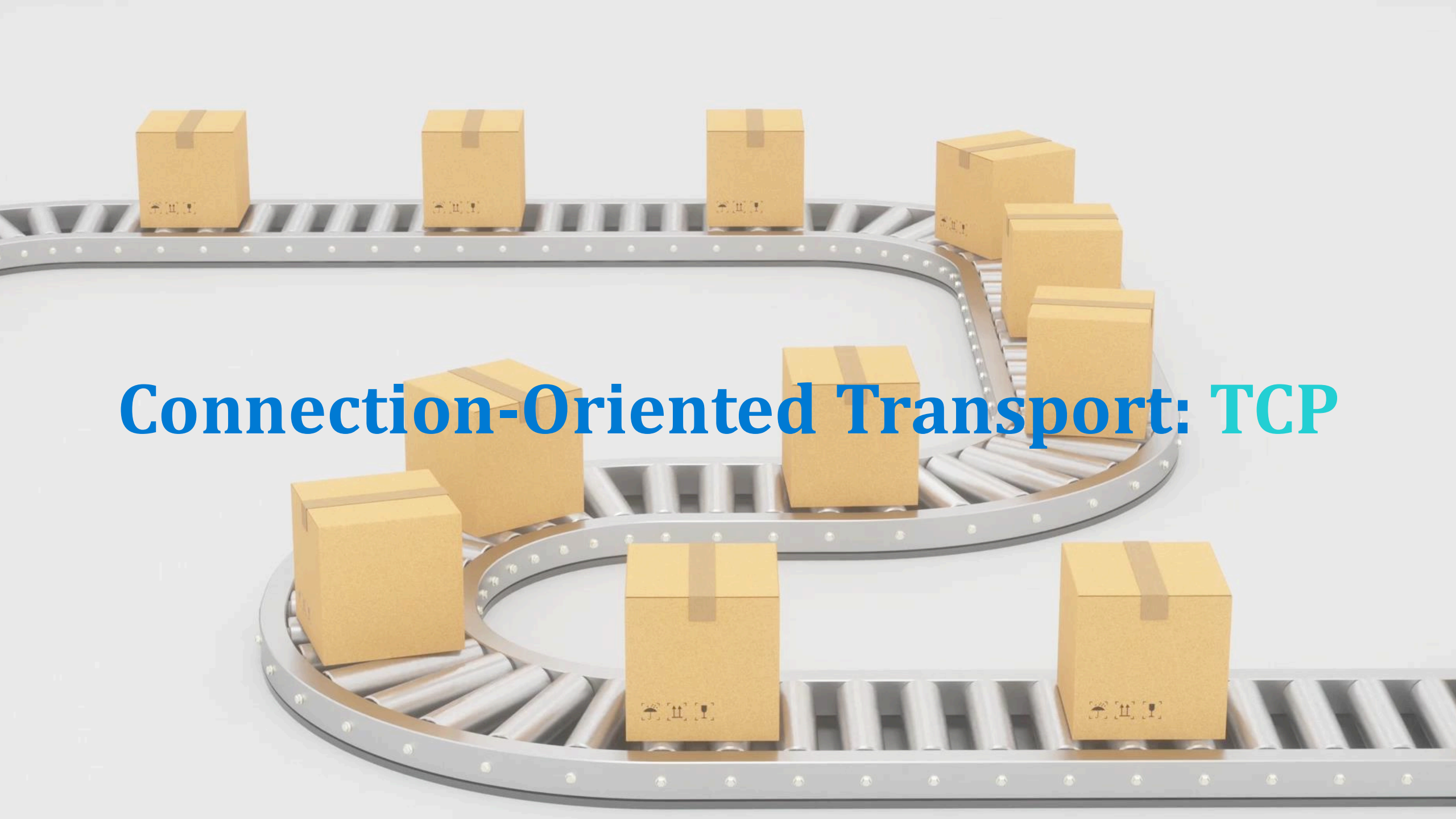
- Receiver can not see sender side.
- Receiver behavior identical in both cases! **Something's (very) wrong!**

Selective Repeat: Dilemma

Q: What relationship between sequence number size and window size to avoid problem in (b)?



(b)

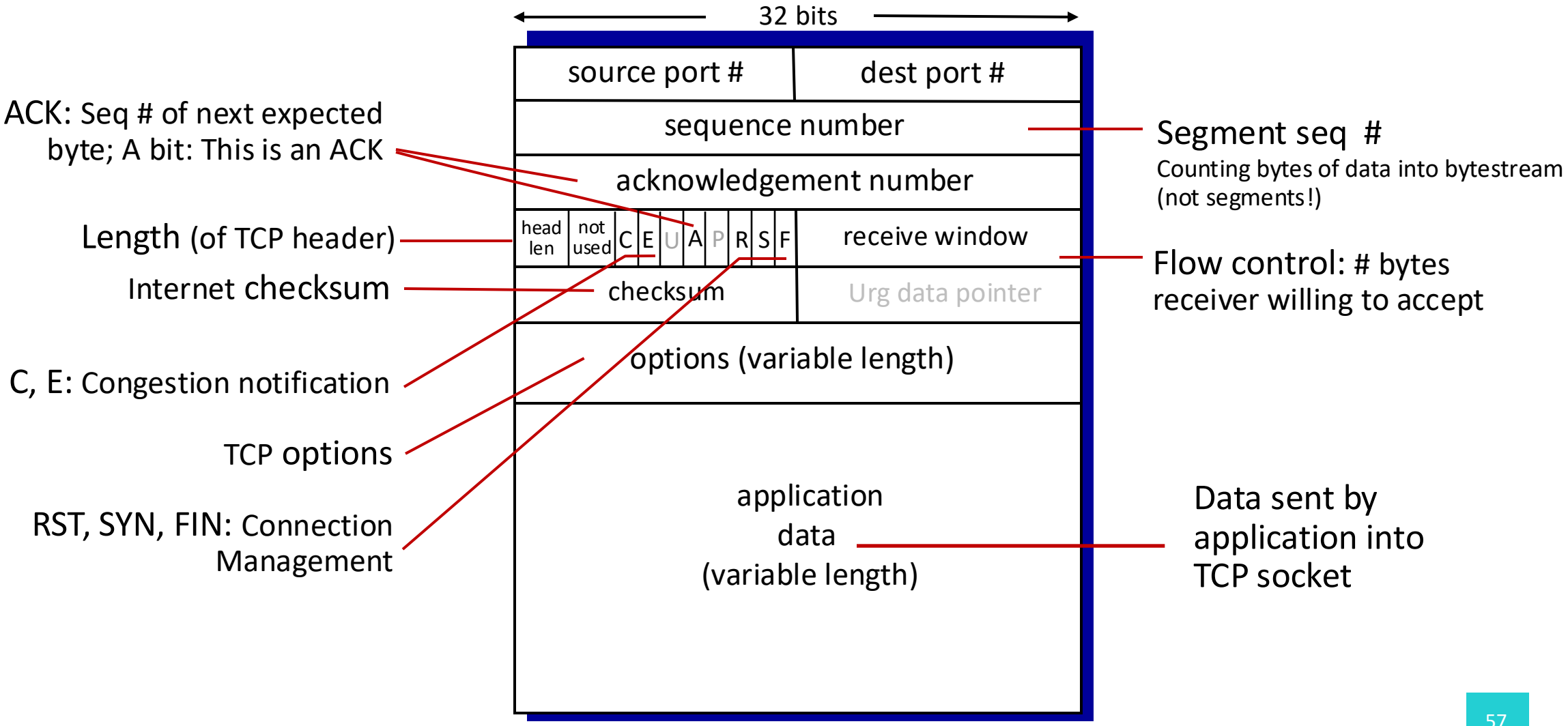


Connection-Oriented Transport: TCP

TCP: Overview (RFCs 793, 1122, 1323, 2018, 2581)

- **Point-to-point:** One sender, one receiver
- **Connection-oriented:** Handshaking (exchange of control messages) initiates sender, receiver state before data exchange
- **Reliable, in-order byte stream:** No message boundaries
- **Pipelined:** TCP congestion and flow control set window size
- **Full duplex data:**
  - Bi-directional data flow in same connection
  - MSS: Maximum Segment Size
- **Flow controlled:** Sender will not overwhelm receiver

TCP Segment Structure



TCP Sequence Numbers and ACKs

Sequence numbers:

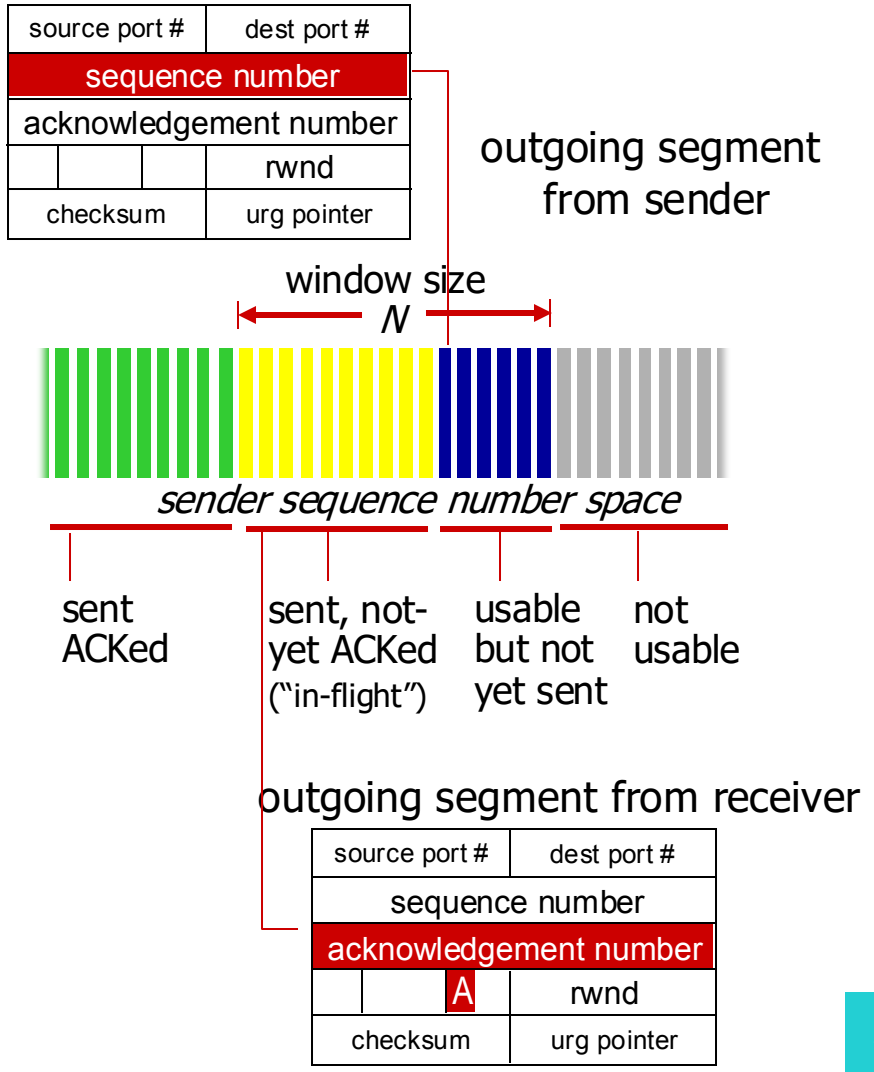
- Byte stream **number** of first byte in segment's data

Acknowledgements:

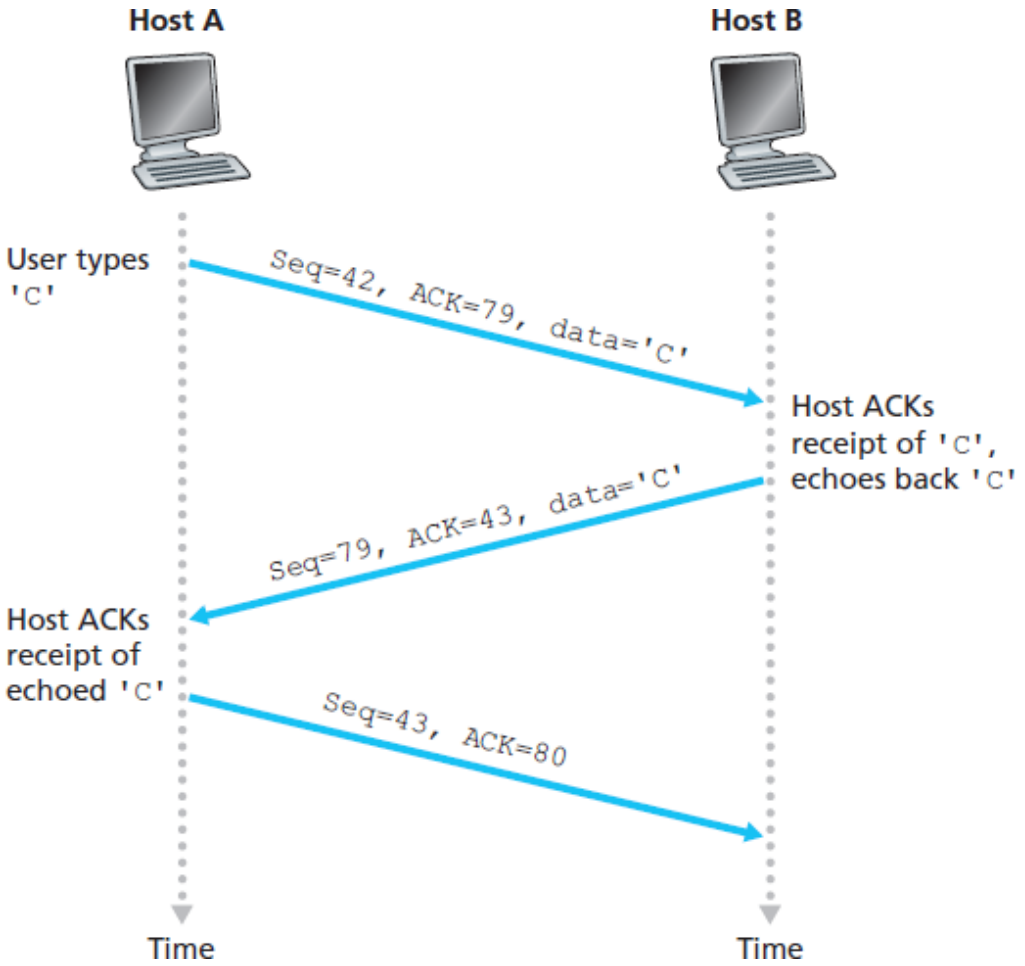
- Sequence number of next byte expected from other side
- Cumulative ACK

Q: How receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



TCP Sequence Numbers and ACKs



TCP RTT & Timeout

Q: How to set TCP timeout value?

- Longer than RTT
  - But RTT varies
- **Too short***:* Premature timeout, unnecessary retransmissions
- **Too long***:* Slow reaction to segment loss

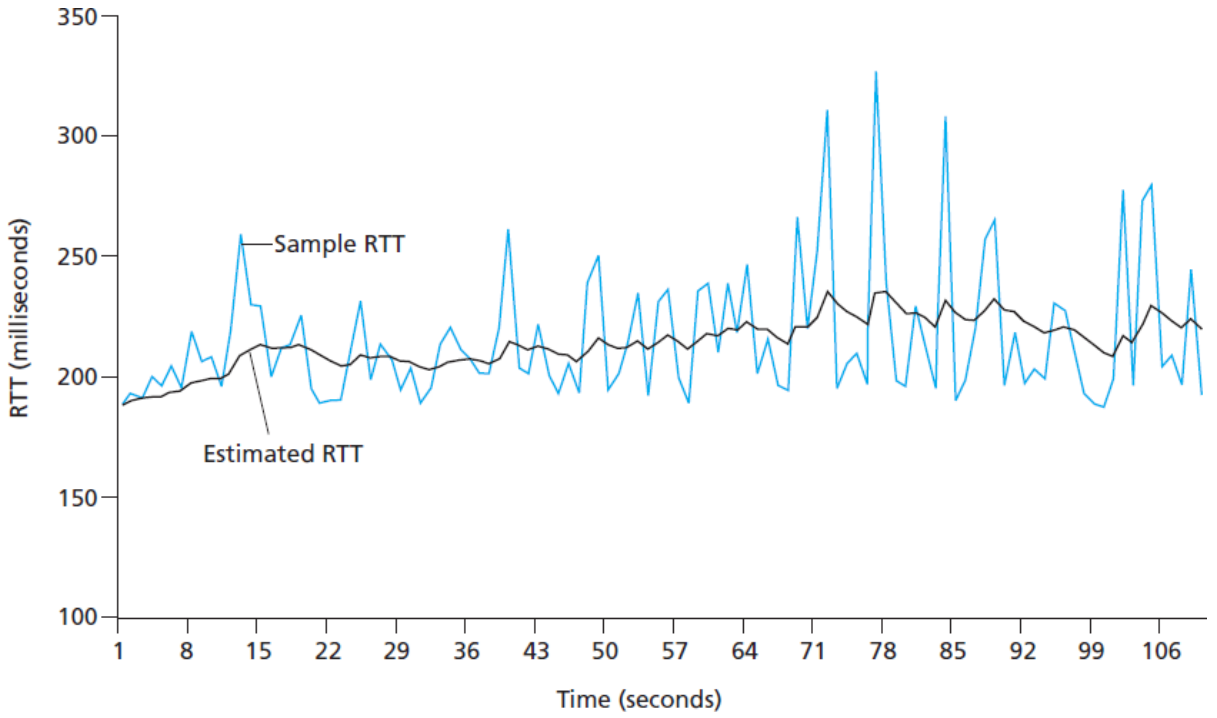
Q: How to estimate RTT?

