UiO : Faculty of Mathematics and Natural Sciences
University of Oslo

Department of Informatics
Networks and Distributed Systems (ND) group

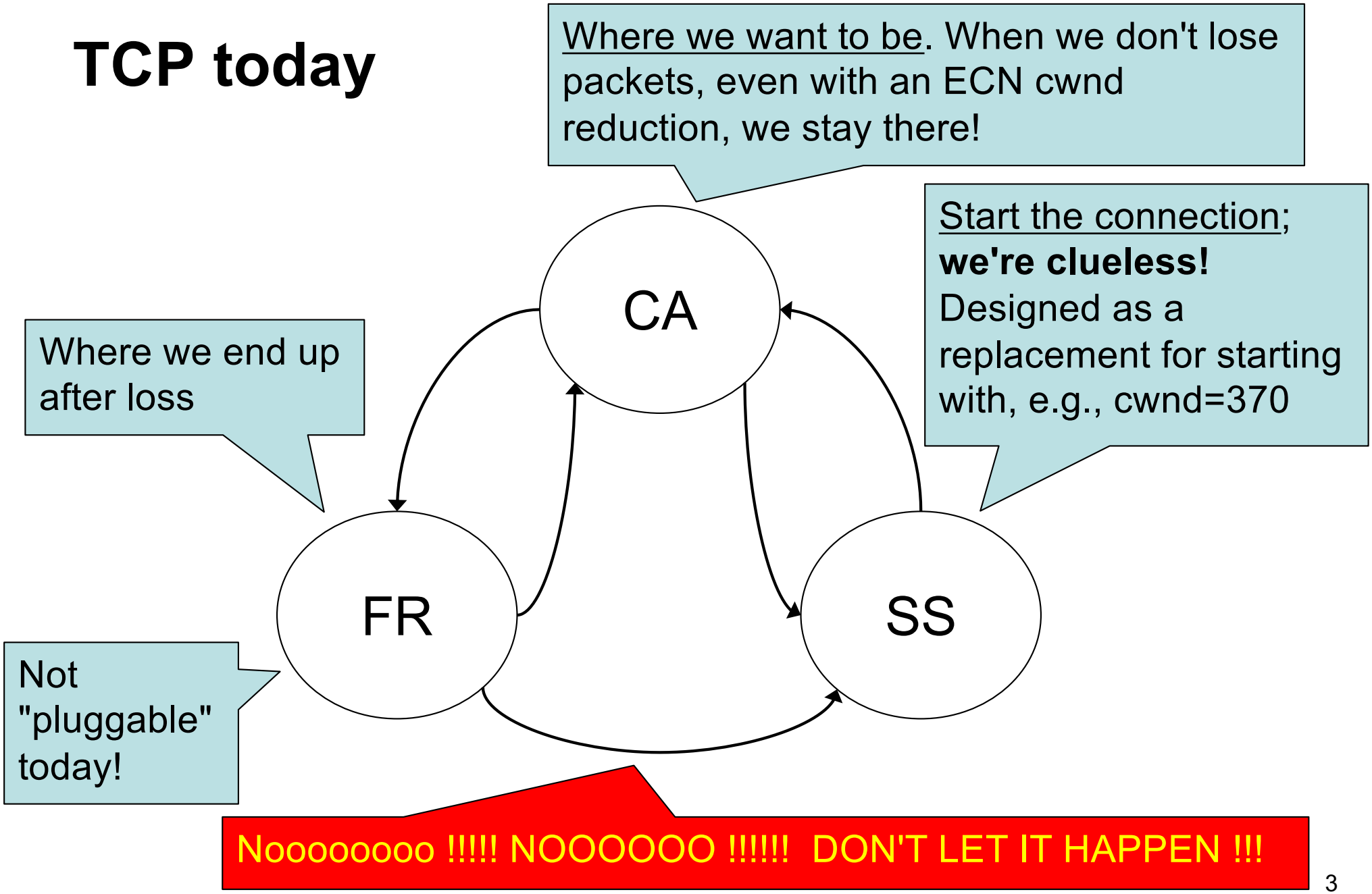
Timer-Based TCP (TBTCP)

Ongoing work with Giuseppe Bianchi and Safiqul Islam

Michael Welzl
INW 2020, Cavalese
31. 1. 2020

Why TBTCP? What's wrong today?

