بسمه تعالی

School of Computer Engineering

# Advanced Computer Networks

Transport Layer and Congestion Control

Part 7

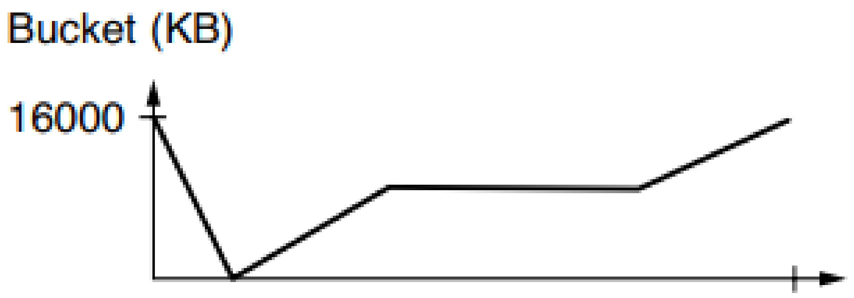
Seyed Hamed Rastegar

Fall 1401

# Congestion Control: Token Bucket

**Token Bucket: Example**

- Finally, the Figure illustrates the bucket level for a token bucket with R = 200 Mbps and a capacity of B = 16, 000 KB.




- This is the smallest token bucket through which the traffic passes unaltered.

# Congestion Control: Token Bucket

**Token Bucket: Example**

- It might be used **at a router in the network** to police the traffic that the host sends.
- However, **if the host is sending traffic that conforms** to the token bucket on which it has agreed with the network, the traffic will fit through that same token bucket run at the router at the edge of the network.
- **If the host sends at a faster or burstier rate**, the token bucket will run out of water.
  - If this happens, a traffic policer will know that the traffic is not as was described.
  - It will then either drop the excess packets or lower their priority, depending on the design of the network.
- **In our example**, the bucket empties only momentarily, at the end of the initial burst, then recovers enough for the next burst.

# Congestion Control: Token Bucket

**Token Bucket: Implementation**

- Leaky and token buckets are easy to implement. We will now describe the operation of a token bucket.
- Even though we have described **water flowing continuously** into and out of the bucket, **real implementations** must work with **discrete quantities**.
- A token bucket is implemented with **a counter for the level of the bucket**.
  - The counter is advanced by $R/\Delta T$ units at every clock tick of $\Delta T$ seconds.
  - This would be 200 Kbit every 1 msec in our example above.
  - Every time a unit of traffic is sent into the network, the counter is decremented, and traffic may be sent until the counter reaches zero.

# Congestion Control: Token Bucket

## Token Bucket: Example

- **When the packets are all the same size**, the bucket level can just be counted in packets (e.g., 200 Kbit is 20 packets of 1250 bytes).

- However, **often variable-sized packets are used**. In this case, **the bucket level can be counted in bytes.**

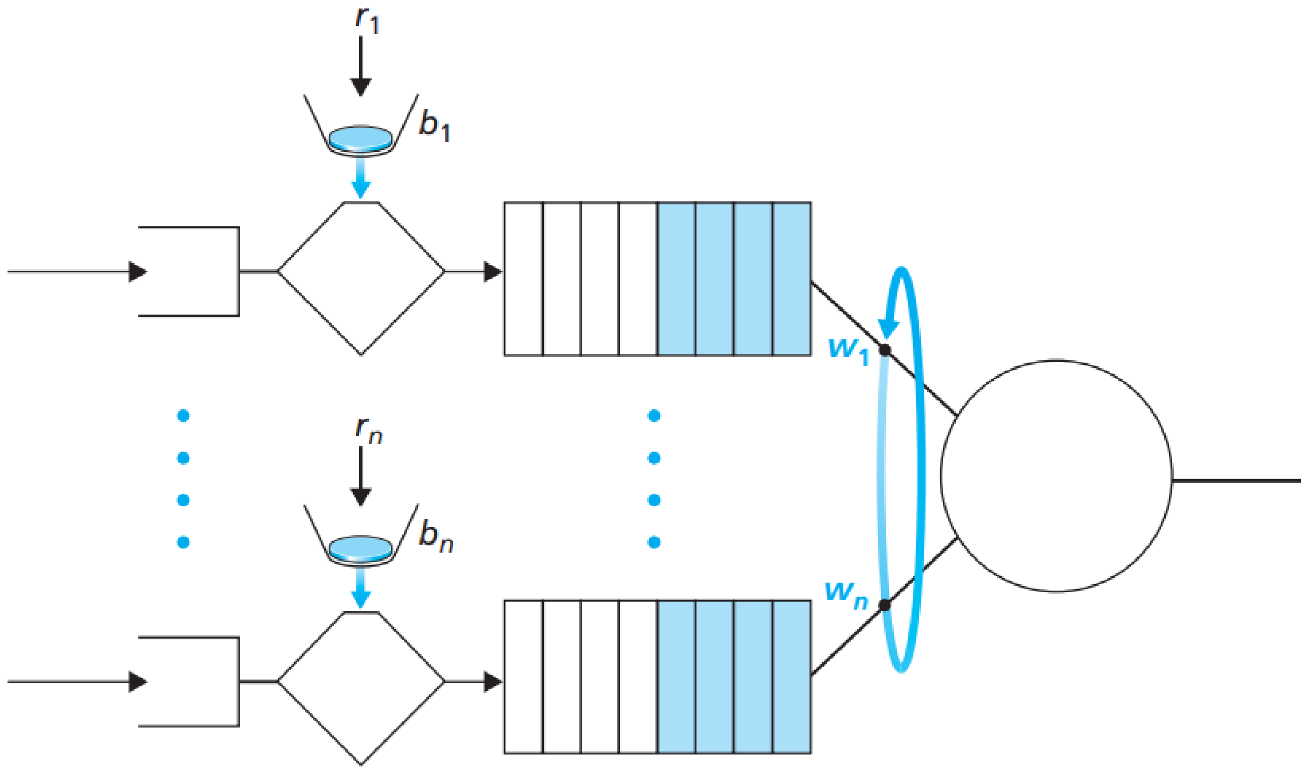
- If the **residual byte count is too low to send a large packet**, the packet must wait until the next tick (or even longer, if the fill rate is small).

# Congestion Control: Token Bucket

**Token Bucket + Weighted Fair Queuing = Provable Maximum Delay in a Queue**

- How the two can be combined to provide a bound on the delay through a router's queue.

- Let's consider a router's output link that multiplexes $n$ flows, each policed by a token bucket with parameters $b_i$ and $r_i$ , $i = 1, \ldots, n$, using WFQ scheduling.



# Congestion Control: Token Bucket

**Token Bucket + Weighted Fair Queuing = Provable Maximum Delay in a Queue**

- Recall from our discussion of WFQ that each flow, *i*, is guaranteed to receive a share of the link bandwidth equal to at least $R \cdot w_i/(\sum w_j)$, where $R$ is the transmission rate of the link in packets/sec.

- **What then is the maximum delay** that a packet will experience while waiting for service in the WFQ (that is, after passing through the token bucket)?

- Let us focus on flow 1. Suppose that flow 1's token bucket is initially full. A burst of $b_1$ packets then arrives to the token bucket policer for flow 1.

# Congestion Control: Token Bucket

**Token Bucket + Weighted Fair Queuing = Provable Maximum Delay in a Queue**

- These packets remove all of the tokens (without wait) from the token bucket and then join the WFQ waiting area for flow 1.

- Since these $b_1$ packets are served at a rate of at least $R \cdot w_i/(\sum w_j)$ packet/sec, the last of these packets will then have a maximum delay, $d_{max}$, until its transmission is completed, where

$$d_{max} = \frac{b_1}{R \times w_1 / \sum w_j}$$

# Congestion Control: Token Bucket

- In practice

support.huawei.com/enterprise/en/doc/EDOC1100020320?section=j00f

NE05E and NE08E V200R006C20SPC600 Feature Description 01(CLI)

Favorite Download Feedback

Contents



Related Documents

NE05E&NE08E V200R006C20SPC600 Product Description 01(CLI)

NE05E, NE08E Hardware Guide

(IPv6 Series eBook) IFIT

## 12.2 Traffic Policing and Traffic Shaping

### 12.2.1 Introduction

#### Definition

- Traffic Policing

  Traffic Policing (TP) is a traffic management technology that is applied at the ingress or egress of a NE to limit specified types of data traffic.

- Traffic Shaping

  The traffic shaping adopts the Generic Traffic Shaping (GTS) to shape the traffic that is irregular or does not conform to the preset traffic features, which is convenient for the bandwidth match between the network upstream and downstream.

