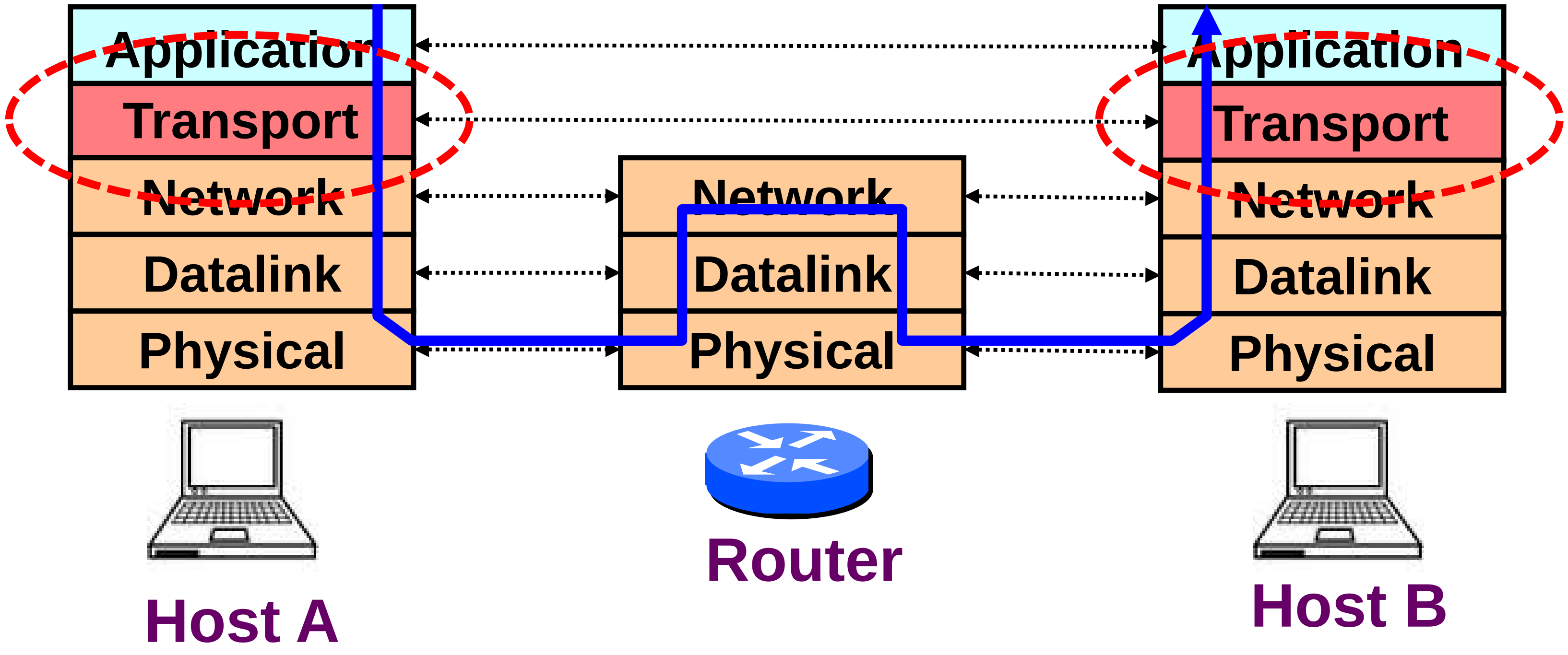
Topic 5 – Transport

Our goals:

- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
  - buffers
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control
  - TCP flow control

Transport Layer

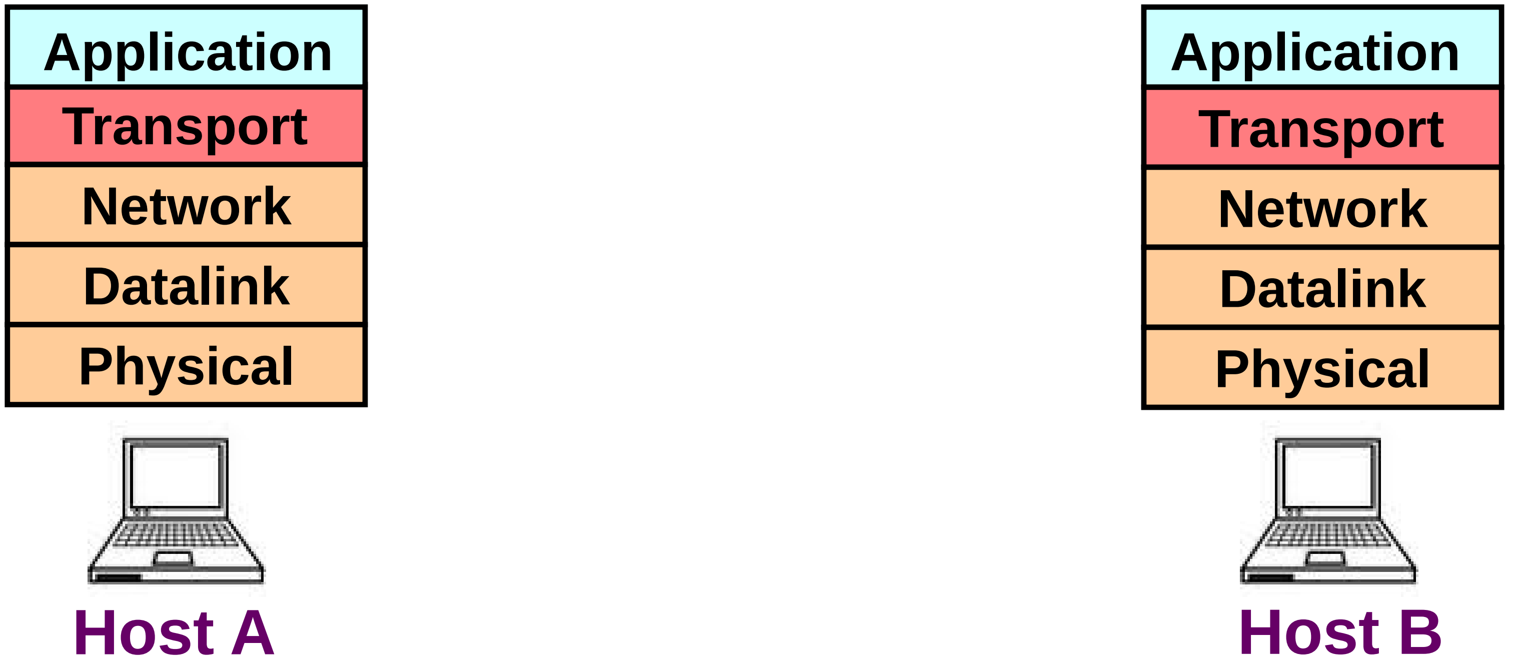
- Commonly a layer at end-hosts, between the application and network layer



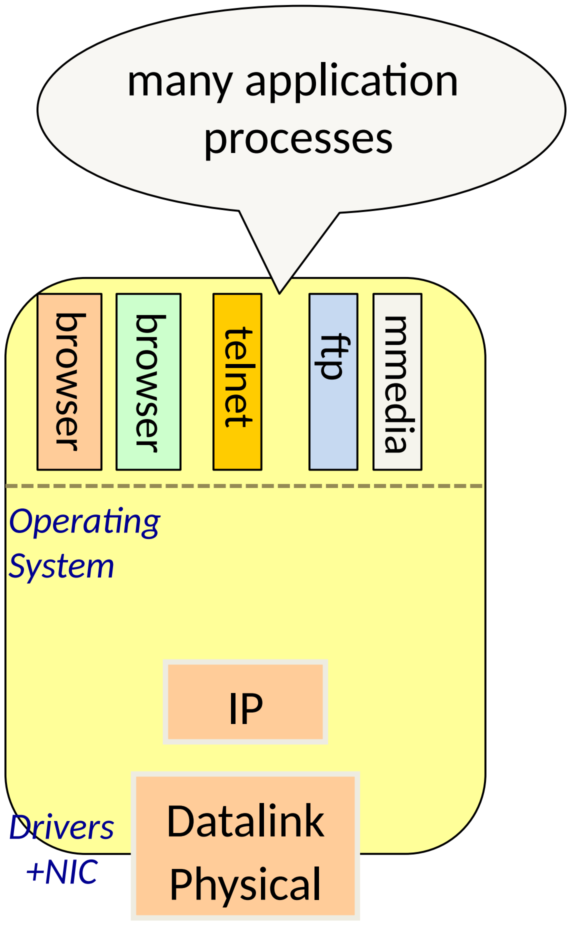
Why a transport layer - 1/3?

1) IP packets are addressed to a host, but end-to-end communication is between application/ processes/tasks at hosts
   - Need at least a way to decide which packets go to which applications (*further multiplexing*)

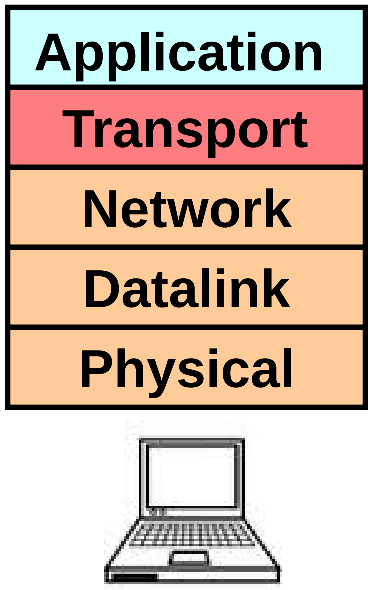
Why a transport layer - 1?



Why a transport layer - 1?

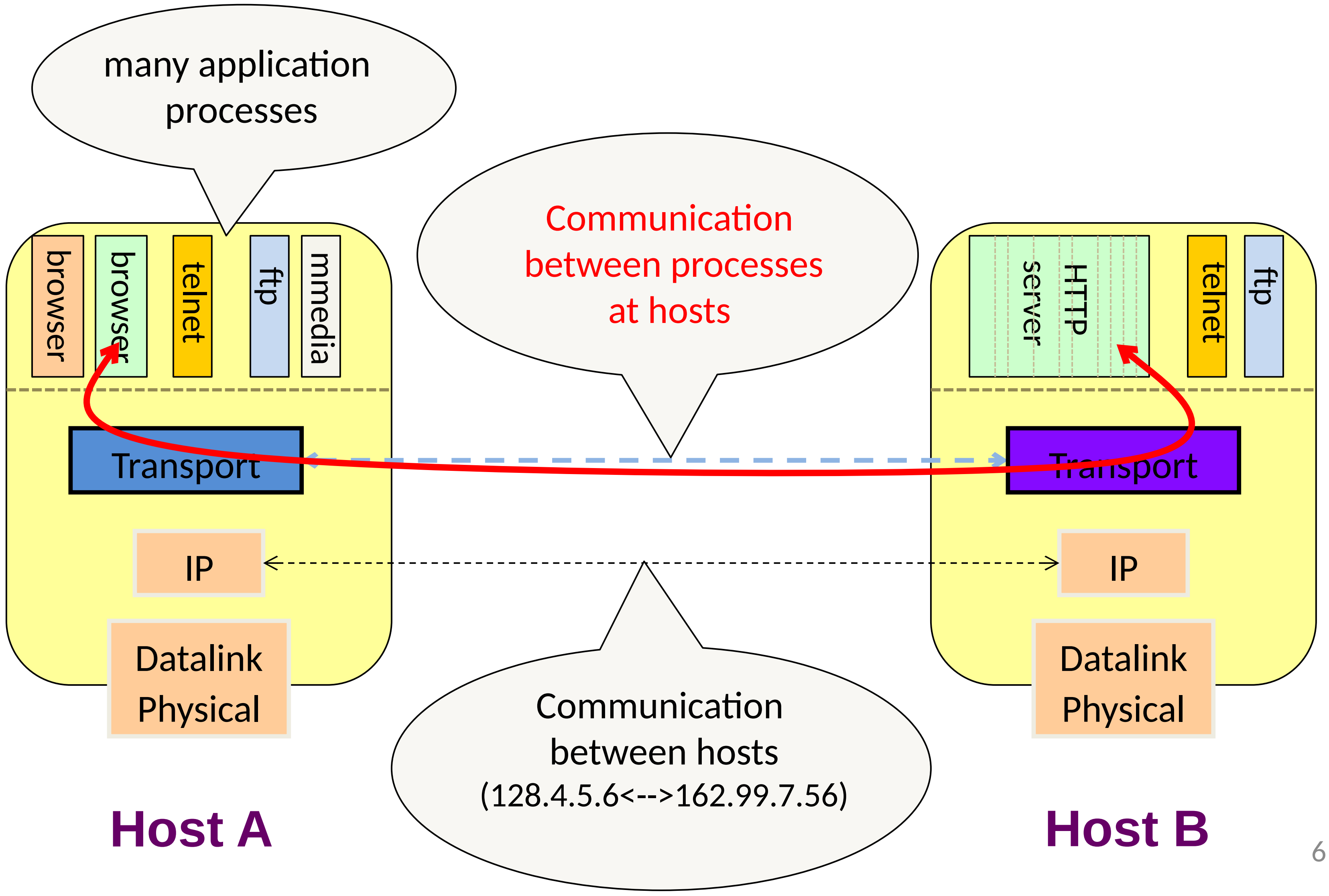


Host A



Host B

Why a transport layer - 1?



many application processes
browser
browser
telnet
ftp
mmedia
Transport
IP
Datalink
Physical
Communication between processes at hosts
HTTP server
telnet
ftp
Transport
IP
Datalink
Physical
Communication between hosts (128.4.5.6<-->162.99.7.56)

Host A

Host B

Why a transport layer - 2/3?

1) IP packets are addressed to a host, but end-to-end communication is between application processes at hosts
   - Need a way to decide which packets go to which applications (mux/demux).

- 2) IP provides a weak service model (*best-effort*)
   - Packets can be corrupted, delayed, dropped, reordered, duplicated.

Why a transport layer - 3/3?

1) IP packets are addressed to a host but end-to-end communication is between application processes at hosts:

2) IP provides a weak service model (*best-effort*)
- Packets can be corrupted, delayed, dropped, reordered, duplicated.

3) IP gives no guidance on how much traffic to send and when
- Dealing with this is tedious for application developers
- We need ‘rate matching’ or ‘flow control’
- Flow control also needed to avoid buffer overrun or congestion control (for reason 2).

Role of the Transport Layer

- Communication between application processes
  - Multiplexing between application processes
  - Implemented using *ports*

Role of the Transport Layer

- Communication between application processes
- Provide common end-to-end services for app layer [optional]
  - Reliable, in-order data delivery
  - Paced data delivery: flow and congestion-control
    - too fast may overwhelm the network
    - too slow is not efficient

(Just like Computer Networking lectures....)

Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
  - also SCTP, MTCP, SST, RDP, DCCP, …

Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- UDP is a minimalist, no-frills transport protocol
  - only provides mux/demux capabilities

Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- UDP is a minimalist, no-frills transport protocol
- TCP is the *totus porcus* protocol
  - offers apps a reliable, in-order, byte-stream abstraction
  - with congestion control
  - but **no** performance (delay, bandwidth, ...) guarantees

Context: Applications and Sockets

- Socket: software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system
  - socketID = socket(…, socket.TYPE)
  - socketID.sendto(message, …)
  - socketID.recvfrom(…)

- Two important types of sockets
  - UDP socket: TYPE is SOCK_DGRAM
  - TCP socket: TYPE is SOCK_STREAM

A socket and a websocket are similar.
Websockets are a stack violation using tunnelling.

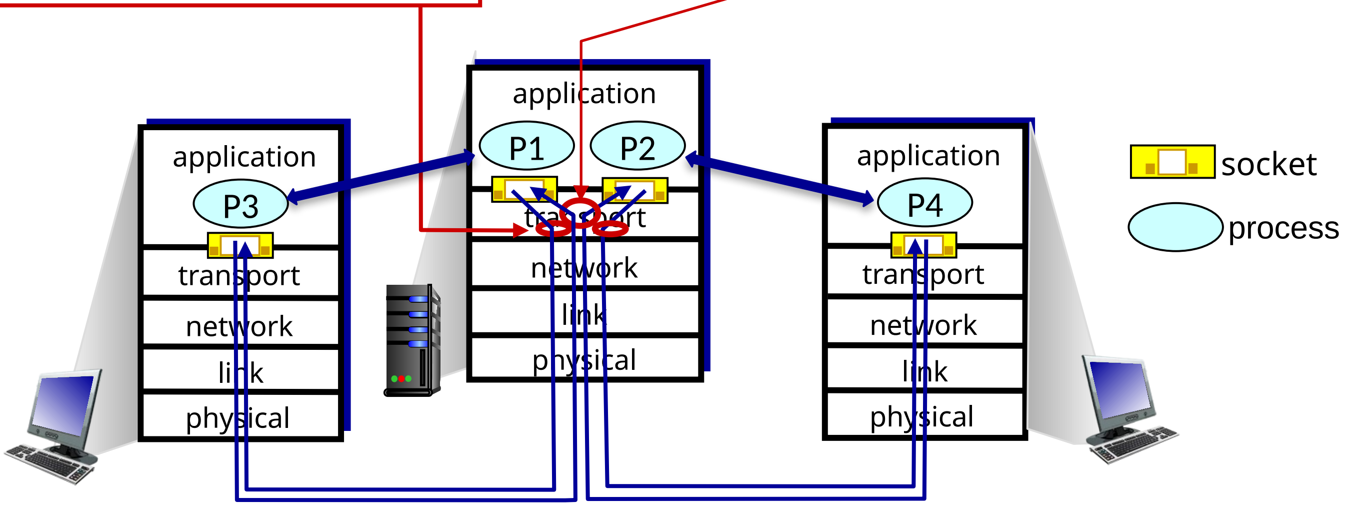
Multiplexing/demultiplexing

multiplexing as sender:

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

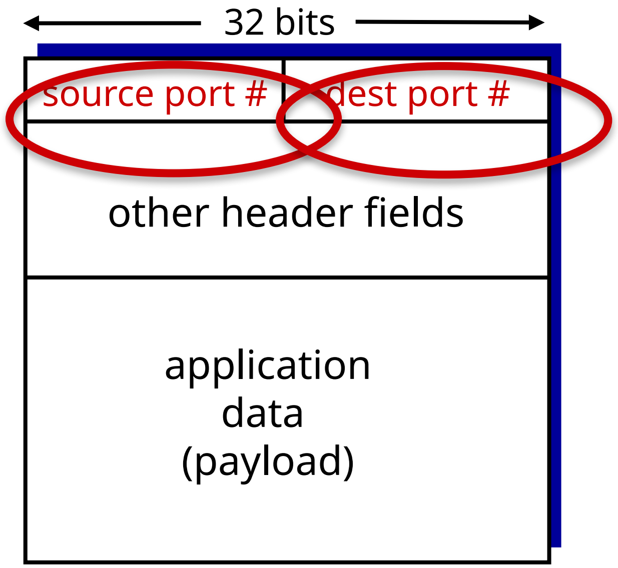
demultiplexing as receiver:

use header info to deliver received segments to correct socket



How demultiplexing Works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one 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

- when creating socket, must specify *host-local* port #:

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

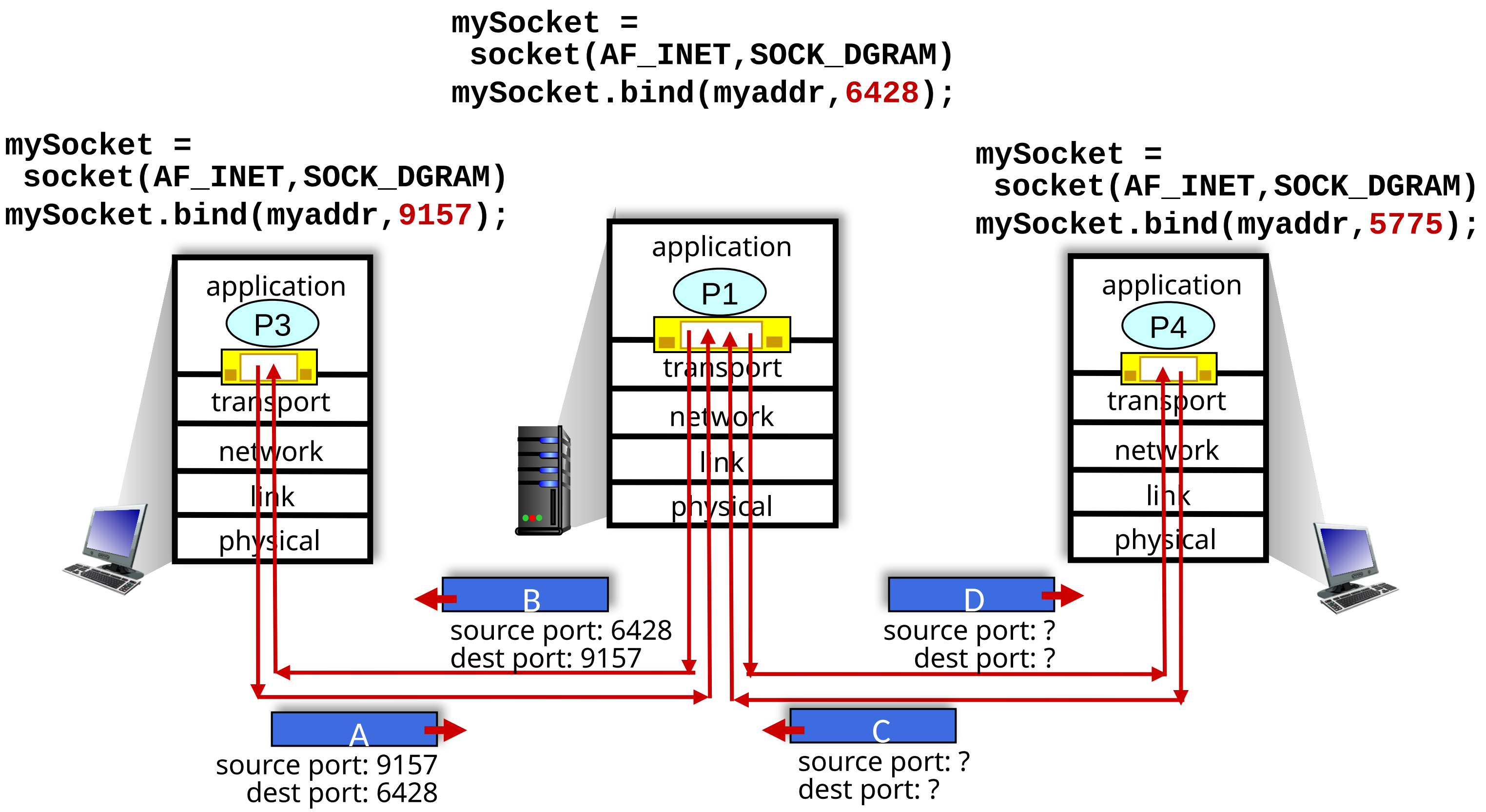
- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #

IP/UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Connectionless demultiplexing: an example



Connection-oriented demultiplexing

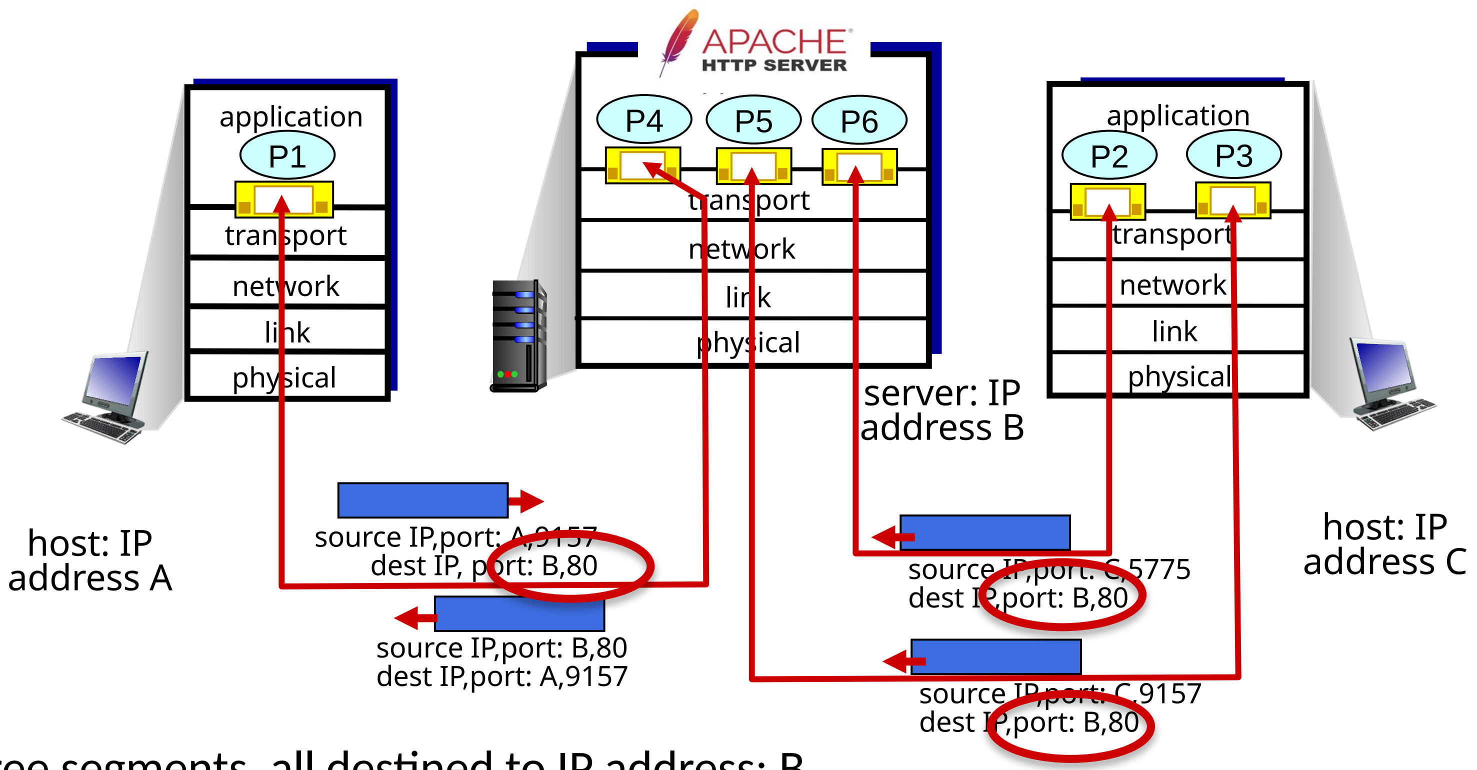
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses *all four values* *(4-tuple)* to direct segment to appropriate socket

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

slight lie alert.... I should say that a common network tuple has FIVE values

- source IP address
- source port number
- dest IP address
- dest port number AND
- *protocol* e.g. TCP (6) or UDP (17)

Connection-oriented demultiplexing: example



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

Ports

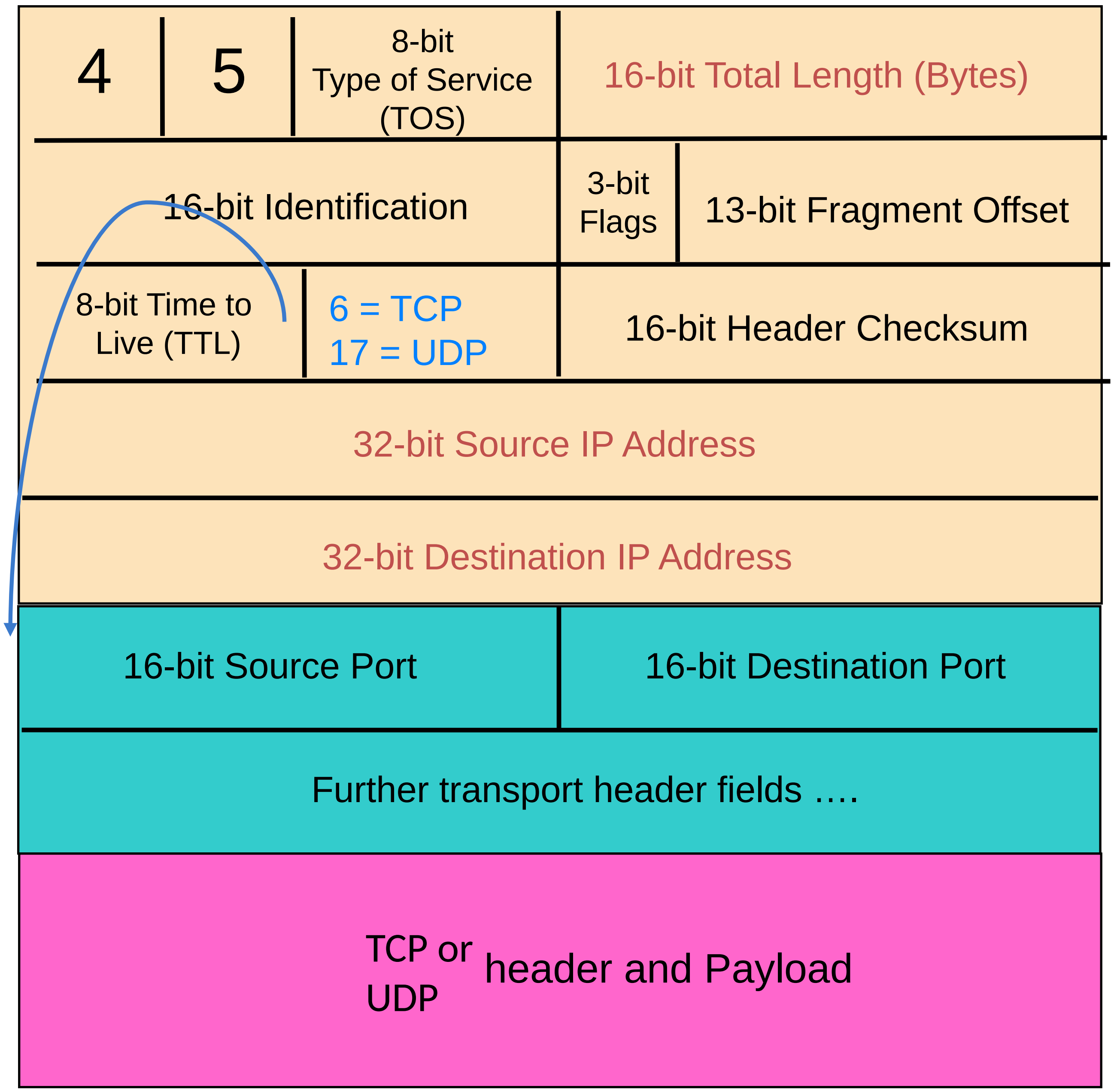
- Problem: deciding which application (or socket) gets which packets

- Solution: ***port*** as a transport layer identifier
  - 16 bit identifier
    - O/S stores mapping between sockets and *ports*
    - a packet carries a source and destination port number in its transport layer header

- For UDP ports (SOCK_DGRAM)
  - O/S stores (local port, local IP address) <--> socket

- For TCP ports (SOCK_STREAM)
  - O/S stores (local port, local IP, remote port, remote IP) <--> socket

- At O/S application layer, ports are typically owned exclusively by one process, but one process may open many ports.
- O/S may have sockets not connected to IP ports (eg for local IPC communications or other network protocol (non-IP) support).


4
5
8-bit Type of Service (TOS)
16-bit Total Length (Bytes)
16-bit Identification
3-bit Flags
13-bit Fragment Offset
8-bit Time to Live (TTL)
6 = TCP
17 = UDP
16-bit Header Checksum
32-bit Source IP Address
32-bit Destination IP Address
16-bit Source Port
16-bit Destination Port
Further transport header fields ….
TCP or UDP
header and Payload

Summary

- Multiplexing, demultiplexing: based on segment (L4) and datagram (L3) header field values.
- **UDP:** demultiplexing using destination port number (only).
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers.

- (Multiplexing/demultiplexing typically also happening at *any* and all other layers)

A socket and a websocket are similar.
Websockets are a stack violation using tunnelling.

More on Ports

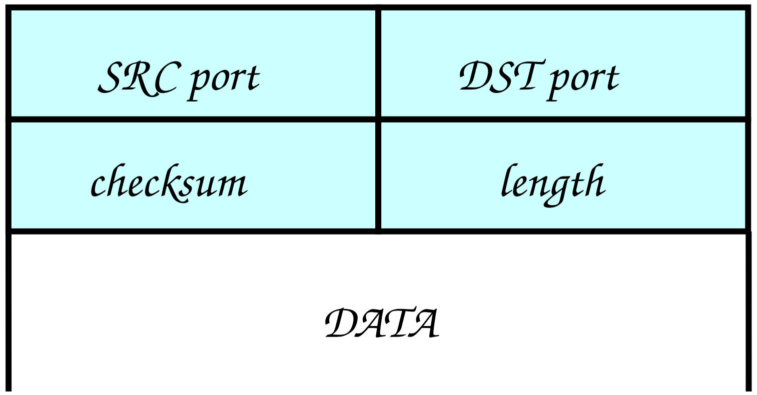
- Separate 16-bit port address space for UDP and TCP
- "Well known" ports (0-1023): everyone agrees which services run on these ports
  - e.g., ssh:22, http:80, https:443
  - hence a client knows which port to seek on a server
- Ephemeral ports (most 1024-65535)
  - dynamically allocated as the source port for a client's connection.

O/S protection may prevent an arbitrary user listening on a low port no.

UDP: User Datagram Protocol

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery

- UDP described in RFC 768 – (1980!)
  - Destination IP address and port to support demultiplexing
  - Optional error checking on the packet contents
    - (*checksum* field of 0 means “don’t verify checksum”) ***not in IPv6!***
    - ((this idea of optional checksum is removed in IPv6))

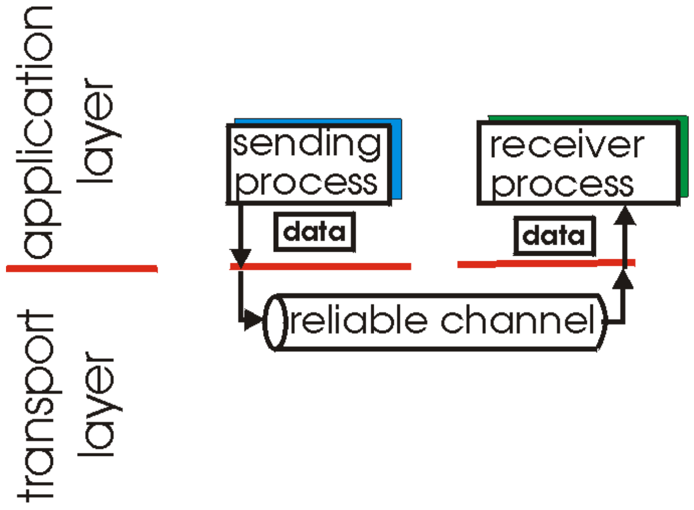


Why a transport layer?

- IP packets are addressed to a host but, end-to-end communication is between application processes at hosts
  - Need a way to decide which packets go to which applications (mux/demux).
- IP provides a weak service model (*best-effort*)
  - Packets can be corrupted, delayed, dropped, reordered, duplicated.

Principles of Reliable Data Transfer

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



(a) provided service

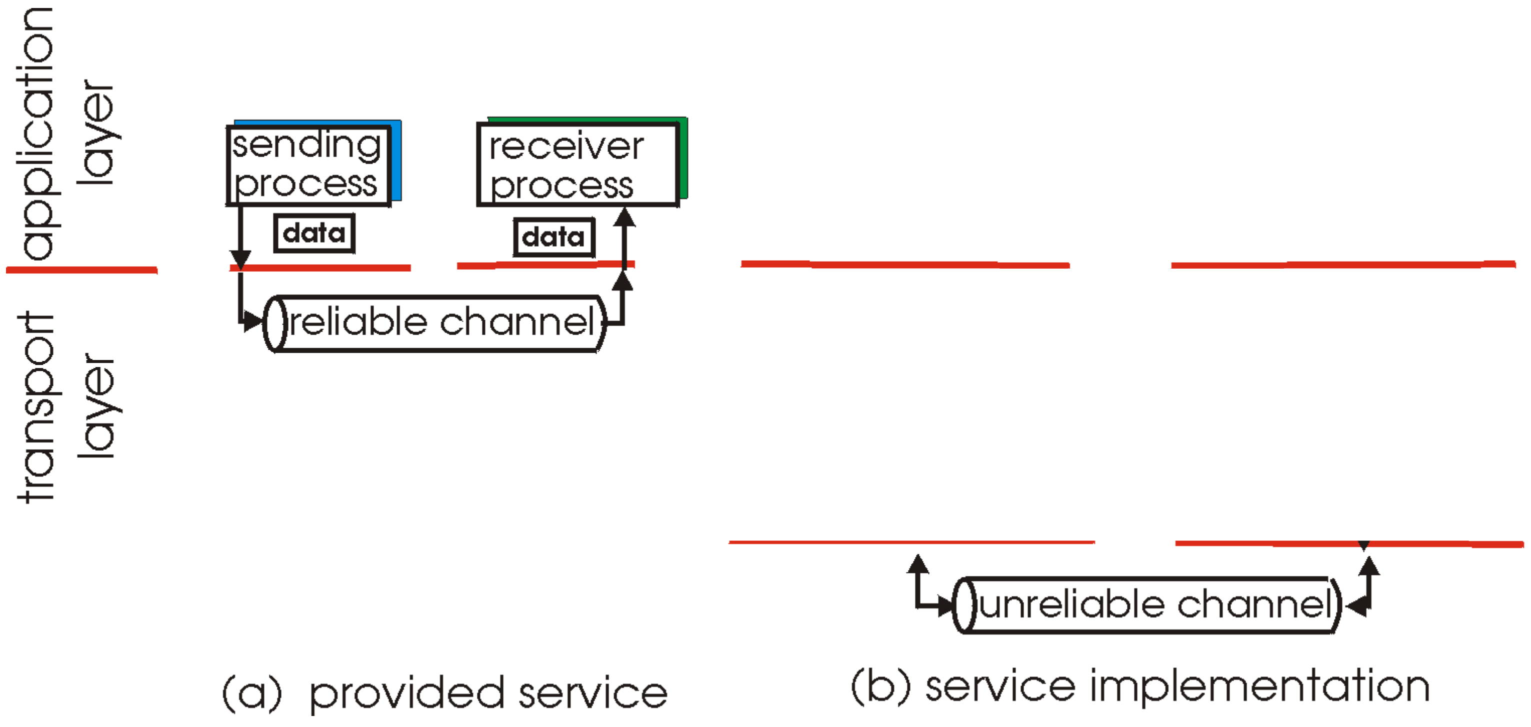
- In a perfect world, reliable transport is easy

But the Internet default is *best-effort*

- All the bad things best-effort can do
  - a packet is corrupted (bit errors)
  - a packet is lost
  - a packet is delayed (*why?*)
  - packets are reordered (*why?*)
  - a packet is duplicated (*why?*)

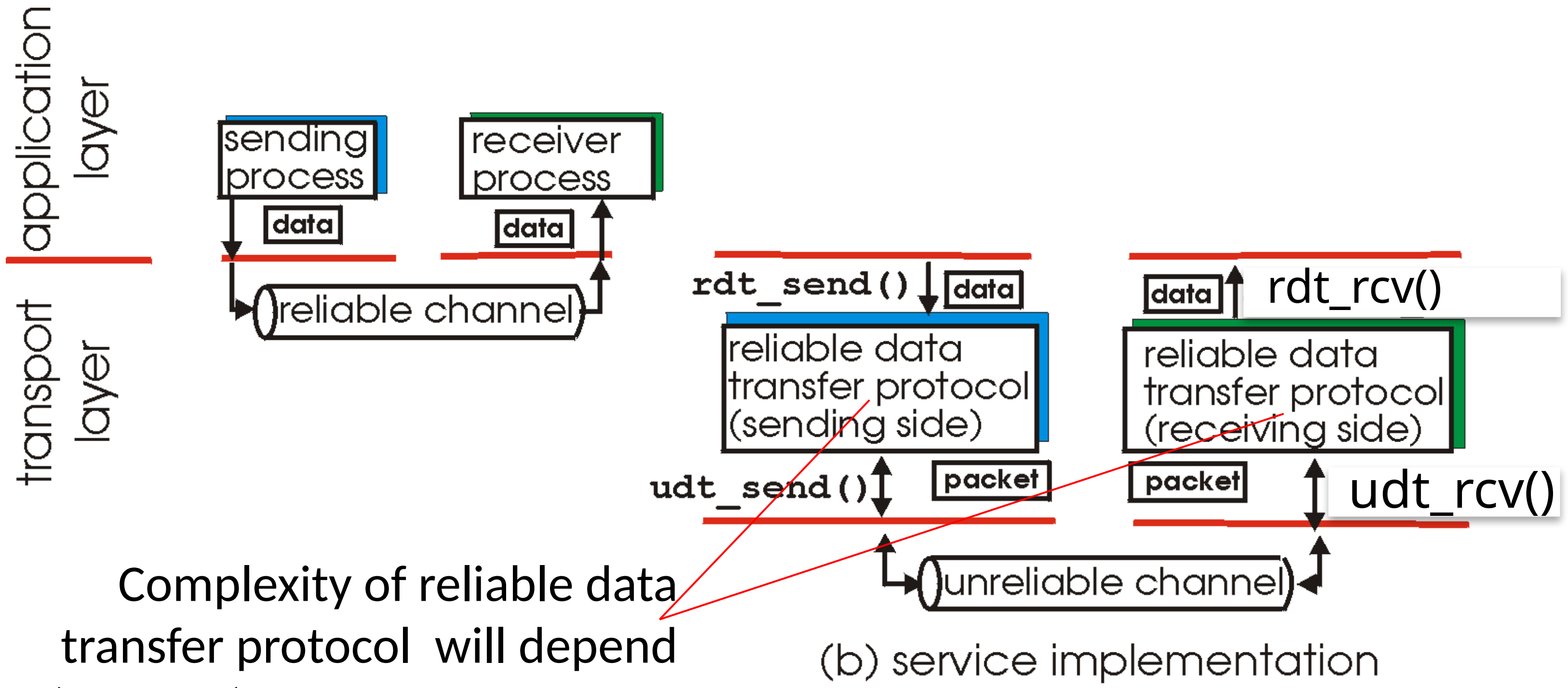
Principles of Reliable data transfer

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



Principles of Reliable data transfer

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

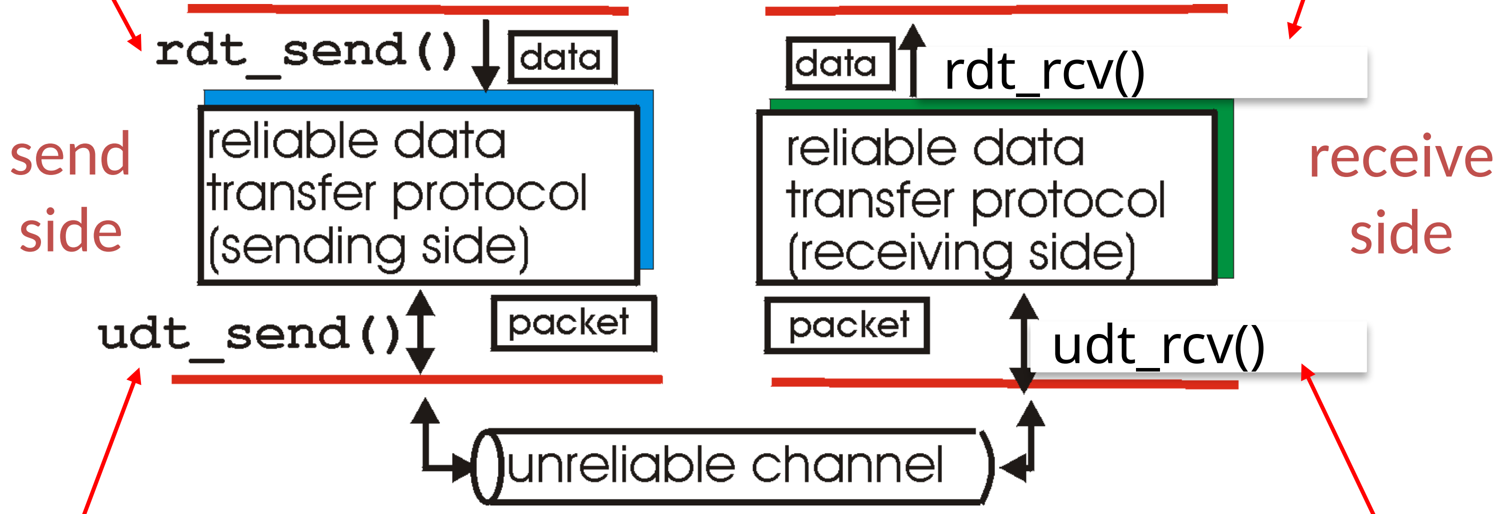


Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

rdt_rcv(): called by **rdt** to deliver data to upper



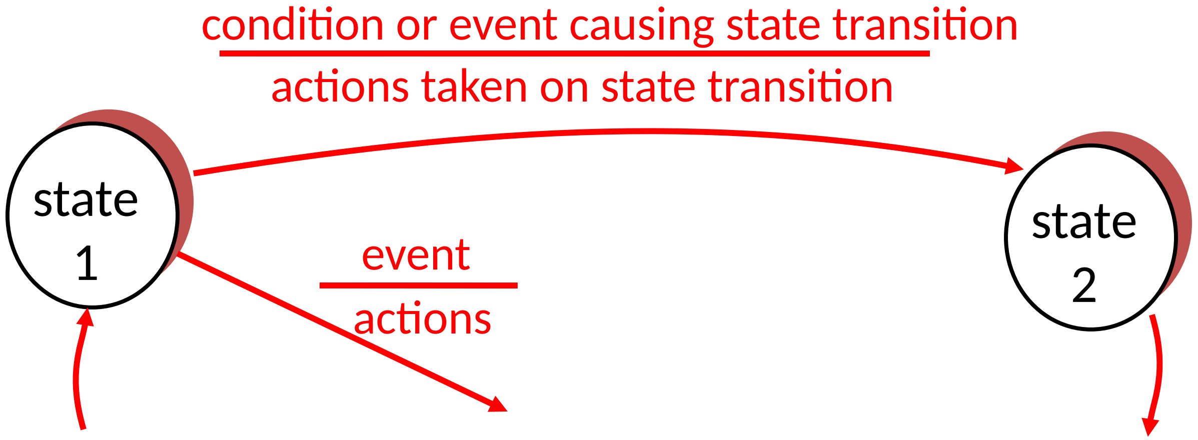
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

udt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

We'll:

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


condition or event causing state transition
actions taken on state transition
state 1
state 2
event
actions

state: when in this "state" next state uniquely determined by next event

KR state machines – a note.

