Chapter 3: roadmap

3.1 Transport-layer services

3.2 Port numbers

3.3 Connectionless transport: UDP

UDP socket programming

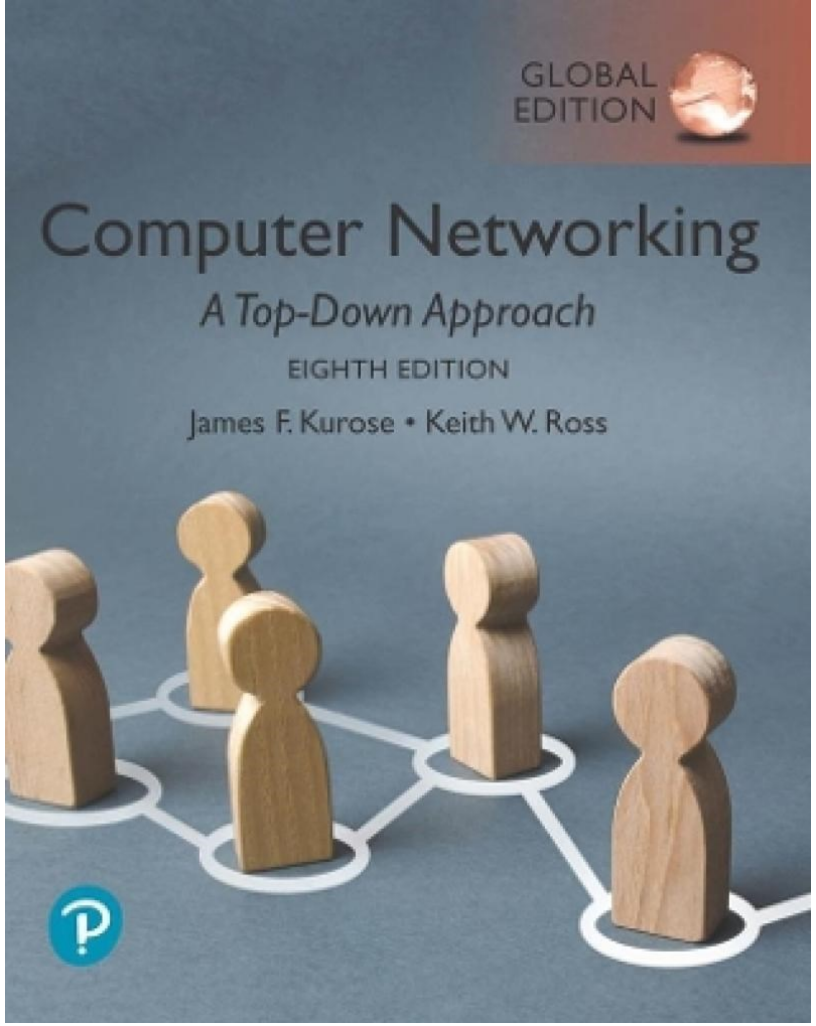
3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

~~3.6 Principles of congestion control~~

3.7 TCP congestion control

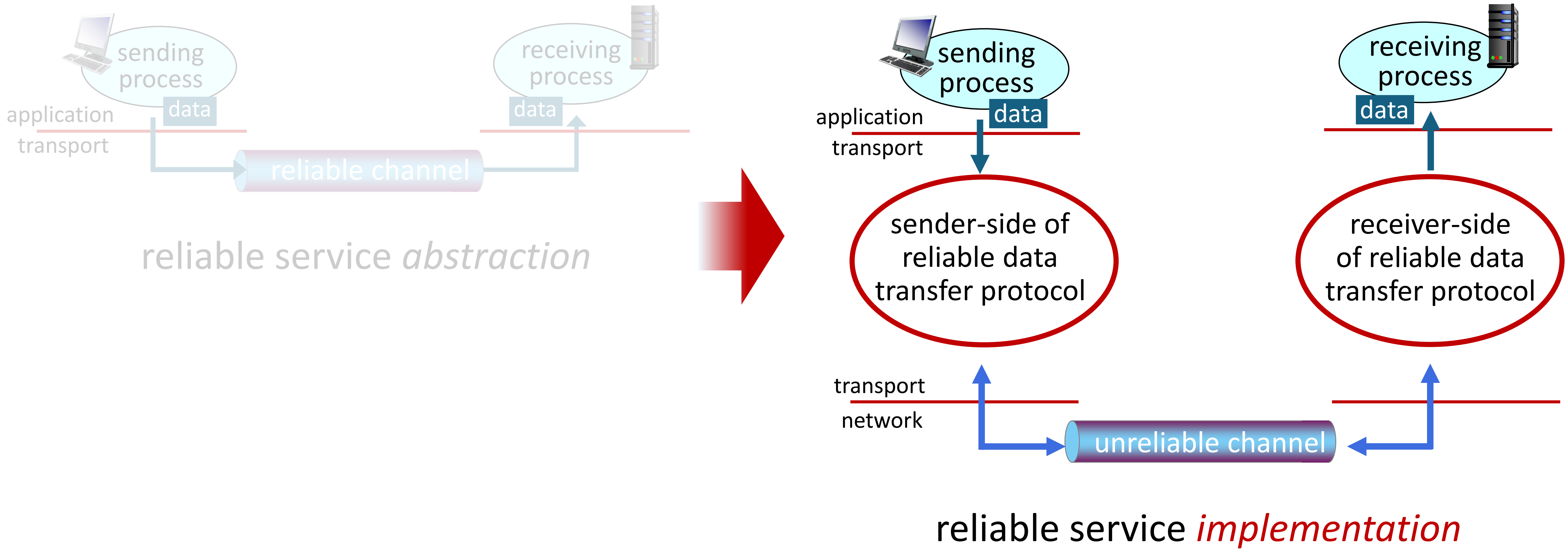
3.8 QUIC: Quick UDP Internet Connections



Principles of reliable data transfer

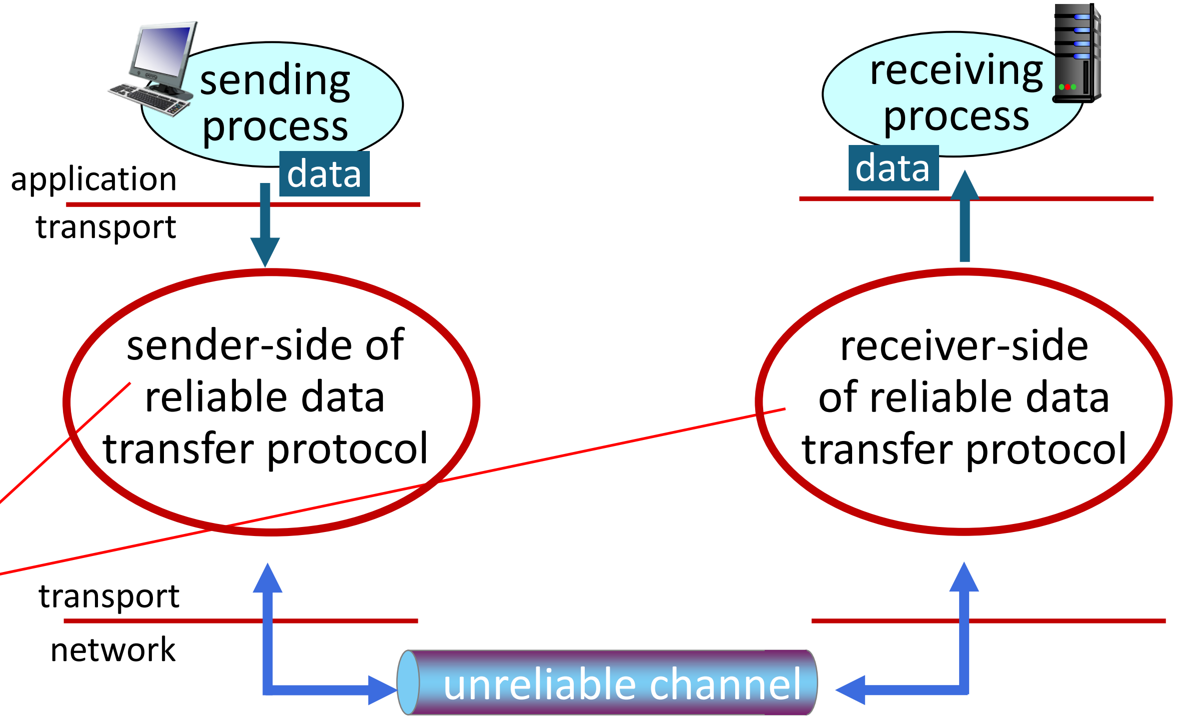
reliable service *abstraction*

Principles of reliable data transfer



Principles of reliable data transfer

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

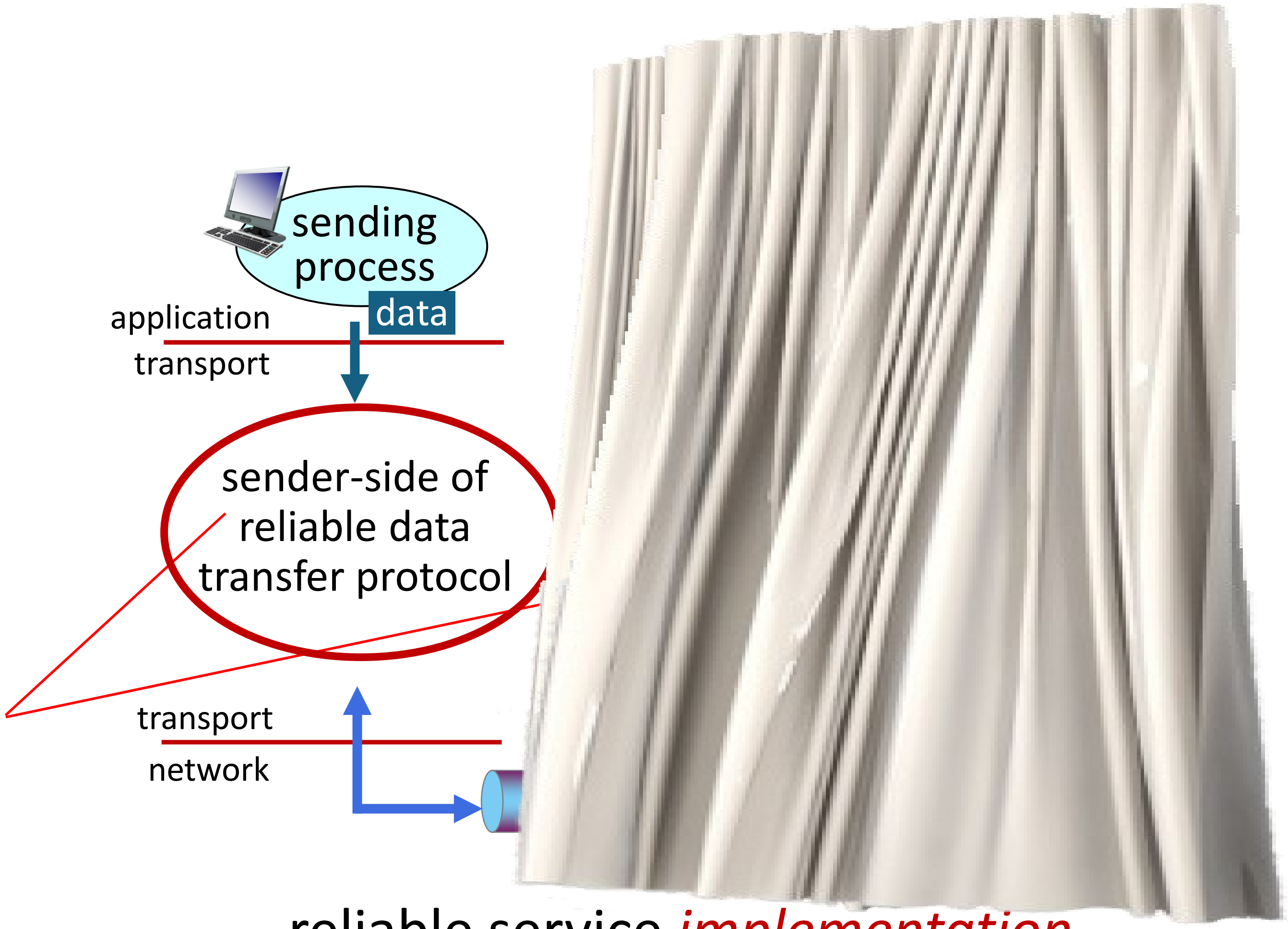


reliable service *implementation*

Principles of reliable data transfer

Sender, receiver do *not* know the “state” of each other, e.g., was a message received?

