Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter

Glenn Judd, *Morgan Stanley*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/judd

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter

Glenn Judd
Morgan Stanley

Abstract

Over the last several years, datacenter computing has become a pervasive part of the computing landscape. In spite of the success of the datacenter computing paradigm, there are significant challenges remaining to be solved—particularly in the area of networking. The success of TCP/IP in the Internet makes TCP/IP a natural candidate for datacenter network communication. A growing body of research and operational experience, however, has found that TCP often performs poorly in datacenter settings. TCP's poor performance has led some groups to abandon TCP entirely in the datacenter. This is not desirable, however, as it requires reconstruction of a new transport protocol as well as rewriting applications to use the new protocol. Over the last few years, promising research has focused on adapting TCP to operate in the datacenter environment.

We have been running large datacenter computations for several years, and have experienced the promises and the pitfalls that datacenter computation presents. In this paper, we discuss our experiences with network communication performance within our datacenter, and discuss how we have leveraged and extended recent research to significantly improve network performance within our datacenter.

1 Introduction

In recent years, datacenter computing has become a pervasive part of the computing landscape. The most visible examples of datacenter computing are the warehouse-scale computers [4] used to run search engines, social networks, and other publicly visible "cloud" applications. Less visible, but no less critical, are datacenter computing platforms used internally by numerous organizations.

In spite of the success of the datacenter computing paradigm, there are significant challenges remaining to be solved—particularly in the area of networking. The pervasiveness of TCP/IP in the Internet makes TCP/IP a natural candidate for datacenter network communication. TCP/IP, however, was not designed for the datacenter environment, and many TCP design assumptions—e.g. a high degree of flow multiplexing, multi-millisecond RTT—do not hold in a datacenter. A growing body of research and operational experience, has found that TCP can perform poorly in datacenter settings.

TCP's poor performance has led some groups to abandon TCP entirely [15]. This is not desirable, however, as it requires reconstruction of a new transport protocol as well as rewriting applications to use the new protocol. Recent research has focused on adapting TCP to operate in the datacenter environment. DCTCP stands out as a particularly promising approach as it utilizes technology available today to dramatically improve datacenter TCP performance.

In this paper, we discuss our experiences with network communication performance within our datacenter and discuss how we have leveraged and extended recent research to significantly improve network performance within our datacenter, without requiring changes to our applications.

The experimental results that we present are often in the form of controlled tests that isolate behavior that we encountered either in actual production TCP and DCTCP usage, or in our efforts to introduce DCTCP into production.

In addition, this paper makes the following specific contributions.

- To the best of our knowledge, this paper presents the first published discussion of DCTCP production deployment.
- We identify shortcomings that make DCTCP as presented and implemented in [1] unusable in our environment, and we present solutions to those shortcomings that we have verified through implementation.
- We demonstrate that commonly used receive buffer tuning algorithms perform poorly in current datacenters.
- We empirically compare DCTCP performance to TCP convergence, and we show that—surprisingly—DCTCP convergence can be superior to TCP convergence. We show that this is due to DCTCP's superior coexistence with common receive buffer tuning algorithms. With correct buffer tuning, TCP convergence, stability, and short-term fairness all exceed that of DCTCP.
- We also discuss results from dramatically reducing RTO_{min} at scale to mitigate incast.

Our discussion will proceed as follows. Section 2 will briefly describe our datacenter environment. Section 3 will discuss the three significant problems that we have encountered with TCP in our datacenter. Section 4 will discuss problems that delayed acknowledgements introduce into datacenter networks, and will analyze solutions. Section 5 will discuss reducing RTO_{min} to mitigate incast. Section 6 will discuss addressing the root cause of incast-induced packet loss using DCTCP. Section 7 will discuss obstacles that prevent DCTCP from being used in our environment, and solutions for those problems. Section 8 will then compare DCTCP performance to that of TCP. Section 9 will investigate the performance of automatic TCP buffer tuning in our environment. Section 10 will briefly discuss related work, before we conclude in Section 11.

2 Setting

The majority of recent work on TCP in the datacenter has either implicitly or explicitly been undertaken in the context of an Internet services setting. Of course, datacenter computation applies to a much broader spectrum of applications, and even within a single datacenter of a single organization, a wide variety of application types may be found.

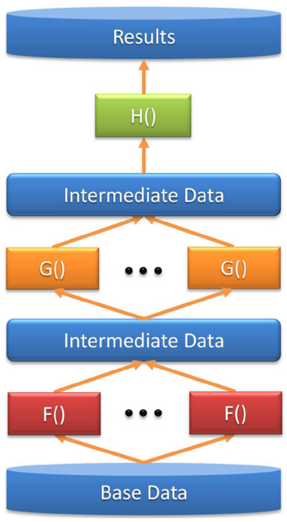

Figure 1: Typical Application Structure

2.1 Overview

The context of this work is a datacenter used largely for two broad types of applications: Monte Carlo simulation and data analysis. A typical application is structured as shown in Figure 1, which shows a common communication-intensive application structure in our datacenter. This application is constructed as a series of transformations (depicted as rectangles) on data (depicted as ovals and rounded rectangles.) Each transformation may read several data elements, and may store several data elements.

Most data access is to one of two highly-parallel distributed data storage systems: a key-value store and a distributed file system. The key-value store tends to generate much higher degrees of incast (discussed at length further in this paper) due to support for bulk reading and writing of values. The distributed file system results in more limited incast as the number of blocks simultaneously read by any particular operation is limited by the file system's read-ahead limit. Further details of these storage systems are outside the scope of this paper, but both are colocated with our computation servers and—thus—are highly parallel.

Monte Carlo simulations tend to be computationally intensive, but even they tend to contain periods of intensive communication. Data analysis applications tend to be storage and communication intensive.

As our datacenter is shared among many applications and distinct user groups, it is very important that applications in our datacenter are as loosely coupled as possible.

Unless otherwise specified, the applications discussed and results presented in this paper were obtained on a 10 Gbps network with a 9K MTU. Also unless otherwise specified, controlled experiments were conducted using iperf as traffic generator sending at the maximum rate allowed. TCP congestion control is CUBIC [6] unless otherwise stated (as CUBIC is the Linux default congestion control.) We conducted several of the controlled experiments using the Linux Reno implementation, but did not observe any significant differences. As such we have left comparisons with Reno (and other TCP variants) as out of scope for this work. Applications in this datacenter do not access the public Internet.

2.2 Traffic Characteristics

To illustrate the type of traffic that our applications generate, we recorded network traffic for a two-minute interval of a representative application (a Monte Carlo simulation) on a single server in this application. Due to the uniform nature of both our applications and our storage systems, the traffic seen by other servers is very similar. (We have verified this with additional samples on other servers.) Figures 2, 3, and 4 summarize flow characteristics of the recorded traffic.

TCP connections in our environment tend to be long-lived. For the purposes of this analysis, we define a flow as packets demarcated by TCP PUSH flags within a single TCP connection.

