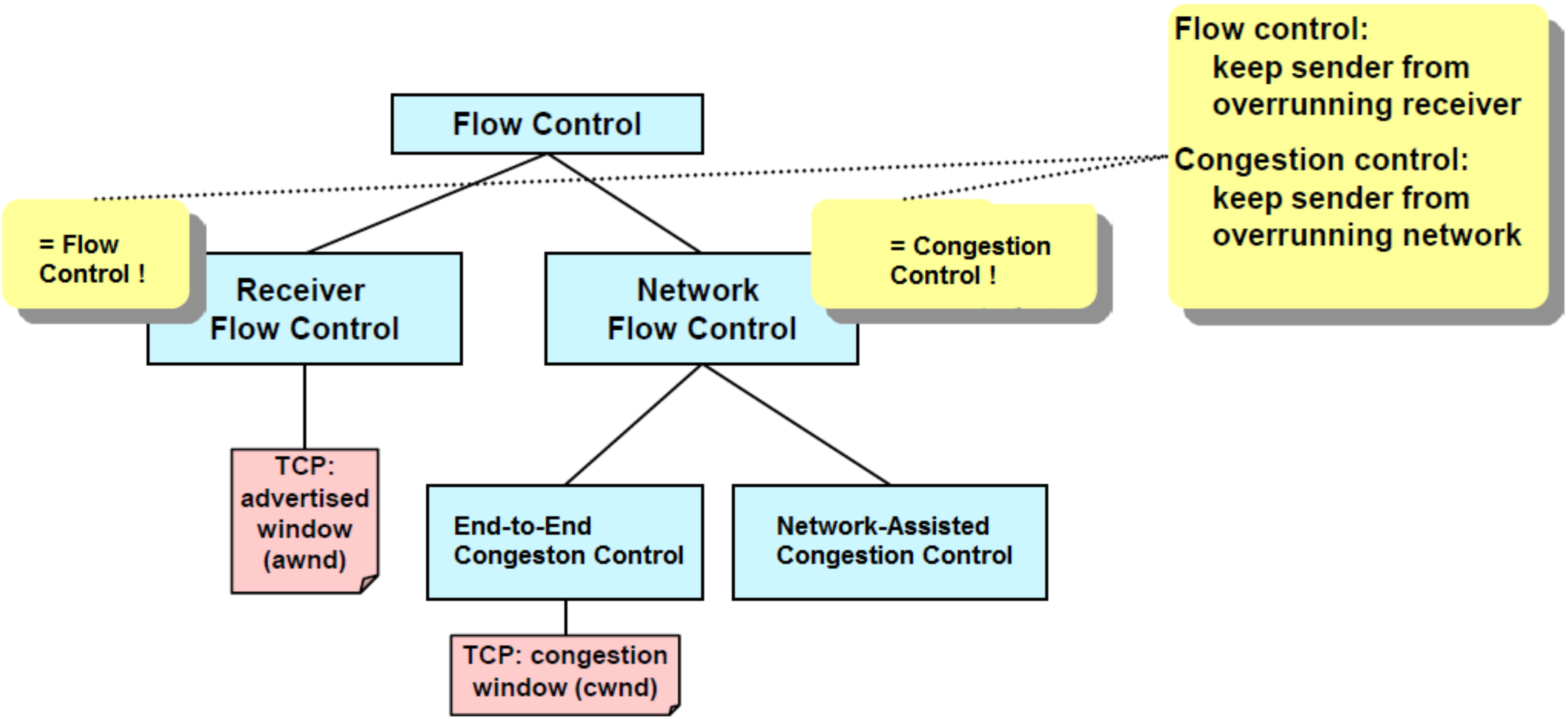
TCP Congestion Control (Part III)

