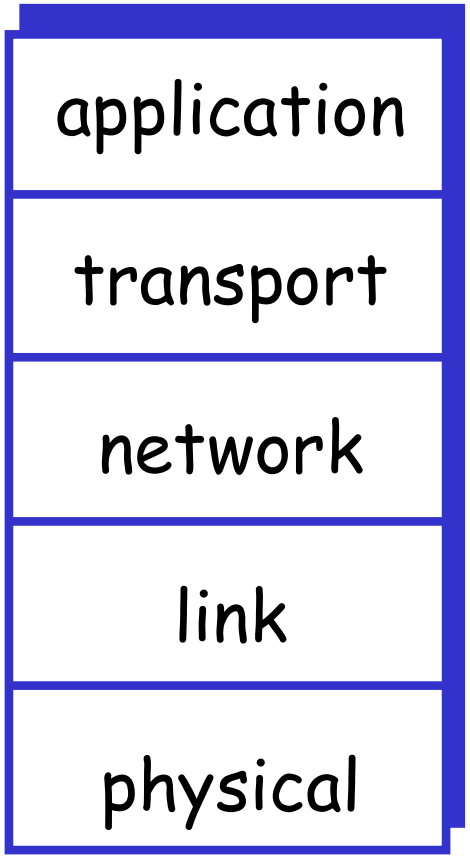
Internet Protocol Stack

- Application: supporting network applications
  - FTP, SMTP, HTTP
- Transport: host-host data transfer
  - TCP, UDP
- Network: routing of datagrams from source to destination
  - IP, routing protocols
- Link: data transfer between neighboring network elements
  - WiFi, Ethernet
- Physical: bits "on the wire"
  - Radios, coaxial cable, optical fibers



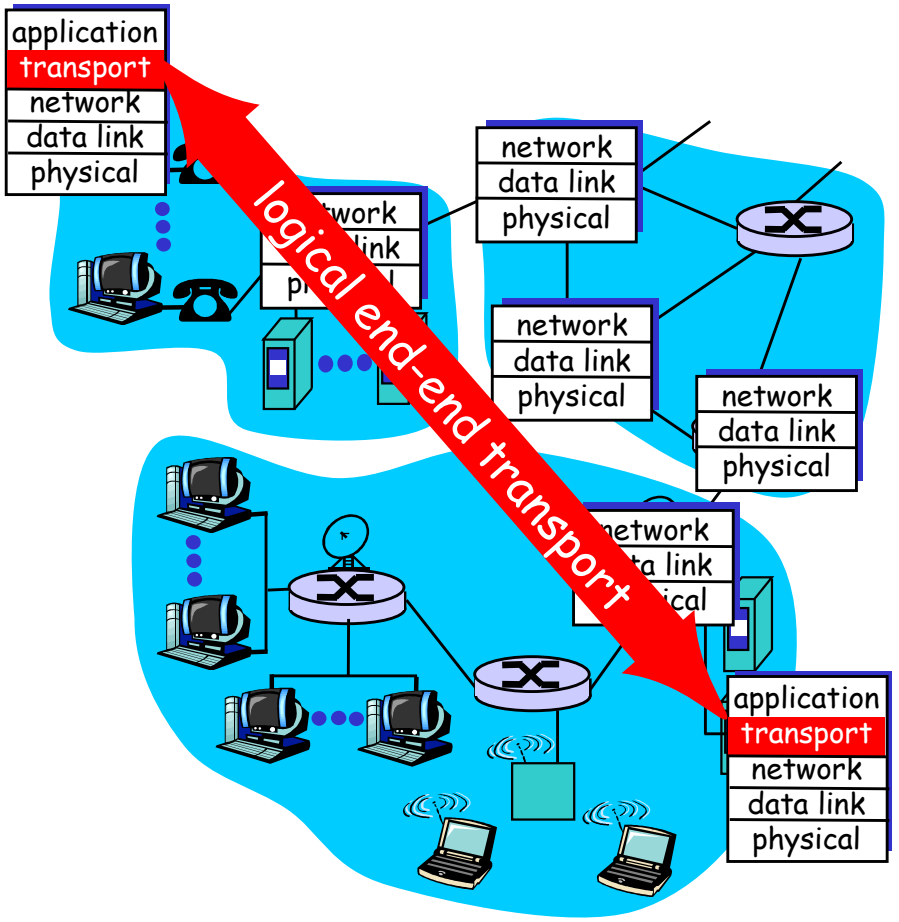
Transport Layer

Roadmap

- Overview transport layer
- Wireless TCP to address the challenges
  - Unreliable links
  - Node mobility

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

- *network layer*: logical communication between hosts
- *transport layer*: logical communication between processes
  - relies on and enhances network layer services

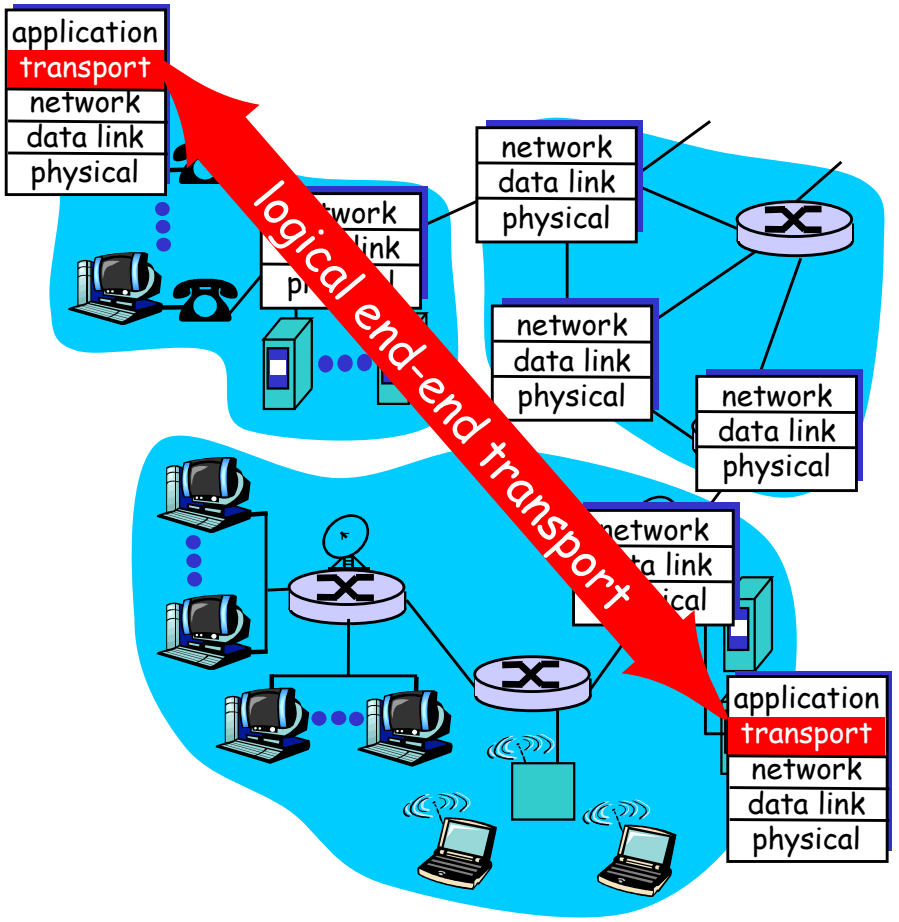
Household analogy:

12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

Internet transport-layer protocols

- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- services not available:
  - delay guarantees
  - bandwidth guarantees



Multiplexing/demultiplexing

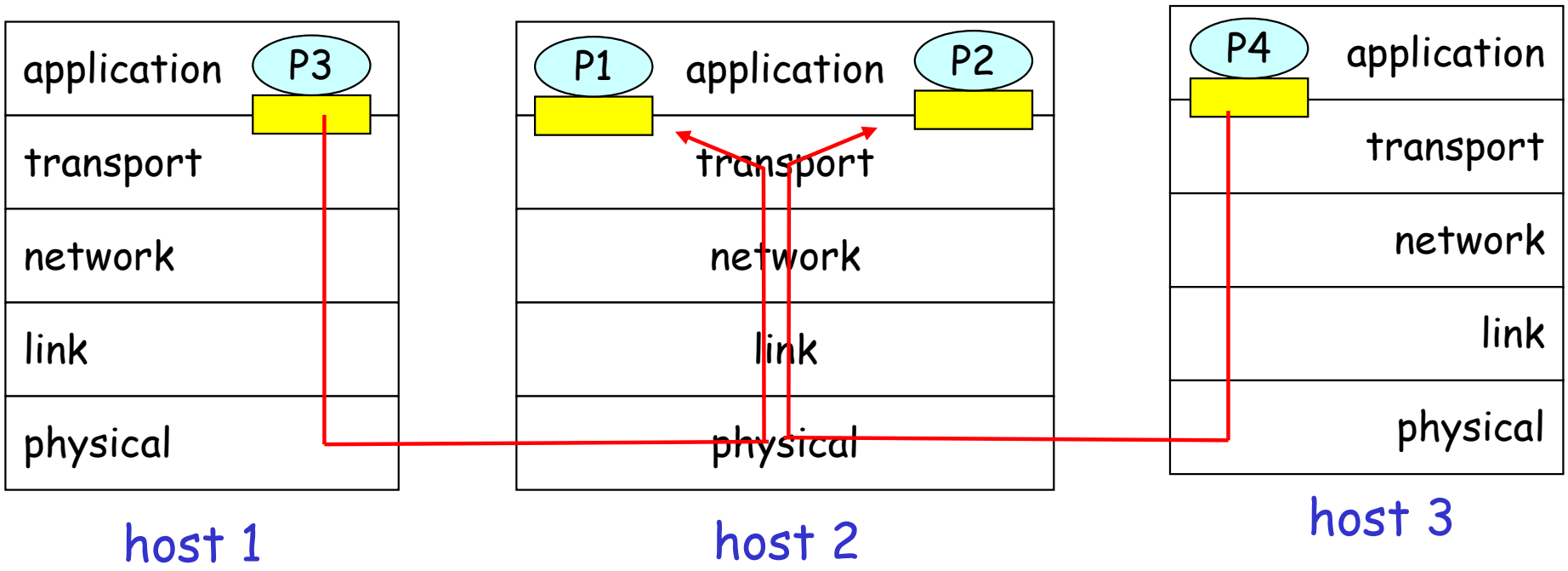
Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

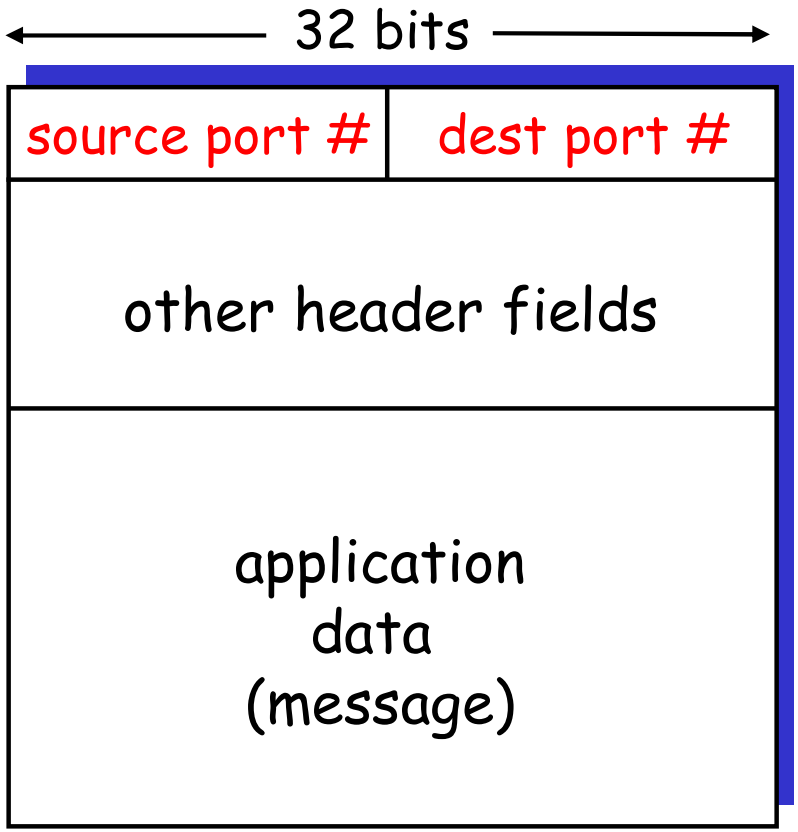
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

= socket = process



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(12534);
DatagramSocket mySocket2 = new
   DatagramSocket(12535);
```

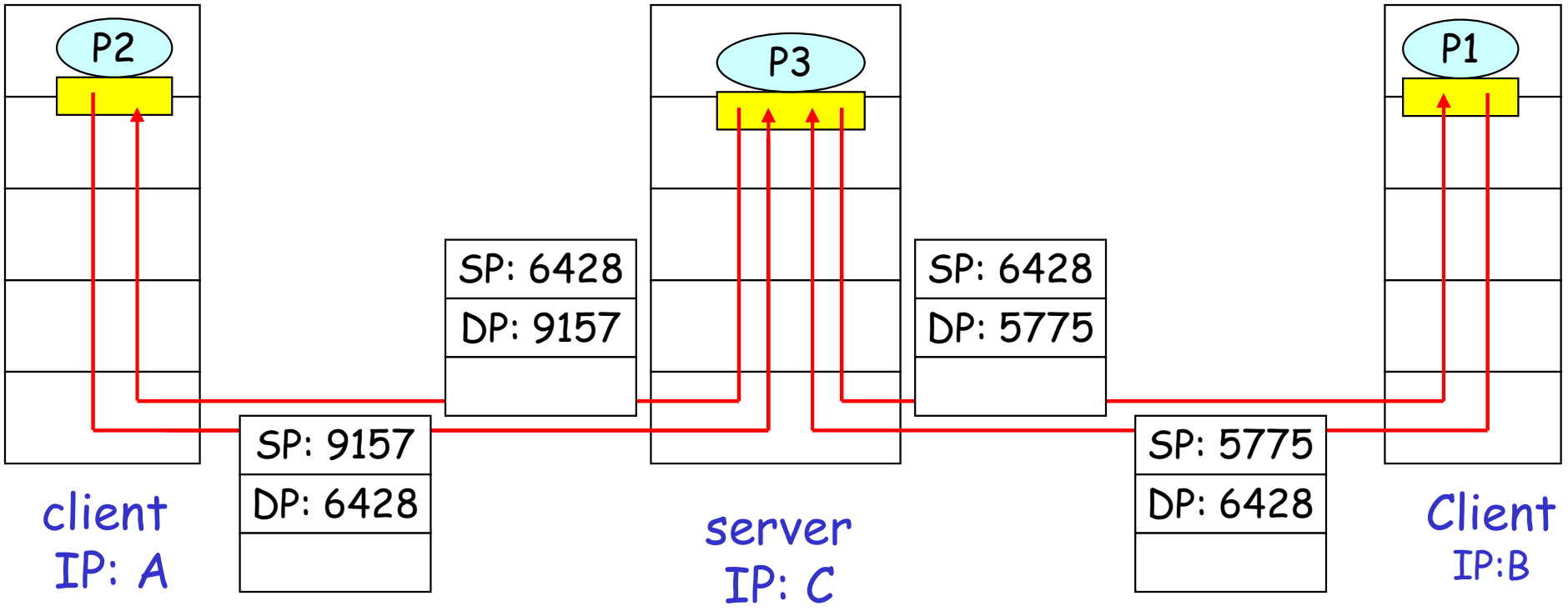
- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux (cont)

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



SP provides "return address"

Send datagram

```
DatagramSocket datagramSocket = new DatagramSocket();
byte[] buffer = "0123456789".getBytes();
InetAddress receiverAddress = InetAddress.getLocalHost();
DatagramPacket packet = new DatagramPacket( buffer, buffer.length,
                                            receiverAddress, 80);
datagramSocket.send(packet);
```

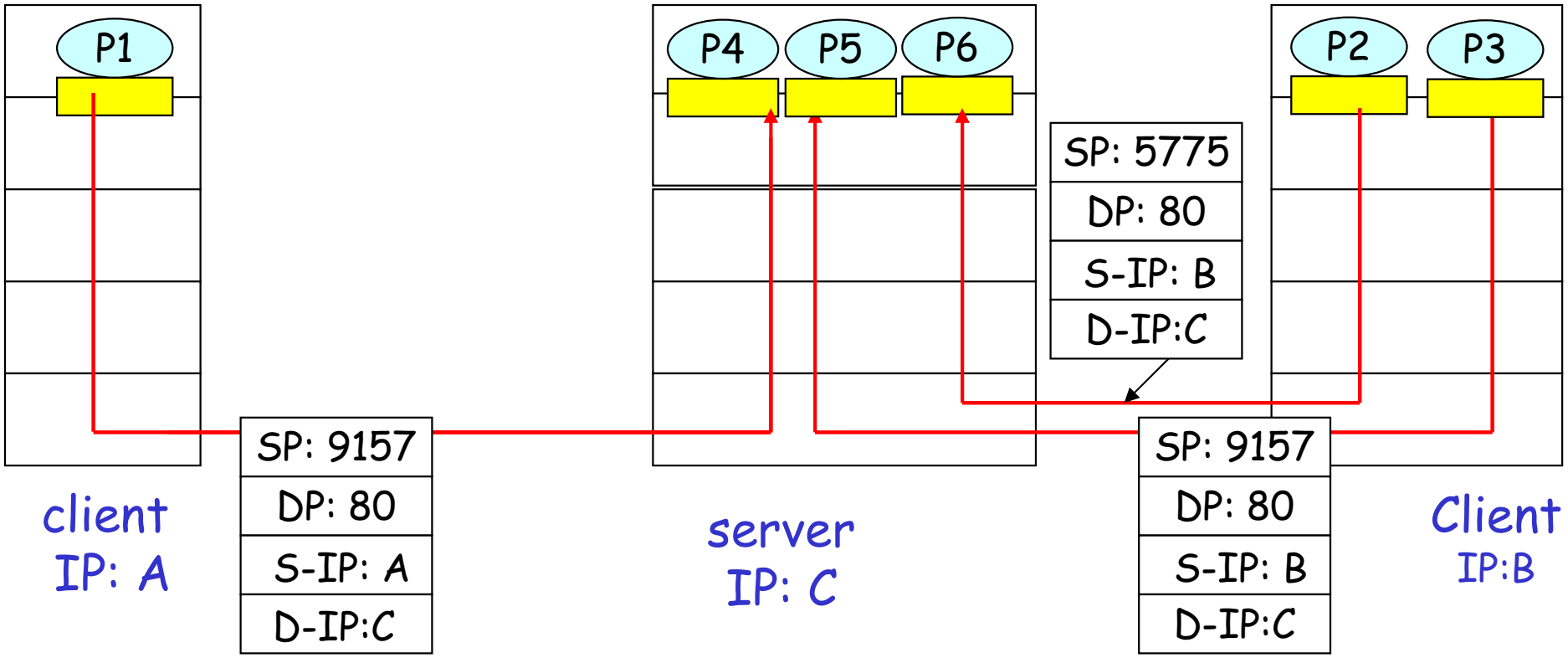
Receive datagram