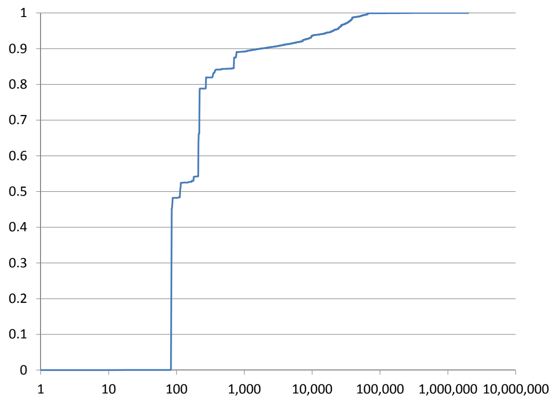

Figure 2: CDF of flow sizes

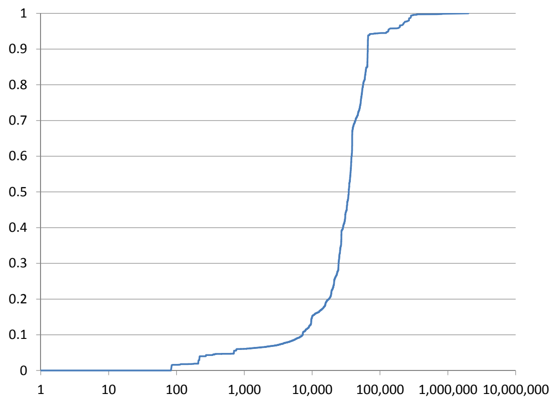

Figure 3: CDF of flow bytes

Figure 2 depicts the cumulative distribution of flow sizes sampled, and illustrates that the vast majority of flows in this application are very small. These small flows largely consist of either data retrieval requests, or simple operation results. (As stated earlier, an individual connection will contain many flows. These flows often occur in quick succession within the connection.)

Figure 3 illustrates the cumulative distribution of flow bytes. (For each flow size, the corresponding point depicts the fraction of total bytes in flows less than or equal to that flow size.) As shown in Figure 3, the majority of bytes are in larger flows—in spite of the large number of small flows. This is due to the fact that while the simple requests and operation results dominate in terms of flow numbers, most bytes on the network are generated by actual value storage or retrieval.

In addition, we also categorized the sampled traffic as shown in Figure 4. This figure shows the fraction of total traffic (measured in bytes) that falls into the given traffic categories. This figure clearly shows that key-value store traffic dominates, followed by distributed file system traffic. Other traffic types are not a significant fraction of the total traffic.

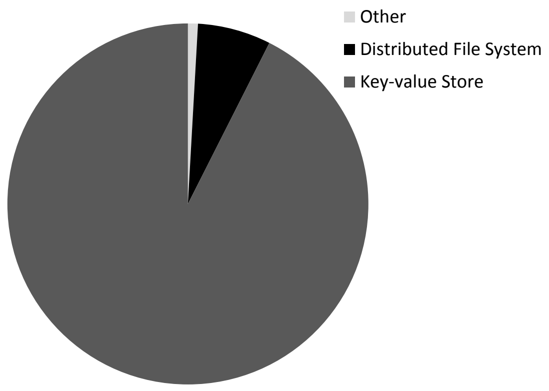

Figure 4: Flow Type Categorization

In summary, key-value store traffic dominates the traffic in the measured application. Most flows generated by this application are very small—too small for traditional congestion control to prevent problems such as incast. The majority of traffic, however, is in contained in larger flows. Thus, congestion control plays an important role in preventing larger flows from experiencing congestion, and in preserving buffer space for small flows.

3 TCP in the Datacenter

Communication intensive datacenter applications present datacenter networks with several performance problems [5]. This is largely due to the fact that TCP—the foundation of many datacenter applications—was not originally designed with the characteristics of modern datacenters in mind. In this section we discuss three significant problems with TCP that we have encountered: delayed ACK induced stalls, incast, and problems with receive buffer tuning.

3.1 Stalls Due to Delayed ACKs

Delayed ACKS in TCP allow TCP to substantially reduce the number of packets sent. Delayed ACKs work by delaying the sending of an ACK for multiple segments. The delayed ACK effectively merges ACKs by cumulatively acknowledging multiple received segments.

Delayed ACKs have an associated timeout to prevent the sender from stalling forever due to a lack of ACKs from the receiver. The default timeout is tens to hundreds of milliseconds. In a datacenter with sub-millisecond RTT, the default delayed ACK timeout is far too large, and we have observed application-level timeouts that were caused by delayed ACKs. Section 4 will discuss resolving this issue.

3.2 Incast

The most vexing problem that TCP encounters in our datacenter network is "TCP incast" [12]. TCP incast occurs whenever a single receiver receives data from multiple senders in a short amount of time. This is a frequent communication pattern in datacenter applications. As depicted in Figure 5, when this situation occurs, the switch to which the receiver is attached is often overloaded: the senders send more data than the receiver can receive; the switch cannot store all of the data; and so the switch discards data that it does not have room for [13]. Unlike delayed ACK-induced timeouts, incast is much more difficult to remedy, and we will spend much of this paper discussing this problem.

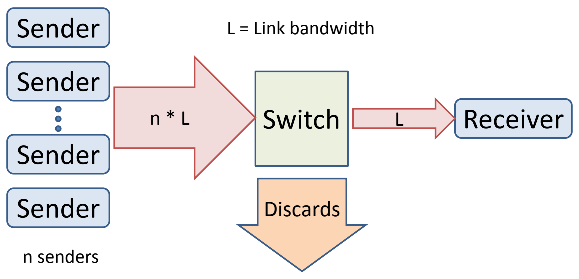

Figure 5: Incast

Previous work discussing incast and other datacenter TCP problems has focused on Internet service applications and shown that TCP performs poorly in datacenters that are servicing these applications. While the nature and structure of our datacenter applications are very different, we still experience similar problems with TCP in our datacenter.

Consider our typical application structure discussed previously and depicted in Figure 1. Each transformation may read several data elements from our distributed storage systems, and may store several data elements into our distributed storage systems. As a result, reads from our distributed storage systems often result in a high degree of TCP incast. Writes to the distributed storage systems also contribute to incast as many writers may be writing to the same storage node.

At a high level, we find that incast produces the following problems at the application layer:

- Communication timeouts and retransmissions
- Lost throughput
- Increased latency
- Latency variance (jitter)

These problems can afflict even "innocent" applications and servers uninvolved in the communication. At the business level, further problems result:

- Application failures
- Idle servers waiting for communication, and increased costs associated with procuring and operating additional servers.
- Application failures even for "innocent bystanders"
- Development effort to work around communication problems
- Effort lost troubleshooting network problems in innocent applications
- Effort lost coordinating among different development groups to avoid communication problems.

3.3 Receive Buffer Tuning

In addition, a very significant problem that we have encountered with TCP in the datacenter is receive buffer tuning [16]. The receive buffer size has a dramatic impact on TCP performance and server RAM utilization.

