Internet Engineering Task Force (IETF)
Request for Comments: 8257
Category: Informational
ISSN: 2070-1721

S. Bensley
D. Thaler
P. Balasubramanian
Microsoft
L. Eggert
NetApp
G. Judd
Morgan Stanley

October 2017

Data Center TCP (DCTCP): TCP Congestion Control for Data Centers

Abstract

This Informational RFC describes Data Center TCP (DCTCP): a TCP congestion control scheme for data-center traffic. DCTCP extends the Explicit Congestion Notification (ECN) processing to estimate the fraction of bytes that encounter congestion rather than simply detecting that some congestion has occurred. DCTCP then scales the TCP congestion window based on this estimate. This method achieves high-burst tolerance, low latency, and high throughput with shallow-buffered switches. This memo also discusses deployment issues related to the coexistence of DCTCP and conventional TCP, discusses the lack of a negotiating mechanism between sender and receiver, and presents some possible mitigations. This memo documents DCTCP as currently implemented by several major operating systems. DCTCP, as described in this specification, is applicable to deployments in controlled environments like data centers, but it must not be deployed over the public Internet without additional measures.

Status of This Memo

Table of Contents

1. Introduction

Large data centers necessarily need many network switches to interconnect their many servers. Therefore, a data center can greatly reduce its capital expenditure by leveraging low-cost switches. However, such low-cost switches tend to have limited queue capacities; thus, they are more susceptible to packet loss due to congestion.

Network traffic in a data center is often a mix of short and long flows, where the short flows require low latencies and the long flows require high throughputs. Data centers also experience incast bursts, where many servers send traffic to a single server at the same time. For example, this traffic pattern is a natural consequence of the MapReduce [MAPREDUCE] workload: the worker nodes complete at approximately the same time, and all reply to the master node concurrently.

These factors place some conflicting demands on the queue occupancy of a switch:

- The queue must be short enough that it does not impose excessive latency on short flows.
- The queue must be long enough to buffer sufficient data for the long flows to saturate the path capacity.
- The queue must be long enough to absorb incast bursts without excessive packet loss.

Standard TCP congestion control [RFC5681] relies on packet loss to detect congestion. This does not meet the demands described above. First, short flows will start to experience unacceptable latencies before packet loss occurs. Second, by the time TCP congestion control kicks in on the senders, most of the incast burst has already been dropped.

[RFC3168] describes a mechanism for using Explicit Congestion Notification (ECN) from the switches for detection of congestion. However, this method only detects the presence of congestion, not its extent. In the presence of mild congestion, the TCP congestion window is reduced too aggressively, and this unnecessarily reduces the throughput of long flows.

Data Center TCP (DCTCP) changes traditional ECN processing by estimating the fraction of bytes that encounter congestion rather than simply detecting that some congestion has occurred. DCTCP then scales the TCP congestion window based on this estimate. This method achieves high-burst tolerance, low latency, and high throughput with shallow-buffered switches. DCTCP is a modification to the processing of ECN by a conventional TCP and requires that standard TCP congestion control be used for handling packet loss.

DCTCP should only be deployed in an intra-data-center environment where both endpoints and the switching fabric are under a single administrative domain. DCTCP MUST NOT be deployed over the public Internet without additional measures, as detailed in Section 5.

The objective of this Informational RFC is to document DCTCP as a new approach (which is known to be widely implemented and deployed) to address TCP congestion control in data centers. The IETF TCPM Working Group reached consensus regarding the fact that a DCTCP standard would require further work. A precise documentation of running code enables follow-up Experimental or Standards Track RFCs through the IETF stream.

This document describes DCTCP as implemented in Microsoft Windows Server 2012 [WINDOWS]. The Linux [LINUX] and FreeBSD [FREEBSD] operating systems have also implemented support for DCTCP in a way that is believed to follow this document. Deployment experiences with DCTCP have been documented in [MORGANSTANLEY].

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Normative language is used to describe how necessary the various aspects of a DCTCP implementation are for interoperability, but even compliant implementations without the measures in Sections 4-6 would still only be safe to deploy in controlled environments, i.e., not over the public Internet.