Assist.Prof.Dr. Fatih ABUT

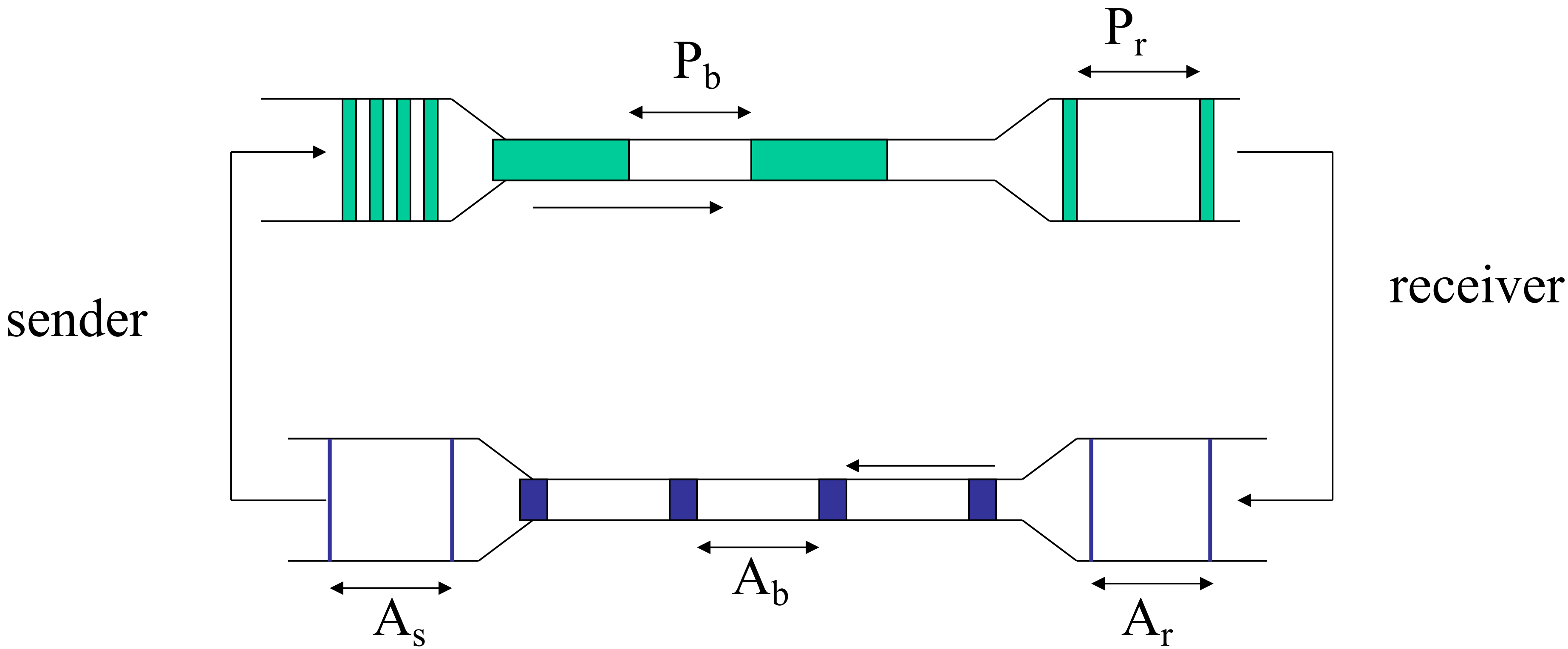
Contents

- Introduction
- Slow-start
- Congestion avoidance
  - Additive Increase
  - Multiplicative Decrease
- Fast retransmit
- Fast recovery
- TCP Fairness

Flow Control: Overview



TCP Self-Clocking Principle



P_b : the minimum packet spacing (the inter-packet interval) on the bottleneck link

P_r : the receiver's network packet spacing [$P_b = P_r$]

A_r : the spacing between acks on the receiver's network

[if the processing time is the same for all packets, $P_b = P_r = A_r$]

