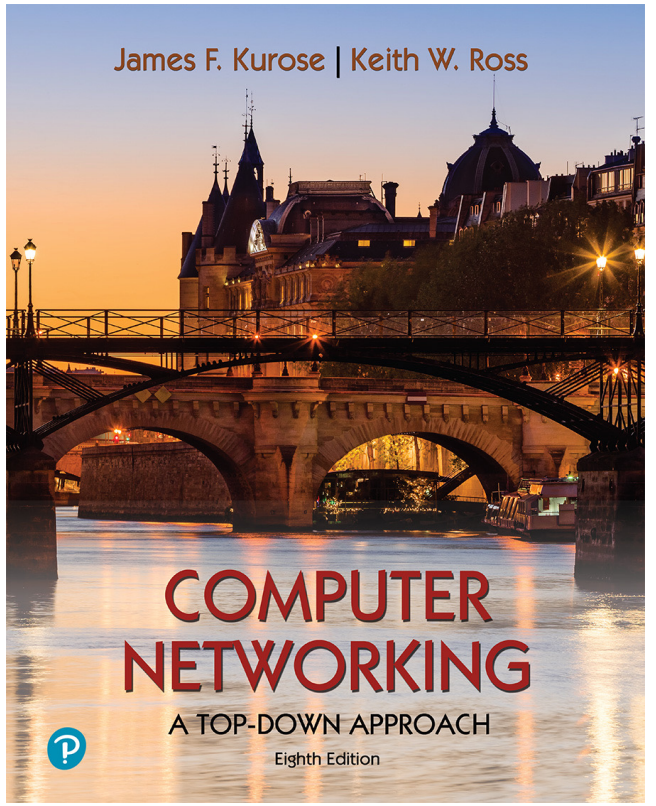
Chapter 3
Evolution of Transport Layer

Prof.ssa Chiara Petrioli
Reti di Elaboratori



In addition material from paperson CUBIC, QUIC, I-TCP, BBR is included. Students can find these references in the course material.

Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Evolving transport-layer functionality

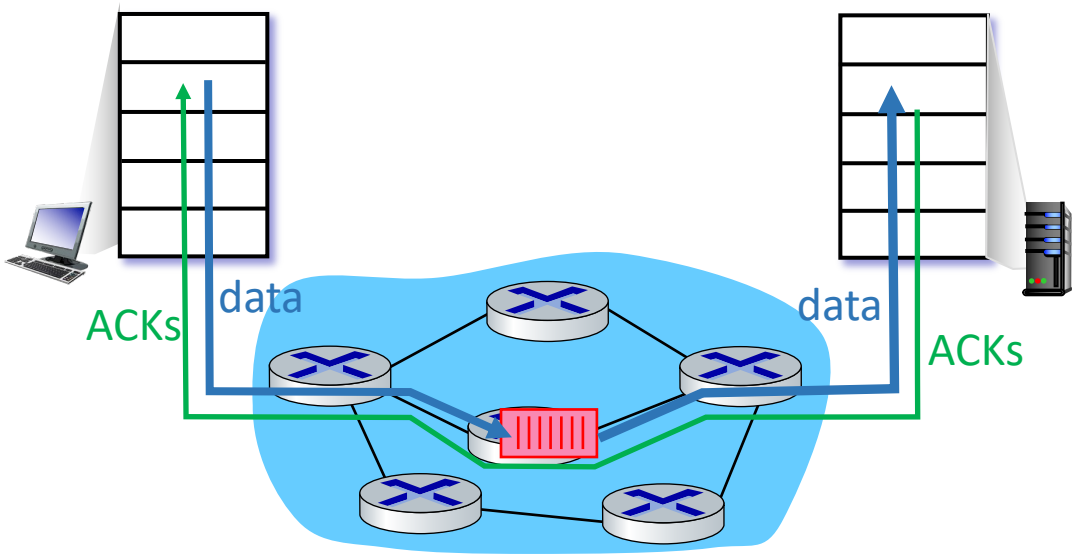
- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

Approaches towards congestion control

End-end congestion control:

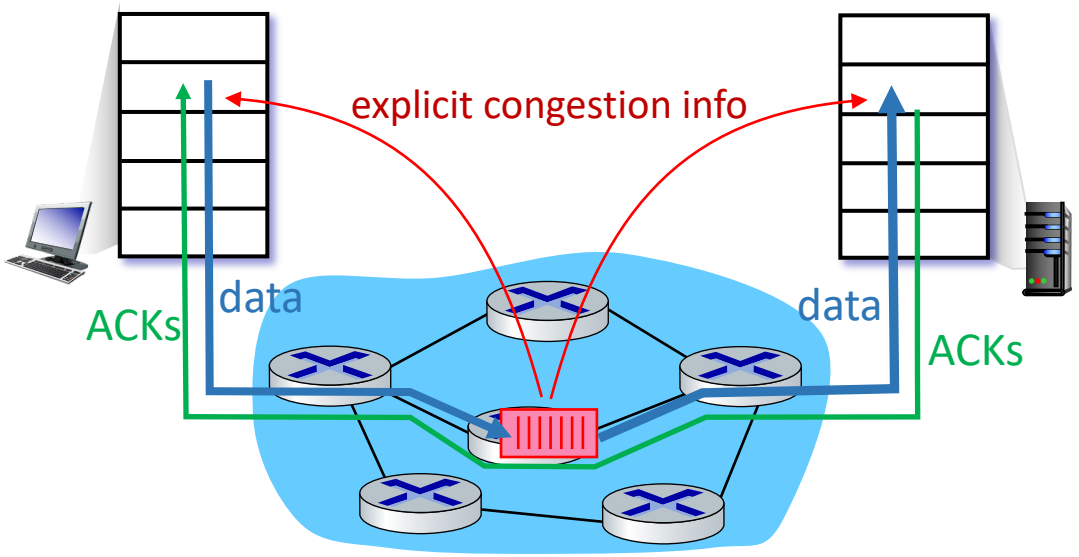
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



Approaches towards congestion control

Network-assisted congestion control:

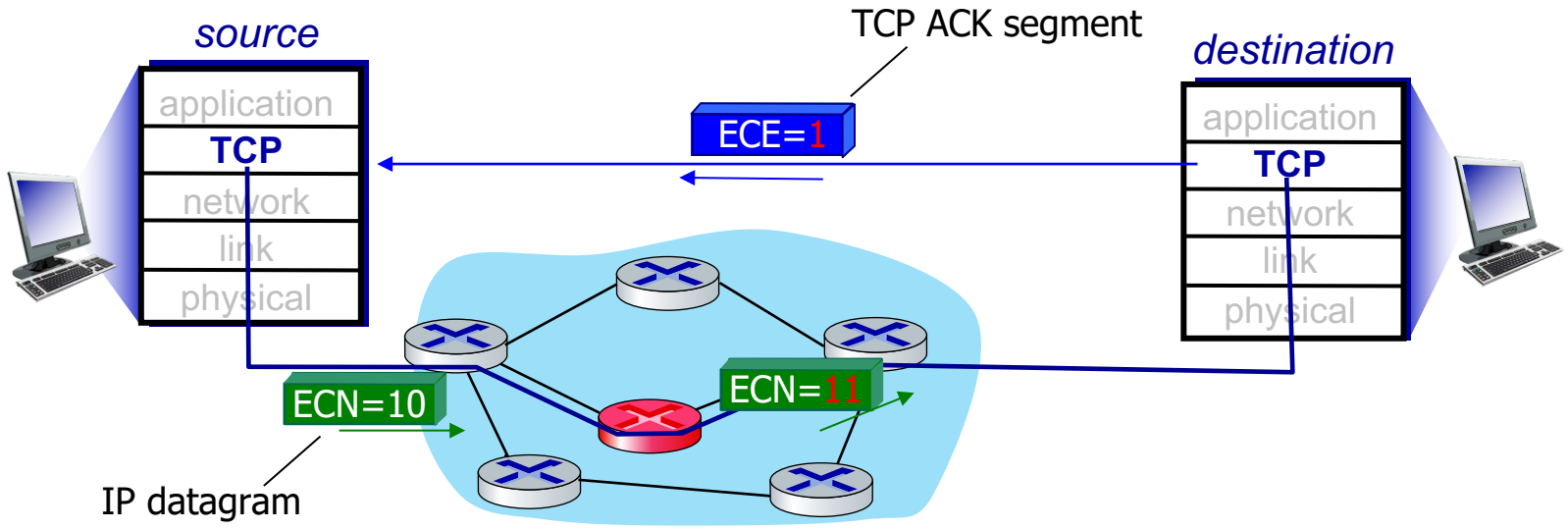
- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECbit protocols



