w12
(1b)

Computer Networks II

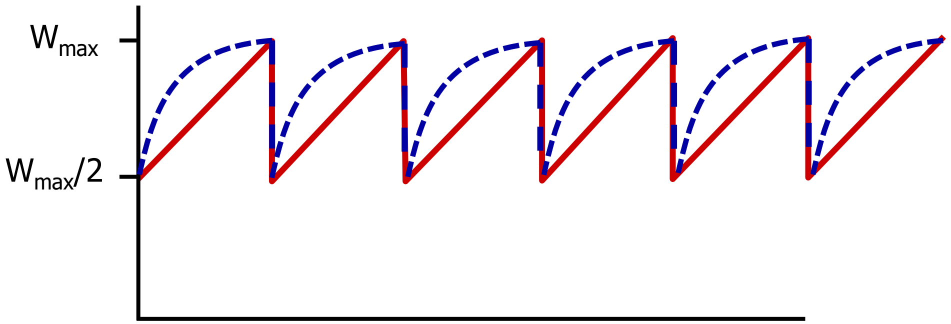
TCP Congestion and Flow Control

Other Variants, TCP Fairness, Flow Control

Amitangshu Pal
Computer Science and Engineering
IIT Kanpur

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*



TCP CUBIC (Cubic TCP) is a congestion control algorithm that aims to improve the probing of available bandwidth compared to traditional Additive Increase Multiplicative Decrease (AIMD) algorithms.

Here's an easy way to understand it:

Probing for Usable Bandwidth:

TCP CUBIC seeks to find the maximum sending rate at which congestion was detected (called W_max) more efficiently than AIMD.
It does this by initially ramping up the sending rate faster after a congestion event but then gradually approaching W_max more slowly.

Insight/Intuition:

When congestion occurs, the bottleneck link's congestion state probably hasn't changed much.
So, TCP CUBIC assumes that the network can handle a higher rate of traffic if it has experienced congestion recently.
How TCP CUBIC Works:

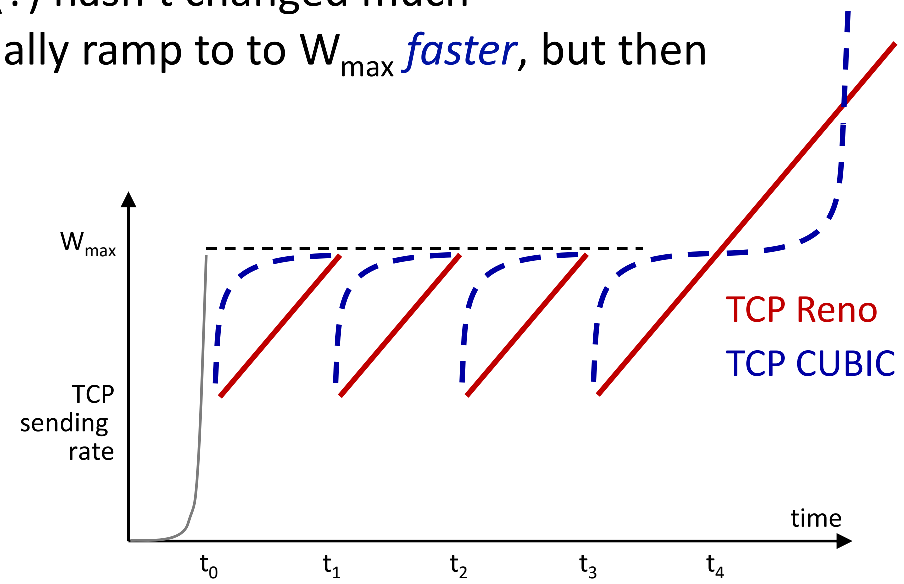
After a Congestion Event:
When TCP CUBIC detects congestion (such as packet loss), it reduces its congestion window or sending rate (similar to AIMD).
But instead of just cutting the rate/window in half like AIMD, TCP CUBIC remembers the rate at which congestion was detected (W_max).
Ramping Up Sending Rate:
After reducing the rate/window on congestion, TCP CUBIC ramps up its sending rate more aggressively initially, trying to reach W_max faster.

Slowing Down Near W_max:
As the sending rate approaches W_max, TCP CUBIC slows down its rate of increase, ensuring that it doesn't overshoot W_max.
TCP CUBIC in Practice:
TCP CUBIC is the default TCP congestion control algorithm in the Linux kernel and is widely used by popular web servers.
It's designed to handle high-speed, high-capacity networks more effectively than traditional AIMD algorithms, especially in scenarios with large bandwidth-delay products.
In simpler terms, TCP CUBIC is like a car accelerating quickly after a red light but then gradually slowing down as it approaches the speed limit. It probes for available bandwidth by ramping up speed faster after congestion but then slowing down as it gets closer to the maximum speed at which congestion was detected

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*

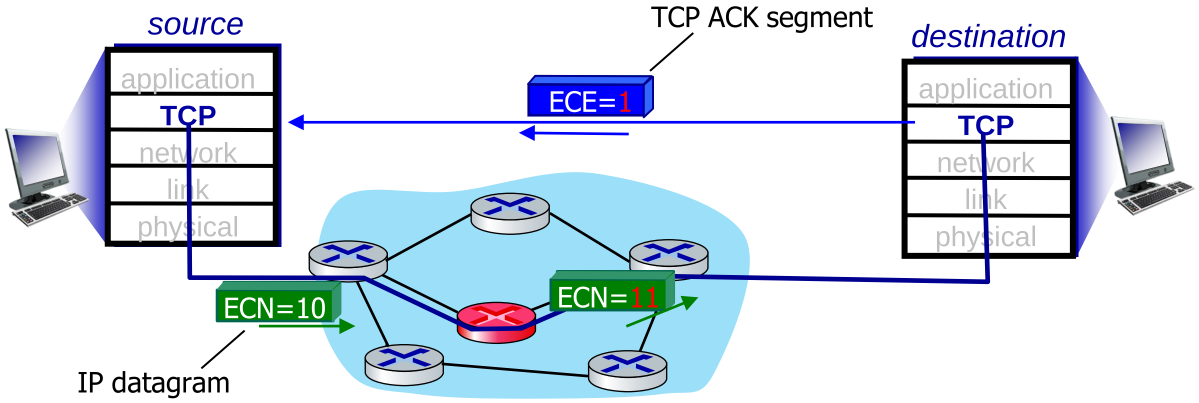
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