TCP today


Where we want to be. When we don't lose packets, even with an ECN cwnd reduction, we stay there!
Start the connection; **we're clueless!** Designed as a replacement for starting with, e.g., cwnd=370
Where we end up after loss
CA
FR
SS
Not "pluggable" today!
Noooooooo !!!!! NOOOOOO !!!!!! DON'T LET IT HAPPEN !!!

Two major operating states

- Good state: ACKs arrive
  - ACK clocking: send a packet when a packet has left the network
- Bad state: back to cwnd=1 + SS ... must prevent at all cost!
  - Entered when RTO timer fires; hence **RTO is VERY conservative**

- Quote from draft-dukkipati-tcpm-tcp-loss-probe-01.txt:
  "To get a sense of just how long the RTOs are in relation to connection RTTs, following is the distribution of RTO/RTT values on Google Web servers. [percentile, RTO/RTT]: [50th percentile, 4.3]; [75th percentile, 11.3]; [90th percentile, 28.9]; [95th percentile, 53.9]; [99th percentile, 214]."

- ACK clocking has made FR very complicated
  - Could simplify it with a faster timer + less drastic interpretation
  - This idea inspired RACK

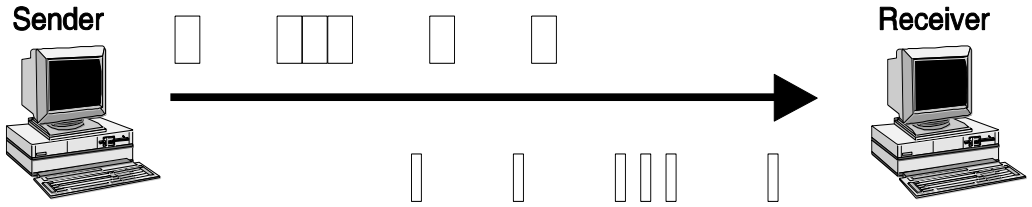
No point being religious about "ACK clocking"

- Q: Is a bursty ACK-clocked TCP better, or a paced non-ACK clocked TCP?
- CA: cwnd+=1 breaks ACK clocking
- SS: breaks ACK clocking; IW in particular!
- FR:
  - TLP/RACK slightly deviate from ACK clocking
  - ACK clocking tries to keep #packets in flight ("pipe") constant; these are really "in flight"+"in queue"
    - Can keep the queue filled
  - Complex, and issues: e.g., can't solve drops of retransmits (going to SS has been called "a feature, not a bug")
  - RACK can handle this... but won't reduce cwnd again

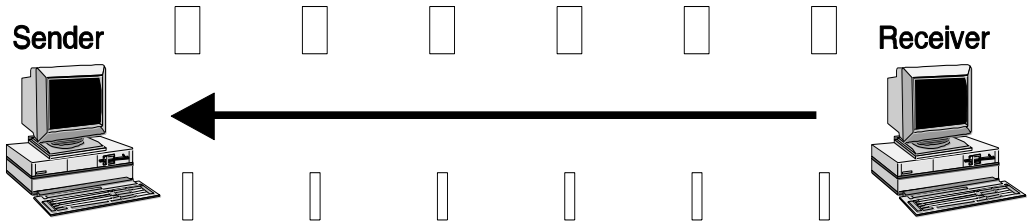
TBTCP

Everything based on time

- **Pacing is generally, usually good**
  - Avoids queue overflows from bursts
- Fast timers are now affordable:
  - **Software:** "Carousel" paper from Google @ SIGCOMM 2017
  - **Hardware:** Salvatore Pontarelli, Giuseppe Bianchi, Michael Welzl: "A Programmable Hardware Calendar for High Resolution Pacing", 2018 IEEE International Conference on High Performance Switching and Routing (HPSR), 17-20 June 2018, Bucharest, Romania.