```
DatagramSocket datagramSocket = new DatagramSocket(80);
byte[] buffer = new byte[10];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
datagramSocket.receive(packet);
```

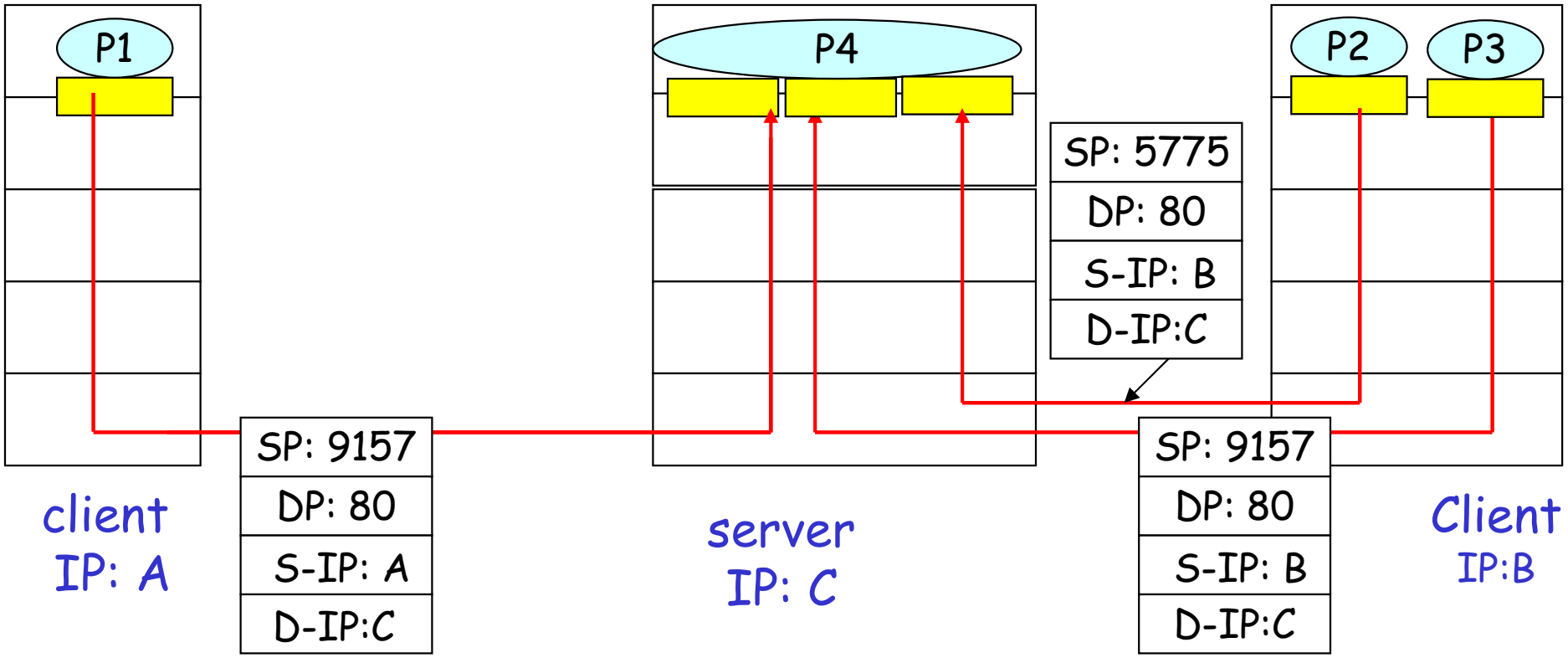
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- receiving host uses all four values to 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 (cont)



Connection-oriented demux: Threaded Web Server



Client side

- Create a socket with the socket() system call
- Connect the socket to the address of the server using the connect() system call
- Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

Server side

- Create a socket with the socket() system call
- Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the listen() system call
- Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
- Send and receive data

TCP Server

```
try (
ServerSocket serverSocket = new ServerSocket(portNumber);
Socket clientSocket = serverSocket.accept();
PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
BufferedReader in = new BufferedReader( new InputStreamReader(clientSocket.getInputStream())); )
```

TCP Client

