# Connection-oriented Transport: TCP

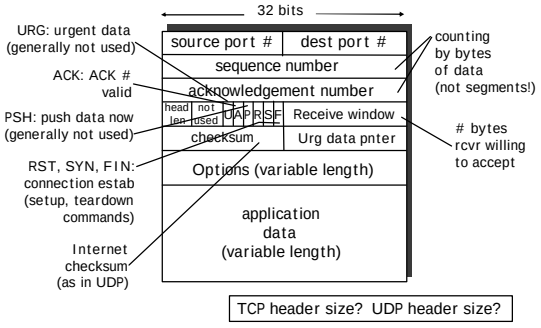
Outline:
- segment structure
- reliable data transfer
- flow control
- connection management

## TCP: Overview

- point-to-point:
  - one sender, one receiver
- reliable data transfer:
  - guaranteed arrival, no error, in order
- pipelined:
  - multiple in-flight segments
- full duplex data:
  - bi-directional data flow in same connection
- connection-oriented:
  - handshaking (exchange of control msgs) to initialize sender, receiver state before data exchange
- flow controlled:
  - sender does not overwhelm receiver
- congestion controlled:
  - sender does not overwhelm the network
- no delay or bandwidth guarantee.



## TCP Segment Structure




TCP header size? UDP header size?



## Outline

- segment structure
- reliable data transfer
- flow control
- connection management



## TCP Reliable Data Transfer

- TCP provides reliable data transfer service on top of IP's unreliable service
- Pipelined transmissions & cumulative ACKs
- Looks like Go-back-N, however the receiver could buffer out-of-order segments
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate ACKs



## TCP Timeout

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

## Estimating TCP Round Trip Time

Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
- **SampleRTT** fluctuates, we want estimated RTT "smoother" to avoid short-term spikes
  - average several recent measurements, not just current **SampleRTT**
- we also want to give more recent measurements higher weight in case things do change



## EWMA – Exponentially Weighted Moving Average

$$\begin{aligned}\text{EstimatedRTT} &= \alpha*\text{SampleRTT}_1 \\ &+ \alpha*(1-\alpha)*\text{SampleRTT}_2 \\ &+ \alpha*(1-\alpha)^2*\text{SampleRTT}_3 \\ &+ \alpha*(1-\alpha)^3*\text{SampleRTT}_4 \\ &+ \ldots\ldots\end{aligned}$$

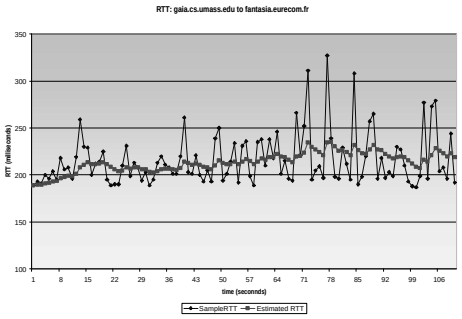
$\text{SampleRTT}_1$ is the most recent sample TTL,
$\text{SampleRTT}_2$ is the next recent sample TTL, etc.

Þ $\text{EstimatedRTT} = (1-\alpha)*\text{EstimatedRTT}_{last} + \alpha*\text{SampleRTT}_1$

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



## Example RTT Estimation






## TCP Timeout

Setting the timeout:

- **EstimtedRTT** plus "safety margin"
  - large variation in **EstimatedRTT ->** larger safety margin
- we need to estimate of how much SampleRTT deviates from EstimatedRTT (EWMA):

$$\text{DevRTT} = (1-\beta)*\text{DevRTT}_{last} + \beta*|\text{SampleRTT}-\text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

Then set timeout interval:

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



## TCP Sender Events and Processing

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 un-ACKed segment)
- timeout vale: we just decided it!!

timeout:

- retransmit segment that caused timeout
- restart timer

ACK rcvd:

- If acknowledges previously unacked segments
  - slide sender window
  - start timer if there are outstanding segments



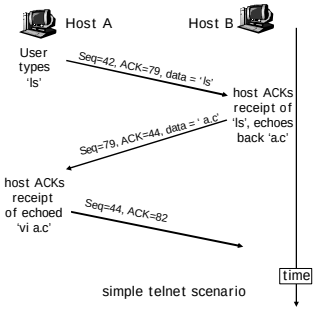
## TCP byte-oriented seq. #'s and ACKs

Seq. #'s:

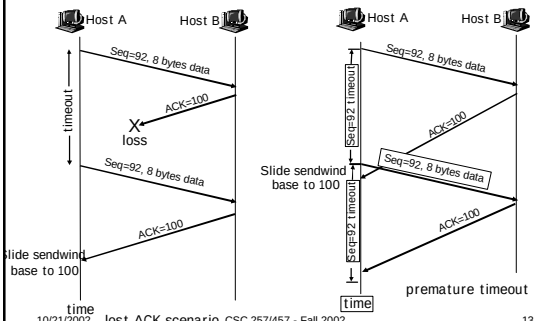
- byte stream "number" of first byte in segment's data

