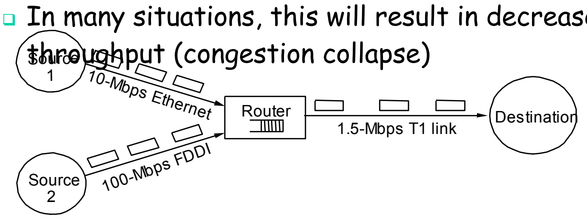
Advanced Computer Networks

TCP Congestion Control

Thanks to Kamil Sarac

What is congestion?

- *Increase in network load results in decrease of useful work done*
  - Different sources compete for resources inside network
  - Why is it a problem?
    - Sources are unaware of current state of resource
    - Sources are unaware of each other
    - In many situations, this will result in decrease in throughput (congestion collapse)



Issues

- How to deal with congestion?
  - pre-allocate resources so as to avoid congestion *(avoidance)*
  - control congestion if (and when) it occurs *(control)*
- Two points of implementation
  - hosts at the edges of the network (transport protocol)
  - routers inside the network (queuing discipline)
- Underlying service model
  - best-effort data delivery

TCP Congestion Control

- Idea
  - assumes best-effort network (FIFO or FQ routers)
  - each source determines network capacity for itself
  - uses implicit feedback
  - ACKs pace transmission (*self-clocking*)
- Challenge
  - determining the available capacity in the first place
  - adjusting to changes in the available capacity

TCP Congestion Control

- TCP sender is in one of two states:
  - slow start OR congestion avoidance
- Three components of implementation

  Original TCP (TCP Tahoe)
  - 1. Slow Start

    2. Additive Increase Multiplicative Decrease (AIMD)

    3. Fast Retransmit
- TCP Reno
  - 3. Fast Recovery
- TCP Vegas
  - Introduces Congestion Avoidance

TCP Congestion Control

- Objective: adjust to changes in the available capacity
- New state variables per connection: **CongestionWindow and (slow start) threshold**
  - limits how much data source has in transit

```
MaxWin = MIN(CongestionWindow,
             AdvertisedWindow)
EffWin = MaxWin - (LastByteSent -
                   LastByteAcked)
```

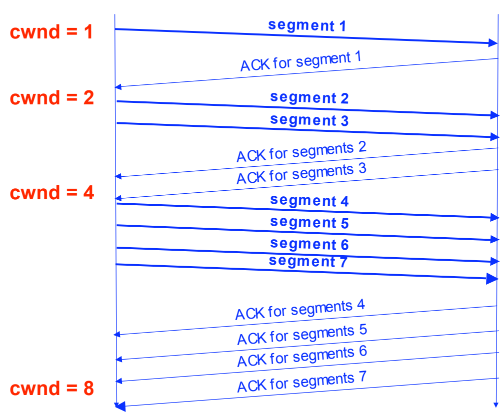
Slow Start

- Initial value: Set **cwnd = 1**
  - Note: Unit is a segment size. TCP actually is based on bytes and increments by 1 MSS (maximum segment size)
- The receiver sends an acknowledgement (ACK) for each packet
  - Note: Generally, a TCP receiver sends an ACK for every other segment.
- Each time an ACK is received by the sender, the congestion window is increased by 1 segment:

  cwnd = cwnd + 1
  - If an ACK acknowledges two segments, cwnd is still increased by only 1 segment.
  - Even if ACK acknowledges a segment that is smaller than MSS bytes long, cwnd is increased by 1.

Slow Start Example

- The congestion window size grows very rapidly
  - For every ACK, we increase cwnd by 1 irrespective of the number of segments ACK'ed
- TCP slows down the increase of *cwnd* when *cwnd > ssthresh*



Congestion Avoidance via AIMD

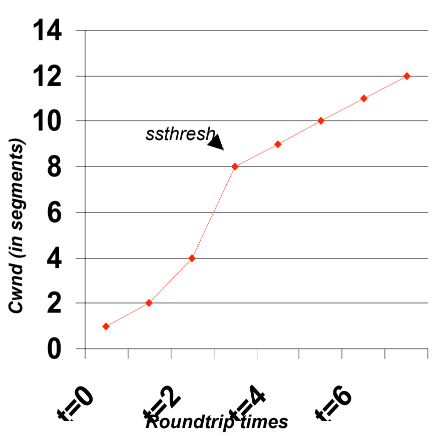
- Congestion avoidance phase is started if cwnd has reached the slow-start threshold value

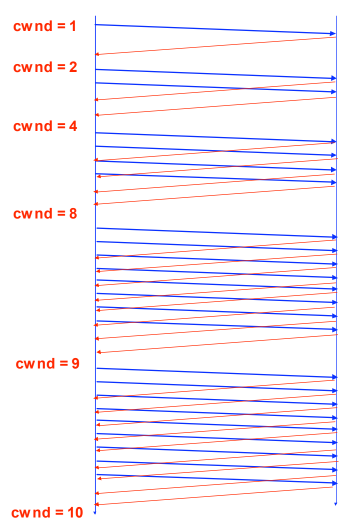
- If cwnd >= ssthresh then each time an ACK is received, increment cwnd as follows:
  - cwnd = cwnd + 1/ cwnd

- So *cwnd* is increased by one only if all *cwnd* segments have been acknowledged.