```
try (
Socket kkSocket = new Socket(hostName, portNumber);
PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
BufferedReader in = new BufferedReader( new InputStreamReader(kkSocket.getInputStream())); )
```

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

UDP: User Datagram Protocol [RFC 768]

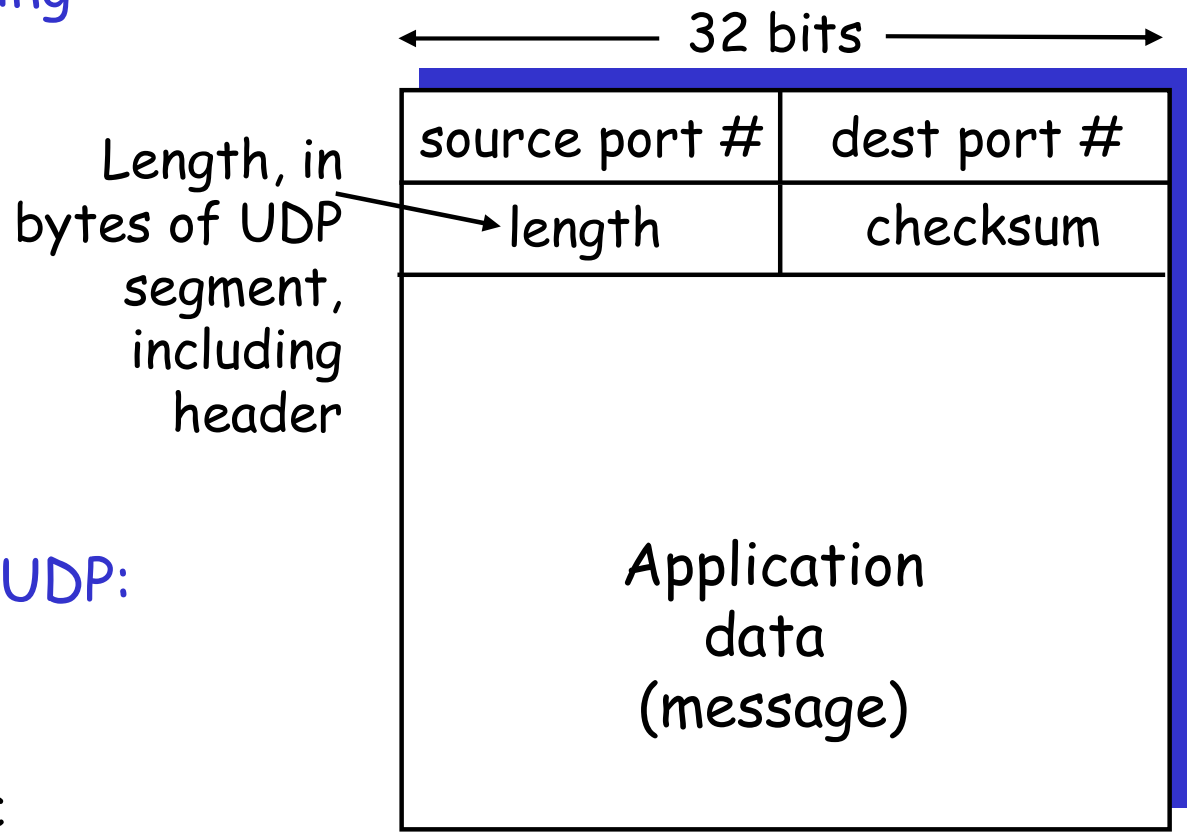
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!



UDP segment format

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- treat segment content as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

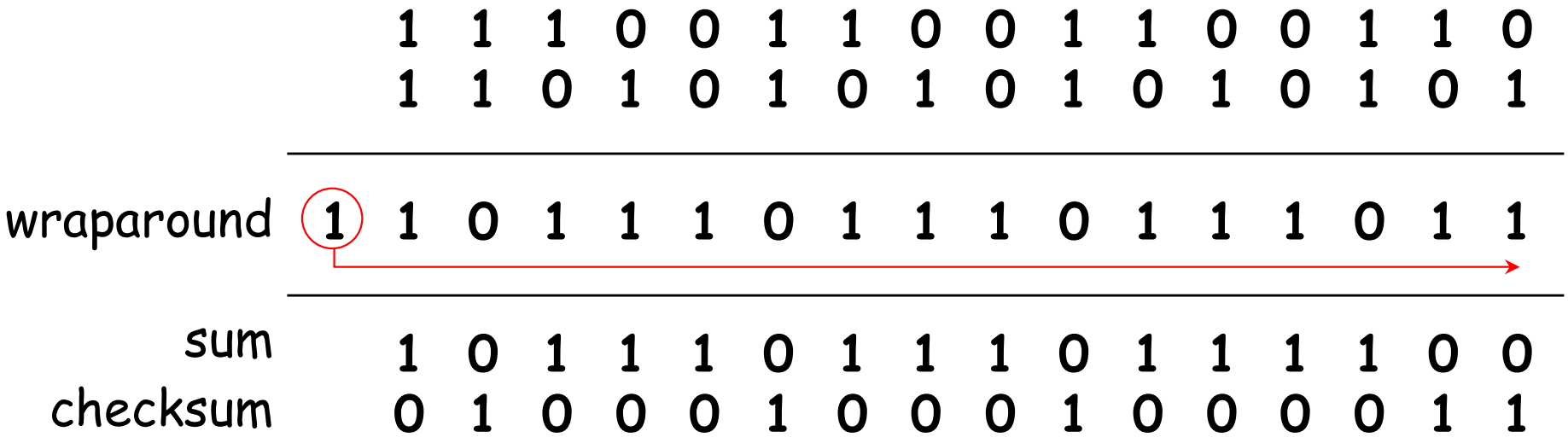
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ....

Internet Checksum Example

- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers
  - Take one's complement
  - Add them
  - Take one's complement

Internet Checksum Example (Cont.)

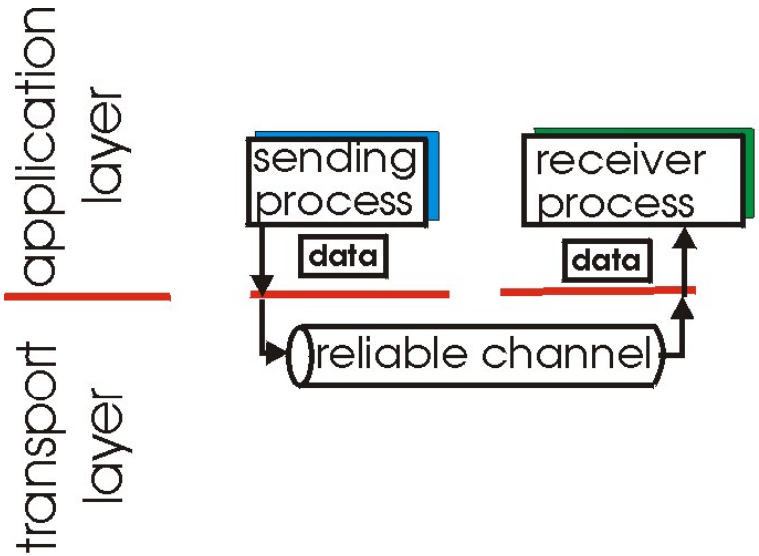
- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers



Benefits: easy to compute; can do incremental update; endian-independent

Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

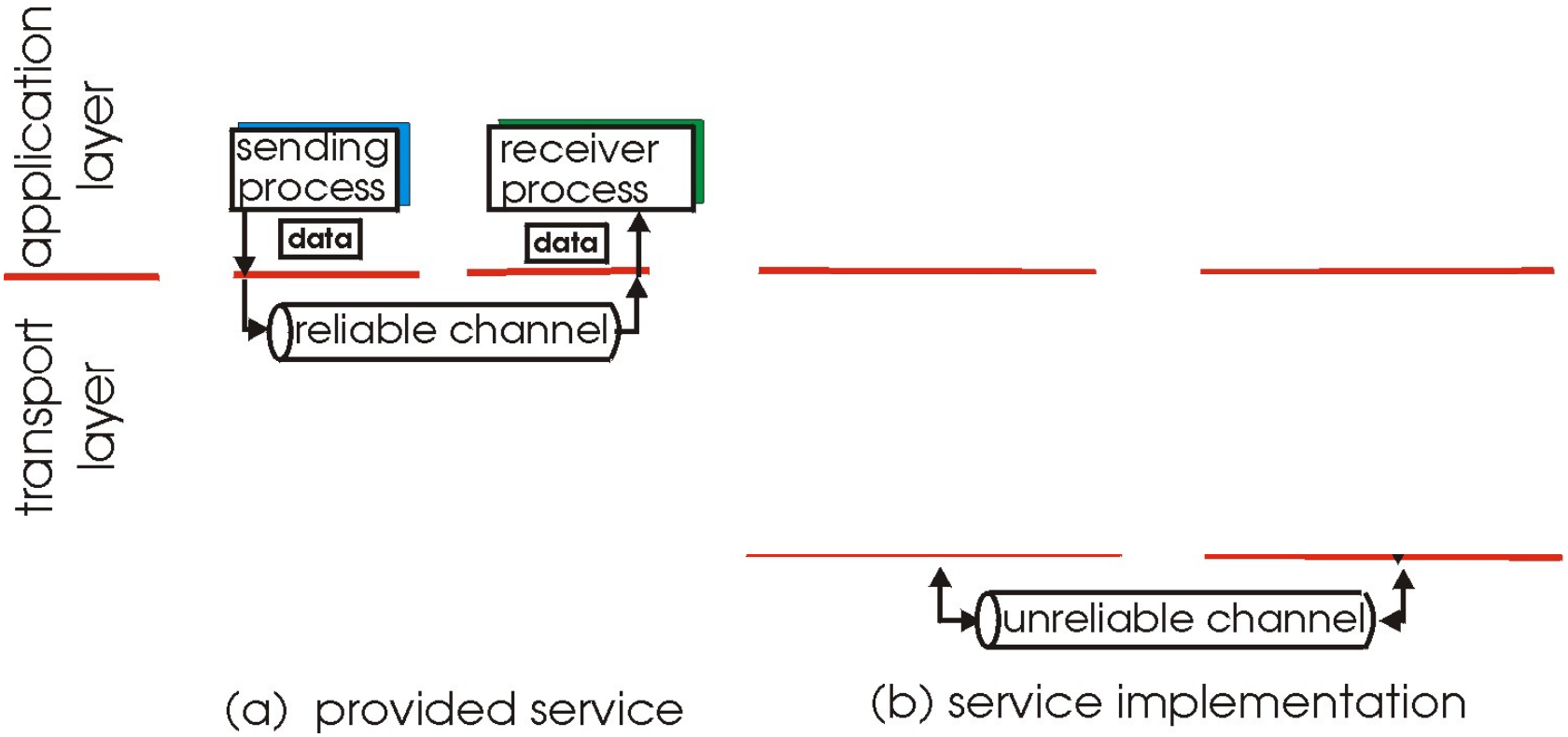


(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable Data Transfer

Reliable data transfer over a reliable channel

- over a reliable channel

Reliable Data Transfer

Reliable data transfer over a reliable channel

- over a reliable channel
- over a channel with error

Reliable Data Transfer

Reliable data transfer over a reliable channel

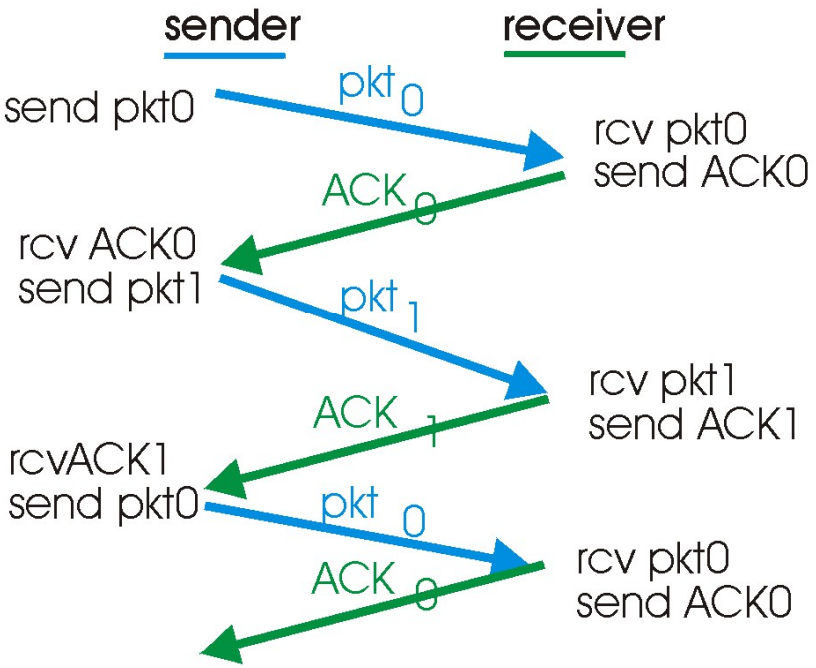
- over a reliable channel
- over a channel with error
  - Checksum + NACK or ACK
- over a channel with error and loss

Reliable Data Transfer

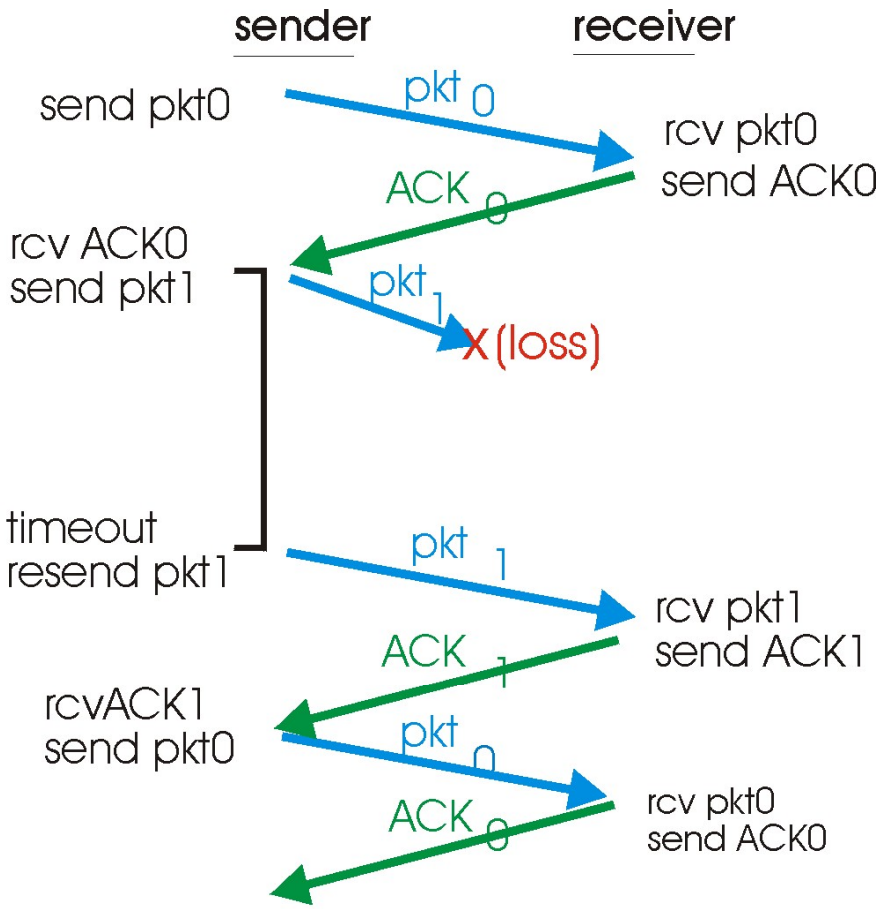
Reliable data transfer over a reliable channel

- over a reliable channel
- over a channel with error
  - Checksum + NACK or ACK
- over a channel with error and loss
  - Checksum + ACK + sequence no. + timeout

Reliable transfer in action

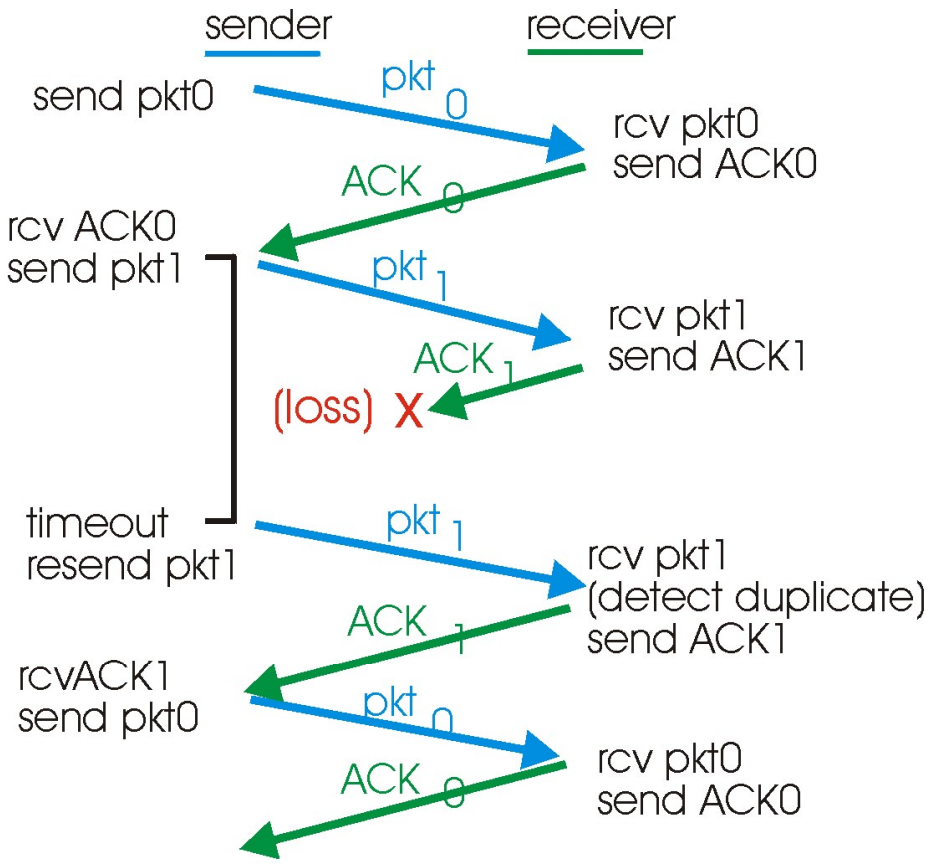


(a) operation with no loss

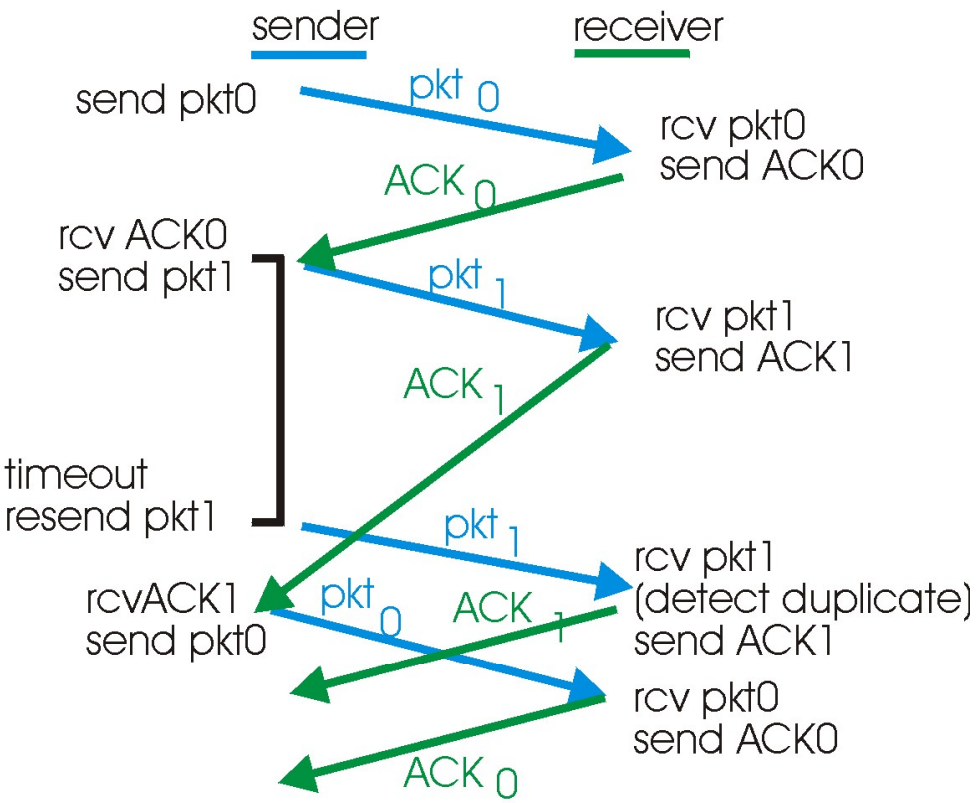


(b) lost packet

Reliable transfer in action



(c) lost ACK



(d) premature timeout

Any problem with the above protocol?

Performance

- It works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

Performance

- It works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

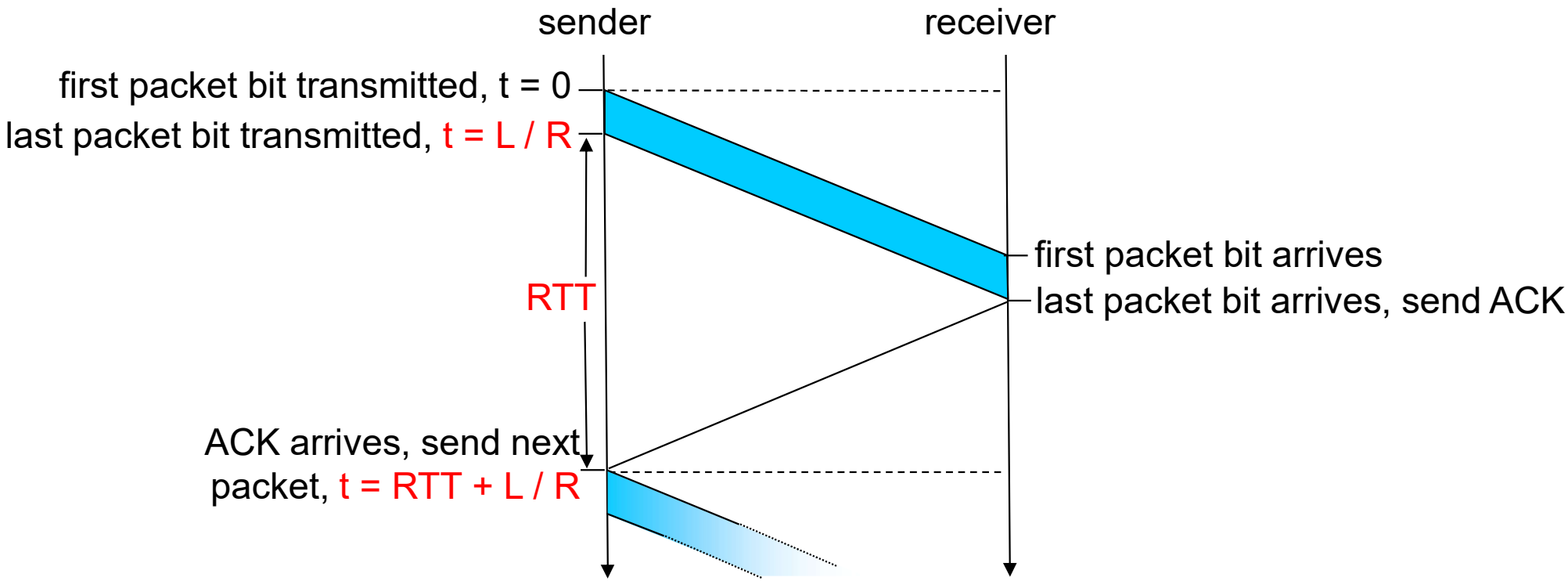
$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{10^9 \text{ b/sec}} = 8 \text{ microsec}$$

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

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

  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

Stop-and-wait operation



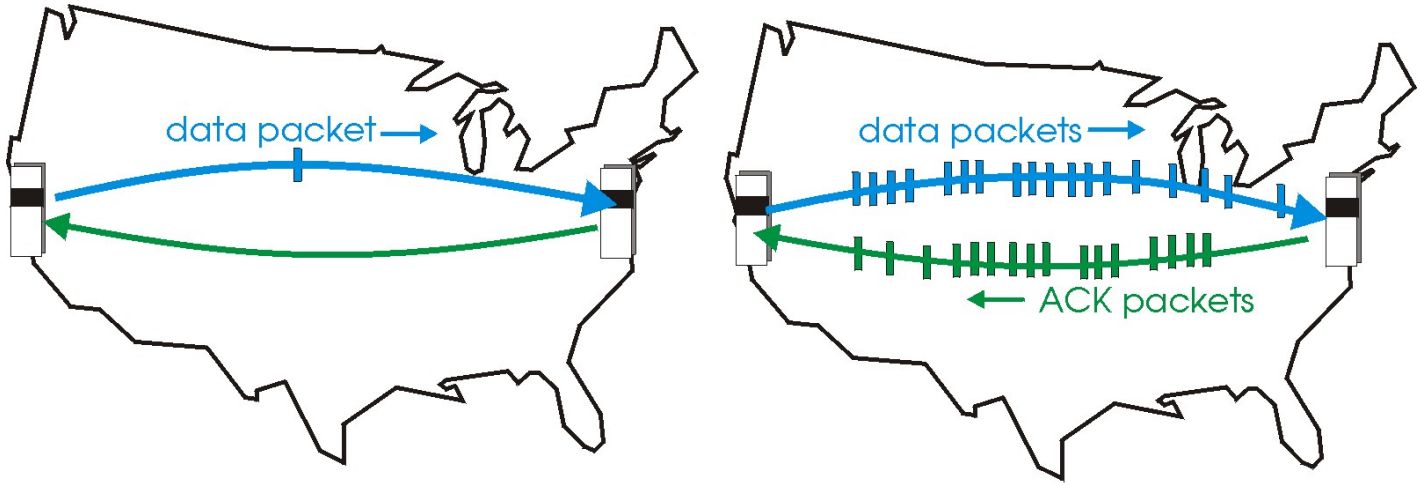
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

How to fix the problem?

Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

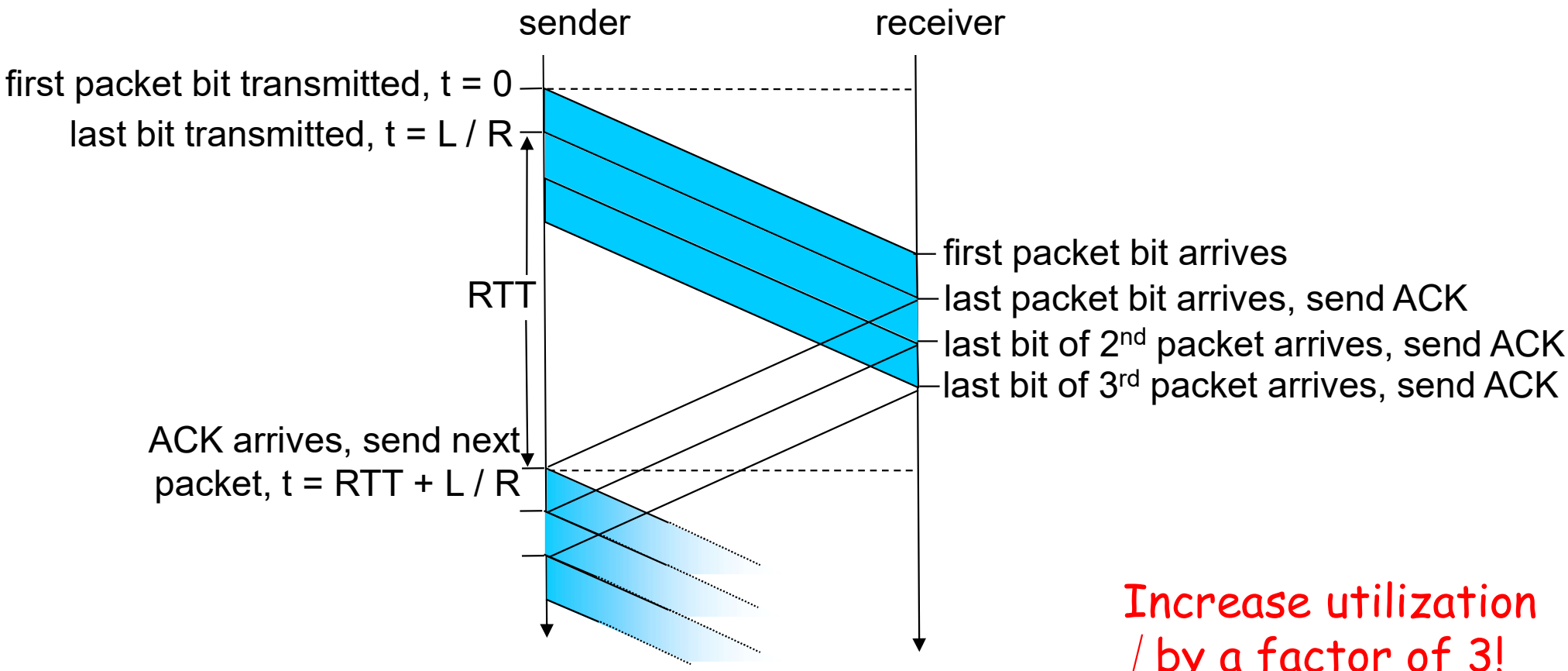


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Pipelining: increased utilization



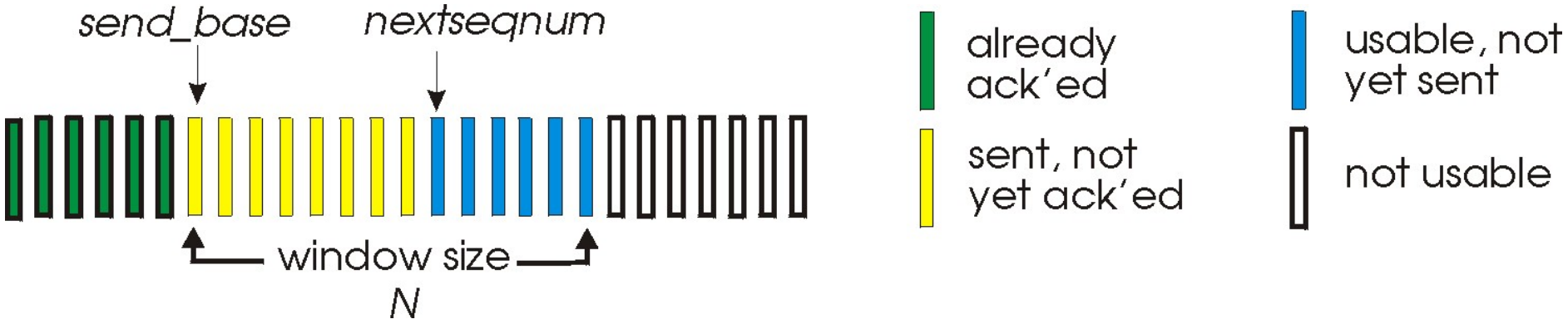
Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N

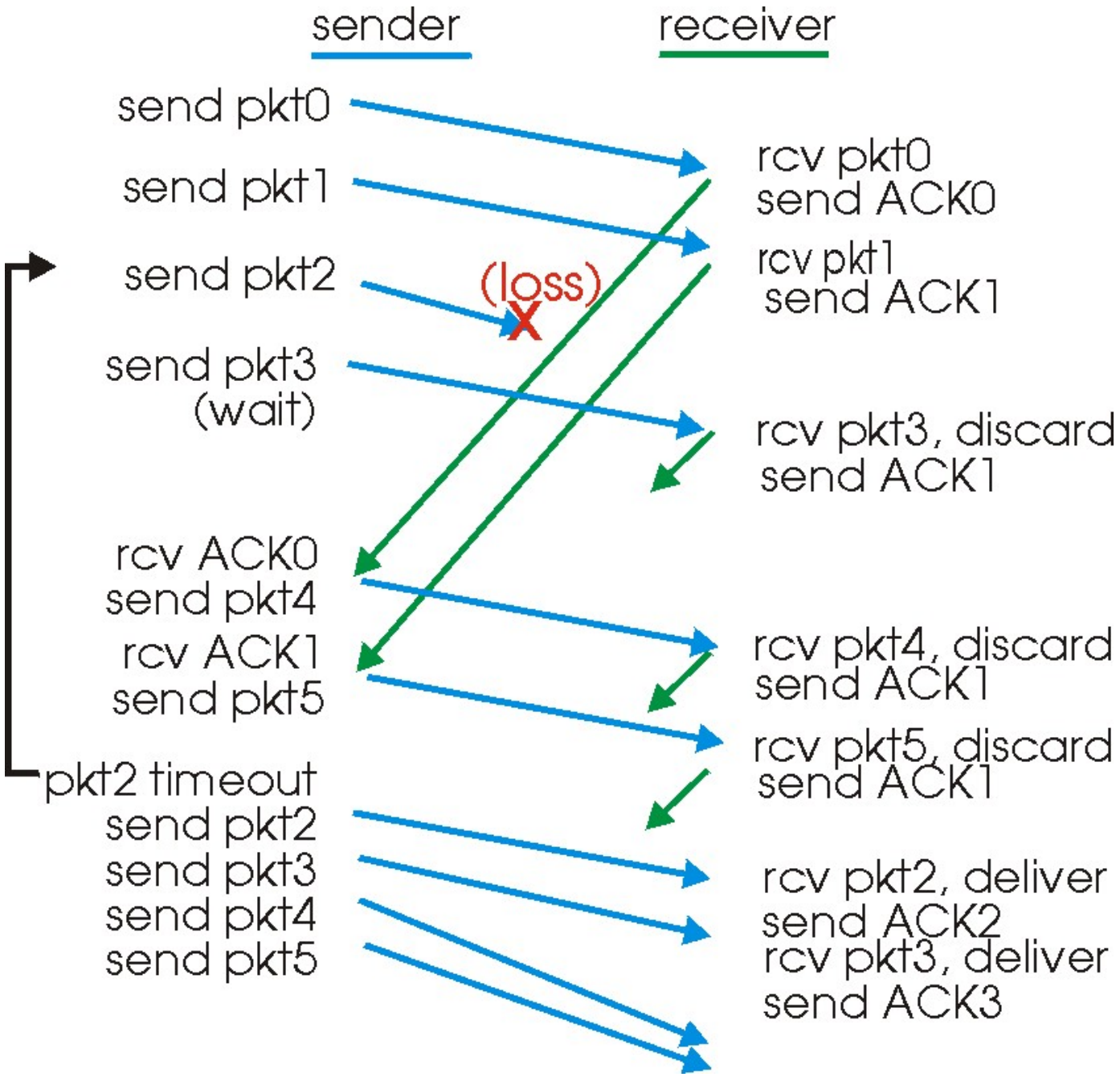
Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n)*: retransmit pkt n and all higher seq # pkts in window

GBN in action

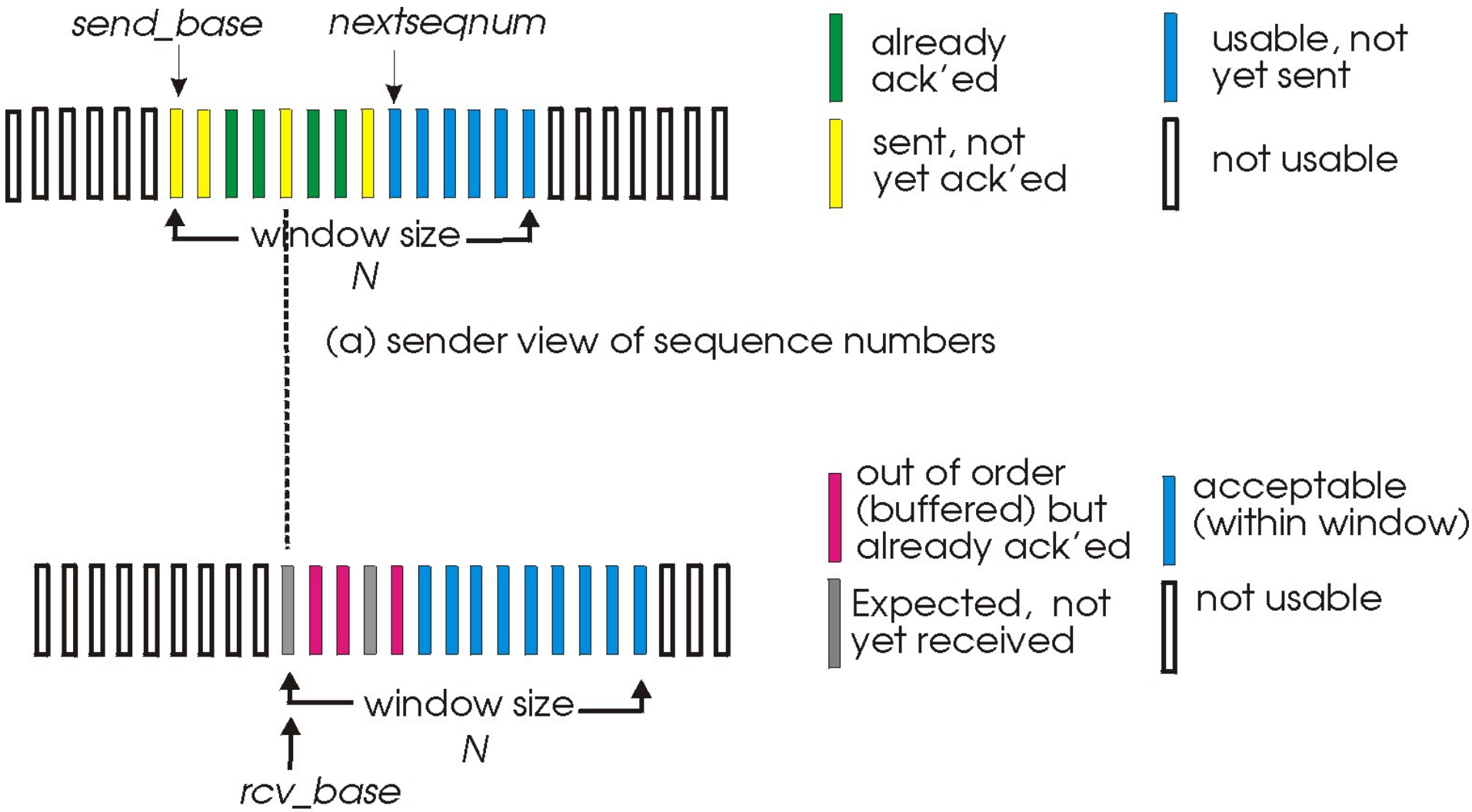


What is the drawback of GBN?
How to eliminate that?

Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above :

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

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

- mark pkt n as received
- if n is smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

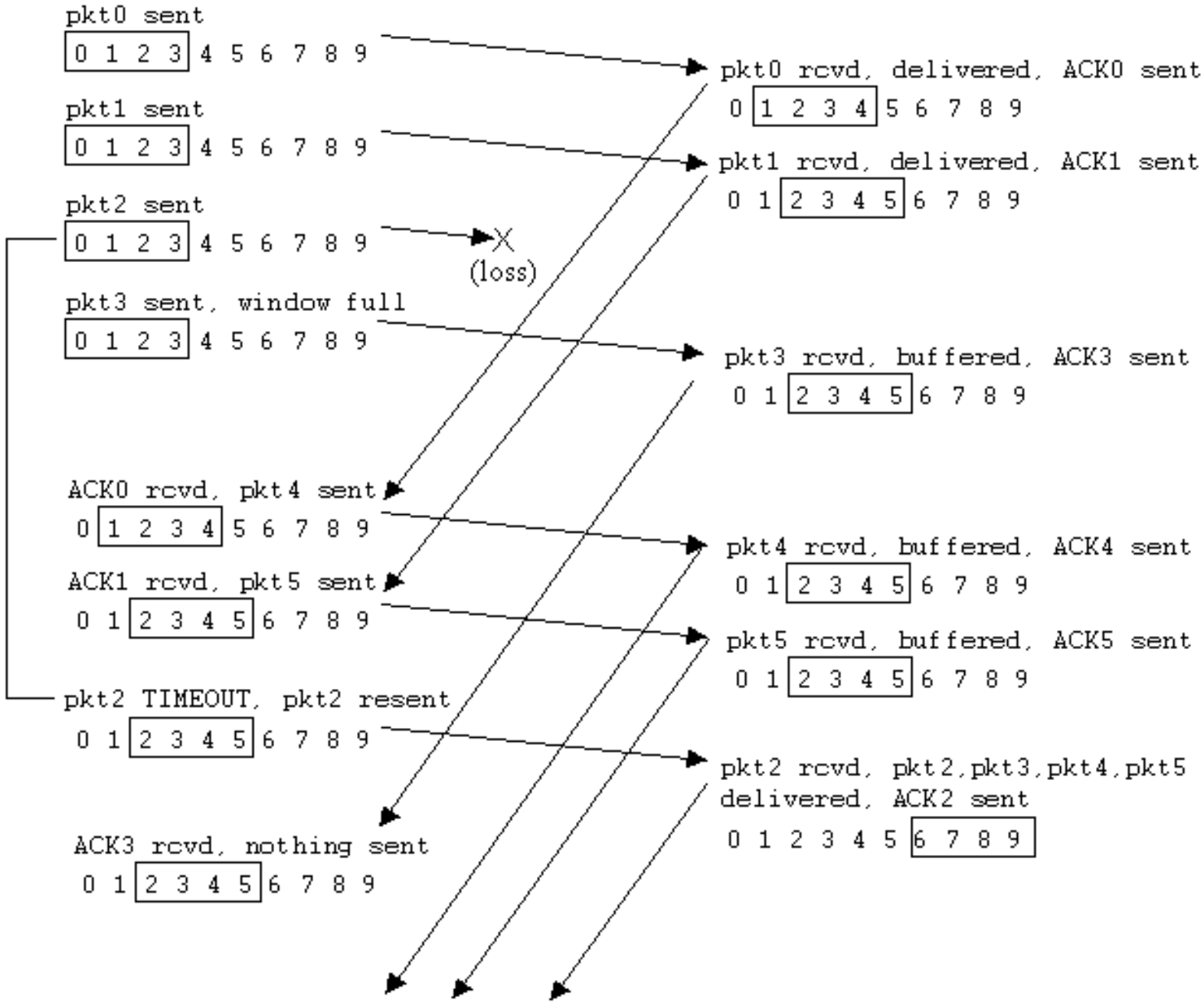
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



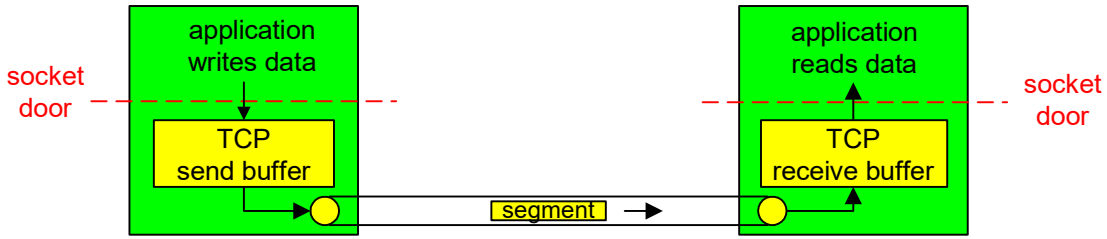
Reliable Data Transfer Mechanisms

Mechanism	Details
Checksum	Detect bit errors
Timer	Detect packet loss at sender
Sequence number	Detect packet loss and duplicates at receiver
ACK	Inform sender that pkt has been received
NACK	Inform sender that pkt has not been received correctly
Window, pipelining	Increase throughput, and adapt to receiver buffer size and network congestion

TCP: Overview

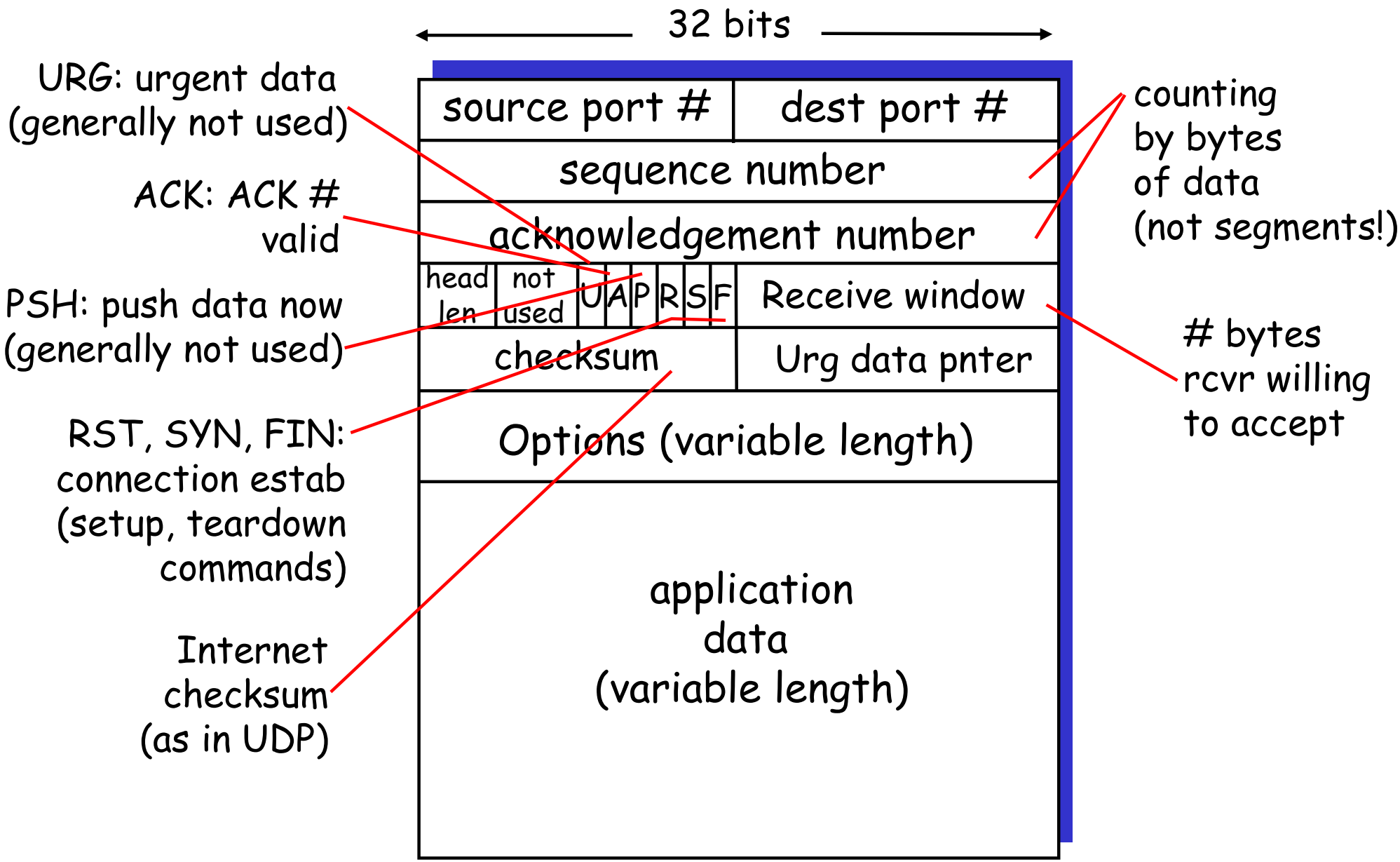
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*



- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **RcvWindow**)
- *client*: connection initiator

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

- *server*: contacted by client

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

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management (cont.)

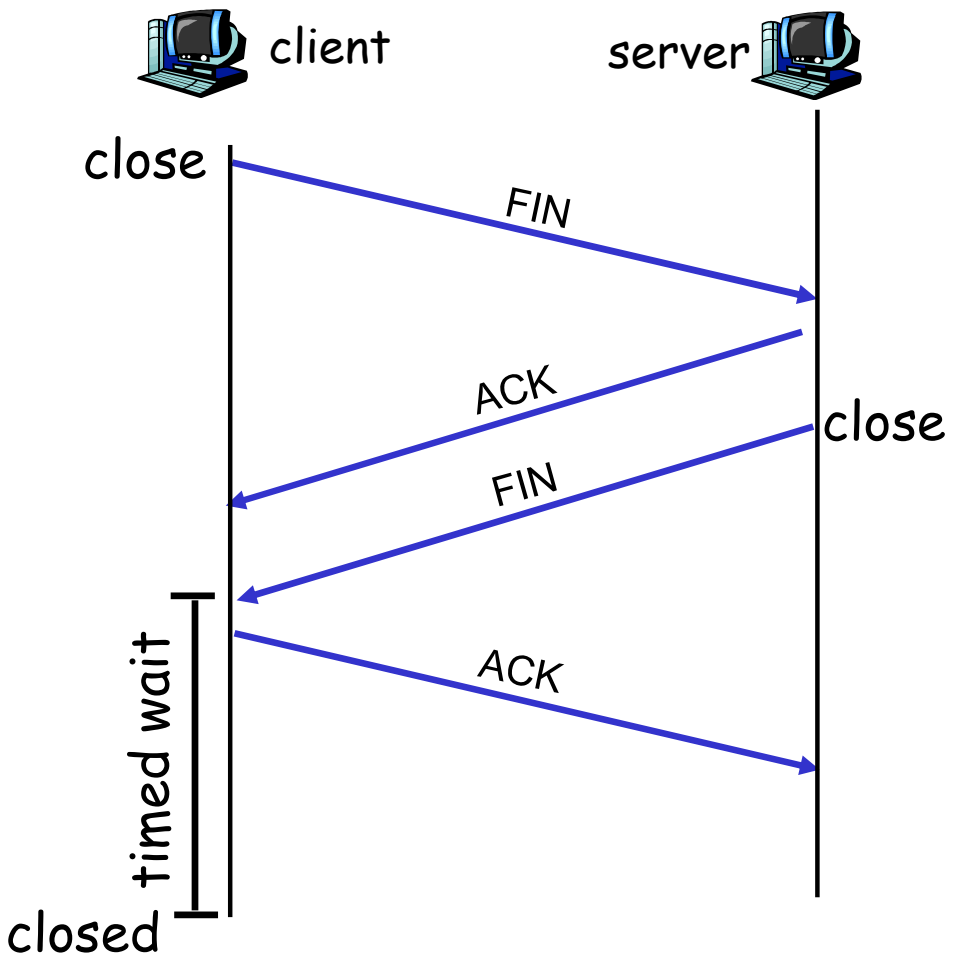
Closing a connection:

client closes socket:

