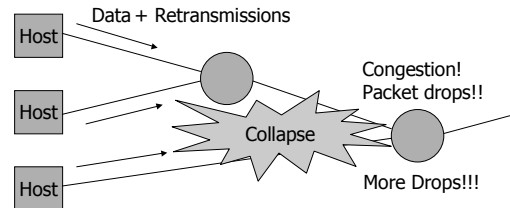


TCP Congestion Control

TCP Tahoe
TCP Reno
TCP Vegas

Internet Congestion Collapse

- In the late 80s, the Internet suffered a congestion collapse

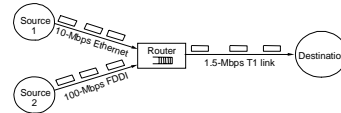


Why is Today's Topic Important?

THE ALGORITHM FOR TCP CONGESTION CONTROL IS THE MAIN REASON WE CAN USE THE INTERNET SUCCESSFULLY TODAY DESPITE LARGELY UNPREDICTABLE USER ACCESS PATTERNS AND DESPITE RESOURCE BOTTLENECKS AND LIMITATIONS. WITHOUT TCP CONGESTION CONTROL, THE INTERNET COULD HAVE BECOME HISTORY A LONG TIME AGO.

Resource Management Solutions

- Handling congestion
 - pre-allocate resources so as to avoid congestion ☹
 - control congestion if (and when) it occurs ☹



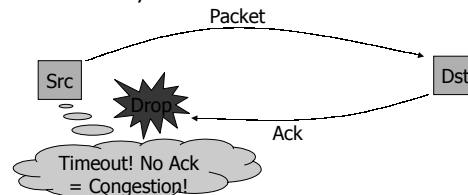
- Two points of implementation
 - routers inside the network (queuing discipline) ☹
 - hosts at the edges of the network (transport protocol) ☹

Problem with 80s Transport Protocol (Early TCP)

- The connection flow rate does not depend on the level of congestion in the network
- The 80s question was: How to re-design TCP to send less traffic when the network gets congested?
 - How to detect congestion?
 - How to make flow rate sensitive to congestion level?

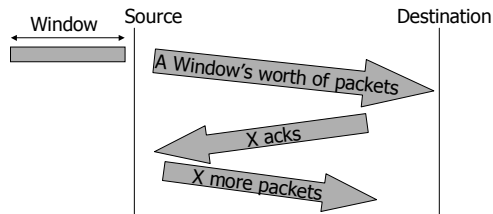
Detecting Congestion

- Packet drops indicate congestion
 - Is that really true?
 - Why does it work?



Controlling Congestion – The Effect of Window Size

- Note that sender's window is equal to the number of sender packets in flight (in the network). Why?



Controlling Congestion

- Reduce window → less packets in the network
- Increase window → more packets in the network
- Idea: Concept of a congestion window – window is smaller when congestion is larger and vice versa

Van Jacobson's Congestion Control

- Van Jacobson (formerly CISCO Chief Technical Officer – now a startup founder) introduced congestion control into TCP in 1988-1989.
- The original version of TCP that implements Van Jacobson's congestion control is known as TCP Tahoe.

Elements of Congestion Control in TCP Tahoe

- Additive increase, multiplicative decrease
- Slow start
- Fast retransmit

Additive Increase, Multiplicative Decrease

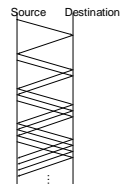
- Each time a packet drop occurs, slash window size in half (multiplicative decrease)
 - Multiplicative decrease is necessary to avoid congestion
- When no losses are observed, gradually increase window size (additive increase)

AIMD (cont)

- Algorithm
 - increment `CongestionWindow` by one packet per RTT (linear increase)
 - divide `CongestionWindow` by two whenever a timeout occurs (multiplicative decrease)
- In practice: increment a little for each ACK

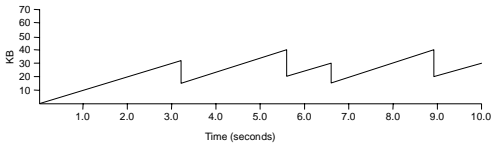
$$\text{Increment} = (\text{MSS} * \text{MSS}) / \text{CongestionWindow}$$

$$\text{CongestionWindow} += \text{Increment}$$



AIMD (cont)