Example of Slow Start/Congestion Avoidance

Assume that *ssthresh = 8*





Responses to Congestion

- So, TCP assumes there is congestion if it detects a packet loss
- A TCP sender can detect lost packets via:
  - Expiration of a retransmission timer
  - Receipt of a duplicate ACK (why?)
- TCP interprets a Timeout as a binary congestion signal. When a timeout occurs, the sender performs:
  - cwnd is reset to one:

    cwnd = 1
  - ssthresh is set to half the current size of the congestion window:

    ssthresh = cwnd / 2

Summary of TCP congestion control

Initially:

cwnd = 1;

ssthresh =
advertised window size;

New Ack received:

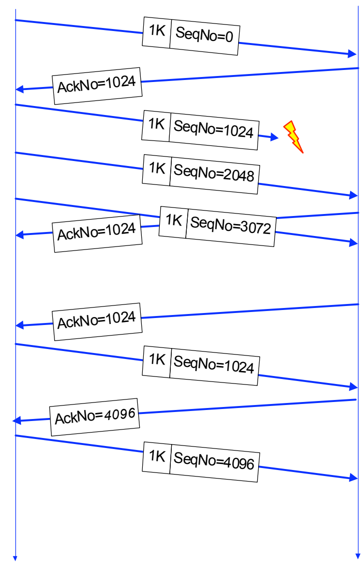
```
if (cwnd < ssthresh)
    /* Slow Start*/
    cwnd = cwnd + 1;
else
    /* Cong. Avoidance */
    cwnd = cwnd + 1/cwnd;
```

Timeout:

```
/* Multiplicative decrease */
ssthresh = cwnd/2;
```

Fast Retransmit

- If three or more duplicate ACKs are received in a row, the TCP sender believes that a segment has been lost.
- Then TCP performs a retransmission of what seems to be the missing segment, without waiting for a timeout to happen.
- Enter slow start:
  ssthresh = cwnd/2
  cwnd = 1



Flavors of TCP Congestion Control

- **TCP Tahoe** (1988, FreeBSD 4.3 Tahoe)
  - Slow Start
  - Congestion Avoidance
  - Fast Retransmit
- **TCP Reno** (1990, FreeBSD 4.3 Reno)
  - Fast Recovery
- **New Reno** (1996)
- **SACK** (1996)

TCP Reno

- Duplicate ACKs:
  - Fast retransmit
  - Fast recovery

  → Fast Recovery avoids slow start

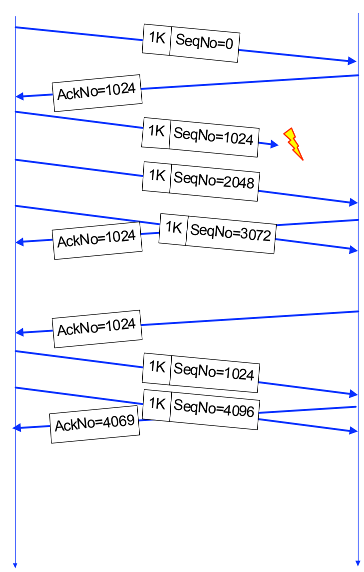
- Timeout:
  - Retransmit
  - Slow Start

- TCP Reno improves upon TCP Tahoe when a single packet is dropped in a round-trip time.

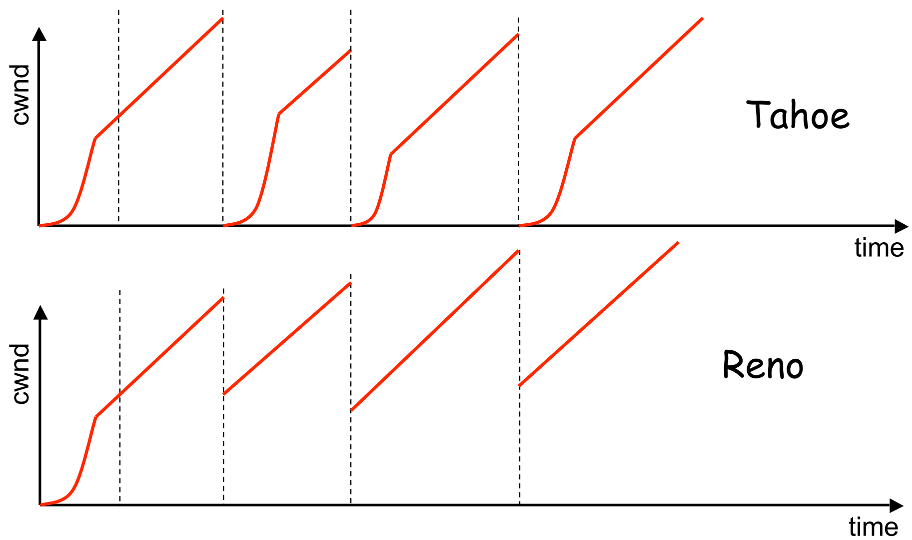
Fast Recovery

- Fast recovery avoids slow start after a fast retransmit
- **Intuition:** Duplicate ACKs indicate that data is getting through
- After three duplicate ACKs set:
  - Retransmit "lost packet"
- On packet loss detected by 3 dup ACKs:
  - ssthresh = cwnd/2
  - cwnd=ssthresh

