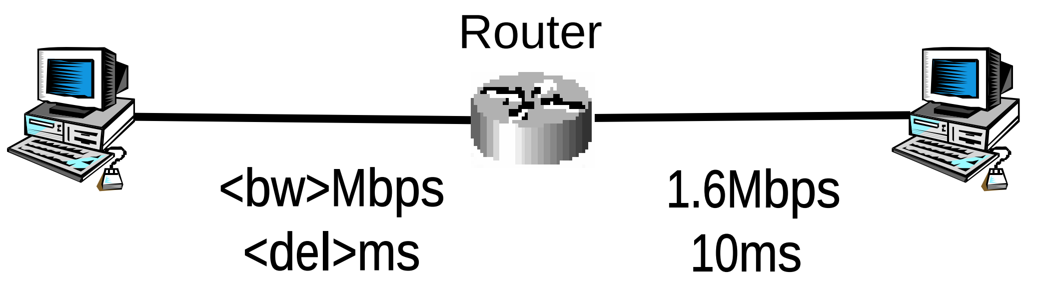
Pipelining examples and self-clocking

G.Bianchi, G.Neglia, V.Mancuso

Università degli Studi di Palermo

Pipelining

Packet size: 1000bytes



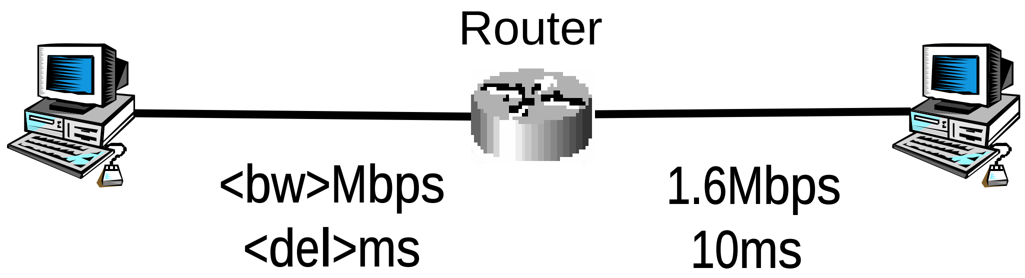
- ➔ **ns ack_clock2.tcl <bw> <del> <qu> <W> <stop>**
- ➔ **ns ack_clock2.tcl 1.6Mb 30ms 100 10 1**
- ➔ **ns ack_clock2.tcl 1.6Mb 30ms 100 17 1**
- ➔ **ns ack_clock2.tcl 1.6Mb 30ms 100 18 1**
- ➔ **ns ack_clock2.tcl 1.6Mb 30ms 100 36 1**

Self-clocking

- ➔**The ACK policy makes the protocol *self-clocking*:**
  - ⇨it dynamically adapts its transmission speed
  - ⇨*trying* to satisfy a conservation principle: a new packet for each old one leaving the network

Self-clocking (example)

Packet size: 1000bytes



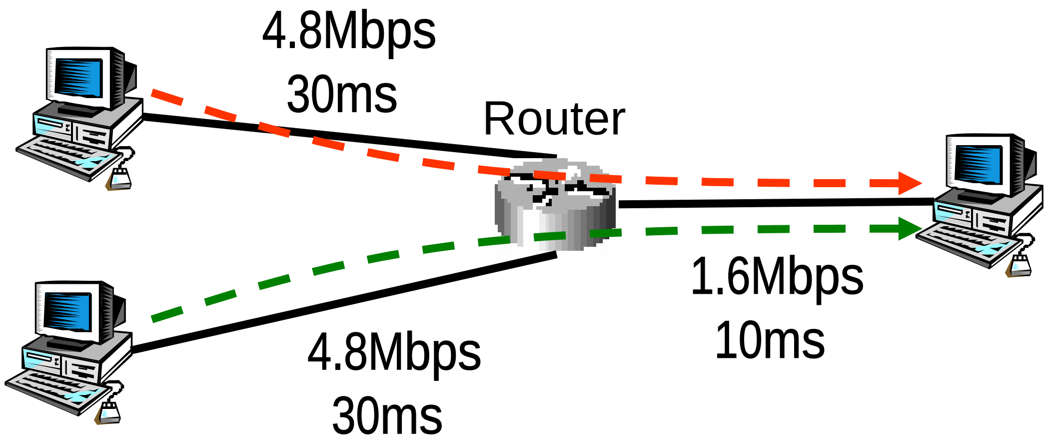
➔ **ns ack_clock2.tcl 4.8Mb 30ms 100 17 1**

➔ **ns ack_clock2.tcl 4.8Mb 30ms 100 18 1**

➔ **ns ack_clock2.tcl 4.8Mb 30ms 100 36 1**

Self-clocking: is it enough?

Packet size: 1000bytes



→ **ns congavd_motivation2.tcl 100 DropTail false false 4**

but…

→ **ns ack_clock2.tcl 4.8Mb 30ms 10 36 1**

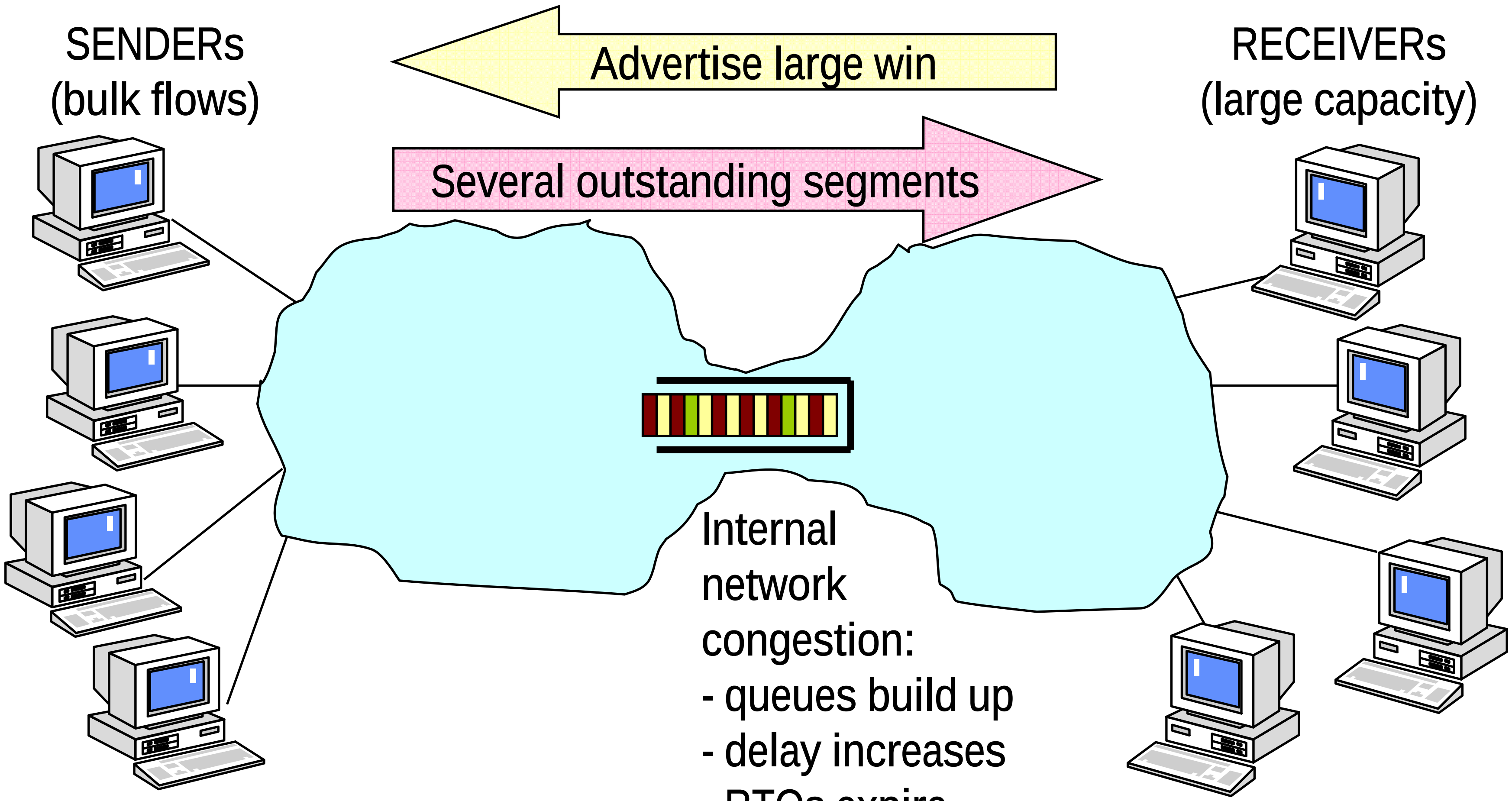
→ **ns congavd_motivation2.tcl 10 DropTail false false 4**

TCP
congestion control

(very good summary in RFC 2581)

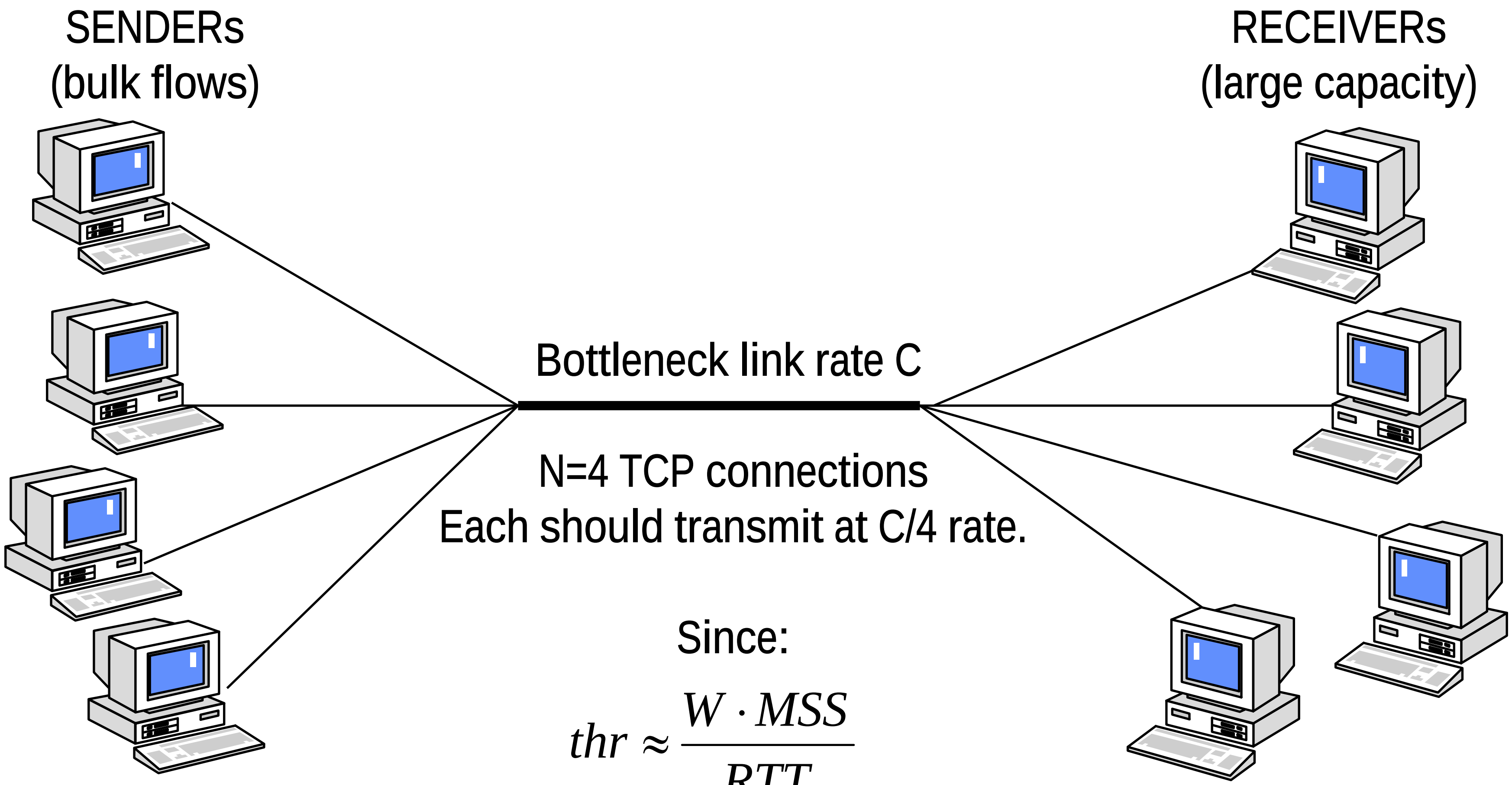
G.Bianchi, G.Neglia, V.Mancuso

The problem of congestion



Internal network congestion:
- queues build up
- delay increases
- RTOs expire
- more segments transmitted -> more congestion!

The goal of congestion control



N=4 TCP connections
Each should transmit at C/4 rate.

Since:

$$thr \approx \frac{W \cdot MSS}{RTT}$$

Each should adapt W accordingly…
How sources can be lead to know the RIGHT value of W??

History of congestion control

- ➔ **Before 1986: the Internet meltdown!**
  - ⇨ No mechanisms employed to react to internal network congestion
- ➔ **1986: Slow Start + Congestion avoidance**
  - ⇨ Van Jacobson, TCP Berkeley
  - ⇨ Proposes idea to make TCP reactive to congestion
- ➔ **1988: Fast Retransmit (TCP Tahoe)**
  - ⇨ Van Jacobson, first implemented in 1988 BSD Tahoe release
- ➔ **1990: Fast Recovery (TCP Reno)**
  - ⇨ Van Jacobson, first implemented in 1990 BSD Reno release
- ➔ **1995-1996: TCP NewReno**
  - ⇨ Floyd (based on Hoe's idea), RFC 2582
  - ⇨ Today the de-facto standard