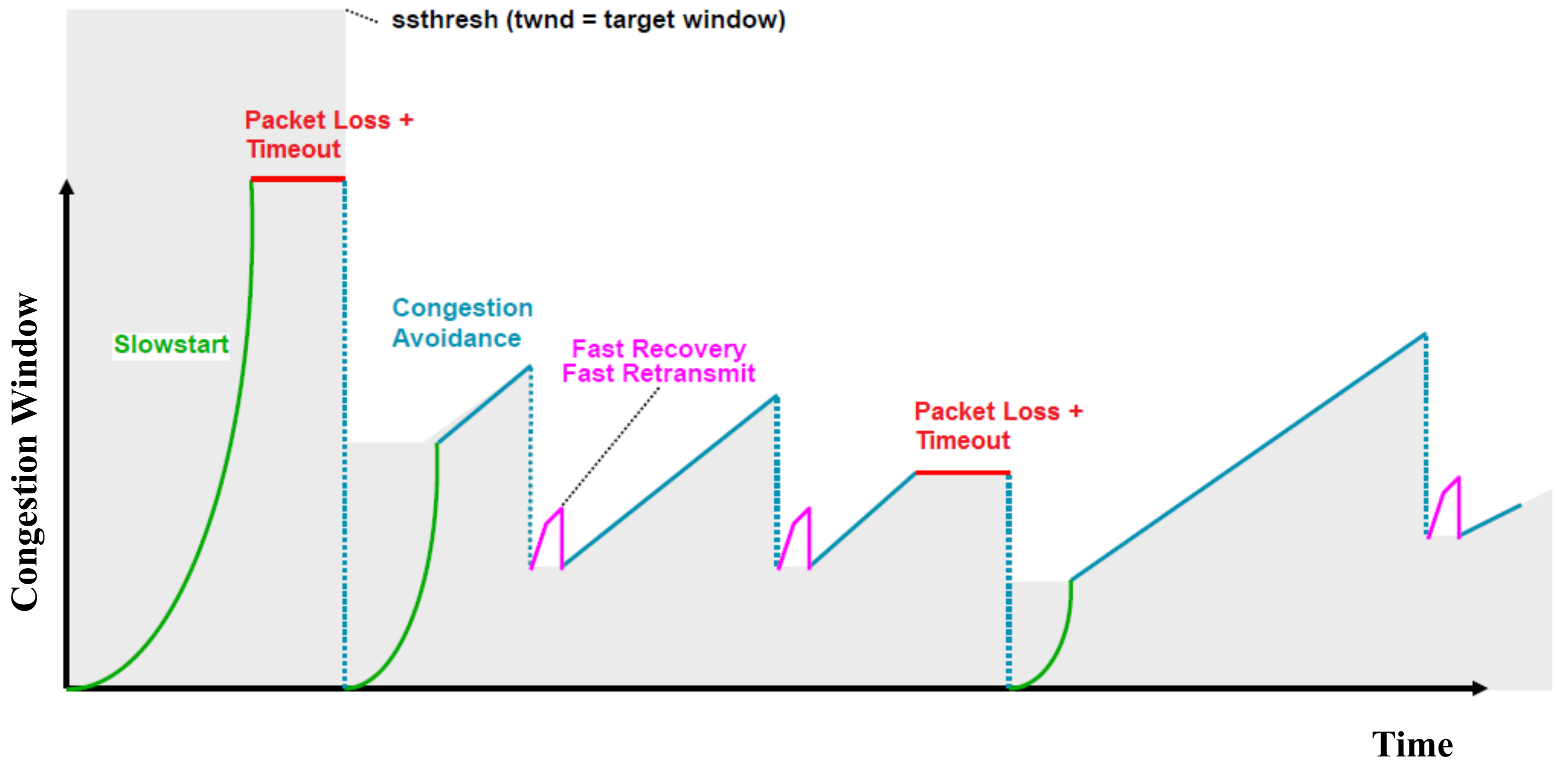
A_b : the ack spacing on the bottleneck link

A_s : the ack spacing on the sender's network [$A_s = P_b$]

A collection of collaborating mechanisms :

- Accurate Retransmission Timeout Estimation
- Slow-Start
- Congestion Avoidance / Multiplicative Decrease
- Fast Retransmit
- Fast Recovery

Typical “Sawtooth" Pattern



Slow Start (1)

- The source starts with cwnd = 1.
- Every time an ACK arrives, cwnd is incremented.

➔ cwnd is effectively doubled per RTT “epoch”.

