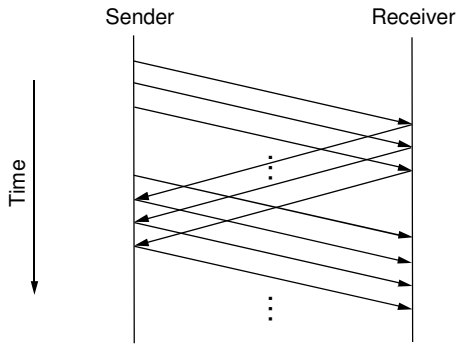
# CSE 461 – TCP Flow Control
# TCP Congestion Control

## Part 0: Sliding Window Review



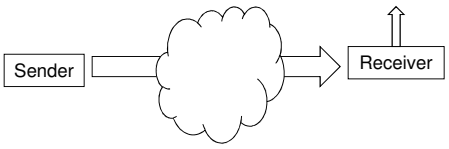
## Part 0: TCP and Sliding Window

- TCP uses sliding window
  - ARQ for reliability
    - Timeout and resend if ACK doesn't arrive
  - Send and receive buffers
    - Sender
      - save copies in case you have to retransmit
    - Receiver
      - Re-order buffer in case segments arrive out of order
      - Re-order buffer in case some segments are lost
      - Speed matching buffer in case segments arrive in bursts

3

## Part 1: There's still a problem…

- A sender may choose a sliding window size that can overwhelm the receiver
  - Sender is more powerful machine than receiver
  - Producing data is cheaper than consuming it
- Effects of buffer overrun on…?
  - Reliability?
  - Performance?
- What can we do about it?
  - Flow control

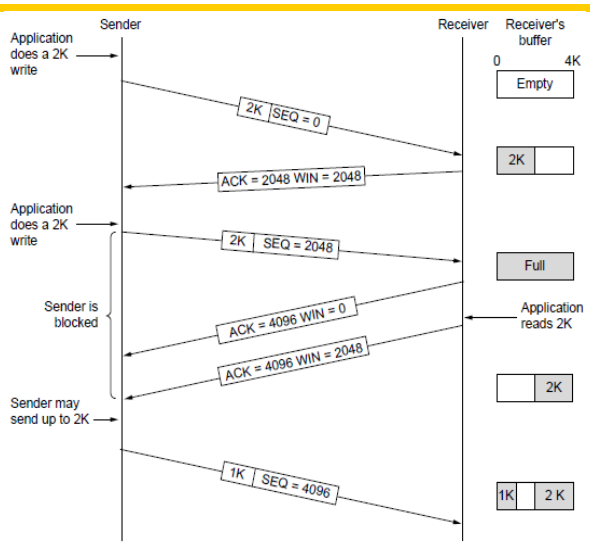


## Flow Control in TCP

- Basic Idea:
  - Let receiver tell sender how much free buffer space receiver has
  - Update the value regularly
- Another take on the basic idea:
  - Decouple acknowledgements from window size

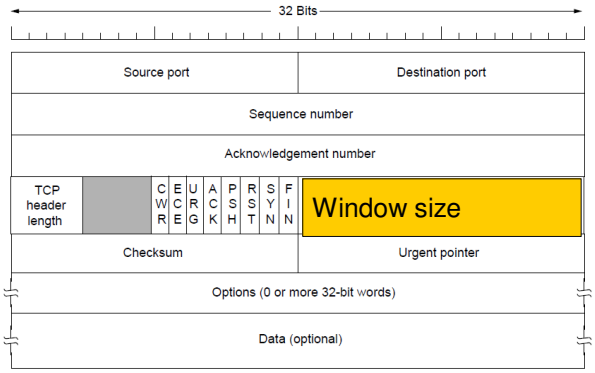
5

## TCP flow control



## TCP Header Format

- Advertised window is used for flow control






## One Additional Problem

- Window advertisement messages can't be part of ARQ scheme
  - Receiver sends segment with:
    - ACK set to last byte received in order + 1
    - Updated window size
    - No data (data length = 0)
      - So, sender sequence number == last data sent sequence number
  - Even if sender ACKs the window advertisement message, receiver can't distinguish that from ACKing the previous data segment

## One Additional Problem (cont.)

- So… window advertisement messages are unreliable
  - Can't be ACK'ed
- Why is that a problem?
  - What can go wrong if one is lost?
- How can you fix the problem?