#### Purpose

To control the traffic that is sent to networks by users is important for ISPs. To control the traffic of certain applications in an enterprise network is a means to manage the network status. A typical application of TP is to monitor the specification of a type of traffic that enters a network. Based on the result, the specification can be limited in a reasonable scope; or the amount of traffic that exceeds the limit is "punished." As a result, the network resources and the interests of a carrier are protected.

A typical application of traffic shaping is to control the normal flow and burst flow of outgoing traffic based on the network connection. Therefore, the packets can be sent at a uniform rate.

#### Benefits

- This feature brings the following benefits to carriers.

  Punish the amount of traffic that exceeds the limit to protect the network resources and the interests of a carrier.

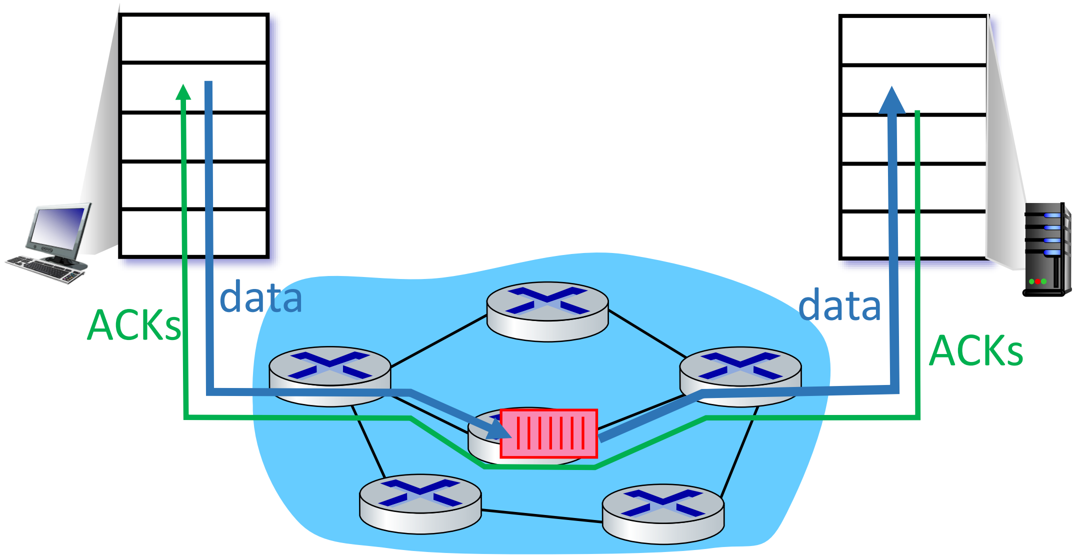
# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
- Principles of congestion control
- Congestion Control Techniques
  - Basic Related Schemes
  - TCP Congestion Control
  - Advanced Schemes

# Approaches towards congestion control

End-end congestion control:

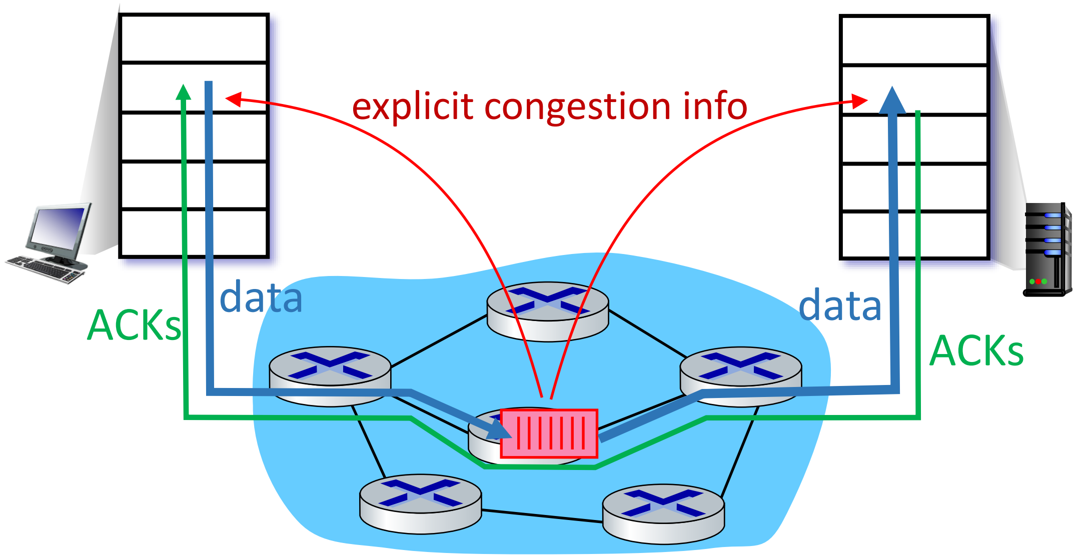
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



# Approaches towards congestion control

## Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECbit protocols



# TCP Congestion Control

- Another key component of TCP is its congestion-control mechanism.

- What we might refer to as "Classic" TCP—the version of TCP standardized in [RFC 2581] and most recently [RFC 5681]—uses end-to-end congestion control rather than network-assisted congestion control, since the IP layer provides no explicit feedback to the end systems regarding network congestion.

# Congestion Control

**Classic TCP Congestion Control: The general approach**

- The approach taken by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion.
  - If a TCP sender perceives that there is little congestion on the path between itself and the destination, then the TCP sender increases its send rate;
  - If the sender perceives that there is congestion along the path, then the sender reduces its send rate.

# Congestion Control

## Classic TCP Congestion Control: The general approach

- But this approach raises three questions.
  - **First**, how does a TCP sender limit the rate at which it sends traffic into its connection?
  - **Second**, how does a TCP sender perceive that there is congestion on the path between itself and the destination?
  - And **third**, what algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

# Congestion Control

**Classic TCP Congestion Control: The general approach**

- Let’s first examine how a TCP sender limits the rate at which it sends traffic into its connection.
- Each side of a TCP connection consists of a receive buffer, a send buffer, and several variables (LastByteRead, rwnd, and so on).
- The TCP congestion-control mechanism operating at the sender keeps track of an additional variable, **the congestion window**.

# Congestion Control

## Classic TCP Congestion Control: The general approach

- The congestion window, denoted **cwnd**, imposes a constraint on the rate at which a TCP sender can send traffic into the network.

- Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of cwnd and rwnd, that is:

$$\mathrm{LastByteSent} - \mathrm{LastByteAcked} \leq \min\{\mathrm{cwnd}, \mathrm{rwnd}\}$$

# Congestion Control

## Classic TCP Congestion Control: The general approach

- In order to focus on congestion control (as opposed to flow control), let us henceforth assume that the TCP receive buffer is so large that the receive-window constraint can be ignored;
  - thus, the amount of unacknowledged data at the sender is solely limited by cwnd.

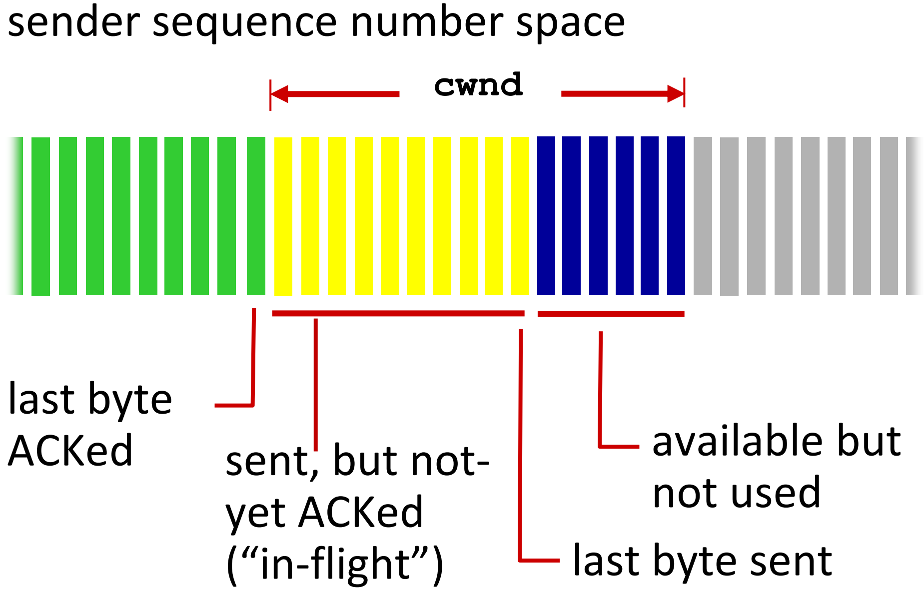
- We also assume that the sender always has data to send, that is, that all segments in the congestion window are sent.

# Congestion Control

**Classic TCP Congestion Control: The general approach**

- The constraint above limits the amount of unacknowledged data at the sender and therefore indirectly limits the sender’s send rate.
- To see this, consider a connection for which loss and packet transmission delays are negligible.
- Then, roughly, at the beginning of every RTT, the constraint permits the sender to send cwnd bytes of data into the connection; at the end of the RTT the sender receives acknowledgments for the data.