Explicit congestion notification (ECN)

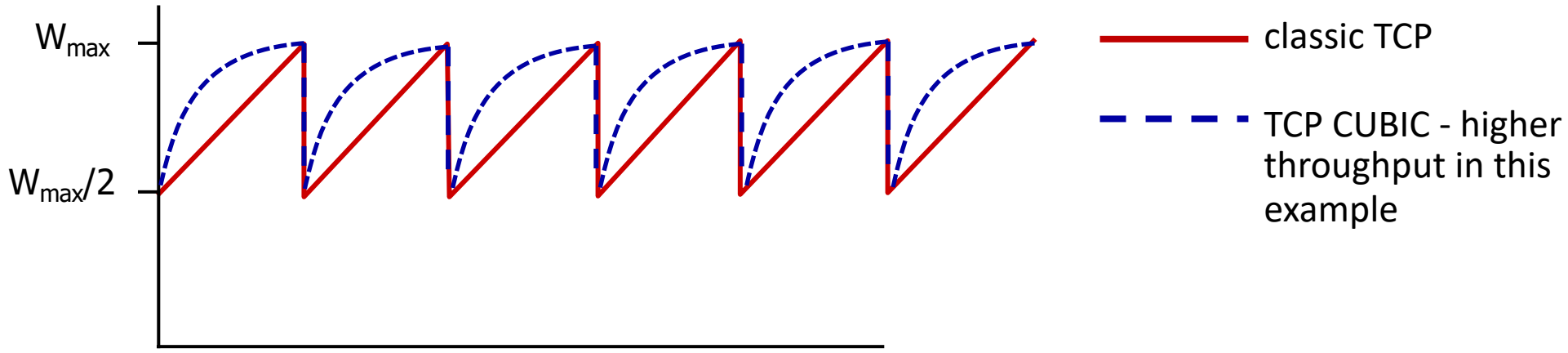
TCP deployments often implement *network-assisted* congestion control:

- 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)



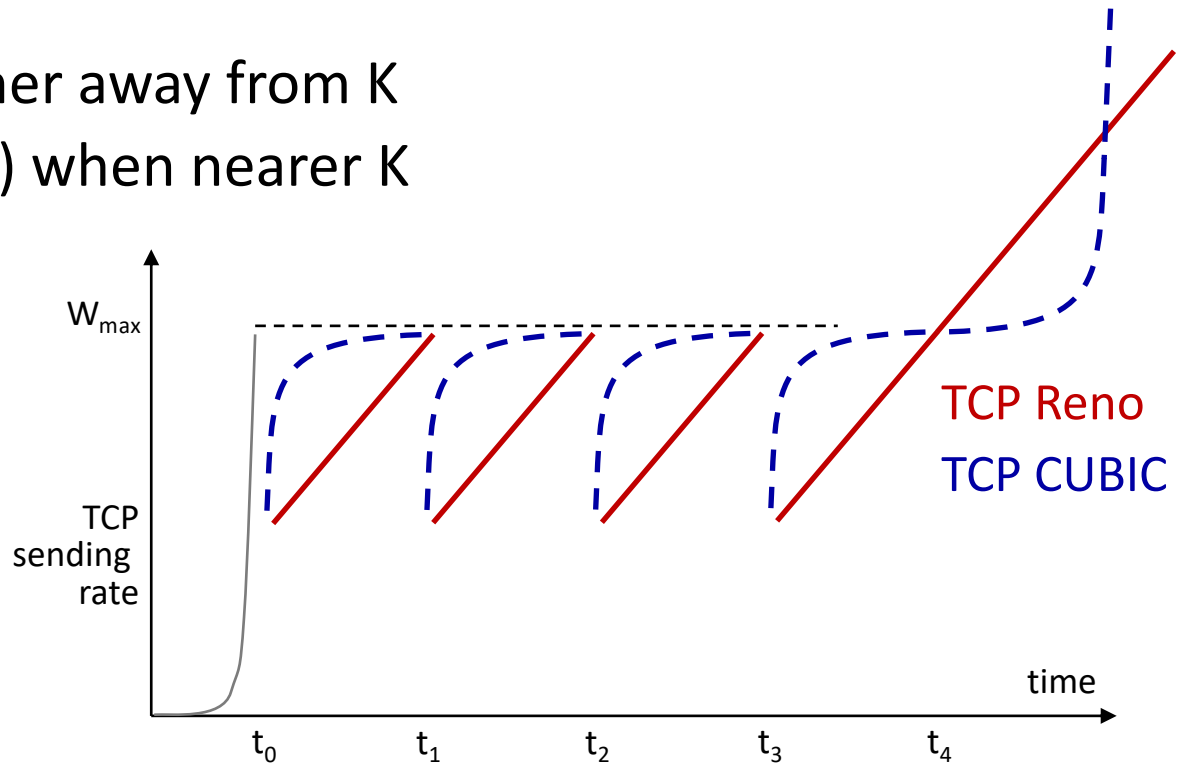
TCP CUBIC

- 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

- 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



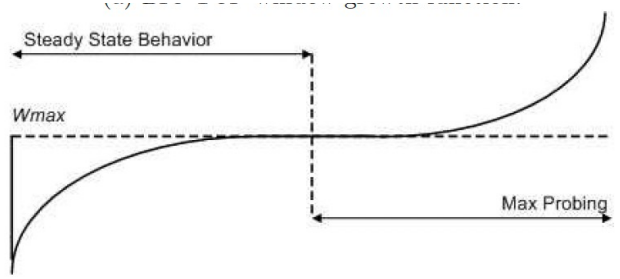
TCP CUBIC

The window growth function of CUBIC uses the following function:

$$W(t) = C(t-K)^3 + W_{max} \quad (1)$$

where C is a CUBIC parameter, t is the elapsed time from the last window reduction, and K is the time period that the above function takes to increase W to W_{max} when there is no further loss event and is calculated by using the following equation:

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}} \quad (2)$$



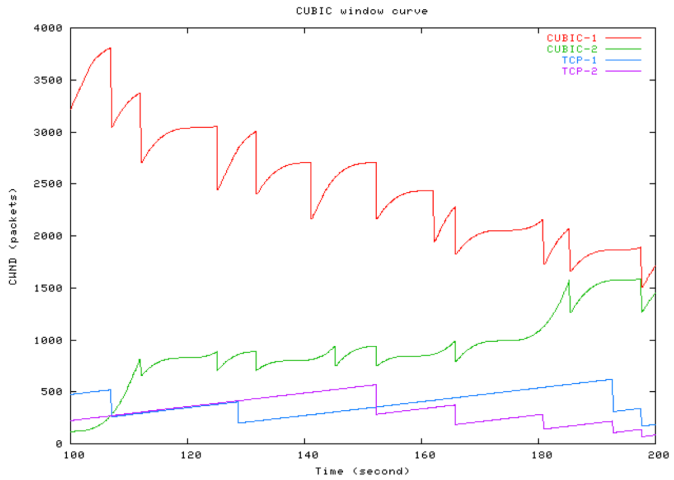


Fig. 3: CUBIC window curves (NS simulation in a network with 500Mbps and 100ms RTT), $C = 0.4$, $\beta = 0.8$.

