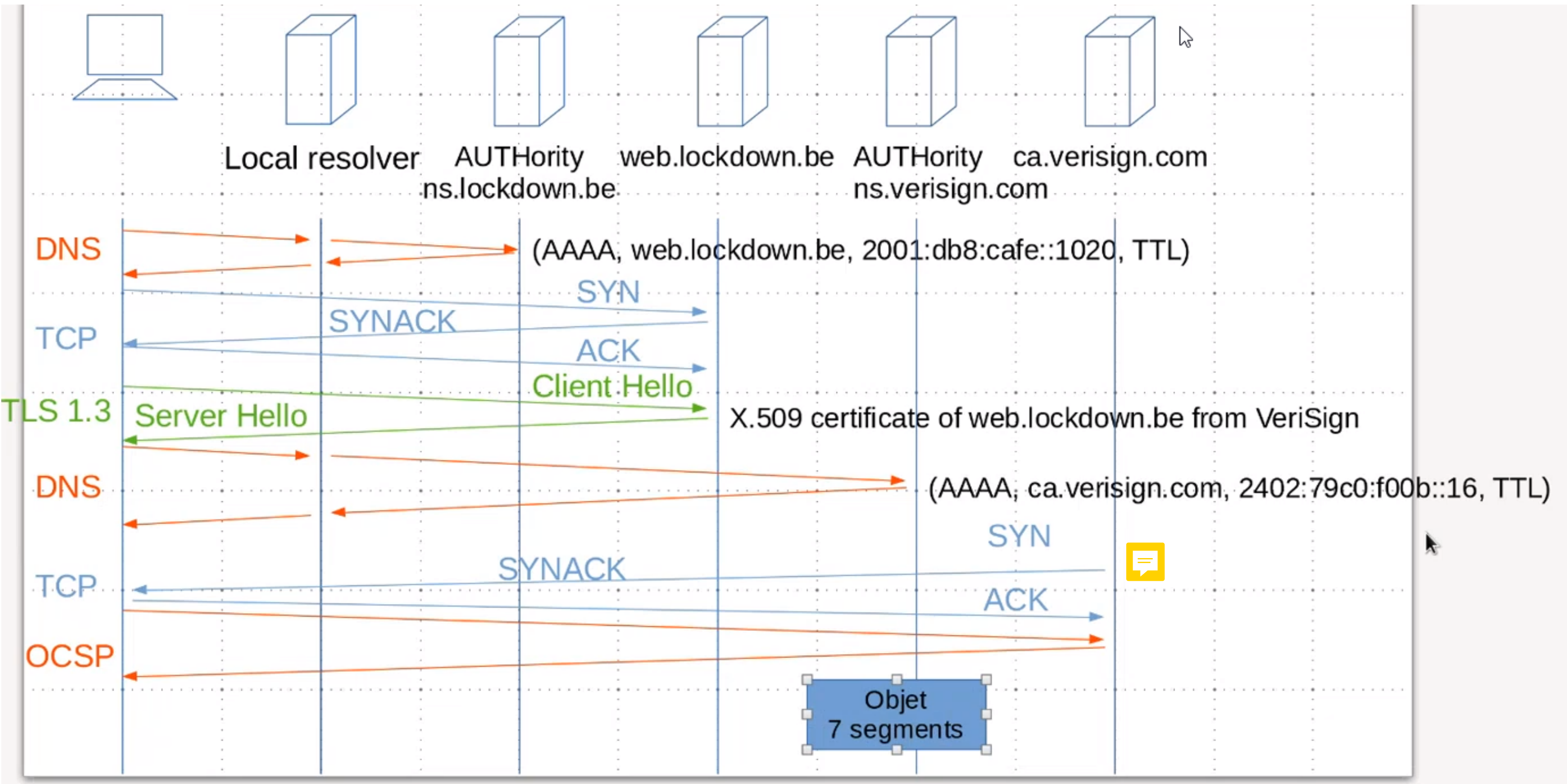
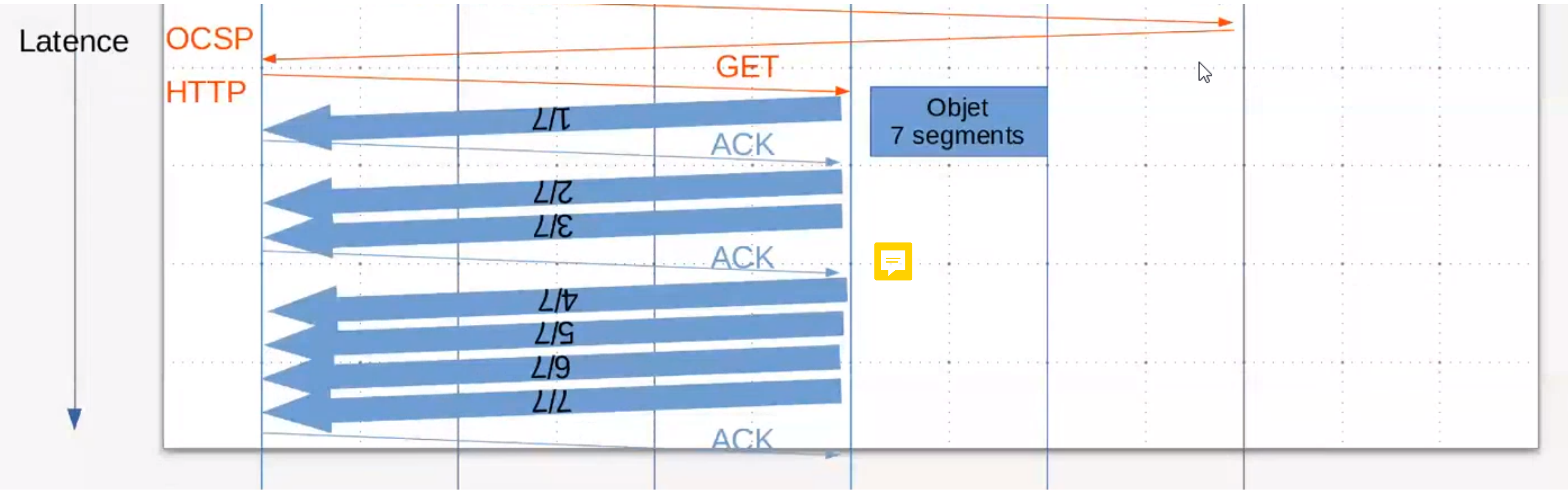
Chapitre 4
Transport Layer

Laurent Schumacher (UNamur)
Dernière mise-à-jour : 20 octobre 2020

UNIVERSITÉ DE NAMUR

Synthèse d'une requête HTTPS vers un site web



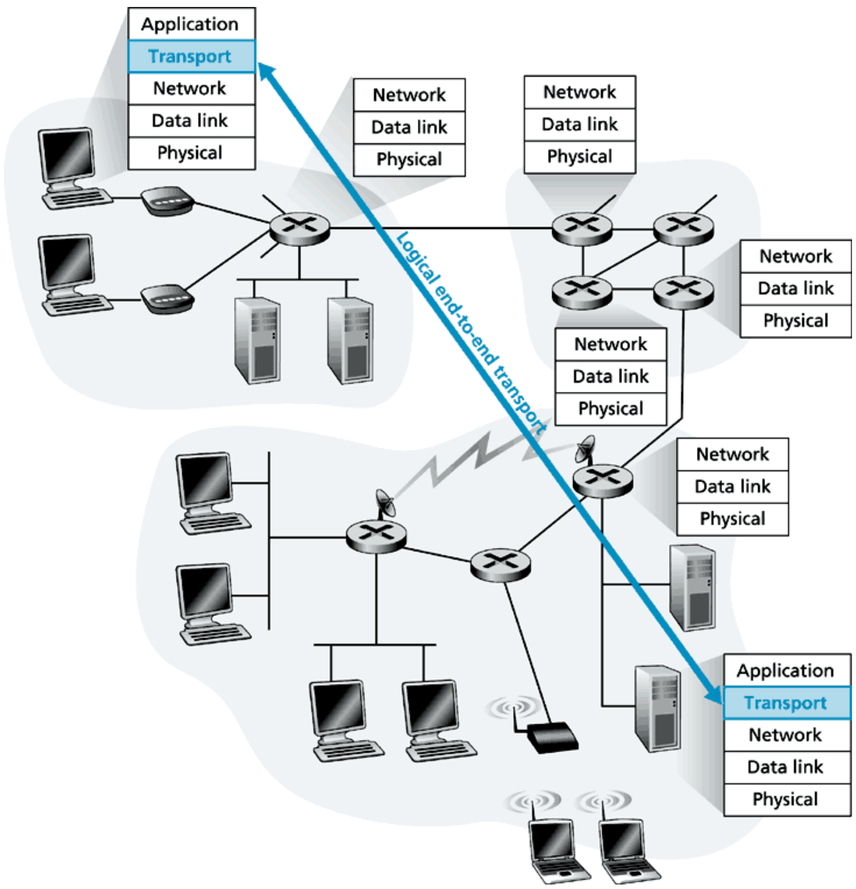


Latence
OCSP
HTTP
GET
Objet
7 segments
1/7
ACK
2/7
3/7
ACK
4/7
5/7
6/7
7/7
ACK

Outline

- Transport-Layer services
- Multiplexing and demultiplexing
- Connectionless transport – UDP
- Principles of reliable data transfer
- Connection-oriented transport – TCP
  - Segment structure
  - Connection management
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control
- Quick UDP Internet Connections – QUIC

Transport Layer Services
Introduction



- Transport-layer protocol provides logical communication between processes on remote end systems
- Hosts look like directly connected
- Sender: breaks application messages into segments passed to network layer
- Receiver: reassembles segments into messages and passes them to application layer

Transport Layer Services
Transport vs. Network Layer

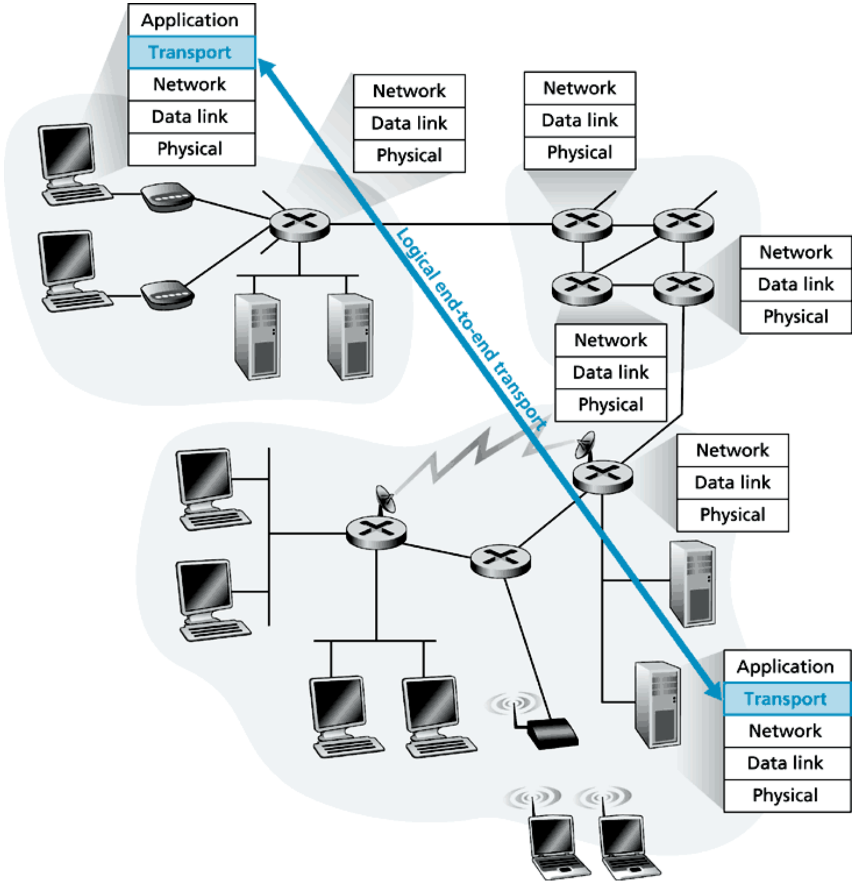
Transport layer relies on and enhances services offered by Network layer

Transport Layer Logical communication between processes
Network Layer Logical communication between hosts

School analogy

Kids	**Processes**
Schools	**Hosts**
Letters	Messages
Alice and Bob	Transport Layer
Postal service	Network Layer

Transport Layer Services
Internet Transport Protocols



- Reliable, in-order delivery (TCP)
  - Connection set-up
  - Flow control
  - Congestion control
- Unreliable, unordered delivery (UDP)
  - No-frills extension of "best-effort" IP
- Services not available
  - Delay guarantees
  - Bandwidth guarantees

Transport Layer Services
Service Description

MPTCP
Multiplexing

TCP
- Reliable delivery service between **processes**
- Connection-oriented
- Correctly and in order
- Flow and congestion control

SCTP
- Reliable delivery service between **processes**
- Connection-oriented
- Correctly, possibly in order
- Flow and congestion control

DCCP
- Unreliable delivery service between **processes**
- Connection-oriented
- Error checking
- Congestion control

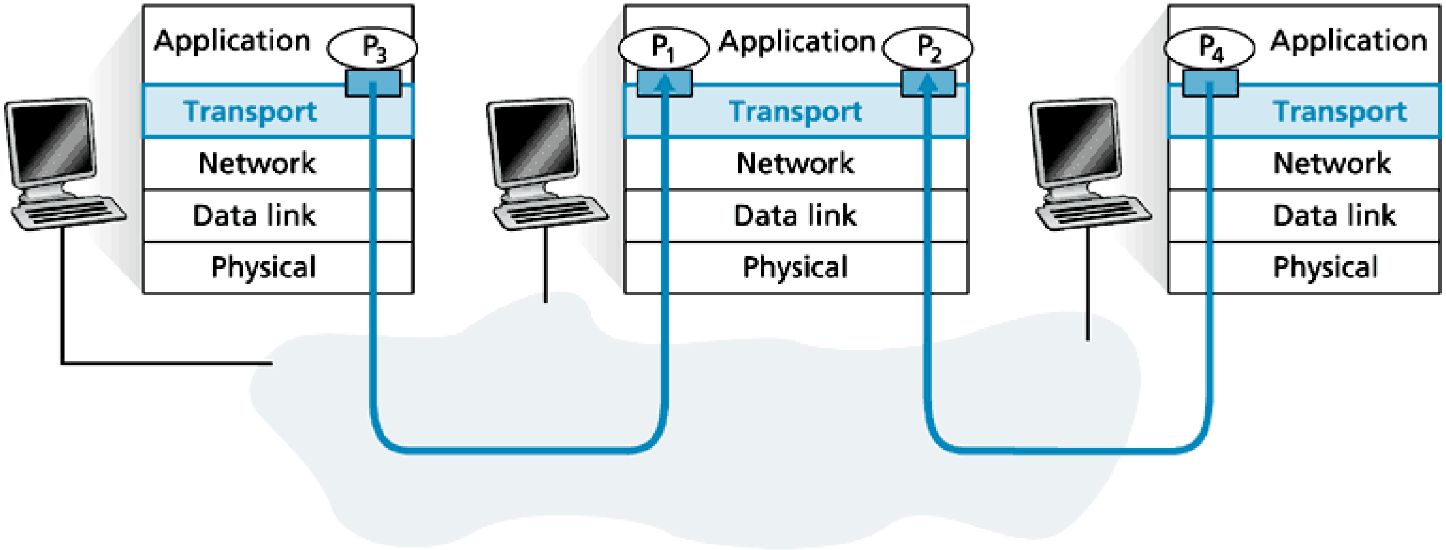
UDP
- Unreliable delivery service between **processes**
- Connectionless
- Error checking

IP
Best-effort, unreliable delivery service between **hosts**
No guarantee (orderly) delivery, no guarantee integrity

Outline

- Transport-Layer services
- Multiplexing and demultiplexing
- Connectionless transport – UDP
- Principles of reliable data transfer
- Connection-oriented transport – TCP
  - Segment structure
  - Connection management
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control
- Quick UDP Internet Connections – QUIC

Multiplexing and demultiplexing

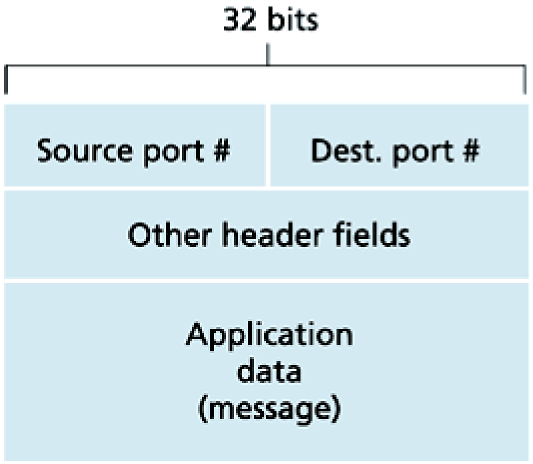


- Multiplexing at sending host: gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)
- School analogy : gathering letters before sending

- Demultiplexing at receiving host : unwrapping data from header, delivering received segments to relevant socket
- School analogy : distributing received letters

Multiplexing and demultiplexing
Identification

TCP/UDP segment format



- Host receives IP datagrams
  - Each datagram has (source IP address, destination IP address)
  - Each datagram carries one transport-layer segment
  - Each segment has (source port number, destination port number)
- Host uses IP addresses and port numbers to direct segment to appropriate socket

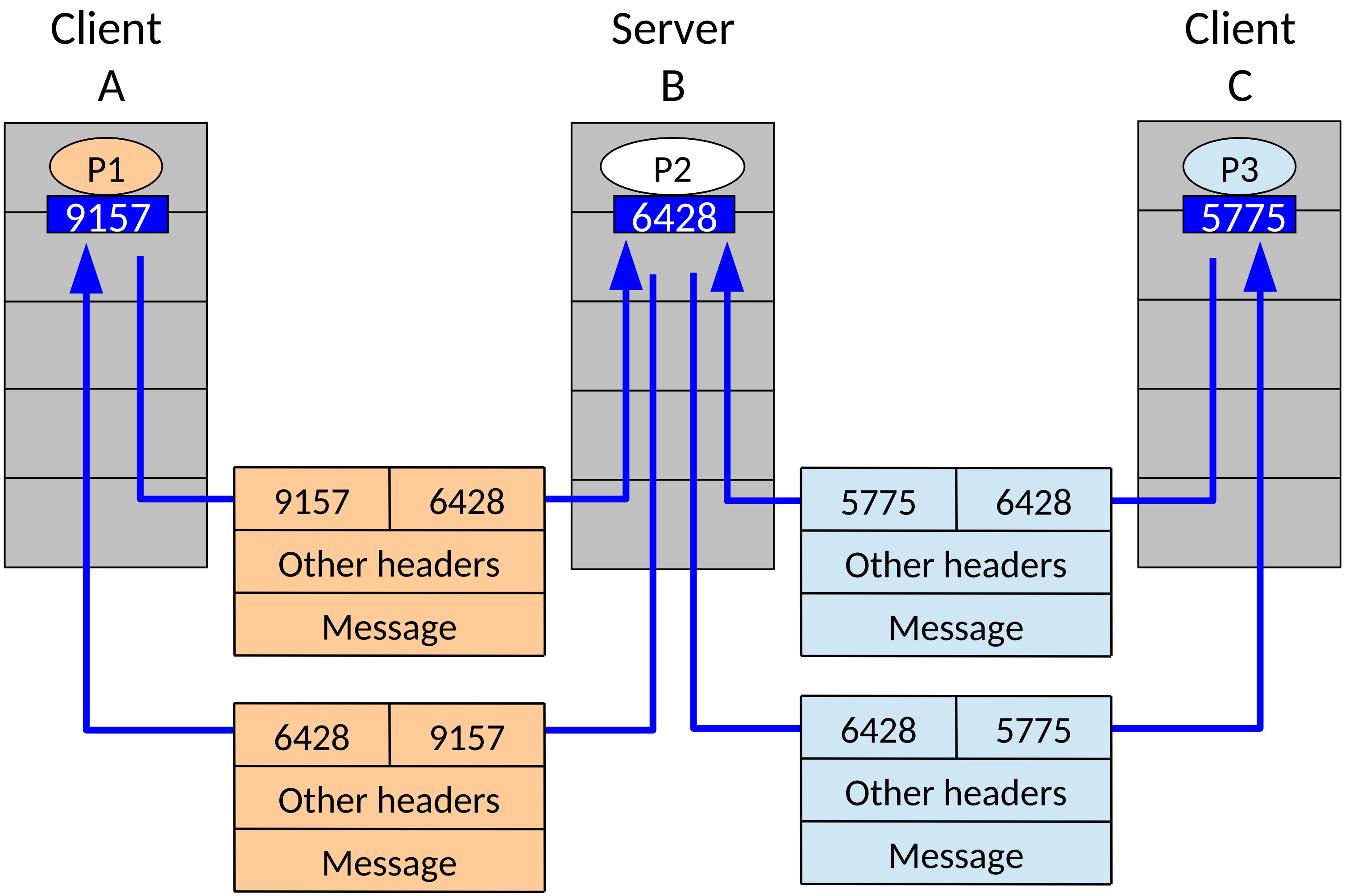
Multiplexing and demultiplexing
Connectionless MUX/DEMUX

- Create sockets with port number

```
DatagramSocket mySocket = new DatagramSocket(19157);
```

- UDP socket identified by 2-tuple: (destination IP address, destination port number)
- Source IP address and source port number used for return address
- When host receives UDP segment
  - Checks destination port number in segment
  - Directs segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

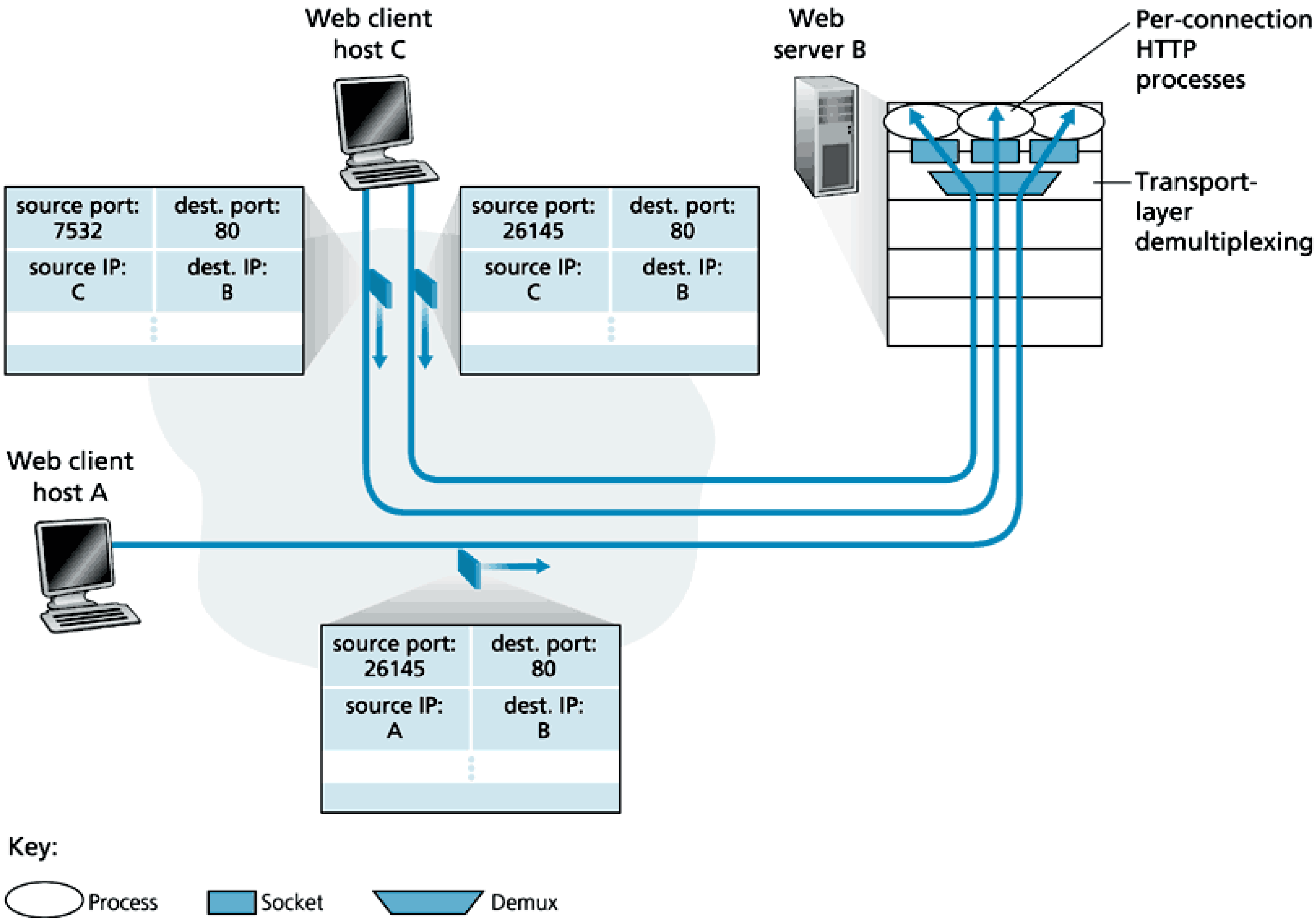
Multiplexing and demultiplexing
Connectionless MUX/DEMUX



Multiplexing and demultiplexing
Connection-oriented MUX/DEMUX

- TCP socket identified by 4-tuple
  1. Source IP address
  2. Source port number
  3. Destination IP address
  4. Destination port number
- Connection identifier
  - Each socket identified by its own 4-tuple
  - Receiving host uses all four values to direct segment to appropriate socket
- Server may support many simultaneous TCP sockets
- One socket per process or thread

Multiplexing and demultiplexing
Connection-oriented MUX/DEMUX



Outline

- Transport-Layer services
- Multiplexing and demultiplexing
- Connectionless transport – UDP
- Principles of reliable data transfer
- Connection-oriented transport – TCP
  - Segment structure
  - Connection management
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control
- Quick UDP Internet Connections – QUIC

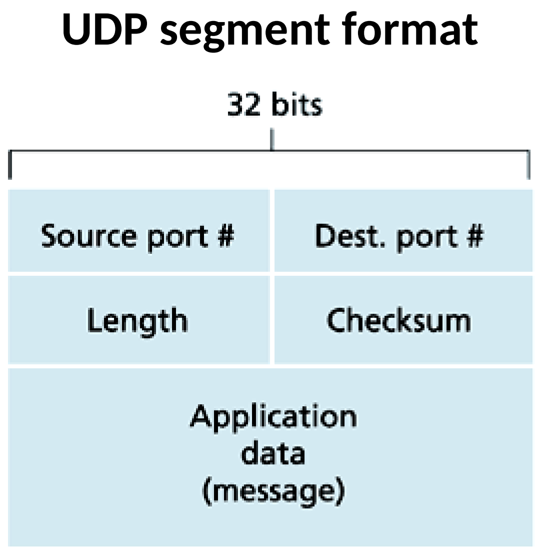
Connection-less transport – UDP
In a nutshell

- “No frills” Internet transport protocol
- “Best effort” service
- UDP segments may be
  - Lost
  - Delivered out of order to application
- Connectionless
  - No handshaking
  - Each UDP segment handled independently of others

Connection-less transport – UDP
In a nutshell

- Pro's
  - No connection establishment → no delay
  - No connection state (buffers, control parameters)
  - Small header overhead (8 instead of 20 with TCP)
  - No congestion control: UDP can blast away as fast as desired
- Con's
  - No congestion control: risk of starvation for TCP
  - Some middleboxes block UDP (2-4% in 2016, source: https://tools.ietf.org/html/rfc8323#ref-EK2016)
- If requested, congestion control implemented in the application

Multiplexing and demultiplexing
Identification



- Often used for streaming multimedia applications
  - Loss tolerant
  - Rate sensitive
- Other UDP uses
  - DNS
  - Simple Network Management Protocol (SNMP)
- Reliable transfer over UDP: add reliability at application layer
- Application-specific error recovery

Multiplexing and demultiplexing
Checksum

- Goal: detect "errors" (e.g. flipped bits) in transmitted segment
- Sender
  - Treat segment contents as sequence of 16-bit integers
  - Checksum: addition (1's complement sum) of segment contents
  - Sender puts checksum value into UDP checksum field
- Receiver
  - Compute checksum of received segment
  - Check if computed checksum equals checksum field value
  - NO - Error detected
  - YES - No error detected. Sure?