- **`SampleRTT:`** Measured time from segment transmission until ACK receipt
  - Ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT **smoother**
  - Average several **recent** measurements, not just current **`SampleRTT`**

TCP RTT & Timeout

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

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: $\alpha = 0.125$



TCP RTT & Timeout

- **Timeout interval: `EstimatedRTT` plus safety margin**

  large variation in `EstimatedRTT` → larger safety margin.

- Estimate SampleRTT deviation from `EstimatedRTT` :

```
DevRTT = (1-β)*DevRTT +
         β*|SampleRTT -EstimatedRTT|

(typically, β = 0.25)
```

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

estimated RTT

"safety margin"

TCP Reliable Data Transfer

- TCP creates RDT service on top of IP's unreliable service
  - Pipelined segments
  - Cumulative acks
  - Single retransmission timer

- Retransmissions triggered by:
  - Timeout events
  - Duplicate acks

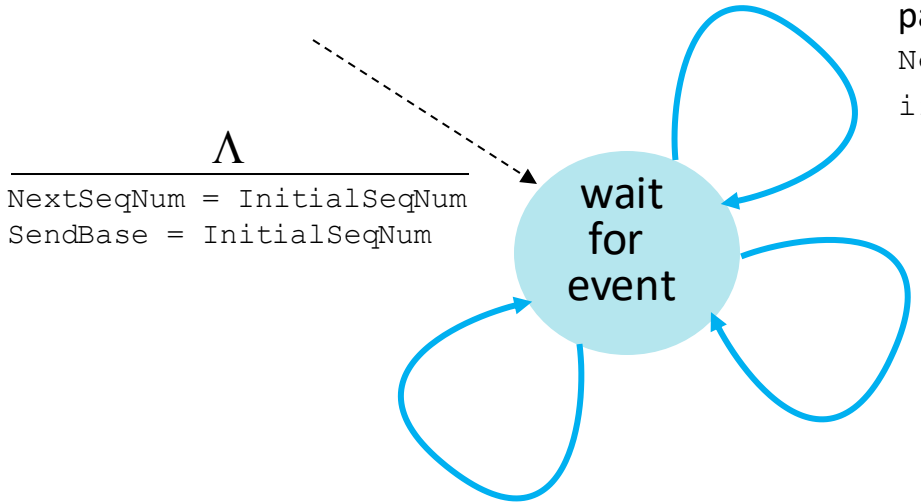
TCP Reliable Data Transfer

- Let’s initially consider simplified TCP sender
    - Ignore duplicate acks
    - Ignore flow control, congestion control

Simplified TCP Sender

- **Data received from application**
  - Create segment with sequence number
  - Sequence number is byte-stream number of first data byte in segment
  - Start timer if not already running
    - Think of timer as for oldest unACKed segment
    - Expiration interval: **`TimeOutInterval`**
- **Timeout**
  - Retransmit segment that caused timeout
  - Restart timer
- **ACK received**
  - If ACK acknowledges previously un-ACKed segments
    - Update what is known to be ACKed
    - Start timer if there are still un-acked segments

Simplified TCP Sender



Data received from application above

Create segment, sequence number: `NextSeqNum`
pass segment to IP

```
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running) start timer
```

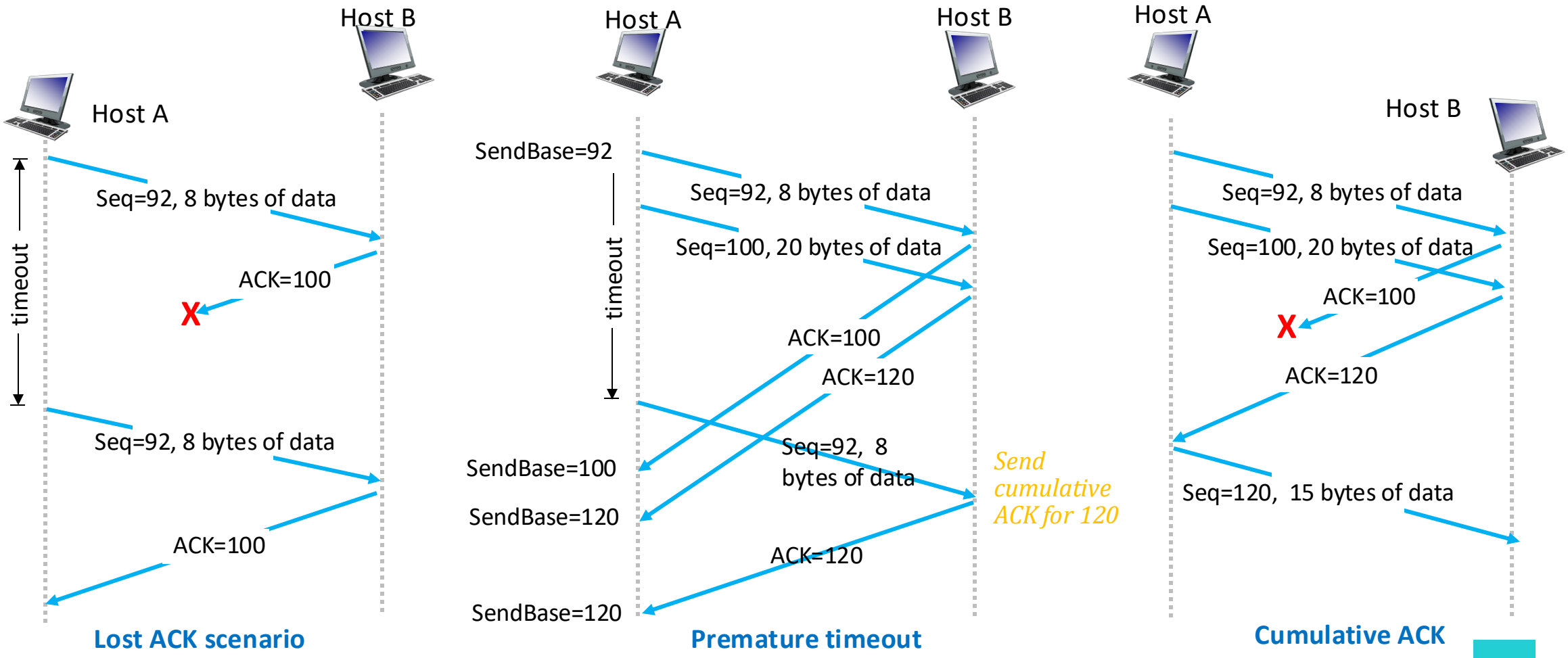
Timeout

Retransmit not-yet-acked segment with smallest sequence number
Start timer

ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase-1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
         start timer
       else stop timer
     }
```

TCP Retransmission Scenarios



TCP ACK Generation (RFC 1122, RFC 2581)

Event at receiver	*TCP receiver action*
Arrival of in-order segment with expected sequence number. All data up to expected sequence number already ACKed	*Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK*
arrival of in-order segment with expected sequence number. One other segment has ACK pending	*Immediately send single cumulative ACK, ACKing both in-order segments*
arrival of out-of-order segment higher-than-expect Sequence number . Gap detected	*Immediately send **duplicate ACK,** indicating Sequence number of next expected byte*
Arrival of segment that partially or completely fills gap	*Immediate send ACK, provided that segment starts at lower end of gap*

TCP Fast Retransmit

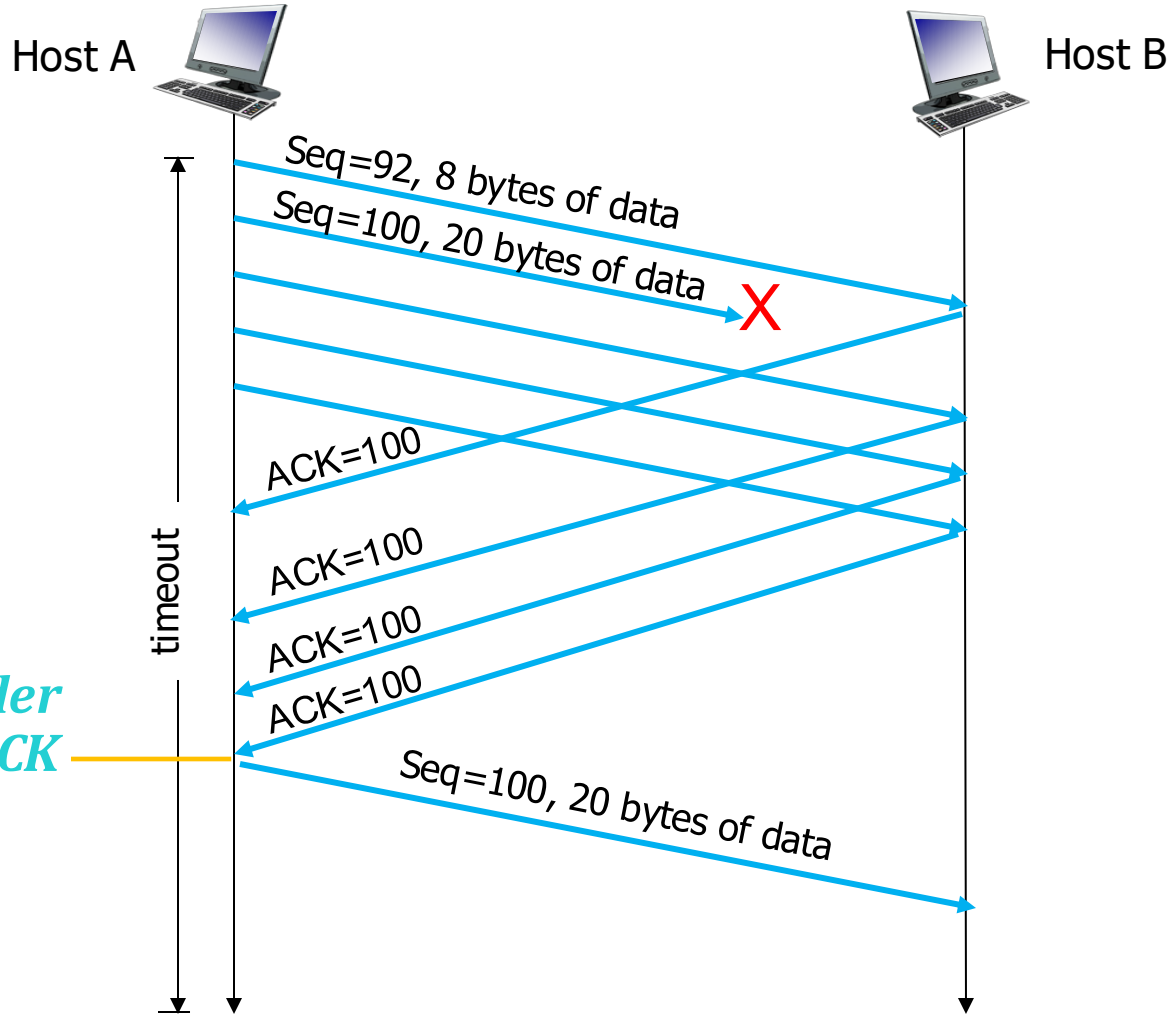
- Time-out period often relatively long
  - Long delay before resending lost packet

- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

- If sender receives 3 ACKs for same data (**Triple Duplicate ACKs**), resend un-ACKed segment with smallest sequence number
  - Likely that un-ACKed segment lost, so don't wait for timeout

TCP Fast Retransmit

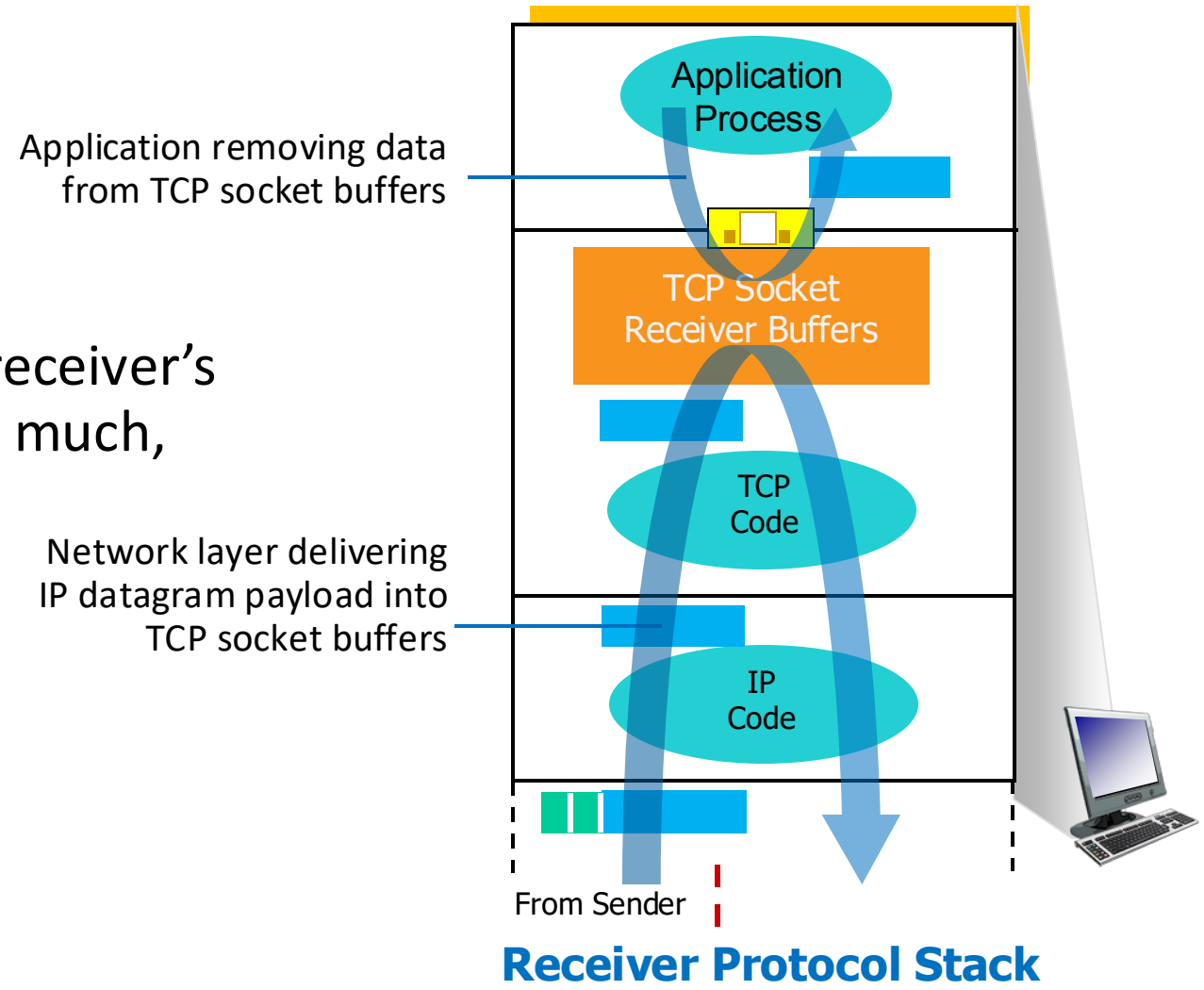


Fast retransmit after sender receipt of triple duplicate ACK

TCP Flow Control

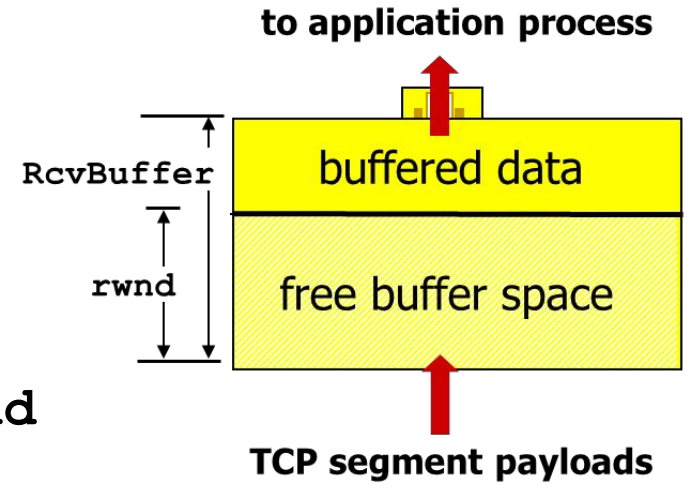
Flow control

- Receiver controls sender
- Sender will not overflow receiver's buffer by transmitting too much, too fast



TCP Flow Control

- Receiver advertises free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - Many operating systems auto-adjust **`RcvBuffer`**
- Sender limits amount of un-ACKed (in-flight) data to receiver's **`rwnd`** value
- Guarantees receive buffer will not overflow

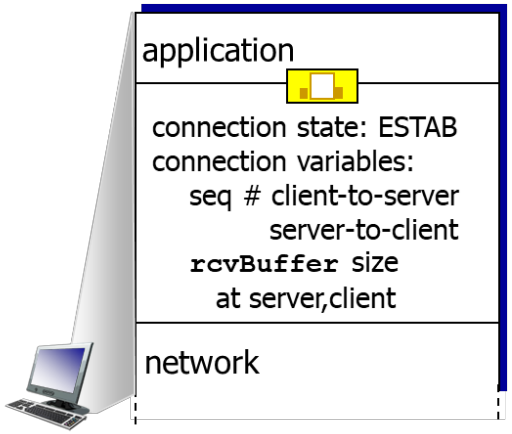


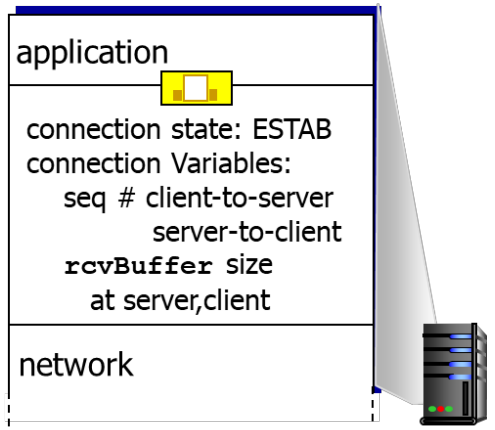
receiver-side buffering

Connection Management

Before exchanging data, sender and receiver **handshake**

- Agree to establish connection
  (each knowing the other willing to establish connection)
- Agree on connection parameters



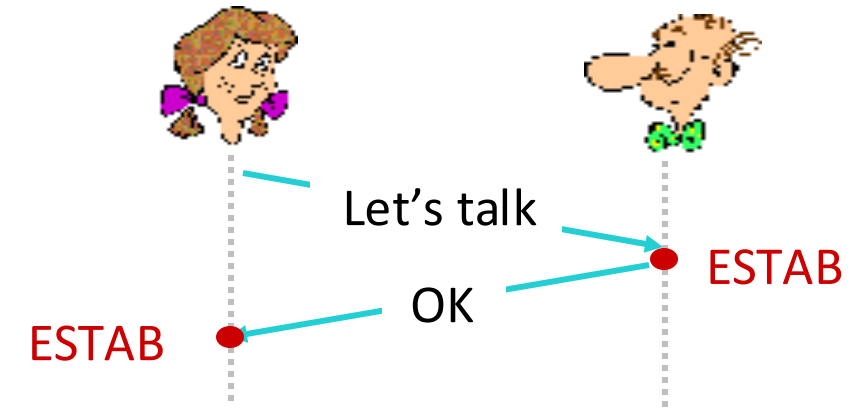


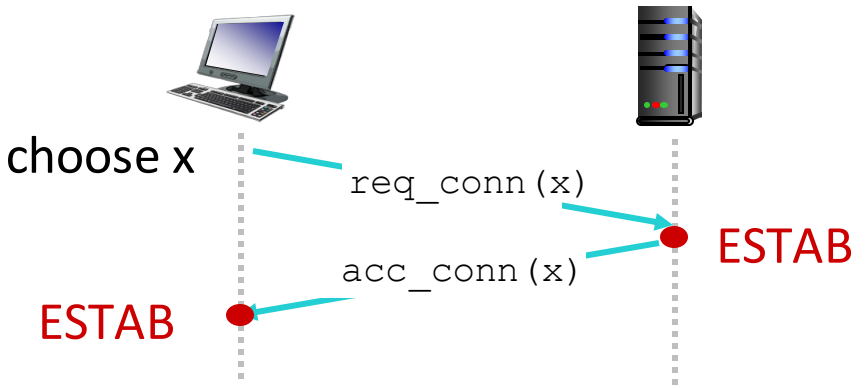

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