3. DCTCP Algorithm

There are three components involved in the DCTCP algorithm:

- o The switches (or other intermediate devices in the network) detect congestion and set the Congestion Encountered (CE) codepoint in the IP header.

- o The receiver echoes the congestion information back to the sender, using the ECN-Echo (ECE) flag in the TCP header.

- o The sender computes a congestion estimate and reacts by reducing the TCP congestion window (cwnd) accordingly.

3.1. Marking Congestion on the L3 Switches and Routers

The Layer 3 (L3) switches and routers in a data-center fabric indicate congestion to the end nodes by setting the CE codepoint in the IP header as specified in Section 5 of [RFC3168]. For example, the switches may be configured with a congestion threshold. When a packet arrives at a switch and its queue length is greater than the congestion threshold, the switch sets the CE codepoint in the packet. For example, Section 3.4 of [DCTCP10] suggests threshold marking with a threshold of K > (RTT * C)/7, where C is the link rate in packets per second. In typical deployments, the marking threshold is set to be a small value to maintain a short average queueing delay. However, the actual algorithm for marking congestion is an implementation detail of the switch and will generally not be known to the sender and receiver. Therefore, the sender and receiver should not assume that a particular marking algorithm is implemented by the switching fabric.

3.2. Echoing Congestion Information on the Receiver

According to Section 6.1.3 of [RFC3168], the receiver sets the ECE flag if any of the packets being acknowledged had the CE codepoint set. The receiver then continues to set the ECE flag until it receives a packet with the Congestion Window Reduced (CWR) flag set. However, the DCTCP algorithm requires more-detailed congestion information. In particular, the sender must be able to determine the number of bytes sent that encountered congestion. Thus, the scheme described in [RFC3168] does not suffice.

One possible solution is to ACK every packet and set the ECE flag in the ACK if and only if the CE codepoint was set in the packet being acknowledged. However, this prevents the use of delayed ACKs, which are an important performance optimization in data centers. If the delayed ACK frequency is n, then an ACK is generated every n packets.

The typical value of n is 2, but it could be affected by ACK throttling or packet-coalescing techniques designed to improve performance.

Instead, DCTCP introduces a new Boolean TCP state variable, DCTCP Congestion Encountered (DCTCP.CE), which is initialized to false and stored in the Transmission Control Block (TCB). When sending an ACK, the ECE flag MUST be set if and only if DCTCP.CE is true. When receiving packets, the CE codepoint MUST be processed as follows:

1. If the CE codepoint is set and DCTCP.CE is false, set DCTCP.CE to true and send an immediate ACK.

2. If the CE codepoint is not set and DCTCP.CE is true, set DCTCP.CE to false and send an immediate ACK.

3. Otherwise, ignore the CE codepoint.

Since the immediate ACK reflects the new DCTCP.CE state, it may acknowledge any previously unacknowledged packets in the old state. This can lead to an incorrect rate computation at the sender per Section 3.3. To avoid this, an implementation MAY choose to send two ACKs: one for previously unacknowledged packets and another acknowledging the most recently received packet.

Receiver handling of the CWR bit is also per [RFC3168] (including [Err3639]). That is, on receipt of a segment with both the CE and CWR bits set, CWR is processed first and then CE is processed.

```
                             Send immediate
                             ACK with ECE=0
                 .-----.     .--------------.     .-----.
    Send 1 ACK  /      v     v              |     |      \
     for every |     .------------.    .------------.     | Send 1 ACK
     n packets |     | DCTCP.CE=0 |    | DCTCP.CE=1 |     | for every
    with ECE=0 |     '------------'    '------------'     | n packets
                \      |     |              ^     ^      /  with ECE=1
                 '-----'     '--------------'     '-----'
                              Send immediate
                              ACK with ECE=1
```

Figure 1: ACK Generation State Machine

3.3. Processing Echoed Congestion Indications on the Sender

The sender estimates the fraction of bytes sent that encountered congestion. The current estimate is stored in a new TCP state variable, DCTCP.Alpha, which is initialized to 1 and SHOULD be updated as follows:

DCTCP.Alpha = DCTCP.Alpha * (1 - g) + g * M

where:

- g is the estimation gain, a real number between 0 and 1. The selection of g is left to the implementation. See Section 4 for further considerations.