Multiplexing and demultiplexing
Checksum example

Sender: three 16-bit words

```
  0110011001100110
  0101010101010101
  0000111100001111
+ ----------------
  1100101011001010

  0011010100110101
  ................
```

1's complement = checksum

Receiver: three 16-bit words + checksum

```
  0110011001100110
  0101010101010101
  0000111100001111
  0011010100110101
+ ----------------
  1111111111111111
```

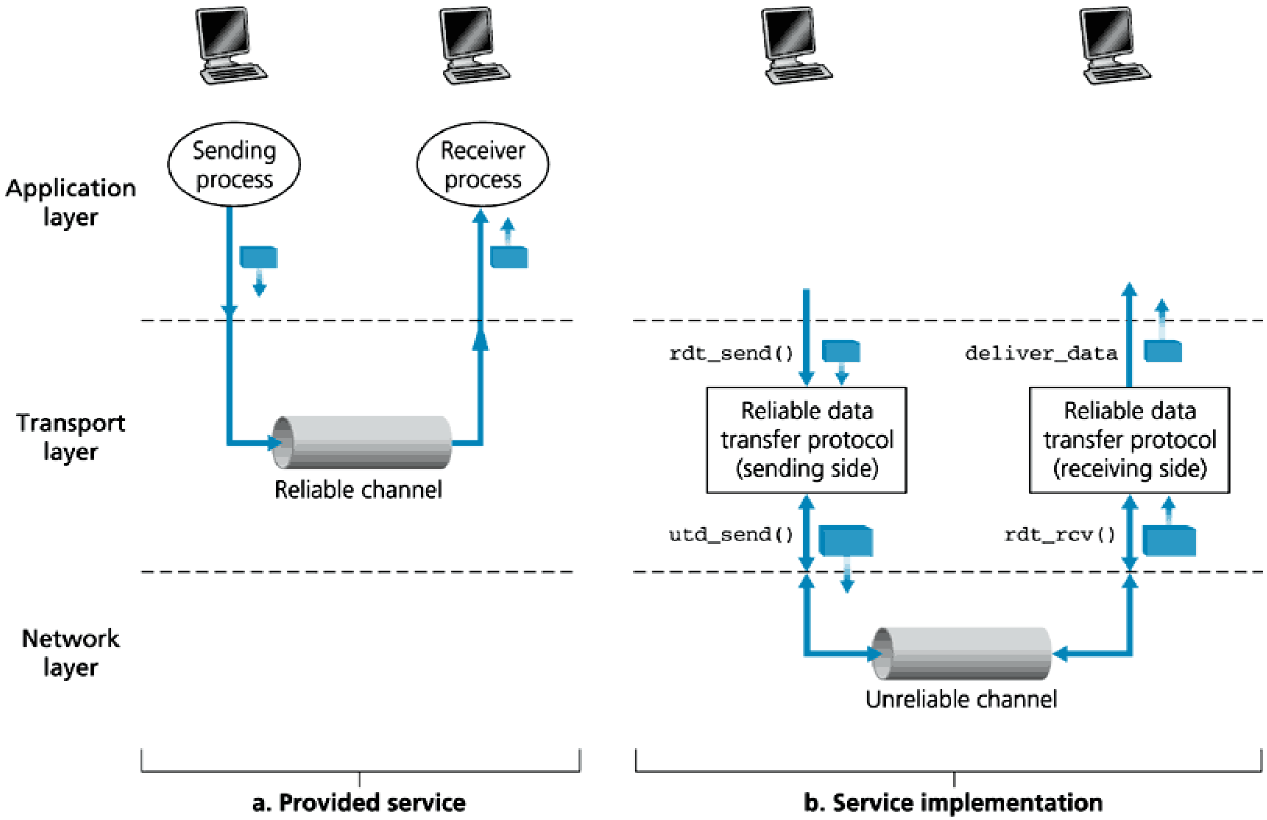
One-bit error detected

Certain two-bit errors undetected

Outline

- Transport-Layer services
- Multiplexing and demultiplexing
- Connectionless transport – UDP
- Principles of reliable data transfer
- Connection-oriented transport – TCP
  - Segment structure
  - Connection management
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control
- Quick UDP Internet Connections – QUIC

Principles of reliable data transfer

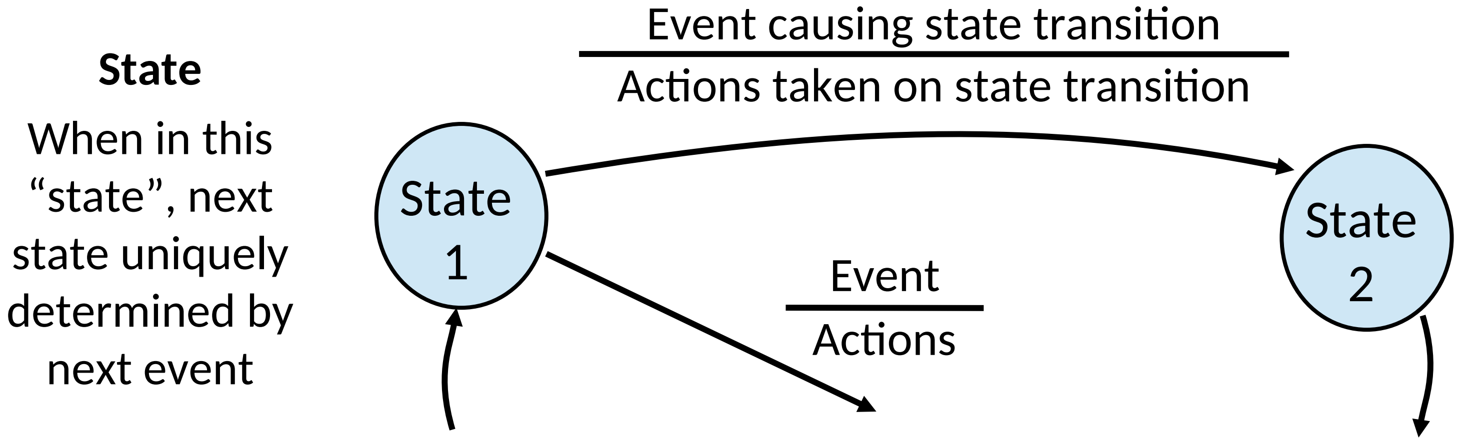


- Reliable data transfer is one of the most important networking topic
- Characteristics of unreliable channel will determine complexity of (fictional) reliable data transfer protocol (`rdt`)

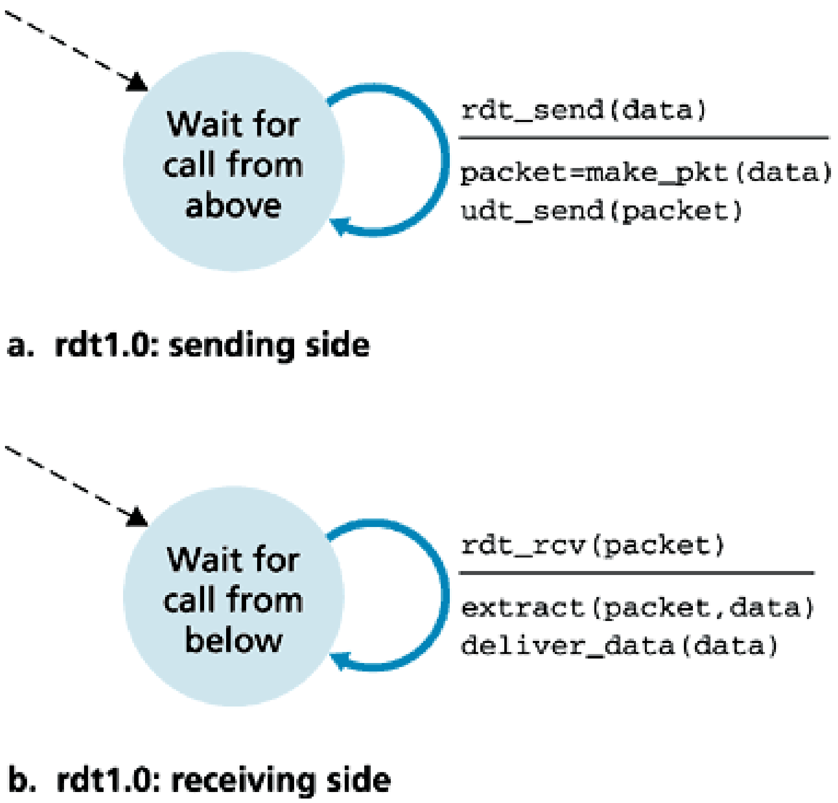
Principles of reliable data transfer
Getting started

Procedure

- Incrementally develop sender, receiver sides of reliable data transfer protocol (`rdt`)
- Consider only unidirectional data transfer
- Control info flow on both directions
- Use finite state machines (FSM)



Principles of reliable data transfer
`rdt1.0` – Reliable transfer over a reliable channel

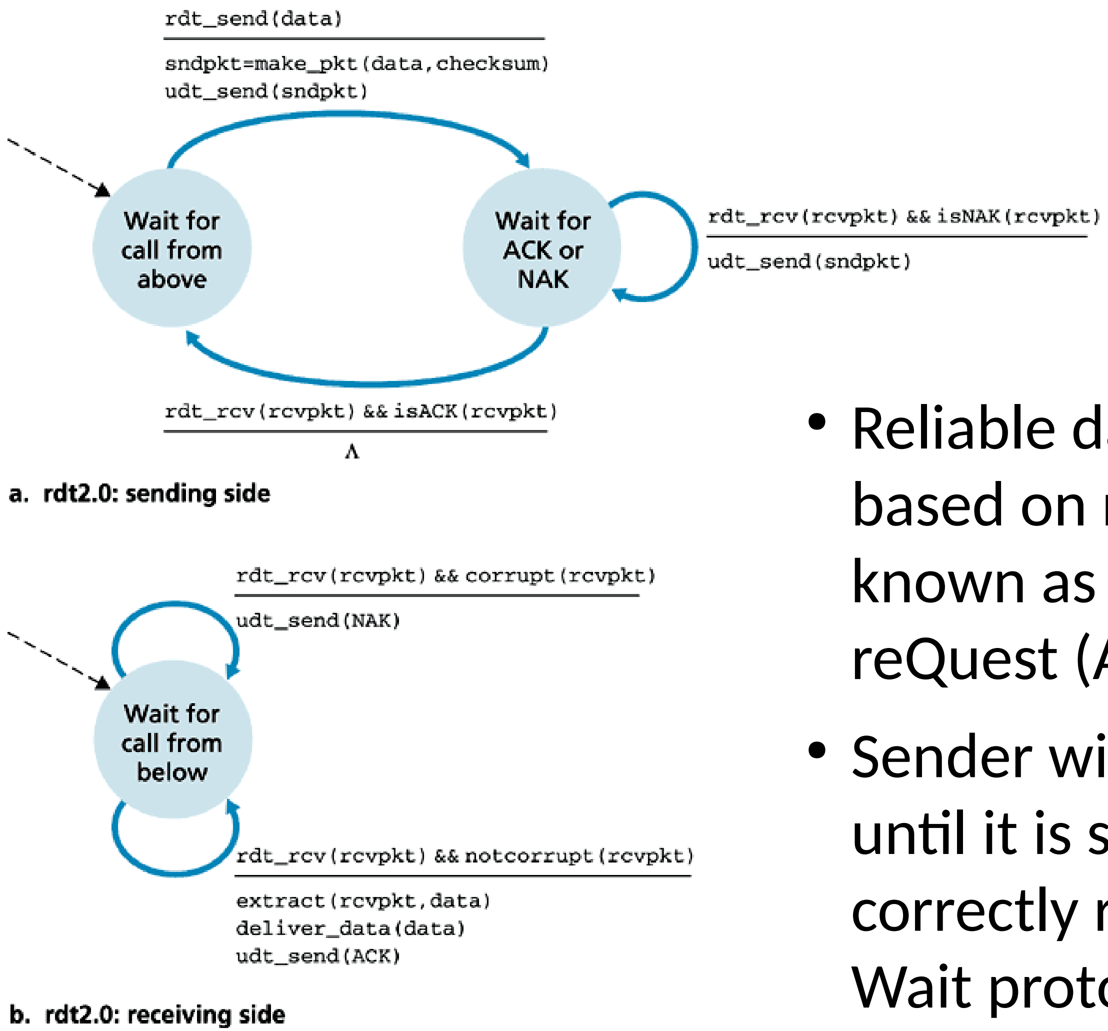


- Underlying channel perfectly reliable
    - No bit errors
    - No loss of packets
- Separate FSMs for sender and receiver
    - Sender sends data into underlying channel
    - Receiver read data from underlying channel

Principles of reliable data transfer
`rdt2.0` – Channel with error bits

- Underlying channel may flip bits in packet
    - UDP checksum to detect bit errors
- How to recover from errors?
    - Acknowledgements (ACKs) : receiver explicitly tells sender that packet received OK
    - Negative acknowledgements (NAKs) : receiver explicitly tells sender that packet had errors ; sender retransmits packet on receipt of NAK
    - Human analogy: dictation
- New mechanisms in rdt2.0 (beyond rdt1.0)
    - Error detection
    - Receiver feedback: control messages (ACK,NAK)
    - Retransmission

Principles of reliable data transfer
`rdt2.0` – FSM Specification



- Reliable data transfer protocols based on retransmissions are known as Automatic Repeat reQuest (ARQ)
- Sender will not send new packet until it is sure previous packet correctly received → Stop-and-Wait protocol (SAW)

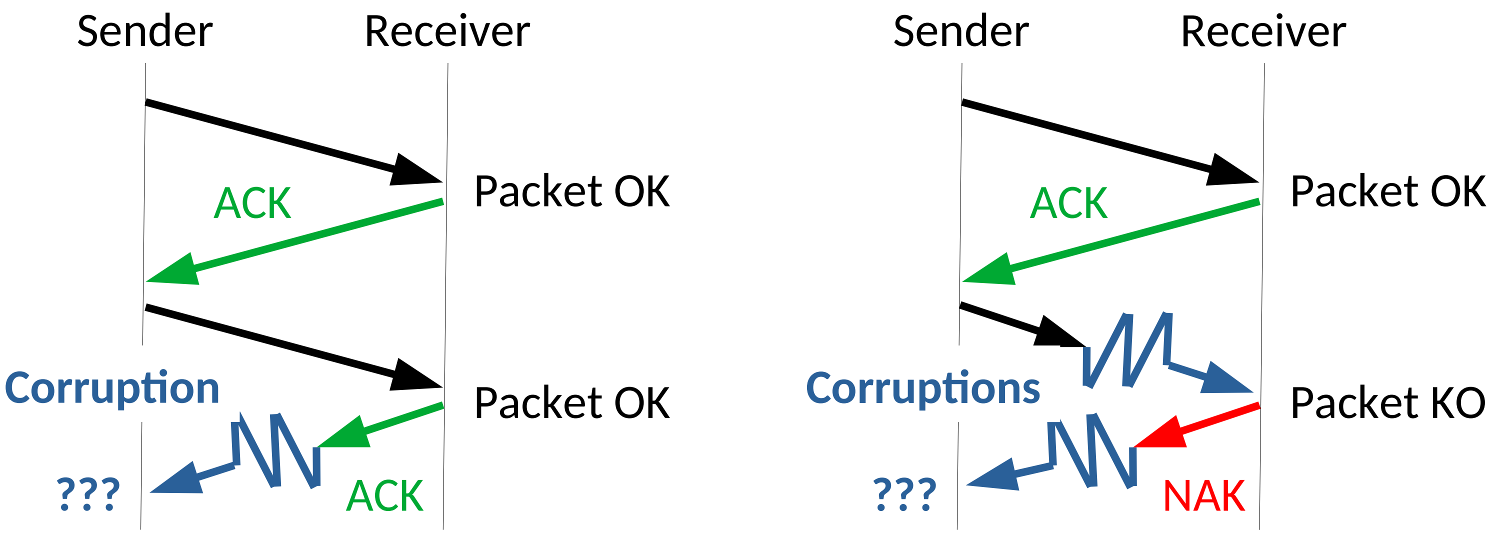
Principles of reliable data transfer

`rdt2.0` – Fatal flaw (1/2)

- What happens if ACK/NAK corrupted?
  - Sender doesn't know what happened at receiver!
  - Can not just retransmit: possible duplicate
- What to do?
  - Sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
  - Retransmit anyway, but this might cause retransmission of correctly received packet
- Simple solution : sender inserts sequence number in packets
  - Solution to unreliability → ACK/NAK
  - Solution to rdt2.0 flaw → sequence number
    - Sender retransmits packet if ACK/NAK corrupted
    - Thanks to sequence number, receiver discards duplicate packets

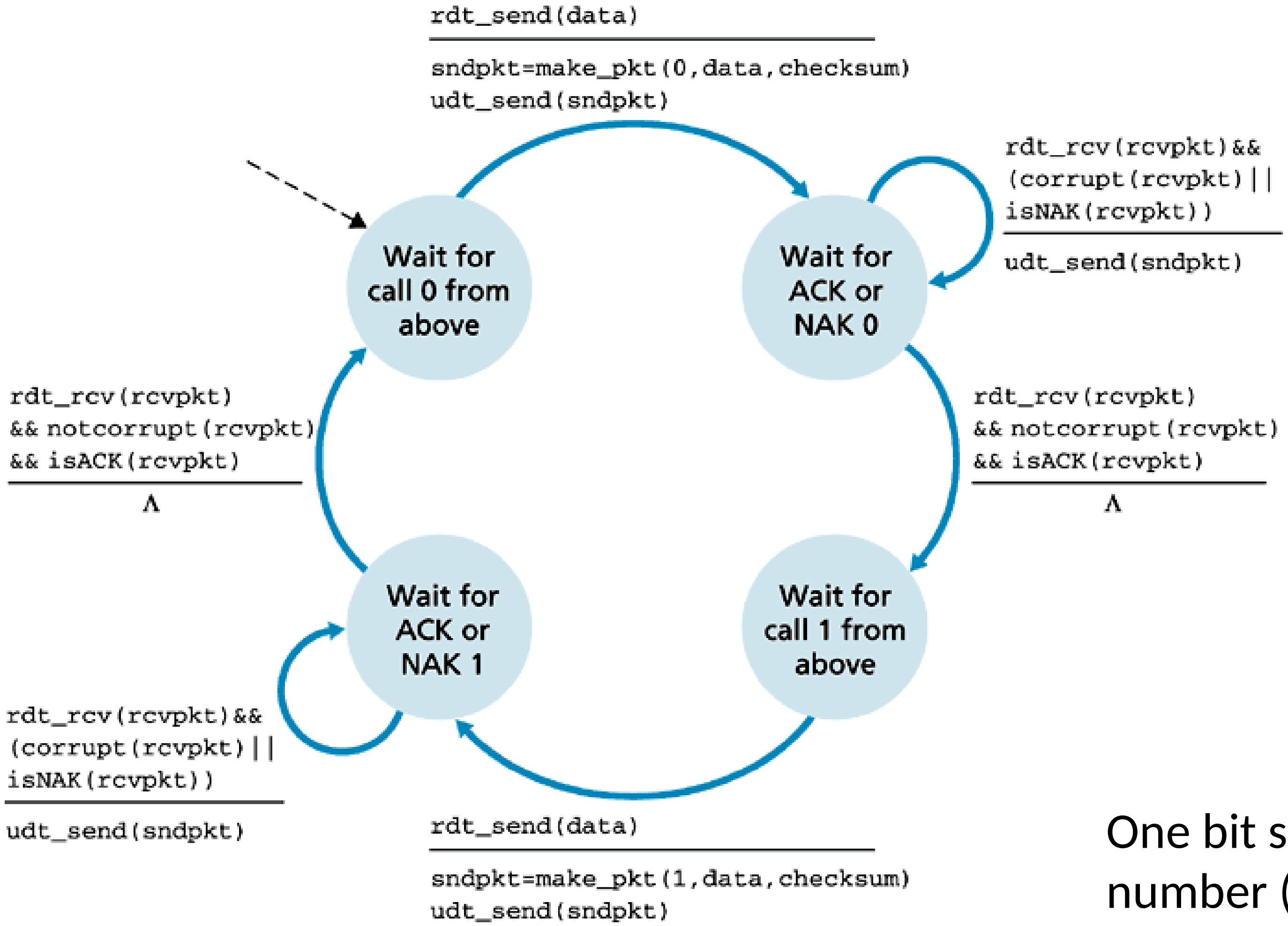
Principles of reliable data transfer
`rdt2.0` – Fatal flaw (2/2)

- Sender receives corrupted feedback
- How to assess the situation?



- Solution : retransmit anyway, but provide means to Receiver to identify duplicates → Sequence numbers

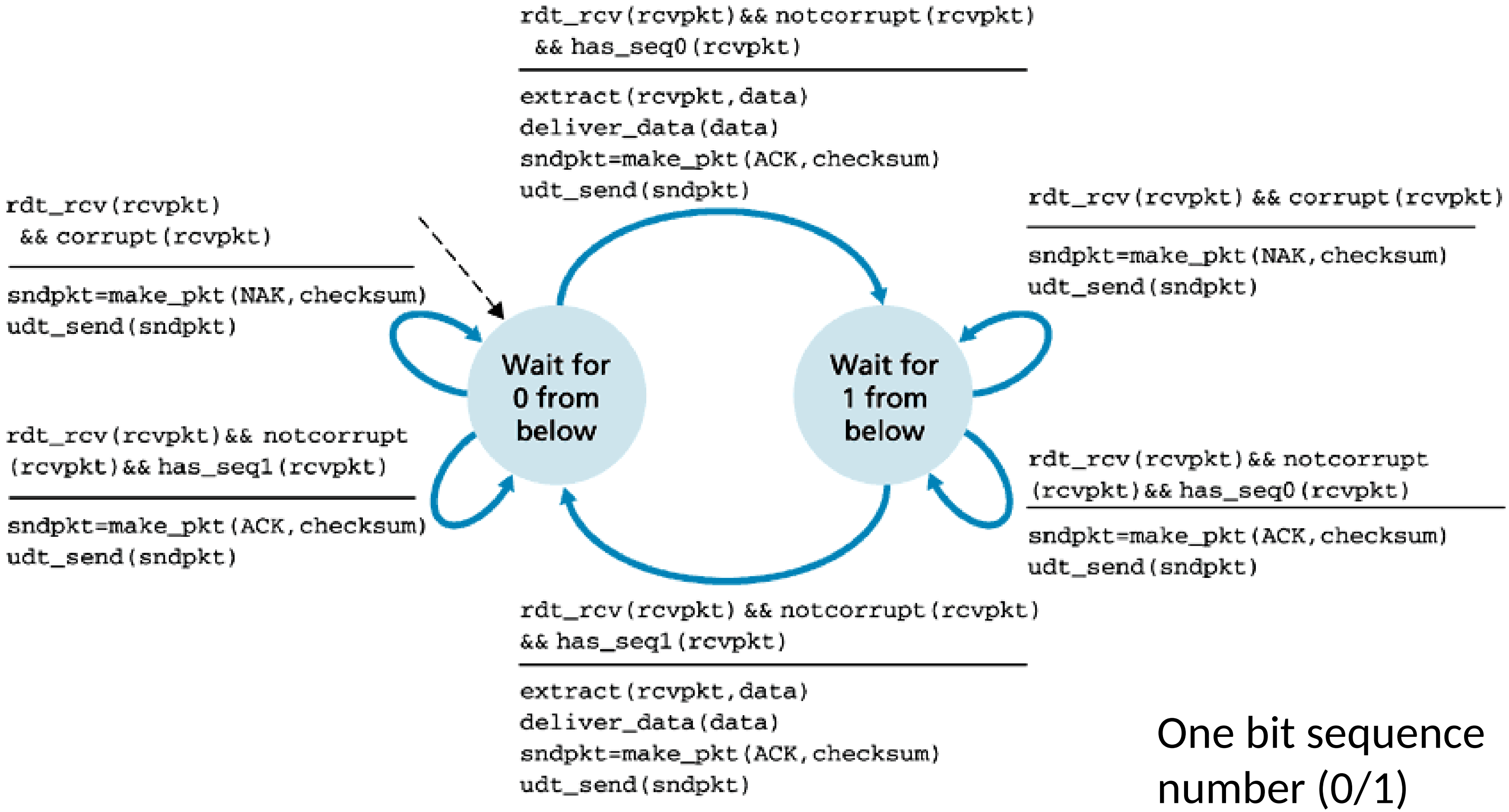
Principles of reliable data transfer
rdt2.1 – Sender FSM



One bit sequence number (0/1)

Principles of reliable data transfer

`rdt2.1` – Receiver FSM



One bit sequence number (0/1)

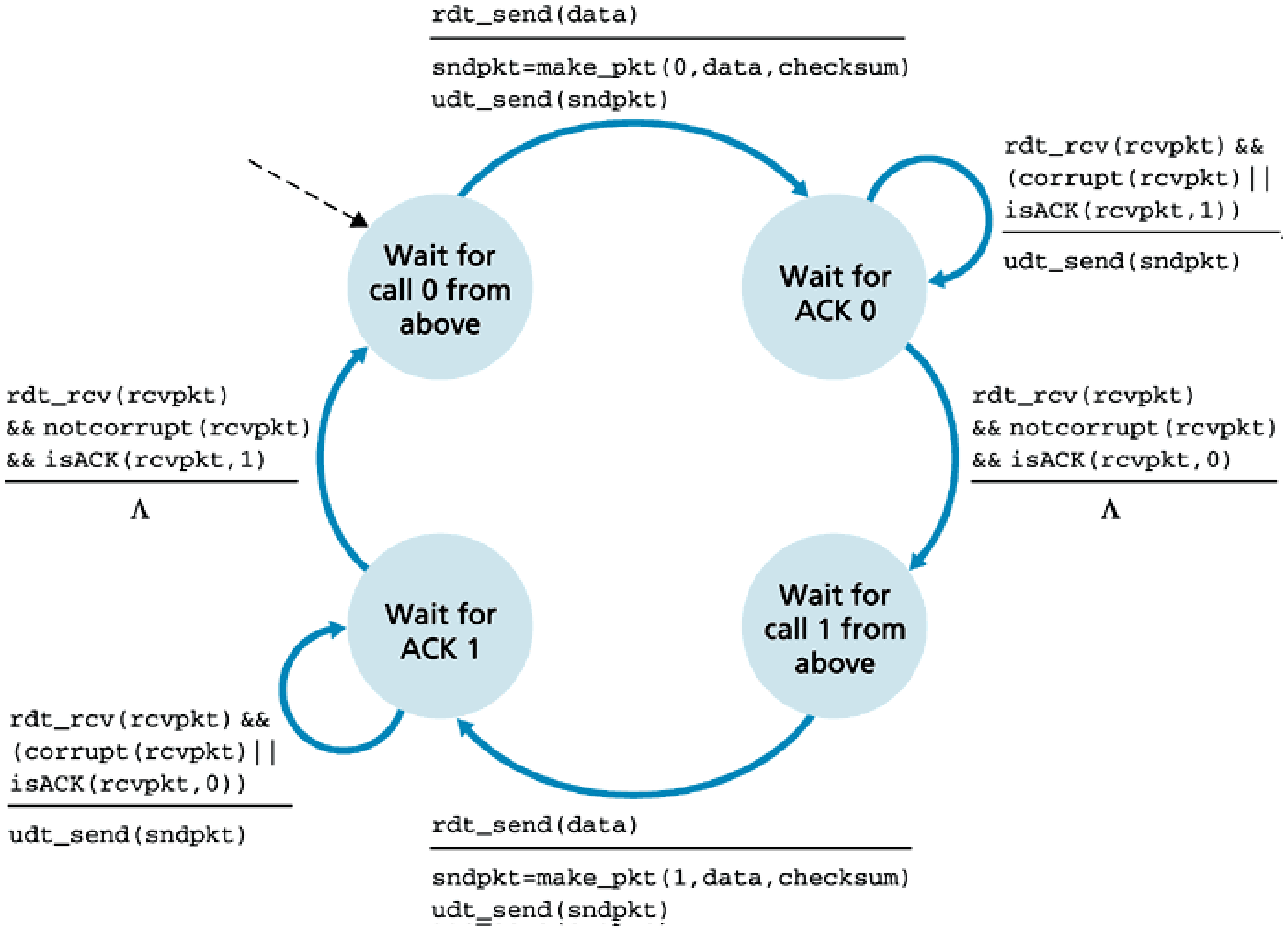
Principles of reliable data transfer
`rdt2.1` – Discussion