```
Socket connectionSocket =
  welcomeSocket.accept();
```

Agree to Establish a Connection

Two-way handshake



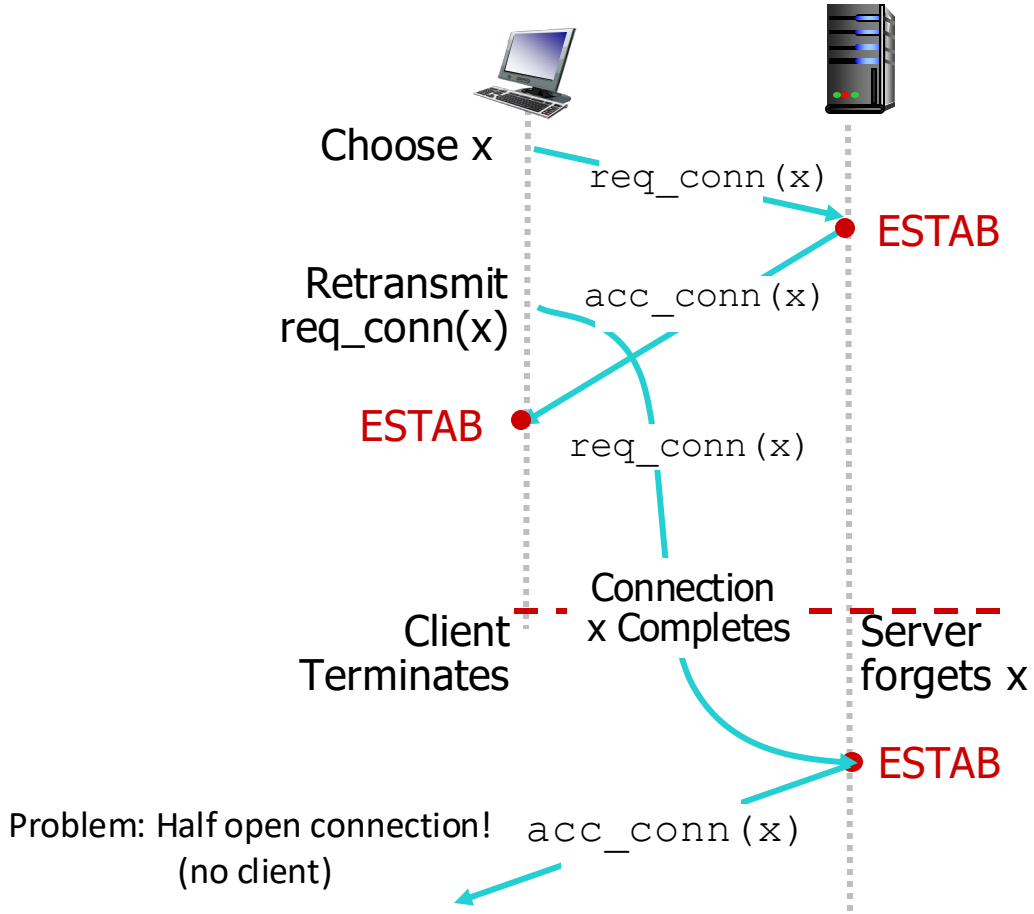


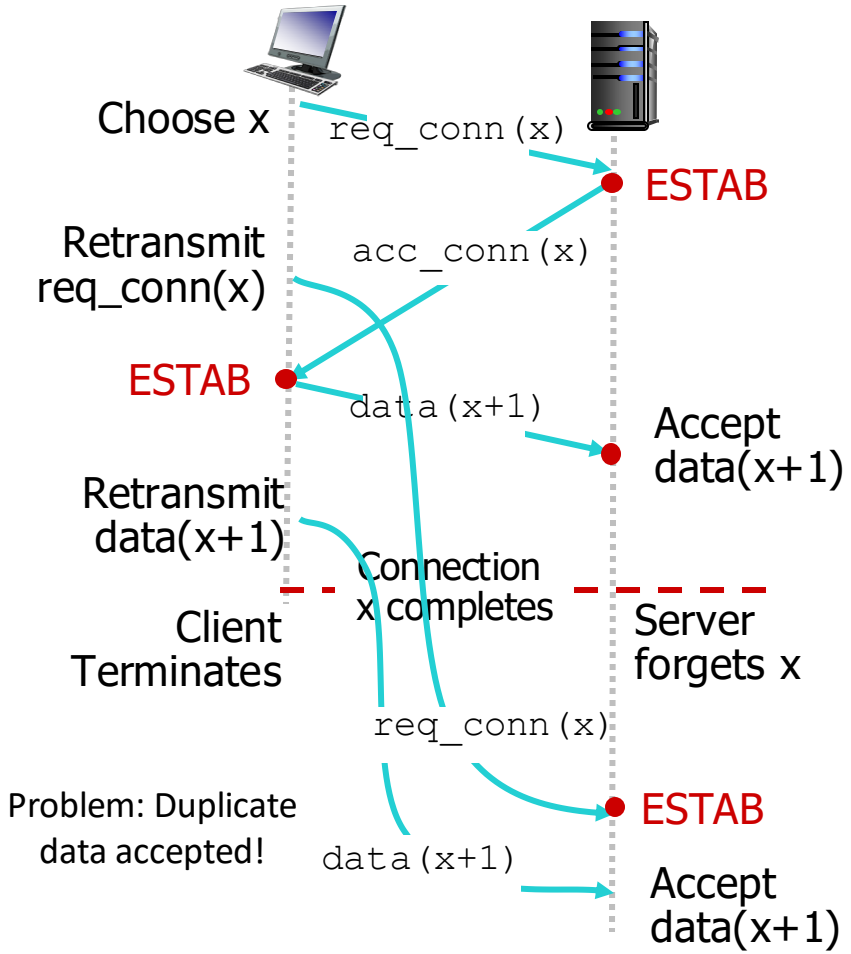
Q: Will two-way handshake always work in network?

- Variable delays
- Retransmitted messages due to message loss
- Message reordering
- Can not **see** other side

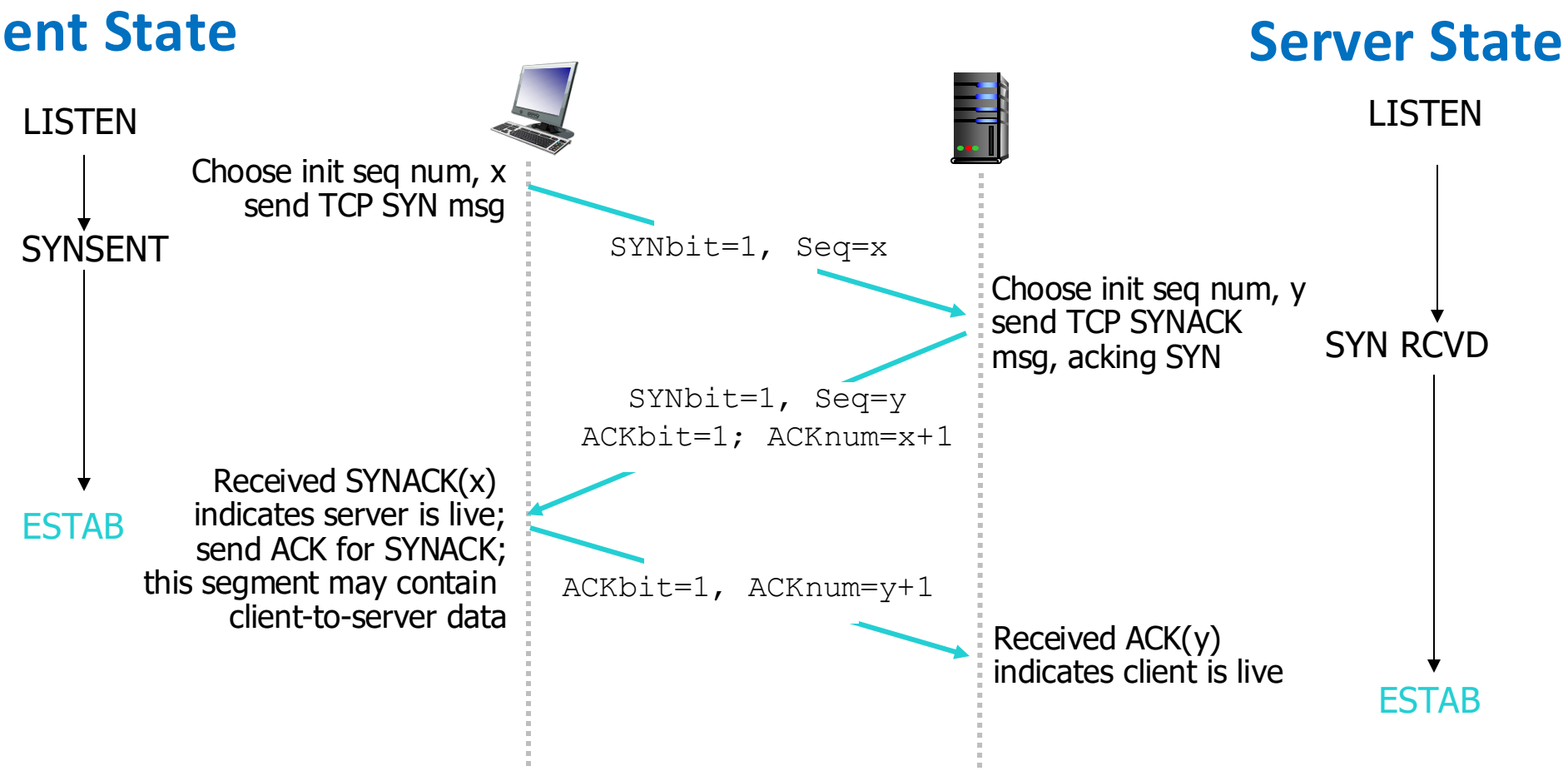
Agreeing To Establish a Connection

- Two-way handshake failure scenarios

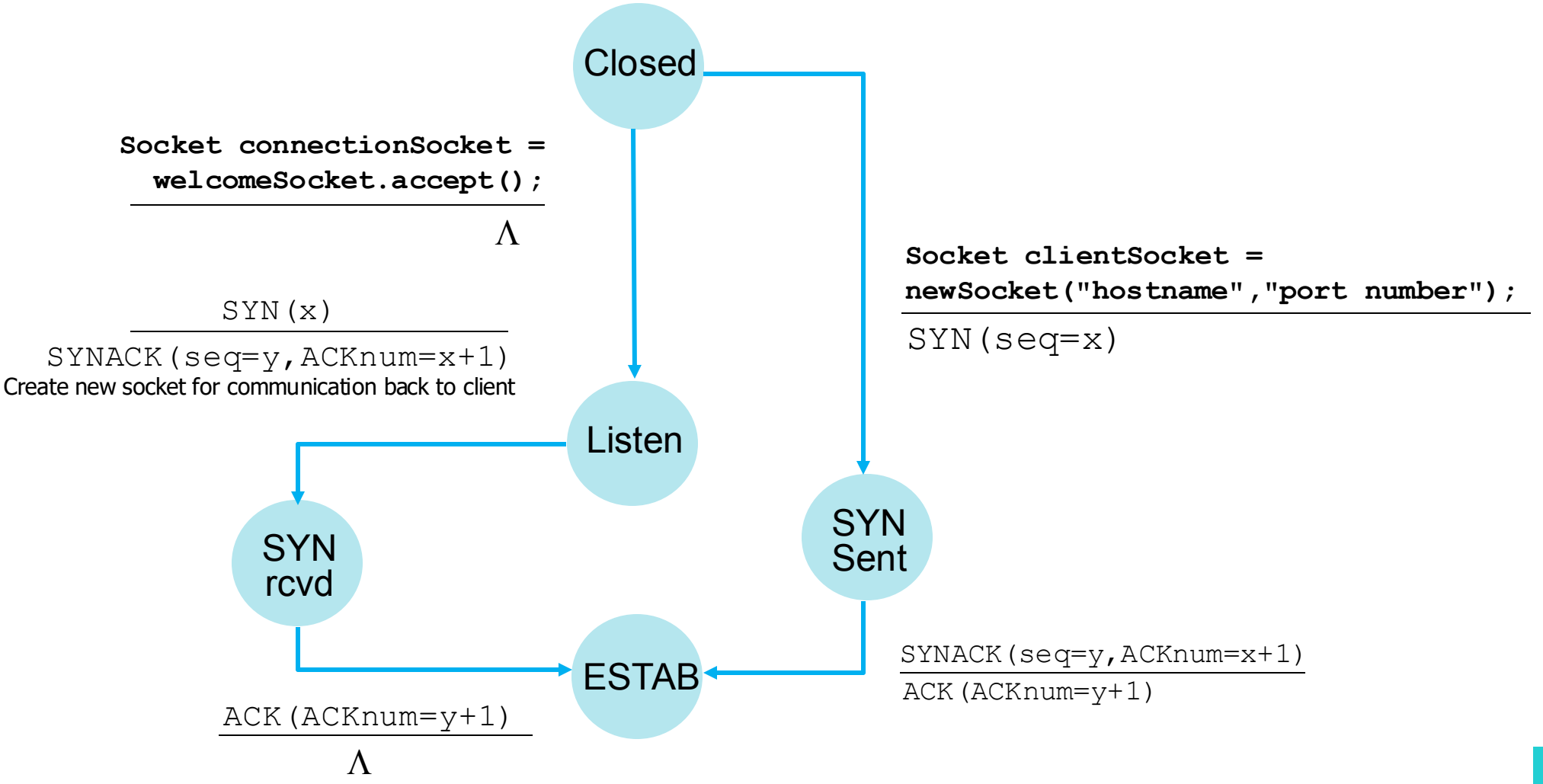




TCP Three-Way Handshake



TCP Three-Way Handshake: FSM



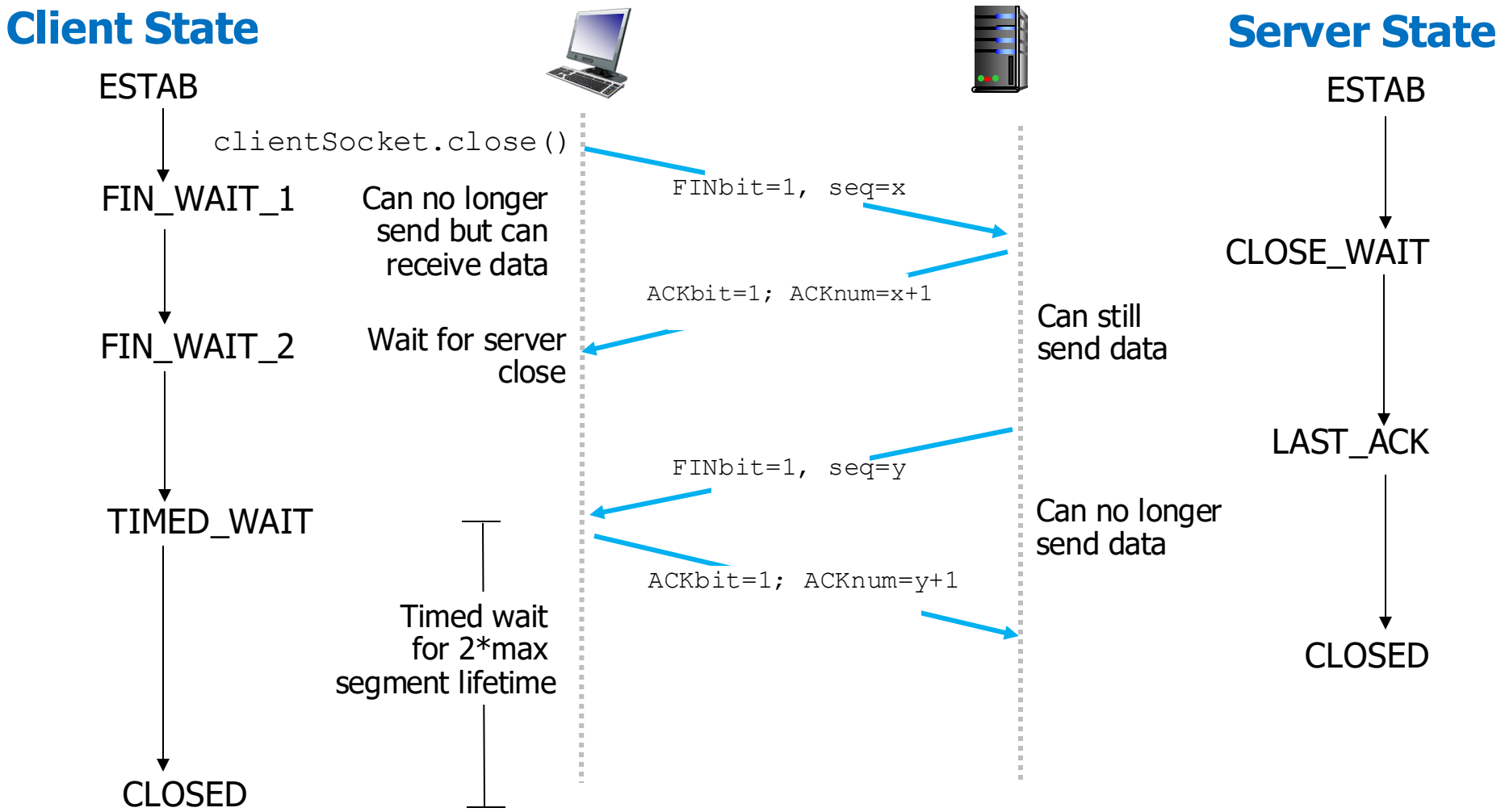
TCP: Closing A Connection

- Client and server each close their side of connection
  - Send TCP segment with FIN bit = 1

- Respond to received FIN with ACK
  - On receiving FIN, ACK can be combined with own FIN

- Simultaneous FIN exchanges can be handled

TCP: Closing A Connection



Congestion Control

Image generated by Google Gemini to depict network congestion

Principles of Congestion Control

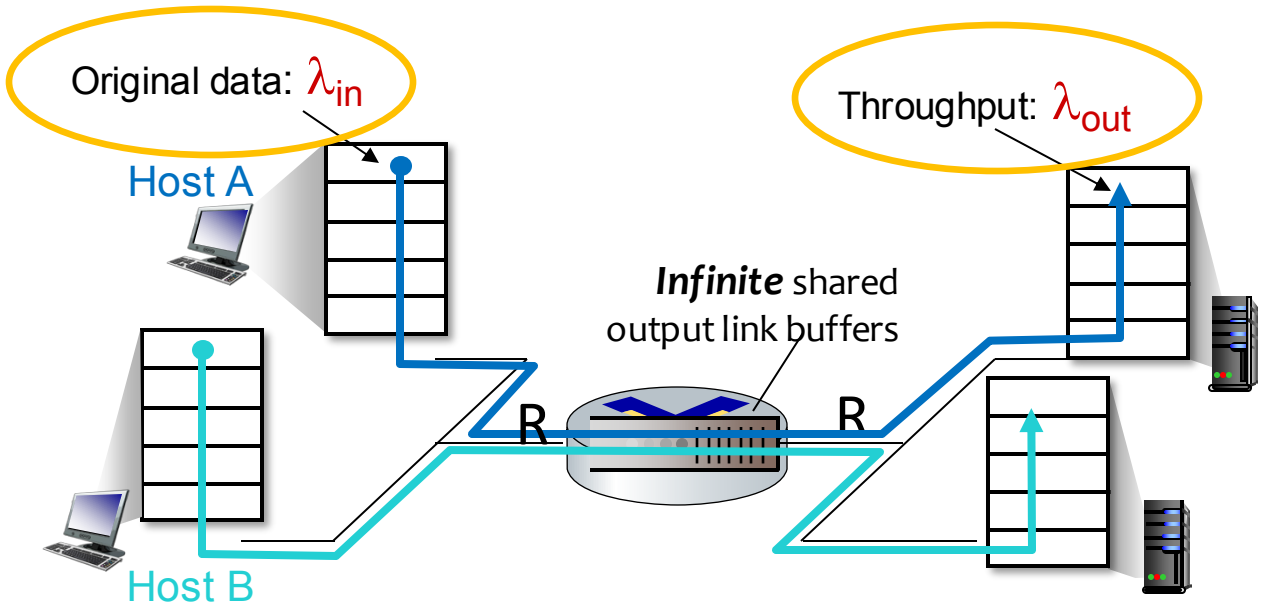
Congestion:

- Informally: Too many sources sending too much data too fast for **network** to handle.
  - Manifestations:
    - Lost packets (buffer overflow at routers)
    - Long delays (queueing in router buffers)

- Different from flow control!

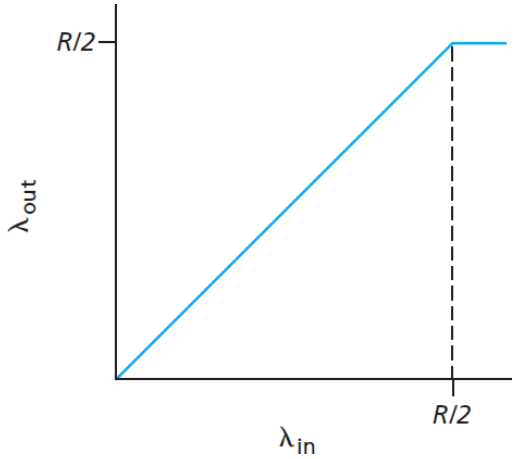
Causes & Costs of Congestion: First Scenario

- Two senders, two receivers
- One router, infinite buffers
- Output link capacity: **R**
- No retransmission

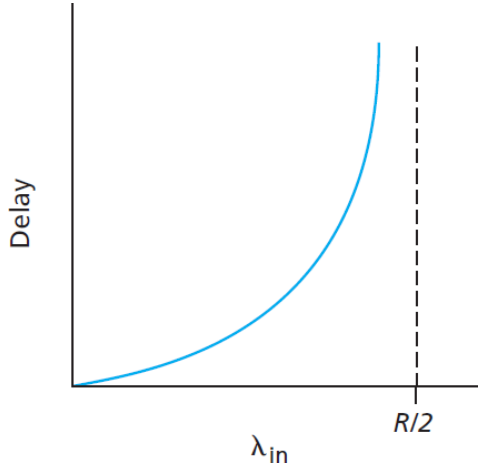


Causes & Costs of Congestion: First Scenario

- Maximum per-connection **throughput**: $\frac{R}{2}$

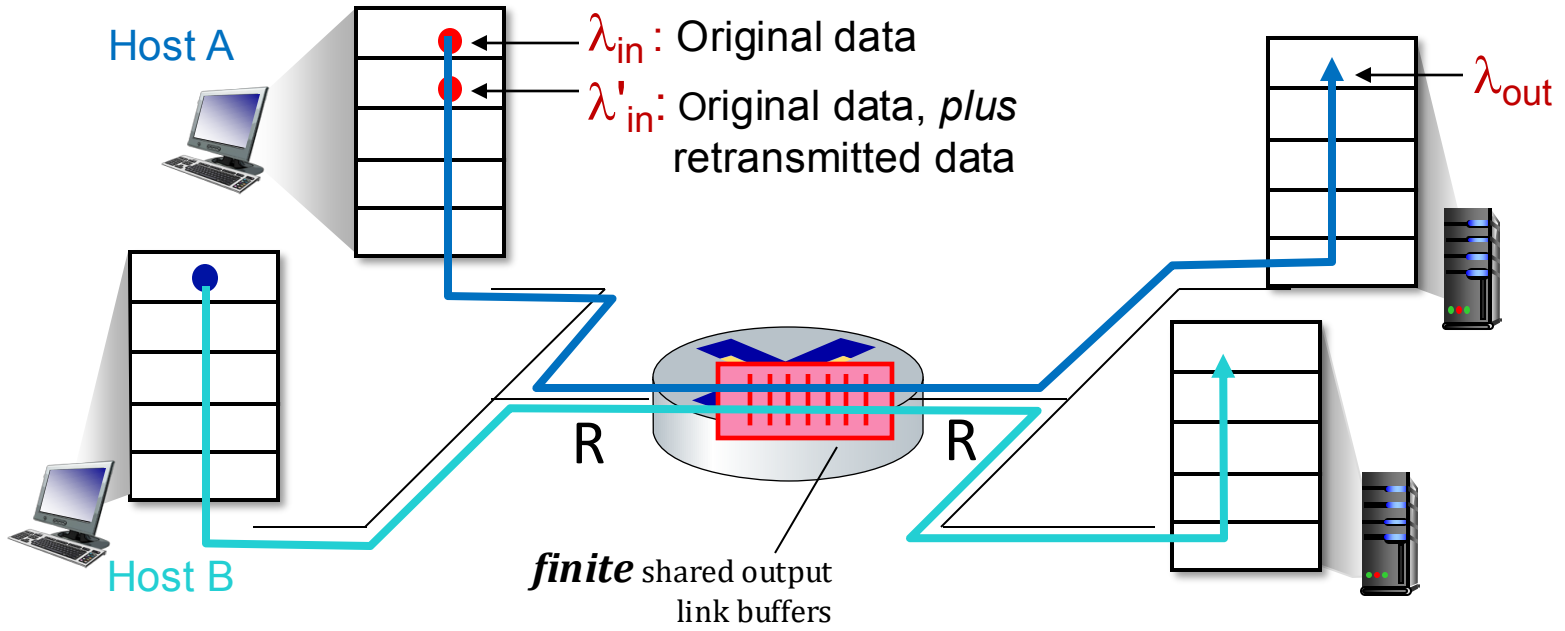


- Large **delays** as arrival rate, λ_{in}, approaches capacity



Causes & Costs of Congestion: Second Scenario

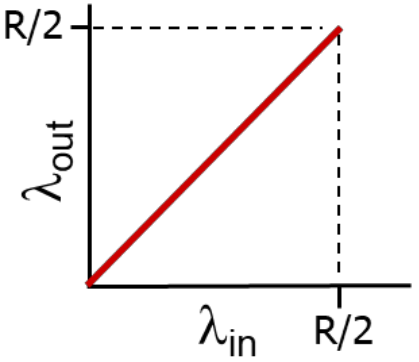
- One router, **finite** buffers
- Sender retransmission of timed-out packet