Beware

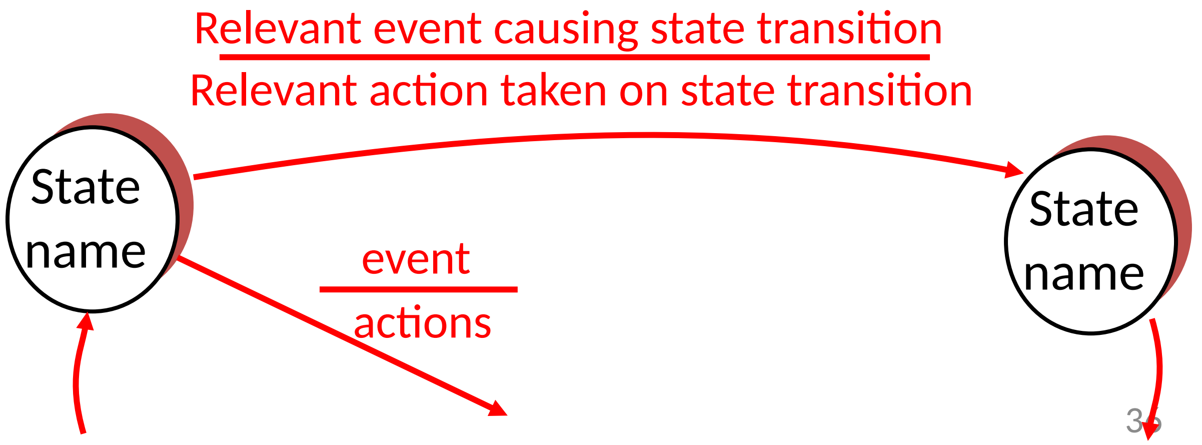
Kurose and Ross has a confusing/confused attitude to state-machines.

I've attempted to normalise the representation.

UPSHOT: these slides have differing information to the KR book (from which this RDT narrative is taken.)

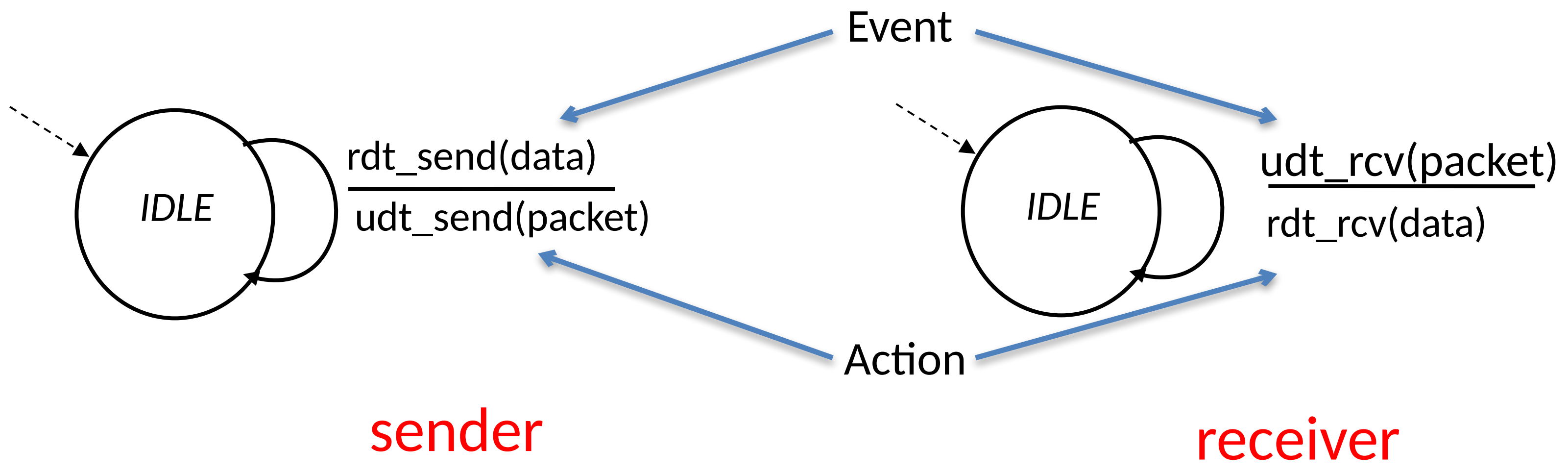
in KR "actions taken" appear wide-ranging, my interpretation is more specific/relevant.

state: when in this "state" next state uniquely determined by next event


Relevant event causing state transition
Relevant action taken on state transition
State name
event
actions
State name

Rdt1.0: reliable transfer over a reliable channel

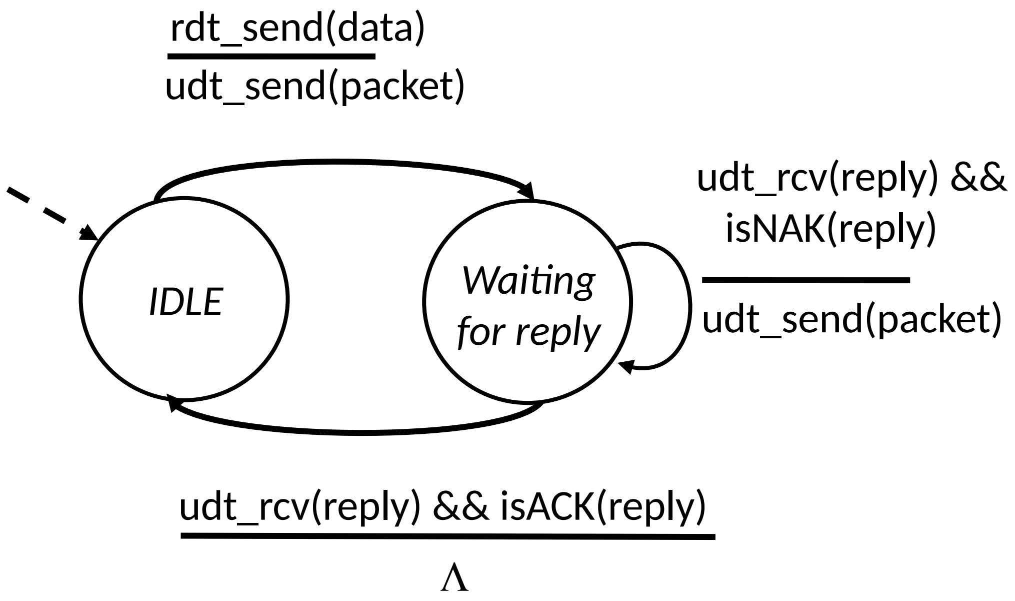
- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



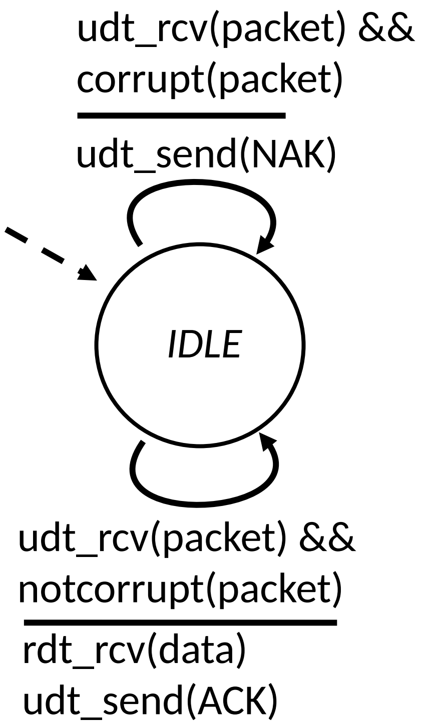
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - Checksum or CRC will be used to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that packet received is OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that packet had errors
  - sender retransmits packet on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) receiver->sender

rdt2.0: FSM specification



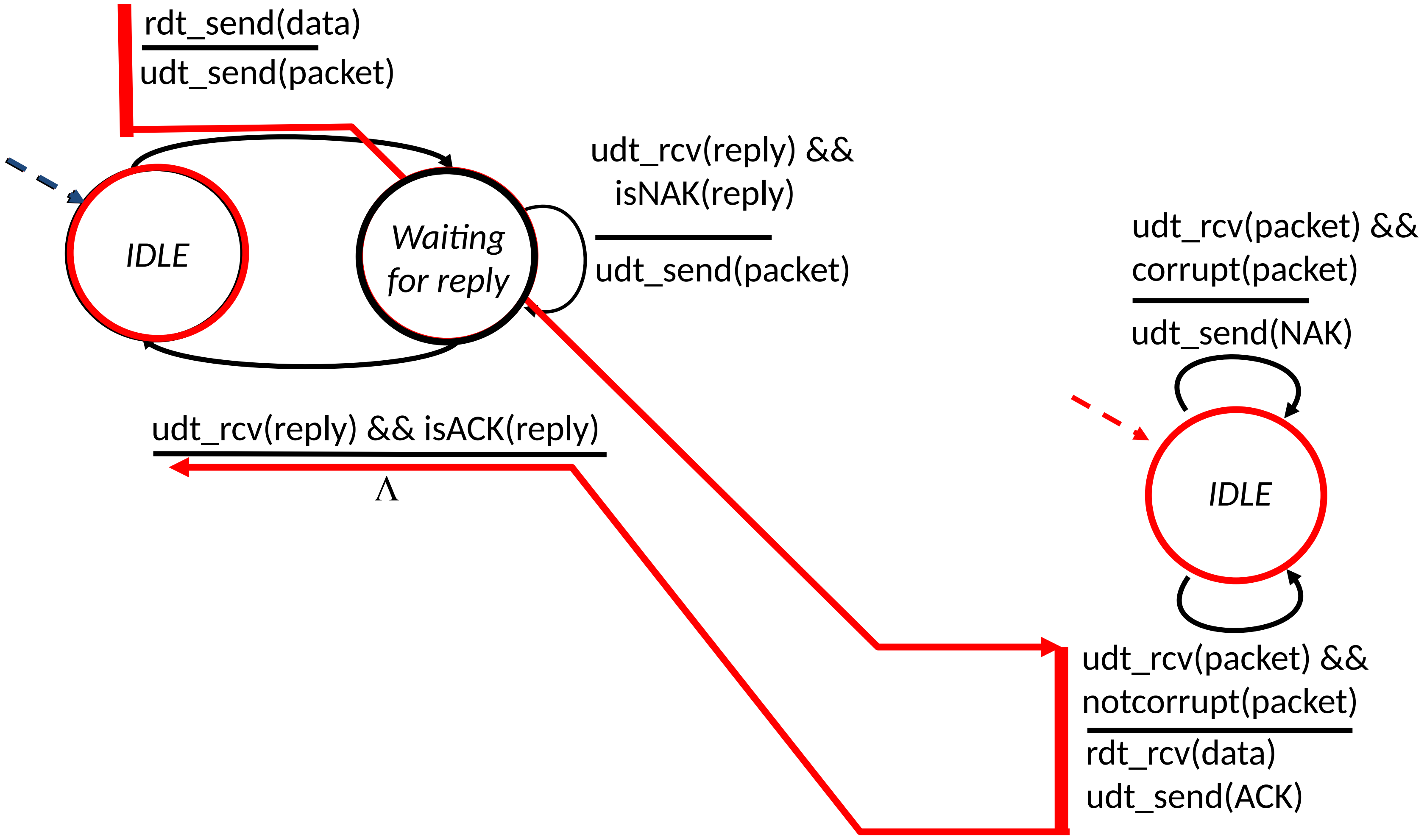
sender


udt_rcv(packet) && corrupt(packet)
udt_send(NAK)
IDLE
udt_rcv(packet) && notcorrupt(packet)
rdt_rcv(data)
udt_send(ACK)

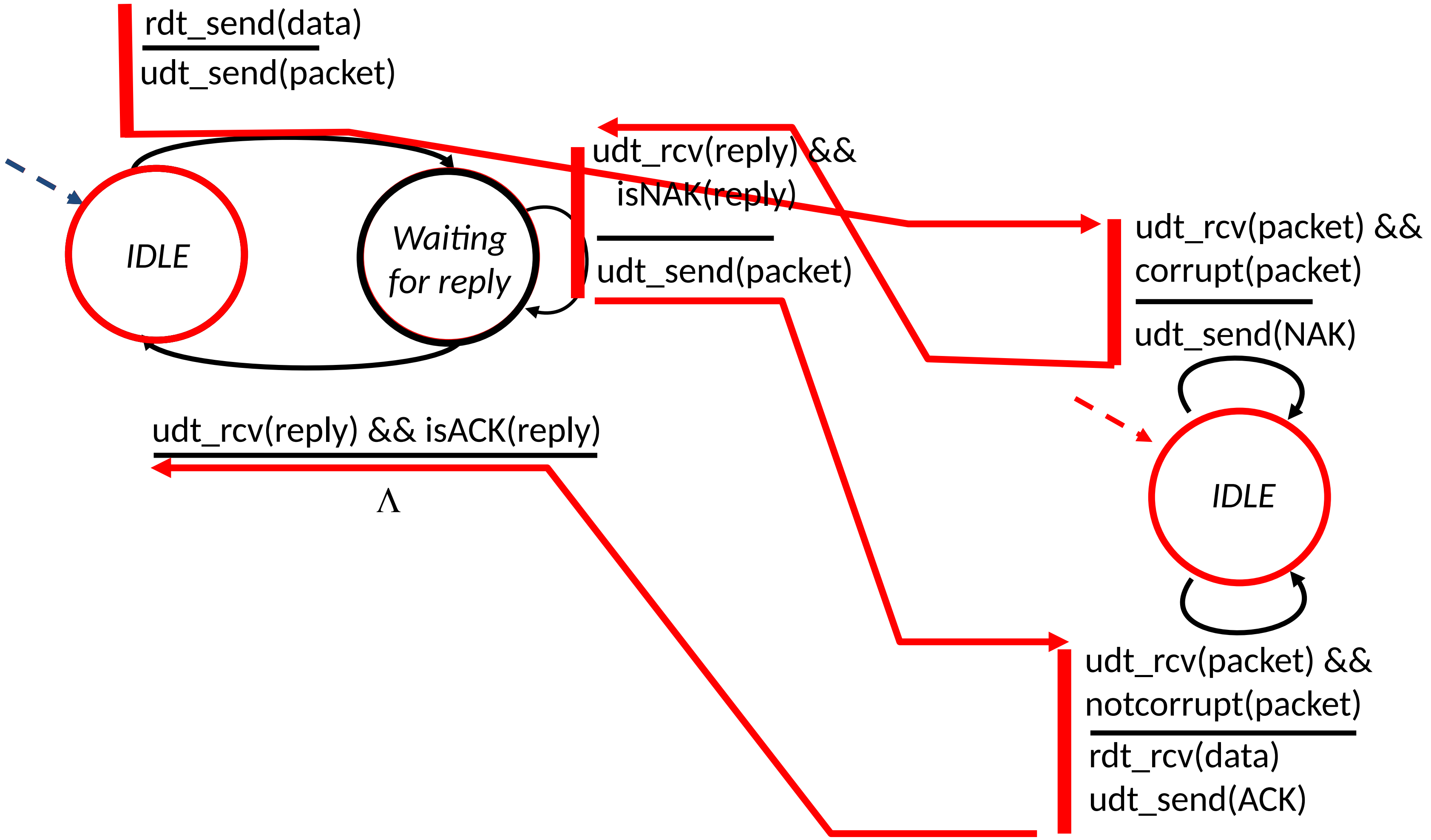
receiver

Note: the sender holds a copy of the packet being sent until the delivery is acknowledged.

rdt2.0: operation with no errors


rdt_send(data)
udt_send(packet)
IDLE
Waiting for reply
udt_rcv(reply) && isNAK(reply)
udt_send(packet)
udt_rcv(reply) && isACK(reply)
Λ
udt_rcv(packet) && corrupt(packet)
udt_send(NAK)
IDLE
udt_rcv(packet) && notcorrupt(packet)
rdt_rcv(data)
udt_send(ACK)

rdt2.0: error scenario


rdt_send(data)
udt_send(packet)
IDLE
Waiting for reply
udt_rcv(reply) && isNAK(reply)
udt_send(packet)
udt_rcv(packet) && corrupt(packet)
udt_send(NAK)
udt_rcv(reply) && isACK(reply)
Λ
IDLE
udt_rcv(packet) && notcorrupt(packet)
rdt_rcv(data)
udt_send(ACK)

rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted? (Not lost entirely for now)

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

Handling duplicates:

- sender adds *sequence number* to each packet
- sender retransmits current packet if ACK/NAK garbled
- receiver discards (doesn't deliver) duplicate packet

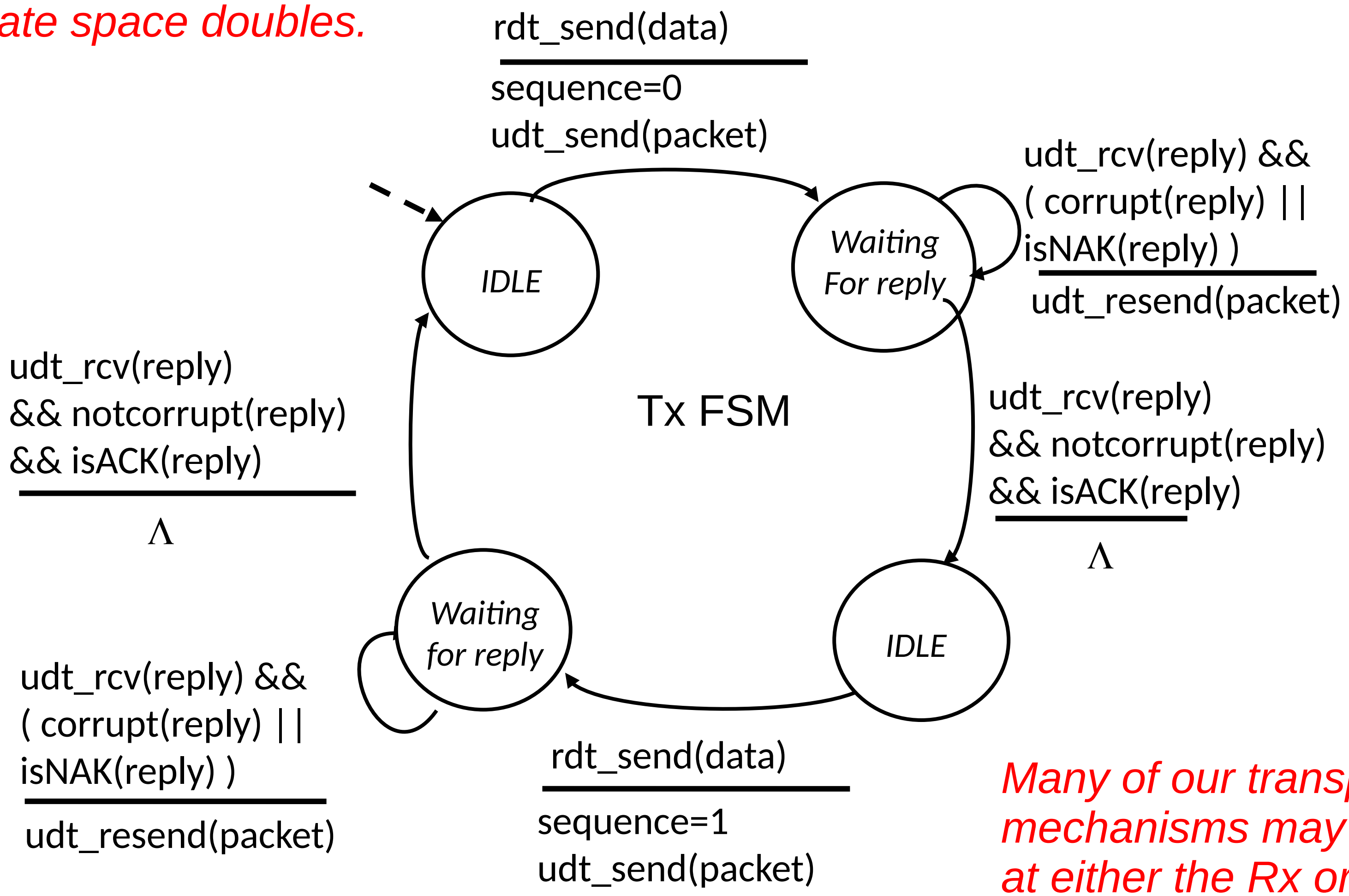
stop and wait

Sender sends one packet, then waits for receiver response

Let's add a 1-bit sequence number.

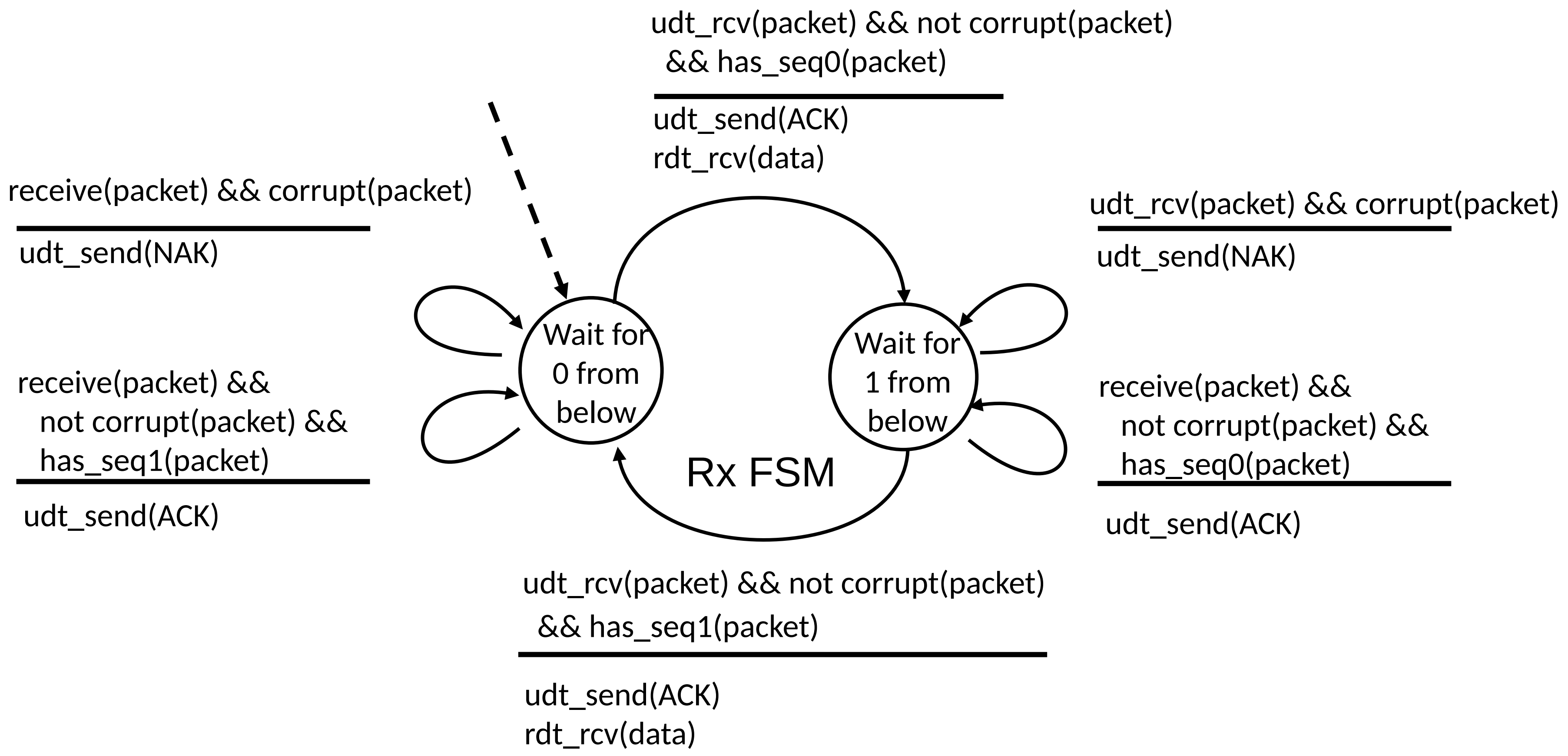
rdt2.1: handles garbled ACK/NAKs

State space doubles.



Many of our transport mechanisms may be provided at either the Rx or Tx. It's a matter of design.

rdt2.1: handles garbled ACK/NAKs


udt_rcv(packet) && not corrupt(packet) && has_seq0(packet)
udt_send(ACK)
rdt_rcv(data)
receive(packet) && corrupt(packet)
udt_send(NAK)
udt_rcv(packet) && corrupt(packet)
udt_send(NAK)
Wait for 0 from below
Wait for 1 from below
receive(packet) && not corrupt(packet) && has_seq1(packet)
udt_send(ACK)
receive(packet) && not corrupt(packet) && has_seq0(packet)
udt_send(ACK)
Rx FSM
udt_rcv(packet) && not corrupt(packet) && has_seq1(packet)
udt_send(ACK)
rdt_rcv(data)

rdt2.1: discussion

Sender:

- seq numbers added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has a 0 or 1 sequence number

Receiver:

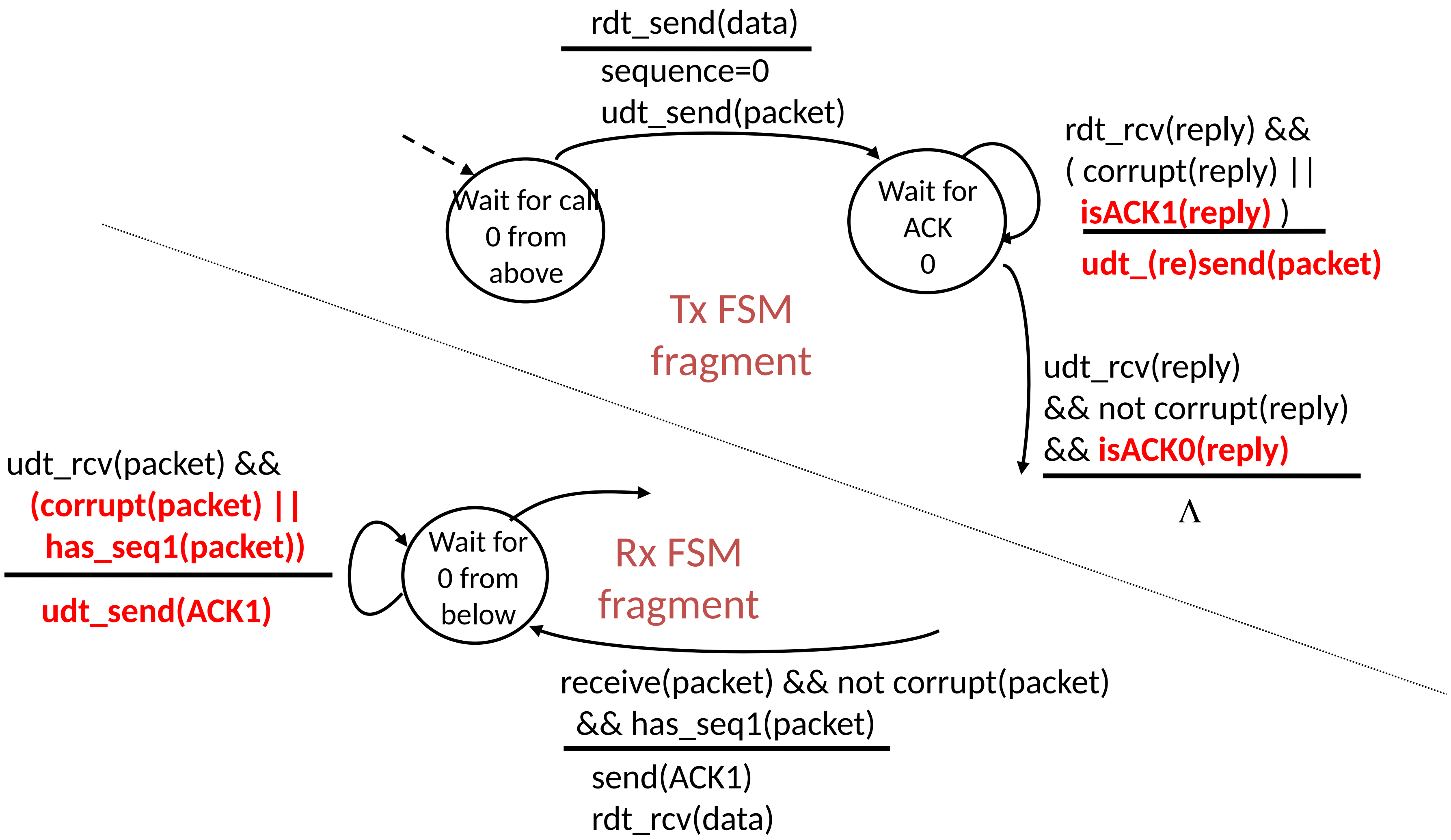
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: Rx *cannot* know if its last ACK/NAK received OK at Tx

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACK messages only
- instead of NAK, Rx sends ACK for last pkt received OK
  - ACK message now includes seq number of pkt being ACK'd
- duplicate ACK at sender convey the same action as NAK did
  - response is to *re-transmit current pkt*

As we will see, TCP uses this approach to be NAK-free.

rdt2.2: sender, receiver fragments


rdt_send(data)
sequence=0
udt_send(packet)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(reply) && (corrupt(reply) || isACK1(reply))
udt_(re)send(packet)
Tx FSM fragment
udt_rcv(reply) && not corrupt(reply) && isACK0(reply)
Λ
udt_rcv(packet) && (corrupt(packet) || has_seq1(packet))
udt_send(ACK1)
Wait for 0 from below
Rx FSM fragment
receive(packet) && not corrupt(packet) && has_seq1(packet)
send(ACK1)
rdt_rcv(data)

rdt3.0: channels with errors *and* loss

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

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

Q: How do *humans* handle lost sender-to-receiver words in conversation?

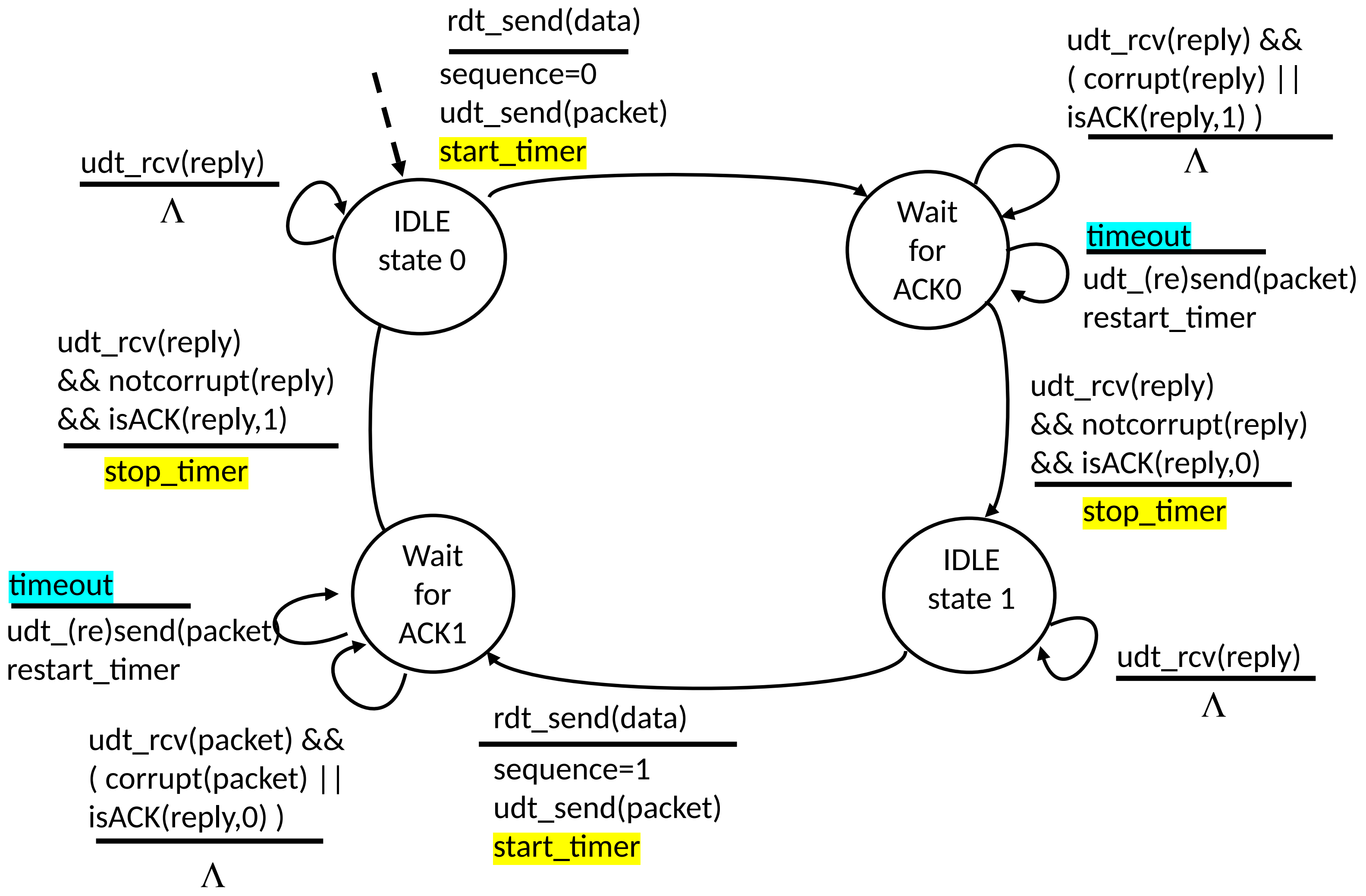
rdt3.0: channels with errors *and* loss

Approach: *Tx* waits “reasonable” amount of time for ACK

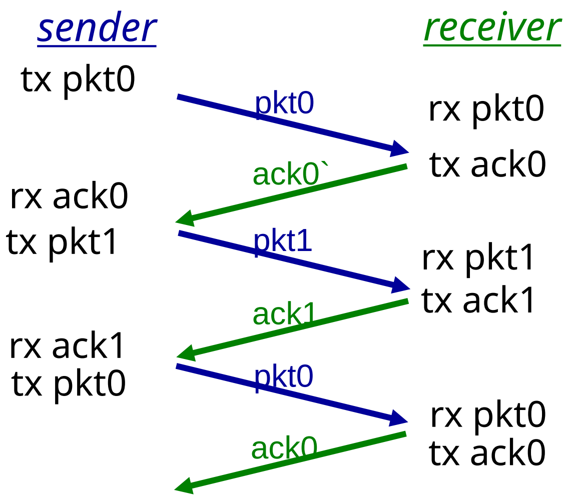
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - Retransmission can be rx’d as a duplicate, but seq numbers already cope with this.
  - Rx must specify seq number of packet being ACK’d
- use countdown timer to interrupt after “reasonable” amount of time

timeout

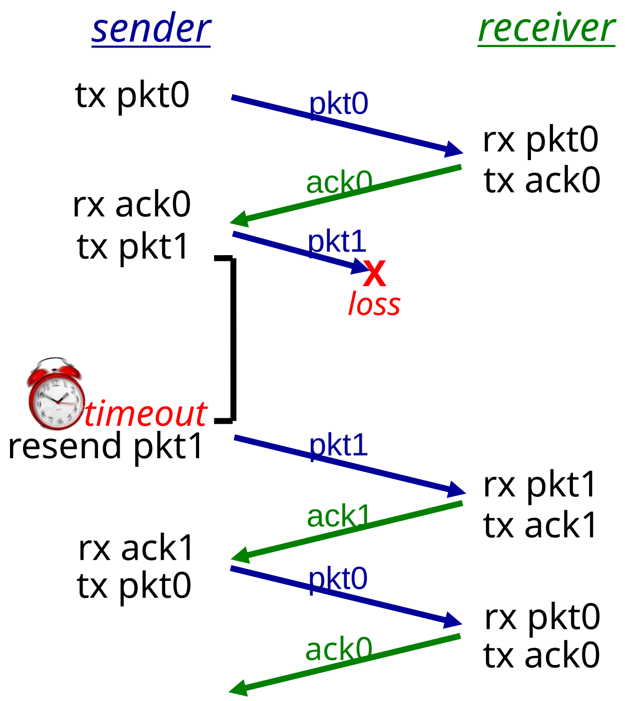
rdt3.0 sender



rdt3.0 in action

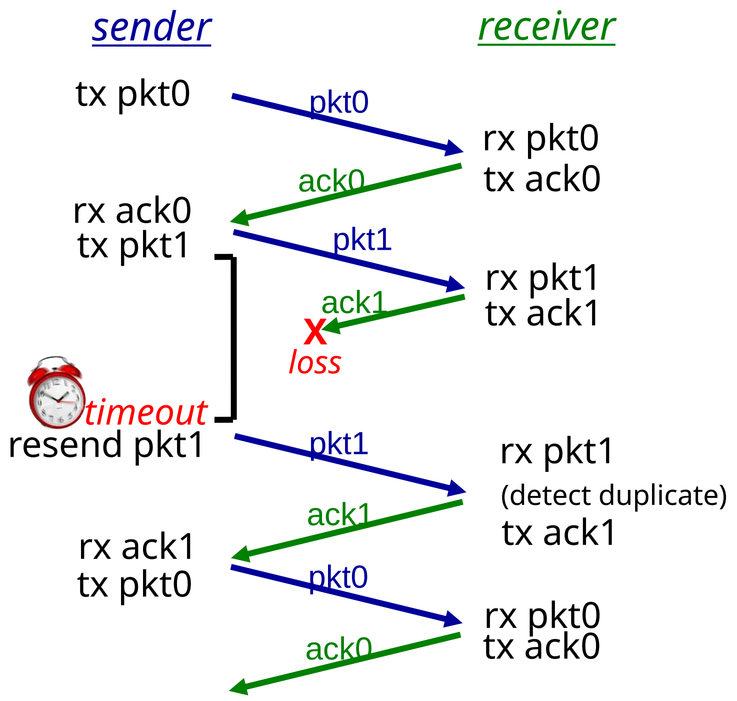


no loss scenario

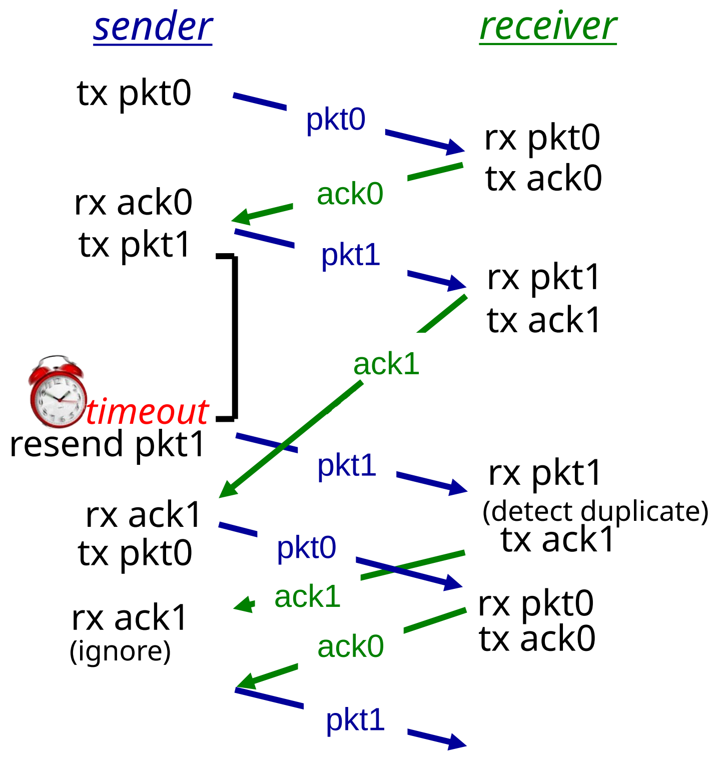


Scenario with packet loss

rdt3.0 in action

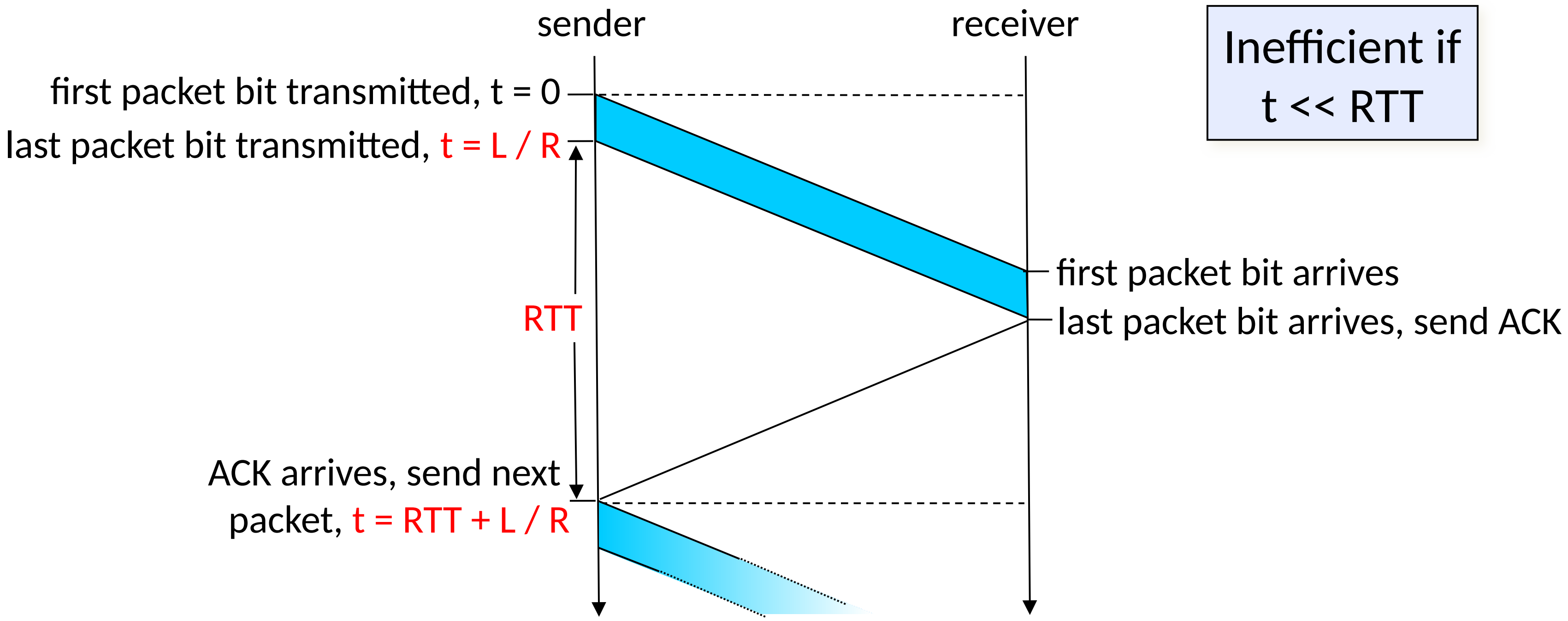


ACK loss scenario



premature timeout/delayed ACK scenario

rdt3.0: 'stop-and-wait' operation


sender
receiver
first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
ACK arrives, send next
packet, t = RTT + L / R