- unless communicated via a message

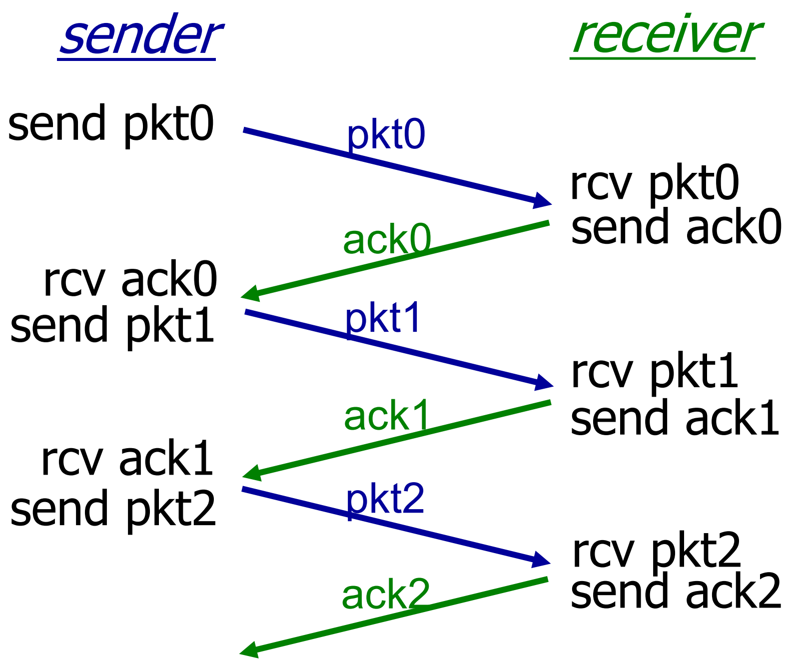


reliable service *implementation*

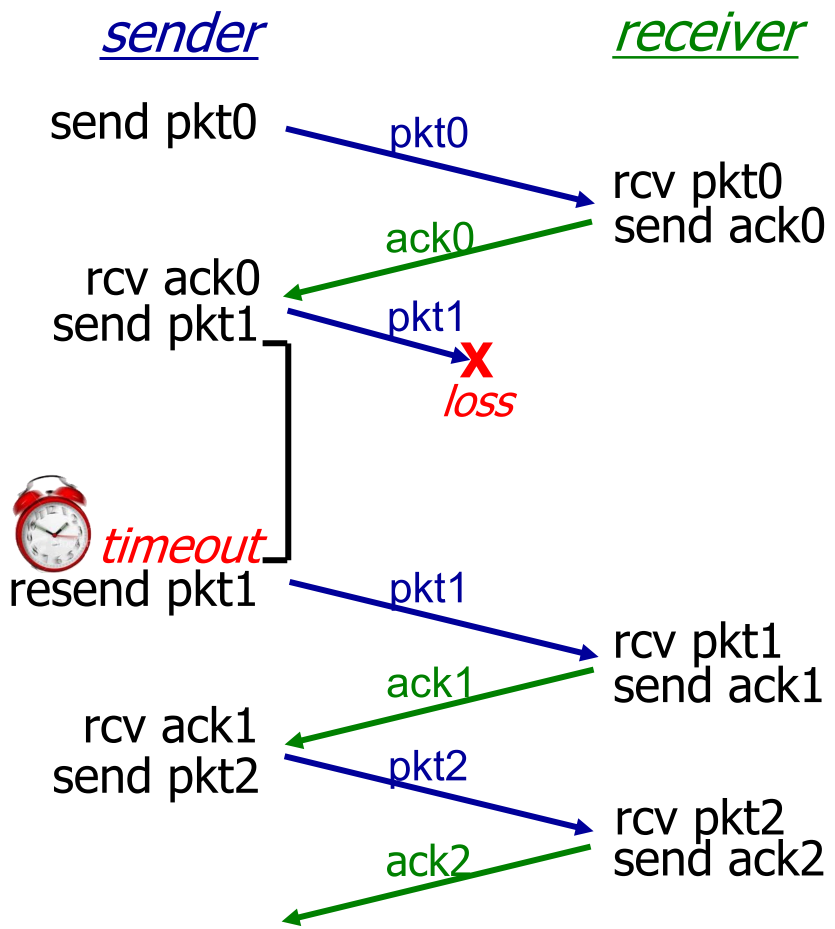
Mechanisms for reliable data transfer (rdt)

1. Acknowledgement: ACK
2. Timer (sender-side)
3. Sequence numbers

Principles of reliable data transfer (rdt3.0)

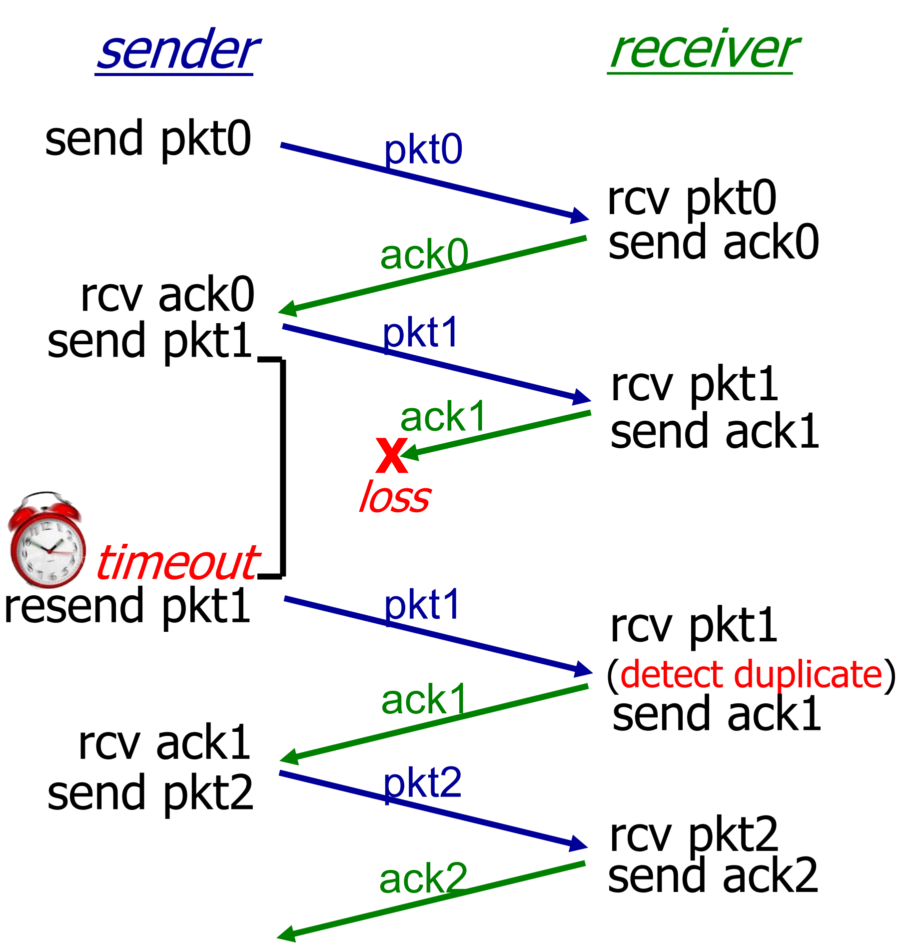


(a) no loss

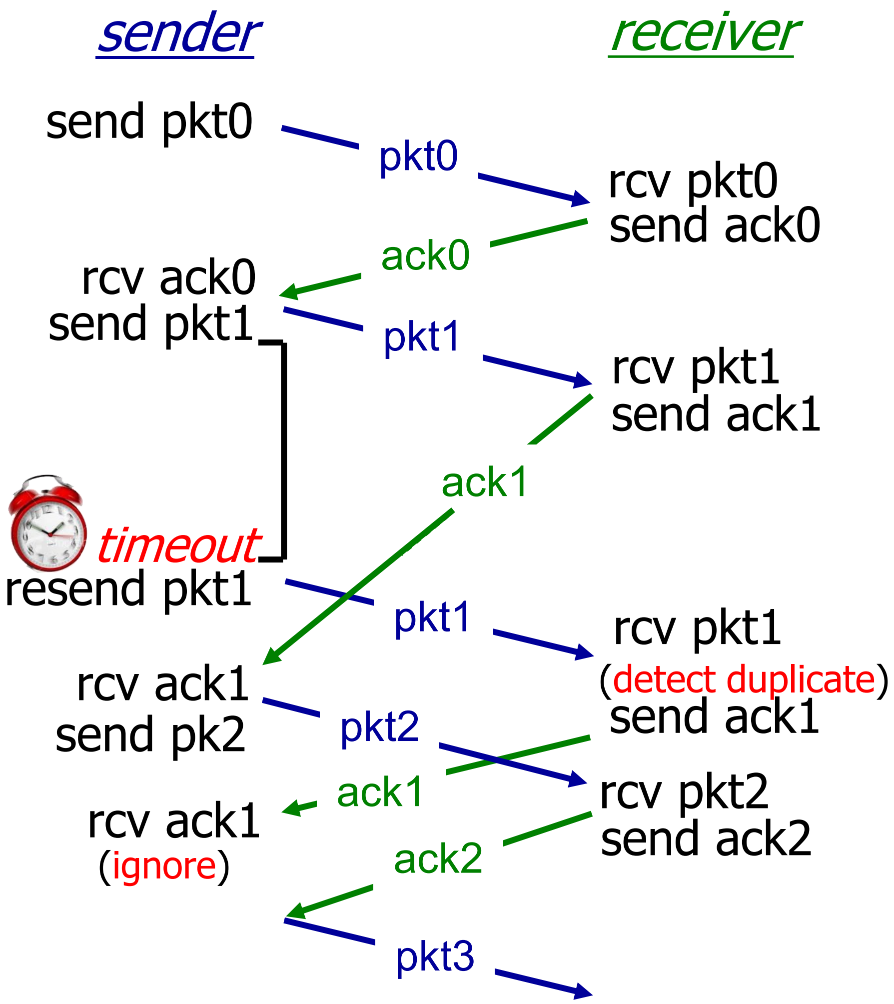


(b) packet loss

Principles of reliable data transfer (rdt3.0)



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance - stop-and-wait

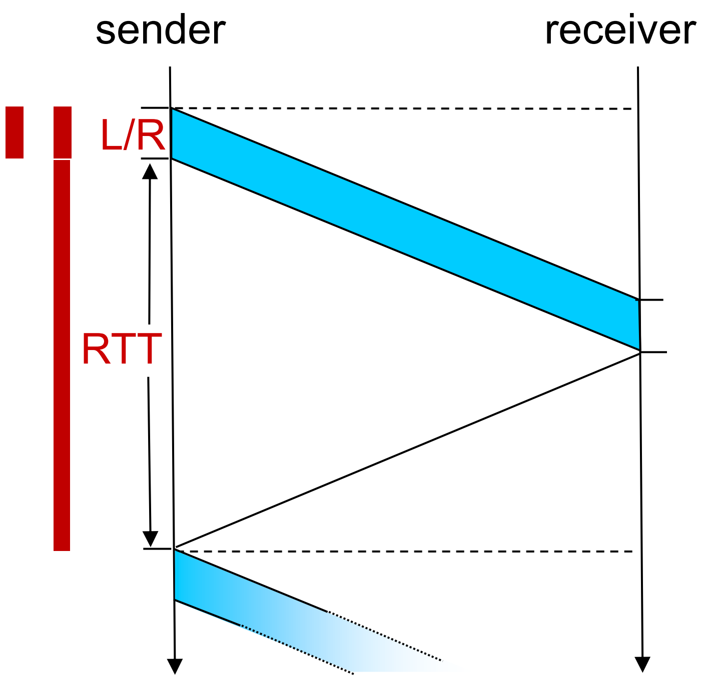
- U_{sender}: *utilization* – fraction of time sender busy sending

- example: 1 Gbps link, 15 ms propagation delay, 1000 bytes packet
  - time to transmit packet into channel:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

- example: 1 Gbps link, 15 ms propagation delay, 1000 bytes packet

$$\text{Utilization } U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{0.008}{30.008} \approx 0.00027 \approx 0.027\%$$