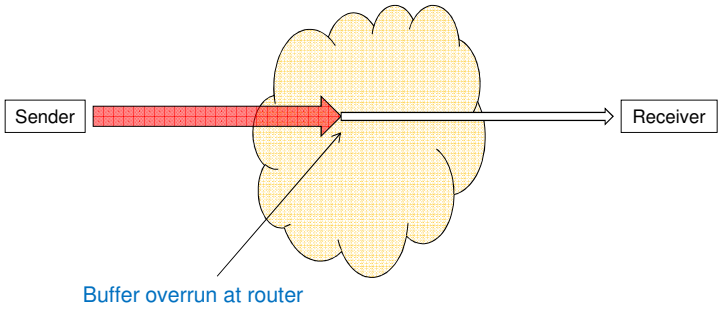
9

## Window size deadlock problem

10

## Part 2: There's still a problem…

Congestion: The bottleneck is inside the network




11

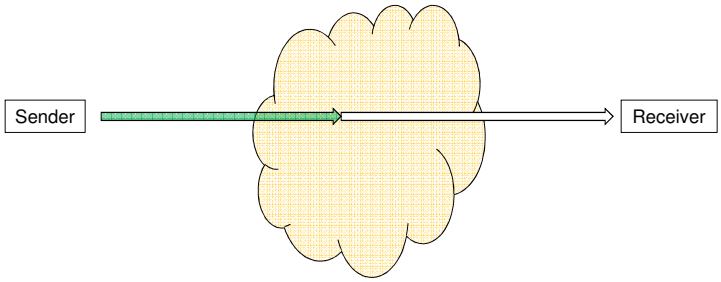
## But memory is cheap…

- Why not just have really big router buffers?
  - Doesn't fix the problem
    - If input rate vs. output rate imbalance persists, it will overrun any buffer size
  - Increases delay and delay variance
    - A packet lucky enough to be queued at the router may have to wait a long time before transmission
    - Sometimes the queue is empty, sometimes it's nearly full…
- Big router buffers aren't a solution

12

## What is a solution?

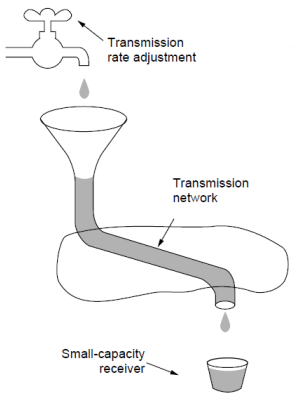
Sender should control the rate at which it sends to avoid overloading the bottleneck router



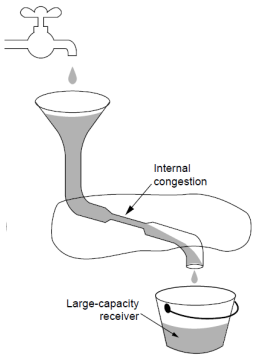

Ideally, packets arrive at bottleneck router just as it has a free transmission slot…