- Three slow start situations:
  - At the very beginning of a connection **{cold start}**.
  - After retransmission timeout
  - After long passive TCP phase

Slow Start

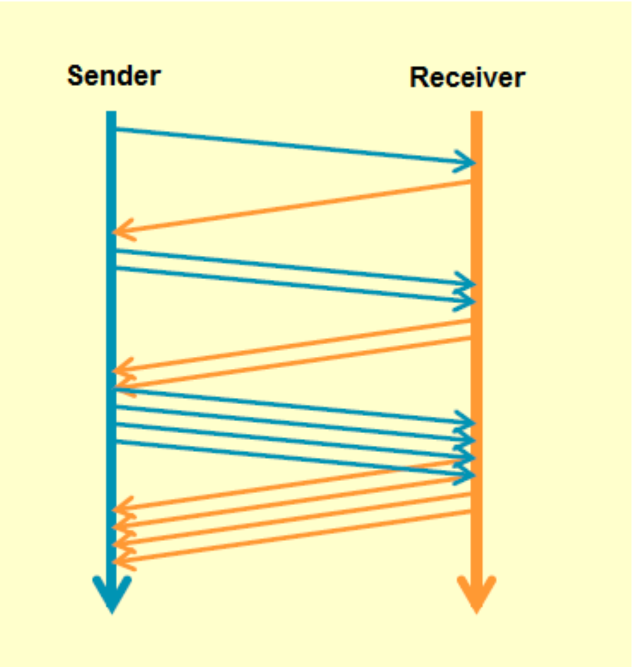
<u>cwnd</u> = SMSS

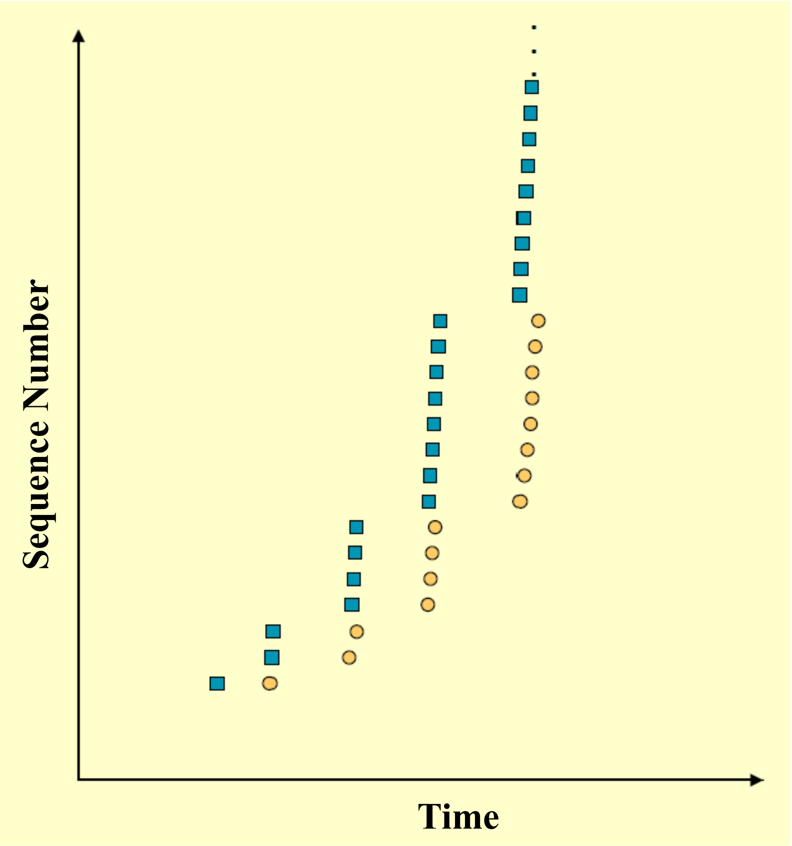
<u>FOR</u> <u>EACH</u> received(ACK)

    cwnd = cwnd + SMSS

<u>UNTIL</u> cwnd >= twnd <u>OR</u> RTO

Slow Start (2)





Congestion Avoidance (AIMD = Additive Increase / Multiplicative Decrease)

- CongestionWindow (cwnd) is a variable held by the TCP source for each connection.

MaxWindow = min (**CongestionWindow** , AdvertisedWindow)

EffectiveWindow = MaxWindow – Unacknowledged Segments

- cwnd is set based on the perceived level of congestion. The Host receives *implicit* (packet drop) or *explicit* (packet mark) indications of internal congestion.