- M is the fraction of bytes sent that encountered congestion during the previous observation window, where the observation window is chosen to be approximately the Round-Trip Time (RTT). In particular, an observation window ends when all bytes in flight at the beginning of the window have been acknowledged.

In order to update DCTCP.Alpha, the TCP state variables defined in [RFC0793] are used, and three additional TCP state variables are introduced:

- DCTCP.WindowEnd: the TCP sequence number threshold when one observation window ends and another is to begin; initialized to SND.UNA.

- DCTCP.BytesAcked: the number of sent bytes acknowledged during the current observation window; initialized to 0.

- DCTCP.BytesMarked: the number of bytes sent during the current observation window that encountered congestion; initialized to 0.

The congestion estimator on the sender MUST process acceptable ACKs as follows:

1. Compute the bytes acknowledged (TCP Selective Acknowledgment (SACK) options [RFC2018] are ignored for this computation):

   BytesAcked = SEG.ACK - SND.UNA

2. Update the bytes sent:

   DCTCP.BytesAcked += BytesAcked

3. If the ECE flag is set, update the bytes marked:

   DCTCP.BytesMarked += BytesAcked

4. If the acknowledgment number is less than or equal to DCTCP.WindowEnd, stop processing. Otherwise, the end of the observation window has been reached, so proceed to update the congestion estimate as follows:

5. Compute the congestion level for the current observation window:

   M = DCTCP.BytesMarked / DCTCP.BytesAcked

6. Update the congestion estimate:

   DCTCP.Alpha = DCTCP.Alpha * (1 - g) + g * M

7. Determine the end of the next observation window:

   DCTCP.WindowEnd = SND.NXT

8. Reset the byte counters:

   DCTCP.BytesAcked = DCTCP.BytesMarked = 0

9. Rather than always halving the congestion window as described in [RFC3168], the sender SHOULD update cwnd as follows:

   cwnd = cwnd * (1 - DCTCP.Alpha / 2)

Just as specified in [RFC3168], DCTCP does not react to congestion indications more than once for every window of data. The setting of the CWR bit is also as per [RFC3168]. This is required for interoperation with classic ECN receivers due to potential misconfigurations.

3.4. Handling of Congestion Window Growth

A DCTCP sender grows its congestion window in the same way as conventional TCP. Slow start and congestion avoidance algorithms are handled as specified in [RFC5681].

3.5. Handling of Packet Loss

A DCTCP sender MUST react to loss episodes in the same way as conventional TCP, including fast retransmit and fast recovery algorithms, as specified in [RFC5681]. For cases where the packet loss is inferred and not explicitly signaled by ECN, the cwnd and other state variables like ssthresh MUST be changed in the same way that a conventional TCP would have changed them. As with ECN, a DCTCP sender will only reduce the cwnd once per window of data across all loss signals. Just as specified in [RFC5681], upon a timeout, the cwnd MUST be set to no more than the loss window (1 full-sized segment), regardless of previous cwnd reductions in a given window of data.

3.6. Handling of SYN, SYN-ACK, and RST Packets

If SYN, SYN-ACK, and RST packets for DCTCP connections have the ECN-Capable Transport (ECT) codepoint set in the IP header, they will receive the same treatment as other DCTCP packets when forwarded by a switching fabric under load. Lack of ECT in these packets can result in a higher drop rate, depending on the switching fabric configuration. Hence, for DCTCP connections, the sender SHOULD set ECT for SYN, SYN-ACK, and RST packets. A DCTCP receiver ignores CE codepoints set on any SYN, SYN-ACK, or RST packets.

4. Implementation Issues

4.1. Configuration of DCTCP

An implementation needs to know when to use DCTCP. Data-center servers may need to communicate with endpoints outside the data center, where DCTCP is unsuitable or unsupported. Thus, a global configuration setting to enable DCTCP will generally not suffice. DCTCP provides no mechanism for negotiating its use. Thus, additional management and configuration functionality is needed to ensure that DCTCP is not used with non-DCTCP endpoints.

Known solutions rely on either configuration or heuristics. Heuristics need to allow endpoints to individually enable DCTCP to ensure a DCTCP sender is always paired with a DCTCP receiver. One approach is to enable DCTCP based on the IP address of the remote endpoint. Another approach is to detect connections that transmit within the bounds of a data center. For example, an implementation could support automatic selection of DCTCP if the estimated RTT is less than a threshold (like 10 msec) and ECN is successfully negotiated under the assumption that if the RTT is low, then the two endpoints are likely in the same data-center network.

[RFC3168] forbids the ECN-marking of pure ACK packets because of the inability of TCP to mitigate ACK-path congestion. RFC 3168 also forbids ECN-marking of retransmissions, window probes, and RSTs. However, dropping all these control packets -- rather than ECN-marking them -- has considerable performance disadvantages. It is RECOMMENDED that an implementation provide a configuration knob that will cause ECT to be set on such control packets, which can be used in environments where such concerns do not apply. See [ECN-EXPERIMENTATION] for details.

It is useful to implement DCTCP as an additional action on top of an existing congestion control algorithm like Reno [RFC5681]. The DCTCP implementation MAY also allow configuration of resetting the value of DCTCP.Alpha as part of processing any loss episodes.

4.2. Computation of DCTCP.Alpha

As noted in Section 3.3, the implementation will need to choose a suitable estimation gain. [DCTCP10] provides a theoretical basis for selecting the gain. However, it may be more practical to use experimentation to select a suitable gain for a particular network and workload. A fixed estimation gain of 1/16 is used in some implementations. (It should be noted that values of 0 or 1 for g result in problematic behavior; g=0 fixes DCTCP.Alpha to its initial value, and g=1 sets it to M without any smoothing.)

The DCTCP.Alpha computation as per the formula in Section 3.3 involves fractions. An efficient kernel implementation MAY scale the DCTCP.Alpha value for efficient computation using shift operations. For example, if the implementation chooses g as 1/16, multiplications of DCTCP.Alpha by g become right-shifts by 4. A scaling implementation SHOULD ensure that DCTCP.Alpha is able to reach 0 once it falls below the smallest shifted value (16 in the above example). At the other extreme, a scaled update needs to ensure DCTCP.Alpha does not exceed the scaling factor, which would be equivalent to greater than 100% congestion. So, DCTCP.Alpha MUST be clamped after an update.

This results in the following computations replacing steps 5 and 6 in Section 3.3, where SCF is the chosen scaling factor (65536 in the example), and SHF is the shift factor (4 in the example):

1. Compute the congestion level for the current observation window:

```
ScaledM = SCF * DCTCP.BytesMarked / DCTCP.BytesAcked
```

2. Update the congestion estimate:

```
if (DCTCP.Alpha >> SHF) == 0, then DCTCP.Alpha = 0

DCTCP.Alpha += (ScaledM >> SHF) - (DCTCP.Alpha >> SHF)

if DCTCP.Alpha > SCF, then DCTCP.Alpha = SCF
```

5. Deployment Issues

DCTCP and conventional TCP congestion control do not coexist well in the same network. In typical DCTCP deployments, the marking threshold in the switching fabric is set to a very low value to reduce queueing delay, and a relatively small amount of congestion will exceed the marking threshold. During such periods of congestion, conventional TCP will suffer packet loss and quickly and drastically reduce cwnd. DCTCP, on the other hand, will use the fraction of marked packets to reduce cwnd more gradually. Thus, the rate reduction in DCTCP will be much slower than that of conventional TCP, and DCTCP traffic will gain a larger share of the capacity compared to conventional TCP traffic traversing the same path. If the traffic in the data center is a mix of conventional TCP and DCTCP, it is RECOMMENDED that DCTCP traffic be segregated from conventional TCP traffic. [MORGANSTANLEY] describes a deployment that uses the IP Differentiated Services Codepoint (DSCP) bits to segregate the network such that Active Queue Management (AQM) [RFC7567] is applied to DCTCP traffic, whereas TCP traffic is managed via drop-tail queueing.