13

## Flow vs. Congestion Control



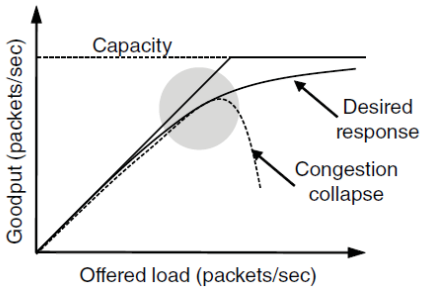

A fast network feeding a low-capacity receiver → flow control is needed



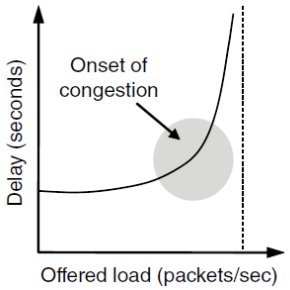

A slow network feeding a high-capacity receiver → congestion control is needed

## When is network "congested"?

Efficient use of bandwidth gives high goodput, low delay

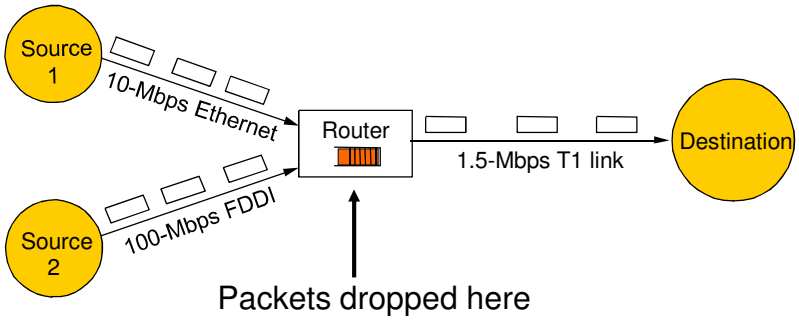



Goodput rises more slowly than load when congestion sets in




Delay begins to rise sharply when congestion sets in

## Why does this congestion collapse occur?



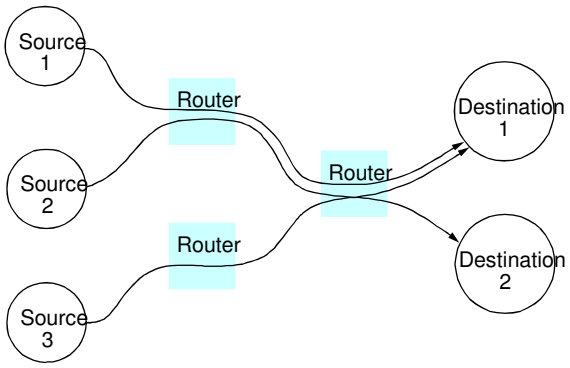

- Buffer intended to absorb bursts when input rate > output
- But if sending rate is persistently > drain rate, queue builds
- Dropped packets represent wasted work; goodput < throughput

## What should the sending rate be?

- Cannot be too close to the capacity
- In practice, a good operating point optimizes the “power” metric
  - Power = goodput/ delay
- It will rise with send rate until delay starts to climb rapidly
  - Knee of the curve gives a good operating point.

## What about fairness?

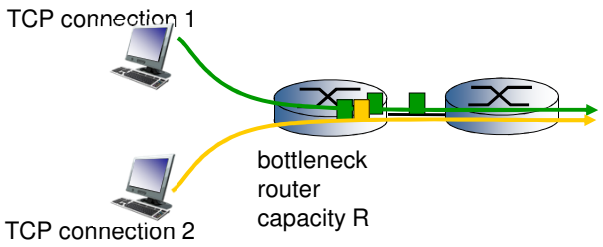
Defining “fairness” is complicated…



## TCP Fairness: Max-Min Fairness

*Fairness goal:* if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K…

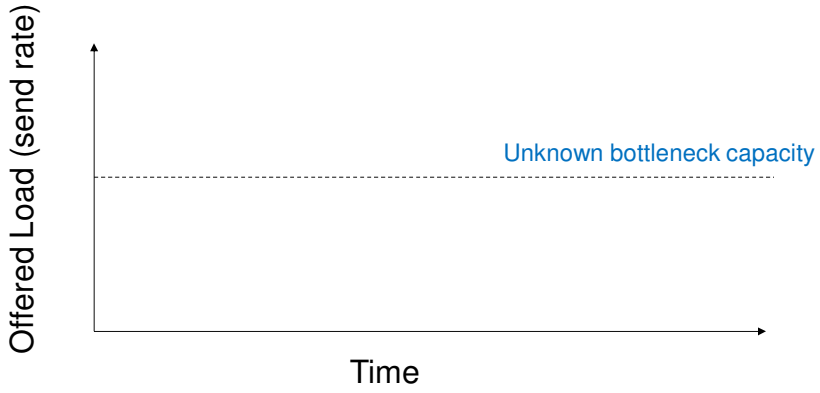
except that if any flows can't use their share, the excess is equally divided among those that can




