# Paced Chirping: Rethinking TCP start-up

Joakim S. Misund
*University of Oslo,*
*Department of Informatics*

Bob Briscoe
*Independent*

## Abstract

Paced Chirping is a replacement of slow start that continuously pulses the network with groups of packets called 'chirps'. The decreasing inter-packet gap of the chirps allows the sender to estimate the available capacity from queueing delay measurements. Each chirp creates a small measurable queue, and the queue is allowed to drain in-between subsequent chirps. Using the capacity estimates Paced Chirping transition to congestion avoidance without overshooting capacity.

Paced Chirping is implemented in Linux using the kernels pacing framework to realise chirps. We have added code to the pacing framework that allows a congestion control module to create chirps with desired characteristics. This does allow for more elaborate start-up schemes, made necessary by increasing capacity and the need for lower latency. Pacing precision is a challenge at high speed due to timing precision. With the proposed NIC API change essayed by Van Jacobson at netdev 0x12 precision can be improved significantly.

Paced Chirping achieves fast acceleration with extremely low maximum queueing delay (couple of milliseconds), and it is a promising step towards a start-up algorithm that can reach utilization fast without hurting latency-sensitive applications. In the acceleration world, Cubic (without hystart) is best and DCTCP is worst. In the delay overshoot world Cubic is worst and DCTCP is best. Paced Chirping is as good as the best in both worlds. The best acceleration of today's schemes gives the worst overshoot and the best overshoot gives the worst acceleration. With paced chirping, the acceleration and the overshoot are as good as the best of both worlds.

## 1 Introduction

**TCP slow start** fumbles in the dark attempting to reach the available capacity of a network path. It is a heuristic the roughly doubles the amount of data it puts into the network each round-trip time until it exceeds the available capacity and overshoots the bottleneck queue causing high latency and multiple congestion events. This affects not only the flow itself but all the flows sharing the bottleneck.

TCP Slow start has to choose between acceleration speed and queueing delay. If it accelerated faster than exponential it causes a greater overshoot, but if it is slower it takes longer to reach full utilization. It seems impossible to solve this dilemma with TCP slow start, thus a new algorithm has to be made. We believe that Paced Chirping is a promising step towards a start-up algorithm that does not have to make a tradeoff between acceleration speed and queue impact.

When TCP detects the first congestion event it transitions to congestion avoidance. In congestion avoidance it is assumed that the current amount of packets is close to the available capacity of the network. However, if this is not the case it will usually take a long time to reach full utilization. TCPs start-up performance is thus fragile to early congestion events caused by transient congestion periods, e.g bursty cross traffic. Once a flow has entered congestion avoidance there is no way back. This means that a premature transition from TCP slow start can result in long convergence times with under-utilization. A new TCP flow can thus be unlucky during the start-up phase.

Although Paced Chirping is currently only implemented and tested for DCTCP it can replace slow start for all congestion controls. We have not yet made an attempt to structure the code so that it can be used by other congestion control modules. Each congestion control module needs its own implementation. It is challenging to make the implementation generic so that it can be an opt-in toggle.

**L4S** is an architecture that aims to keep queueing delay extremely low. Simply put it makes two changes; one to the network and one to the end system. In the network, queues are kept small by marking packets early with explicit congestion notification (ECN). The end-system (sender) reacts to the extent of marks rather than the presence of marks. The combination of the changes creates a network with extremely low queueing delay and high throughput.

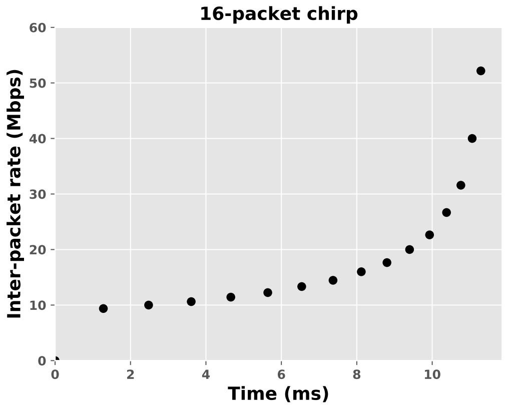


Figure 1: Packets in a chirp are sent with decreasing inter-packet interval (increasing inter-packet rate).

