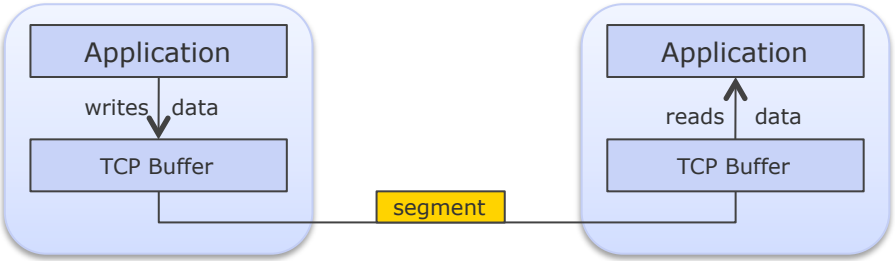
CONTENT of this CHAPTER

- Fundamental Goals of the Transport Layer
- Concepts for the Transport Layer
- Standard Transport Layer Protocols
  - UDP
  - TCP
  - MPTCP
  - SCTP
  - DCCC

Transmission Control Protocol (TCP): Overview

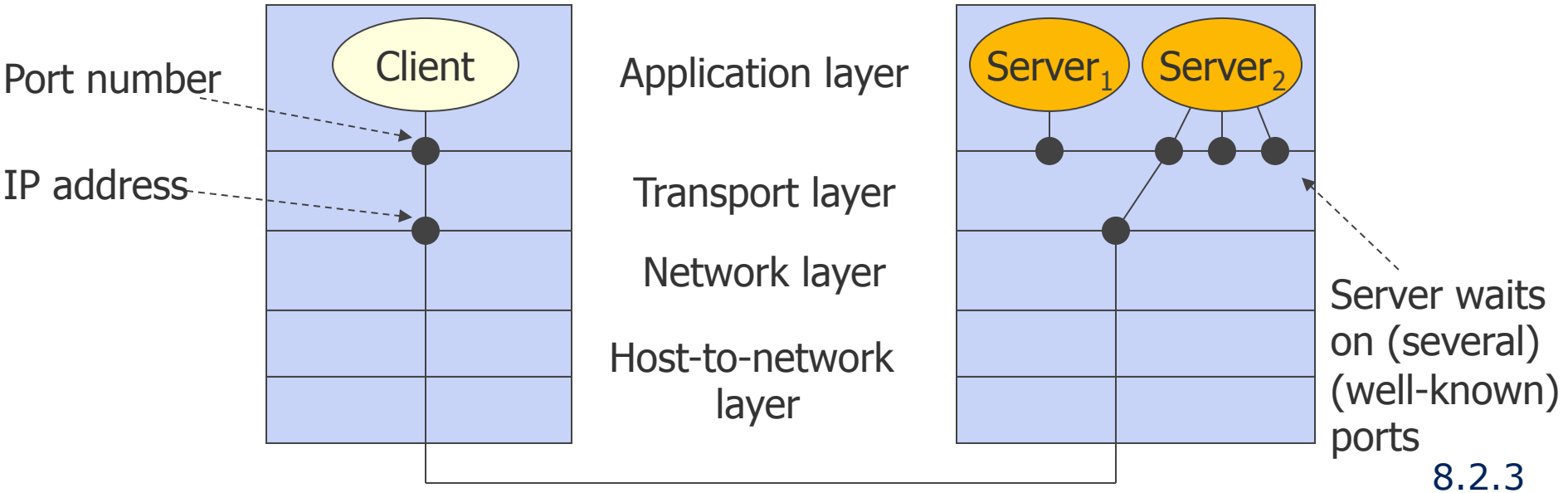
- TCP is specified in RFC 793 (published in 1981)
  - Complemented by others since, e.g. RFCs 1122, 1323, 2018, 5681
- Point-to-point
  - One sender, one receiver
- Full duplex
  - Bi-directional flow in same connection
- Send & receive buffers
  - Data=byte stream, split into "segments"
  - MSS: maximum segment size 64 KByte



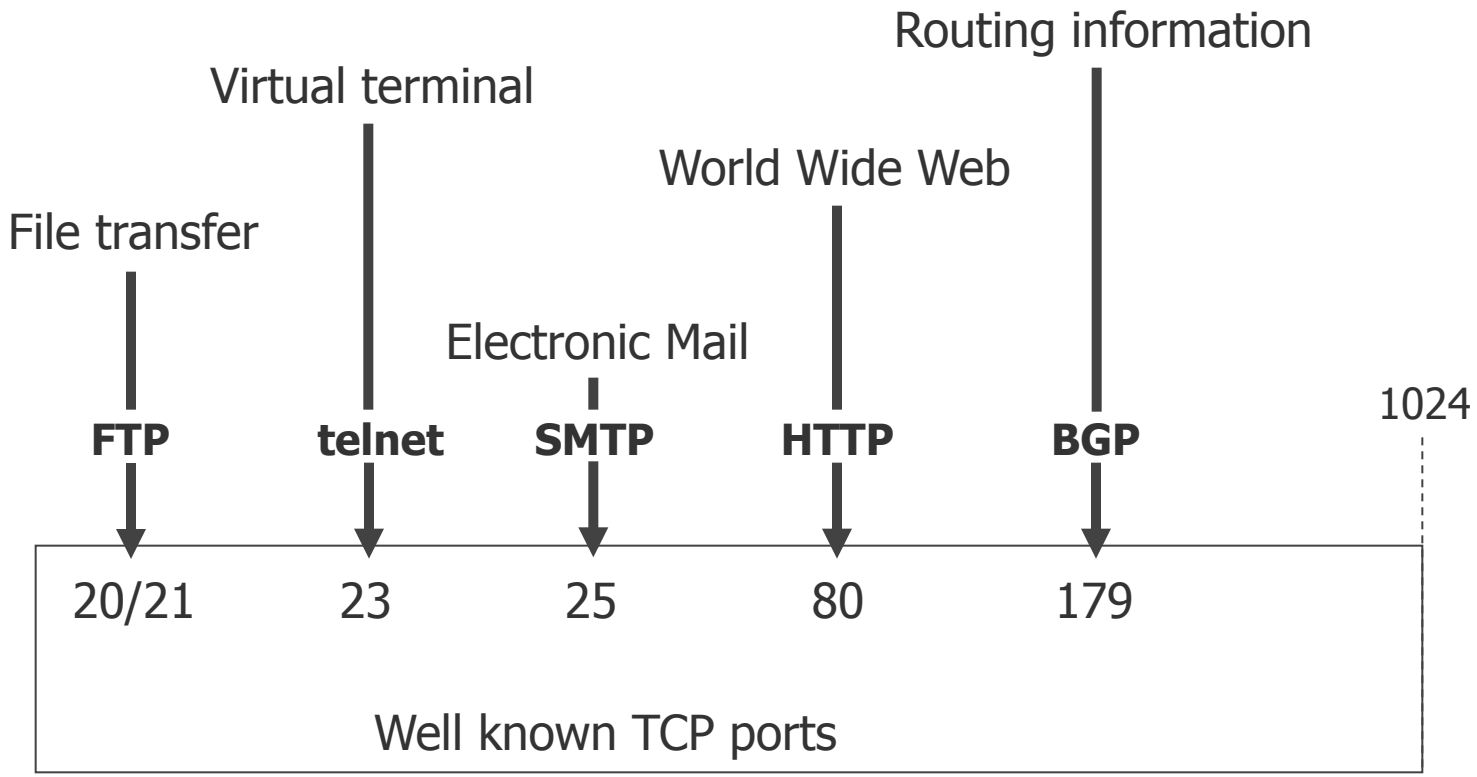
- Connection-oriented
  - Handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- Reliable delivery of segments
- Segments delivered in order
- Flow controlled
  - Sender will not overwhelm receiver
- Congestion controlled
  - Sender will not overwhelm network
- Pipelined
  - Sliding window: TCP congestion and flow control set window size
- "Limited" QoS

Transmission Control Protocol (TCP): Overview

- TCP sends and receives segments to realize:
  - Connection establishment
  - Agreement on a window size
  - Data transmission
  - Sending of confirmations
  - Connection termination
  - "Urgent" messages outside of flow control
- For an application, sockets are the access points to the network, identified by IP address/port number tuple: (IP Address$_1$, Port$_1$, IP Address$_2$, Port$_2$)



Examples of TCP-based Applications



- Streaming (proprietary protocols) use more and more TCP
  - e.g. YouTube started using TCP a few years ago

Content of this Section

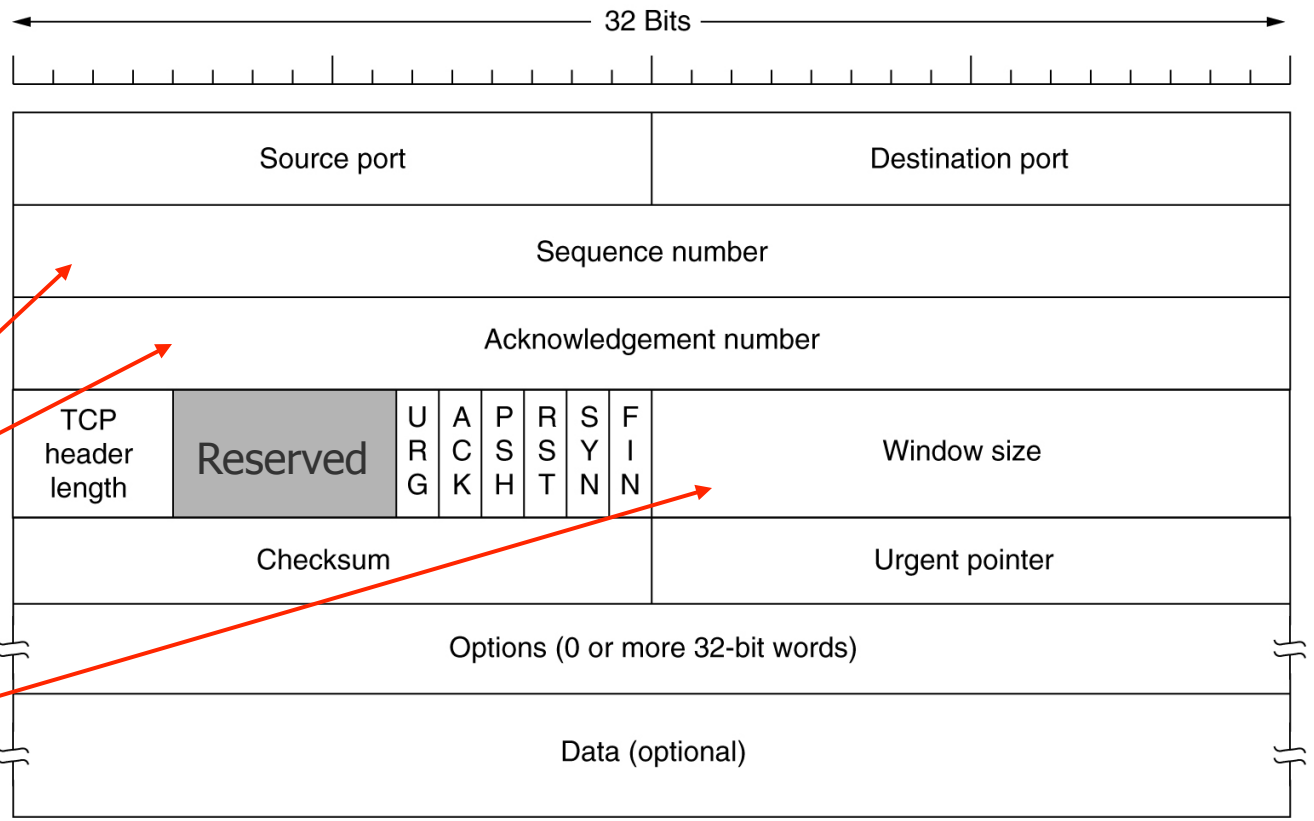
- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

The TCP Header

- 20 byte header
- Plus options
- Up to 65495 data bytes

Counting by bytes of data (not segments!)

Number of bytes receiver willing to accept



The TCP Header

- **Source and Destination Port**: port number of sender resp. receiver
- **Sequence Number/Acknowledgment Number**: Segments have a 32 bit sequence and acknowledgement number for the window mechanism in flow control (Sliding Window).
  - Sequence and acknowledgement number count bytes!
  - The acknowledgement number indicates the next expected byte!
  - Sequence numbers begin not necessarily with 0! A random value is chosen to avoid a possible mix-up of old (late) segments.
  - Piggybacking, i.e., an acknowledgement can be sent in a data segment.
- **Header Length**: As in case of IP, also the TCP header has an indication of its length. The length is counted in 32-bit words.
- **Window Size**: Size of the receiver's buffer for the connection.
  - Used in flow control: the window of a flow indicates, how many bytes at the same time can be sent.
  - The size of the buffer indicates, the number of bytes the receiver can accept.
  - The window of flow control is adapted to this value.

The TCP Header

- **Flags**:
  - URG: Signaling of special important data, e.g., abort, Ctrl-C
  - ACK: This bit is set, if this is an acknowledgement
  - PSH: Immediate transmission of data, no more waiting for further data
  - RST: Reset a connection, e.g., during a host crash or a connecting rejection
    - Generally problems arise when a segment with set RST bit is received
  - SYN: set to 1 for connection establishment
  - FIN: set to 1 for connection termination
- **Urgent flag**: indicates, at which position in the data field the urgent data ends (byte offset of the current sequence number).
- **Options**:
  - Negotiation of a **window scale:** Window size field can be shifted up to 14 bits ➡ allowing windows of up to 2^{30} bytes
  - Use of **Selective Repeat** instead of **Go-Back-N** in the event of an error
  - Indication of the **Maximum Segment Size** (**MSS**) to determine the size of the data field

TCP Pseudo Header

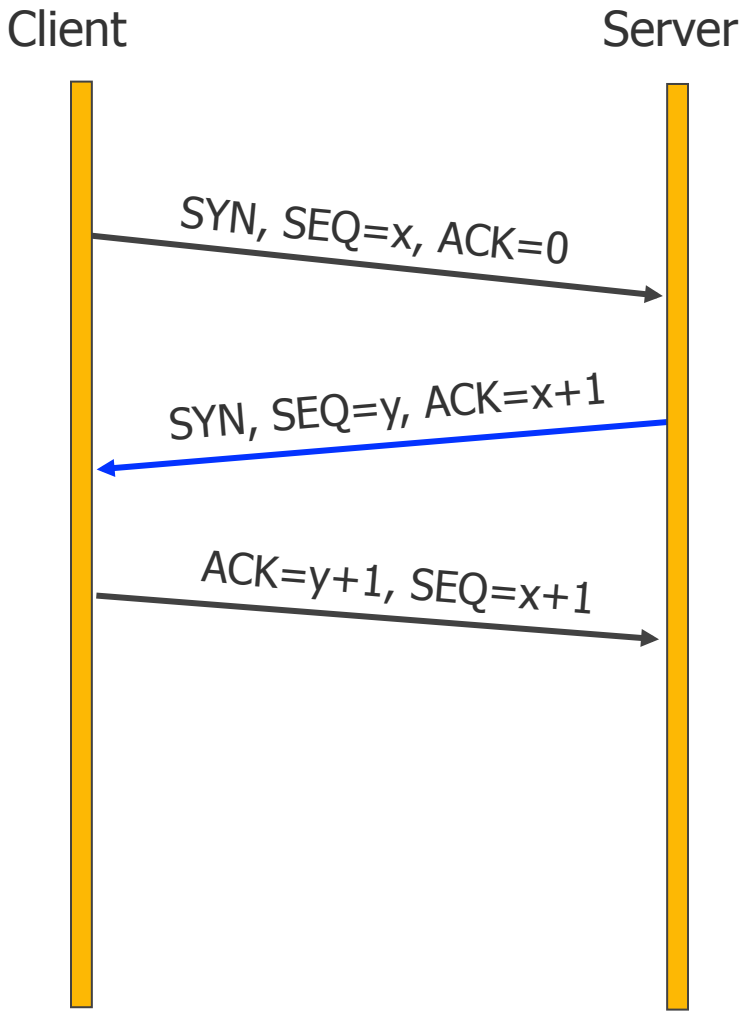
- **Checksum**: serves among other things for the verification that the packet was delivered to the correct device.
  - The checksum is computed using a **pseudo header**. The pseudo header is placed in front of the TCP header, the checksum is computed based on both headers (the checksum field is here 0).
    - The checksum is computed as the 1-complement of the sum of all 16-bit words of the segment including the pseudo header.
  - The receiver also places the pseudo header in front of the received TCP header and executes the same algorithm (the result must be 0).

