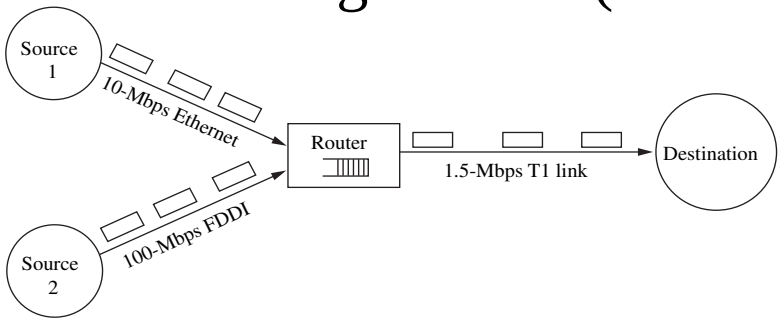
Copyright 1996. All rights reserved. Permission to reproduce this document for not for profit educational purposes is hereby granted. This document may not be reproduced for commercial purposes without the express written consent of the author.

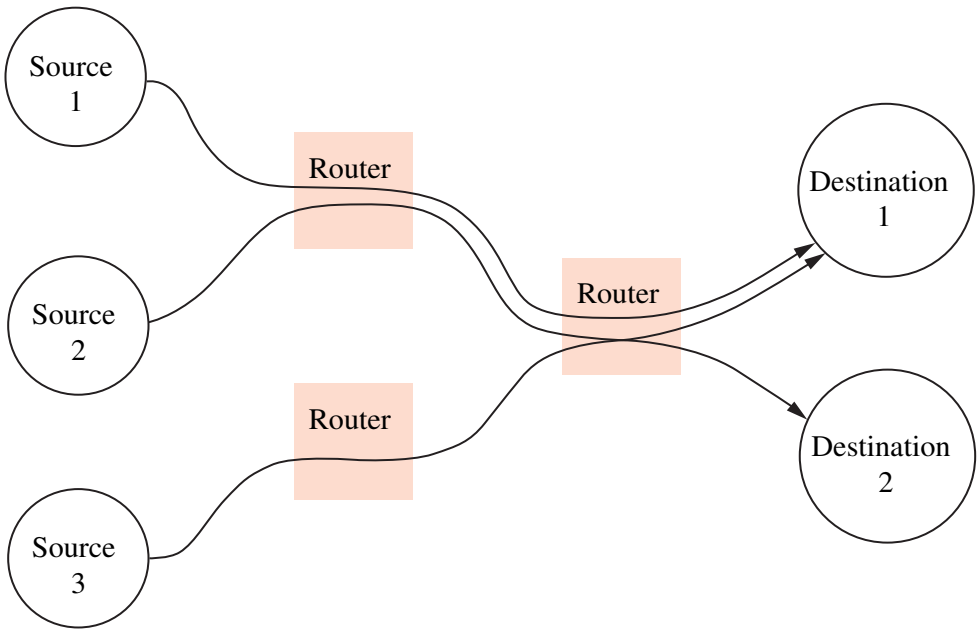
Congestion Control Issues

- Two sides of the same coin
  - pre-allocate resources so at to avoid congestion
  - send data and control congestion if (and when) is occurs

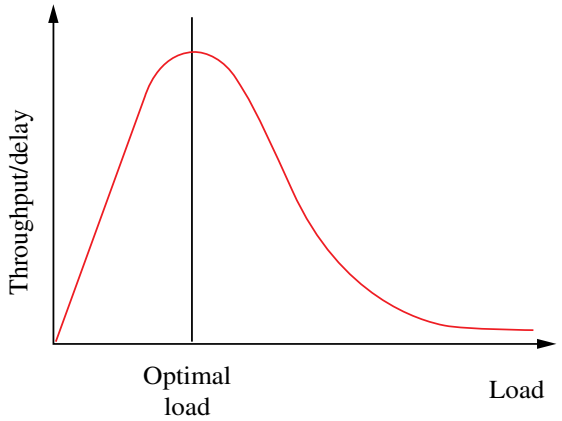


- Two points of implementation
  - hosts at the edges of the network (transport protocol)
  - routers inside the network (queuing discipline)
- Underlying service model
  - best-effort (assume for now)
  - multiple *qualities of service* (later)

- Connectionless flows
  - sequence of packets sent between source/destination pair
  - maintain *soft state* at the routers