A consequence of having a low marking threshold is that the risk of exceeding the marking threshold prematurely increases. TCP Slow start usually exhibit bursty behaviour causing it to exceed the marking threshold early. This can result in under-utilization and long convergence time.

During congestion avoidance TCP gently probes for more capacity. Regular TCP uses additive increase. TCP CUBIC [2] uses a cubic function to make the probing faster in high BDP scenarios, but can be even slower to converge otherwise. The problem traditional TCP faces is uncertainty. It does not know if more capacity has become available or if it is in-between congestion events. This is because congestion notifications (loss, or ECN) are infrequent. In L4S congestion notifications are frequent and a flow can thus infer that more capacity has become available from lack of congestion notifications. This opens the possibility to use Paced Chirping in congestion avoidance when frequent and periodic ECN marks are absent. This can only be done for scalable congestion controls like DCTCP and TCP Prague because they react proportional to marks which keeps the marks coming frequently.

## 2 Paced Chirping

Paced Chirping attempts to identify the available capacity of a network path by gently probing the network continuously with groups of packets called chirps. In this section we will go through and explain the algorithm and its building blocks. Paced Chirping is also described in [5].

A *chirp* is a group of packets sent at an increasing rate. The increase in rate is relized by decreasing the inter-send gaps between the packets. Figure 1 shows the inter-packet rates of a chirp and how they relate to the inter-packet time gaps. Paced Chirping is insensitive to the accuracy in the inter-packet gaps as long as they are decreasing.

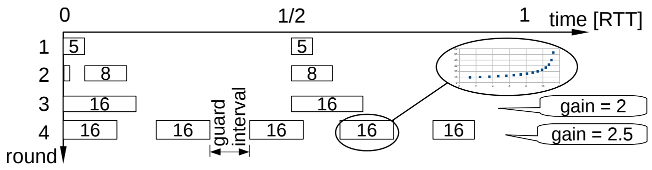


Figure 2: The arrangement of chirps in each round as a flow starts up).

The difference in queueing delay experienced by each packet in a chirp is analyzed to get an estimate of the available capacity. The analysis is the same as that of pathchirp [6]. It looks for increasing trends in queueing delay, called excursions, to estimate the available capacity. Paced Chirping maintains an exponential moving average of all the individual estimates. This moving average is used to calculate the average gap of later chirps.

Paced chirping uses multiple chirps spread over a round-trip-time with enough room in-between to allow the queue to relax. This is shown in Figure 2. The time between subsequent chirps is called a *guard interval*. Its value depends on the number of chirps in the current round, the round-trip-time and the current available capacity estimate. Critically, the guard interval allows the bottleneck queue to relax in-between chirps. This prevents persistent build-up of packets at the bottleneck queue. We have experimented with adapting the guard interval based on an EWMA of the variability of previous estimates. This way if the estimates are close to one another Paced Chirping can accelerate fast to the available capacity, and when the estimates are noisy it can be more cautious.

A key aspects of Paced Chirping is that it can probe for available capacity without committing to sending at the capacity it probes for. This allows it to increase the confidence in the estimate before it commits and hence reduce the chances of overshooting. It is made possible by the guard interval in-between chirps, which controls the average rate over a sequence of chirps. The guard interval can be adjusted according to the confidence in the current estimate.

The number of chirps in a round is controlled by a variable M and a gain called g. The initial value of M is 2 (2 chirps). After the third round M is set to $M * g$ each round. We have mostly used $g = 2$, but have experimented with smaller and bigger values. This value of g gives us an increase similar to exponential increase.

Paced Chirping terminates once it 'fills the pipe'; when the number of in-flight packets is sufficiently close to the estimated bandwidth delay-product. We currently have a legacy transition period from when we had a linked list and no direct interaction between the kernel and the congestion control module, but we believe that we can remove it with the latest code. Paced Chirping is also terminated on packet loss.