## How to effect congestion control?

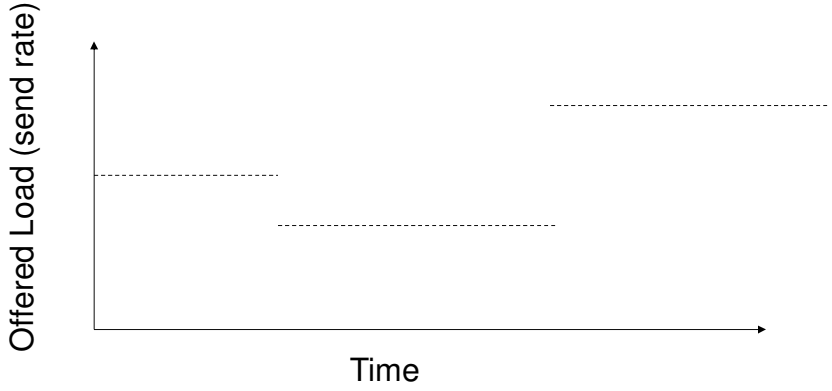
- The bottleneck router knows its state
  - It seems like a natural choice from being in charge of throttling the sender, but…
- TCP and the Internet have substantial deployment when the need for congestion control is realized
  - Using routing to go around congested portions of the Internet hasn't worked out…
- Approach: modify the TCP implementations on end hosts, and deploy updated implementations host-by-host
  - Good idea, but you need a solution that doesn't require router cooperation
  - Actual solution: even better! Requires changes only to the sender's implementation

## Sender-based congestion control




21

## It's actually a little more complicated




22

## TCP congestion control

- No long-term history information used
  - Path to some destination right now may not be similar to path last time
- "Learn" current bottleneck capacity
  - Start conservatively
  - Ramp up rate if things look good
  - Reduce rate if things look not so good
  - Slash rate if things look bad

## How to ``learn" about bottlenecks?

- Implicit feedback
  - No (new) router feedback
  - No (new) receiver feedback
- Use losses
  - If you time out and retransmit, assume a congestion loss has occurred
- Learning the bottleneck capacity
  - Increase the sending rate until there is a loss
  - When convincing losses occur, decrease the sending rate

## Controlling Send Rate

- The send rate is controlled using a congestion window
- If everything remains fixed, the send rate is
  send rate = CWND / RTT
- Sender's rate is limited by the min of the flow control and congestion windows
- How to increase/decrease rate means how to adjust CWND
  - How?

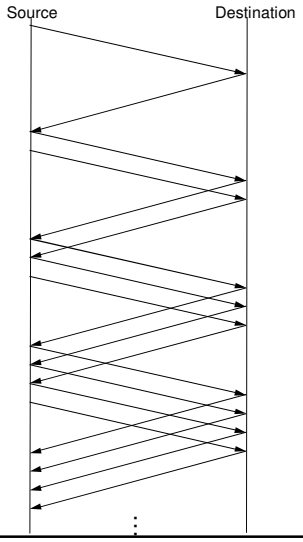
25

## AIMD (Additive Increase/Multiplicative Decrease)

- Additive Increase
  - Ramp up slowly
  - In particular, increase rate by a constant for every t seconds that things continue to look good
- Multiplicitive Decrease
  - Ramp down quickly
  - Ramp down by a fraction of your current rate, not a constant
    - If you're going fast, you ramp down a lot
    - If you're going slowly, your ramp down a little
  - If all flows detect the bottleneck at the same time, total reduction is a fraction of the bottleneck load
    - Works no matter what the bottleneck capacity is, in absolute terms

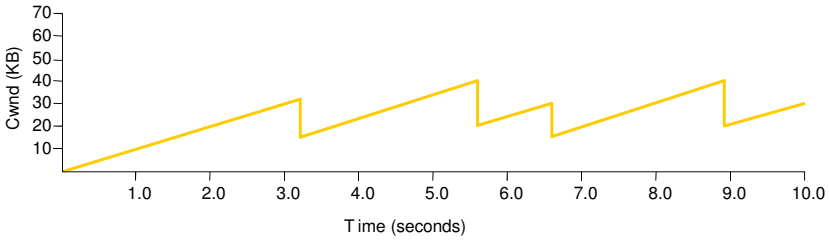
## TCP's AIMD Mechanism

- Additive constant for increase:
  1 MSS
- Time interval = 1 RTT
  - Rate of increase = MSS / RTT
- Simple implementation:
  - Each ACK received
    Cwnd += 1/Cwnd MSS
- Multiplicative constant for decrease:
  - Cwnd /= 2
- TCP uses a version of this