Inefficient if t << RTT

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

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\,\text{microseconds}$$

- U $_{\text{sender}}$: utilization – fraction of time sender busy sending

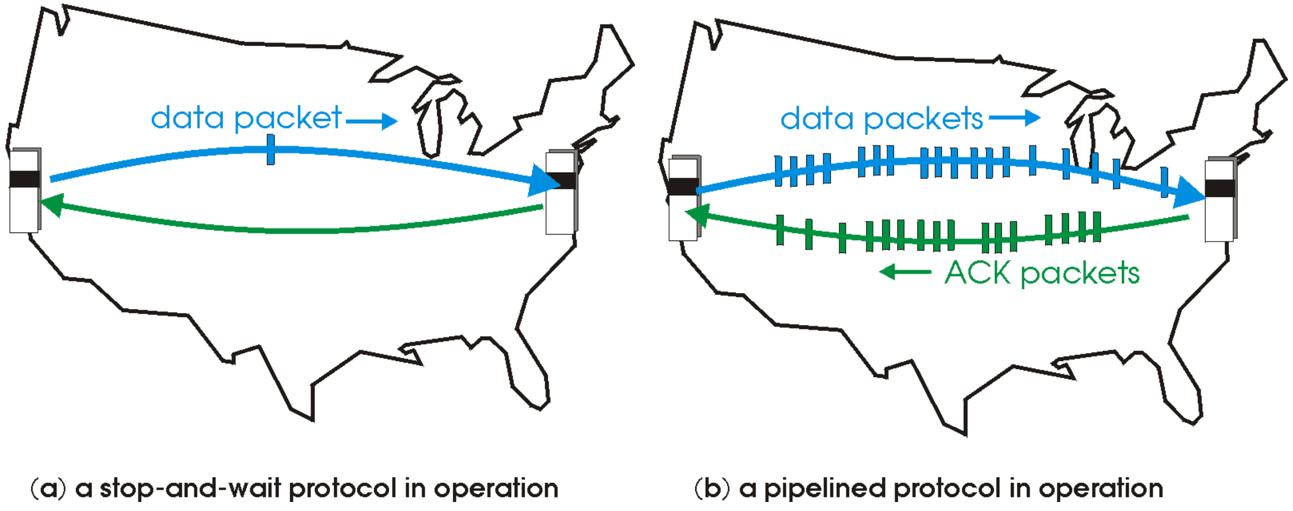
- 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link

- The network protocol limits use of physical resources!

Pipelined (Packet-Window) protocols

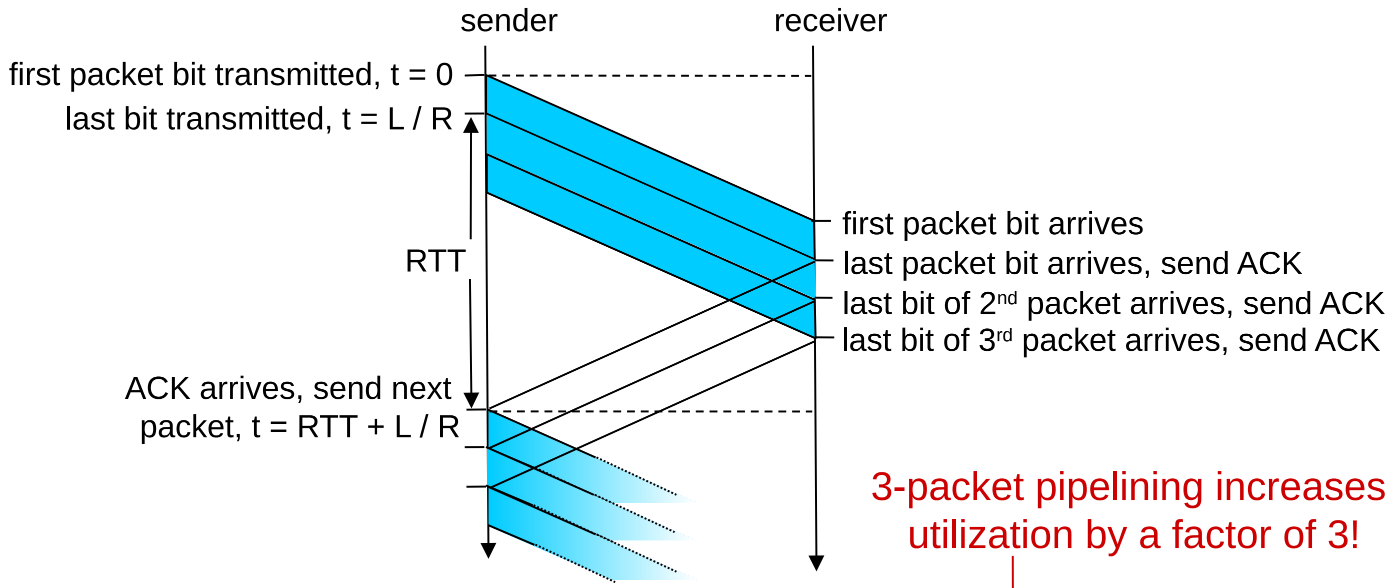
Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- 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

Pipelining: increased utilization


sender
receiver
first packet bit transmitted, t = 0
last bit transmitted, t = L / R
RTT
ACK arrives, send next packet, t = RTT + L / R
first packet bit arrives
last packet bit arrives, send ACK
last bit of 2nd packet arrives, send ACK
last bit of 3rd packet arrives, send ACK

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

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

Why not just use bigger packets?

“Having read the distributed systems course, Part Ib students should be in a position to riff on the core ideas presented so far, coming up with their own special-purpose transport protocols.”

For that course, TCP is merely an interesting ‘case study’ !

A ‘Sliding’ Packet Window

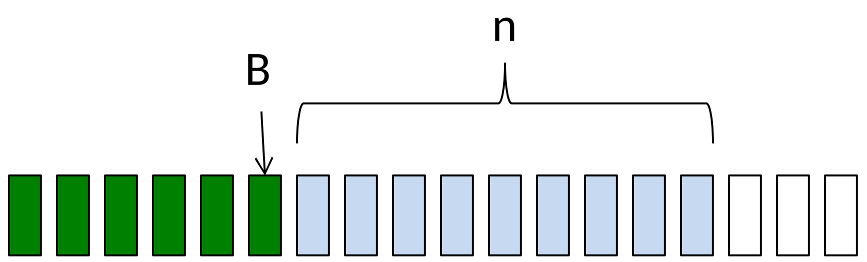
- window = set of adjacent sequence numbers
  - The size of the set is the window size
  - Window size is fixed at *n* *(or varies according to a control algorithm)*.

- General idea: have span of up to *n* packets unACK'd at a time

  - Sender (Tx) can send packets in its window
  - Receiver (Rx) can accept packets in its window
  - Window of acceptable packets “slides” forward on successful reception/acknowledgment.

Acknowledgements (1)

- At Rx

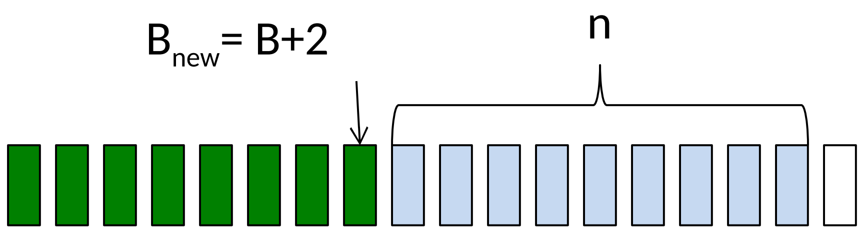


🟩 Received and ACK'd

▨ Acceptable but not yet received

□ Cannot be received

- After receiving B+1, B+2

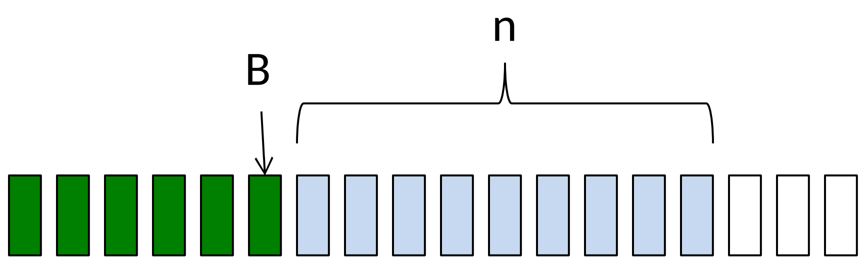


+1 here is because we switch to cumulative ack encoding where the field in packet denotes the next one expected.

- Rx sends ACK(B_{new}+1)

Acknowledgements (2)

- At Rx

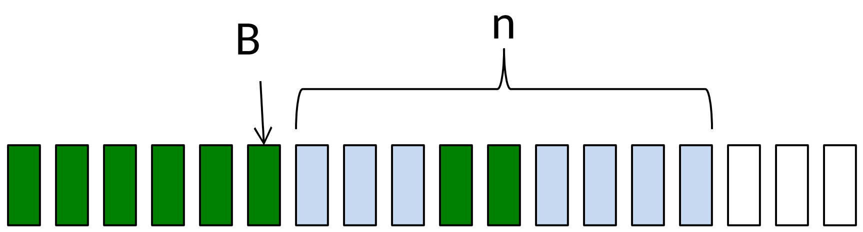


🟩 Received and ACK'd

🟦 Acceptable but not yet received

⬜ Cannot be received

- After receiving B+4, B+5



Oh…. how do we recover?

- Receiver sends ACK(B+???)

Acknowledgements with Sliding Window

How to deal with segment loss:

- Two common options
  - Go-Back-N (GBN)
  - Selective Repeat (SR)

    Also called Selective Acknowledgement (SACK)

As will be shown, sometimes some of the benefit of SR accrues using GBN acknowledgements.

Go-Back-N (GBN)

- Sender (Tx) transmits up to *n* unacknowledged packets
- n is less than the window size (n < N)

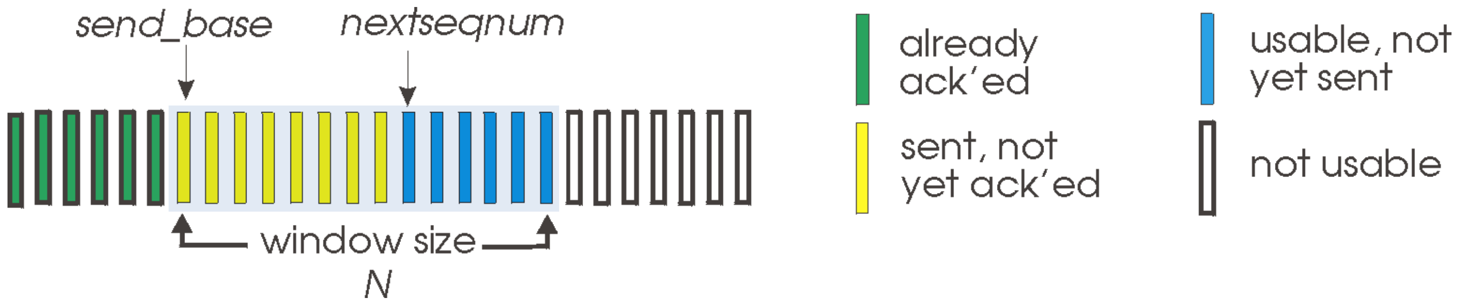
- Rx only accepts packets in order
  - discards out-of-order packets (i.e., packets other than *B+1*)
- Rx uses cumulative acknowledgements
  - ie. sequence number in ACK = next expected in-order sequence no

- Tx sets timer for 1st outstanding ack (A+1)
- If timeout, re-transmit (at some rate) *A+1*, … , *A+n*

The N in Go-Back-N is not the window size N.

Go-Back-N: Tx behaviour

- Tx: ‘window’ of up to N, consecutive transmitted but un-ACK’d pkts
  - k-bit seq number in pkt header

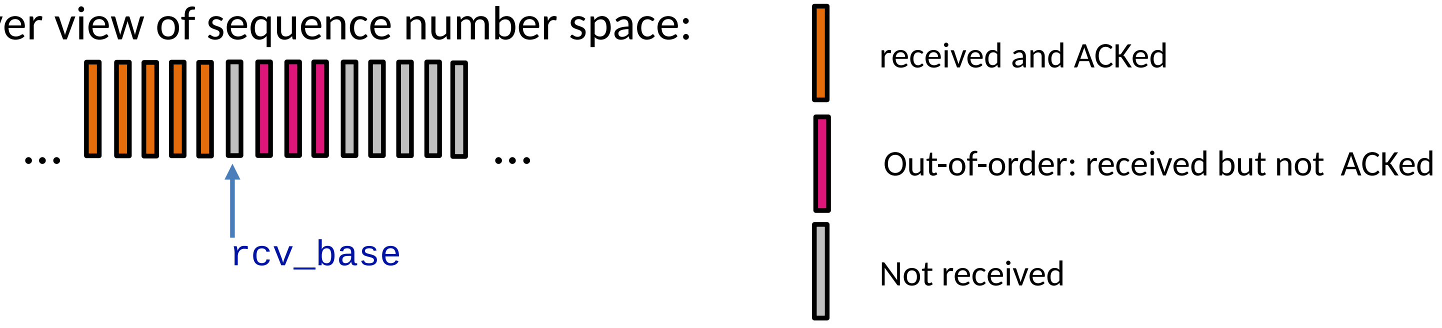


- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq number *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest (lowest n) un-ack’d packet
- *timeout(n):* re-transmit packet n and then (at some rate) re-send all higher packets in window

Go-Back-N: receiver

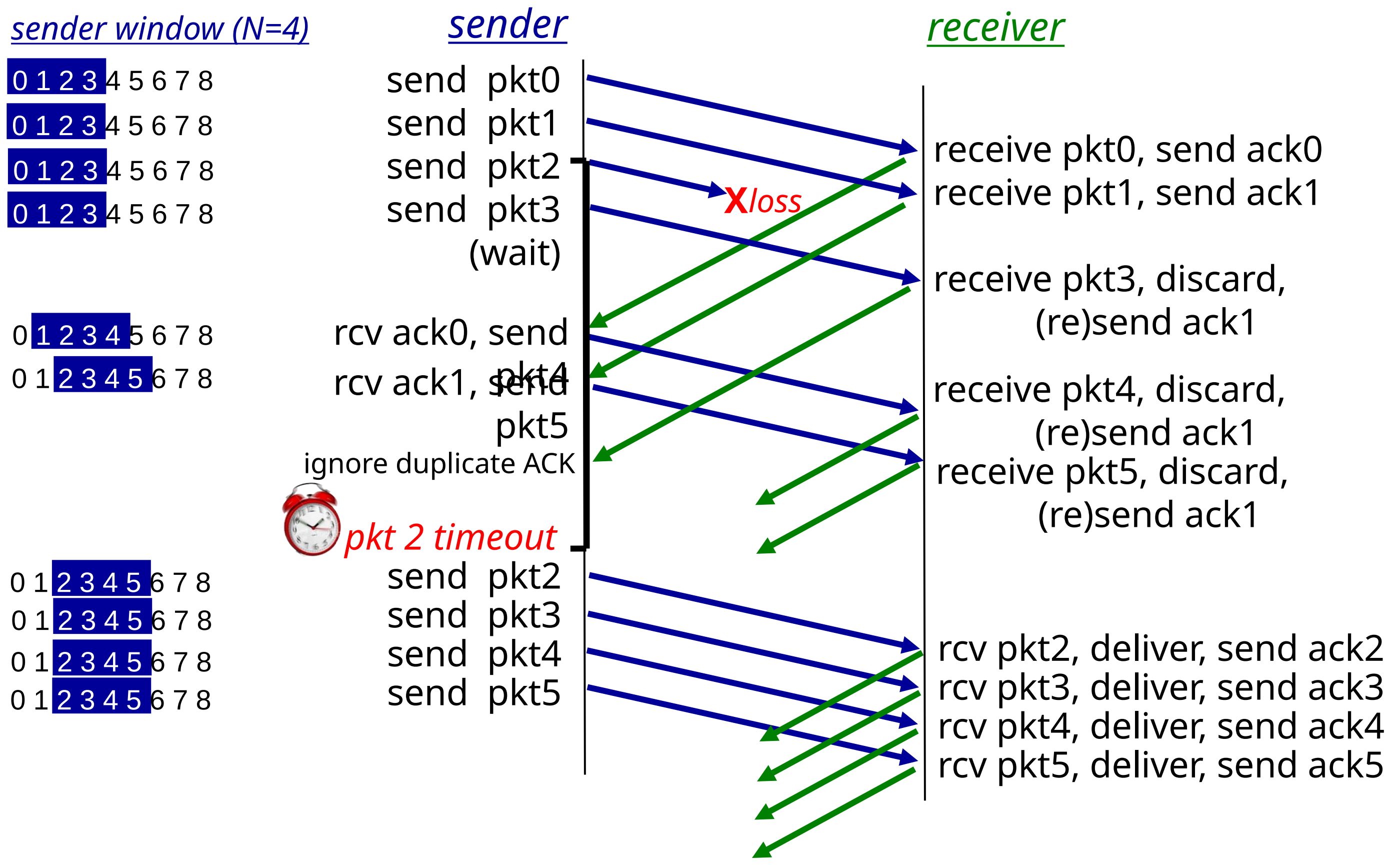
- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq number
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order segment:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest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

Receiver view of sequence number space:

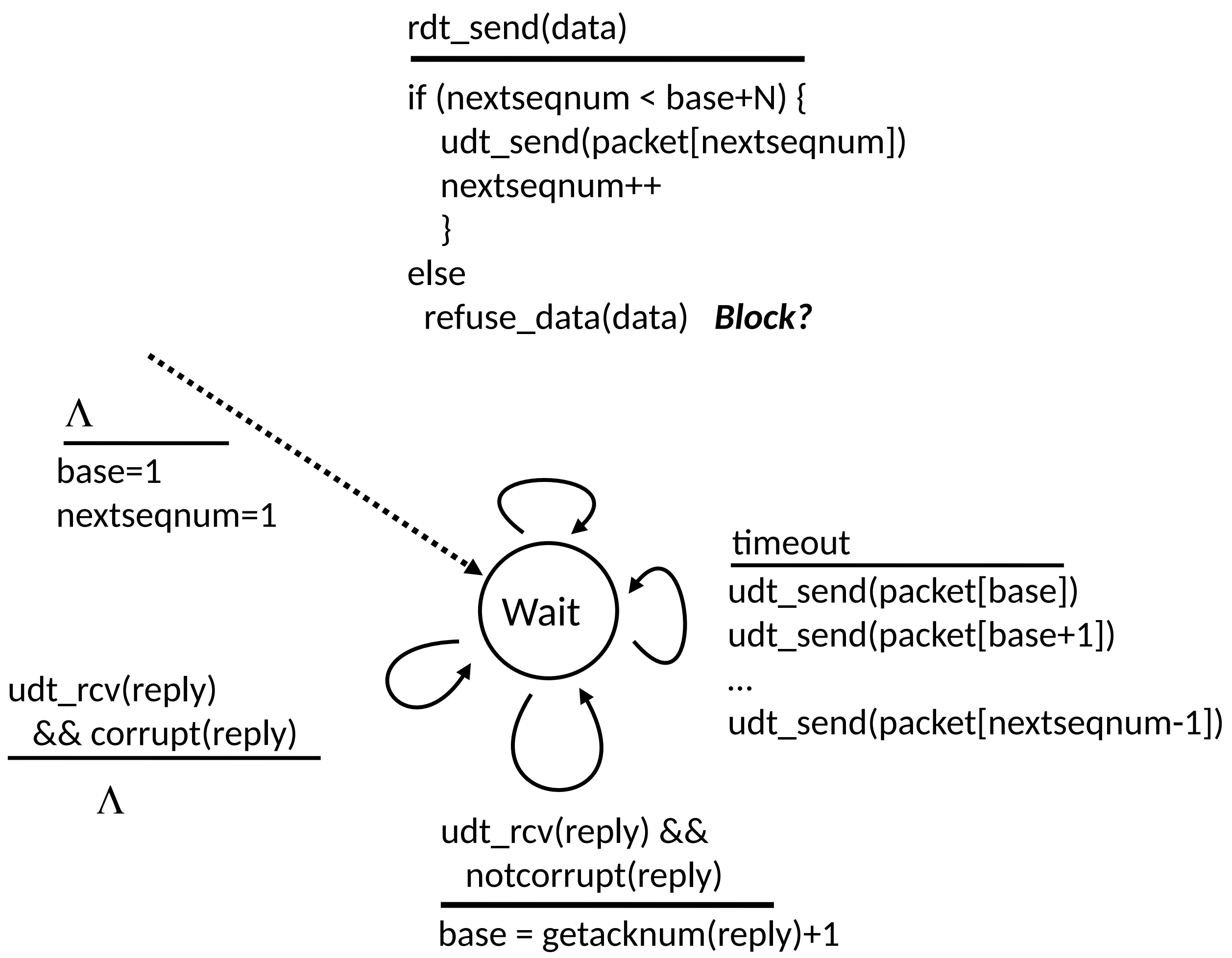


Having a choice about whether or how much to buffer out-of-order segments enables seamless choice over various compatible Rx complexities.

Go-Back-N in action

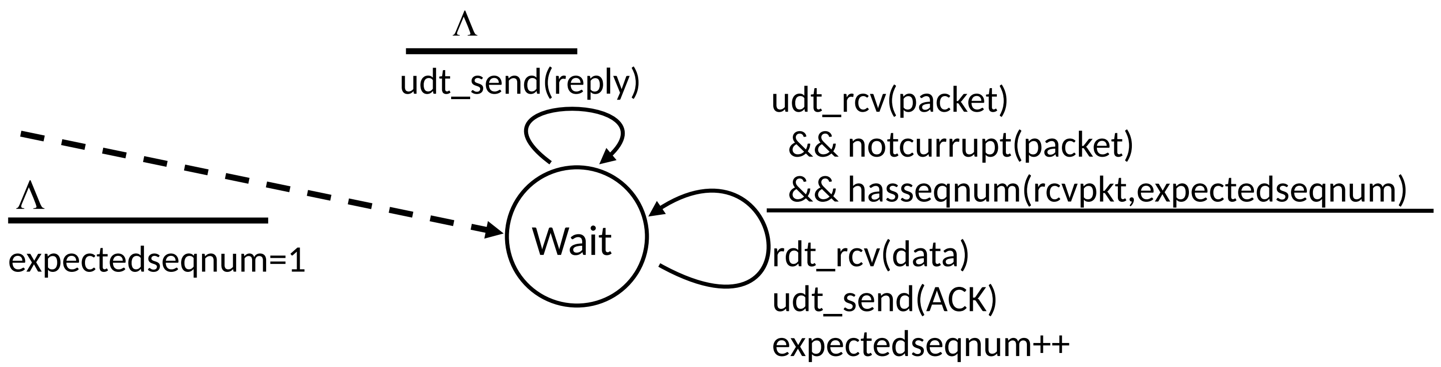


GBN: sender extended FSM


rdt_send(data)
if (nextseqnum < base+N) {
udt_send(packet[nextseqnum])
nextseqnum++
}
else
refuse_data(data) ***Block?***
Λ
base=1
nextseqnum=1
Wait
timeout
udt_send(packet[base])
udt_send(packet[base+1])
…
udt_send(packet[nextseqnum-1])
udt_rcv(reply) && corrupt(reply)
Λ
udt_rcv(reply) && notcorrupt(reply)
base = getacknum(reply)+1

GBN FSM not lectured in 25/26.

GBN: receiver extended FSM



ACK-only: always send an ACK for correctly-received packet with the highest *in-order* seq 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 seq number

GBN FSM not lectured in 25/26.

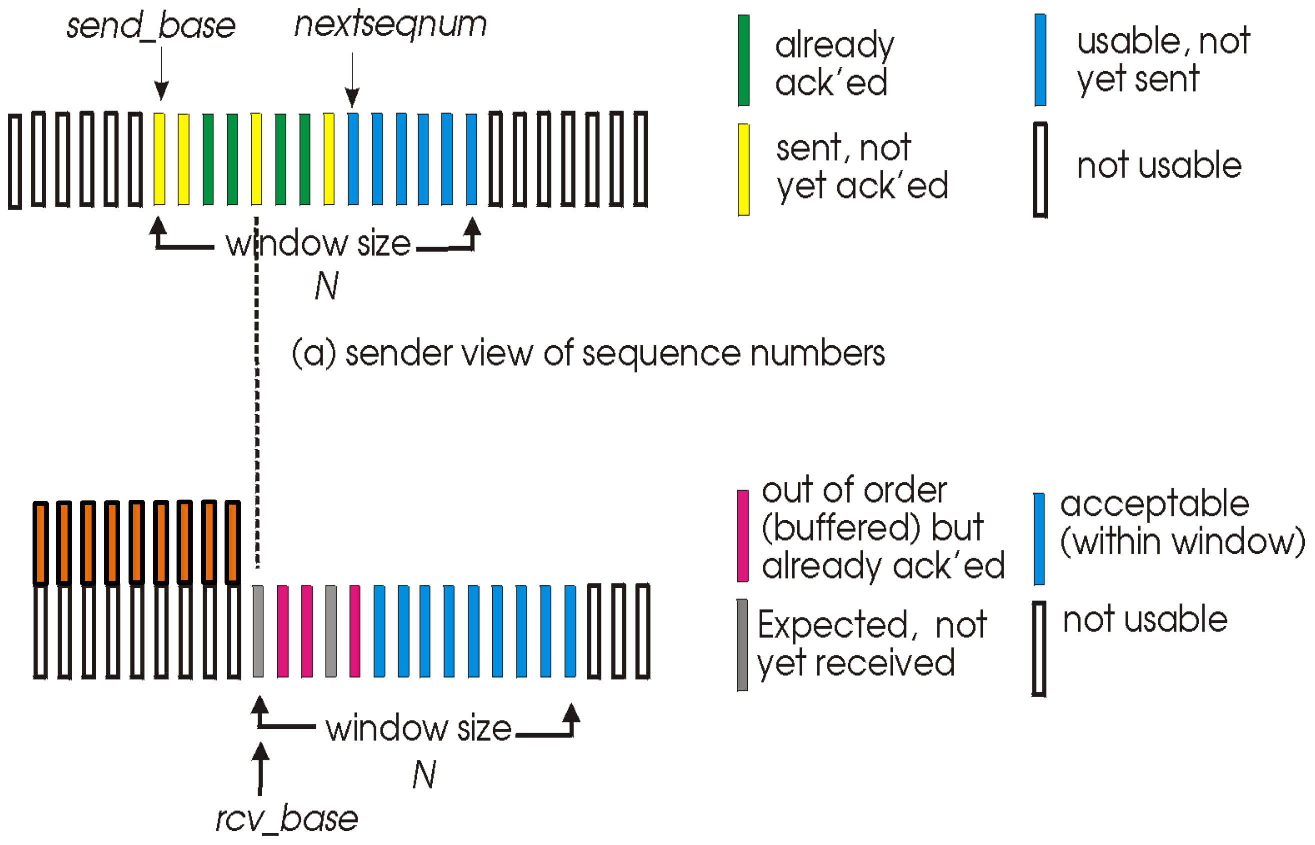
Selective Repeat (the main alternative)

- Rx *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- Tx times-out/retransmits individually for unACK’d packets
  - sender maintains timer for each unACK’d pkt
- Tx window
  - *N* consecutive seq numbers
  - limits seq numbers of sent, unACK’d packets

This is also known as Selective Acknowledgement or simply SACK.

Selective repeat window views

Tx and Rx share common window size, N, but Rx runs behind…



send_base
nextseqnum
already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable
window size N
(a) sender view of sequence numbers
out of order (buffered) but already ack'ed
Expected, not yet received
acceptable (within window)
not usable
window size N
rcv_base
(b) receiver view of sequence numbers

Note to lecturer: is already ack'd colouring accurate in detail?

Selective Repeat: sender and receiver

sender

On data from above:

- if next available seq no in window, send packet, else block application

On timeout(*n*):

- resend packet *n*, restart timer

On ACK(*n*) in [send_base, send_base+N-1]:

- mark packet *n* as received
- if n smallest unACK'd packet, advance window (send_base) to next unACK'd seq number.

receiver

On packet *n* in [rcv_base, rcv_base+N-1] with ok checksum/CRC:

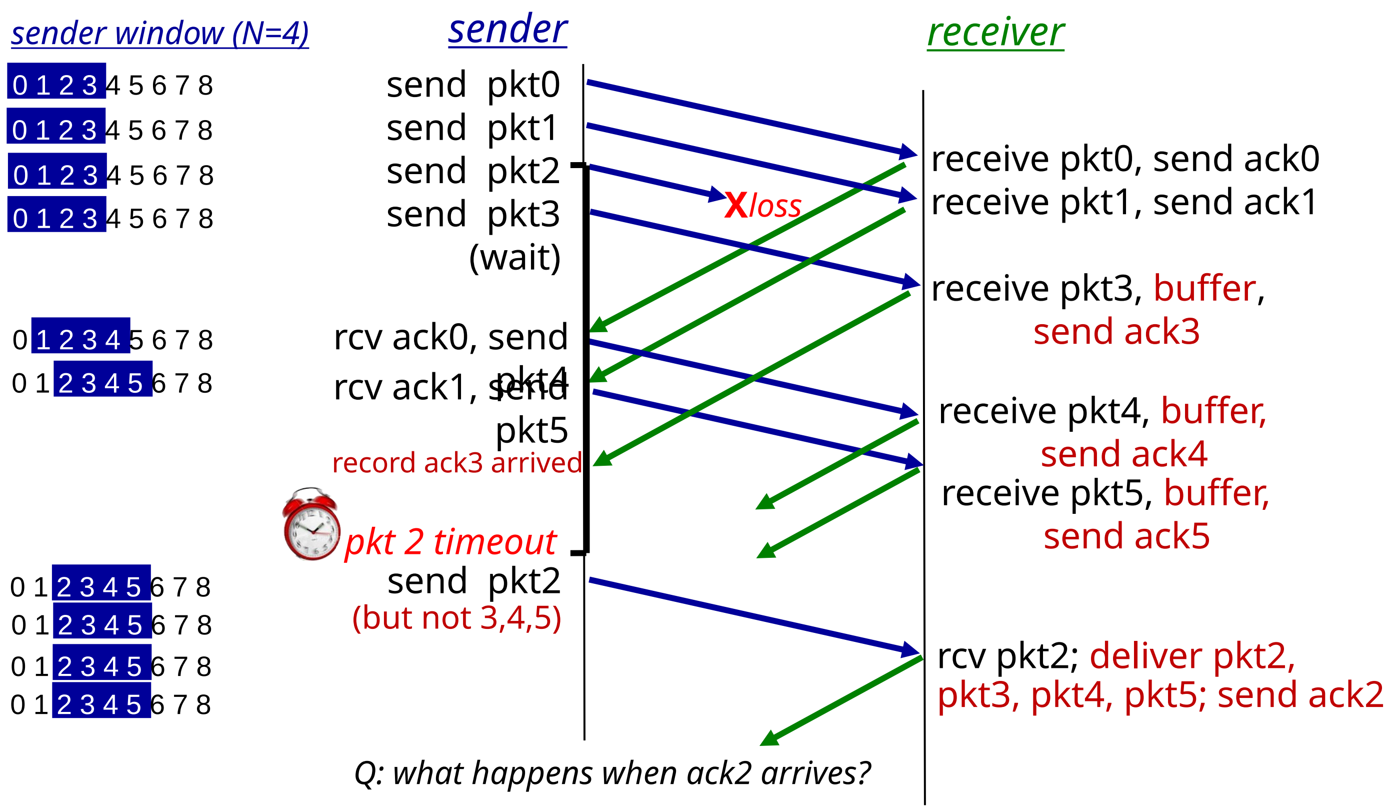
- send ACK(*n*)
- If out-of-order: Buffer the segment
- Else
  - Deliver any earlier buffered segments in their correct order
  - Deliver current segment
  - Advance window (rcv_base) to next not-yet-received packet

Otherwise:

- ignore

Note: this slide still using notation ACK(n) for n'th packet, instead of ACK(n+1).

Selective Repeat in action

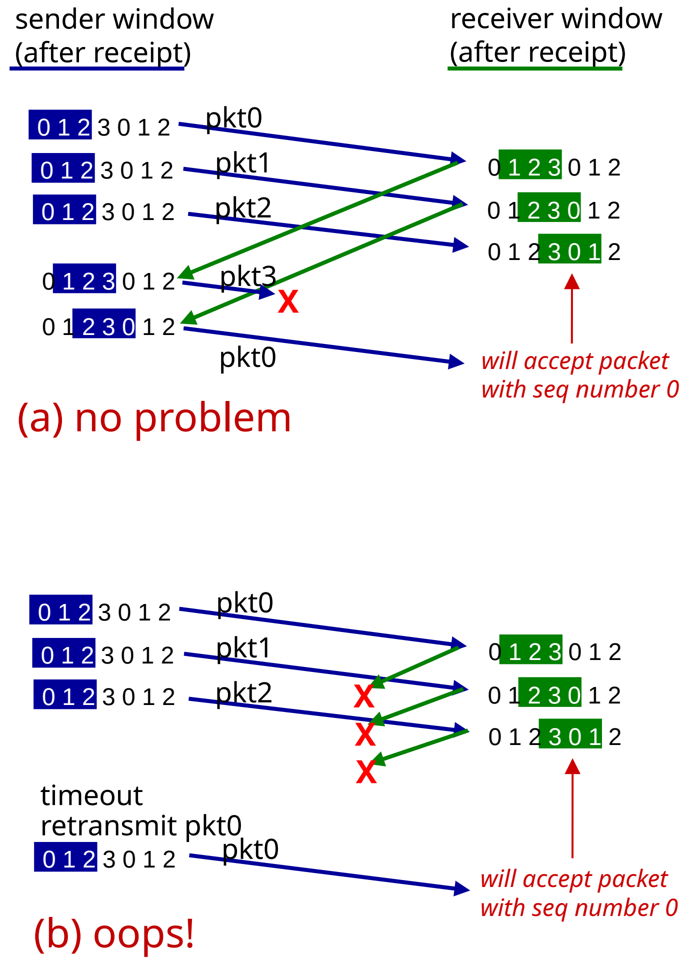


Q: what happens when ack2 arrives?

Sequence number encoding issue (bit width)

SR example:

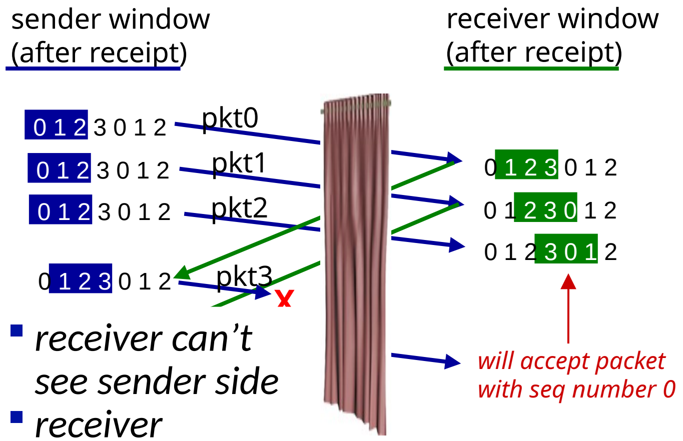
- seq nos: 0, 1, 2, 3 (base 4 counting)
- window size=3


sender window (after receipt)
receiver window (after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
pkt3
X
pkt0
will accept packet with seq number 0
(a) no problem
X
X
X
timeout
retransmit pkt0
pkt0
will accept packet with seq number 0
(b) oops!