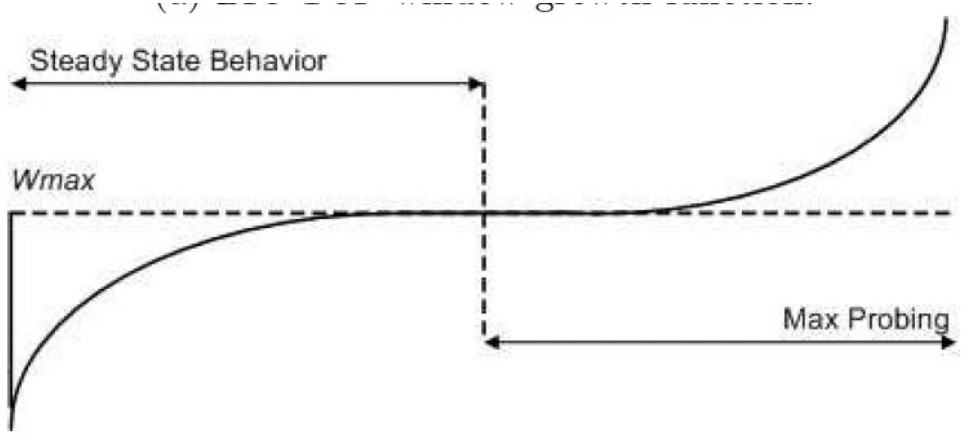
TCP CUBIC

The window growth function of CUBIC uses the following function:

$$W(t) = C(t-K)^3 + W_{max} \quad (1)$$

where C is a CUBIC parameter, t is the elapsed time from the last window reduction, and K is the time period that the above function takes to increase W to W_{max} when there is no further loss event and is calculated by using the following equation:

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}} \quad (2)$$



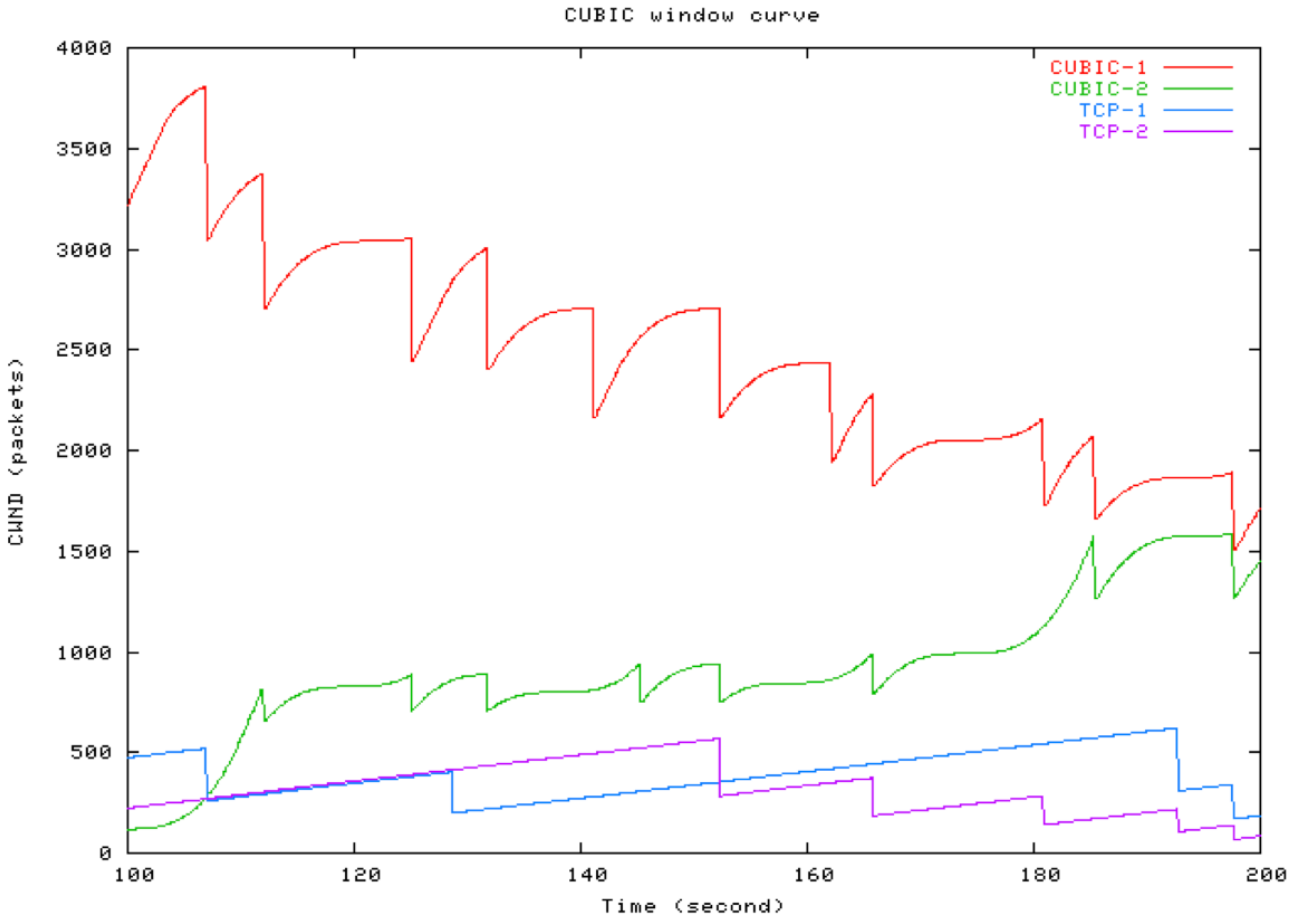
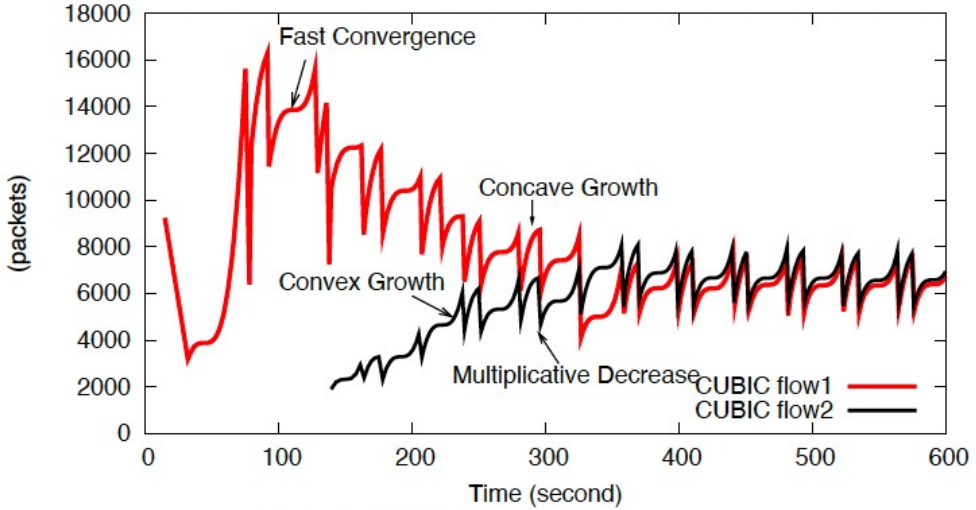
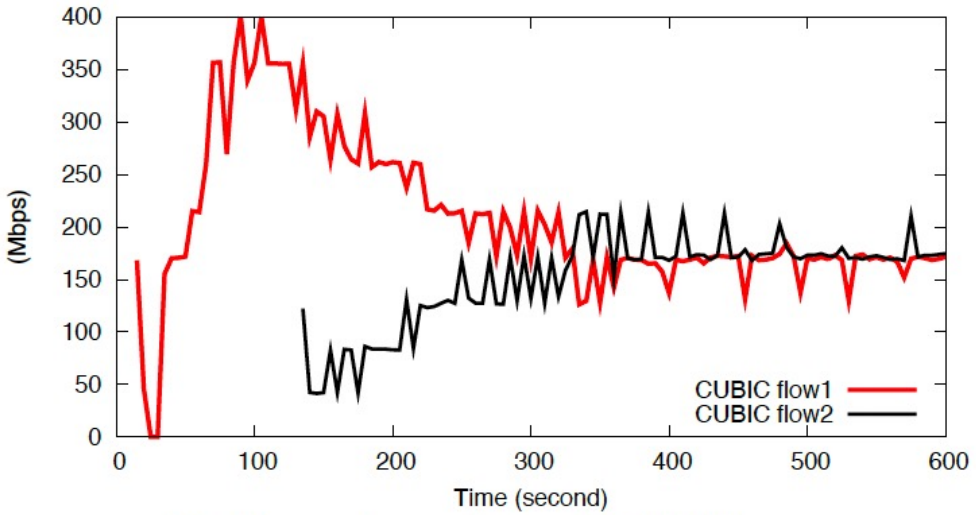


Fig. 3: CUBIC window curves (NS simulation in a network with 500Mbps and 100ms RTT), $C = 0.4$, $\beta = 0.8$.

TCP CUBIC

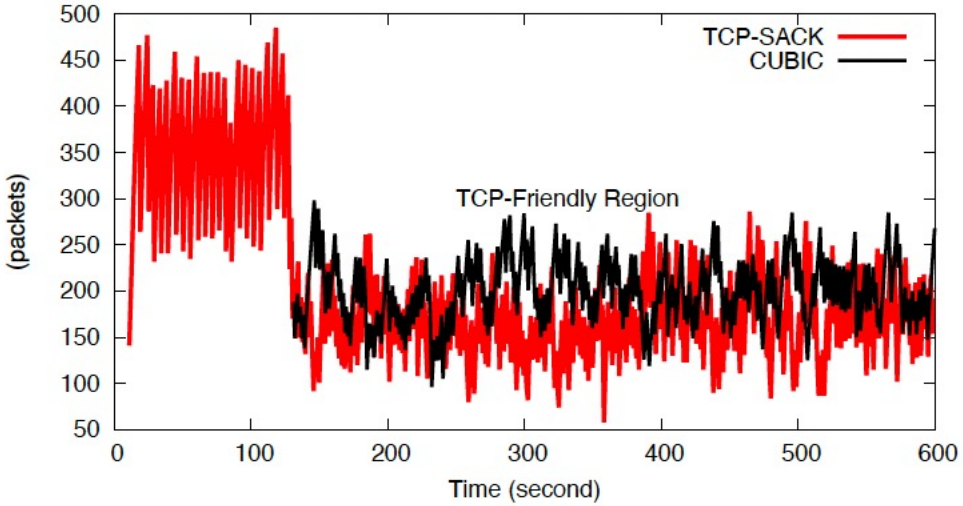


(a) CUBIC window curves.

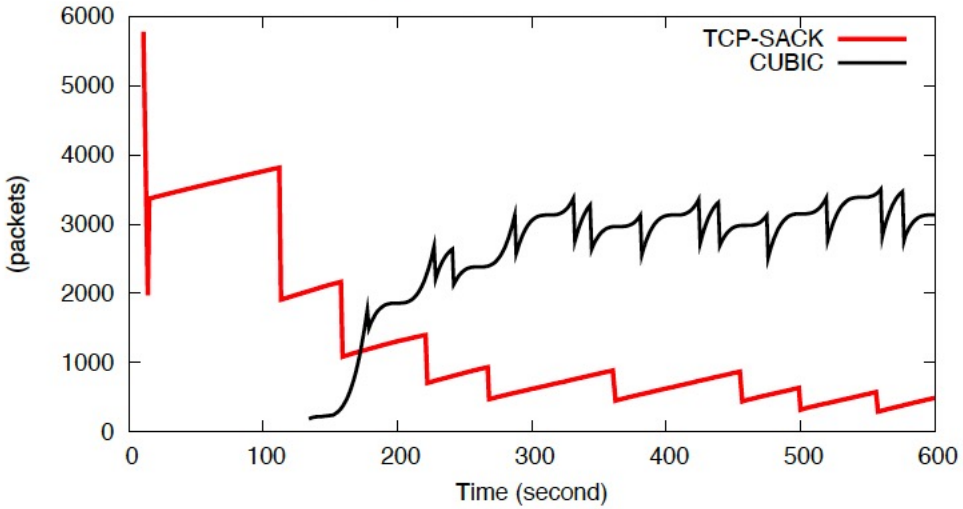


(b) Throughput of two CUBIC flows.

Figure 4: Two CUBIC flows with 246ms RTT.



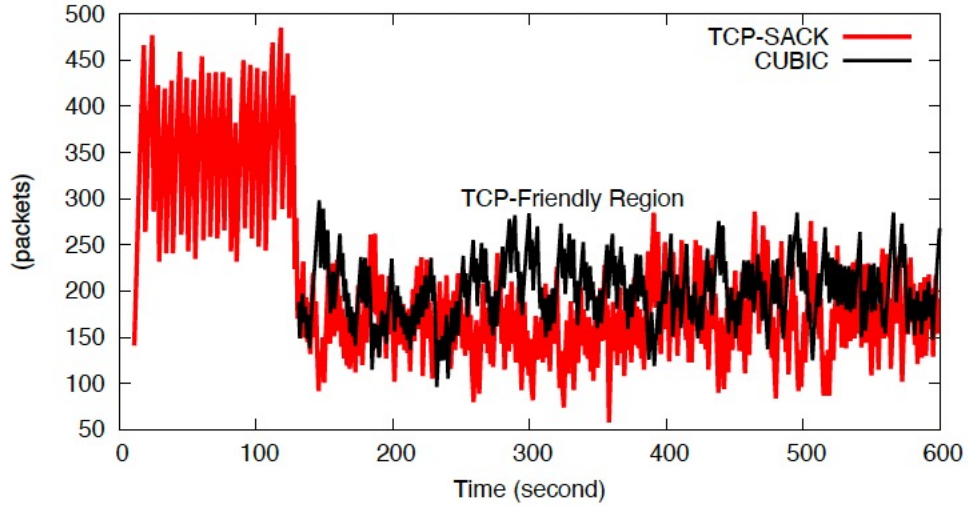
(a) RTT 8ms.



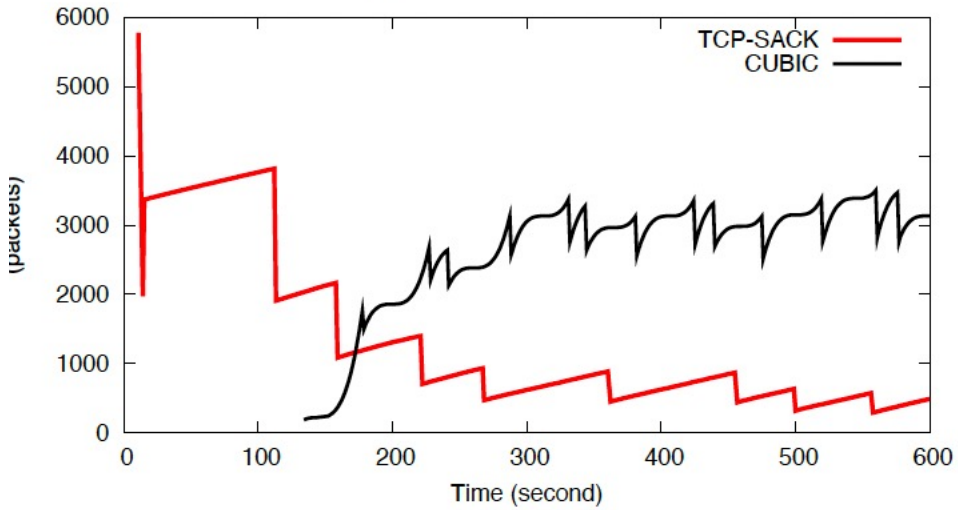
(b) RTT 82ms.

Figure 5: One CUBIC flow and one TCP-SACK flow. Bandwidth is set to 400Mbps.

TCP-friendliness defines whether a protocol is being fair to TCP, and it is critical to the safety of the protocol. When a protocol is used, we need to make sure that its use does not unfairly affect the most common network flows (namely TCP). Many different definitions of this property are found in the literature. The most commonly cited one is by [3]: under high loss rate regions where TCP is well-behaving, the protocol must behave like TCP, and under low loss rate regions where TCP has a low utilization problem, it can use more bandwidth



(a) RTT 8ms.

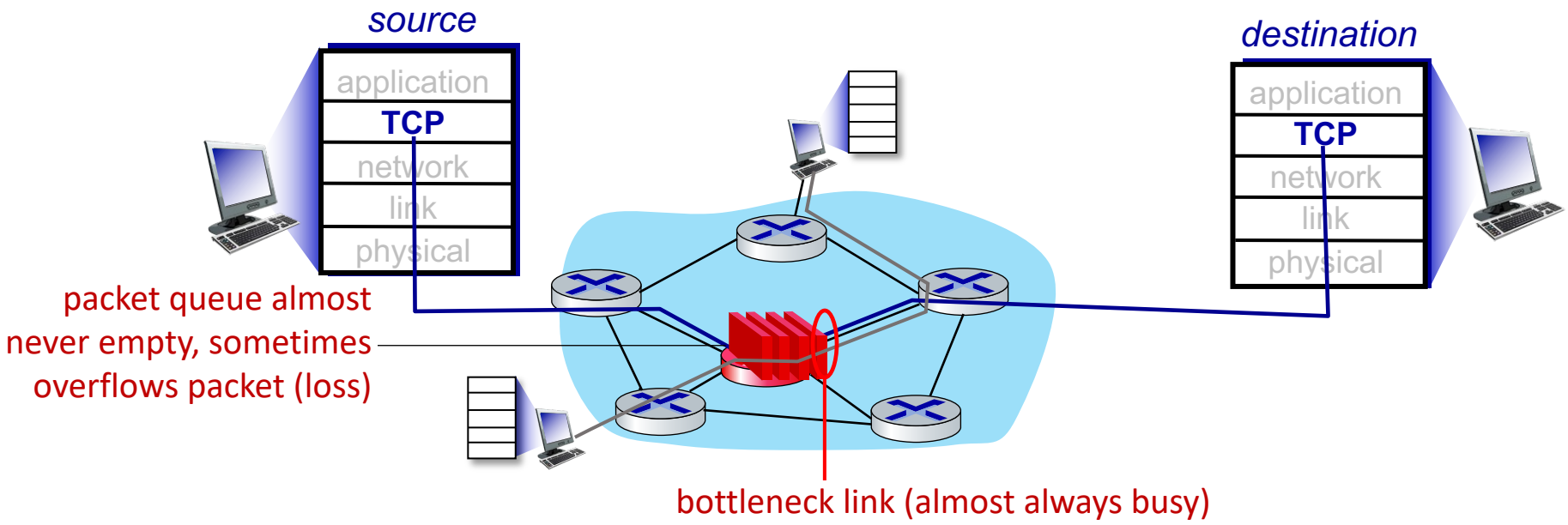


(b) RTT 82ms.

Figure 5: One CUBIC flow and one TCP-SACK flow. Bandwidth is set to 400Mbps.

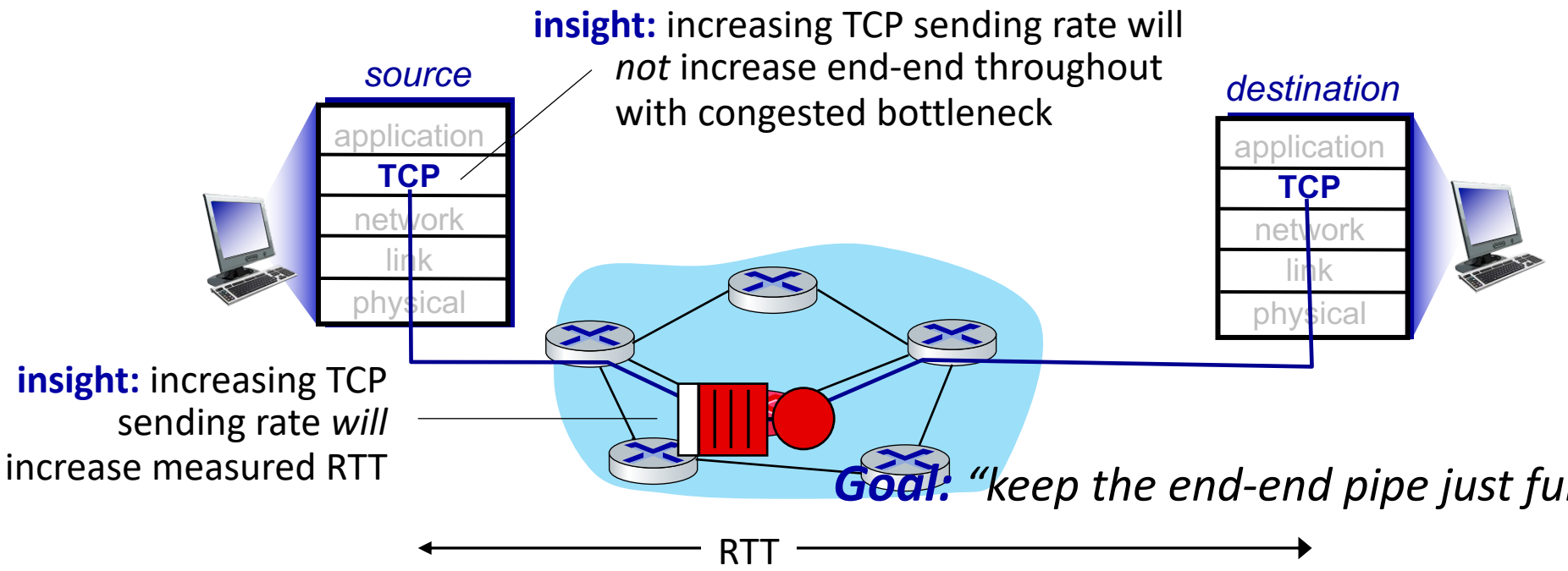
TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



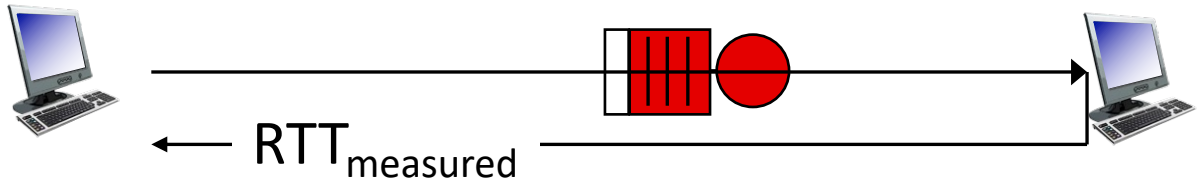
TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



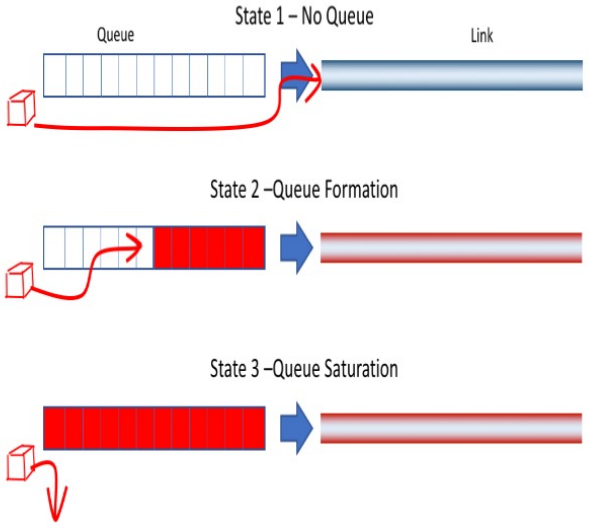
Delay-based TCP congestion control

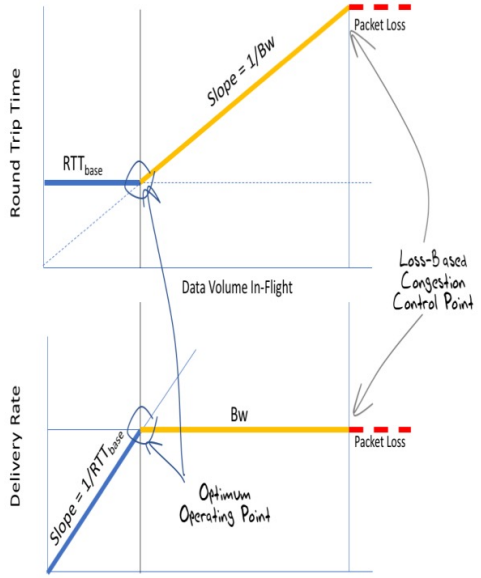
Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