Additive Increase

- Additive Increase is a reaction to perceived available capacity.
- Linear Increase basic idea:: For each "cwnd's worth" of packets sent, increase cwnd by 1 packet.
- In practice, cwnd is incremented fractionally for each arriving ACK.

Multiplicative Decrease

* The key assumption is that a dropped packet and the resultant timeout are due to congestion at a router or a switch.

  Multiplicate Decrease: TCP reacts to a timeout by halving cwnd.

- Although cwnd is defined in bytes, the literature often discusses congestion control in terms of packets (or more formally in MSS == Maximum Segment Size).
- cwnd is not allowed below the size of a single packet.

Congestion Avoidance Algorithm (1)

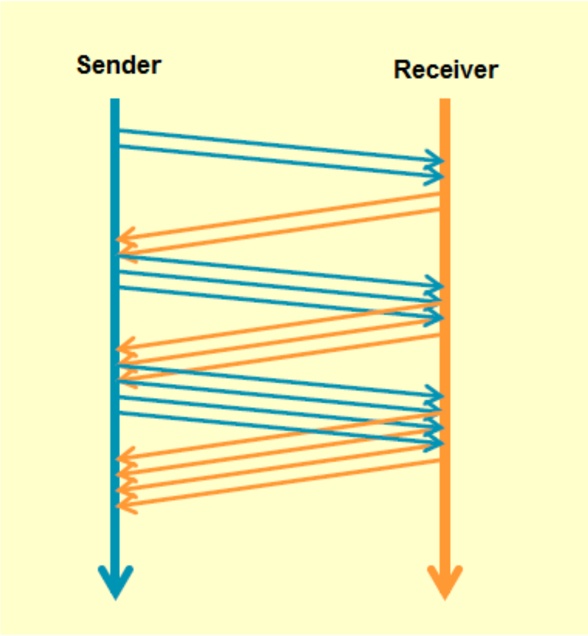
Congestion Avoidance

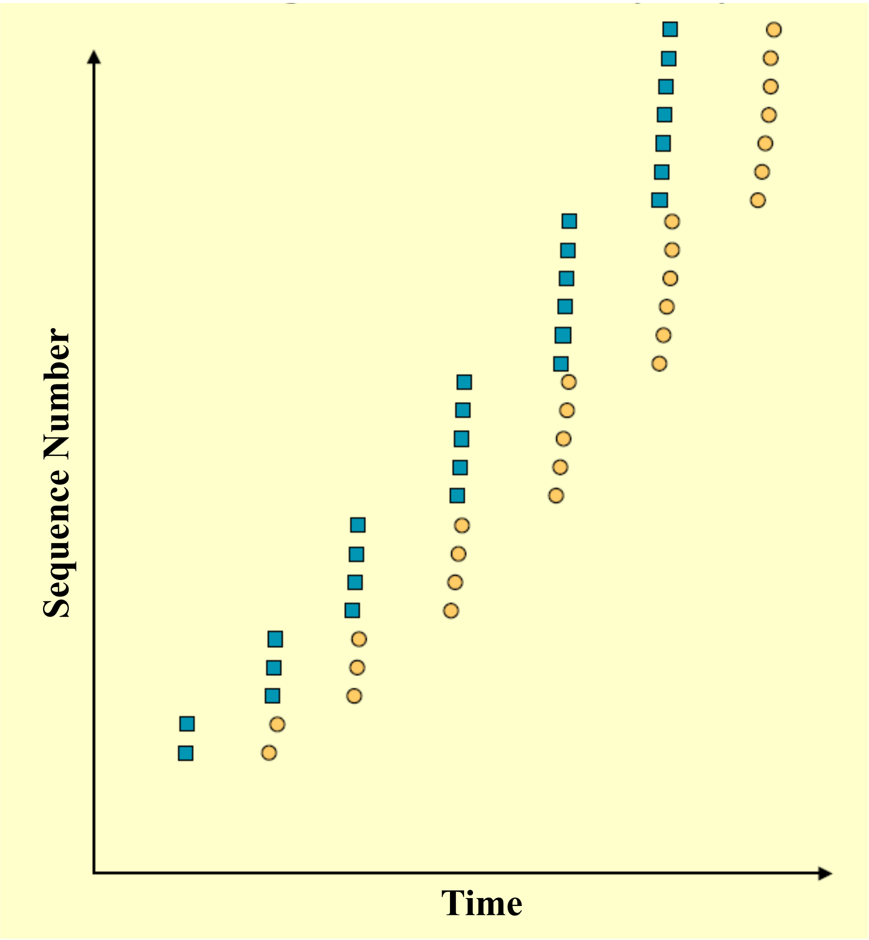
```
UNTIL LossEvent
  FOR EACH received(ACK)
    cwnd = cwnd + SMSS * (SMSS/cwnd)
  ENDFOR
ENDUNTIL
twnd := cwnd/2

IF intelligent
  Perform Fast Retransmit or Fast Recovery
ELSE
  cwnd :=1
  Perform SlowStart
ENDIF
```