```
clientSocket.close();
```

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



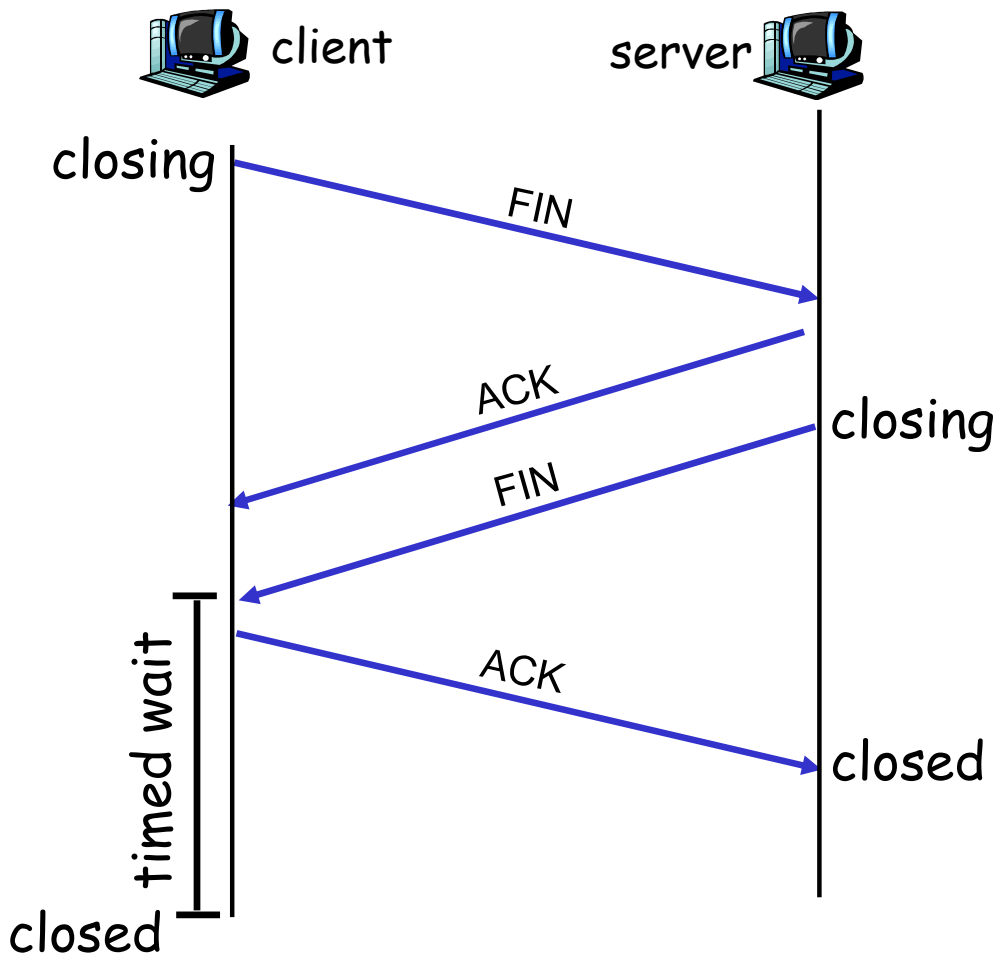
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



TCP seq. #'s and ACKs

Seq. #'s:

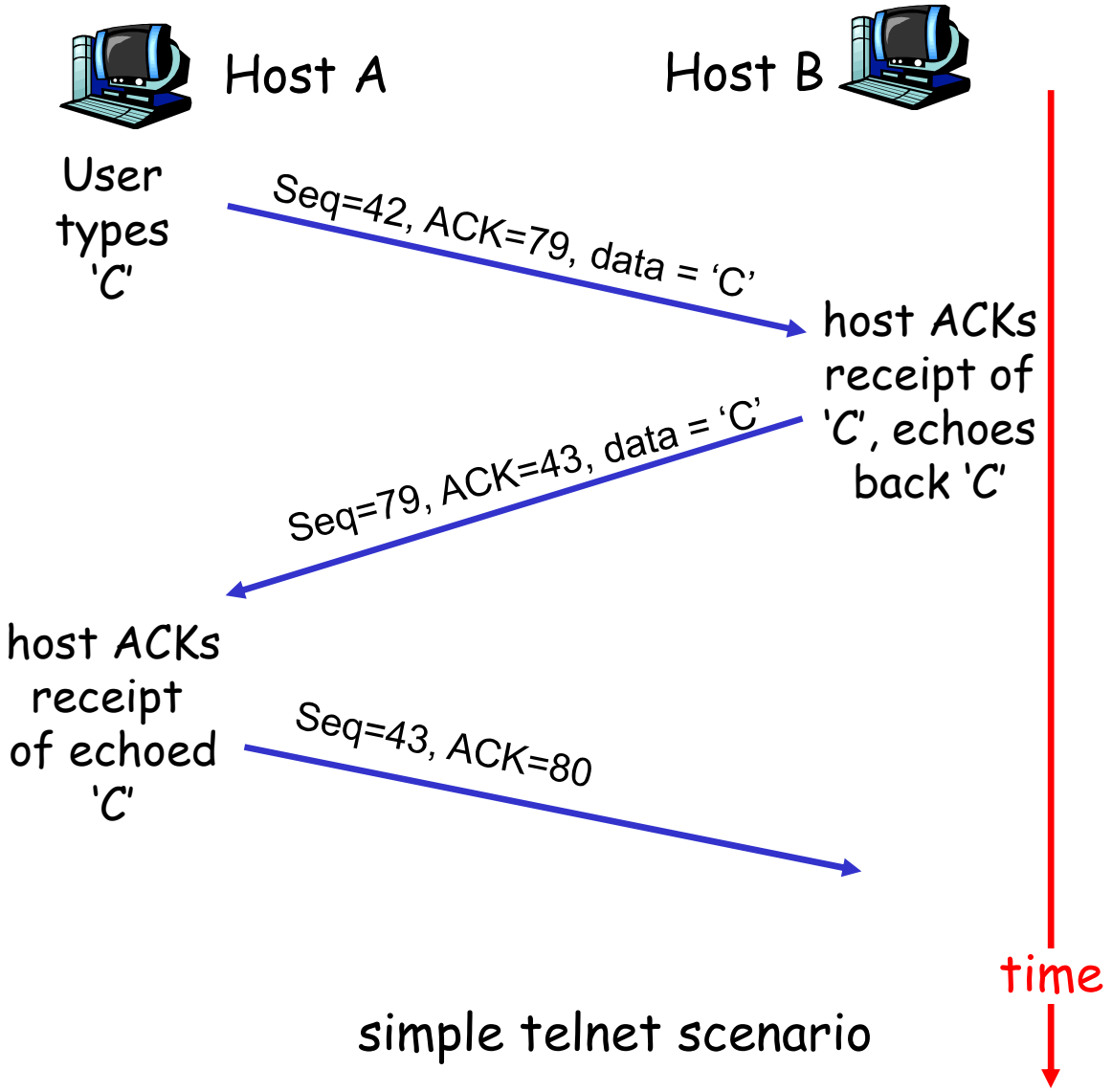
- byte stream "number" of first byte in segment's data

ACKs:

- seq # 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



simple telnet scenario

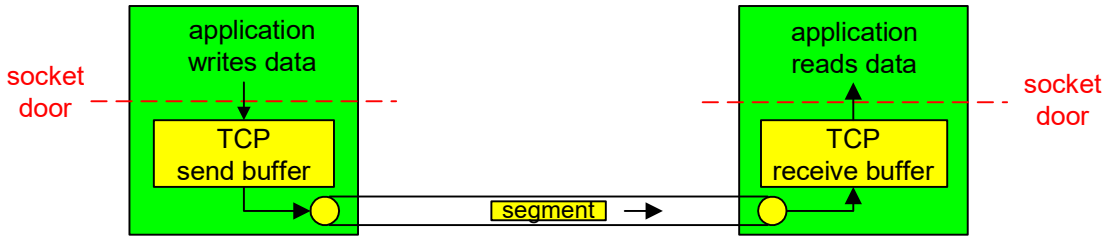
Reliable Data Transfer Mechanisms

Mechanism	Details
Checksum	Detect bit errors
Timer	Detect packet loss at sender
Sequence number	Detect packet loss and duplicates at receiver
ACK	Inform sender that pkt has been received
NACK	Inform sender that pkt has not been received correctly
Window, pipelining	Increase throughput, and adapt to receiver buffer size and network congestion

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*



- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP Timeout

Q: how to set TCP timeout value?

TCP Round Trip Time and 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?

TCP Round Trip Time and 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?

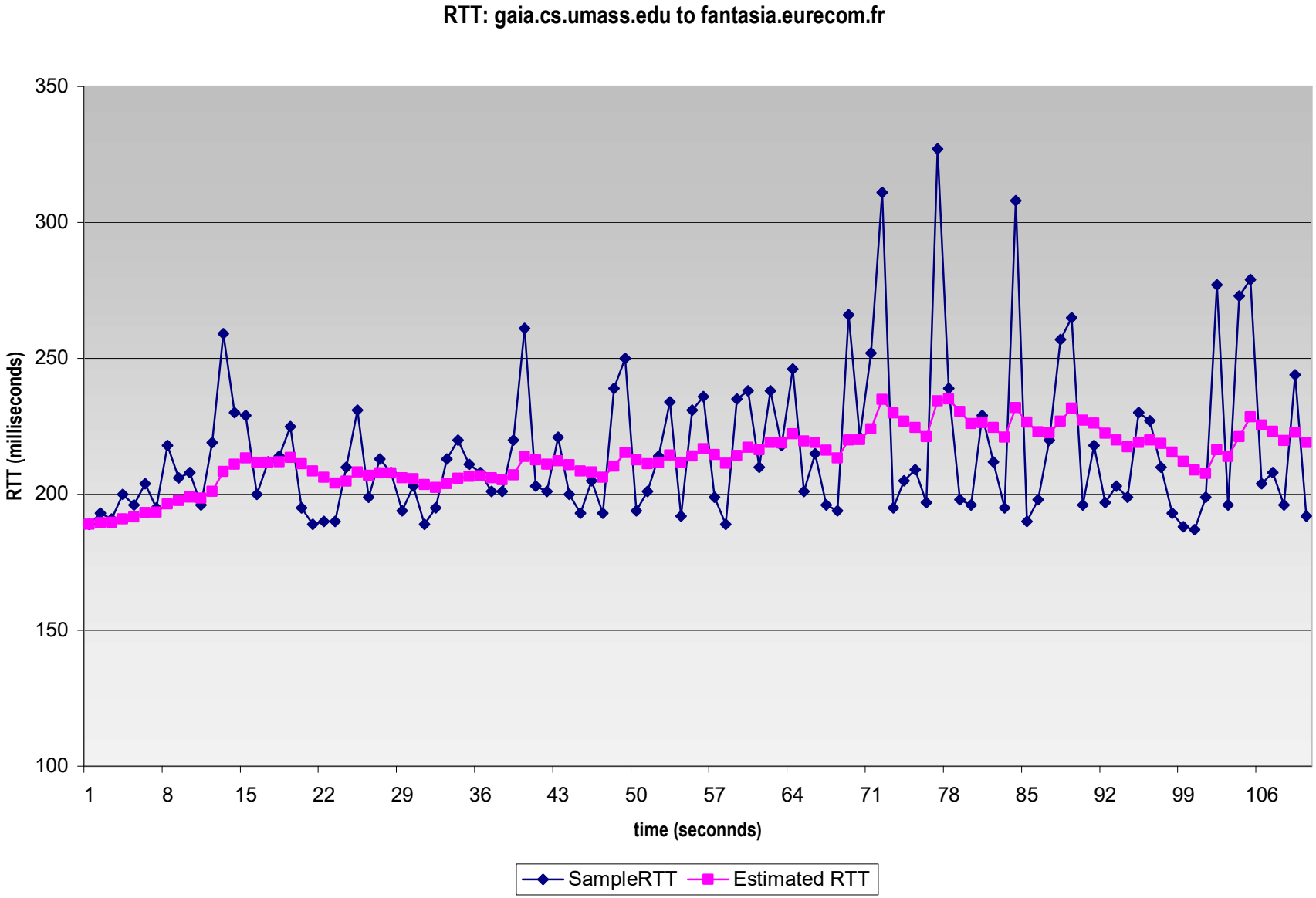
- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
  - Why?
- **SampleRTT** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

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

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout

- **EstimtedRTT** plus "safety margin"
  - large variation in **EstimatedRTT** -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

$$\text{DevRTT} = (1-\beta)*\text{DevRTT} + \beta*|\text{SampleRTT}-\text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

Then set timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4*\text{DevRTT}$$

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 loop (forever) {
   switch(event)

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

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

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

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

TCP sender (simplified)

Comment:

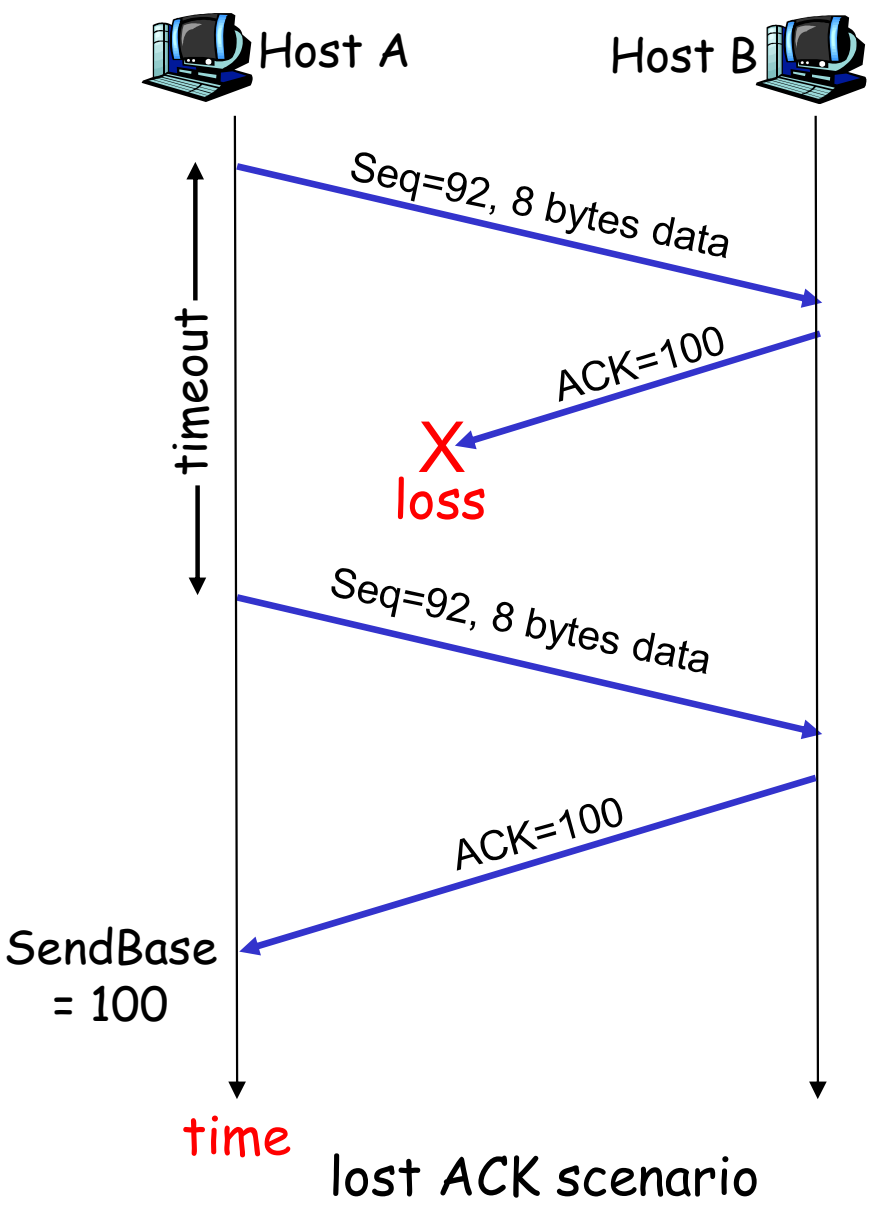
- SendBase-1: last cumulatively ack'ed byte

Example:

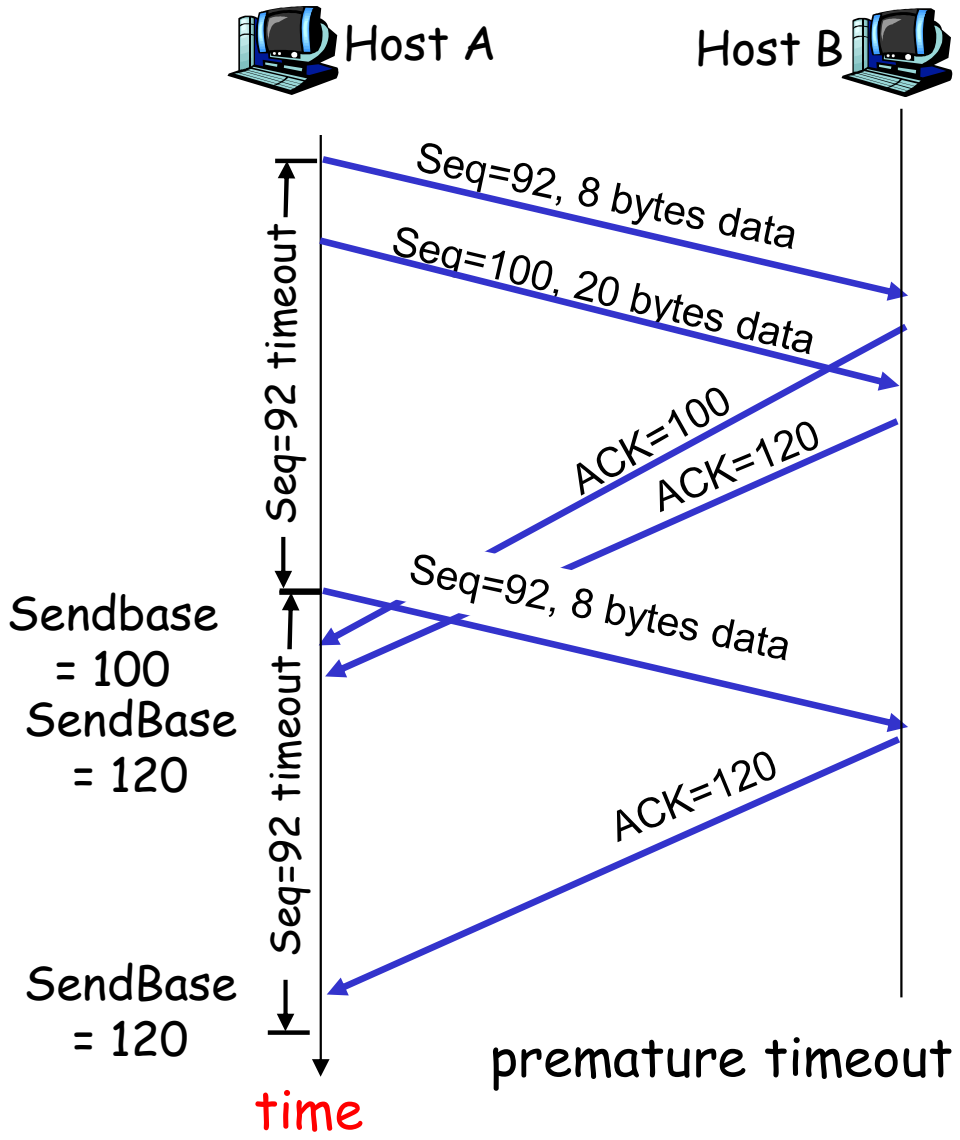
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ;

y > SendBase, so that new data is acked

TCP: retransmission scenarios

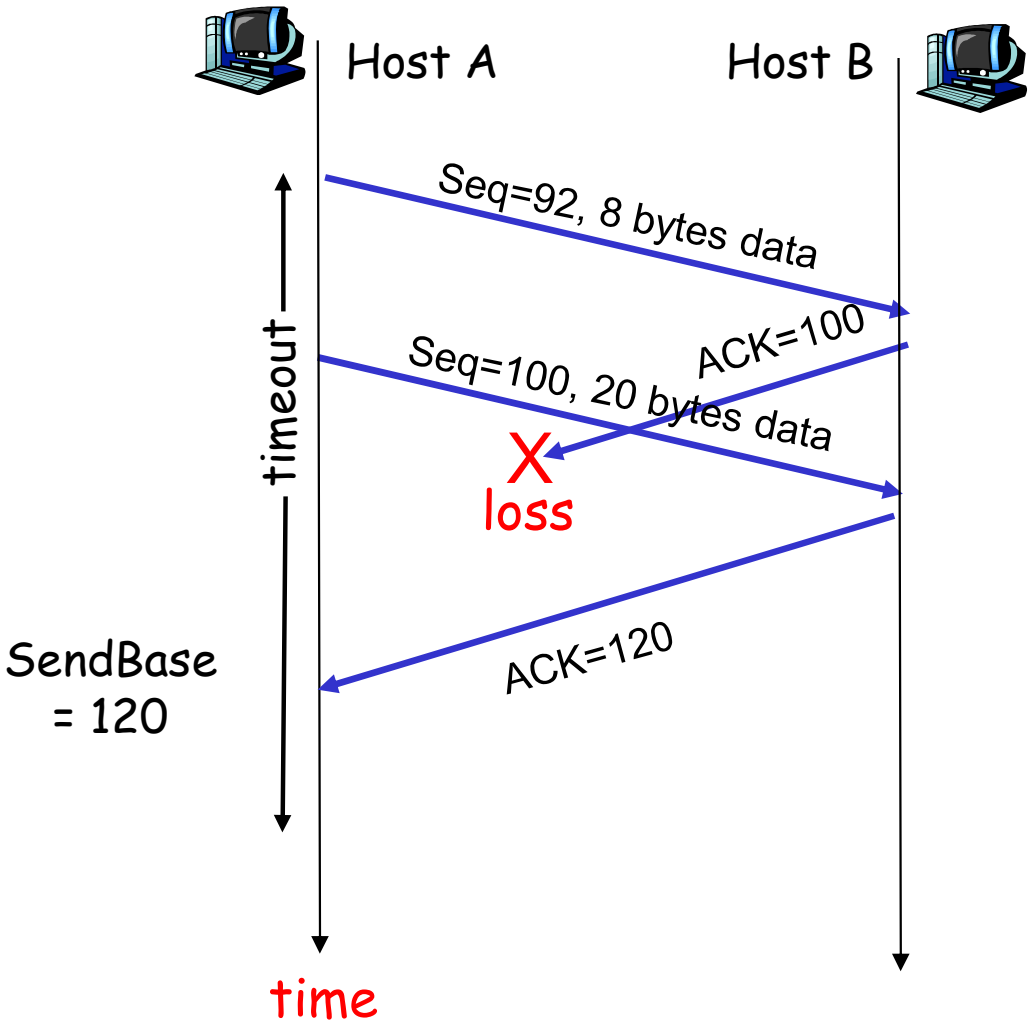


lost ACK scenario



premature timeout

TCP retransmission scenarios (more)



Cumulative ACK scenario

Sending ACKs is an overhead.
How to reduce the overhead?

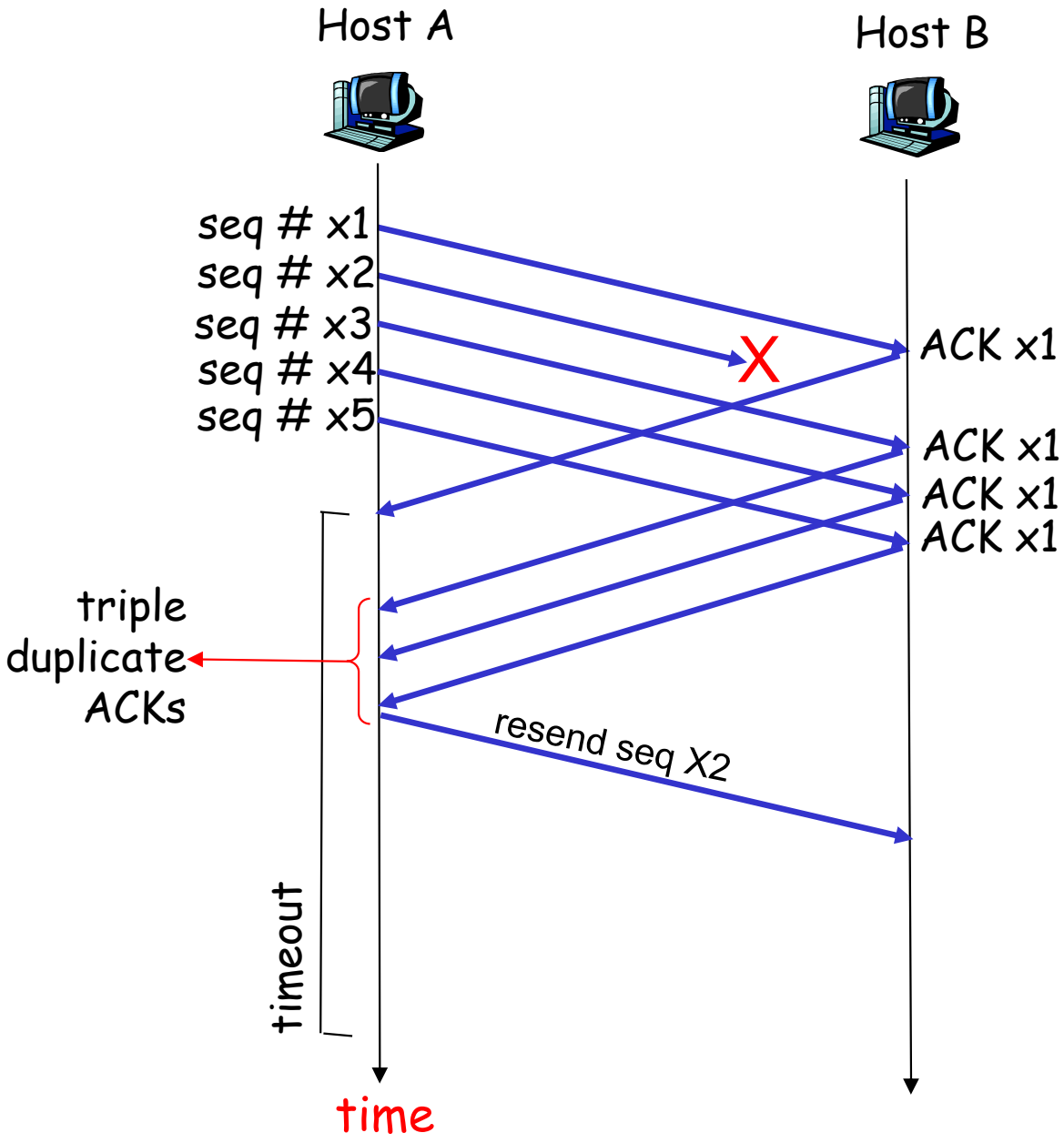
TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. 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 seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediately send ACK, provided that segment starts at lower end of gap

Retransmission upon timeout incurs significant delay. Can we retransmit sooner?

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 for that segment
- If sender receives 3 ACKs for same data, it assumes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires
  - Why 3 dup acks?


Host A
Host B
seq # x1
seq # x2
seq # x3
seq # x4
seq # x5
X
ACK x1
ACK x1
ACK x1
ACK x1
triple
duplicate
ACKs
resend seq X2
timeout
time

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
              }
            else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                }
```

