KHOA CÔNG NGHỆ THÔNG TIN-ĐHKHTN
CSC11004 - MẠNG MÁY TÍNH NÂNG CAO

TCP CONGESTION CONTROL

Lê Ngọc Sơn

TPHCM, 9-2024

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

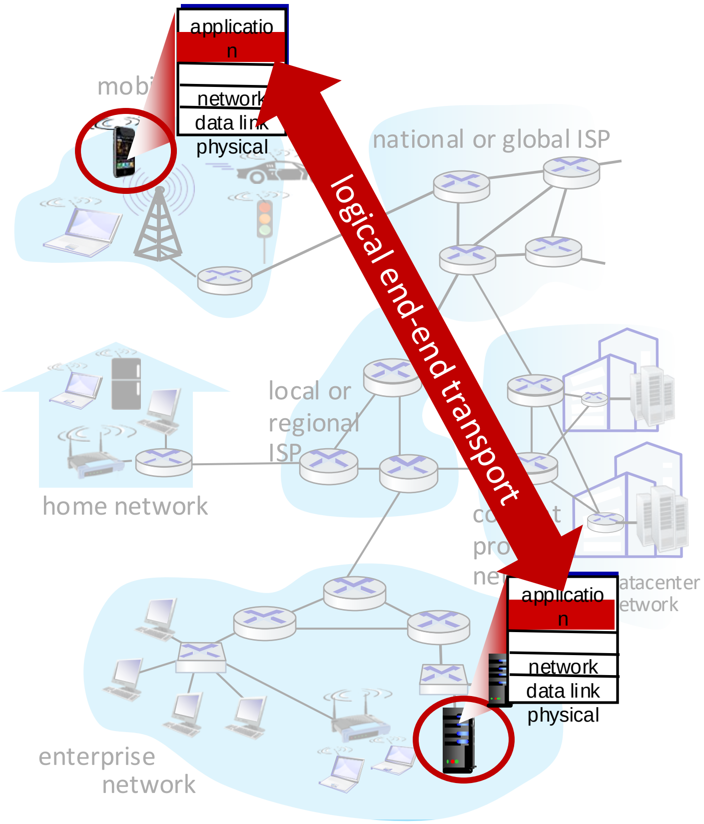
fit@hcmus

Agenda

- Transport-layer services
- UDP and TCP
- Principles of congestion control
- Congestion control approaches
- Modern Challenges and Solutions
- Case Studies
- TCP Congestion Control variants

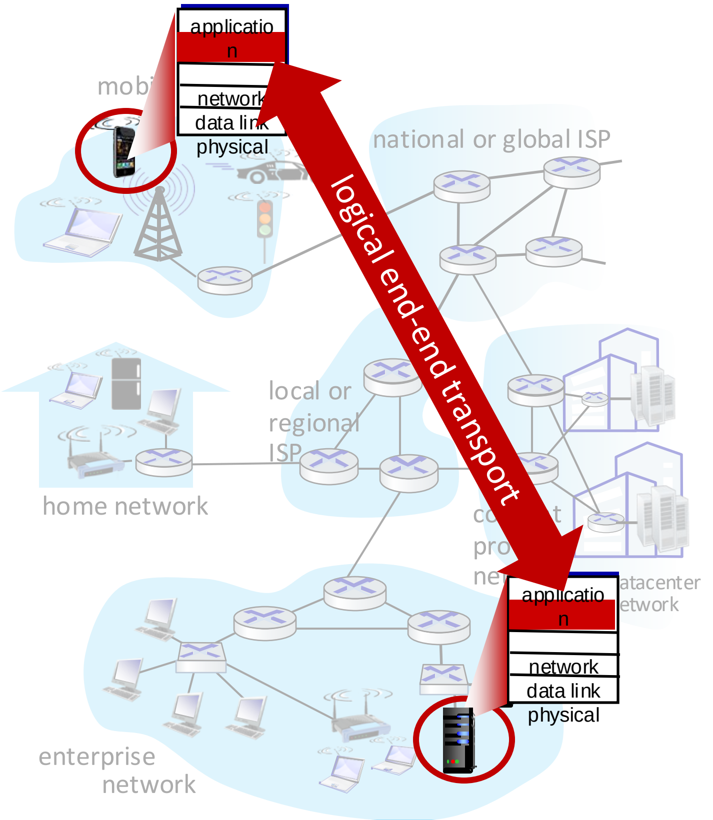
Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



Two principal Internet transport protocols

- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



UDP: User Datagram Protocol

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

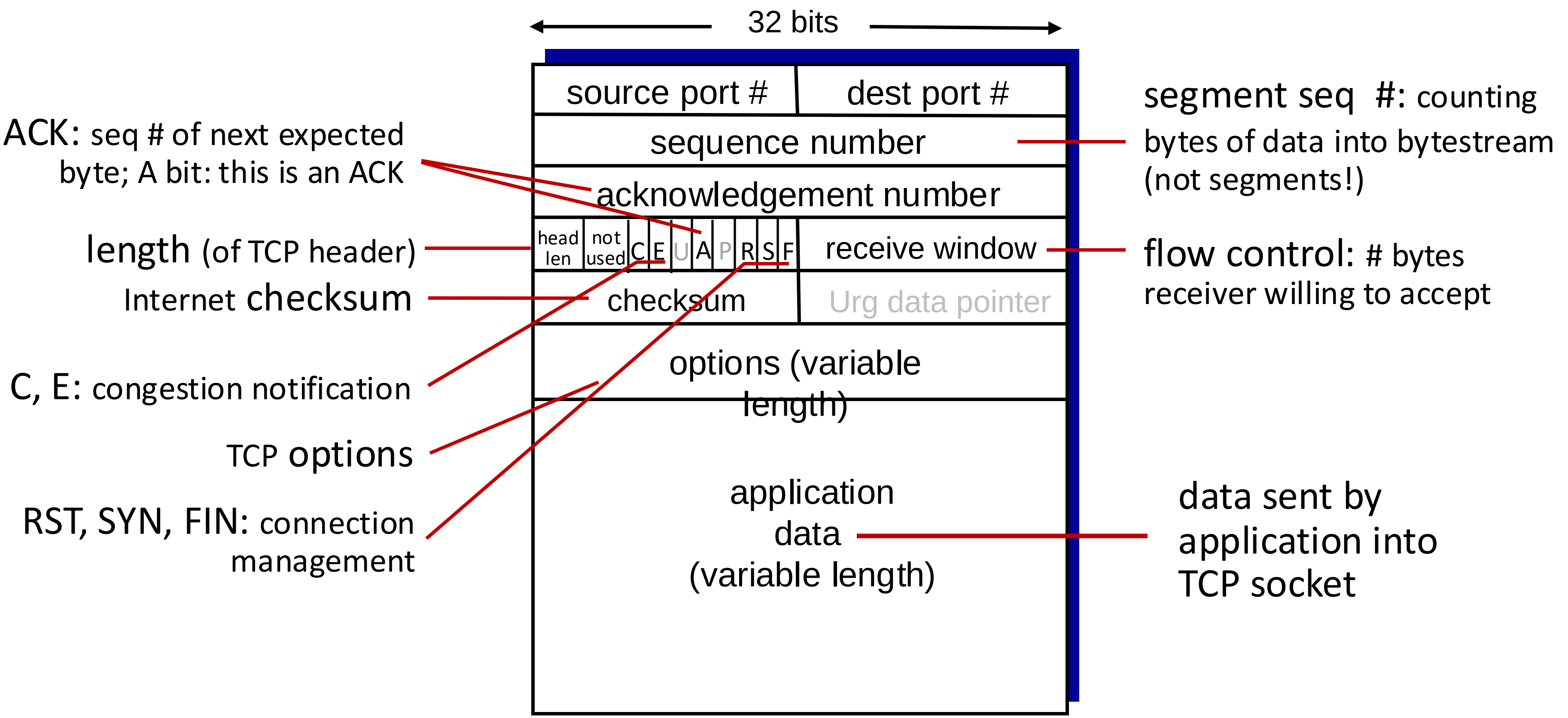
Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP sequence numbers, ACKs