Source address (IP)		
Destination address (IP)		
00000000	Protocol = 6	Length of the TCP segment

Content of this Section

- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

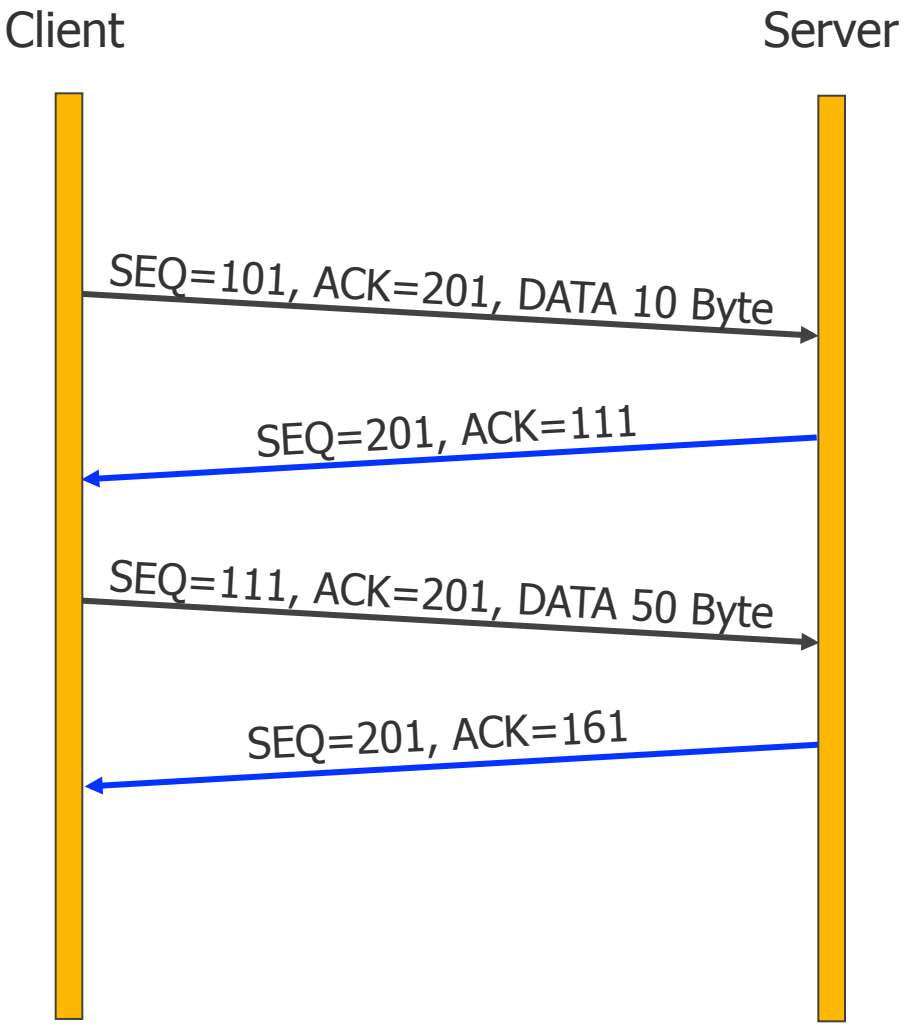
TCP Connection Management: 1. Connection Establishment



Three Way Handshake

- The server waits for connection requests using LISTEN and ACCEPT.
- The client uses the CONNECT operation by indicating IP address, port number, and the acceptable maximum segment size (MSS).
- CONNECT sends a segment with SYN bit set.
- If destination port of the CONNECT is the port number on which the server waits, connection is accepted (else rejected with RST)
- If accept, the server also sends a segment with SYN and ACK bits set to the client and acknowledging the client's SYN segment.
- The client acknowledges the SYN segment of the server. The connection is established.
- Remark: there can be only 1 connection per (IP Address$_1$, Port$_1$, IP Address$_2$, Port$_2$) tuple

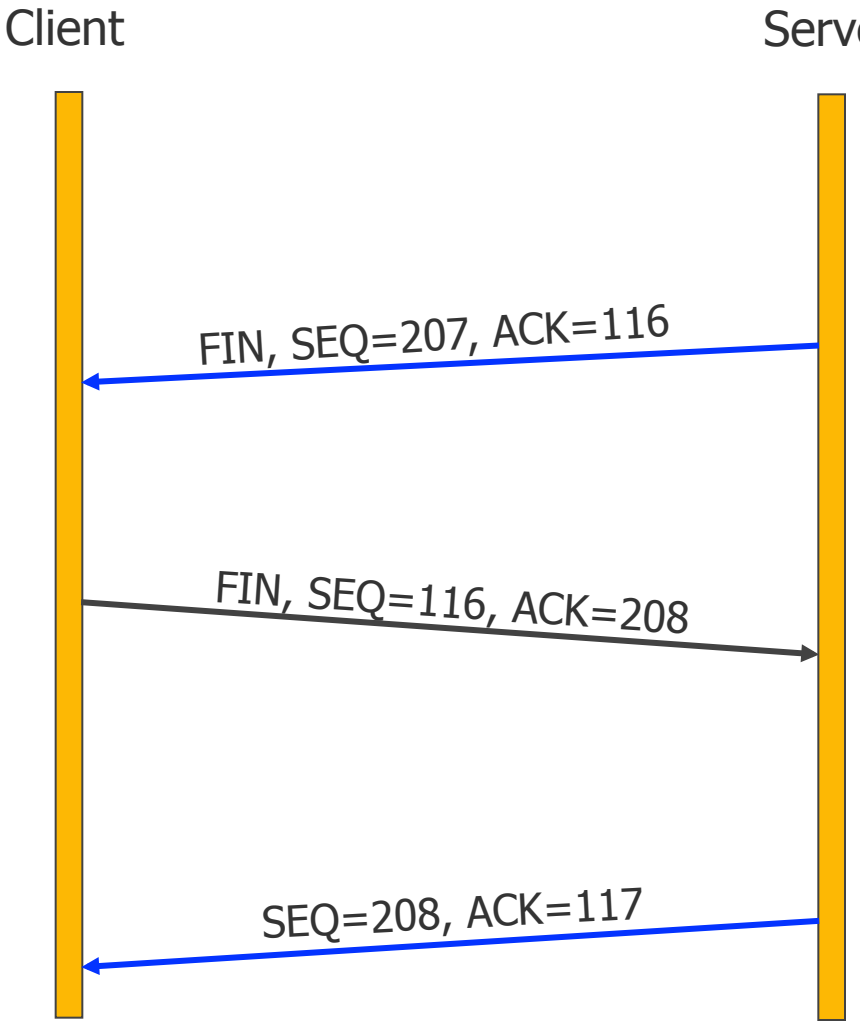
TCP Connection Management: 2. Data Transmission



- Full-duplex connection
- Segmentation of a byte stream into segments.
  - Usual sizes are 1500, 536, or 512 byte (avoids IP fragmentation)
  - All hosts have to accept segments of 536 byte +20 byte = 556 byte
- Usual acknowledgement mechanism:
  - Cumulative: segments up to ACK-1 are confirmed. If timeout before an ACK, the sender retransmits.
- Usual procedure for repeating:
  - Go-Back-N or Selective Repeat

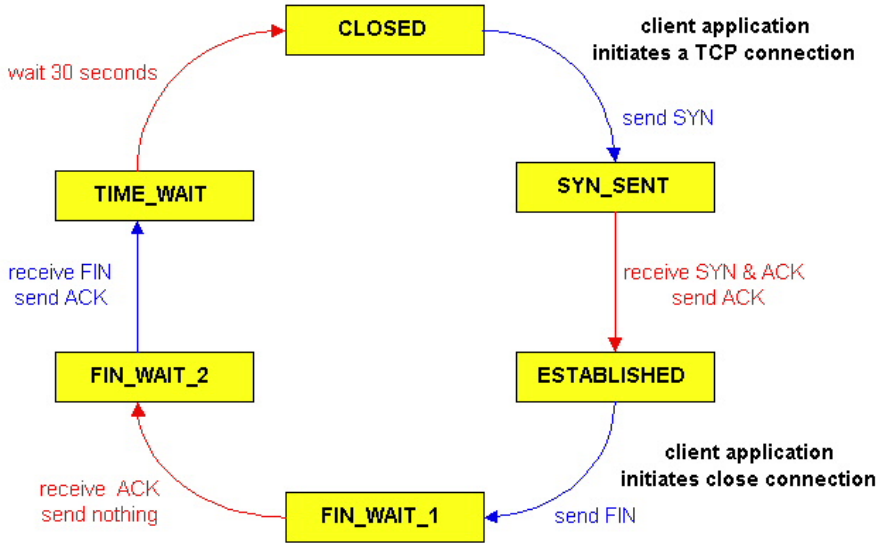
- Remark: sequence numbers and acknowledgements are not for segments but for bytes of transported data!

TCP Connection Management: 3. Connection Termination



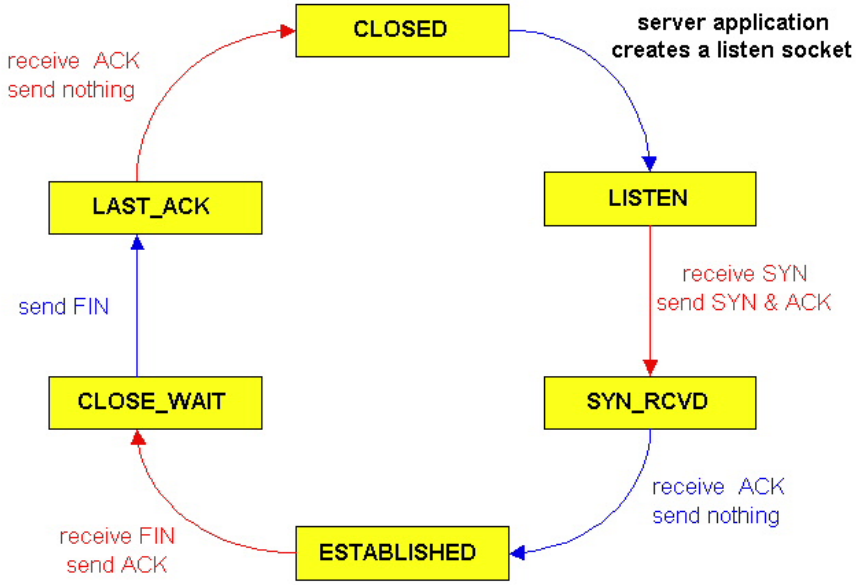
- Termination as two simplex connections
- Send a FIN segment
- If the FIN segment is confirmed, the direction is "switched off". The opposite direction remains however still open, data can be still further sent.
  - Half-open connections!
- Use timers to avoid packet loss.

TCP Connection Management: Finite State Machine



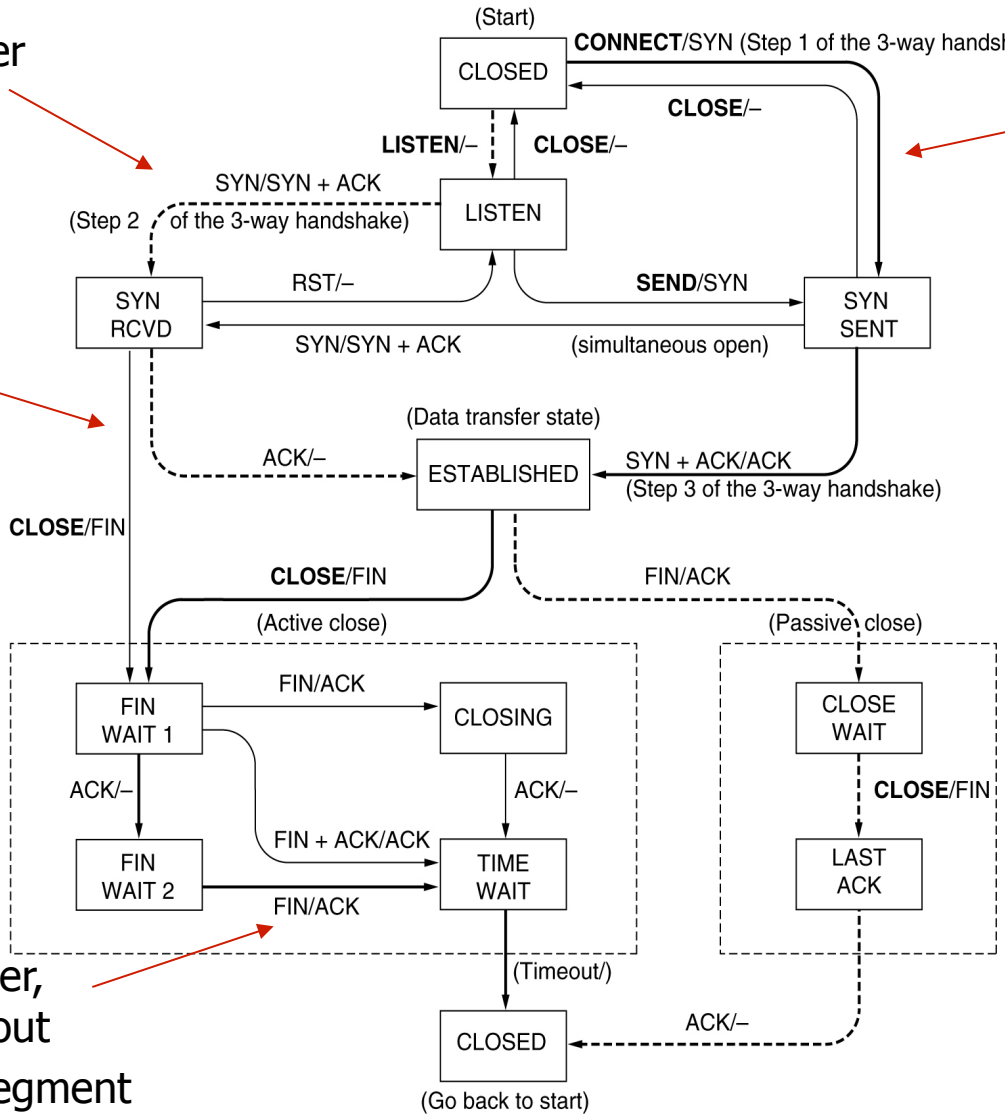
TCP client lifecycle

TCP server lifecycle



The Entire TCP Connection: Finite State Machine

Freie Universität Berlin



Event/action pair

- Event: System call by user, arrival of segment, timeout
- Action: Send a control segment

States during a TCP Session

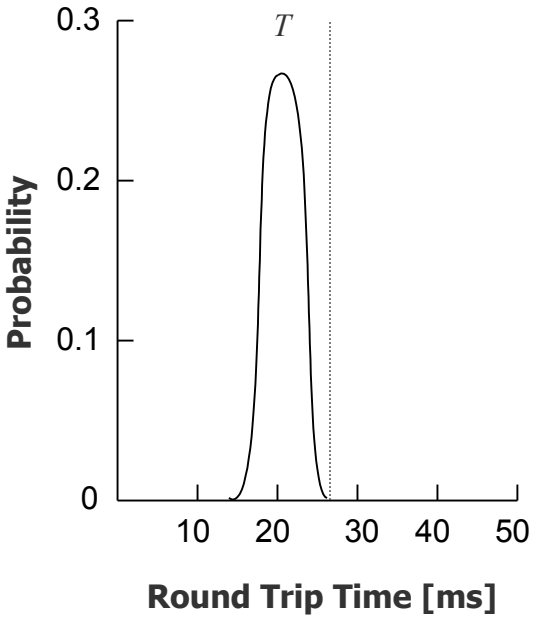
State	Description
CLOSED	No active communications
LISTEN	The server waits for a connection request
SYN RCVD	A connection request was received and processed, wait for the last ACK of the connection establishment
SYN SENT	Application began to open a connection
ESTABLISHED	Connection established, transmit data
FIN WAIT 1	Application starts a connection termination
FIN WAIT 2	The other side confirms the connection termination
TIME WAIT	Wait for late packets
CLOSING	Connection termination
CLOSE WAIT	The other side initiates a connection termination
LAST ACK	Wait for late packets

Content of this Section

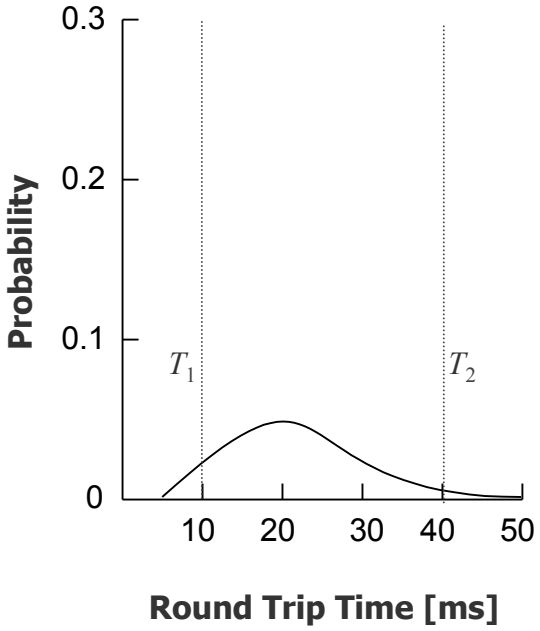
- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

