TCP Inigo

Fighting Congestion With Both Hands

Andrew Shewmaker (agshew@gmail.com) Carlos Maltzahn
Katia Obraczka
Scott Brandt
University of California Santa Cruz
This research was partially funded by a Google Faculty Award

November 15, 2014

Abstract

The Land of Net is always in peril from unscrupulous "princes" and six-fingered apps. TCP Inigo is the hero of this story. He functions surprisingly well, considering he has a drinking problem or two and prefers to fight with his left hand even though he's right handed. Join TCP Inigo as he overcomes his faults and defeats the six-fingered apps who killed his buffer.

1 Introduction

> I am not *the* TCP ... My name is Westwood; I inherited the name TCP from the previous protocol, just as the next will inherit it from me. The protocol I inherited it from is not the real TCP either. Its name was Reno. The original TCP has been retired fifteen years and living like a king in Tahoe.
>
> - The Dread Pirate TCP Westwood

This paper will not delve into the long history of TCP. It builds off of Data Center TCP (DCTCP) [1]. DCTCP improves congestion control by using ECN markings to estimate the extent of congestion instead of just the presence of it. Given congestion markings on all of the packets in a window, DCTCP will halve the window just like traditional TCP does when it detects packet loss or when TCP+ECN does when it sees a single ECN marking. If DCTCP sees fewer markings it will back off proportionally less.

DCTCP is transitioning out of academic status, with support already in or coming to MS Windows, Linux, and FreeBSD. It also serves as the basis for continuing academic research. HULL [4] adds packet pacing and phantom queues to DCTCP, sacrificing some utilization in order to reduce latencies. Deadline-aware DCTCP (D2TCP) [13] modifies the congestion response further to enable fine-grained, soft QoS. And the IETF is discussing other ways to improve congestion notification and DCTCP.

2 Drinking Problems

Below is a list of some significant issues with DCTCP. It inherited some from previous TCP's, and some are specific to its method of measuring congestion. Since TCP Inigo is based on DCTCP, then it had better deal with them.

DCTCP assumes receiver-side modification

This causes problems with incremental rollout and heterogenous environments. One-sided DCTCP [6] avoids this requirement, but must make compromises that reduce its effectiveness if the receiver has been modified.

DCTCP is vulnerable to ack-loss

ACKs might be lost due to congestion on the reverse path or an intermittent link fault. In this situation, DCTCP's congestion estimate becomes inaccurate, which results in overshooting and undershooting the optimal window size [7].

DCTCP can exhibit instability

The longer the RTT of a flow, the more delayed the signal of congestion is in getting back to a sender. Delayed signals, along with DCTCP's suggested single threshhold value used to mark packets in switches, can cause instability in some circumstances [5, 8]. Chen, et. al suggest using a double threshhold [5], showing that it is able to handle approximately five more flows than a single threshhold before collapsing due to incast. But the double threshold implementation is more complicated, and that increase in complexity is likely to inhibit its adoption. Zats, et. al propose direct and more informative congestion feedback with FastLane [15]. While improved congestion notification like FastLane's would benefit response time and stability, it requires significant changes in switches and hosts.

DCTCP responds to congestion only once per window

As in regular TCP, this exacerbates round-trip-time unfairness where flows with large RTTs get a smaller portion of the bottleneck bandwidth. The original authors of DCTCP propose a modification in which it responds to each ACK [2], and simulation results looked promising. Unfortunately, this idea is problematic to implement because it requires subtracting less than one packet from the window for each ACK reporting congestion.

DCTCP only works if ECN is enabled and configured

DCTCP wasn't intended for use outside the data center, so requiring ECN isn't much of an issue. However, it would be nice if the benefits of DCTCP could be experienced in every situation that standard TCP is currently deployed. Unfortunately, it doesn't appear likely that ECN will become

common across the Internet or that switches will be configured to mark ECN according to DCTCP's guidelines.

DCTCP halves its window upon packet loss

A behavior inherited from traditional TCP, halving the window on packet loss causes under-utilization of wireless networks since packet loss may be due to noise rather than congestion. Wireless DCTCP [14] modifies DCTCP with an ACK-based estimate of available bandwidth similar to TCP Westwood in order to tolerate random packet loss. However, estimates like that can easily be too volatile due to issues like ACK compression, and smoothing techniques can make the estimate not responsive enough. Average or current RTTs are inherently volatile and must be used with extreme care or avoided.