Sequence numbers:

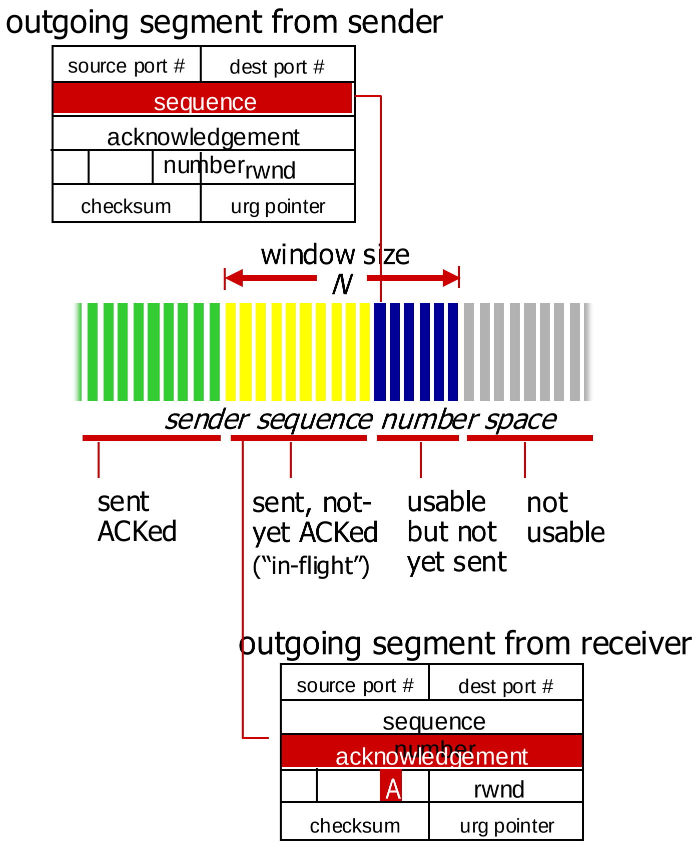
- byte stream "number" of first byte in segment's data

Acknowledgements:

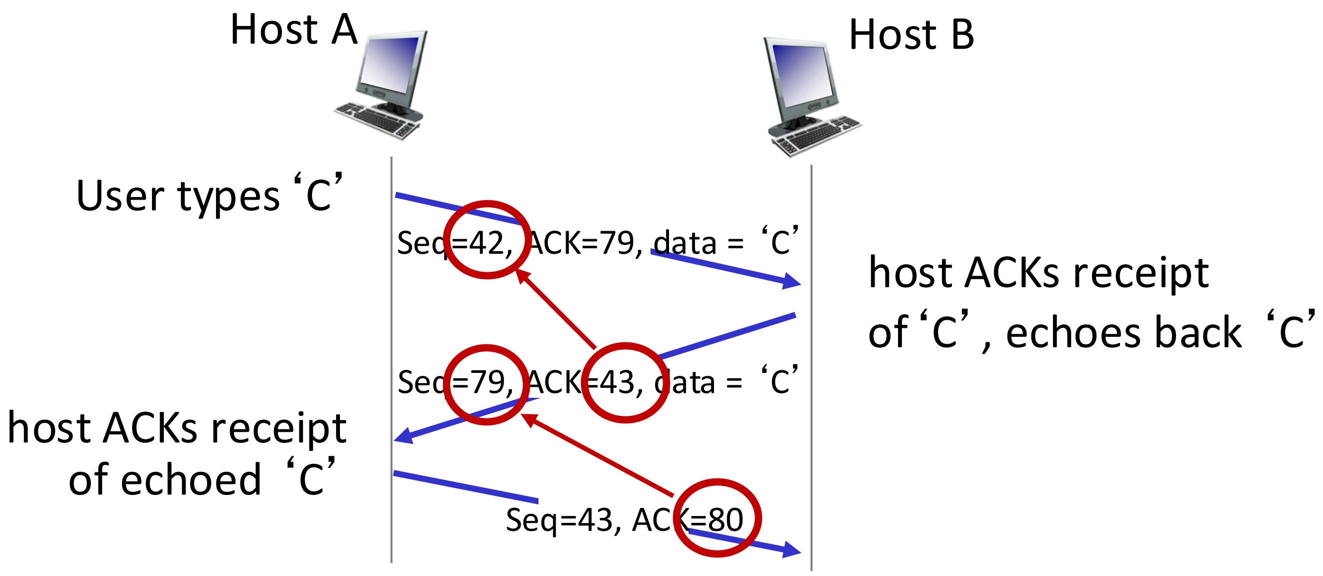
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- *A:* TCP spec doesn't say, - up to implementor



TCP sequence numbers, ACKs



simple telnet scenario

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

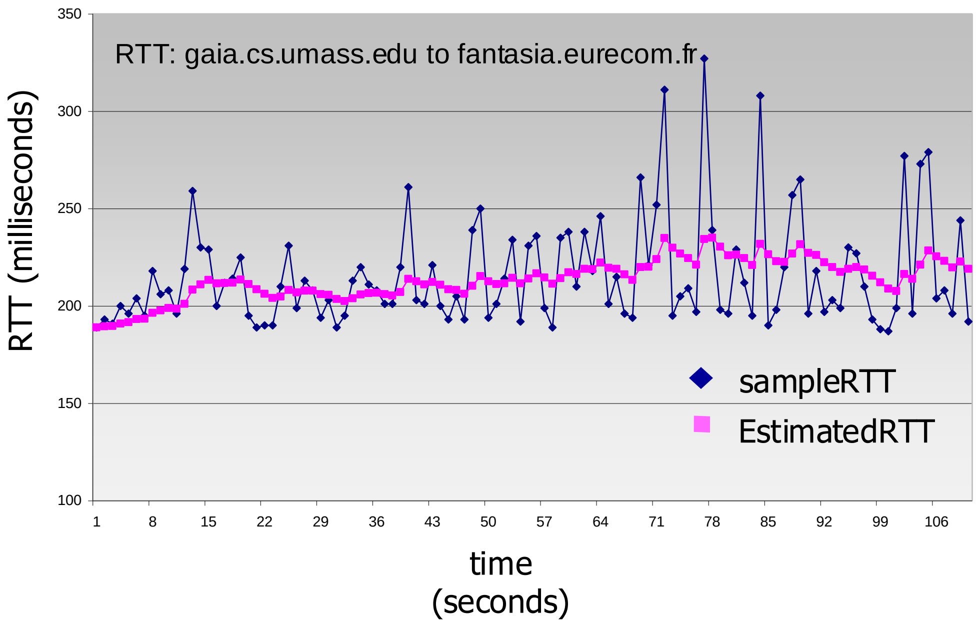
Q: how to estimate RTT?

- `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 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



fit@hcmus

TCP round trip time, timeout

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

$$\texttt{TimeoutInterval} = \underbrace{\texttt{EstimatedRTT}}_{\text{estimated RTT}} + \underbrace{\texttt{4*DevRTT}}_{\text{"safety margin"}}$$

