

Network Working Group
Internet-Draft
Intended status: Informational
Expires: May 5, 2016

S. Bensley
Microsoft
L. Eggert
NetApp
D. Thaler
P. Balasubramanian
Microsoft
G. Judd
Morgan Stanley
November 2, 2015

Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters
draft-ietf-tcpm-dctcp-01

Abstract

This informational memo describes Datacenter TCP (DCTCP), an improvement to TCP congestion control for datacenter traffic. DCTCP uses improved 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, the lack of a negotiating mechanism between sender and receiver, and presents some possible mitigations.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 5, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	2
2.	Terminology	3
3.	DCTCP Algorithm	4
3.1.	Marking Congestion on the Switches	4
3.2.	Echoing Congestion Information on the Receiver	4
3.3.	Processing Congestion Indications on the Sender	5
3.4.	Handling of SYN, SYN-ACK, RST Packets	7
4.	Implementation Issues	7
5.	Deployment Issues	9
6.	Known Issues	10
7.	Implementation Status	11
8.	Security Considerations	11
9.	IANA Considerations	11
10.	Acknowledgements	11
11.	References	12
11.1.	Normative References	12
11.2.	Informative References	12
	Authors' Addresses	14

[1.](#) Introduction

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

Network traffic in a datacenter is often a mix of short and long flows, where the short flows require low latencies and the long flows require high throughputs. Datacenters 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 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:

- o The queue must be short enough that it does not impose excessive latency on short flows.
- o The queue must be long enough to buffer sufficient data for the long flows to saturate the path capacity.
- o The queue must be short 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 early detection of congestion, rather than waiting for packet loss to occur. 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.

Datacenter TCP (DCTCP) improves 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.

It is recommended that DCTCP be deployed in a datacenter environment where the endpoints and the switching fabric are under a single administrative domain. This protocol is not meant for uncontrolled deployment in the global Internet.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [[RFC2119](#)].

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 accordingly (cwnd).

3.1. Marking Congestion on the Switches

The switches in a datacenter 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 $K > (RTT * C)/7$, where C is the sending rate in packets per second. 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, sender and receiver MUST 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 code point 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 code point was set in the packet being acknowledged. However, this prevents the use of delayed ACKs, which are an important performance optimization in datacenters.

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, send an ACK for any previously unacknowledged packets and set DCTCP.CE to true.
2. If the CE codepoint is not set and DCTCP.CE is true, send an ACK for any previously unacknowledged packets and set DCTCP.CE to false.
3. Otherwise, ignore the CE codepoint.

The handling of the "Congestion Window Reduced" (CWR) bit is also exactly as per [\[RFC3168\]](#) including [\[RFC3168-ERRATA3639\]](#). That is, on receipt of a segment with both the CE and CWR bits set, CWR is processed first and then ECE is processed.

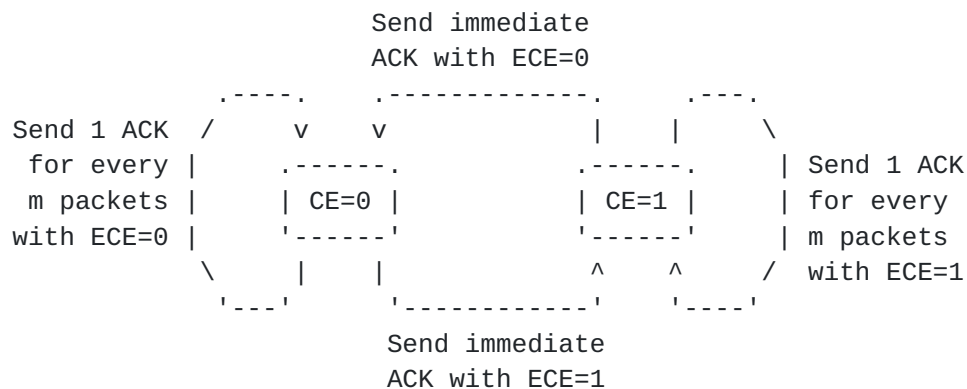


Figure 1: ACK generation state machine. DCTCP.CE abbreviated as CE.