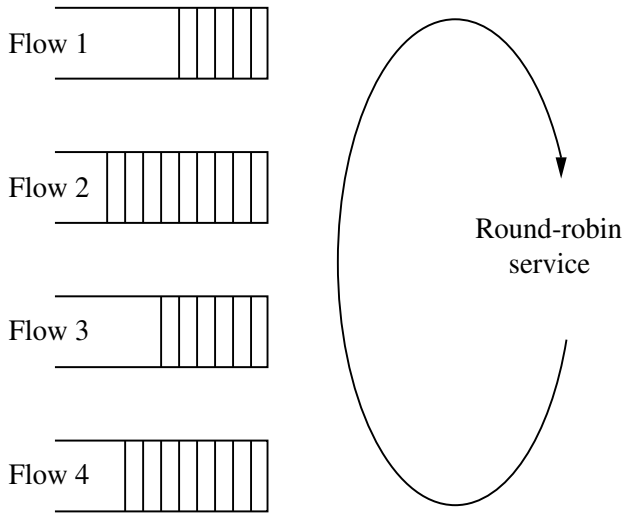


- Taxonomy
  - router-centric versus host-centric
  - reservation-based versus Feedback-based
  - window-based versus rate-based
- Evaluation
  - fairness
  - power (ratio of throughput to delay)



Queuing Disciplines

- First-In-First-Out (FIFO)
  - does not discriminate between traffic sources
- Fair Queuing (FQ)
  - explicitly segregates traffic based on flows
  - ensures no flow captures more than its share of capacity
  - variation: weighted fair queuing (WFQ)

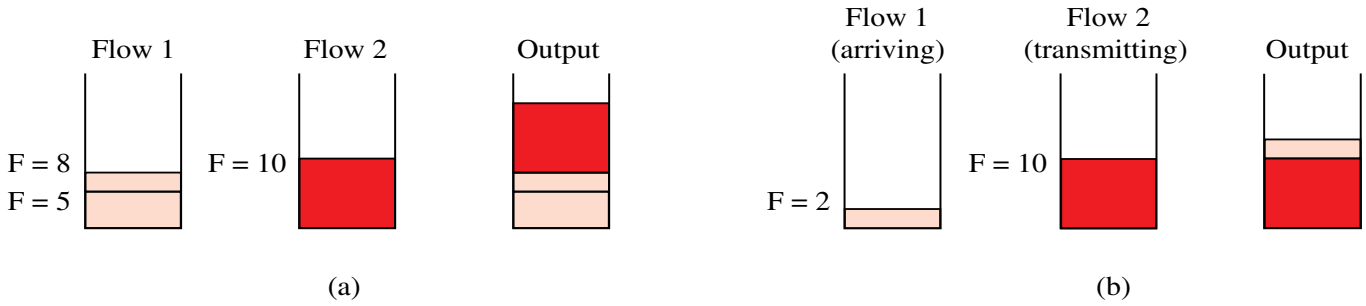


- Problem: packets not all the same length
  - really want bit-by-bit round robin
  - not feasible to interleave bits (schedule on packet basis)
  - simulate by determining when packet would finish
- For a single flow
  - suppose clock ticks each time a bit is transmitted
  - let P^i denote the length of packet i
  - let S^i denote the time when start to transmit packet i
  - let F^i denote the time when finish transmitting packet i
  - $F^i = S^i + P^i$

- When does router start transmitting packet *i*
  - If before router finished packet *i-1* from this flow, then immediately after last bit of i^{-1} (F^{i-1})
  - If no current packets for this flow, then start transmitting when arrives (call this A_i)
- Thus: $F^i = MAX(F^{i-1}, A^i) + P^i$

- For multiple flows
  - calculate F^i for each packet that arrives on each flow
  - treat all F^i's as timestamps
  - next packet to transmit is one with lowest timestamp

- Not perfect: can't preempt the packet currently being transmitted
- Example



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

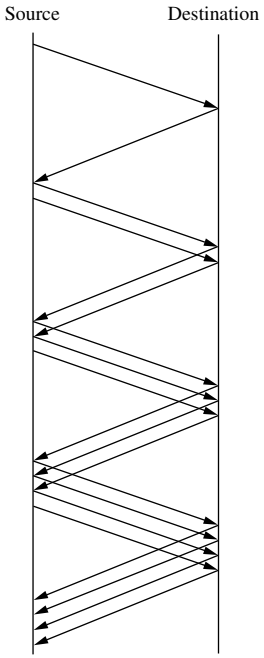
Additive Increase/Multiplicative Decrease