- Sender
  - Sequence number added to packet
  - Two sequence numbers (0,1) will suffice
  - Must check if received ACK/NAK corrupted
  - Twice as many states
  - State must “remember” whether “current” packet has 0 or 1 sequence number
- Receiver
  - Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected packet sequence number
  - Receiver can not know if its last ACK/NAK received OK at Sender

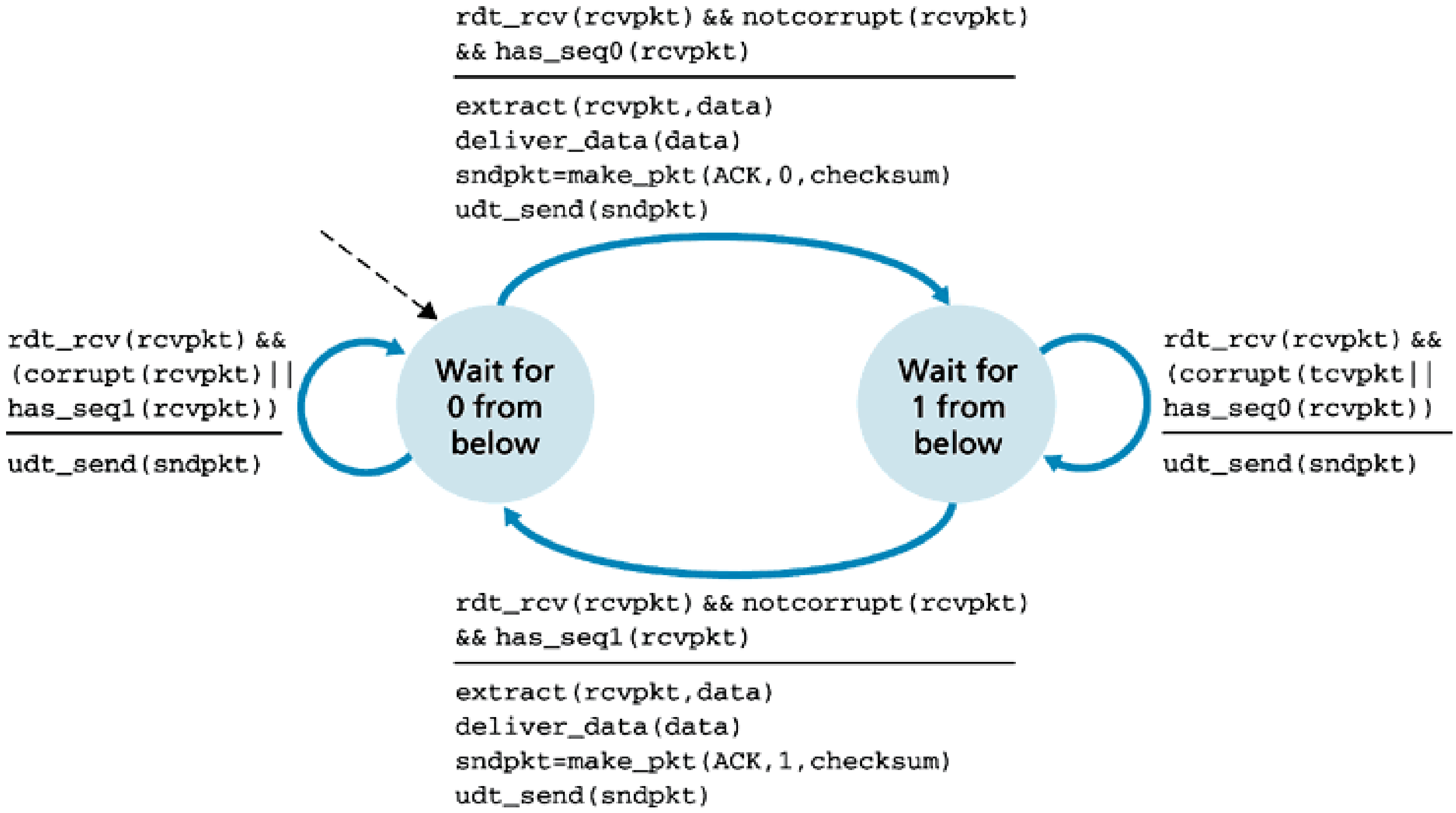
Principles of reliable data transfer
`rdt2.2` – NAK-free protocol

- Same functionality as `rdt2.1`, using ACKs only
- Instead of NAK, receiver sends ACK for last packet received OK
- Receiver must explicitly include sequence number of packet being ACKed
- Duplicate ACKs at Sender result in same action as NAK: retransmit current packet

Principles of reliable data transfer
rdt2.2 – Sender FSM



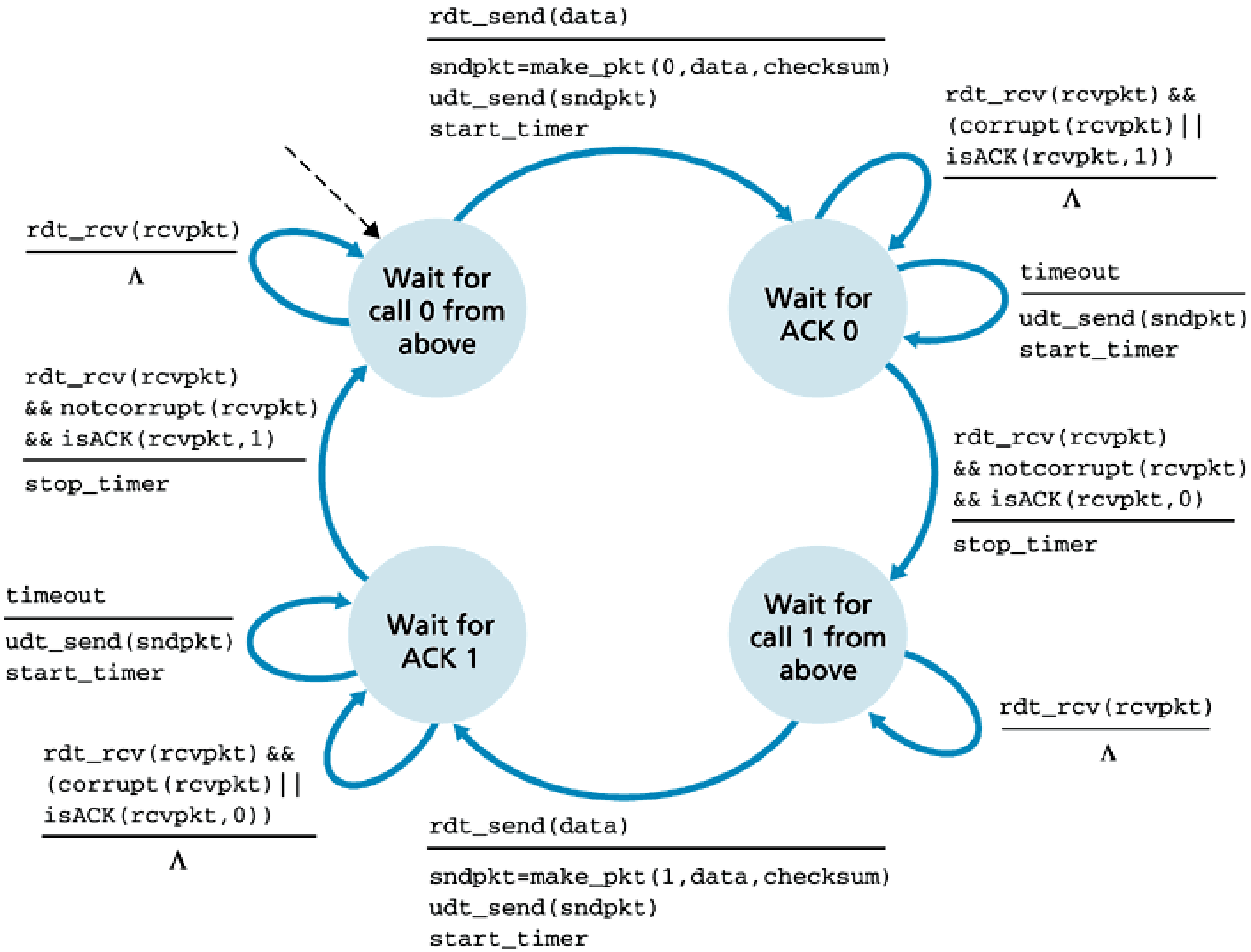
Principles of reliable data transfer
`rdt2.2` – Receiver FSM



Principles of reliable data transfer
`rdt3.0` – Channel with bit errors and losses

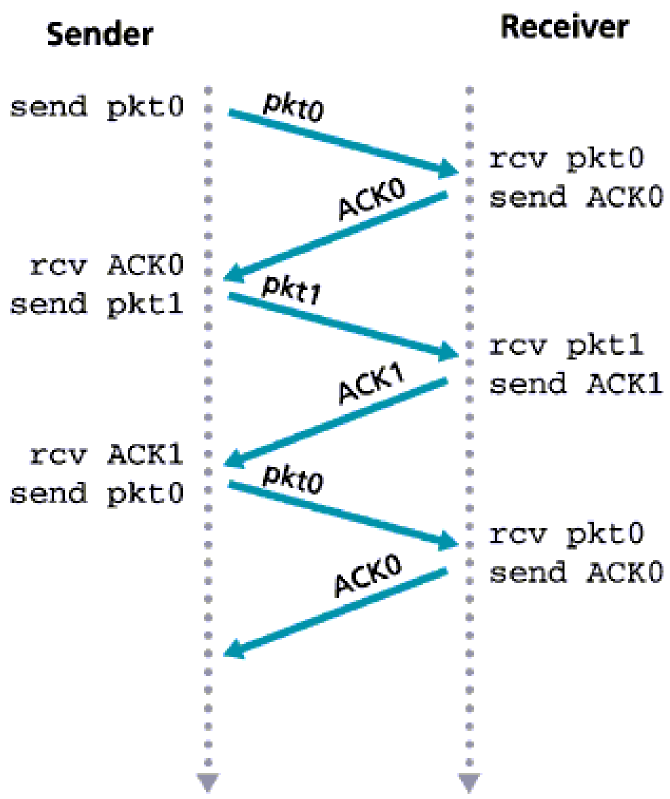
- Underlying channel can also lose packets (data or ACKs)
- Checksum, ACKs, retransmissions, sequence numbers will be of help, but not enough
- How to deal with loss? Sender waits long enough until it is certain that data or ACK lost, then retransmits
- How long should the sender wait? Data expected ASAP
- Sender waits “reasonable” amount of time for ACK ; retransmits if no ACK received in this time
- If packet (or ACK) just delayed (not lost), retransmission will be duplicate, but use of sequence numbers already handles this
- Requires countdown timer

Principles of reliable data transfer
`rdt3.0` – Sender FSM

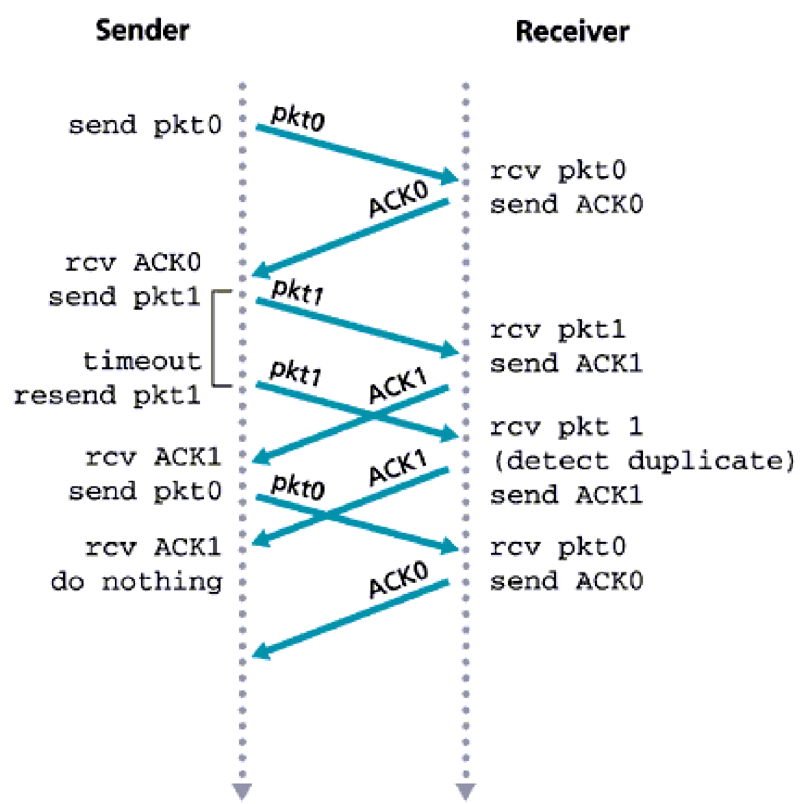


Principles of reliable data transfer
`rdt3.0` – Protocol in action (1/2)

No loss



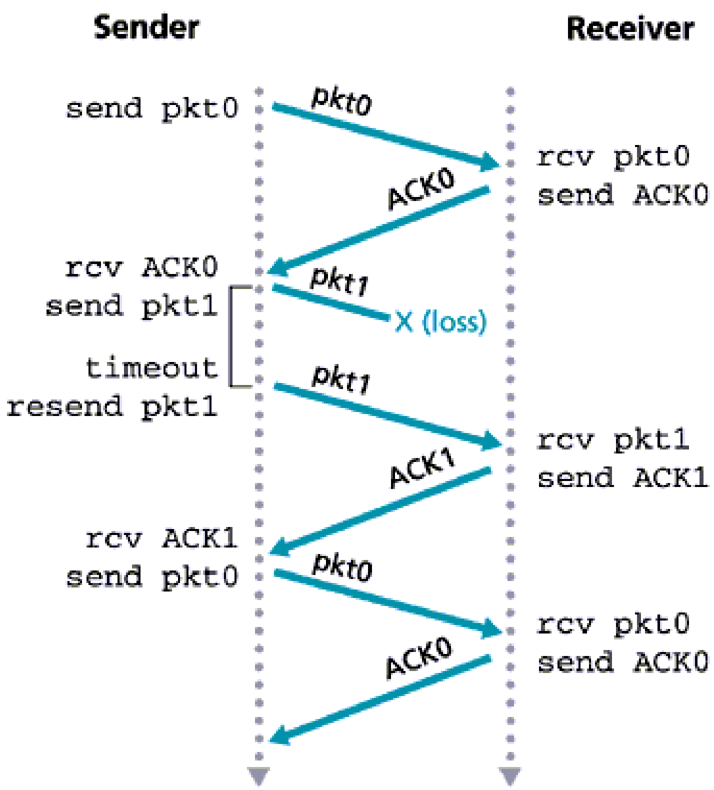
Premature timeout



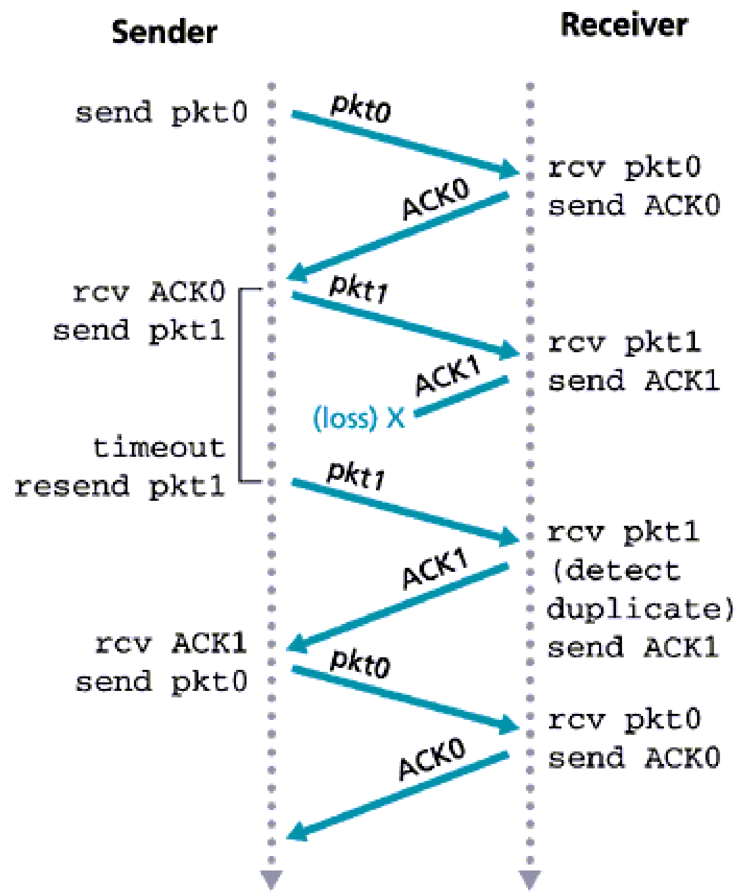
One-bit sequence number → Alternating-bit protocol

Principles of reliable data transfer
`rdt3.0` – Protocol in action (2/2)

Lost packet



Lost ACK



Principles of reliable data transfer

Summary of `rdt` versions

`rdt1.0`

Reliable channel

`rdt2.0` Checksum, ACK/NAK, retransmissions

`rdt2.1` Sequence numbers

`rdt2.2` Numbered ACK

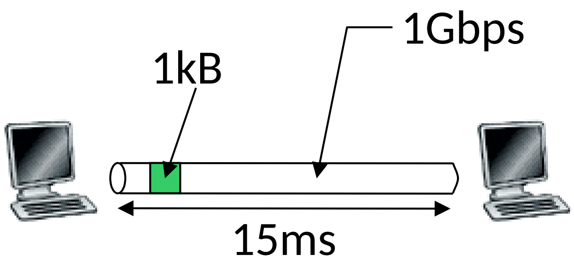
Channel with bit errors

`rdt3.0` Countdown timer

Channel with bit errors and losses

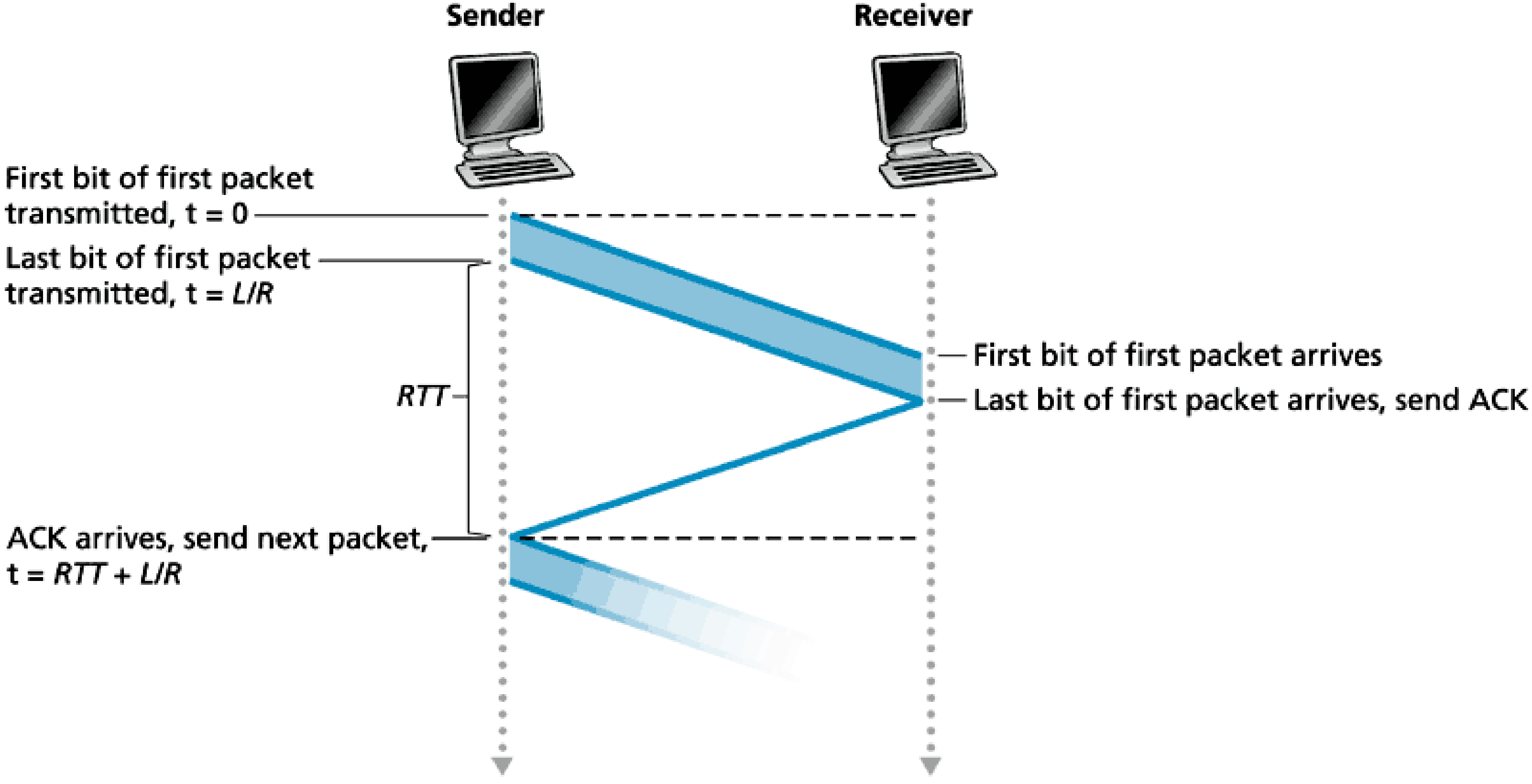
Principles of reliable data transfer
`rdt3.0` – Performance

- `rdt3.0` is a Stop-And-Wait (SAW) protocol
- Assume
  - 1 Gbps link
  - 15 ms E2E propagation delay
  - 1 kB packet



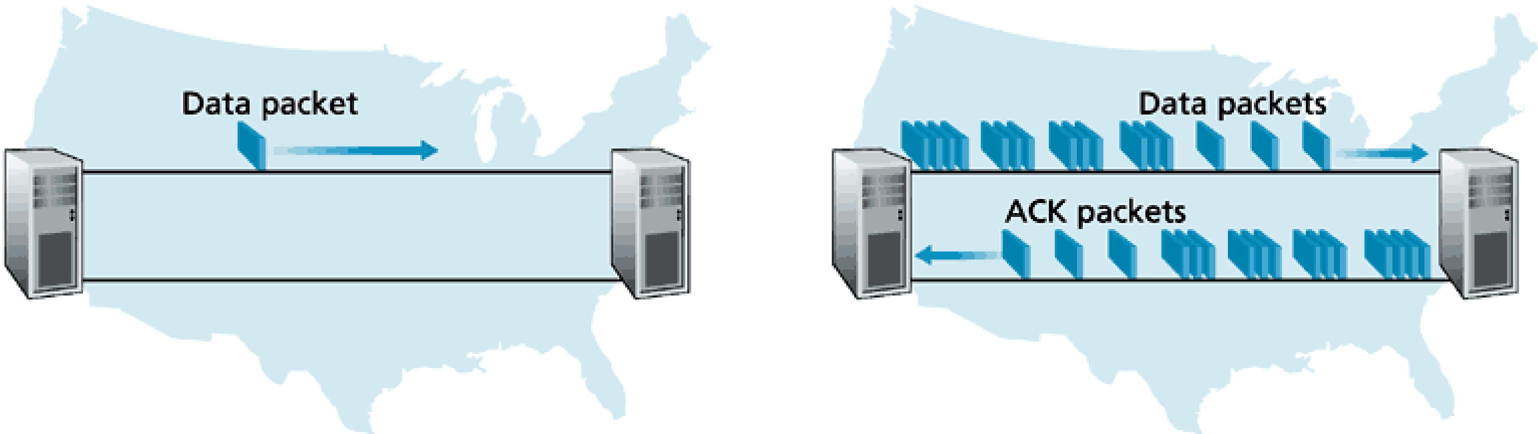
- Transmission delay = L/R = 8 µs << d/s = 15 ms
- 1 kB every 30 ms → 267 kbps throughput over 1 Gbps link
- Network protocol limits use of physical resources!