TCP approach for detecting and controlling congestion

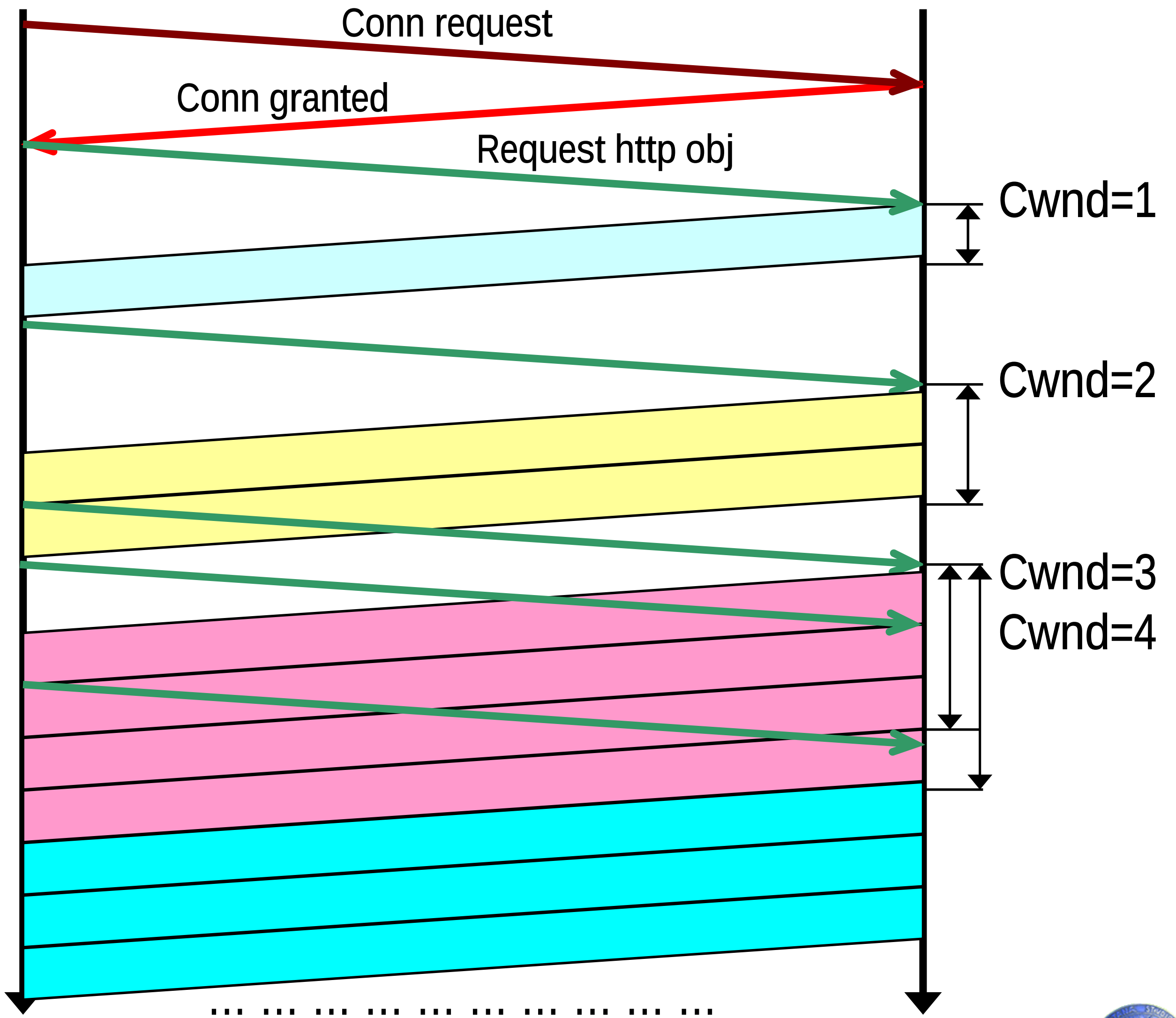
- ➔ **IP protocol does not implement mechanisms to detect congestion in IP routers**
  - →Unlike other networks, e.g. ATM
- ➔ **necessary indirect ways (TCP is an end-to-end protocol)**
- ➔ **TCP approach: congestion detected by lack of acks**
  - » couldn't work efficiently in the 60s & 70s (error prone transmission lines)
  - » OK in the 80s & 90s (reliable transmission)
  - » what about wireless networks???
- ➔ **Controlling congestion: use a SECOND window (congestion window)**
  - →Locally computed at sender
  - →Outstanding segments: min(receiver_window, congestion_window)

Starting a TCP transmission

- **A new offered flow may suddenly overload network nodes**
  - receiver window is used to avoid recv buffer overflow
  - But it may be a large value (16-64 KB)
- **Idea: slow start**
  - Start with small value of cwnd
  - And increase it as soon as packets get through
    - Arrival of ACKs = no packet losts = no congestion
- **Initial cwnd size:**
  - Just 1 MSS!
  - Recent (1998) proposals for more aggressive starts (up to 4 MSS) have been found to be dangerous

Slow start – exponential increase

- ➔ First start: set congestion window
  cwnd = 1MSS
- ➔ send cwnd segments
  - ⇨ assume cwnd <= receiver win
- ➔ upon successful reception:
  - ⇨ Cwnd +=1 MSS
  - ⇨ i.e. double cwnd every RTT
  - ⇨ until reaching receiver window advertisement
  - ⇨ <u>OR a segment gets lost</u>



Detecting congestion and restarting

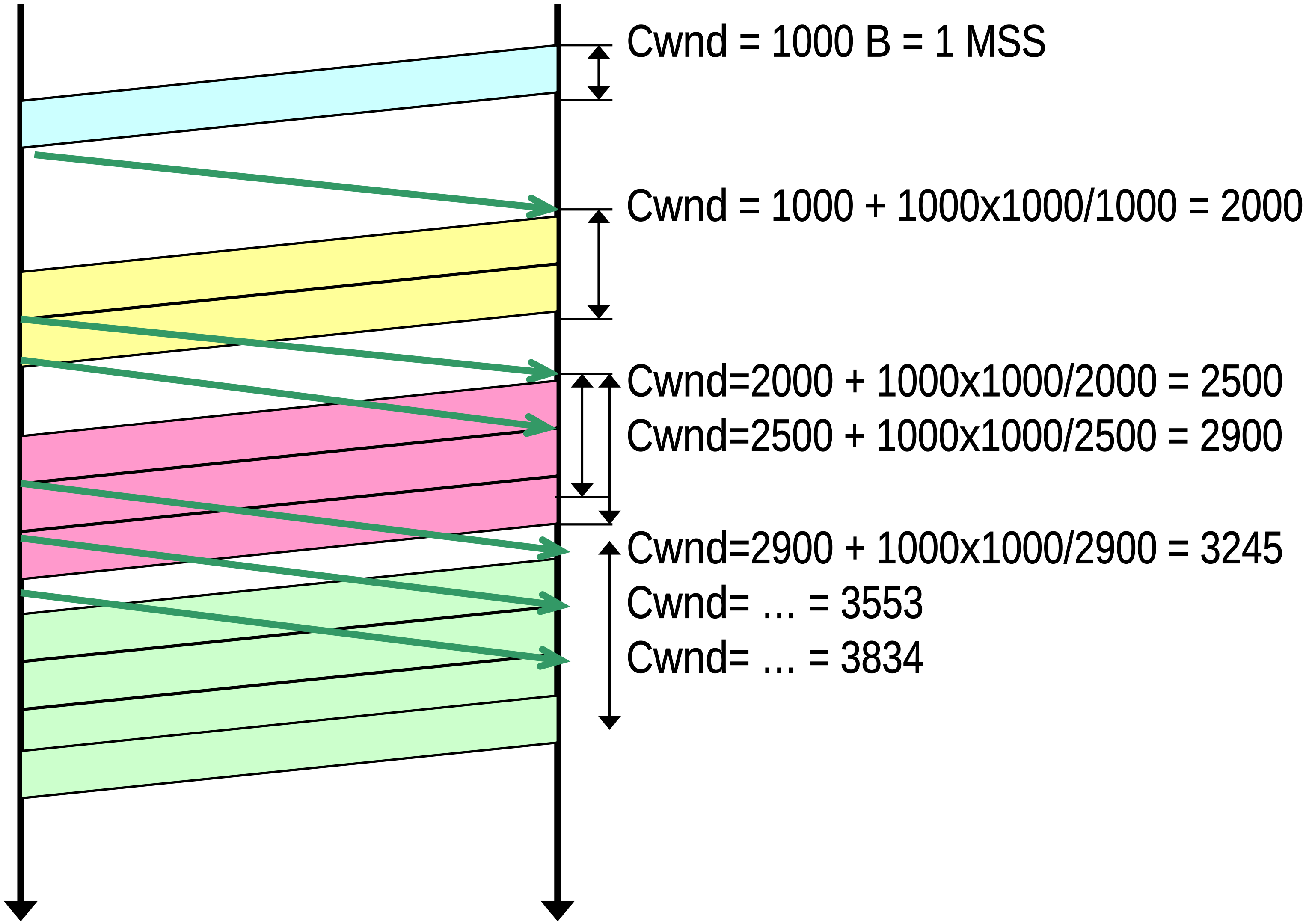
- ➔ **Segment gets lost**
  - ⇨ Detected via RTO expiration
  - ⇨ Indirectly notifies that one of the network nodes along the path has lost segment
    - » Because of full queue
- ➔ **Restart from cwnd=1 (slow start)**
- ➔ **But introduce a supplementary control: slow start threshold**
    - →sstresh = max(cwnd/2, 2MSS)
  - ⇨ The idea is that we now KNOW that there is congestion in the network, and we need to increase our rate in a more careful manner…
  - ⇨ ssthresh defines the "congestion avoidance" region