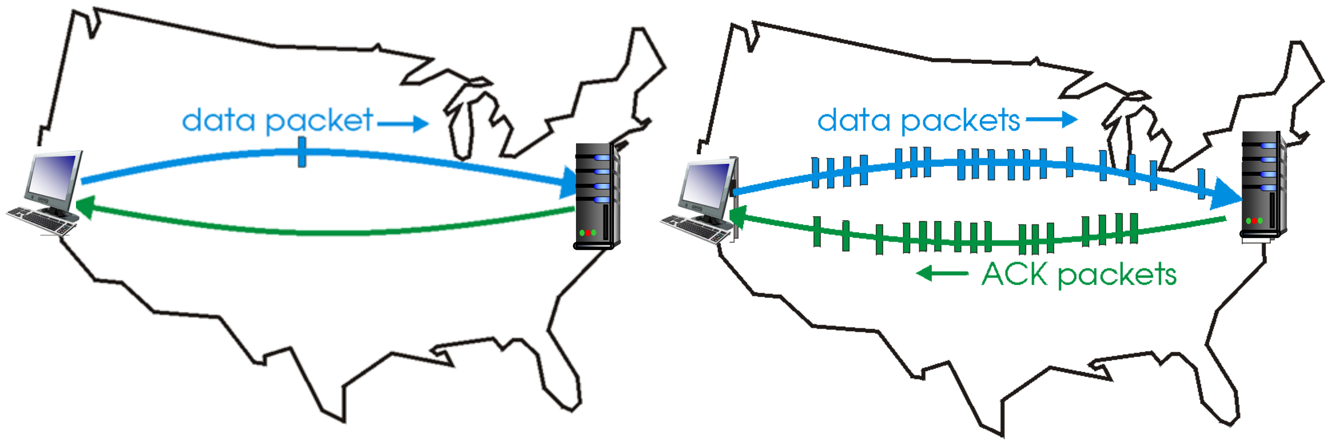


- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

rdt3.0: pipelined protocols operation

pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

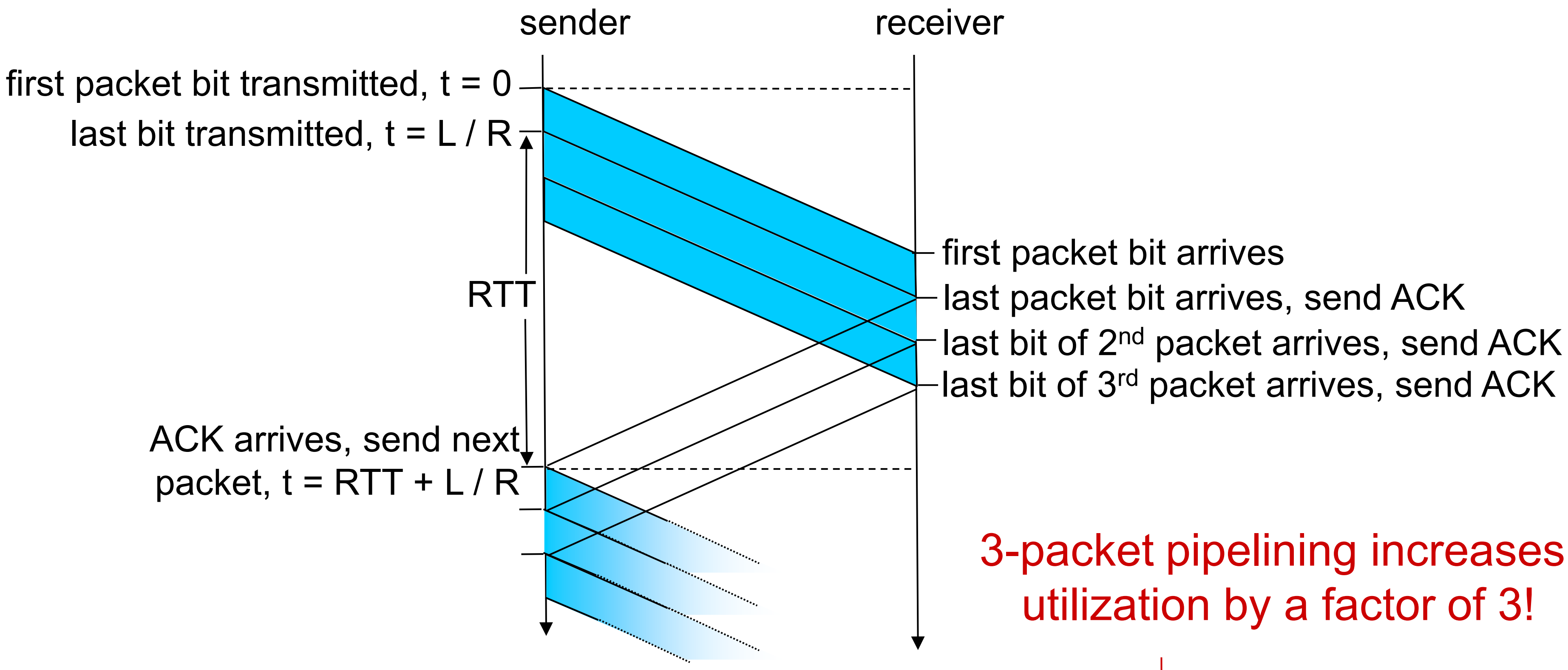
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

(b) a pipelined protocol in operation

Pipelining: increased utilization



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

$$U_{sender} = \frac{3 \cdot L/R}{RTT + L/R} = \frac{0.0024}{30.008} \approx 0.00081 \approx 0.081\%$$

Pipelined protocols: overview

Go-back-N:

- sender can have up to N unack'ed packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unack'ed packet
  - when timer expires, retransmit *all* unack'ed packets

Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- receiver sends *individual ack* for each packet
- sender maintains timer for *each* unack'ed packet
  - when timer expires, retransmit only that unack'ed packet

Chapter 3: roadmap

3.1 Transport-layer services

3.2 Port numbers

3.3 Connectionless transport: UDP

UDP socket programming

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

~~3.6 Principles of congestion control~~

3.7 TCP congestion control

3.8 QUIC: Quick UDP Internet Connections

3.5 Connection-oriented transport: TCP

1. TCP overview
2. Segment structure
3. The TCP connection
4. Opening and closing TCP connections
5. Reliable data transfer
6. TCP round trip time, timeout
7. Flow control

3.5.1 TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- Connection-orientation
  - handshaking establishes a TCP connection
  - point-to-point: one sender, one receiver
  - full duplex : bi-directional data flow in same connection
- Data segmentation
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- Reliable, in-order *byte steam*
- Pipelining
  - Cumulative ACKs
- Flow control
  - sender will not overwhelm receiver
- Congestion control

Comparison of UDP and TCP

UDP:

- Port numbers
- Integrity check
- Connectionless data transmission
- No data segmentation
- Not reliable data transfer

TCP:

- Port numbers
- Integrity check
- Connection-oriented data transmission
- Data segmentation
- Reliable data transfer
  - flow control and congestion control

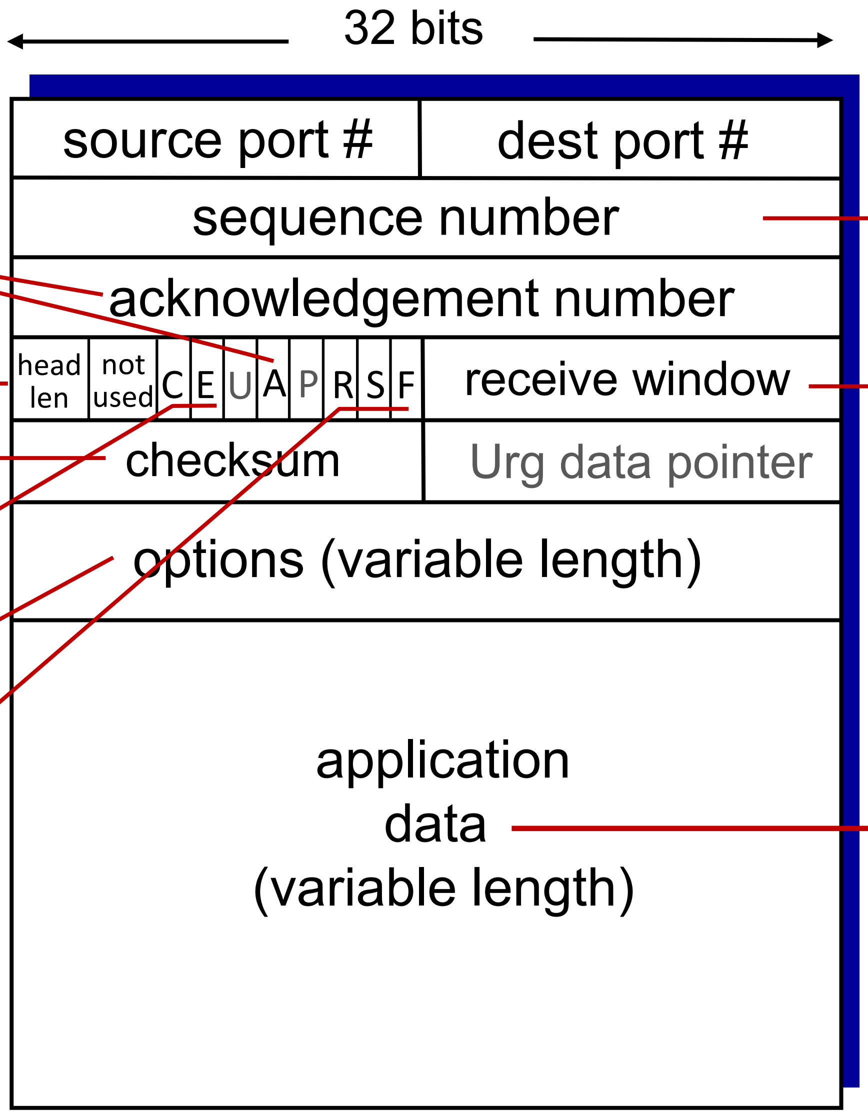
services not available:

delay guarantees and bandwidth guarantees

3.5 Connection-oriented transport: TCP

1. TCP overview
2. Segment structure
3. The TCP connection
4. Opening and closing TCP connections
5. Reliable data transfer
6. TCP round trip time, timeout
7. Flow control

3.5.2 TCP segment structure



ACK: seq # of next expected byte; A bit: this is an ACK

length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

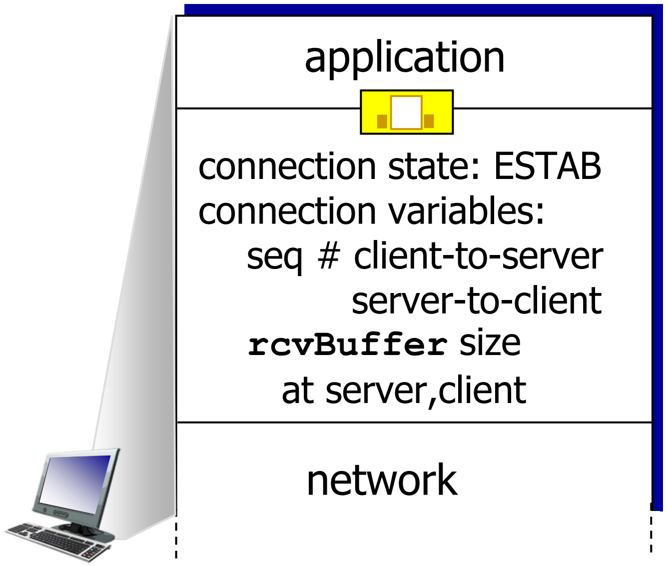
data sent by application into TCP socket

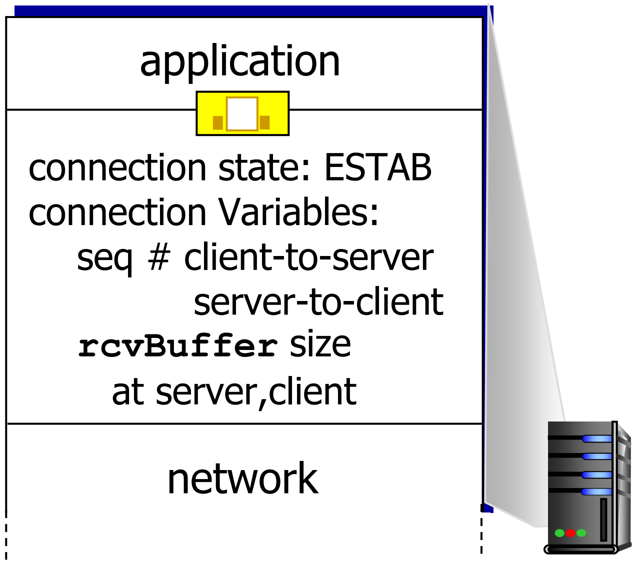
3.5 Connection-oriented transport: TCP

1. TCP overview
2. Segment structure
3. The TCP connection
4. Opening and closing TCP connections
5. Reliable data transfer
6. TCP round trip time, timeout
7. Flow control

3.5.3 TCP connection management

before exchanging data, sender/receiver "handshake":
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)






```
clientSocket = socket(AF_INET,SOCK_STREAM)

connect(clientSocket, hostName,portNumber)
```

```
serverSocket = socket(AF_INET,SOCK_STREAM)
bind(serverSocket,serverPort)
serverSocket.listen(1)
connectionSocket = accept(serverSocket)
```

TCP connection management

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - destination IP address
  - destination port number
- receiver uses *all four values (4-tuple)* to direct segment to appropriate socket

- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

Comparison of connection management

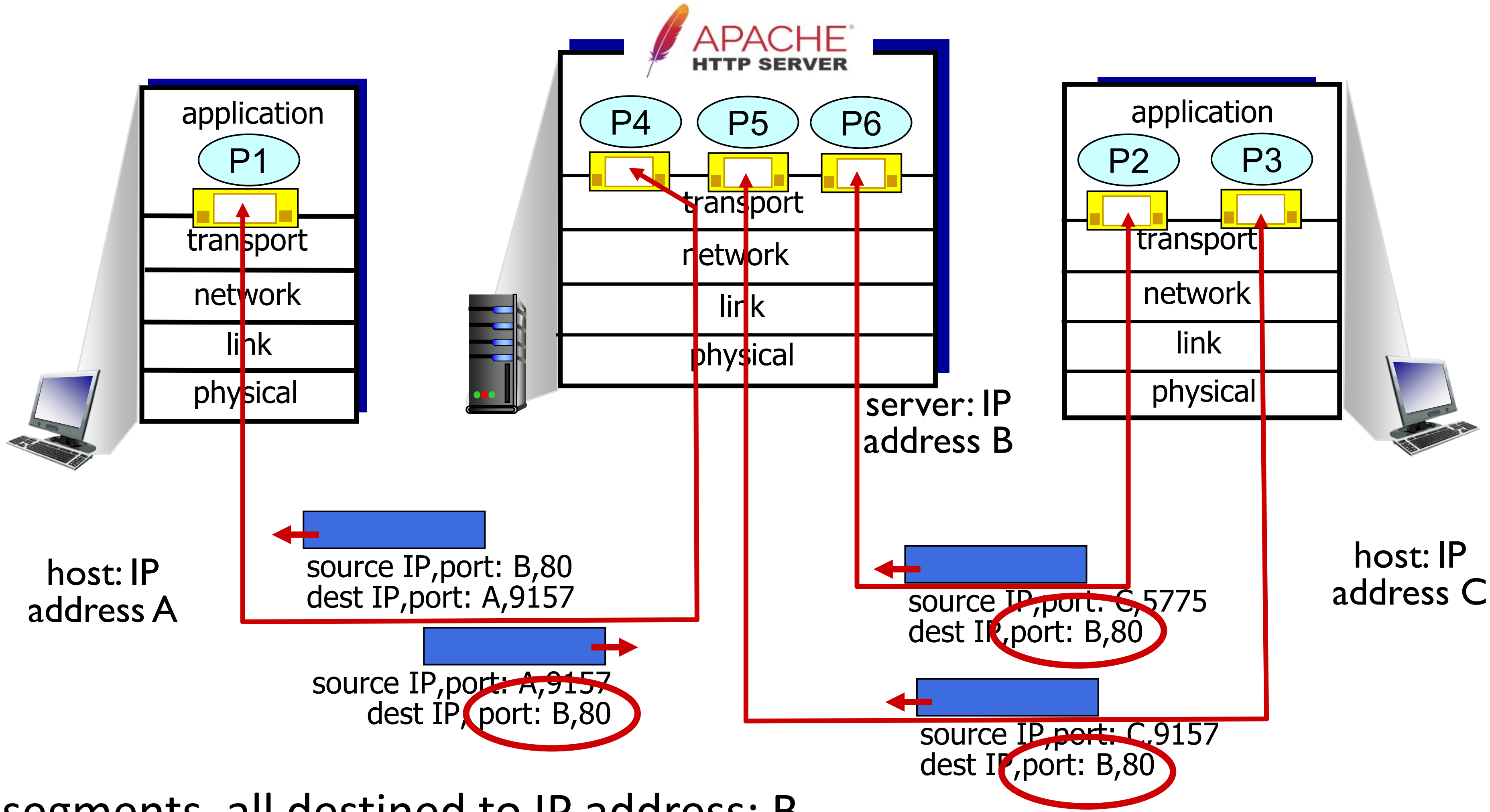
UDP: target socket identified using 2-tuple:

- destination IP and destination port number

TCP: target socket identified using 4-tuple:

- source and destination IP addresses
- source and destination port numbers

TCP connections and sockets: example

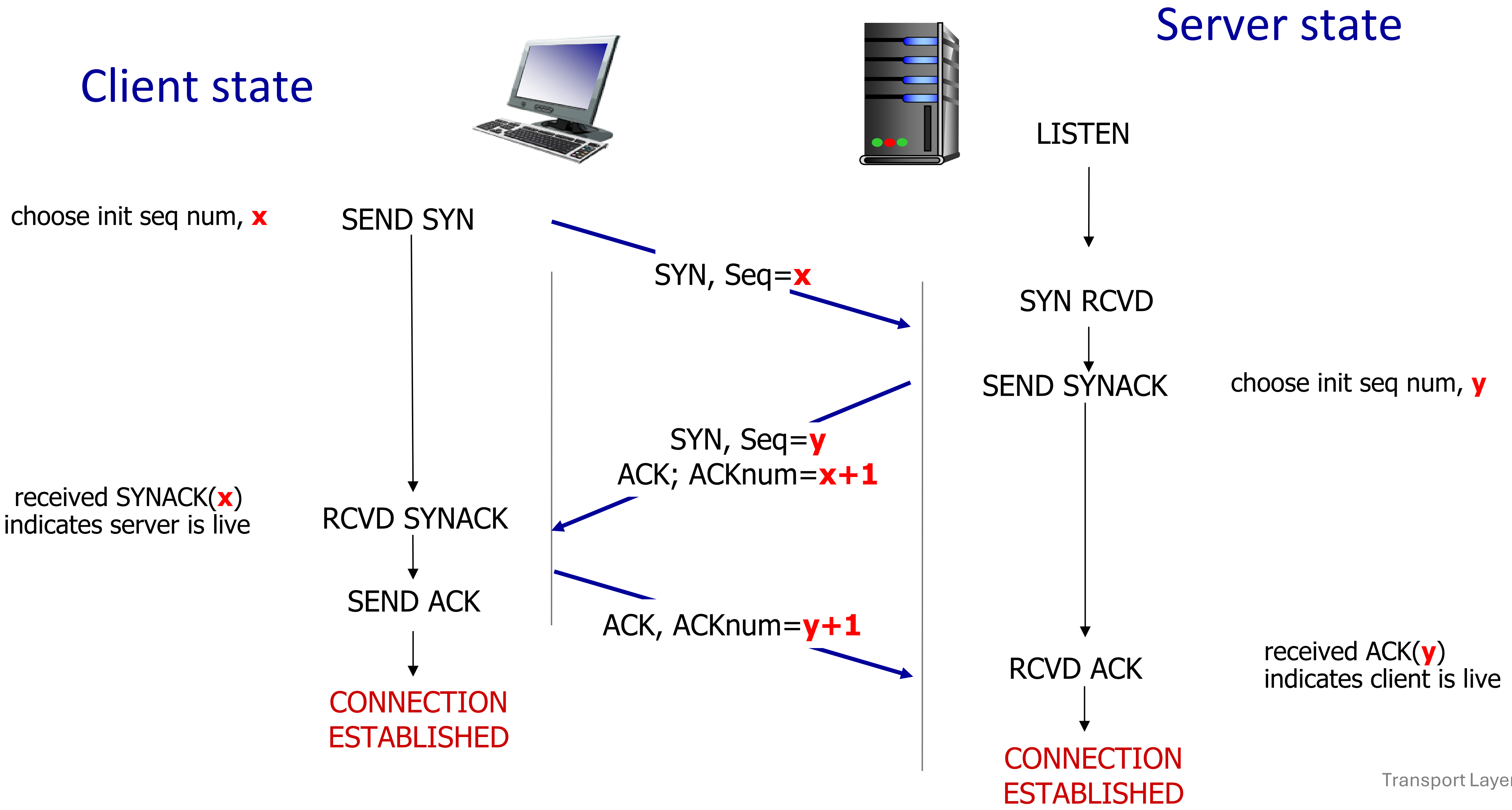


Three segments, all destined to IP address: B,
destination port: 80 are demultiplexed to *different* sockets

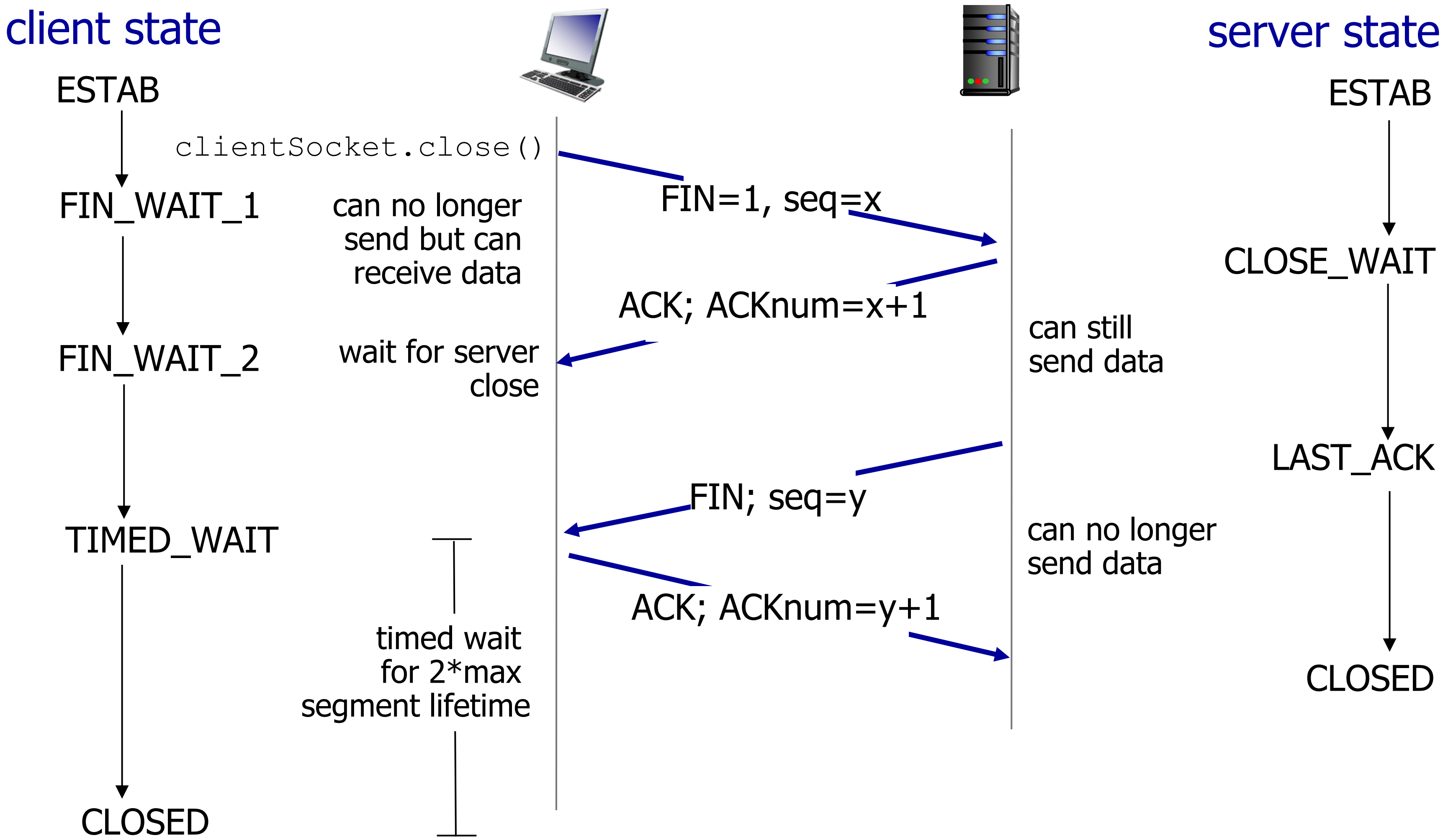
3.5 Connection-oriented transport: TCP

1. TCP overview
2. Segment structure
3. The TCP connection
4. Opening and closing TCP connections
5. Reliable data transfer
6. TCP round trip time, timeout
7. Flow control

New TCP connection: 3-way handshake



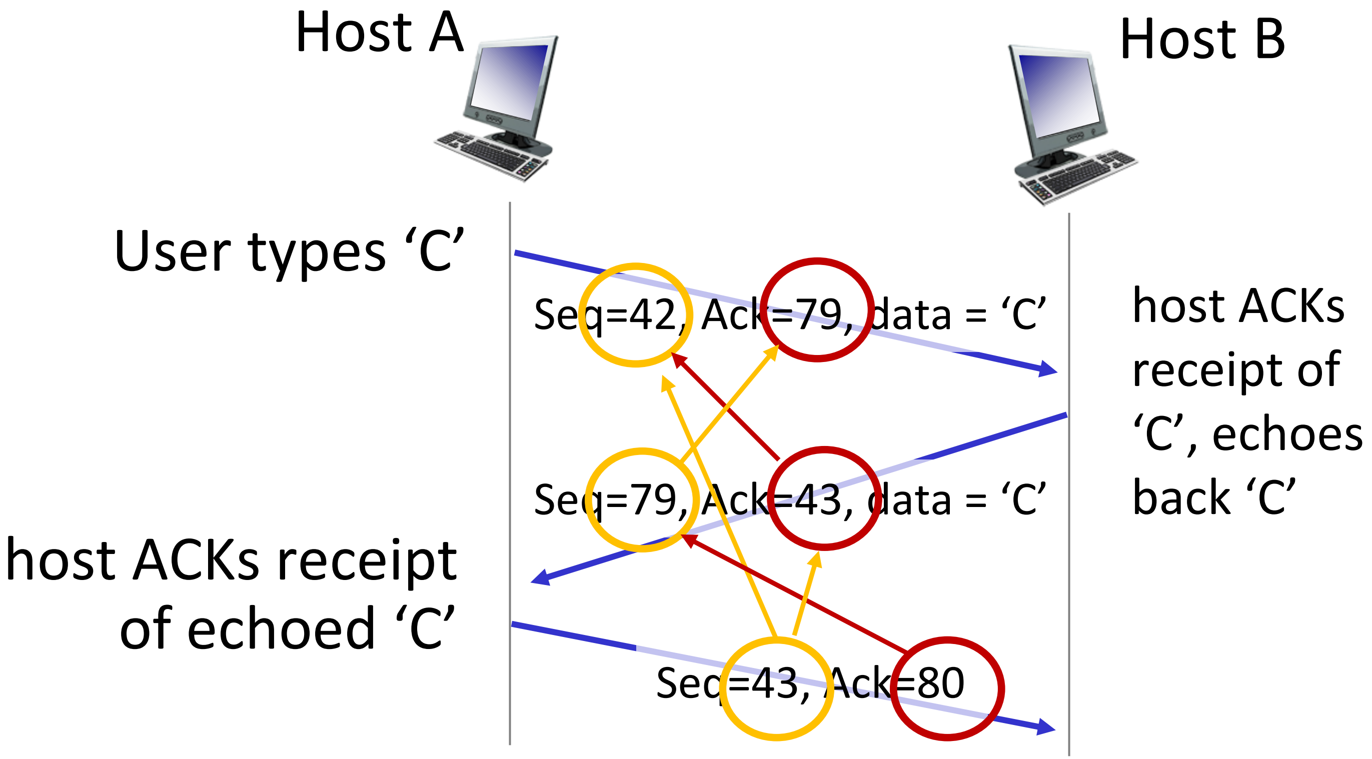
Closing a TCP connection



3.5 Connection-oriented transport: TCP

1. TCP overview
2. Segment structure
3. The TCP connection
4. Opening and closing TCP connections
5. Reliable data transfer
6. TCP round trip time, timeout
7. Flow control

TCP sequence numbers, ACKs



simple telnet scenario

SEQ-number:

- ACK-number of received segment
- incremented by payload size of previously sent segment

ACK-number:

- seq. number + payload size of received segment
- the expected seq. number of next received segment

Why random sequence numbers?