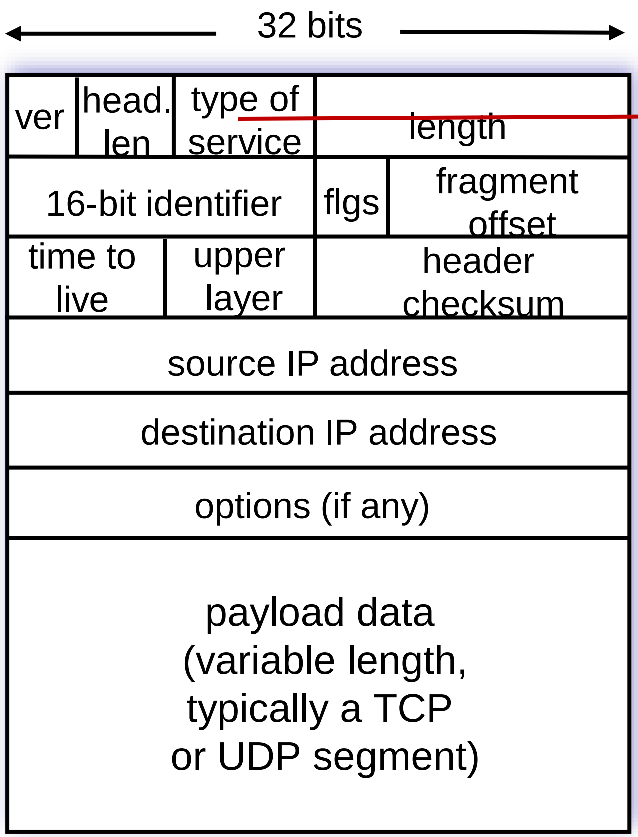
Explicit congestion notification (ECN)

TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)

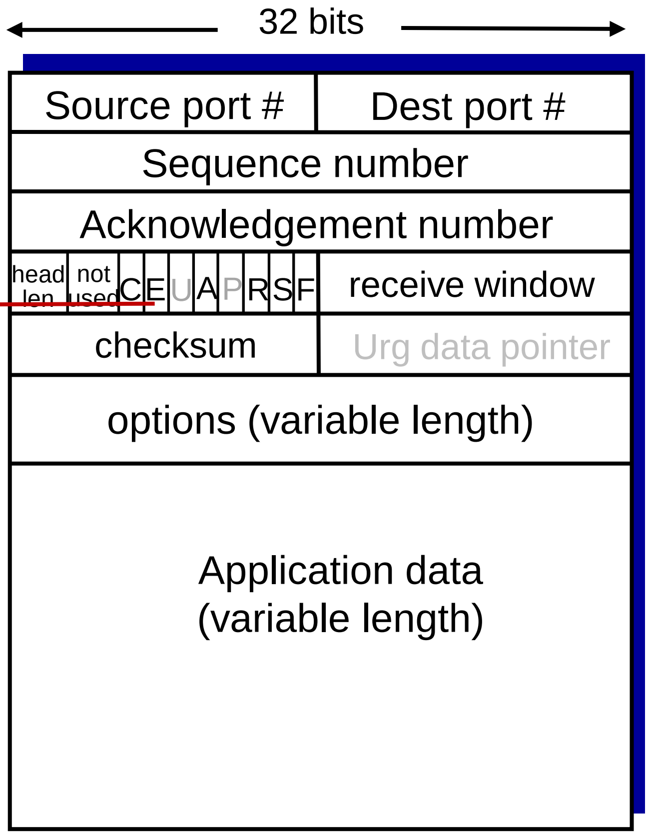


Explicit congestion notification (ECN)



"type" of service:
- diffserv (0:5)
- ECN (6:7)

C, E:
congestion notification



1. Network-Assisted Congestion Control:

ECN allows routers in the network to notify the sender of congestion instead of waiting for packet loss to occur.
Two bits in the IP header's Type of Service (ToS) field are used to mark congestion by the network router.
2. Marking by Routers:

Routers mark packets with these two bits when they experience congestion.
The policy to determine when to mark packets is chosen by the network operator based on various factors like traffic load, link utilization, etc.
3. Congestion Indication Carried to Destination:

When marked packets reach their destination, the congestion indication is carried to the TCP receiver.
4. Notifying Sender:

The receiver sets the ECN-Echo (ECE) bit on the acknowledgment (ACK) segment to notify the sender of congestion.
This indicates to the sender that congestion is occurring in the network.
5. Involvement of IP and TCP:

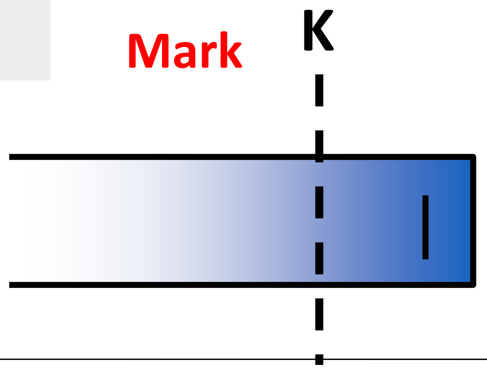
ECN involves both IP and TCP layers.
IP Header: ECN bit marking is done in the IP header.
TCP Header: TCP header has bits to handle ECN - the ECN-Echo (ECE) bit and the Congestion Window Reduced (CWR) bit.
In summary, ECN enables routers to mark packets to indicate congestion, and this information is communicated to the TCP sender via the ECN-Echo (ECE) bit in TCP acknowledgments. This allows for more proactive congestion control without relying solely on packet loss as an indication of congestion.

DCTCP: Two Key Ideas

1. React in proportion to the **extent** of congestion, not its **presence**.
   - ✓ Reduces **variance** in sending rates, lowering queuing requirements.

ECN Marks	TCP	DCTCP
1 0 1 1 1 1 0 1 1 1	Cut window by 50%	Cut window by 40%
0 0 0 0 0 0 0 0 0 1	Cut window by 50%	Cut window by 5%

2. Mark based on **instantaneous** queue length.
   - ✓ Fast feedback to better deal with bursts.



In simple terms, DCTCP adjusts its sending rate based on how severe the congestion is (not just if congestion is present) and uses the instantaneous queue length to quickly respond to congestion events, ensuring a smoother and more efficient data transmission in data center networks.

1. Proportional Reaction to Congestion:

Traditional TCP algorithms often react to congestion by cutting the congestion window (sending rate) in half when they detect any sign of congestion, such as packet loss.
DCTCP, however, reacts proportionally to the extent of congestion. This means it adjusts its congestion window based on how severe the congestion is, rather than just reacting if congestion is present.
For example, if the network is only moderately congested, DCTCP might reduce its congestion window by a smaller proportion compared to a severe congestion scenario.
This allows DCTCP to respond more accurately to different levels of congestion and avoid overreacting, which can lead to unnecessary reductions in throughput.

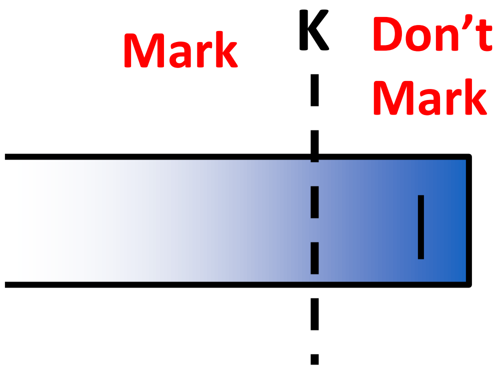
Maintaining High Throughput:

By reacting proportionally and reducing the congestion window by a smaller proportion, DCTCP can maintain higher throughput even during congestion.
It allows DCTCP to balance the need for congestion control with the goal of maximizing throughput.
DCTCP achieves this by ensuring that its response to congestion is more nuanced and adaptive to varying levels of congestion in the network.
In summary, DCTCP's approach to congestion control is more nuanced and adaptive compared to traditional TCP algorithms. By reacting proportionally and reducing the congestion window by a smaller proportion, DCTCP maintains high throughput while still appropriately responding to congestion in the network.

Data Center TCP Algorithm

Switch side:

- Mark packets when **Queue Length > K**



Sender side:

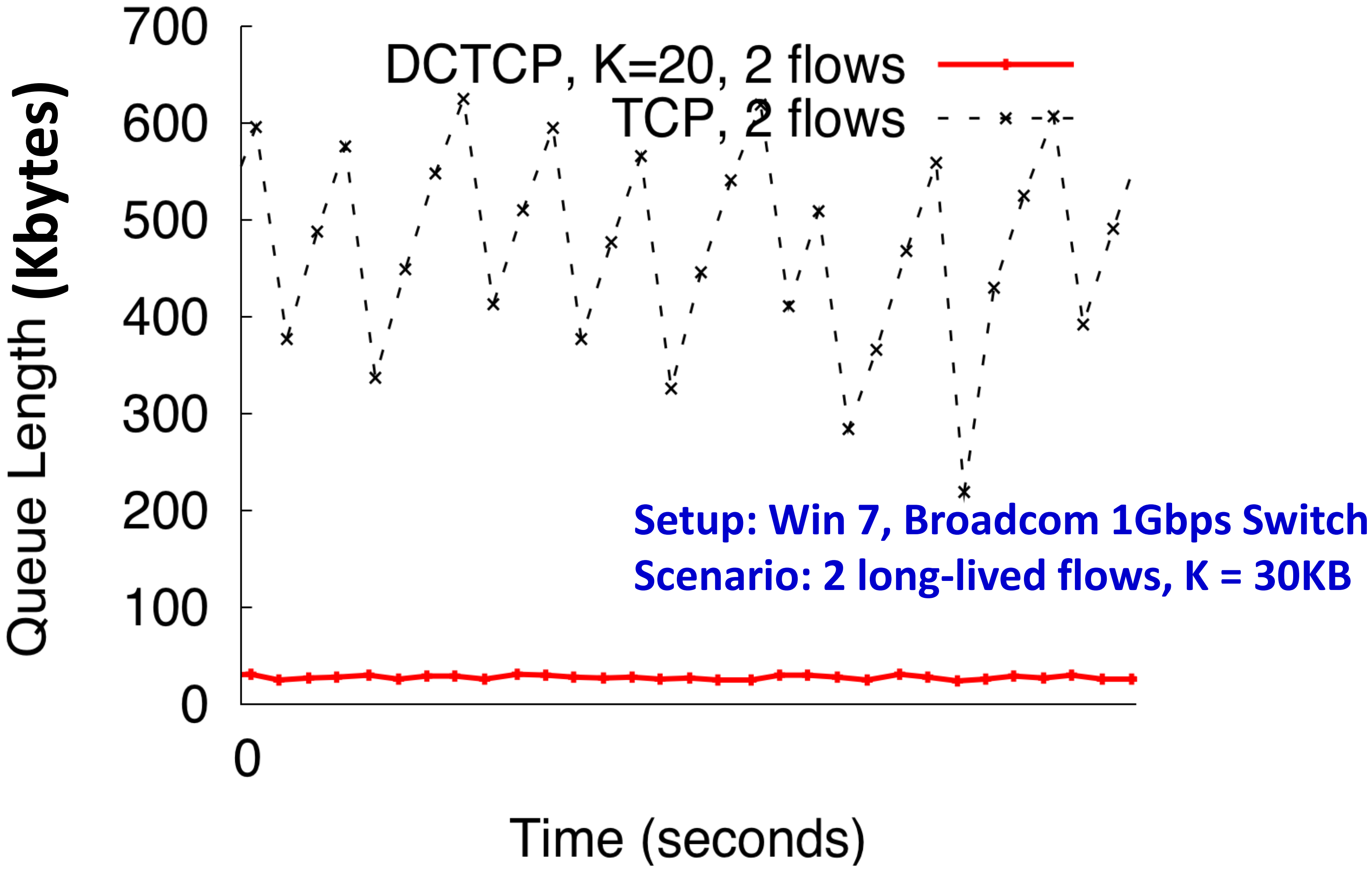
– Maintain running average of ***fraction*** of packets marked **(α)**.

In each RTT:

$$F = \frac{\#\ of\ marked\ ACKs}{Total\ \#\ of\ ACKs} \qquad \alpha \leftarrow (1-g)\alpha + gF$$

➢ **Adaptive window decreases:** $Cwnd \leftarrow (1 - \frac{\alpha}{2})Cwnd$

DCTCP in Action



RTO Calculation

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?

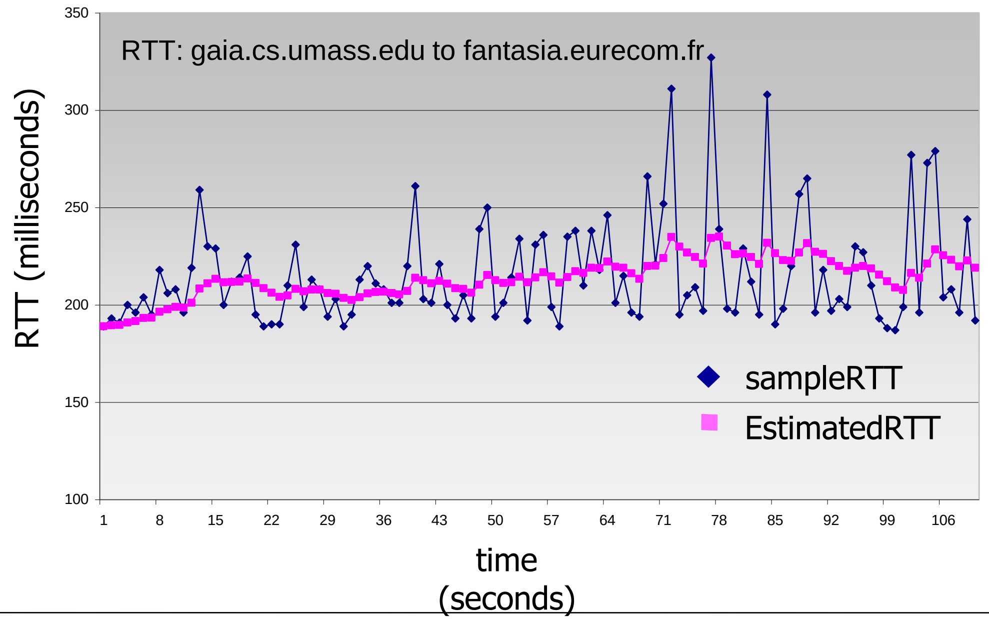
- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current `SampleRTT`

TCP estimates the round-trip time using the SampleRTT and then smoothens this estimate to calculate the EstimatedRTT. The timeout value is set longer than the EstimatedRTT to accommodate variations in RTT but not too long to avoid slow reaction to segment loss. By averaging several recent measurements, TCP ensures a reliable estimate of the round-trip time and sets an appropriate timeout value for retransmissions.

TCP round trip time, timeout

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- **`DevRTT`**: EWMA of **`SampleRTT`** deviation from **`EstimatedRTT`**:

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

(typically, β = 0.25)

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`:** want a larger safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

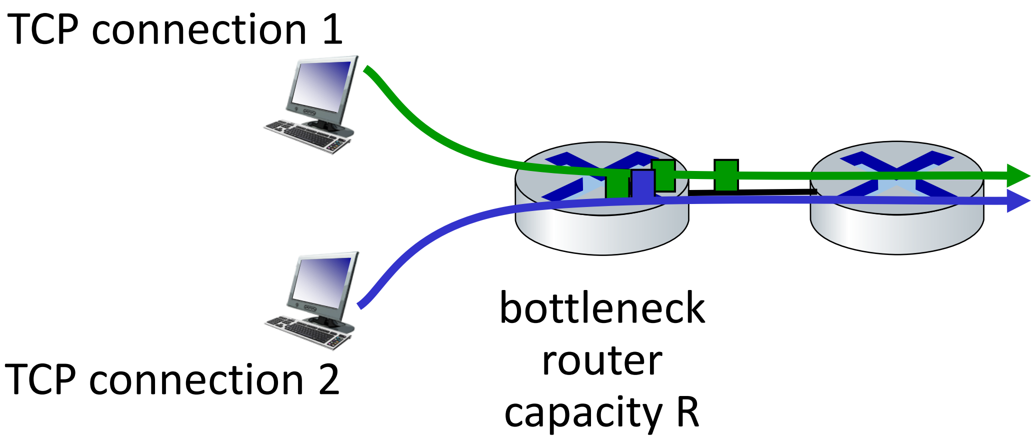
"safety margin"

More reading: http://research.protocollabs.com/captcp/doc-socket-statistic-module.html

TCP Fairness

TCP fairness

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

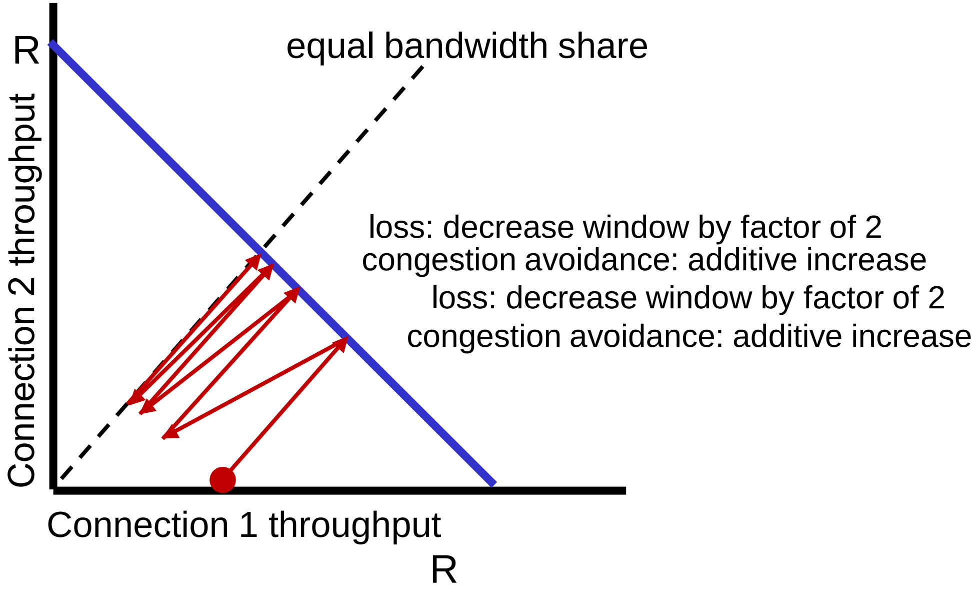


TCP fairness is achieved through a combination of congestion control algorithms, slow start mechanisms, congestion avoidance, and fair queuing mechanisms, all aimed at ensuring that every connection gets a fair share of the available network bandwidth. This helps maintain a balanced and efficient use of network resources among all TCP connections.

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?

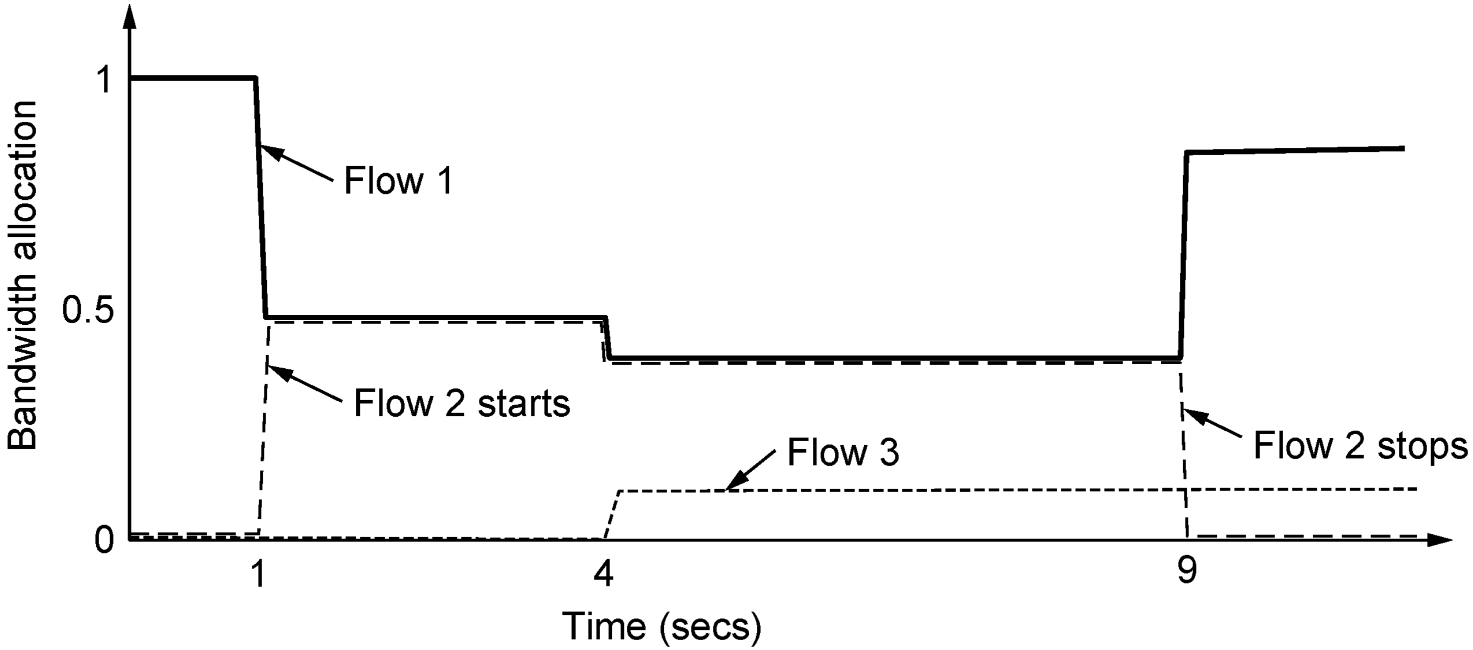
A: Yes, under idealized assumptions:

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

Is TCP Fair?

Example: two competing TCP sessions:

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



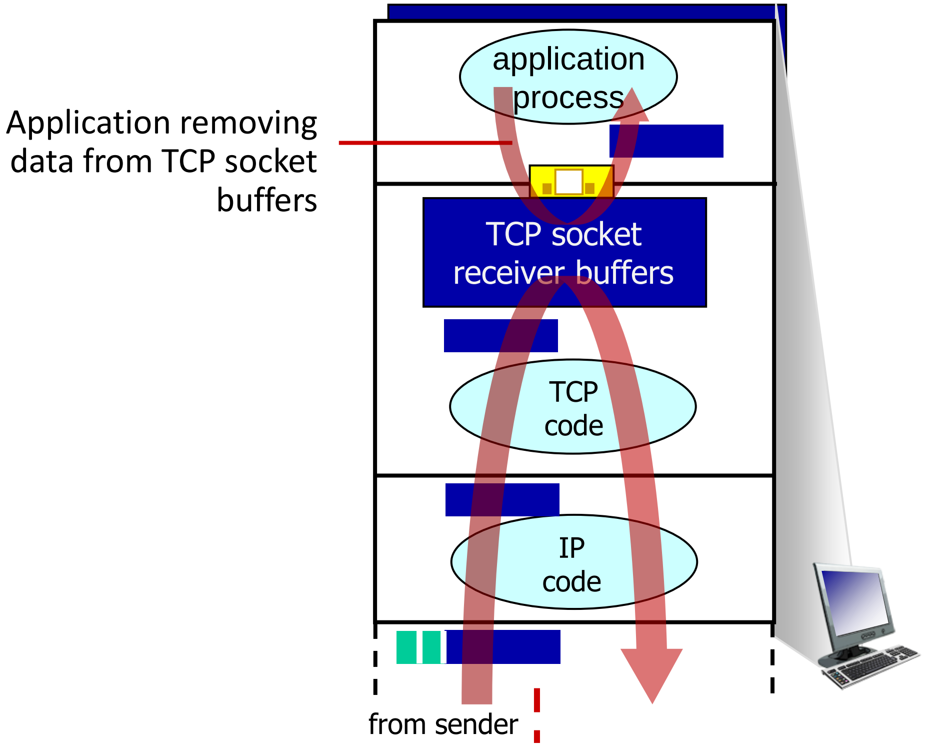
TCP Flow Control

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

flow control

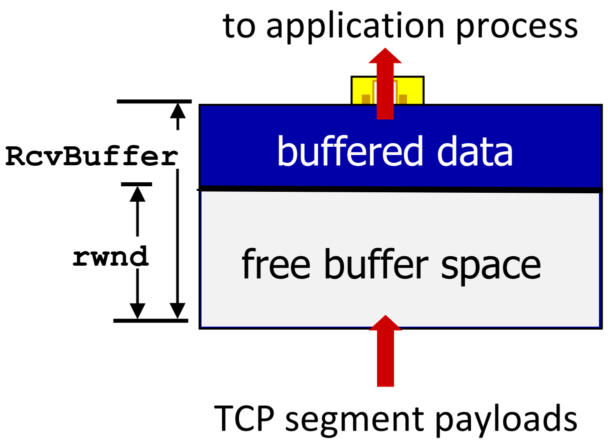
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast



receiver protocol stack

TCP flow control

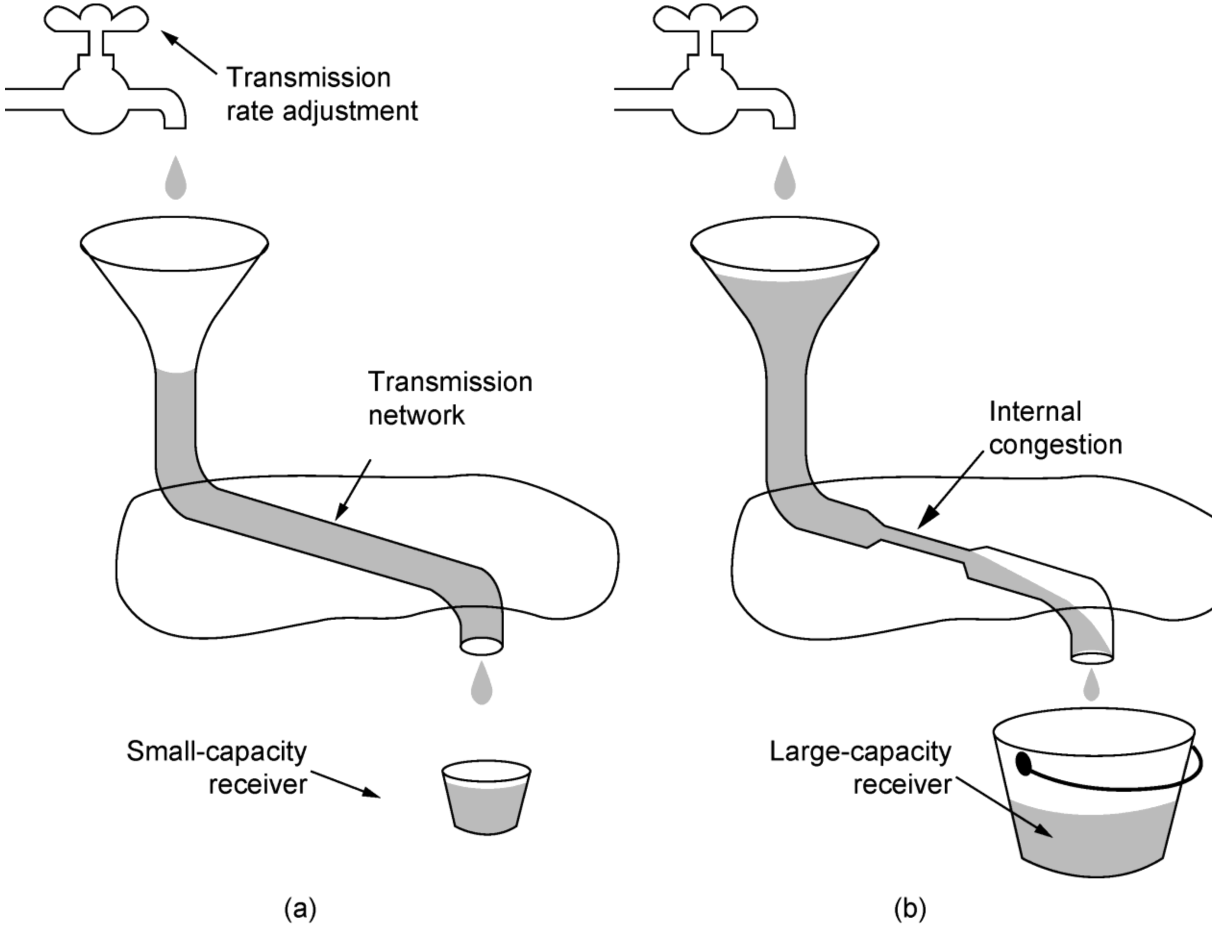
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP receiver-side buffering

TCP flow control

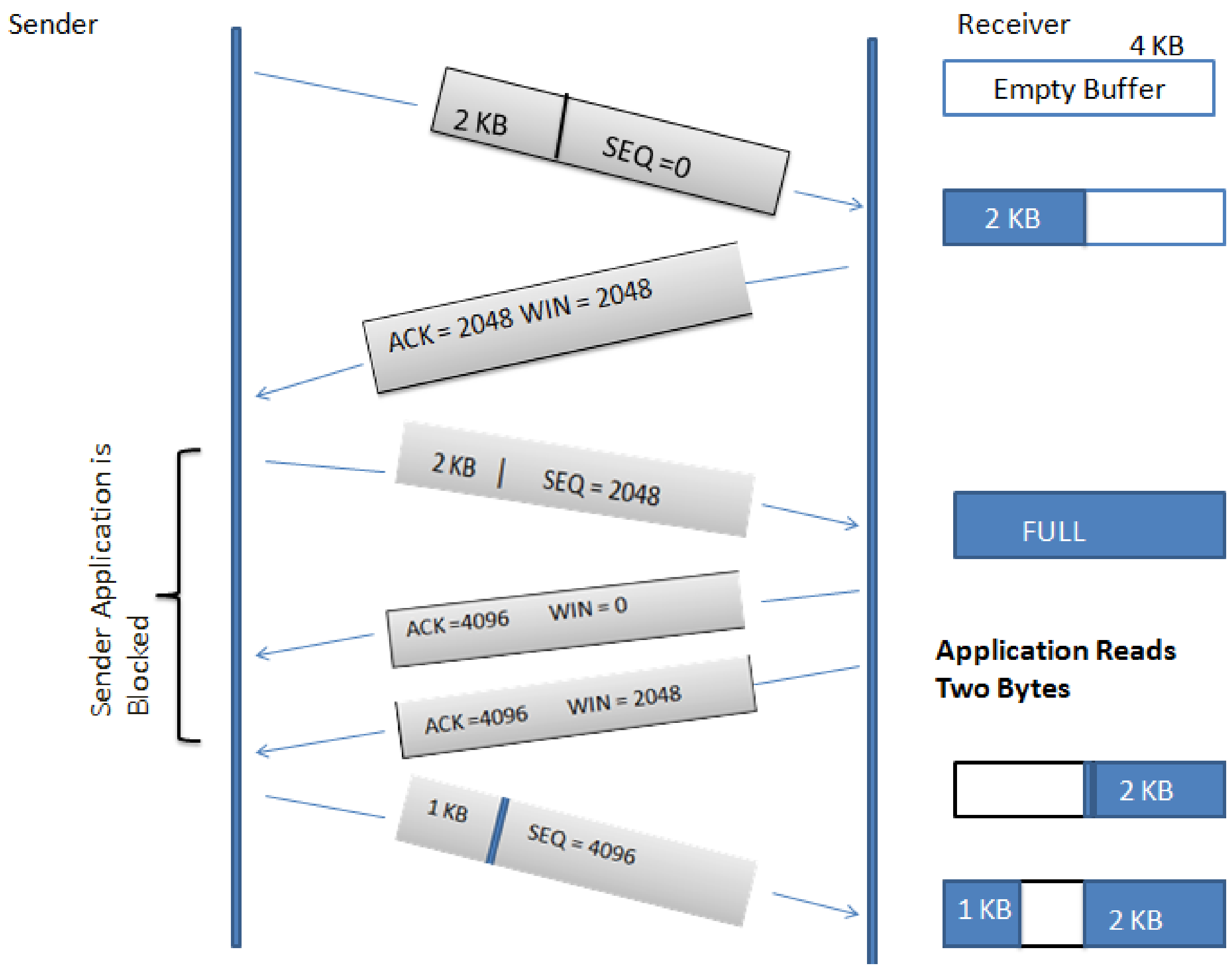
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



(a) A fast network feeding a low-capacity receiver.
(b) A slow network feeding a high-capacity receiver.