# TCP congestion control: details




TCP sending behavior:

- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- TCP sender limits transmission: **`LastByteSent- LastByteAcked ≤ cwnd`**
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)
- *Thus the sender's send rate is roughly cwnd/RTT bytes/sec. By adjusting the value of cwnd, the sender can therefore adjust the rate at which it sends data into its connection.*

# Congestion Control

**Classic TCP Congestion Control: The general approach**

- How a TCP sender perceives that there is congestion on the path between itself and the destination?

  - Let us define a “loss event” at a TCP sender as the occurrence of either a timeout or the receipt of three duplicate ACKs from the receiver.

  - When there is excessive congestion, then one (or more) router buffers along the path overflows, causing a datagram (containing a TCP segment) to be dropped.

  - The dropped datagram, in turn, results in a loss event at the sender—either a timeout or the receipt of three duplicate ACKs—which is taken by the sender to be an indication of congestion on the sender-to-receiver path

# Congestion Control

**Classic TCP Congestion Control: The general approach >> Self-clocking**

- Consider the more optimistic case when the network is congestion-free, that is, when a loss event doesn't occur.
- In this case, acknowledgments for previously unacknowledged segments will be received at the TCP sender.
- TCP takes the arrival of these acknowledgments as an **indication** that all is well use acknowledgments to increase its congestion window size (and hence its transmission rate).
  - Note that if acknowledgments **arrive at a relatively slow rate** (e.g., if the end-end path has high delay or contains a low-bandwidth link), then the congestion window will be increased at **a relatively slow rate**.
  - On the other hand, if acknowledgments arrive at a **high rate**, then the congestion window **will be increased more quickly**.
- Because TCP uses acknowledgments to trigger (or clock) its increase in congestion window size, TCP is said to be **self-clocking.**

# Congestion Control

**Classic TCP Congestion Control: The general approach**

- Given the mechanism of adjusting the value of cwnd to control the sending rate, the critical question remains:

  - ❑ How then do the TCP senders determine their sending rates such that they don’t congest the network but at the same time make use of all the available bandwidth?

  - ❑ Are TCP senders explicitly coordinated, or is there a distributed approach in which the TCP senders can set their sending rates based only on local information?

# Congestion Control

**Classic TCP Congestion Control: The general approach**

- TCP answers these questions using the following **guiding principles**:
  - A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost.
  - An acknowledged segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment.
  - Bandwidth probing

# Congestion Control

## Classic TCP Congestion Control: The general approach

- The celebrated TCP congestion-control algorithm, which was first described by Jacobson in 1988 and is standardized in RFC 5681.

- The algorithm has three major components:
    - slow start,
    - congestion avoidance,
    - fast recovery.

# Congestion Control

## Classic TCP Congestion Control: The general approach

- **Some Remarks:**

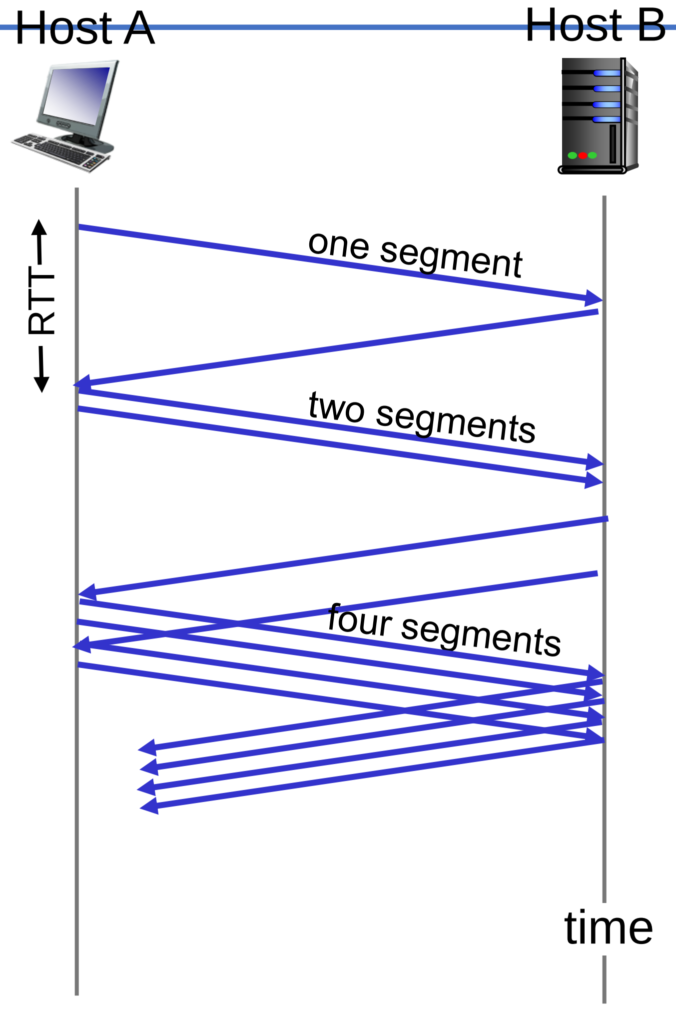
❖ Slow start and congestion avoidance are **mandatory** components of TCP, differing in how they increase the size of cwnd in response to received ACKs.

❖ Slow start increases the size of cwnd more rapidly (despite its name!) than congestion avoidance.

❖ Fast recovery is recommended, but not required, for TCP senders.

# TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



# Congestion Control

## Classic TCP Congestion Control: Slow start

- But when should this exponential growth end?

- Slow start provides several answers to this question.

❖ **First**, if there is a loss event (i.e., congestion) indicated by a timeout, the TCP sender sets the value of cwnd to 1 and begins the slow start process anew.

  - It also sets the value of a second state variable, **ssthresh** (shorthand for "slow start threshold") to cwnd/2—half of the value of the congestion window value when congestion was detected.

# Congestion Control

## Classic TCP Congestion Control: Slow start

- The **second** way in which slow start may end is directly tied to the value of ssthresh.
  - Since ssthresh is half the value of cwnd when congestion was last detected, it might be a bit reckless to keep doubling cwnd when it reaches or surpasses the value of ssthresh.
  - Thus, when the value of cwnd equals ssthresh, **slow start ends and TCP transitions into congestion avoidance mode.** TCP increases cwnd more cautiously when in congestion-avoidance mode.
- The **final** way in which slow start can end is if three duplicate ACKs are detected, in which case TCP performs a fast retransmit and **enters the fast recovery state**, as discussed further.

# Congestion Control

## Classic TCP Congestion Control: Congestion avoidance

- On entry to the congestion-avoidance state, the value of cwnd is approximately half its value when congestion was last encountered.

- Thus, rather than doubling the value of cwnd every RTT, TCP adopts a more conservative approach and increases the value of cwnd by just a single MSS every RTT [RFC 5681].

- This can be accomplished in several ways. **A common approach** is for the TCP sender to increase cwnd by MSS bytes (MSS/cwnd) whenever a new acknowledgment arrives.

# Congestion Control

## Classic TCP Congestion Control: Congestion avoidance

- For example, if MSS is 1,460 bytes and cwnd is 14,600 bytes, then 10 segments are being sent within an RTT.

- Each arriving ACK (assuming one ACK per segment) increases the congestion window size by 1/10 MSS, and thus, the value of the congestion window will have increased by one MSS after ACKs when all 10 segments have been received.

- ❑ *But when should congestion avoidance's linear increase (of 1 MSS per RTT) end?*

# Congestion Control

## Classic TCP Congestion Control: Congestion avoidance

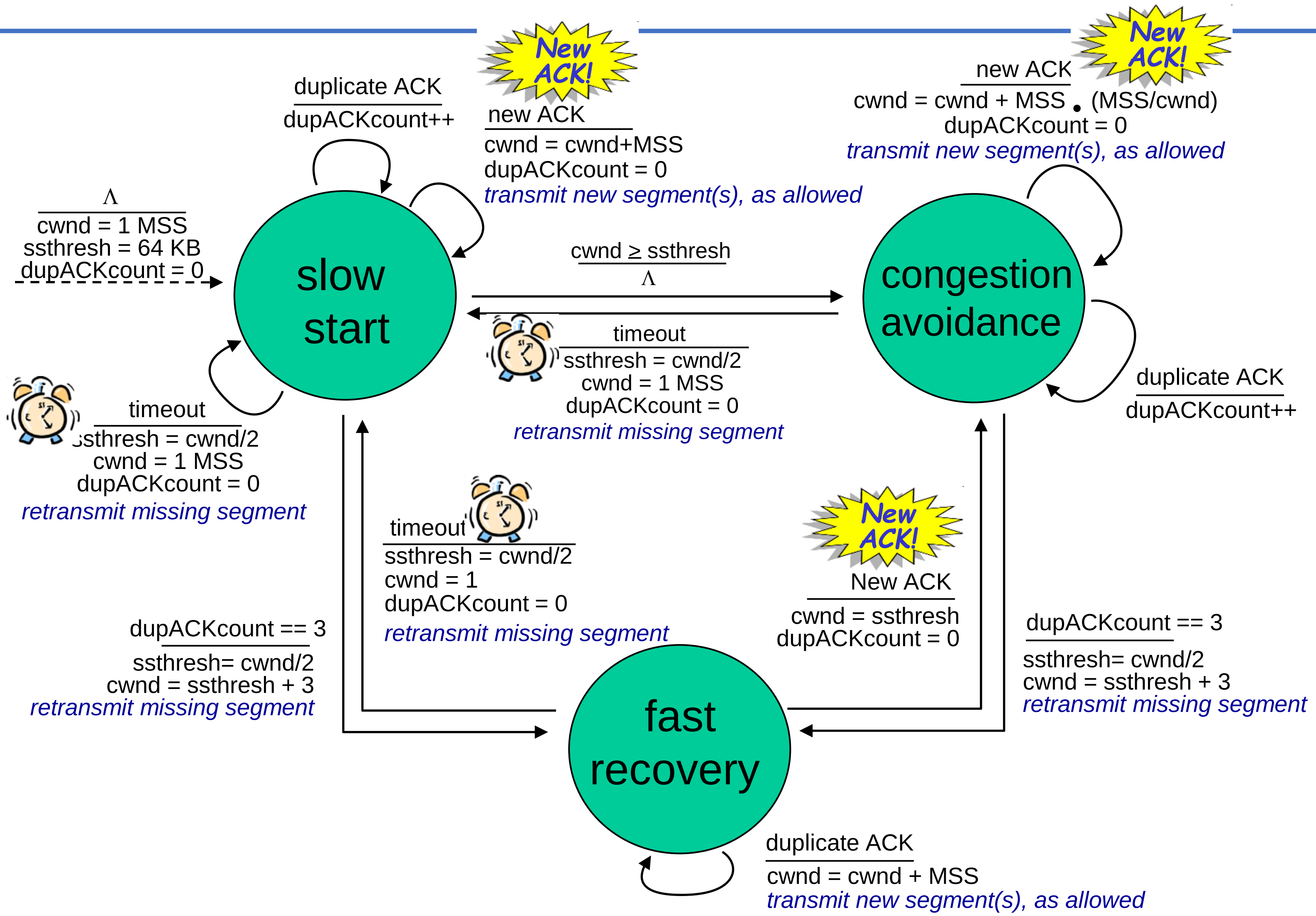
- TCP's congestion-avoidance algorithm behaves the same when a **timeout** occurs as in the case of slow start: The value of cwnd is set to 1 MSS, and the value of ssthresh is updated to half the value of cwnd when the loss event occurred.
- Recall, however, that a loss event also can be triggered by a **triple duplicate ACK** event.
  - In this case, the network is continuing to deliver some segments from sender to receiver (as indicated by the receipt of duplicate ACKs).
  - So TCP's behavior to this type of loss event should be less drastic than with a timeout-indicated loss: TCP halves the value of cwnd (adding in 3 MSS for good measure to account for the triple duplicate ACKs received) and records the value of ssthresh to be half the value of cwnd when the triple duplicate ACKs were received. **The fast-recovery state is then entered.**

# Congestion Control

## Classic TCP Congestion Control: Fast Recovery

- In fast recovery, the value of cwnd is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state.
- Eventually, when an ACK arrives for the missing segment, TCP enters the **congestion-avoidance state**.
- If a timeout event occurs, fast recovery transitions to the **slow-start state** after performing the same actions as in slow start and congestion avoidance: The value of cwnd is set to 1 MSS, and the value of ssthresh is set to half the value of cwnd when the loss event occurred.

# Summary: TCP congestion control



# Congestion Control

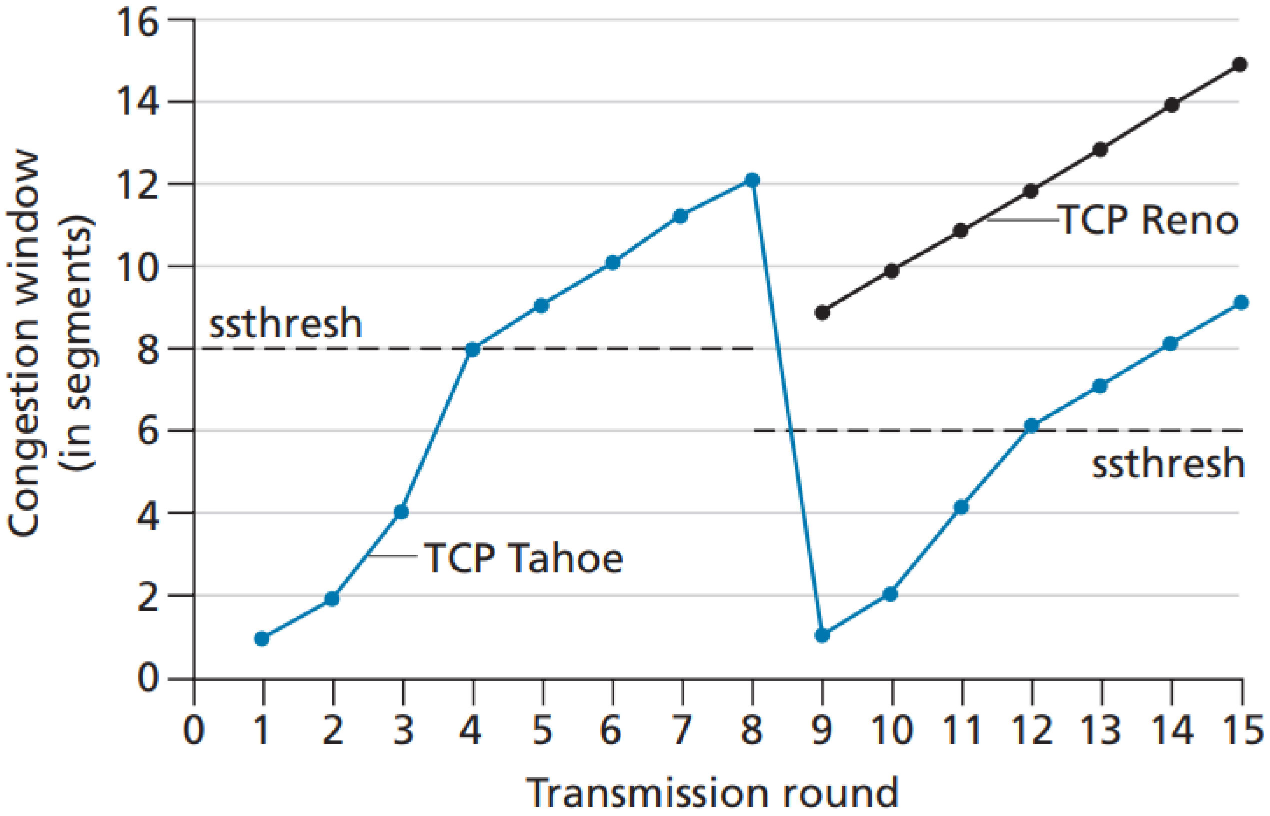
## Classic TCP Congestion Control: Fast Recovery

- Fast recovery is a recommended, but not required, component of TCP [RFC 5681].
- It is interesting that an early version of TCP, known as **TCP Tahoe**, unconditionally cut its congestion window to 1 MSS and entered the slow-start phase after either a timeout-indicated or triple-duplicate-ACK-indicated loss event.
- The newer version of TCP, **TCP Reno**, incorporated fast recovery.

# Congestion Control

## Classic TCP Congestion Control

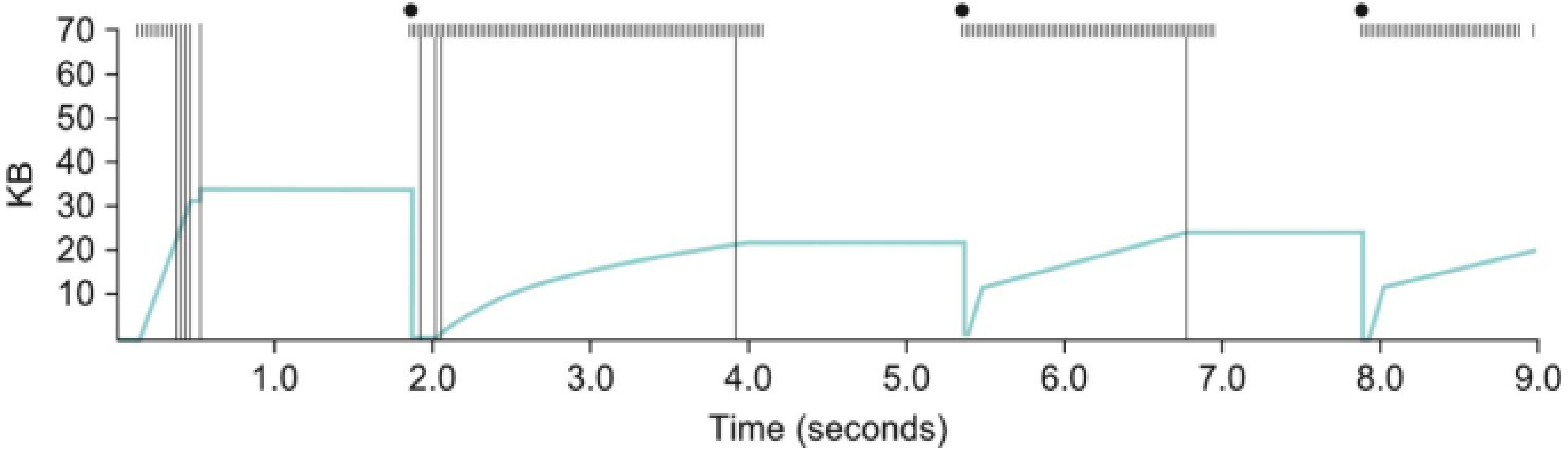
- Example



# Congestion Control

## Classic TCP Congestion Control

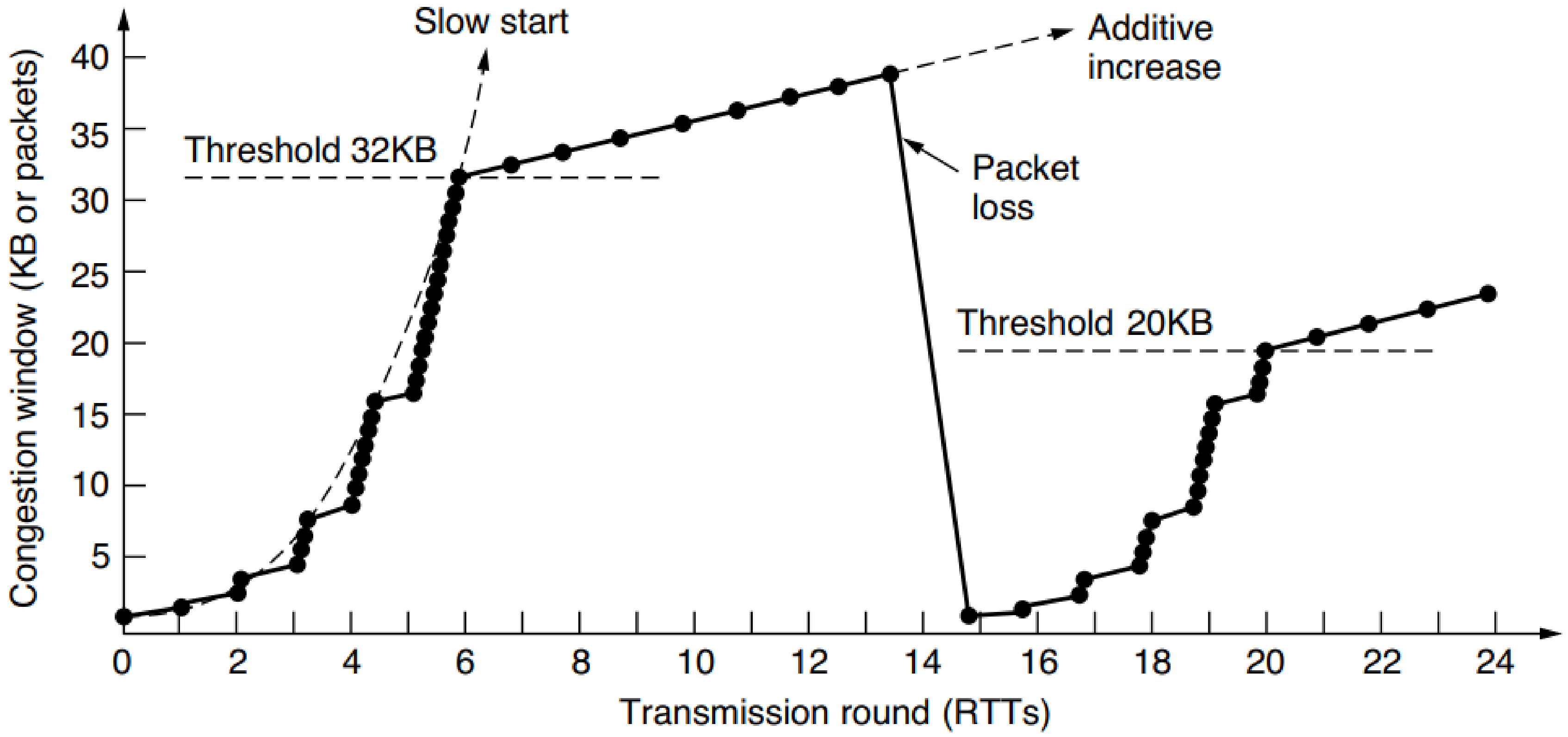
- Example: a trace of TCP congestion control mechanism (similar to Tahoe)