Principles of reliable data transfer
`rdt3.0` – SAW Operation



$$U_{Sender} = \frac{\text{Time when sender busy}}{\text{Time when channel busy}} = \frac{L / R}{RTT + L / R} = 0.00027$$

Principles of reliable data transfer
Pipelining

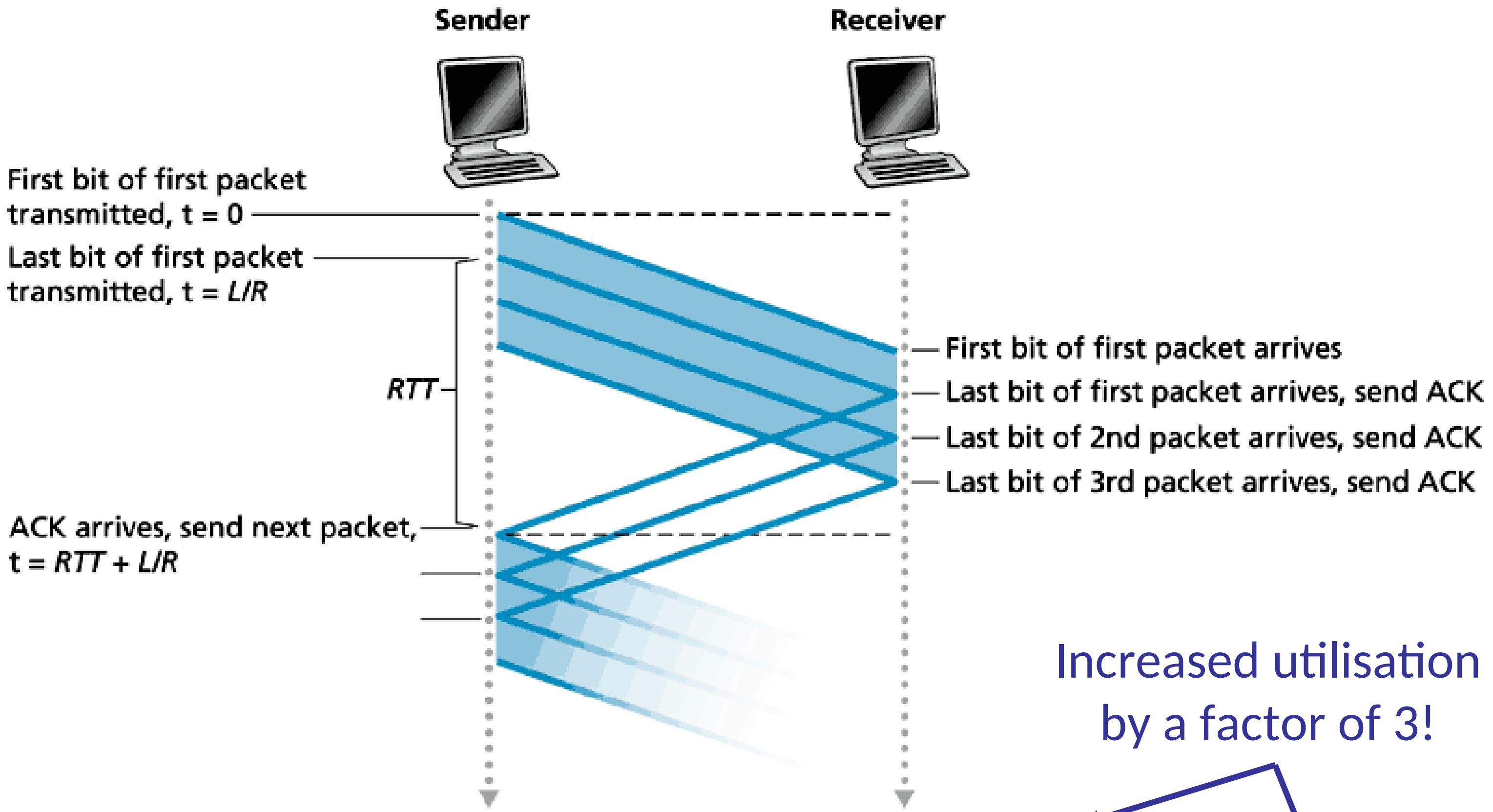


a. A stop-and-wait protocol in operation

b. A pipelined protocol in operation

- Pipelining
    - Sender allows multiple, “in-flight”, yet-to-be-acknowledged packets
    - Range of sequence numbers must be increased
    - Requires buffering at sender and/or receiver
- Two generic forms of pipelined protocols: Go-Back-N, Selective Repeat

Principles of reliable data transfer
Performance of pipelined protocols

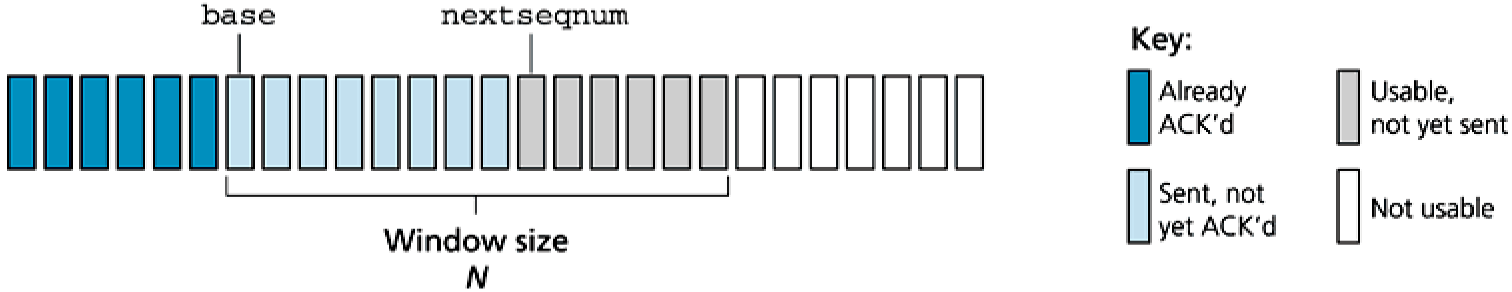


Increased utilisation by a factor of 3!

$$U_{Sender} = \frac{\text{Time when sender busy}}{\text{Time when channel busy}} = \frac{3\ L / R}{RTT + L / R} = 0.00081$$

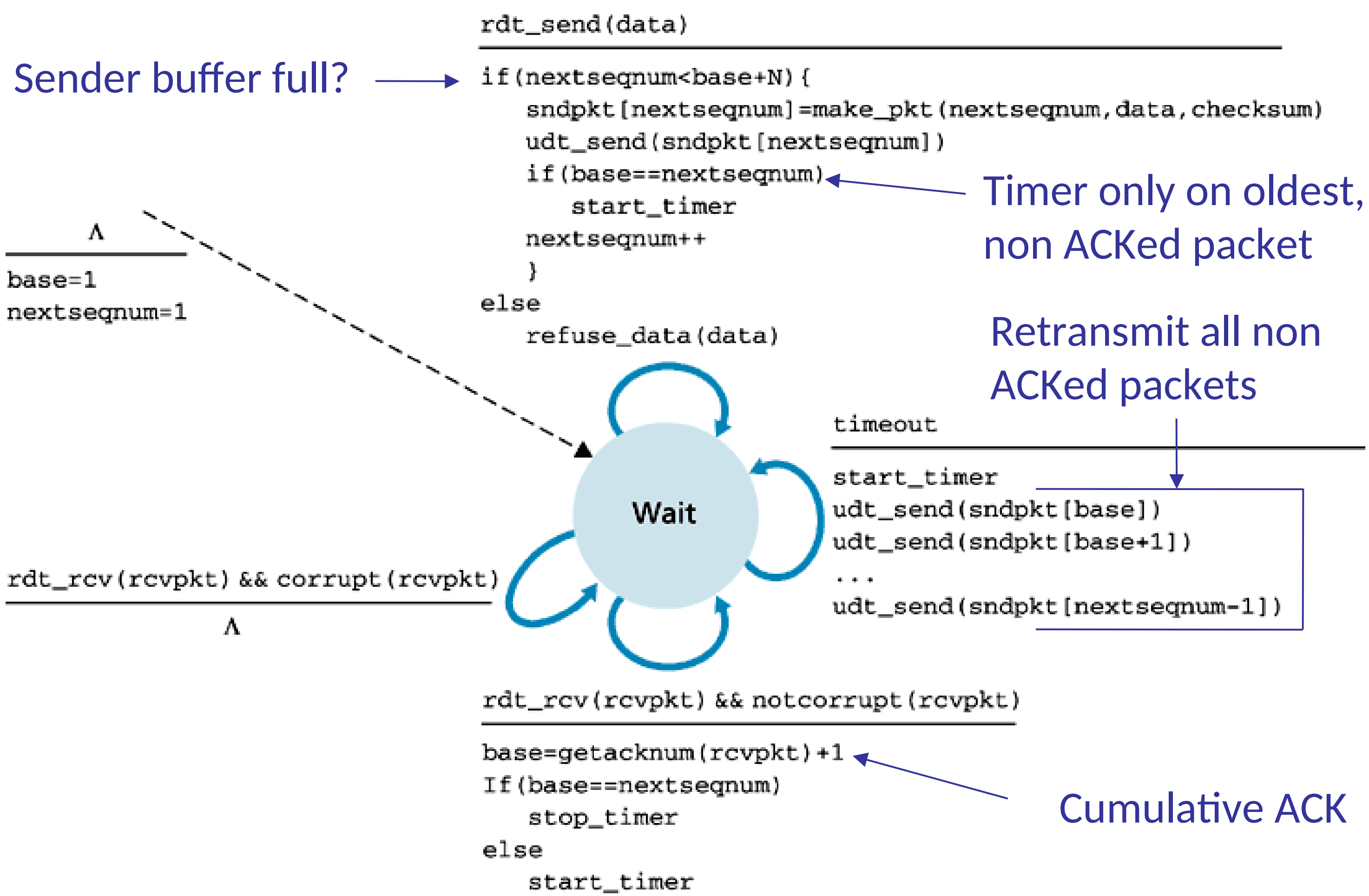
Principles of reliable data transfer
Go-Back-N - Window

- *k*-bit sequence number → 2^k packets in transit
- "Window" of up to *N* consecutive unacknowledged packets



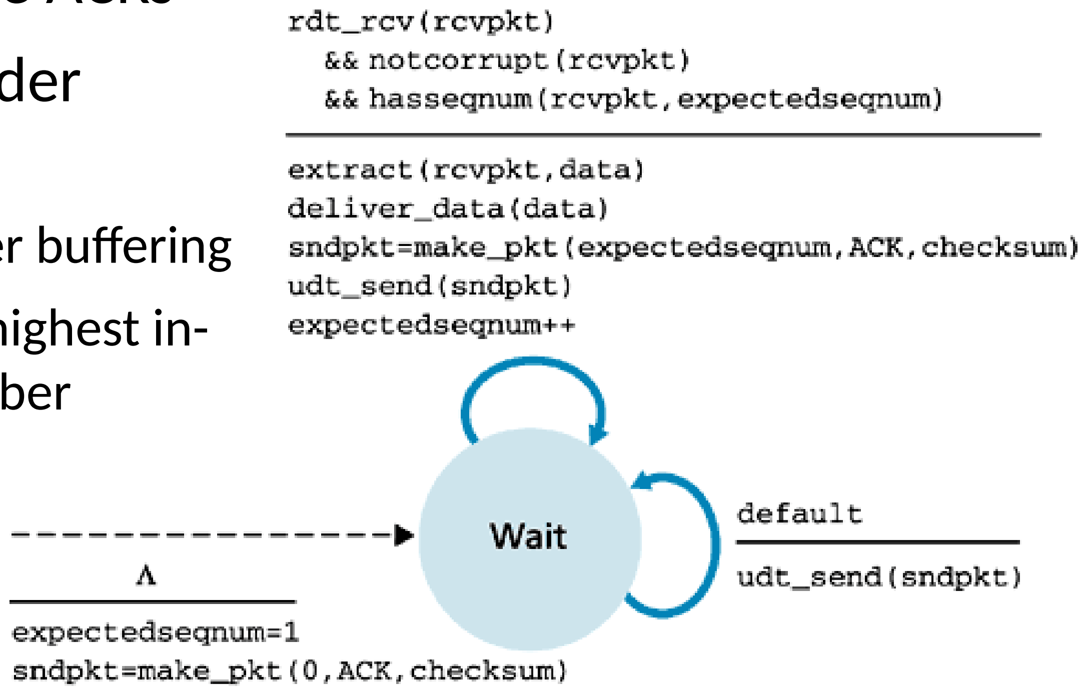
- Cumulative ACK - ACK(*n*)
  - ACKs all packets up to, including sequence number *n*
  - May deceive duplicate ACKs (see receiver)
- Timer for each in-flight packet
- `timeout(n)`: retransmit packet n and all higher sequence numbered packets in window

Principles of reliable data transfer
Go-Back-N - Extended Sender FSM

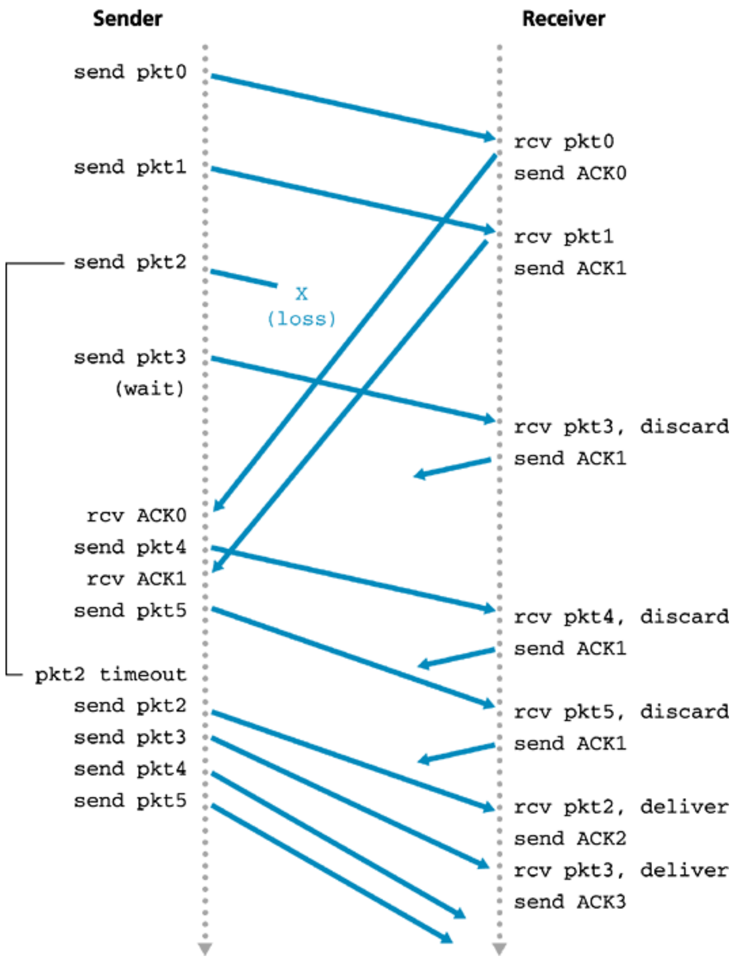


Principles of reliable data transfer
Go-Back-N - Extended Receiver FSM

- Always send ACK for correctly-received packet with highest in-order sequence number
- May generate duplicate ACKs
- Reception of out-of-order packet
  - Discard → no receiver buffering
  - Re-ACK packet with highest in-order sequence number



Principles of reliable data transfer
Go-Back-N – Protocol in action

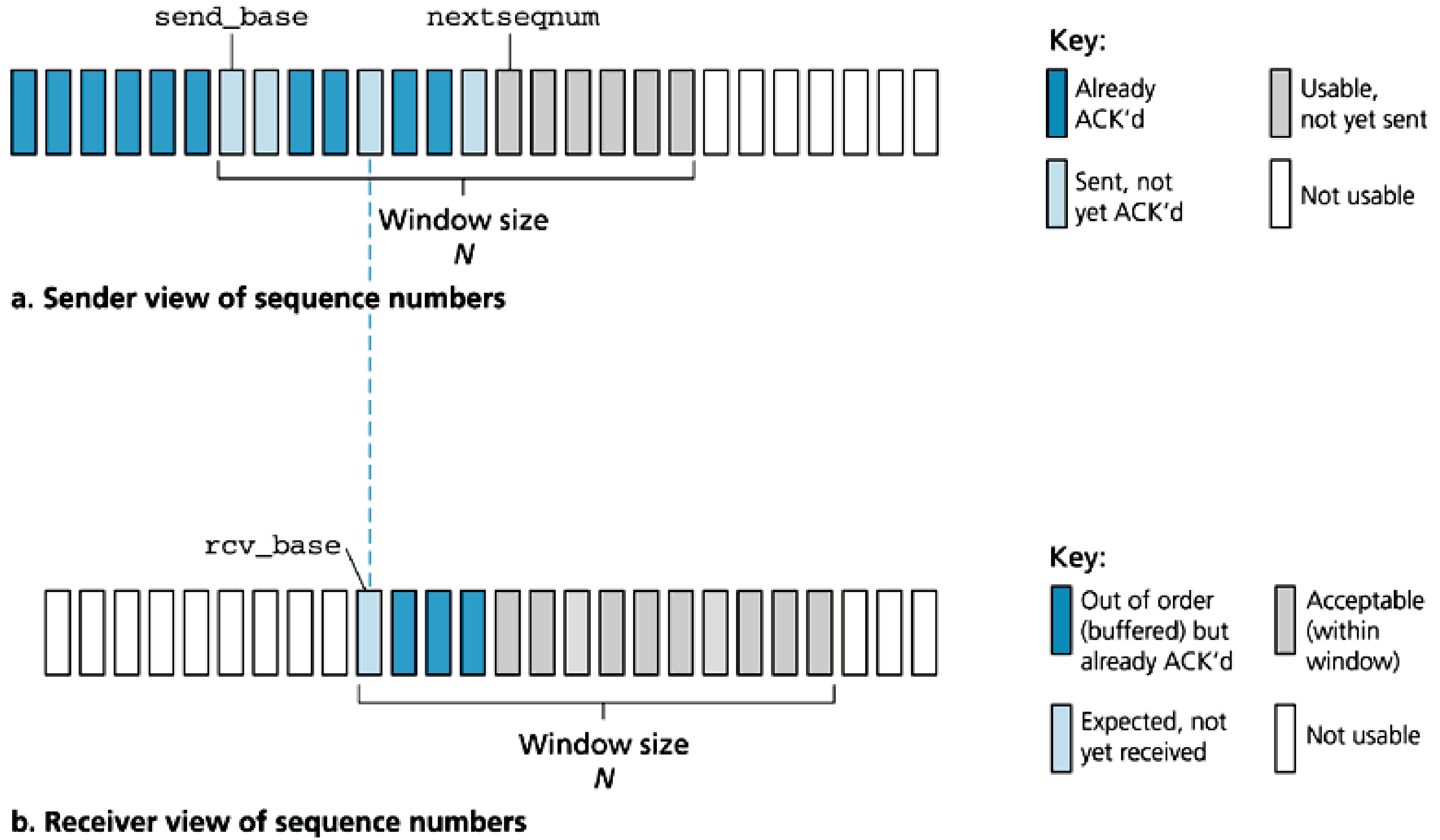


- Window size = 4 packets
- Go-Back-N incorporates
  - Sequence numbers
  - Cumulative ACKs
  - Checksums
  - Timeout/retransmit

Principles of reliable data transfer
Pipelined protocols – Selective Repeat

- Main drawback of Go-Back-N: a single packet error may cause many packets to be retransmitted
- Receiver
  - Individually acknowledges all correctly received packets
  - Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender
  - Only resends packets for which ACK not received
  - Timer for each unACKed packet
- Sender window
  - *N* consecutive sequence numbers
  - Again limits sequence numbers of sent, unACKed packets

Principles of reliable data transfer
Selective Repeat - Window



Principles of reliable data transfer
Selective Repeat - Actions

Sender

Data from above

If next available sequence number in window, send packet

timeout(n)

Resend packet *n*, restart timer

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

- Mark packet *n* as received
- if *n* smallest unACKed packet, advance window base to next unACKed sequence number

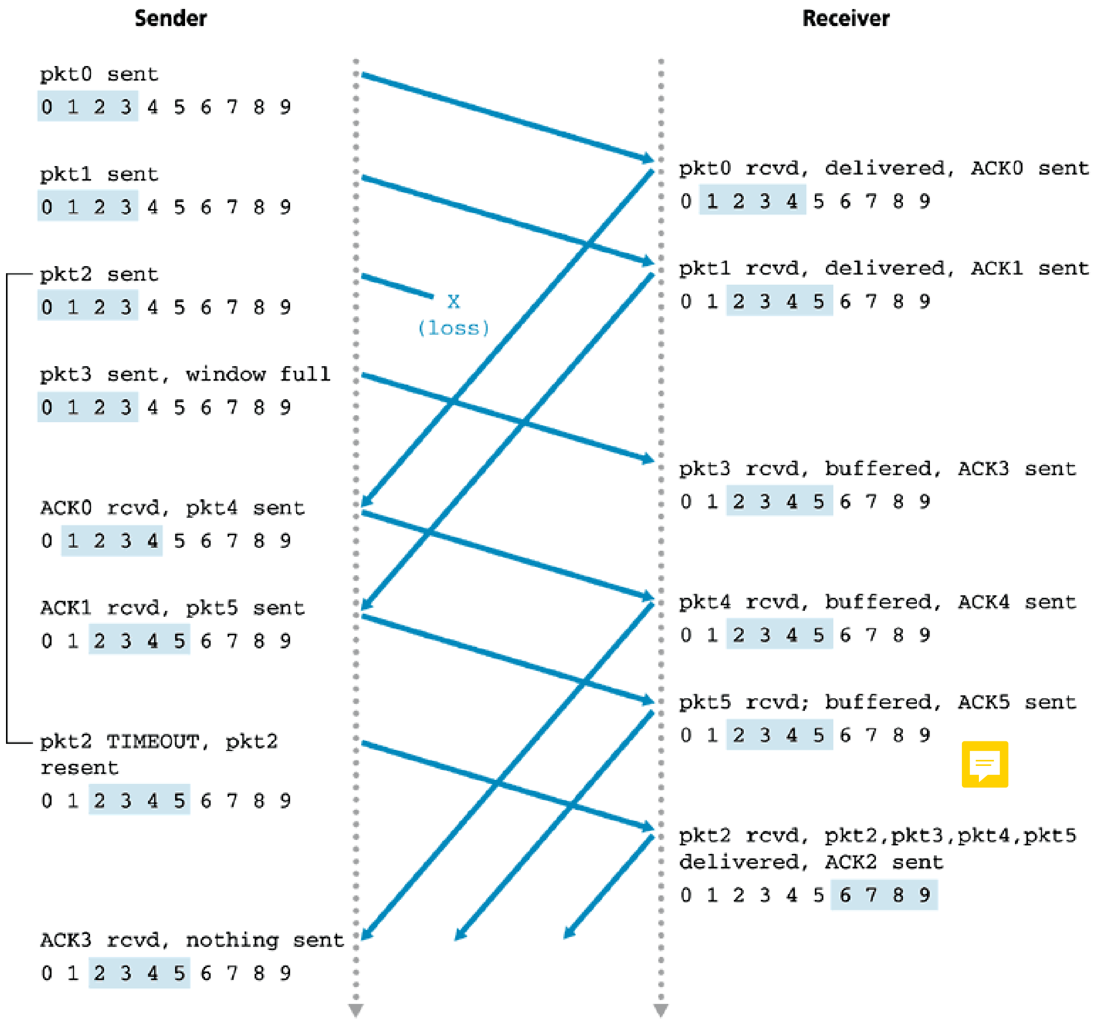
Receiver

- Packet *n* in [rcvbase, rcvbase+N-1]
  - Send ACK(*n*)
  - In-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet
  - Out-of-order: buffer
- Packet *n* in [rcvbase-N, rcvbase-1]

  Re-ACK(*n*)
- Otherwise

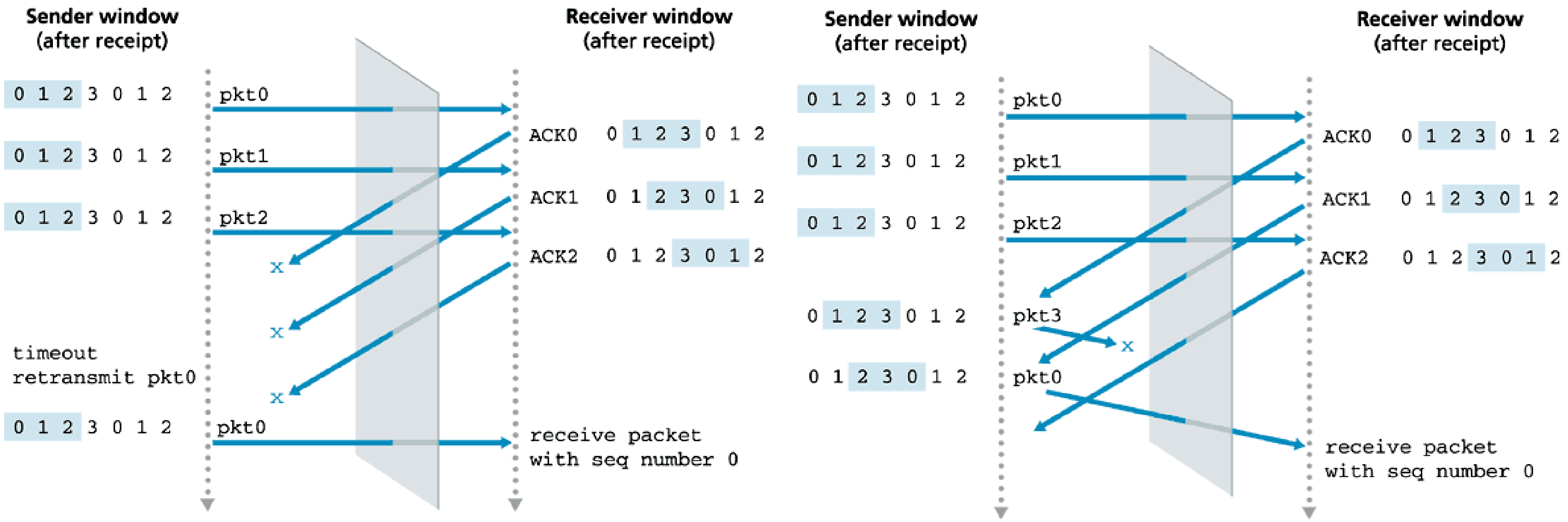
  Ignore

Principles of reliable data transfer
Selective Repeat – Protocol in action



Principles of reliable data transfer
Pipelined protocols – Dilemma

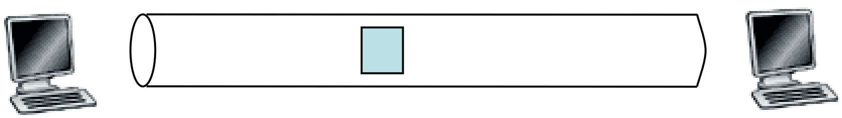
- 2-bit sequence numbering; window size = 3 segments



- Receiver sees no difference in two scenarios!
- Incorrectly passes duplicate data as new
- Solution: k-bit numbering such that $2^{(k-1)}$ > window size

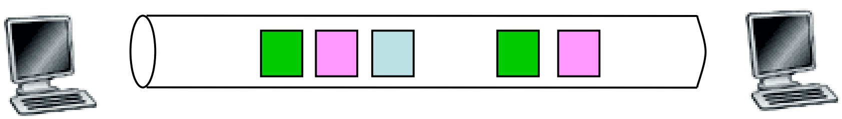
Principles of reliable data transfer
Summary of ARQ protocols

- Stop-And-Wait



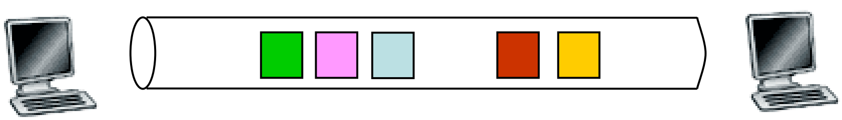
Single packet on the fly

- Go-Back-N



Several packets, possibly repeated, on the fly

- Selective Repeat



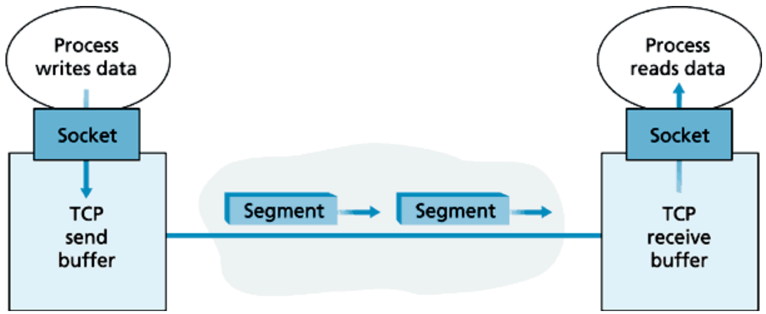
Several original packets on the fly

Outline

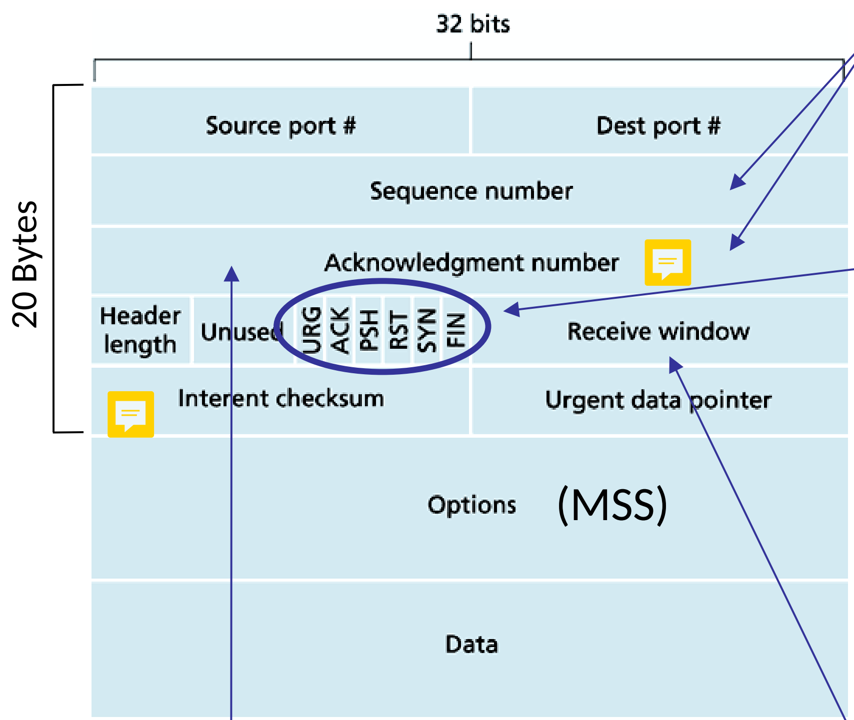
- Transport-Layer services
- Multiplexing and demultiplexing
- Connectionless transport – UDP
- Principles of reliable data transfer
- Connection-oriented transport – TCP
  - Segment structure
  - Connection management
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control
- Quick UDP Internet Connections – QUIC