- Objective: adjust to changes in the available capacity
- New state variable per connection: **`CongestionWindow`**
  - limits how much data source has in transit

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

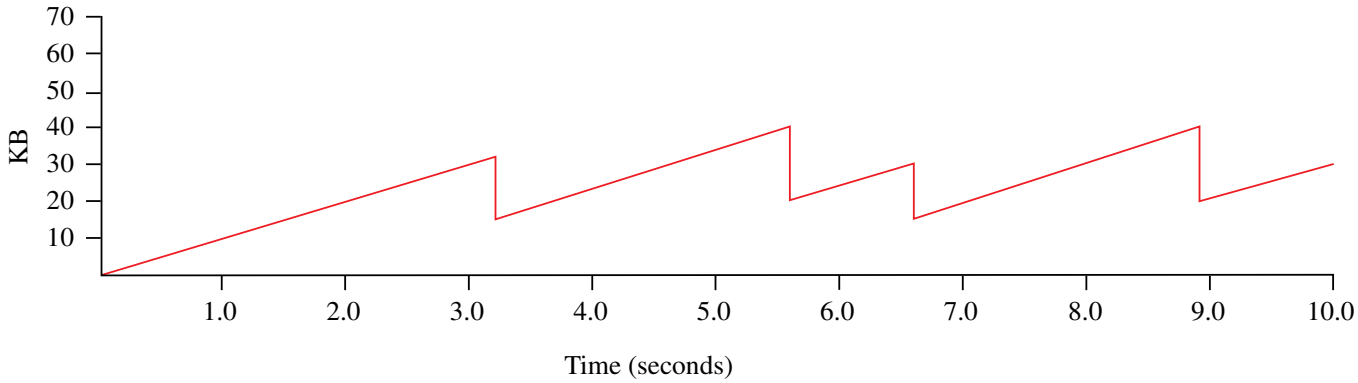
- Idea:
  - increase **`CongestionWindow`** when congestion goes down
  - decrease **`CongestionWindow`** when congestion goes up
- Question: how does the source determine whether or not the network is congested?
- Answer: a timeout occurs
  - timeout signals that a packet was lost
  - packets are seldom lost due to transmission error
  - lost packet implies congestion

- Algorithm:
  - increment **`CongestionWindow`** by one packet per RTT (*linear increase*)
  - divide **`CongestionWindow`** by two whenever a timeout occurs (*multiplicative decrease*)



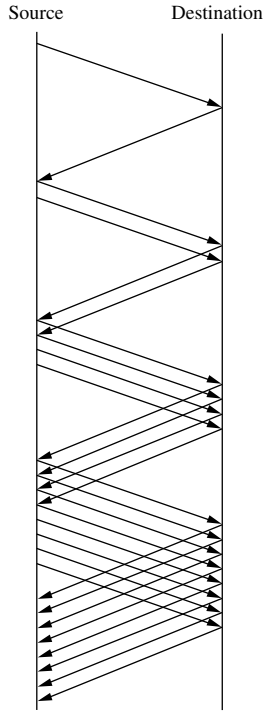
- In practice: increment a little for each ACK
  - Increment = (MSS * MSS)CongestionWindow
  - CongestionWindow = Increment