Congestion avoidance

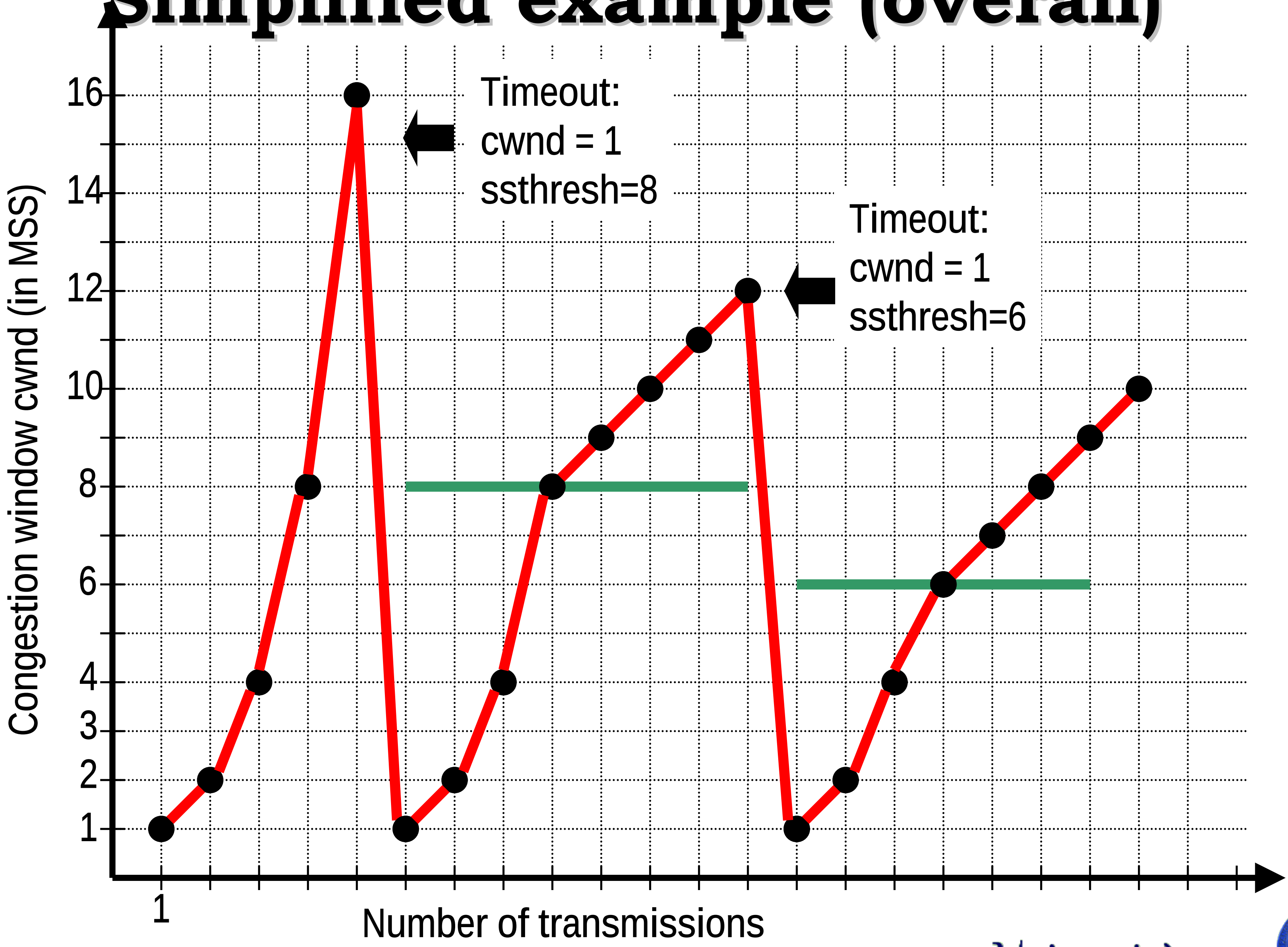
- **If cwnd < ssthresh**
  - Slow start region: Increase rate exponentially
- **If cwnd >= ssthresh**
  - Congestion avoidance region : Increase rate linearly
  - At rate 1 MSS per RTT
    - Practical implementation:
      cwnd += MSS*MSS/cwnd
    - Good approximation for 1 MSS per RTT
    - Alternative (exact) implementations: count!!
- **Which initial ssthresh?**
  - » ssthresh initially set to 65535: unreachable!

In essence, congestion avoidance is flow control imposed by sender while advertised window is flow control imposed by receiver

Congestion avoidance example

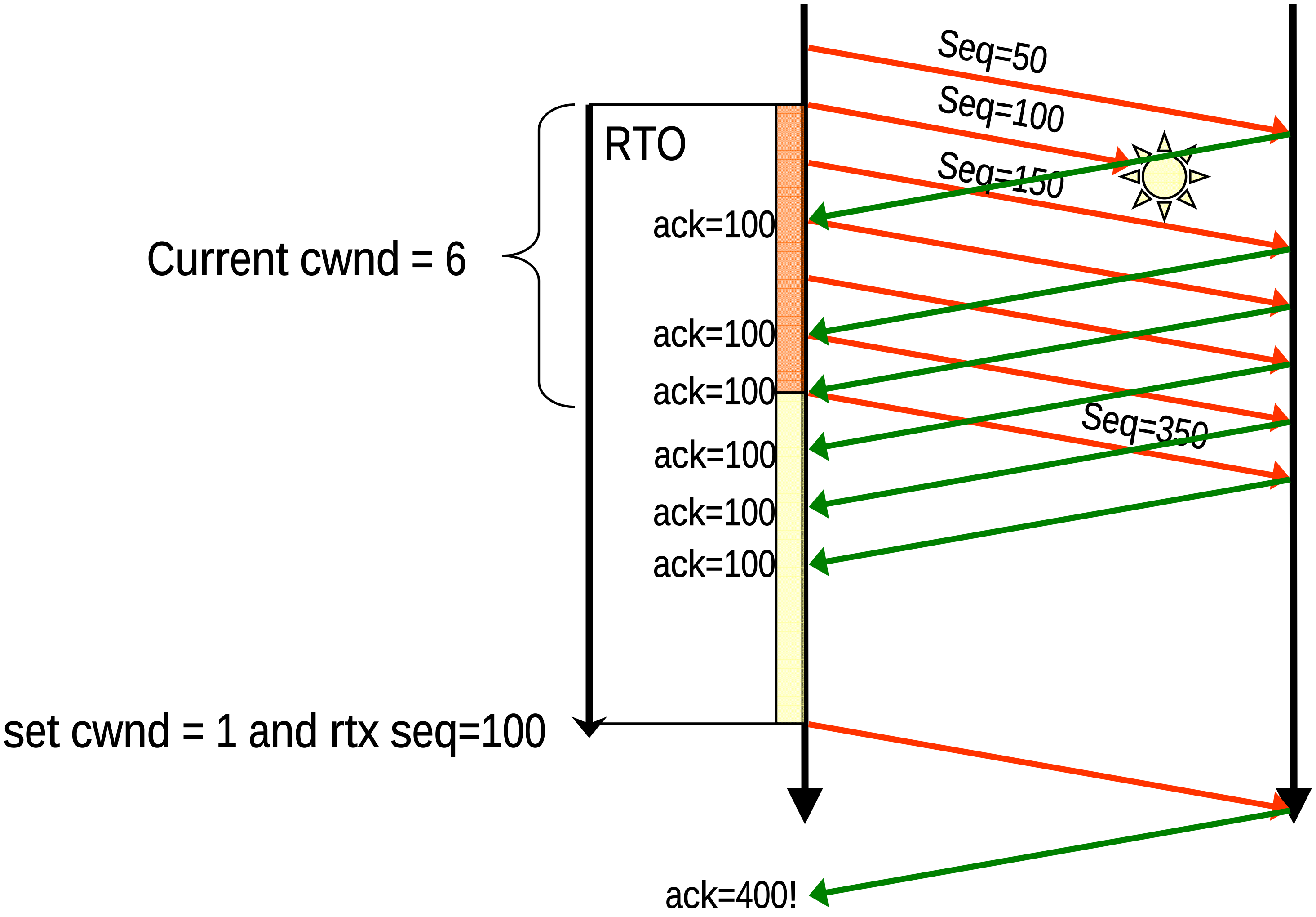


Simplified example (overall)



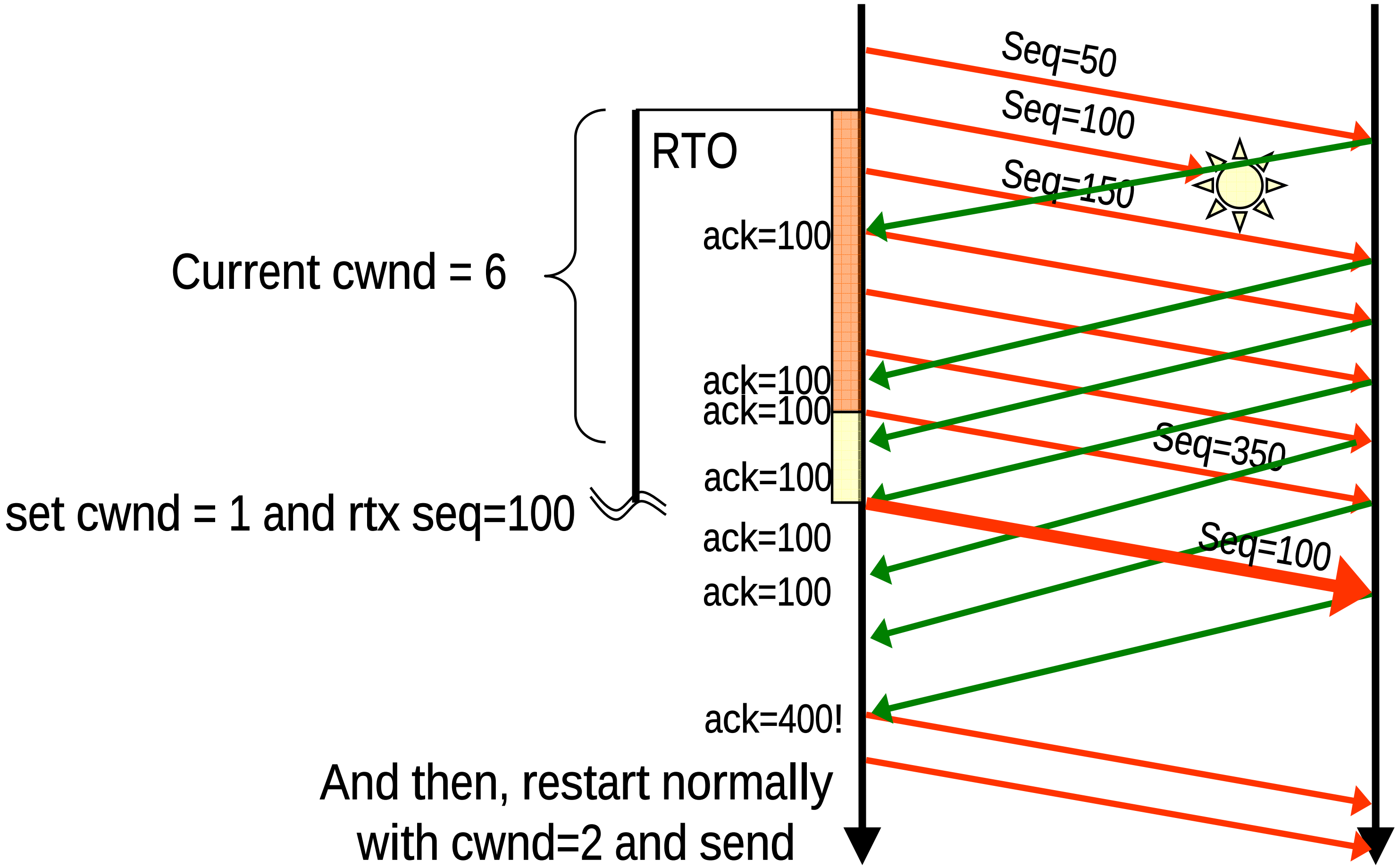
G.Bianchi, G.Neglia, V.Mancuso

What happens AFTER RTO?
(without fast retransmit)



And then, restart normally with cwnd=2 and send seq=400,450

TCP TAHOE
(with fast retransmit)



And then, restart normally
with cwnd=2 and send
seq=400,450

Same as before, but shorter time to recover packet loss!

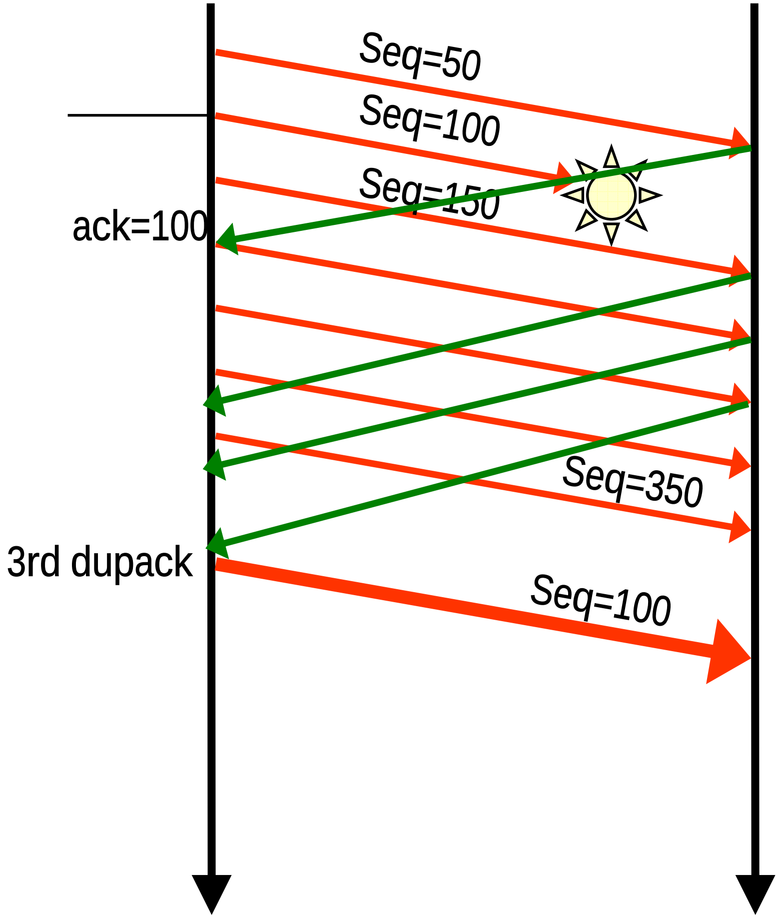
Motivations for fast recovery

FAST RECOVERY:

- ⇨ The phase following fast retransmit (3 duplicate acks received)

⇨ TAHOE approach: slow start, to protect network after congestion

⇨ However, since subsequent acks have been received, no hard congestion situation should be present in the network: slow start is a too conservative restart!