(i) without pacing



(ii) with pacing

Key algorithmic aspects

(historical order)

Finished; used in:
Giuseppe Bianchi, Michael Welzl, Angelo Tulumello, Francesco Gringoli, Giacomo Belocchi, Marco Faltelli, Salvatore Pontarelli: "XTRA: Towards Portable Transport Layer Functions", IEEE Transactions on Network and Service Management, 10/19.
Also: demo @ SIGCOMM 2018

1. FR
   - MUCH simpler!
   - Ignores DupACKs
   - If SACK is enabled, they are parsed; this only avoids unnecessary retransmissions)
     - Not much performance difference
   - Could obviously optimize! Point was to see how good a very simple timer-based algorithm can be

Design 90% done... just beginning to evaluate

2. SS and CA
   - Interpolated SS and CA curves instead of ACK-driven
   - General algorithm with pluggable increase equation

FR algorithm (roughly)

- Send new segment → arm timer with 2×RTT

Everywhere: most recent sample

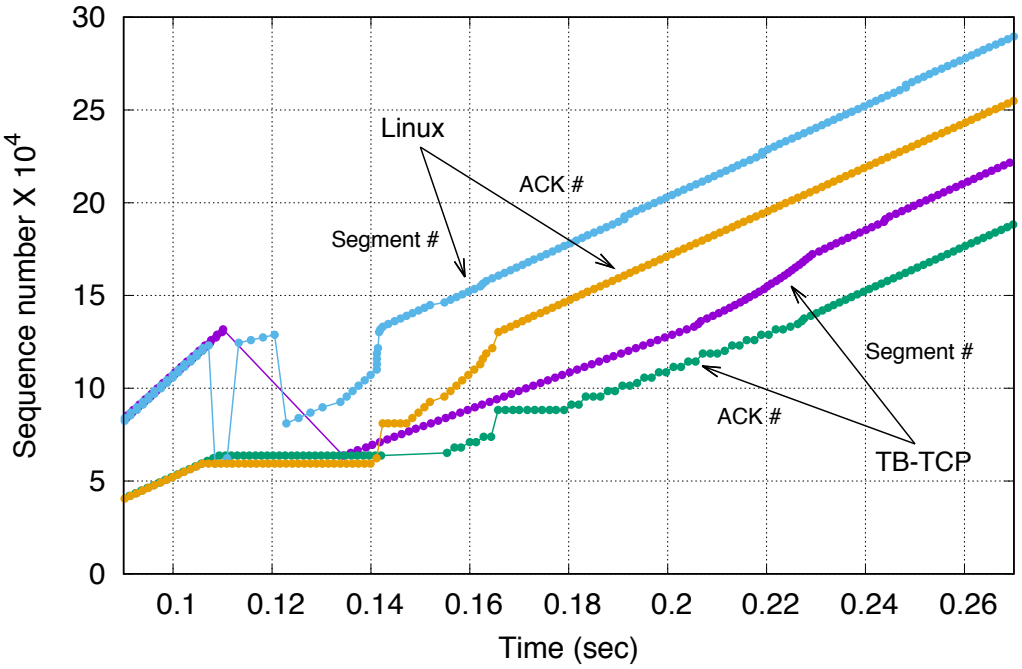
Similar to RACK's TLP timer

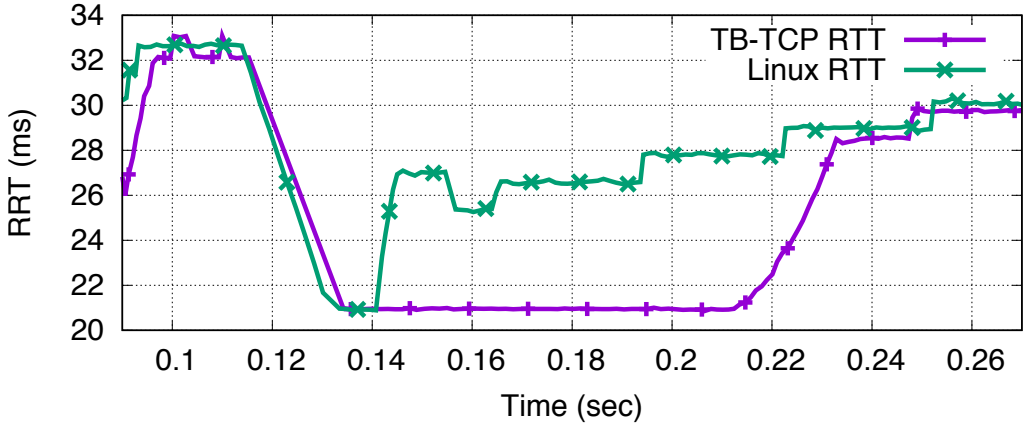
- Timer fires → FR begins