Timer Management with TCP

- TCP uses several timers, e.g. timer T to schedule retransmissions
  - Problem: how to select the retransmission timer value T?
  - Solution: Estimation of probability density of RTT, and set T > estimated RTT
    - Similar to RTT estimation done at the Link Layer
    - But it is typically more difficult to accurately estimate RTT at the Transport Layer



Link Layer



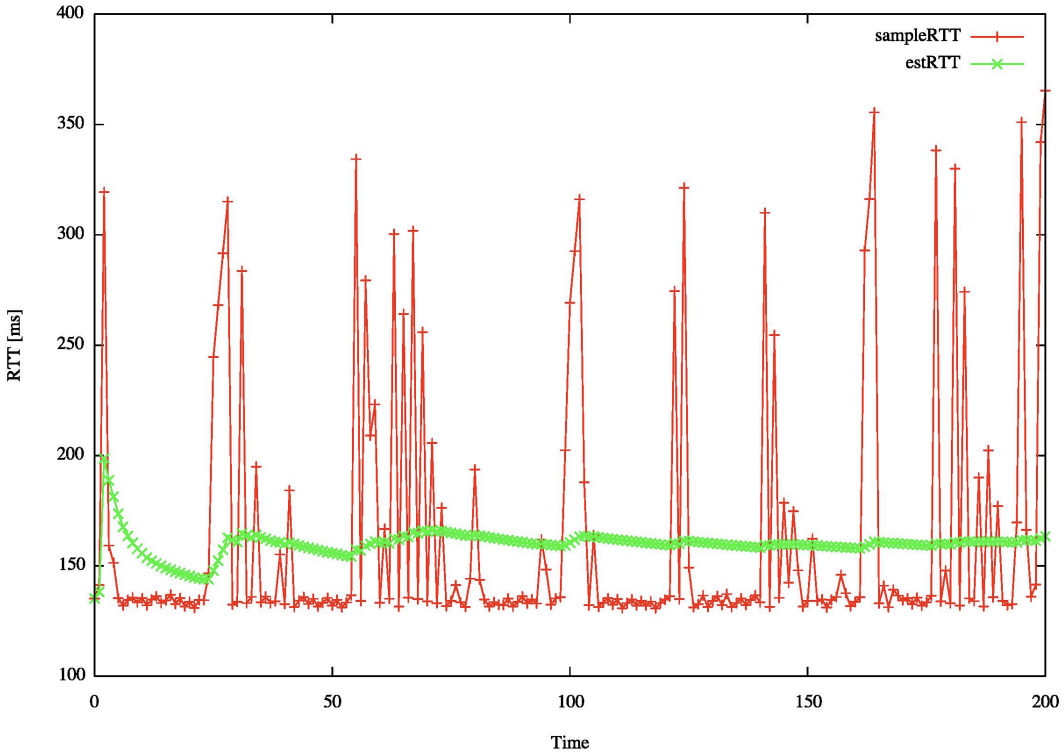
Transport Layer

Impact of imprecision:

- T_1 is too small: too many retransmissions
- T_2 is too large: delay to recover from packet loss

Sampling and Estimating RTT

- RTT typically varies dynamically



- How to estimate RTT?
  - SampleRTT: measured time from segment transmission until ACK receipt
  - Problem: SampleRTT will vary too much: need "smoother" estimated RTT
  - Solution: use average several recent measurements, not just current SampleRTT

Timer Management with TCP: Retransmission Timer

- **Algorithm of Jacobson (1988)**
  - Dynamic algorithm for RTT estimation, which adapts the timer by current measurements of the network performance
  - TCP manages a variable RTT (Round Trip Time) for each connection
  - RTT is momentarily the best **estimation** of the round trip time
  - When sending a segment, a timer is started which measures the time the acknowledgement needs and initiates a retransmission if necessary.
  - If the acknowledgement arrives before expiration of the timer (after a time unit $sampleRTT$), RTT is updated:

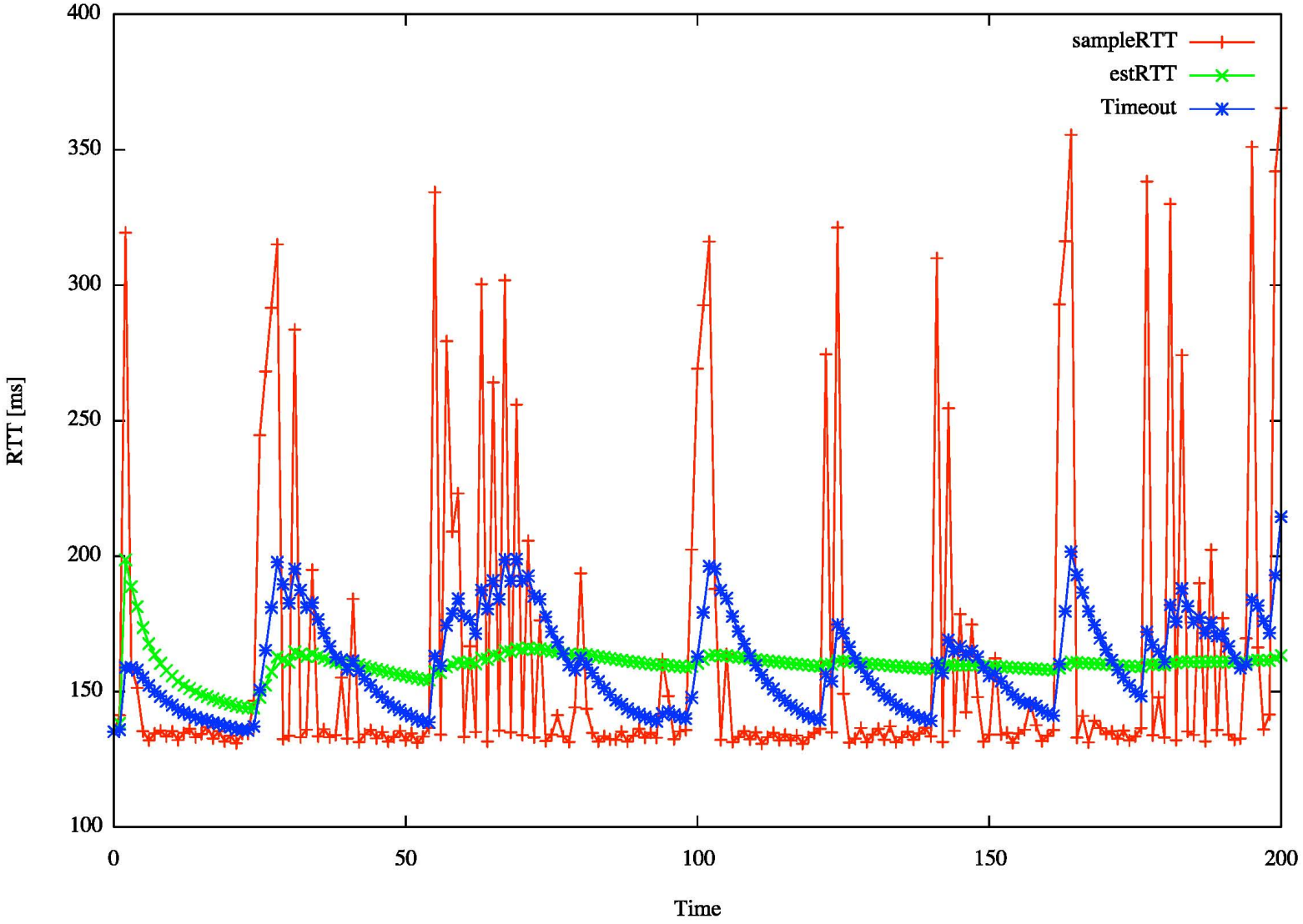
$$RTT = \alpha \times RTT + (1 - \alpha) \times sampleRTT$$

  - α is a smoothing factor, typically 0.875
  - The choice of the timeout is based on RTT

$$Timeout = \beta \times RTT$$

  - Typically, β is chosen as 2.

Timer Management with TCP



Timer Management with TCP: Retransmission Timer

- **Problem:** $\beta \times RTT$ is not reactive enough to sudden changes
- **Improvement (Jacobson):** set timer proportionally to the standard deviation of the arrival time of acknowledgements
  - Computation of standard deviation:

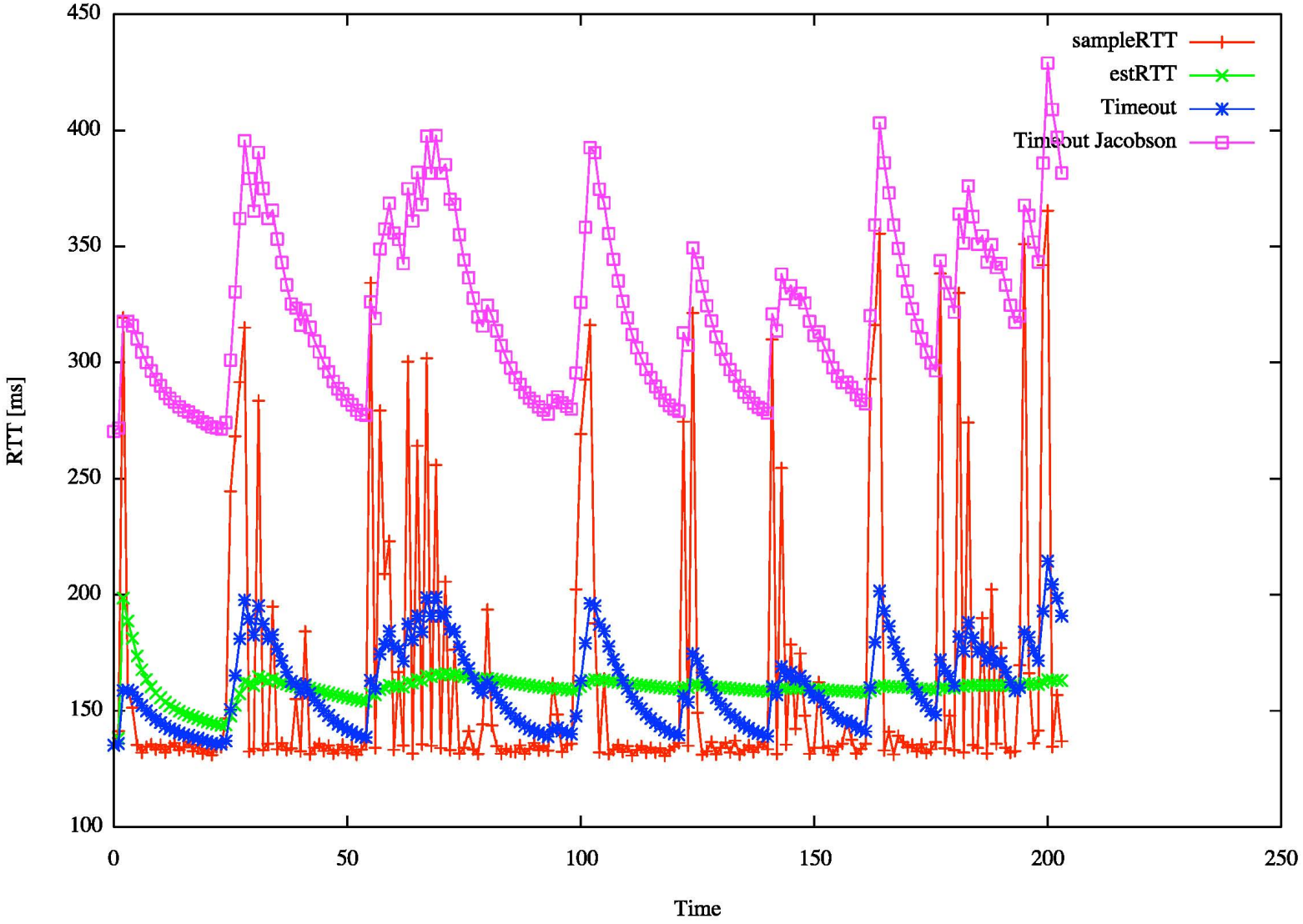
$$devRTT = \gamma \times devRTT + (1 - \gamma) \times |RTT - sampleRTT|$$

  - Typically $\gamma = 0.75$
  - Standard timeout interval:

$$Timeout = RTT + 4 \times devRTT$$

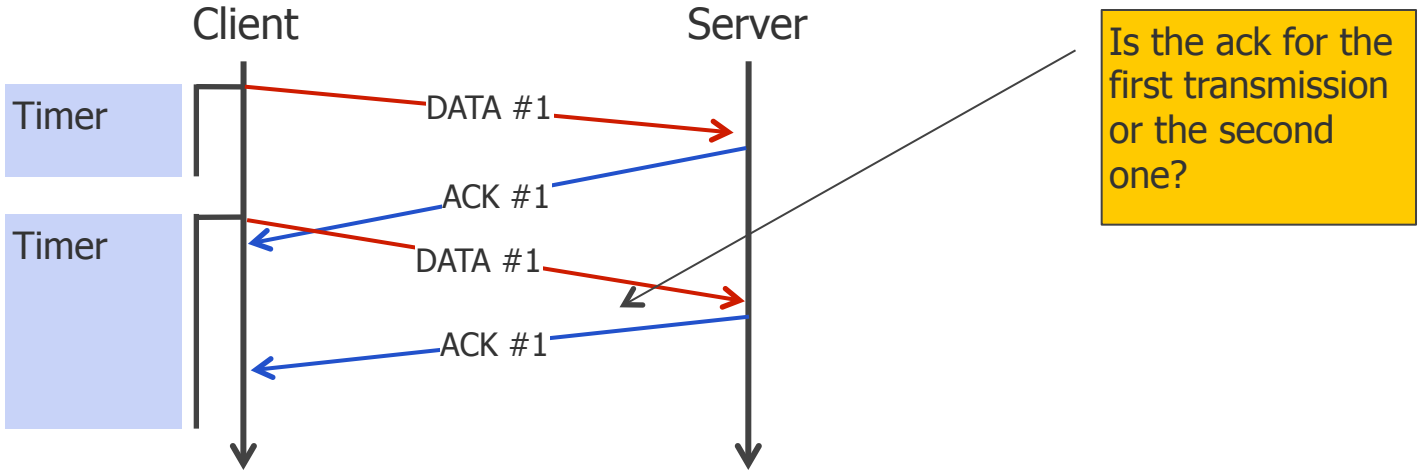
  - The factor 4 was determined exprimentally, and kept because it is fast and simple to use in computations.

Timer Management with TCP



Timer Management with TCP: Retransmission Timer

- Karn's Algorithm
  - Do not update RTT on any segments that have been **retransmitted**.
    - Exponential backoff: timeout is doubled on each failure until the segments get through.
  - Very simple proposal, which is used in most TCP implementations (though it is optional)



Timer Management with TCP: Other Timers

Persistence timer
- Prevents a deadlock with a loss of the buffer release message of a receiver
- With expiration of the timer, the sender transfers a test segment. The response to this transmission contains the current buffer size of the receiver. If it is still zero, the timer is started again.

Keep-alive timer
- If a connection is inactive for a long time, at expiration of the timer it is examined whether the other side is still living.
- If no response is given, the connection is terminated.
- Disputed function, not often implemented.

Time Wait timer
- During the termination of a connection, the timer runs for the double packet life time to be sure that no more late packets arrive.