DCTCP is oblivious to deadline and utilization requirements

Like traditional TCP, DCTCP emphasizes fairness. Coarse-grained QoS via priority classes is separately provided by other layers in the network. Deadline-aware DCTCP (D2TCP) [13] adds awareness of deadlines and remaining work to DCTCP's congestion response, enabling fine-grained per-connection, soft QoS. The gamma function it uses to modify the congestion response is inadequate because under high congestion an urgent flow will back off similarly to a non-urgent flow—instead it should actually increase its window size. Therefore, D2TCP gives soft QoS support to TCP, but it could do significantly better.

DCTCP gets bullied by Reno

Newer variants of TCP that seek to minimize queue buildup tend to get bullied by TCP Reno, which will push until packet loss. Some variants try to counteract Reno's bullying, but DCTCP does not. So, an organization wanting to use DCTCP would either have to upgrade every TCP stack or configure the switches to separate DCTCP and Reno into separate queues. That may be difficult or impossible in a multi-tenant environment or when external traffic must be allowed in.

3 TCP Inigo

Here is how Inigo will deal with each of the previously described issues.

DCTCP assumes receiver-side modification

See next item.

DCTCP is vulnerable to ack-loss

A TCP Inigo receiver will use an experimental option as outlined in [12]. Each ACK will include a measure of congestion, calculated by the receiver

in the same way as the sender. That will give Inigo senders an accurate measure of forward path congestion even with ACK loss. The sender could even estimate congestion on the reverse path by comparing its idea of forward path congestion to the receiver's. If the receiver does not recognize and use the experimental option, the sender can be set to either refuse to connect or use one-sided congestion calculations [6].

DCTCP can exhibit instability

In their analysis of QCN, Alizadeh, et al. showed how the averaging principle [3] can add stability to a congestion protocol. Inigo will follow their advice to dampen window adjustments during the congestion avoidance phase by setting the window size to the average of the previous two windows after a half window's worth of packets have been sent.

DCTCP responds to congestion only once per window

The DCTCP RTT-fairness enhancement [2] could be made practical by responding to congestion every *cong_interval* ACKs instead of every ACK, where $4 < cong\_interval < window$. A minimum of 4 ACKs is used so that the averaging principle can continue to be applied to the smallest *cong_interval*. Some may argue that changes to window size shouldn't be made more frequently than one RTT since responding more quickly leaves no time to observe the effect of the last change. However, the more frequent window reductions proposed are proportionally smaller and designed to sum up to nearly the same amount of change that would occur under persistent congestion if the window size was only modified once per RTT. And the benefit that flows with longer RTTs aren't penalized compared to those with shorter RTTs is important for connections outside the data center or in complex intranet topologies.

DCTCP only works if ECN is enabled and configured

Inigo will approximate DCTCP's ECN-based measure of congestion via the ratio $f : packets_{delayed} \rightarrow windowsize$. Delay will either be based on RTTs or Relative Forward Delays (RFDs) if the timestamp fidelity allows it. In this way, the benefits of DCTCP can come to more diverse networks, including the Internet. This is described in more detail in section 4.

DCTCP halves its window upon packet loss

Inigo will assume some amount of packet loss is normal, and beyond that approximate DCTCP's measure of congestion via the ratio $f : packets_{lost} \rightarrow windowsize$. This should enable Inigo to take better advantage of wireless networks than other variants of TCP. The adaptation of DCTCP's congestion measurement to packet loss is described in section 5.

DCTCP is oblivious to deadline and utilization requirements

Inigo will calculate urgency as in D2TCP [13], which is similar to what I proposed in my master's thesis and PhD proposal [11]. But instead of modifying the congestion response with a gamma function it will use a probabalistic response to give stronger guarantees. If a bottleneck is overloaded with too many flows claiming high urgency, then they will push the network toward packet loss and Inigo's behavior will become similar to Reno's. See section 6 for more information.

DCTCP gets bullied by Reno

Inigo's tolerance of some packet loss and its notion of urgency should make coexistence with Reno feasible. Whereas many TCP variants that are careful to not overflow buffers tend to get bullied by TCP Reno, Inigo will be significantly less aggressive in backing off due to packet loss. In addition, if Inigo enters slow start whenever its congestion ratio is zero, then it will push back effectively against Reno's bullying. Furthermore, by using a notion of urgency calculated using deadlines and the amount of work remaining, Inigo can ignore some amount of congestion. In the end, Inigo will play rough if other flows play rough, but it will still halve its window under chronic congestion, so it shouldn't push the network to collapse.