  - Reduce ssthresh following the congestion control algorithm (typically ½), set cwnd = ssthresh
  - Calculate min. pacing delay as RTT/ssthresh
  - For each additional firing timer, retransmit a segment, limit rate to pacing rate
  - Optional: using a local "round" counter, back off again when a retransmit is lost (real "exponential backoff")

- Generally, whenever an ACK arrives, cancel any outstanding timers of segments that are ACKed
  - In FR, a “full ACK” (i.e., covering all the data that was outstanding when loss was detected) ends FR

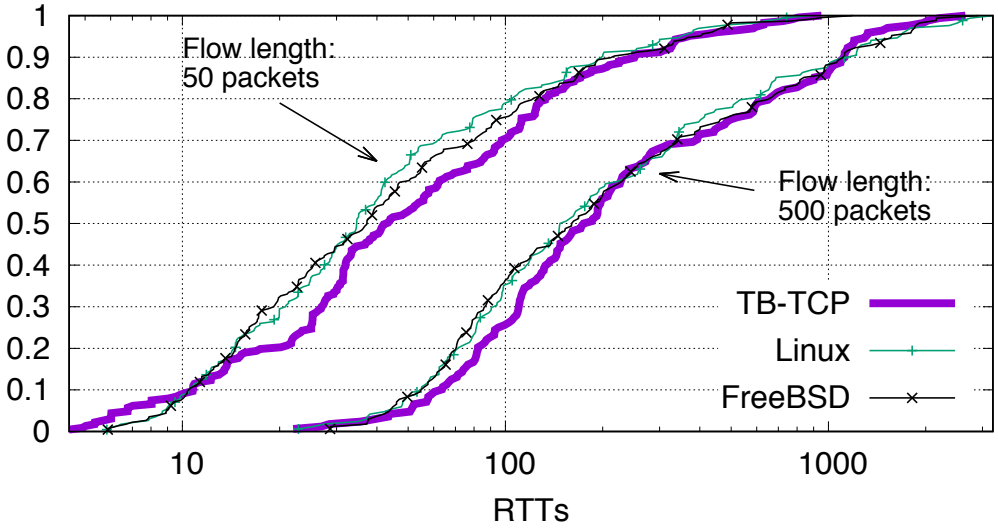
How this looks... ...and how it performs

"Hacked" ns-3 TCP, used as traffic generator talking to ncat receiver in a testbed





CDF of FCT of the slowest of 6 parallel flows, starting at the same time



~ 400 experiments with many different capacities, RTTs, queue lengths

Side note: do not trust ns-3's native TCP!

Fast Recovery is either broken or very inefficiently implemented (FreeBSD and Linux are very different)
(case in point for FR complexity!)