- Example trace: sawtooth behavior



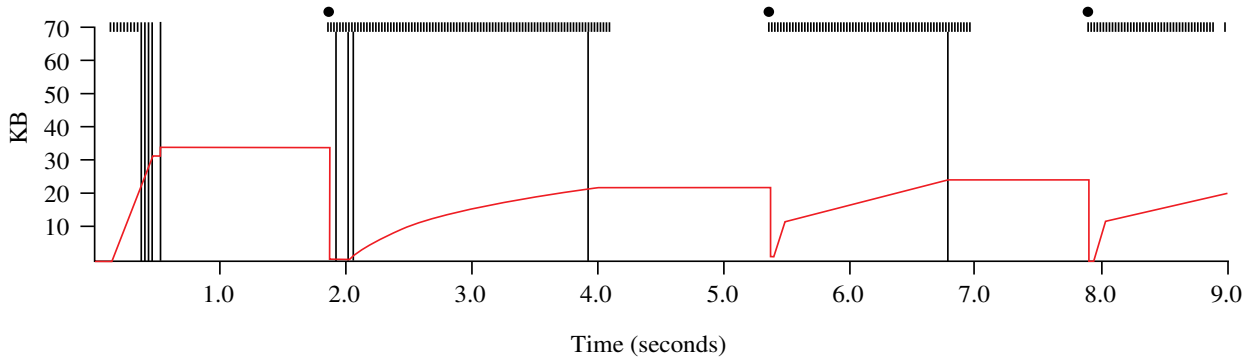
Slow Start

- Objective: determine the available capacity in the first place
- Idea:
  - begin with **`CongestionWindow`** = 1 packet
  - double **`CongestionWindow`** each RTT (increment by 1 packet for each ACK)



- Exponential growth, but slower than all in one blast
- Used...
  - when first starting connection
  - when connection goes dead waiting for a timeout

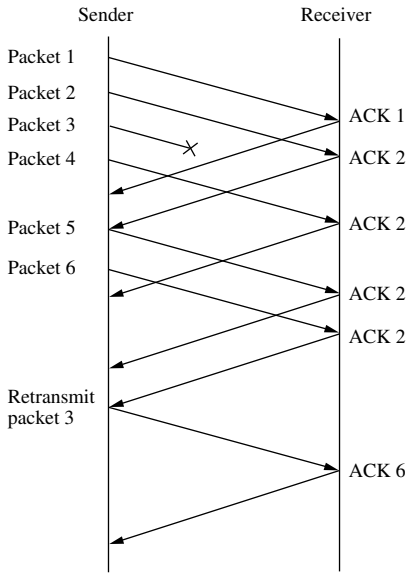
- Example trace



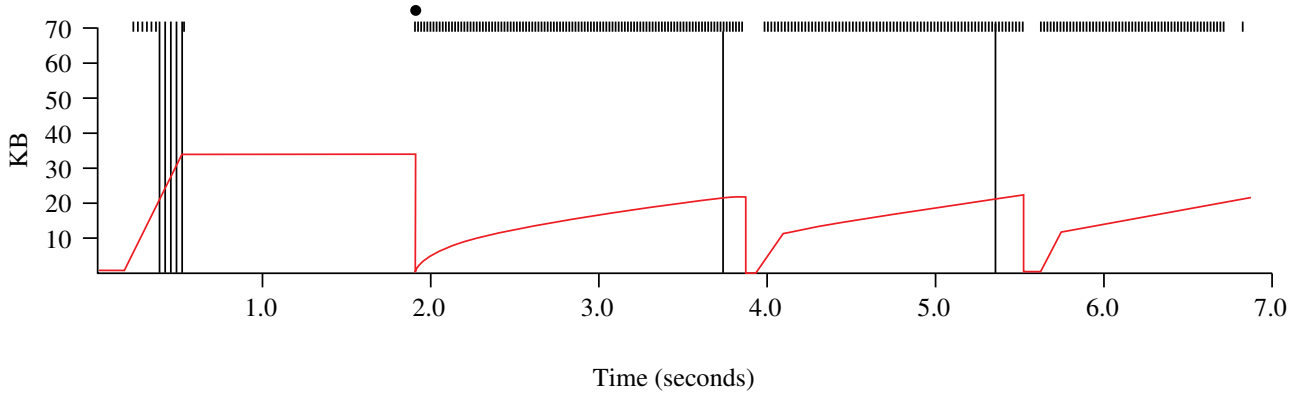
- Problem: lose up to half a **`CongestionWindow`**'s worth of data

Fast Retransmit and Fast Recovery

- Problem: coarse-grain TCP timeouts lead to idle periods
- Fast retransmit: use duplicate ACKs to trigger retransmission