- Application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
- Transport-layer input includes **retransmissions**: $\lambda'_{in} \geq \lambda_{in}$

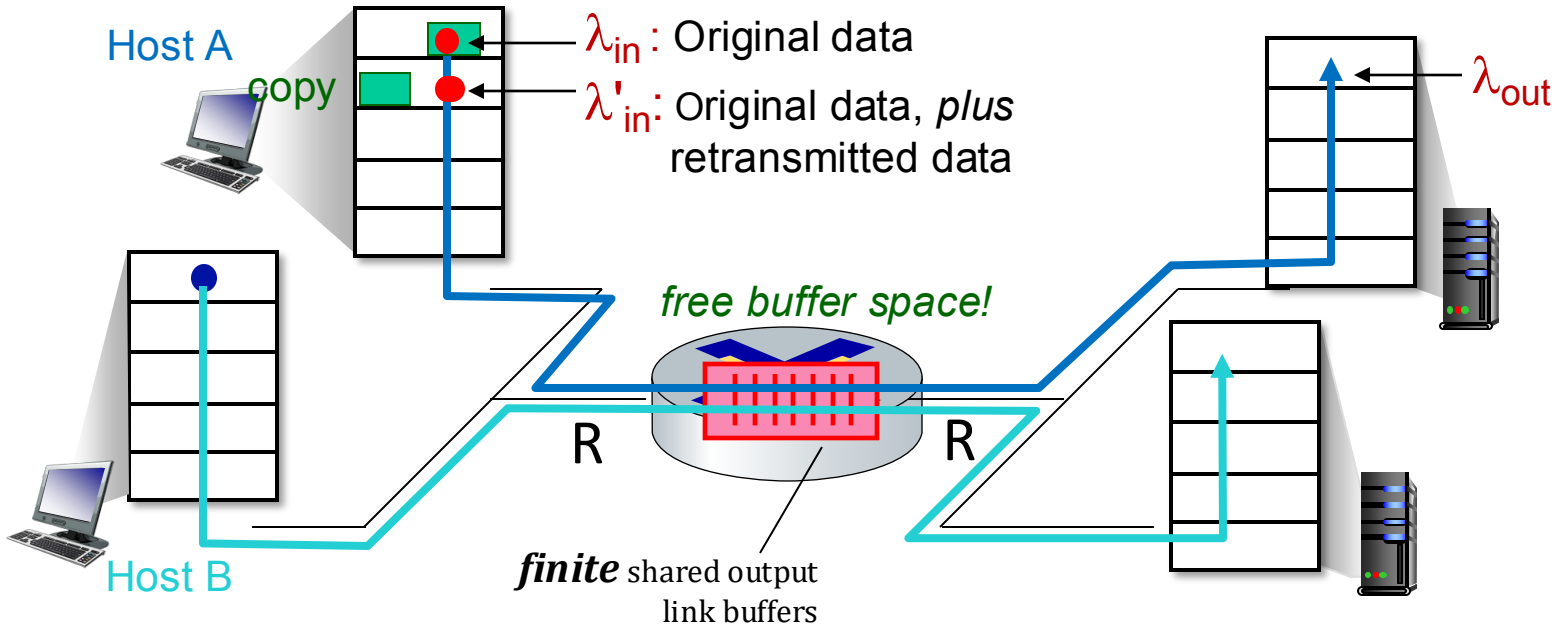


Causes & Costs of Congestion: Second Scenario

Idealization: Perfect knowledge

- Sender sends only when router buffers available

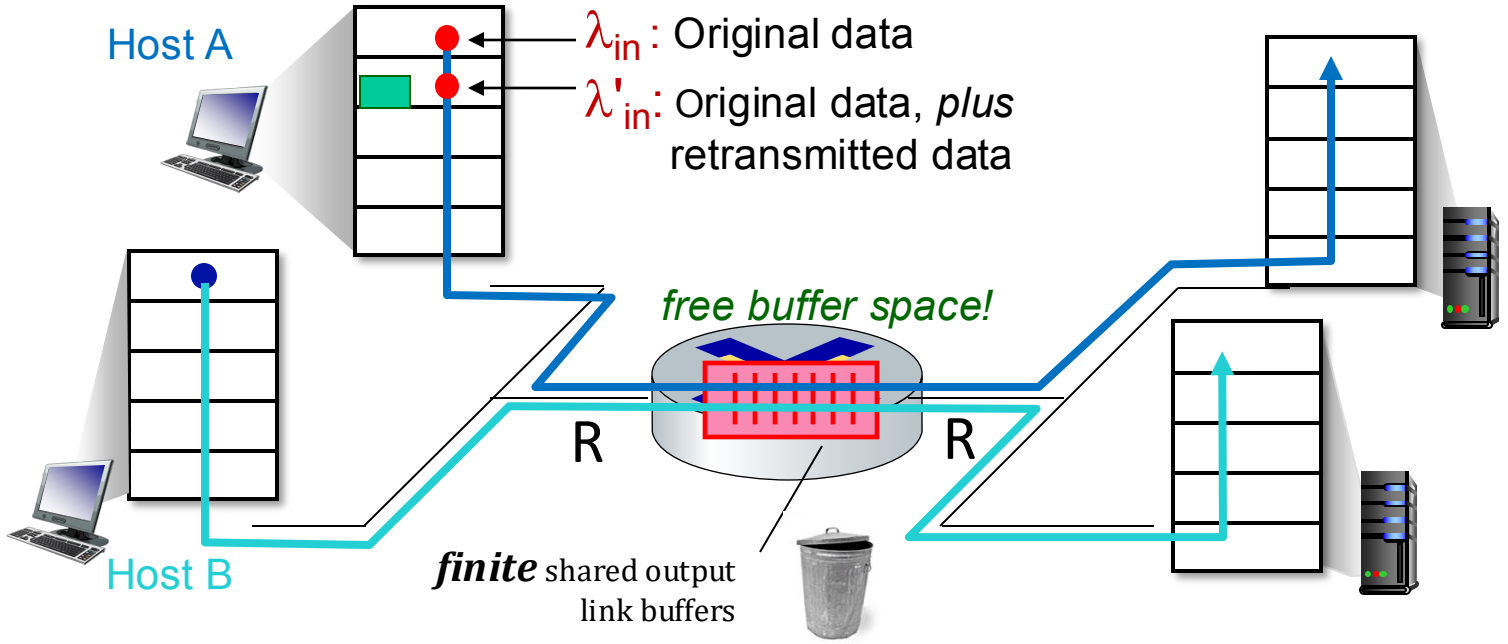




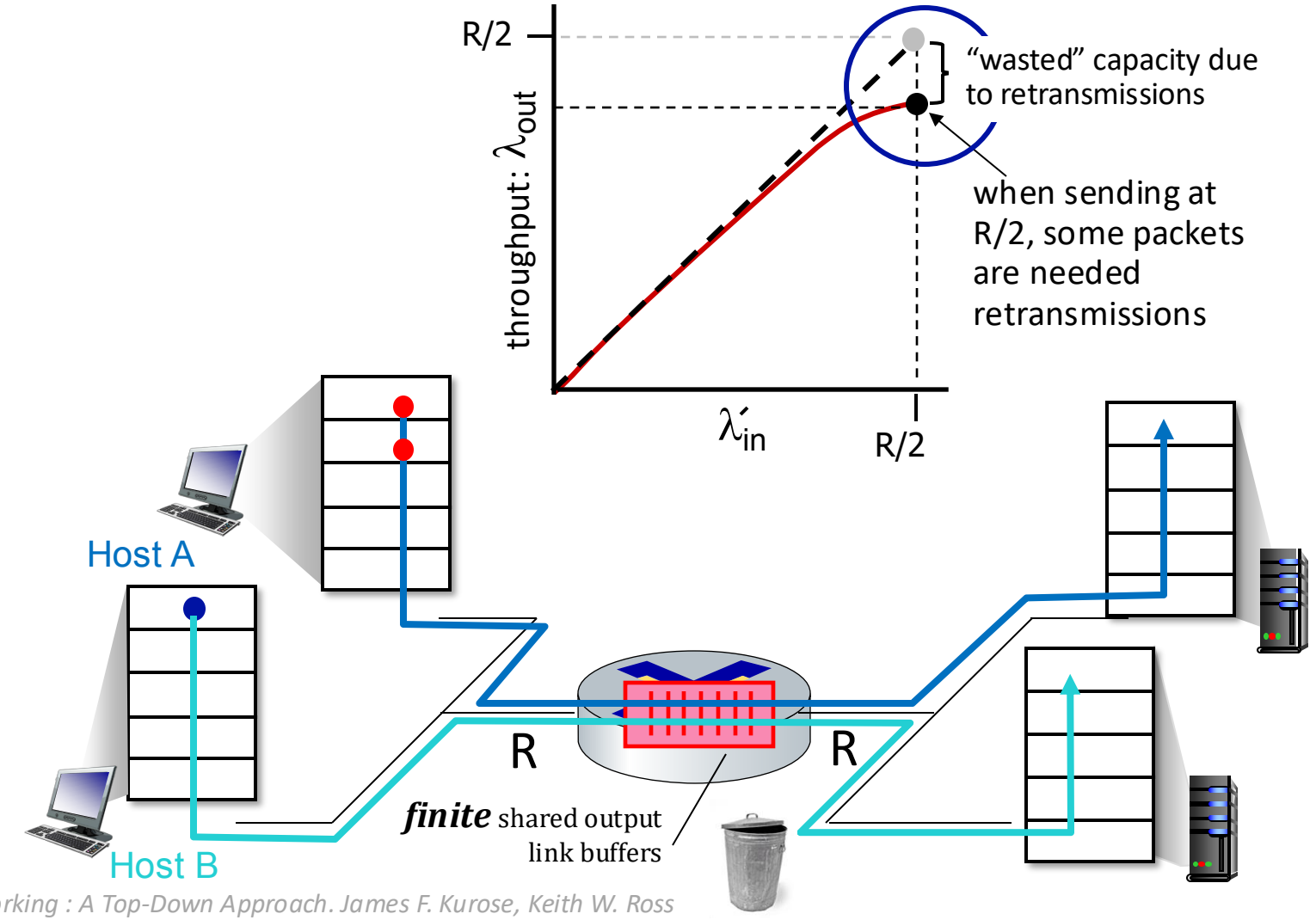
Causes & Costs of Congestion: Second Scenario

Idealization: Known loss

- Packets can be lost, dropped at router due to full buffers
- Sender only resends if packet **known** to be lost



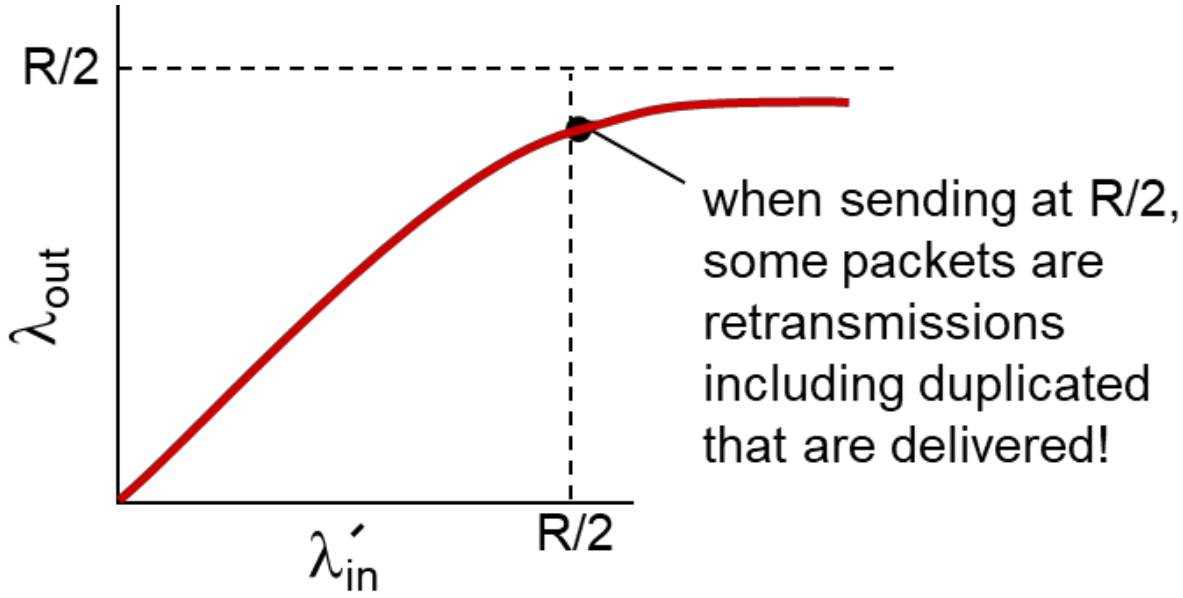
Causes & Costs of Congestion: Second Scenario



Causes & Costs of Congestion: Second Scenario

Realistic: Duplicates

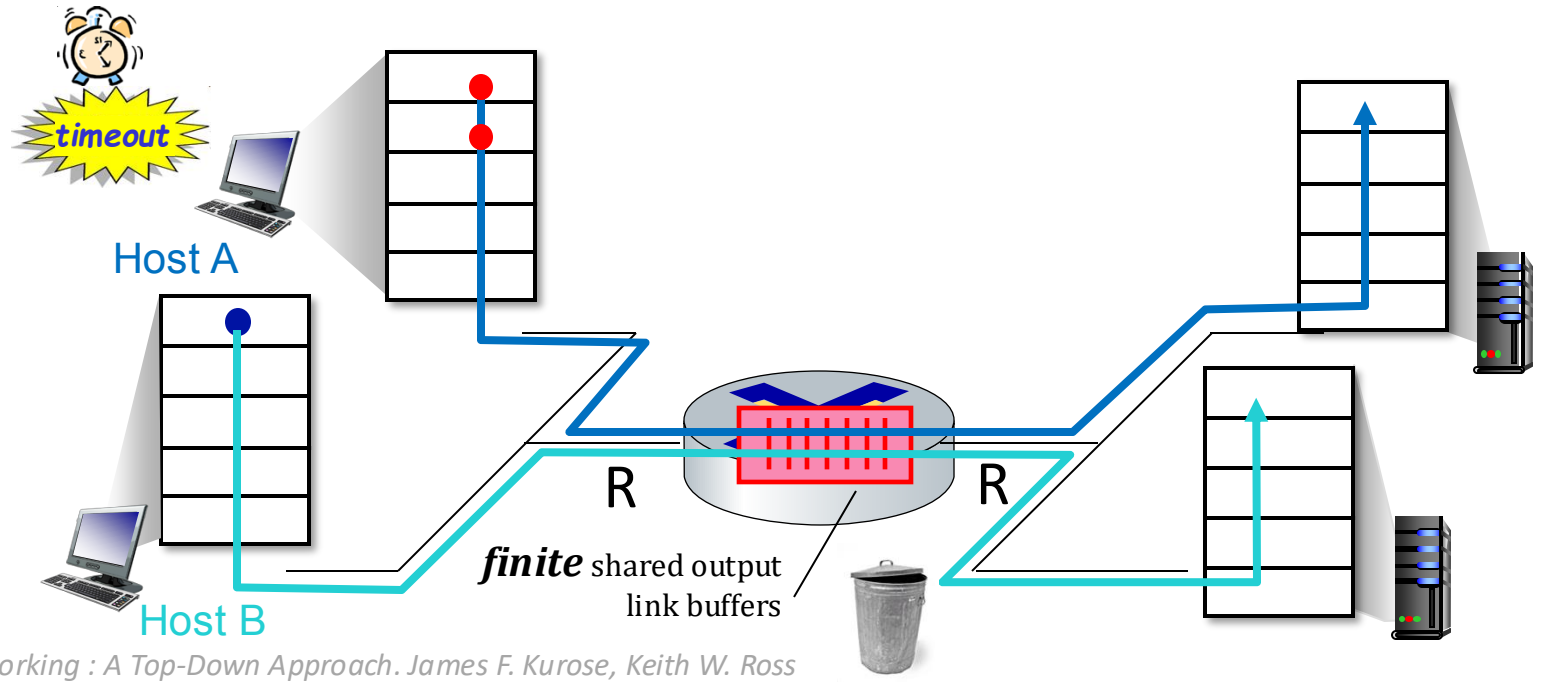
- Packets can be lost, dropped at router due to full buffers
- Sender times out prematurely, sending **two** copies, both of which are delivered



Causes & Costs of Congestion: Second Scenario

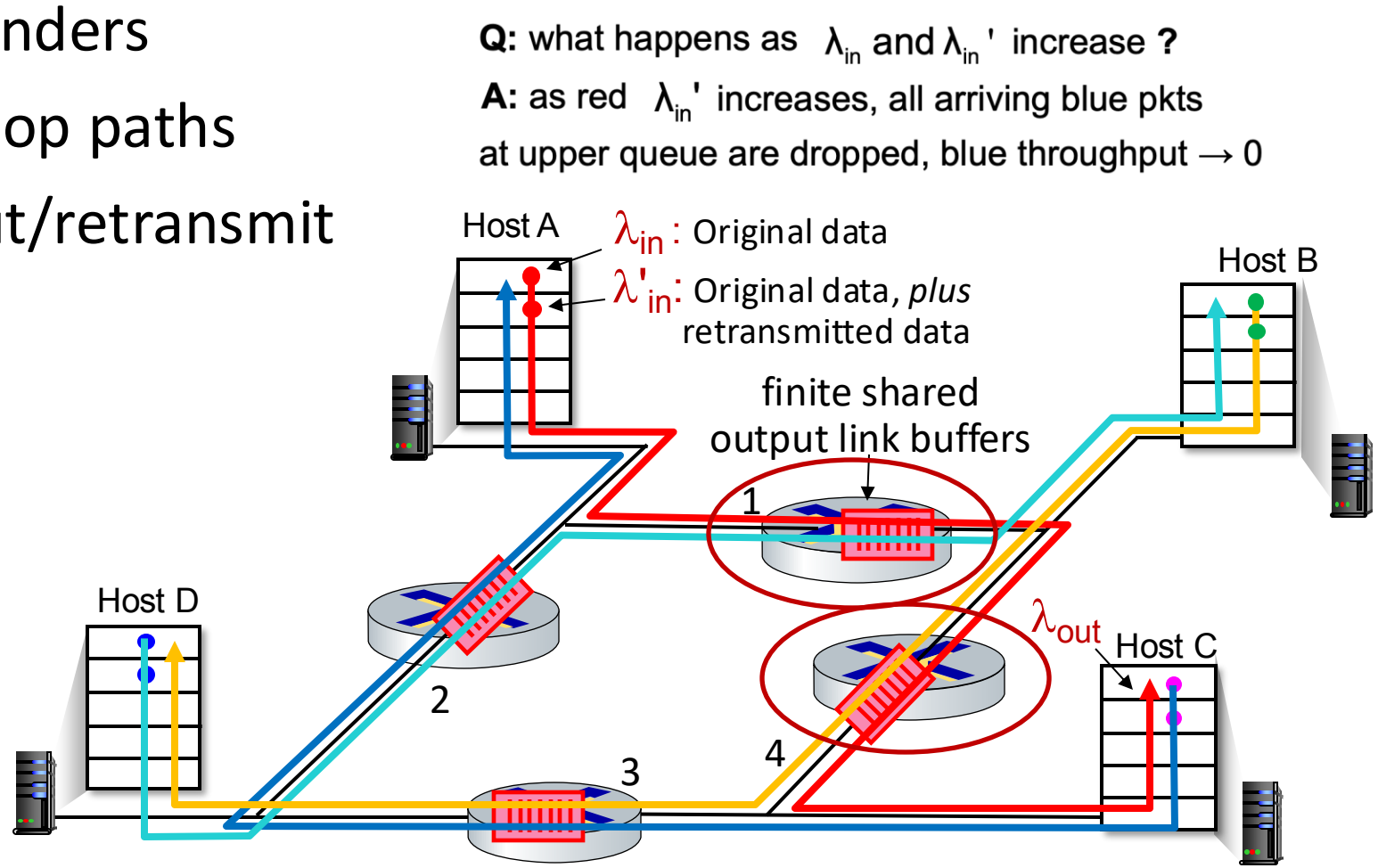
Costs of congestion:

- More work (retransmit) for given **goodput**
- Unneeded retransmissions: link carries multiple copies of packet
  - Decreasing goodput

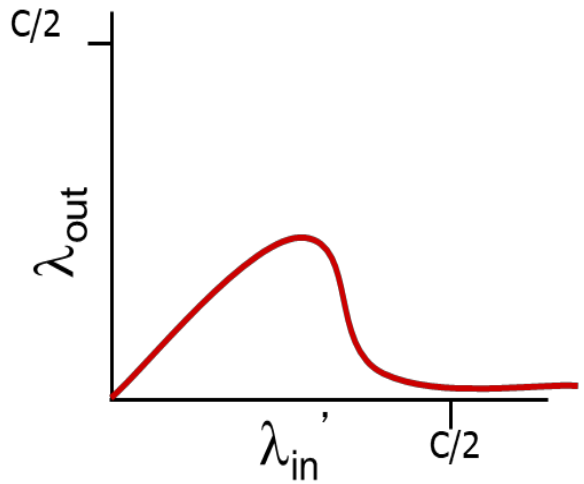


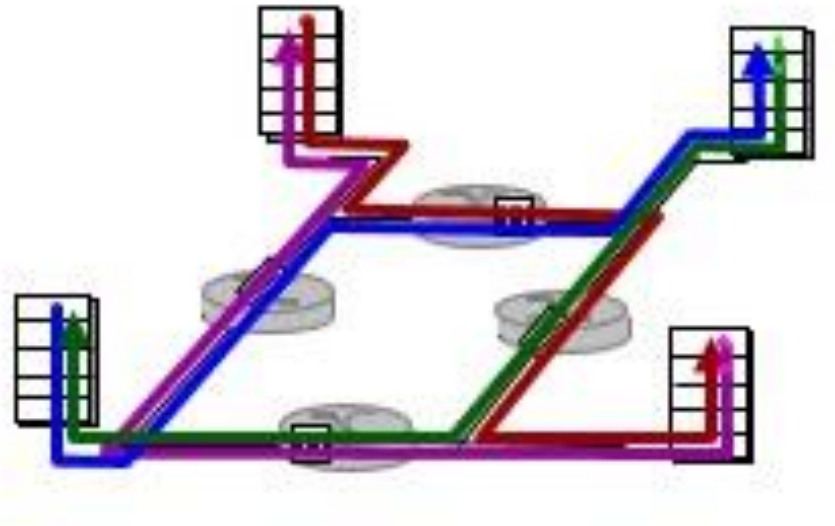
Causes & Costs of Congestion: Third Scenario

- Four senders
- Multi-hop paths
- Timeout/retransmit



Causes & Costs of Congestion: Third Scenario





Another cost of congestion

- When packet dropped, any upstream transmission capacity used for that packet was wasted!

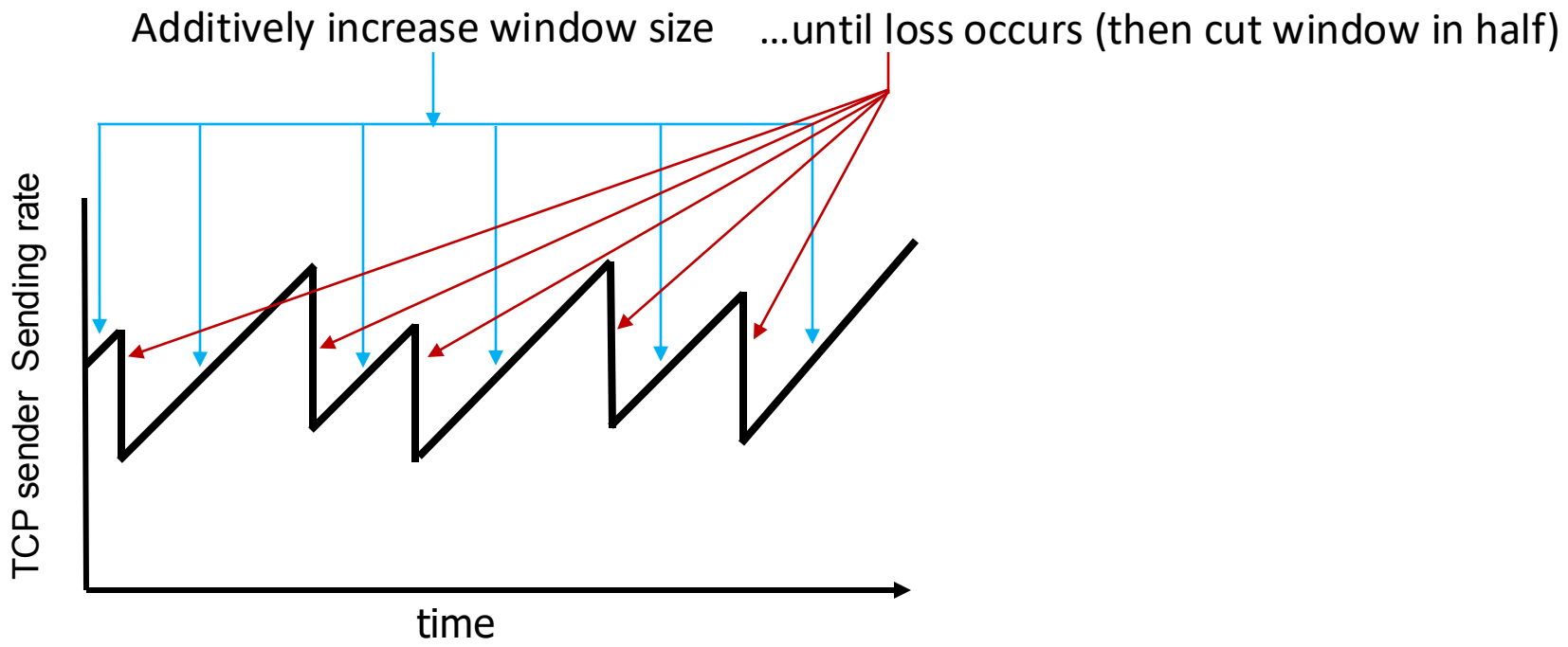
Image generated by Google Gemini to depict network congestion

TCP Congestion Control: AIMD

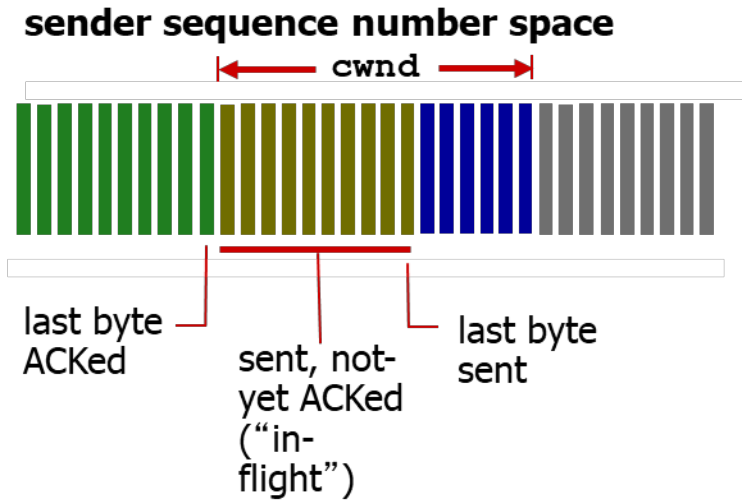
- **Approach:** Sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - **Additive Increase**
    - Increase **`cwnd`** by 1 MSS every RTT until loss detected
