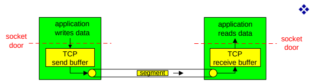
# TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

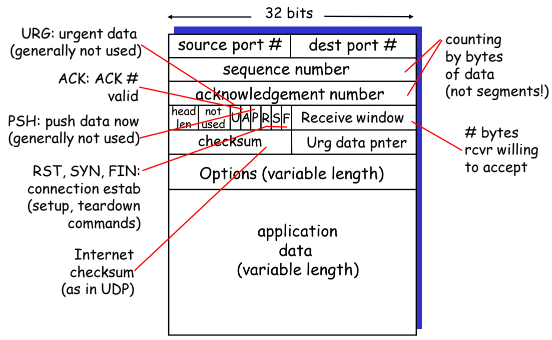
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*




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



# TCP segment structure



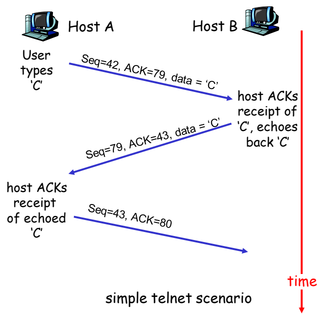



# TCP seq. #'s and ACKs

Seq. #'s:
- byte stream "number" of first byte in segment's data

ACKs:
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor






# TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?
- `SampleRTT`: measured time from segment transmission until ACK receipt
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several recent measurements



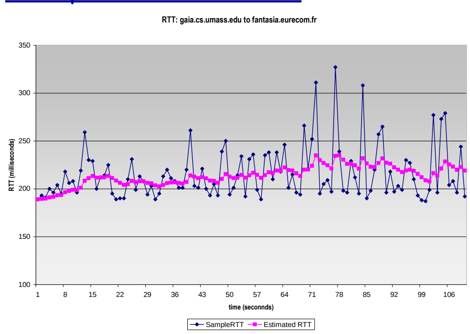
# TCP Round Trip Time and Timeout

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

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



# Example RTT estimation:



# TCP Round Trip Time and Timeout

## Setting the timeout

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

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```



# TCP reliable data transfer

- TCP creates reliable service on top of IP's unreliable service
- pipelined segments
- cumulative acks
- TCP uses single retransmission timer
- retransmissions are triggered by:
  - timeout events
  - duplicate acks



# TCP sender events:

**data rcvd from app:**

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)

**timeout:**

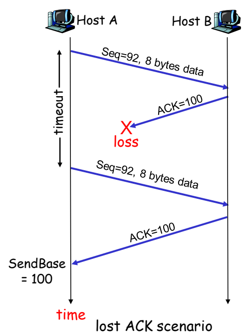
- retransmit segment that caused timeout
- restart timer

**Ack rcvd:**

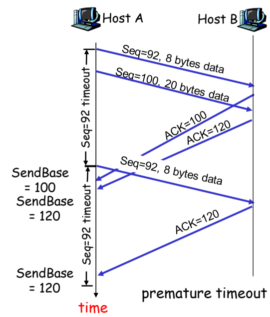
- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments



# TCP: retransmission scenarios



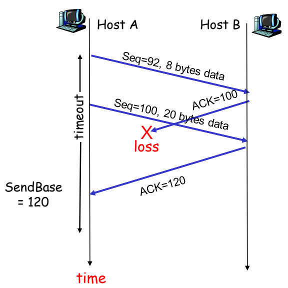

lost ACK scenario




premature timeout



# TCP retransmission scenarios (more)

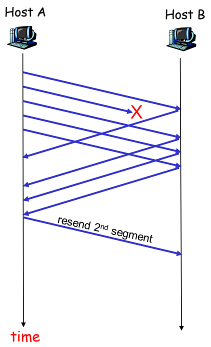



Cumulative ACK scenario



# Fast Retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.
- if sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

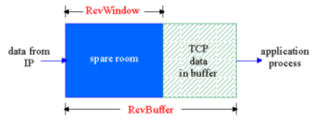



Resending a segment after triple duplicate ACK



# TCP Flow Control

- receive side of TCP connection has a receive buffer:




- app process may be slow at reading from buffer

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

- speed-matching service: matching the send rate to the receiving app's drain rate



# TCP Connection Management

## Three way handshake:

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

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

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



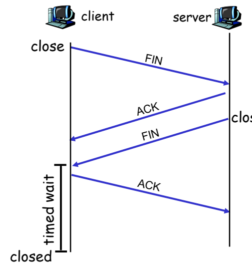
# TCP Connection Management (cont.)

## Closing a connection:

client closes socket:
`clientSocket.close();`

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

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



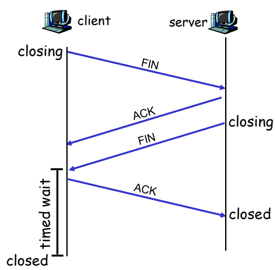



# TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

.






# Principles of Congestion Control

## Congestion:

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

## Approaches towards congestion control

Two broad approaches towards congestion control:

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

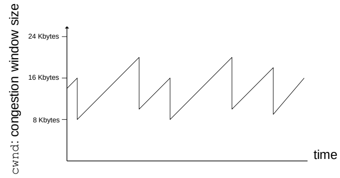
**network-assisted congestion control:**
- routers provide feedback to end systems



## TCP congestion control: additive increase, multiplicative decrease

- *approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **cwnd** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **cwnd** in half after loss

saw tooth behavior: probing for bandwidth






## TCP Congestion Control: details

- sender limits transmission:
  **LastByteSent-LastByteAcked ≤ cwnd**
- roughly,

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

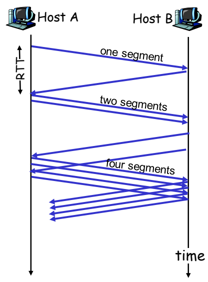
- **cwnd** is dynamic, function of perceived network congestion

How does sender discover congestion?
- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (**cwnd**) after loss event



## TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received






## Refinement: inferring loss

- after 3 dup ACKs:
  - **cwnd** is cut in half
  - window then grows linearly
- <u>but</u> after timeout event:
  - **cwnd** instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

Philosophy:
- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario