Extensions to FreeBSD Datacenter TCP for Incremental Deployment Support

Midori Kato
Fixstars Solutions
midori.kato@fixstars.com

Lars Eggert
NetApp
lars@netapp.com

Alexander Zimmermann
NetApp
alexandz@netapp.com

Rodney Van Meter
Keio University
rdv@sfc.keio.ac.jp

Hideyuki Tokuda
Keio University
hxt@sfc.keio.ac.jp

Abstract

Datacenter TCP (DCTCP) achieves low latencies for short flows while maintaining high throughputs for concurrent bulk transfers, but requires changes to both endpoints, which presents a deployment challenge. This paper presents extensions to DCTCP that enables one-sided deployment when peers implement standard TCP/ECN functionality. This makes DCTCP significantly easier to deploy incrementally. This paper also improves DCTCP in two-sided deployments by refining ECN processing and the calculation of the congestion estimate. A FreeBSD kernel implementation of these DCTCP improvements demonstrates better performance than the original DCTCP variant, and validates that incremental one-sided deployments see benefits similar to those previously only achievable in two-sided deployments.

1 Introduction

Datacenter workloads are diverse and range from web application hosting, content and media streaming, databases and virtual desktop infrastructures (VDI) to big-data analytics and other specialized applications. A common theme among these workloads is that their traffic mix consists of concurrent short interactive and long bulk-transfer flows. Efficiently supporting such a traffic mix requires network mechanisms that can maintain low and predictable latencies for short flows, absorb traffic bursts, and maintain high throughputs for bulk flows. Traditional TCP congestion control over a FIFO switch fabric fails to achieve these goals.

Datacenter TCP (DCTCP) [1, 2] is a recently proposed TCP variant that addresses this limitation. DCTCP assumes that Explicit Congestion Notification (ECN) [14] is enabled in the network fabric and uses this information to estimate network congestion more accurately than TCP. Based on this congestion estimate, DCTCP adjusts its sending rate with more precision than TCP, avoiding long standing queues that increase latencies. DCTCP first shipped in a production OS with Microsoft Windows Server 2012 and is enabled by default for paths with RTTs of less than 10 ms.

DCTCP is not the only recent proposal in this space. D^2TCP [15] and D^3 [18] provide deadline-aware flow scheduling, i.e., a stronger service model than DCTCP and TCP. VCP [19], RCP [7] and XCP [11] augment the switching fabric in order to derive even finer-grained congestion information. Other proposals [5, 12] extend ECN to enable new uses or return more accurate congestion information.

Many of these other proposals are difficult to deploy, because they require modifications to switch queuing mechanisms. Even DCTCP, which only requires standard ECN support from the network fabric, has deployment challenges, because both the sender and receiver must be modified, i.e., it requires a *two-sided* deployment. And DCTCP has no built-in negotiation mechanism to detect if a peer

supports it; a sender needs out-of-band information about whether a given peer supports DCTCP[1]. This is hurdle for incremental deployment, because as this paper shows in later sections, enabling DCTCP to communicate with a peer that does not support it can seriously reduce performance. This is problematic in particular for a network which contains expensive service such as storage appliance because the available protocol is restricted from operating system upgrading.

This paper addresses the deployment challenge to such networks by describing a variant of DCTCP in Section 3 that can be deployed *one-sided*, i.e., on only one of the endpoints of a connection.

As long as the other endpoint is ECN-capable, the one-sided variant achieves many of the same performance benefits of the original two-sided DCTCP. If the peer is DCTCP-capable, performance is increased over conventional DCTCP, due to our second contribution, which is presented in Section 4. We describe additional optimizations to the original two-sided DCTCP algorithm at startup and after an idle period, timeout or packet loss. Together, these modifications address DCTCP's deployment problem, because a sender can always enable our modified DCTCP, without risking performance degradation when communicating with legacy peers.