- Results



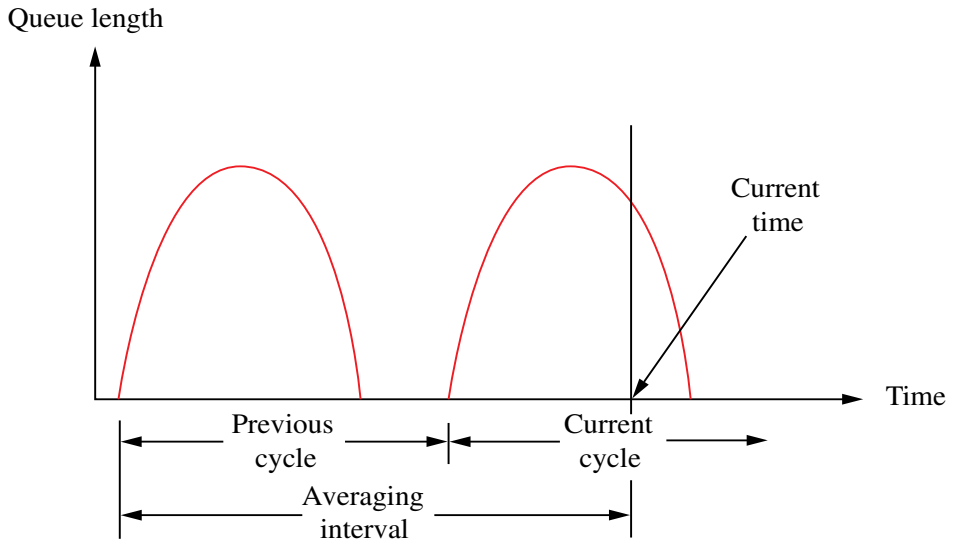
- Fast recovery: remove the slow start phase; go directly to half the last successful **`CongestionWindow`**

Congestion Avoidance Mechanisms

- TCP's strategy
  - to control congestion once it happens
  - to 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, and reduce the rate at which hosts send data just before packets start being discarded
  - we call this congestion *avoidance*, to distinguish it from congestion *control*
- Two possibilities
  - router-centric: DECbit and RED Gateways
  - host-centric: TCP Vegas

DECbit

- Add binary congestion bit to each packet header
- Router
  - monitors average queue length over last busy+idle cycle
  - set congestion bit if average queue length greater than 1 when packet arrives
  - attempts to balance throughput against delay



- End Hosts
  - destination echos bit back to source
  - source records how many packets resulted in set bit
  - if less than 50% of last window's worth had bit set, then increase **`CongestionWindow`** by 1 packet
  - if 50% or more of last window's worth had bit set, then decrease **`CongestionWindow`** by 0.875 times

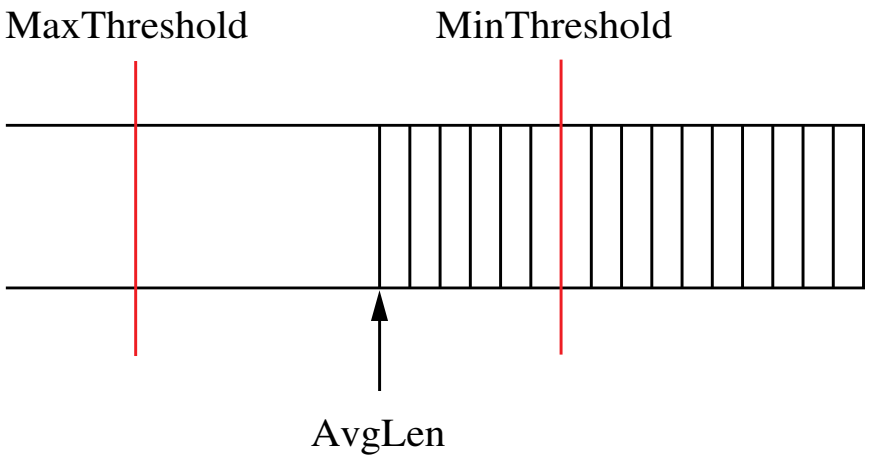
Random Early Detection (RED) Gateways

- Notification is implicit
  - just drop the packet (TCP will timeout)
  - could make 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: fills in the details
  - compute average queue length

    `AvgLen=(1-Weight)*AvgLen+Weight*SampleLen`

    - 0 < **`Weight`** < 1 (usually 0.002)
    - **`SampleLen`** is queue length each time a packet arrives

– two queue length thresholds
  - if AvgLen Š MinThreshold then
  - enqueue the packet
  - if MinThreshold < AvgLen < MaxThreshold
  - calculate probability P
  - drop arriving packet with probability P
  - if MaxThreshold Š AvgLen
  - drop arriving packet



– probability P
  - not fixed
  - function of **`AvgLen`** and how long since last drop (**`count`**)
    keeps track of new packets that have been queued while **`AvgLen`** has been between the two thresholds