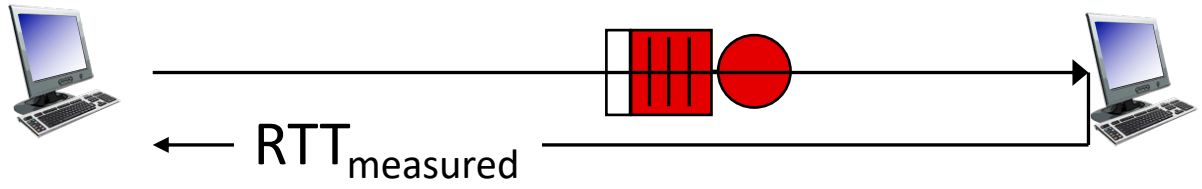
- uncongested throughput with congestion window `cwnd` is cwnd/RTT_{min}





Delay-based TCP congestion control

Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{\text{RTT}_{measured}}$$

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window `cwnd` is cwnd/RTT_{min}

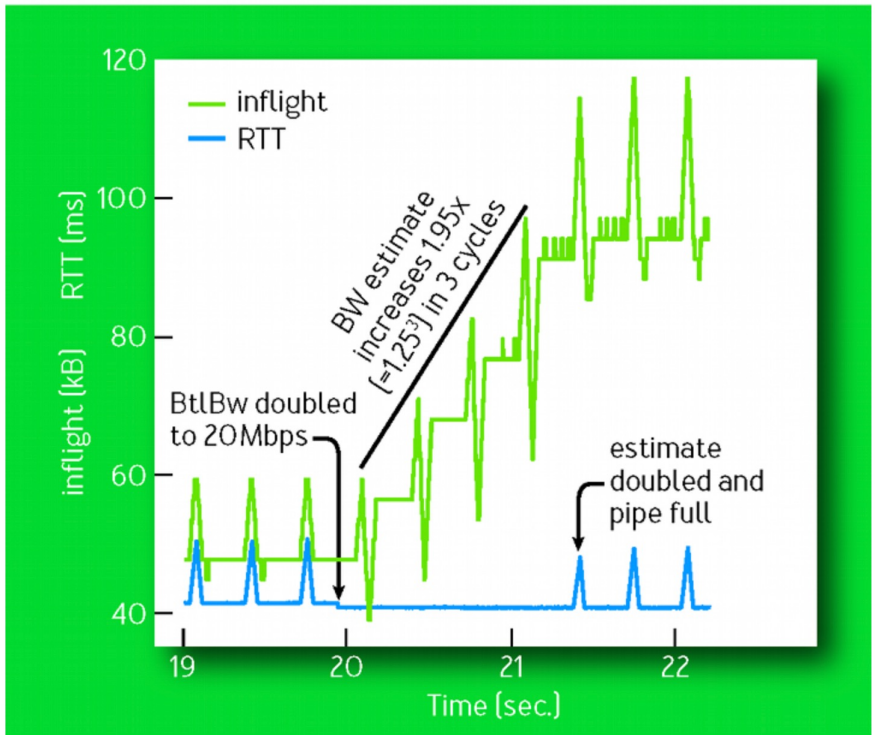
if measured throughput "very close" to uncongested throughput
    increase `cwnd` linearly /* since path not congested */
else if measured throughput "far below" uncongested throughout
    decrease `cwnd` linearly /* since path is congested */

Delay-based TCP congestion control

- congestion control without inducing/forcing loss
- maximizing throughout ("keeping the just pipe full… ") while keeping delay low ("…but not fuller")
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network

BBR

FIGURE 3: **BANDWIDTH CHANGE**



Evolving transport-layer functionality

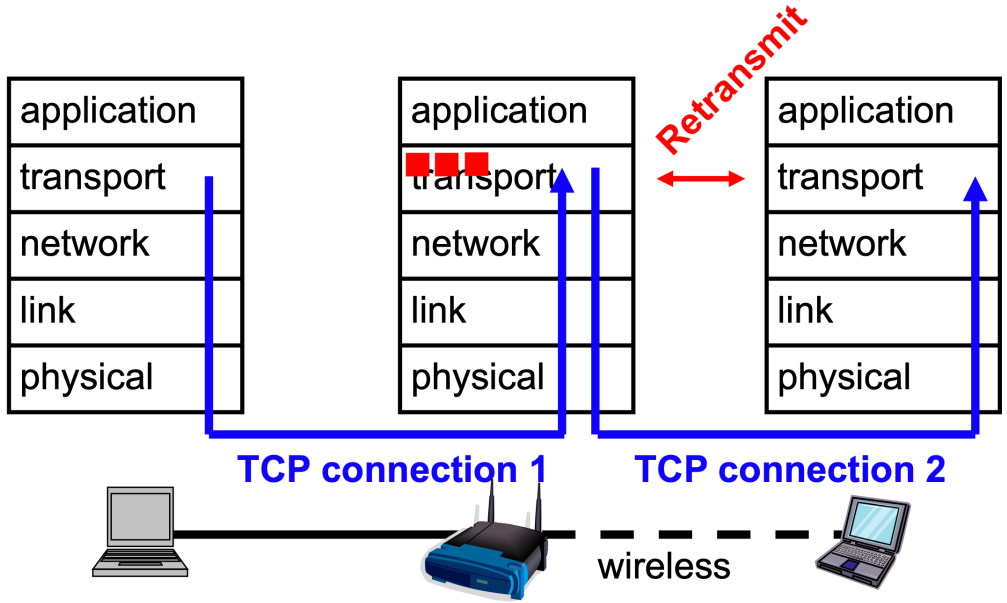
- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

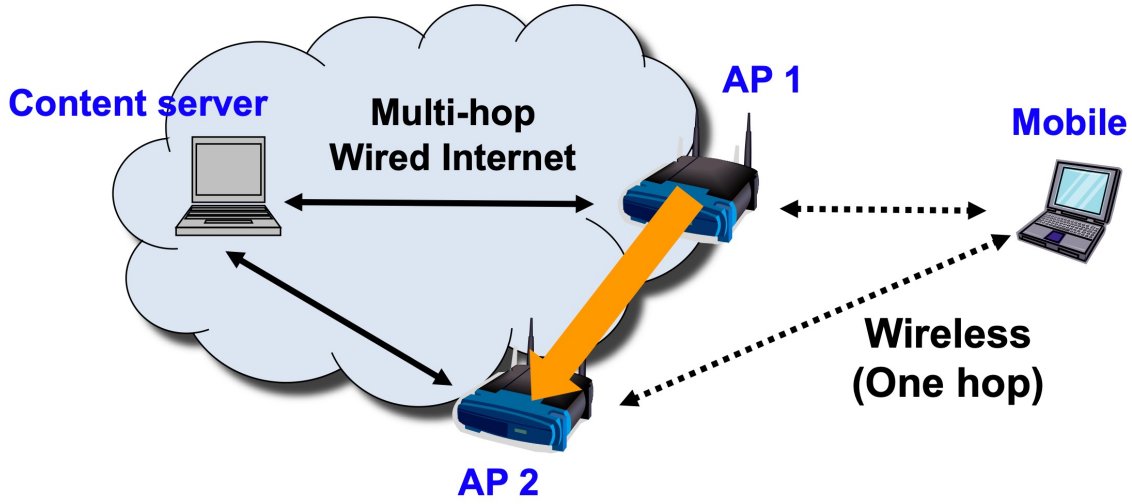
Transport over wireless access links

- TCP interprets packet loss as a sign of congestion → TCP sender reduces congestion window
- However, even if congestion leads to loss it is not necessarly true that loss is due to congestion!
  - Wireless access often used. Wireless links suffers from higher BER and thus packet losses.
    - Packet loss over wireless links con also occur due to interference, random channel errors, cellular or WLAN handoff leading to temporary loss not due to congestion but to features of the wireless link → reducing window is too conservative and leads to poor throughput performance
- Solutions?
  - Mask wireless losses so that the TCP sender will not slow down → split connection approach, TCP Snoop.
  - Explicitly notify sender about cause of loss.