Also, please make sure that you don't have an endless queue, and that cwnd really reaches the right limit

General algorithm for paced increase

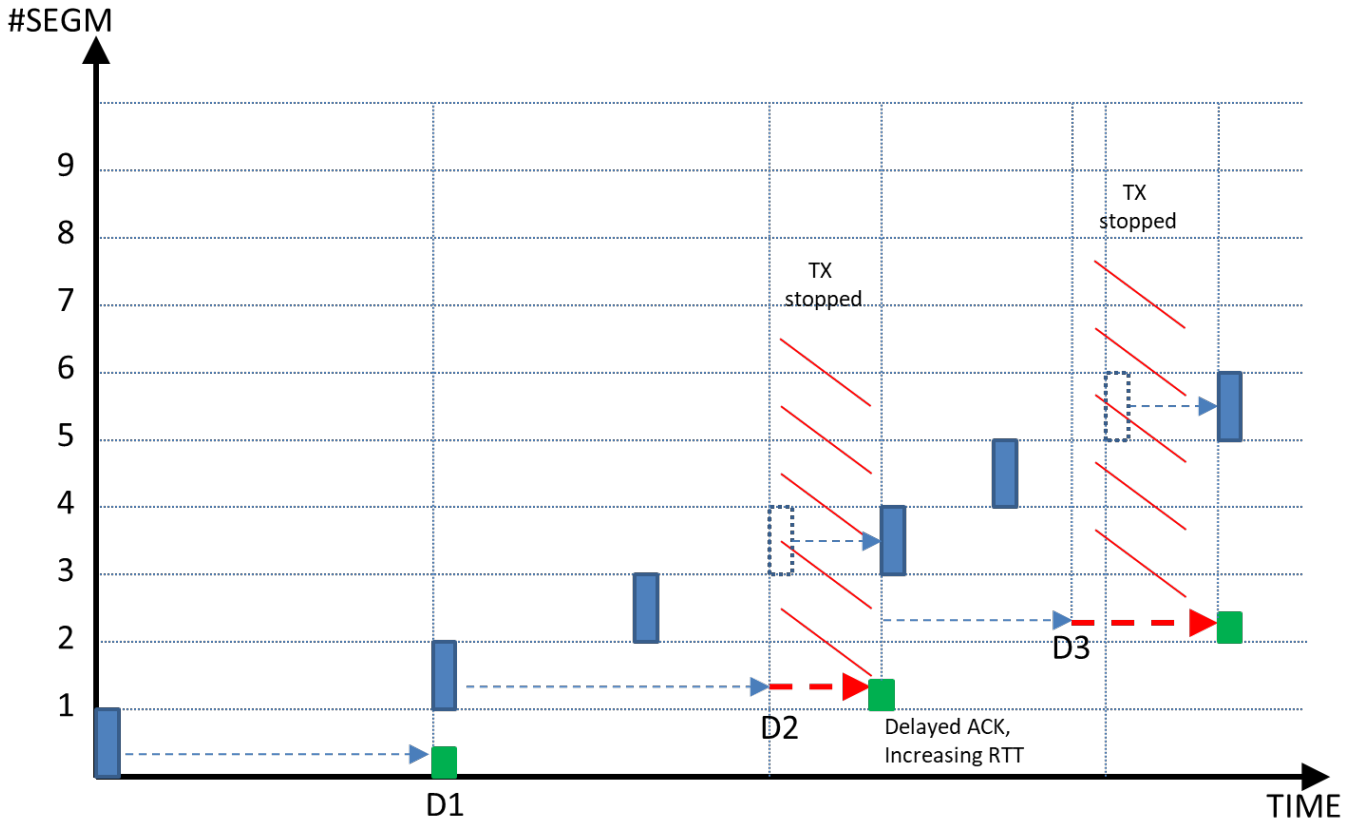
High-level overview

- Increase function F(RTT, N) (N=packet number) yields dt (time until next scheduled event)
  - E.g. **SS:** $dt = RTT \log_2 (1+1/N)$

- Usage of F
  - when transmitting, to calculate the next transmission
  - Initially, to calculate expected ACK arrival
  - Upon ACK arrival, to update next expected arrival time
  - Make a decision based on difference: real arrival time – expected arrival time

Use real arrival time; also: update RTT!