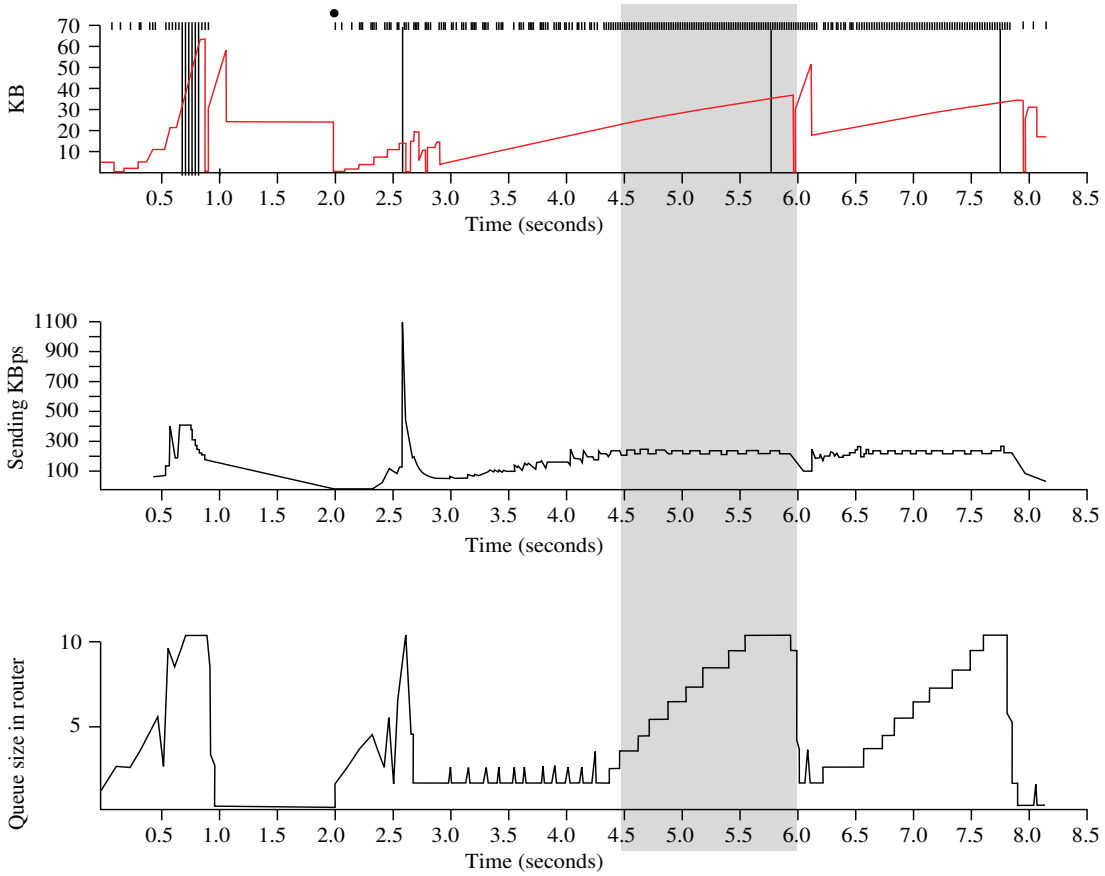
```
TempP = MaxP * (AvgLen - MinThreshold)
        /(MaxThreshold - MinThreshold)
P = TempP/(1 - count * TempP)
```

- Notes
  - probability of dropping a particular flow's packet(s) is roughly proportional to the share of the bandwidth that flow is currently getting
  - **`MaxP`** is typically set to 0.02, meaning that when the average queue size is halfway between the two thresholds, the gateway drops roughly one out of 50 packets.
  - if traffic is bursty, then **`MinThreshold`** should be sufficiently large to allow link utilization to be maintained at an acceptably high level
  - difference between two thresholds should be larger than the typical increase in the calculated average queue length in one RTT; setting **`MaxThreshold`** to twice **`MinThreshold`** is reasonable for traffic on today's Internet