a duplicate ACK for already ACKed segment

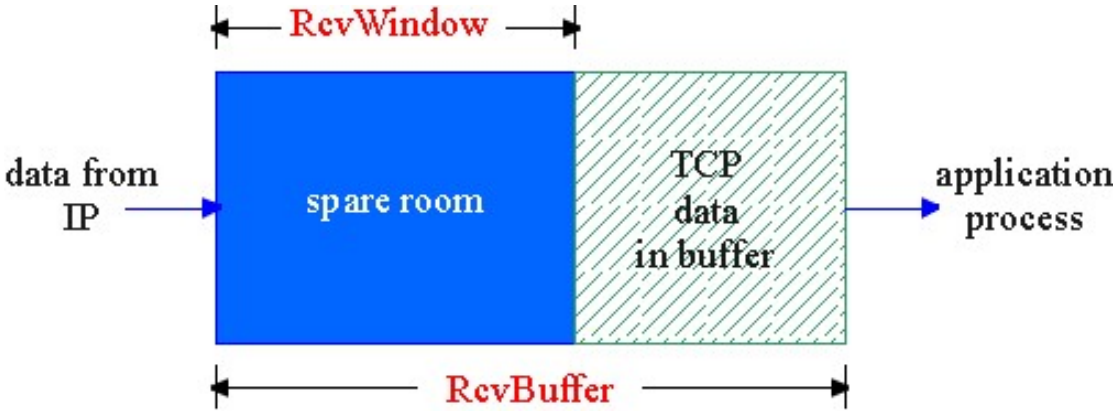
fast retransmit

TCP Flow Control

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

- receive side of TCP connection has a receive buffer:



- app process may be slow at reading from buffer

- speed-matching service: matching the sending rate to the receiving app's drain rate
- Rcvr advertises spare room by including value of **`RcvWindow`** in segments
- Sender limits unACKed data to **`RcvWindow`**
  - guarantees receive buffer doesn't overflow

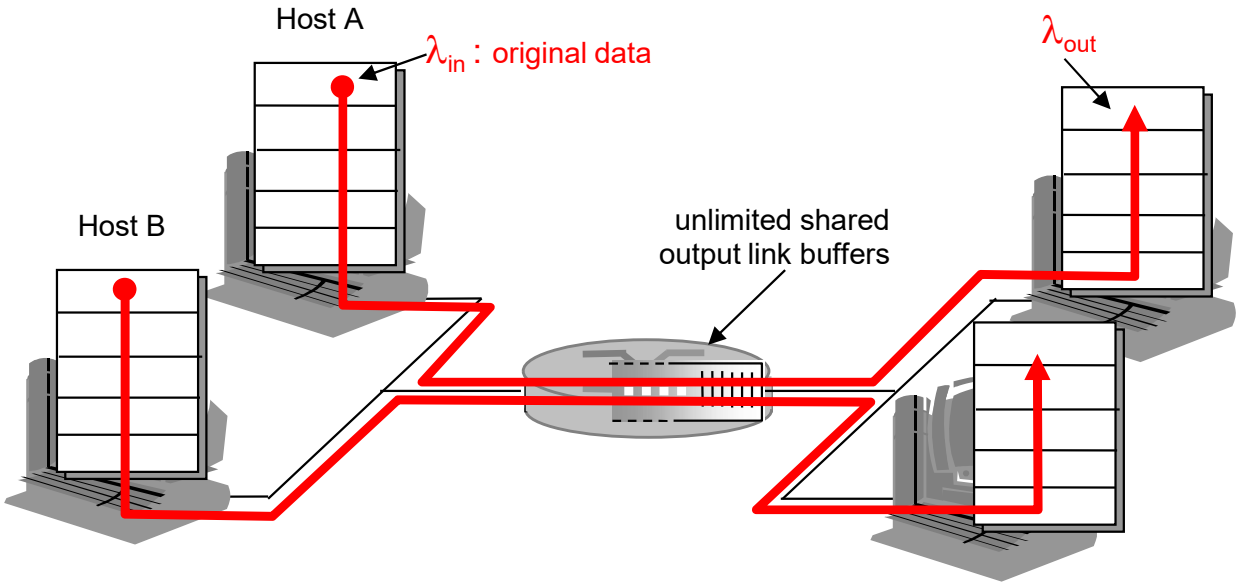
Principles of Congestion Control

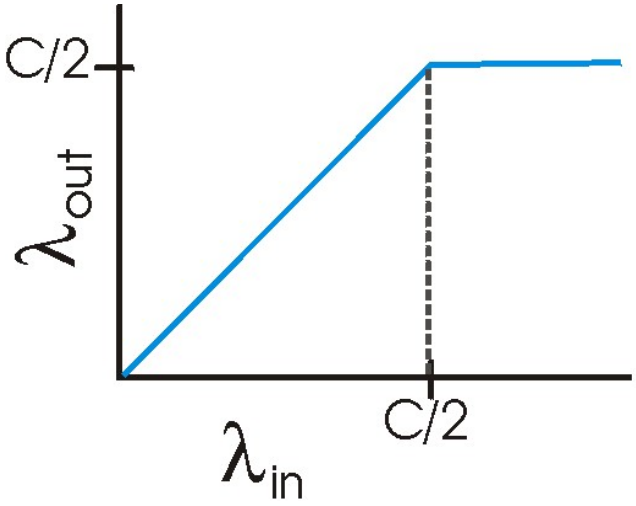
Congestion:

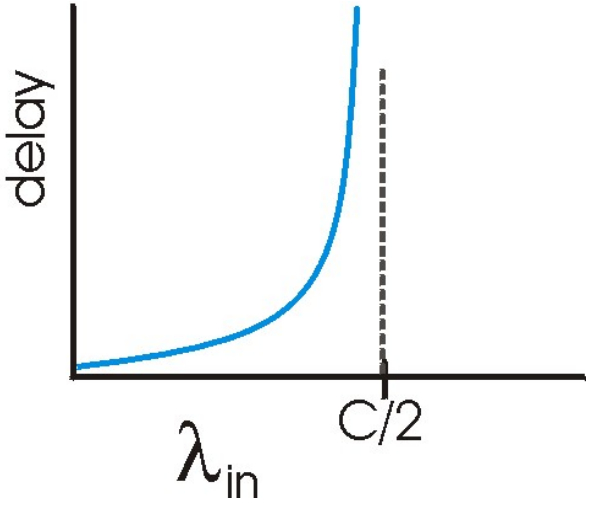
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission



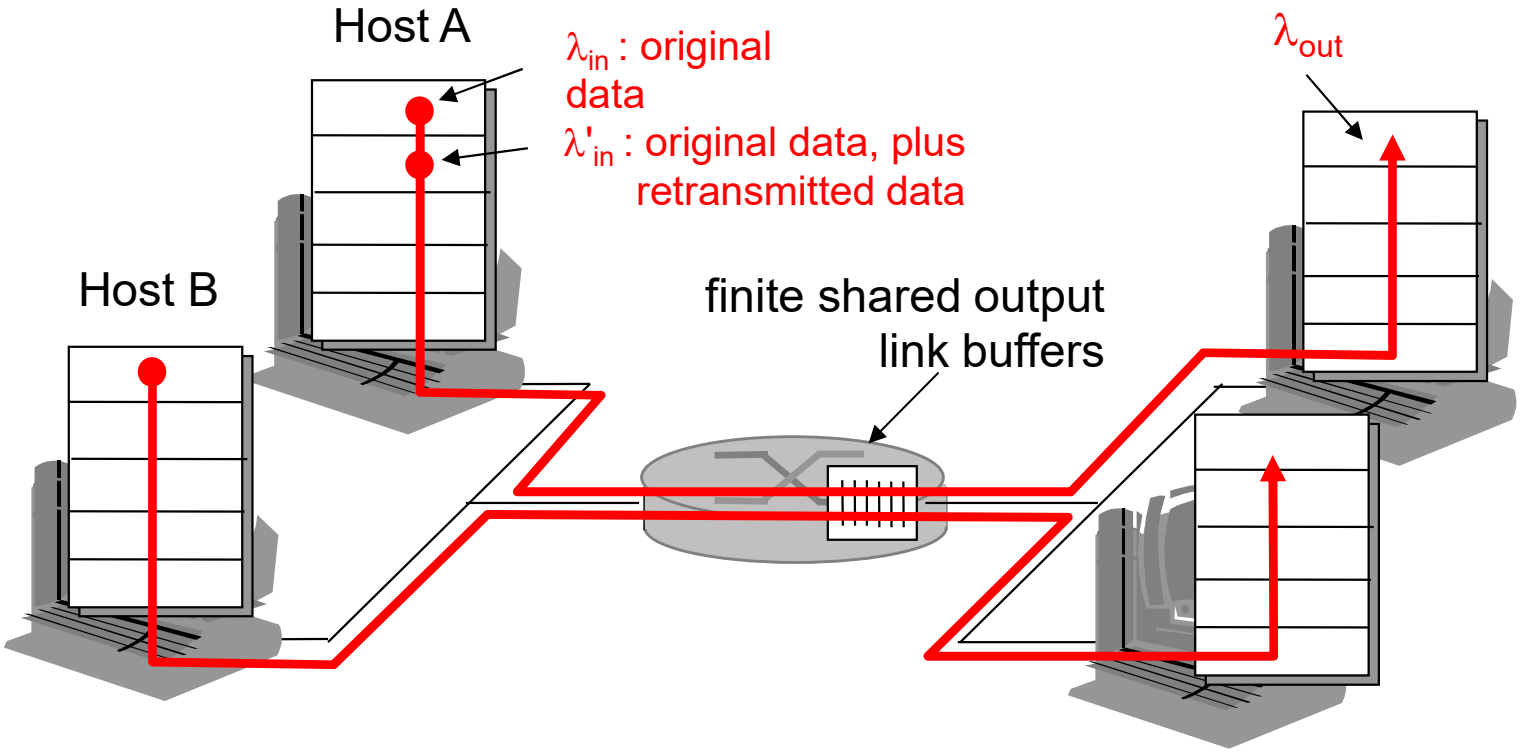




- large delays when congested
- maximum achievable throughput

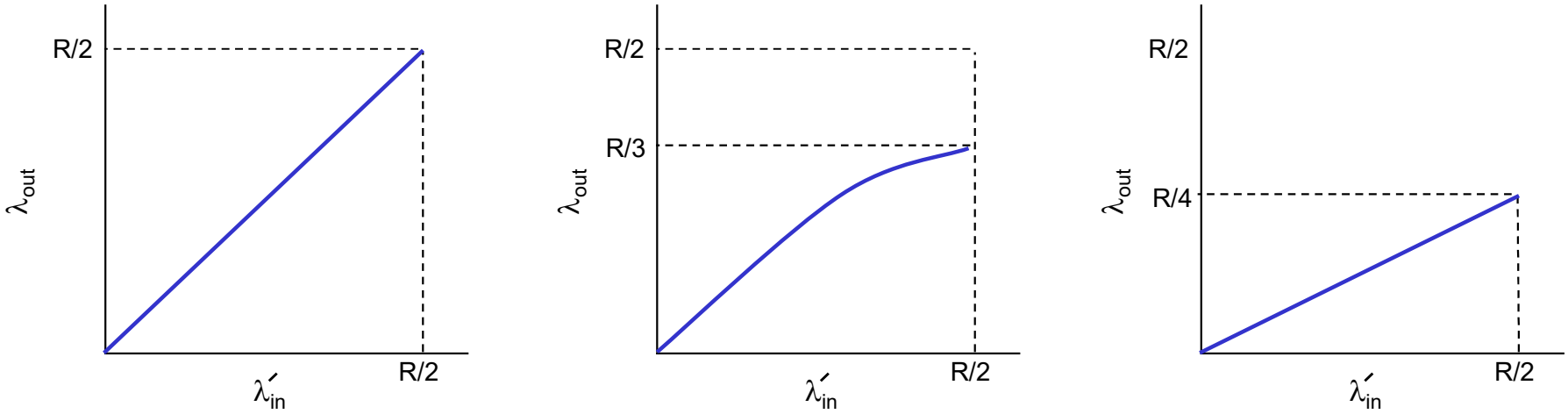
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet



Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



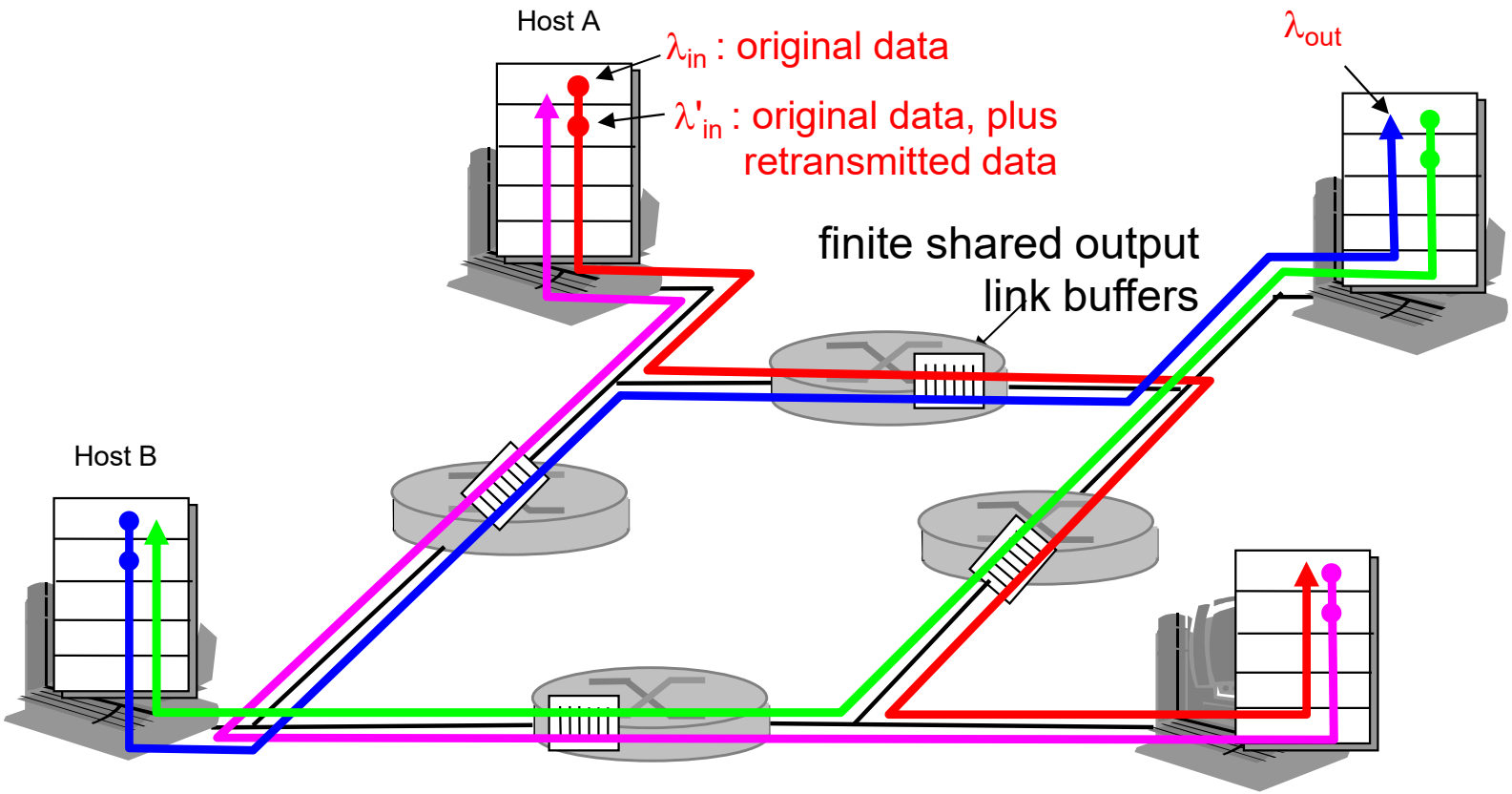
"costs" of congestion:

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

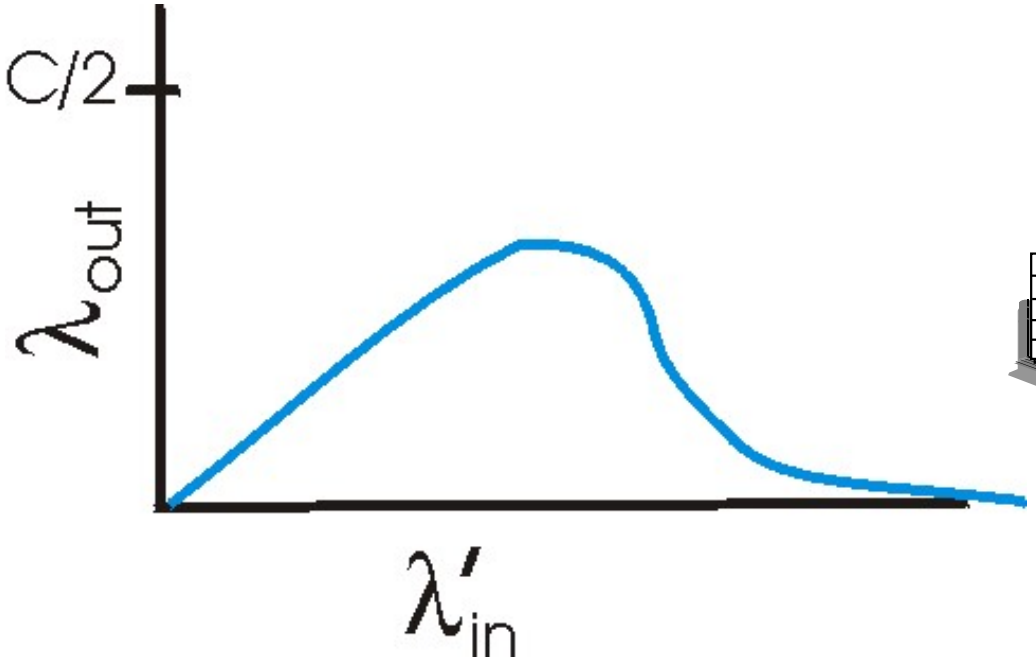
Causes/costs of congestion: scenario 3

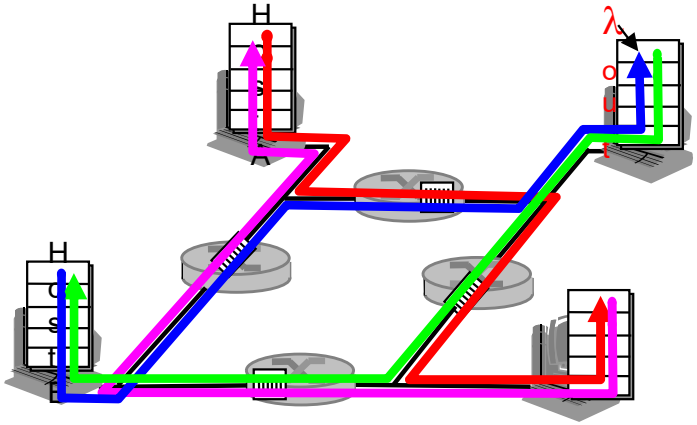
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as λ_{in} and λ'_{in} increase ?



Causes/costs of congestion: scenario 3





another "cost" of congestion:

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

Recap

- How to achieve reliable communication?
- What is flow control?
- What is congestion control?
- Why do we need congestion control?

How to avoid network congestion?

Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:

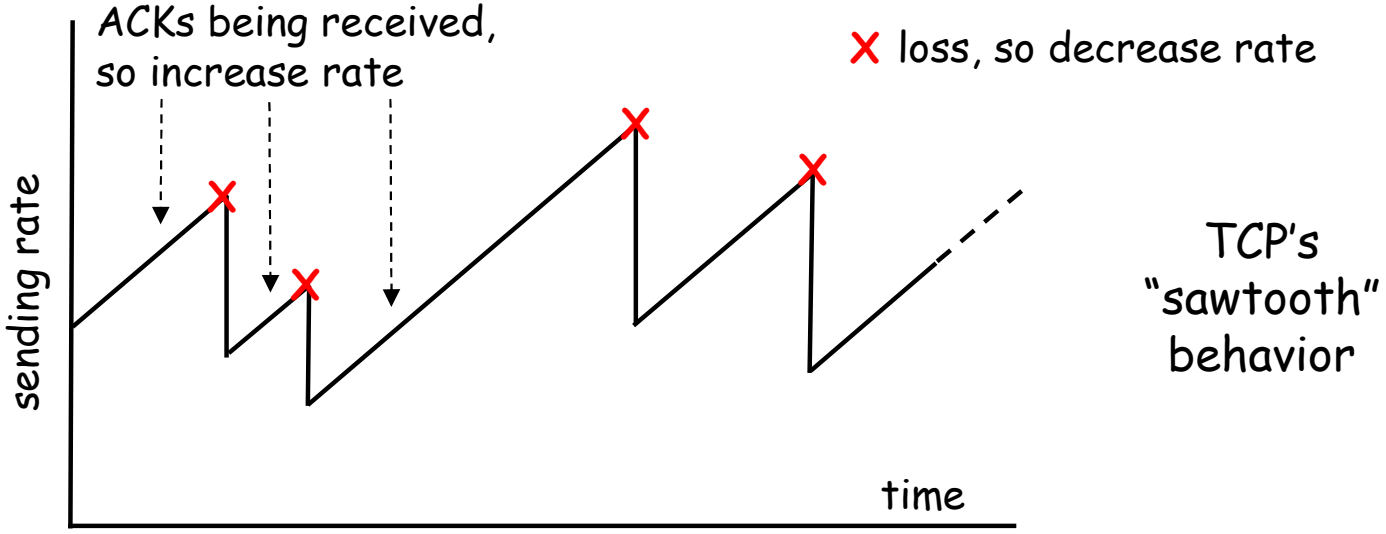
- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at (XCP)

TCP congestion control:

- *goal:* TCP sender should transmit as fast as possible, but without congesting network
  - Q: how to find rate *just* below congestion level
- decentralized: each TCP sender sets its own rate, based on *implicit* feedback:
  - *ACK:* segment received (a good thing!), network not congested, so increase sending rate
  - *lost segment:* assume loss due to congested network, so decrease sending rate

TCP congestion control: bandwidth probing

- "probing for bandwidth": increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate
  - continue to increase on ACK, decrease on loss (since available bandwidth is changing, depending on other connections in network)



- Q: how fast to increase/decrease?
  - details to follow

TCP Congestion Control: details

- sender limits rate by limiting number of unACKed bytes "in pipeline":

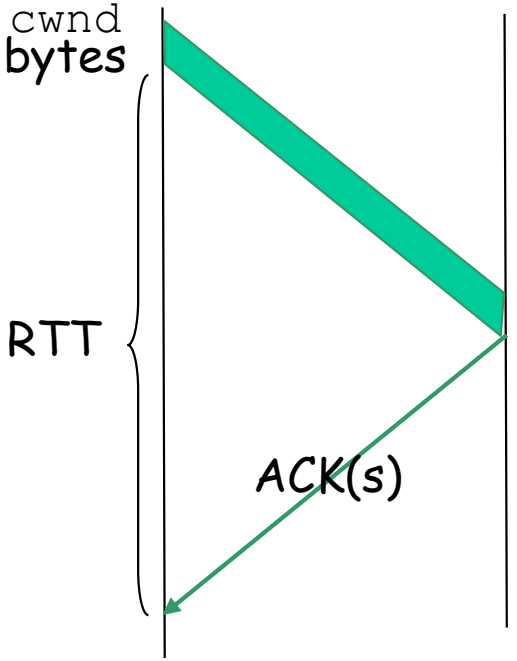
  LastByteSent-LastByteAcked ≤ cwnd

  - **cwnd:** differs from **rwnd** (how, why?)
  - sender limited by **min(cwnd,rwnd)**

- roughly,

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

- **cwnd** is dynamic, function of perceived network congestion



TCP Congestion Control: more details

segment loss event: reducing cwnd

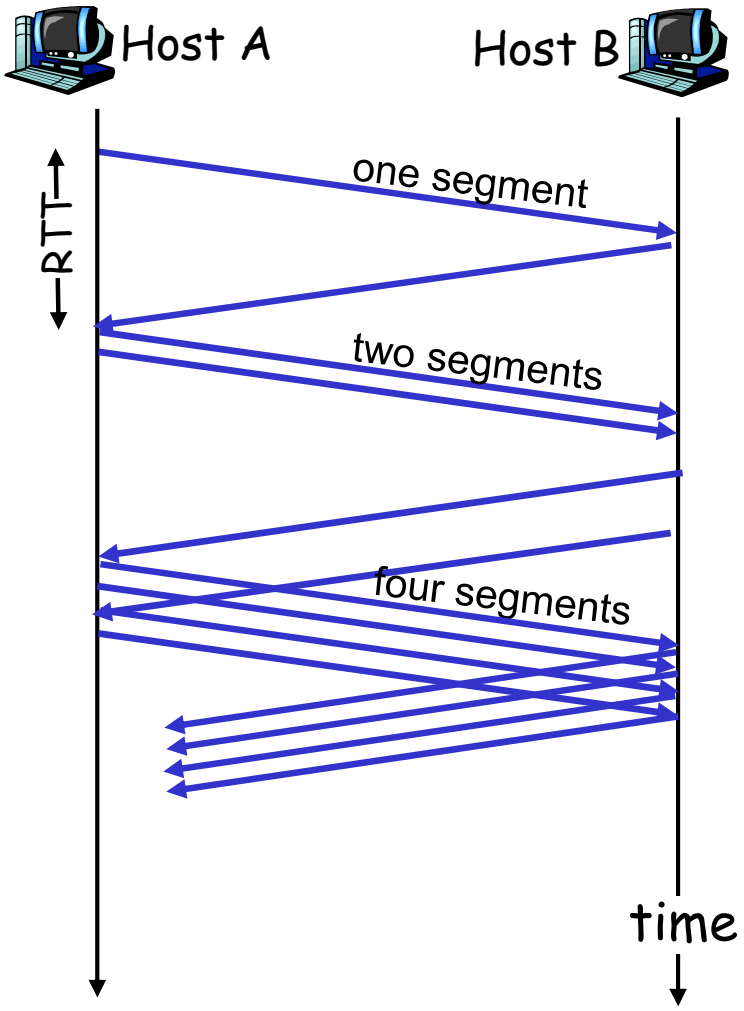
- timeout: no response from receiver
  - cut **cwnd** to 1
- 3 duplicate ACKs: at least some segments getting through (recall fast retransmit)
  - cut **cwnd** in half, less aggressively than on timeout

ACK received: increase cwnd

- Slow start phase:
  - increase exponentially fast (despite name) at connection start, or following timeout
- congestion avoidance:
  - increase linearly

TCP Slow Start

- when connection begins, **cwnd** = 1 MSS
  - example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- increase rate exponentially until first loss event or when threshold reached
  - double **cwnd** every RTT
  - done by incrementing **cwnd** by 1 for every ACK received