- Trace: sawtooth behavior

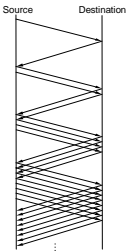


Problems

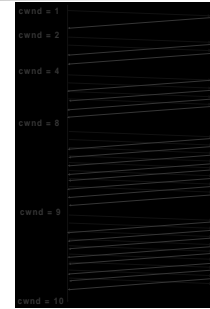
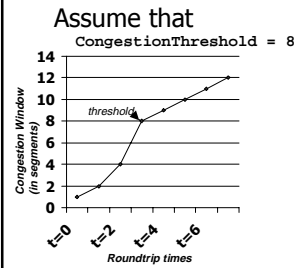
- What should the window size be
 - Initially?
 - Upon packet loss and timeout?
- Pessimistic window size? (e.g., 1)
 - Additive increase is too slow in ramping up window size – short connections will not fully utilize available bandwidth
- Optimistic window size?
 - Large initial burst may cause router queue overflow

Slow Start

- Objective: determine the available capacity quickly
- Idea:
 - Use `CongestionThreshold` as an optimistic `CongestionWindow` estimate
 - begin with `CongestionWindow = 1` packet
 - double `CongestionWindow` each RTT (increment by 1 packet for each ACK)
 - When `CongestionThreshold` is crossed, use additive increase



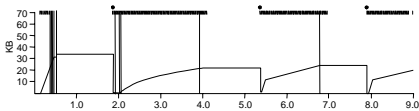
Slow Start Illustration



Figures by Jorg Liebeherr

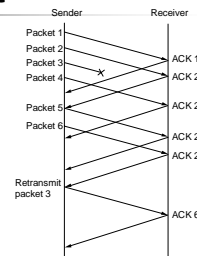
Slow Start (cont)

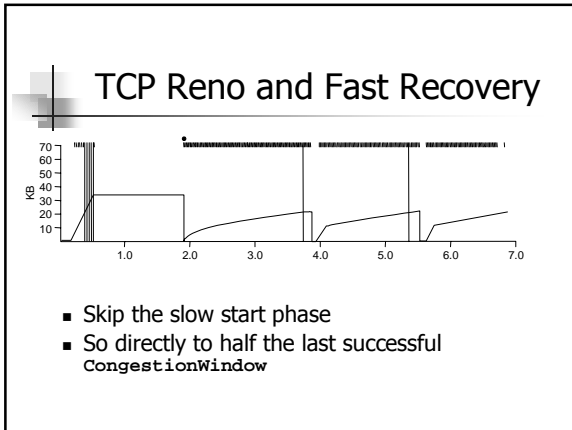
- Used...
 - when first starting connection
 - when connection goes dead waiting for timeout
- Real TCP Trace



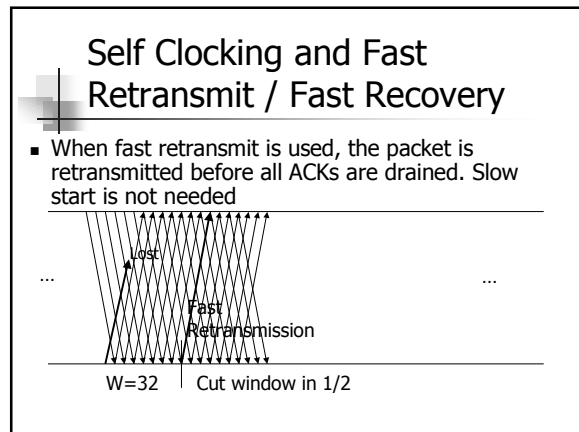
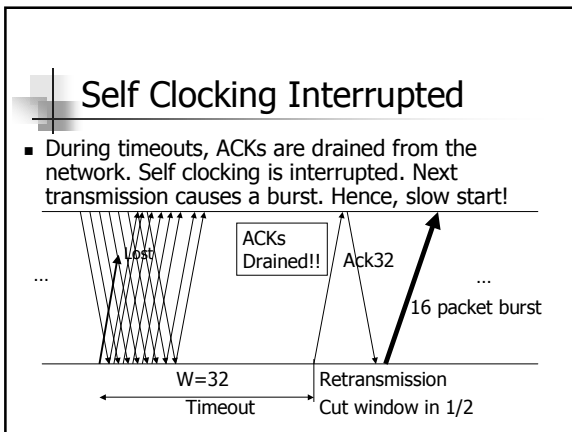
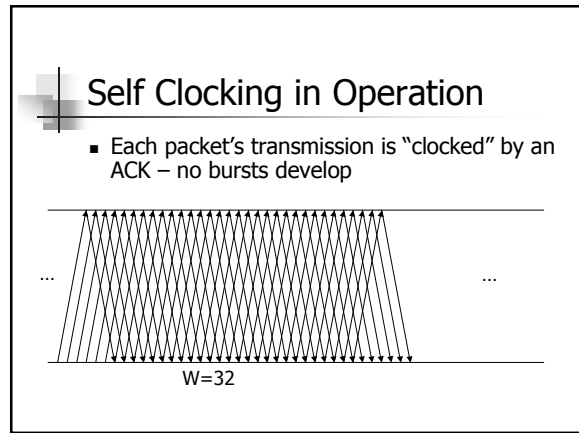
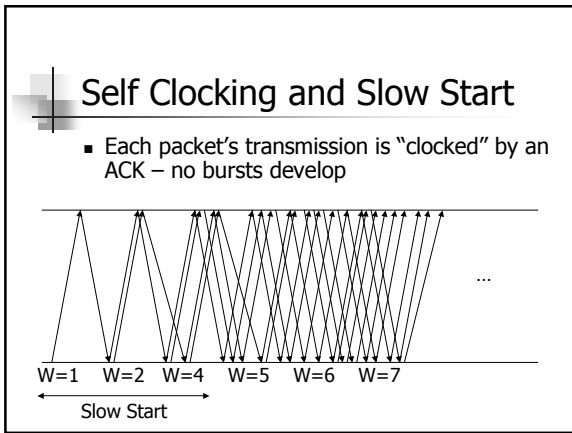
Fast Retransmit