Section 5 presents an experimental evaluation of these improvements based on our FreeBSD kernel implementation of DCTCP, using a Cisco Nexus 3548 switch with support for ECN threshold marking. The results confirm that our modifications improve data transmission times and fairness between short and long flows for two-sided DCTCP, and demonstrate that a one-sided deployment of DCTCP is now feasible.

2 DCTCP Overview

DCTCP uses ECN in a non-standard way, in order to derive a more precise estimate of the congestion on the transmission path. A standard ECN-capable TCP endpoint regards ECN as a congestion event [14]. When an ECN-capable endpoint receives a packet with the "congestion experienced" (CE) bit set by the switch fabric, it echoes it back to the sender by setting the "Echo CE" (ECE) bit in *all* its acknowledgments (ACKs) until seeing a "congestion window reduced" (CWR) indication from the sender. On the other hand, a DCTCP receiver sends back information about incoming (or missing) CE marks by setting (or clearing) ECE marks whenever there is a change in the inbound CE bits [4].

The resulting stream of ECE marks allows the DCTCP sender to estimate the fraction of sent bytes that experienced congestion during a time period. The sender uses the fraction of marked bytes in the next congestion window calculation. It calculates congestion window ($cwnd$) per RTT as:

$$cwnd \leftarrow cwnd * \left(1 - \frac{\alpha}{2}\right)$$

The variable α is an estimator of the congestion on the path. Small values of α indicate light congestion; α approaching one indicates heavy congestion. When α equals one, the DCTCP sender behaves identical to a ECN-capable TCP sender. The value for α is computed as:

$$\alpha \leftarrow \alpha * (1 - g) + g * M$$

The value g is the estimation gain (a constant $\frac{1}{16}$ in the Windows Server 2012 implementation), and M is the fraction of bytes estimated to have experienced congestion during the last RTT.

DCTCP relies on an ECN-enabled switch fabric. Individual switches can employ different queueing mechanisms to generate ECN marks, such as RED [9] or BLUE [8]. Similar to the original DCTCP paper [2], this paper uses a simple threshold marking approach in Section 5, which marks CE when the instantaneous queue length exceeds a pre-defined threshold.

3 Enabling One-Sided DCTCP Deployment

This section proposes a variant of DCTCP that can be deployed only on one endpoint of a connection including the one-sided DCTCP problem explanation. A FreeBSD implementation, which contains the new variant will yield many of the benefits of a two-sided deployment as long as the peer is ECN-capable[2].

[1] This is contrast to most other TCP extensions, which negotiate their use in the SYN exchange.

[2] Most operating systems have been ECN-capable for years; support can typically enabled through a simple configuration change.

3.1 Emitting CWRs at One-Sided Senders

A severe performance issue can manifest when a regular DCTCP sender communicates with a non-DCTCP ECN-capable peer. The DCTCP paper [2] and specification [4] do not require that the sender set the CWR flag, probably because a DCTCP receiver would only ignore it. When sending to a non-DCTCP ECN-capable receiver, never sending CWR causes the receiver to keep emitting ACKs with the ECE mark set while waiting for a CWR mark to arrive. This ECE-marked ACK stream in turn appears to the DCTCP sender as if *all* sent bytes experienced congestion, causing it to shrink $cwnd$ (and hence throughput) to the minimum.

In order to facilitate one-sided deployment, a DCTCP sender should set the CWR mark after receiving an ECE-marked ACK once per RTT. This avoids the performance issue identified above. It is also safe in two-sided deployments, because a regular DCTCP receiver will simply ignore the CWR mark. Our FreeBSD kernel implementation therefore always enables this modification.

With this modification, DCTCP performance when communicating with a ECN-capable non-DCTCP receiver becomes similar to that achievable in a two-sided deployment of DCTCP. The ECE stream of ECN-capable TCP receiver in the ACK stream makes *cwnd* of the one-sided DCTCP sender set to the relative small value. However, it never leads to the significant impact of the DCTCP performance.