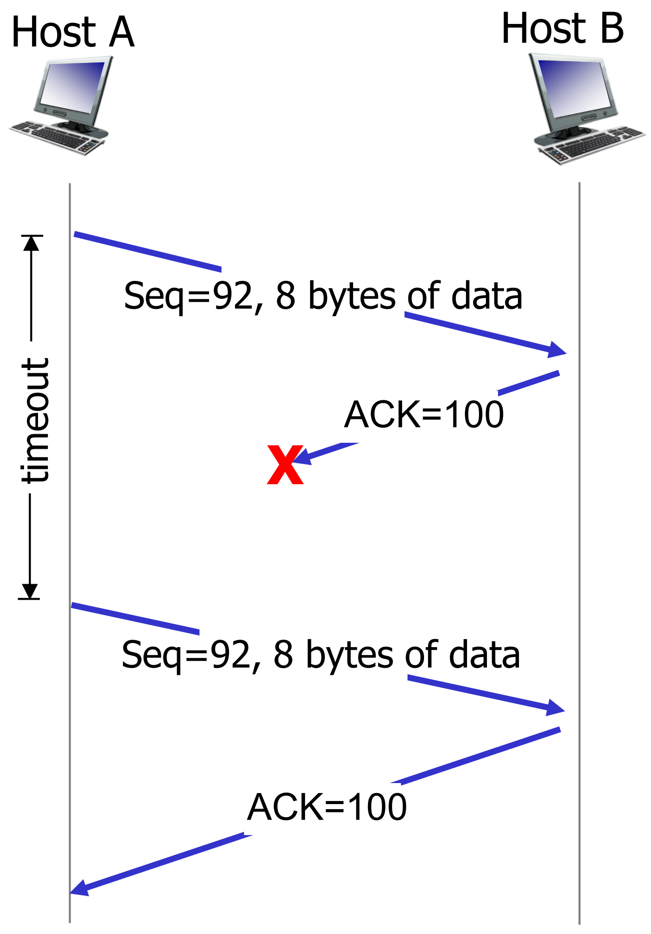
Multi-session interference.

- If all sessions started their sequence numbers at 1, then it would be much easier to end up in situations where you mix up packets from various sessions between two hosts (though there are other measures in place to avoid this, like randomizing the source port).

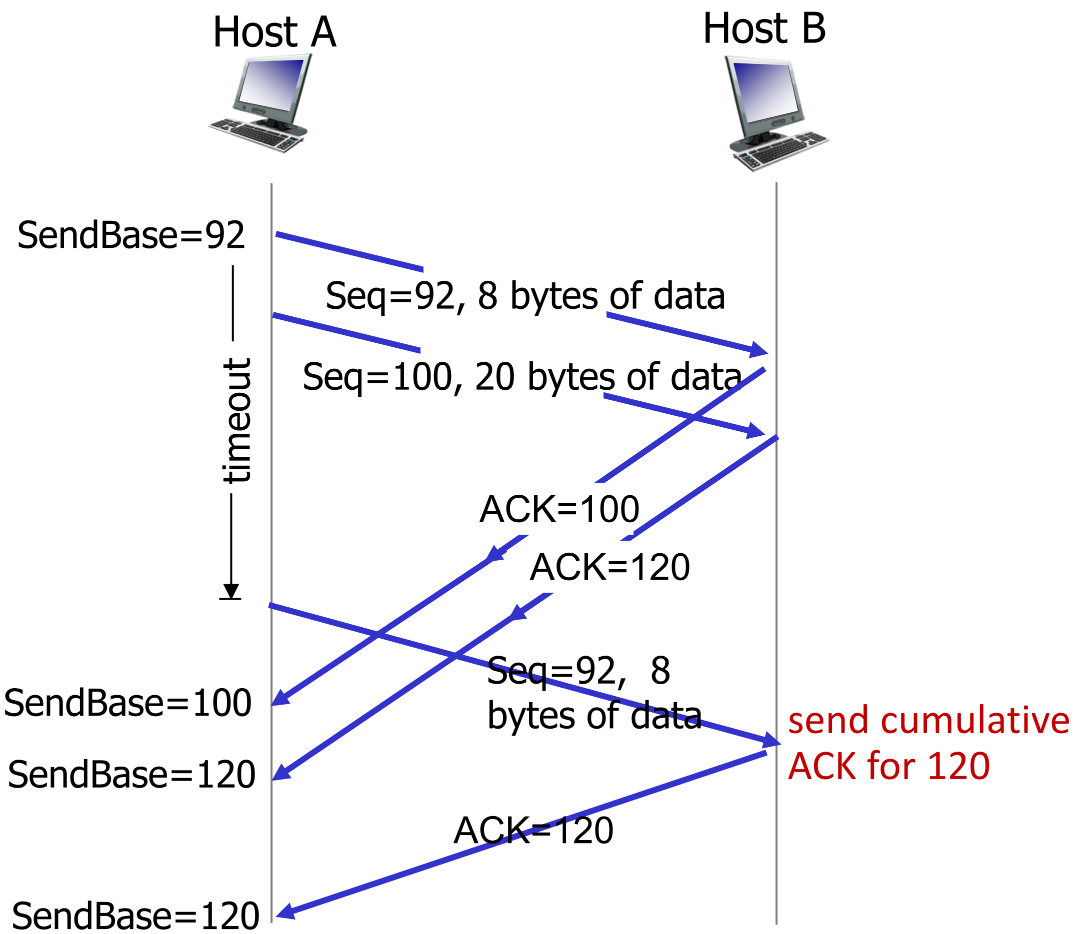
Security

- TCP sequence number randomization is a technique that aims to make TCP sequence numbers unpredictable and hard to guess by attackers. This can improve network security by reducing the chances of TCP spoofing attacks

TCP: retransmission scenarios

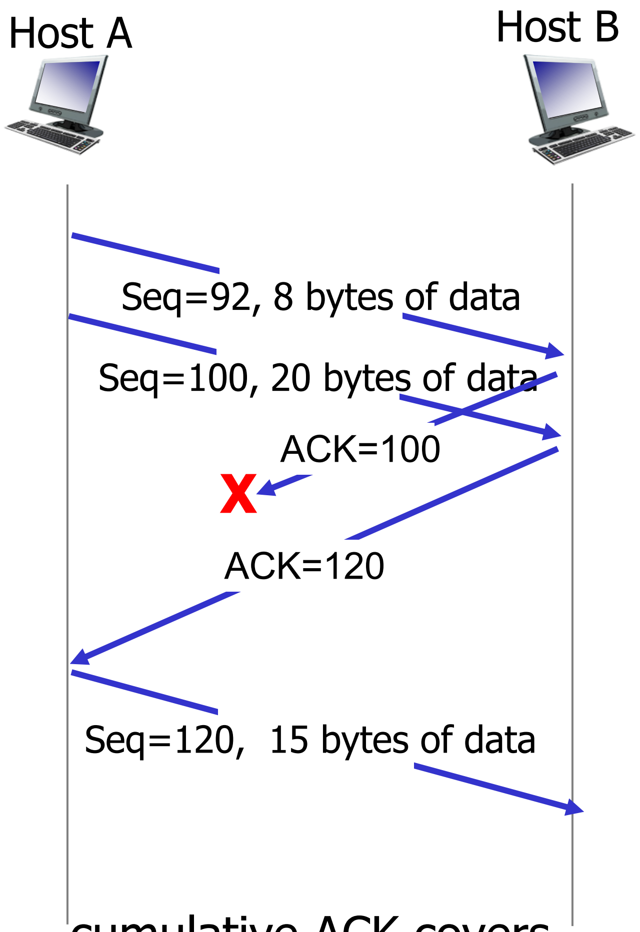


lost ACK scenario



premature timeout

TCP: retransmission scenarios



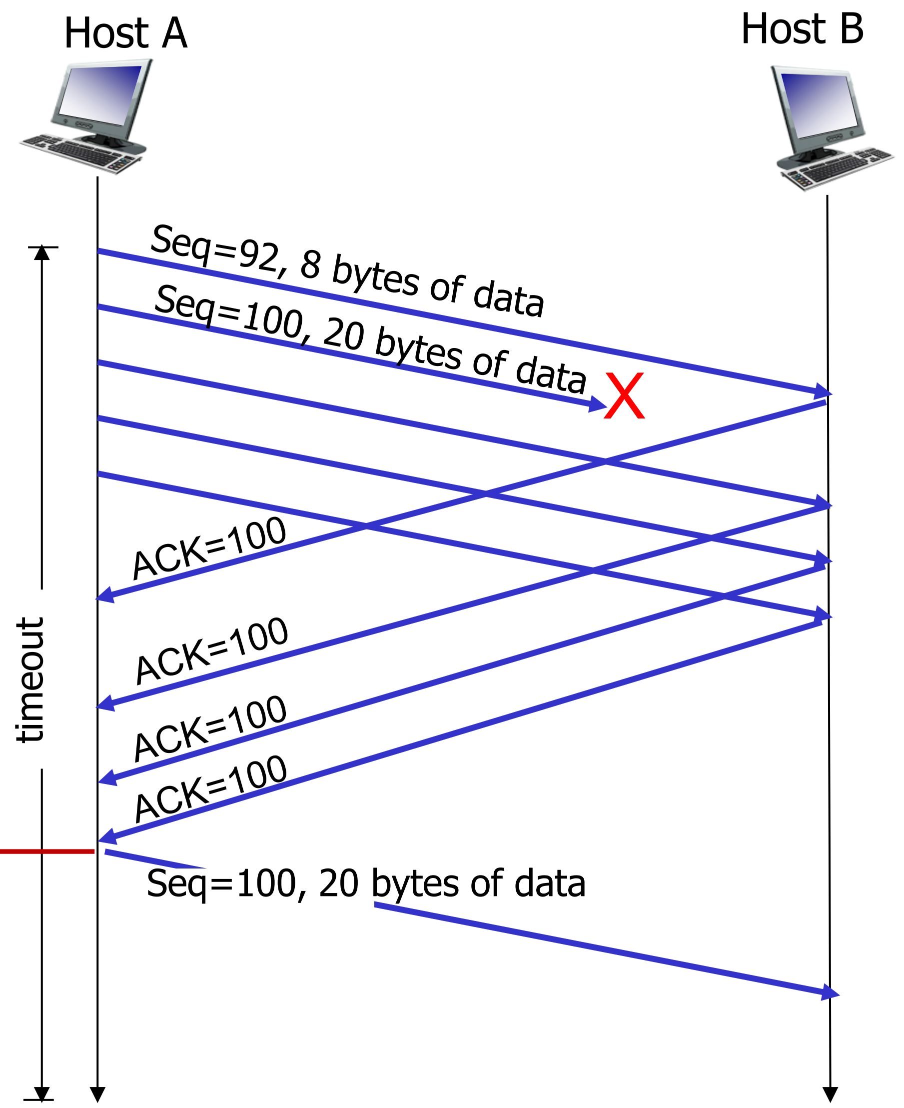
cumulative ACK covers
for earlier lost ACK

TCP fast retransmit

TCP fast retransmit

if sender receives 3 additional ACK's for same data ("triple duplicate ACK's"), resend unACK'ed segment with smallest seq #

- likely that unACK'ed segment lost, so don't wait for timeout



Receipt of three duplicate ACK's indicates 3 segments received after a missing segment – lost segment is likely. So, retransmit!