- Problem: coarse-grain TCP timeouts lead to idle periods
- Fast retransmit:
 - Send an ack on every packet reception
 - Send duplicate of last ack when a packet is received out of order
 - Use duplicate ACKs to trigger retransmission



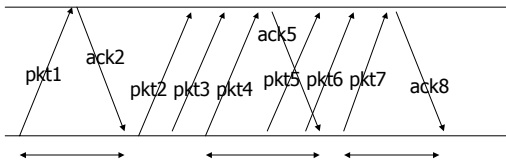


- ### Recap of Congestion Control
- TCP Reno and its derivatives are most widely used today
 - Performance issue: try to avoid forcing the source to go to slow-start (Why?)
 - Source goes to slow-start initially and upon timeouts (what's the rationale for that?)
 - Source cuts congestion window in half upon a fast retransmit (why not go to slow start?)
 - Single packet drops can be caught by the fast retransmit/fast recovery algorithm – source does not go to slow start (Why?)
 - Multiple consecutive packet drops will most likely force the source into slow start (why?)



Tahoe/Reno Retransmission Timers

- The retransmission timers are set based on round-trip time (RTT) measurements that TCP performs



© Jörg Liebeherr

Round-Trip Time Measurements

- TCP uses RTT mean and variance estimators, called $srtt$ and $rttvar$:

$$srtt_{n+1} = \alpha RTT + (1 - \alpha) srtt_n$$

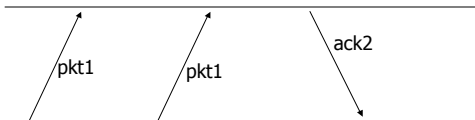
$$rttvar_{n+1} = \beta (| RTT - srtt_{n+1} |) + (1 - \beta) rttvar_n$$

- RTO is set to the mean RTT plus four times the estimated RTT variance.

$$RTO_{n+1} = srtt_{n+1} + 4 rttvar_{n+1}$$

Problem

- If a segment is retransmitted, TCP can't compute the RTT.
 - Why?



Karn's Algorithm

- Don't update RTT on any segments that have been retransmitted.
- When TCP retransmits, set:

$$RTO_{n+1} = \max (2 RTO_n, 64)$$

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
 - called congestion avoidance
- Two possibilities
 - router-centric: DECbit and RED Gateways
 - host-centric: TCP Vegas

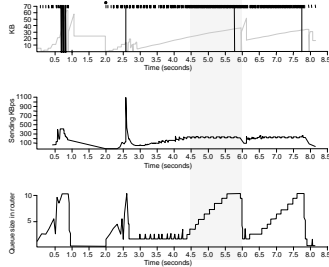
TCP Vegas

- Uses congestion avoidance instead of congestion control
 - Vegas: Congestion avoidance: Predict and avoid congestion before it occurs
 - Tahoe, Reno: Congestion control: React to congestion after it occurs
- Question: How to predict congestion?