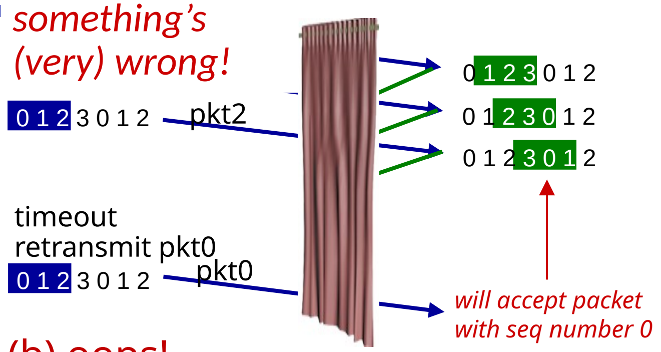
Sequence number encoding issue (bit width)

SR example:

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


sender window (after receipt)
receiver window (after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
pkt3
X
0 1 2 3 0 1 2
will accept packet with seq number 0

- *receiver can't see sender side*
- *receiver behaviour identical in both cases!*
- *something's (very) wrong!*



(b) oops!

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

SR solution:

maximum allowable window size = half the sequence number space.

Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size (n) is large enough. Throughput is ~ (n/RTT)
  - Stop & Wait is like n = 1.
- Sender has to buffer all unacknowledged packets, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to its buffer limits
- Implementation complexity depends on protocol details (GBN vs. SR)

Recap: components of a solution

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)

- Reliability protocols use the above to decide when and what to retransmit or acknowledge

What does TCP do?

Most of our previous tricks + a few more beside

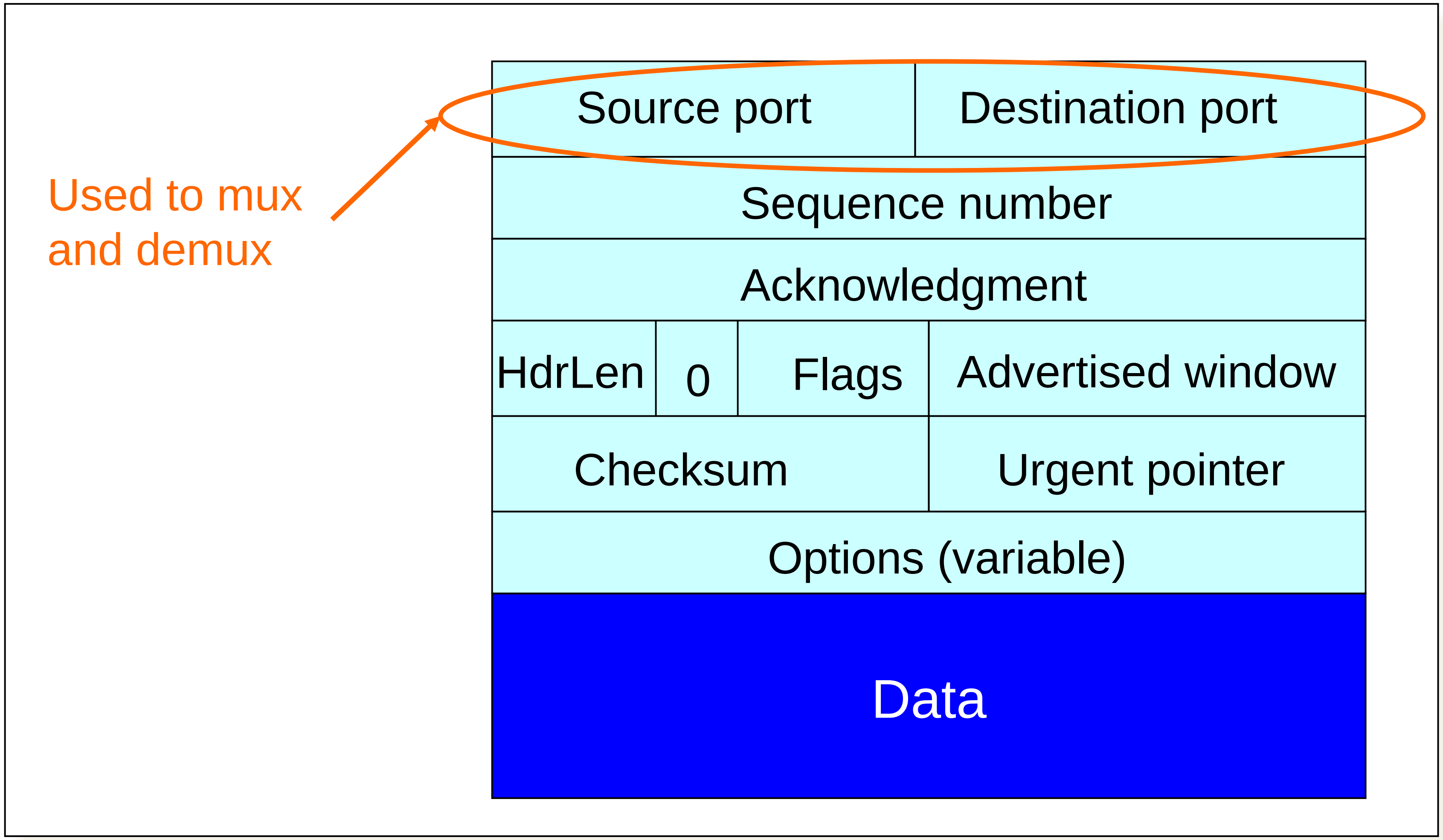
- Sequence numbers are byte offsets
- Sender and receiver maintain a sliding window
- Receiver sends cumulative acknowledgements (like GBN)
- Sender maintains a single retx. timer
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retx
- Introduces timeout estimation algorithms

TCP provides end-to-end flow control with application-level ‘backpressure’ but we’ll discuss this under ‘congestion control’ heading.

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

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP Header

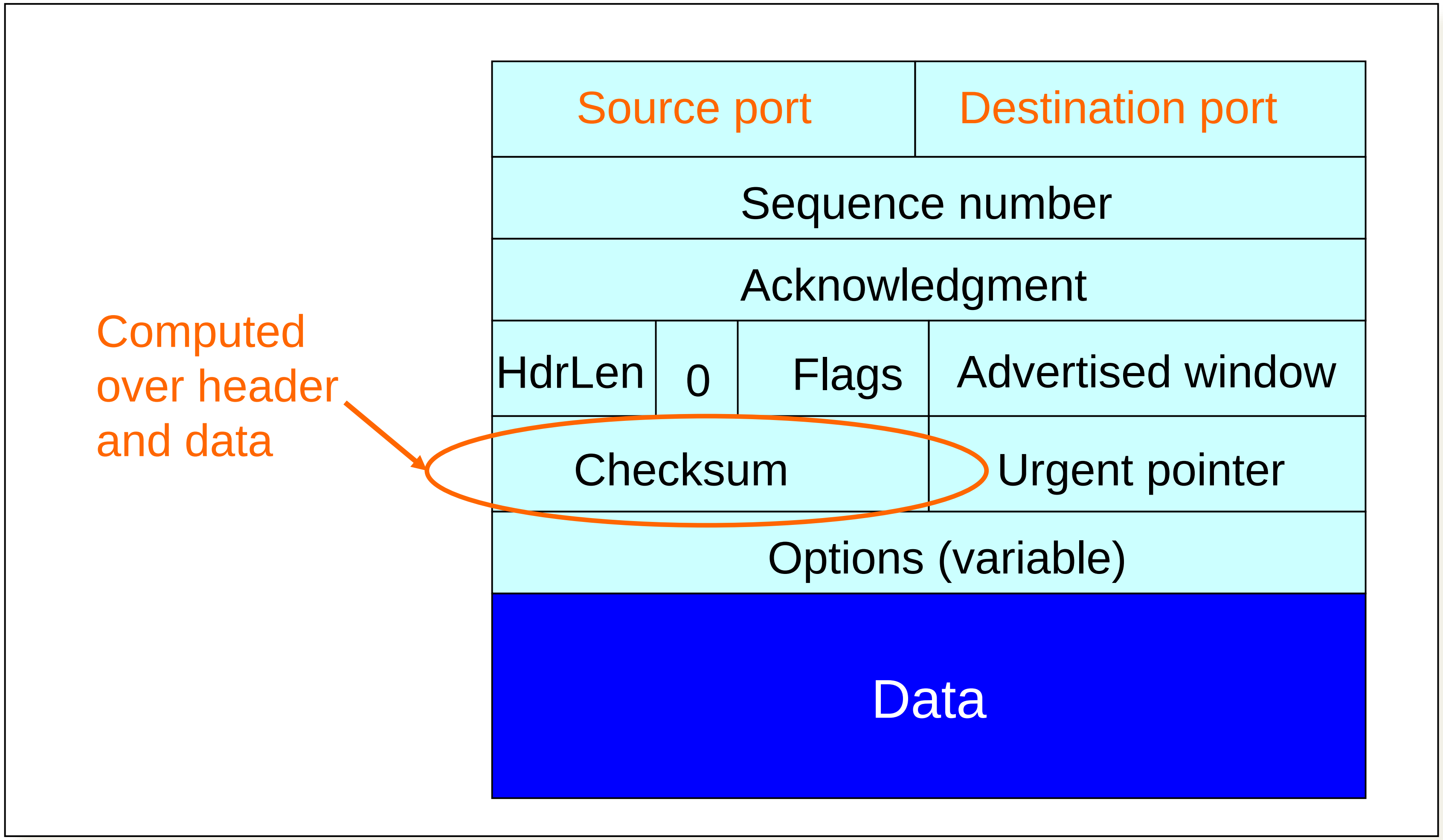

Used to mux and demux
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

What does TCP do?

Many of our previous ideas, but some key differences

- Checksum

TCP Header

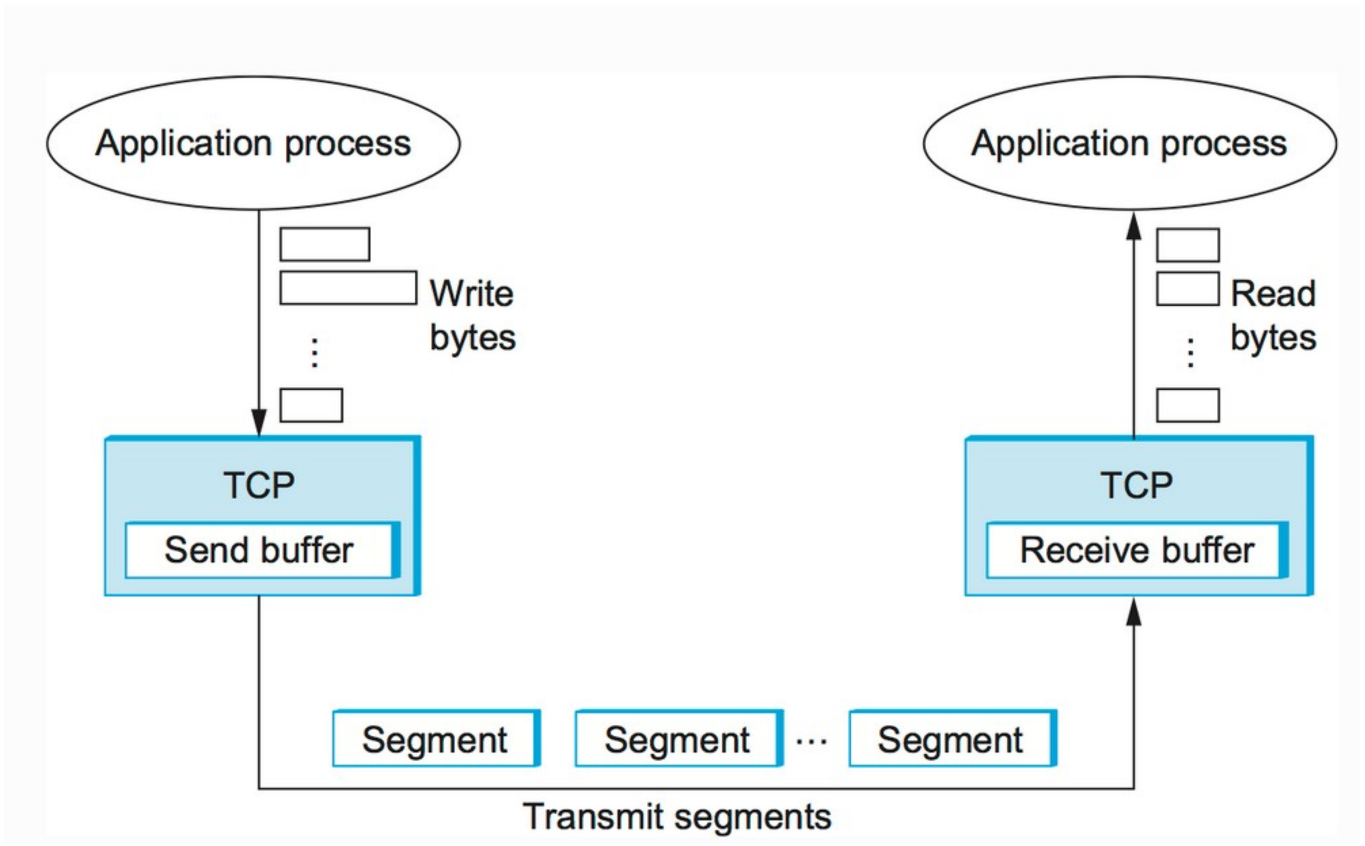

Computed
over header
and data
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

What does TCP do?

Many of our previous ideas, but some key differences

- Checksum
- **Sequence numbers are byte offsets**

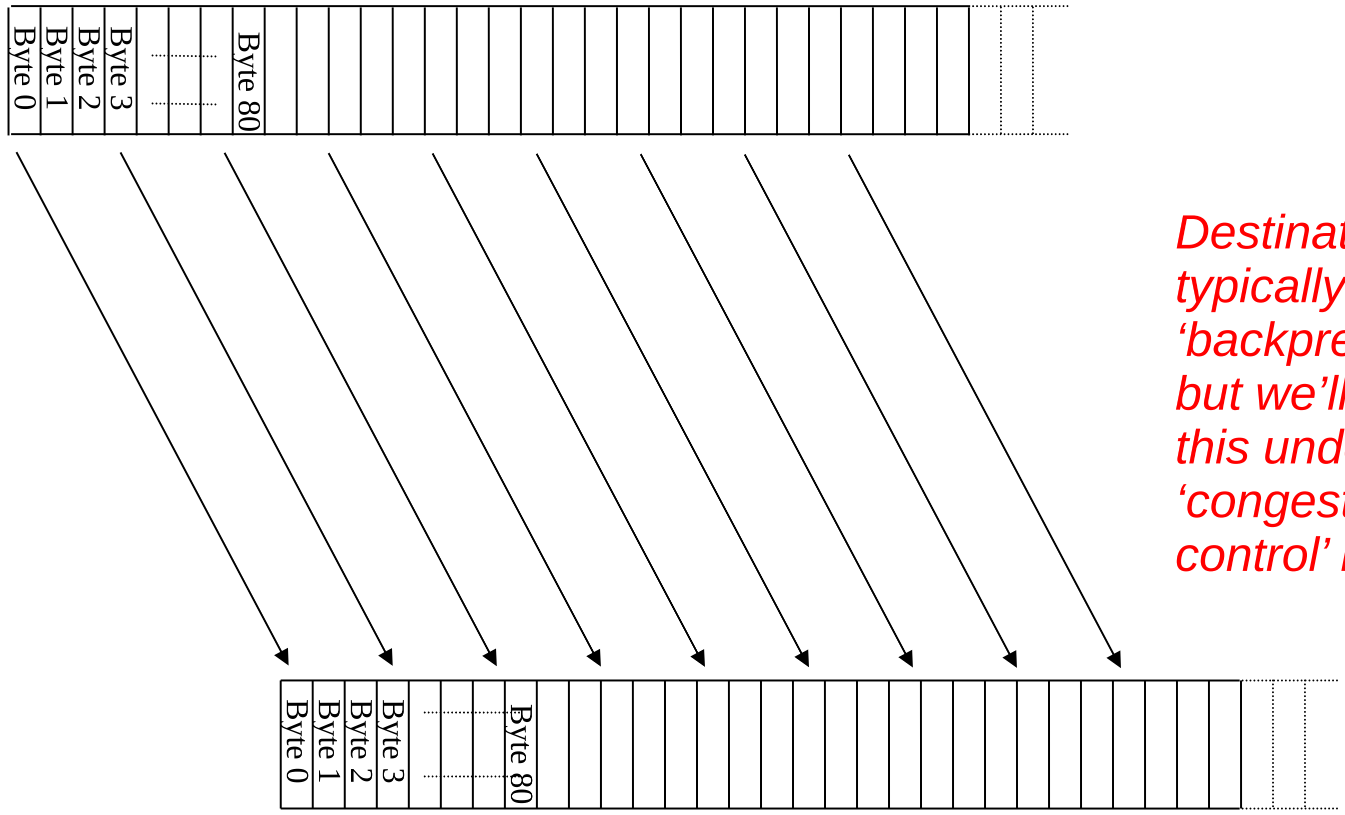
TCP reliable ‘bytestream’ service



From Peterson and Davie (figure 127)

Stream of Bytes – Bytestream

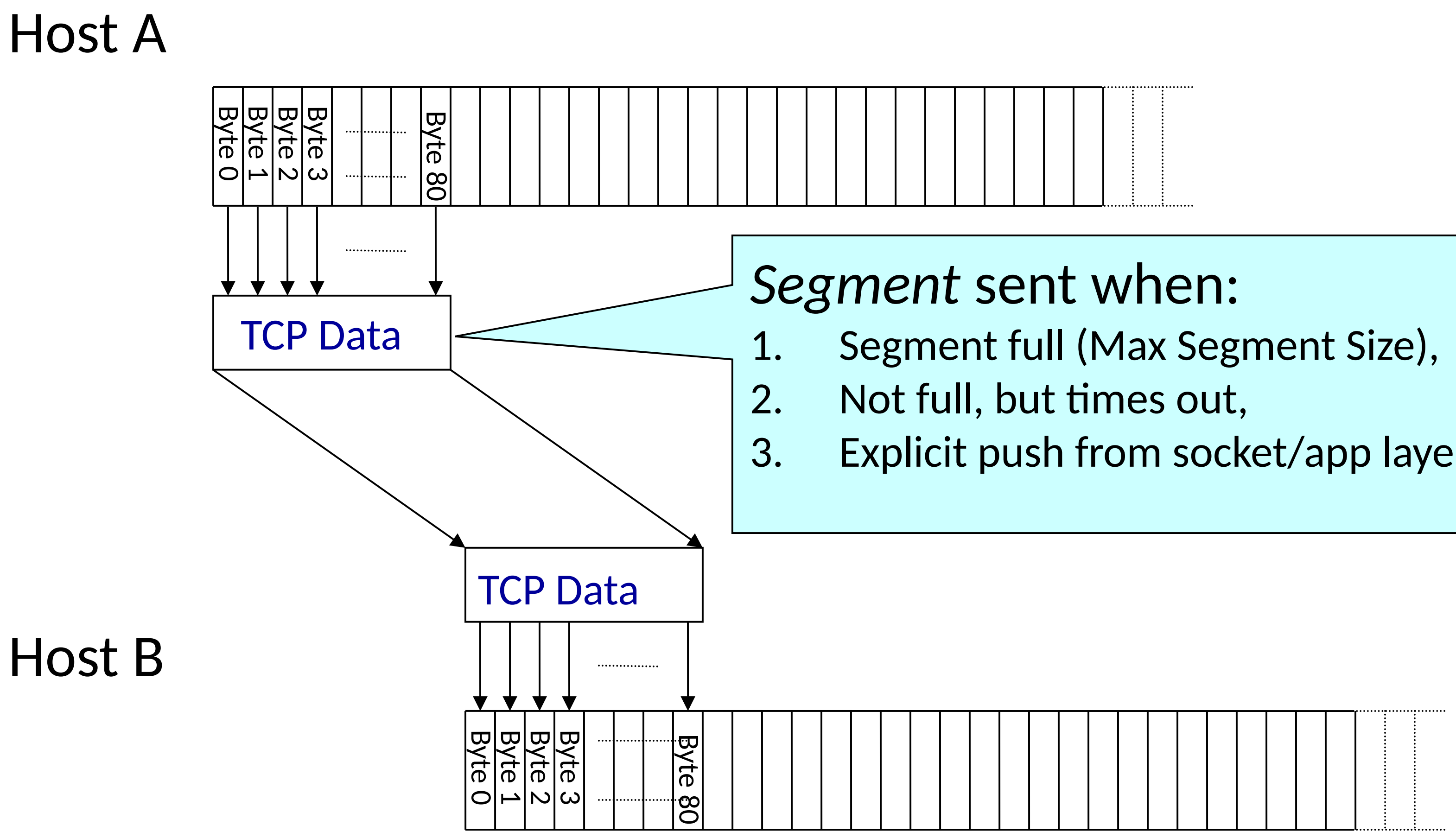
Application @ Host A



Destination typically offers ‘backpressure’ but we’ll discuss this under ‘congestion control’ heading.

Application @ Host B

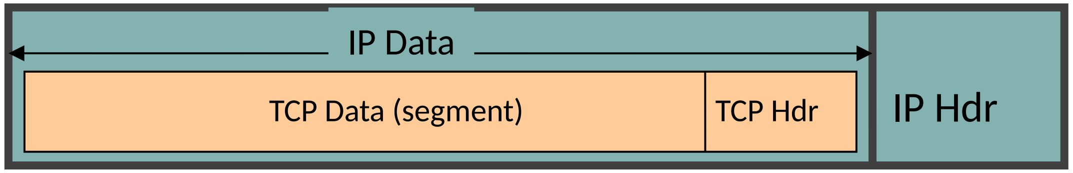
… Provided Using TCP 'Segments'



Segment sent when:

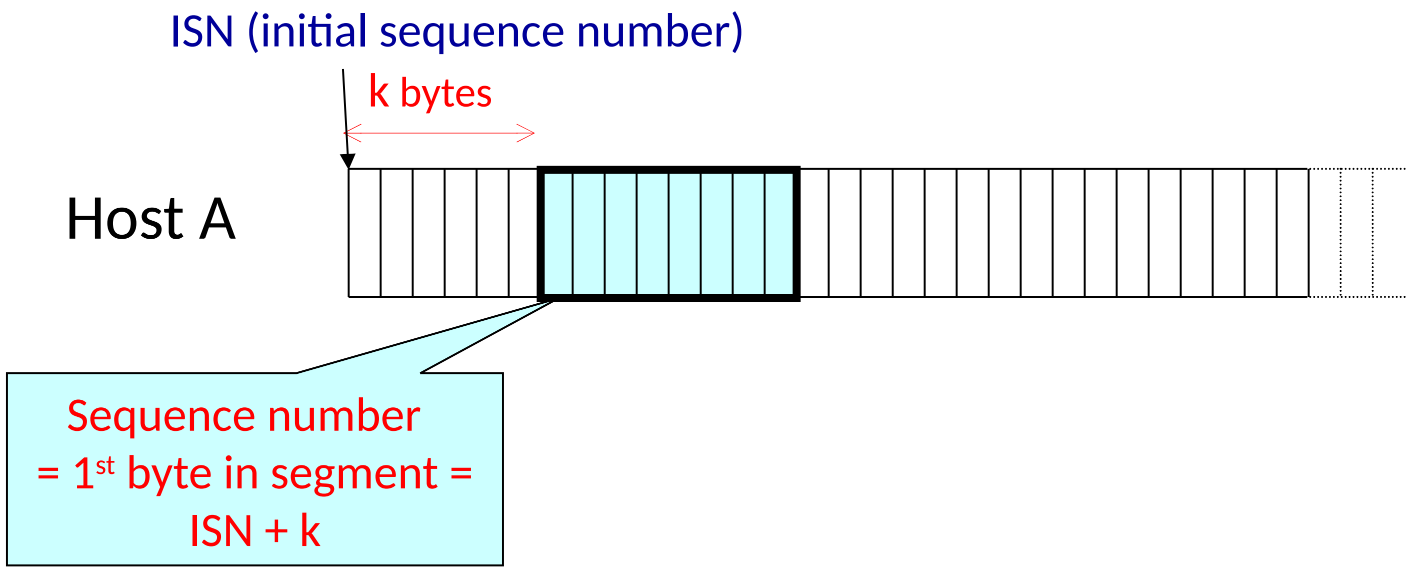
1. Segment full (Max Segment Size),
2. Not full, but times out,
3. Explicit push from socket/app layer.

TCP Segment

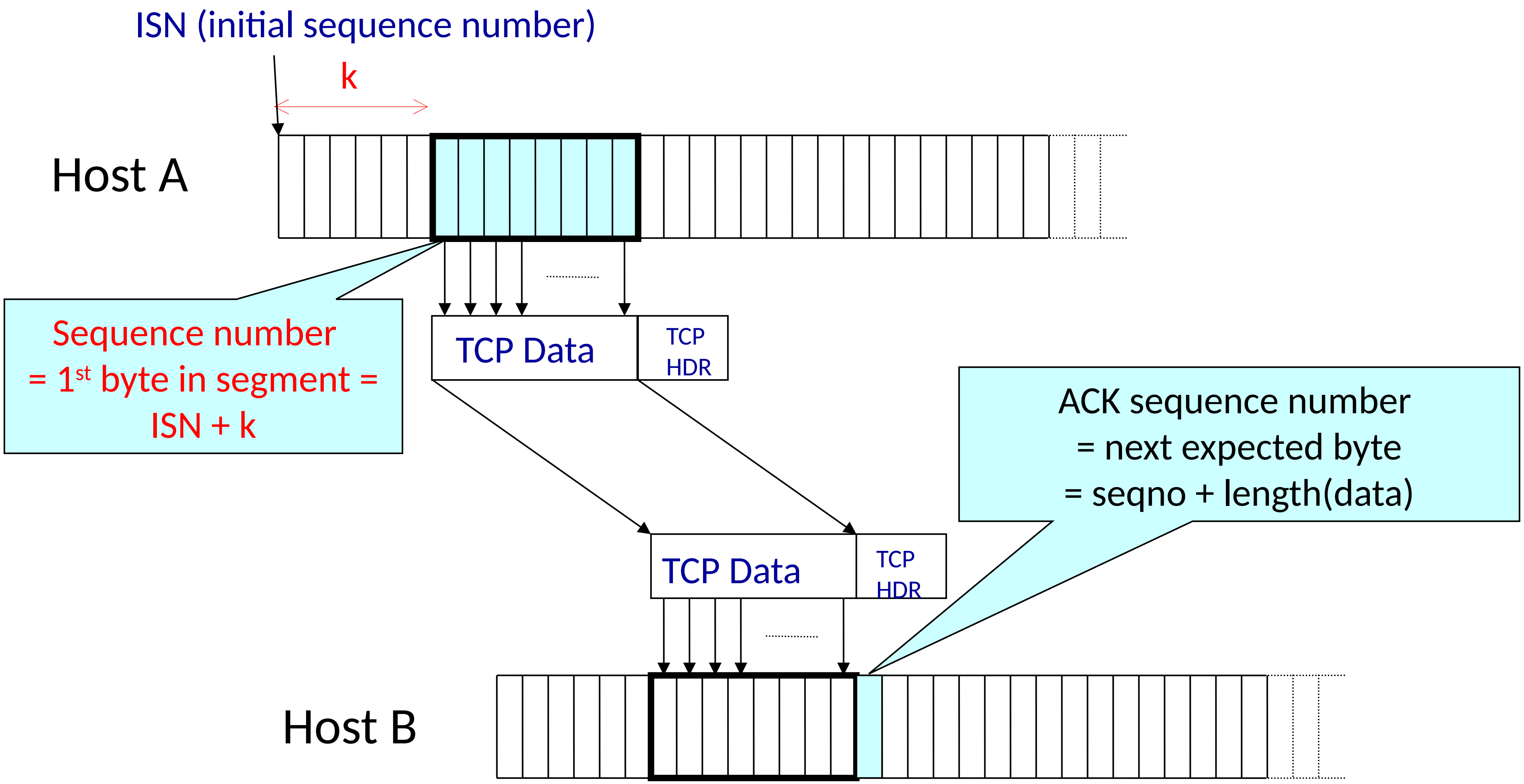


- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header $\geq$ 20 bytes long
- TCP **segment**
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU – (IP header) – (TCP header)

Sequence Numbers

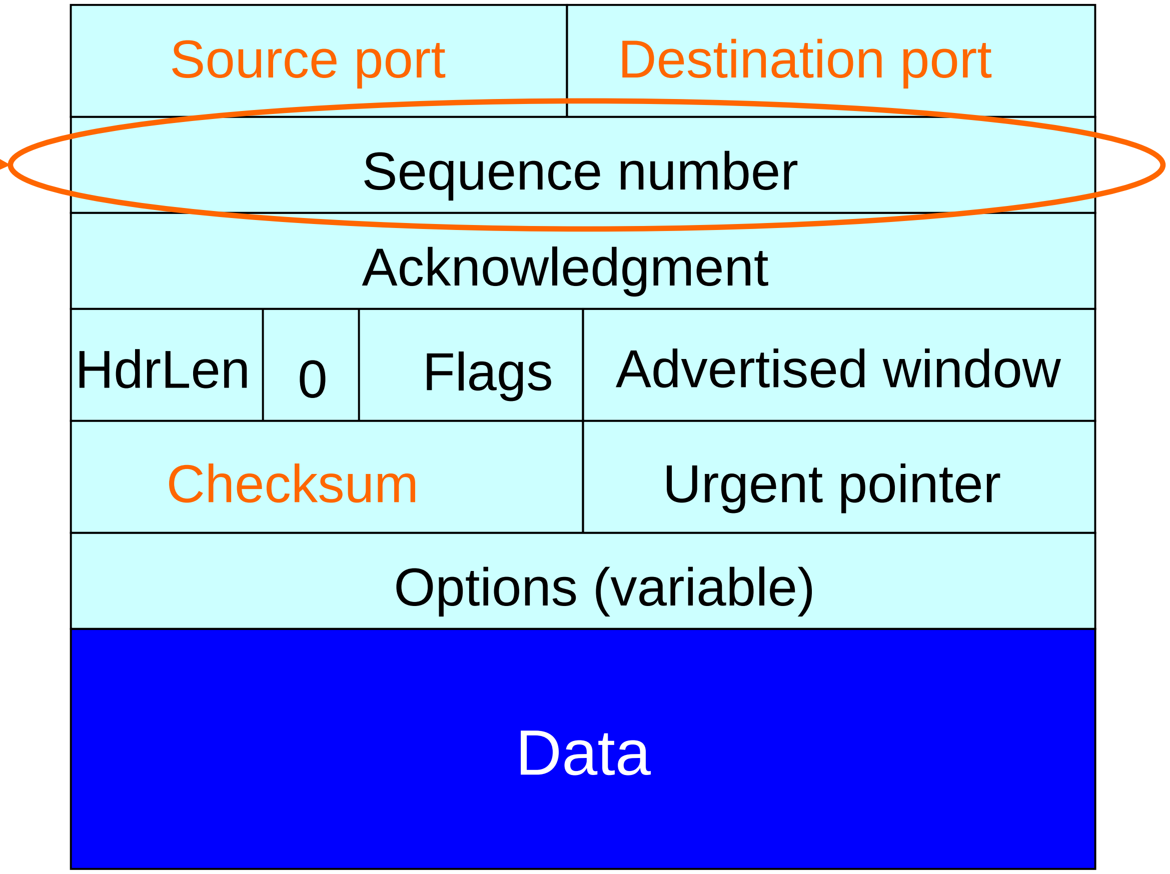


Sequence Numbers

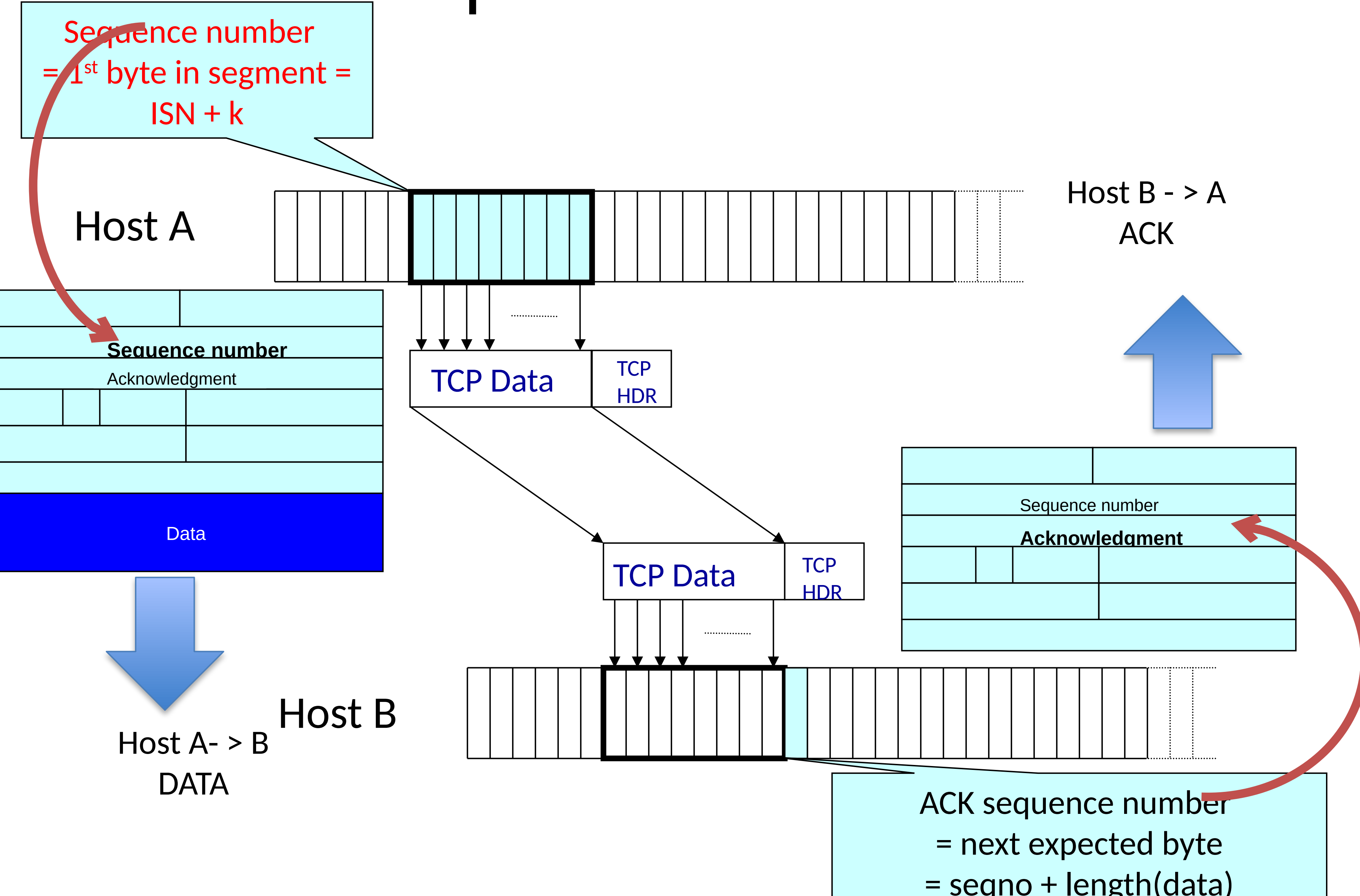


TCP Header

Starting byte offset of data carried in this segment



Sequence Numbers



TCP Sequences and ACKs

Sequence acknowledgement is given in terms of BYTES (not packets); the window also in terms of bytes.

number of packets ~= window size (bytes) / segment size

TCP is full-duplex by default:

- two independently flows of sequence numbers

Server/Client does not blindly map to Source/Tx or Destination/Rx

Duplex ‘piggybacking’ increases efficiency, but many applications or flows may only use data segments in one direction.

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Default receiver sends cumulative acknowledgements (like GBN)

ACKing and Sequence Numbers

- Sender sends packet
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, ….X+B-1]

- Upon receipt of packet, receiver sends an ACK
  - If all data prior to X already received:
    - ACK acknowledges X+B (because that is next expected byte)
  - If highest in-order byte received is Y s.t. (Y+1) < X
    - ACK acknowledges Y+1
    - Even if this has been ACK’d before

Normal Pattern

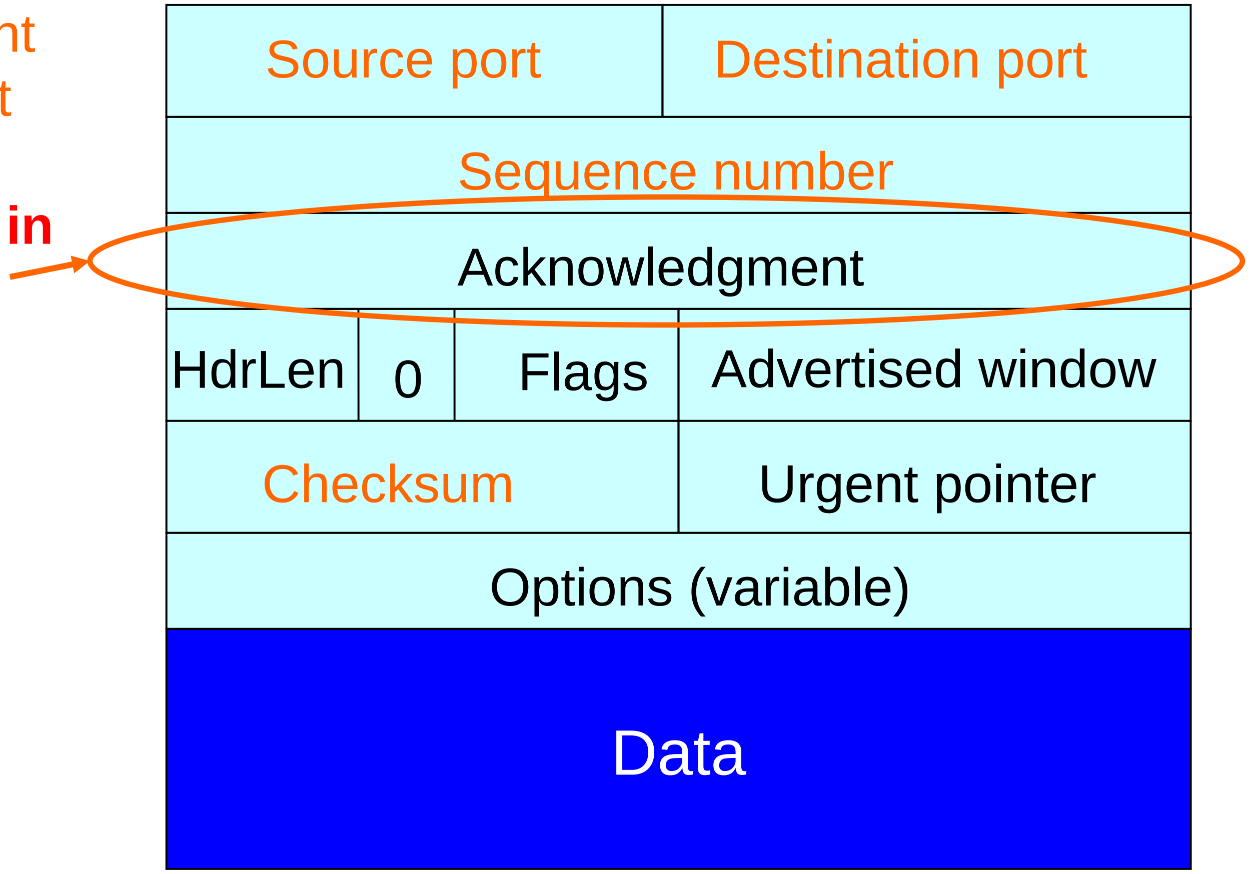
- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B

- Seqno of next packet is same as last ACK field

TCP Header

Acknowledgment gives seqno just beyond highest seqno received **in order**

("What Byte is Next")



What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers can buffer out-of-sequence packets (like SR)

Loss with cumulative ACKs

- Sender sends packets with 100B and seqnos.:
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, …

- Assume the fifth packet (seqno 500) is lost, but no others

- Stream of ACKs will be:
  - 200, 300, 400, 500, 500, 500, 500,…

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers may not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission

Loss detection on duplicate ACKs

- 'Duplicate ACKs' are a sign of an isolated loss
  - The lack of ACK progress meant 500 hasn't been delivered
  - Stream of ACKs means *some* packets are being delivered

- Therefore, instead of waiting for normal timeout:
  - Detect some number, k, extra (duplicate) ACKs for a segment
    - TCP uses k=3
  - Resend the following segment straightaway (aka fast re-transmit).
  - Do not implement rate or window kickdown (see later).

What does TCP do?

Most of our previous tricks, but a few differences

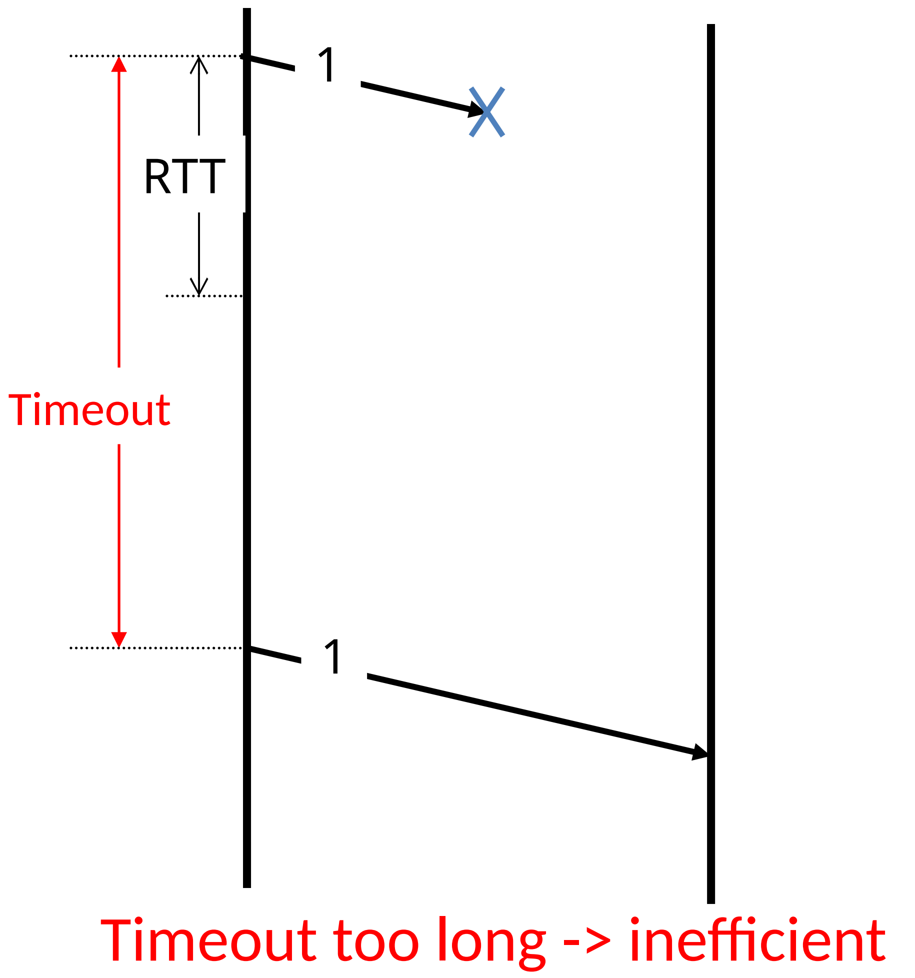
- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout.