Content of this Section

- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

TCP Reliable Data Transfer

- Basic principles of reliable transfer with TCP
  - TCP creates reliable data transfer service on top of IP's unreliable service
  - Pipelined segments (sliding window + cumulative acks)
  - TCP uses single retransmission timer
- Retransmissions are triggered by:
  - Timeout events
  - Duplicate acks
- Let's initially consider a simplified TCP sender:
  - Ignore duplicate acks
  - Ignore flow control, congestion control

TCP Sender Events

- Event type 1: Data received from app
  - Create segment with sequence number
  - Sequence number is byte-stream number of first data byte in segment
  - Start timer (if not already running: think of timer as for oldest unacked segment)
  - Expiration interval: TimeOutInterval
- Event type 2: Timeout
  - Retransmit segment that caused timeout
  - Restart timer
- Event type 3: ACK received
  - If it acknowledges previously unacked segments
    - Update what is known as acked
    - Start timer if there are outstanding segments

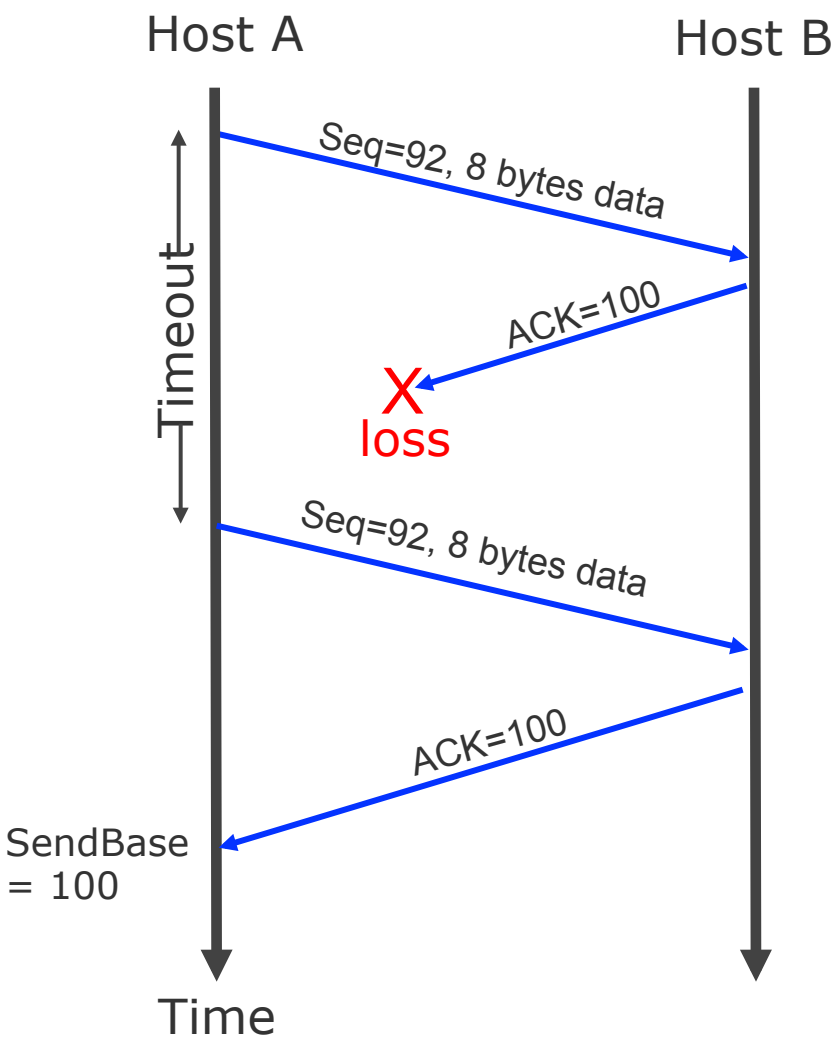
TCP Sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
    switch(event)

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

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

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

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

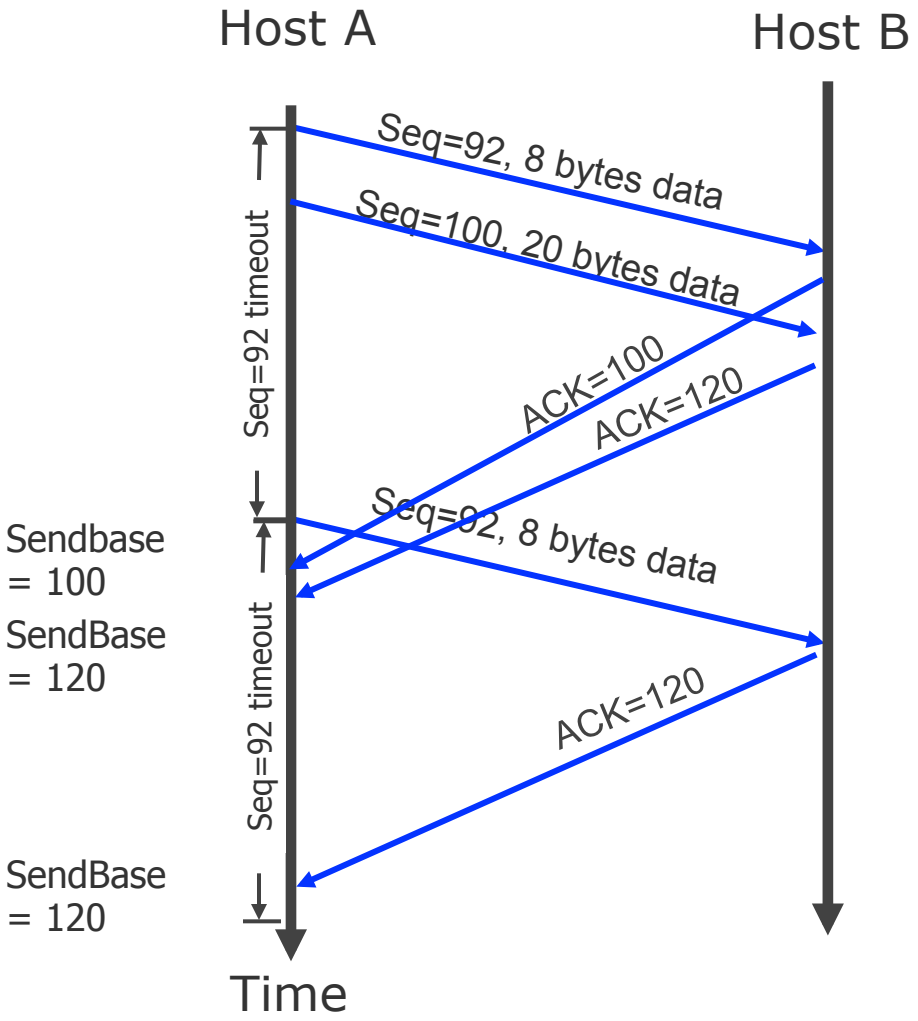
Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+; y > SendBase, so that new data is acked

TCP Retransmission Scenarios

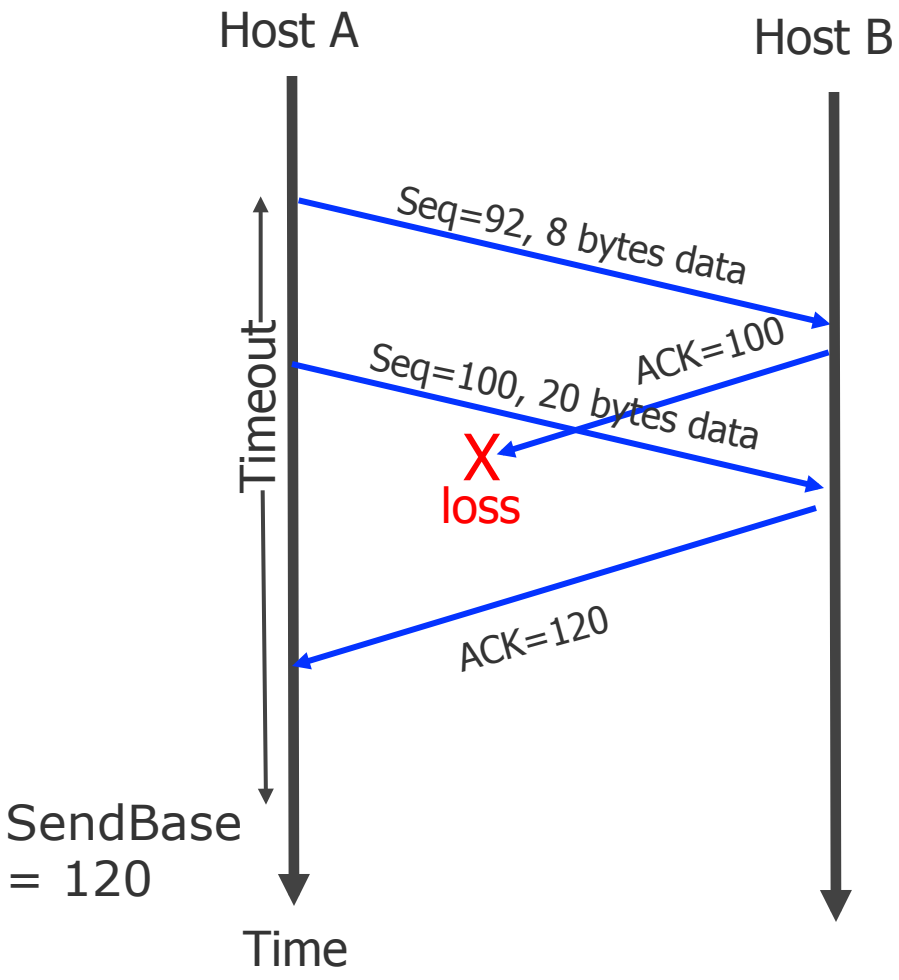


Lost ACK scenario



Premature timeout

TCP Retransmission Scenarios

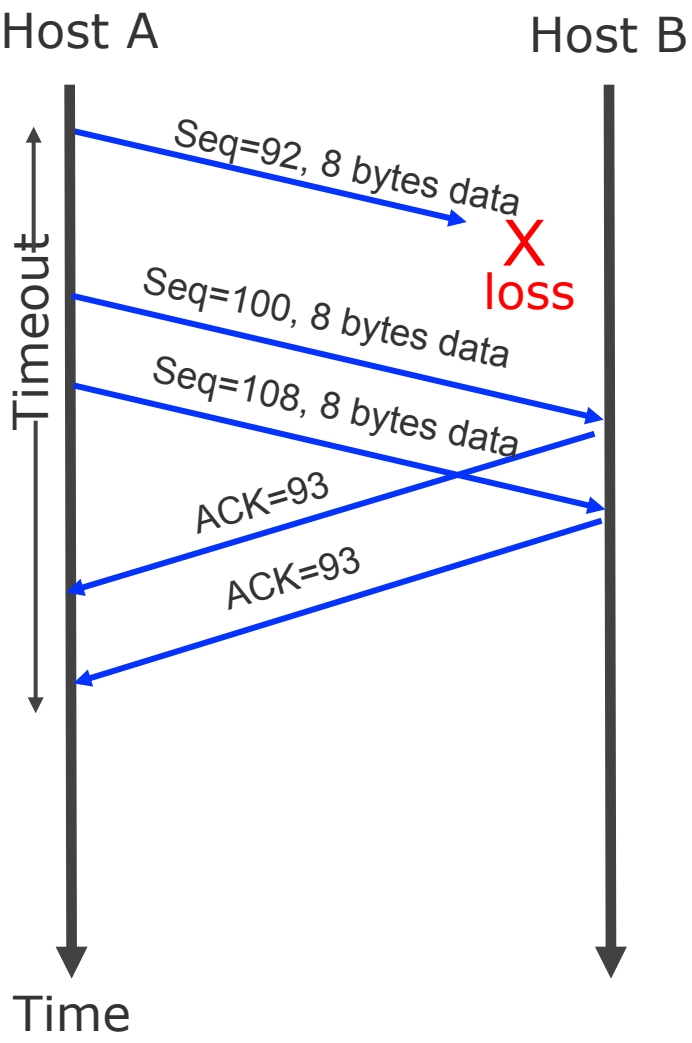


Cumulative ACK scenario

TCP ACK generation [RFC 1122, RFC 2581]

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

TCP Receiver: Duplicate Acking



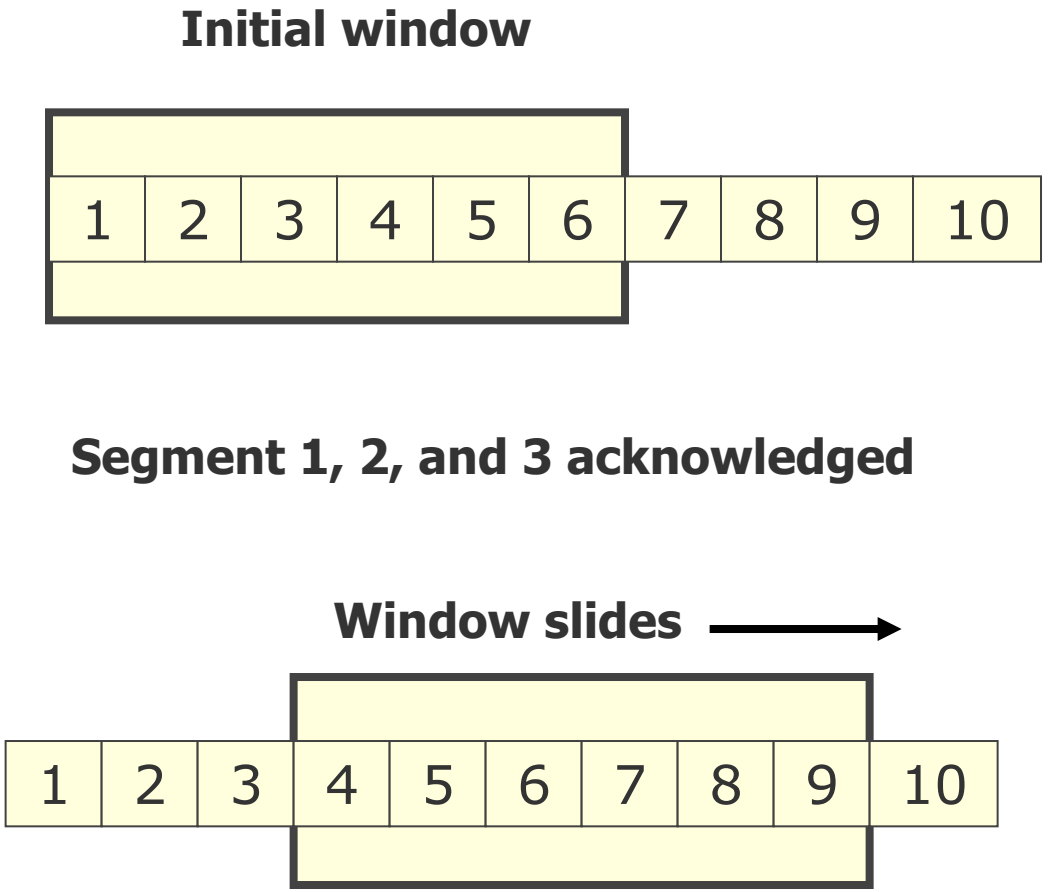
Duplicate ACK scenario

Content of this Section

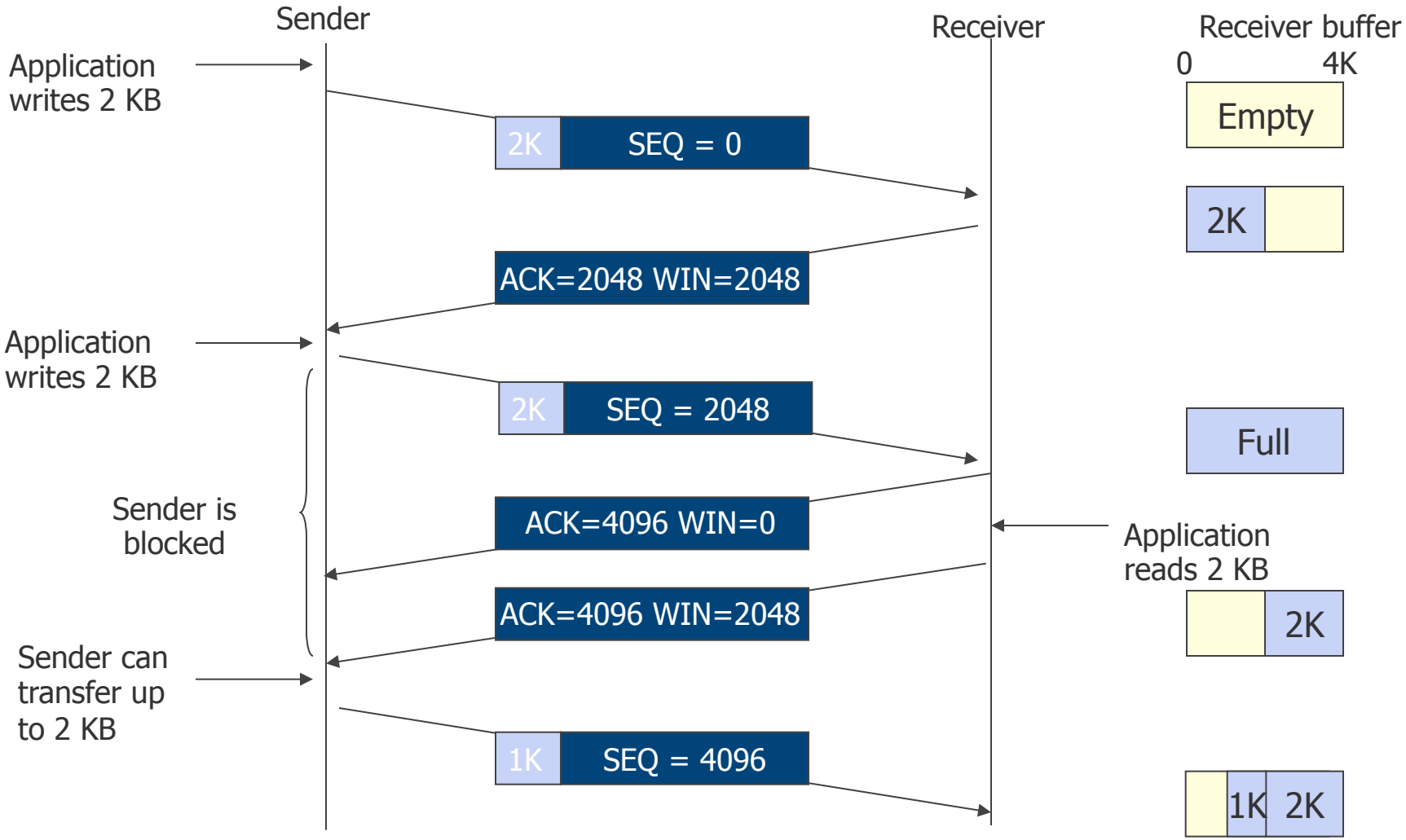
- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