Connection-oriented transport
TCP connection

- Point-to-point: one sender, one receiver
- Full duplex: bi-directional data flow in connection
- Characterised by Maximum Segment Size (MSS)
- Reliable, in-order byte stream service
- Connection-oriented: handshaking initialises sender and receiver states before data exchange
- Flow controlled: sender will not overwhelm receiver
- Pipelined
  - TCP congestion and flow control set window size
  - Send and receive buffers



Connection-oriented transport
TCP segment structure



Counting in bytes (not segments)

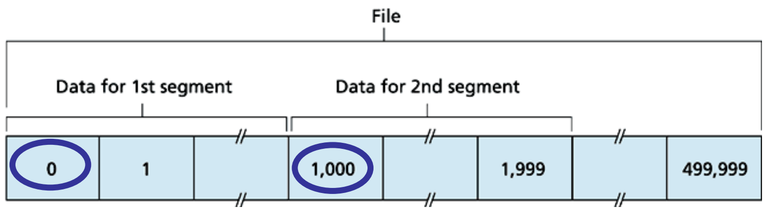
URG: urgent data (generally not used)
ACK: signals ACK number is valid
PSH: push data now (generally not used)
RST, SYN, FIN: connection establishment (setup, teardown commands)

Flow control: number of bytes receiver is willing to accept, up to 65 536 bytes

Piggybacking: ACK data A→B in segment B→A

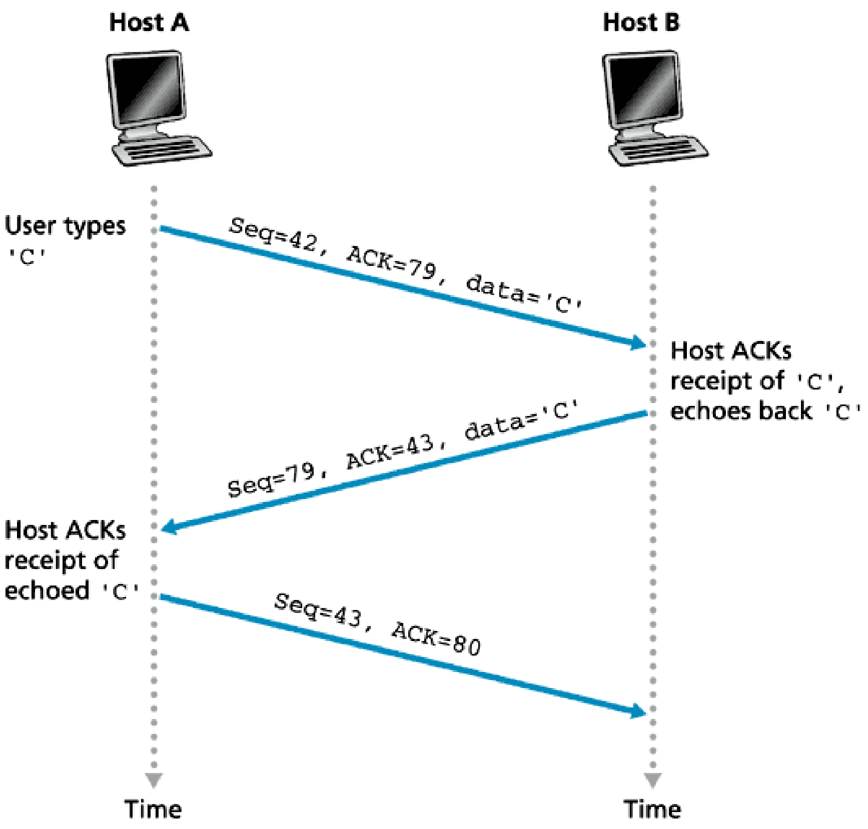
Connection-oriented transport
TCP sequence numbering (1/2)

- Byte stream “number” of first byte in data

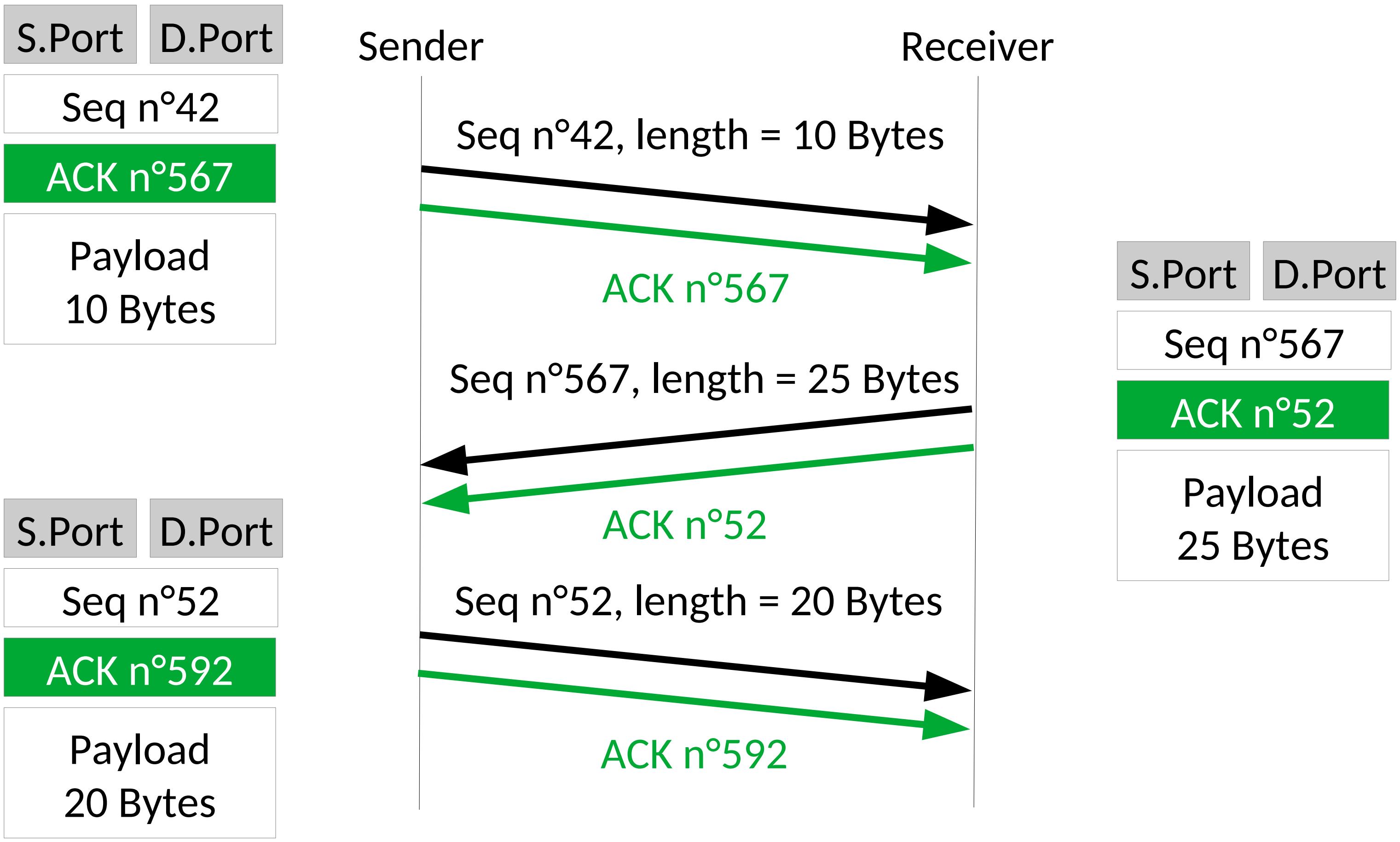


MSS =1 kB

- ACKs = sequence number of next byte expected from other side
- Out-of-order segments? No rules in TCP RFCs

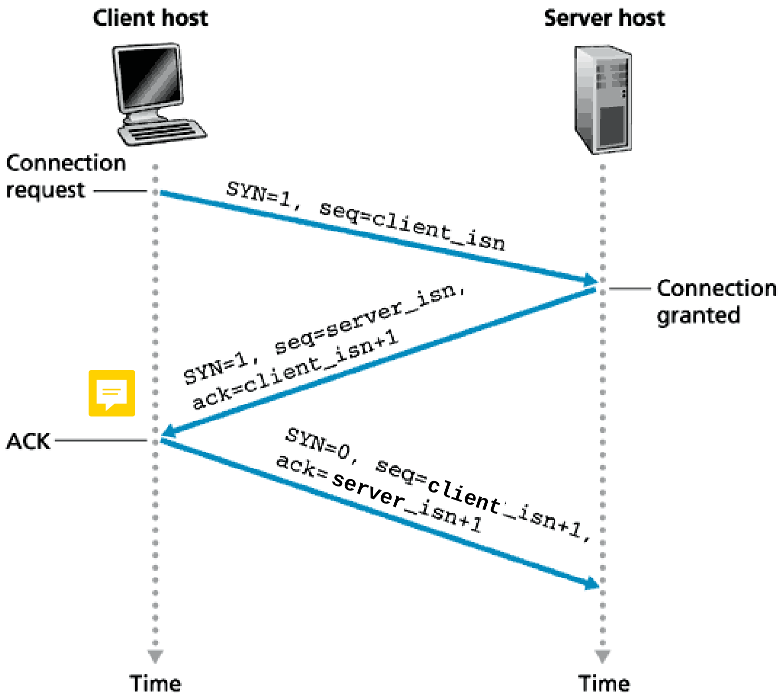


Connection-oriented transport
TCP sequence numbering (2/2)



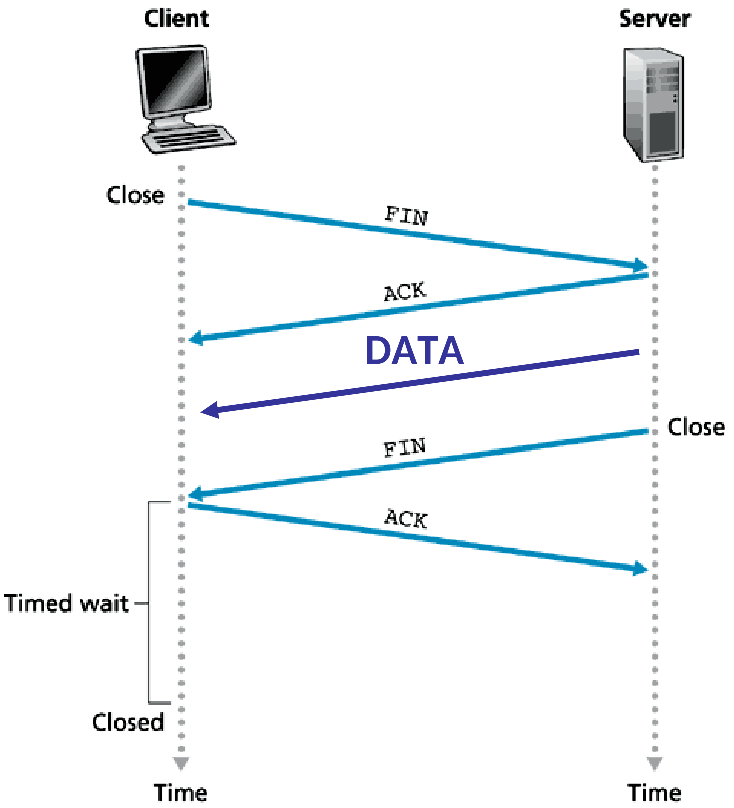
Connection-oriented transport
TCP connection opening

- Hosts establish a “connection” before exchange
- Initialisation TCP variables (sequence numbers, buffers, flow control info)
- Three-way handshake
  - Client host sends TCP **SYN** segment to server (initial sequence number, no data)
  - Server host receives SYN, replies with **SYNACK** segment, allocates buffers and specifies server ISN
  - Client receives SYNACK, replies with **ACK** segment, which may contain data

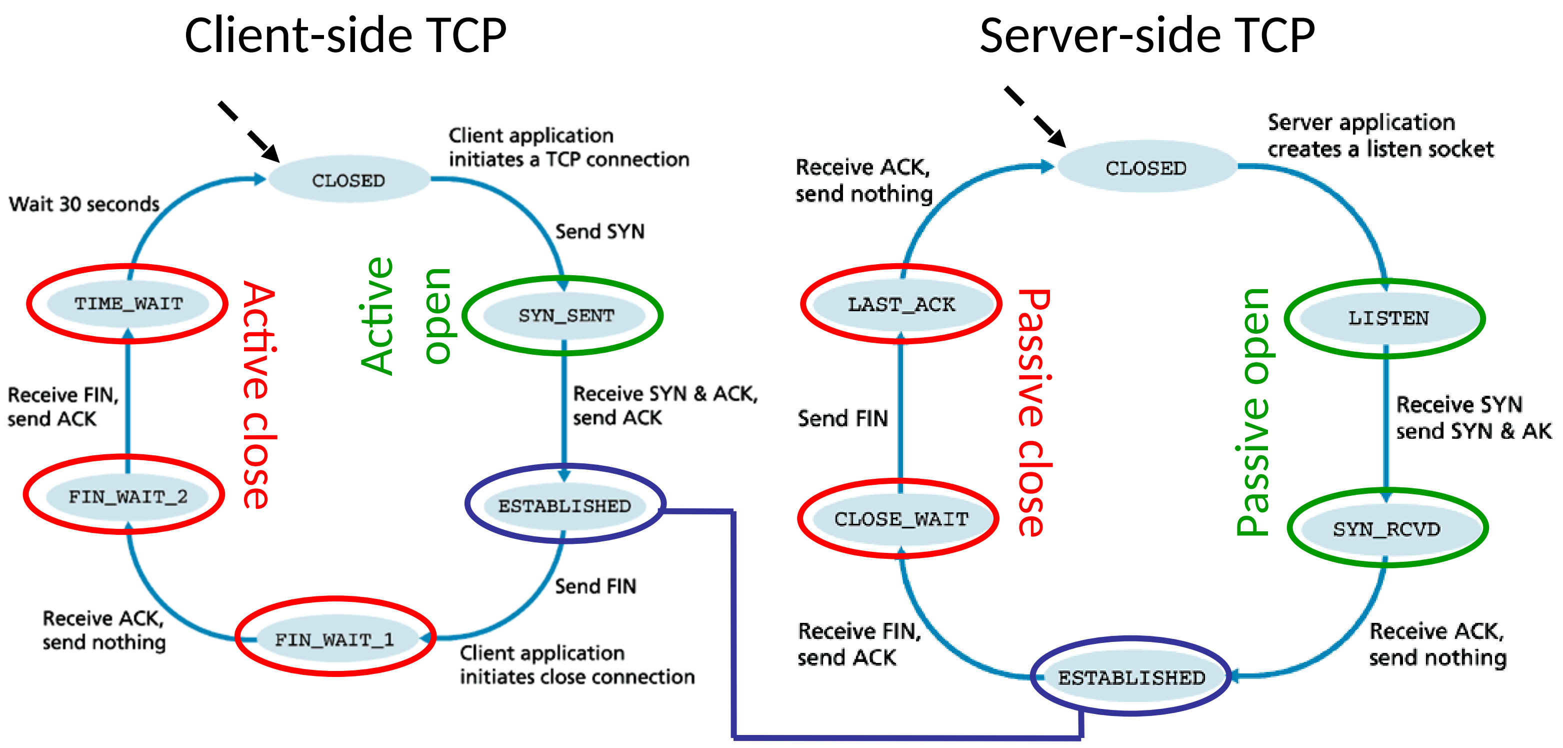


Connection-oriented transport
TCP connection tear-down

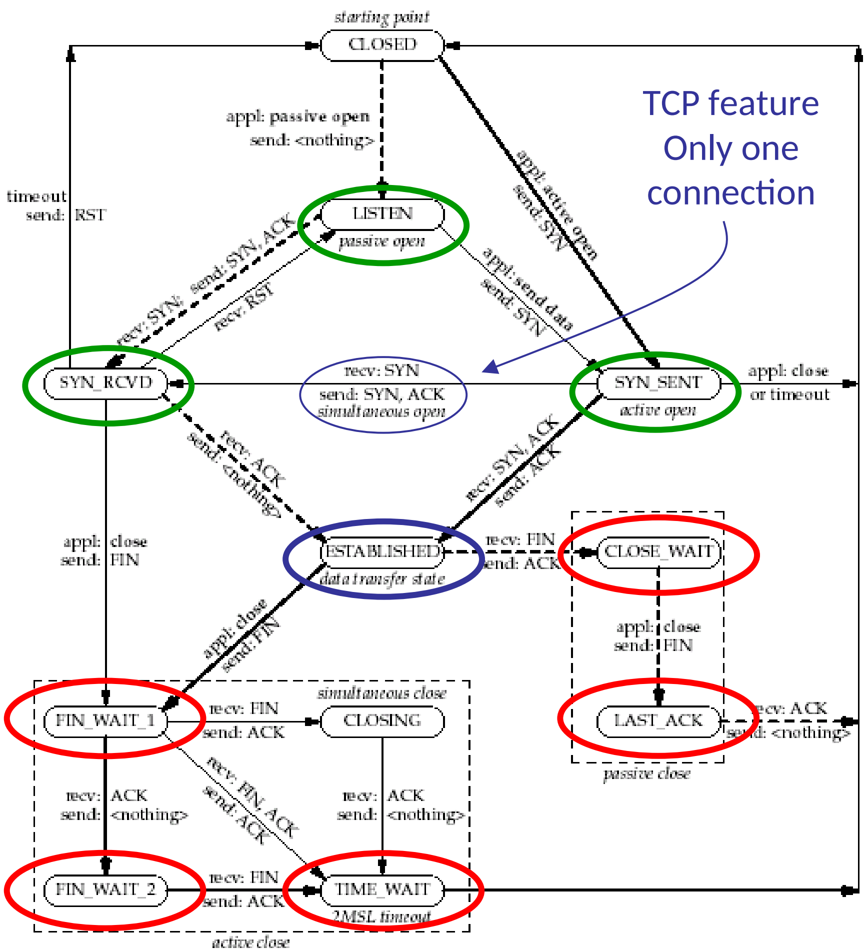
- Four segments: full-duplex implies shut down of two independent connections
- Client sends **FIN** control segment to server
- Server receives **FIN**, replies with **ACK**, closes connection and sends **FIN**
- Client
  - Receives **FIN**, replies with **ACK**
  - Enters “Timed wait”, will respond with ACK to received **FINs**
- Server receives **ACK**. Connection closed.



Connection-oriented transport
Simplified TCP State Transition Diagram

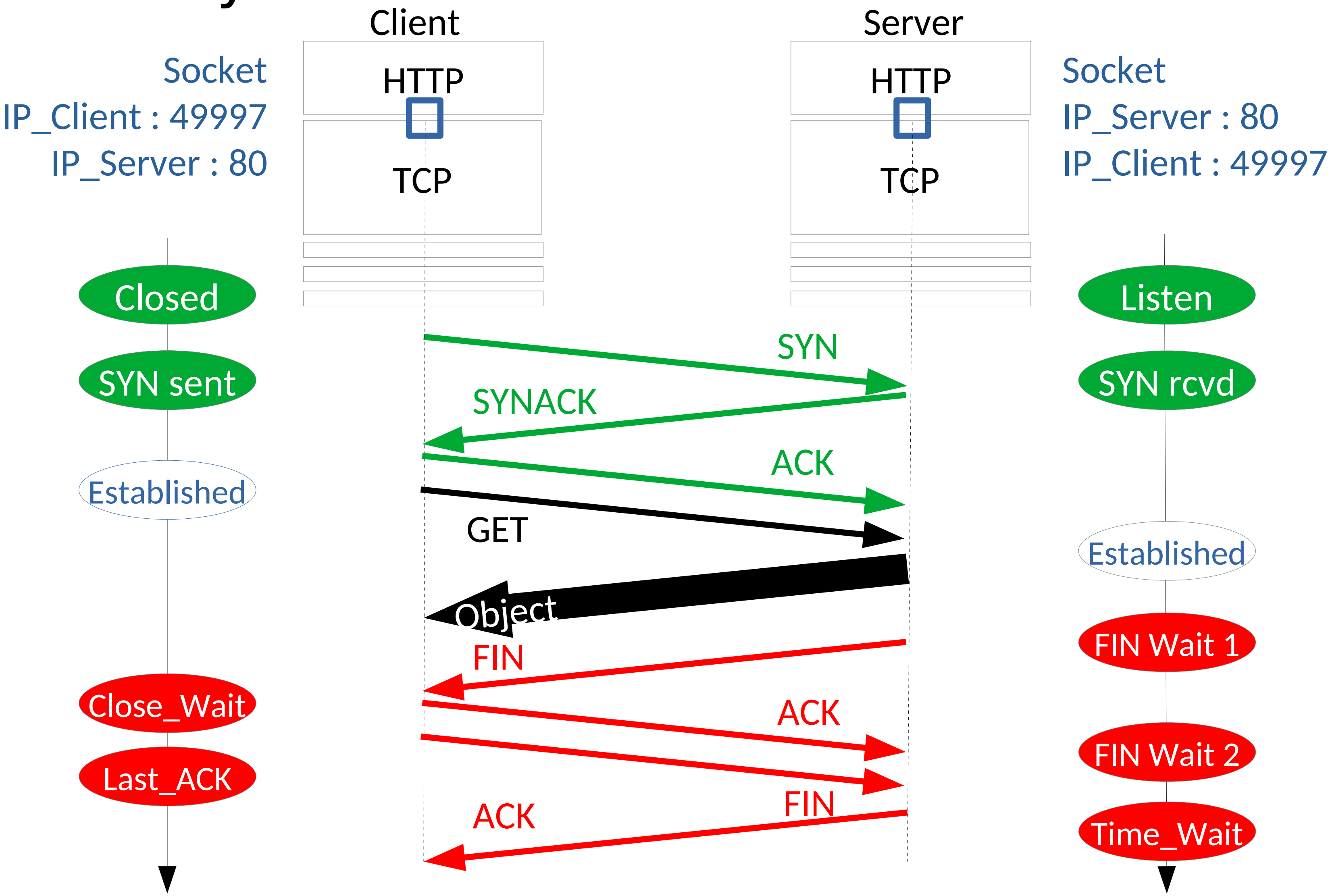


Connection-oriented transport
Steven's TCP State Transition Diagram



- Previous slide shows typical transitions (darker solid/dashed arrows)
- TIME_WAIT state also known as 2MSL wait state
- In TIME_WAIT
  - ACK can be resent if lost
  - Socket 4-uple can not be reused (issue for servers)
  - Old messages not regarded as new
- MSL (Maximum Segment Lifetime) is MAX amount of time a segment can exist before being discarded
- Usually 30-120 s

Connection-oriented transport
Summary



Connection-oriented transport
TCP reliable data transfer

- TCP creates this reliable service on top of IP's unreliable service, using
  - Pipelined segments
  - Cumulative ACKs
  - Single retransmission timer
- Retransmissions are triggered by
  - Timeout events
  - Duplicate ACKs (dupACK)
- Initially consider simplified TCP sender
  - Ignore dupACKs
  - Ignore flow and congestion control

Connection-oriented transport
Simplified TCP Sender

- Using only timeouts to recover lost segments

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
  switch(event)

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

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

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

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

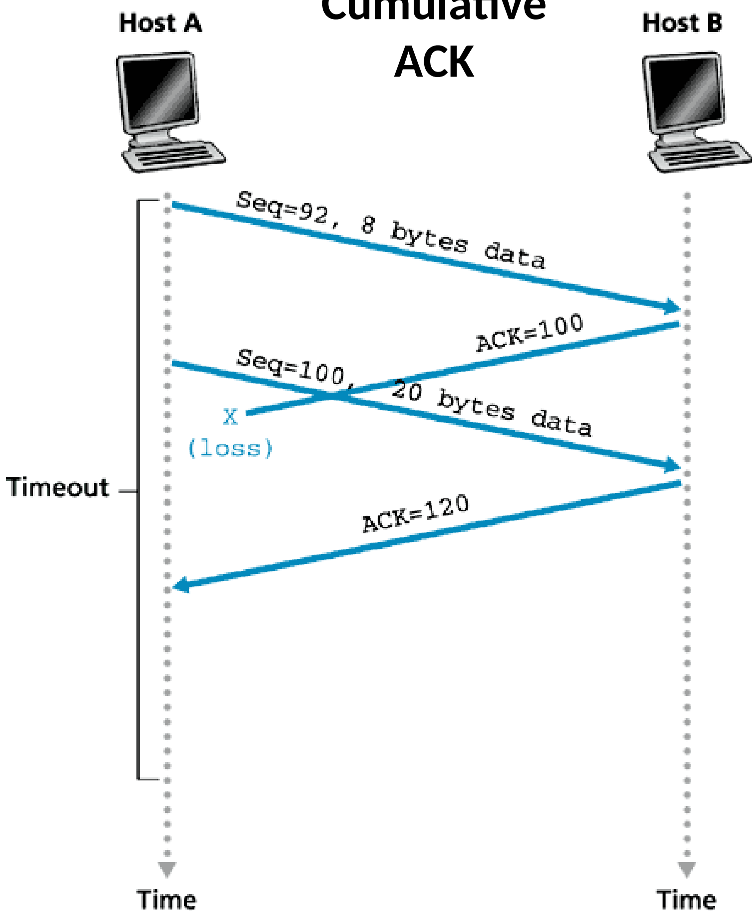
SendBase-1: last cumulatively ACKed byte
y > SendBase, new data is ACKed

Three major events

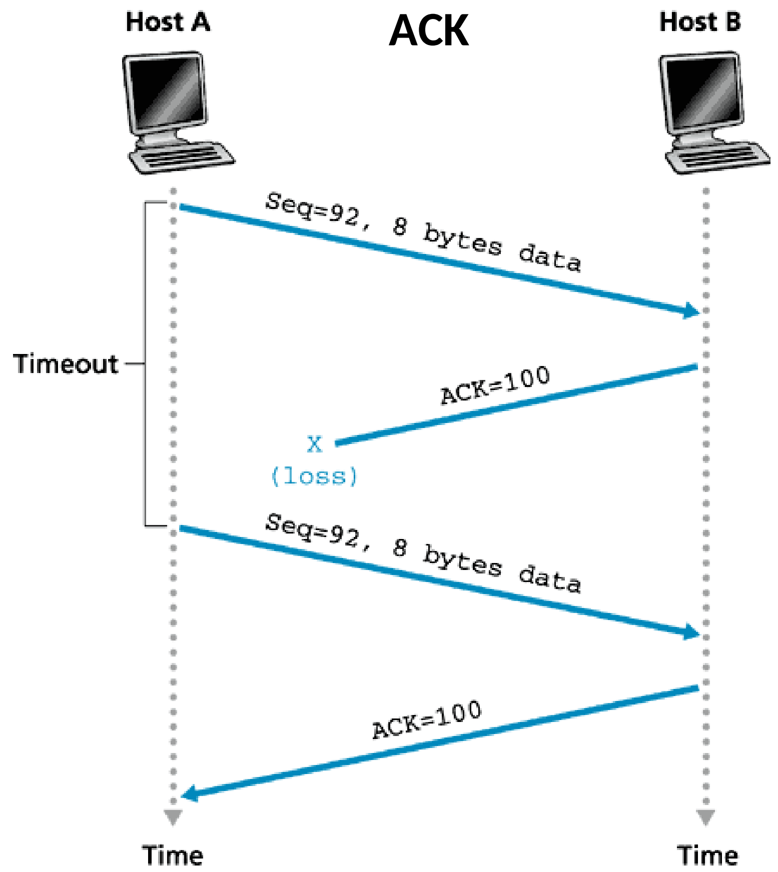
- Data receive from application
  - Create segment
  - Sequence number is byte-stream number of first data byte in segment
  - Start timer if not already running (timer tracks oldest unacked segment)
- Timer timeout
  - Retransmit segment that caused timeout
  - Restart timer
- ACK received
  - If acknowledges previously non ACKed segments
    - Update what is known to be ACKed
    - Start timer if there are outstanding segments