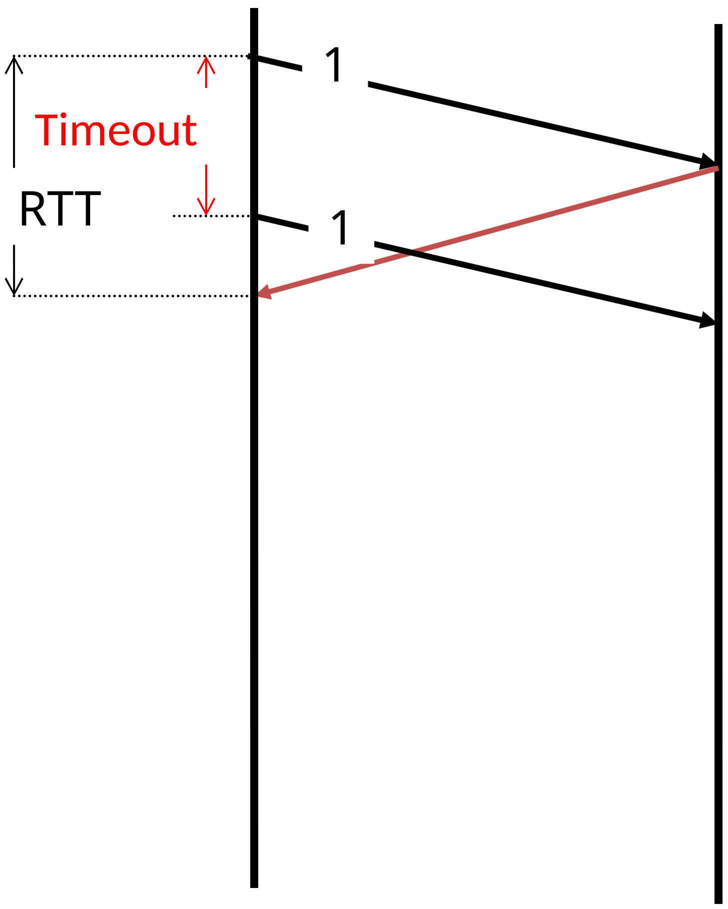
Retransmission Timeout

- If Tx hasn’t received an ACK by timeout, resend the first packet in the window.

- How do we pick a timeout duration?

Timing Illustration





Timeout too long -> inefficient

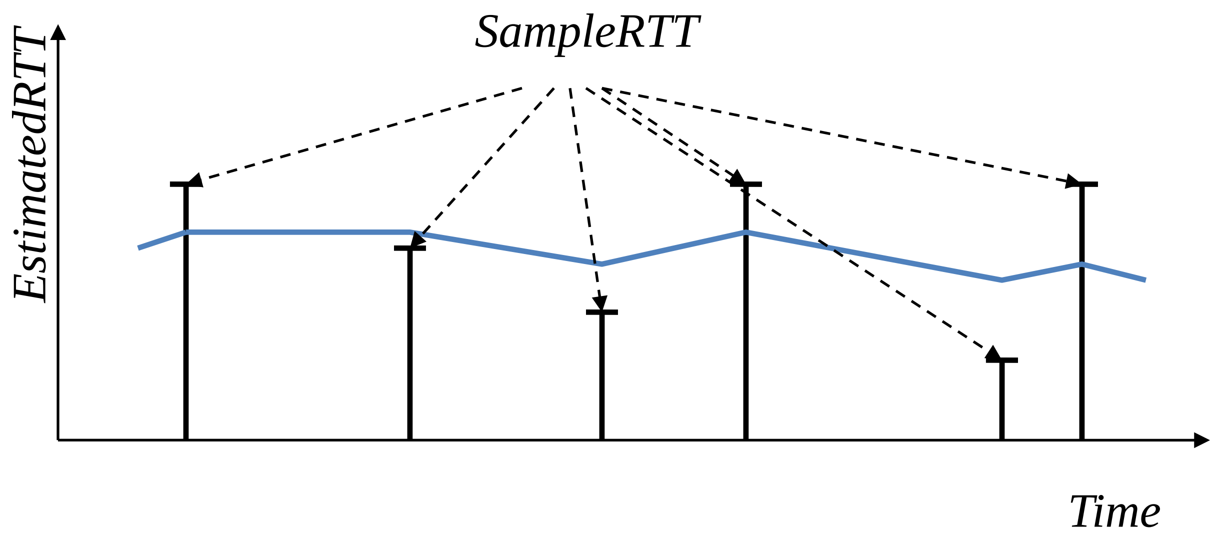
Timeout too short -> duplicate packets

Retransmission Timeout

- If haven't received ack by timeout, retransmit the first packet in the window
- How to set timeout?
  - Too long: connection has low throughput
  - Too short: retransmit packet that was just delayed
- Solution: make timeout proportional to RTT
- But how do we measure RTT?

RTT Measurement Jitter

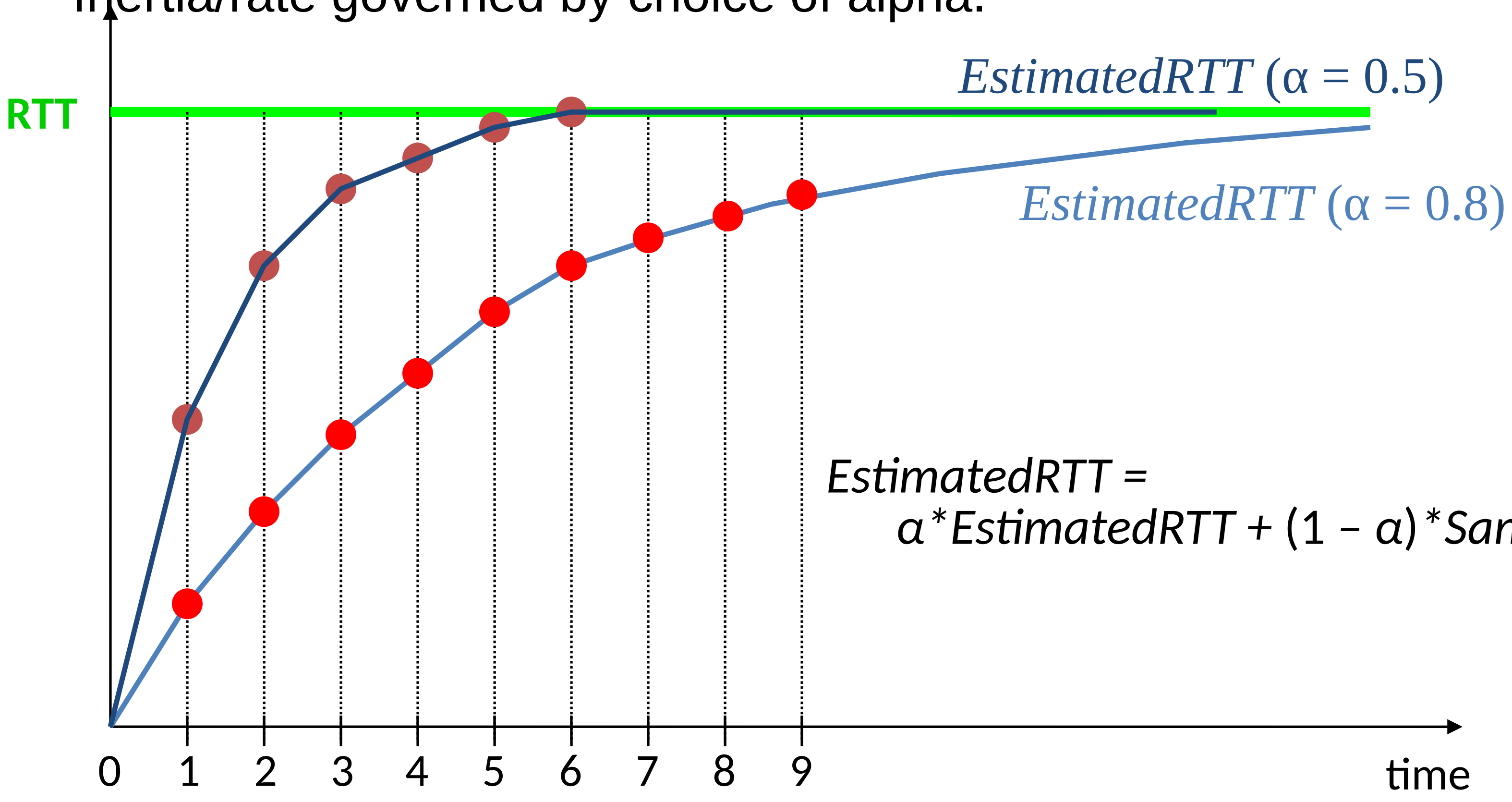
- Raw sample of RTT are noisy
- Linear interpolation not sensible



- Instead use exponential averaging:
  - *EstimatedRTT = α*EstimatedRTT + (1 – α)*SampleRTT*

Exponential-Weighted Moving Average (EWMA)

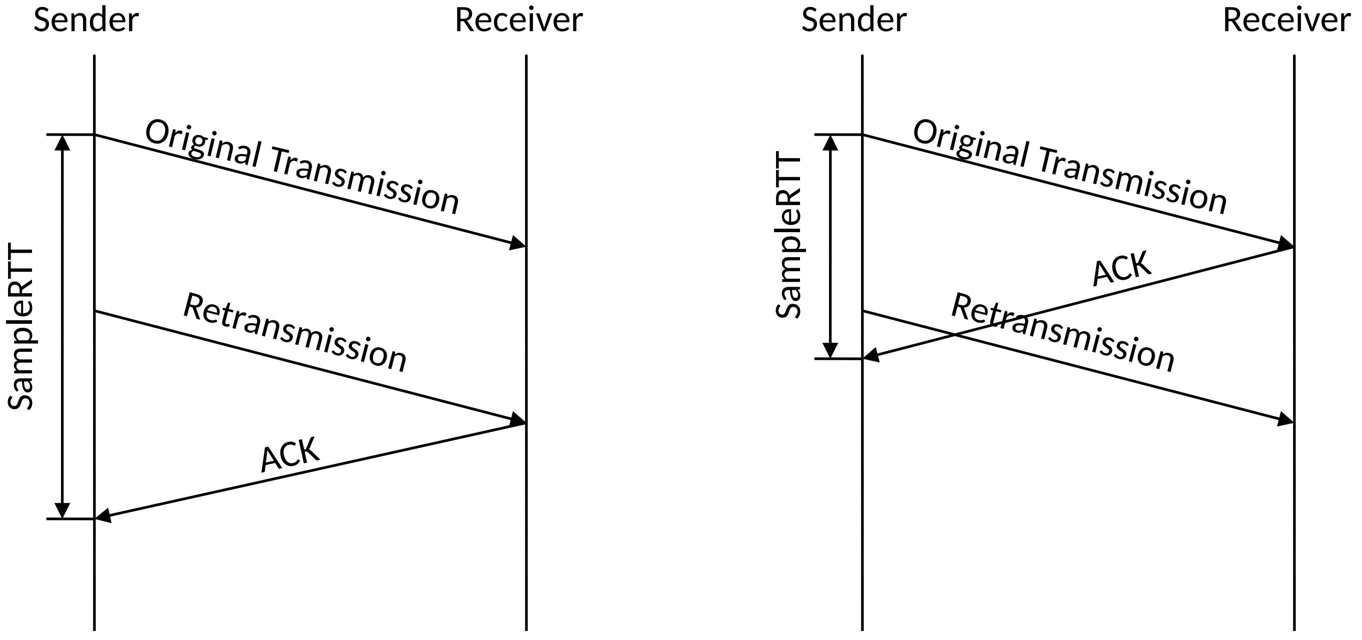
Implements a low-pass filter, rather like a running average.
Rejects high frequency jitter in individual measurements.
Converges perfectly when average RTT is a per-connection constant.
If varying, older values receive geometrically less significance.
Inertia/rate governed by choice of alpha.



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

Problem: Ambiguous Measurements

- How do we differentiate between the real ACK, and ACK of the retransmitted packet?



Karn/Partridge Algorithm

Discard junk measurements

- Measure *SampleRTT* only for original transmissions
  - Once a segment has been retransmitted, do not use it for any further measurements.
- Compute EstimatedRTT using $\alpha = 0.875$
- Baseline timeout value (RTO) = 2 × EstimatedRTT

Factor of two was a hard-coded margin for late acks:

- *Factor of two is too slow (wasteful)?*
- *Factor of two is too cautious (wasteful)?*

Backoff to avoid too many retransmissions

- Baseline timeout value (RTO) = 2 × EstimatedRTT
- Backoff/kickdown
  - Every time RTO timer expires, set RTO ← 2·RTO
  - (Up to maximum ≥ 60 sec)
  - Every time new measurement comes in (= successful original transmission), collapse RTO back to 2 × EstimatedRTT
- Implements an exponential backoff

Jacobson/Karels Algorithm

Dynamic timeout margin

- Problem: need to better capture variability in RTT
  - Directly measure deviation

- Deviation = **|** SampleRTT – EstimatedRTT **|**
- EstimatedDeviation: exponential average of Deviation

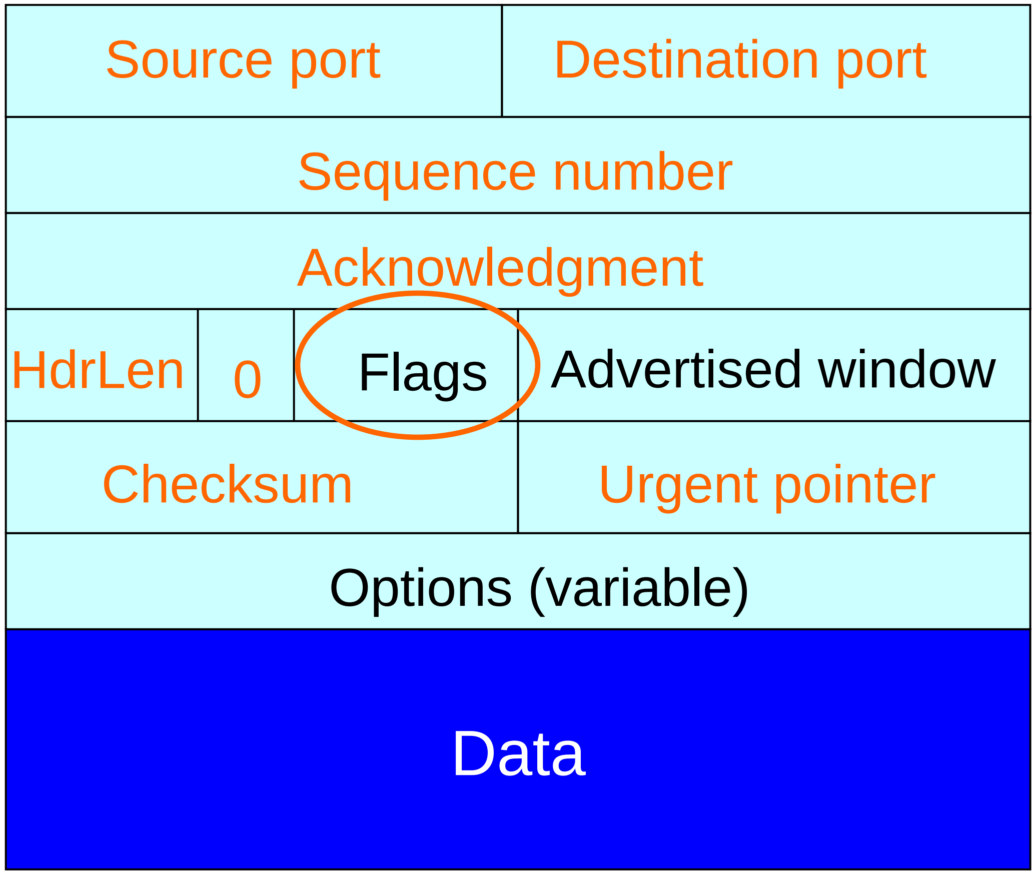
- RTO = EstimatedRTT + 4 x EstimatedDeviation

What does TCP do?

Most of our previous ideas, but some key differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

TCP Header: What's left?



TCP Connection Establishment and Initial Sequence Numbers

TCP connection establishment and teardown will not be lectured by DJG in 25/26.

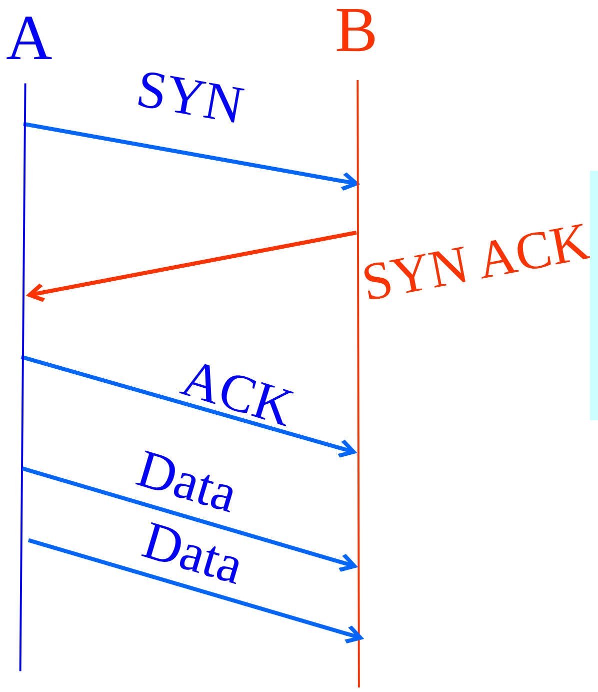
There will be no detailed discussion of the flags field.

Next slide to be covered is ‘application rate matching’.

Initial Sequence Number (ISN)

- Sequence number for the very first byte
- Why not just use ISN = 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … small chance an old packet is still in flight
- TCP therefore requires changing ISN
- Hosts exchange ISNs when they establish a connection

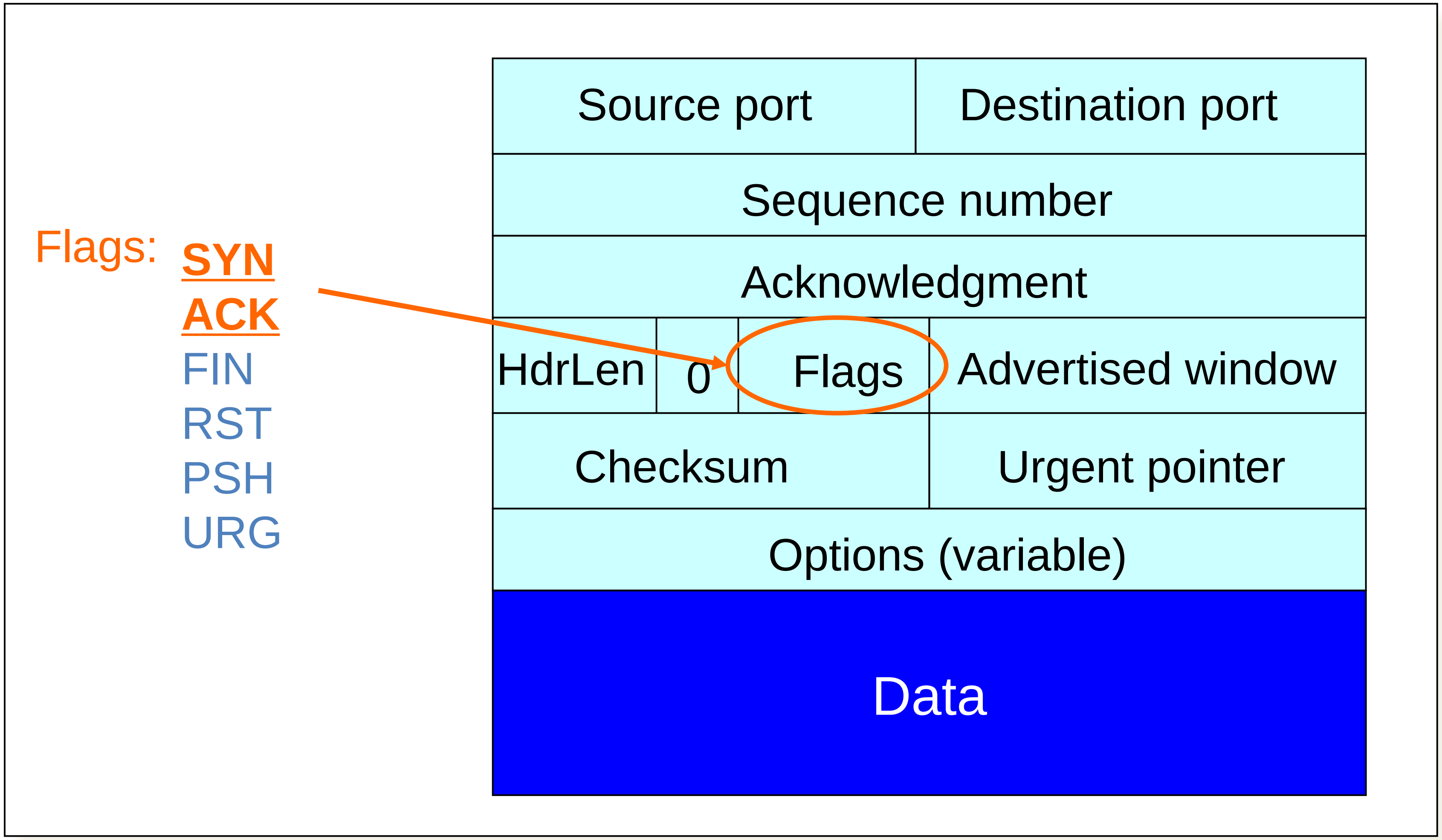
Establishing a TCP Connection



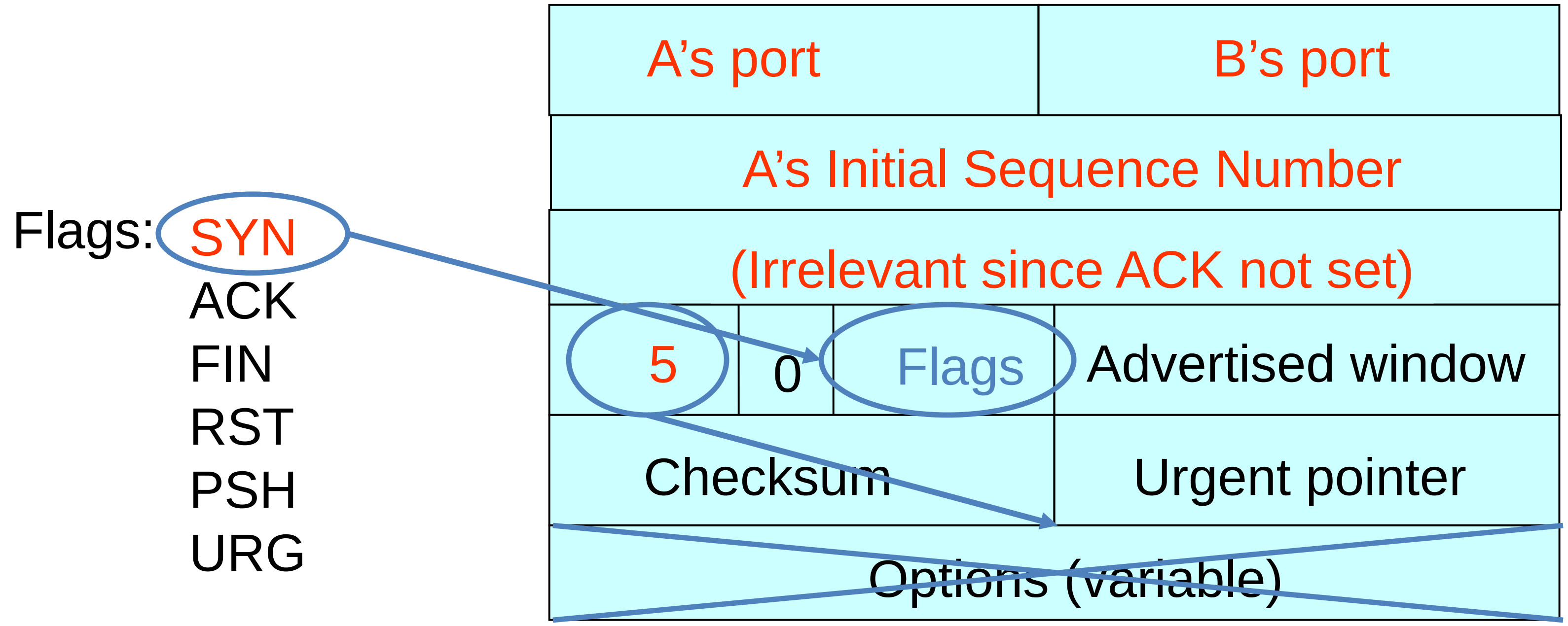
Each host tells its ISN to the other host.

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open; “synchronize sequence numbers”) to host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

TCP Header

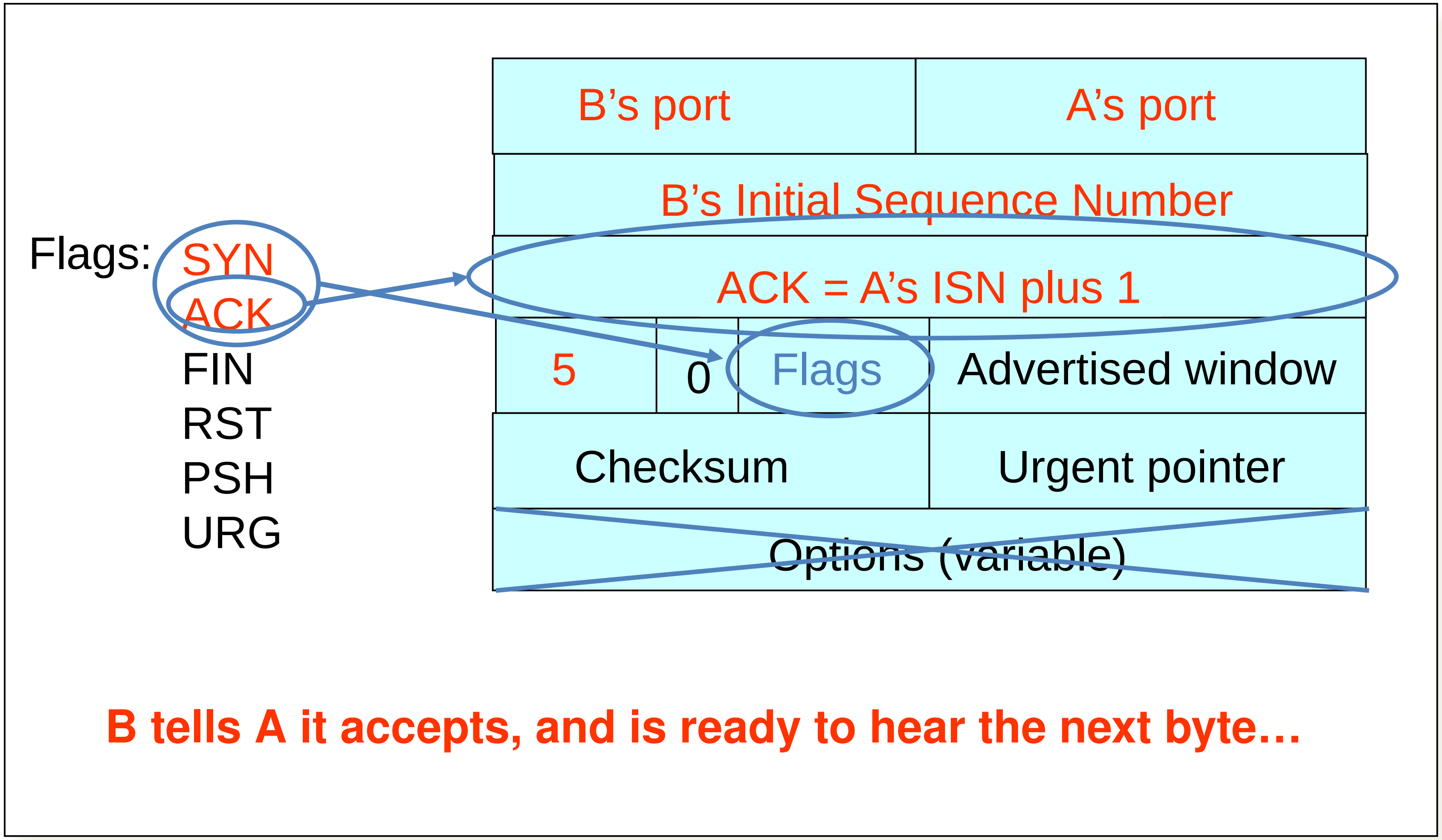

Flags:
SYN
ACK
FIN
RST
PSH
URG
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

Step 1: A's Initial SYN Packet



A tells B it wants to open a connection…

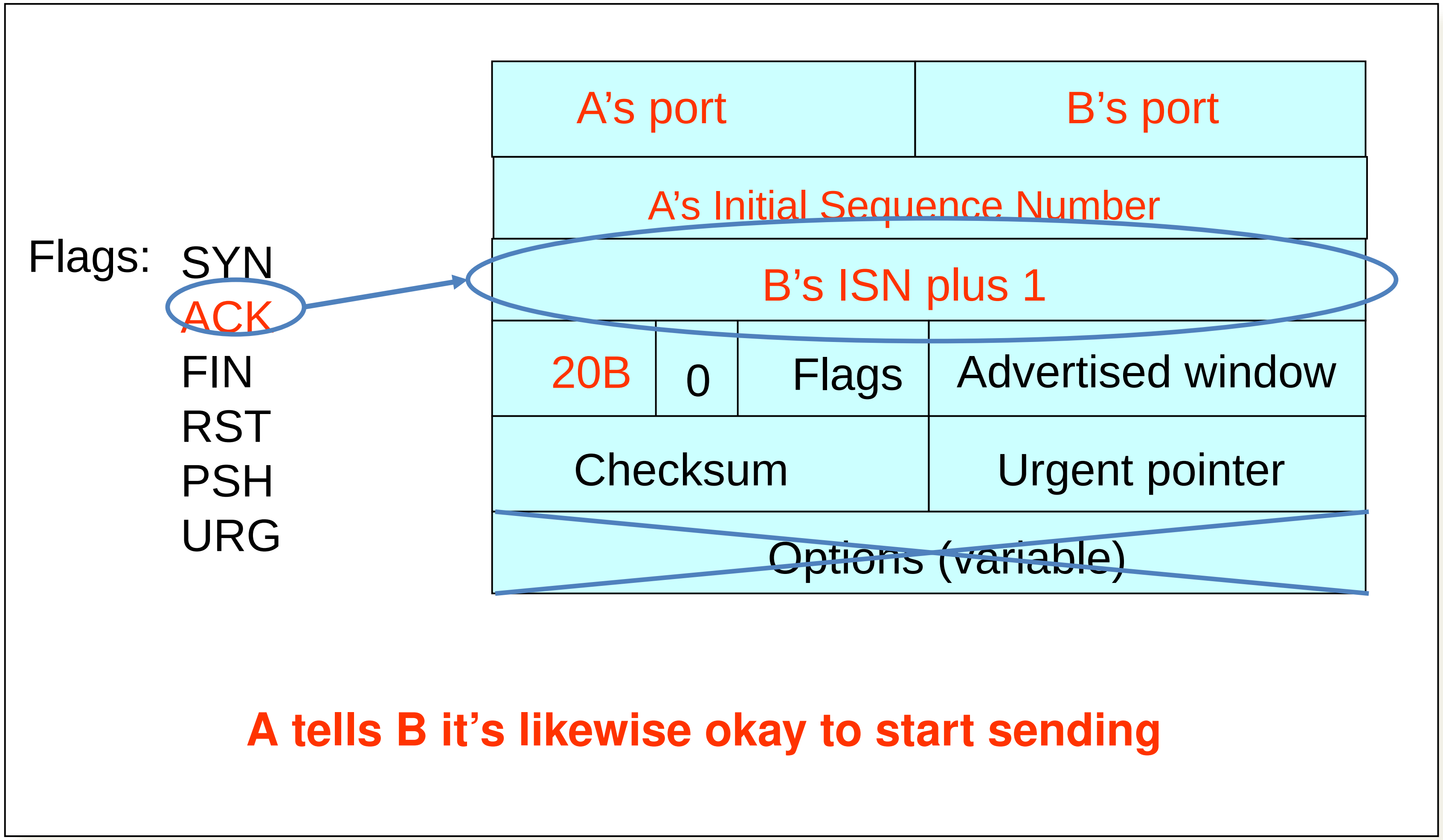
Step 2: B's SYN-ACK Packet



B tells A it accepts, and is ready to hear the next byte…

… upon receiving this packet, A can start sending data

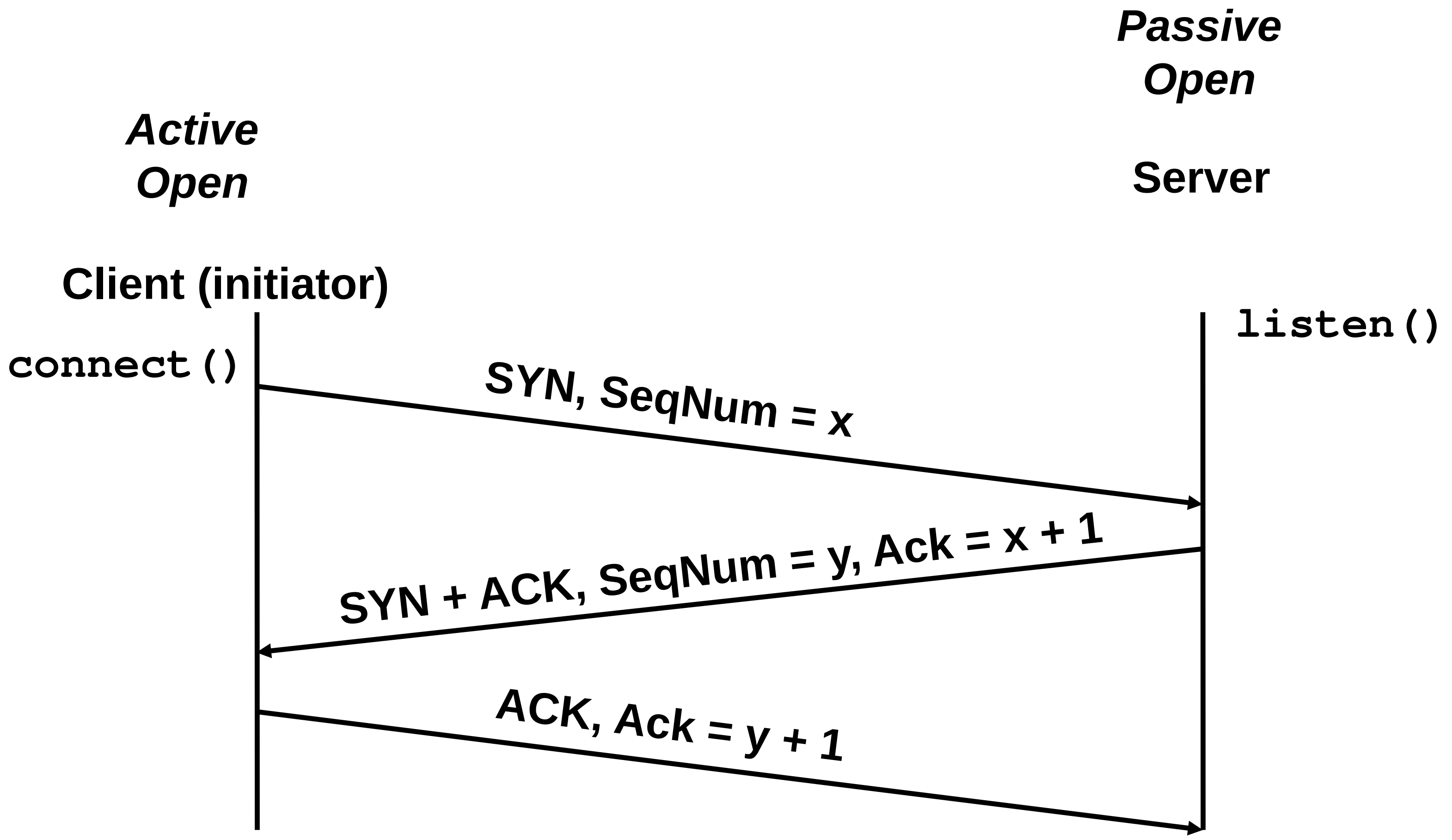
Step 3: A's ACK of the SYN-ACK



A tells B it's likewise okay to start sending

… upon receiving this packet, B can start sending data

Timing Diagram: 3-Way Handshaking

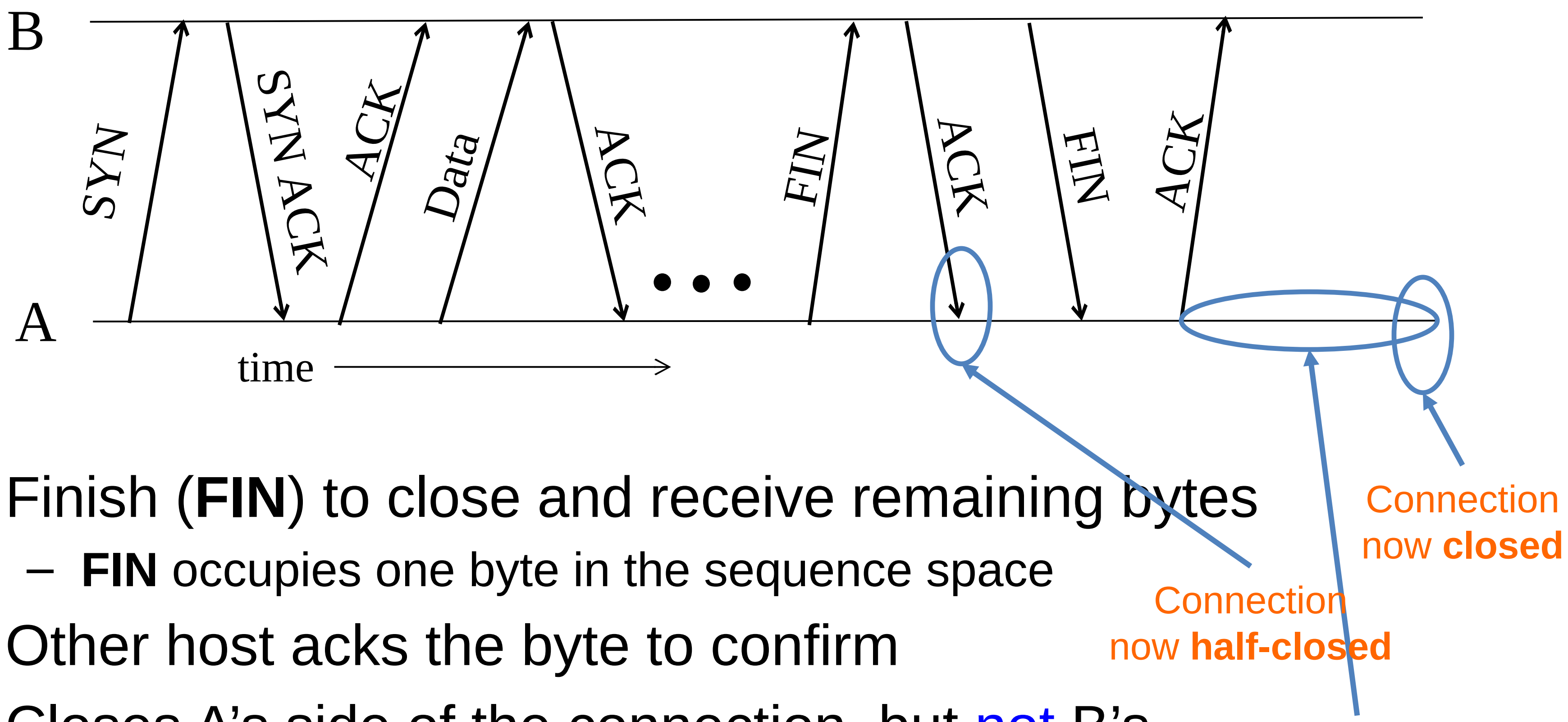


What if the SYN Packet Gets Lost?

- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or:
  - Server discards the packet (e.g., it’s too busy)
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and waits for the SYN-ACK
  - … and retransmits the SYN if needed
- How should the TCP sender set the timer?
  - Sender has no idea how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - **SHOULD** (RFCs 1122 & 2988) use default of 3 seconds
    - Some implementations instead use 6 seconds

Tearing Down the Connection

Normal Termination, One Side At A Time



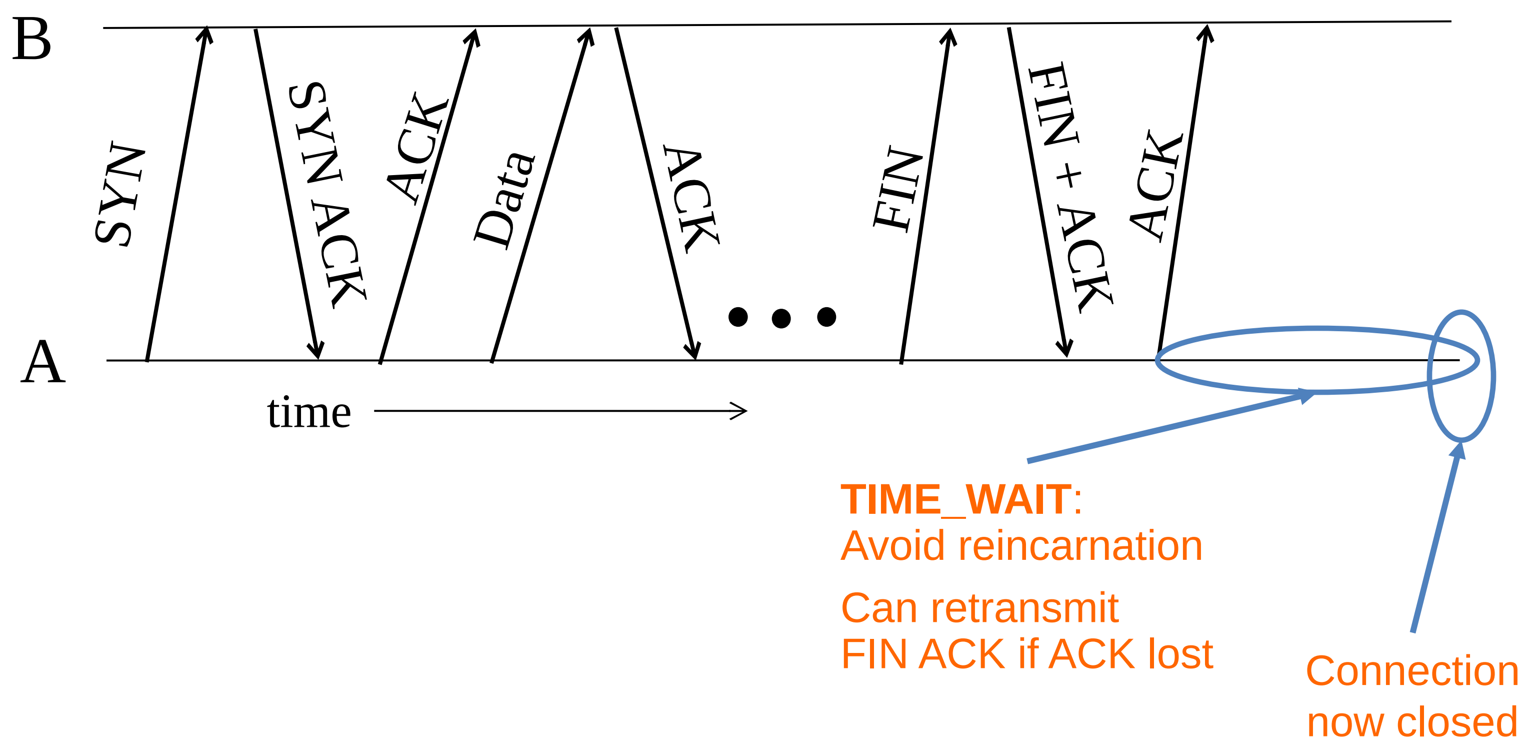
- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host acks the byte to confirm
- Closes A's side of the connection, but not B's
  - Until B likewise sends a **FIN**
  - Which A then acks

TIME_WAIT:

Avoid reincarnation

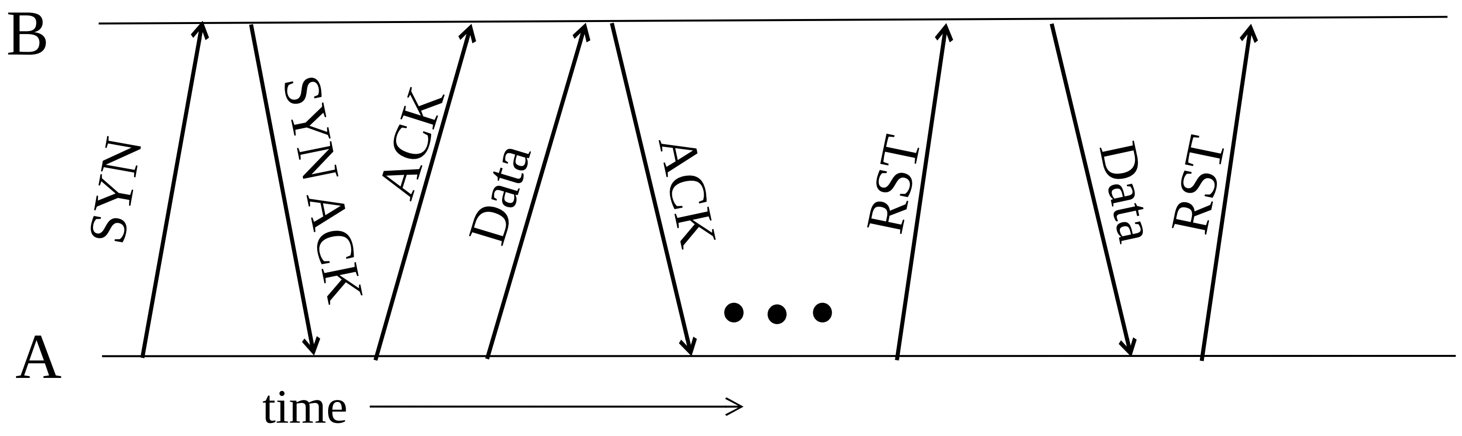
B will retransmit FIN if ACK is lost

Normal Termination, Both Together



- Same as before, but B sets **FIN** with their ack of A's **FIN**

Abrupt Termination

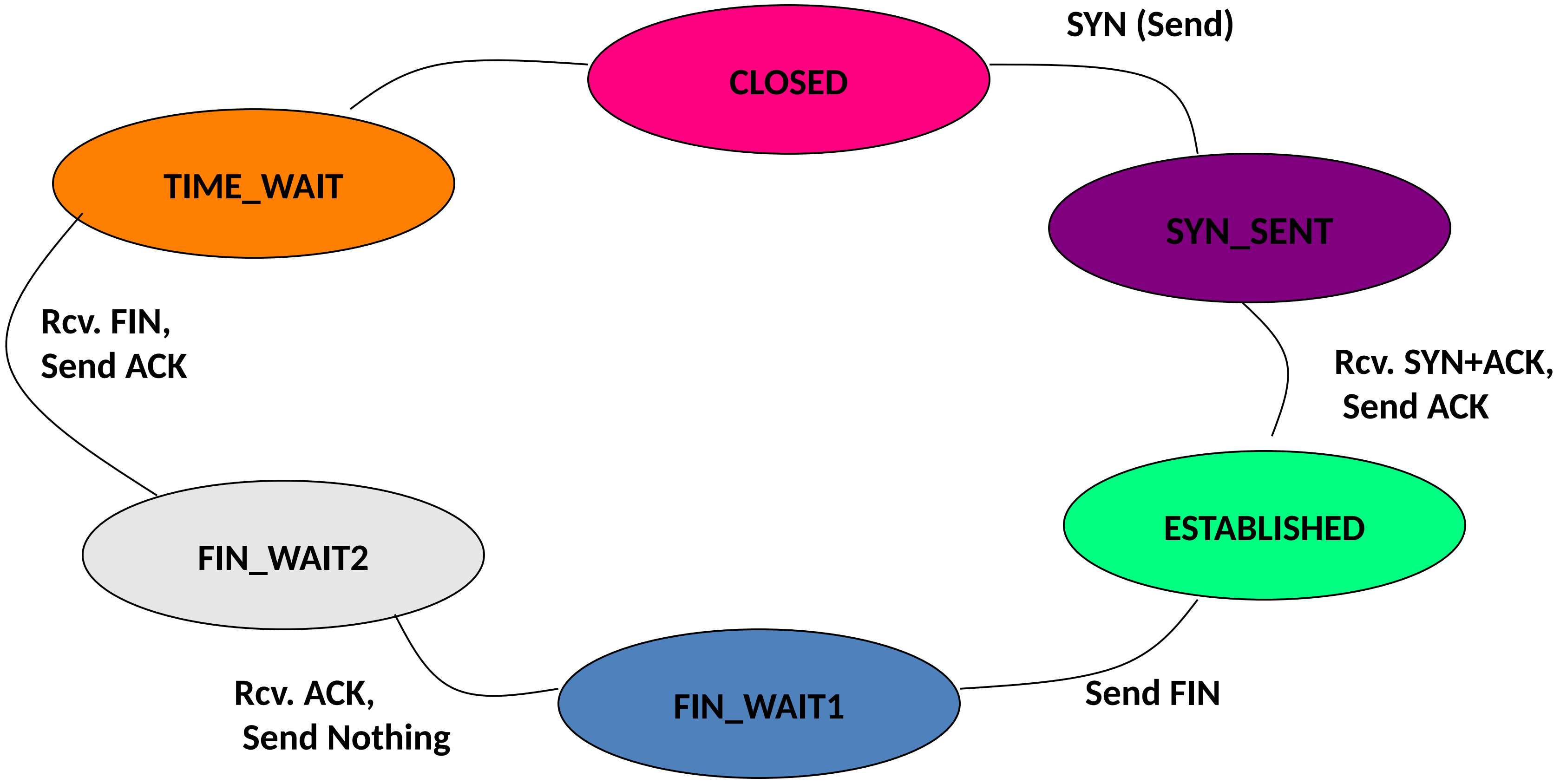


- A sends a RESET (**RST**) to B
  - E.g., because application process on A crashed
- That's it
  - B does not ack the **RST**
  - Thus, **RST** is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another **RST**

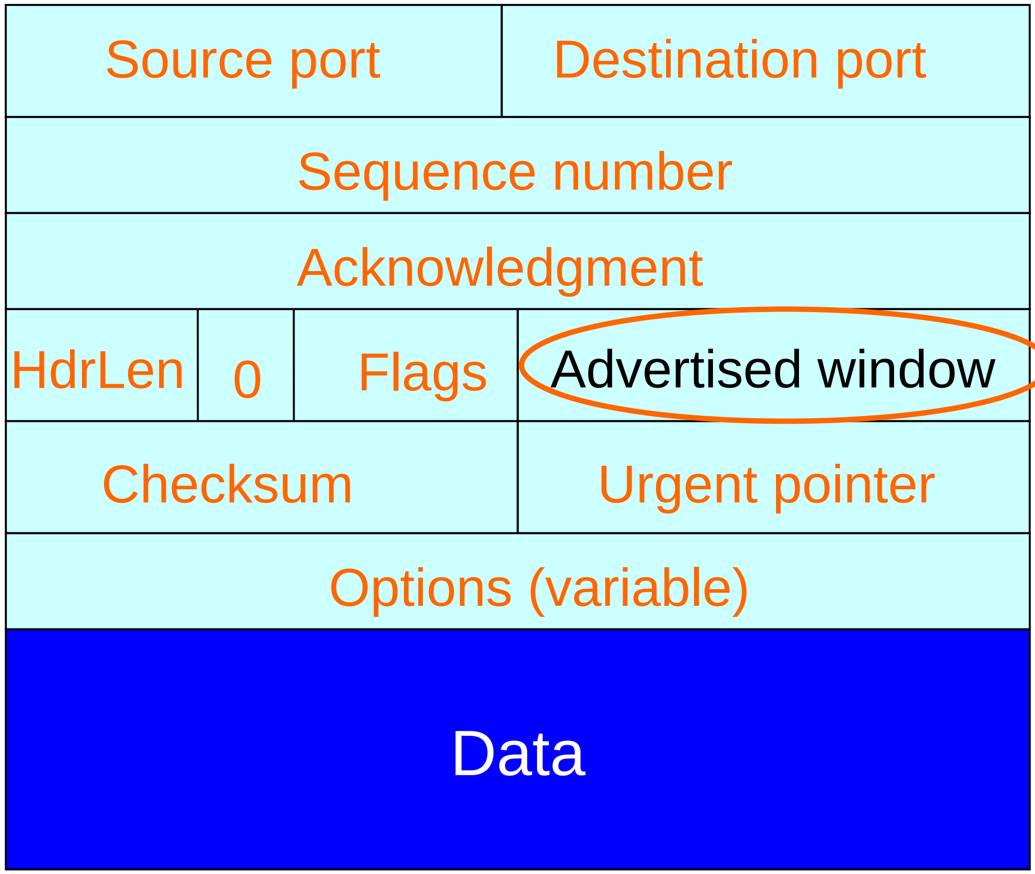
TCP State Transitions

Data, ACK exchanges are in here

An Simpler View of the Client Side



TCP Header

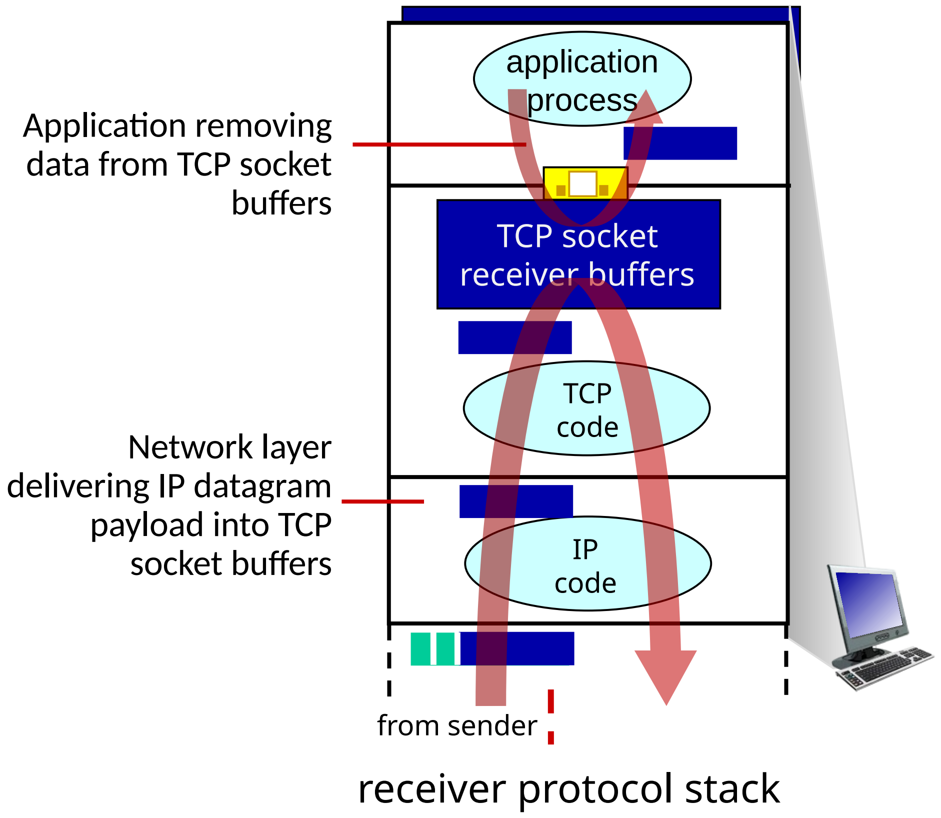


Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP

TCP window sizing (application rate matching)

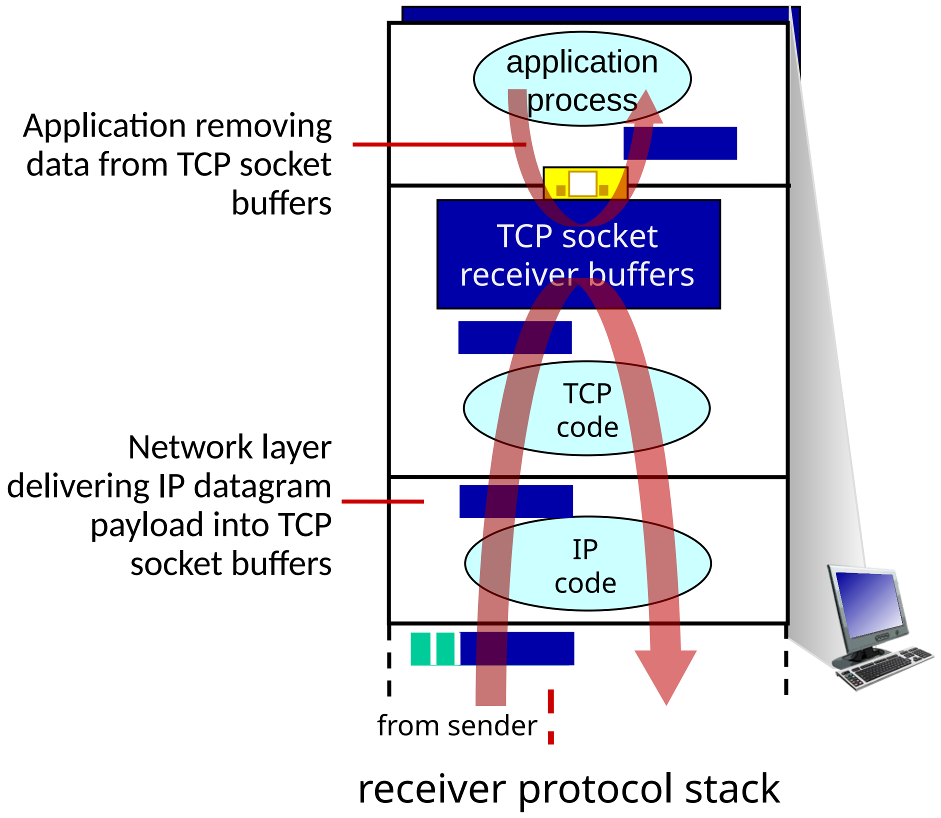
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



application process
Application removing data from TCP socket buffers
TCP socket receiver buffers
TCP code
Network layer delivering IP datagram payload into TCP socket buffers
IP code
from sender
receiver protocol stack

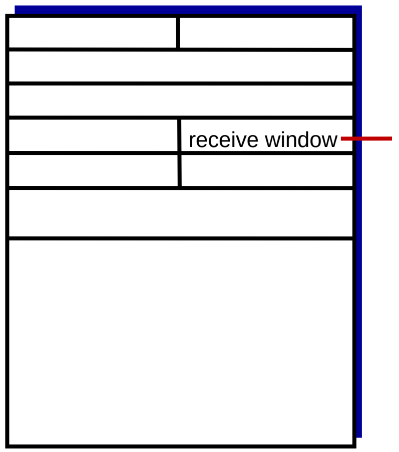
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

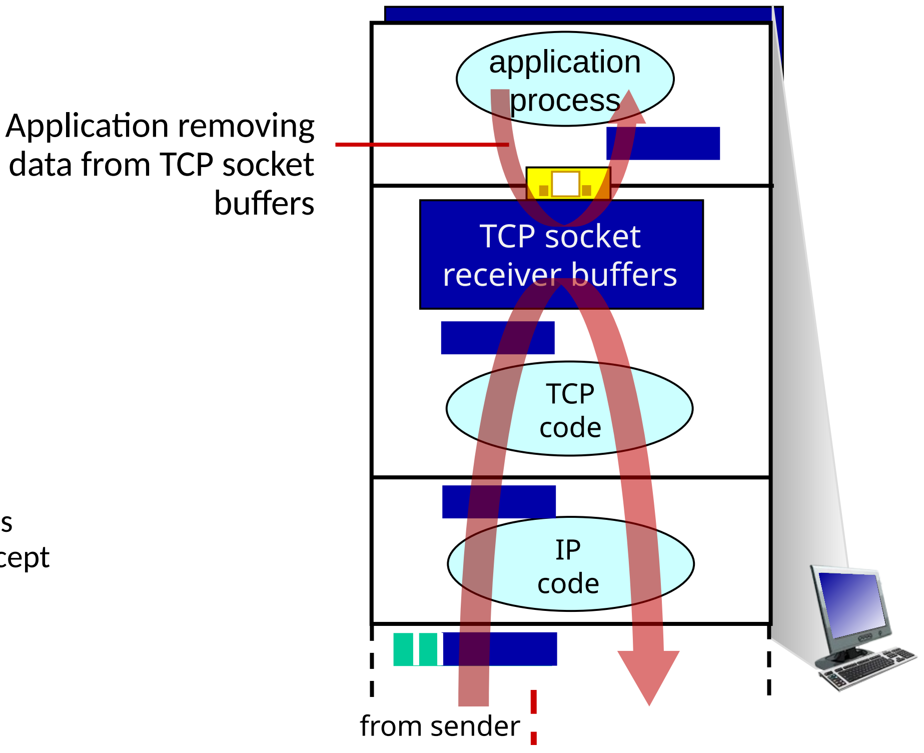

Application removing data from TCP socket buffers
application process
TCP socket receiver buffers
TCP code
Network layer delivering IP datagram payload into TCP socket buffers
IP code
from sender
receiver protocol stack

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



flow control: # bytes receiver willing to accept



Application removing data from TCP socket buffers
application process
TCP socket receiver buffers
TCP code
IP code
from sender

receiver protocol stack

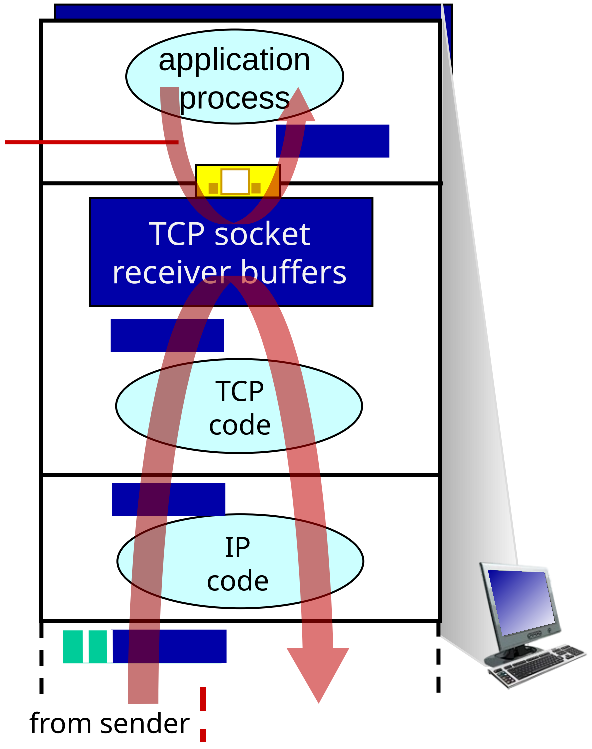
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

flow control

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

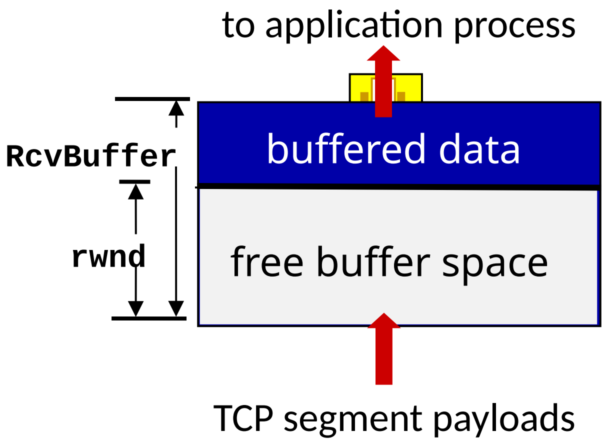
Application removing data from TCP socket buffers



receiver protocol stack

TCP flow control

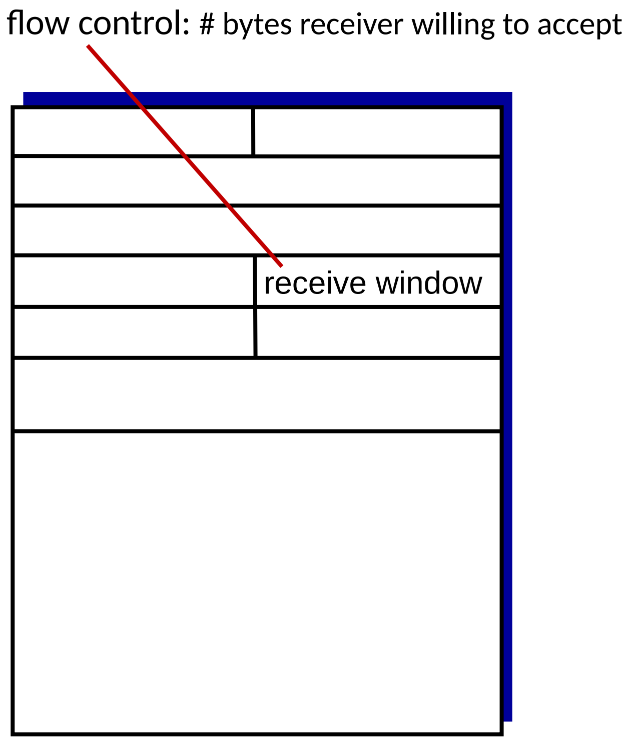
- TCP receiver 'advertises' free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- Tx limits amount of unACK'd ('in-flight') data to received **`rwnd`**
- guarantees Rx buffer will not overflow



TCP receiver-side buffering

TCP flow control

- TCP receiver 'advertises' free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACK'd ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP segment format

Later we'll have rwnd and cwnd.

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling sender's rate
- What's missing?

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- **Congestion Control in TCP**

We have seen:

- Flow control: adjusting the sending rate to keep from overwhelming a slow *receiver*

Now lets attend…

- Congestion control: adjusting the sending rate to keep from overloading the *network*

Principles of congestion control

Congestion:

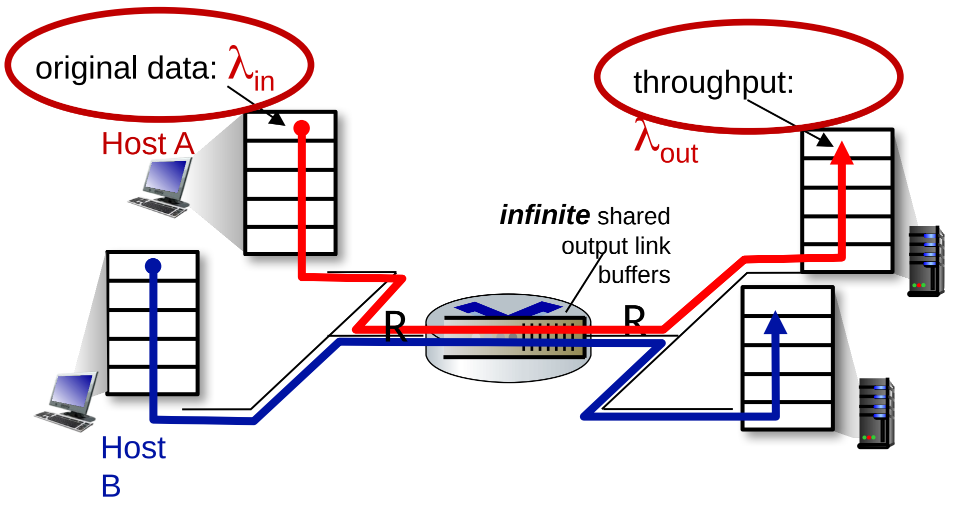
- informally: 'too many sources sending too much data too fast for *network* to handle'
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!

congestion control: too many senders, sending too fast

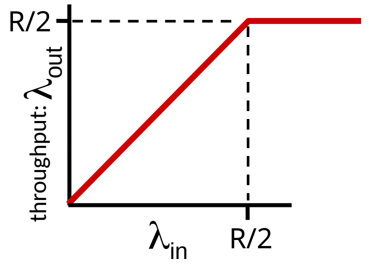
flow control: one sender too fast for one receiver

Causes/costs of congestion: scenario 1

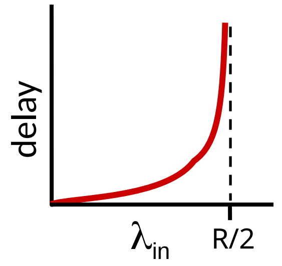
Simplest scenario:
- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed


original data: λ_{in}
throughput: λ_{out}
Host A
Host B
infinite shared output link buffers
R
R

Q: What happens as arrival rate λ_{in} approaches R/2?



maximum per-connection throughput: R/2

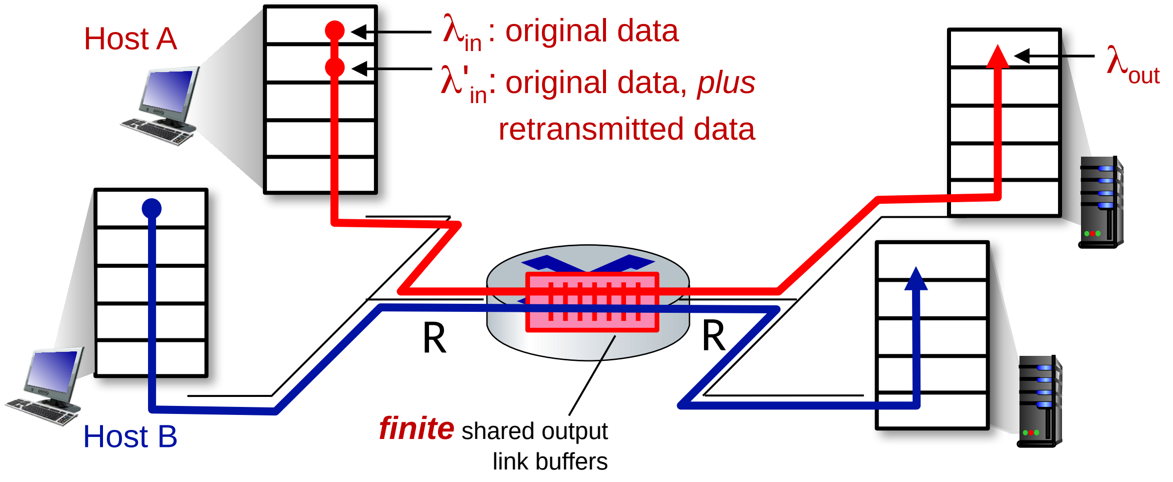


large delays as arrival rate λ_{in} approaches capacity

These router drawings have traffic combining outside the router! Please assume traffic arrives at distinct input ports, each of capacity R. Assume also that the subsequent split is in a further router, not shown.

Causes/costs of congestion: scenario 2

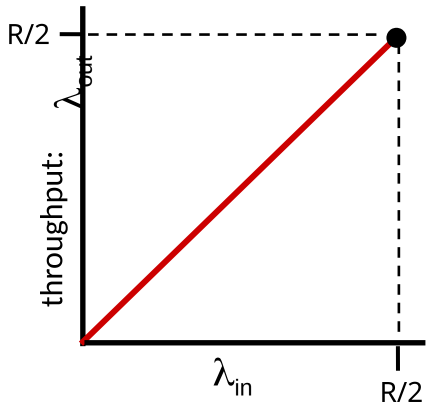
- one router, *finite* buffers
- sender retransmits lost, timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

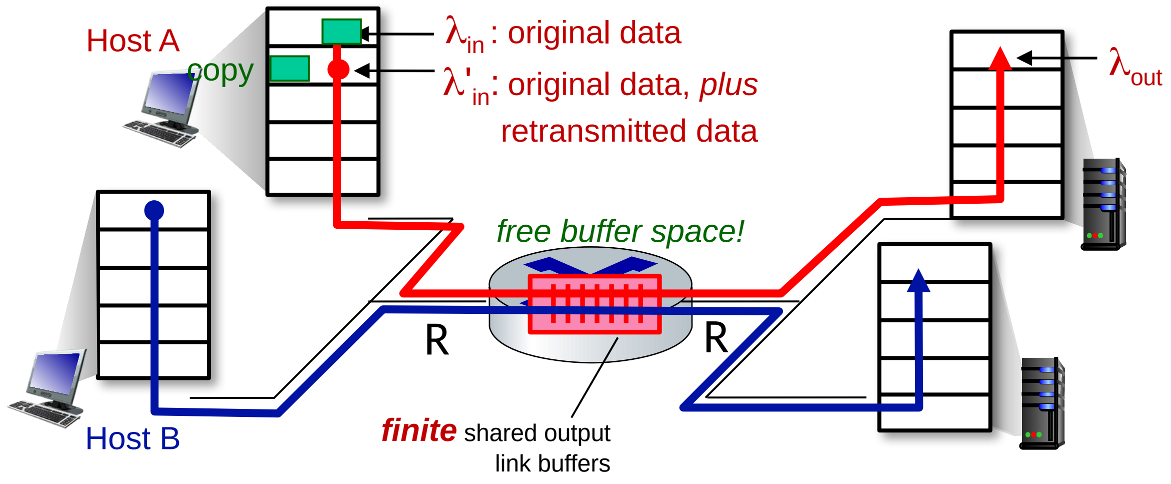

Host A
λ_{in} : original data
λ'_{in}: original data, *plus* retransmitted data
λ_{out}
Host B
R
R
finite shared output link buffers

Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available

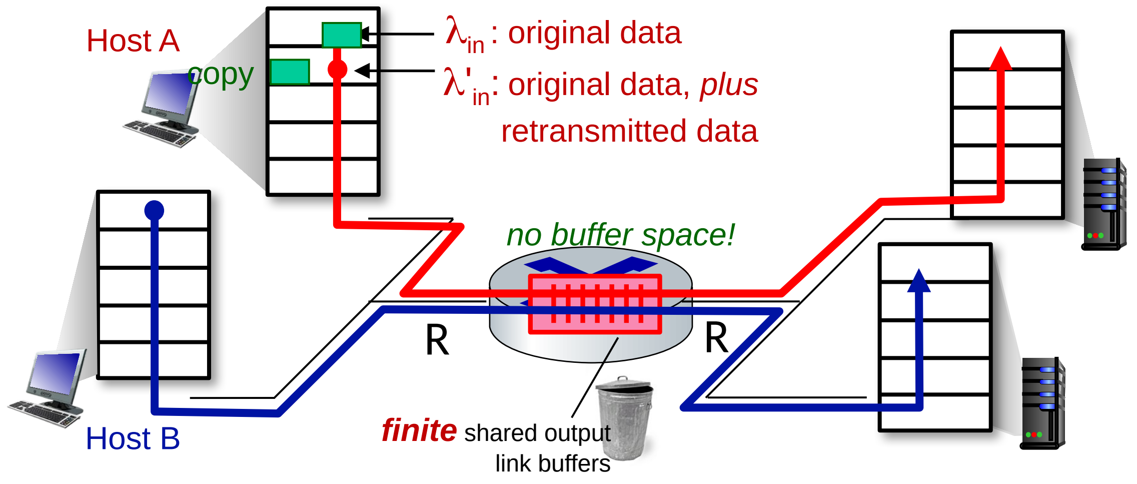



Host A
copy
λ_{in} : original data
λ'_{in}: original data, *plus* retransmitted data
λ_{out}
free buffer space!
R
R
finite shared output link buffers
Host B

Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

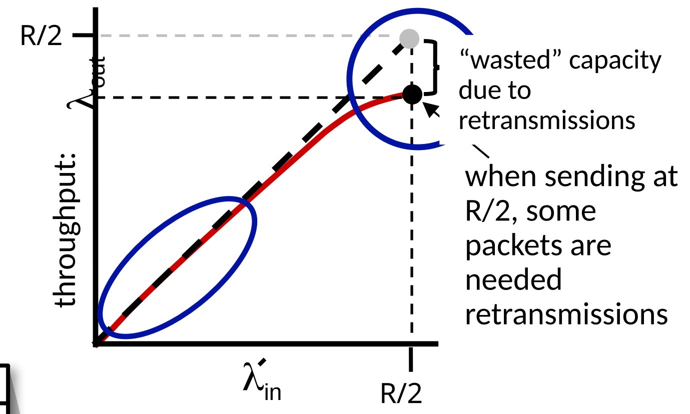
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost

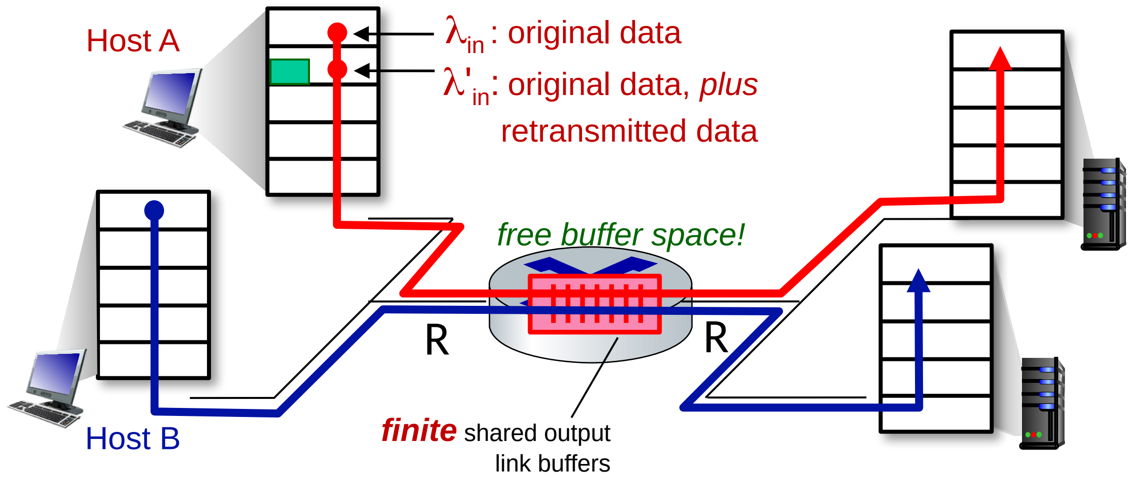

Host A
copy
λ_{in} : original data
λ'_{in}: original data, *plus* retransmitted data
no buffer space!
R
R
Host B
finite shared output link buffers

Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



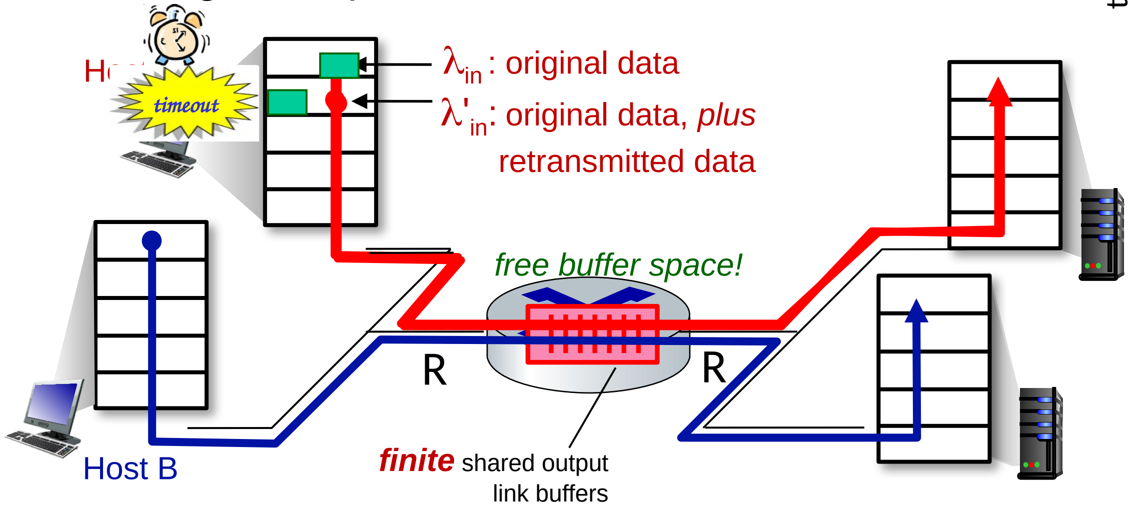


Host A
λ_{in} : original data
λ'_{in}: original data, *plus* retransmitted data
free buffer space!
R
R
finite shared output link buffers
Host B

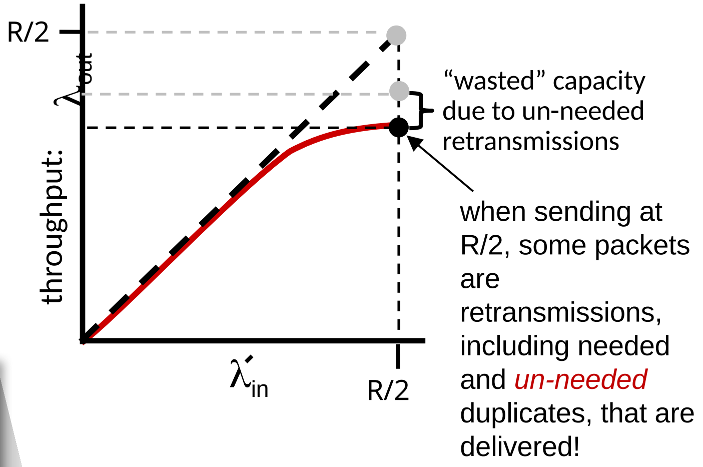
Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



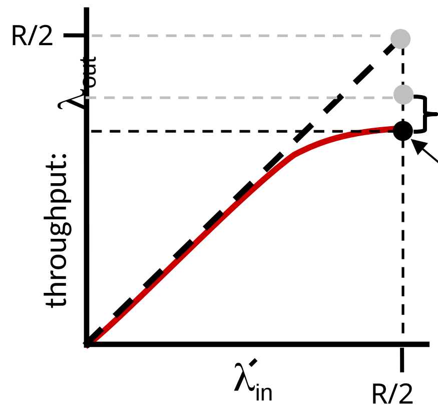
timeout
H
λ_{in} : original data
λ'_{in}: original data, *plus* retransmitted data
free buffer space!
R
R
Host B
finite shared output link buffers



Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



"wasted" capacity due to un-needed retransmissions

when sending at R/2, some packets are retransmissions, including needed and *un-needed* duplicates, that are delivered!