## 2.1 Fast acceleration and low latency

To see the benefit of using Paced Chirping instead of slow start we ran flow completion time experiments comparing Paced Chirping to DCTCP (slow start without hystart) and Cubic. Each experiment has 1000 flows with Pareto distributed flow sizes mimicking recent real-world measurements since the introduction of HTTP/2 [4]. Alpha and mean are set to 0.5 and 900 Bytes respectively, and the sizes are kept in the range [1KB, 5MB]. The inter-arrival times of the flows are exponentially distributed with various mean values configured, called intensity. The network is configured with a 15 ms RTT and a 100 Mb/s capacity. DCTCP and paced chirping are run with a 1 ms marking threshold, while Cubic is run with a tail-drop queue of 1 BDP. We vary the intensity to test performance under various loads. The experiment is run with and without a greedy background flow.

Figure 3 shows the flow completion time and empirical CDF of the queueing delay under the different conditions. Cubic achieves good FCT, but at the expense of a significant queue. DCTCP does not handle the low marking threshold well and exits slow start early which reduces the FCT for longer flows. On the other hand the queuing delay is excellent. Flows that use paced chirping finish as fast as Cubic but with queueing delay nearly as low as DCTCP; far lower than Cubic. Paced chirping does not make the trade-offs that Cubic and DCTCP have to make.

## 3 When to Trigger Paced Chirping

It is obvious that Paced Chirping should be triggered at start-up and re-start after RTO or idle. Scalable congestion controls opens the possibility of triggering Paced Chirping during congestion avoidance.

Traditional TCP variants needs to have a very low congestion event rate to keep utilization high because of its conservative reaction to congestion events. A consequence of this is that congestion events have to happen very infrequent. When congestion events are infrequent it is difficult to know if one is in-between congestion events or if more capacity has become available. TCP CUBIC uses time as an indication for how aggressive it can be. As the time since the last congestion event increases it becomes increasingly aggressive. This improves the time it takes to acquire available capacity in many scenarios by increasing the congestion window according to a cubic function.

Figure 4 shows how TCP CUBIC acquires capacity as the available capacity increase from 50Mbps to 100Mbps, and 50Mbps to 800Mbps over a 100ms round-trip time path. It takes roughly 15 seconds, or 150 round-trip-times, when the capacity goes from 50Mbps to 100Mbps. The number of round-trip-times needed to go from 50Mbps to 800Mbps is roughly the double, 300 round-trip-times. The takeaway is that as capacity increases so does the time it takes TCP CU-