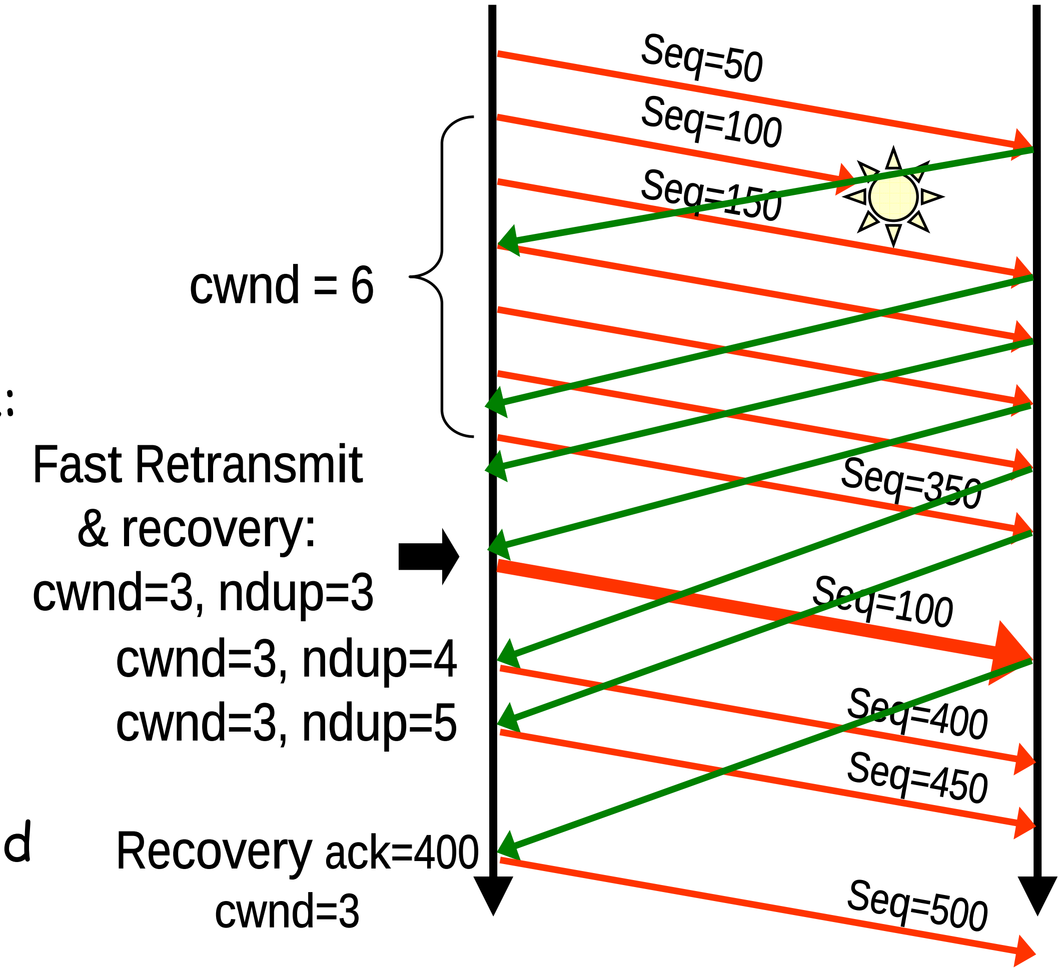


Università degli Studi di Palermo

Fast recovery rules

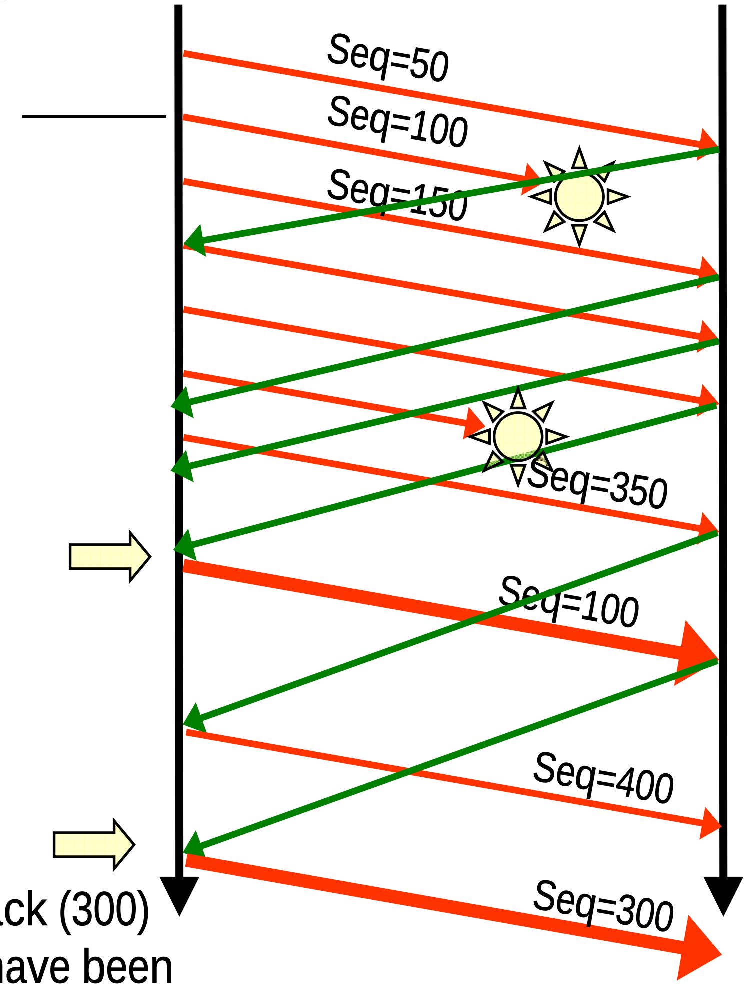
FAST RECOVERY RULES:

- ⇨ Retransmit lost segment
- ⇨ **Set cwnd = ssthresh = cwnd/2**
- ⇨ **Restart with congestion avoidance (linear)**
- ⇨ start fast recovery phase:
  - ⇨ Set counter for duplicate packets ndup=3
  - ⇨ Use "inflated" window: w = cwnd+ndup
  - ⇨ Upon new dup_acks, increase ndup, not cwnd (and send new data)
  - ⇨ Upon recovery ack, "deflate" window setting ndup=0



What about multiple losses?