Split connection/Indirect TCP (I-TCP)

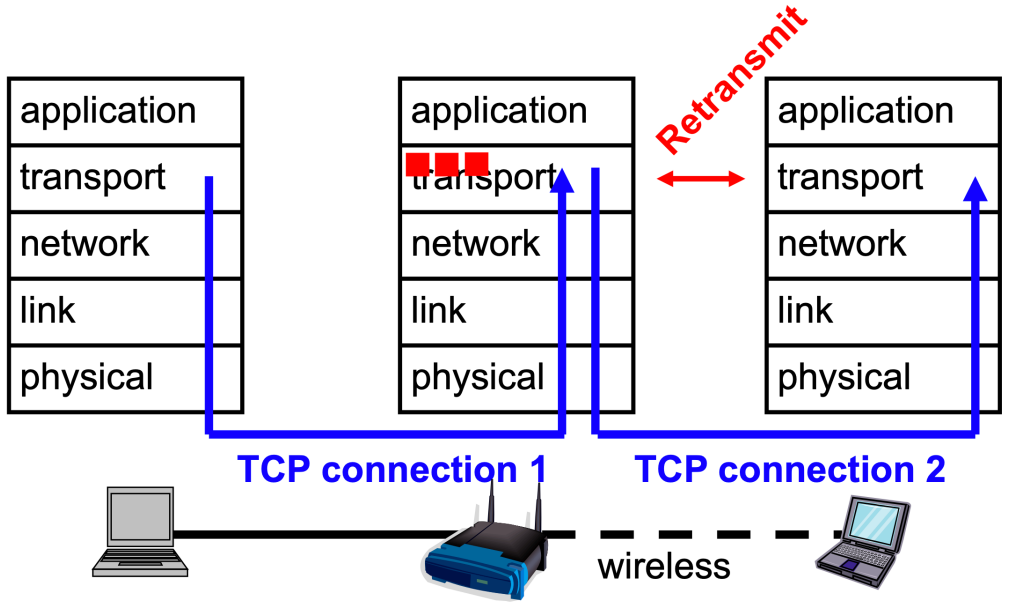


– On handoff from AP 1 to AP 2, **connection state** must **move from AP 1 to AP 2**

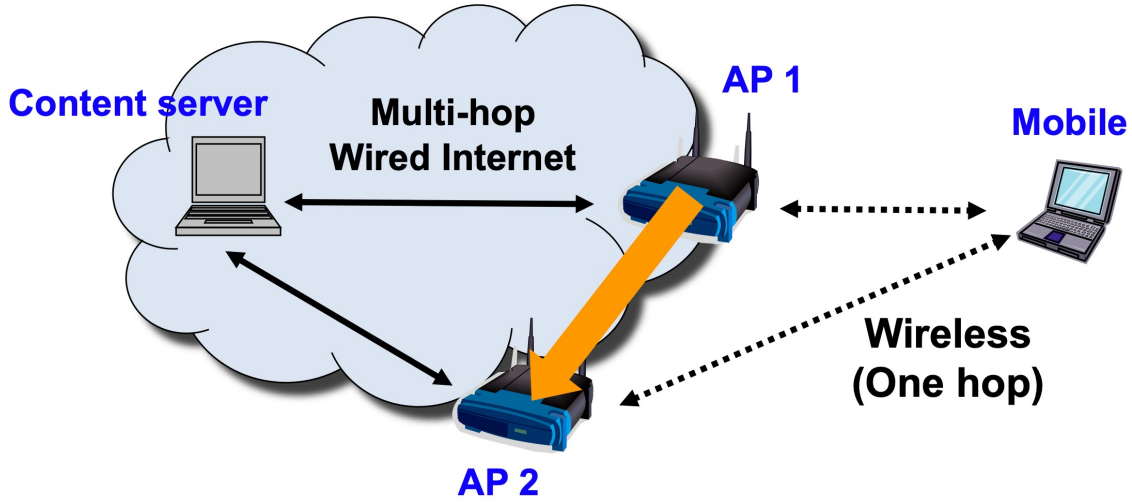


- Local retransmission or dedicated transport layer over the wireless link
- Effects of wireless link errors do not propagate on TCP connection 1

Split connection/Indirect TCP (I-TCP)



– On handoff from AP 1 to AP 2, **connection state** must **move from AP 1 to AP 2**



- Not end to end
- Requires buffering and maintaining state at the AP
- State must be moved in case of handoff
- Sometimes congestion can also occur over the wireless link

Evolving transport-layer functionality

- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

- moving transport–layer functions to application layer, on top of UDP
  - HTTP/3: QUIC

QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)

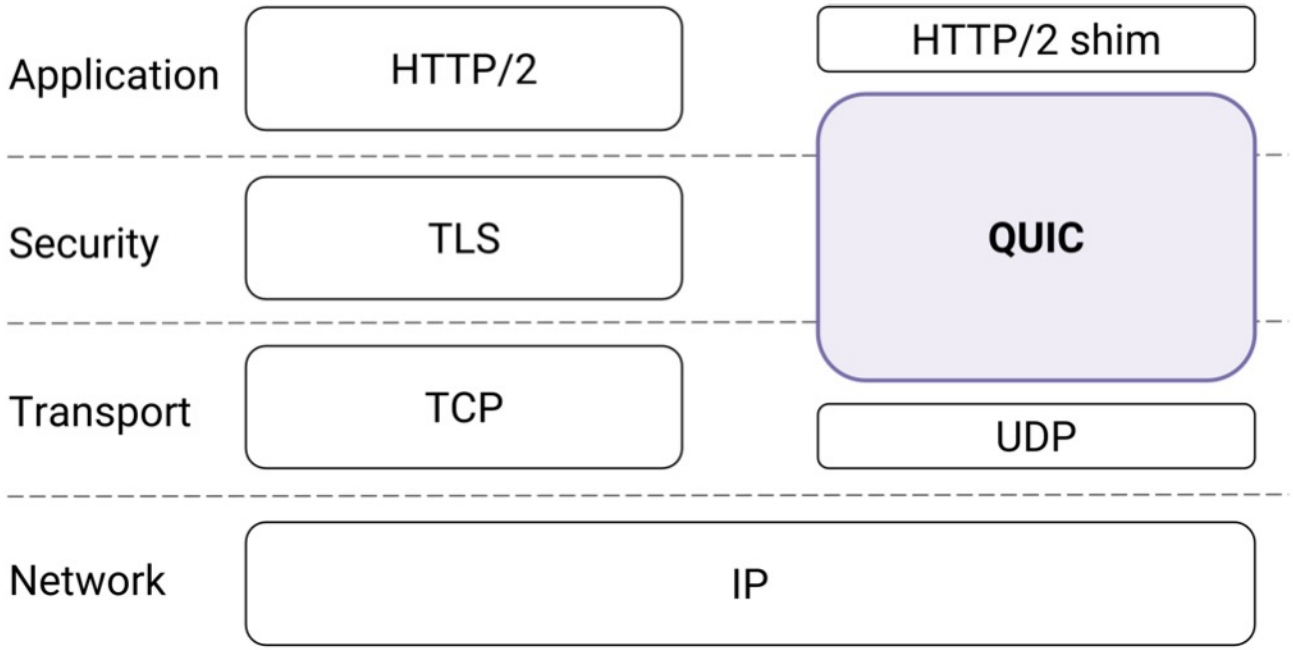


Figure 1: QUIC in the traditional HTTPS stack.

QUIC: Why?

- TCP is commonly implemented in the Operatiing System (OS) kernel. As a result, even if TCP modifications were deployable, pushing changes to TCP stacks typically requires OS upgrades. This coupling of the transport implementation to the OS limits deployment velocity of TCP changes.
- Growth in latency-sensitive web services and use of the web as a platform for applications is placing unprecedented demands on reducing web latency.
- The Internet is rapidly shifting from insecure to secure traffic, which adds delays. The generality of TCP and TLS continues to serve Internet evolution well, but the costs of layering have become increasingly visible with increasing latency demands on the HTTPS stack. TCP connections commonly incur at least one round-trip delay of connection setup time before any application data can be sent, and TLS adds two round trips to this delay . While network bandwidth has increased over time, the speed of light remains constant. Most connections on the Internet, and certainly most transactions on the web, are short transfers and are most impacted by unnecessary handshake round trips.
- QUIC encrypts transport headers and builds transport functions atop UDP, avoiding dependence on vendors and network operators and moving control of transport deployment to the applications that directly benefit from them
- HTTP/2 multiplexes multiple objects and recommends using a single TCP connection to any server. TCP's bytestream abstraction, however, prevents applications from controlling the framing of their communications and imposes a "latency tax" on application frames whose delivery must wait for retransmissions of previously lost TCP segments.