## AIMD Sawtooth Pattern

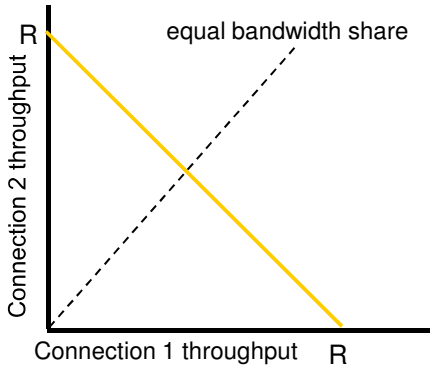
- TCP has evolved, but is based on the notion of AIMD



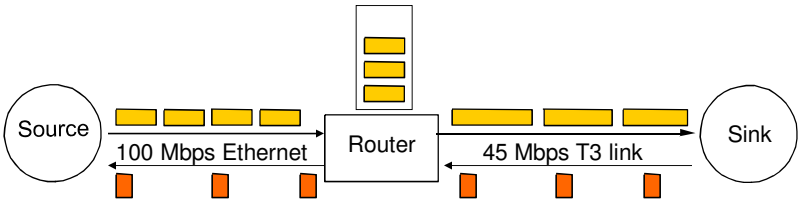
## Why is TCP fair?

Two competing sessions:

- additive increase gives each a slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally




## Bonus Property: "Self-Clocking" the sending rate




- ACKs pace transmissions at approximately the bottleneck rate
- So just be sending packets we can discern the "right" sending rate (called the packet-pair technique)

## TCP has evolved…

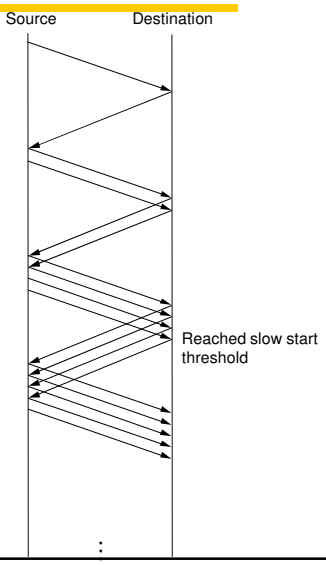
- Increasing CWND by one MSS every RTT can be very slow
  - Example: If a src can send at a rate of 1.5Gbps, how many RTTs will it take before src can send at capacity? (assume 1500 byte MSS)
- Reacting to a loss may take a long time since it takes forever to learn that there has been a loss
  - Related to the retransmission timeout, which is related to the mean and variance of the RTT

## TCP congestion control

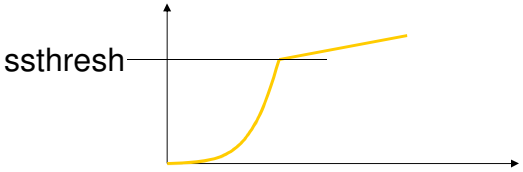
- Slow Start (not really slow)
  - Pick a threshold and increase exponentially until you hit the threshold, then do additive increase
  - (This is "slow" relative to just starting out by sending a full flow-control window as fast as you can.)
- Fast Retransmit
  - Triple duplicate ACKs hint that a loss has occurred

## TCP "Slow Start"

- Until slow start threshold
  - Each ACK received:
    CWND += 1 MSS
  - Doubles CWND every RTT
- When the slow start threshold is reached, start additive increase
  - CWND +=1 / CWND



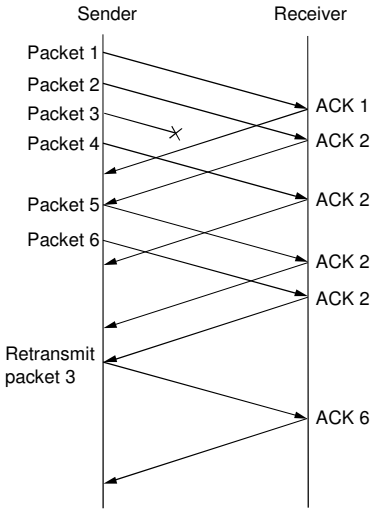

## Combining Slow Start and multiplicative decrease




- Slow start is used initially, and after every timeout
- Instead of multiplicative decrease
  - **Reduce the slow start threshold by half**