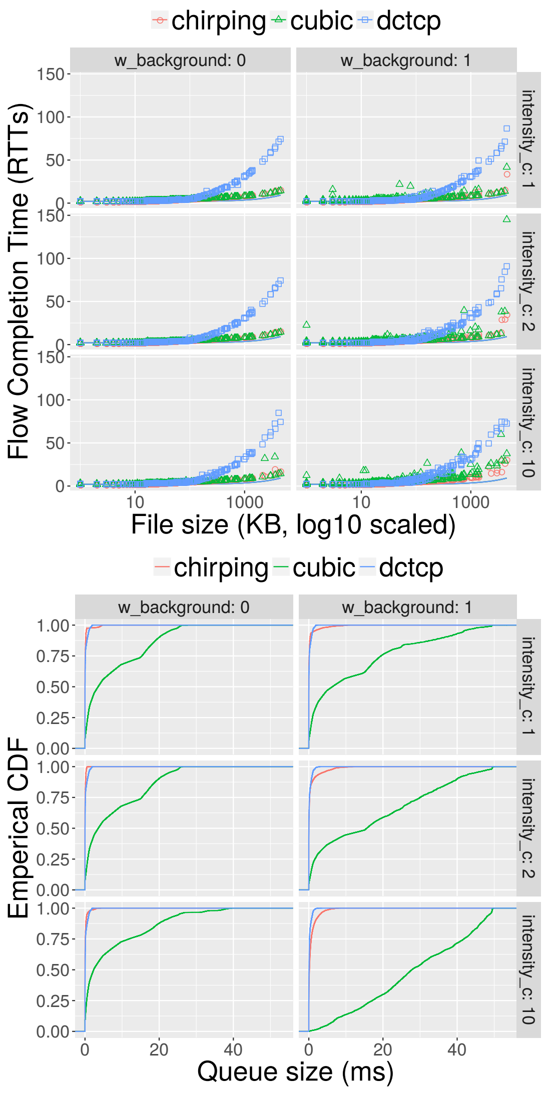


Figure 3: Flow Completion Time and queueing delay for Cubic, DCTCP and Paced Chirping. DCTCP and Cubic makes opposite trade-offs between latency and FCT, while Paced Chirping achieves both low latency and good FCT. ECN-marking threshold 1 ms or 1 BDP tail-drop; RTT 50 ms; capacity 100 Mb/s.

BIC to reach it.

Data Center TCP (DCTCP) [1], and other scalable congestion controls, can make more informed decisions about whether they are in-between congestion events or more capacity has become available compared to traditional TCP variants because congestion events are more frequent [1].

We have implemented Paced Chirping in ns-3 and done some very early experiments on turning Paced Chirping on in congestion avoidance when more capacity becomes available. The results are promising. It accelerates fast with little

[1]On average 2 marks per round-trip-time

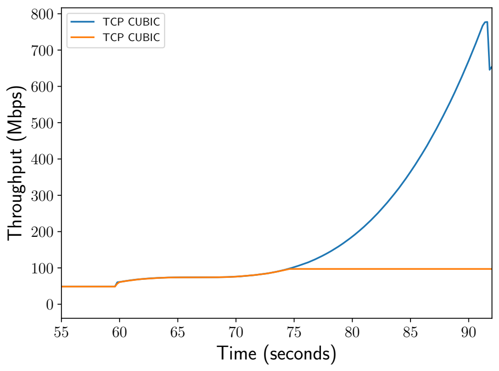


Figure 4: Comparison of TCP CUBIC performance when capacity goes from 50Mbps to 100Mbps and 800Mbps. Round-trip-time 100ms

if any overshoot of the bottleneck queue.

Figure 5 shows the throughput evolution of different TCP variants as capacity of the network is doubles during congestion avoidance. The flows are run separately in different experiments. The round-trip-time is 100ms, the capacity goes from 50Mbps to 100Mbps. The queue length is half a BDP at 50Mbps for Bic and Cubic, and 1ms at 50Mbps for both DCTCP variants. The modified DCTCP variant uses Paced Chirping when it detects lack of expected ECN marks, and the unmodified DCTCP variant uses additive increase.

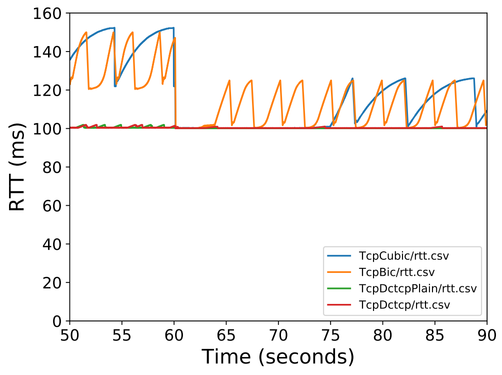
## 4 Implementation

Paced Chirping is implemented as two parts. The first is modifications to the pacing framework in the Linux kernel that allows any congestion control module to realise chirps. The second part is logic added to the DCTCP congestion control module that implements Paced Chirping.

## Status

The latest version of paced chirping logic and modifications to the pacing framework are implemented on the net-next branch (v5.0-rc8). A roadmap of the implementation(s) and how it can be installed can be found at `github.com/joakimmisund/PacedChirping`. The code is open sourced under GPLv2.