"costs" of congestion:

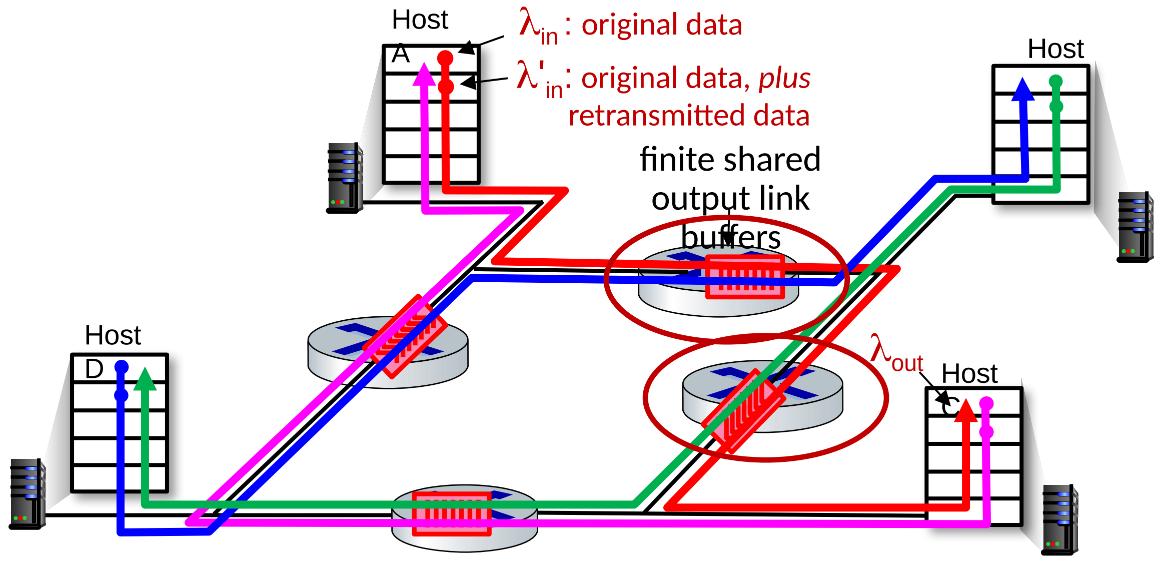
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

Causes/costs of congestion: scenario 3

- *four* senders
- *multi-hop* paths
- timeout/retransmit
- perfect/near symmetry

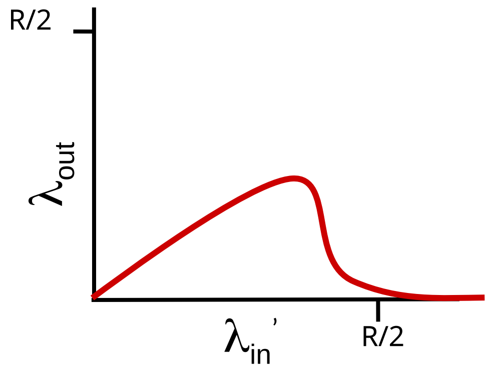
Q: what happens as λ_{in} and λ_{in}' increase ?

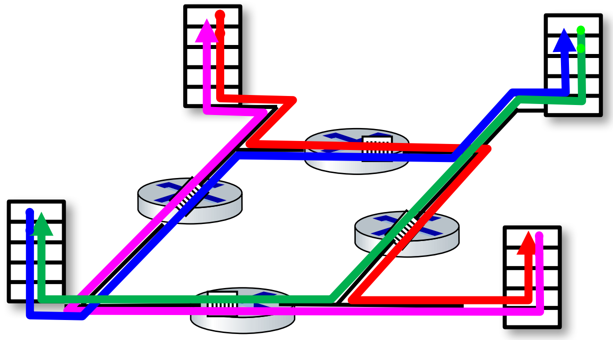
A: despite symmetry, blue traffic at top router has half the chance of red traffic, and can be excessively dropped.


Host A
λ_{in} : original data
λ'_{in}: original data, *plus* retransmitted data
Host B
finite shared output link buffers
Host D
λ_{out}
Host C

Traffic that has gone further before being dropped wastes more resource.

Causes/costs of congestion: scenario 3





So a further 'cost' of congestion:

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

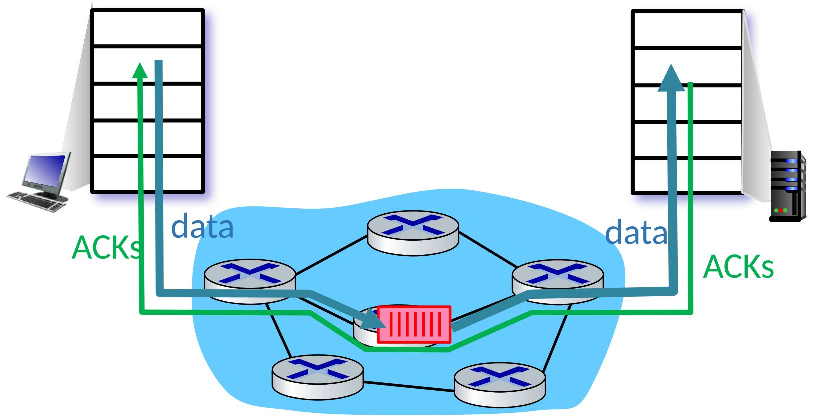
Causes/costs of congestion: insights

- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decrease effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream

Approaches towards congestion control (e2e vs explicit)

1/ End-end congestion control:

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

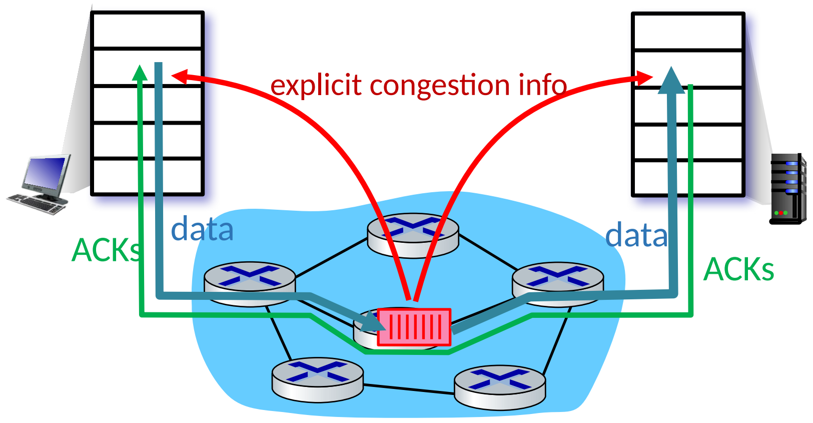


Approaches towards congestion control (e2e vs explicit)

2/ Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate

- TCP ECN, ATM, DECbit protocols



Three Issues to Consider

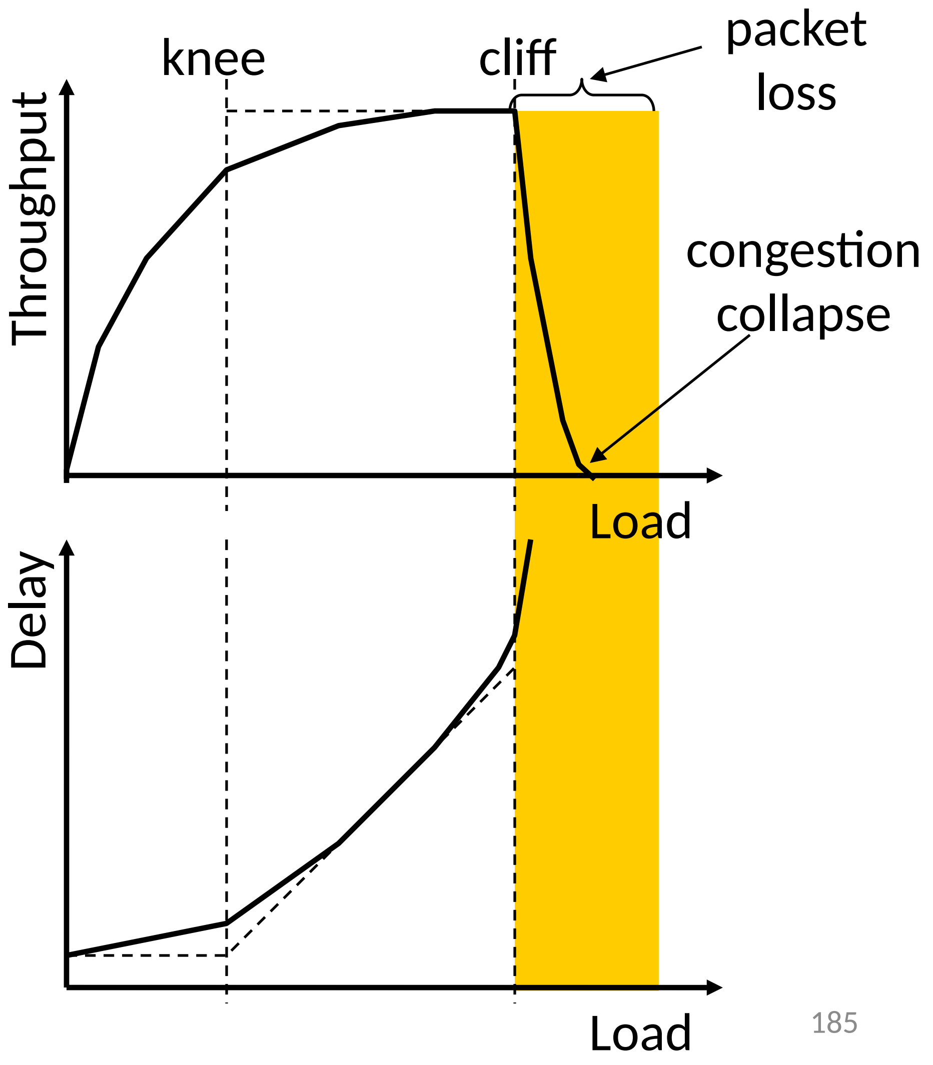
- Discovering the available (bottleneck) bandwidth

- Adjusting to variations in bandwidth

- Sharing bandwidth between flows

View from a single flow

- Knee – point after which
  - Throughput increases slowly
  - Delay increases fast
- Cliff – point after which
  - Throughput starts to drop to zero (congestion collapse)
  - Delay approaches infinity



General Approaches in Theory

(0) Send without care

– Many packet drops

General Approaches in Theory

(0) Send without care

(1) Reservations

- Pre-arrange bandwidth allocations
- Requires negotiation before sending packets
- Low utilization

General Approaches in Theory

(0) Send without care

(1) Reservations

(2) Pricing

- Don’t drop packets for the high-bidders
- Requires payment model

General Approaches in Theory

(0) Send without care

(1) Reservations

(2) Pricing

(3) Dynamic Adjustment

- Hosts probe network; infer level of congestion; adjust
- Network reports congestion level to hosts; hosts adjust
- Combinations of the above
- Simple to implement but suboptimal, messy dynamics.

General Approaches in Theory

(0) Send without care

(1) Reservations

(2) Pricing

(3) Dynamic Adjustment

All three techniques have their place

- *Generality* of dynamic adjustment has proven powerful
- Doesn’t presume business model, traffic characteristics, application requirements; does assume good citizenship

Who Takes Care of Congestion?

- Network? End hosts? Both?

- TCP's approach:
  - **End hosts** adjust sending rate
  - Based on **implicit feedback** from network

- As seen, not the only approach
  - A consequence of history rather than planning

Some History: TCP in the 1980s

- Sending rate only limited by flow control
  - Packet drops senders (repeatedly!) retransmit a full window’s worth of packets

- Led to “congestion collapse” starting Oct. 1986
  - Throughput on the NSF network dropped from 32Kbits/s to 40bits/sec

- “Fixed” by Van Jacobson’s development of TCP’s congestion control (CC) algorithms

Jacobson’s Approach

- Extend TCP’s existing window-based protocol but adapt the window size in response to congestion
  - required no upgrades to routers or applications!
  - patch of a few lines of code to TCP implementations

- A pragmatic and effective solution
  - but many other approaches exist

- Extensively improved on since
  - topic now sees less activity in ISP contexts
  - but is making a comeback in datacenter environments

TCP's Approach in a Nutshell

- TCP connection has window
  - Controls number of packets in flight
  - (Intended to avoid Rx buffer overrun)
- Sending rate: ~Window/RTT
- Vary window size to control sending rate

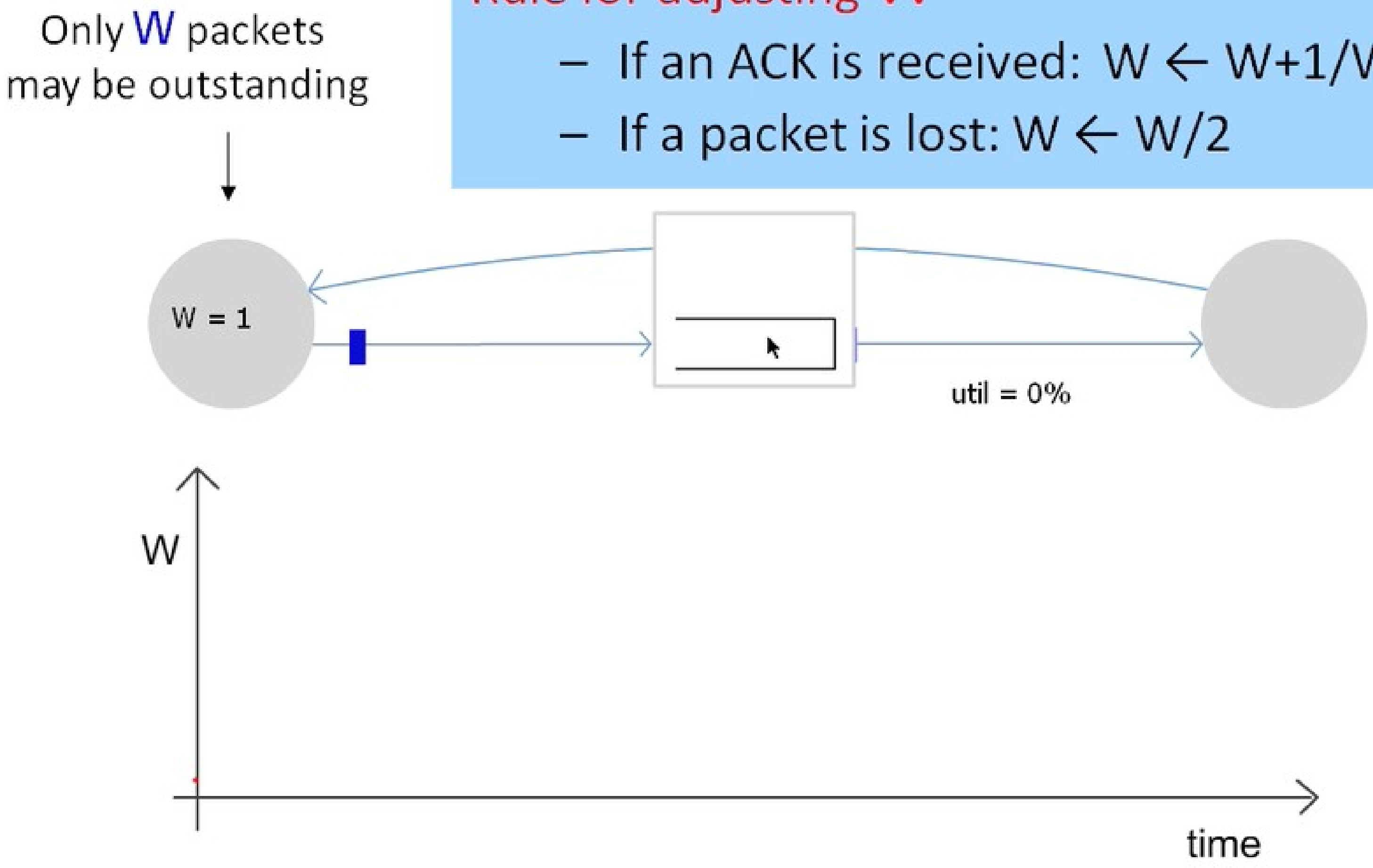
Buffer Sizing Story

	Rule-of-thumb
# of packets	1,000,000
Intuition	TCP Sawtooth
Assume	Single TCP Flow, 100% Utilization
Evidence	Simulation, Emulation

Continuous ARQ (TCP) adapting to congestion

Rule for adjusting W

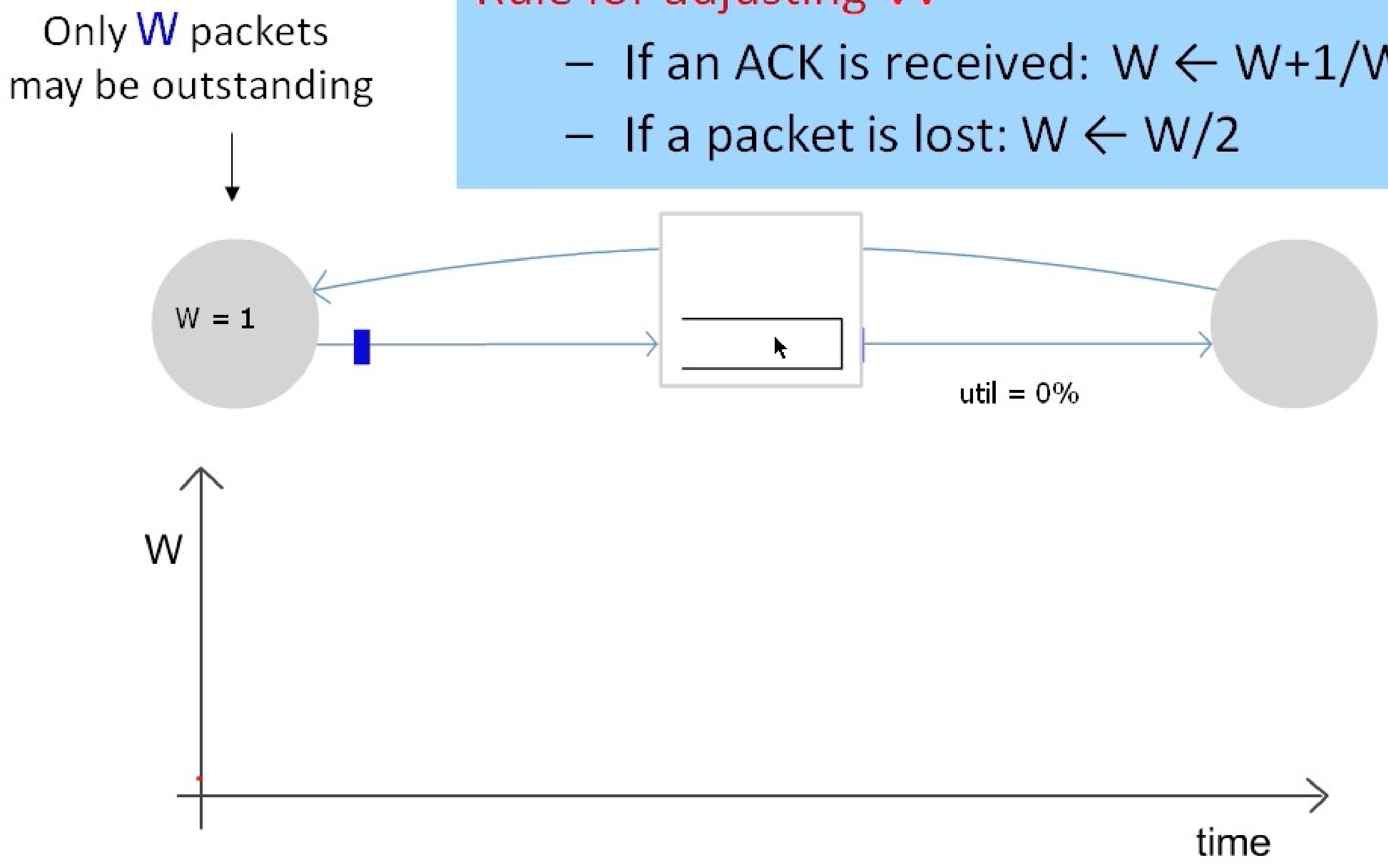
- If an ACK is received: W ← W+1/W
- If a packet is lost: W ← W/2



Continuous ARQ (TCP) adapting to congestion

Rule for adjusting W

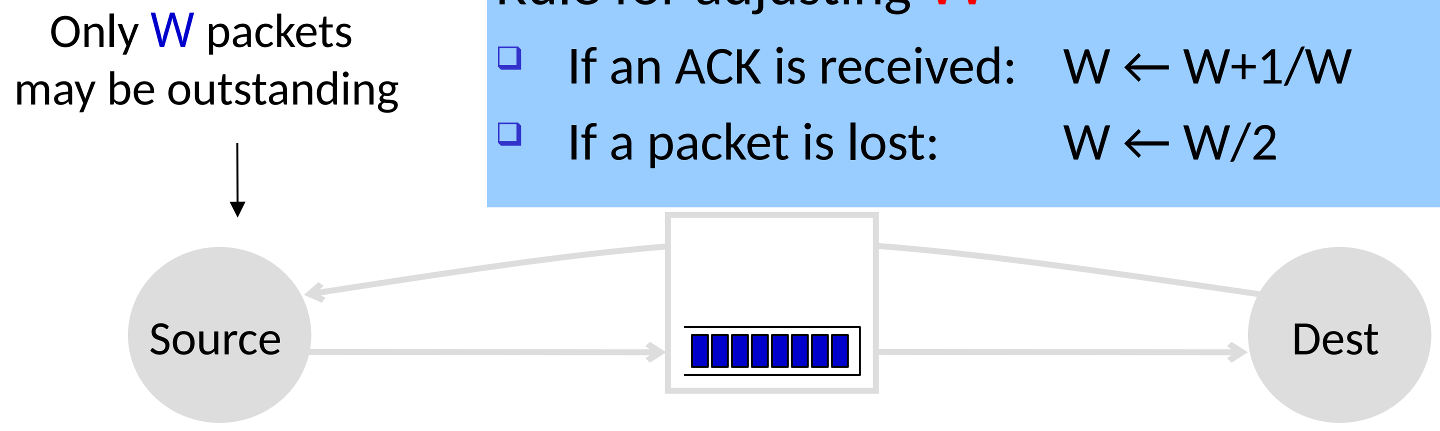
- If an ACK is received: W ← W+1/W
- If a packet is lost: W ← W/2

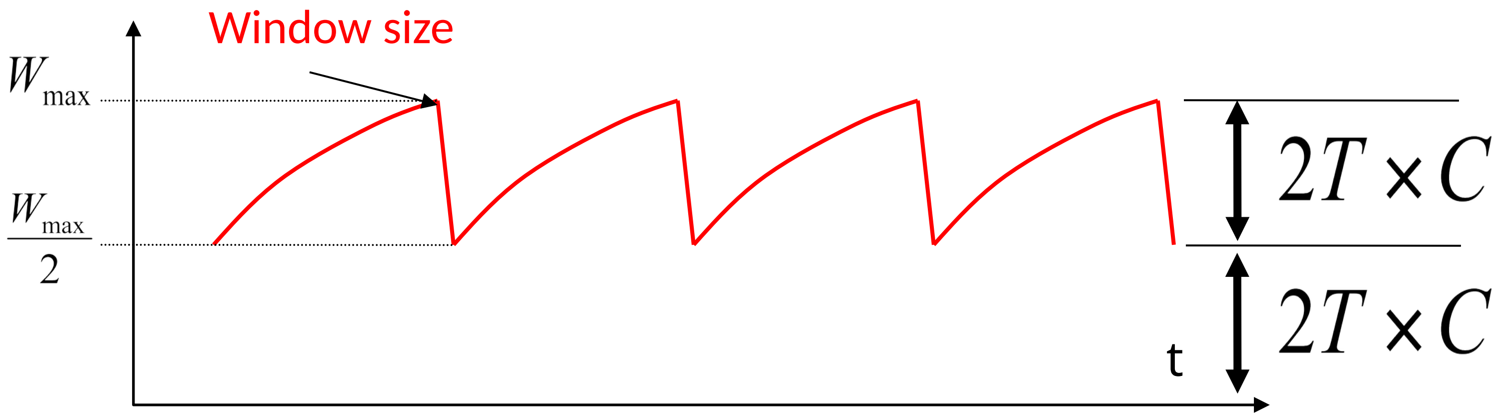


Rule-of-thumb – Intuition

Rule for adjusting W

- If an ACK is received: W ← W+1/W
- If a packet is lost: W ← W/2





Buffers in Routers

So how large should the buffers be?

Buffer size matters

- Packet loss
  - Queue overload, and subsequent packet loss
- End-to-end delay
  - Transmission, propagation, and queueing delay
  - The only variable part is queueing delay

Two windowing limits per flow

- Flow control window: AdvertisedWindow (RWND)
  - How many bytes can be sent without overflowing Rx buffers,
  - Determined by Rx and reported to Tx.
- Congestion Window: CWND
  - How many bytes can be sent without overflowing routers.
  - Computed by Tx using congestion control algorithm.

- Sender-side window = MIN(CWND,RWND)
  - Assume for this material that RWND >> CWND

Note

- This lecture will talk about CWND in units of MSS
  - (Recall MSS: Maximum Segment Size, the amount of payload data in a TCP packet)
  - This is only for pedagogical purposes

- **In reality this is a SIMPLIFICATION:**

  Real implementations maintain CWND in bytes

Two Basic Questions

- How does the sender detect congestion?
- How does the sender adjust its sending rate?
  - To address three issues
    - Finding available bottleneck bandwidth
    - Adjusting to bandwidth variations
    - Sharing bandwidth

(Recall) Detecting Congestion

- Packet delays
  - Tricky: noisy signal (delay often varies considerably)

- Router tell end-hosts they’re congested

- Packet loss
  - Fail-safe signal that TCP already has to detect
  - Complication: non-congestive loss (checksum errors)

- Two indicators of packet loss
  - No ACK after certain time interval: timeout
  - Multiple duplicate ACKs

Not All Losses the Same

- Duplicate ACKs: isolated loss
  - Still getting ACKs
- Timeout: much more serious
  - Not enough packets in progress to trigger duplicate-acks, OR
  - Suffered several losses
- We will adjust rate differently for each case

Rate Adjustment

- Basic structure:
  - Upon receipt of ACK (of new data): increase rate
  - Upon detection of loss: decrease rate

- How we increase/decrease the rate depends on the phase of congestion control we're in:
  - Discovering available bottleneck bandwidth *vs.*
  - Adjusting to bandwidth variations

Bandwidth Discovery with Slow Start

- Goal: estimate available bandwidth
  - start slow (for safety)
  - but ramp up quickly (for efficiency)

- Consider
  - RTT = 100ms, MSS=1000bytes
  - Window size to fill 1Mbps of BW = 12.5 packets
  - Window size to fill 1Gbps = 12,500 packets
  - Either is possible!

"Slow Start" Phase

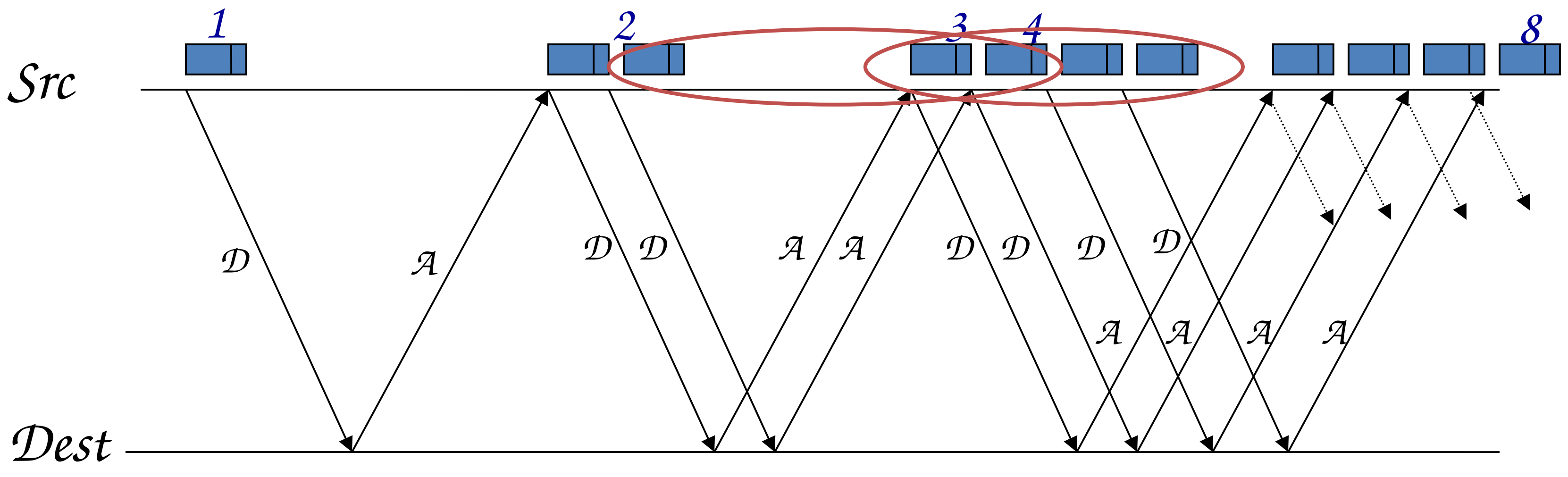
- Sender starts at a slow rate but increases **exponentially** until first loss

- Start with a small congestion window
  - Initially, CWND = 1
  - So, initial sending rate is MSS/RTT

- Double the CWND for each RTT with no loss

Slow Start in Action

- For each RTT: double CWND
- Simpler implementation: for each ACK, CWND += 1



Adjusting to Varying Bandwidth

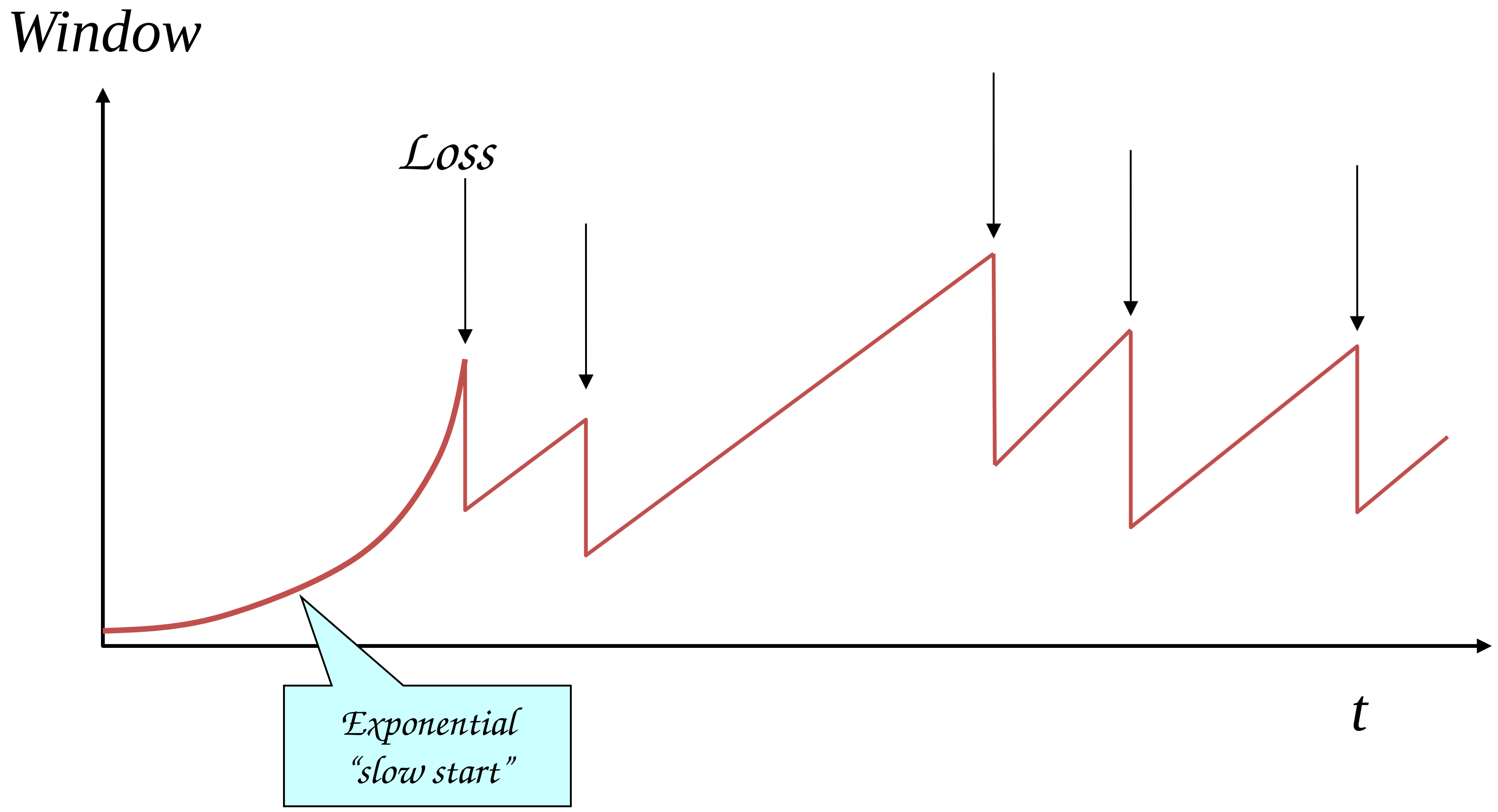
- Slow start gave an estimate of available bandwidth
- Now, want to track variations in this available bandwidth, oscillating around its current value
  - Repeated probing (rate increase) and backoff (rate decrease)
- TCP uses: ‘Additive Increase Multiplicative Decrease’ (AIMD)
  - We’ll see why shortly…

AIMD

- Additive increase
  - Window grows by one MSS for every RTT with no loss
  - For each successful RTT, CWND := CWND + 1

- Multiplicative decrease
  - On loss of packet, divide congestion window in **half**
  - On loss, CWND := CWND/2

Leads to the TCP "Sawtooth"



Slow-Start vs. AIMD

- When does a sender stop Slow-Start and start Additive Increase?
- Introduce a ‘slow-start threshold’ (ssthresh)
  - Initialized to a large value
  - On timeout, ssthresh := CWND/2
- When CWND == ssthresh, sender switches from slow-start to AIMD-style increase

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD (slow-start, congestion avoidance)

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD (slow-start, congestion avoidance)

    and Fast-Recovery

One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss

Example (in units of MSS, not bytes)

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet number 101
    - i.e., receiver expecting next packet to have seq. no. 101

- 10 packets [101, 102, 103,…, 110] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

The problem – A timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) : only now can we transmit new packets
- Plus no packets in flight so ACK “clocking” (to increase CWND) stalls for another RTT

Solution: Fast Recovery

Idea: Grant the sender temporary ‘credit’ for each dupACK so as to keep packets in flight

- If dupACKcount = 3
  - ssthresh := cwnd/2
  - cwnd := ssthresh + 3
- While in fast recovery
  - cwnd := cwnd + 1 for each additional duplicate ACK
- Exit fast recovery after receiving new ACK
  - set cwnd := ssthresh

Example

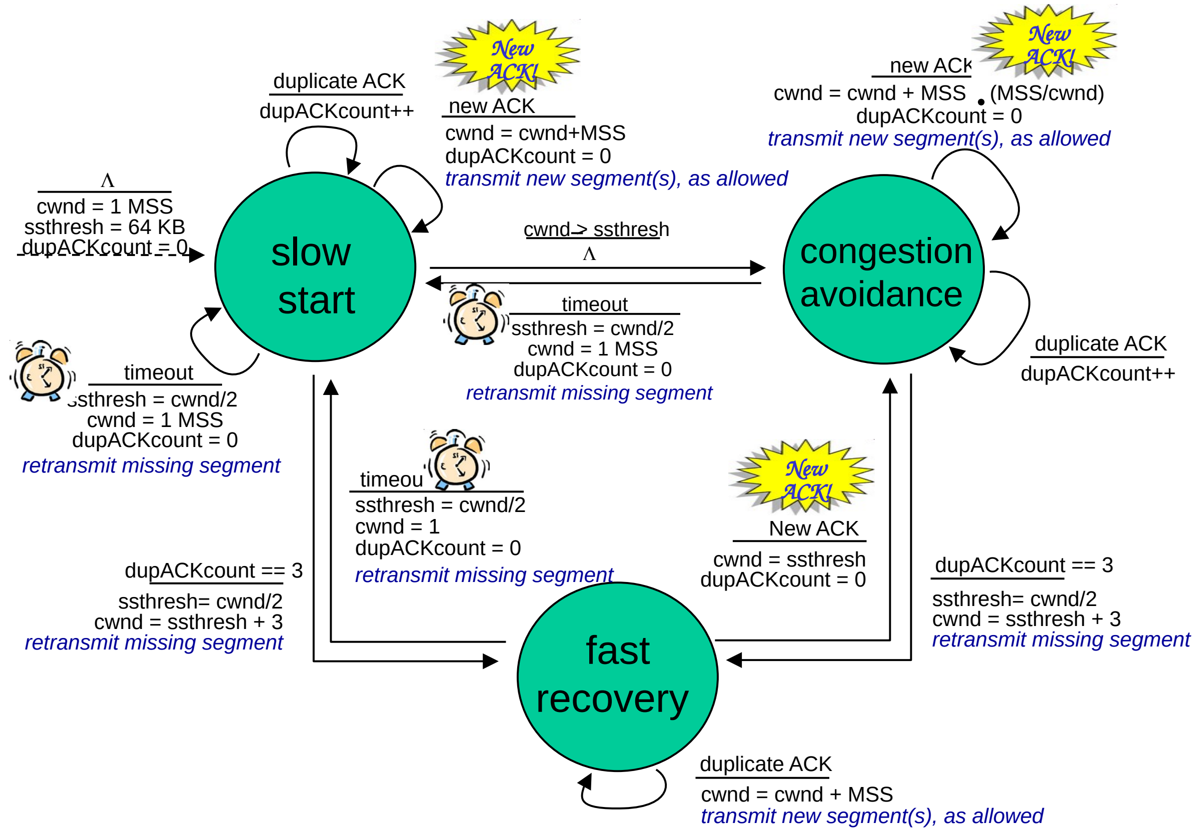
- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101

- 10 packets [101, 102, 103,..., 110] are in flight
  - Packet 101 is dropped

Timeline

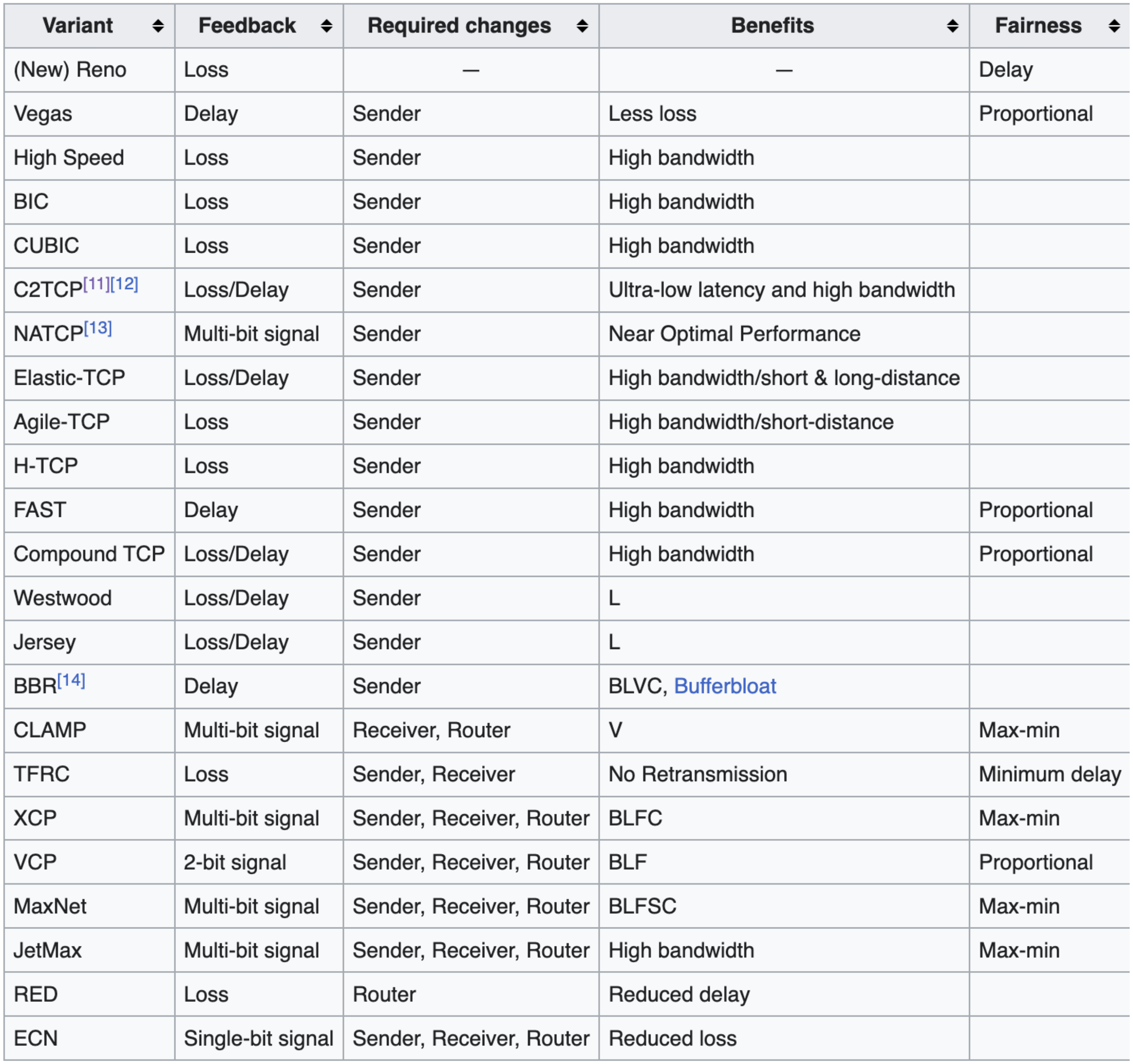
- ACK 101 (due to 102) cwnd=10 dup#1
- ACK 101 (due to 103) cwnd=10 dup#2
- ACK 101 (due to 104) cwnd=10 dup#3
- REXMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 105) cwnd= 9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd = 5 (xmit 115)   exiting fast recovery
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd = 5 + 1/5   back in congestion avoidance

Summary: TCP congestion control



- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD (slow-start, congestion avoidance)
    and Fast-Recovery

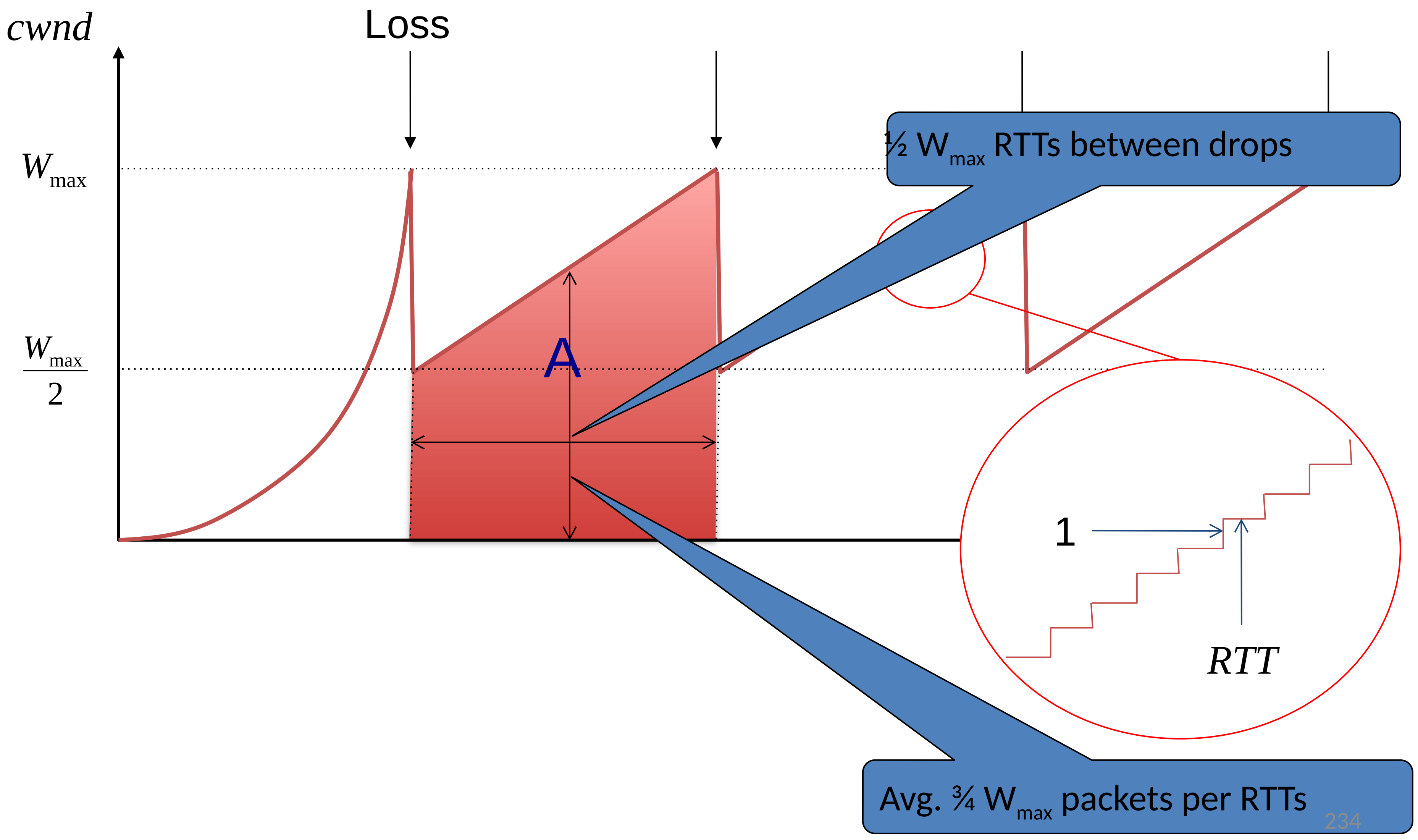
Congestion avoidance algorithm has been a fertile field….