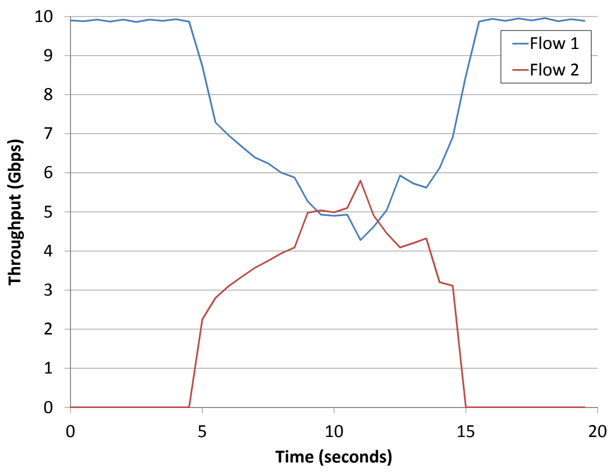

Figure 6: TCP convergence

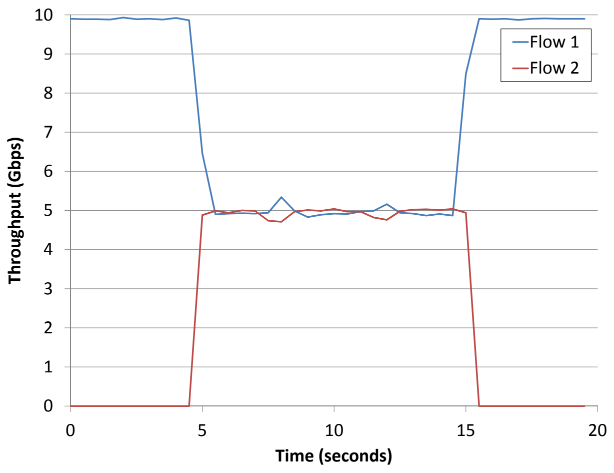

Figure 7: DCTCP convergence

To illustrate this, consider the results of a simple two-flow throughput experiment. Both flows were sent from distinct servers to a common receiver. The first flow ran for 20 seconds. The second flow started 5 seconds later, and ran for a total of 10 seconds. The results are shown in Figure 6.

TCP convergence, fairness, and stability in this test are all extremely poor. TCP should be able to converge within a few RTT, not several seconds. (While [11] discusses some detailed problems with TCP-CUBIC convergence, the behavior shown in Figure 6 is far worse than is expected.)

Figure 7 repeats this test for DCTCP. Surprisingly, while [2] finds that DCTCP converges more slowly than TCP, Figure 7 shows DCTCP dramatically outperforming TCP with respect to stability, convergence, and short-term fairness.

The source of this unexpected behavior is receive buffer tuning. This will be addressed in detail in Section 9.

3.4 Summary

The problems discussed above are significant, and historically we worked around them at the application layer. In the following sections, we discuss how we have largely eliminated these problems, dramatically increased our network performance, and removed the need for application-level workarounds.

4 Delayed ACKs

As discussed earlier, delayed acknowledgements can cause significant problems. Delayed ACK timeouts are—by default—far too large for a datacenter setting. Fortunately, there are two simple alternatives to remedy this problem: 1) eliminate delayed acknowledgements, or 2) reduce the delayed acknowledgement timeout. We have investigated both approaches.

If ACKs could be generated without cost, the ideal ACK delay would be zero, and an ACK would be generated for every single packet. Unfortunately, while eliminating delayed ACKs eliminates the possibility of any sender stall, it does so at the cost of generating a significant number of packets. We do not find this increased load to be a problem in our network, but we do find it to be problematic on our end servers.

Figure 8 illustrates this behavior. In this test, one or two senders send to a single receiver. Delayed ACKs are delayed a maximum of 0 (i.e. no delayed ACKs), 1, or 40 milliseconds. The total CPU % utilized by IRQ daemons on the receiver for the given test is plotted for each test (100% is the equivalent of 1 CPU completely busy). This test exhibits essentially no difference for delays of 1 and 40 milliseconds. Turning delayed acknowledgements entirely off, however, produces a sharp increase in CPU utilization for both one and two flows. (Repeating this test yields similar results with insignificant variation.)

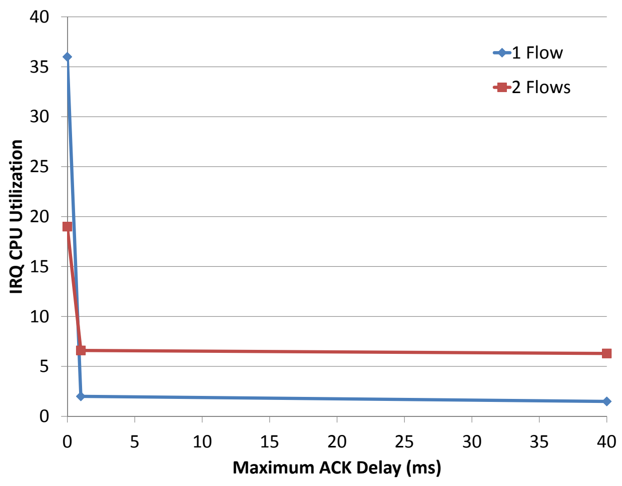

Figure 8: Delayed ACK CPU Utilization

For this reason, in our network we now lower the delayed ACK timeout as much as possible without turning delayed ACKS off. Those constraints yield a delayed ACK setting of 1 ms. We will still incur an occasional stall, but the stall is not long enough to cause significant issues at our application layer. (Some applications, however, may benefit from turning off delayed ACKs entirely.)

5 Reducing RTO_{min}

During the most communication-intensive phases of our application, we found that our applications were experiencing large numbers of incast-induced TCP timeouts. At the application layer, this resulted in a long tail on our task completion times. The effects of incast are clearly seen in Figure 9 which shows a TCP sequence graph from a single flow of a production application during a heavy all-to-all incast. The duplicate sequence numbers visible are packet losses and retransmissions that were successfully handled by TCP. The 200 ms pauses in the flow, however, are due to whole-window loss induced TCP timeouts incuring the RTO_{min} penalty.

Previous work [13] has proposed a simple technique to mitigate the effects of incast-induced TCP timeouts: reduce RTO_{min}. We employed this technique in our datacenter, and the benefits can be seen in Figure 10 which shows TCP sequence plot of a flow experiencing incast. As with Figure 9, loss is visible, as is a timeout, but timeouts are reduced to 5 ms which is the minimum effective RTO_{min} that our servers support.

As shown in Figures 9 and 10, reducing RTO_{min} significantly improved the performance of TCP in our datacenter by mitigating the effect of TCP timeouts. It did

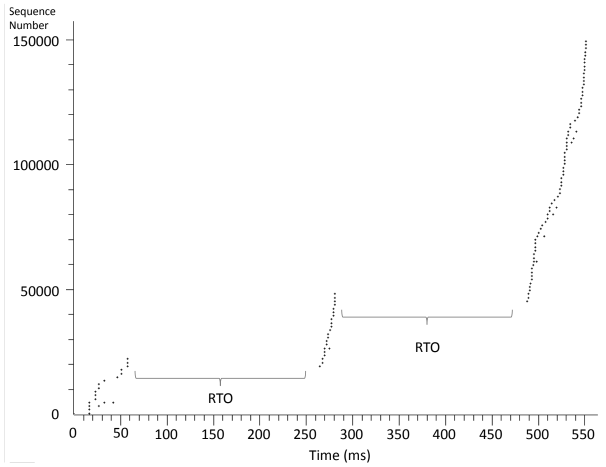

Figure 9: RTO_{min} 200 ms

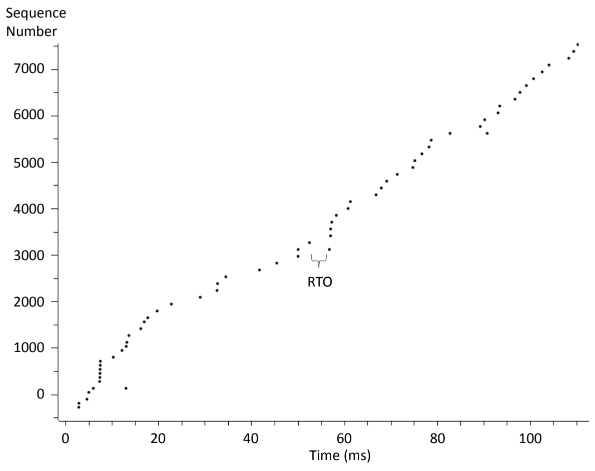

Figure 10: RTO_{min} 5 ms

not, however, prevent timeouts. In fact, the rate of packet loss in our network *increased* significantly after we applied the RTO_{min} change. This is expected as lowering the RTO_{min} does not prevent packet loss and timeouts, it just mitigates the effects. Moreover, lower timeout values will increase the number of contending flows which will tend to increase the overall number of lost packets.

In short, we found that reducing RTO_{min} greatly reduced the impact of incast on our applications. Network and server stability were not impacted by this change even when running on a cluster of over 2,000 servers. Innocent applications (applications not involved in the incast) were, however, still impacted. Moreover, network performance was still far from ideal. We were still incurring (much smaller) timeouts and the usual TCP latency. In the next section, we discuss addressing the root cause of incast.

6 DCTCP

Subsequent work on datacenter TCP has proposed several techniques to actually reduce packet losses due to incast, rather than just mitigate the effects of lost packets. Of these techniques, one of the most promising for deployment in our datacenter is DCTCP. DCTCP possesses several features that make it a particularly promising approach for us: it relies on capabilities that are available in current hardware and software, an implementation is available [8], and it does not contain features that we cannot use. (In particular, we decided against leveraging work that relies on flow priority or deadlines as our connections are long-lived and utilized for many different types of communication. As a result, communicating priority or deadline information to the network layer would be difficult or impossible for our applications.)

Our primary objectives for moving to DCTCP were to: eliminate TCP timeouts (or nearly eliminate them), reduce latency, and reduce the network-induced coupling of applications. In particular, we wanted to protect "innocent bystanders" from aggressive applications.

In the following sections, we first discuss obstacles to reaping these benefits, and how we extended DCTCP to overcome these obstacles, followed by some discussion of our extended DCTCP's performance.

7 DCTCP Deployment Challenges

7.1 Coexistence with TCP

In motivating the design of DCTCP, [1] states "[a datacenter] network is largely homogeneous and under a single administrative control. Thus backward compatibility, incremental deployment and fairness to legacy protocols are not major concerns." For actual usage in our datacenter, however, these are *all* major concerns. We do not have the luxury of a "big bang" deployment for several reasons.

- There are multiple applications running in our datacenter with distinct ownership. It is critical that one application moving to DCTCP does not negatively impact any application using conventional TCP. Recall that one of our major arguments for deploying DCTCP is to *reduce* the coupling of applications.
- Many critical services cannot be moved to DCTCP. Even for applications with owners willing to make the move to DCTCP, there are services used by those applications that we simply cannot move to DCTCP. For instance, many of our applications leverage file servers that do not support DCTCP.

Unfortunately, DCTCP and TCP do not naturally coexist well. To demonstrate this, we conducted a simple test where one TCP flow and one DCTCP flow both send at maximum rate from distinct servers to a single receiver. (Again, these experiments are conducted on a 10 Gbps network using iperf as sender and receiver.) The TCP flow lasts for a total of 20 seconds. The DCTCP flow lasts for 10 seconds and starts 5 seconds *after* the TCP flow starts.

The results are shown in Figure 11. As soon as the DCTCP flow starts, the TCP flow almost completely stops while the DCTCP flow completely saturates the link. This is an extremely negative result, and essentially the complete opposite of what we require. (Note that it is possible to delay the onset of this behavior through configuration settings, but this will not solve the fundamental problem.)

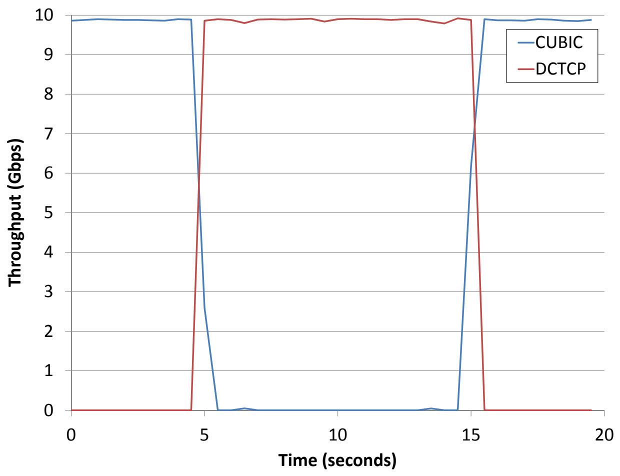

Figure 11: DCTCP Coexistence with TCP

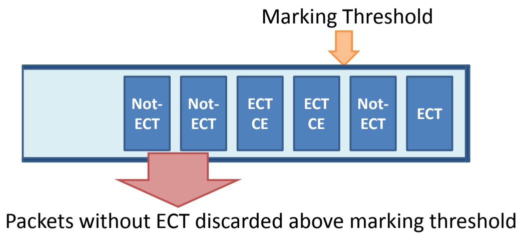

Figure 12: Switch RED ECN Implementation

The reason that DCTCP traffic dominates conventional TCP traffic is due to RED/ECN AQM behavior which is as follows for a switch configured for DCTCP. As depicted in Figure 12, when the switch queue length is below the marking threshold (there is only one threshold for DCTCP), any packet that arrives is simply queued irrespective of ECT status. When the queue length is over the marking threshold, however, all ECT packets are marked with CE, but non-ECT packets are *dropped*. In DCTCP, the marking threshold is set very low value to reduce queueing delay, thus a relatively small amount of congestion will exceed the marking threshold. During such periods of congestion, conventional TCP will suffer packet losses and quickly scale back *cwnd*. DCTCP, on the other hand, will use the fraction of marked packets to scale back *cwnd*. Only when all packets are marked will *cwnd* be scaled back as far as conventional TCP. Thus rate reduction in DCTCP will be much lower than that of conventional TCP, and DCTCP traffic will dominate conventional TCP traffic traversing the same link.