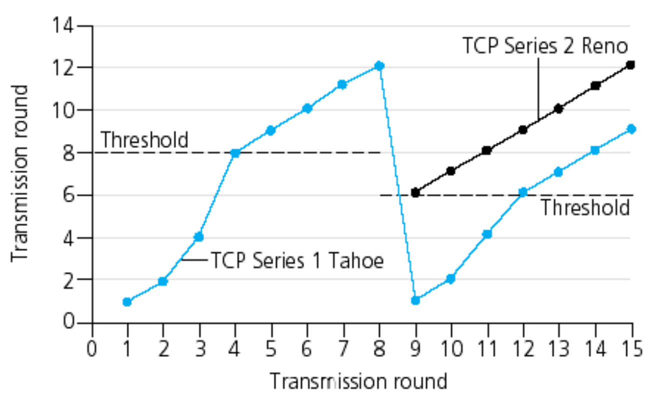
  enter congestion avoidance



TCP Tahoe and TCP Reno
(for single segment losses)



TCP CC



TCP New Reno

- When multiple packets are dropped, Reno has problems
- Partial ACK:
  - Occurs when multiple packets are lost
  - A partial ACK acknowledges some, but not all packets that are outstanding at the start of a fast recovery, takes sender out of fast recovery

  →Sender has to wait until timeout occurs
- **New Reno:**
  - Partial ACK does not take sender out of fast recovery
  - Partial ACK causes retransmission of the segment following the acknowledged segment
- New Reno can deal with multiple lost segments without

SACK

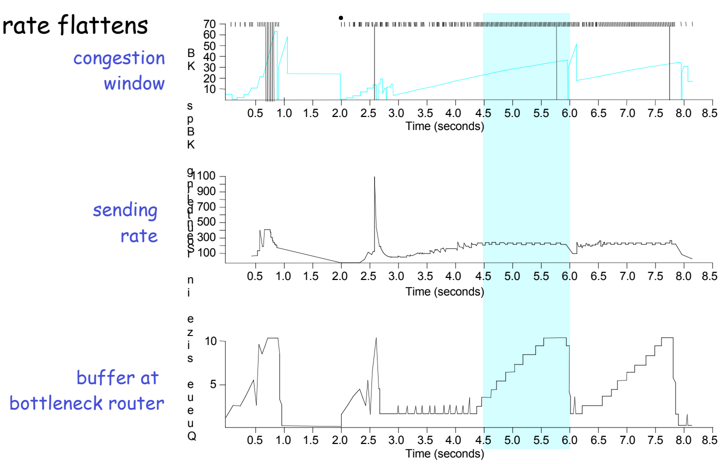
- SACK = Selective acknowledgment
- <u>Issue</u>: Reno and New Reno retransmit at most 1 lost packet per round trip time
- **Selective acknowledgments:** The receiver can acknowledge non-continuous blocks of data (SACK 0-1023, 1024-2047)
- Multiple blocks can be sent in a single segment.
- TCP SACK:
  - Enters fast recovery upon 3 duplicate ACKs
  - Sender keeps track of SACKs and infers if segments are lost.

Congestion Avoidance

- TCP's strategy
  - control congestion once it happens
  - repeatedly increase load in an effort to find the point at which congestion occurs and then back off
- Alternative strategy
  - predict when congestion is *about* to happen
  - reduce rate before packets start being discarded
  - call this congestion *avoidance*, instead of congestion *control*
- Two possibilities
  - host-centric: TCP Vegas
  - router-centric: DECbit and RED Gateways

Congestion Avoidance in TCP
(TCP Vegas)

- Idea: source watches for some sign that router's queue is building up and congestion will happen; e.g.,
  - RTT grows
  - sending rate flattens



Algorithm

- Let **BaseRTT** be the minimum of all measured RTTs (commonly the RTT of the first packet)
- If not overflowing the connection, then

```
ExpectRate = CongestionWindow/BaseRTT
```

- Source calculates sending rate (**ActualRate**) once per RTT
- Source compares **ActualRate** with **ExpectRate**

```
Diff = ExpectRate - ActualRate
if Diff < a
    increase CongestionWindow linearly
else if Diff > b
    decrease CongestionWindow linearly
else
    leave CongestionWindow unchanged
```