3.2 Delayed ACKs at One-Sided Receivers

As mentioned in the previous section, the standard ECN-capable TCP endpoint handles ECN as a congestion event. Because of this, a congestion reaction to the ECN-capable endpoint is in the same manner of the data retransmission. It refrains from new data transmission for an RTT which corresponds to the duration responding to ECE. The FreeBSD kernel name this duration “congestion recovery”.

After congestion recovery, ECN-capable TCP start increasing window size. The size depends on the sender detection of the use of delayed ACK at receiver side. Delayed ACK sends an ACK every m packets[3] reaches, it allows a sender to increase $cwnd$ by two full-sized segments (SMSS); otherwise, by one SMSS [3].

[3] FreeBSD sets m to two

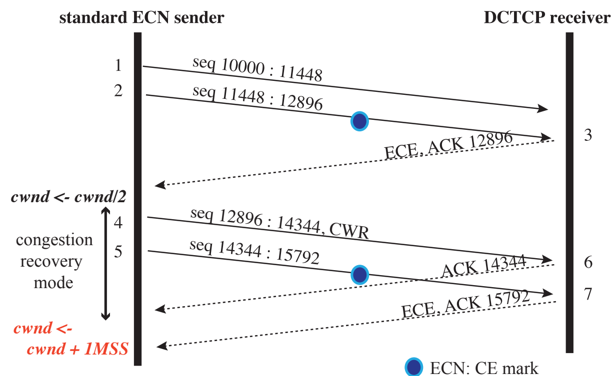

Figure 1: One-sided DCTCP sequence diagram illutrating problematic behavior.

This window increase adjustment rises to another performance issue when a ECN-capable TCP sender is communicating with a DCTCP receiver. Figure 1 shows the problematic situation. This in turn causes the sender to halve $cwnd$ and, enter and exit congestion recovery. The sender, then, starts increasing $cwnd$. The ACK behavior of DCTCP can cause mis-detection at the sender. Here, the non-DCTCP sender incorrectly determines that delayed ACKs are not in use, and hence increases $cwnd$ by one instead of two SMSS. This is the minor point but we found this leads lower throughput compared to ECN capable TCP, which no user expects.

In order to eliminate this performance issue and facilitate one-sided deployment, a one-sided DCTCP receiver should always delay an ACK for incoming packets marked with CWR, which is the only indication of congestion recovery exit. This modification impacts the accuracy of calculating α for two-sided DCTCP flows, however, this is only of minor concern, because α is calculated as a weighted average. Therefore, our FreeBSD kernel implementation always delays ACKs for incoming CWR packets.

4 Improving Two-Sided DCTCP

This section motivates and describes additional improvements to DCTCP that are unrelated to enabling one-sided deployment, also enabled in our FreeBSD implementation.

4.1 ECE Processing

A congestion control reaction in TCP can be triggered by several events: First, an out-of-order ACK will trigger *fast recovery* [10]. Second, an ACK with an ECE mark will trigger *congestion recovery*. (Other events include timeouts or receiving multiple duplicate ACKs.) When one of these two cases happens, a sender stops increasing $cwnd$ for roughly one RTT.

For DCTCP, there is no reason for this behavior. DCTCP does not use ECN marks to detect congestion *events*, it uses ECN marks to estimate and quantify the *extent* of congestion, and then reacts proportionally to that extent. Therefore, there is no need to stop $cwnd$ from increasing. In addition, a DCTCP sender frequently enters congestion recovery, because it tries to adapt to the available path bandwidth, aggravating the issue. Our FreeBSD implementation therefore does not enter congestion recovery when receiving an ECE mark for two-sided DCTCP flows.

4.2 Congestion Estimation

The DCTCP algorithm [2] can be improved in terms of managing α; namely, how α is initialized, and how α is calculated after an idle period, timeout or packet loss.

4.2.1 Initialization of α

According to the draft DCTCP specification [4], α is conservatively initialized to one. This initial value determines how DCTCP reacts when receiving the first CE mark. When α is one, $cwnd$ will be halved on the first CE mark. This is the most conservative behavior, because it causes a reaction as if *all* bytes sent during the last RTT experienced congestion.