4 Measuring Congestion Via Delay

Inigo seeks to approximate DCTCP's ECN-based congestion measurement with one based on timestamps. A person might be tempted to compare the current RTT to the minimum observed RTT plus three times the median absolute deviation (i.e. Is $RTT < RTT\_min + 3 * mdev$?). This is similar to, but strictly less than the retransmit timeout.

However, that approach is problematic. First, it will take some time to warm up the mdev, although it would perhaps be sufficiently warmed up by the end of slow start. Second, as congestion increases, the mdev will increase, which will make it less likely that a new larger RTT will count as a mark of congestion. Third, creating a good average RTT estimator is difficult, although the machine learning technique employed by Nunes, et al. [9] shows promise.

It is safer to avoid the use of an average RTT or its deviation. A better criteria would be if $rtt < (rtt\_min + pkt\_thresh\_time)$, then count it as a mark of congestion. The approximate number of packets, pkt_thresh, necessary for DCTCP to achieve 100% throughput is 17% of the bandwidth delay product, according to Alizadeh, et. al [2]. And we can see that from an end host's perspective that the corresponding delay threshhold would simpy be:

$$pkt\_thresh\_time = 0.17 rtt\_min \tag{1}$$

Obviously, this threshhold makes small RTT flows more sensitive to congestion than large RTT flows. On the other hand, TCP's congestion avoidance has traditionally been biased in the other direction due to the fact that a window

grows by one packet per RTT. The overall impact will need to be measured, and may necessitate a sublinear *pkt_thresh_time* that balances the desire to enable 100% throughput for large RTT flows with the need for fairness.

As an alternative to RTT, Inigo could use Relative Forward Delay (RFD) as introduced by TCP Santa Cruz [10]. RFD is the difference in one way delays, but can be used in the common scenario of unsynchronized clocks. The major advantage of RFD over RTT is that it doesn't conflate congestion on the forward and reverse paths. TCP Santa Cruz uses it to model the depth of the bottleneck queue. Unfortunately, TCP Santa Cruz never saw implementation outside of the ns-2 simulator. Also, when I have attempted to use RFD in a modified Linux TCP, the RFDs didn't make sense. I believe this was due to inadequate TCP timestamps–the granularity wasn't fine enough in my tests and the TX timestamp is created too high in the stack, which means measurements include the time a packet spends in the sender's buffers.

RTTs suffer from those problems too, but RFDs appear to be more sensitive, perhaps due to their smaller size. It might be possible to improve TCP timestamps or take advantage of the IEEE 1588 Precision Time Protocol, which creates a TX timestamp either in the NIC or just before. If using those timestamps is feasible, then one might ask why not simply implement TCP Santa Cruz's method of adapting to congestion. One flaw in its bottleneck queue model is that it assumes the queue is empty at the beginning of a flow's existence, where it might already be congested. It might take a while before TCP Santa Cruz's queue model drops below zero and reset its baseline, but in the meantime a flow would not be getting its fair share. I believe it would be better to use RFD with DCTCP's technique.

5 Measuring Congestion Via Loss

It is normal for a wireless network to experience loss, and many academic papers report losses from 1 to 2.5% on a typical network. In that scenario a few losses don't indicate packet loss and chronic congestion, as most TCP variants assume. Furthermore, more frequent losses don't indicate an overflowing buffer, nor do they necessarily indicate chronic congestion, although the likelihood does increases the busier the wireless network is. Therefore TCP Inigo will assume 2.5% packet loss, and it will use the ratio of unexpected lost packets to window size. Since the receiver will also be measuring the extent of congestion—via ECN, timestamps, and loss— the sender will also be able to perform some sanity checking of its loss-based measure of congestion too.

While it may be desirable for Inigo to detect if it is using a wireless device and modify its congestion response accordingly, tolerance of some packet loss is appropriate for wired networks too. That's beause as link rates increase toward a medium's error rate, loss becomes inevitable. Also, if the ECN and timestamp-based methods of measuring congestion are doing their jobs and all flows are well behaved, then packet loss due to congestion should become rare. And if one or more flows aren't well behaved (e.g. TCP Reno, or too many deadline-driven flows are claiming to be urgent), then this limited toleration of packet loss means that Inigo won't be easily bullied.

