Flow-start: Faster and Less Overshoot with Paced Chirps

Joakim Misund, Simula
Bob Briscoe, Independent
Mar 2018
Problem: originally L4S/ECN-specific
Solution: delay-based

• Original problem
  – DCTCP's ECN marking prob. higher\(^1\) than Classic drop
  – A flow pushing into existing traffic exits slow start earlier

• Solution
  – TCP Prague intended for public Internet
  – Unlike DCTCP, cannot assume ECN support at bottleneck
    • even if new flow experiences ECN marking
    • as available capacity increases, could reveal a non-ECN b'neck
  – Must use delay-sensing, with ECN only to improve precision

\(^1\) Deliberate: 2 marks per RTT in steady state (control scales to any rate)
Problem: DCTCP slow start

- Throughput convergence
  - awful

- Queuing Impact
  - excellent
Solution: Paced Chirping

- Throughput convergence
  - excellent

- Queuing Impact
  - excellent

- Implemented
  - kernel module and modified the API to pacing in linux kernel
  - still only one outstanding timer per connection

- But not tested to extremes
Approach (1/3)

- **Packet chirps**
  - continually pulse queue by a few packets, then relax

- **Samples available (and max) capacity**
  - available: rate where ACKs spread from sent pattern (after filtering noise within chirp)
  - max: ACK rate of last 2 packets

- **Maximizes ratio of capacity-information-rate to harm (queue delay)**
  - run each (per chirp) measurement through EWMA
Using chirps to measure available capacity

- This example measures constant available capacity
- Code to interpret chirps filters noise to measure varying available capacity
Approach (2/3): paced chirps

- Avg rate of each chirp depends on EWMA of available capacity measured in previous rounds
- Noisy, but increasingly frequent measurements
- Queue delay solely depends on chirp geometry
- Notice, chirp length reduces
  - as available capacity measured in last round increases
Approach (3/3): adaptive gain

- Growth in #chirps per RTT depends on a gain variable
  - vary gain dependent on stability of available capacity samples
- Push-in a little harder than available capacity grows:
  - other flows yield
  - activity-triggered link scheduler expands per-user capacity
- When shift from paced chirps to ACK clocking?
  - when chirps fill the round trip
  - or …? (to be determined, perhaps using ECN for extra precision?)

Still, queue delay solely depends on chirp geometry, not gain.
Approach (3/3): adaptive gain

- Growth in #chirps per RTT depends on a gain variable
  - vary gain dependent on stability of available capacity samples
- Push-in a little harder than available capacity grows:
  - other flows yield
  - activity-triggered link scheduler expands per-user capacity
- When shift from paced chirps to ACK clocking?
  - when chirps fill the round trip
  - or …? (to be determined, perhaps using ECN for extra precision?)

Still, queue delay solely depends on chirp geometry, not gain.
Caveats

• Delayed ACKs & ACK thinning
  • we need rcvr to suppress delayed ACKs during SS
  • Linux rcvr quickacks during SS - detected heuristically
    - but our modified start-up phase confuses its heuristics
  • sender could request quickacks with a 1-bit option
    - Could put arrival times in ACKs (as in QUIC)

• Bursty MACs and schedulers
  • Our initial experiments are with simple Ethernet
  • Shared upstream links (LTE, DOCISIS, GPON) all time-slotted
  • Averaging within chirps designed to cope, but may need tweaking
  • EWMA designed to cope with numerous noisy measurements
Closing the loop

- Paced chirps: not just for slow-start
  - for whenever the closed loop signal is lost
- With Scalable CC, e.g. TCP Prague, DCTCP, etc.
  - after 1 round trip without marks
  - start paced chirps to rapidly find new operating point

Takes a lot longer with an unscalable CC
- e.g. 500-1000 round trips for Cubic at 800Mb/s
Why two phases?

1) Start-up:
   • paced chirping

2) Steady state:
   • regular congestion avoidance, preferably scalable (e.g. DCTCP, TCP Prague, Relentless, Scalable TCP)

• Benefit of not chirping in steady state
  • using ACK clock reduces timer burden on servers (large majority of packets are sent in steady state)
  • cuts out noise (each chirp is a signal for the sender, but noise for other flows)
Further Work Needed

**Research**
- Termination condition – when to stop pushing in
- Improving noise filtering & precision of chirps
  - esp. for bursty MACs: LTE/5G, DOCSIS, GPON
- Exploiting ECN if available
- Initial avg. gap for a wide range of possible networks
- Evaluation over much wider range of conditions & iterate design
  - much lower/higher BDP, hi as well as lo stat. mux. bottlenecks, etc.

**Engineering**
- handling loss, reordering during slow start
- TFO when RTT estimate is stale in the first RTT
- mimic QUIC’s ACKS listing arrival times in other protocols
Summary

- TCP slow-start is mimicked in most transport protocols
  - an open loop phase characterized by arbitrary numbers
- Paced chirping
  - closes the open loop – frequent startup information
  - queue delay solely depends on geometry of each chirp, not pace of chirps
  - maximizes ratio: \( \frac{\text{capacity-information-rate}}{\text{harm}} \)
- Initial research
  - much more testing and development to do
Flow-start: Faster and Less Overshoot with Paced Chirps

Q&A and Spare Slides
Measuring Available Capacity using Chirps

• Find inter-packet gap where path delay starts to persistently increase

  1) Record each inter-packet path delay increase

      \[ \Delta q_n = q_n - q_{n-1} \]

      where \( q = ts_{rcv} - ts_{snd} \); \( ts \) are timestamps; and \( n \) is the packet number

  2) Ideally one-way delay: timestamp each packet:
     • when sent (not when scheduled to send)
     • when received
       - in practice use when ACK rec'd (round trip delay)

  3) Filter out noise. Simple example filter:
     • only count an increasing trend of more than \( L \) packets
to count as an increase

      \[ \Delta q \geq \frac{\max_{i=1}^n (\Delta q_i)}{F} \]

     • default: \( L=5, \ F=1.5 \)
Linux Pacing Framework

```
struct tcp_sock
   pacing_timer
   used_pacing_rate_list
   pacing_rate_list

1. Use rate if avail.
2. Put used rate

tcp_internal_pacing

tcp_pacing_check
```

modifications in red
Linux kernel
Structure to set up per-packet rates

tcp_internal_pacing
1. Check if available rate
2. Use rate, put time and snd_nxt
3. Move entry to used-list

tcp_sock

used_pacing_rate_list

pacing_rate_list

rate
seq
timestamp
rate
seq
timestamp
rate
seq
timestamp
rate
seq
timestamp
rate
seq
timestamp