An initial α of one may be a good choice for latency-sensitive applications, because potential queue buildup is minimized. However, for throughput-sensitive applications, a smaller initial α may be preferred, so that $cwnd$ is not reduced quite as drastically.

Our FreeBSD implementation allows flows to select an initial α that is appropriate for them, and defaults to setting α to zero.

4.2.2 Idle Periods

A second related improvement opportunity is if – and how – α should be re-initialized after an idle period. The draft DCTCP specification [4] does not currently discuss this issue. The choices are to re-initialize α to the value used at the beginning of the connection, to keep using the α that was in effect before the idle period, or to age or otherwise adjust that α.

There are tradeoffs surrounding this choice, similar to the TCP "slow-start restart" [17] issue. If the path characteristics after an idle period are similar to before it started, using the last α is reasonable. Otherwise, re-initializing α to a more conservative value is safer. Our FreeBSD implementation chooses to re-initialize α after an idle period longer than the retransmission timeout (RTO)[4].

4.2.3 Timeouts and Packet Loss

Because DCTCP tries hard to avoid queue buildup and overflow, the probability for packet losses or TCP timeouts is lower than for standard TCP. However, it is still important that the algorithm handle packet loss correctly.

The draft DCTCP specification [4] defines the update interval for α as one RTT. In order to track whether this interval has expired, DCTCP compares received ACKs against the sequence number of outgoing packets. This simple approach is problematic when a packet loss occurs. In that case, the incoming (cumulative) ACKs never changes until a retransmission arrives at the receiver, and its ACK arrives back at the sender. This process can take two or more RTTs, during which DCTCP will not update α.

To avoid this problem, DCTCP should update α when it detects either a duplicate ACK or a timeout. This addition is part of our FreeBSD kernel implementation.

5 Experimental Evaluation

This section evaluates the performance of the proposed DCTCP modifications for one-sided and two-sided deployments through experiments with a FreeBSD imple-

[4] Whether 1 RTO is an appropriate amount of time is an open question, cf. [17]; this value was chosen because of ease of implementation in FreeBSD.

mentation in a testbed, using `flowgrind` [20] as a traffic generator.

The testbed consists of four x86 machines running FreeBSD-CURRENT and one Cisco Nexus 3548 switch. Each machine is equipped with two dual-core Intel Xeon E5240 CPUs running at 3 GHz, 16 GB RAM and one four-port Intel PRO/1000 1 G Ethernet card.

Three machines act as senders (S1–S3) and one hosts two receiver processes (R1–R2) on two different IP addresses assigned to the same network interface. S1 sends data to R1. S2 and S3 send data to R2. The Nagle algorithm [13] was disabled, and the TCP host cache is flushed before each run.

The Cisco Nexus 3548 shares a large buffer pool among four adjacent ports [6]. Due to its deep buffers and a low configured ECN marking threshold of 20 packets, no packet loss occurs during the experiments when DCTCP is used.

The experiments focus on two different scenarios, an *incast* scenario and a *bulk transfer* scenario. In all figures, error bars illustrate the standared deviation. In the *incast* [16] scenario, many flows start at the same time and converge in the same egress queue of the switch. If these flows use regular TCP, they overflow the queue and experience packet loss. The experiment creates an incast situation by starting ten flows at the same time. The transfer size of the flows is one parameter of the experiment and varies between 10–800 KB. The metrics for this experiment are the transmission time and the smoothed RTT (SRTT)[5] as calculated by TCP.

The second *bulk transfer* scenario models a case where short latency-sensitive flows share a path with several bulk transfers. With regular TCP, short flows running concurrently with bulk transfers take longer to finish than when they run in isolation, due to longer standing queues and a therefore higher propagation delay. The desired behavior for short flows is short completion times, whereas high throughputs should be maintained for bulk transfers. The experiment runs ten short flows in parallel to eight bulk transfers. The start times of the short flows are staggered by 500 ms, which is long enough for them to reach steady-state. The transfer size of the bulk transfers is 40 MB, the transfer size of short flows is one parameter. The metric of interest in this scenario is the transmission time.