TCP Sender (simplified)

event: data received from application

- create segment with seq number
- seq number is byte-stream number of first data byte in segment
- start timer if not already running
  - for oldest unACKed segment
  - expiration interval: **`TimeOutInterval`**

event: timeout

- retransmit segment that caused timeout
- restart timer

event: ACK received

- update what is known to be ACKed segments
- start timer if there are still unACKed segments

TCP Receiver: ACK generation [RFC 5681]

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

3.5 Connection-oriented transport: TCP

1. TCP overview
2. Segment structure
3. The TCP connection
4. Opening and closing TCP connections
5. Reliable data transfer
6. TCP round trip time, timeout
7. Flow control

3.5.6 TCP round trip time, 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

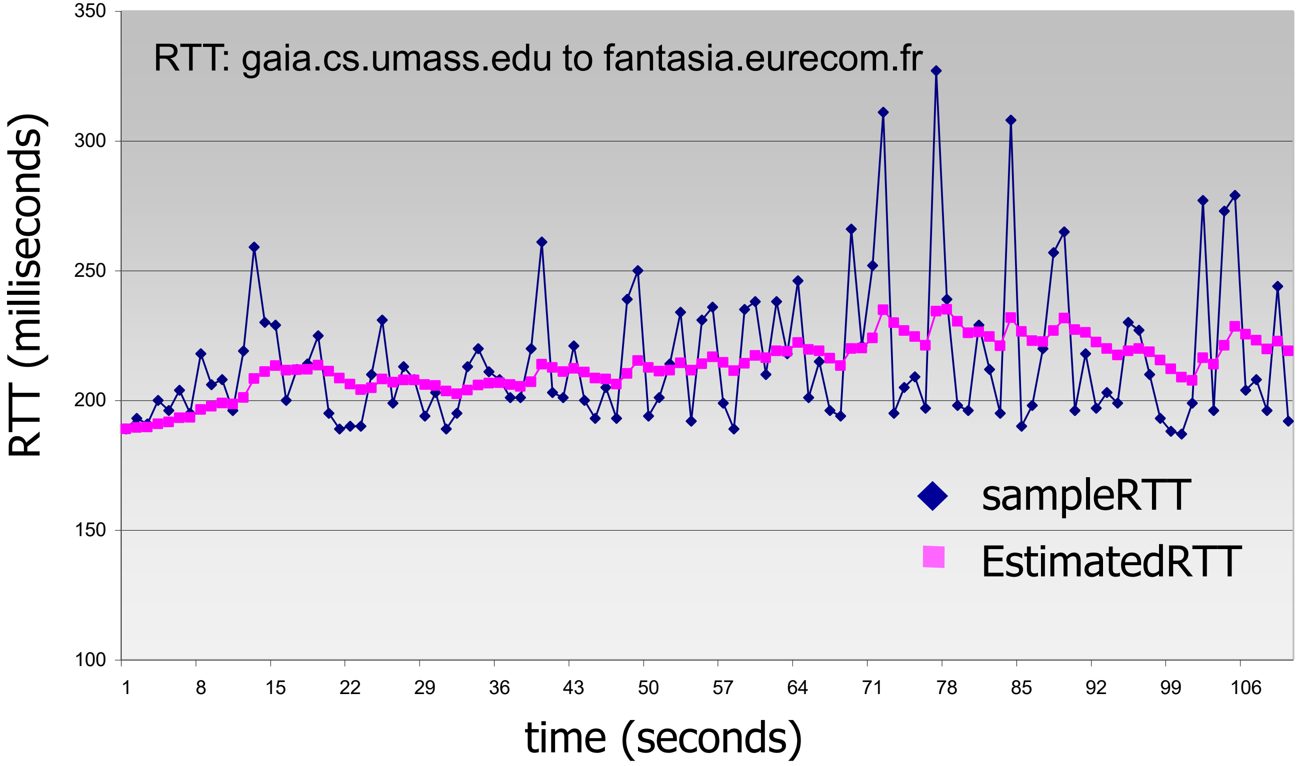
Q: how to estimate RTT?

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

TCP round trip time, timeout

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

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus “safety margin”
  - large variation in **`EstimatedRTT`**: want a larger safety margin

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

estimated RTT “safety margin”

- **`DevRTT`**: EWMA of **`SampleRTT`** deviation from **`EstimatedRTT`**:

$$\texttt{DevRTT = (1-}\beta\texttt{)*DevRTT + }\beta\texttt{*}\underbrace{\texttt{|SampleRTT - EstimatedRTT|}}_{\text{deviation, difference}}$$

(typically, $\beta = 0.25$)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

3.5 Connection-oriented transport: TCP

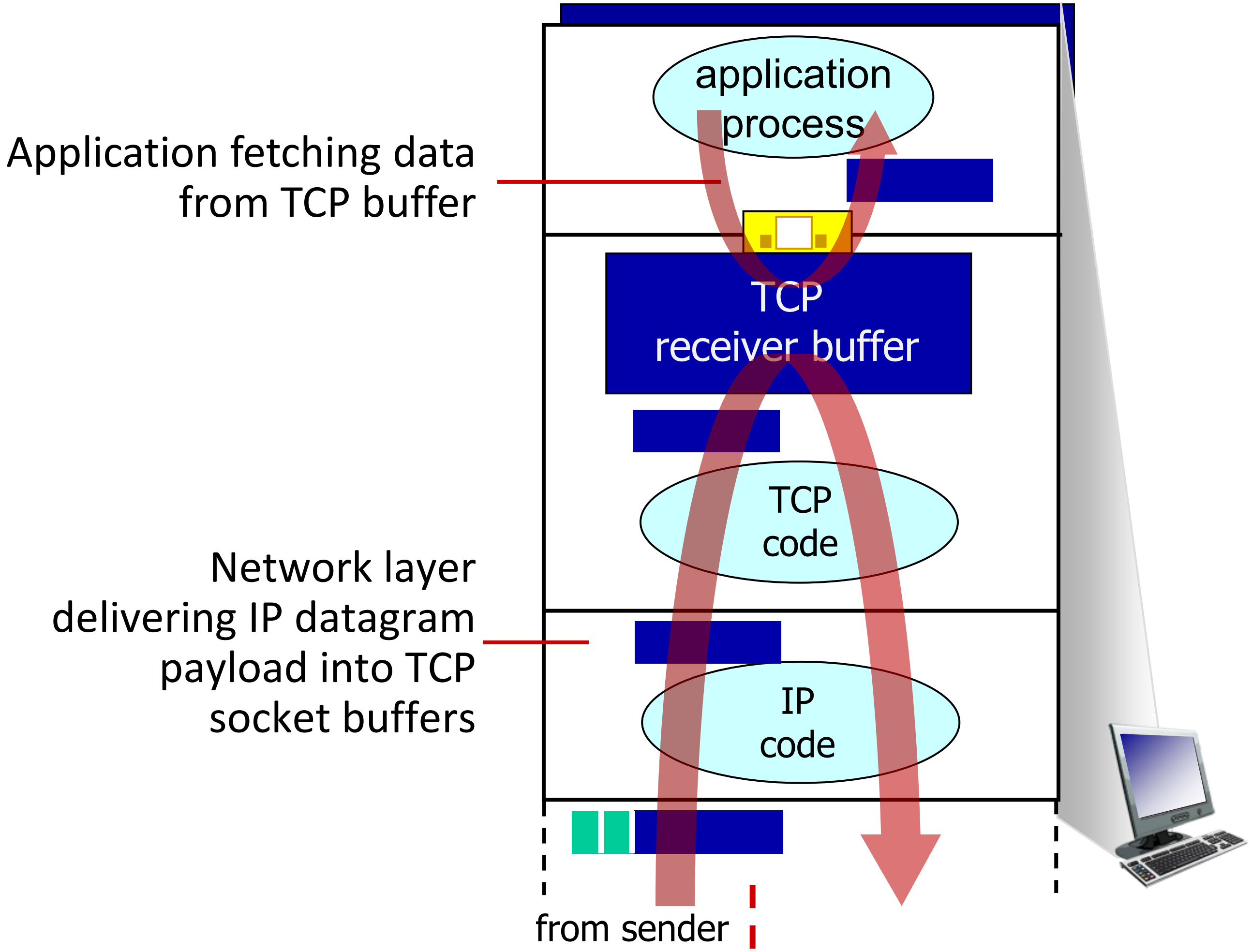
1. TCP overview
2. Segment structure
3. The TCP connection
4. Opening and closing TCP connections
5. Reliable data transfer
6. TCP round trip time, timeout
7. Flow control

3.5.7 TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from receiver buffer?

flow control

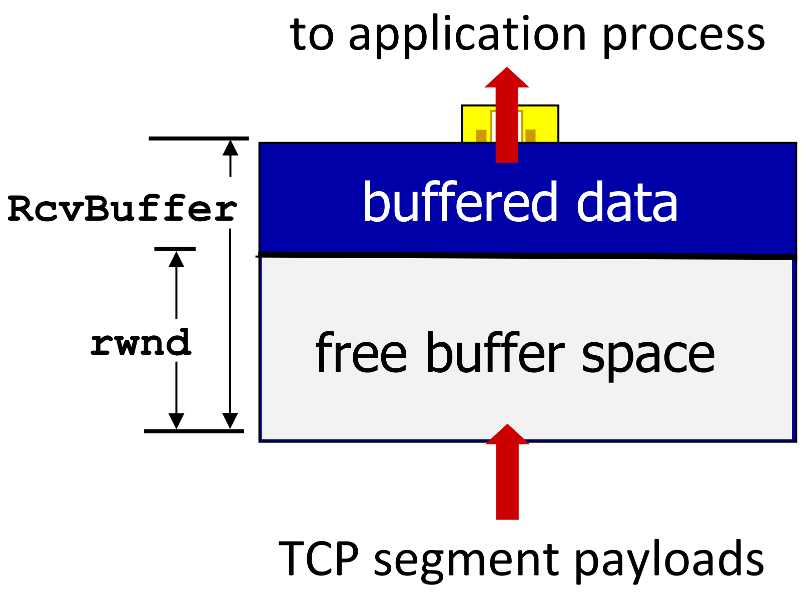
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast



receiver protocol stack

TCP flow control

- TCP receiver "advertises" free buffer space in *receive window* (**`rwnd`**) field in TCP header of ACK packets
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto adjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow
- *receive window* is 16 bit --> 65 535 bytes

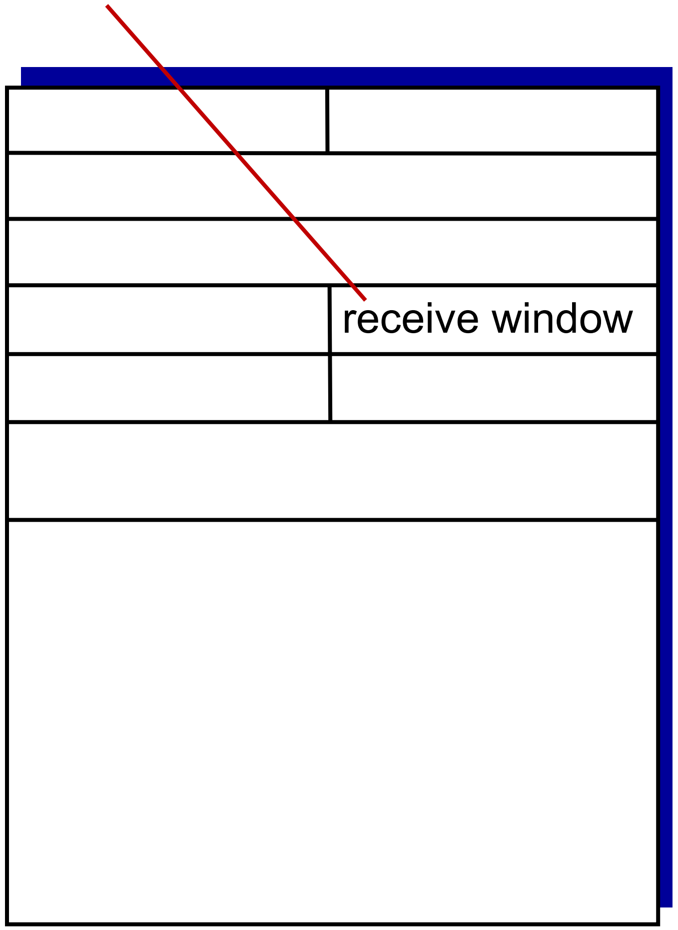


TCP receiver-side buffering

TCP flow control

- TCP receiver "advertises" free buffer space in *receive window* (**`rwnd`**) field in TCP header of ACK packets
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto adjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow
- *receive window* is 16 bit --> 65 535 bytes

flow control: # bytes receiver willing to accept



TCP segment format

Chapter 3: roadmap

3.1 Transport-layer services

3.2 Port numbers

3.3 Connectionless transport: UDP

UDP socket programming

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

~~3.6 Principles of congestion control~~

3.7 TCP congestion control

3.8 QUIC: Quick UDP Internet Connections

Principles of congestion control

Congestion:

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

congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver

Chapter 3: roadmap

3.1 Transport-layer services

3.2 Port numbers

3.3 Connectionless transport: UDP

UDP socket programming

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

~~3.6 Principles of congestion control~~

3.7 TCP congestion control

3.8 QUIC: Quick UDP Internet Connections

TCP congestion control: AIMD

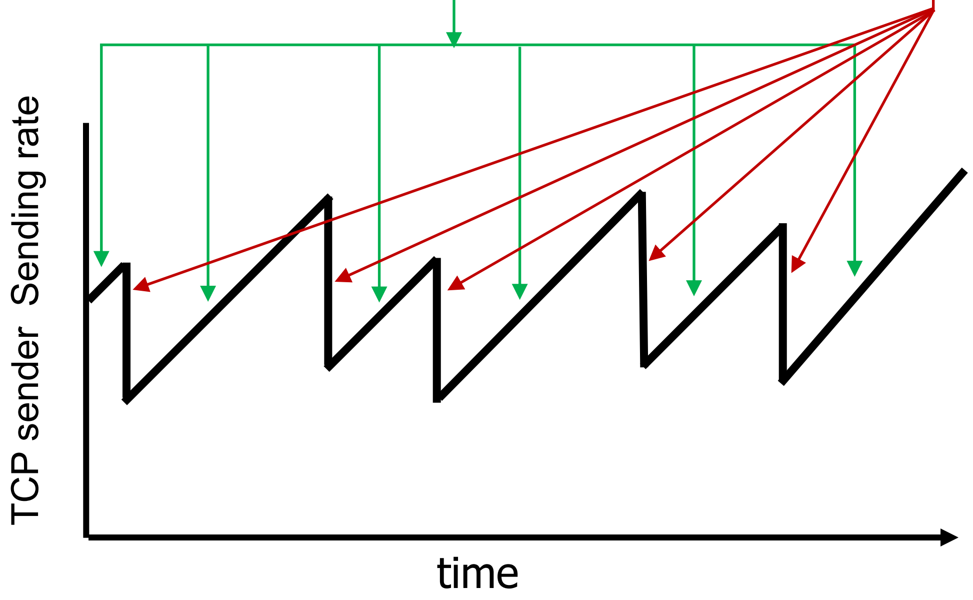
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