TCP flow control

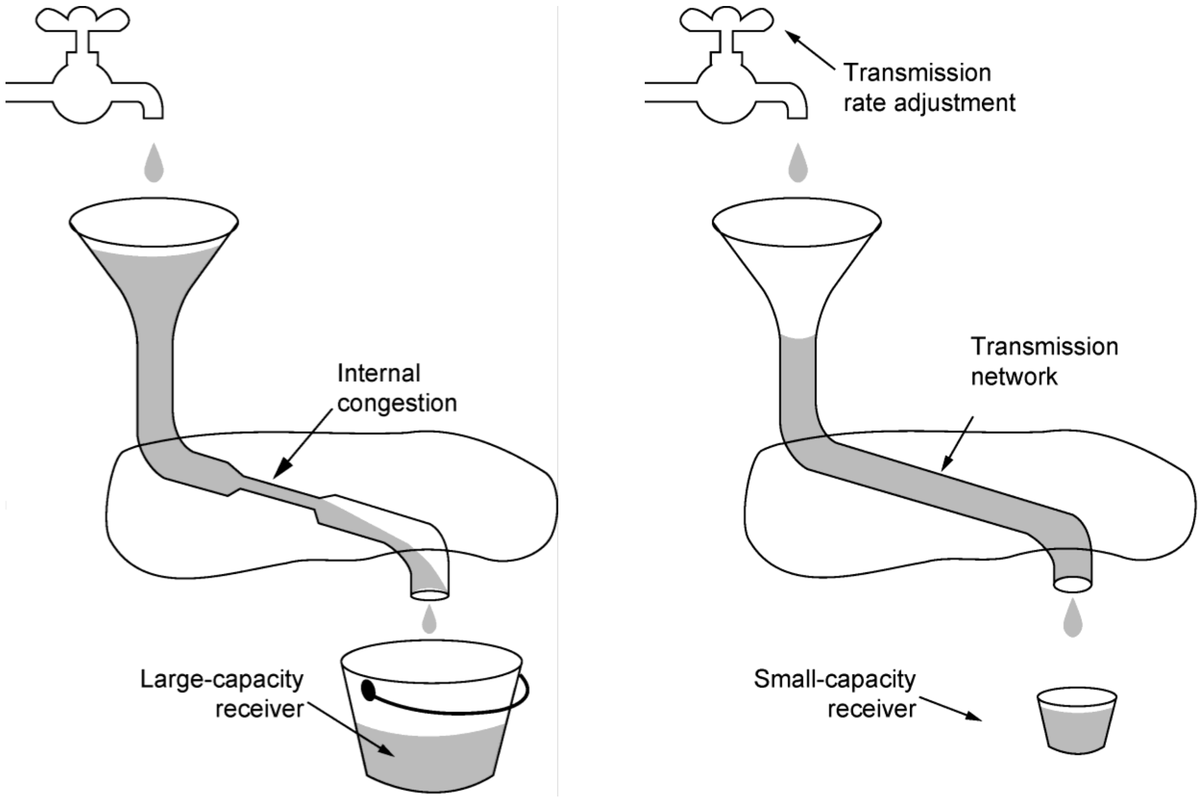
- TCP receiver "advertises" free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed ("in-flight") data to received **rwnd**
- guarantees receive buffer will not overflow



Src: https://upload.wikimedia.org/wikipedia/commons/5/57/Flow_Control_in_TCP.png

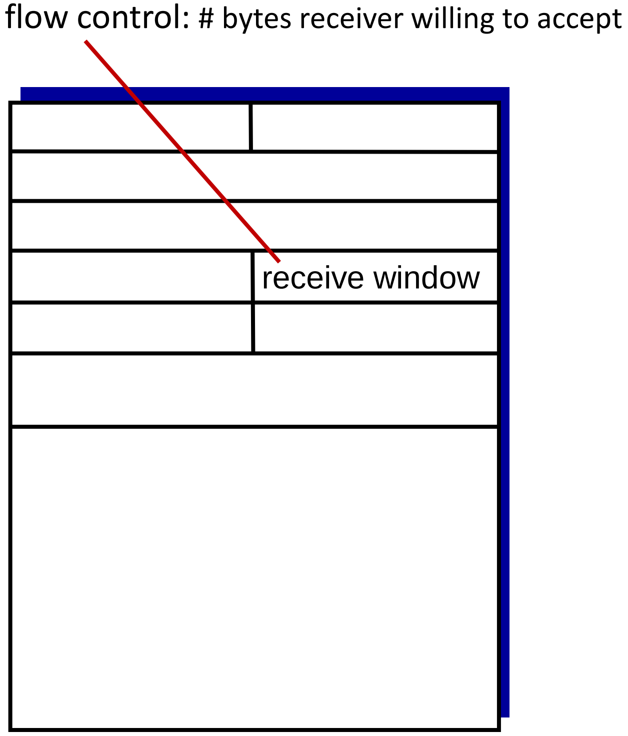
TCP Congestion Control vs Flow Control

- Congestion control
  - Sender will not overwhelm the network
- Flow control
  - Sender will not overwhelm the receiver



TCP flow control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP segment format