# Congestion Control

## Classic TCP Congestion Control

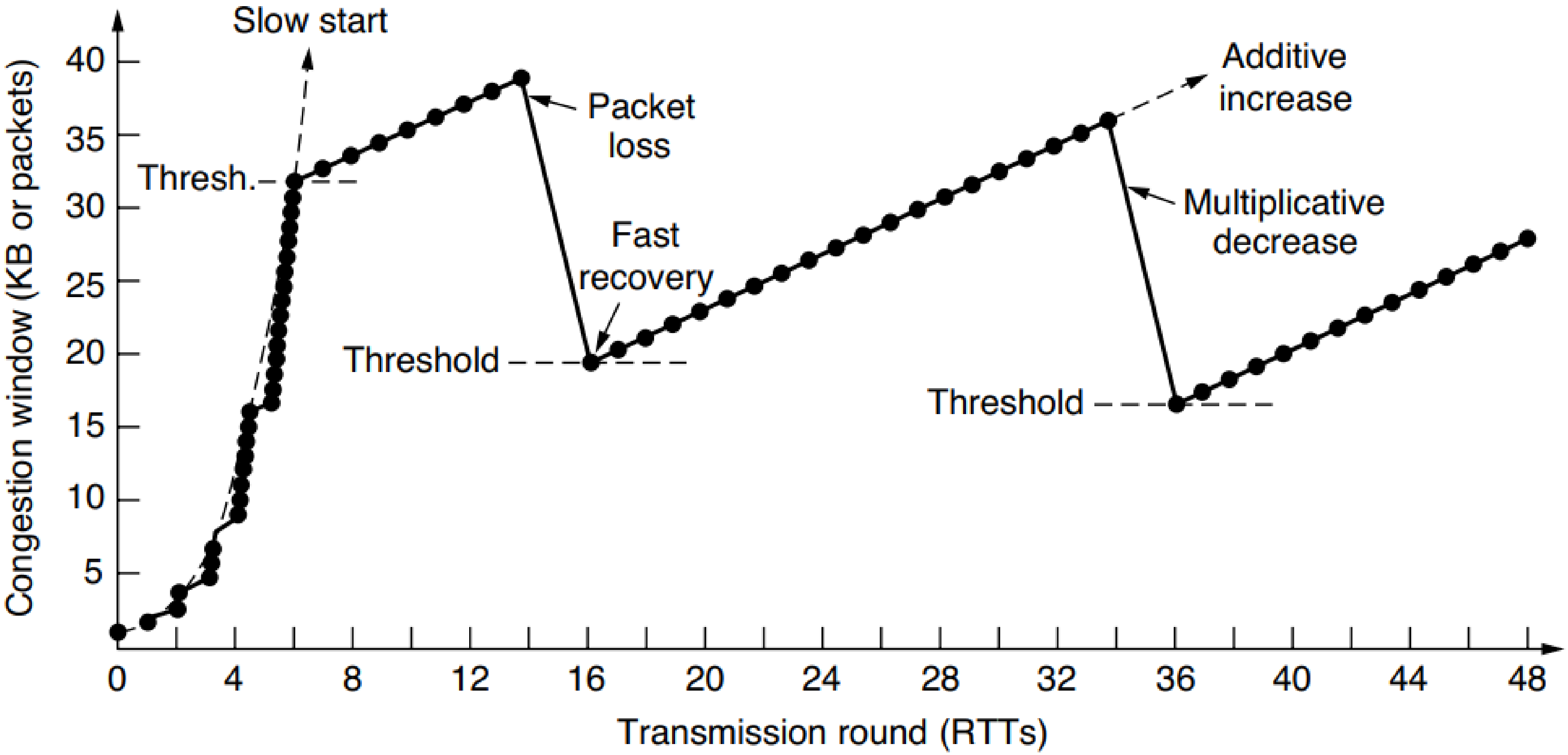
- TCP Tahoe



# Congestion Control

## Classic TCP Congestion Control

- TCP Reno



# TCP congestion control: AIMD

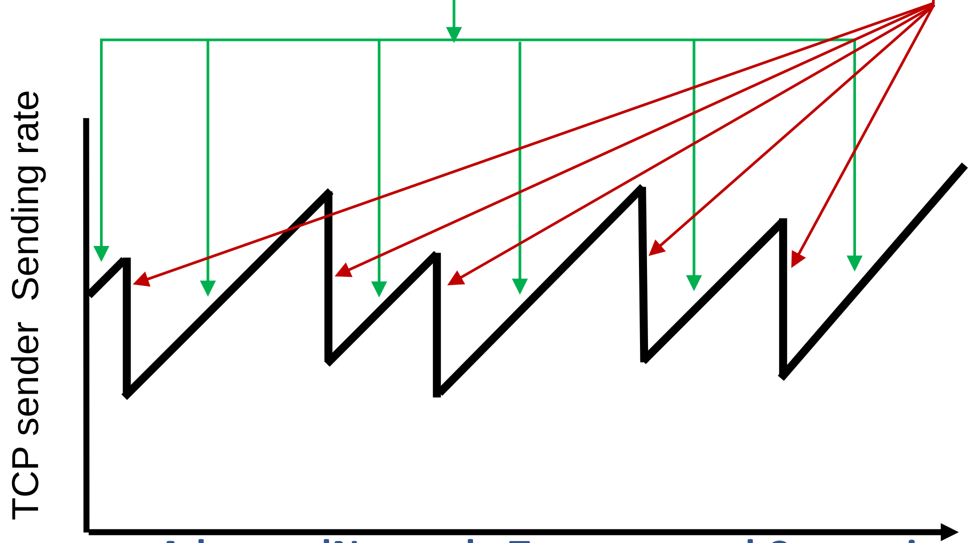
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

*Additive Increase*

increase sending rate by 1 maximum segment size every RTT until loss detected

*Multiplicative Decrease*

cut sending rate in half at each loss event




**AIMD** sawtooth behavior: *probing* for bandwidth

# TCP congestion control: AIMD

*Multiplicative decrease* detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

# Congestion Control

## Classic TCP Congestion Control: Retrospective

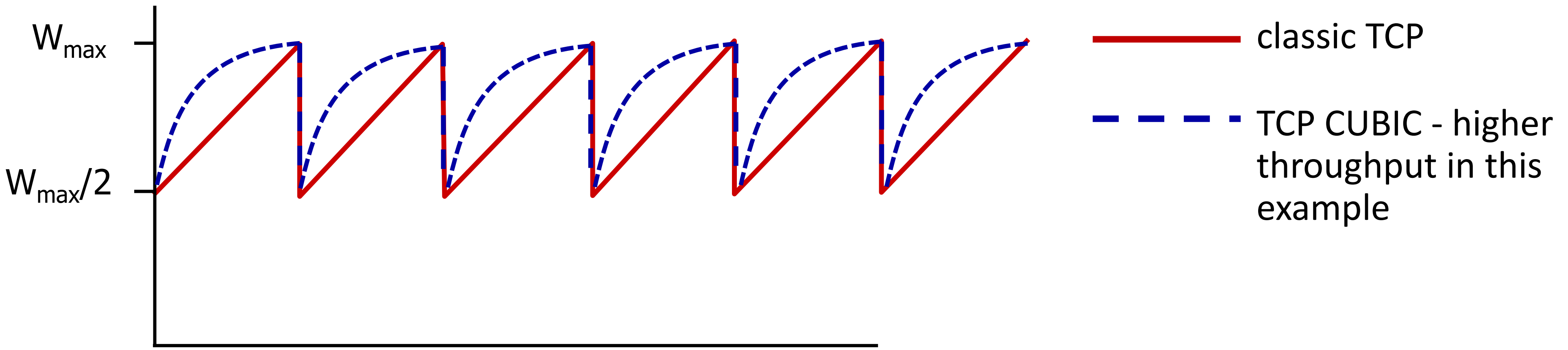
- TCP’s AIMD algorithm was developed based on a tremendous amount of engineering insight and experimentation with congestion control in operational networks.

- Ten years after TCP’s development, theoretical analyses showed that TCP’s congestion-control algorithm serves as a distributed asynchronous-optimization algorithm that results in several important aspects of user and network performance being simultaneously optimized by F. Kelly in 1998.

- A rich theory of congestion control has since been developed by R. Srikant.

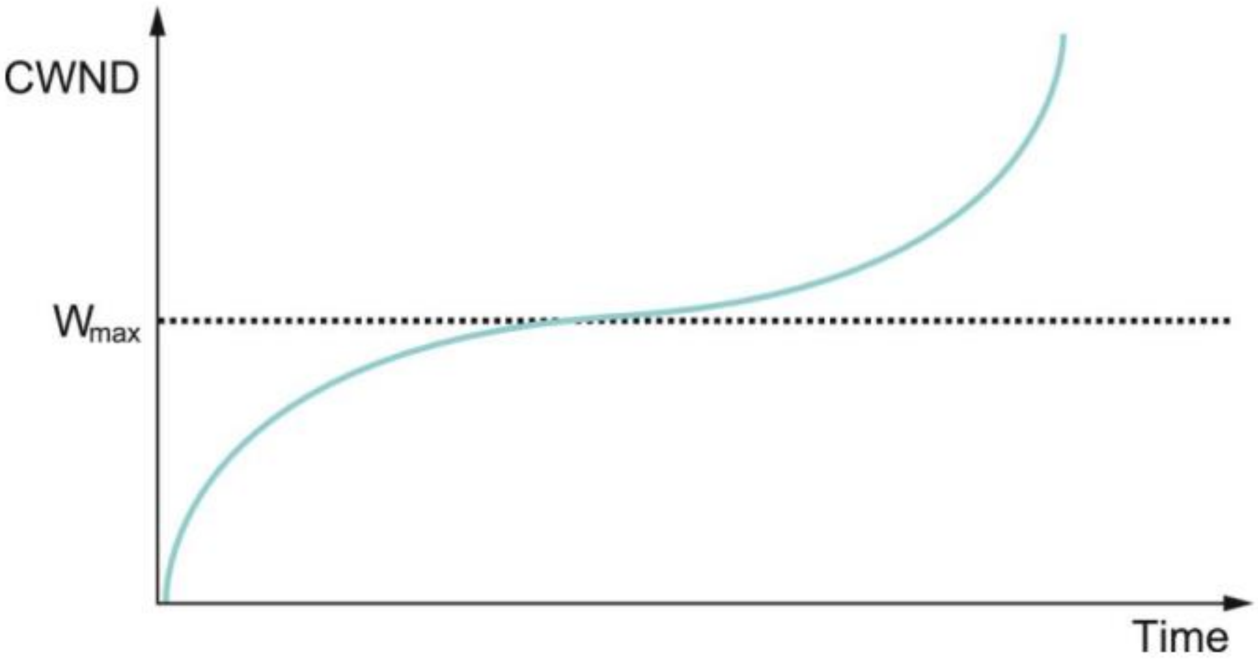
# Congestion Control: TCP CUBIC

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - $W_{max}$: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to $W_{max}$ *faster*, but then approach $W_{max}$ more *slowly*



# Congestion Control: TCP CUBIC

- Generic cubic function illustrating the change in the congestion window as a function of time.