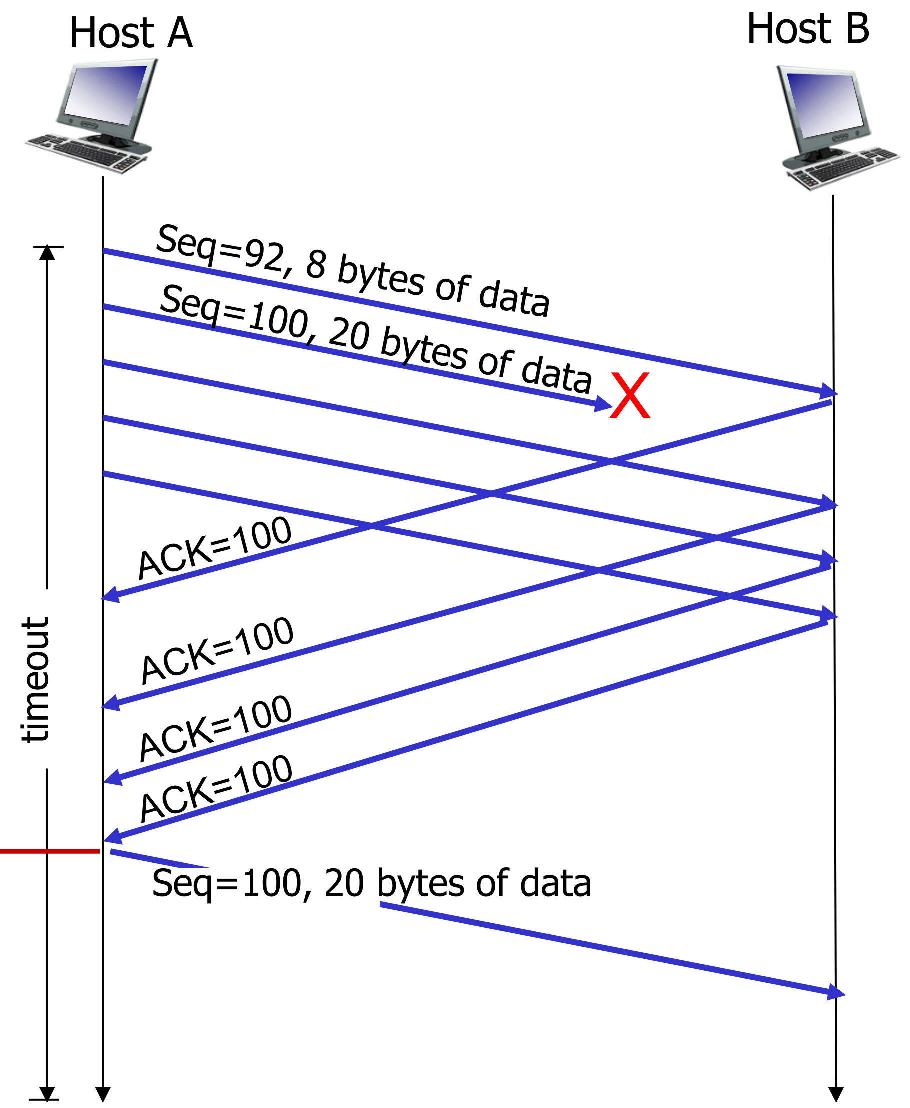
- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

TCP fast retransmit

TCP fast retransmit

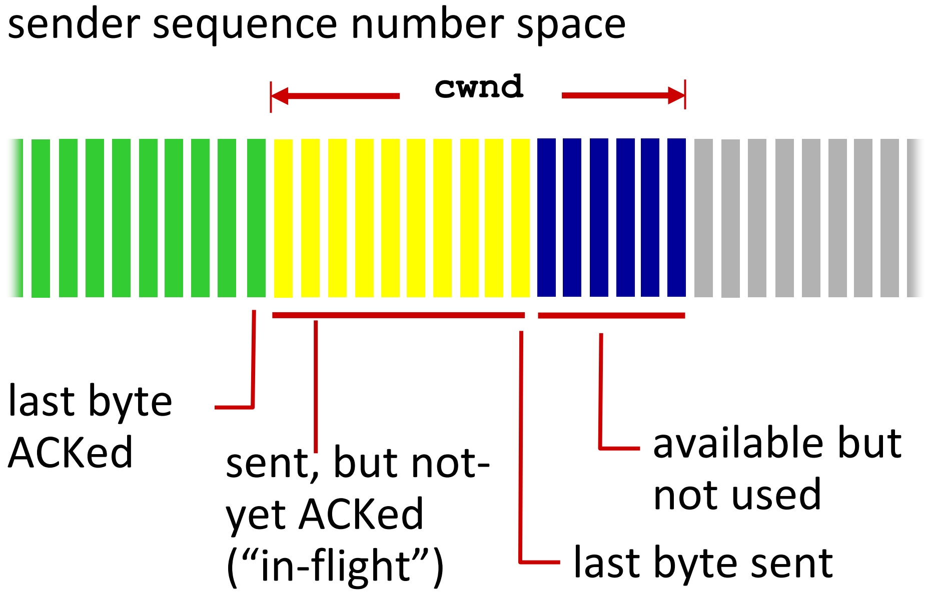
if sender receives 3 additional ACK's for same data ("triple duplicate ACK's"), resend unACK'ed segment with smallest seq #

- likely that unACK'ed segment lost, so don't wait for timeout



Receipt of three duplicate ACK's indicates 3 segments received after a missing segment – lost segment is likely. So, retransmit!

TCP congestion control: details



TCP sending behavior:

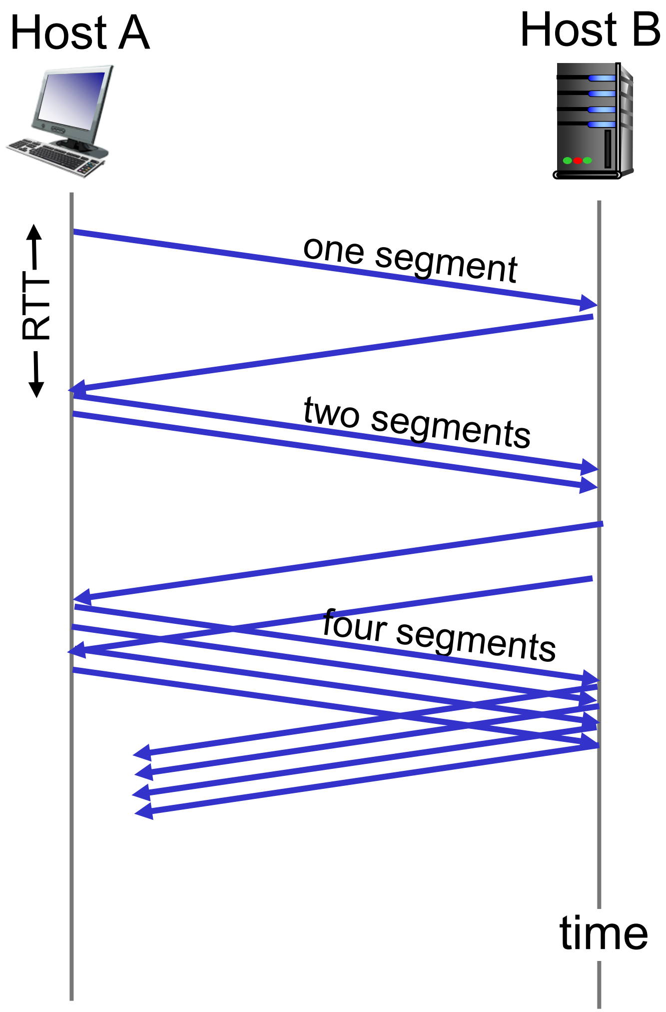
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

- TCP sender limits transmission: **`LastByteSent - LastByteAcked ≤ cwnd`**
- **`cwnd`** is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

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

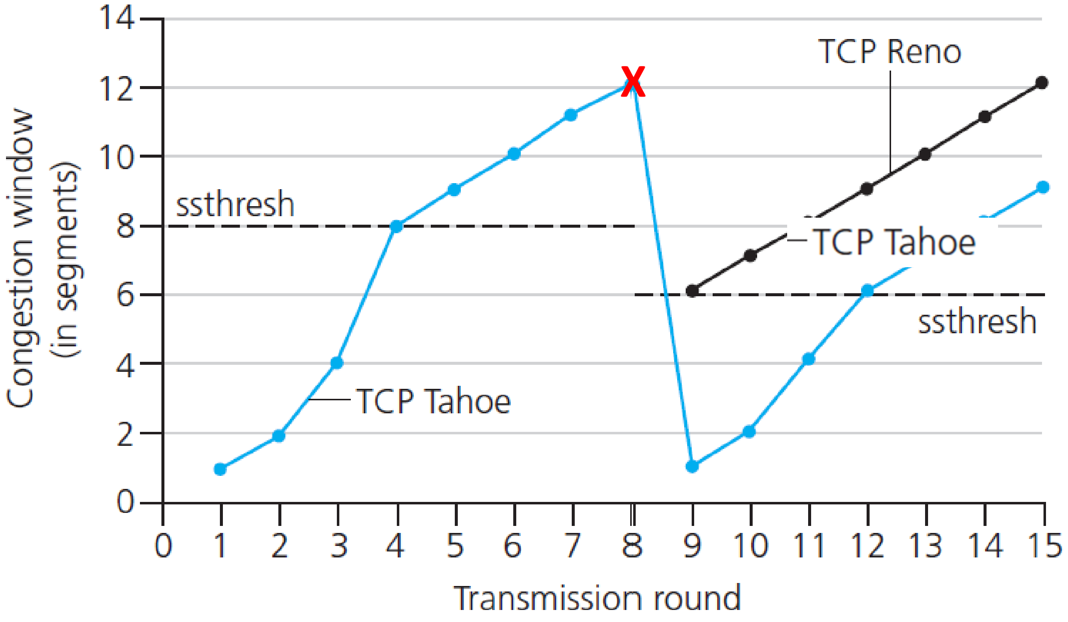


TCP: from slow start to congestion avoidance

- TCP Tahoe: Cut to 1 MSS (maximum segment size) when loss detected by timeout
- TCP Reno: Cut in half on loss detected by triple duplicate ACK

Q: when should the exponential increase switch to linear?

A: when **`cwnd`** gets to 1/2 of its value before timeout.