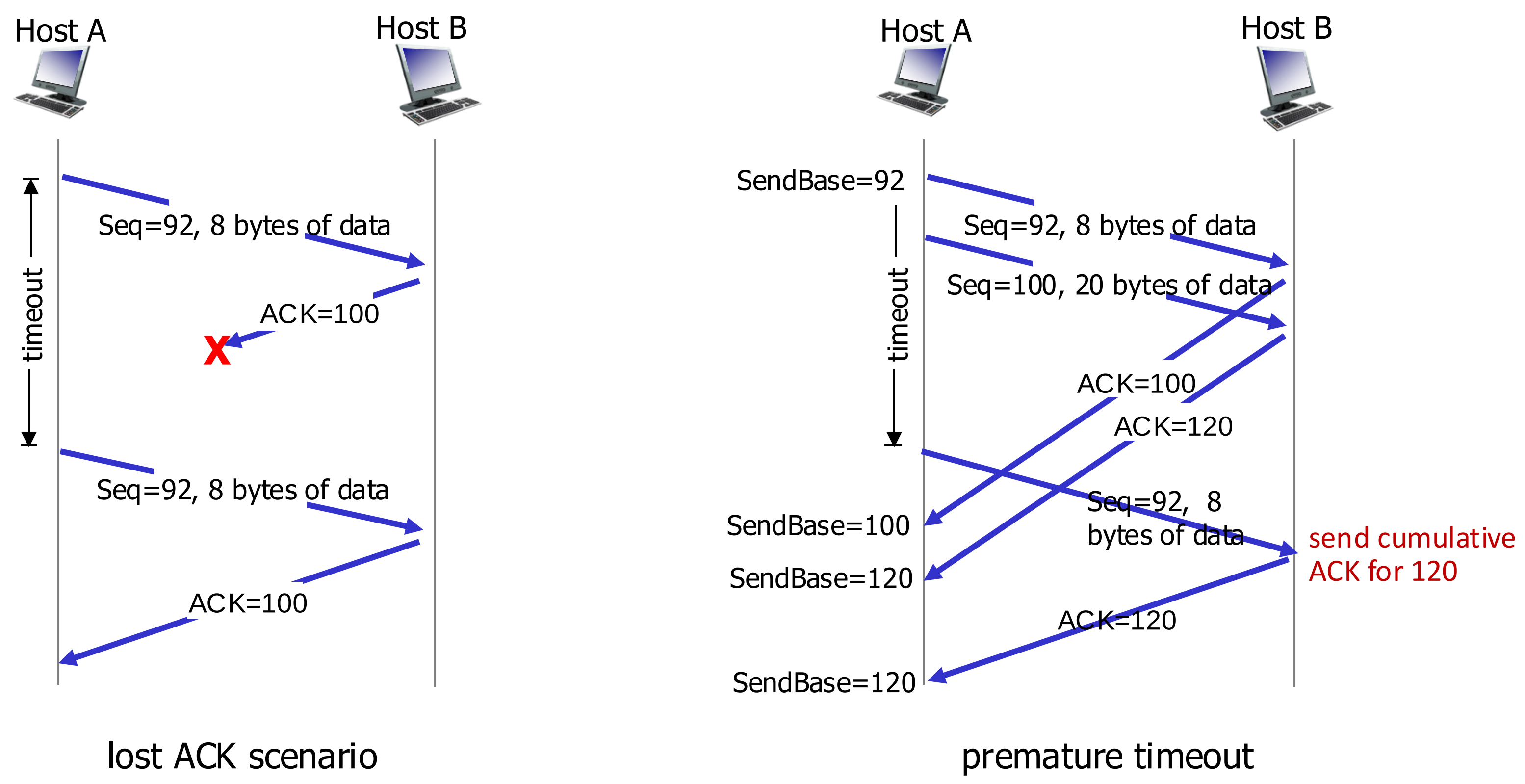
- **`DevRTT`**: EWMA of **`SampleRTT`** deviation from **`EstimatedRTT`**:

$$\texttt{DevRTT} = (1-\beta)\texttt{*DevRTT} + \beta\texttt{*|SampleRTT-EstimatedRTT|}$$

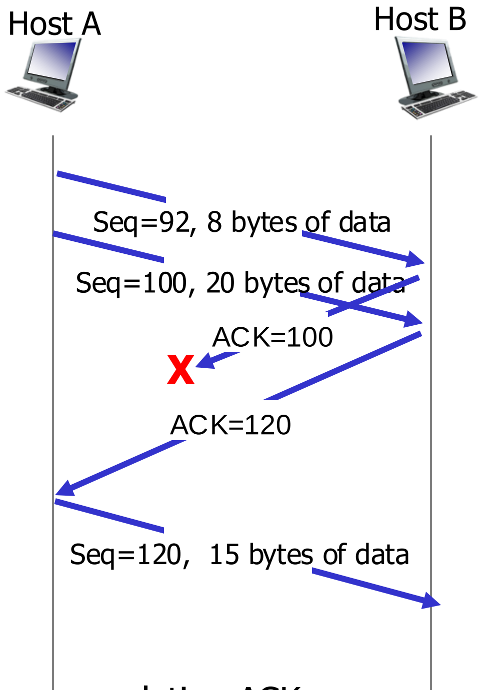
(typically, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

TCP: retransmission scenarios



TCP: retransmission scenarios



cumulative ACK covers
for earlier lost ACK

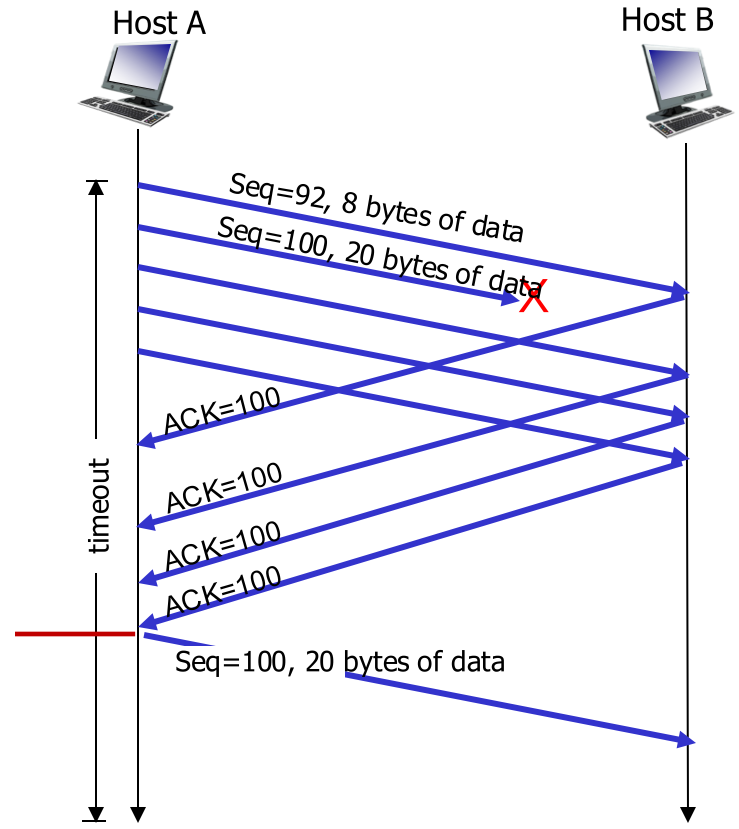
TCP fast retransmit

TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!