[5] The SRTT approximates the queuing delay at the switch; however, the timer resolution of the FreeBSD kernel is 1 ms, which limits the accuracy of the estimation.

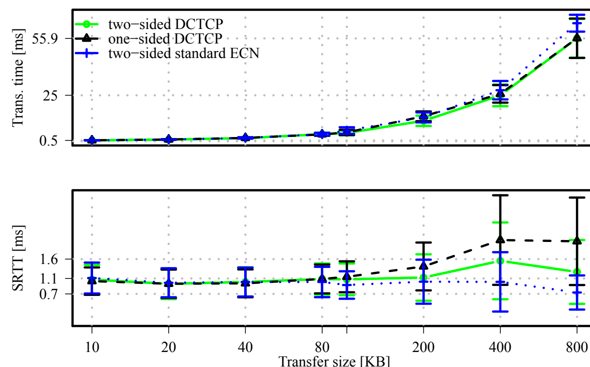

Figure 2: Incast results for one-sided DCTCP senders, transmission times (top) and SRTT (bottom).

5.1 Results for One-Sided DCTCP

The evaluation for one-sided deployments is split into results for one-sided DCTCP senders and for one-sided DCTCP receivers; the peer always uses standard TCP with ECN. As a baseline, results where both sides use standard TCP with ECN are also included. We also compare these results to those obtained for two-sided deployments of original DCTCP and of standard TCP with ECN.

5.1.1 One-Sided DCTCP Senders

The *incast* results in Figure 2 show that one-sided DCTCP achieves a performance that is identical to two-sided DCTCP. When compared to standard TCP with ECN, both one- and two-sided DCTCP have a somewhat higher queuing delay, while they still achieve short transmission times. The higher queuing delay results from DCTCP behavior after slow start. Because one- and two-sided DCTCP react to congestion based on the available bandwidth at the bottleneck link, they have a higher queuing delay until they converge on an adjusted value for $cwnd$.

Figure 3 shows the results for the *bulk transfer* scenario, when eight bulk transfers are running concurrently with ten short interactive flows. The performance of the bulk transfers show no significant difference in any of the cases. This validates that DCTCP is able to achieve bulk transfer

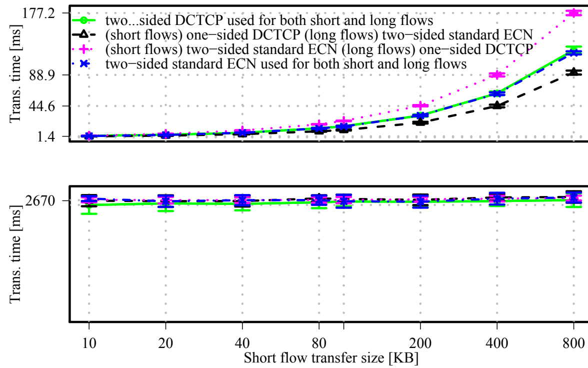

Figure 3: Bulk transfer results for one-sided DCTCP senders, transmission times of short flows (top) and eight bulk transfers (bottom).

throughputs that are comparable with standard TCP with ECN.

When one-sided DCTCP is used for short flows, they experience up to 28 ms (23 %) shorter transmission times than over a two-sided deployment of standard TCP with ECN. Because one-sided DCTCP senders react to congestion more accurately, their short flows take bandwidth from bulk transfers over standard TCP with ECN, and completes more quickly.

When one-sided DCTCP is used for bulk transfers, short flows using standard TCP with ECN experience up to 57 ms (32 %) longer transmission times than when the bulk transfers use also standard TCP with ECN. The DCTCP bulk transfers use all available bandwidth when the short flows using standard TCP with ECN start up, and they hence take a longer time to complete.