As both TCP and DCTCP must service the same servers in our network, we resort to utilizing IP DSCP bits to segregate DCTCP traffic from conventional TCP traffic. AQM is applied to DCTCP traffic, while TCP traffic is managed via drop-tail queueing.

7.2 Non-compliant switches

While we are fortunate enough to have support for ECN marking on our top-of-rack switches, this is the only location in our network that supports ECN marking. Higher-level switches are purely drop-tail. DCTCP must gracefully support transit over non-ECN switches without impacting either the behavior of DCTCP traffic or conventional traffic. Our tests show that DCTCP successfully resorts to loss-based congestion control when transiting a congested drop-tail link.

7.3 Non-technical challenges

Even without any technical challenges, altering the network in a major enterprise is a difficult undertaking. Network administrators are, necessarily, risk-averse. A reliable network is a business-critical requirement. Thus, network innovations are often viewed as presenting significantly more risk than reward.

We were able to present a compelling case for DCTCP implementation due to the following:

- Reduction in coupling. Application coupling was a known phenomenon in our datacenter. Conventional TCP's strong coupling of unrelated applications causes problems as discussed previously. DCTCP's promise to greatly reduce the coupling between applications meant that our network administrators would directly benefit from reduced troubleshooting requests from applications experiencing mysterious network performance issues caused by unrelated applications.
- Timing. We timed our DCTCP roll-out to coincide with the deployment of new network switches in our environment. We worked with our network administrators to ensure that the switch features necessary to support DCTCP were available from day one.
- Primum non nocere. Our support for conventional TCP and non-ECN compliant switches enabled us to guarantee that we would not harm existing applications.

7.4 Connection Establishment

Segregating DCTCP from conventional TCP removed one potential showstopper from our DCTCP deployment effort. Nevertheless, we encountered one other major problem in DCTCP that had the potential to prevent DCTCP adoption in our network: we found that under load, DCTCP would fail to establish network connections due to a lack of ECT in SYN and SYN-ACK packets.

[1] does not discuss setting ECT on SYN and SYN-ACK packets. The Stanford implementation [8] does *not* set ECT on either SYN or SYN-ACK packets. This is in line with RFC 3168 [14] which states *"A host MUST NOT set ECT on SYN or SYN-ACK packets."* RFC 5562 [10] (derived from ECN+ [9]) proposes setting ECT on SYN-ACK packets, but maintains the restriction of no ECT on SYN packets.

RFC 3168 and RFC 5562 prohibit ECT in SYN packets due to security concerns regarding malicious SYN packets with ECT set. These RFCs, however, are intended for general Internet use, and do not directly apply to DCTCP. In our internal network, we do not tolerate the compromised servers necessary for an attacker to send such packets. Moreover, the Stanford implementation's adoption of these RFCs likely owes more to its leveraging of the existing ECN support in Linux than anything else.

We find that setting ECT on SYN and SYN-ACK is critical for the practical deployment of DCTCP. Without this feature, SYN and SYN-ACK packets will be dropped whenever there is even minor congestion. As discussed in Section 7.1, and depicted in Figure 12, whenever the queue length is greater than the marking threshold, non-ECT packets are dropped. Thus, if SYN and SYN-ACK packets are non-ECT they will be dropped with high probability. We modified DCTCP to apply ECT to both SYN and SYN-ACK packets. We refer to this implementation as "DCTCP+" to distinguish it from the original DCTCP implementation. (Following the naming convention of ECN+ which extended ECN with ECT on SYN-ACK only.)

To measure the effect of this issue, we conducted an experiment where we disabled ECT for SYN packets and attempted to establish a DCTCP connection (with no SYN or SYN-ACK ECT) in the presence of a number of competing DCTCP+ flows which were already established and sending data at maximum rate. As shown in Figure 13, as the number of competing flows increases, it quickly becomes hard, then impossible, to establish a connection when SYN packets are non-ECT. Thus, we utilize DCTCP+ in our deployment, which marks both SYN and SYN-ACK packets as ECT.

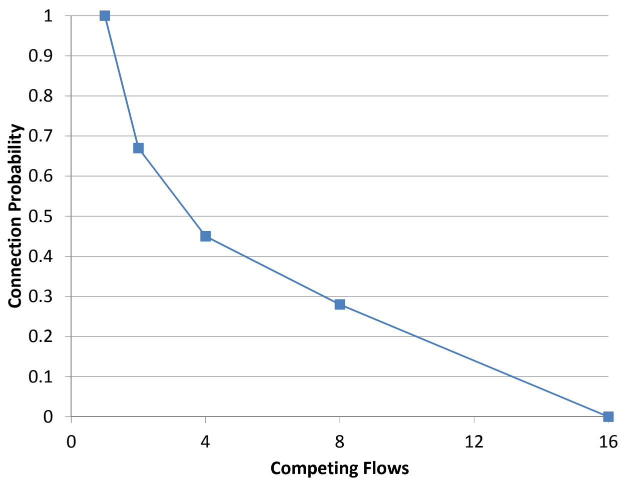

Figure 13: Connection Probability without SYN ECT

Note that given our support for conventional TCP, we could use DSCP to cause SYN and SYN-ACK packets *only* to be treated as conventional TCP. We do not take this approach as it would split packets from a single flow across two separate paths in our network which is highly undesirable.

8 DCTCP+ Performance

We now discuss several elements of DCTCP+ performance illustrating where DCTCP+ does well, and where there is room for improvement.

8.1 Incast Throughput and Fairness with Buffer Tuning Active

We first measured performance in an incast scenario similar to that in Figure 5. In this case, a single receiver received traffic from 19 senders for a total of 10 seconds (as discussed previously all experiments in this section are conducted on a 10 Gbps network). Importantly, automatic receive buffer tuning is *on* for this test; we will later show that this has a dramatic effect on TCP performance but very little for DCTCP+. Figures 14 and 15 show summarized throughput statistics for all 19 flows for each experiment. DCTCP+ fairly distributes the link bandwidth among flows resulting in a very narrow throughput distribution while fully utilizing the link. TCP is also able to fully utilize the link, but does so very inefficiently as flows stall due to a combination of packet loss and incorrectly sized receive buffers. The link is able to remain utilized, however, as other flows step in and utilize the missing bandwidth. Nevertheless, the median throughput is lower, and there is a large variation among flow throughput. In short, under DCTCP+, flow performance is fast and reliable while under TCP, packet loss and the poor performance of buffer auto tuning causes extremely variable throughput.