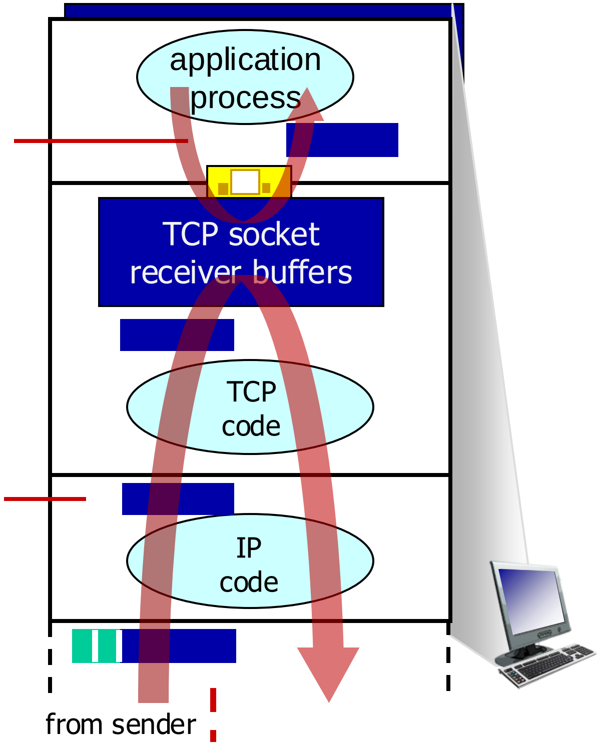


TCP flow control

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

Application removing data from TCP socket buffers

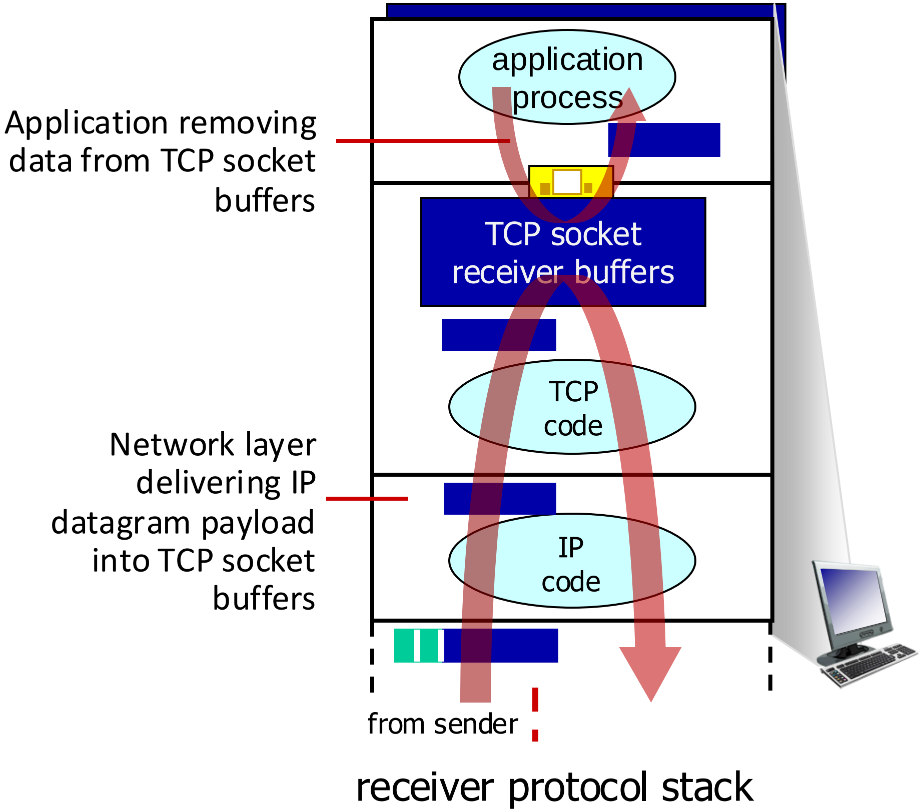
Network layer delivering IP datagram payload into TCP socket buffers



receiver protocol stack

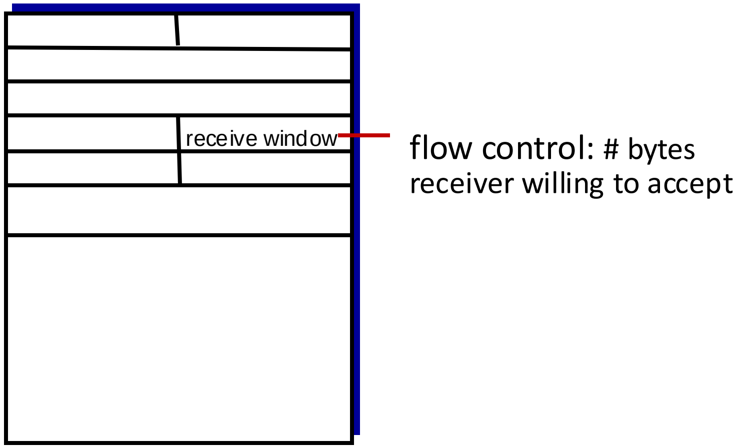
TCP flow control

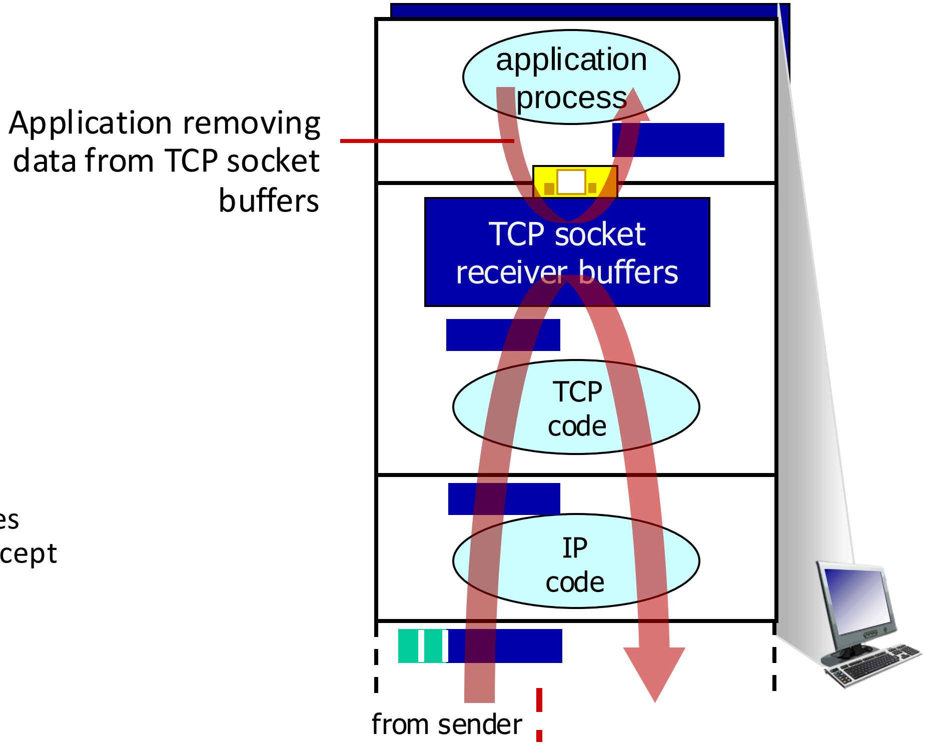
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



TCP flow control

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



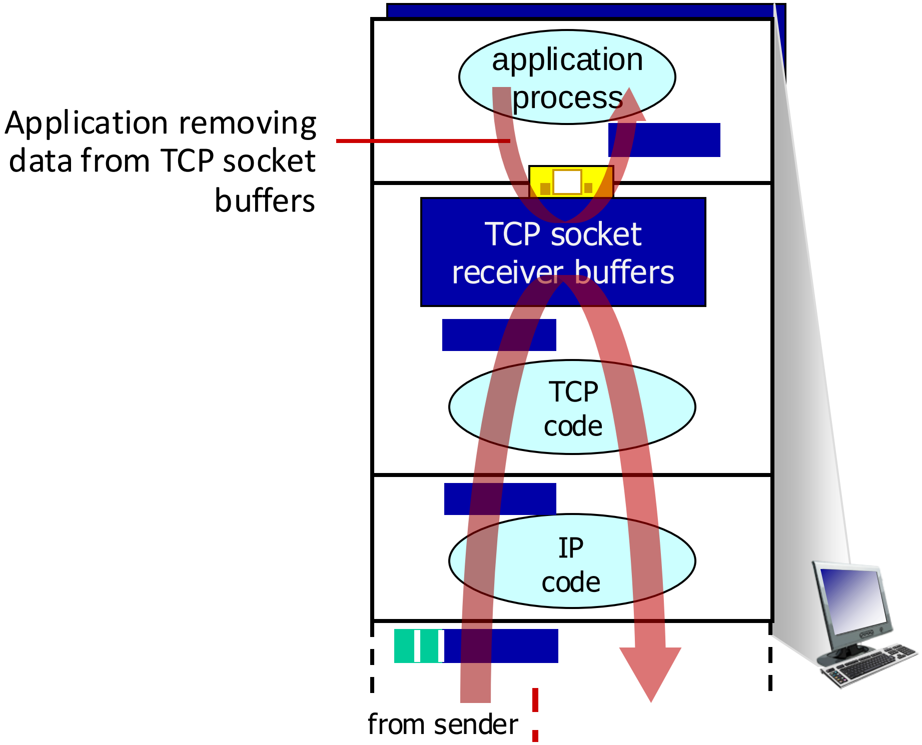


receiver protocol stack

fit@hcmus

TCP flow control

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



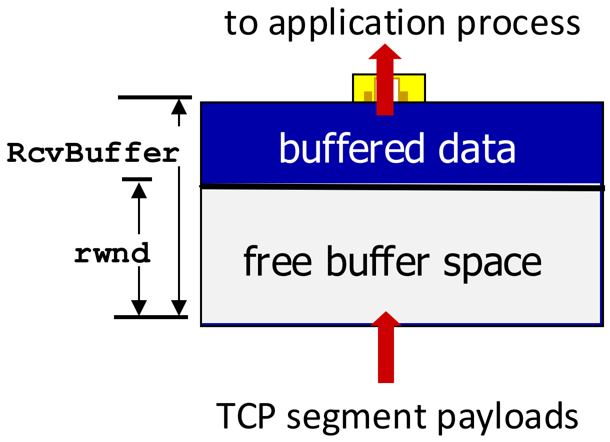
receiver protocol stack

flow control

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

TCP flow control

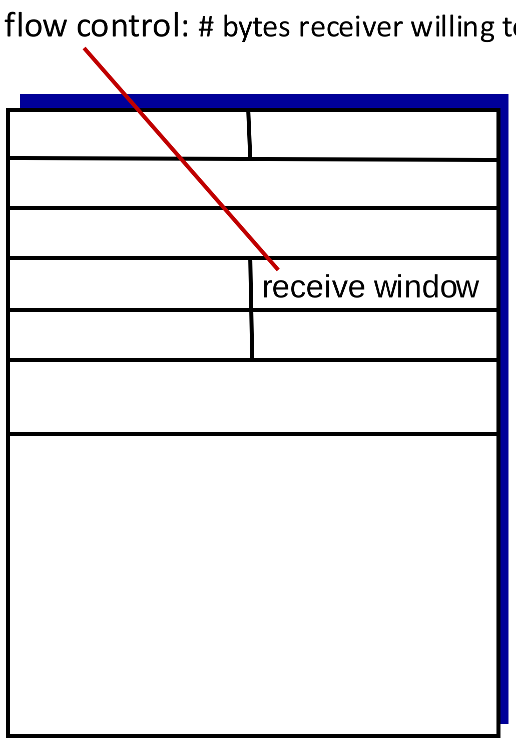
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees 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



TCP segment format

Principles of congestion control

Congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!

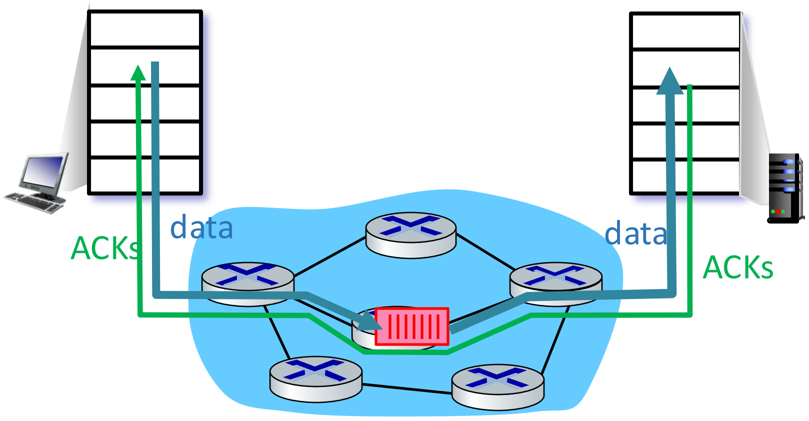
congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver

Approaches towards congestion control

End-end congestion control:

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

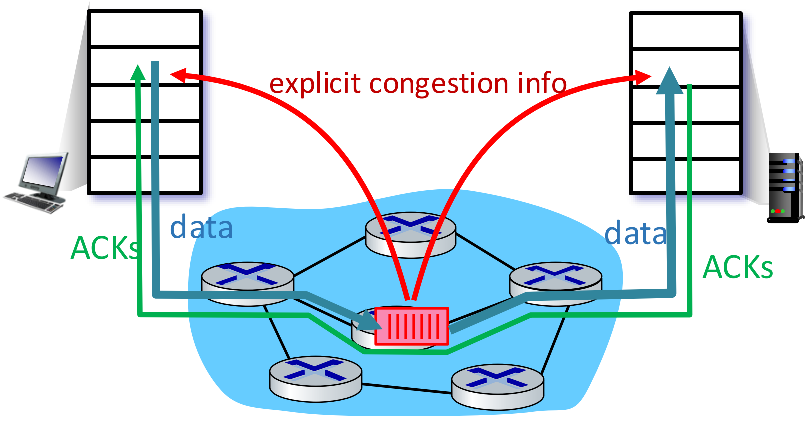


Approaches towards congestion control

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 End-End Congestion control - Approaches

Loss-based algorithms:
TCP Tahoe, Reno,
Cubic

Delay-based algorithms:
TCP Vegas

Loss-based Congestion Control

- Congestion control mechanisms that adjust the transmission rate based on packet loss.
- Detect network congestion when packets are dropped and respond by reducing the sending rate.

Key Principles

- **Packet Loss as Congestion Signal**: Algorithms assume packet loss indicates network congestion.
- **Multiplicative Decrease**: Upon detecting loss, the sender reduces the congestion window, often halving the sending rate.
- **Additive Increase**: The sender gradually increases the congestion window (sending rate) once packet loss ceases, probing for available bandwidth.

TCP congestion control: AIMD

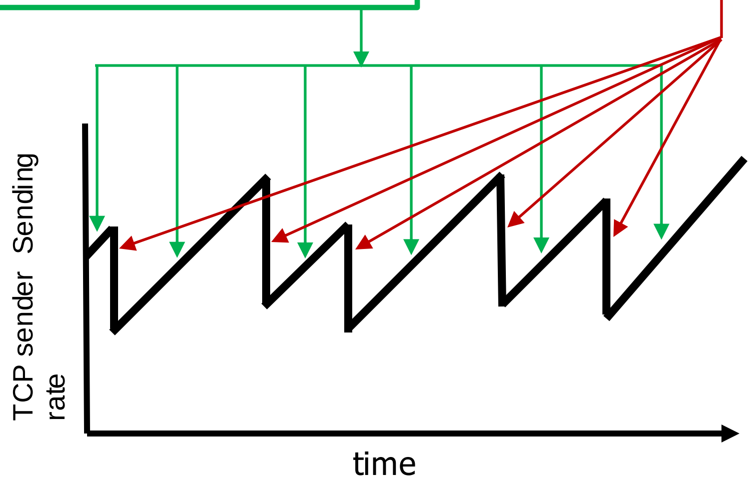
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

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 AIMD: more

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)

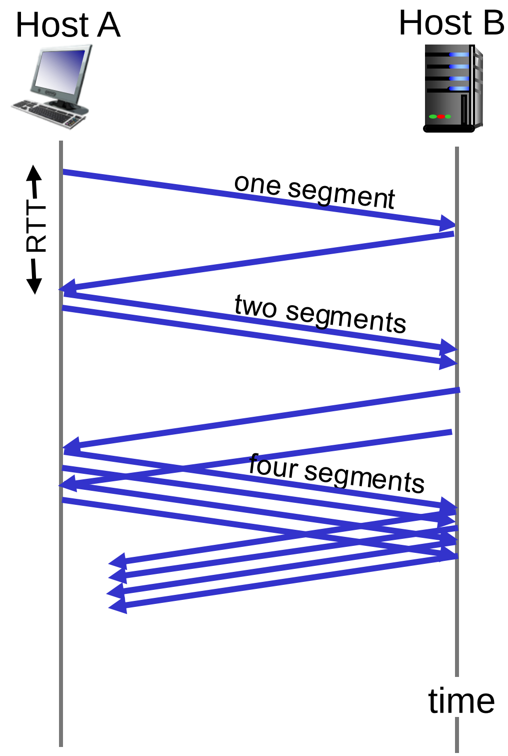
Why AIMD?

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

fit@hcmus

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



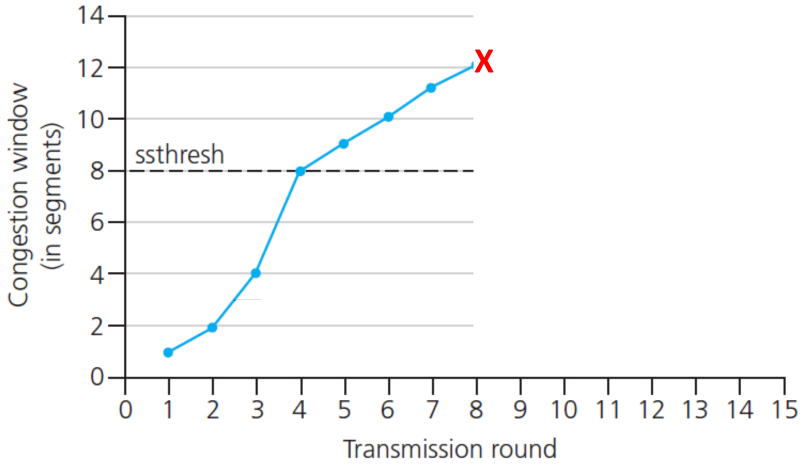
TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.

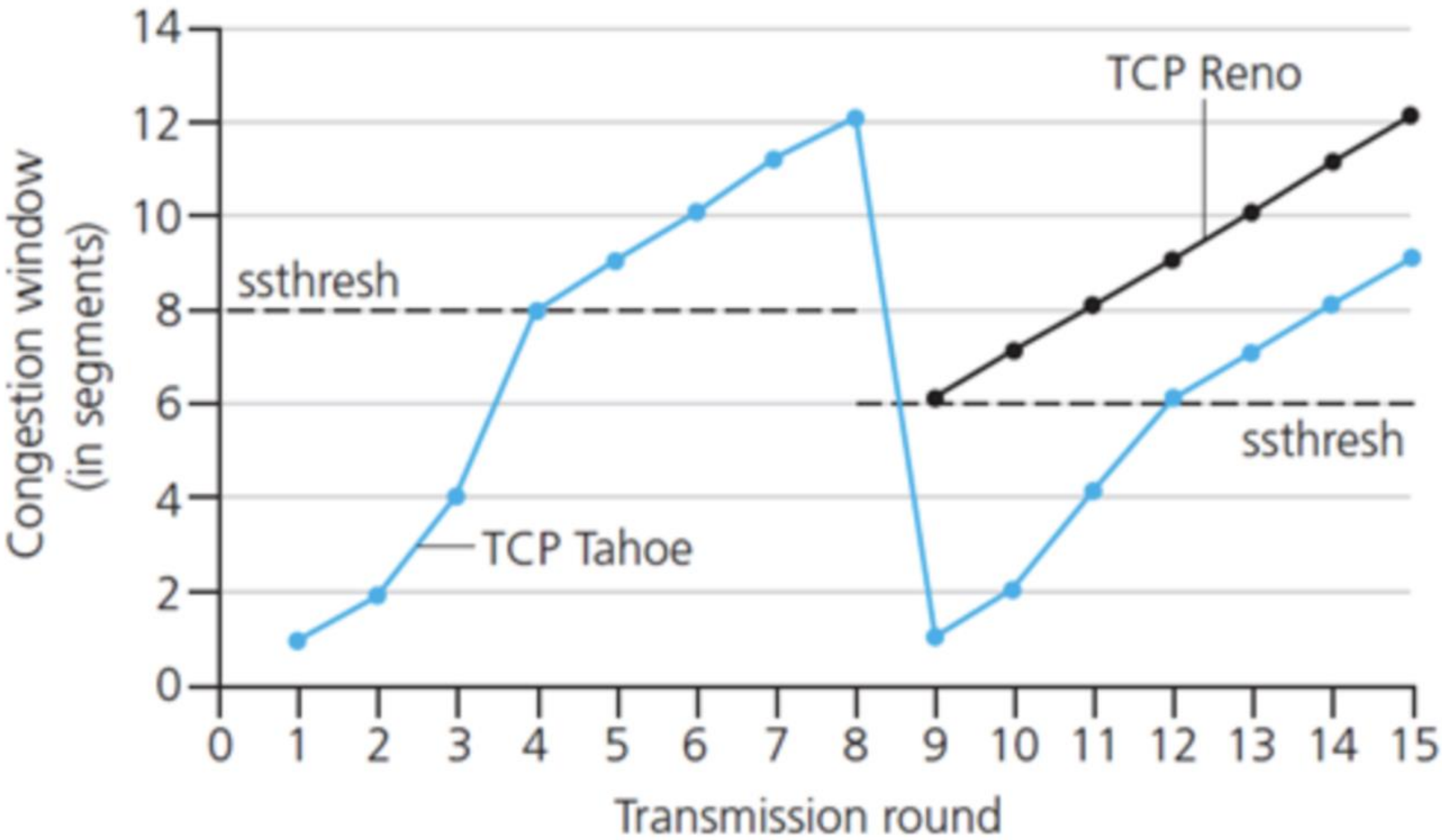
Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event



fit@hcmus

TCP Tahoe and TCP Reno



Loss-based Algorithm Advantages

- **Simplicity**:

  Packet loss is an easy-to-detect congestion signal.

- **Widespread Adoption**:

  Standardized and used in the majority of TCP implementations

- **Effectiveness in Typical Networks**:

  Works well in networks with regular bandwidth and latency characteristics.

Modern Challenges in Congestion Control

As network speeds increase, more devices become connected, and the nature of data traffic changes, traditional congestion control algorithms are increasingly being tested.

- **Bufferbloat**: Excessive buffering leads to increased latency.
- **Fairness**: Different TCP flows compete for bandwidth, causing fairness issues.
- **Long Fat Networks**: In high-speed networks, traditional TCP struggles with throughput.

Bufferbloat

What?

Bufferbloat occurs when large buffers in routers or network devices become full, causing significant delays in packet transmission. This typically happens because older congestion control algorithms fill up network buffers before reacting to congestion signals like packet loss.

Why ?

Large buffers lead to **increased latency**, which can severely impact real-time applications such as video streaming, VoIP, and online gaming. This causes jittery audio or video, lag in online gaming, and general performance degradation.

Traditional loss-based algorithms like **TCP Reno** are slow to react, only reducing sending rates after packet loss occurs, which may happen too late to prevent large latency spikes caused by bufferbloat.

Fairness

- What?

Fairness refers to how bandwidth is shared between different connections in a network. Ideally, all flows should have a fair share of network resources. However, this is not always the case.

- Why?

In scenarios where multiple congestion control algorithms (e.g., **TCP Reno** vs. **TCP BBR**) are competing, newer algorithms like **BBR**, which aggressively optimize throughput, can dominate bandwidth. This results in **unfair distribution**, where flows using older or more conservative algorithms get less bandwidth.

Loss-based algorithms favor connections with lower round-trip times (RTT), causing **unfair competition** between connections with different RTTs. In such cases, flows with longer RTTs receive less bandwidth.

High-Speed, Long Distance Networks (Long Fat Networks LFNs)

LFNs refer to networks with **high bandwidth and long latency**, such as intercontinental fiber-optic links or satellite networks. These networks have a large **bandwidth-delay product** (BDP), meaning they can carry many packets simultaneously due to their high capacity and long delay.

Traditional loss-based algorithms like **TCP Reno** and **TCP Tahoe** perform poorly in LFNs because of their slow **congestion window growth** and **aggressive reductions** after packet loss. As a result, they fail to fully utilize the available bandwidth.

Congestion control algorithms need to be more responsive in LFNs to avoid underutilization and maintain high throughput over long distances.

Solutions

Delay-Based Algorithms (e.g., TCP Vegas):

- Monitor round-trip time (RTT) to detect congestion before packet loss.
- Advantage: Can prevent bufferbloat and detect congestion earlier.

Hybrid Approaches (e.g., TCP Compound):

- Use both delay, loss and other information to manage congestion more effectively.
- Balances throughput, latency, and fairness in modern networks.

fit@hcmus

Delay-based Congestion Control

- Congestion control mechanisms that rely on **round-trip time (RTT)** or **queueing delay** as indicators of congestion.
- These algorithms adjust the transmission rate based on increasing delays before packet loss occurs.

cdio 4.0
fit@hcmus

Key Principles

- **RTT as Congestion Signal**: Instead of waiting for packet loss, delay-based algorithms monitor changes in the round-trip time (RTT) or packet queueing delays.
- **Avoiding Congestion**: When RTT increases (indicating congestion), the sender reduces the transmission rate to prevent packet loss.
- **Congestion Window Adjustment**:
  - If RTT is near its minimum (no congestion), the sender increases the sending rate.
  - If RTT is increasing (indicating congestion), the sender decreases the sending rate.

Examples of Delay-based Algorithms

- **TCP Vegas**:
  - Measures RTT to detect early signs of congestion.
  - Adjusts the congestion window to keep the network "just full enough."
- **TCP FAST**:
  - An enhancement of TCP Vegas, designed for high-speed networks.
  - Provides higher throughput and lower latency by fine-tuning RTT-based control.

fit@hcmus

Delay-based Algorithms Pros & Cons

Early Congestion Detection: Detect congestion before packet loss occurs, making it possible to avoid packet loss altogether.

Lower Latency: Better suited for real-time applications like video streaming and online gaming since delay-based algorithms prevent bufferbloat.

Increased Efficiency: Utilizes network resources more efficiently by minimizing unnecessary retransmissions.

Sensitivity to RTT Fluctuations: Delay-based algorithms may misinterpret natural RTT fluctuations as congestion, leading to unnecessary reductions in the sending rate.

Requires Accurate RTT Measurement: Effectiveness depends on the accuracy and consistency of RTT measurements, which can be difficult in some networks.

Unfairness in Mixed Environments: When competing with loss-based algorithms (like TCP Reno), delay-based algorithms may reduce their sending rate too much, resulting in unfair bandwidth sharing.

cdio 4.0

fit@hcmus

Hybrid Approach Congestion Control

Loss-Based + Delay-Based

combines traditional **loss-based congestion control** with **delay-based control**, using both **packet loss** and **RTT (Round-Trip Time)** measurements to optimize the sending rate.

Loss-Based + Delay-Based Hybrid Algorithms

- What ?

Combine both **delay-based** and **loss-based** congestion control mechanisms.

Use delay (RTT) to detect congestion early and packet loss to handle severe congestion.

- Why ?

Aim to maintain high throughput like loss-based algorithms (e.g., TCP Reno) while minimizing latency, similar to delay-based algorithms (e.g., TCP Vegas).

Balance between efficiency and low latency, ideal for high-speed networks.

TCP Compound

- Combines both delay and loss signals to optimize performance in Windows
- It uses a loss-based congestion window to handle traditional congestion control and a delay-based window to fine-tune the sending rate based on RTT measurements.

Other approaches - Model-based Congestion Control

A **model-based approach** in congestion control refers to using an explicit mathematical model of the network's behavior to guide decision-making, rather than relying on traditional reactive signals like packet loss or delay increases. The model attempts to predict or estimate key characteristics of the network, such as available bandwidth and round-trip time (RTT), and uses these estimates to control the rate at which data is sent. This proactive strategy contrasts with traditional congestion control mechanisms, which react to events like packet loss after they occur.

Why hybrid approaches are important ?

In modern networks, relying solely on one signal—whether it's packet loss, delay, or bandwidth—is often insufficient to optimize performance. Networks today are much more diverse, including:

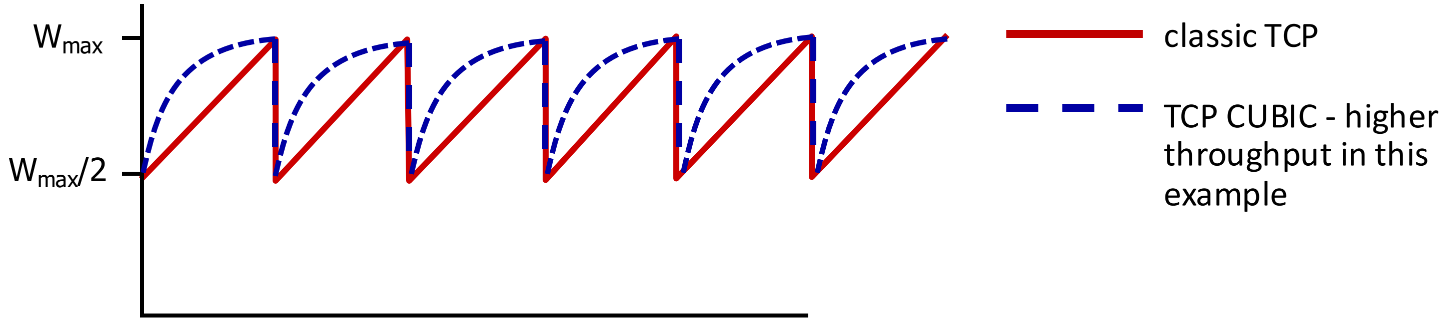
- **High-speed links** (e.g., fiber optics, 5G)
- **Mobile networks** (with varying latency and throughput)
- **Low-latency, high-demand applications** (like gaming, video streaming, and cloud services)

Thus, hybrid approaches offer better flexibility and responsiveness to different network conditions.

Case study: TCP CUBIC

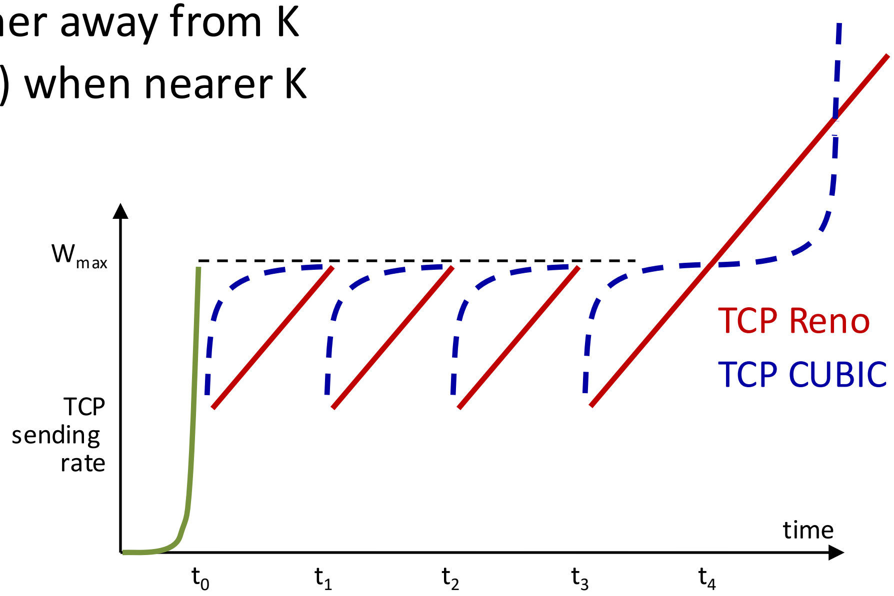
- Is there a better way than AIMD to “probe” for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn’t changed much

  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



Case Study: TCP CUBIC

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



cdio 4.0
fit@hcmus

Case Study: Google's TCP BBR

What ?

A modern congestion control algorithm developed by Google.

Stands for **Bottleneck Bandwidth and RTT**.

Why?

Traditional loss-based algorithms (e.g., TCP Reno, TCP Cubic) struggle in high-bandwidth, high-latency environments.

BBR was designed to improve throughput and reduce latency by estimating bottleneck bandwidth and minimum RTT instead of relying on packet loss.

fit@hcmus

How does TCP BBR Work ?