  - **Multiplicative Decrease**
    - Cut **`cwnd`** in half after loss

TCP Congestion Control: AIMD

AIMD **Sawtooth** behavior:
Probing for bandwidth



TCP Congestion Control: Details



- Sender limits transmission

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

cwnd is dynamic, function of perceived network congestion

TCP sending rate:

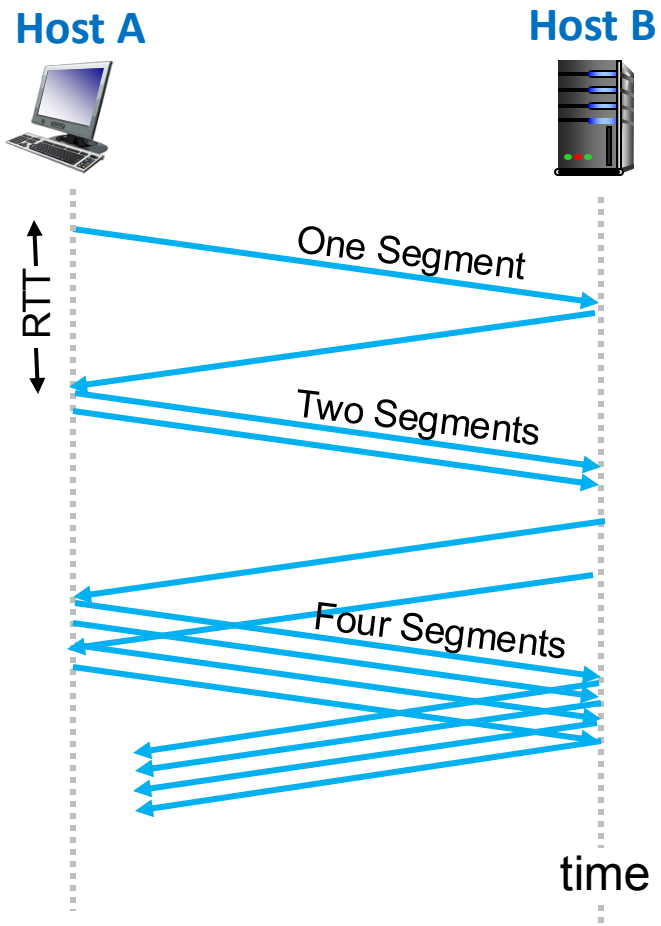
- **Roughly:** Send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

TCP Slow Start

- When connection begins
  increase rate exponentially until first loss event
  - Initially **`cwnd`** = 1 MSS
  - Double **`cwnd`** every RTT
  - Done by incrementing **`cwnd`** for every ACK received

- **Summary:** Initial rate is slow but ramps up exponentially fast



TCP: Detecting & Reacting to Loss

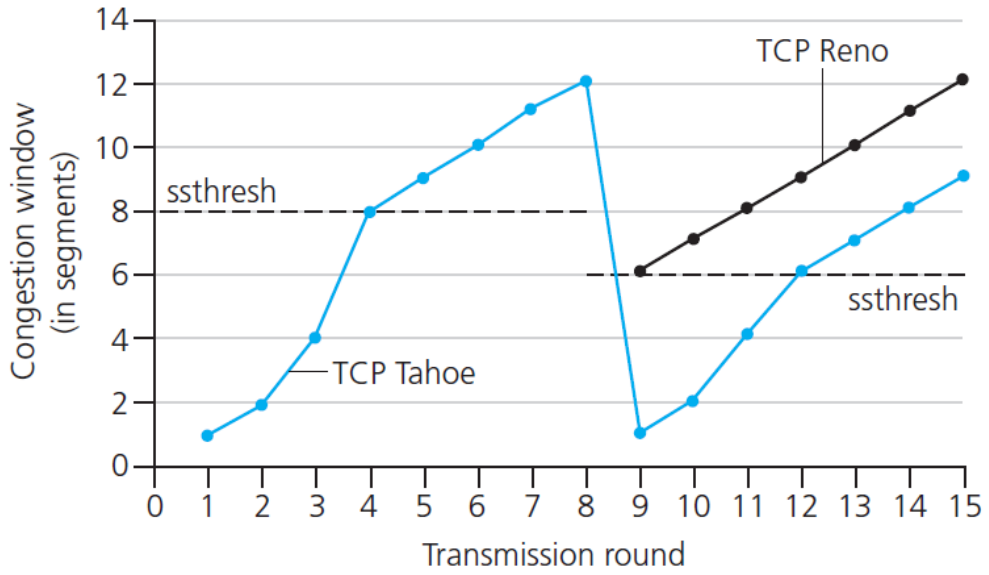
- Loss indicated by timeout:
  - **cwnd** set to 1MSS;
  - Window then grows exponentially (as in slow start) to threshold, then grows linearly

- Loss indicated by 3 duplicate ACKs: TCP RENO
  - Duplicate ACKs indicate network capable of delivering some segments
  - **cwnd** is cut in half window then grows linearly

- TCP Tahoe always sets **cwnd** to 1 (timeout or **3** duplicate acks)

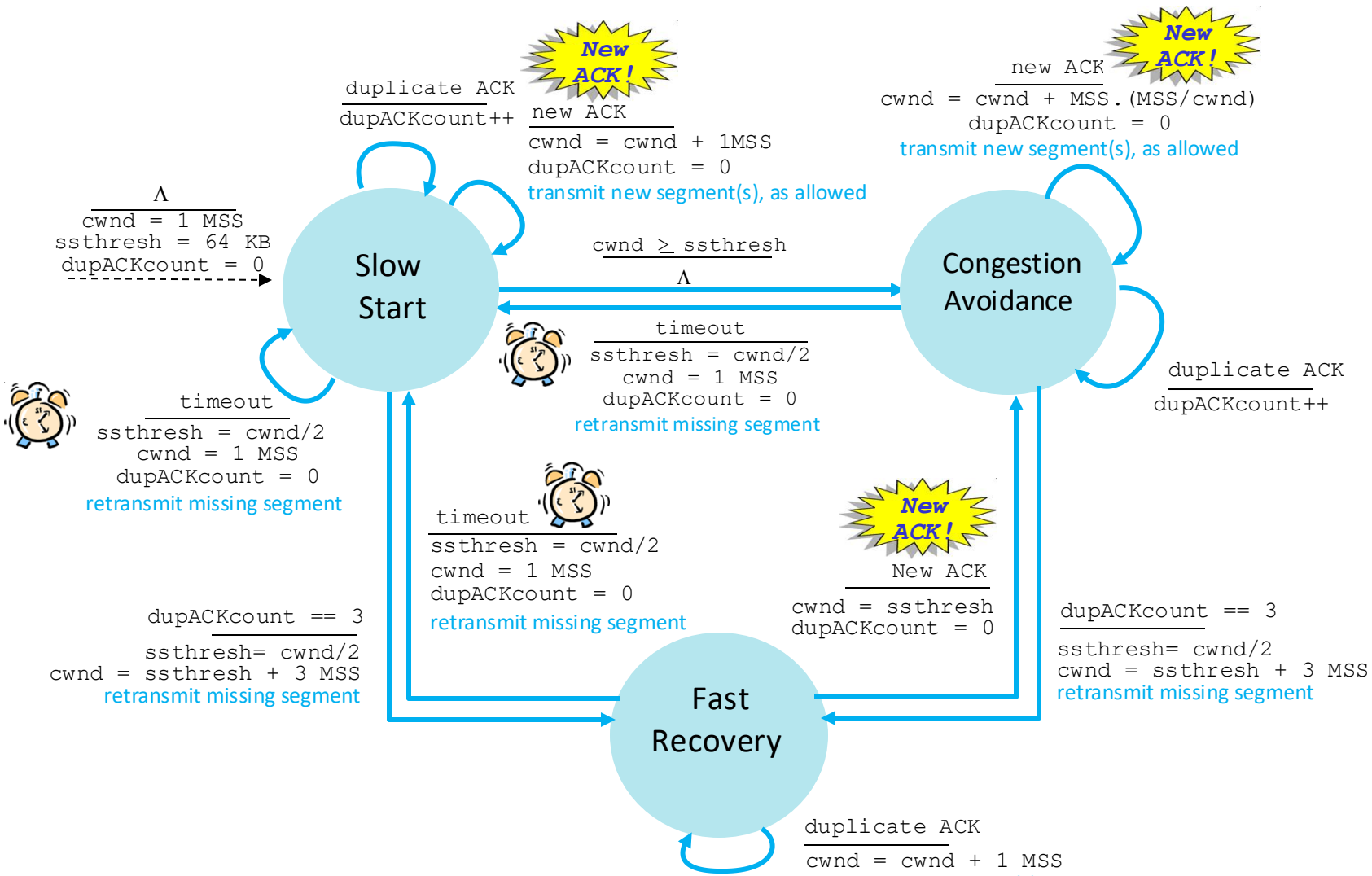
TCP: Switching from Slow Start to CA

- **Q:** When should the exponential increase switch to linear?
- **A: W**hen **`cwnd`** gets to ½ of its value before timeout.



- **Implementation**
  - Variable **`ssthresh`**
  - On loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event

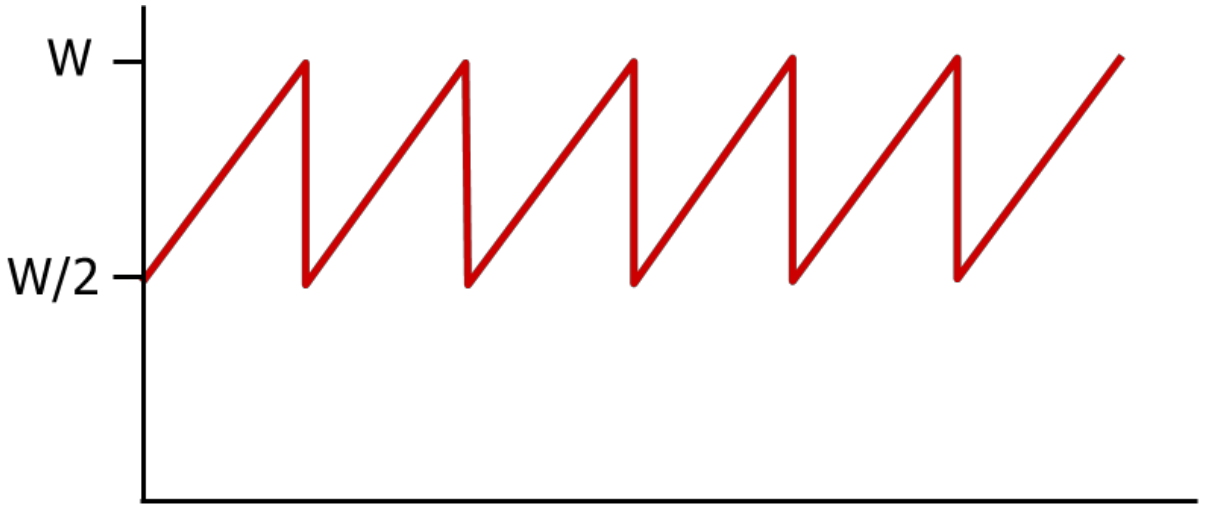
TCP Congestion Control



TCP Throughput

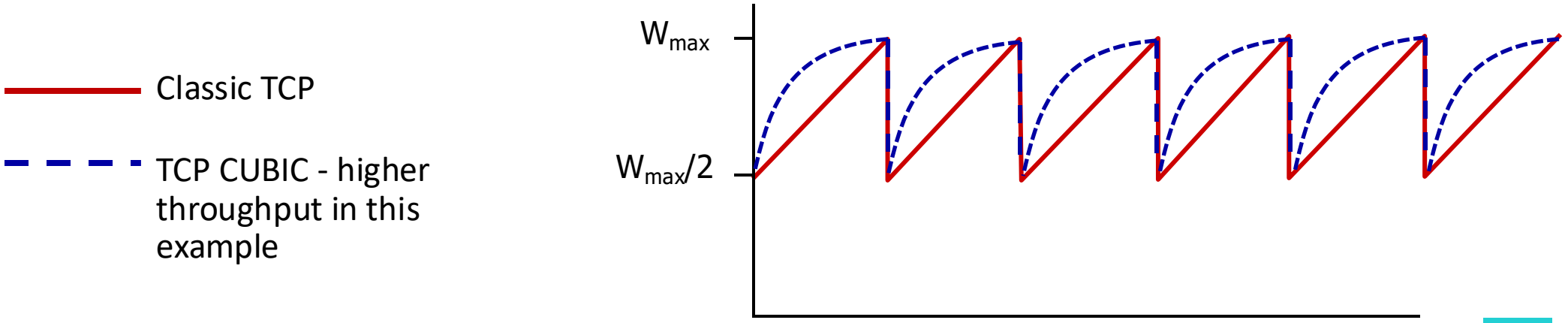
- Average TCP throughput as function of window size, RTT?
  - Ignore slow start, assume always data to send

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes / sec}$$



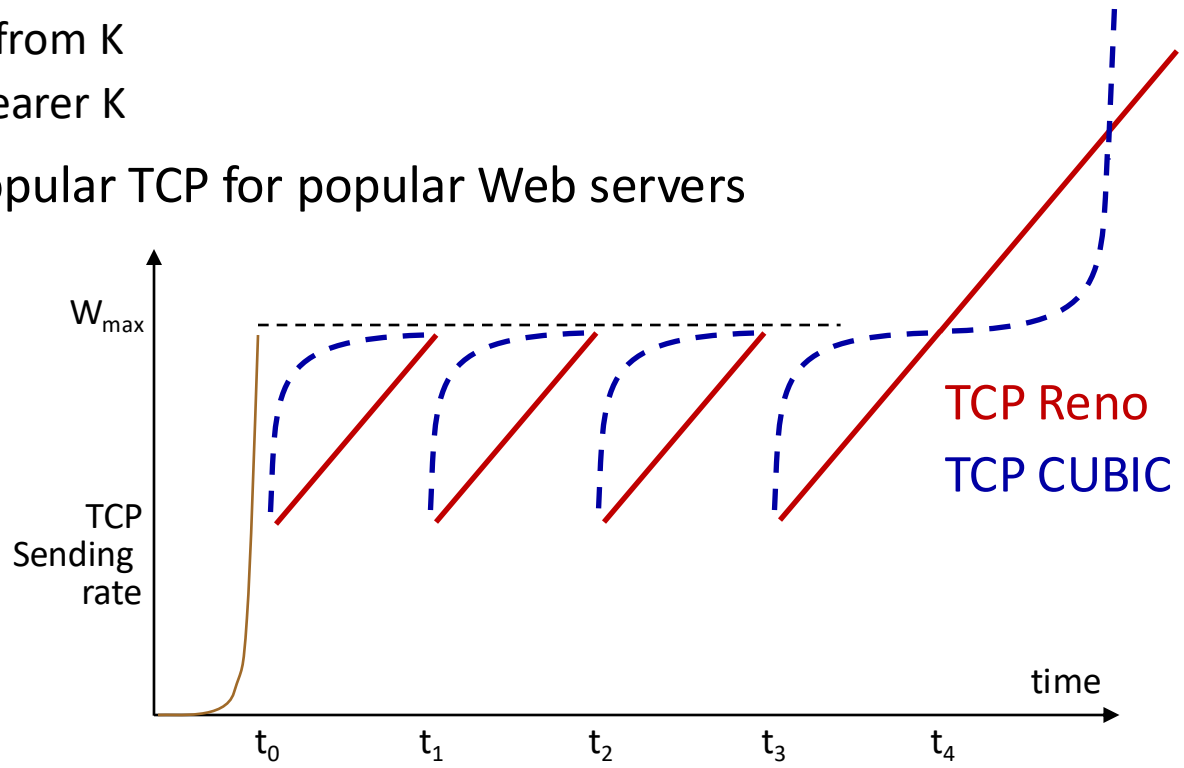
TCP CUBIC