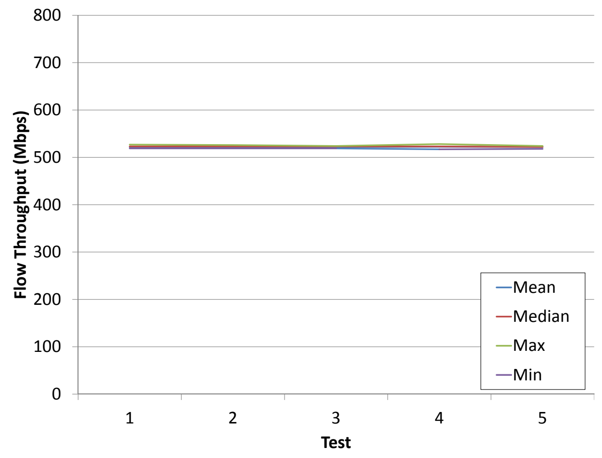

Figure 14: DCTCP single-receiver incast

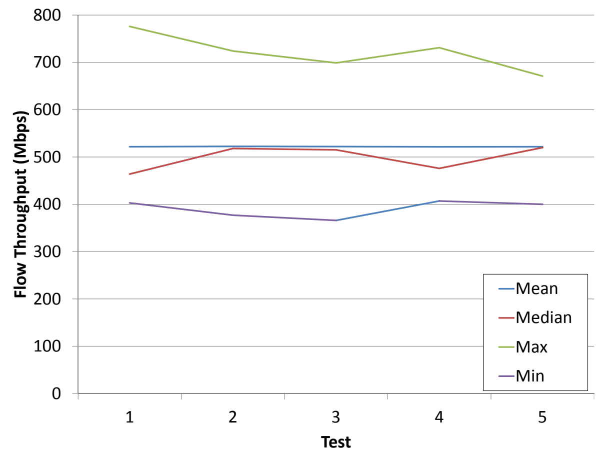

Figure 15: TCP single-receiver incast (buffer tuning active)

	TCP	DCTCP+
Mean	4.01	0.0422
Median	4.06	0.0395
Maximum	4.20	0.0850
Minimum	3.32	0.0280
σ	0.167	0.0106

Table 1: Per-packet latency in ms

Moreover, as shown in Table 1, per-packet latency under TCP is two orders of magnitude greater than per-packet latency under DCTCP+. DCTCP+'s reliably low latency enables higher-layer applications to reliably communicate in a very short time span. Under TCP (particularly before we lowered RTO_{min}), our applications needed added logic to deal with the unpredictable latency and throughput that incast induced. DCTCP+'s consistently superb performance make it a superior transport protocol to TCP within our datacenter.

8.2 Scale

The scalability afforded by datacenter computing lies at the heart of applications ranging from web search engines, to the Monte Carlo simulations and data analytics running in our datacenter. Realizing the benefits of scale, however, is challenging for many components of networked systems. For DCTCP+ to be an effective datacenter transport mechanism, it must scale with the applications that it supports.

In this section, we examine the scalability of DCTCP+ experimentally and analytically.

Incast traffic patterns are particularly difficult to scale. We examined DCTCP+ support at scale for incast by sending large numbers of long-lived flows (20 seconds) from many senders to a single receiver. Each flow was generated by a distinct server using iperf. Ideally, we should see that—as with TCP—the link would be fully utilized and each flow would receive a fair share of the link, but—unlike TCP—latency would remain low.

The results of this test are shown in Tables 2 and 3. Table 2 shows that throughput and long-term fairness are excellent through 500 servers. Table 3, however, exhibits some problems. The first problem to notice is that latency is relatively high even for 100 servers. At 300 servers, the high latency shows that the receive queue is entirely full, and at 400 servers significant amounts of traffic are lost and numerous timeouts are occurring. By 500 servers, 8.7% of packets sent are retransmissions, and timeouts are very significant; as a result, short term flow fairness will be poor. In a nutshell, it seems that at this scale, DCTCP+ is performing no better than TCP.

Senders	Total	Mean	Max	Min	σ
100	9,901	99.0	99.3	88.6	1.06
200	9,900	49.7	49.9	46.1	0.35
300	9,901	33.2	34	31.2	0.36
400	9,894	24.9	28.5	20.2	1.01
500	9,895	20.0	23.9	13.8	1.42

Table 2: Scale Test: Throughput (Mbps)

		Retransmissions		
Senders	RTT (ms)	Total	%	RTO
100	1.60	0	0	0
200	3.11	0	0	0
300	4.38	3	0	0
400	4.42	702	4.6	274
500	4.44	1110	8.7	655

Table 3: Scale Test: Latency and Retransmissions

Why is latency so high? Shouldn't the switch be marking packets causing DCTCP+ to back off before latency gets so high? Packet traces from a sender involved in this test show that for all cases, the switch is marking 100% of packets in steady state, yet DCTCP+ is still sending packets. In other words, even when the switch is telling DCTCP+ to fall back aggressively, DCTCP+ refuses to fall back enough to prevent congestion.

The source of this behavior is in the *cwnd* update procedure of DCTCP+. According to [1], DCTCP+ updates *cwnd* as:

$$cwnd \leftarrow cwnd \times (1 - \alpha/2)$$

Actual TCP implementations, however, are more intricate, and the Linux implementation in [8] updates *cwnd* as follows:

```
cwnd_new = max(tp->snd_cwnd
               - ((tp->snd_cwnd
                   * tp->dctcp_alpha)>>11),
               2U);
```

In other words, irrespective of measured congestion, DCTCP+ will always be willing to send two segments. This effectively puts a lower limit on DCTCP+ transmission rate per sender of:

$$TransmissionRate \geq \frac{SegmentSize \times 2}{RTT}$$

The resulting load for our scale test is shown in Table 4.

Senders	Load (Gbps)
100	3.27
200	6.55
300	9.82
400	13.09
500	16.36

Table 4: DCTCP+ Load vs. Scale

By 300 servers, load is nearly at the capacity of the link, and at higher scales, the load exceeds the link capacity. The result is the significant packet drops, retransmissions, and timeouts shown above. In effect, once the load due to the DCTCP+ minimum transmission rate exceeds the link capacity, DCTCP+ congestion control is no longer in effect, and TCP congestion control takes over. Hence, at scales higher than 300 in this test, DCTCP+ congestion control is no longer in effect.

DCTCP+ scale can be extended by reducing the minimum transmission rate per server. This can be done by applying the *cwnd* cap logic found elsewhere in the Linux TCP implementation.

```
cwnd_new = max(tp->snd_cwnd
               - ((tp->snd_cwnd
                   * tp->dctcp_alpha)>>11),
               1U);
cwnd_new = min(cwnd_new,
               tcp_packets_in_flight(tp) + 1U);
```

With this addition, under a congested network, only one packet will be allowed per RTT and the scaling will double – just over 600 servers can send at full rate to a single receiver without the minimum DCTCP+ transmission rate exceeding the link capacity.