Connection-oriented transport
TCP retransmission scenarii (1/2)

Cumulative ACK

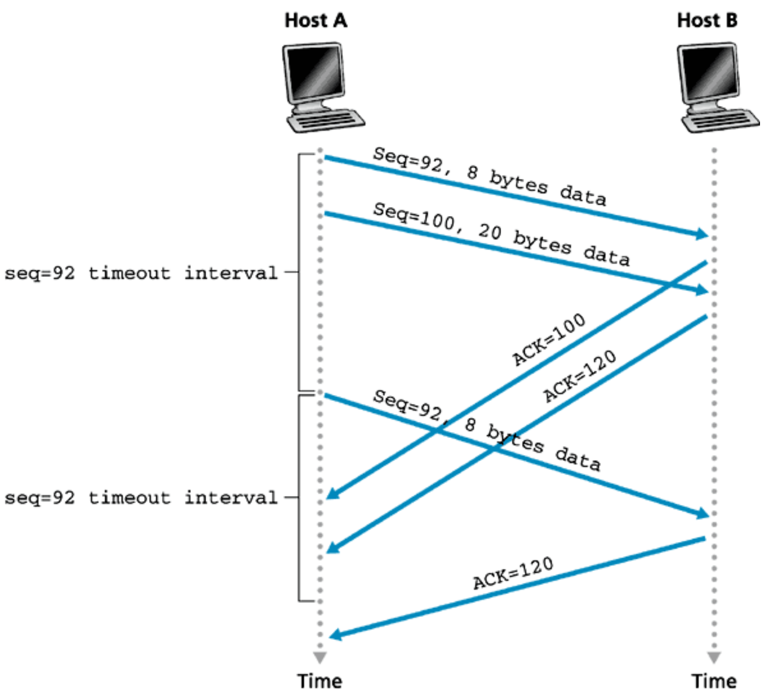


Lost ACK

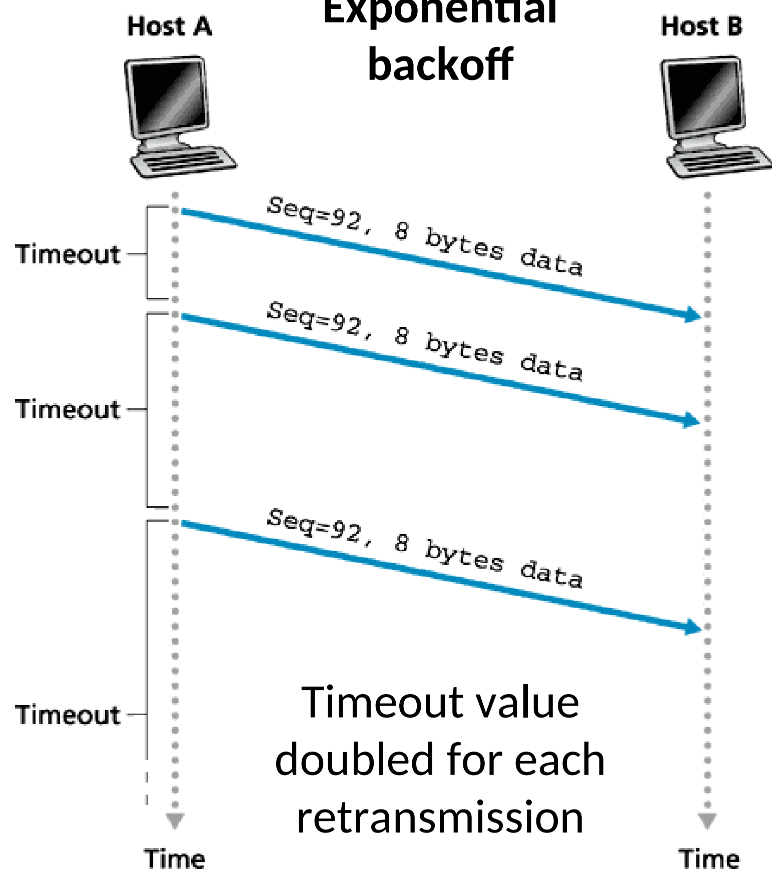


Connection-oriented transport
TCP retransmission scenarii (2/2)

Timeout



Exponential backoff



Timeout value doubled for each retransmission

Connection-oriented transport
TCP ACK generation

- As recommended in RFC 5681

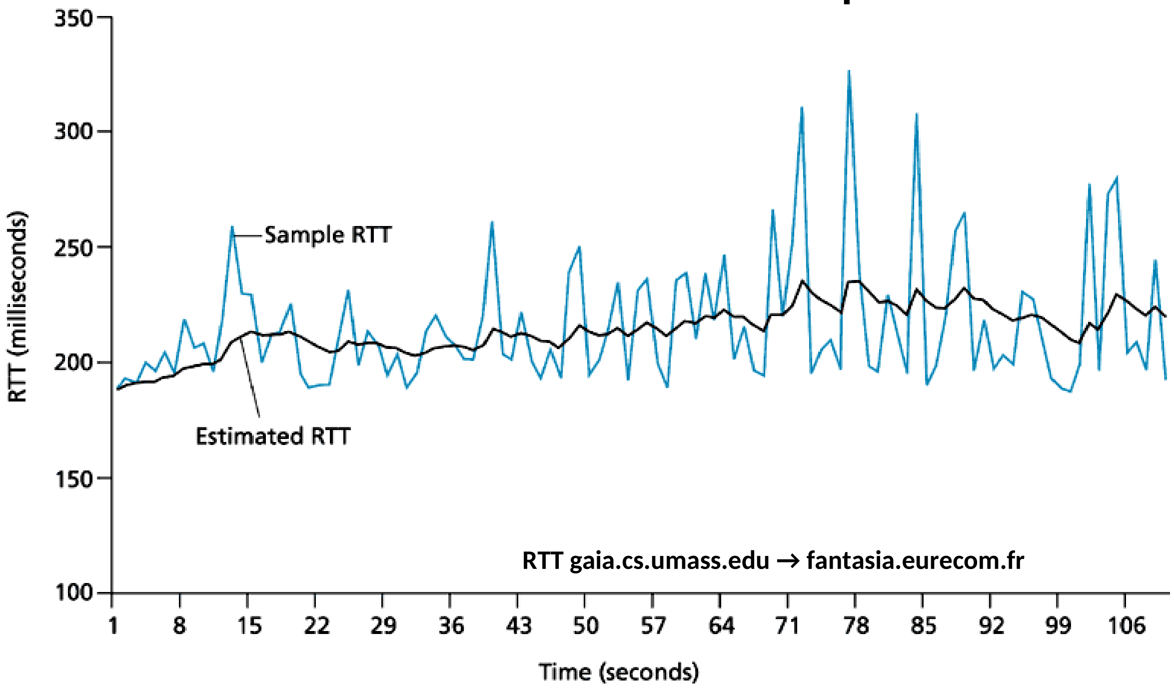
Event at receiver	TCP receiver action
Arrival of in-order segment with expected sequence number. All data up to expected sequence number already ACKed	Delayed ACK. Wait up to 500 ms for arrival of another in-order segment. If no arrival within this interval, send ACK.
Arrival of in-order segment with expected sequence number. All data up to expected sequence number already ACKed	Immediately send single cumulative ACK, ACKing both segments
Arrival of out-of-order segment with higher-than-expected sequence number. Gap detected	Immediately send duplicate ACK, indicating sequence number of next expected byte (lower end of gap)
Arrival of segment that partially or completely fills in gap in received data	Immediately send ACK, provided that segment starts at lower end of gap

Connection-oriented transport
Computing TCP timeout value

- Longer than RTT, but RTT varies
  - Too short
    - Premature timeout
    - Unnecessary retransmissions
  - Too long: slow reaction to segment loss
- Estimate RTT?
  - `SampleRTT`: measured time from segment transmission until ACK receipt, ignoring retransmissions (Karn’s algorithm)
  - `SampleRTT` will vary, want estimated RTT “smoother”
  - Average several recent measurements, not just current `SampleRTT`

Connection-oriented transport
RTT estimation

- Exponential weighted moving average
- EstimatedRTT(t) = (1 – α) * EstimatedRTT(t-1) + α * SampleRTT(t) ; α = 1/8



- Influence of past samples decreases exponentially

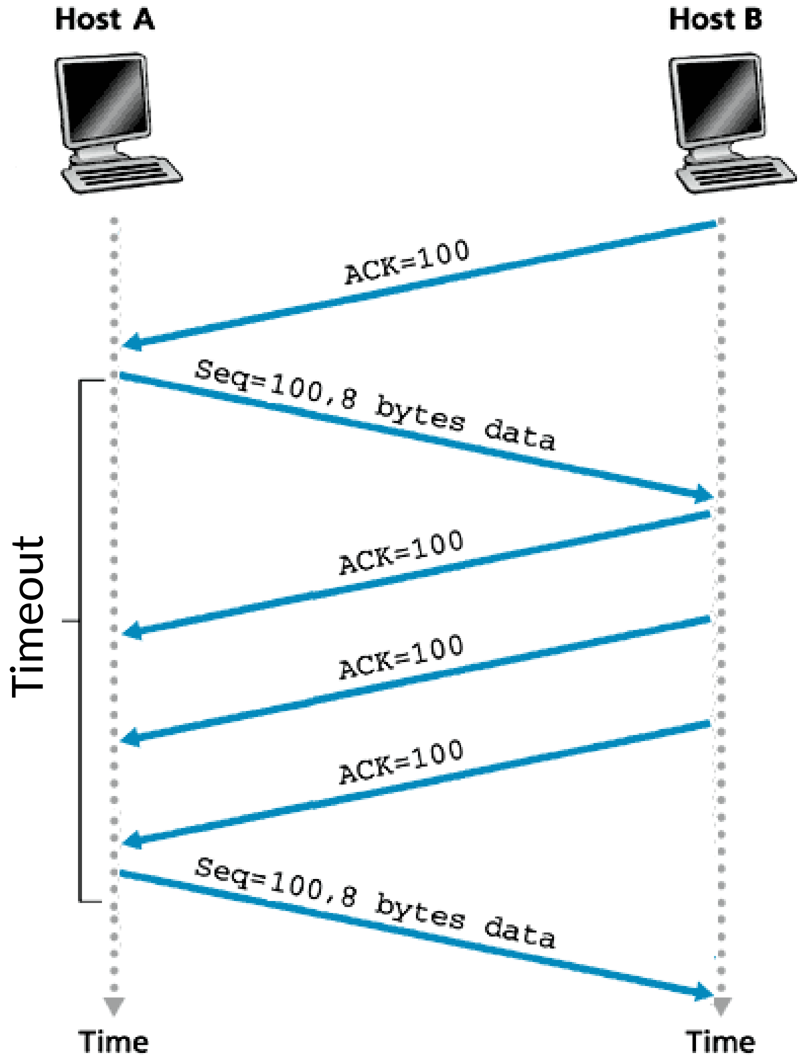
Connection-oriented transport
Setting timeout

- `EstimatedRTT` plus “safety margin”
- Large variation in `EstimatedRTT` → larger margin
- Inspiration from statistics
  - Mean = `EstimatedRTT`
  - Standard deviation = `DevRTT` where
    `DevRTT`(t) = (1-β) * `DevRTT`(t-1)
    + β * | `EstimatedRTT(t) – SampleRTT(t)` | ; β = 1/4
- Eventually,
  `TimeoutInterval = EstimatedRTT + 4 * DevRTT`
- Initially, `EstimateRTT` = 0s and `DevRTT` = 3s

Connection-oriented transport
Fast retransmit

- Time-out period often relatively long
- Long delay before resending lost packet
- Detect lost segments via duplicate ACKs
- If a single segment is lost among a burst, there will likely be many duplicate ACKs (dupACK).
- Fast retransmit
  - If sender receives 3 dupACKs for the same data, it supposes that segment after ACKed data was lost
  - Segment sent again before timer expires