The code is still research and we are focusing on developing the algorithms rather than making the current production ready (which it is not). We have not observed any kernel freezes with the current version. We encourage you to experiment with it (at your own risk) and give us feedback.



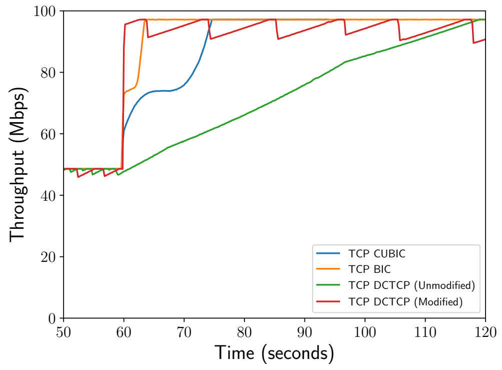


Figure 5: Comparison of ca performance when capacity doubles (from 50 to 100 Mbps) over a 100ms round-trip-time path

### 4.1 Kernel code changes

In this subsection we will discuss the changes we have made in the kernel to enable congestion control modules to send chirps. The code should work with both internal pacing and external pacing (FQ, NIC), but has only been tested with internal pacing.

We will start this subsection with the different parts of the kernel we have modified. First, we will discuss the data types, variables and the callback we have added. Second, discuss the added logic.

#### Data types, variables, Callback

Listings 1 shows the changes made to the tcp header-file under include/linux. We have introduces a new struct called chirp which describes how a chirp should look like. It has the number of packet and the number of packets sent. It has a gap that is the time inserted between subsequent packets. It has a gap_step which is a value that is to be subtracted from gap each time gap is used. There are two TCP sequence numbers that denote the first sequence number of the first packet in the chirp and the first sequence number after the last packet in the chirp. The pointer is an optional pointer to memory where the actual inter-send times can be put. This is

Listing 1: New struct in tcp_sock

```
struct chirp {
 u16 packets;
 u16 packets_out;
 u32 gap_ns;
 u32 gap_step_ns;
 u32 guard_interval_ns;
 u32 begin_seq;
 u32 end_seq;
 u64 *scheduled_gaps;
};

struct tcp_sock {

                ...

 u32 is_chirping;
 struct chirp chirp;
 u32 disable_cwr_upon_ece;
 u32 disable_kernel_pacing_calculation;

                ...
};
```

useful if the kernel is unable to realise the inter-packet gaps accurately.

In tcp_sock we have added one instance of struct_chirp and three flags. is_chirping indicates whether or not the congestion control is currently chirping. The other two are for disabling reaction to ECN marks and disabling default pacing rate calculation. These to are not important for the general chirping framework, but are necessary for Paced Chirping. disable_cwr_upon_ecn prevents the kernel from entering CWR when it receives an ECN mark. Every mark is give to the congestion control module, so the information is not lost. disable_kernel_pacing_calculation prevents the kernel from updating the pacing rate (sk_pacing_rate) which is necessary during the transition phase from Paced Chirping to congestion avoidance.

Listings 2 shows the new callback added to the congestion control ops structure. The return value indicates whether or not a new packet should be sent. 1 means that a new packet should not be sent, and 0 means that a packet should be sent. It can thus block sending of packets by returning 1 (or any true value), which might be necessary when external pacing is used to prevent excessive queueing between the TCP stack and the NIC. The implementation of new_chirp is expected to fill in the struct_chirp in tcp_sock with information about the next chirp. It can choose not to and return 0 to send packets without any pacing. This is useful if the application does not have enough data to use Paced Chirping.

## Logic

The logic to realize the chirps are located in tcp_write_xmit and tcp_update_skb_after_send. The code is listed in listing 3 and 4.