TCP Flow Control: Sliding Window

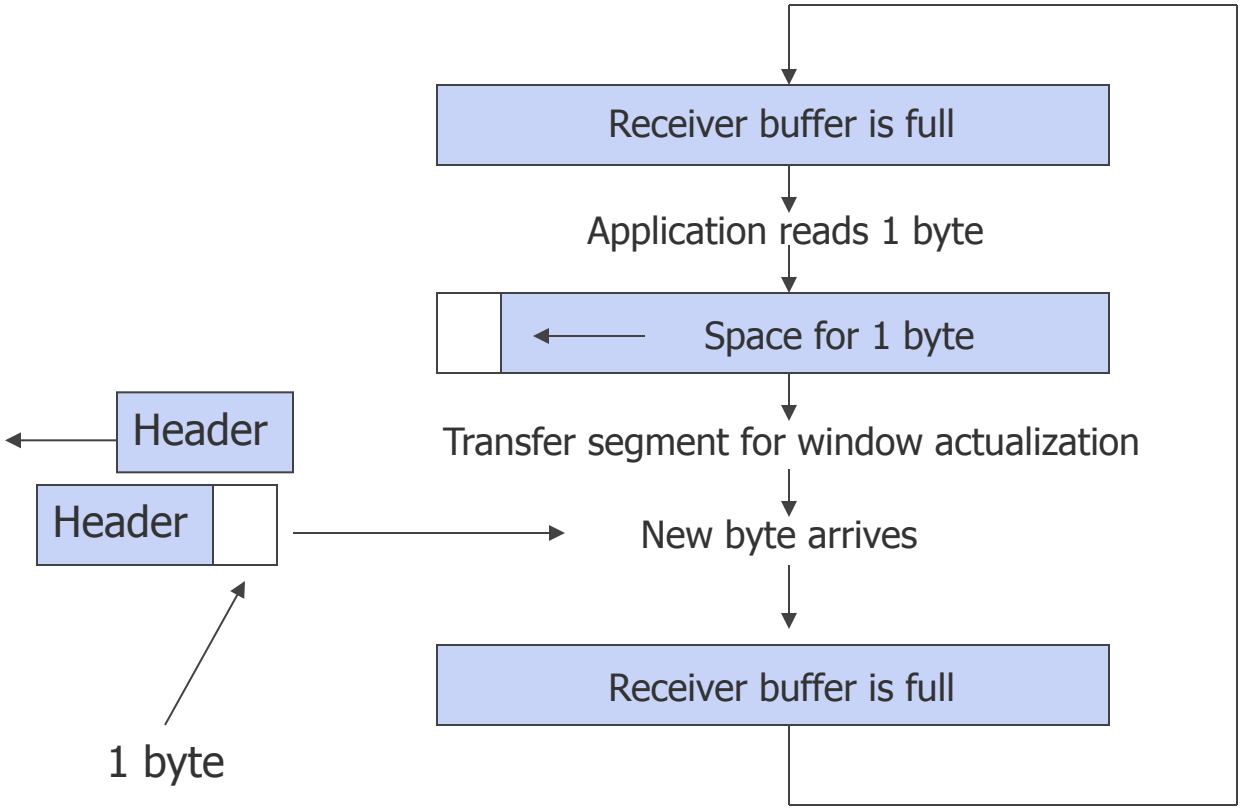
- To provide reliable data transfer, a sliding window mechanism is used.
- Similar to Layer 2, but:
  - Sender sends **bytes** according to the window size
    - Window is shifted by **n** byte as soon as an ACK for **n** bytes arrives
  - The window size can be changed dynamically during the transmission phase
  - Remark: Urgent data (URGENT flag is set) is sent immediately disregarding the sliding window mechanism



TCP Flow Control: Sliding Window, Example

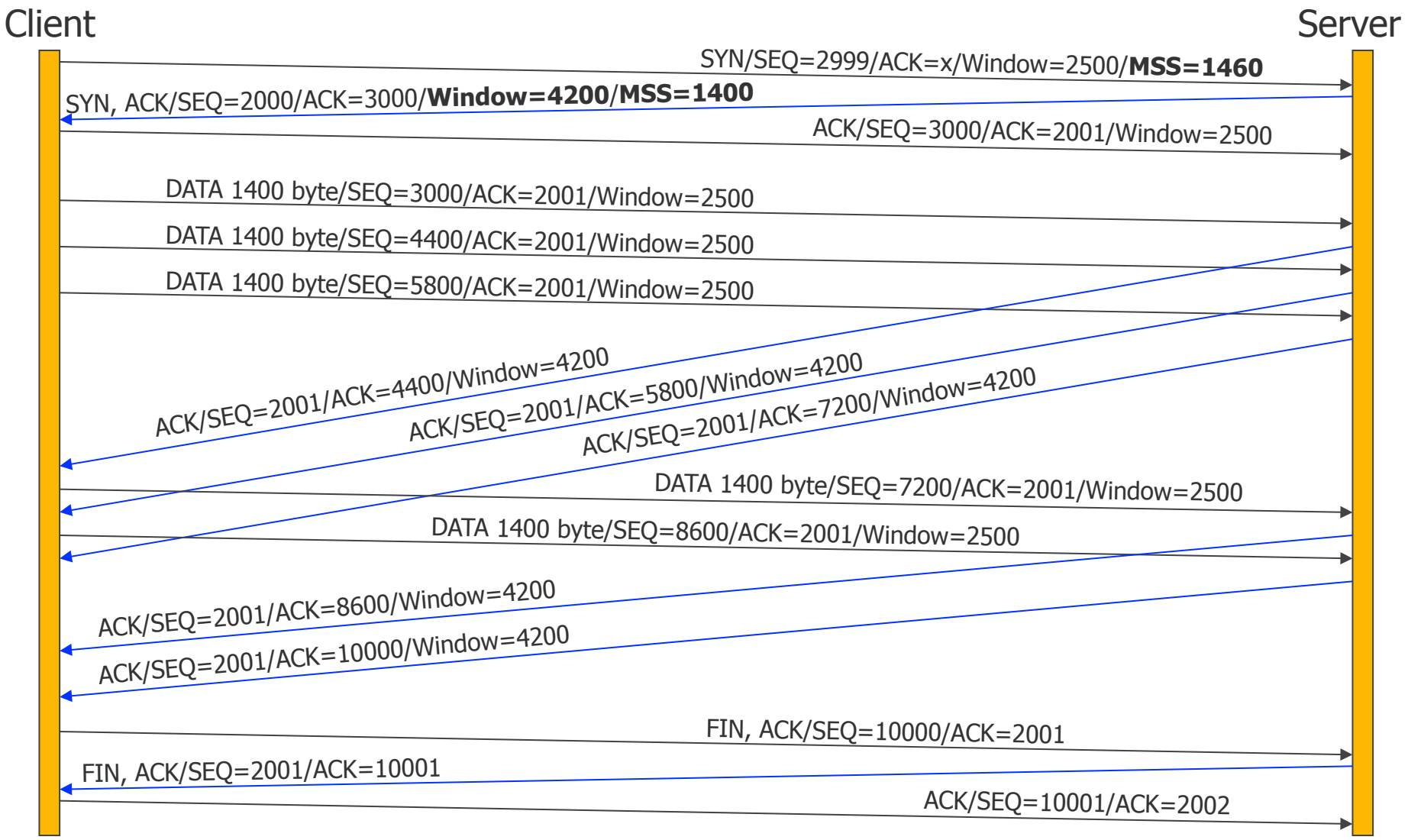


"Silly Window" Syndrome



➔ Before updating the window size, the receiver must wait until its buffer is reasonably empty again!

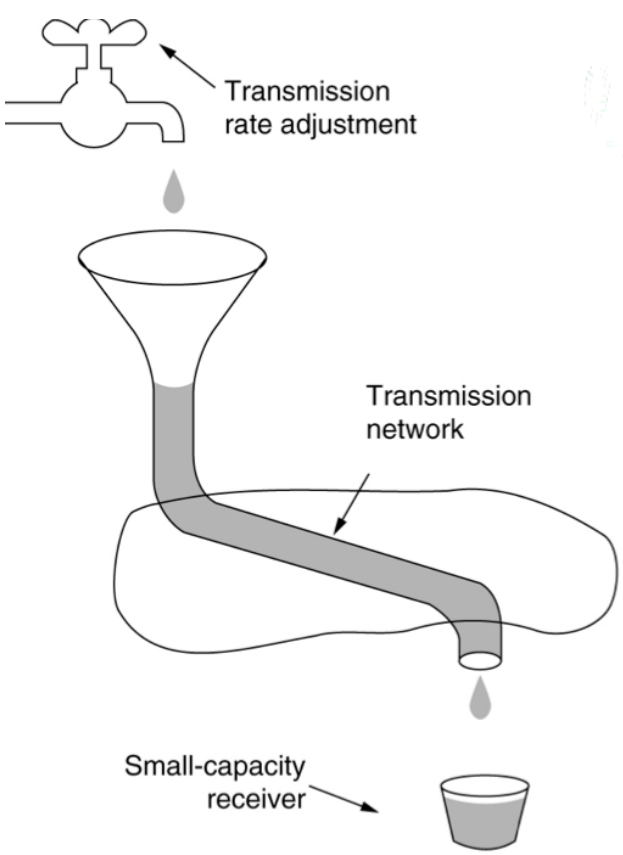
Example of a Complete TCP Session

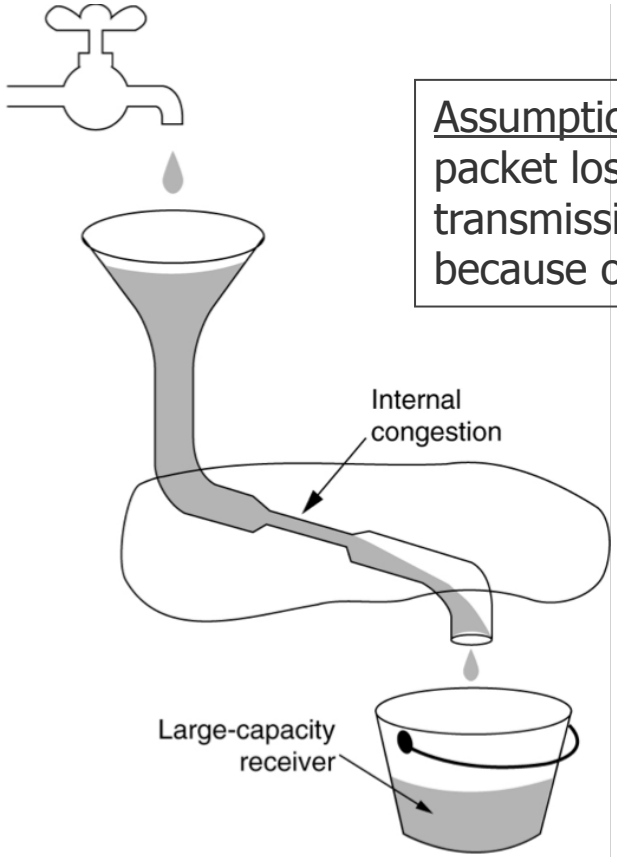


Content of this Section

- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

Flow Control *vs* Congestion Control





Assumption:
packet loss is rarely because of transmission errors, more often because of overload situations.

Adjusting to receiver capacity:
Flow Control Window

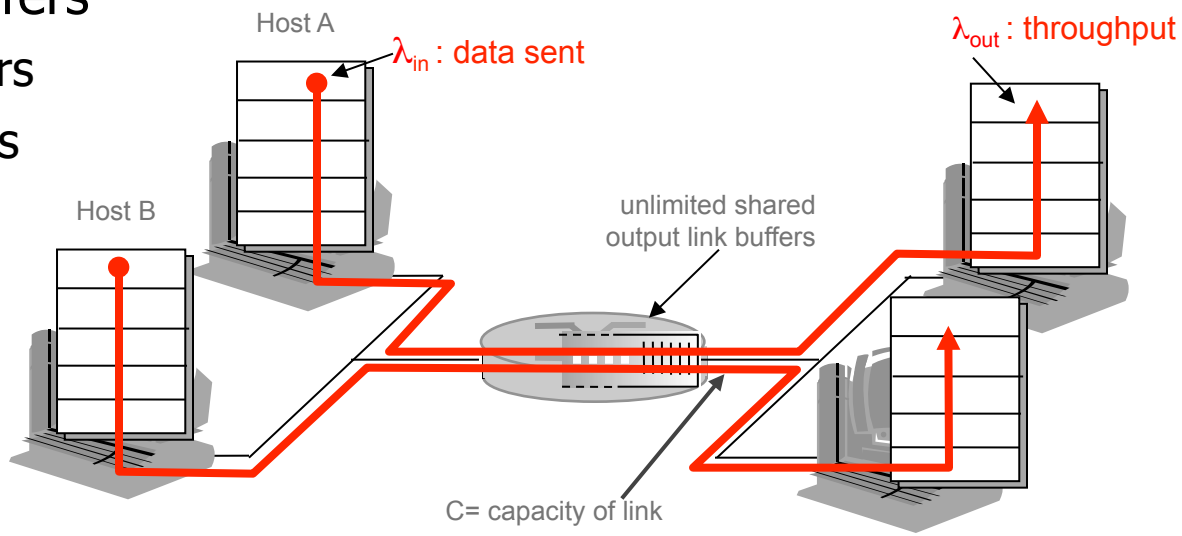
Adjusting to network capacity:
Congestion Window

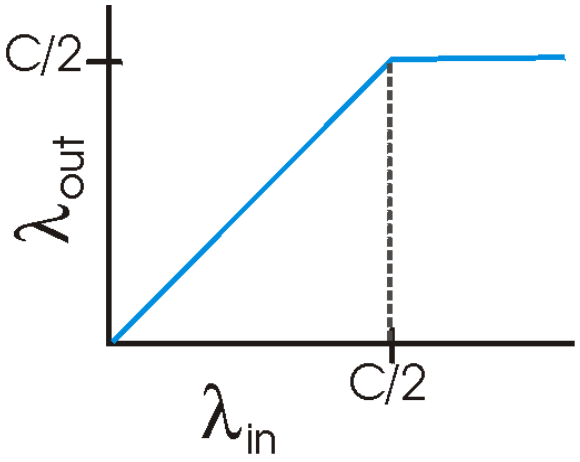
Principles of Congestion Control

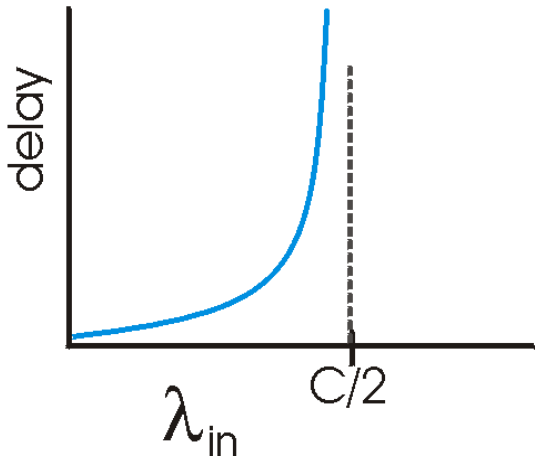
- Congestion:
  - "too many sources sending too much data too fast for the network to handle it"
    - Different from flow control!
  - Manifestations:
    - Lost packets (buffer overflow at routers)
    - Long delays (queueing in router buffers)
  - Solving this is one of the main challenges of the Internet!