Loss Event can be triggered by three dupACKs or RTO

Congestion Avoidance Algorithm (2)





Fast Retransmit

- Coarse timeouts remained a problem, and Fast retransmit was added with TCP Tahoe.
- Since the receiver responds every time a packet arrives, this implies the sender will see duplicate ACKs.

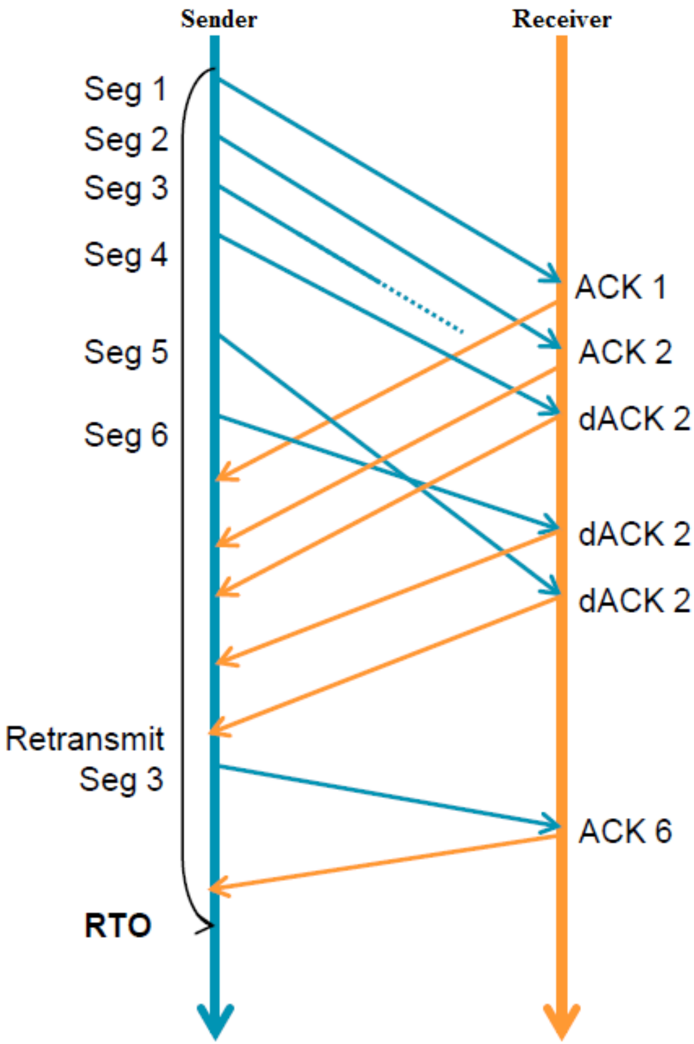
Basic Idea:: *use* ***duplicate ACKs*** *to signal lost packet.*

Fast Retransmit

Upon receipt of *three* duplicate ACKs, the TCP Sender retransmits the lost packet.

Fast Retransmit

- Generally, fast retransmit eliminates about half the coarse-grain timeouts.
- This yields roughly a 20% improvement in throughput.
- Note – fast retransmit does not eliminate all the timeouts due to small window sizes at the source.



Fast Retransmit

Based on <u>three</u> duplicate ACKs

Fast Recovery (1)

- Fast recovery was added with TCP Reno.
- Basic idea: When fast retransmit detects three duplicate ACKs, start the recovery process from congestion avoidance region and use ACKs in the pipe to pace the sending of packets.