TCP Vegas

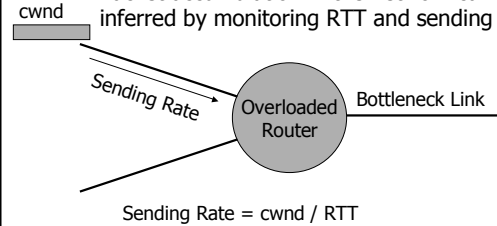
- Idea: source watches for some sign that router's queue is building up and congestion will happen too; e.g.,

- RTT grows
- sending rate flattens



Observation

- Packet accumulation in the network can be inferred by monitoring RTT and sending rate



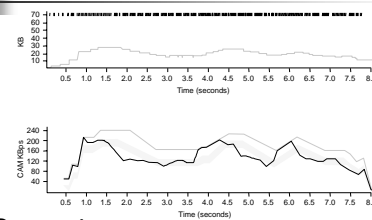
Algorithm

- Let **BaseRTT** be the minimum of all measured RTTs (commonly the RTT of the first packet)
- If not overflowing the connection, then
- ExpectedRate** = $\text{CongestionWindow} / \text{BaseRTT}$
- Source calculates **ActualRate** once per RTT
- Source compares **ActualRate** with **ExpectedRate**

```

Diff = ExpectedRate - ActualRate
if Diff <  $\alpha$ 
  increase CongestionWindow linearly
else if Diff >  $\beta$ 
  decrease CongestionWindow linearly
else
  leave CongestionWindow unchanged
    
```

Algorithm (cont)



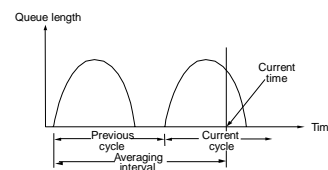
- Parameters
 - $\alpha = 1$ packet
 - $\beta = 3$ packets

Congestion Avoidance in the Network Layer: DECbit

- In the network:
 - Router computes average queue length
 - If average > 1, set congestion bit in packet header
- Destination copies congestion bit onto the ACK
- On the source:
 - Source maintains a sliding window (as in TCP).
 - If less than 50% of the ACKs in last window have congestion bit set, increase window by 1.
 - If more than 50% of the ACKs have congestion bit set, shrink window to 0.875 of its current size.

DECbit

- Computing the average queue length
 - Average carried out over last busy+idle intervals plus the current busy interval



Congestion Avoidance in RED

- Source:
Sally Floyd and Van Jacobson, "Random Early Detection Gateways for Congestion Avoidance," IEEE/ACM Transactions on Networking, Vol. 1, No. 4, pp. 397-413, August 1993.

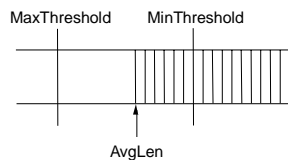
Random Early Detection (RED)

- Notification is implicit
 - just drop the packet (TCP will timeout)
 - could be made explicit by marking the packet
- Early random drop
 - rather than wait for queue to become full, drop each arriving packet with some drop probability whenever the queue length exceeds some drop level

RED Details

- Compute average queue length
$$\text{AvgLen} = (1 - \text{Weight}) * \text{AvgLen} + \text{Weight} * \text{SampleLen}$$
$$0 < \text{Weight} < 1 \text{ (usually } 0.002)$$

sampleLen is queue length each time a packet arrives



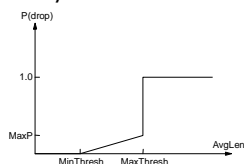
RED Details (cont)

- Two queue length thresholds

```
if AvgLen <= MinThreshold then
  enqueue the packet
if MinThreshold < AvgLen < MaxThreshold then
  calculate probability P
  drop arriving packet with probability P
if MaxThreshold <= AvgLen then
  drop arriving packet
```

RED Details (cont)

- Computing probability P (first pass):
$$P = \text{MaxP} * (\text{AvgLen} - \text{MinThreshold}) / (\text{MaxThreshold} - \text{MinThreshold})$$
- Drop Probability Curve



RED Details (cont)

- Problem with:
$$P = \text{MaxP} * (\text{AvgLen} - \text{MinThreshold}) / (\text{MaxThreshold} - \text{MinThreshold})$$
 - Drop probability is insensitive to time since last drop – two or more drops may occur back to back forcing the source into slow start
- Improvement:
$$\text{TempP} = \text{MaxP} * (\text{AvgLen} - \text{MinThreshold}) / (\text{MaxThreshold} - \text{MinThreshold})$$
$$P = \text{TempP} / (1 - \text{count} * \text{TempP})$$

RIO: RED with In and Out

- Packets fall into two categories
 - In profile (high priority)
 - Out of profile (low priority).
- Out packets are dropped first