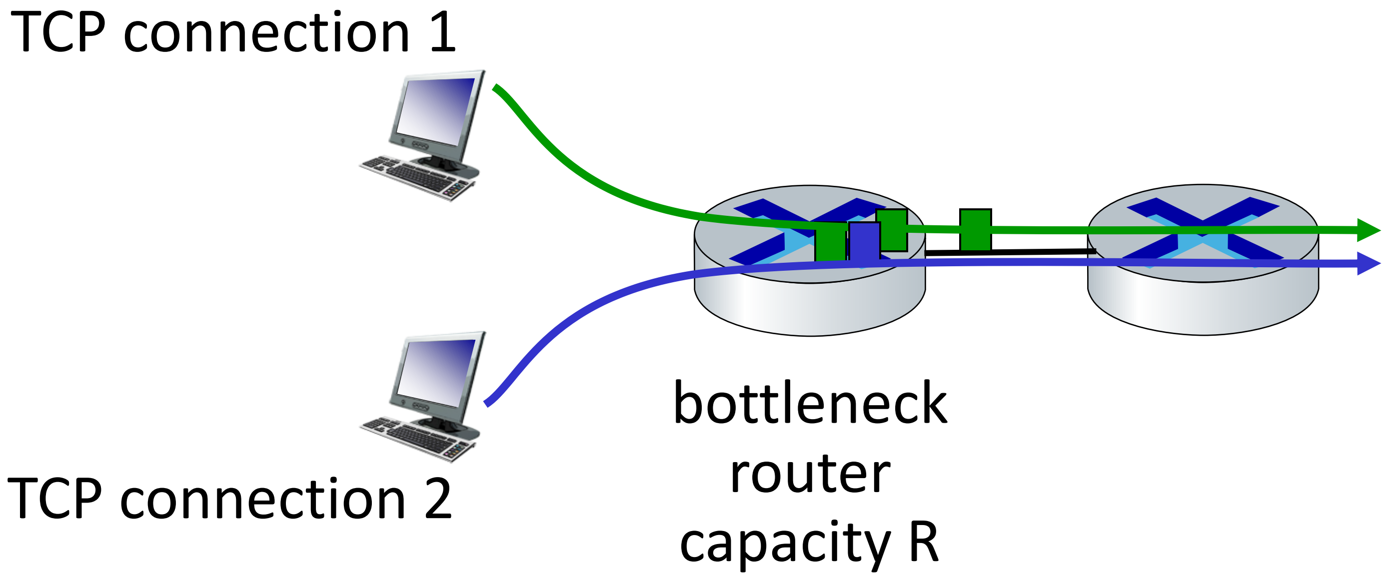
Implementation:

- variable **`ssthresh`**
- on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

TCP fairness

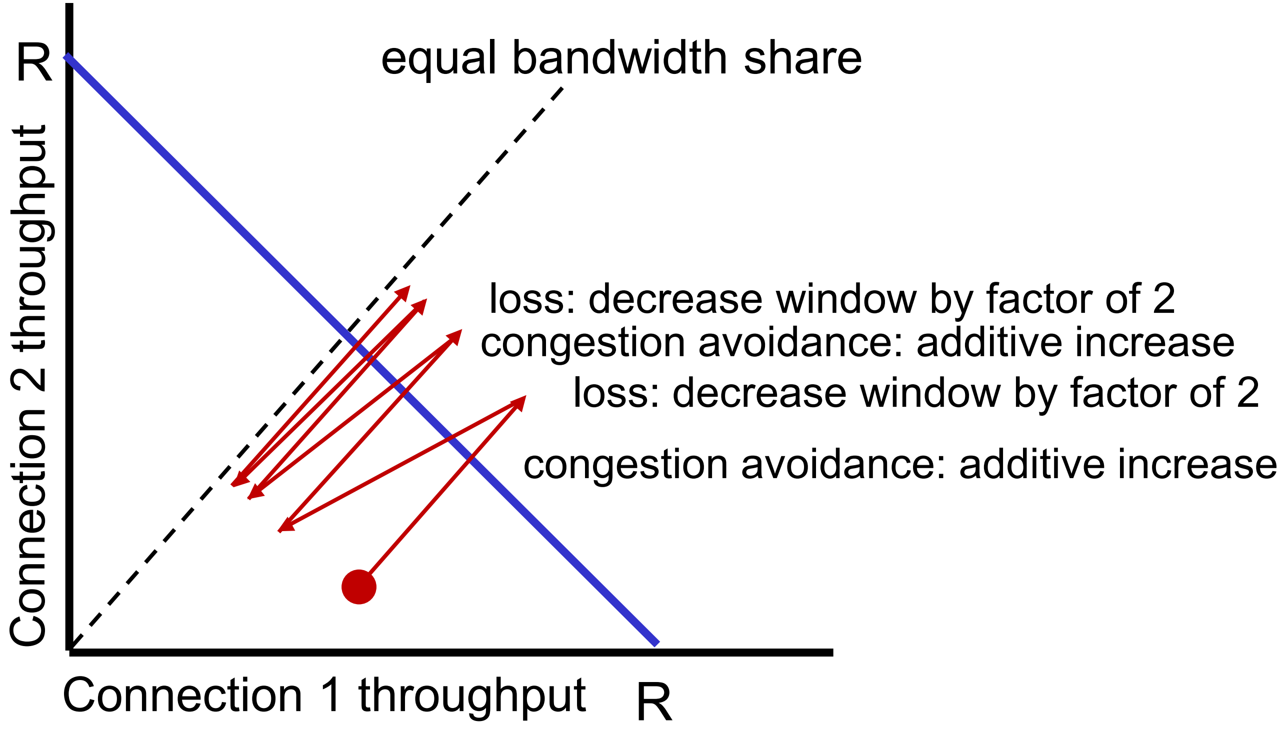
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Fairness: must all network apps be "fair"?

Fairness and UDP

- multimedia apps 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
- there is no "Internet police" policing use of congestion control

Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

Transport layer: roadmap

3.1 Transport-layer services

3.2 Port numbers

3.3 Connectionless transport: UDP

UDP socket programming

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

~~3.6 Principles of congestion control~~

3.7 TCP congestion control

3.8 QUIC: Quick UDP Internet Connections

QUIC: Quick UDP Internet Connections

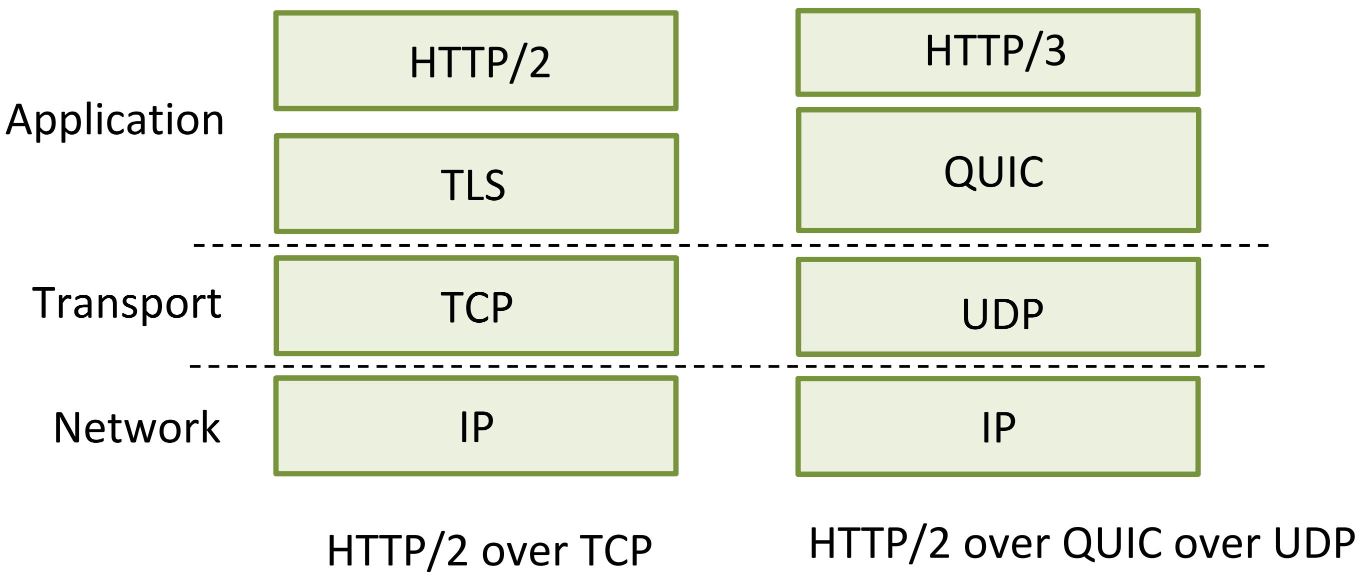
- TCP, UDP: principal transport protocols for 40 years
- different "flavors" of TCP developed, for specific scenarios:

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 application layer, on top of UDP
  - HTTP/3: QUIC

QUIC: Quick UDP Internet Connections

- QUIC is an application-layer protocol, on top of UDP
  - HTTP/3, increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)

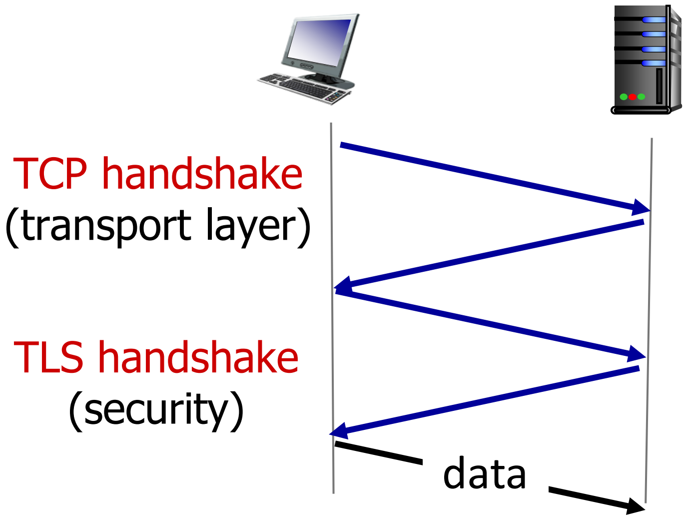


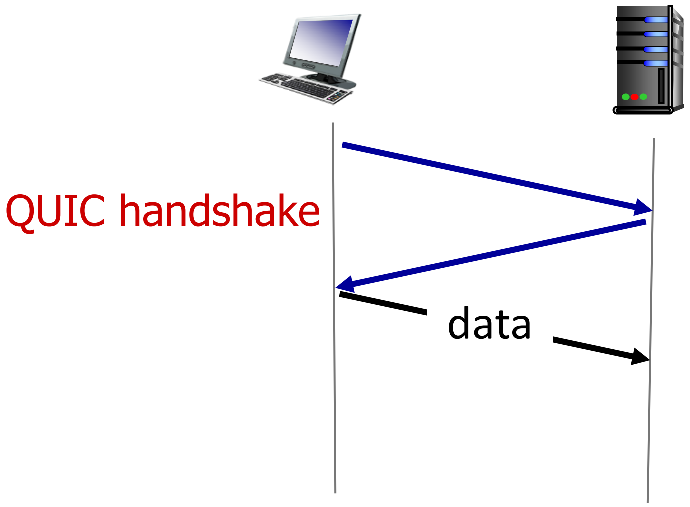
QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- Connection-oriented: Two endpoints
- Reliable data transfer: In-order packet delivery, retransmissions of lost packets (on application-level)
- Security: authentication, encryption
- Congestion control
- HTTP/3 multiple application-level "streams" of 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 or 3 serial handshakes

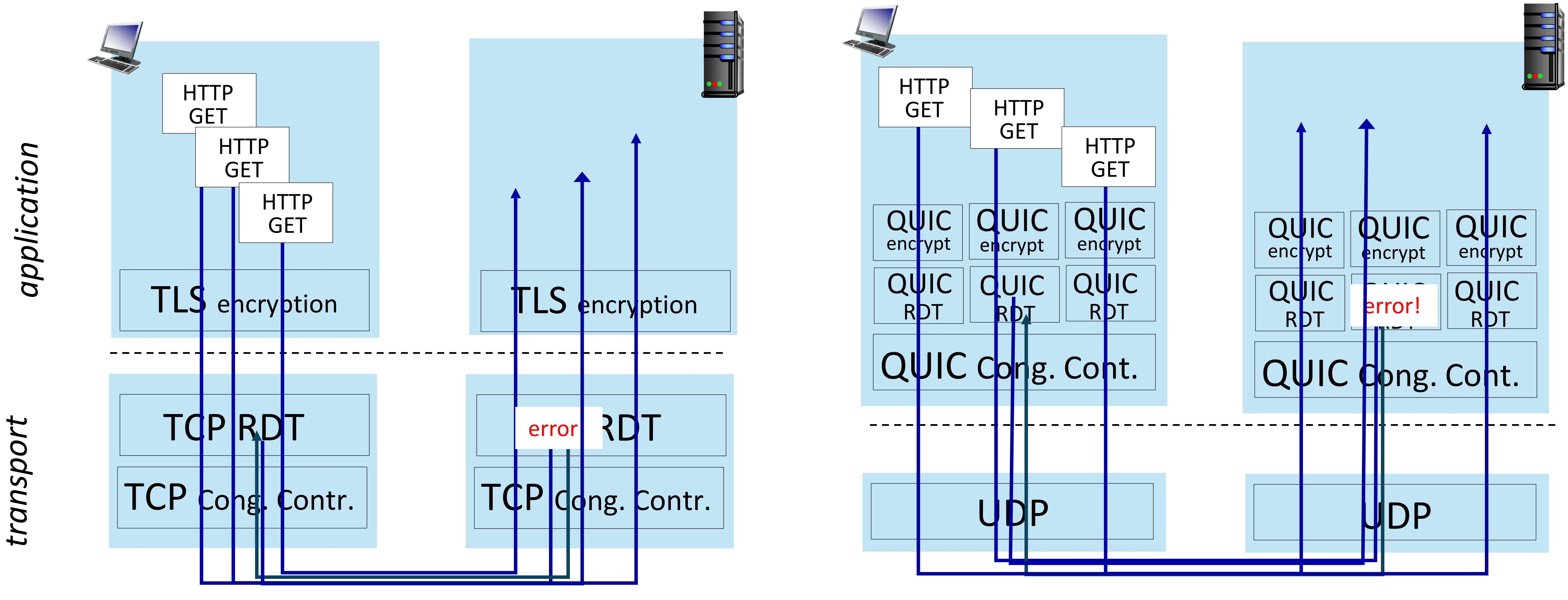
QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

HOL blocking problem

- HTTP/1.1 sending multiple HTTP requests, all over a single TCP connection
- Since TCP provides reliable, in-order byte delivery, this means that the multiple HTTP requests must be delivered in-order at the destination HTTP server.
- If a packet of one HTTP request are lost, TCP at the HTTP server cannot restore the remaining HTTP requests until the lost packet is retransmitted and correctly received
- A variant of the HOL blocking problem (Section 2.2.5)

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

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

Chapter 3: summary

- principles behind transport layer services:
  - port numbers
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

Up next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network-layer chapters:
  - data plane
  - control plane