Note that just as in DCTCP's ECN-based congestion ratio, when the loss ratio appoaches one within a RTT, then Inigo will still halve its window. What

about particarly noisy wireless networks? If the observed loss is consistently greater than 2.5%, regardless of window size, then Inigo will increase its expectation to what it's observing. In the corner case where Inigo sees large amounts of packet loss, then it will initially back off, but it will eventually push harder as it's expected packet loss increases. If a lower loss ratio is seen at some later time, then the expected loss ratio will be reset accordingly.

It should be noted that most DCTCP experiments published have reported very little packet loss, if any. If there is, then DCTCP should respond like Reno. Since Inigo will be modifying the response to packet loss, we'll have to test for unforseen interactions between its congestion control and error control.

Other solutions to fix TCP's suboptial use of wireless networks have been proposed, but they tend to horrendously violate layer separation and there are different types of wireless networks. It would be nice to solve this once, completely in the transport layer.

6 QoS Support

Deadline Response Principles:

- devolve to proportional fairness
- slow start: $Win[2 * W, N * W], N > 2$
- cong avoid: $Win[W/2, W + 3]$
  - growth enhanced by urgency
  - backoff enhanced by lack of urgency
- desynchronize flows
- transition from mouse to elephant flow
- add admission control for real guarantees
- kernel-friendly algorithm

Inigo will keep D2TCP's [13] calculation of urgency based on the ratio of time to completion to deadline. But Inigo will add support for periodic deadlines, as well as dynamic adjustment of them. In particular, Inigo will by default set the amount of work and deadline as appropriate for a mouse flow for one deadline, then after that use values specified by the application or fall back to deadline-unaware behavior (i.e. become a best-effort elephant flow).

It's a good idea to keep slow start exponential so that even low urgency flows suck up available bandwidth. But urgent flows should grow faster. Ideally, we could set urgent flows to the line rate, but we can't assume that is known. But we might go faster than $2 * W$.

Flows might be synchronized, causing a bottleneck buffer to fill suddenly. Desynchronization via a frequency shift (window adjustment) and/or a phase shift (busy loop) bounded by the flow's remaining work and deadline could reduce the maximum depth of the bottleneck buffer.

```
T_c = B / (3/4 W)                    # time to complete as in D2TCP
d = min(1000 * T_c / D, 2000)        # urgency: 0 < d < 2000

if (prandom_u32_max(2000) < d)       # urgent flows more likely to grow
       W *= 2 + (d + 500) / 1000     # urgent flows grow more
else
       W = 2W                        # response in [2W, 5W] per RTT
```

Figure 1: Probabilistic Deadline Slow Start

6.1 Deadline Aware Slow Start

The congestion ratio a is scaled as in DCTCP to 1000 (really 1024). Urgency d is initially scaled to 1000 and capped at 2000, similarly to D2TCP, but the range will have to be re-evaluated at different link rates. Also, note that the random function is more kernel friendly than D2TCP's gamma function, which requires some sort of fractional power function or lookup table.

6.2 Deadline Aware Cong Avoid

```
T_c = B / (3/4 W)                    # time to complete as in D2TCP
d = min(1000 * T_c / D, 2000)        # urgency: 0 < d < 2000
a = (1 - g) a + g * cong_ratio       # extent of congestion as in DCTCP

if (prandom_u32_max(2000) > d)       # urgent less likely to back off
       b = a + a * (2000-d+500) / 1000
       b = min(b, 1000) / 1000       # adjusted cong ratio: a < b < 3*a
       W = W (1000 - b / 2) / 1000   # response in [W/2, W] per RTT
else
       # urgent flows more likely to grow and grow more
       # response in [W+1, W+3] per RTT
       W += 1 + (d + 500) / 1000
```

Figure 2: Probabilistic Deadline Cong Avoid

If cong ratio $a = 0$ (i.e. no congestion) and urgency $d = 0$, then there is a 100% chance of backoff. Specifically, the adjusted cong ratio $b = 3 * a = 0$ and so $W = W$.

But if the urgency $d = 1800$, then there is a 10% chance of backoff. If it does back off, then the adjusted cong ratio $b = a = 0$ and $W = W$. On the other hand, it has a 90% chance of growth $W = 1 + (1800 + 500)/1000 = W + 3$.

If cong ratio $a = 1000$ (i.e. chronic congestion) and urgency $d = 0$, then there is a 100% chance of backoff. The adjusted cong ratio $b = 3 * a$, but is capped at 1000, so $W = 0.5 * W$.