TCP Vegas

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

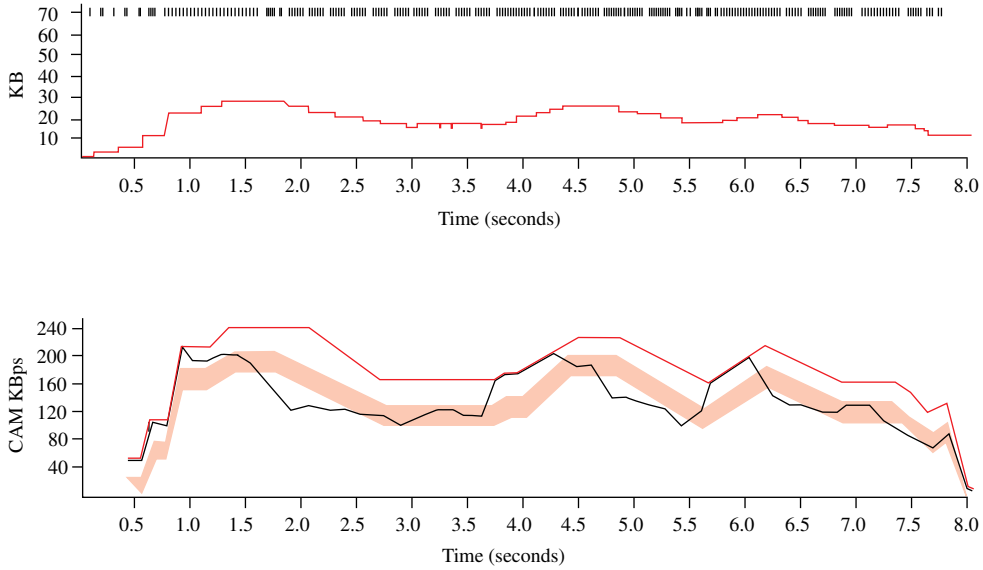
- Intuition


KB
70
60
50
40
30
20
10
0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5
Time (seconds)
Sending KBps
1100
900
700
500
300
100
0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5
Time (seconds)
Queue size in router
10
5
0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5
Time (seconds)

- Algorithm
  - let **`BaseRTT`** be the minimum of all measured RTTs (commonly the RTT of the first packet)
  - if not overflowing the connection, then
    - ExpectedRate = CongestionWindow / BaseRTT
  - source calculates current sending rate (**`ActualRate`**) once per RTT
  - source compares **`ActualRate`** with **`ExpectedRate`**

```
Diff = ExpectedRate - ActualRate if Diff < α
-->increase CongestionWindow linearly else if
   Diff > β
-->decrease CongestionWindow linearly else
-->leave CongestionWindow unchanged
```

- Parameters
  - α: 1 packet
  - β: 3 packets
- Example trace



- Even faster retransmit
  - keep fine-grained timestamps for each packet
  - check for timeout on first duplicate ACK

Virtual Clock

- Different approach
  - uses reservations rather than feedback
  - rate-based rather than window-based
  - router-centric rather than host-centric
- Idea
  - modeled after time-division multiplexing (TDM)
  - statistical multiplexing to accommodate bursty traffic
  - uses logical (rather than real) time

Defining Rate

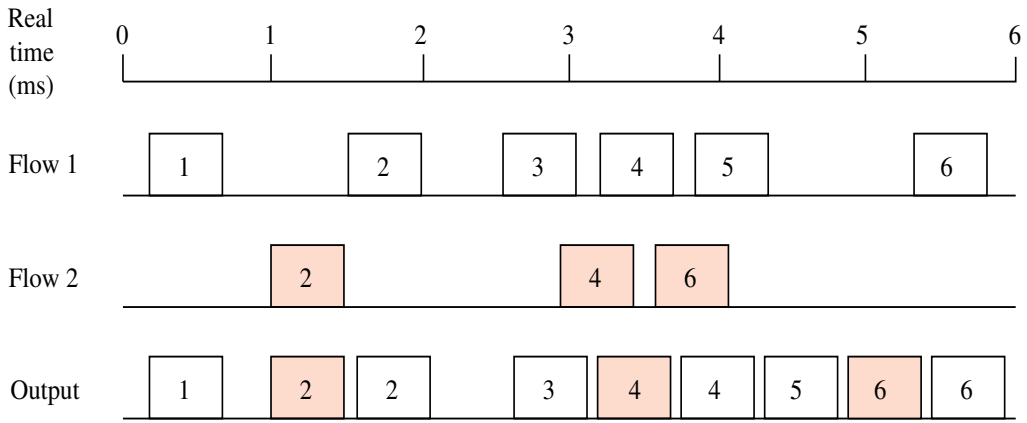
- Explicit flow setup phase
  - source indicates its needs
  - network grants or denies request
- Resource needs
  - average rate (**AR**)
  - average interval (**AI**)
  - example: **AR** = 10 packets per second and **AI** = 100ms
  - range: **1/AR Š AI Š** total flow duration
  - datagram traffic gets it's own flow; granted some capacity