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: Sending rate at which congestion loss was detected
  - Congestion state of bottleneck link probably (?) hasn't changed much
  - After cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then Approach W_{max} more *slowly*



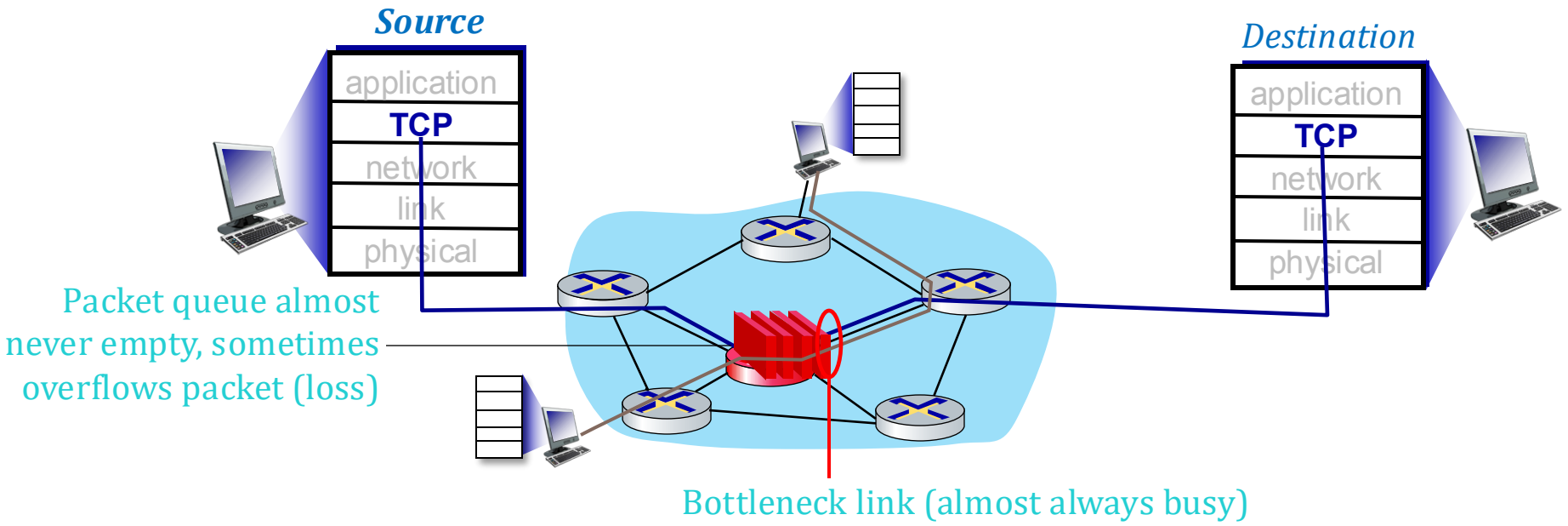
TCP CUBIC

- K: Point in time when TCP window size will reach W_{max}
  - K itself is tunable
- Increase W as a function of the *cube* of the distance between current time and K
  - Larger increases when further away from K
  - Smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



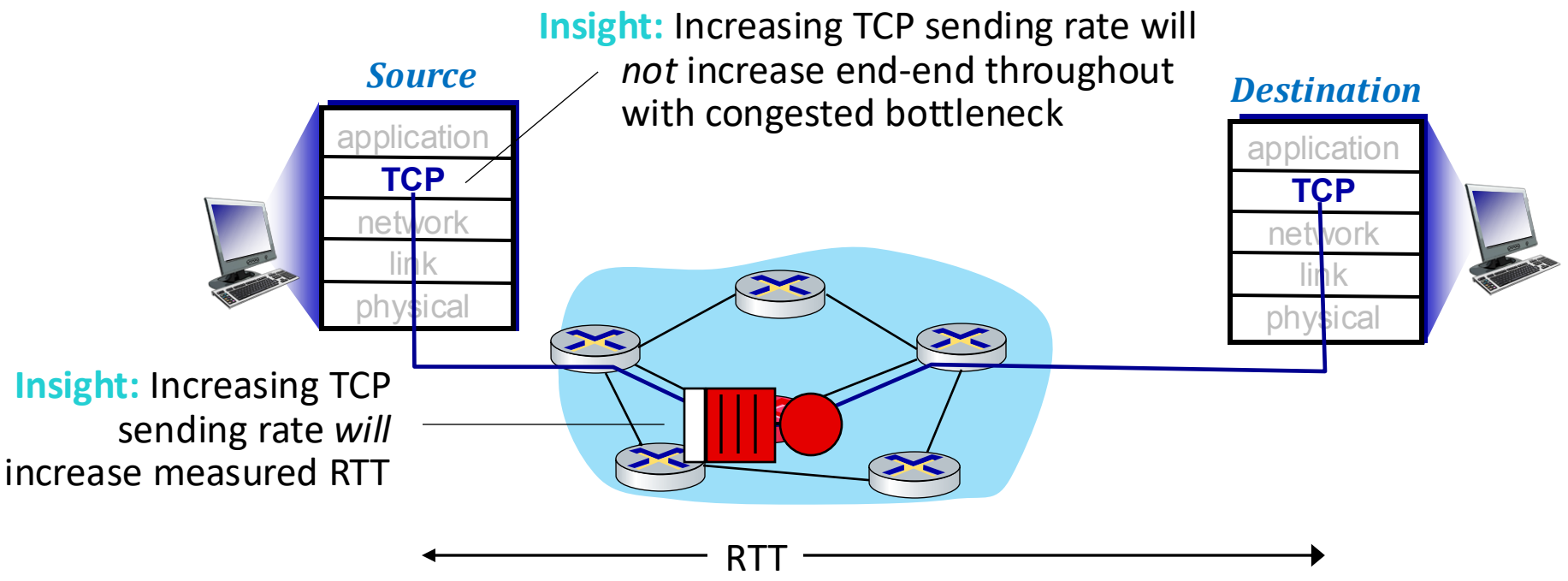
TCP & The Bottleneck Link

TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: The bottleneck link.



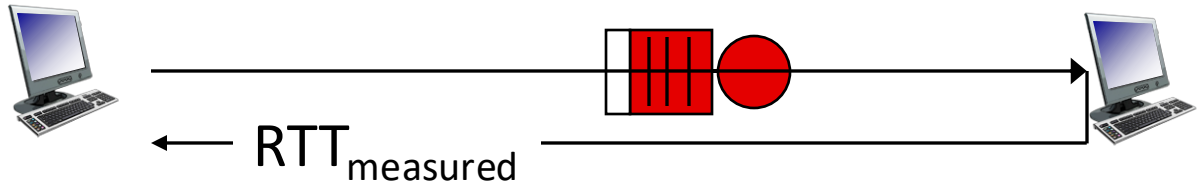
TCP & The Bottleneck Link

Understanding congestion: Useful to focus on congested bottleneck link



Delay-based TCP Congestion Control

- Keep bottleneck link busy transmitting, but avoid high delays and buffering
  - RTT_{min} - Minimum observed RTT (uncongested path)
  - Uncongested throughput with congestion window cwnd is $cwnd/RTT_{min}$



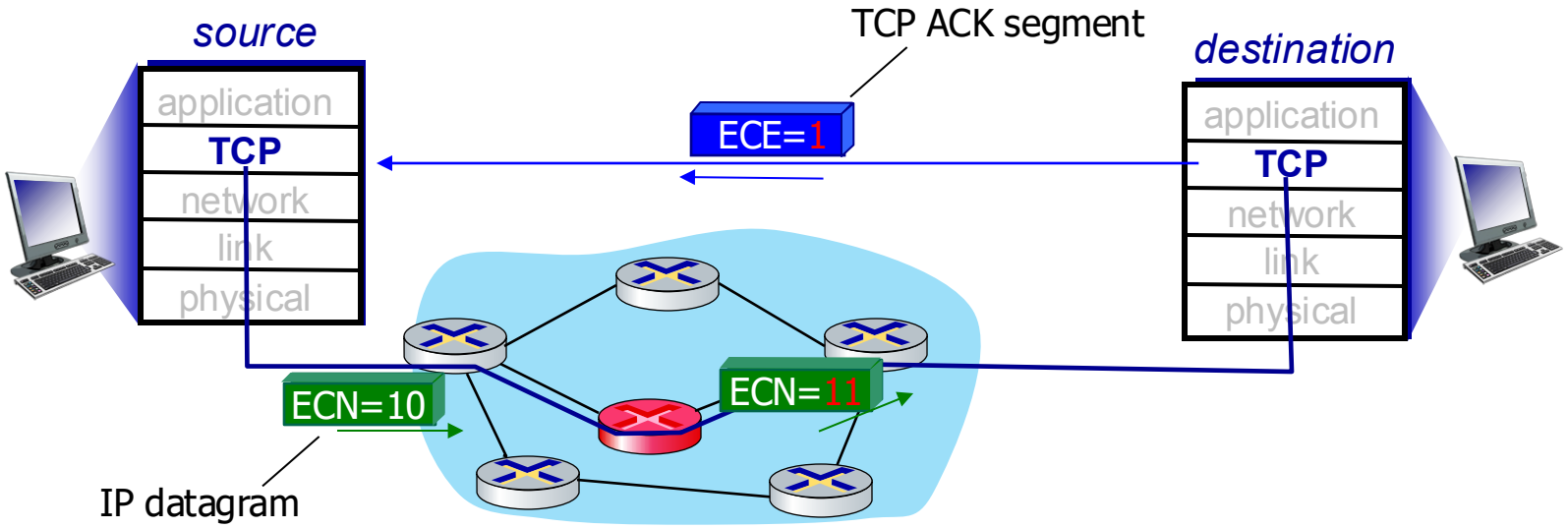
$$\text{Measured throughput} = \frac{\text{Bytes sent in last RTT interval}}{RTT_{measured}}$$

Delay-based TCP Congestion Control

- Congestion control without inducing or forcing loss
- Maximizing throughout while keeping delay low
- A number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network

ECN: Explicit Congestion Notification

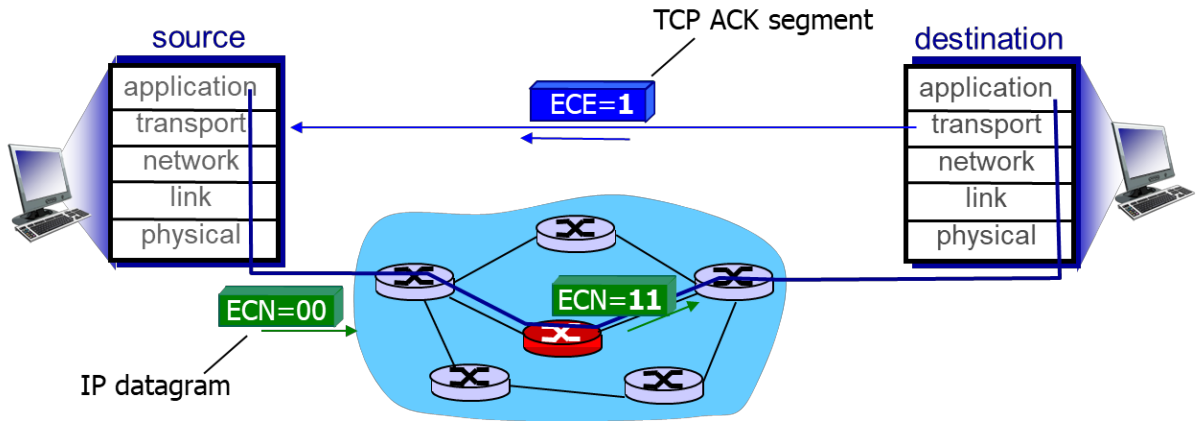
- TCP deployments often implement *network-assisted* congestion control:
  - Two bits in IP header (ToS field) marked *by network router* to indicate congestion
    - *Policy* to determine marking chosen by network operator
  - Congestion indication carried to destination
  - Destination sets ECE bit on ACK segment to notify sender of congestion
  - Involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



Explicit Congestion Notification

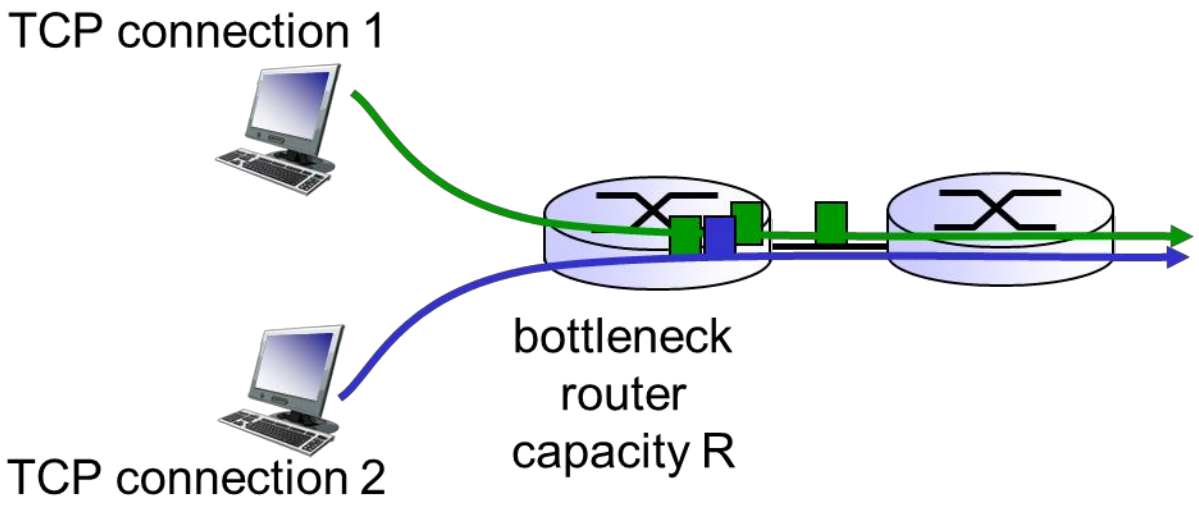
Network-assisted congestion control:

- Two bits in IP header (ToS field) marked **by network router** to indicate congestion
- Congestion indication carried to receiving host
- Receiver (seeing congestion indication in IP datagram) sets ECE bit on receiver-to-sender ACK segment to notify sender of congestion.



TCP Fairness

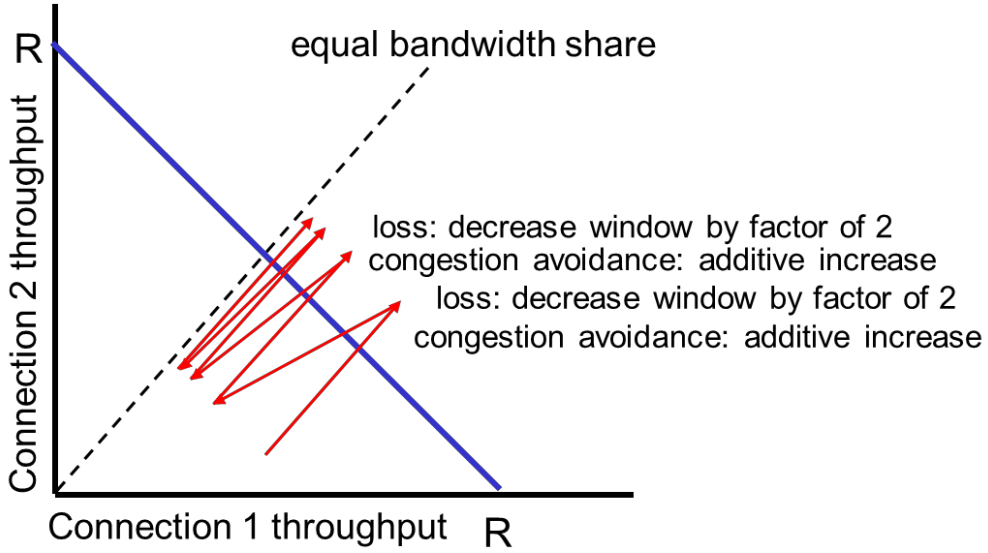
- **Fairness goal***:* If K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



WHY TCP is Fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughput increases
- Multiplicative decrease decreases throughput proportionally



Fairness

Fairness and UDP

- Multimedia applications often do not use TCP: Do not want rate throttled by congestion control
- Instead use UDP: Send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- Application can open multiple parallel connections between two hosts
- Web browsers do this
- Example, link of rate R with 9 existing connections
  - New application asks for 1 TCP, gets rate: R/10
  - New application asks for 11 TCPs, gets: R/2

Evolving Transport-Layer Functionality

- TCP & UDP: Principal transport protocols for 40 years
- Different flavors of TCP developed, for specific scenarios:

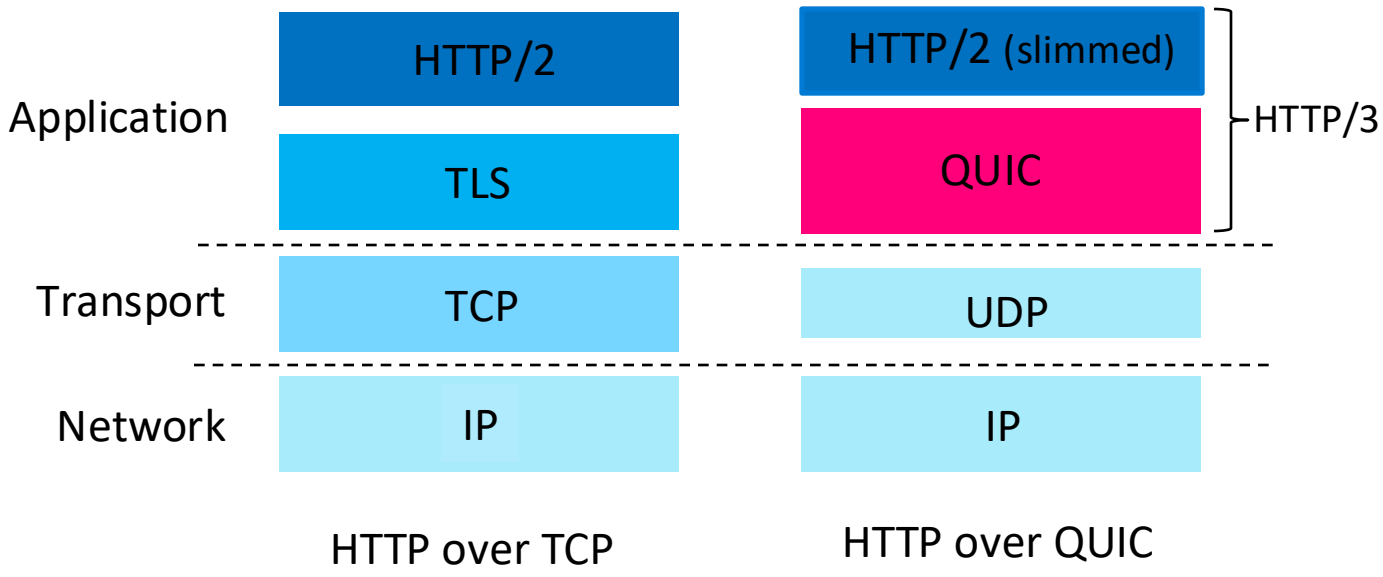
Scenario	Challenges
Long, fat pipes (large data transfers)	*Many packets in flight; loss shuts down pipeline*
Wireless networks	*Loss due to noisy wireless links, mobility; TCP treat this as congestion loss*
Long-delay links	*Extremely long RTTs*
Data center networks	*Latency sensitive*
Background traffic flows	*Low priority, background TCP flows*

- Moving transport–layer functions to the application layer, on top of the UDP
  - HTTP/3: QUIC

QUIC

QUIC: Quick UDP Internet Connections

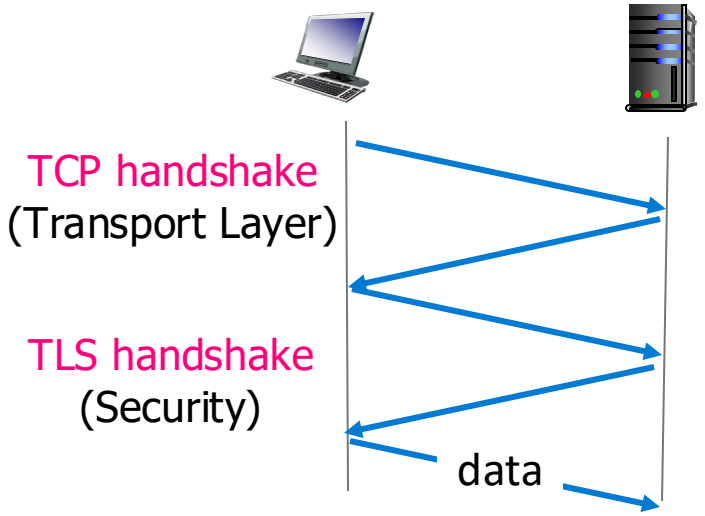
- Motivation