While this change may result in additional delayed acknowledgements, our initial evaluation indicates that lowering the delayed acknowledgement timeout as discussed in Section 4 mitigates this concern. We leave a full evaluation for future work.

8.3 Operational Experience

We have been running DCTCP+ at a scale of approximately 600 servers for nearly one and a half years as of this writing. While quantifying the isolated benefits of DCTCP+ is ongoing work, qualitatively, we have found DCTCP+ to be a stable transport protocol and with the RTO_{min} reduction, delayed ACK reduction, and DCTCP+ all in place, we no longer observe any application-layer issues that are caused by TCP. This is a significant improvement.

9 Receive Buffer Tuning

Figures 6 and 7 previously showed an unexpected result: TCP converging more slowly than DCTCP, and generally performing very poorly. Careful analysis of Figures 17a&b in [1] shows that the creators of DCTCP observed similar behavior experimentally (though this particular behavior was not discussed in [1]): DCTCP outperforms TCP in their experiment with respect to stability, convergence, and short-term fairness. We repeat this convergence experiment in our network under several scenarios—first on a 1 Gbps network, then on a 10 Gbps network. The 1 Gbps result for TCP is shown in Figure 16. This closely matches the results from [1].

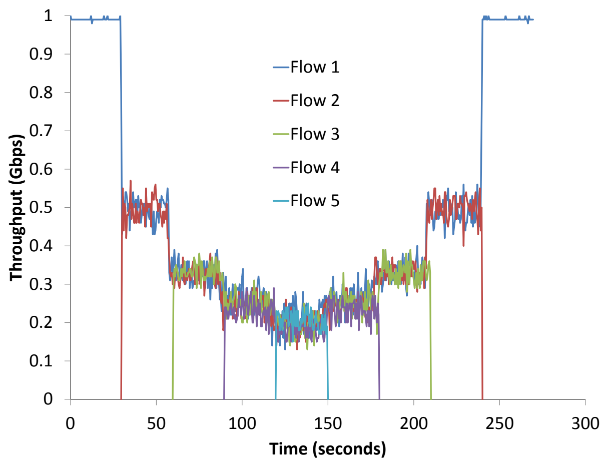

Figure 16: 1G, TCP, Buffer Tuning On

Figure 17 shows that moving to a 10 Gbps network exacerbates the problems with convergence, fairness and stability. We find that, as with the 1 Gbps result presented in [1], at 10 Gbps DCTCP+ convergence is superior to TCP convergence as shown in Figure 18.

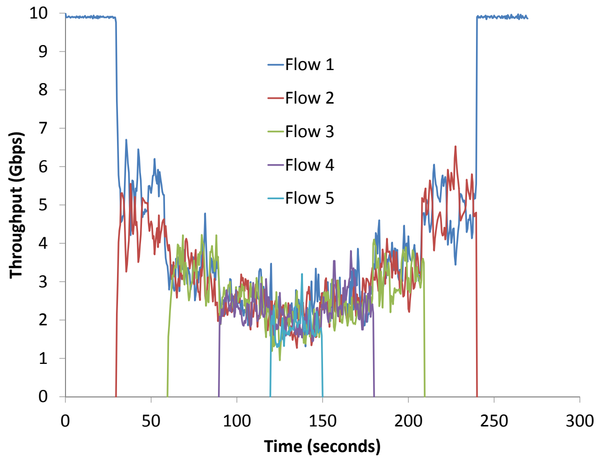

Figure 17: TCP, Buffer Tuning On

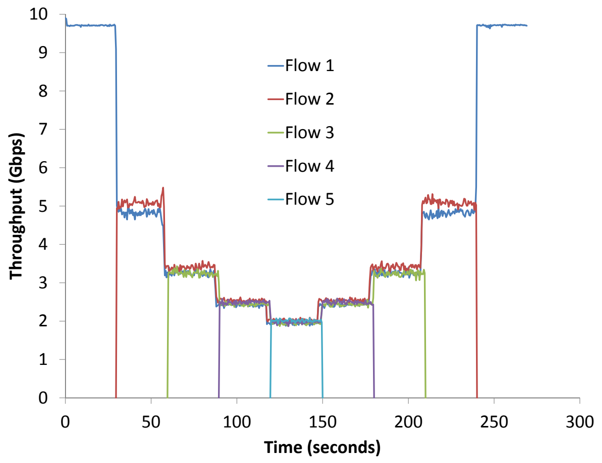

Figure 18: DCTCP+, Buffer Tuning On

These results seemingly defy [2] which showed DCTCP converging more slowly than TCP. The cause of this problem has a simple explanation: receive buffer tuning. Historically, network developers were tasked with setting TCP buffer sizes manually. Getting the buffer sizes right is important for both network and end-system performance: undersized buffers hurt network throughput; overly generous buffer sizes consume RAM, impact system performance, and limit application scale. It is possible to manually set buffer sizes to attempt to strike a balance, but this is very undesirable as it binds the performance of an application to the behavior of a particular network. Moreover, it fails to allow dynamic memory management to take into account a server's memory state.

To overcome these limitations, several approaches have been developed to dynamically set TCP buffer sizes. Unfortunately, in a datacenter setting, these algorithms can perform poorly. In principle, the receive buffer of an application should be set to the bandwidth delay product (BDP) of a link. The trouble is that inside of a datacenter, propagation delay is extremely small—approximately four orders of magnitude less than the queueing delay of a congested link! As a result, the bandwidth delay product of a link varies significantly, and—worse—is a function of the receive buffer size. The strong feedback present in receive buffer tuning a TCP link makes tuning a difficult problem. The tuning algorithm takes many seconds to adapt from the low-latency congestion-free regime to the high latency congested regime. As a result, TCP performance in our datacenter is very poor when automatic receive buffer tuning is enabled. This also explains why DCTCP+ is able to outperform TCP: DCTCP+ keeps latency far lower than TCP. As a result, the tuning algorithm experiences far less feedback and has a much easier time finding the correct buffer size.