	Variant	Feedback	Required changes	Benefits	Fairness
	(New) Reno	Loss	—	—	Delay
	Vegas	Delay	Sender	Less loss	Proportional
	High Speed	Loss	Sender	High bandwidth	
	BIC	Loss	Sender	High bandwidth	
	CUBIC	Loss	Sender	High bandwidth	
●	C2TCP[11][12]	Loss/Delay	Sender	Ultra-low latency and high bandwidth	
	NATCP[13]	Multi-bit signal	Sender	Near Optimal Performance	
	Elastic-TCP	Loss/Delay	Sender	High bandwidth/short & long-distance	
	Agile-TCP	Loss	Sender	High bandwidth/short-distance	
●	H-TCP	Loss	Sender	High bandwidth	
	FAST	Delay	Sender	High bandwidth	Proportional
	Compound TCP	Loss/Delay	Sender	High bandwidth	Proportional
	Westwood	Loss/Delay	Sender	L	
	Jersey	Loss/Delay	Sender	L	
●	BBR[14]	Delay	Sender	BLVC, Bufferbloat	
	CLAMP	Multi-bit signal	Receiver, Router	V	Max-min
	TFRC	Loss	Sender, Receiver	No Retransmission	Minimum delay
●	XCP	Multi-bit signal	Sender, Receiver, Router	BLFC	Max-min
	VCP	2-bit signal	Sender, Receiver, Router	BLF	Proportional
	MaxNet	Multi-bit signal	Sender, Receiver, Router	BLFSC	Max-min
	JetMax	Multi-bit signal	Sender, Receiver, Router	High bandwidth	Max-min
	RED	Loss	Router	Reduced delay	
	ECN	Single-bit signal	Sender, Receiver, Router	Reduced loss	

TCP Throughput Equation

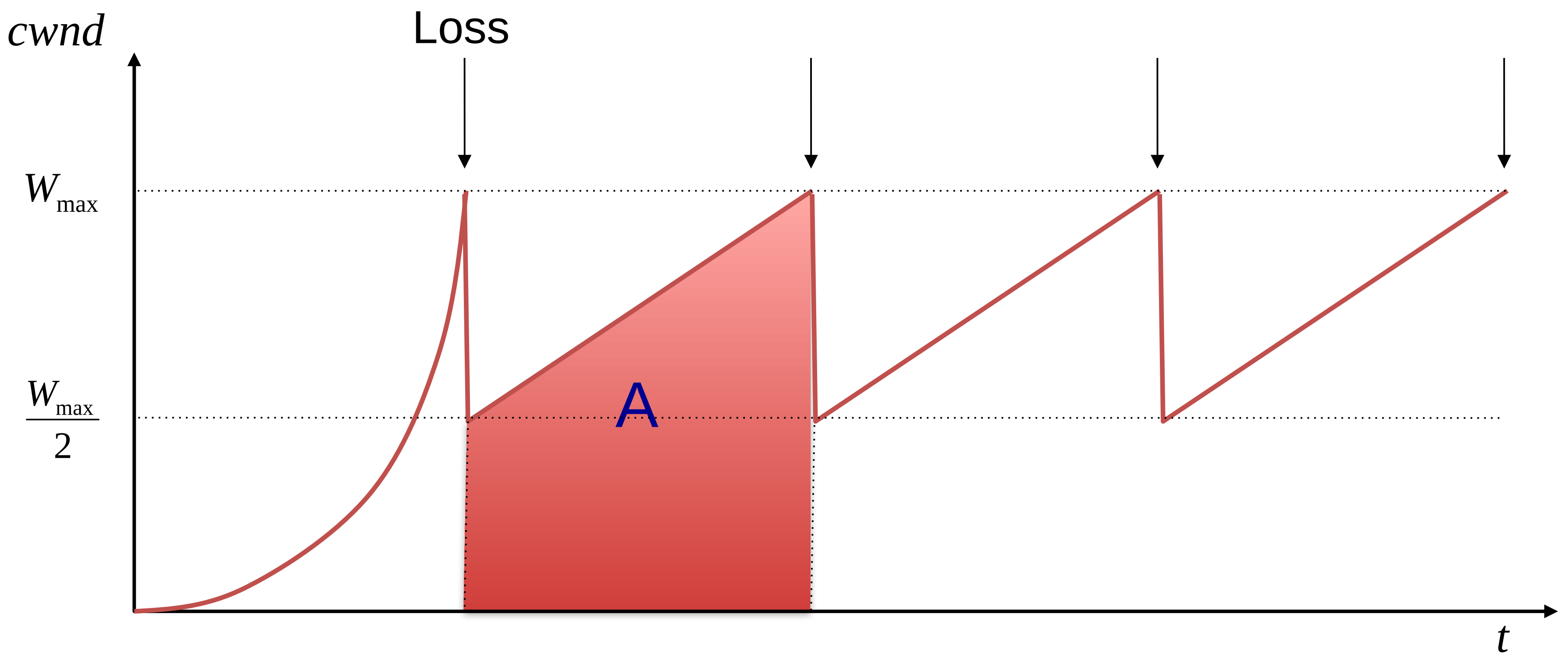
A Simple Model for TCP Throughput



½ $W_{\max}$ RTTs between drops

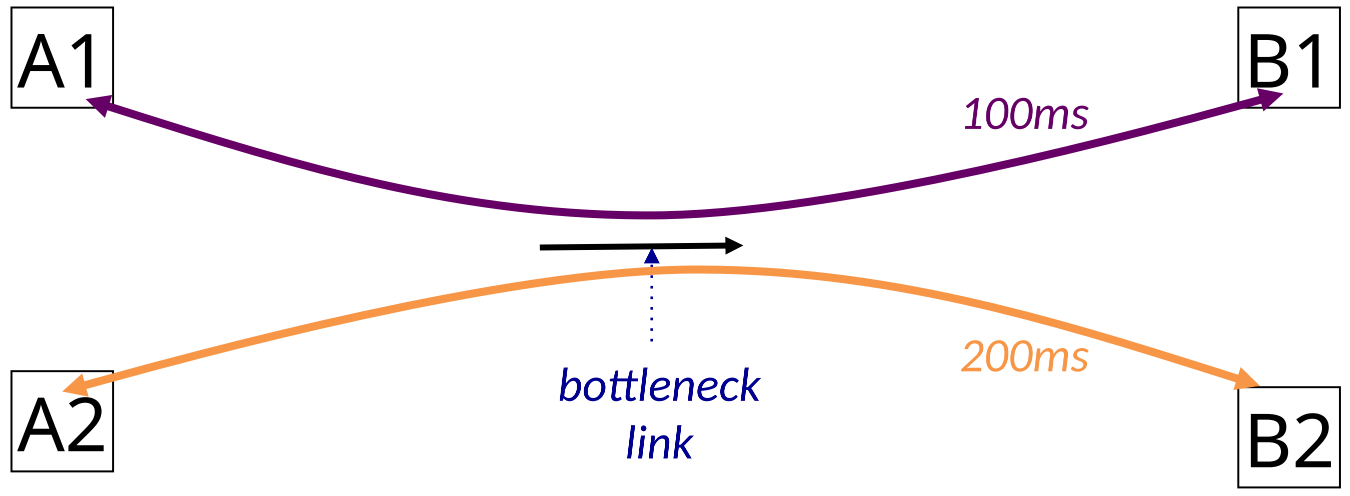
Avg. ¾ $W_{\max}$ packets per RTTs

A Simple Model for TCP Throughput



Implications (1): Different RTTs

- Flows get throughput inversely proportional to RTT
- TCP unfair in the face of heterogeneous RTTs!



Implications (2): High Speed TCP

- Assume RTT = 100ms, MSS=1500bytes
- What value of *p* is required to reach 100Gbps throughput
  - ~ 2 x 10^{-12}
- How long between drops?
  - ~ 16.6 hours
- How much data has been sent in this time?
  - ~ 6 petabits
- These are not practical numbers!

Adapting TCP to High Speed

- Once past a threshold speed, increase CWND faster
  - A proposed standard [Floyd'03]: once speed is past some threshold, change equation to $p^{-.8}$ rather than $p^{-.5}$
  - Let the additive constant in AIMD depend on CWND

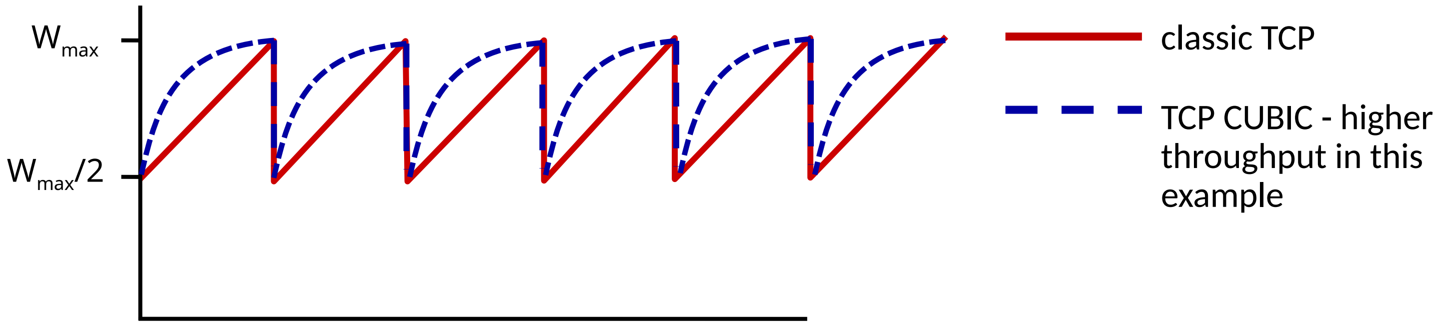
- Other approaches?
  - Multiple simultaneous connections (*hacky* but works today)
  - Router-assisted approaches (will see shortly)

Implications (3): *Rate*-based CC

- TCP throughput is "choppy"
  - repeated swings between W/2 to W
- Some apps would prefer sending at a steady rate
  - e.g., streaming apps
- A solution: "Equation-Based Congestion Control"
  - ditch TCP's increase/decrease rules and just follow the equation
  - measure drop percentage *p*, and set rate accordingly
- Following the TCP equation ensures we're "TCP friendly"
  - i.e., use no more than TCP does in similar setting

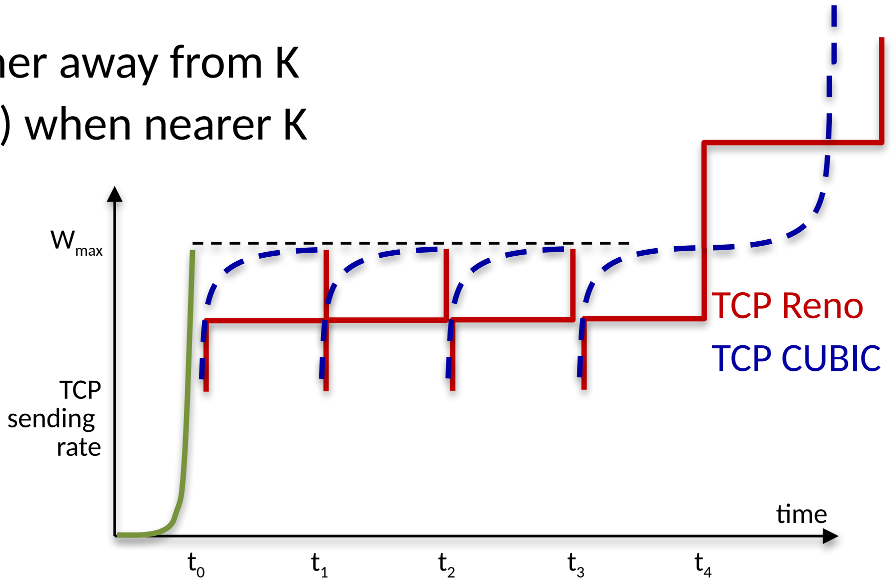
TCP CUBIC

- 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*



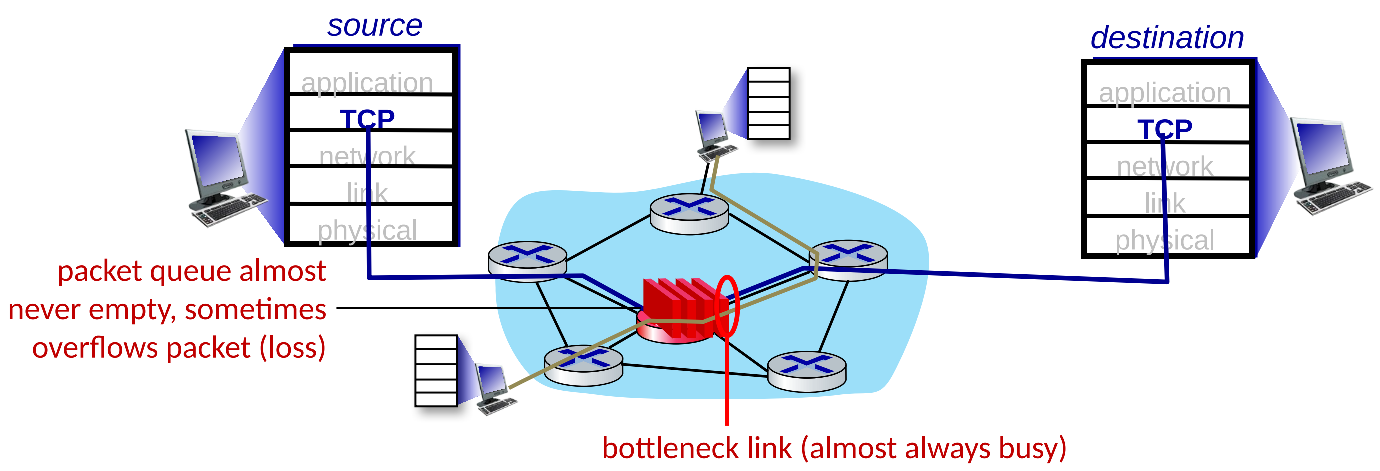
TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- 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



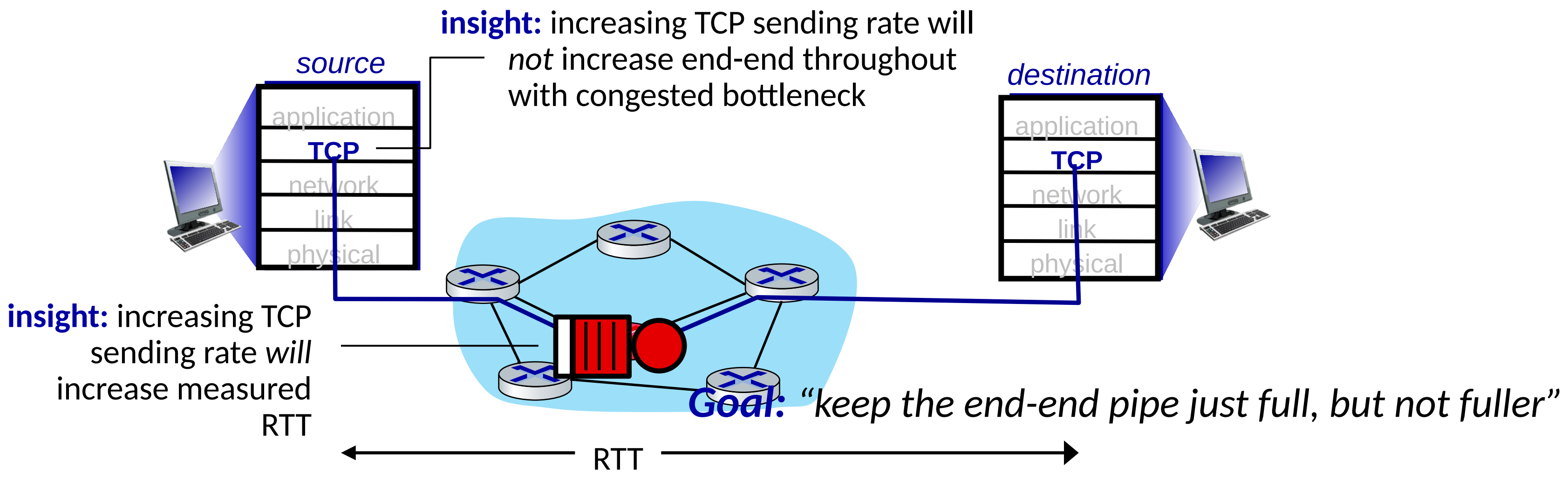
TCP and the congested “bottleneck link”

- TCP (classic, CUBIC) increase TCP’s sending rate until packet loss occurs at some router’s output: the *bottleneck link*


source
application
TCP
network
link
physical
destination
application
TCP
network
link
physical
packet queue almost never empty, sometimes overflows packet (loss)
bottleneck link (almost always busy)

TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



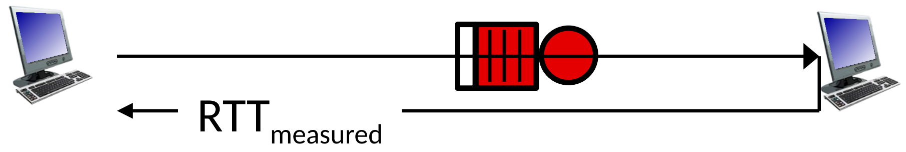
insight: increasing TCP sending rate will *not* increase end-end throughout with congested bottleneck

insight: increasing TCP sending rate *will* increase measured RTT

Goal: *"keep the end-end pipe just full, but not fuller"*

Delay-based TCP Congestion Control

Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{\text{RTT}_{\text{measured}}}$$

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window `cwnd` is $\text{cwnd}/\text{RTT}_{\text{min}}$

```
if measured throughput "very close" to  uncongested throughput
    increase cwnd linearly              /* since path not congested */
else if measured throughput "far below" uncongested throughout
    decrease cwnd  linearly             /* since path is congested */
```

Delay-based TCP Congestion Control

- congestion control without inducing/forcing loss
- maximizing throughout ("keeping the just pipe full... ") while keeping delay low ("…but not fuller")
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network

Recap: TCP problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- AIMD impractical for high speed links
- Sawtooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Routers tell endpoints if they're congested

Routers tell endpoints what rate to send at

Routers enforce fair sharing

Could fix many of these with some help from routers!

Router-Assisted Congestion Control

- Three tasks for CC:
  - Isolation/fairness
  - Adjustment*
  - Detecting congestion

* This may be *automatic* eg loss-response of TCP

How can routers ensure each flow gets its “fair share”?

Fairness: General Approach

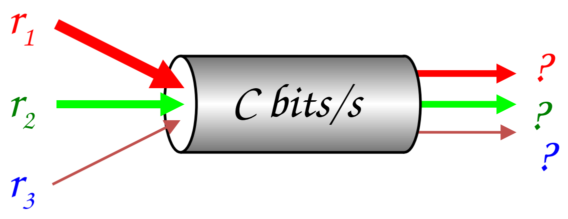
- Routers classify packets into "flows"
  - (For now) flows are packets between same source/destination
- Each flow has its own FIFO queue in router
- Router services flows in a fair fashion
  - When line becomes free, take packet from next flow in a fair order
- What does "fair" mean exactly?

Max-Min Fairness

- Given set of bandwidth demands r_i and total bandwidth C, max-min bandwidth allocations are:

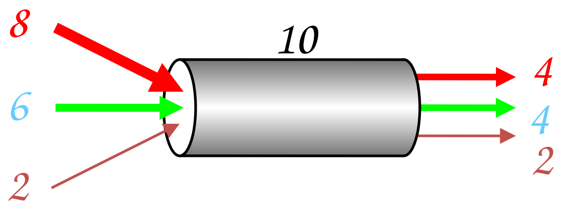
$$a_i = \min(f, r_i)$$

where f is the unique value such that Sum(a_i) = C



Example

- $C = 10; \quad r_1 = 8, r_2 = 6, r_3 = 2; \quad N = 3$
- $C/3 = 3.33 \rightarrow$
  - Can service all of r_3
  - Remove r_3 from the accounting: $C = C - r_3 = 8$; $N = 2$
- $C/2 = 4 \rightarrow$
  - Can't service all of r_1 or r_2
  - So hold them to the remaining fair share: $f = 4$



$f = 4$:
min(8, 4) = 4
min(6, 4) = 4
min(2, 4) = 2

Max-Min Fairness

- Given set of bandwidth demands r_i and total bandwidth C, max-min bandwidth allocations are:

$$a_i = \min(f, r_i)$$

- where f is the unique value such that $\mathrm{Sum}(a_i) = C$

- Property:
  - If you don't get full demand, no one gets more than you

- This is what round-robin service gives if all packets are the same size

How do we deal with packets of different sizes?

- Mental model: Bit-by-bit round robin (“fluid flow”)

- Can you do this in practice?

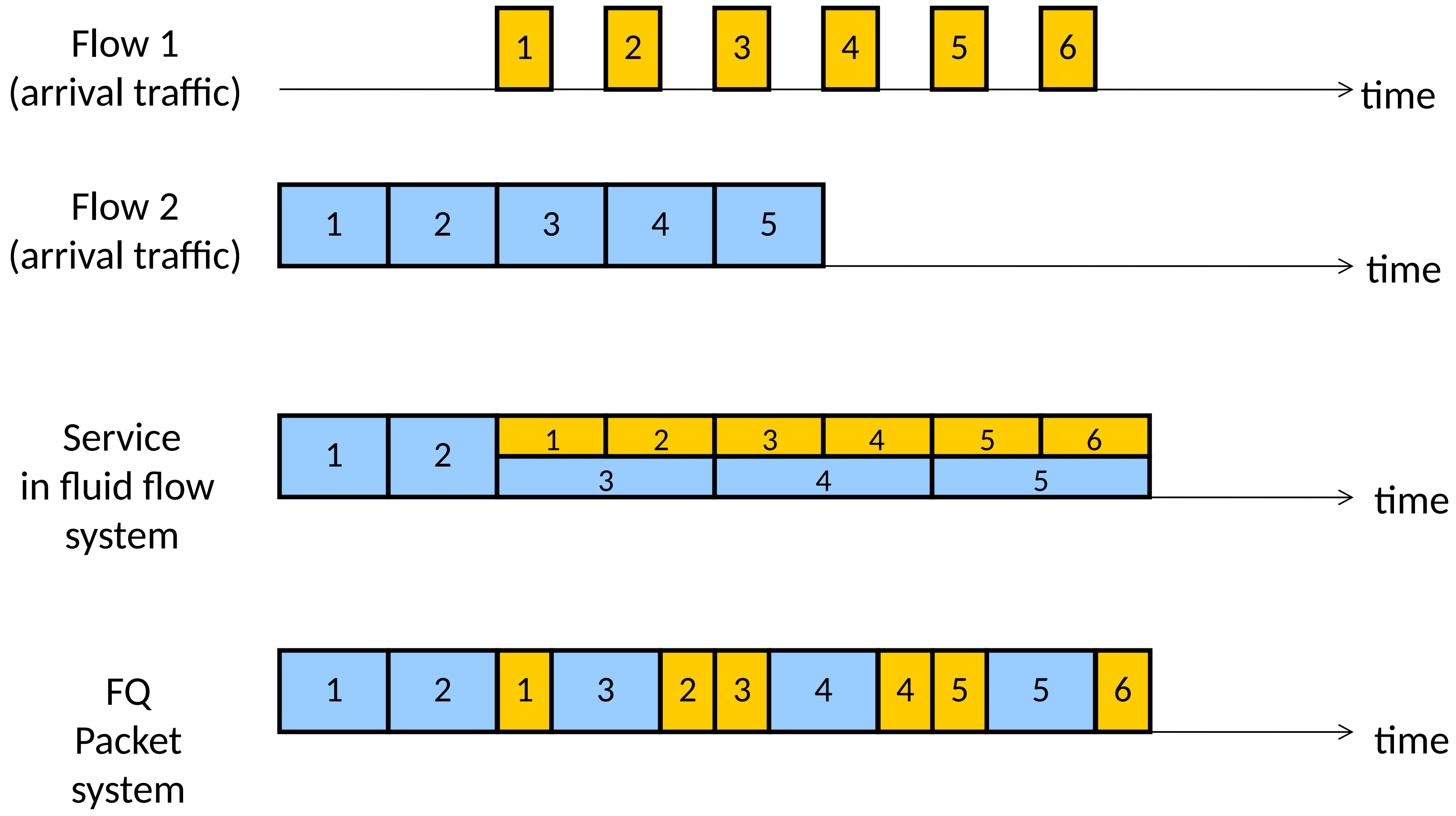
- No, packets cannot be preempted

- But we can approximate it
  - This is what “fair queuing” routers do

Fair Queuing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left the router *if* flows are served bit-by-bit
- Then serve packets in the increasing order of their deadlines

Example


Flow 1
(arrival traffic)
1
2
3
4
5
6
time
Flow 2
(arrival traffic)
1
2
3
4
5
time
Service
in fluid flow
system
1
2
1
2
3
4
5
6
3
4
5
time
FQ
Packet
system
1
2
1
3
2
3
4
4
5
5
6
time

Fair Queuing (FQ)

- Think of it as an implementation of round-robin generalized to the case where not all packets are equal sized

- Weighted fair queuing (WFQ): assign different flows different shares

- Today, some form of WFQ implemented in almost all routers
  - Not the case in the 1980-90s, when CC was being developed
  - Mostly used to isolate traffic at larger granularities (e.g., per-prefix)

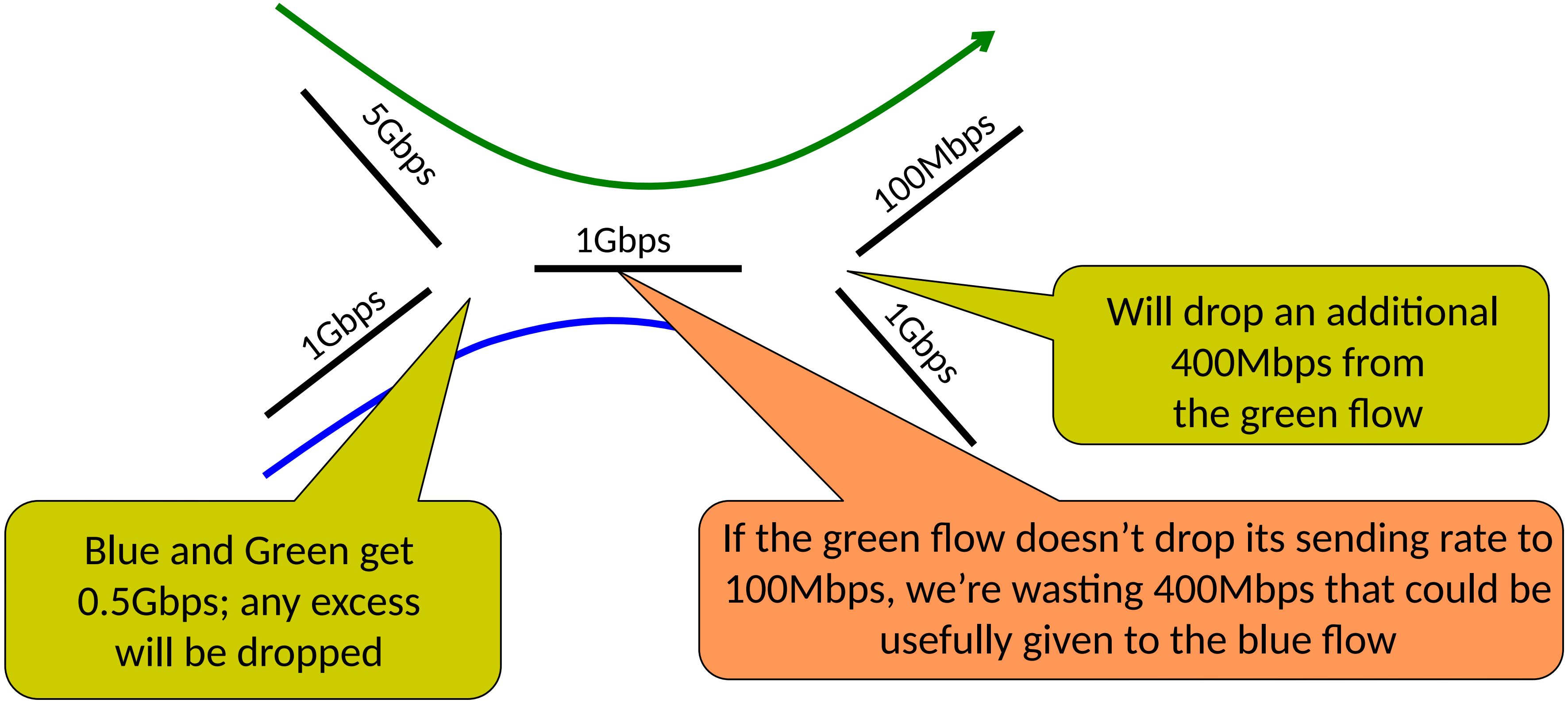
FQ vs. FIFO

- FQ advantages:
  - Isolation: cheating flows don't benefit
  - Bandwidth share does not depend on RTT
  - Flows can pick any rate adjustment scheme they want

- Disadvantages:
  - More complex than FIFO: per flow queue/state, additional per-packet book-keeping

FQ in the big picture

- FQ does not eliminate congestion it just manages the congestion


5Gbps
100Mbps
1Gbps
1Gbps
1Gbps
Will drop an additional 400Mbps from the green flow
Blue and Green get 0.5Gbps; any excess will be dropped
If the green flow doesn't drop its sending rate to 100Mbps, we're wasting 400Mbps that could be usefully given to the blue flow

FQ in the big picture

- FQ does not eliminate congestion it just manages the congestion
  - robust to cheating, variations in RTT, details of delay, reordering, retransmission, *etc*.

- But congestion (and packet drops) still occurs

- And we still want end-hosts to discover/adapt to their fair share!

- What would the end-to-end argument say w.r.t. congestion control?

Fairness is a controversial goal

- What if you have 8 flows, and I have 4?
  - Why should you get twice the bandwidth
- What if your flow goes over 4 congested hops, and mine only goes over 1?
  - Why shouldn't you be penalized for using more scarce bandwidth?
- And what is a flow anyway?
  - TCP connection
  - Source-Destination pair?
  - Source?

Explicit Congestion Notification (ECN)

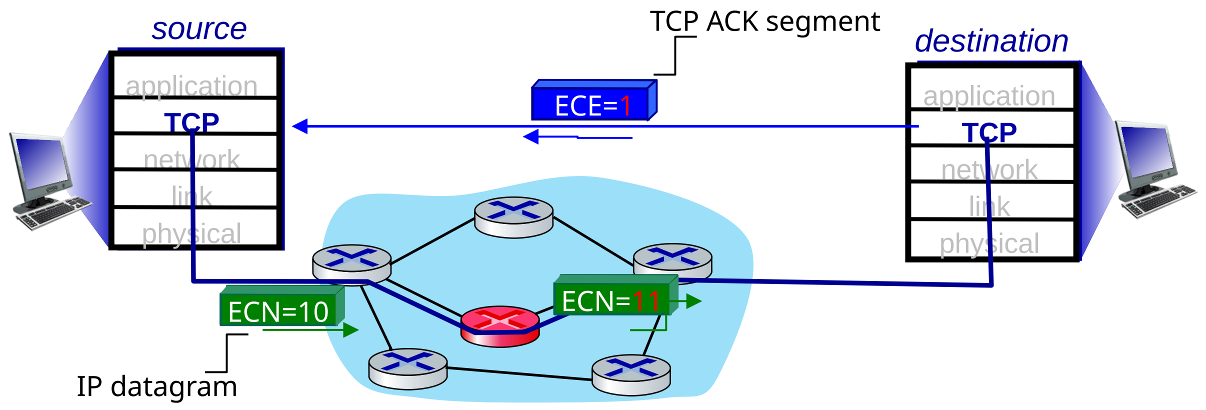
- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for when routers set the bit
  - tradeoff between (link) utilization and (packet) delay
- Congestion semantics can be exactly like that of drop
  - I.e., endhost reacts as though it saw a drop

- Advantages:
  - Doesn't confuse corruption with congestion; recovery w/ rate adjustment
  - Can serve as an early indicator of congestion to avoid delays
  - Easy (easier) to incrementally deploy
    - defined as extension to TCP/IP in RFC 3168 (uses diffserv bits in the IP header)

Explicit congestion notification (ECN)

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)



Securing TCP

Vanilla TCP & UDP sockets:

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

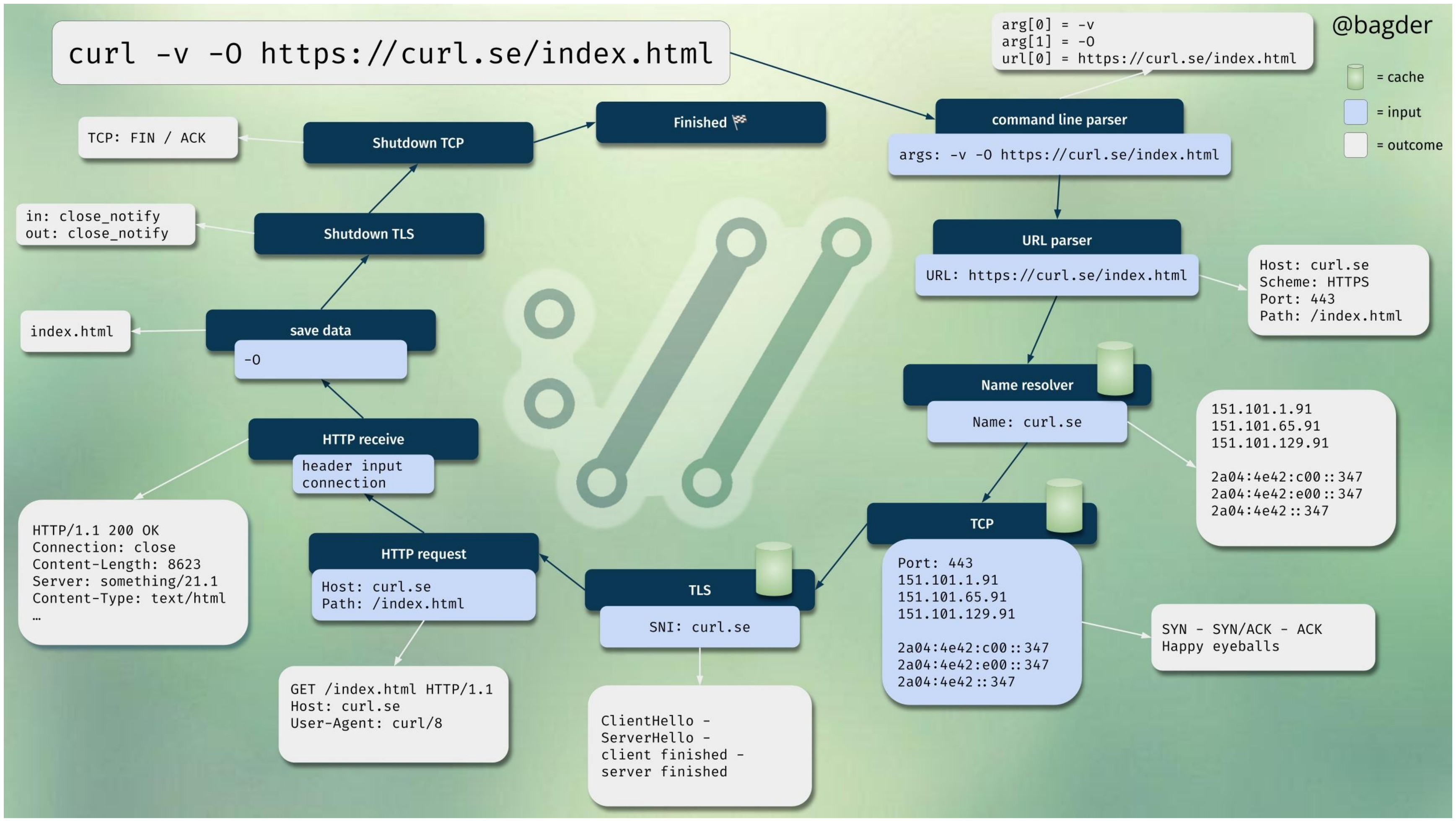
TLS implemented in application layer

- apps use TLS libraries, that use TCP in turn
- cleartext sent into "socket" traverse Internet *encrypted*

SSL vs. TLS

Simple: SSL is deprecated

TLS refers to secure socket layers in actual use.



```
curl -v -O https://curl.se/index.html
```

```
arg[0] = -v
arg[1] = -O
url[0] = https://curl.se/index.html
```

```
args: -v -O https://curl.se/index.html
```

```
URL: https://curl.se/index.html
```

```
Host: curl.se
Scheme: HTTPS
Port: 443
Path: /index.html
```

```
Name: curl.se
```

```
151.101.1.91
151.101.65.91
151.101.129.91

2a04:4e42:c00::347
2a04:4e42:e00::347
2a04:4e42::347
```

```
Port: 443
151.101.1.91
151.101.65.91
151.101.129.91

2a04:4e42:c00::347
2a04:4e42:e00::347
2a04:4e42::347
```

```
SYN - SYN/ACK - ACK
Happy eyeballs
```

```
SNI: curl.se
```

```
ClientHello -
ServerHello -
client finished -
server finished
```

```
Host: curl.se
Path: /index.html
```

```
GET /index.html HTTP/1.1
Host: curl.se
User-Agent: curl/8
```

```
header input
connection
```

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 8623
Server: something/21.1
Content-Type: text/html
…
```

```
-O
```

```
index.html
```

```
in: close_notify
out: close_notify
```

```
TCP: FIN / ACK
```

Transport Recap

A "*big bag*":

Multiplexing, reliability, error-detection, error-recovery,
flow and congestion control, ….

- UDP:
  - Minimalist - multiplexing and error detection

- TCP:
  - somewhat hacky
  - but practical/deployable
  - good enough to have raised the bar for the deployment of new approaches
  - though the needs of datacenters change the status quos

- Beyond TCP (discussed in Topic 6):
  - QUIC / application-aware transport layers