# Congestion Control: TCP CUBIC

- Specifically, CUBIC computes the congestion window as a function of time (t) since the last congestion event:

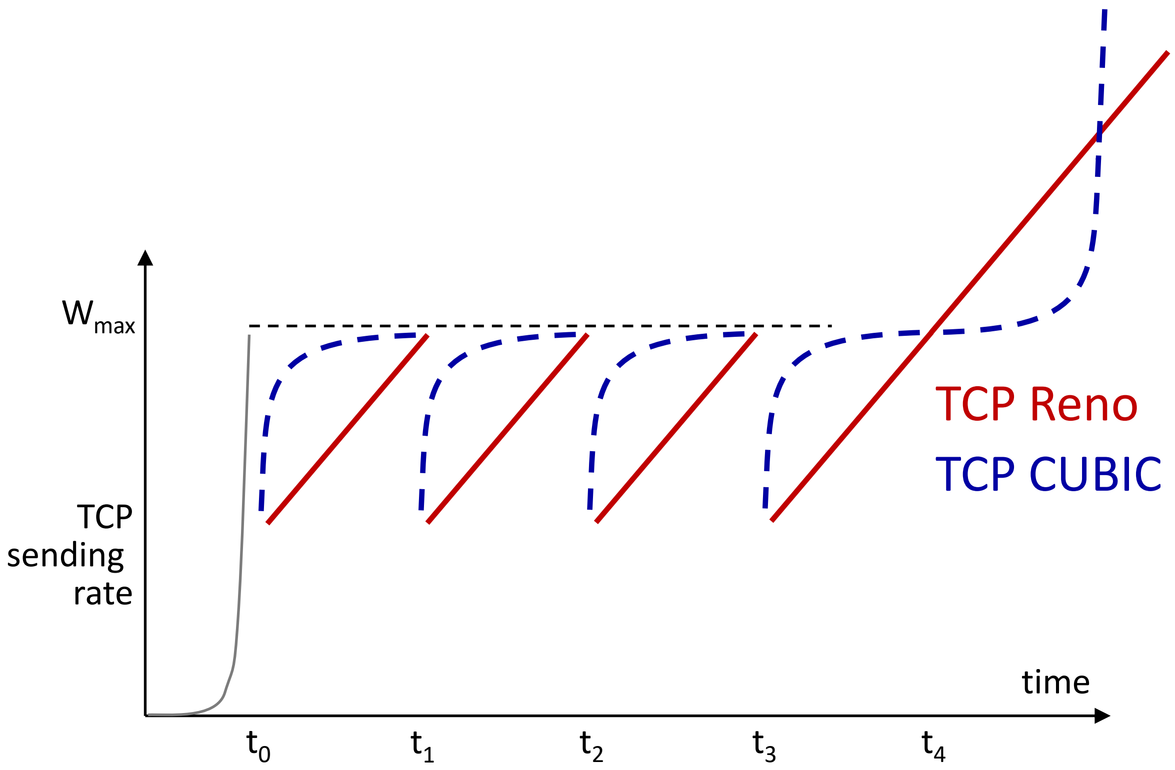
$$\mathrm{CWND}(t) = C \times (t - \mathrm{K})^3 + W_{max}$$

- K: point in time when TCP window size will reach $W_{max}$
  - K itself is tuneable

- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K

# Congestion Control: TCP CUBIC

- TCP CUBIC default in Linux, most popular TCP for popular Web servers



# Congestion Control: TCP CUBIC

**Bandwidth-Delay Product (BDP)**

- Consider we are multiply bandwidth of a link (from source to destination) by its RTT, and name it BDP.

- What does the value of BDP tell us?
  - It is the amount of data a TCP sender push into the network until the receipt of the first ACK.
  - The amount of data in the buffer space of router is close to BDP.

- Networks with large BPD (said to be more than 12.5 kB) are called long-fat networks (LFNs).

# Congestion Control: TCP CUBIC

**Bandwidth-Delay Product (BDP)**

- CUBIC's primary goal is to support networks with large delay × BW products, or LFNs.
- Such networks suffer from the original TCP algorithm requiring too many round-trips to reach the available capacity of the end-to-end path.
- CUBIC does this by being more aggressive in how it increases the window size, but of course the trick is to be more aggressive without being so aggressive as to adversely affect other flows.

# Congestion Control: TCP CUBIC

**Bandwidth-Delay Product (BDP)**

- One important aspect of CUBIC's approach is to adjust its congestion window at regular intervals, based on the amount of time that has elapsed since the last congestion event (e.g., the arrival of a duplicate ACK) rather than only when ACKs arrive (the latter being a function of RTT).

- This allows CUBIC to behave fairly when competing with short-RTT flows, which will have ACKs arriving more frequently.

- The idea is to start fast but slow the growth rate as you get close to $W_{max}$, be cautious and have near-zero growth when close to $W_{max}$, and then increase the growth rate as you move away from $W_{max}$. The latter phase is essentially probing for a new achievable $W_{max}$.

# Congestion Control: TCP CUBIC

## Classic TCP Congestion Control: TCP Cubic

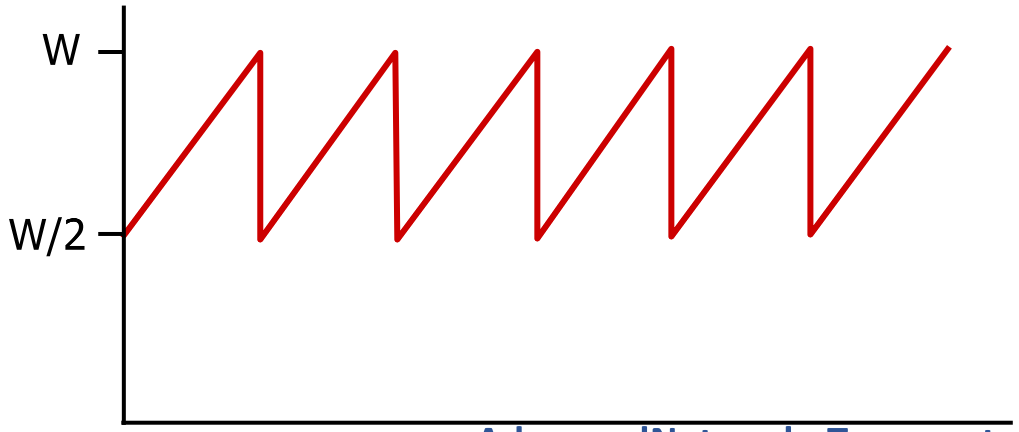
- TCP CUBIC has recently gained wide deployment.

While measurements taken around 2000 on popular Web servers showed that nearly all were running some version of TCP Reno, more recent measurements of the 5000 most popular Web servers shows that nearly 50% are running a version of TCP CUBIC, which is also the default version of TCP used in the Linux operating system.

# TCP Reno throughput: A Macroscopic Description

- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume there is always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



# Congestion Control

## TCP Splitting, Optimizing The Performance Of Cloud Services

- For cloud services such as search, e-mail, and social networks, it is desirable to provide a high-level of responsiveness, ideally giving users the illusion that the services are running within their own end systems (including their smartphones).

- This can be a major challenge, as users are often located far away from the data centers responsible for serving the dynamic content associated with the cloud services.

- Indeed, if the end system is far from a data center, then the RTT will be large, potentially leading to poor response time performance due to TCP slow start.

# Congestion Control

## TCP Splitting, Optimizing The Performance Of Cloud Services

- As a case study, consider the delay in receiving a response for a search query.

- Typically, the server requires three TCP windows during slow start to deliver the response [Pathak 2010].

- Thus the time from when an end system initiates a TCP connection until the time when it receives the last packet of the response is roughly 4 × RTT (one RTT to set up the TCP connection plus three RTTs for the three windows of data) plus the processing time in the data center.

# Congestion Control

## TCP Splitting, Optimizing The Performance Of Cloud Services

- These RTT delays can lead to a noticeable delay in returning search results for a significant fraction of queries.

- Moreover, there can be significant packet loss in access networks, leading to TCP retransmissions and even larger delays.

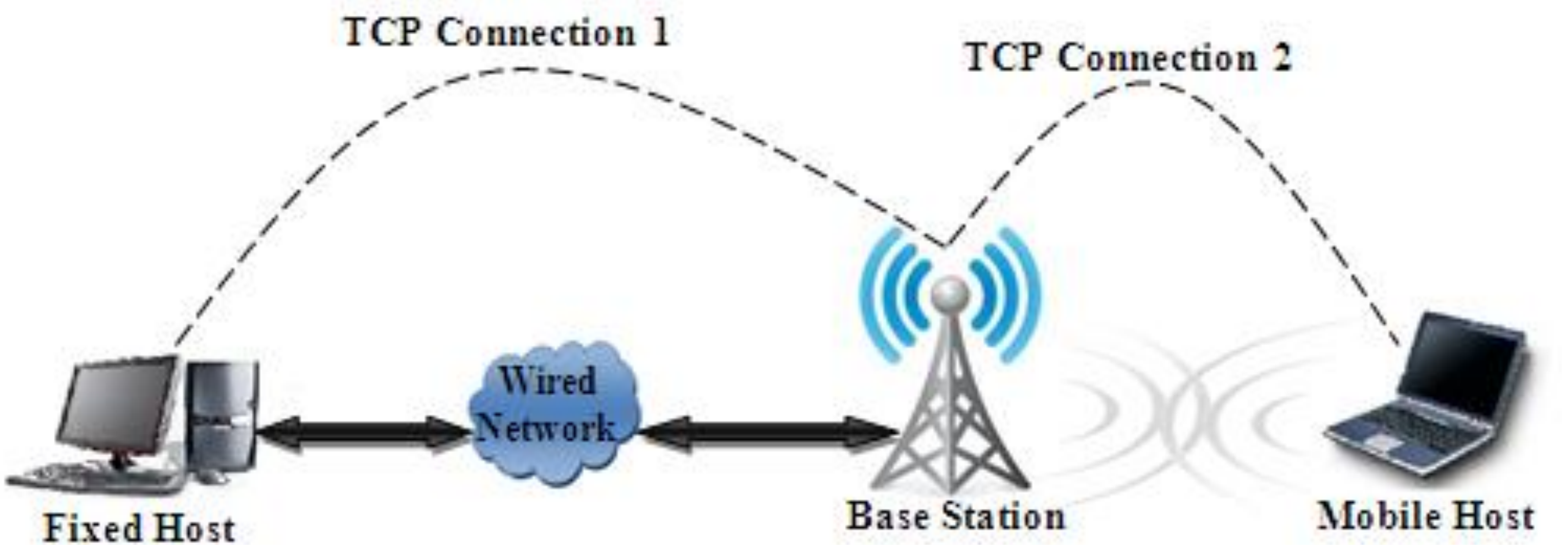
➢ One way to mitigate this problem and improve user-perceived performance is to