## Fast Retransmit

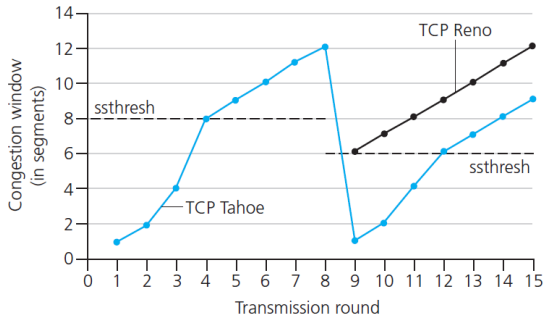
- TCP uses cumulative acks, so duplicate acks start arriving after a packet is lost.
- We can use this fact to infer which packet was lost, instead of waiting for a timeout.
- 3 duplicate acks are used in practice




## Fast Recovery

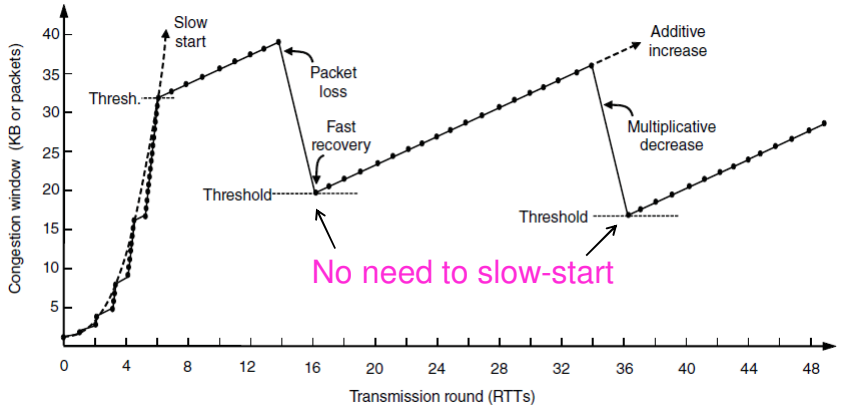
- Use AIMD when there are single packet losses. Only slow start the first time.
- On experiencing a loss
  - Set the slow start threshold to half the current CWND
  - Start at slow start threshold and do additive increase

## Slow start example




L17.37

## TCP Congestion Control



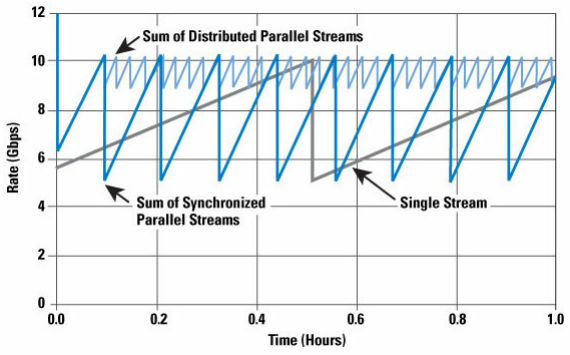
## High-speed, High-delay TCP

- Single connection over 10 Gbps / 70 msec. RTT / 1500 byte packets
  - 256MB of total buffer on the path
- Slow start increases send rate exponentially
  - After 17 RTT (1.2 sec.) sending at 11.2 Gbps
  - Losses cause halving of send rate followed by linear increase
- Result?
  - Takes 41 sec. to get to recover to 10Gbps
  - Takes 34 min 22 sec to saturate buffers and detect losses
  - This requires a bit error rate no greater than $10^{-14}$
    - or we'll be responding to bit errors rather than congestion
  - 1.95 TB sent between losses
  - 7.55 Gbps long-term transmission rate

39

## One Approach: "Parallel" TCP

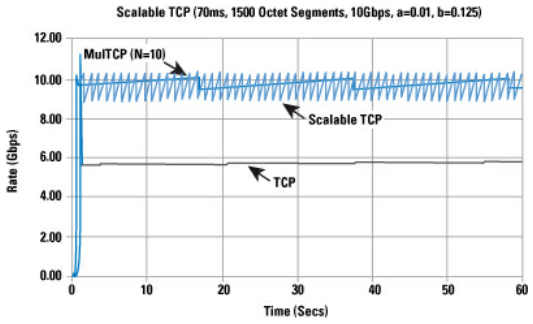
- Pretend a single TCP stream is really N TCP streams




http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_9-2/gigabit_tcp.html

40

# Scalable TCP




Additive increase: a packets/ACK
Multipliciative decrease: factor of (1-b)

41