The results of the incast and bulk transfer scenarios validate that one-sided DCTCP senders achieve a performance that is very similar to that of a two-sided DCTCP deployment. In the incremental deployment path, although competitive ECN-capable TCP flows must compromise transmission time, one-sided DCTCP shows similarity to two-sided ECN-capable TCP/DCTCP deployment.

5.1.2 One-Sided DCTCP Receivers

The *incast* results in Figure 4 show that one-sided DCTCP receivers remark identical performance to that of a two-sided deployment of standard TCP with ECN. This validates that as expected, our modified DCTCP receivers behave like ECN-capable TCP receivers in a partial deployment when communicating with ECN-capable TCP senders.

Figure 5 shows the results for eight bulk transfers in parallel with ten short flows. The performance of the bulk transfers is almost identical in all cases, as expected. The performance of the short flows, however, differs. For example, in 800 KB data transmission, short flows to a one-sided DCTCP receiver takes extra 25 ms (17 %) compared to the transmission time of short flows to a ECN-capable TCP receiver. When one-sided DCTCP is used for bulk transfers, the transmission time is reduced.

This is caused by disabling delayed ACKs and at the same time configuring a low ECN marking threshold at the switch. Why we can say it because a one-sided DCTCP receiver behaves as a ECN-capable TCP receiver except for delayed ACK. Although – in theory – the number of transmitted packets is same with or without delayed ACKs [3], when the switch uses a low ECN marking threshold, the number of packets the sender transmits at any time affects the probability to receive CE marks. A sender that communicates with a peer using delayed ACKs transmits a larger burst of packets on each ACK compared to when a receiver is not using delayed ACKs. With a relatively large number of bulk transfers (for a given bandwidth) and receivers using delayed ACKs, a single bulk transfer can easily occupy the entire queue. The result is that the flows not using delayed ACKs tend to receive more CE marks, reduce their $cwnd$ more, and hence have longer transmission times.

We see the same transmission times in the bulk transfer scenario that the number of short flows is changed from eight to two From this result, we can conclude that one-sided DCTCP achieves similar transmission times only when the number of bulk transfers is small. The tradeoffs surrounding the use of delayed ACKs at one-sided DCTCP receivers must be better understood, which is future work.

Overall, the experimental results show that one-sided DCTCP flows are fair to concurrent two-sided ECN flows in an incremental deployment secenario. In the result of incast experiment, one-sided DCTCP shows identically same plots to ECN-capable TCP. In bulk transfer experiments, plots differs slightly between one-sided DCTCP and ECN-capable TCP. But this new finding will be a hint to complete a DCTCP design as a protocol.

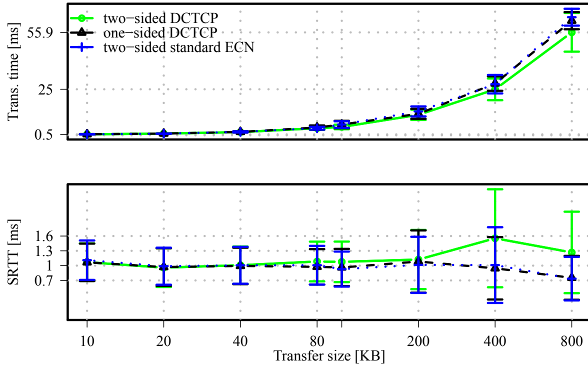

Figure 4: Incast results for one-sided DCTCP receivers, transmission times (top) and SRTT (bottom).

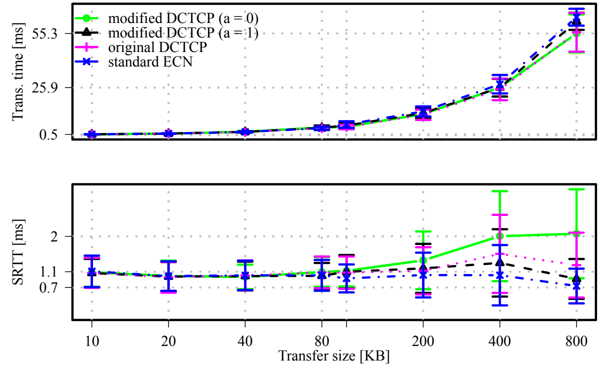

Figure 6: Incast results for two-sided DCTCP, transmission times (top) and SRTT (bottom).