Causes & Impact of Congestion: Scenario 1

- Scenario with infinite buffers
  - Two senders, two receivers
  - One router, infinite buffers
  - Retransmissions ignored



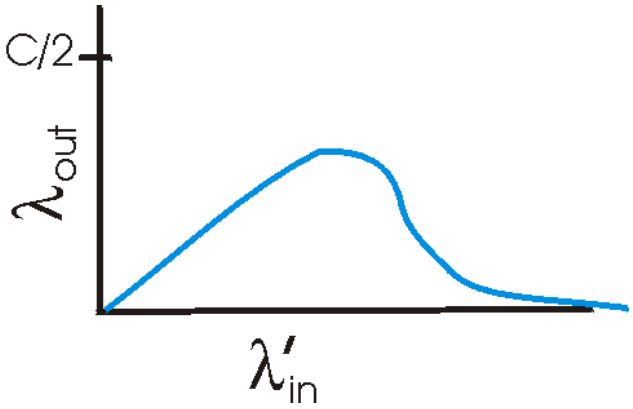


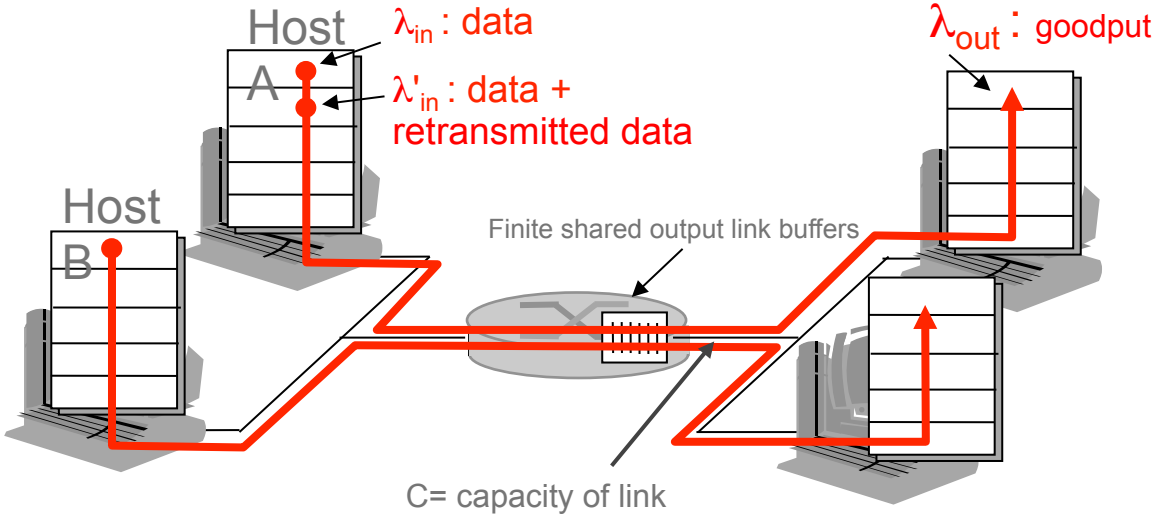


- Large delays when congested
- Maximum achievable throughput
- Question: how to automatically detect & share fairly network capacity?

Causes & Impact of Congestion

- Scenario with finite buffers
  - One router, finite buffers (➔ if full, incoming packets are dropped)
  - Sender retransmits dropped packets
  - Offered load λ'_{in}: Original data + retransmitted data





- More (re)transmissions for same goodput!
  - When packets are dropped, any "upstream" transmission capacity used for that was wasted!
- Question: how to automatically avoid congestion ?

Congestion Control Approaches

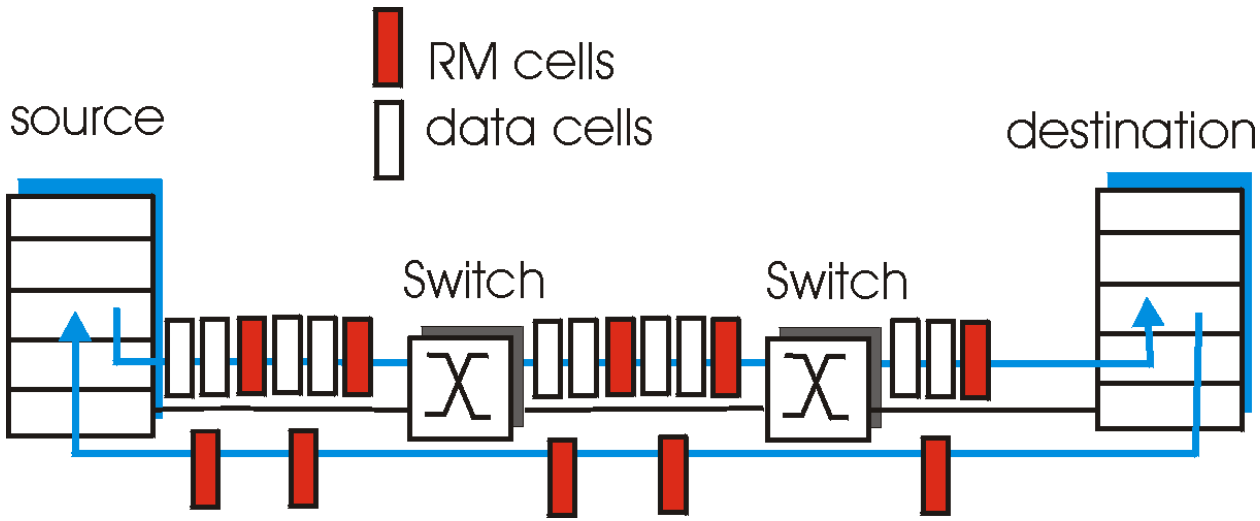
Two main approaches for congestion control:

- End-to-end congestion control:
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - Approach taken by TCP

- Network-assisted congestion control:
  - Routers provide feedback to end systems, e.g.
    - Explicit rate sender should send at
    - Flag indicating congestion
      - Approach taken by ATM, or TCP/IP ECN

Network-assisted Congestion Control in ATM

- ABR: available bit rate:
  - "elastic service" with ATM
  - If sender's path "underloaded":
    - Sender should use available bandwidth
  - If sender's path congested:
    - Sender throttled to minimum guaranteed rate
- RM (resource management) cells:
  - Sent interspersed with data cells
  - Bits in RM cell set by switches on path
    - NI bit: no increase in rate (mild congestion)
    - CI bit: congestion indication
    - Option to indicate ER (explicit rate)
  - Receiver sends RM cells back to sender
    - bits are not modified on the way back

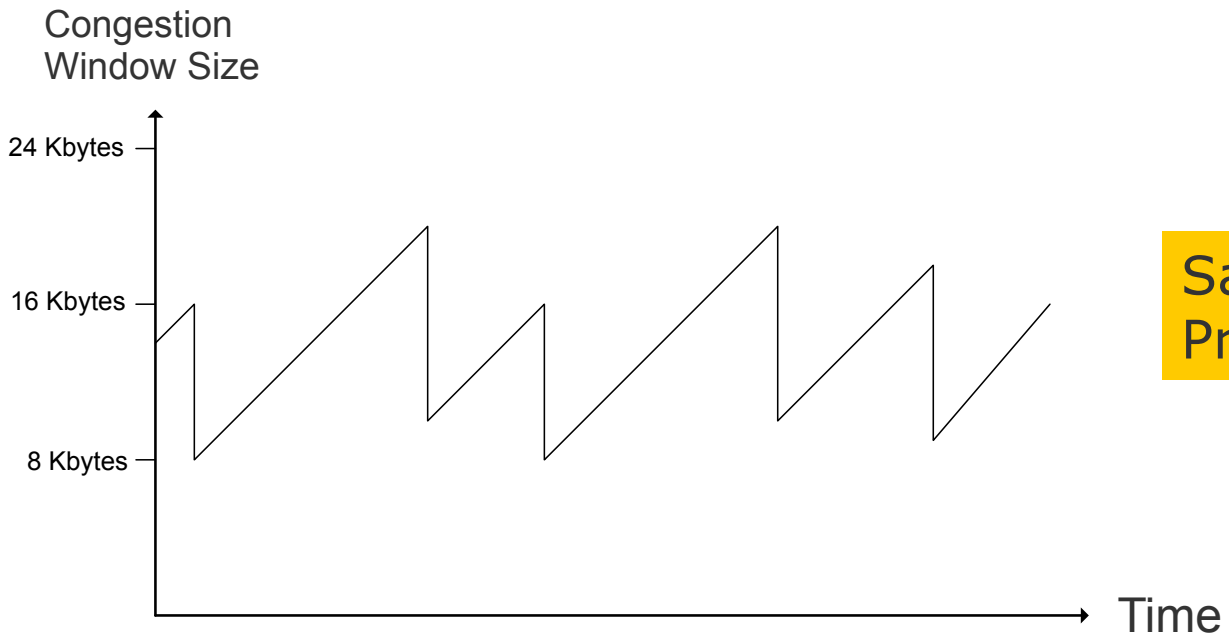


Network-assisted Congestion Control in TCP/IP

- Explicit Congestion Notification (ECN)
  - Specified in RFC3168
  - Not all end-points support this option
- When both endpoints support ECN they mark their IP packets accordingly
  - Sender sets ECT (ECN-Capable Transport) code point in IP packets
  - Remark: This is happening at the network layer!
- IP packets that traverse congested areas can then be marked by routers
  - A congested router sets both ECN bits in IP packet header to signal congestion
  - Offers alternative to simply dropping the packet
  - Remark: This is *still* happening at the network layer!
- Upon receiving a ECN-marked IP packet, the receiving end-point transport layer echoes back the signal to the sender's transport layer
  - Purpose is to signal it should reduce its sending rate

End-to-end Congestion Control in TCP

- Approach: increase transmission rate (window size), probing for usable bandwidth until loss occurs (signaling congestion), using an algorithm based on **AIMD (additive increase, multiplicative decrease)**
  - Additive Increase: increase **CongWin** by 1 MSS every RTT until loss detected
  - Multiplicative Decrease: cut **CongWin** in half after loss



Saw tooth behavior:
Probing for bandwidth

TCP Congestion Control: Overview

- Sender throttles transmissions with the minimum of the congestion window and the receiver window, i.e. at all times:

$$LastByteSent - LastByteAcked \leq \min(CongWin, ReceiverWin)$$

- Typically *CongWin* < *ReceiverWin*
  - Rough estimation of sending rate:

$$\text{rate} = \frac{CongWin}{RTT} \text{ Bytes/sec}$$

- CongWin is dynamic, function of perceived network congestion
  - Sender perceives congestion via detection of loss event
    - Timeout **or** 3 duplicate acks
  - TCP sender reduces rate (CongWin) after loss event

- Three mechanisms regulate CongWin size evolution:
  - AIMD
  - Slow start
  - Congestion avoidance

TCP Slow Start

- When connection begins, CongWin = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - Initial sending rate = 20 kbit/s

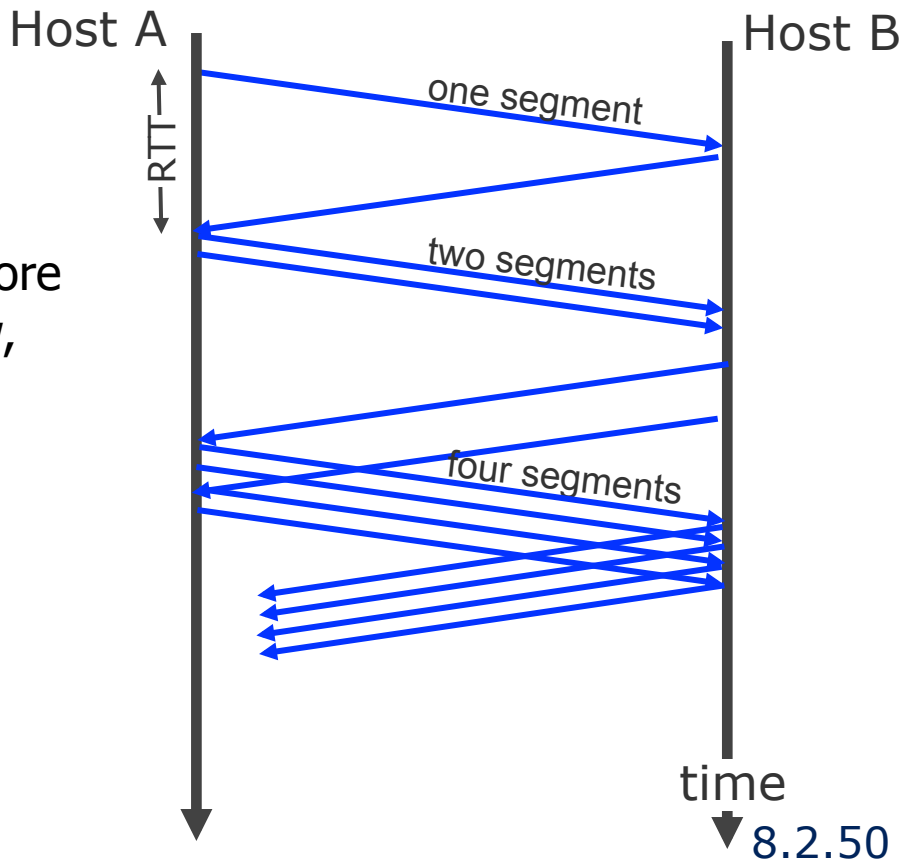
- Problem: Available throughput is typically much bigger than MSS/RTT

  ➔ It is desirable to quickly ramp up to a bigger rate

- Solution: upon initialization increase rate exponentially until first loss event

TCP Slow Start

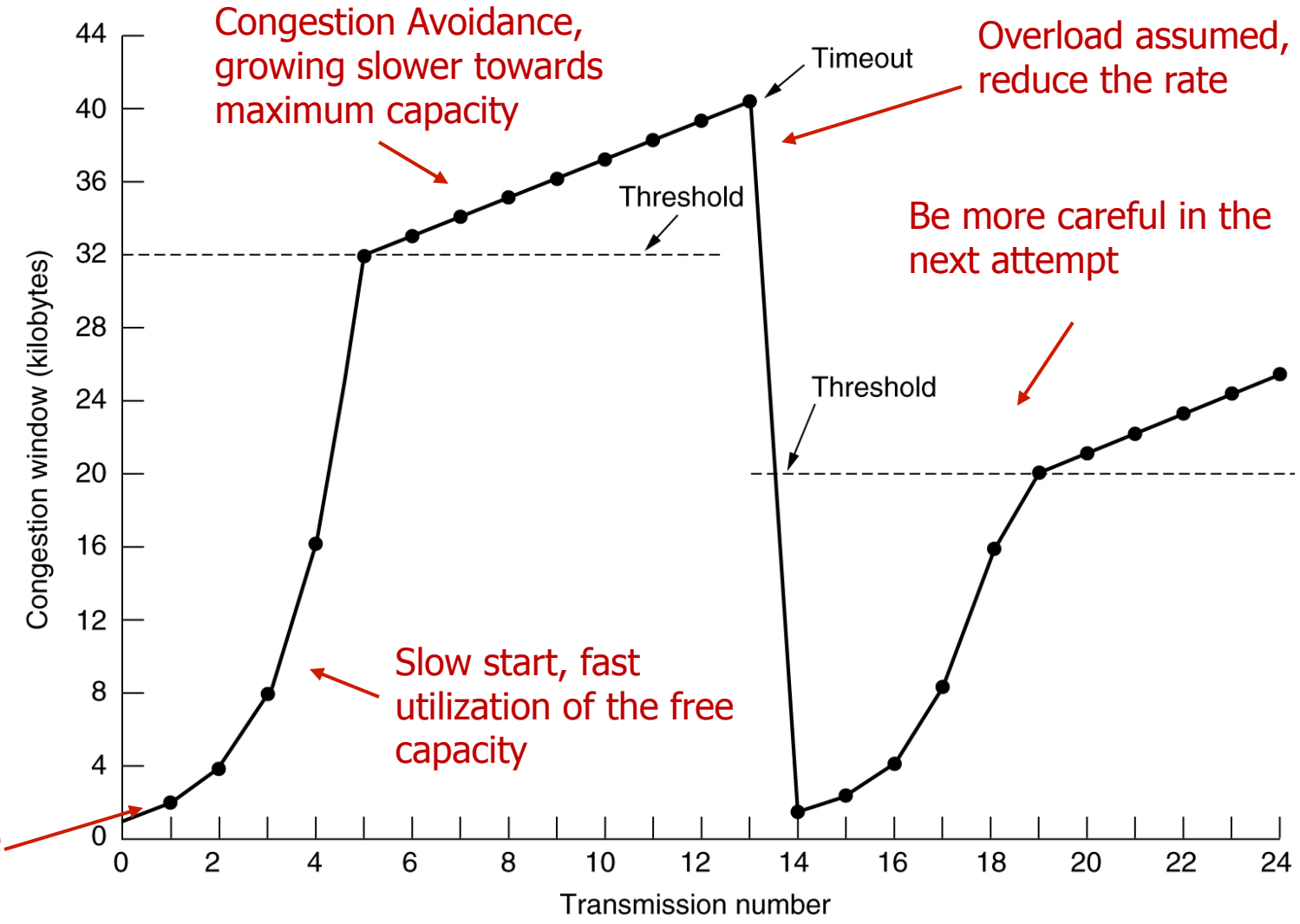
- Phase 1: Upon TCP connection establishment, the sender initializes the congestion window to one **maximum segment size (MSS)**, and the receiver window to the value specified by the other end of the connection.
- Phase 2: A segment with MSS bytes of data is sent
- Phase 3: **Slow Start Algorithm**
  - When an acknowledgement arrives before timeout, double the congestion window, otherwise reset it to the initial value.
  - Congestion window growth stops when reaching the flow control window size