Listing 2: New cong ops callback

```
struct tcp_congestion_ops {

                ...

 /* call when congestion control indicates that it is
     sending chirps
  * and stack does not have a chirp description available
      .
  */
 u32 (*new_chirp)(struct sock *sk);

                ...
};
```

We will start by looking at tcp_write_xmit. First, The code checks three things in sequence. It checks if chirping has been requested. If it is then it check if the current chirp description is used. If the number of packets sent is greater or equal to the number of packets in the chirp then the chirp description is used. The third check is two purposed. The return value whether or not a new packet should be sent. If the congestion control returns 1 (or any true value) break is executed preventing sending of the packet. If the congestion control returns 0 the packet is sent as part of or not part of a chirp. The congestion control can return 0 without providing a new chirp description. The second purpose is to allow the congestion control module to fill in the chirp descriptor in tcp_sock. (Maybe a NULL pointer check should be added?)

Second, it forces the segment limit to be MSS. Paced Chirping only works if it controls the inter-packet gaps of individual MSS sized packets. Time-based TSO with dynamic inter-packet gaps might be possible in the future. At netdev 0x12 Van Jacobson held a keynote arguing for changing the NIC API from what to send to what to send and when to send it[3]. If NIC vendors move to such an API it might be possible to realise chirps in the NIC, which can improve performance and pacing accuracy.

## Discussion

A challenge with using external pacing is that congestion control module gets a chirp request before the previous chirps guard interval has expired. One way to deal with this is to apply the guard interval using the internal pacing clock. An other solution might be to have a check similar to TCP small queues (either in the congestion control module or kernel) to prevent excessive scheduling of chirps.

The pacing framework provides only an upper bound on the data rate (lower bound on the inter-packet gaps).This means that it might give you a larger gap than requested. One situation where this can happen is when the guard interval is applied using internal pacing and the CPU goes into a power-saving state.

The chirp description is tailored for Paced Chirping. There

Listing 3: tcp_write_xmit

```
 * but cannot send anything now because of SWS or another
       problem.
 */
static bool tcp_write_xmit(struct sock *sk, unsigned int
     mss_now, int nonagle,

              ...

  if (tcp_pacing_check(sk))
   break;

  if (tp->is_chirping &&
      tp->chirp.packets <= tp->chirp.packets_out &&
      inet_csk(sk)->icsk_ca_ops->new_chirp(sk)) {
   break;
  }

              ...

  limit = mss_now;
  if (!tp->is_chirping && tso_segs > 1 && !tcp_urg_mode(
       tp))

   limit = tcp_mss_split_point(sk, skb, mss_now,
         min_t(unsigned int,
        cwnd_quota,
        max_segs),
         nonagle);

              ...
}
```

might come other algorithms that want to send packets in a different manner. The API could be made more generic by having the gaps calculated in the congestion control module removing all calculation from the pacing framework. This would require either more memory in struct chirp or having the congestion control module allocate memory for each chirp and have a pointer to it in struct chirp.

We have yet to run the algorithm with reordering of packets. The start and end sequence numbers are absolutely critical for a congestion control module to identify which chirp an acknowledgement belongs to. We foresee that reordering in the border between chirps can pose a challenge.

## 4.2 Using the framework (congestion control module)

To use the framework a congestion control module has to do the following things.

**Tell kernel to chirp** Set is_chirping to 1 (or any true value). The kernel will check if this is set, and only then ask the congestion control module for chirp descriptions as discussed above.

**Provide kernel with chirp descriptions** Implement new_chirp callback. It should do one of the three first ac-

Listing 4: tcp_update_skb_after_send

```
static void tcp_update_skb_after_send(struct sock *sk,
     struct sk_buff *skb,
          u64 prior_wstamp)
{

              ...

 if (sk->sk_pacing_status != SK_PACING_NONE) {
  unsigned long rate = sk->sk_pacing_rate;

  if (tp->is_chirping) {
   if (tp->chirp.packets > tp->chirp.packets_out) {

    struct chirp *chirp = &tp->chirp;
    u64 len_ns = chirp->gap_ns;
    u64 credit = tp->tcp_wstamp_ns - prior_wstamp;

    chirp->gap_ns = (chirp->gap_step_ns > chirp->gap_ns)
         ?
     0 : chirp->gap_ns - chirp->gap_step_ns;
    chirp->packets_out++;

    if (chirp->packets_out == 1U) {
     chirp->begin_seq = tp->snd_nxt;
     credit = 0;
    }

    if (chirp->packets_out == chirp->packets) {
     tp->tcp_wstamp_ns += chirp->guard_interval_ns; /*Don
          't care about credits here*/
     chirp->end_seq = tp->snd_nxt + skb->len;
     inet_csk(sk)->icsk_ca_ops->new_chirp(sk);
    } else {
     /* take into account OS jitter */
     len_ns -= min_t(u64, len_ns / 2, credit);
     tp->tcp_wstamp_ns += len_ns;
     if (chirp->scheduled_gaps) {
      chirp->scheduled_gaps[chirp->packets_out] = credit
           + len_ns;
     }
    }
   }
  }
 }

              ...
}
```