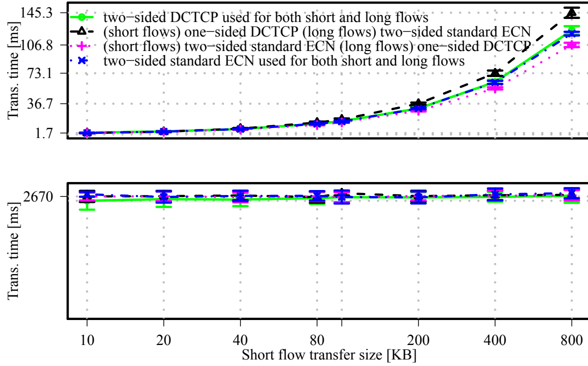

Figure 5: Bulk transfer results for one-sided DCTCP receivers, transmission times of short flows (top) and eight bulk transfers (bottom).

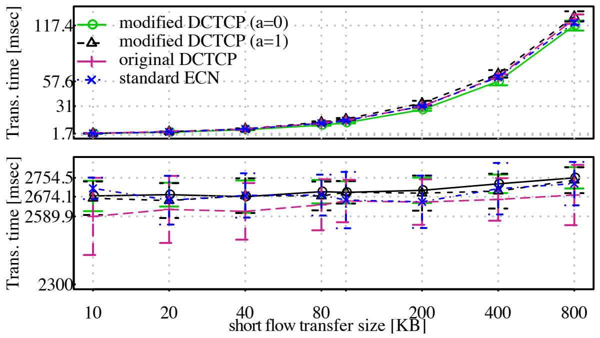

Figure 7: Bulk transfer results for two-sided DCTCP, transmission times of short flows (top) and eight bulk transfers (bottom).

5.2 Results for Two-Sided DCTCP

The focus of the evaluation of two-sided DCTCP deployments is on validating our modifications to ECE processing (see Section 4.1) and the selection of the initial α (see Section 4.2).

In both the incast and the bulk transfer scenarios, four results are plotted in each figure, as labeled. The first two explore the effect of different initial α values (zero and one) on our modified DCTCP. The other results – two-sided standard TCP with ECN and original DCTCP – are used as baselines for comparison.

Figure 6 shows the incast results. The performance of our modified DCTCP with an initial α of zero is very similar to that of the original DCTCP (less than 1 ms difference.) With an initial α of one, our modified DCTCP almost behaves like a standard TCP with ECN (less than 3 ms difference.) We conclude that the modifications to ECE processing have a very small benefit in the incast scenario.

Figure 7 shows the results of the bulk transfer secnario with eight bulk flows. Overall, there is less than 5 % difference in the transfer times of short flows. Our modified DCTCP behaves very similar to the original DCTCP (less than 10 ms difference.) For the bulk transfers, our modified DCTCP shows has a 110 ms (2 %) longer transmission time than the original DCTCP, but also has a substantially lower variability in the result, with the standard deviation

declining from 6 % to 2 %. Because the modified ECE processing never stops $cwnd$ from increasing, modified DCTCP bulk transfers achieve higher throughputs than bulk transfers using traditional DCTCP .

When the initial α is set to one, our modified DCTCP behaves like standard TCP with ECN. The difference in performance for short flows is at most 4 ms (7 %); bulk transfer performance is similar.

These experimental results validate that the modified ECE processing has benefits for the transfer time of bulk transfers. When the initial α is one, the transmission time is improved by a very small amount compared to standard TCP with ECN, while the queuing latency is similar. This validates that our modified DCTCP with an initial α of one behaves like standard TCP with ECN.