TCP Congestion Avoidance: Inferring Loss

- Refinement 1: introduction of a threshold value **ssthresh**
  - Initially threshold set to 64 kbyte
  - Above threshold, CongWin increases linearly
    - add 1 MSS each time
  - In case of timeout, the threshold modified, set **to half of the maximum window size** reached before the timeout
  - In case of timout, CongWin is set back to 1 MSS
- Refinement 2: after 3 duplicate ACKs
  - CongWin is cut in half
  - Window then grows linearly
- Rationale:
  - 3 dup ACKs is interpreted as a mild congestion scenario, i.e. the network is still capable of delivering some segments
  - Timeout is interpreted as a severe congestion scenario, i.e. need to "reset"

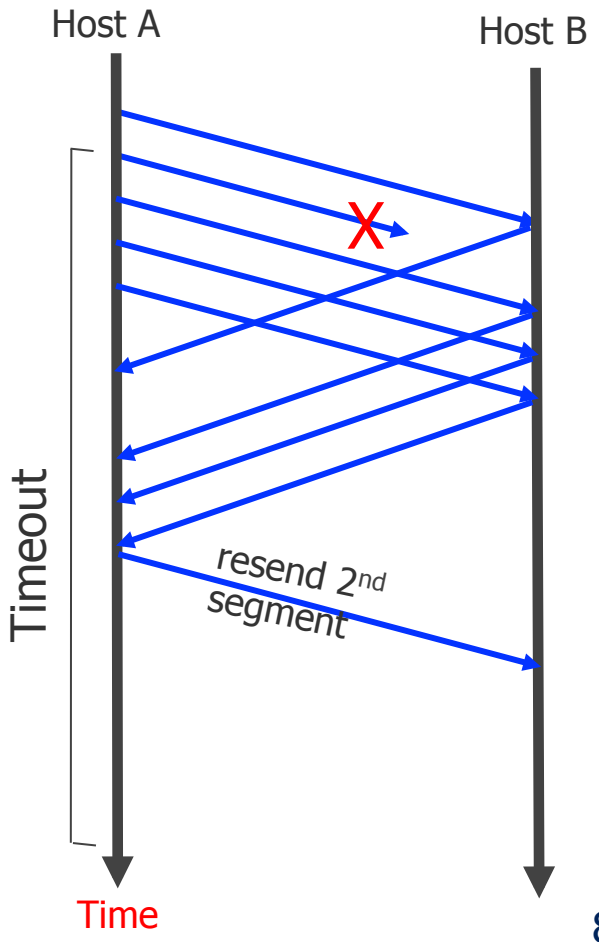
Example of TCP Slow Start + Congestion Avoidance



Fast Retransmit and Recovery (FRR)

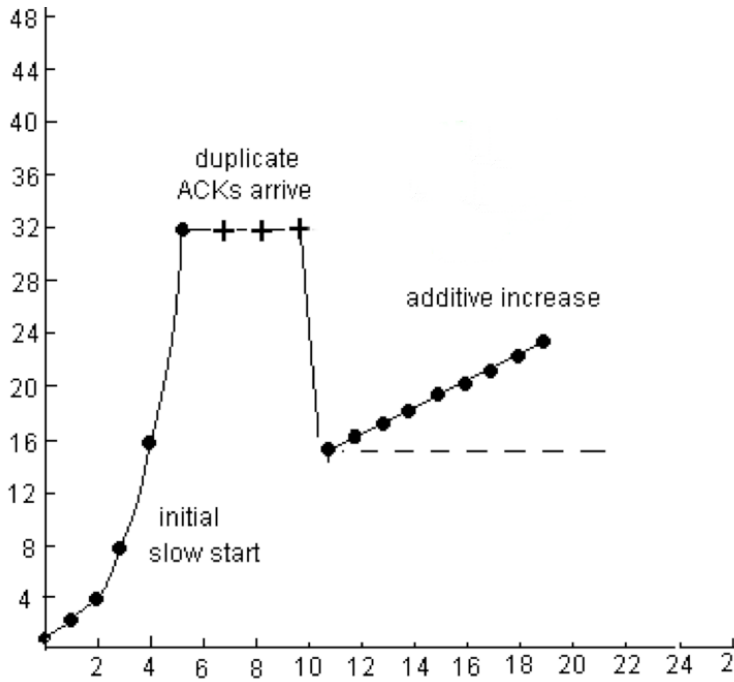
- Slow Start is not well suited when only a single packet is lost…
  - Time-out period often relatively long ➡ Long delay before resending lost packet
  - e.g. brief interference on a wireless link?

- **Fast Retransmit**
  - the receiver sends duplicate ACKs immediately when out-of-order segments arrive
  - if the sender receives **triple duplicate ACKs**, it retransmits the missing segment
    - Hopefully, the acknowledgement for the repeated segment arrives before the timeout thus avoiding a new slow start phase
  - Works well to recover from single packet loss



Fast Retransmit and Recovery (FRR)

- Fast Retransmit can be enhanced to adapt better in case of congestion
- **Fast Recovery**
  - When the third ACK is received, reduce
    ssthresh = max(ssthresh/2, 2×MSS)
  - Retransmit the missing segment, and set
    cwnd = ssthresh + 3*MSS
  - For each subsequent duplicate ACK, increment **cwnd** by 1 MSS
  - If timeout during fast recovery, go back to slowstart
- Rationale: try to skip the slow start phase if unnecessary because congestion is not severe



TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

Summary: TCP Congestion Control

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

- There are several TCP versions, which differ on congestion control:
  - TCP Tahoe: Triple duplicate ACKs followed by slow start (fast retransmit without fast recovery)
  - TCP Reno: Triple duplicate ACKs followed by fast retransmit and fast recovery
  - Other versions exist: Vegas, New Reno, Data Center TCP…

Content of this Section

- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

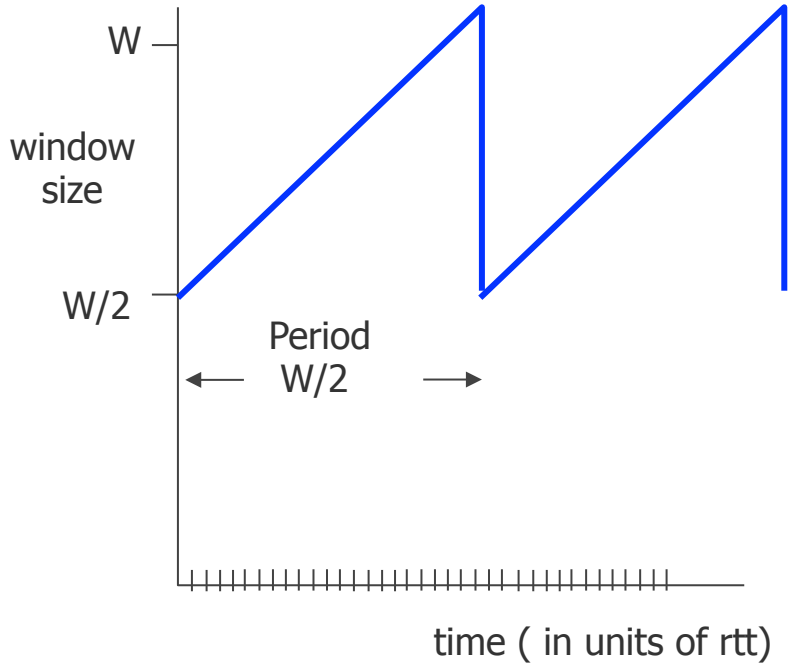
TCP Throughput

- What's the average throughout of TCP given a window size, the MSS, the RTT and the packet loss rate *p*?

Assumptions:

- Simplification: no slow start, only "steadier" congestion avoidance phase
- Infinitely long TCP flow
- Periodic losses

⇒ Renewal process where the window increases from W/2 to W at rate of one segment more per RTT, then a loss occurs and the window falls back to W/2, etc.

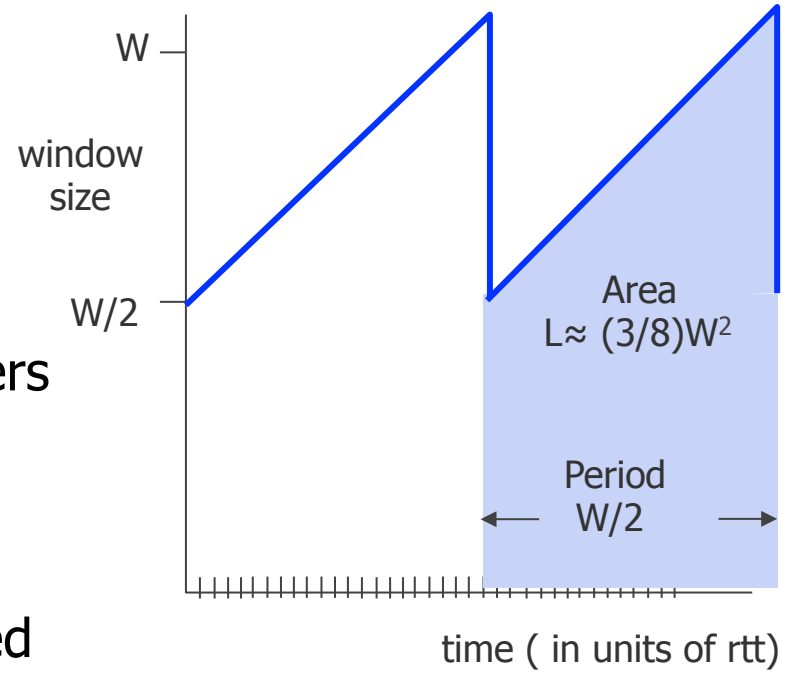


See paper Mathis, Matthew, et al. "The macroscopic behavior of the TCP congestion avoidance algorithm." *ACM SIGCOMM Computer Communication Review* 27.3 (1997): 67-82.

TCP Throughput

With these simplifications, using on renewal process theory, the throughput of TCP is:

$$B = \frac{\text{Avg number of bytes sent per cycle}}{\text{Avg duration of a cycle}}$$



- Remark 1: by definition each period delivers 1/p segments of MSS bytes (remember probablility loss is p with periodic losses)
- Remark 2: total number of segments acked is area under sawtooth, by period that is then $(W/2)^2 + \frac{1}{2}(W/2)^2 = (3/8)W^2 \Rightarrow W = \sqrt{\frac{8}{3p}}$
- So $B = \frac{MSS * \frac{3W^2}{8}}{RTT * \frac{W}{2}} = \frac{\frac{MSS}{p}}{RTT * \sqrt{\frac{2}{3p}}}$ which gives $\boxed{B = \frac{MSS * C}{RTT * \sqrt{p}}}$ with $C = \sqrt{\frac{3}{2}}$

TCP Throughput: Example with "long, fat pipes"

- Example: 1500 byte segments, 100ms RTT, we want 10 Gbit/s throughput

- If packet loss is ignored: B=W/RTT
  - receiver window is the bottleneck, and must be 125 Mbytes
  - corresponds to max 83,333 in-flight segments

- If packet loss is taken into account: $B = \frac{1.22 \cdot MSS}{RTT\sqrt{p}}$

➡One can derive the required max packet loss rate *p*

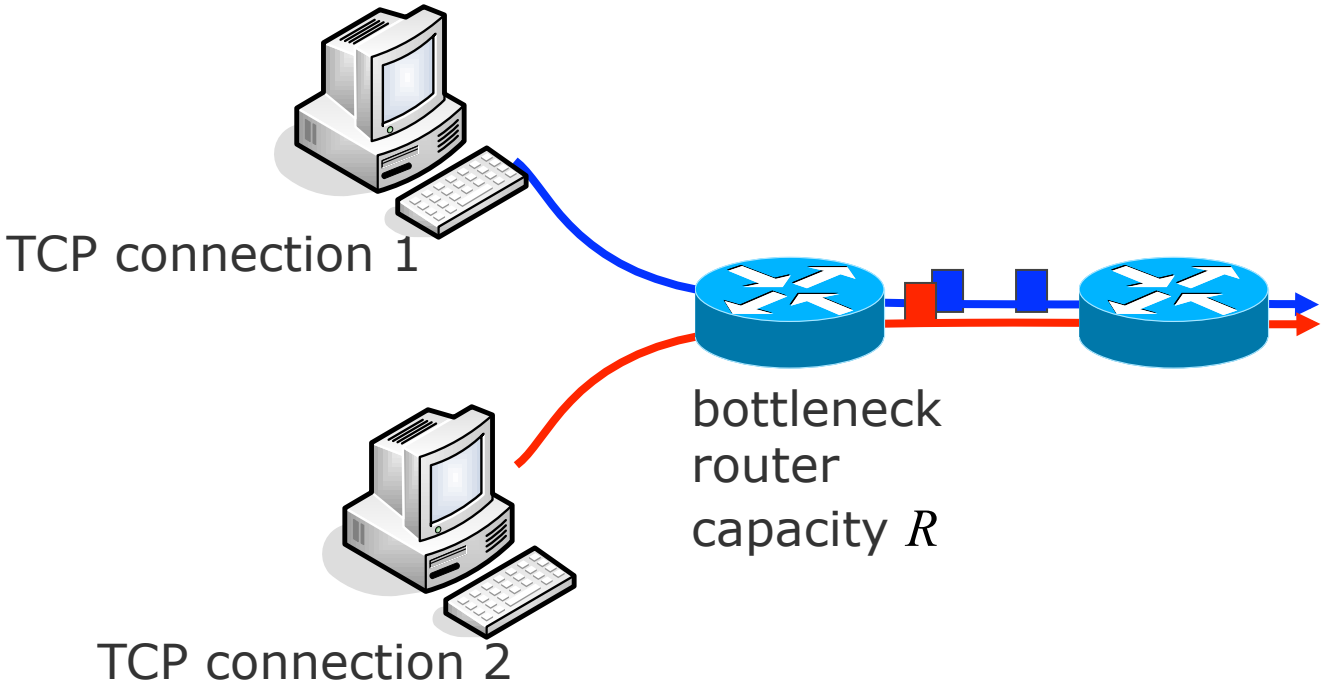
There are limitations to this approach:
- Model assumes long TCP session (most are short in reality)
- Model assumes a single TCP session and "black-box" network
  - e.g. knowledge of RTTs and loss rates as fixed values is unrealistic
- Other models have been designed to overcome these limitations

Content of this Section

- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

TCP Fairness

- General idea: if k flows share same bottleneck link of bandwidth R, each flow should have average rate of R/k
  - Notion of max-min fairness
  - But over which time scale?
  - With which TCP version(s)?