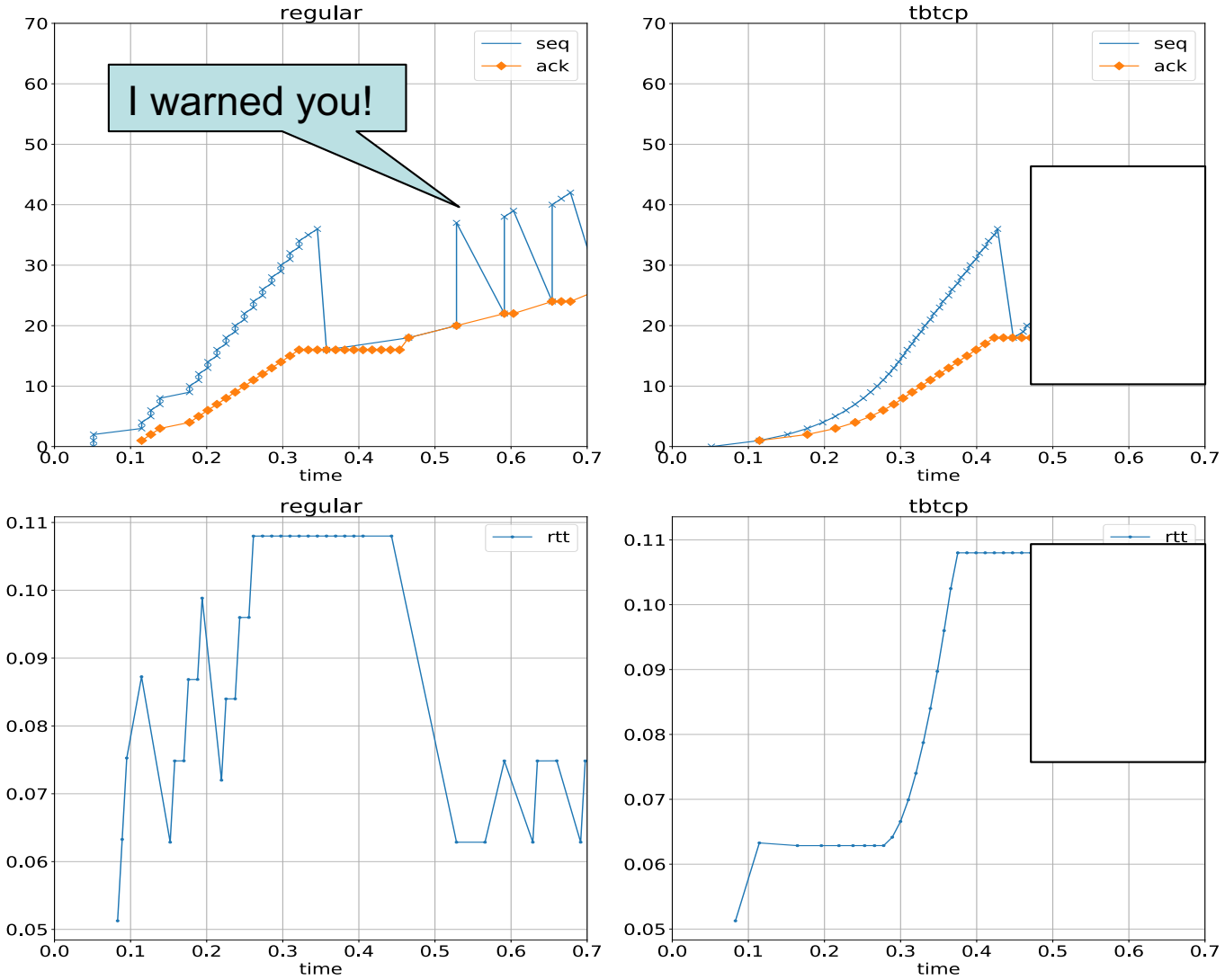
SS example



- In SS, could also hard code: always wait for an ACK, this allows exactly two data packets
  - But this approach is more general!
  - Different increase? Just plug in a different equation