ACKs:

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

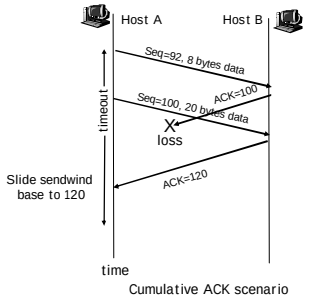


## TCP in Action: Retransmissions






## TCP in Action: Cumulative ACK



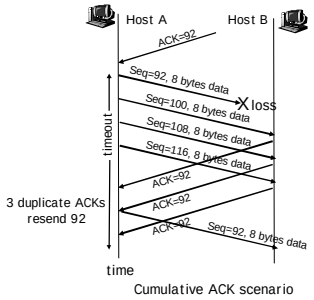



## Fast Retransmission

- 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.
- When receiver receives out-of-order segments, it re-ACK the last in-order byte
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmission: resend segment before timer expires, restart timer



## TCP in Action: Duplicate ACKs and Fast Retransmission





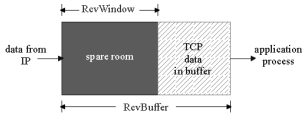

## Outline

- segment structure
- reliable data transfer
- flow control
- connection management



## TCP Flow Control

- receive side of TCP connection has a receive buffer:

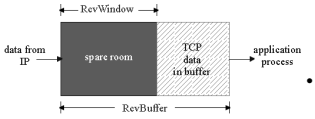



- app process may be slow at reading from buffer

flow control: sender does not 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 Flow Control: how it works




(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

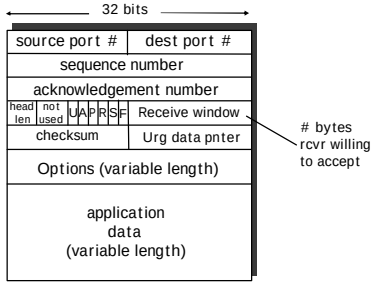
**= RcvWindow**

**= RcvBuffer-[LastByteRcvd - LastByteRead]**

- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - guarantees receive buffer doesn't overflow



## Where is **RcvWindow** in each segment?






## Outline

- segment structure
- reliable data transfer
- flow control
- connection management



## TCP Connection Management

- Establishment:
  - TCP sender, receiver establish "connection" before exchanging data segments
  - initialize TCP variables: starting seq. #s, MSS, buffers, flow control info (e.g. **RcvWindow**)
- MSS is the maximum TCP segment size each side is willing to accept
  - typically the largest segment size fit into a link-layer frame
  - What is MSS for a IPv4 host connected with Ethernet?
- Teardown:
  - freeing up resources after mutually close



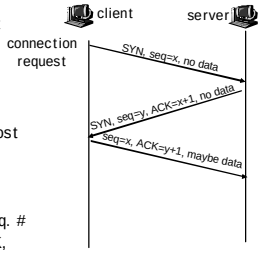
## TCP Connection Establishment

Three way handshake:

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

Step 2: server (passive open) 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 Teardown

Closing a connection:

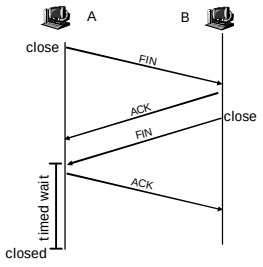
close socket: **close(sockfd);**

Step 1: A (active closing host) sends TCP FIN control segment to server

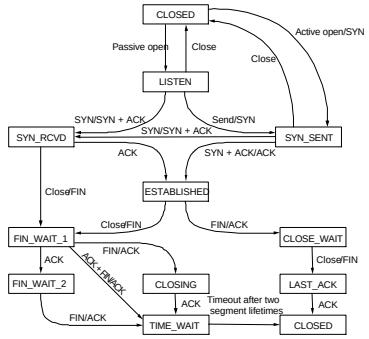
Step 2: B (passive closing host) receives FIN, replies with ACK. Closes connection, sends FIN.

Step 3: A receives FIN, replies with ACK.
- Enters "timed wait" – resend ACK in case it is lost

Step 4: B receives ACK. Connection closed.



## TCP State Transition Diagram






## Disclaimer

- Parts of the lecture slides contain original work of James Kurose, Larry Peterson, and Keith Ross. The slides are intended for the sole purpose of instruction of computer networks at the University of Rochester. All copyrighted materials belong to their original owner(s).