Listing 5: new_chirp callback

```
static u32 dctcp_new_chirp (struct sock *sk)
{
  /* Fill in tp->chirp and return 0*/
  /* Do not fill in tp->chirp and return 0*/
  /* Do not fill in tp->chirp and return 1*/
  /* Fill in tp->chirp and return 1 ?? */
}

static struct tcp_congestion_ops dctcp = {
  ...
  .new_chirp = dctcp_new_chirp,
  ...
};
```

tions in listing 5.

**Enable pacing** Pacing can be enabled by writing:

```
cmpxchg(&sk->sk_pacing_status,
        SK_PACING_NONE,
        SK_PACING_NEEDED);
```

If you set is_chirping to 1 but forget to enable pacing things might go wrong.

### Discussion

An useful check the congestion control module might want to do is to check that there is enough data in the send buffer for a chirp of size N. Maybe something like provide chirp if:

```
SK_TRUESIZE(tp->mss_cache) *
(N + tp->packets_out) <=
sk->sk_wmem_queued
```

This can allow for fallback to regular slow start if the flow is too small to use Paced Chirping, and remove unnecessary overhead associated with chirping.

In previous versions of Paced Chirping the congestion control module communicated the desired gaps through a linked list of gaps. However, we realized that this was not flexible enough for many scenarios. One of them being smaller flows that does not need or can not use Paced Chirps. In the previous versions the chirp was scheduled before the congestion control module could look at the amount of data the flow had scheduled.

## 5 Further work

- Conduct experiments over variable-rate links to identify potential challenges and improvements to the noise filtering.
- Improve precision and interaction with existing slows by exploiting ECN.
- Evaluate over the Internet.
- Handle loss and reordering.
- Handle delayed acks, and possibly look into how one-way delay can be obtained and used.

## 6 Conclusion

Paced Chirping can become the solution to slow starts dilemma; choosing between acceleration speed and queue impact. Although there is much more work to be done we have show the potential of Paced Chirping.

The code is open-sourced, and we encourage researches and curious people to try it out and experiment with it.

## References

[1] ALIZADEH, M., ET AL. Data Center TCP (DCTCP). *Proc. ACM SIGCOMM'10, Computer Communication Review 40*, 4 (Oct. 2010), 63–74.

[2] HA, S., RHEE, I., AND XU, L. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review 42*, 5 (2008), 64–74.

[3] JACOBSON, V. Evolving from AFAP: Teaching NICs about time. `https://www.netdevconf.org/0x12/session.html?evolving-from-afap-teaching-nics-about-time`, July 2018.

[4] MANZOOR, J., DRAGO, I., AND SADRE, R. How HTTP/2 is changing Web traffic and how to detect it. In *Network Traffic Measurement and Analysis Conference (TMA), 2017* (June 2017), pp. 1–9.

[5] MISUND, J., AND BRISCOE, B. Paced Chirping: Rapid flow start with very low queuing delay. In *Proc. IEEE Global Internet Symp.* (May 2019), IEEE.

[6] RIBEIRO, V. J., RIEDI, R. H., BARANIUK, R. G., NAVRATIL, J., AND COTTRELL, L. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *Passive and Active Measurement Workshop (PAM'03)* (2003).