- ➔ **TCP Reno optimized for single loss**
- ➔ **Performance drawbacks with multiple losses in same window**
- ➔ **Improvement: NewReno**
  - ⇨ Distinguish recovery ack from partial ack
    - →Equal to the recovery ack, but does not recover for all ndup segments
  - ⇨ Does not exit fast recovery when partial ack received
  - ⇨ Retransmit segment immediately following partial ack, assuming it was lost

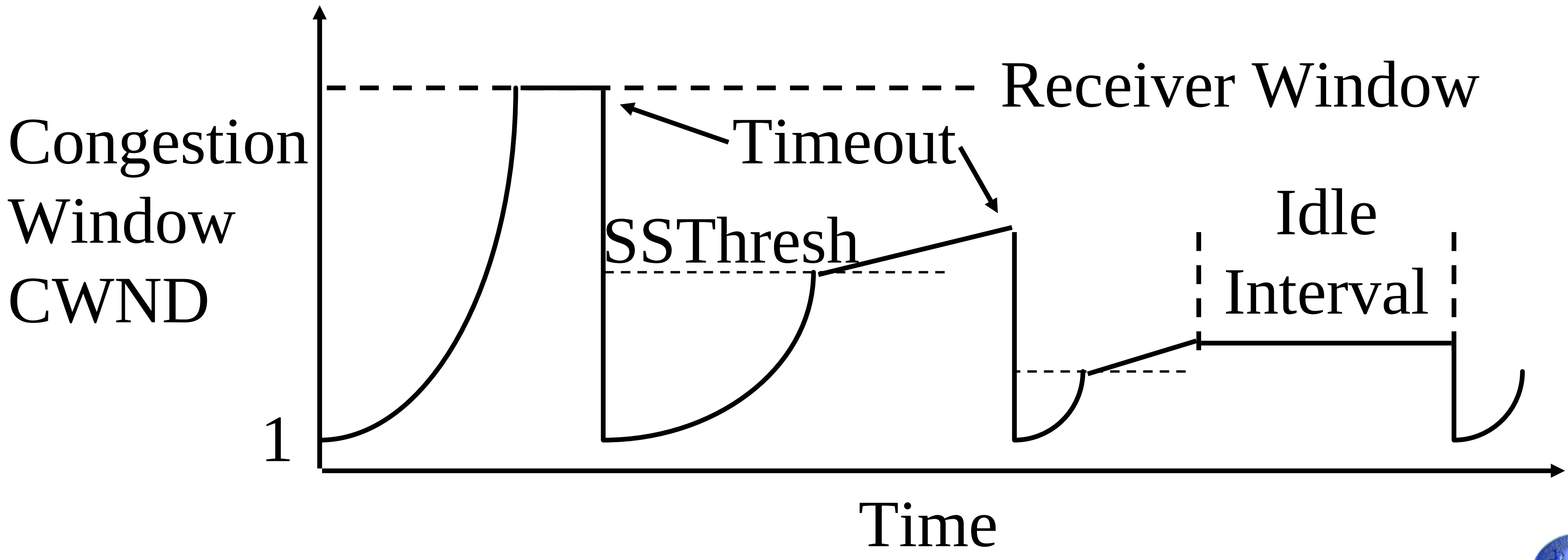


Partial ack (300)
Would have been
recovery ack if ack=400

Università degli Studi di Palermo

Idle periods

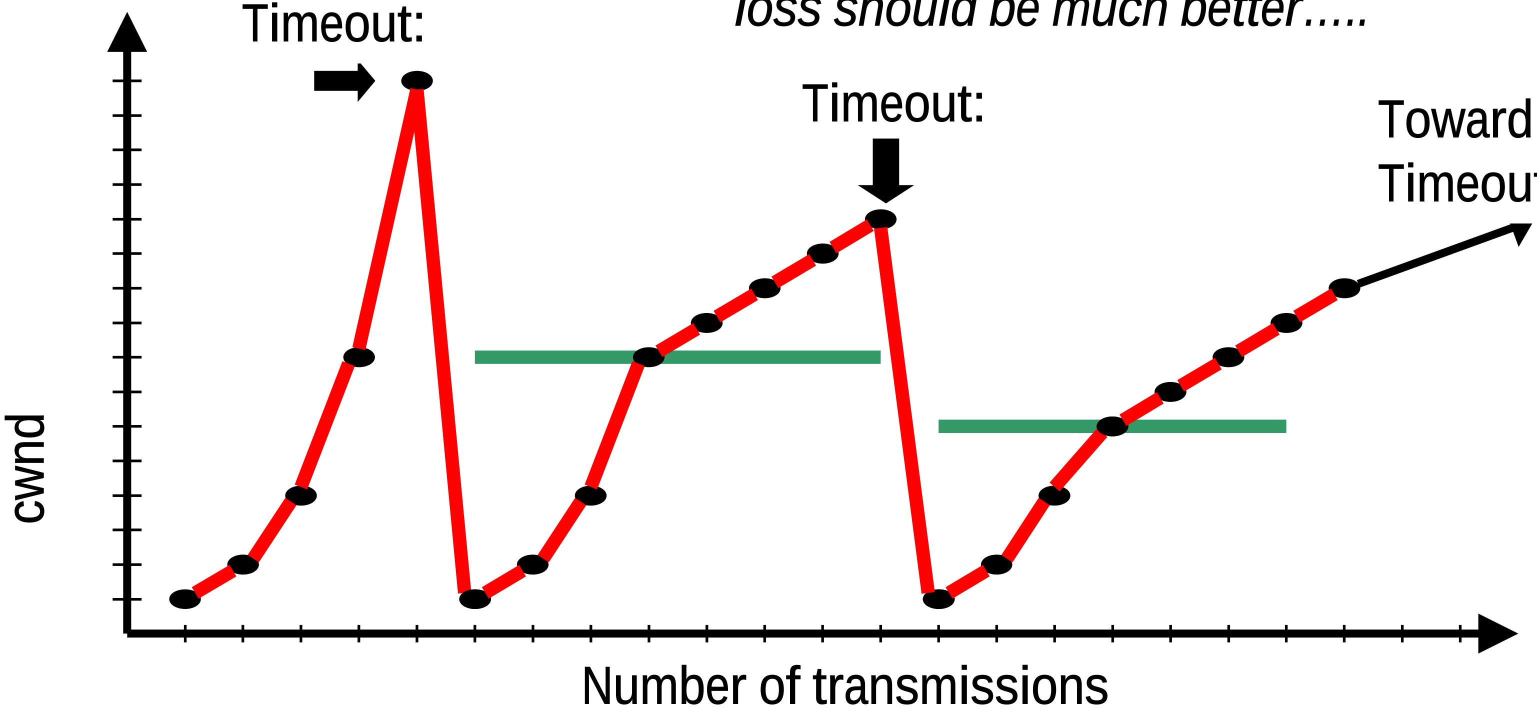
➔**After a long idle period (exceeding one RTO), reset the congestion window to one.**



Further TCP issues

Timeout = packet loss occurrence in an internal network router
TCP (both Tahoe & Reno) does not AVOID packet loss
Simply REACTS to packet loss

CONCLUSION: a TCP able to AVOID packet loss should be much better…..



G.Bianchi, G.Neglia, V.Mancuso

Università degli Studi di Palermo

TCP Vegas (1995)

➔ **Avoids packet loss by predicting it!**
- ⇨ Approach: monitor RTT
- ⇨ when RTT shows increase, deduce that congestion is going to occur
- ⇨ and thus preventively reduce cwnd
- ⇨ but not down to as low as slow start