Deployments should take into account segregation of non-TCP traffic as well. Today’s commodity switches allow configuration of different marking/drop profiles for non-TCP and non-IP packets. Non-TCP and non-IP packets should be able to pass through such switches, unless they really run out of buffer space.

Since DCTCP relies on congestion marking by the switches, DCTCP’s potential can only be realized in data centers where the entire network infrastructure supports ECN. The switches may also support configuration of the congestion threshold used for marking. The proposed parameterization can be configured with switches that implement Random Early Detection (RED) [RFC2309]. [DCTCP10] provides a theoretical basis for selecting the congestion threshold, but, as with the estimation gain, it may be more practical to rely on experimentation or simply to use the default configuration of the device. DCTCP will revert to loss-based congestion control when packet loss is experienced (e.g., when transiting a congested drop-tail link, or a link with an AQM drop behavior).

DCTCP requires changes on both the sender and the receiver, so both endpoints must support DCTCP. Furthermore, DCTCP provides no mechanism for negotiating its use, so both endpoints must be configured through some out-of-band mechanism to use DCTCP. A variant of DCTCP that can be deployed unilaterally and that only requires standard ECN behavior has been described in [ODCTCP] and [BSDCAN], but it requires additional experimental evaluation.

6. Known Issues

DCTCP relies on the sender's ability to reconstruct the stream of CE codepoints received by the remote endpoint. To accomplish this, DCTCP avoids using a single ACK packet to acknowledge segments received both with and without the CE codepoint set. However, if one or more ACK packets are dropped, it is possible that a subsequent ACK will cumulatively acknowledge a mix of CE and non-CE segments. This will, of course, result in a less-accurate congestion estimate. There are some potential considerations:

- o Even with an inaccurate congestion estimate, DCTCP may still perform better than [RFC3168].

- o If the estimation gain is small relative to the packet loss rate, the estimate may not be too inaccurate.

- o If ACK packet loss mostly occurs under heavy congestion, most drops will occur during an unbroken string of CE packets, and the estimate will be unaffected.

However, the effect of packet drops on DCTCP under real-world conditions has not been analyzed.

DCTCP provides no mechanism for negotiating its use. The effect of using DCTCP with a standard ECN endpoint has been analyzed in [ODCTCP] and [BSDCAN]. Furthermore, it is possible that other implementations may also modify behavior in the [RFC3168] style without negotiation, causing further interoperability issues.

Much like standard TCP, DCTCP is biased against flows with longer RTTs. A method for improving the RTT fairness of DCTCP has been proposed in [ADCTCP], but it requires additional experimental evaluation.

7. Security Considerations

DCTCP enhances ECN; thus, it inherits the general security considerations discussed in [RFC3168], although additional mitigation options exist due to the limited intra-data-center deployment of DCTCP.

The processing changes introduced by DCTCP do not exacerbate the considerations in [RFC3168] or introduce new ones. In particular, with either algorithm, the network infrastructure or the remote endpoint can falsely report congestion and, thus, cause the sender to reduce cwnd. However, this is no worse than what can be achieved by simply dropping packets.

[RFC3168] requires that a compliant TCP must not set ECT on SYN or SYN-ACK packets. [RFC5562] proposes setting ECT on SYN-ACK packets but maintains the restriction of no ECT on SYN packets. Both these RFCs prohibit ECT in SYN packets due to security concerns regarding malicious SYN packets with ECT set. However, these RFCs are intended for general Internet use; they do not directly apply to a controlled data-center environment. The security concerns addressed by both of these RFCs might not apply in controlled environments like data centers, and it might not be necessary to account for the presence of non-ECN servers. Beyond the security considerations related to virtual servers, additional security can be imposed in the physical servers to intercept and drop traffic resembling an attack.

8. IANA Considerations

This document does not require any IANA actions.

9. References

9.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <https://www.rfc-editor.org/info/rfc2018>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <https://www.rfc-editor.org/info/rfc3168>.