Fast Recovery

After Fast Retransmit, half cwnd and commence recovery from this point using linear additive increase 'primed' by left over ACKs in pipe.

Fast Recovery (2)

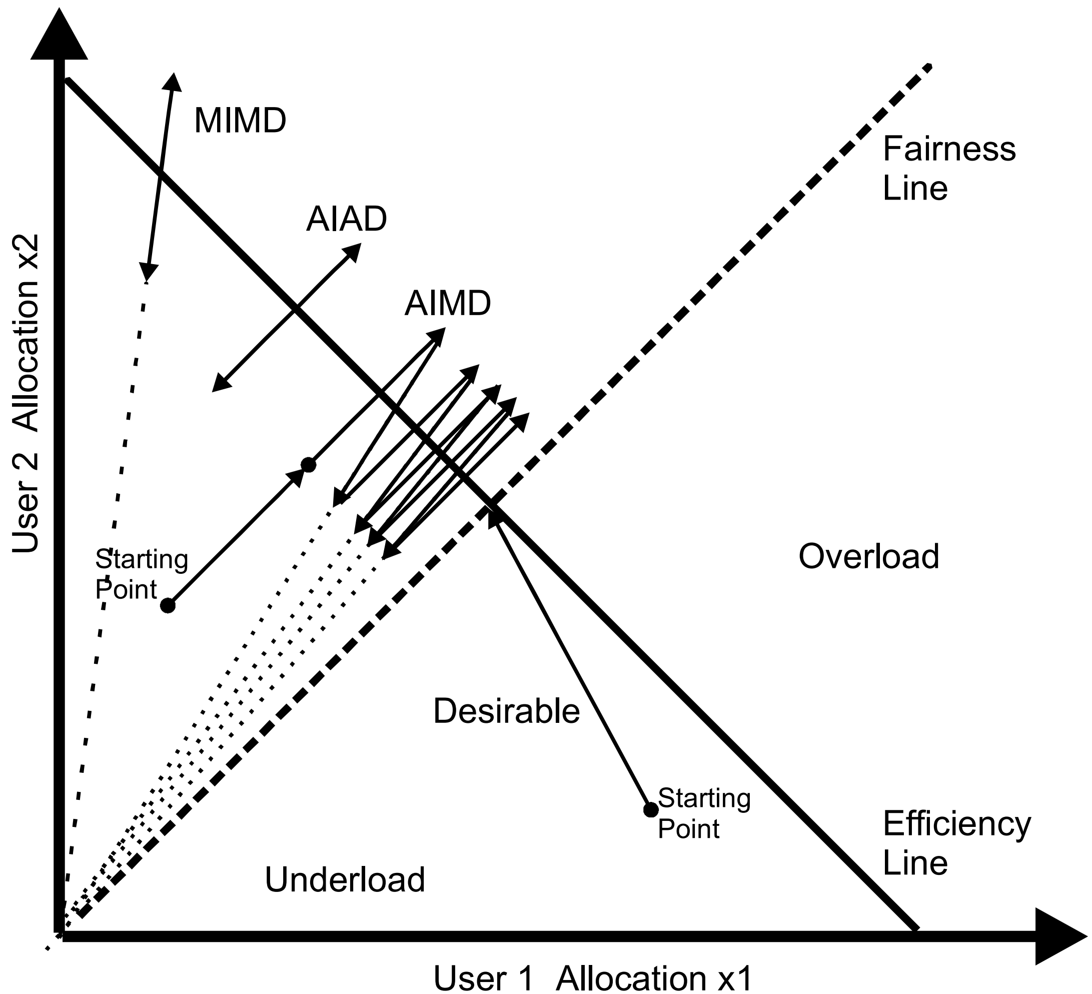
Fast Recovery

```
AS-SOON-AS #dACK=3
   twdn := 0.5 * cwdn  -- multiplik. decrease
   cwnd := twnd + 3 * SMSS -- inflating
   send(LostSegment)
END AS-SOON-AS

FOR-EACH-ADDITIONAL dACK
   cwnd := cwnd + SMSS
END-FOR-EACH ADDITIONAL

AS-SOON-AS regular-ACK
   cwnd := twnd -- deflating
END AS-SOON-AS
```