Virtual Clock as a Queuing Discipline

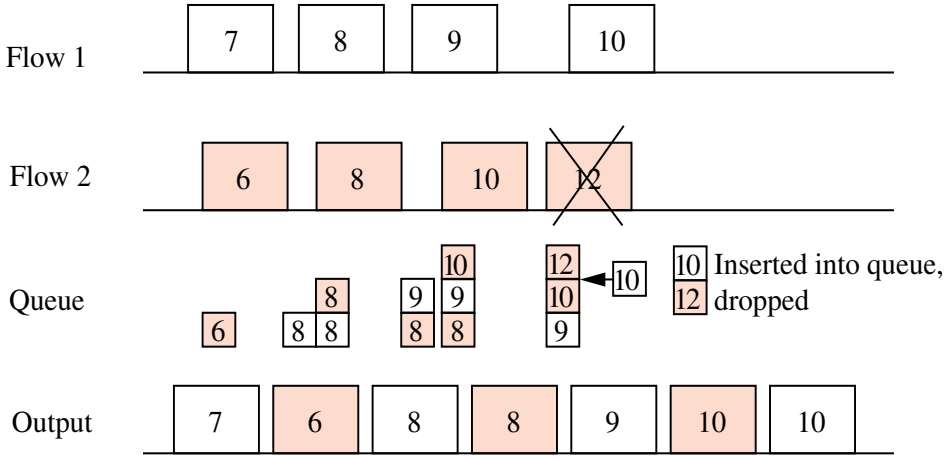
- Similar to weighted fair queuing
- Router variables
  - **`VClock`**^{i} for each flow *i*
  - clock tick (**`VTick`**^{i}) for flow *i*
  - **`VTick`**^{i} **`= 1/AR`**^{i} (assumes all packets on flow *i* are the same size)
  - example: flow *i* has **`AR`** = 200 packets per second, then **`VTick`**^{i} = 5ms

- Router algorithm
  - first packet: **`VClock`**^{i} **`= RealTime`**
  - each packet thereafter: **`VClock`**^{i} **`+= VTicki`**
  - timestamp each packet with **`VClock`**^{i}
  - packets queued and serviced in according to timestamps
  - when buffer space is full, packet with largest timestamp is dropped

- Example of interleaved service proportional to **AR**
  - flow 1: **AR** = 1000 packets per second
  - flow 2: **AR** = 500 packets per second



- Example of discarding a packet



Virtual Clock as a Flow Meter

- Note
  - have not yet usee **AI**
  - possible for flow to “save up” credits and use for burst
- Additional mechanism
  - when setup flow *i*, compute $\mathbf{AIR}^i = \mathbf{AR}^i \times \mathbf{AI}^i$
  - upon receiving $\mathbf{AIR}^i$ bytes on flow *i*...
    - if $\mathbf{VClock}^i - \mathbf{RealTime} > \mathbf{Threshold}$, then send advisory to source
    - if $\mathbf{VClock}^i < \mathbf{RealTime}$, then reset $\mathbf{VClock}^i$ to **RealTime**
  - source not allowed to accumulate “unused time” during one **AI** period and use it in another

- Last detail
  - source can still accumulate credit *within* an **`AI`**
  - problem: using one variable (**`VClock`**) to control both queuing order and flow monitoring
  - solution: introduce auxiliary copy of **`VClock`**
  - upon receiving each packet...

`AuxVClock`^{i} **`= MAX(RealTime, AuxVClock`**^{i}**`)`**
`VClock`^{i} **`= VClock`**^{i} **`+ VTick`**^{i}
`AuxVClock`^{i} **`= AuxVClock`**^{i} **`+ VTick`**^{i}
`Stamp packet with current value of AuxVClock`^{i}

- Thus
  - **`VClock`**^{i} resynchronized with real time once for every **`AIR`**^{i} bytes of data on flow *i*
  - **`AuxVClock`**^{i} resynchronized with real time once for every packet on flow *i*