[RFC5562] Kuzmanovic, A., Mondal, A., Floyd, S., and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets", RFC 5562, DOI 10.17487/RFC5562, June 2009, <https://www.rfc-editor.org/info/rfc5562>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

9.2. Informative References

[ADCTCP] Alizadeh, M., Javanmard, A., and B. Prabhakar, "Analysis of DCTCP: Stability, Convergence, and Fairness", DOI 10.1145/1993744.1993753, Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, June 2011, <https://dl.acm.org/citation.cfm?id=1993753>.

[BSDCAN] Kato, M., Eggert, L., Zimmermann, A., van Meter, R., and H. Tokuda, "Extensions to FreeBSD Datacenter TCP for Incremental Deployment Support", BSDCan 2015, June 2015, <https://www.bsdcan.org/2015/schedule/events/559.en.html>.

[DCTCP10] Alizadeh, M., Greenberg, A., Maltz, D., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and M. Sridharan, "Data Center TCP (DCTCP)", DOI 10.1145/1851182.1851192, Proceedings of the ACM SIGCOMM 2010 Conference, August 2010, <http://dl.acm.org/citation.cfm?doid=1851182.1851192>.

[ECN-EXPERIMENTATION] Black, D., "Explicit Congestion Notification (ECN) Experimentation", Work in Progress, draft-ietf-tsvwg-ecn-experimentation-06, September 2017.

[Err3639] RFC Errata, Erratum ID 3639, RFC 3168, <https://www.rfc-editor.org/errata/eid3639>.

[FREEBSD] Kato, M. and H. Panchasara, "DCTCP (Data Center TCP) implementation", January 2015, <https://github.com/freebsd/freebsd/commit/8ad879445281027858a7fa706d13e458095b595f>.

[LINUX] Borkmann, D., Westphal, F., and Glenn. Judd, "net: tcp: add DCTCP congestion control algorithm", LINUX DCTCP Patch, September 2014, <https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/commit/?id=e3118e8359bb7c59555aca60c725106e6d78c5ce>.

[MAPREDUCE]
Dean, J. and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation, October 2004, <https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/dean.html>.

[MORGANSTANLEY]
Judd, G., "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter", Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, May 2015, <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/judd>.

[ODCTCP] Kato, M., "Improving Transmission Performance with One-Sided Datacenter TCP", M.S. Thesis, Keio University, 2013, <http://eggert.org/students/kato-thesis.pdf>.

[RFC2309] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J., and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet", RFC 2309, DOI 10.17487/RFC2309, April 1998, <https://www.rfc-editor.org/info/rfc2309>.

[RFC7567] Baker, F., Ed. and G. Fairhurst, Ed., "IETF Recommendations Regarding Active Queue Management", BCP 197, RFC 7567, DOI 10.17487/RFC7567, July 2015, <https://www.rfc-editor.org/info/rfc7567>.

[WINDOWS] Microsoft, "Data Center Transmission Control Protocol (DCTCP)", May 2012, <https://technet.microsoft.com/en-us/library/hh997028(v=ws.11).aspx>.

Acknowledgments

The DCTCP algorithm was originally proposed and analyzed in [DCTCP10] by Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan.

We would like to thank Andrew Shewmaker for identifying the problem of clamping DCTCP.Alpha and proposing a solution for it.

Lars Eggert has received funding from the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866 ("SSICLOPS"). This document reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

Authors' Addresses

Stephen Bensley
Microsoft
One Microsoft Way
Redmond, WA 98052
United States of America

Phone: +1 425 703 5570
Email: sbens@microsoft.com

Dave Thaler
Microsoft

Phone: +1 425 703 8835
Email: dthaler@microsoft.com

Praveen Balasubramanian
Microsoft

Phone: +1 425 538 2782
Email: pravb@microsoft.com

Lars Eggert
NetApp
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 151 120 55791
Email: lars@netapp.com
URI: http://eggert.org/

Glenn Judd
Morgan Stanley

Phone: +1 973 979 6481
Email: glenn.judd@morganstanley.com