Connection-oriented transport
Fast retransmit in working

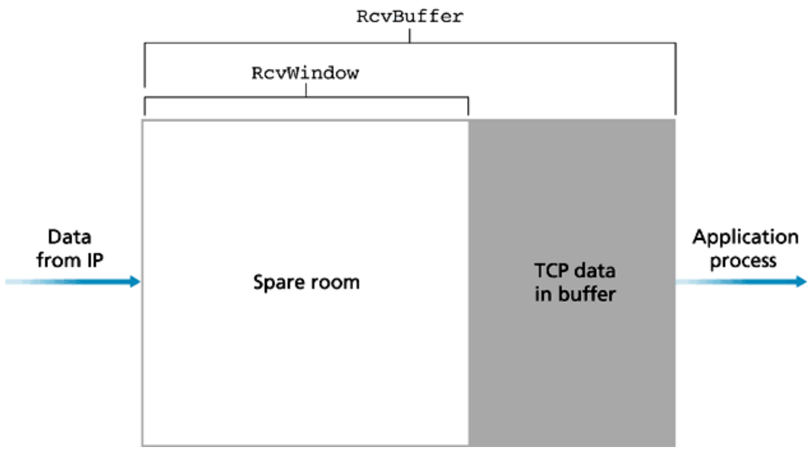


Connection-oriented transport
Fast retransmit algorithm

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
               SendBase = y
               if (there are currently not yet
                        acknowledged segments)
                  start timer
               }
          else { /* a duplicate ACK for already ACKed
                 segment */
               increment number of duplicate ACKs
                  received for y
               if (number of duplicate ACKs received
                  for y = 3) {
                  /* TCP fast retransmit */
                  resend segment with sequence number y
                  }
               break;
```

Connection-oriented transport
Flow control (1/2)

- Receive side of TCP connection has a receive buffer
- Application process may be slow at reading from buffer
- Speed-matching service: matching the send rate to the receiving application's drain rate



Flow control

Sender will not overflow receiver's buffer by transmitting too much, too fast

Connection-oriented transport
Flow control (2/2)

- Suppose out-of-order segments discarded
- Spare room in receive buffer (`RcvWindow`)

  = `RcvBuffer` - [`LastByteRcvd` – `LastByteRead`]

  - Receiver advertises spare room by including value of `RcvWindow` in segments
  - Sender limits non ACKed data to `RcvWindow`
  - Guarantees receive buffer does not overflow
- If receive buffer full (`RcvWindow = 0`)
  - Transmission suspended until non zero `RcvWindow` announced
  - Probe to avoid deadlock due to lack of updates

Outline

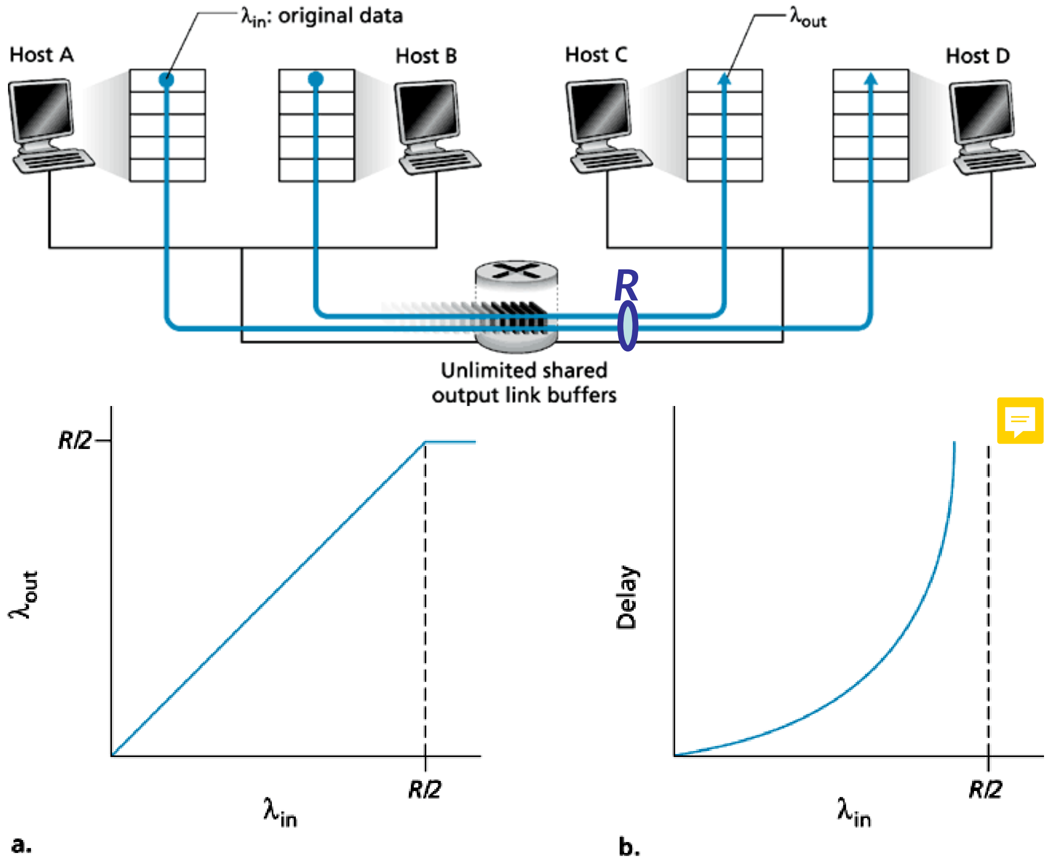
- Transport-Layer services
- Multiplexing and demultiplexing
- Connectionless transport – UDP
- Principles of reliable data transfer
- Connection-oriented transport – TCP
  - Segment structure
  - Connection management
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control
- Quick UDP Internet Connections – QUIC

Principles of congestion control

- Informal definition of congestion: “too many sources sending too much data too fast for network to handle”
- Different from flow control
- Manifestations
  - Lost packets (buffer overflow at routers)
  - Long delays (queueing in router buffers)

Principles of congestion control
Congestion scenario #1

- Two connections sharing a single hop with infinite buffers



No error recovery

No retransmission

No flow control

No congestion control

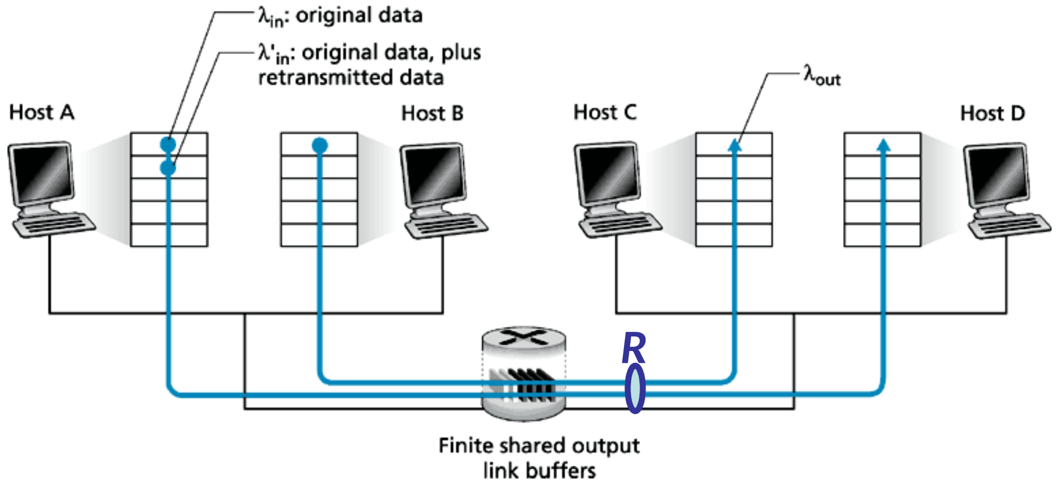
A and B competing for outgoing rate *R*

They achieve MAX throughput *R*/2

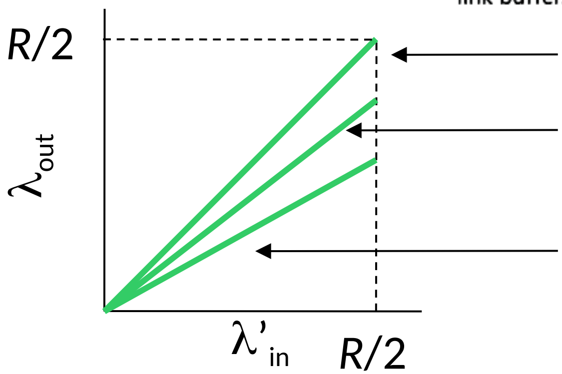
Impact: large queueing delays

Principles of congestion control
Congestion scenario #2

- Two reliable connections sharing a single hop with finite buffers



Error recovery
Retransmission
Packet loss
No congestion control
$\forall\ \lambda_{in} < \lambda'_{in}$



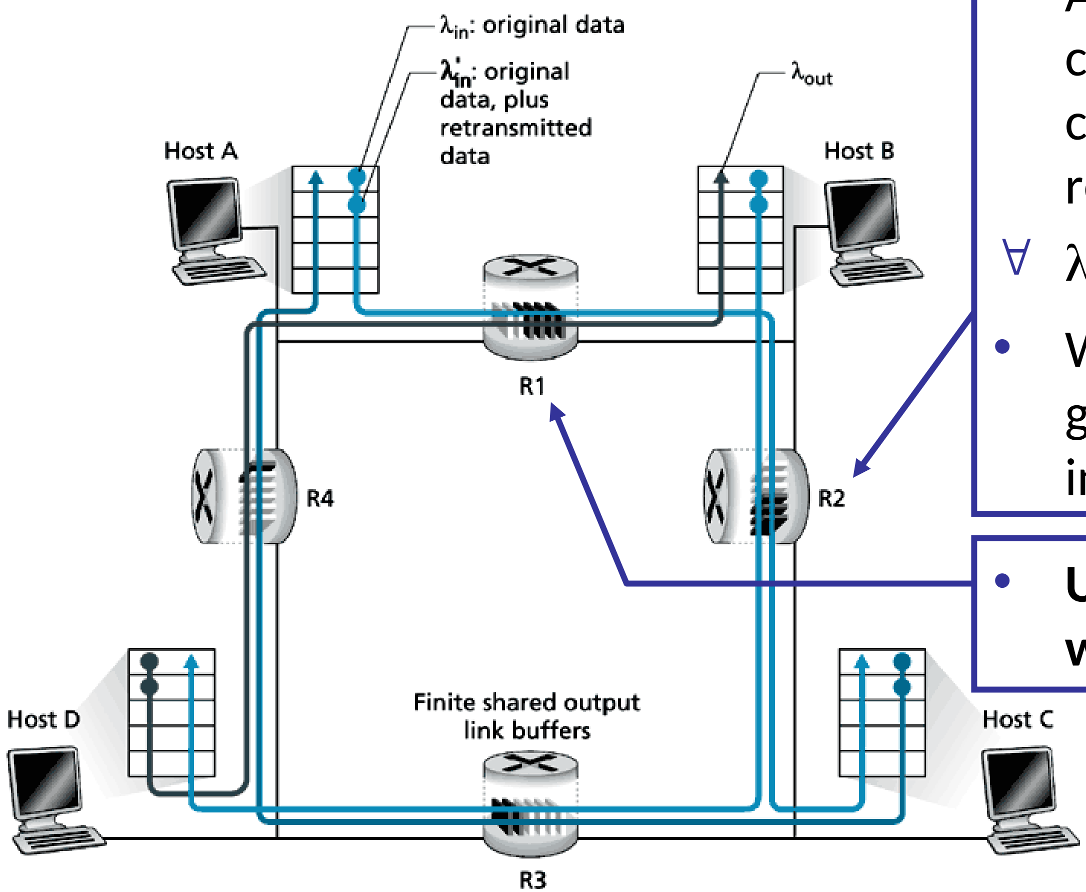
« Clever » sender: no retransmission

« Patient » sender: retransmission compensates for lost packets

« Normal » sender: delays cause **unnecessary retransmissions** of delayed packets

Principles of congestion control
Congestion scenario #3

- Multihop scenario



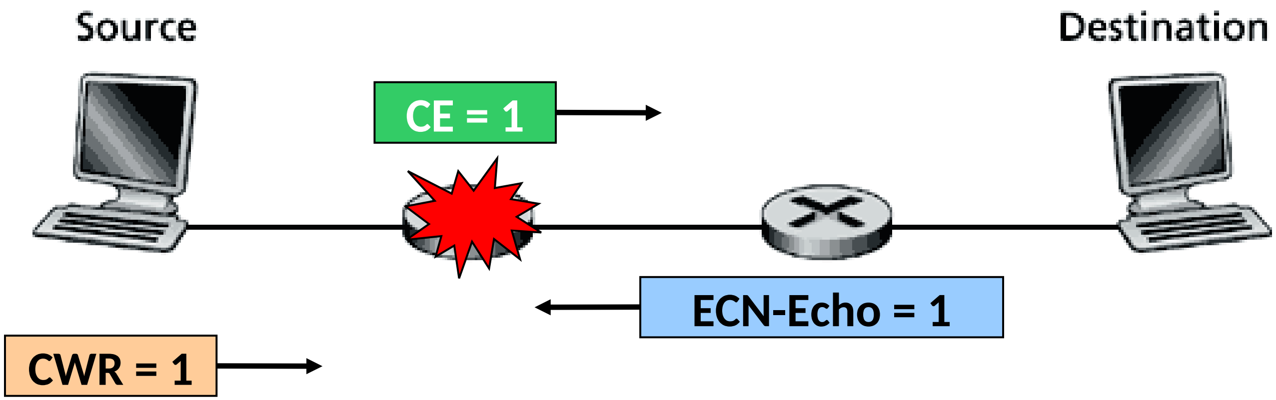
- A-C and B-D connections compete for R2 resources
- $\forall$ λ'_{in} >> λ_{out} <<
- Whenever buffer gets free, B-D fills it in, blocking A-C

- **Upstream resources wasted**

Principles of congestion control
Two main strategies

- Network-assisted congestion control
  - Routers provide feedback to end systems
  - Single bit indicating congestion (TCP/IP ECN, SNA, DECbit, ATM)
  - Explicit mention of the rate the sender should send at
  - Approach adopted by DataCenter TCP (DCTCP, 2010)
- End-end congestion control
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - Approach taken by standard TCP (1980’s)

Principles of congestion control
Explicit Congestion Notification (ECN, RFC 3168)



1. Congestion notification – Packet that caused congestion is tagged. CE bit of IP header set on (CE = Congestion Experienced, bit 7 of IPv4 TOS byte)
2. Informing receiver – Upon reception of CE = 1, inform sender by raising ECN-Echo flag (bit 9 in the Reserved field of the TCP header) in returning ACK until a PDU with CWR on is received (Congestion Window Reduced, bit 8 in the Reserved field of the TCP header).
3. Informing sender – Upon reception of ECN-Echo = 1, behave as if PDU lost, start congestion avoidance and raise CWR flag in next PDU.

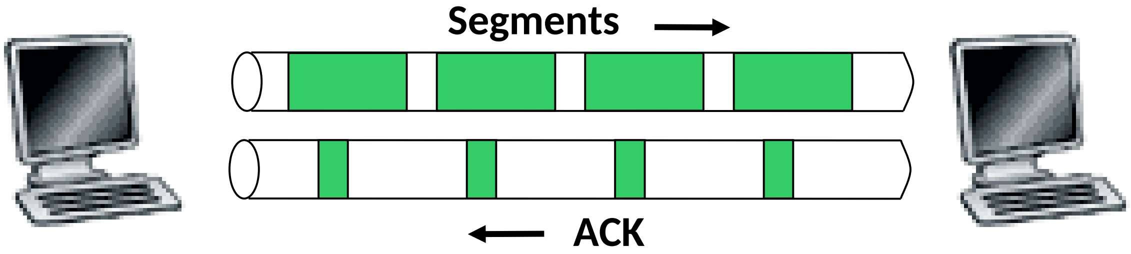
RFC under revision: draft-black-tsvwg-ecn-experimentation

Outline

- Transport-Layer services
- Multiplexing and demultiplexing
- Connectionless transport – UDP
- Principles of reliable data transfer
- Connection-oriented transport – TCP
    - Segment structure
    - Connection management
    - Reliable data transfer
    - Flow control
- Principles of congestion control
- TCP congestion control
- Quick UDP Internet Connections – QUIC

TCP Congestion Control
Bandwidth Delay Product (BDP)

- TCP Self-clocking : assuming that delayed ACKs are not used, at equilibrium, every arriving ACK indicates that a segment has left the network and triggers the transmission of a new segment



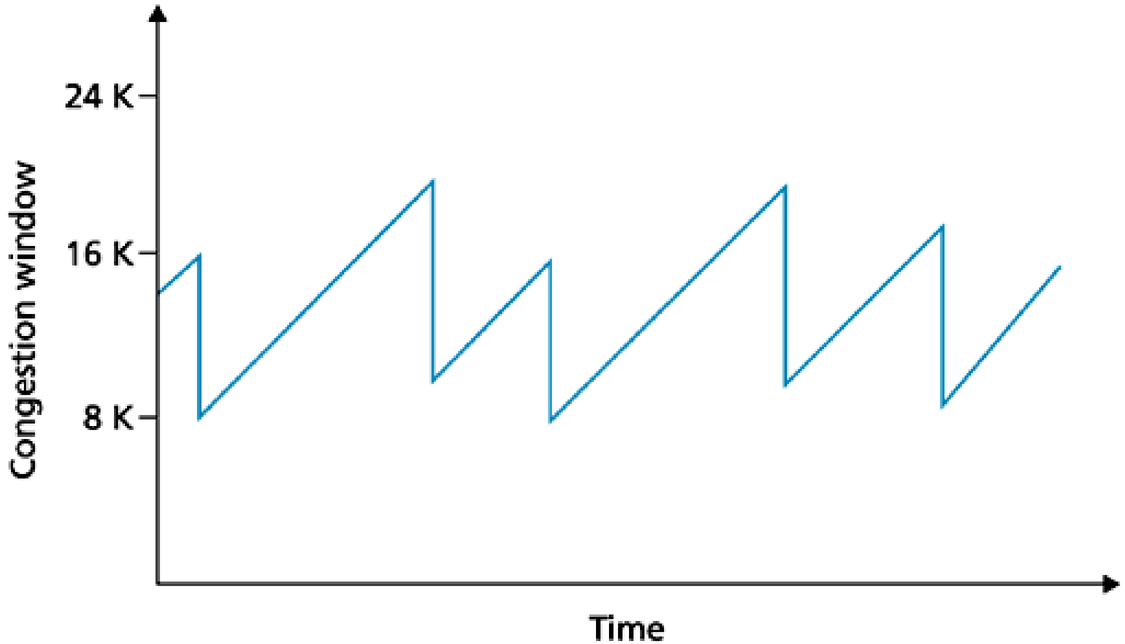
- Bandwidth Delay Product = capacity of the pipe

Access	Bandwidth	Delay	Product
Ethernet	100 Mbps	3 ms	293 kib = 36.6 kiB
Optical (trans-continental)	1 Gbps	60 ms	57.2 Mib = 7.2 MiB
Satellite	1.5 Mbps	500 ms	732.5 kib = 91.6 kiB

TCP Congestion Control
Mechanisms

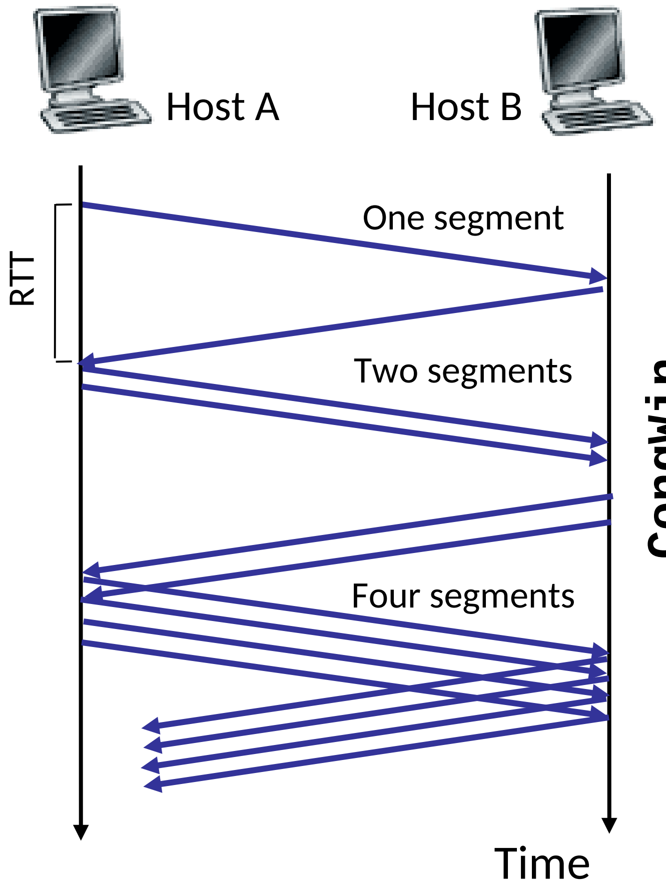
- E2E congestion control
- Additional variable: congestion window **`CongWin`**
  - Sender limits transmission
  - LastByteSent-LastByteAcked $\leq$ CongWin
  - Roughly, sender achieves CongWin/RTT
- How does sender perceive congestion?
  - Loss event = timeout or 3 duplicate ACKs
  - TCP sender reduces rate (CongWin) after loss event
- Three mechanisms
  - Additive-Increase, Multiplicative-Decrease (AIMD)
  - Slow start
  - Reaction to timeout events

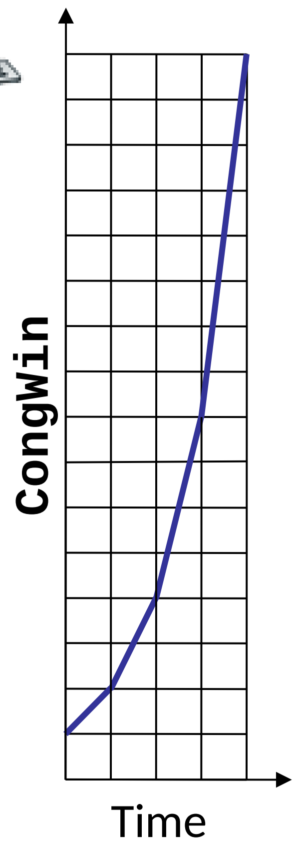
TCP Congestion Control
Additive Increase, Multiplicative Decrease (AIMD)



- Additive Increase
  - Increment **`CongWin`** every successful RTT
  - Linear increase a.k.a. Congestion Avoidance
- Multiplicative Decrease
  - Cut **`CongWin`** in half after loss

TCP Congestion Control
Slow Start





- When connection begins,
  - CongWin = 1 MSS
  - MSS = 500 Bytes
  - RTT = 200 ms

  → Initial rate = 20 kbps
- Available bandwidth may be >> MSS/RTT
- Slow Start: increase rate exponentially fast until first loss event
- Initial CongWin :
  - 2 to 4 MSS (RFC 3390)
  - 10 MSS (RFC 6928)

TCP Congestion Control

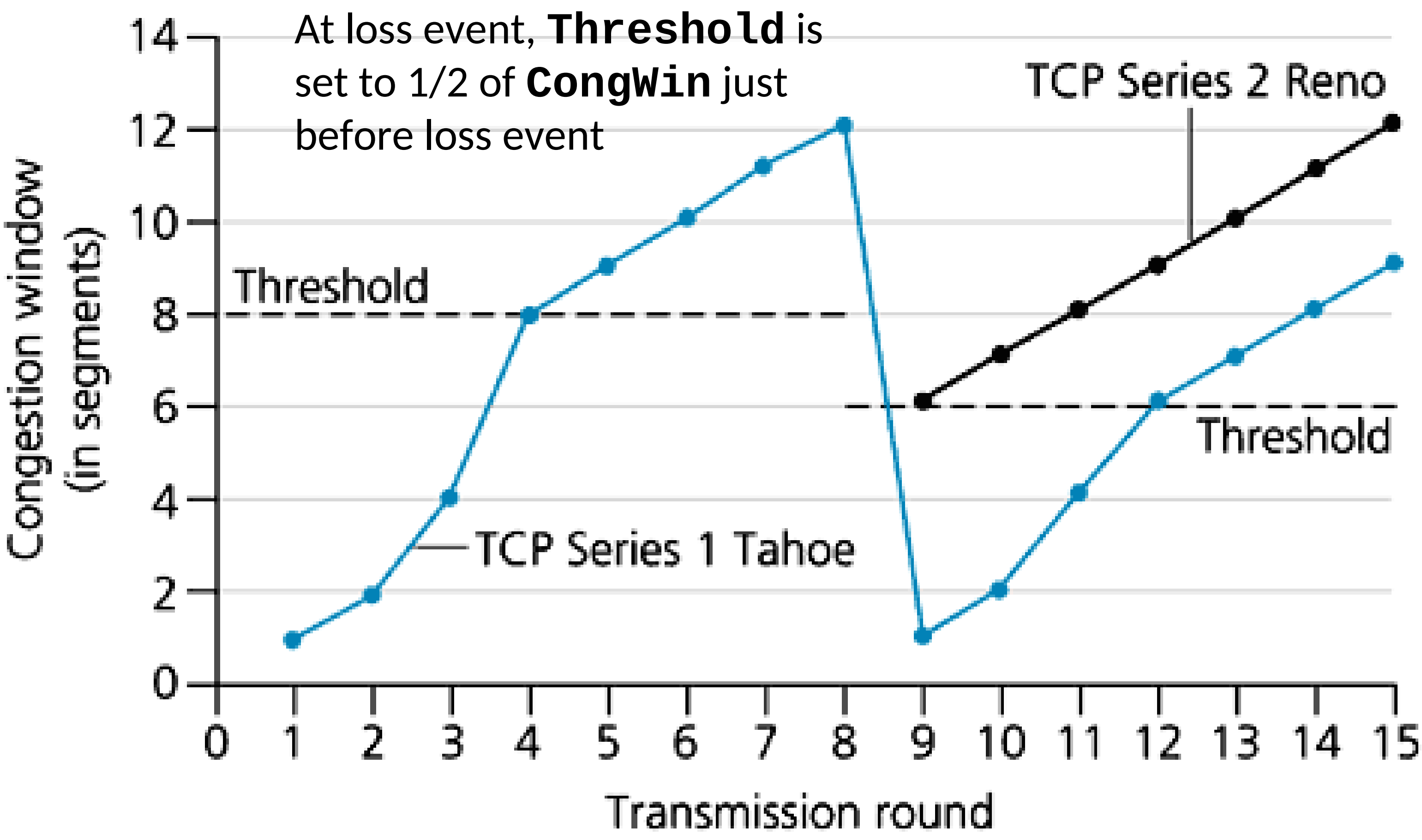
Reaction to timeout events

- TCP congestion control reacts differently to a loss event detected via three duplicate ACKs than to a loss event detected via timeout
  - Three duplicate ACKs indicates network capable of delivering some segments
  - Timeout before three duplicate ACKs is “more alarming”
- After timeout event
  - `CongWin` instead set to 1 MSS
  - Window then grows exponentially (Slow Start)
  - To a threshold (`sstresh`), then grows linearly
- After three dupACKs
  - `CongWin` is cut in half
  - Window then grows linearly

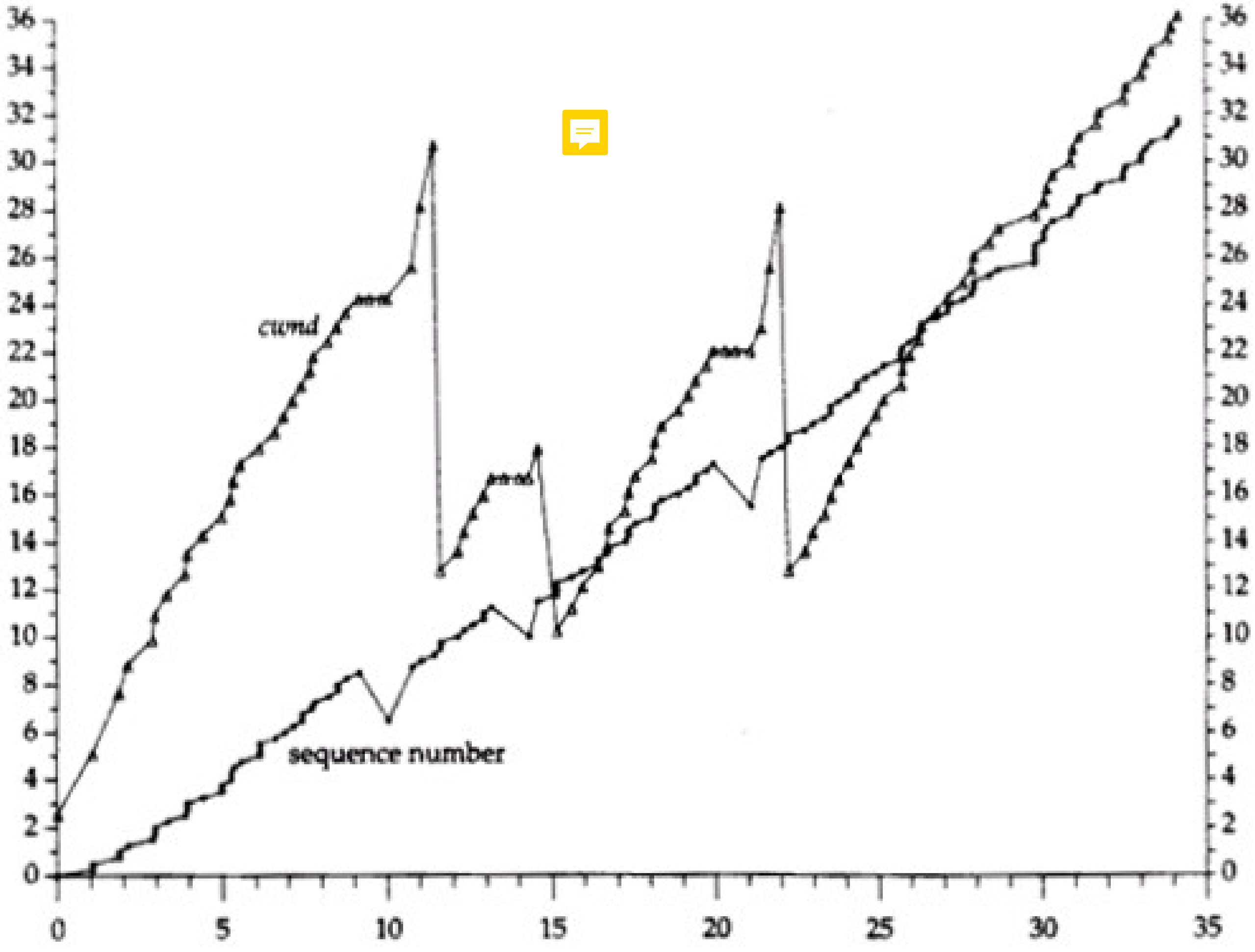
TCP Congestion Control
Summary

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

TCP Congestion Control
Implementation (1/3)

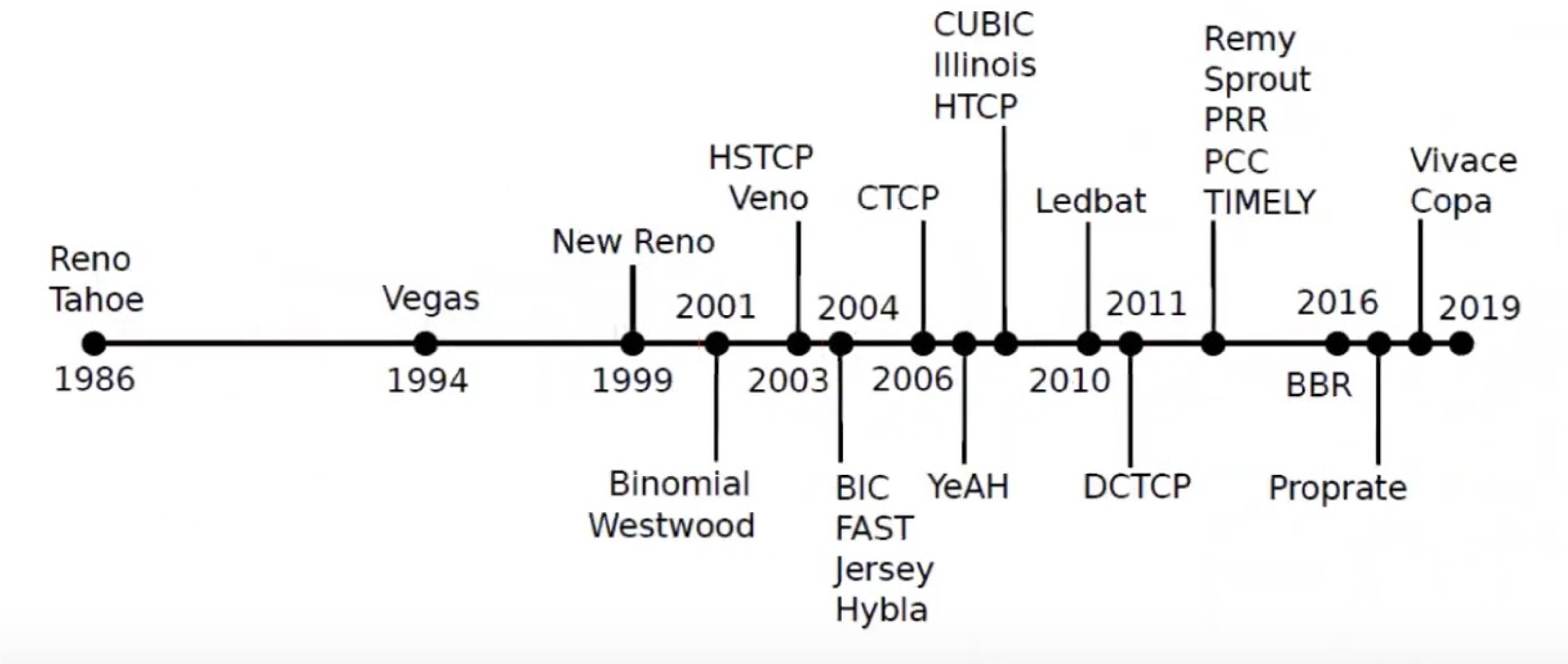


TCP Congestion Control
Implementation (2/3)



Stevens, R., « TCP/IP Illustrated » vol.1, p. 315, 1995

TCP Congestion Control
Implementation (3/3)

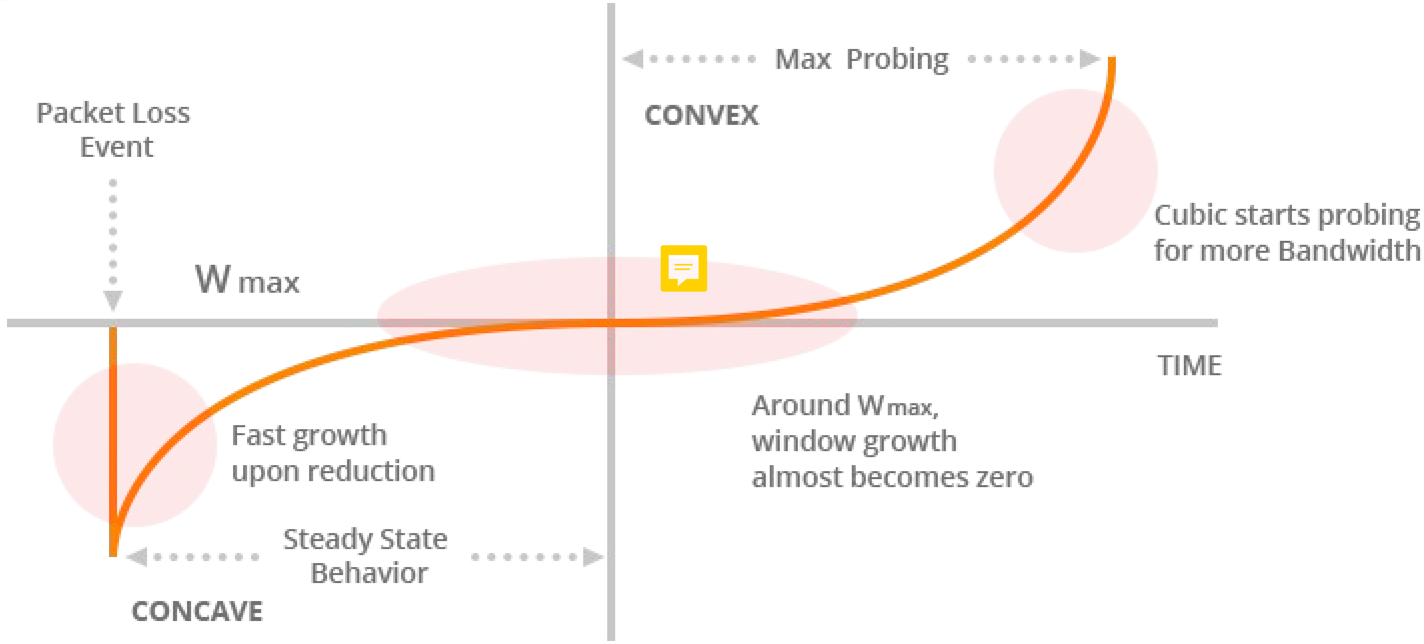


Source : the Great TCP Congestion Control Census, ACM SIGMETRICS 2020
https://www.youtube.com/watch?v=oImBLTue6So

TCP Congestion Control

CUBIC Congestion Control (RFC 8312, February 2020)

- Low utilisation of large BDP networks due to linear increase function of standard TCP congestion control schemes
- Cubic increase function instead of linear
- Part of Linux kernel since 2006, MacOS since 2014 and Windows 10 since 2017

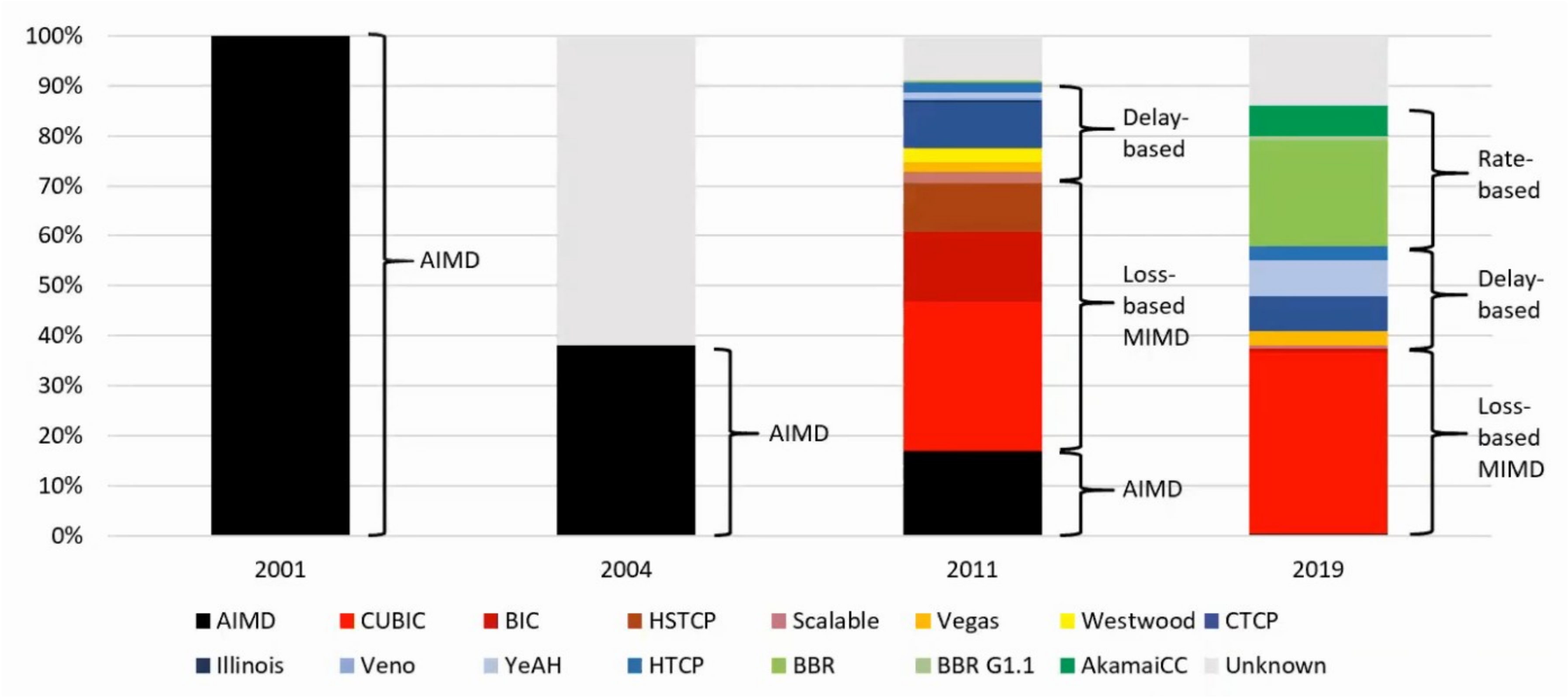