But again, high urgency $d = 1800$ would basically override the typical congestion response, just as before with a 10% chance of backoff and 90chance that the window would increase by 3 packets.

6.3 Admission Control

In order to make absolute guarantees of deadlines or utilization, Inigo will require admission control. Scalable admission control is difficult because there are multiple domains: a host TX port, one or more subnetwork domains, and potentialy a host RX port outside of a subnetwork domain.

The host ports could perform admission control using something like a Linux control group (cgroup). When a connection is being established, a network cgroup could be checked to make sure that the reserved utilization does not exceed 100%. The network cgroup would also be referenced by the Inigo sender when calculating urgency and decrementing the amount of work remaining.

How should a connection communicate a reservation to the SDN? A traditional client/server protocol would require explicit additional programming and network configuration effort. Why not use another TCP experimental option? It would provide information the SDN could use to choose a route for a new flow and it could communicate a reservation across multiple domains.

Each domain would check the new TCP reservation option and leave it alone if they agree to it, or change it to indicate only a lesser guarantee can be made. The receiver echoes the option back to the sender, who might decide to continue in the face of a lesser guarantee than desired. Note that each switch does not need to perform access control—just once per SDN domain.

If within an SDN domain, then access to the receiving host's RX port is controlled via the nearest switch TX port. However, a wireless p2p network would necessitate access control on the receiving host. And in any case the receiver might know that it can't handle as much data as can be delivered. Since a server may need to perform a lot of work per transaction, and there is little to be gained from sending data that must be dropped or suffer long delays in buffers outside the network stack, it would make sense to link the net cgroup to others (e.g. disk cgroup). That way, we can compose end-to-end guarantees.

7 Conclusion

The enhancements proposed in TCP Inigo are intended to spread the benefits of DCTCP to the Internet and wireless networks while making it more robust in the face of ACK loss, mismatched receivers, Reno, and more. It also promises per-flow QoS with the potential for hard guarantees with the assistance of an SDN controller for admission control. All of this still needs to be implemented and evaluated, of course.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). *ACM SIGCOMM computer communication review*, 41(4):63–74, 2011.

[2] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of dctcp: stability, convergence, and fairness. In *Proceedings of the ACM SIG-*

METRICS joint international conference on Measurement and modeling of computer systems, pages 73–84. ACM, 2011.

[3] Mohammad Alizadeh, Abdul Kabbani, Berk Atikoglu, and Balaji Prabhakar. Stability analysis of qcn: the averaging principle. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 49–60. ACM, 2011.

[4] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI*, pages 253–266, 2012.

[5] Wen Chen, Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Ease the queue oscillation: Analysis and enhancement of dctcp. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 450–459. IEEE, 2013.

[6] M. Kato. Improving transmission performance with one-sided datacenter tcp. Master's thesis, Keio University, 2014.

[7] M. Kuehlewind, R. Scheffenegger, and B. Briscoe. Problem statement and requirements for a more accurate ecn feedback. Internet draft, IETF.

[8] Abhisek Mukhopadhyay and Priya Ranjan. Nonlinear instabilities of d2tcp-ii. In *Technology, Informatics, Management, Engineering, and Environment (TIME-E), 2013 International Conference on*, pages 99–104. IEEE, 2013.

[9] Bruno Astuto A Nunes, Kerry Veenstra, William Ballenthin, Stephanie Lukin, and Katia Obraczka. A machine learning approach to end-to-end rtt estimation and its application to tcp. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–6. IEEE, 2011.

[10] Christina Parsa and J. J. Garcia-Luna-Aceves. Improving TCP congestion control over internets with heterogeneous transmission media. In *Proceedings of the 7th IEEE International Conference on Network Protocols (ICNP)*. IEEE, 1999.

[11] A. Shewmaker. Efficient performance guarantees on storage networks. Technical report, 2012.

[12] J. Touch. Shared use of experimental tcp options. RFC 6994, IETF.

[13] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[14] Jingyuan Wang, Yunjing Jiang, Yuanxin Ouyang, Chao Li, Zhang Xiong, and Junxia Cai. Tcp congestion control for wireless datacenters. *IEICE Electronics Express*, 10(12):20130349–20130349, 2013.

[15] David Zats, Anand P Iyer, Randy H Katz, Ion Stoica, and Amin Vahdat. Fastlane: An agile congestion signaling mechanism for improving datacenter performance. Technical report, DTIC Document, 2013.