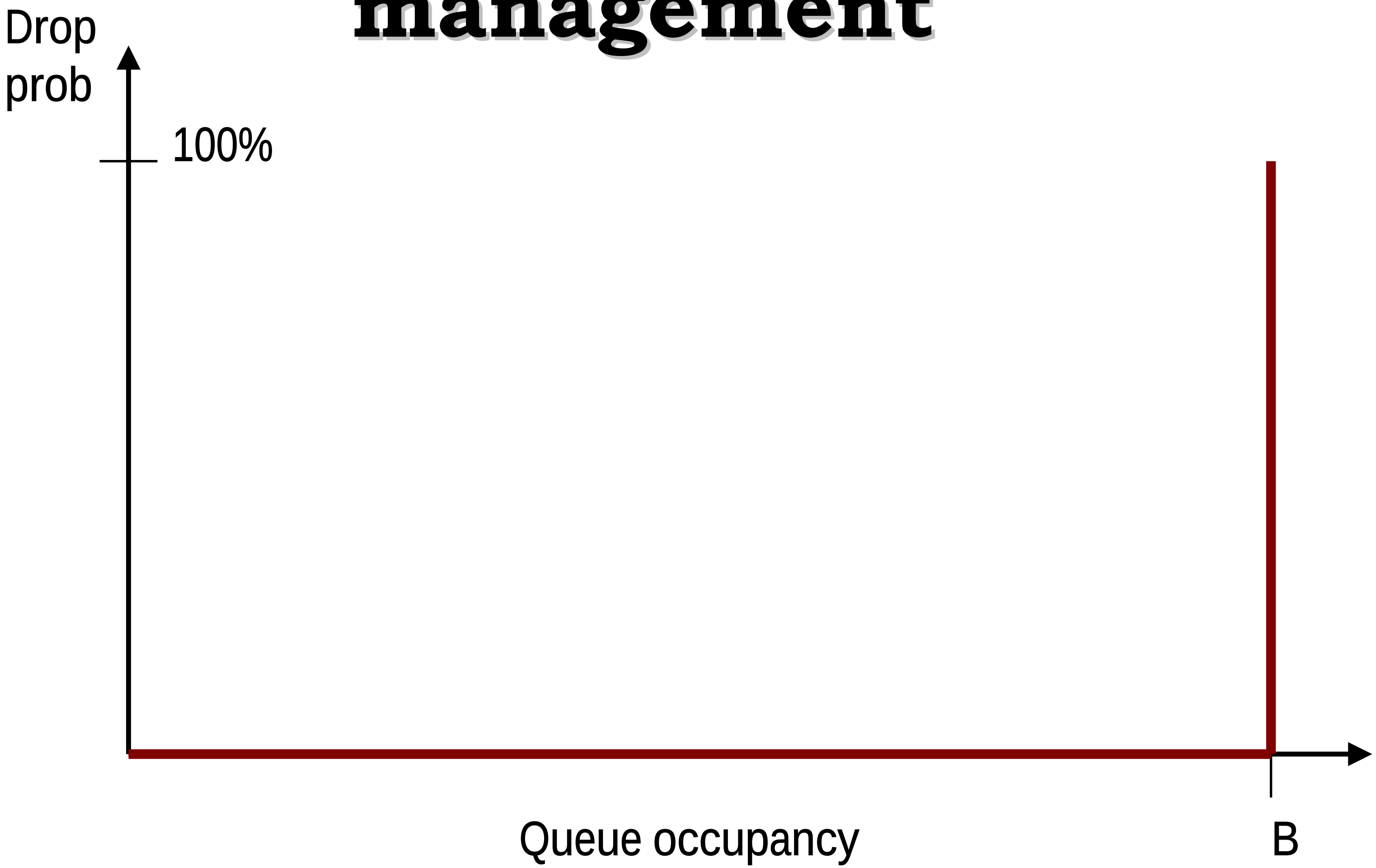
➔ **A problem: DOES NOT WORK WHEN OTHER TERMINALS USE TAHOE/RENO!!!!**
- ⇨ Vegas reduces rate to avoid congestion
- ⇨ while Tahoe/Reno grab the available bandwidth!!

A typical problem in Internet Protocol design: need to live with legacy apps and protoc

Recent Trends in congestion control

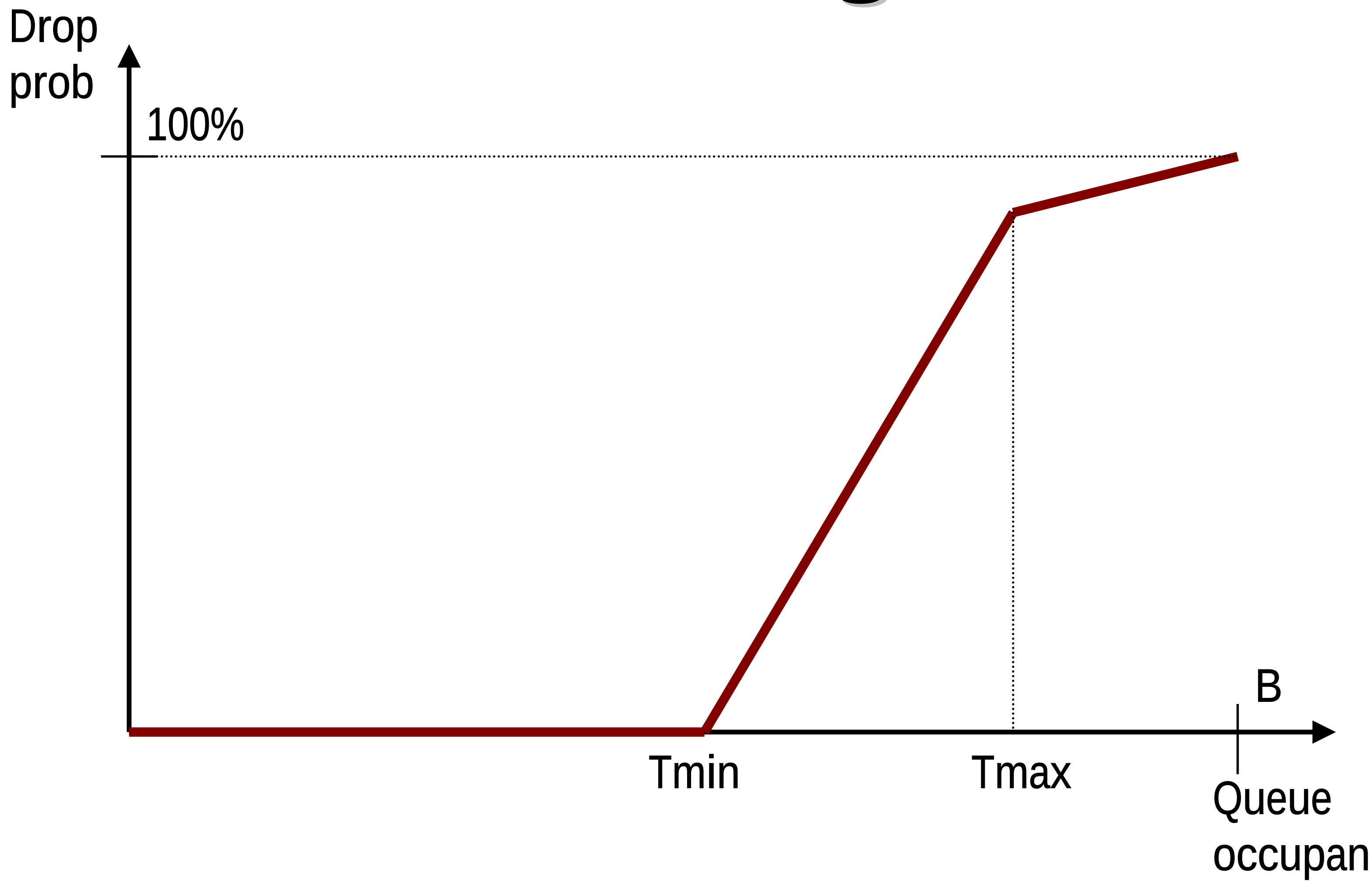
- ➔**End to end TCP congestion control not sufficient!**
- ➔**Active Queue Management (1994, 1998+)**
  - ⇨RED queueing discipline

Standard (Drop-tail) buffer management



Università degli Studi di Palermo

RED buffer management



Università degli Studi di Palermo

Fairness with UDP traffic

➔A serious problem for TCP

- ⇨in heavy network load, TCP reduces transmission rate. Non congestion-controlled traffic does not.
- ⇨Result: in link overload, TCP throughput vanishes!

Mixing TCP & UDP traffic