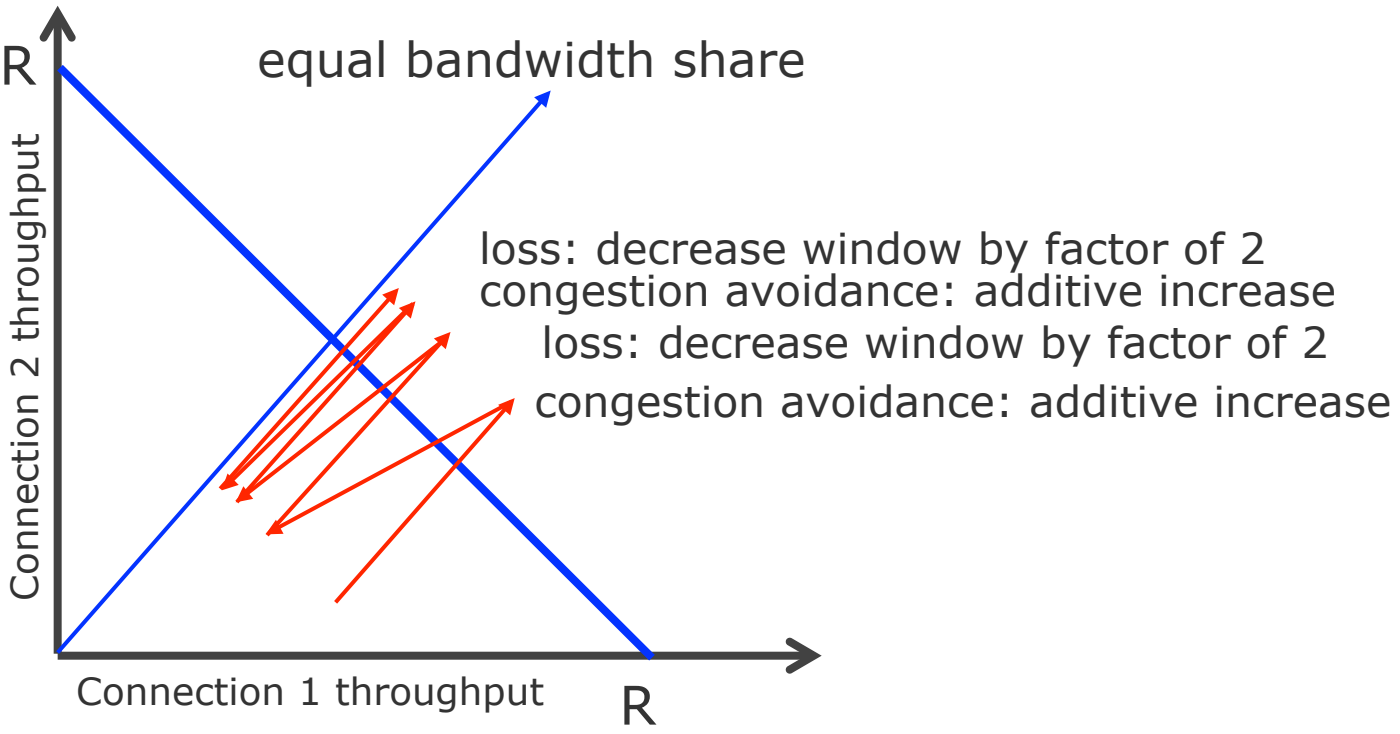


Why is TCP fair? Example of Competing Flows

Example with two competing TCP flows:

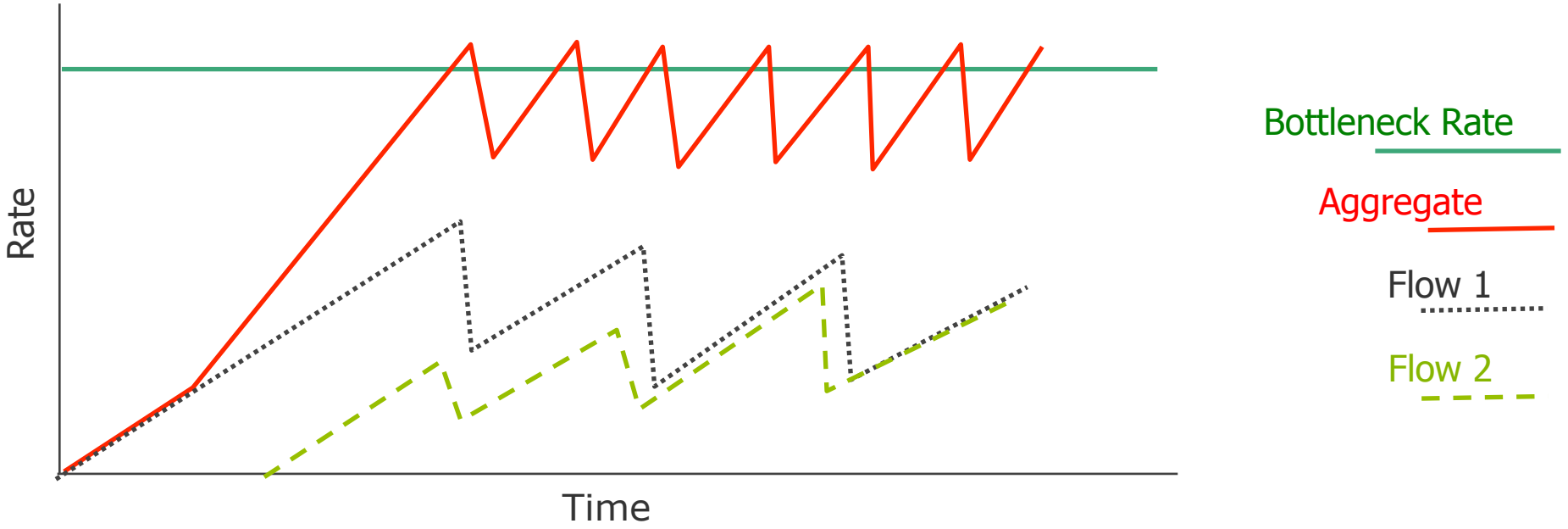
- Below capacity, no loss: Additive increase gives slope of 1, as throughout grows
- Above capacity, loss occurs: Multiplicative decrease diminishes throughput

- Convergence to equal bandwidth sharing



TCP and Aggregate Utility

- Problem: if packet loss is synchronized in different TCP flows, there are times when the capacity is too under-utilized, which is fair to noone
  - Notion of aggregate utility



- Solution: Random Early Detection (RED) in routers.
  - Routers track how their buffers fill up
  - If a buffer threatens to fill up soon, the router begins to drop packets randomly
  - Randomness breaks synchronization, provides better aggregate utility

TCP Fairness Summary

- TCP fairness
  - Vague: formal definition is not easy to express (time scale, TCP versions…)
  - But important: due to the prevalance of TCP (90% of the Internet trafic)
- Max-min fairness
  - Aims to give each session equal access to each link's bandwidth
- Proportional fairness
  - Aims for fairness while maximizing aggregate session utility
- Fairness can be difficult to implement using only end-to-end means
  - e.g., may requires fair queuing etc.
  - Can be easier to implement using in-network packet marking (e.g., ECN)
- Fairness can also be difficult to achieve because of:
  - Cheating: misbehaving TCP flows
  - Cohabitation: with non-TCP trafic e.g. UDP

Fairness: Dealing with Cohabitation & Cheating

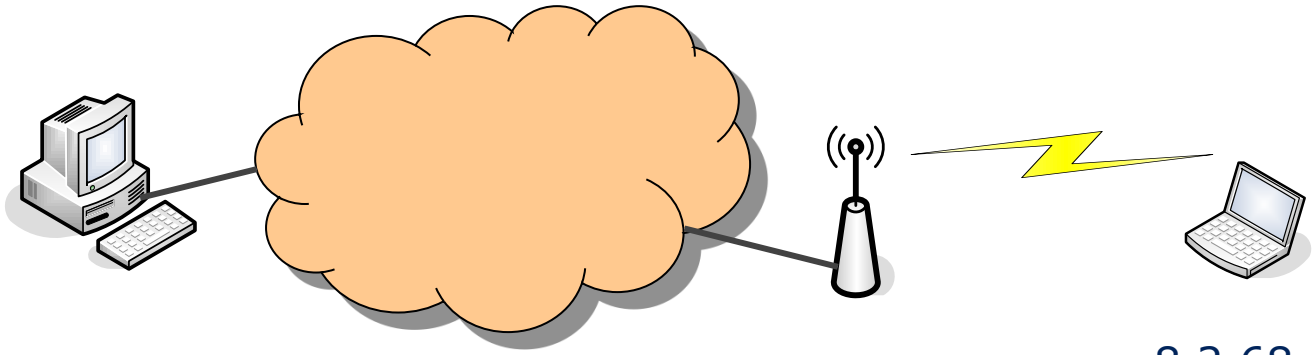
- Fairness and UDP
  - Multimedia apps often don't use TCP
    - do not want rate throttled by congestion control
  - Instead use UDP:
    - pump audio/video at constant rate, tolerate packet loss
  - TCP friendliness
    - Cohabitation with TCP flows?
    - Need to compete in fair manner with TCP congestion control with for example B proportional to $\frac{1}{RTT\sqrt{p}}$
    - e.g. DCCP (later in the chapter)
- Fairness and parallel TCP connections
  - nothing prevents app from opening parallel connections
  - Multiple TCP flows between 2 hosts
  - e.g. web browsers do this
  - If link of rate R supports 9 connections;
    - new app sets up 1 TCP connection, gets rate R/10
    - new app sets up 11 TCP connections, gets R/2
    - how to enforce fairness?

Content of this Section

- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

TCP in Wireless Networks

- The transport layer protocol should be independent of lower layers
  - But TCP is optimized for wired networks
  - e.g. TCP assumes that packet loss is due to congestion in the network
- In wireless networks packet loss occurs due to other causes
  - Handoffs, medium bigger and bursty bit-error rate, bigger delays…
- Performance of TCP in wireless networks is poor
  - e.g., performance of cumulative acking is poor in case of bursty losses
- Approaches to solve the performance problem
  - Split-connection TCP: the end-to-end connection is broken in two parts
  - Modifications of TCP (may lead to backward compatibility issues)
  - New transport protocols

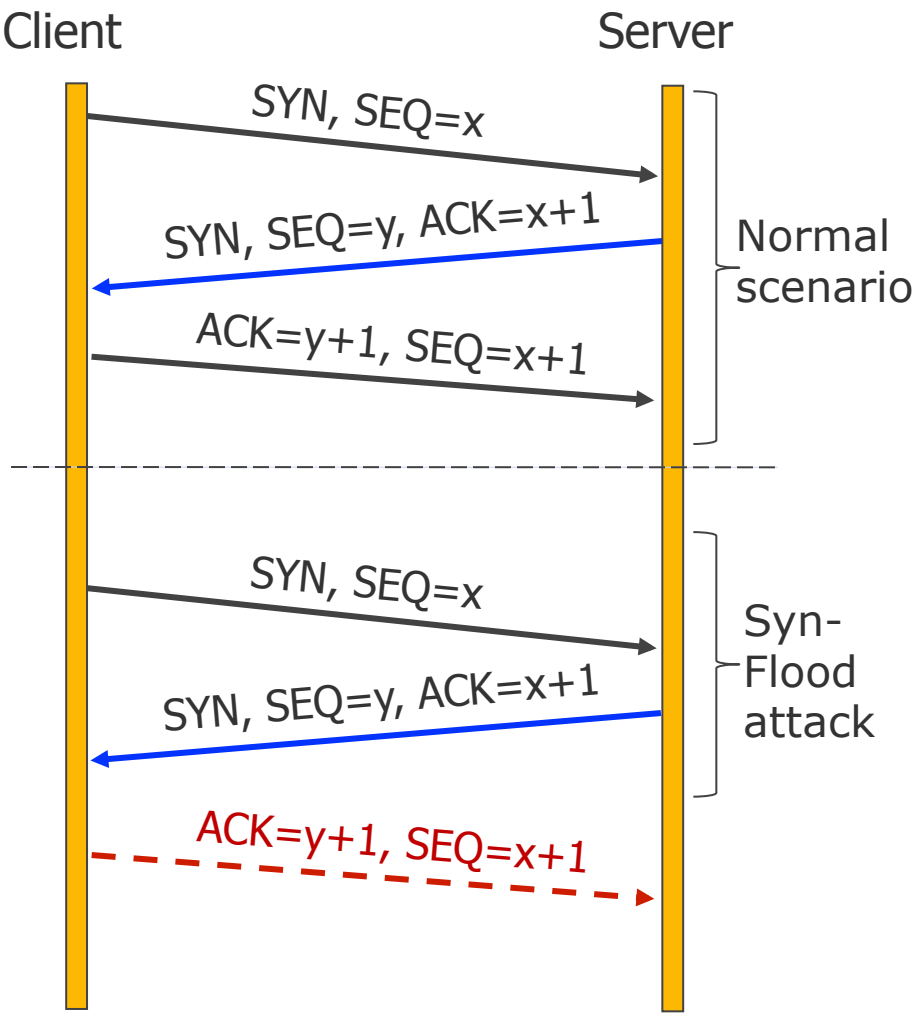


Content of this Section

- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

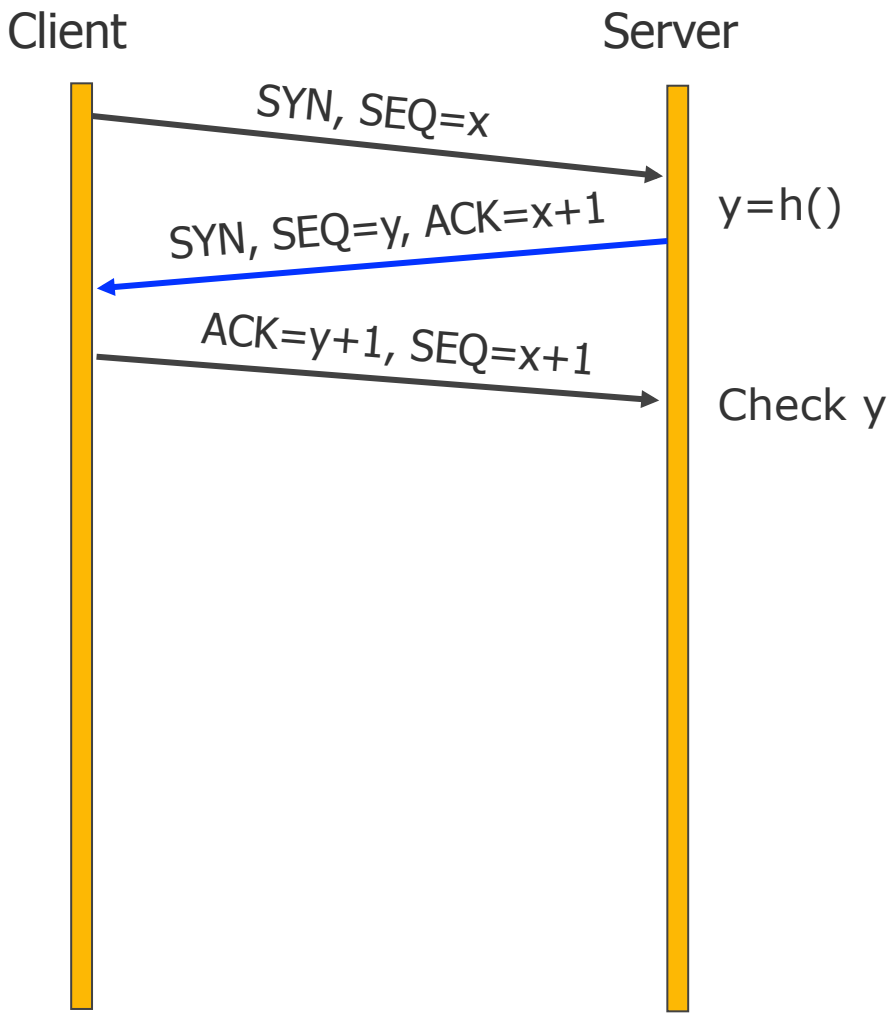
TCP and Security

- Like all network protocol, TCP is vulnerable to attacks, and its security must be assessed
- Example of attack: **SYN Flood**
  - During connection establishment the client does not finish the Three Way Handshake procedure
  - Leaves a half open connection but the server's operating system still reserves resources
  - Can hamper/clog the server if repeated many times: other clients cannot connect to the server.
  - This category of attack is called **Denial of Service** (DoS)



SYN Flood Countermeasure

- Problem: SYN flood achieves DoS
- Solution: SYN cookies
  - The server does not create a half-open connection
  - Server computes an initial sequence number y based on a hash function
    - This is the cookie
  - When client returns with ACK the server recomputes the hash function and checks it
    - For legitimate connection the check will be successful
    - Only then is the connection created



Content of this Section

- TCP Header Format
- TCP Connection Management
- TCP Timer Management
- TCP Reliable Transfer Management
- TCP Flow Control
- TCP Congestion Control
- TCP Throughput
- TCP Fairness
- TCP and Wireless
- TCP and Security
- Tools for TCP

TCP: Some Tools

- netstat: Displays protocol statistics and TCP/IP network connections
  - netstat -n: display IP addresses
  - netstat -b: display executable
  - netstat -r: routing table

```
x:\>netstat -n

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    127.0.0.1:3055         127.0.0.1:3056         ESTABLISHED
  TCP    127.0.0.1:3056         127.0.0.1:3055         ESTABLISHED
  TCP    160.45.114.21:2114     130.133.8.114:80       CLOSE_WAIT
  TCP    160.45.114.21:3027     160.45.113.73:1025     ESTABLISHED
  TCP    160.45.114.21:3029     160.45.113.73:1025     ESTABLISHED
  TCP    160.45.114.21:3043     160.45.113.89:1200     ESTABLISHED
  TCP    160.45.114.21:3362     207.46.108.69:1863     ESTABLISHED
  TCP    160.45.114.21:3704     130.133.8.114:80       CLOSE_WAIT
  TCP    160.45.114.21:3705     130.133.8.114:80       CLOSE_WAIT
  TCP    160.45.114.21:3907     160.45.114.28:139      ESTABLISHED
  TCP    160.45.114.21:3916     160.45.113.100:445     ESTABLISHED
```

TCP: More Tools

- iperf
  - A tool to measure throughput in a network via TCP and UDP stream generation
- TTCP – Test TCP
  - Similar to iperf (measure throughput via TCP/UDP streams)