6 Conclusion

This paper presented modifications to DCTCP that make it more incrementally deployable in environments where some peers may use regular ECN-enabled TCP. It also presented some general performance improvements to DCTCP. The experimental evaluation of our FreeBSD kernel implementation, which is slated to be merged into FreeBSD-CURRENT in the near future, validates our hypothesis that the proposed modifications improve performance for two-sided DCTCP deployments, and improve the performance of one-sided deployments to a point where it is comparable to the two-sided case, while being easier to deploy incrementally.

7 Acknowledgements

We thank Mirja Kühlewind, Richard Scheffenegger and Michio Honda for feedback on ECN and DCTCP, Cisco for donating a Nexus 3548 for experimentation, and Lucien Avramov from Cisco for help with the switch configuration. We also appreciate to Hiren Panchasara for the deligent support when we merge our DCTCP impelementation into the FreeBSD kernel.

8 References

[1] M. Alizadeh et al. "Analysis of DCTCP: Stability, Convergence, and Fairness". In: *Proc. ACM SIGMETRICS*. 2011, pp. 73–84.

[2] M. Alizadeh et al. "Data Center TCP (DCTCP)". In: *Proc. ACM SIGCOMM*. 2010, pp. 63–74.

[3] M. Allman. *TCP Congestion Control with Appropriate Byte Counting (ABC)*. RFC 3465. IETF, Feb. 2003.

[4] S. Bensley et al. *Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters*. Internet-Draft draft-bensley-tcpm-dctcp-00. Work in Progress. IETF, Feb. 2014.

[5] B. Briscoe et al. *Re-ECN: A Framework for adding Congestion Accountability to TCP/IP*. Internet-Draft draft-briscoe-conex-re-ecn-motiv-02. Work in Progress. IETF, July 2013.

[6] *Cisco Nexus 3548 Switch Architecture*. Tech. rep. C11-715262-01. Cisco, Sept. 2012.

[7] N. Dukkipati et al. "Processor Sharing Flows in the Internet". In: *Proc. IWQoS*. 2005, pp. 271–285.

[8] W.-c. Feng et al. "The BLUE Active Queue Management Algorithms". In: *IEEE/ACM Trans. Netw.* 10.4 (Aug. 2002), pp. 513–528.

[9] S. Floyd et al. "Random Early Detection Gateways for Congestion Avoidance". In: *IEEE/ACM Trans. Netw.* 1.4 (Aug. 1993), pp. 397–413.

[10] T. Henderson et al. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582. IETF, Apr. 2012.

[11] D. Katabi et al. "Congestion Control for High Bandwidth-delay Product Networks". In: *Proc. ACM SIGCOMM*. 2002, pp. 89–102.

[12] M. Kühlewind et al. *More Accurate ECN Feedback in TCP*. Internet-Draft draft-kuehlewind-tcpm-accurate-ecn-02. Work in Progress. IETF, June 2013.

[13] J. Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. IETF, Jan. 1984.

[14] K. K. Ramakrishnan et al. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. IETF, Sept. 2001.

[15] B. Vamanan et al. "Deadline-aware Datacenter TCP (D2TCP)". In: *Proc. ACM SIGCOMM*. 2012, pp. 115–126.

[16] V. Vasudevan et al. "Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication". In: *Proc. ACM SIGCOMM*. 2009, pp. 303–314.

[17] V. Visweswaraiah et al. *Improving Restart of Idle TCP Connections*. Tech. rep. 97-661. University of Southern California, Nov. 1997.

[18] C. Wilson et al. "Better Never Than Late: Meeting Deadlines in Datacenter Networks". In: *Proc. ACM SIGCOMM*. 2011, pp. 50–61.

[19] Y. Xia et al. "One More Bit is Enough". In: *Proc. ACM SIGCOMM*. 2005, pp. 37–48.

[20] A. Zimmermann et al. "Flowgrind – A New Performance Measurement Tool". In: *Proc. IEEE GLOBECOM*. Dec. 2010, pp. 1–6.