  - Protocol Entrenchment
  - Implementation Entrenchment
  - Handshake Delay
  - Head of Line Blocking Delay
- Services
  - Congestion Control
  - Encryption
  - HTTP move to QUIC
    (Multiplexed Streams, Server-initialized Streams, etc.)

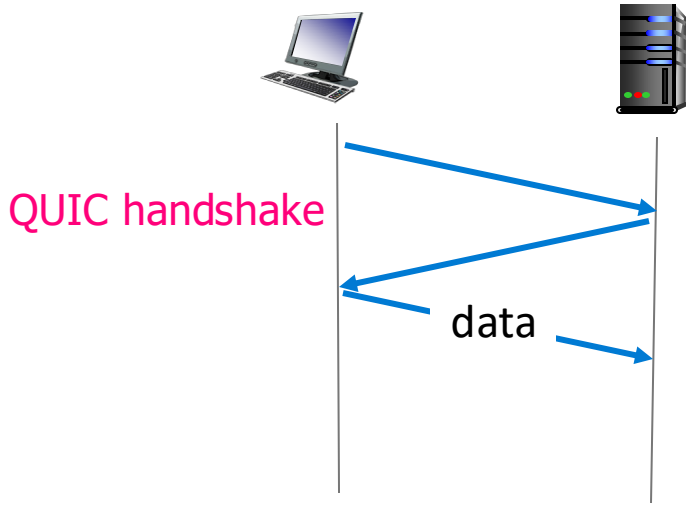


QUIC

- Similar connection establishment, error control, and congestion control approaches
- **Error and congestion control:** “Readers familiar with TCP’s loss detection and congestion control will find algorithms here that parallel well-known TCP ones.” [from QUIC specification]
- **Connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT
- Multiple application-level streams multiplexed over single QUIC connection
  - Separate reliable data transfer & security
  - Common congestion control

QUIC: Connection Establishment





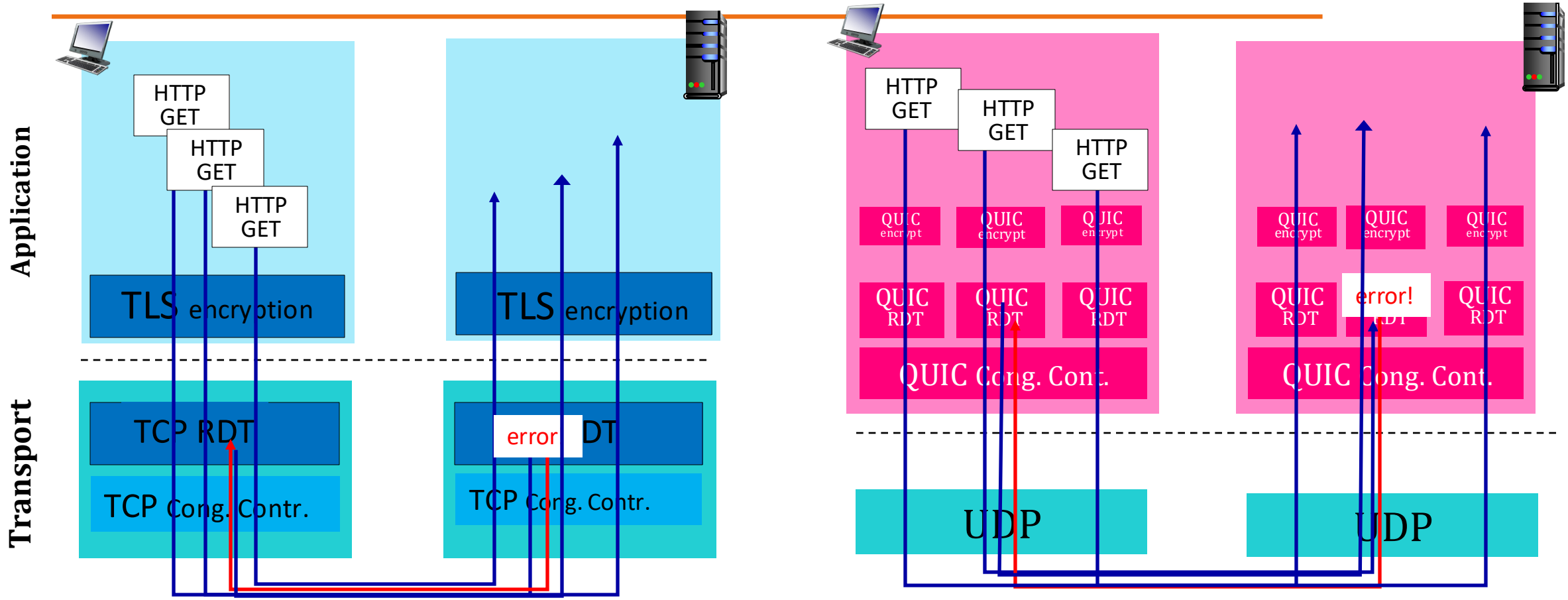
TCP (Reliability, Congestion Control State)
+ TLS (Authentication, Crypto State)

2 Serial Handshakes

QUIC: Reliability, Congestion Control, Authentication, Crypto State

1 Handshake

QUIC: Streams: Parallelism, No HOL Blocking



(a) HTTP 1.1

(b) HTTP/2 with QUIC: no HOL blocking

Summary

- Multiplexing and Demultiplexing
- Connectionless Transport :UDP
- Reliable Data Transfer
- Connection-oriented Transport: TCP
  - Segment Structure
  - Reliable Data Transfer
  - Flow Control
  - Connection Management
- Congestion Control
- TCP Congestion Control
- QUIC

Acknowledgements

- The following materials have been used in preparation of this presentation:

  [1] Textbook and (edited) Slides: Computer Networking: A Top-Down Approach
  James Kurose, Keith Ross
  7th and 8th Edition, Pearson
  http://gaia.cs.umass.edu/kurose_ross/

  [2] Reference: Computer Networks: A Systems Approach
  https://www.systemsapproach.org/book.html

- Recommended Additional Resources:

  [1] Interactive Exercises (Chapter Three)
  http://gaia.cs.umass.edu/kurose_ross/interactive/