Key Innovation

- BBR doesn’t rely on packet loss as a congestion signal.
- It uses **real-time measurements** of available bandwidth and the **minimum observed RTT**.

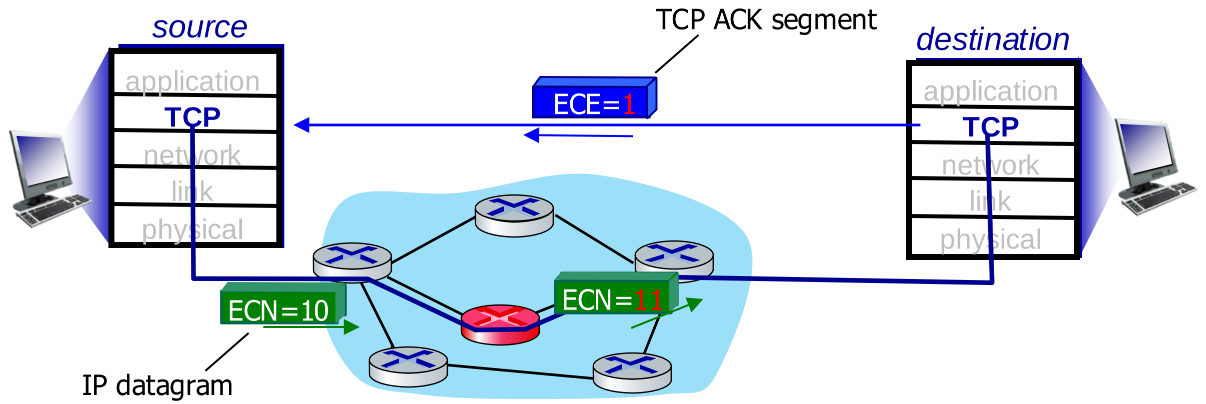
Bandwith Probing:

- TCP BBR continuously probes the network to measure the bottleneck bandwidth and adjusts the sending rate accordingly.
- BBR estimates the **minimum RTT** to determine the lowest latency that the network can achieve.

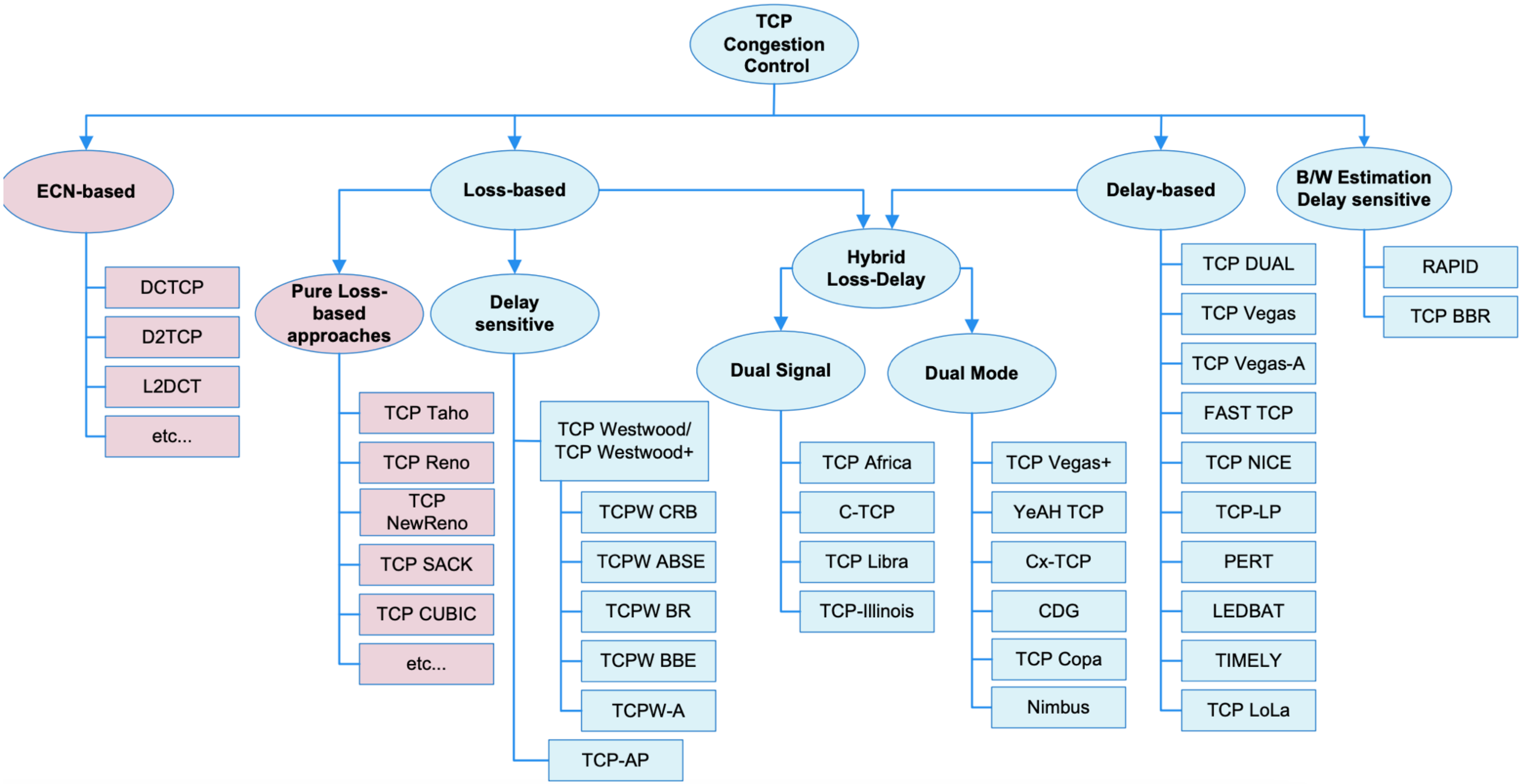
TCP **Network-Assisted** Congestion Control Approach

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)



TCP Congestion Control Variants



Other variants

Variant	Feedback	Required changes	Benefits
(New) Reno	Loss	—	—
Vegas	Delay	Sender	Less loss
High Speed	Loss	Sender	High bandwidth
BIC	Loss	Sender	High bandwidth
CUBIC	Loss	Sender	High bandwidth
C2TCP[9][10]	Loss/Delay	Sender	Ultra-low latency and high bandwidth
NATCP[11]	Multi-bit signal	Sender	Near Optimal Performance
Elastic-TCP	Loss/Delay	Sender	High bandwidth/short & long-distance
Agile-TCP	Loss	Sender	High bandwidth/short-distance
H-TCP	Loss	Sender	High bandwidth
FAST	Delay	Sender	High bandwidth
Compound TCP	Loss/Delay	Sender	High bandwidth
Westwood	Loss/Delay	Sender	L
Jersey	Loss/Delay	Sender	L
BBR[12]	Delay	Sender	BLVC, Bufferbloat
CLAMP	Multi-bit signal	Receiver, Router	V
TFRC	Loss	Sender, Receiver	No Retransmission
XCP	Multi-bit signal	Sender, Receiver, Router	BLFC
VCP	2-bit signal	Sender, Receiver, Router	BLF
MaxNet	Multi-bit signal	Sender, Receiver, Router	BLFSC
JetMax	Multi-bit signal	Sender, Receiver, Router	High bandwidth
RED	Loss	Router	Reduced delay
ECN	Single-bit signal	Sender, Receiver, Router	Reduced loss

TCP Congestion Control in OSs

- Linux:

Default: **TCP Cubic**

Available Options: TCP Reno, TCP NewReno, TCP Vegas, TCP BBR

- macOS, iOS

Default: **TCP Cubic** (since macOS 10.13)

Available Options: TCP Reno, TCP NewReno

- Windows:

Default: **TCP Compound**

Available Options: TCP Reno, TCP NewReno

- Android:

Default: **TCP Cubic** (inherits from Linux kernel)

Available Options: TCP Reno, New Reno, BBR, Vegas

Question?