1. Deploy front-end servers closer to the users,
2. Utilize TCP splitting by breaking the TCP connection at the front-end server.

# Congestion Control

## TCP Splitting, Optimizing The Performance Of Cloud Services

- With TCP splitting, the client establishes a TCP connection to the nearby front-end, and the front-end maintains a persistent TCP connection to the data center with a **very large TCP congestion window**.



# Congestion Control

**TCP Splitting, Optimizing The Performance Of Cloud Services**

- With this approach, the response time roughly becomes

$$4 \times RTT_{FE} + RTT_{BE} + \text{processing time},$$

where $RTT_{FE}$ is the round-trip time between client and front-end server, and $RTT_{BE}$ is the round-trip time between the front-end server and the data center (back-end server).

- If the front-end server is close to client, then this response time approximately becomes RTT plus processing time, since $RTT_{FE}$ is negligibly small and $RTT_{BE}$ is approximately RTT.

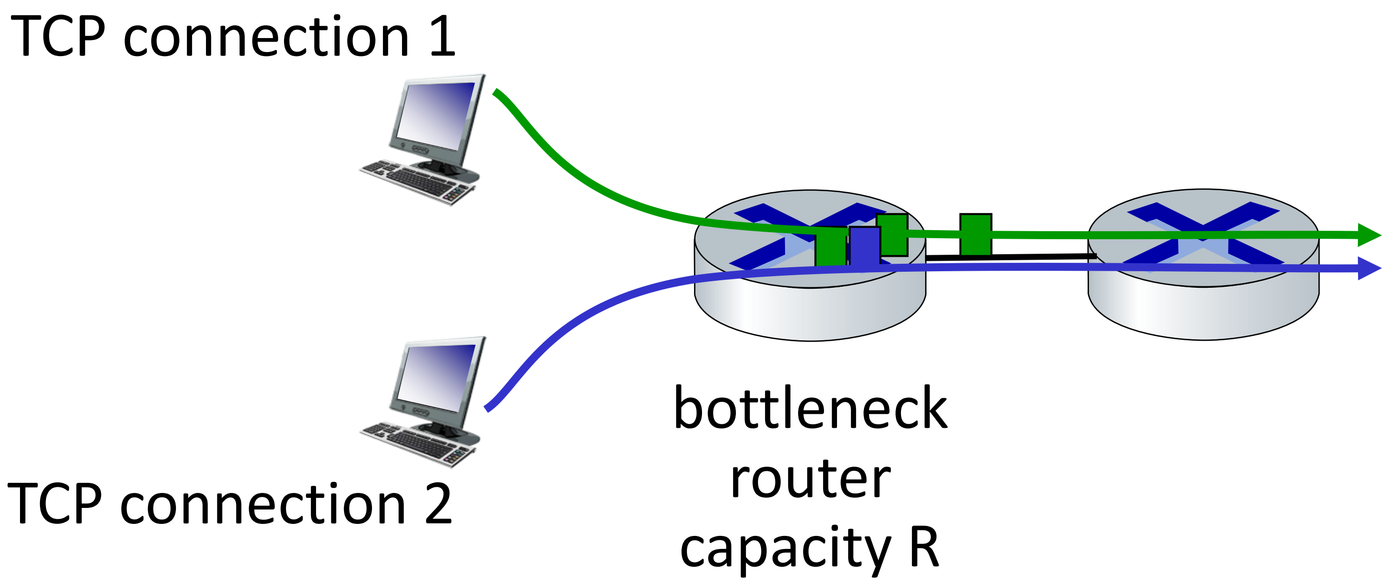
# Congestion Control

## TCP Splitting, Optimizing The Performance Of Cloud Services

- In summary, TCP splitting can reduce the networking delay roughly from $4 \times$ RTT to RTT, significantly improving user-perceived performance, particularly for users who are far from the nearest data center.
- TCP splitting also helps reduce TCP retransmission delays caused by losses in access networks.
- Google and Akamai have made extensive use of their CDN servers in access networks to perform TCP splitting for the cloud services they support.

# TCP fairness

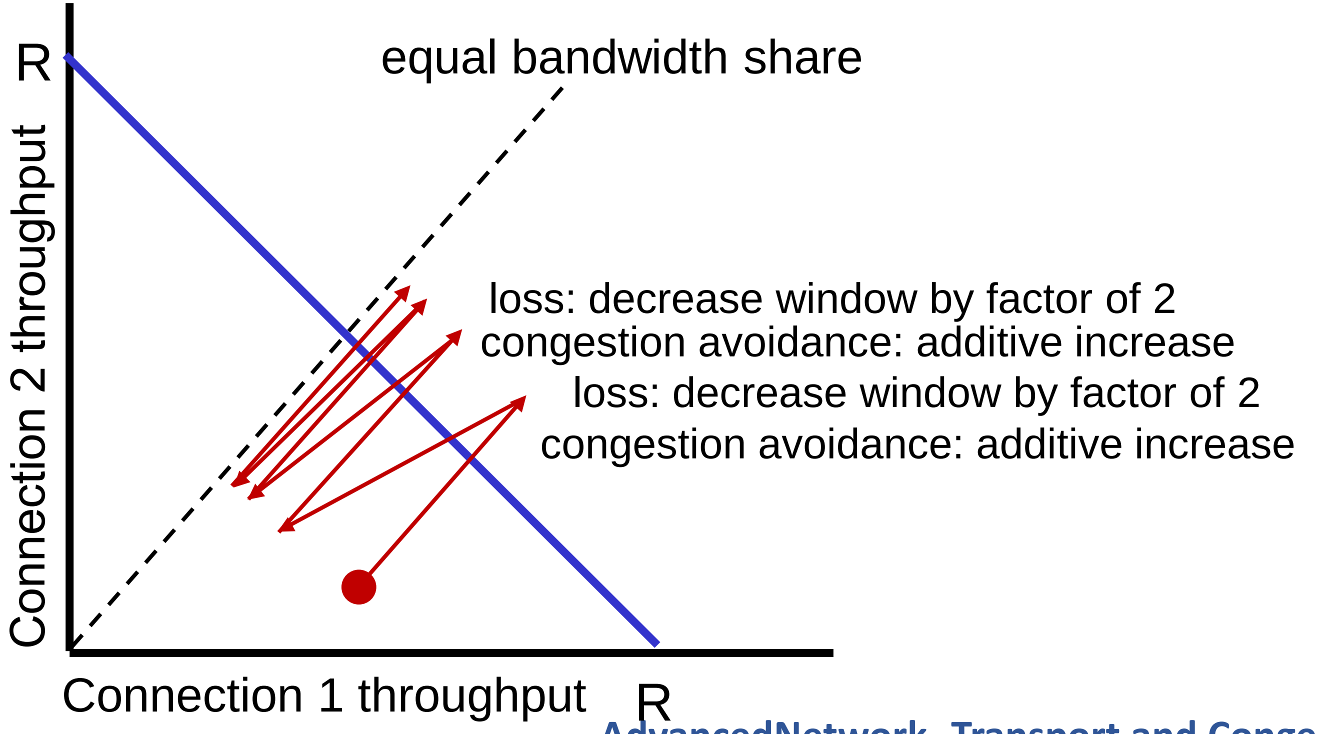
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



# Q: is TCP Fair?

Example: two competing TCP sessions:

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




*Is* TCP fair?

*A:* Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

# Fairness: must all network apps be "fair"?

## Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no "Internet police" policing use of congestion control

## Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2