TCP Fairness



UDP: User Datagram Protocol [RFC 768]

- “bare bones”, “best effort” transport protocol
- *connectionless:*
  - no handshaking between UDP sender, receiver before packets start being exchanged
  - each UDP segment handled independently of others
- Just provides multiplexing/demultiplexing

Pros:

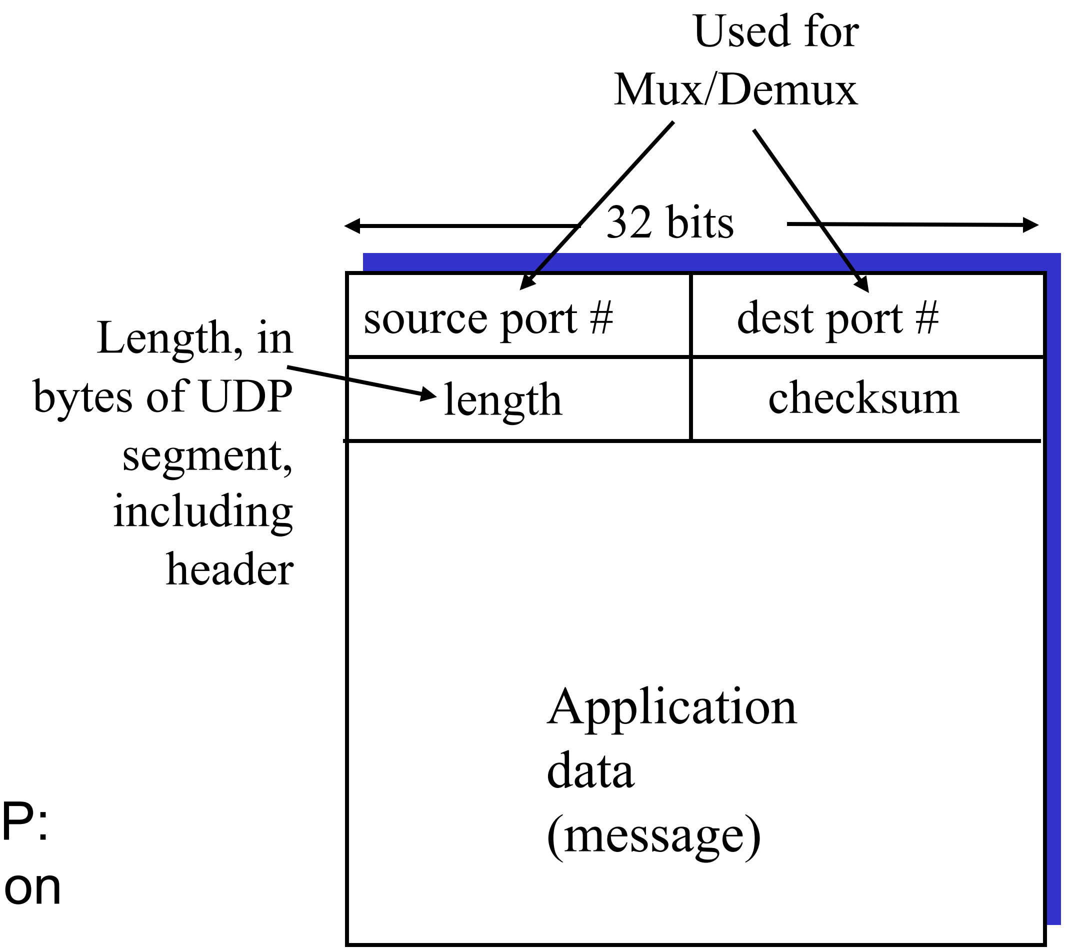
- No connection establishment
  - No delay to start sending/receiving packets
- Simple
  - no connection state at sender, receiver
- Small segment header
  - Just 8 bytes of header

Cons:

- “best effort” transport service means, UDP segments may be:
  - lost
  - delivered out of order to app
- no congestion control: UDP can blast away as fast as desired

UDP more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!



UDP segment format