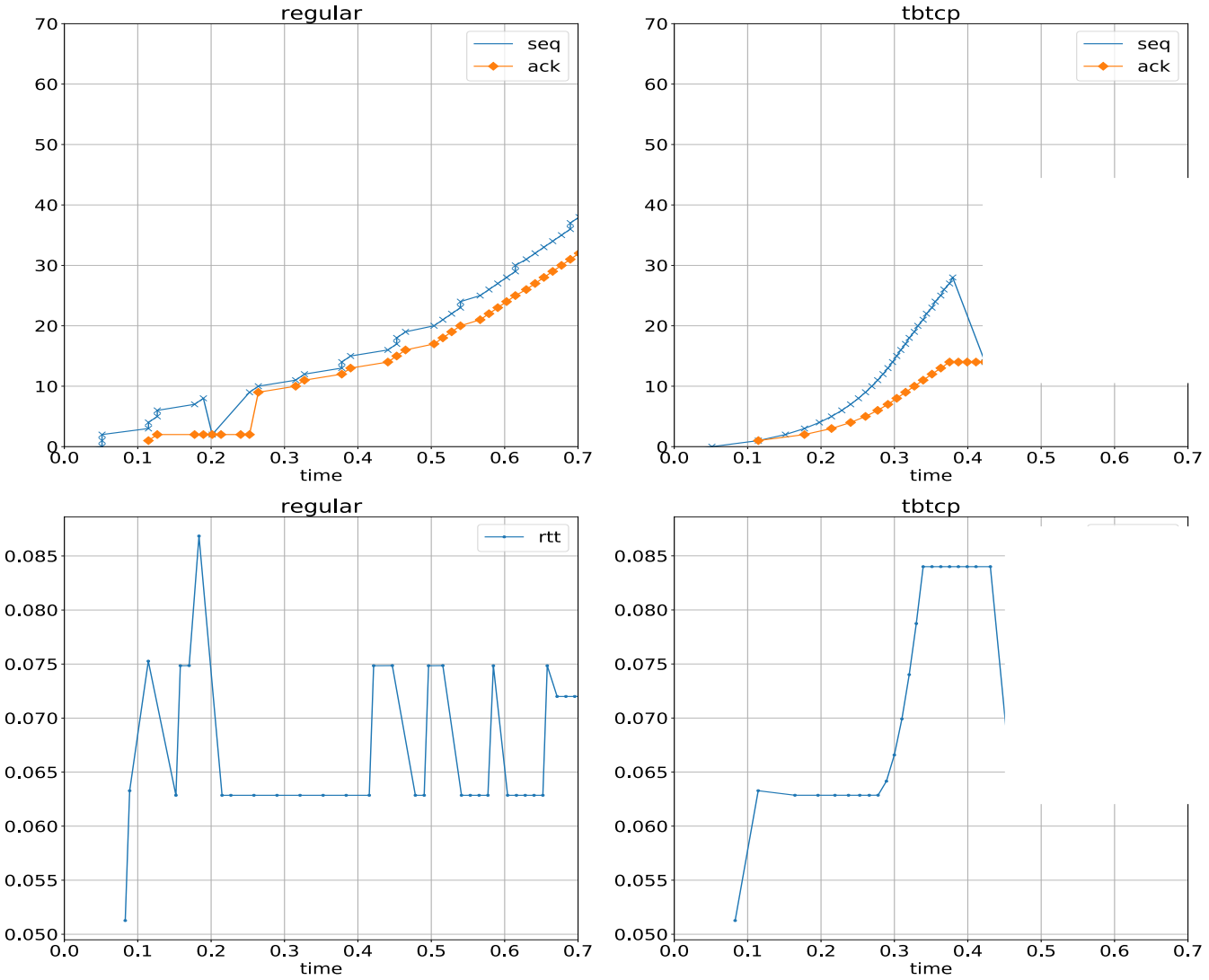
SS example in (ns-3) action

c=1Mb, RTT=50ms, qlen= 4 pkts (a BDP)