QUIC: Quick UDP Internet main points

- QUIC is an encrypted transport: packets are authenticated and encrypted, preventing modification and limiting ossification of the protocol by middleboxes.
- QUIC uses a cryptographic handshake that minimizes handshake latency for most connections by using known server credentials on repeat connections and by removing redundant handshake-overhead at multiple layers in the network stack.
- QUIC eliminates head-of-line blocking delays by using a lightweight data-structuring abstraction, streams, which are multiplexed within a single connection so that loss of a single packet blocks only streams with data in that packet.
- It allows connections to migrate across IP address changes by using a Connection ID to identify connections instead of the IP/port -tuple
- It provides flow control to limit the amount of data buffered at a slow receiver and ensures that a single stream does not consume all the receiver's buffer by using per-stream flow control limits.
- It provides a modular congestion control interface for experimenting with various controllers

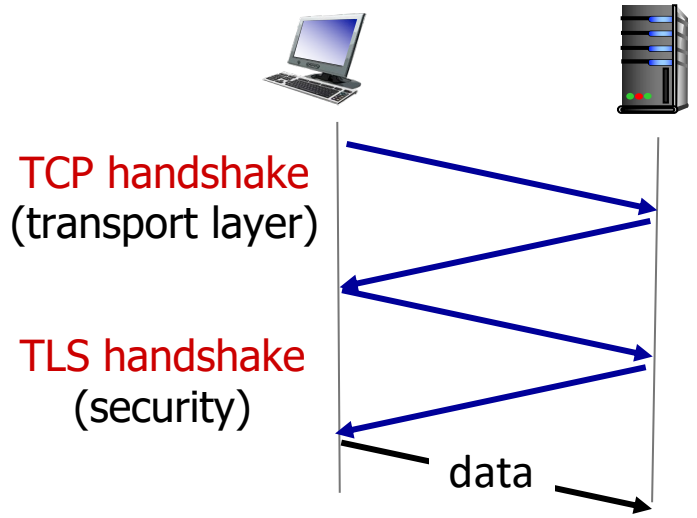
QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT
- **flow control**

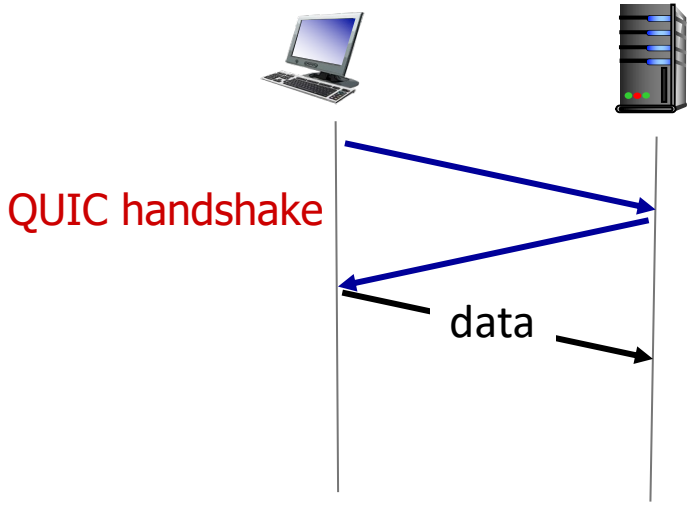
- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

QUIC: Connection establishment



TCP (reliability, congestion control state) + TLS (authentication, crypto state)

- 2 serial handshakes



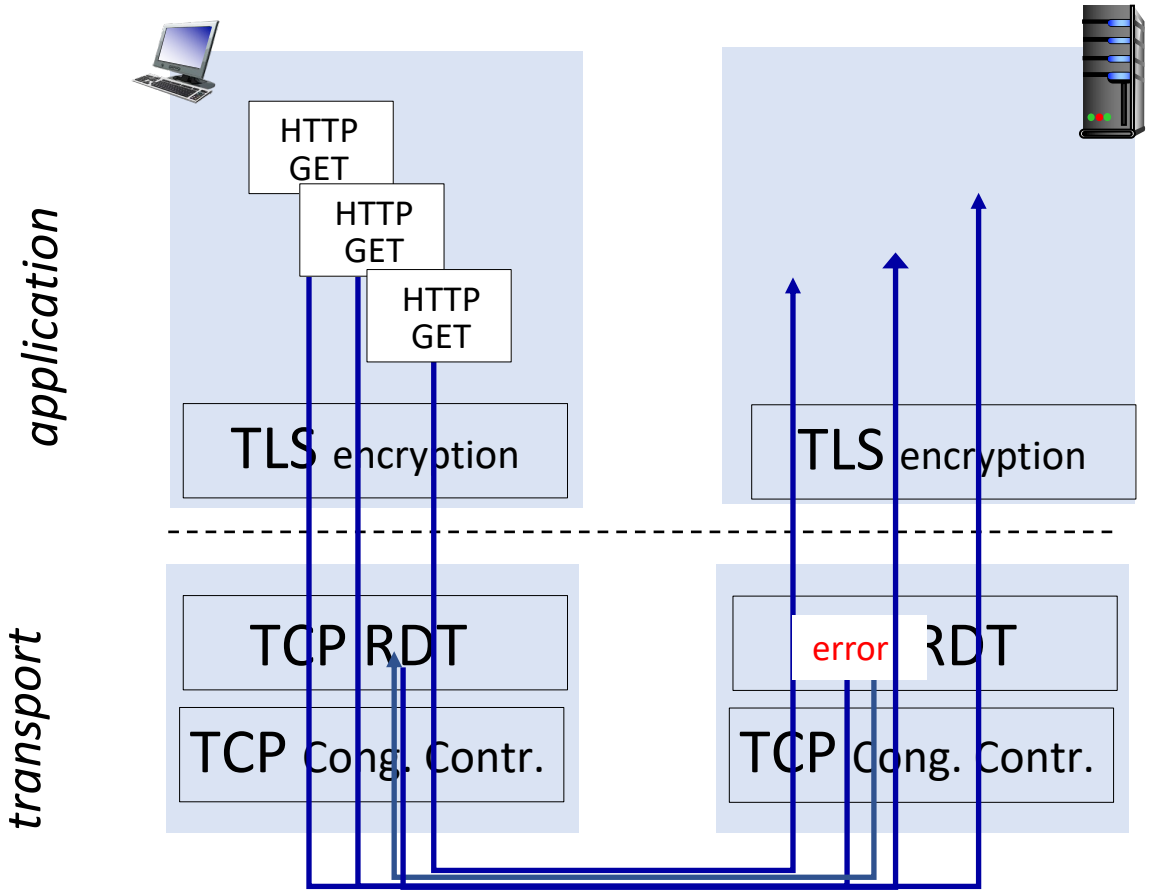
QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: Quick UDP Internet main points

- QUIC is an encrypted transport: packets are authenticated and encrypted, preventing modification and limiting ossification of the protocol by middleboxes.
- QUIC uses a cryptographic handshake that minimizes handshake latency for most connections by using known server credentials on repeat connections and by removing redundant handshake-overhead at multiple layers in the network stack.
- QUIC eliminates head-of-line blocking delays by using a lightweight data-structuring abstraction, streams, which are multiplexed within a single connection so that loss of a single packet blocks only streams with data in that packet.
- It allows connections to migrate across IP address changes by using a Connection ID to identify connections instead of the IP/port -tuple
- It provides flow control to limit the amount of data buffered at a slow receiver and ensures that a single stream does not consume all the receiver's buffer by using per-stream flow control limits.
- It provides a modular congestion control interface for experimenting with various controllers

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

QUIC: performance