Source : https://www.noction.com/wp-content/uploads/2018/02/CUBIC-Function-with-Concave-and-Convex-Profiles.png

TCP Congestion Control
Bottleneck Bandwidth and Round-trip (BBR)

- Loss-based congestion control no longer relevant: packet loss no longer means congestion
- A connection runs with the highest throughput and lowest delay when (rate balance) the bottleneck packet arrival rate equals $BtlBw$ and (full pipe) the total data in flight is equal to the BDP (= $BtlBw \times Rtprop$)
- BBR estimates bottleneck bandwidth $BtlBw$ and propagation delay $Rtprop$ with ACKs
- Introduced by Google in 2016 (BBRv1) ; BBRv2 on its way
- Part of Linux kernel and QUIC
- Source: https://queue.acm.org/detail.cfm?id=3022184

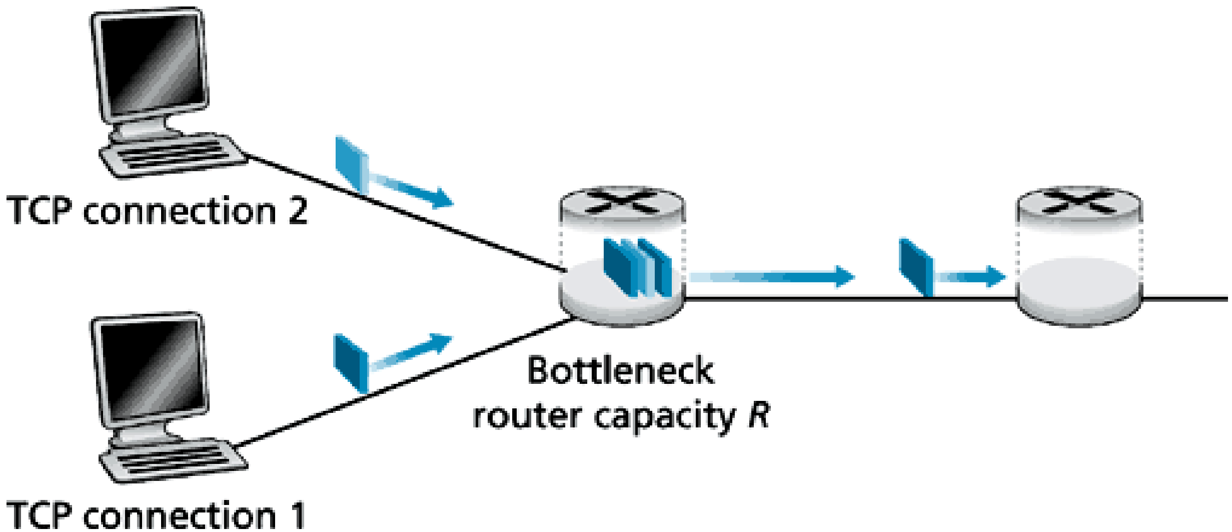
TCP Congestion Control
Evolution of TCP Ecosystem



Source : the Great TCP Congestion Control Census, ACM SIGMETRICS 2020
https://www.youtube.com/watch?v=oImBLTue6So

TCP Congestion Control
Fairness

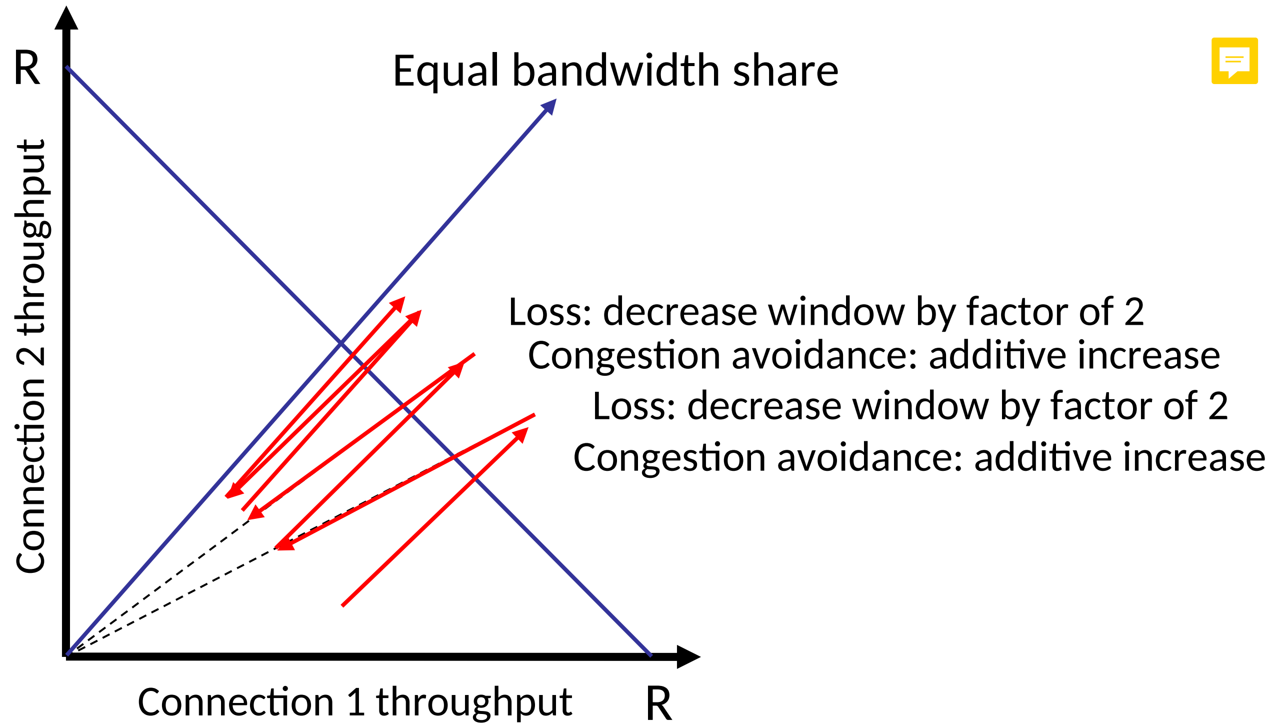
- Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R / K*



- Is AIMD fair, given TCP connections start at different times and may thus have different window sizes at a given instant?

TCP Congestion Control
Fairness explanation

- Two competing sessions
  - Additive Increase gives slope of 1, as throuhgputs increase
  - Multiplicative decrease reduces throughputs proportionally



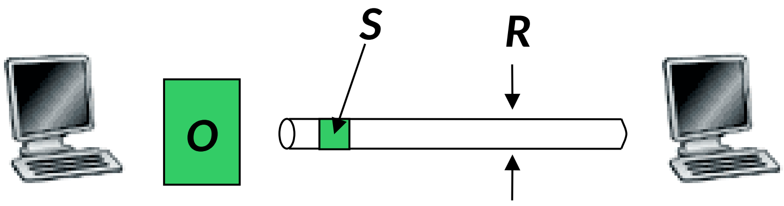
TCP Congestion Control

UDP vs. TCP on fairness

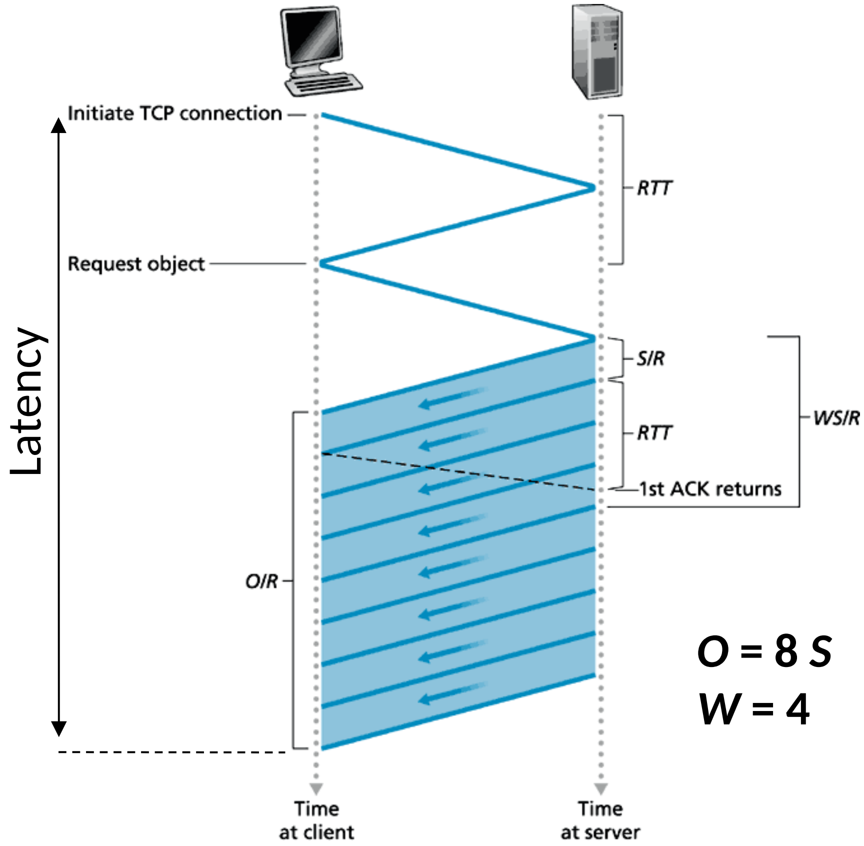
Fairness with UDP	Fairness with TCP
Some applications do not use TCP Do not want rate throttled by congestion control Instead use UDP Pump data at constant rate Must tolerate packet loss, and recover it in Application Layer Mitigation: DCCP, TCP Friendly Rate Control (TFRC, RFC 5348)	Nothing prevents application from opening parallel connections between two hosts. Web browsers do this Example: link of rate R supporting 9 connections New application asks for 1 TCP, gets rate R/10 New application asks for 9 TCPs, gets R/2

TCP Congestion Control
Modelling latency

- How long does it take to receive an object from a Web server after sending a request (latency)?
- Ignoring congestion, delay is influenced by
  - TCP connection establishment
  - Data transmission delay
  - Slow Start
- Assume
  - One link of rate *R*
  - MSS of size *S* (bits)
  - Object of size *O* (bits)
  - No retransmissions (no loss, no corruption)
  - Fixed `CongWin` size of *W* segments



TCP Congestion Control
Latency – Static `CongWin` (1/2)

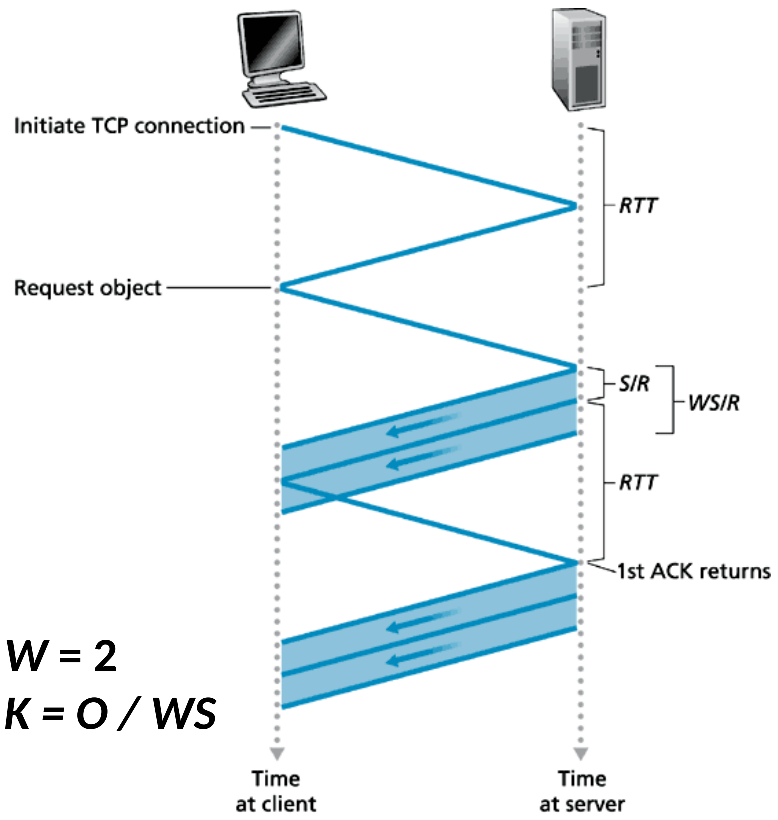


$O = 8\,S$
$W = 4$

- First case: WS/R > RTT + S/R
- ACK for first segment in window returns before window's worth of data sent
- Delay = 2 RTT + O / R

TCP Congestion Control

Latency – Static `CongWin` (2/2)



- Second case: WS/R < RTT + S/R
- ACK for first segment in window returns after window's worth of data sent
- Sender can be idle
- Delay =

2 RTT **Connection set-up**

+ O/R **Transmission delay**

+ (K-1) [(S/R + RTT) – WS/R] **Server idle**

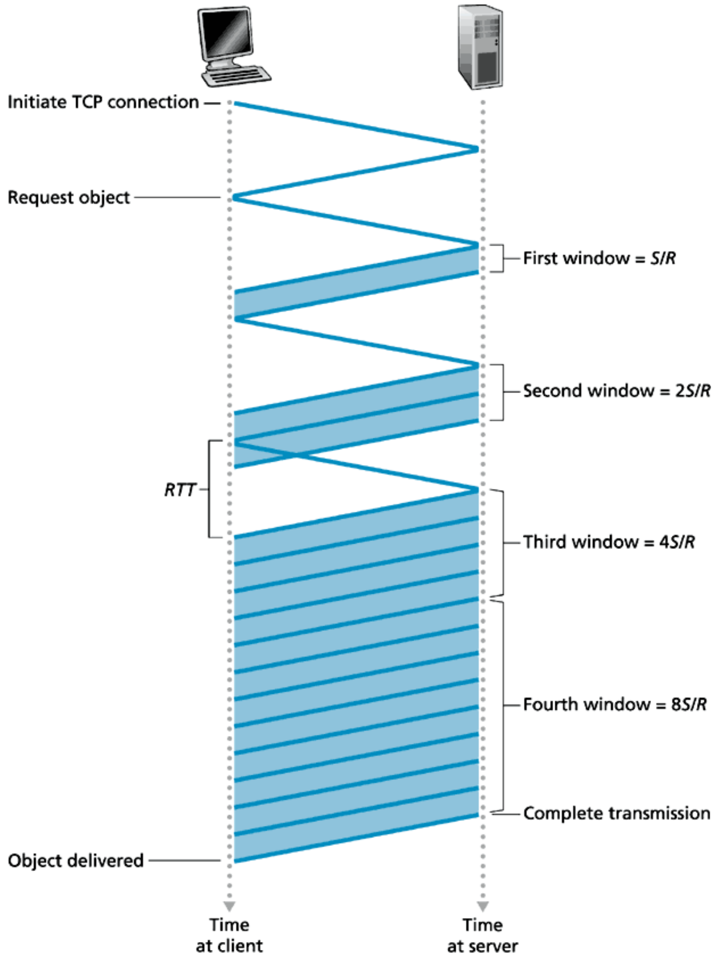
TCP Congestion Control
Latency – Dynamic `CongWin` (1/2)

- Window grows according to slow start
- The delay for one object is

$$\text{Latency} = 2\ \text{RTT} + \frac{O}{R} + P\left(\text{RTT} + \frac{S}{R}\right) - \left(2^P - 1\right)\frac{S}{R}$$

- where
  - K is the number of windows that cover the object
  - Q is the number of times the server idles if the object were of infinite size
  - P = min {Q, K-1}, is the number of times TCP idles at server

TCP Congestion Control
Latency – Dynamic `CongWin` (2/2)



- Delay components
  - 2 RTT for connection establishment and request
  - O/R to transmit object
  - Time server idles due to Slow Start
- Example
  - O/S = 15 segments
  - K = 4 dynamic windows

    (1 + 2 + 4 + 8 = 15)
  - Q = 2
  - P = min{Q, K-1} = 2
  - Server idles P = 2 times

TCP Congestion Control

Latency – Comparison

- Compare latencies with/without congestion control

$$\frac{\text{Latency}}{\text{Minimum latency}} \leq 1 + \frac{P}{\left\lceil \frac{O/R}{\text{RTT}} \right\rceil + 2}$$

- If RTT << O / R, TCP slow start does not significantly increase latency
- Slow start can significantly increase latency if
  - RTT is relatively large
  - Object size is relatively small

A Roadmap to TCP Specification Docs
RFC 6247 – Basic functionalities

TCP header parsing		RFC 793, RFC 2460, RFC 2873
State machine		RFC 793
Congestion control	Tahoe – Slow Start, Congestion Avoidance, Fast Retransmit	RFC 2581
	Reno – Fast Recovery	
RTO computation		RFC 1122, RFC 2988

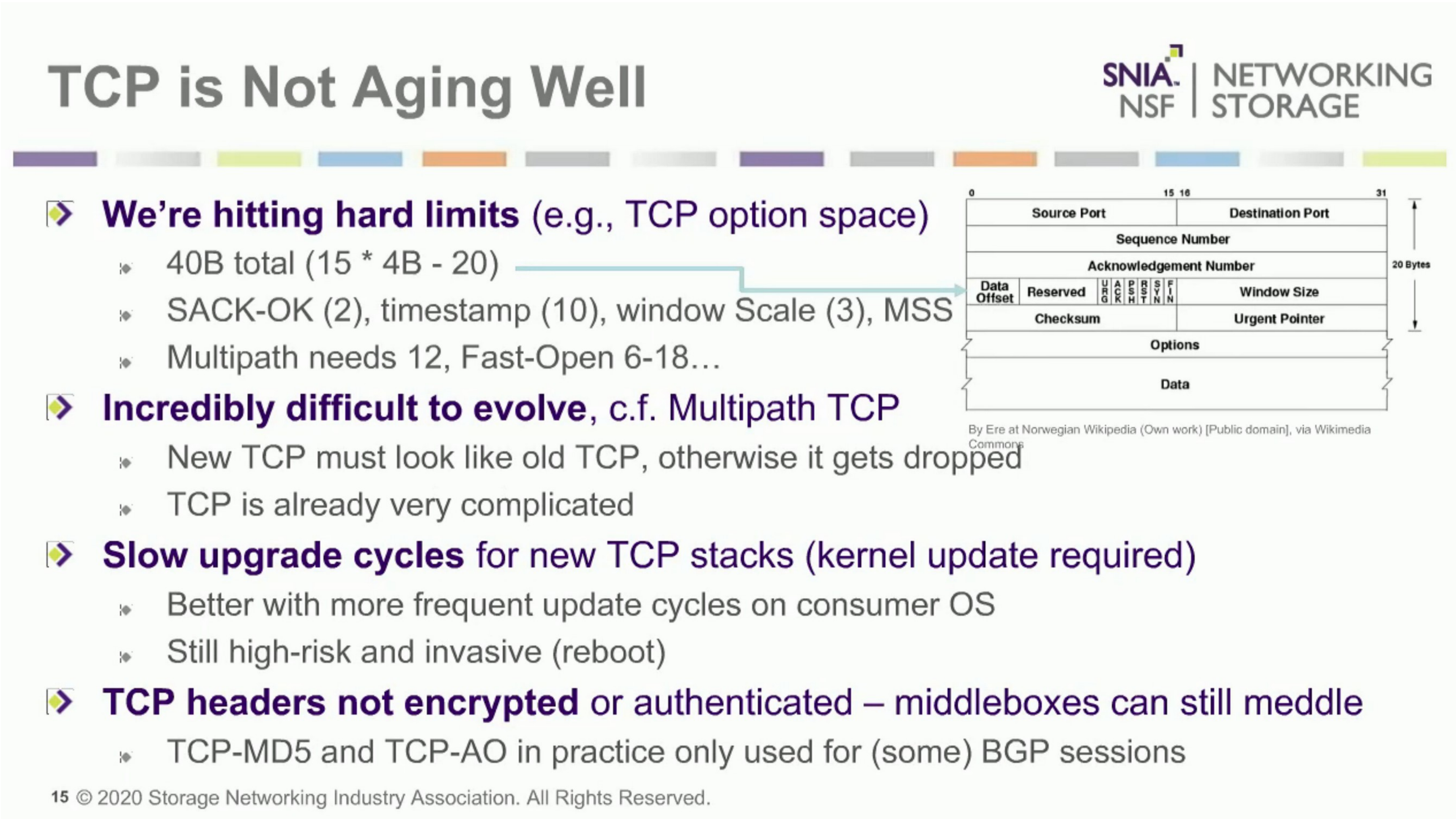
A Roadmap to TCP Specification Docs
RFC 6247 – Standard enhancements

Fundamental changes	Extensions for high performance (large bandwidth-delay product paths)	RFC 1323
	Explicit Congestion Notification (ECN)	RFC 3168, RFC 3540
Congestion Control and Loss Recovery with cumulative ACKs	NewReno – Partial ACKs in Fast Recovery	RFC 3782
	Eifel + F-RTO – Handling spurious time-outs	RFC 3522, RFC 3708, RFC 4015
	Increased initial window	RFC 3390
	Limited transmit	RFC 3042
SACK-based congestion control and loss recovery		RFC 2018, RFC 2883, RFC 3517
Dealing with forged segments		RFC 1948, RFC 2385

A Roadmap to TCP Specification Docs
RFC 6247 – Experimental extensions

Sharing information among TCP connections for congestion control purposes	RFC 2140, RFC 3124
Performance Enhancing Proxies (PEP) – Split connection protocols (I-TCP, M-TCP, MTCP)	RFC 3135
TCP-Friendly Rate Control (TFRC) – Smoother congestion control	RFC 3448
Appropriate Byte Counting (ABC)	RFC 3465
High-Speed TCP	RFC 3649
Duplicate feed-back to avoid spurious retransmissions	RFC 3708
Limited Slow Start	RFC 3742
Forward Acknowledgement (FACK) – Improved TCP SACK	-
Vegas – RTT-sensitive congestion control	-
Veno – Mix of Vegas and Reno	-
Westwood - Sender-side-only modification to NewReno in order to better handle large bandwidth-delay product paths	-

TCP getting old...



Outline

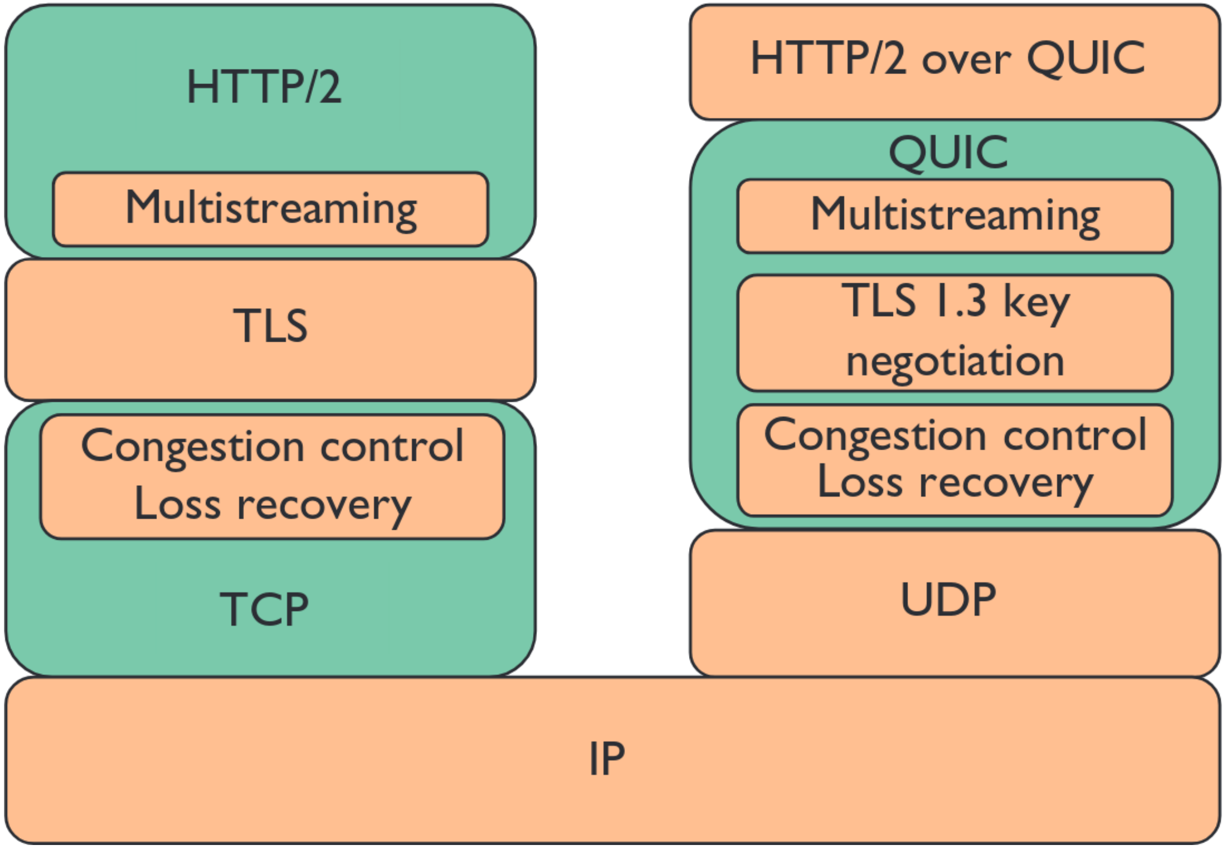
- Transport-Layer services
- Multiplexing and demultiplexing
- Connectionless transport – UDP
- Principles of reliable data transfer
- Connection-oriented transport – TCP
    - Segment structure
    - Connection management
    - Reliable data transfer
    - Flow control
- Principles of congestion control
- TCP congestion control
- Quick UDP Internet Connections – QUIC

QUIC
draft-ietf-quic-transport-31.txt, September 2020

Key mechanisms and benefits

- Low-latency connection establishment
  - No transport handshake (UDP)
  - Session resumption for encryption (TLS)
- Multiplexing without head-of-line blocking
- Authenticated and encrypted header and payload
- Rich signaling for congestion control and loss recovery
- Stream and connection flow control
- Connection migration and resilience to NAT rebinding
- Version negotiation

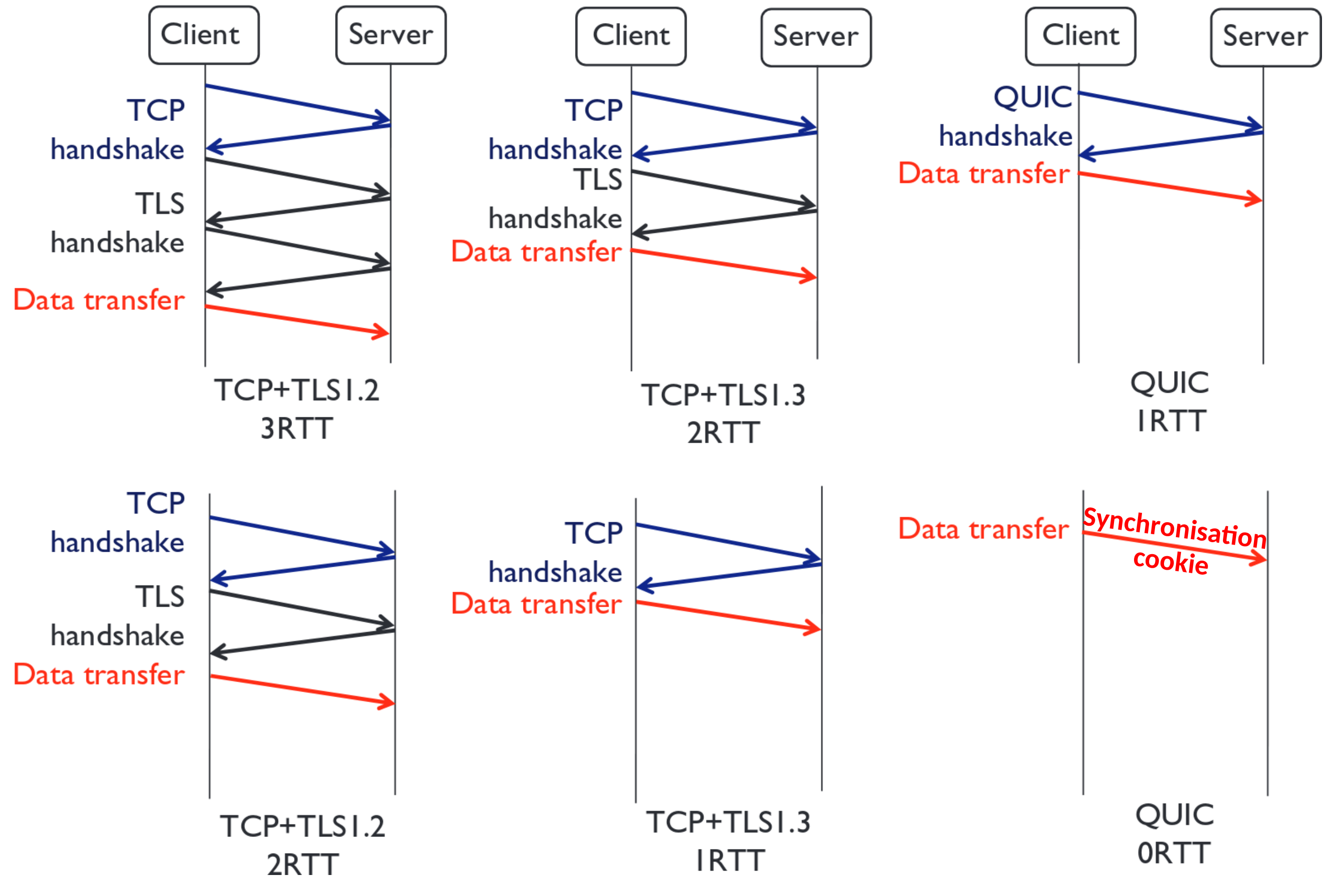
QUIC
Architecture



Source: Innovating Transport with QUIC: Design Approaches and Research Challenges, IEEE Internet Computing, Mar.-Apr. 2017

QUIC
Reduced latency

First connection



Source: Innovating Transport with QUIC: Design Approaches and Research Challenges, IEEE Internet Computing, Mar.-Apr. 2017

Summary

- Principles behind transport layer services:
  - Multiplexing and demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- Implementation in the Internet
  - UDP
  - TCP
- Next
  - Leaving the network “edge” (application, transport layers)
  - Into the network “core”

Review questions

- Describe why an application developer may choose to run an application over UDP rather than over TCP
- Is it possible for an application to enjoy reliable data transfer when the application runs over UDP? If so, how?
- Suppose host A sends two TCP segments to host B over TCP. The first segment has sequence number 90, the second has sequence number 110
    - How much data is in the first segment?
    - Suppose the first segment is lost but the second segment arrives at B. In the ACK that B sends to A, what will be the ACK number?
- Consider congestion control in TCP. When the timer expires at the sender, how does `sstresh` evolve?