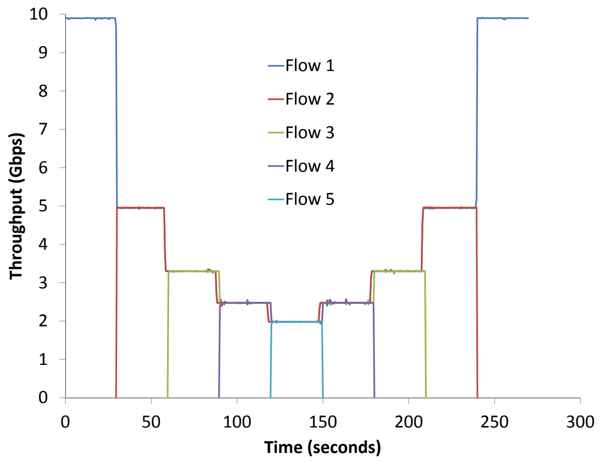

Figure 19: TCP, Buffer Tuning Off

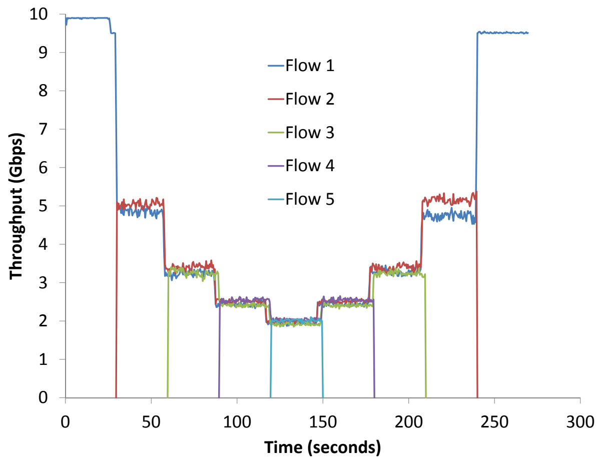

Figure 20: DCTCP+, Buffer Tuning Off

Turning off receive buffer tuning, and manually setting the receive buffer size to be greater than the maximum delay bandwidth product possible, results in much better behavior for TCP, as shown in Figure 19. With this change, TCP stability, convergence, and fairness all exceed that of DCTCP+. DCTCP+ performance, on the other hand, is not changed significantly by manually setting the buffer size, as shown in Figure 20.

In summary, receive buffer tuning can have a dramatic impact on TCP performance. The anomalous results shown in Figures 17a&b of [1], and discussed in this paper, are explained by poor tuning of the TCP receive buffer. With proper receive buffer sizing, TCP stability, convergence, and fairness outperform DCTCP+. Achieving proper receive buffer sizing, however, is much more difficult under TCP than DCTCP+ due to the massive dynamic range of latencies that even two competing flows can generate.

10 Related Work

TCP incast was first discussed by Nagle et al. [12]. Phanishayee et al. [13] explored solutions such as reducing RTO_{min}. Vasudevan et al. [17] proposed reducing RTO_{min} further using fine-grained timers. Instead of this, we simply reduced RTO_{min} as far as our kernel was capable of.

Yu et al. [20] analyze application performance in the datacenter network of an Internet service provider; they identify several performance problems caused by applications, the end-server network stacks, and the network itself. We independently have encountered similar problems in a completely different context, and we believe that the problems encountered in [20] are general problems likely to be found widely in datacenter communication. To fix the problems with delayed acknowledgements, [20] suggests either reducing the delayed ack timeouts or disabling delayed acks. Our work goes further by analyzing the tradeoff between these two options.

Wu et al. [19] also observe that switches running RED/ECN drop non-ECT packets, but do not discuss the impact of this behavior on DCTCP.

Semke et al. [16] developed a method of automatically tuning TCP buffers that is the basis of the current Linux autotuning algorithm.

There has been a good deal of work—such as [7] [18] [21]—on achieving superior congestion control than that attainable by DCTCP by incorporating knowledge of flow priorities and deadlines into congestion control. Unfortunately, these techniques are not readily applicable in our environment.

pFrabric [3] takes a clean-slate approach to datacenter communication. This is promising work, but outside of the scope of our work as we were restricted to techniques that we could run in production today.

11 Conclusion

TCP has been tremendously successful in the Internet, and is a ubiquitous protocol that is critical to countless applications. TCP support in datacenters promises to allow these applications to run alongside new applications. Unfortunately, however, experience has shown that TCP's design assumptions break down inside modern datacenters, and performance is often inadequate.

In this paper, we have shown that leveraging recent work overcomes the major deficiencies of TCP inside of the datacenter. We have shown that DCTCP coexistence with TCP is critical in our environment, and demonstrated how this can be accomplished. Moreover, we have shown how a small extension to DCTCP—employing ECT in SYN and SYN-ACK packets—removes a potentially fatal problem with DCTCP.

Nevertheless, this work has also highlighted areas for future work. Despite the dramatic impact on performance that it can have in current implementations, receive buffer auto tuning can perform very poorly. In addition, we have shown how DCTCP scale can be improved; ideally DCTCP would scale even further before filling queues and reverting to TCP.

In closing, deploying recently developed improvements to TCP (along with our extensions) has dramatically improved TCP performance in our datacenter, without requiring any modifications to our applications or distributed storage systems.

Acknowledgements

We would like to thank our reviewers for their feedback. In addition, we greatly appreciate the assistance of Daniel Varga, Wai-Hong Lui, Vikas Chawla, Eric Hagberg, John MacIntyre, Peter King, Johnathan Edwards, Sam Shteingart, Michael Jansen, Jason Greenberg, Matthew Whitehead, Daniel Borkmann, Florian Westphal, and Frank Hirtz,

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of SIGCOMM 2010*, 2010.

[2] M. Alizadeh, A. Javanmard, and B. Prabhakar. Analysis of dctcp: Stability, convergence, and fairness. In *Proceedings of SIGMETRICS 2011*, 2011.

[3] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of SIGCOMM 2013*, 2013.

[4] L. A. Barroso and U. Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[5] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement,

analysis, and implications. In *Proceedings of SIGCOMM 2011*, 2011.

[6] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 2008.

[7] C.-Y. Hong, M. Caesar, and P. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of SIGCOMM 2012*, 2012.

[8] A. Kabbani, M. Yasuda, and M. Alizadeh. Dctcp-linux. In *https://github.com/myasuda/DCTCP-Linux*, 2012.

[9] A. Kuzmanovic. The power of explicit congestion notification. In *Proceedings of SIGCOMM 2005*, 2005.

[10] A. Kuzmanovic, A. Mondal, S. Floyd, and K. Ramakrishnan. Adding explicit congestion notification (ecn) capability to tcp's syn/ack packets. In *RFC 5562*, 2009.

[11] D. Leith, R. Shorten, and G. McCullagh. Experimental evaluation of cubic-tcp. In *Proceedings of PFLDnet 2008*, 2008.

[12] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster - delivering scalable high bandwidth storage. In *Proceedings of Supercomputing 2004*, 2004.

[13] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *Proceeding of FAST 2008*, 2008.

[14] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ecn) to ip. In *RFC 3168*, 2001.

[15] J. Rothschild. High performance at massive scale: Lessons learned at facebook. In *mms://video-jsoe.ucsd.edu/calit2/JeffRothschildFacebook.wmv*.

[16] J. Semke, J. Mahdavi, and M. Mathis. Automatic tcp buffer tuning. In *Proceedings of SIGCOMM 1998*, 1998.

[17] V. Vasudevan, A. hanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proceedings of SIGCOMM 2010*, 2010.

[18] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review*, 2011.

[19] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang. Tuning ecn for data center networks. In *Proceedings of CoNEXT 2012*, 2012.

[20] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *Proceedings of NSDI 2011*, 2011.

[21] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: reducing the flow completion time tail in datacenter networks. In *Proceedings of SIGCOMM 2012*, 2012.