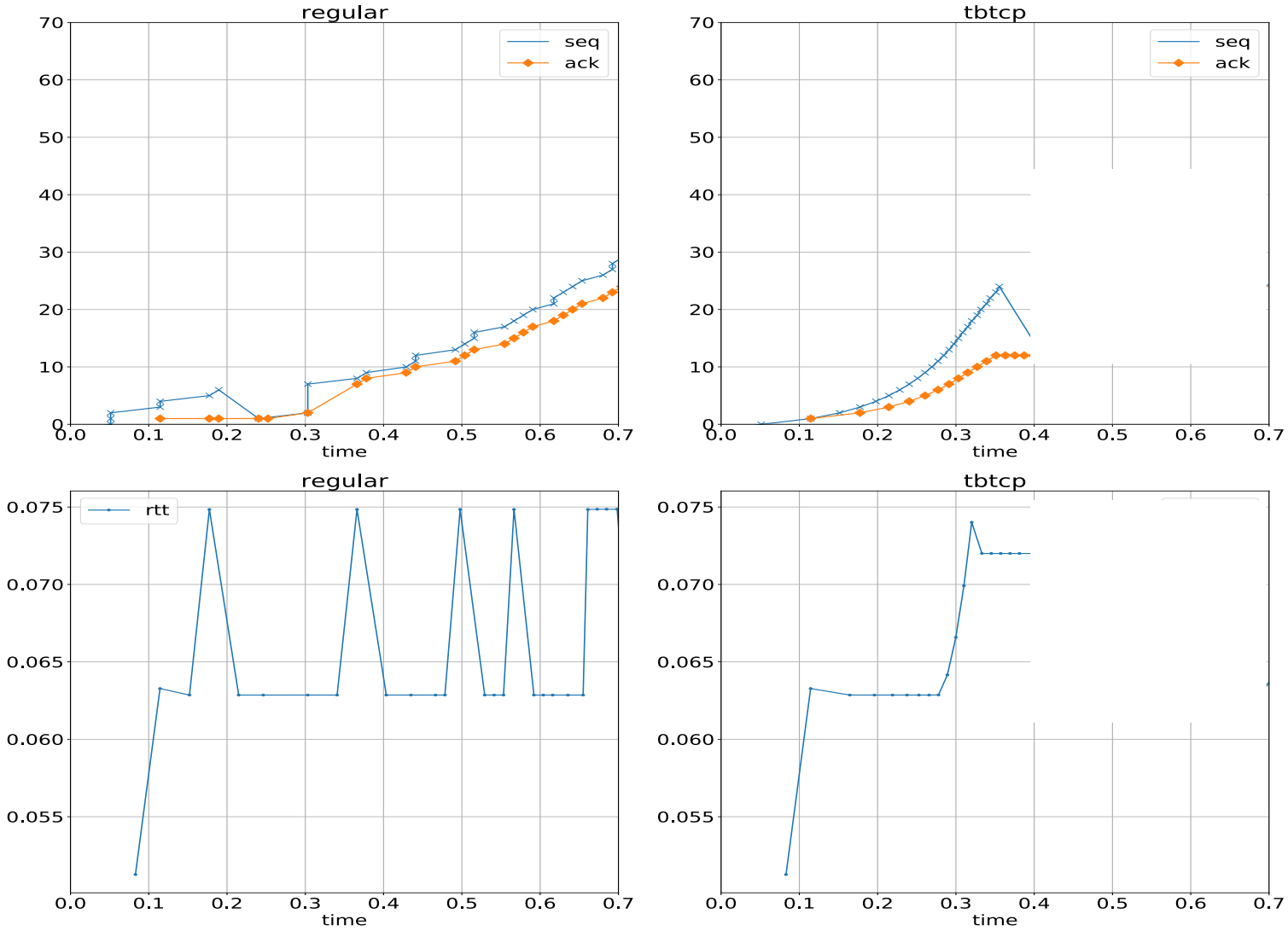
SS example in (ns-3) action /3

c=1Mb, RTT=50ms, qlen= 2 pkts (1/2 BDP)



SS example in (ns-3) action /2

c=1Mb, RTT=50ms, qlen= 1 pkts (1/4 BDP)



Conclusion

- Purely timer-based TCP
  - Result is more simple and more general
  - Now working on a first complete (ns-3) implementation + tests
  - Can we make it always outperform normal TCP? (needs some extras)
  - Else, what are good use cases?

Thank you!