TCP: congestion avoidance

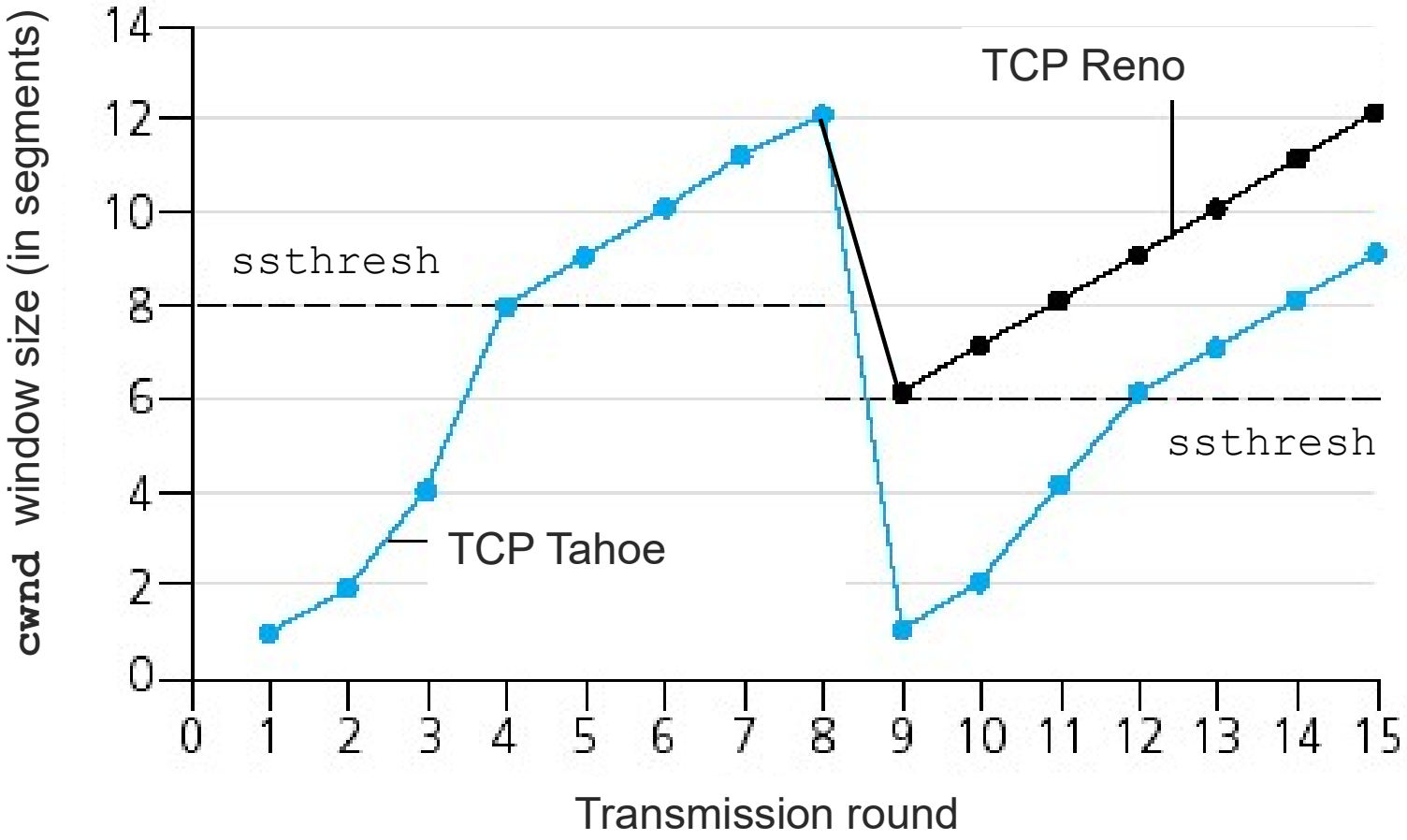
- when **cwnd > ssthresh** grow **cwnd** linearly
  - increase **cwnd** by 1 MSS per RTT
  - approach possible congestion slower than in slowstart
  - implementation: **cwnd = cwnd + MSS/cwnd** for each ACK received

AIMD

- ACKs: increase **cwnd** by 1 MSS per RTT: additive increase
- loss: cut **cwnd** in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease

Popular "flavors" of TCP



TCP Congestion Control

- When **CongWin** is below **Threshold**, sender in slow-start phase, window grows exponentially.
- When **CongWin** is above **Threshold**, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, **Threshold** set to **CongWin/2** and **CongWin** set to **Threshold**.
- When timeout occurs, **Threshold** set to **CongWin/2** and **CongWin** is set to 1 MSS.

TCP Cubic

Core Principles of TCP CUBIC Congestion Control

Cubic Growth Function

CUBIC uses a cubic growth function to adjust congestion window based on time since last congestion, enabling aggressive ramp-up when underutilized.

Improved Throughput Utilization

CUBIC maximizes bandwidth in high-speed, long-distance networks by avoiding slow linear window growth.

RTT Fairness and Scalability

CUBIC's time-based window growth reduces sensitivity to round-trip time differences, enhancing fairness among network flows.

Industry Adoption and Recovery

CUBIC is widely adopted in modern operating systems for its fast congestion recovery and scalability in broadband applications.

Why CUBIC is Preferred Over TCP Reno

Limitations of TCP Reno

Reno reacts only to packet loss, causing slow window recovery and poor throughput in high-latency networks.

CUBIC's Cubic Growth Function

CUBIC uses a cubic growth function to efficiently probe bandwidth and accelerate window growth after congestion.

Improved Fairness and Performance

CUBIC bases growth on time, reducing RTT dependency and improving fairness and throughput across networks.

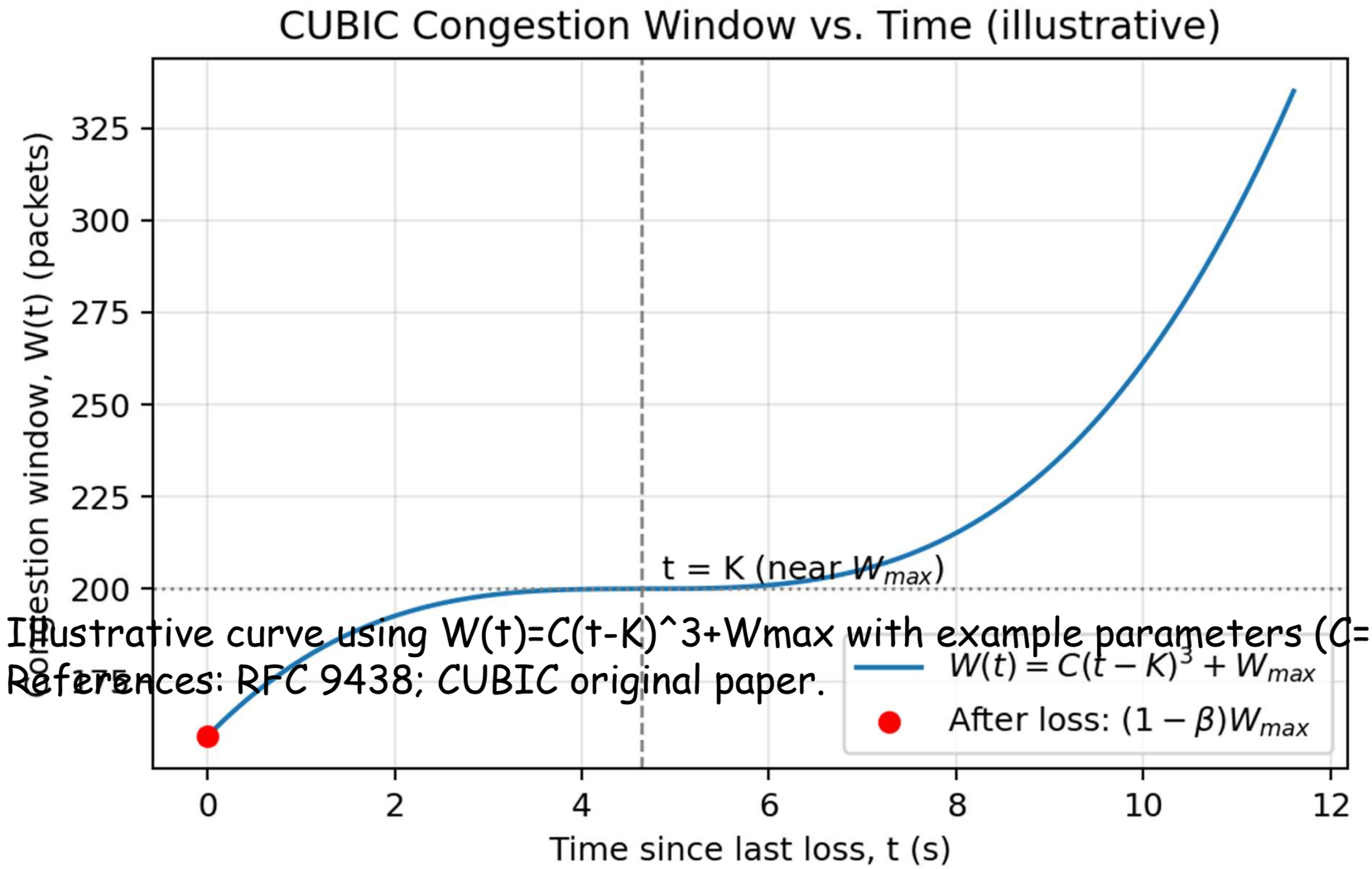
Adoption in Modern Systems

Modern OS kernels prefer CUBIC for its stable performance and better adaptation to high-speed networks.

CUBIC Window Formula

- Window growth
  - W(t) = C*(t-K)^3 + Wmax
  - K = (Wmax*beta/C)^(1/3)
  - Concave (rapid catch up) → Plateau → Convex (aggressive probing)
  - beta = 0.7, C = 0.4
- Multiplicative decrease rule
  - Wnew = (1-beta) Wmax

CUBIC Window Growth Curve



Illustrative curve using W(t)=C(t-K)^3+Wmax with example parameters (C=0.4, beta=0

References: RFC 9438; CUBIC original paper.

Where TCP CUBIC is Commonly Used

Widespread OS Adoption

TCP CUBIC is the default congestion control in most modern Linux-based systems powering servers and devices worldwide.

Optimized for High-Throughput Networks

CUBIC excels in broadband, fiber-optic, and multi-gigabit connections with low latency and packet loss.

Use in Data Centers and CDNs

Data centers, backbone networks, and content delivery networks rely on CUBIC for efficient bandwidth utilization.

Support for Streaming and Testing

CUBIC is preferred for multi-gigabit streaming and backbone network testing due to its bandwidth probing behavior.

Technical and Practical Limitations of CUBIC

Loss-Based Congestion Control

CUBIC reduces throughput when packet loss occurs, even if loss is due to noise, causing inefficiency in wireless networks.

Lack of Early Congestion Detection

Unlike delay-based algorithms, CUBIC reacts only after packet drops, potentially increasing bufferbloat and latency.

Fairness and Stability Issues

CUBIC may cause fairness problems when coexisting with other flows and produce unstable throughput in lossy environments.

TCP BBR

Introduction to TCP BBR Congestion Control

Model-Based Congestion Control

BBR uses a predictive model estimating bottleneck bandwidth and round-trip time for optimal sending rates.

Performance Advantages

BBR achieves higher throughput and lower latency by avoiding loss-based congestion signals and minimizing queue buildup.

Suitability for Modern Networks

BBR is ideal for wireless, high-bandwidth, long-distance, and latency-sensitive applications in contemporary internet environments.

Implementation and Impact

BBR is implemented in TCP and QUIC, enhancing stability, fairness, and performance across internet architectures.

Limitations of Traditional Loss-Based Congestion Control

Packet Loss as Congestion Signal

Traditional TCP algorithms interpret packet loss as a sign of congestion, increasing transmission until loss occurs.

Limitations in Wireless Networks

Wireless networks experience random packet loss from interference and fading, misleading loss-based congestion control.

Bufferbloat Impact

Larger router buffers cause queueing delays without packet loss, confusing loss-based congestion algorithms.

Need for Modern Algorithms

The mismatch between old strategies and new networks motivates model-based algorithms like BBR to improve performance.

Core Principles Behind BBR's Model-Based Design

Decoupling Congestion Control

BBR separates congestion control from packet loss by estimating bottleneck bandwidth and minimum RTT continuously.

Delivery Rate Sampling and Probing

BBR uses delivery rate sampling on acknowledgments and periodic probing to maintain accurate network condition models.

Control Variables in BBR

BBR controls pacing rate and congestion window to optimize packet transmission and limit in-flight data.

Minimizing Queuing Delay

BBR operates near the Kleinrock optimal point maximizing throughput with minimal queuing delay, preventing bufferbloat.

BBR State Machine and Operational Phases

Startup and Drain Phases

BBR starts with high pacing gain in STARTUP to find bottleneck bandwidth quickly, then moves to DRAIN to clear excess data and queues.

Probe Bandwidth Cycle

In PROBE_BW, BBR cycles through PROBE_UP, PROBE_DOWN, PROBE_CRUISE, and PROBE_REFILL to test and maintain optimal bandwidth use.

Probe RTT Phase

During PROBE_RTT, inflight data is reduced to measure minimum RTT, ensuring accurate latency estimates for stable performance.

Adaptive Congestion Control

BBR balances aggressive bandwidth discovery with queue prevention to maintain performance across diverse network conditions.

Evolution of BBR: Versions 1 through 3

BBRv1 Fundamentals

BBRv1 introduced a model-based congestion control but faced fairness challenges against loss-based flows like Reno and CUBIC.

BBRv2 Enhancements

BBRv2 improved fairness and coexistence by adjusting pacing and reacting to congestion signals, reducing loss in shallow buffers.

BBRv3 Refinements

BBRv3 further enhanced responsiveness, fairness, and RTT-awareness for stable performance in dynamic and mixed traffic environments.

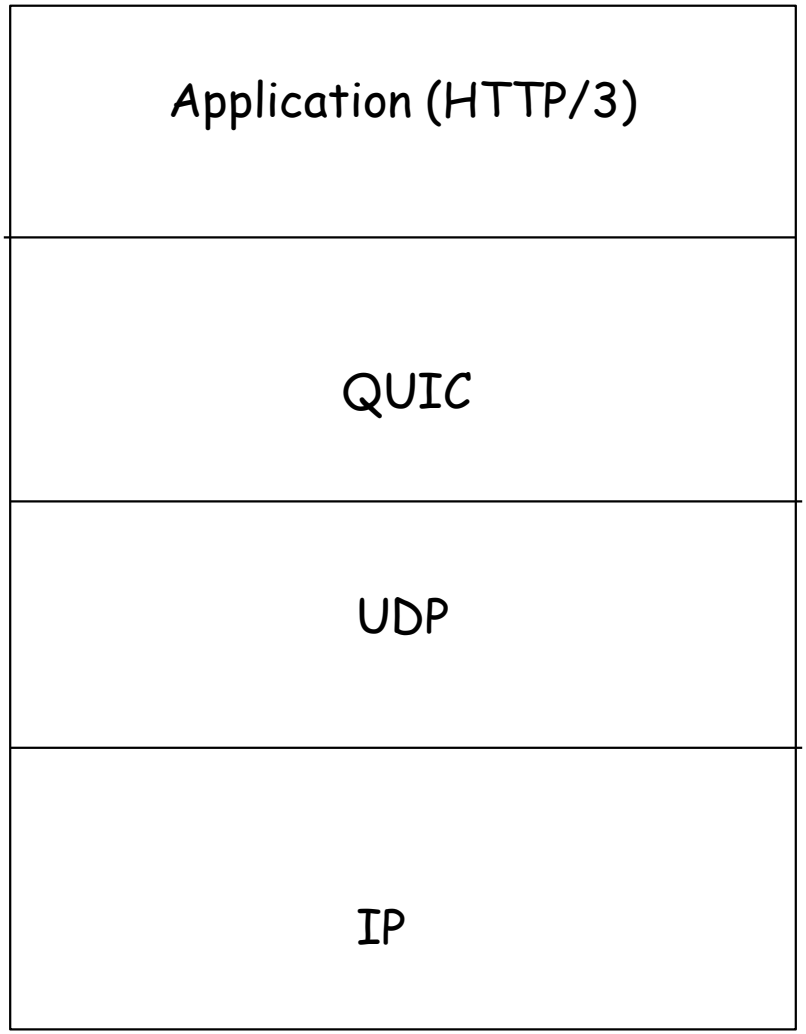
BBR Formulas

- BW = max (Δdelivered / Δt)
- Cwnd = cwnd_gain * BW * Rtmin
- Pacing_rate = pacing_gain * BW

- Startup: pacing_gain = 2.885, cwnd_gain=2
- Drain: pacing_gain = 1/2.885
- Probe_BW: pacing_gain = {1.25, 0.75, 1.0}, cwnd = 2 x BDP
- Probe_RTT: cwnd=4 pkts, pacing_rate = BW

QUIC

QUIC



- Transport protocol implemented in user space, running over UDP
- Provides classic transport functions:
  - Reliablity
  - Congestion control
  - Flow control
  - Loss recovery
  - multiplexing

Why QUIC Was Created

Enable rapid transport-layer innovation

Reduced Latency and Handshake Overhead

QUIC integrates connection setup and cryptographic negotiation, minimizing handshake delays for faster data transmission.

Elimination of Head-of-Line Blocking

By supporting independent data streams, QUIC prevents packet loss in one stream from stalling all traffic.

Resistance to Protocol Ossification

QUIC encrypts metadata to avoid interference by middleboxes, enhancing privacy and allowing protocol evolution.

What is QUIC?

Multiplexed Connections

QUIC supports multiplexed data streams to avoid head-of-line blocking issues seen in TCP multiplexing.

Integrated TLS 1.3 Security

TLS 1.3 is integrated directly into QUIC's transport layer, reducing connection setup time and overhead.

Connection Migration

QUIC enables connection migration to maintain active sessions despite changes in IP addresses or networks.

User-Space Congestion Control

QUIC's user-space congestion control allows faster adaptation and performance optimization across networks.

Core Features of QUIC

UDP Foundation with TCP Features

QUIC operates over UDP but adds TCP-like capabilities such as reliable delivery and congestion control.

Stream Multiplexing

Supports multiple independent data streams in a single connection, eliminating head-of-line blocking.

Connection Migration

Allows seamless session transitions across different network paths without new handshakes.

Integrated Security with TLS 1.3

Built-in TLS 1.3 ensures encrypted communication, reducing latency and enhancing security.

Where QUIC Is Used

Browser Support

Major web browsers like Chrome, Edge, Firefox, and Safari support QUIC as the default transport protocol for faster web interactions.

Google's Extensive Use

Google uses QUIC extensively, with over half of Chrome's connections to Google servers operating on QUIC to speed up services.

Foundation for HTTP/3

QUIC is the transport basis for HTTP/3, enabling faster, secure, and resilient web experiences as HTTP/3 adoption grows.

Global Integration

Content delivery networks, cloud platforms, and service providers adopt QUIC to improve latency, reliability, and mobile connection migration.

Limitations of QUIC

Protocol Complexity

QUIC's architecture is complex and rapidly evolving, making it difficult to fully understand and implement.

Encrypted Metadata Challenges

Encryption of metadata hinders traffic inspection and traditional network troubleshooting methods.

UDP Deployment Issues

Legacy firewalls and middleboxes may block or throttle UDP traffic, affecting QUIC connections.

Performance and Tooling Limitations

User-space QUIC implementations can use more CPU and lack mature monitoring and tuning tools.