3.3. Processing 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 MUST be updated as follows:

$$\text{DCTCP.Alpha} = \text{DCTCP.Alpha} * (1 - g) + g * M$$

where

- o 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.
- o 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:

- o DCTCP.WindowEnd: The TCP sequence number threshold for beginning a new observation window; initialized to SND.UNA.
- o DCTCP.BytesSent: The number of bytes sent during the current observation window; initialized to zero.
- o DCTCP.BytesMarked: The number of bytes sent during the current observation window that encountered congestion; initialized to zero.

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

1. Compute the bytes acknowledged (TCP SACK options [\[RFC2018\]](#) are ignored):

$$\text{BytesAcked} = \text{SEG.ACK} - \text{SND.UNA}$$

2. Update the bytes sent:

$$\text{DCTCP.BytesSent} += \text{BytesAcked}$$

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

$$\text{DCTCP.BytesMarked} += \text{BytesAcked}$$

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:

$$M = \text{DCTCP.BytesMarked} / \text{DCTCP.BytesSent}$$

6. Update the congestion estimate:

$$\text{DCTCP.Alpha} = \text{DCTCP.Alpha} * (1 - g) + g * M$$

7. Determine the end of the next observation window:


```
DCTCP.WindowEnd = SND.NXT
```

8. Reset the byte counters:

```
DCTCP.BytesSent = DCTCP.BytesMarked = 0
```

Rather than always halving the congestion window as described in [RFC3168], when the sender receives an indication of congestion (ECE), the sender MUST update cwnd as follows:

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

Thus, when no bytes sent experienced congestion, DCTCP.Alpha equals zero, and cwnd is left unchanged. When all sent bytes experienced congestion, DCTCP.Alpha equals one, and cwnd is reduced by half. Lower levels of congestion will result in correspondingly smaller reductions to cwnd.

Just as specified in [RFC3168], TCP should not react to congestion indications more than once for every window of data. The setting of the "Congestion Window Reduced" (CWR) bit is also exactly as per [RFC3168].

3.4. Handling of SYN, SYN-ACK, RST 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. These RFCs, however, are intended for general Internet use, and do not directly apply to a controlled datacenter environment. The switching fabric can drop TCP packets that do not have the ECT set in the IP header. If SYN and SYN-ACK packets for DCTCP connections do not have ECT set, they will be dropped with high probability. For DCTCP connections, the sender SHOULD set ECT for SYN, SYN-ACK and RST packets.

4. Implementation Issues

As noted in [Section 3.3](#), the implementation MUST 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. The Microsoft implementation of DCTCP in Windows Server 2012 uses a fixed estimation gain of 1/16.

The implementation must also decide when to use DCTCP. Datacenter servers may need to communicate with endpoints outside the

datacenter, 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, there is additional management and configuration overhead required to ensure that DCTCP is not used with non-DCTCP endpoints.

Potential 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 datacenter. For example, Microsoft Windows Server 2012 (and later versions) supports automatic selection of DCTCP if the estimated RTT is less than 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 datacenter network.

It is RECOMMENDED that an implementation deal with loss episodes in the same way as conventional TCP. In case of a timeout or fast retransmit or any change in delay (for delay based congestion control), the cwnd and other state variables like ssthresh must be changed in the same way that a conventional TCP would have changed them. It would be useful to implement DCTCP as additional actions on top of an existing congestion control algorithm like NewReno. The DCTCP implementation MAY also allow configuration of resetting the value of DCTCP.Alpha as part of processing any loss episodes.

To prevent incast throughput collapse, the minimum RTO (MinRTO) used by TCP should be lowered significantly. The default value of MinRTO in Windows is 300 msec, which is much greater than the maximum latencies inside a datacenter. In Microsoft Windows Server 2012 (and later), the MinRTO value is configurable, allowing values as low as 10 msec on a per-subnet or per-port basis (or even globally.) A lower MinRTO value requires a correspondingly lower delayed ACK timeout on the receiver. It is RECOMMENDED that an implementation allow configuration of lower timeouts for DCTCP connections.

In the same vein, it is also RECOMMENDED that an implementation allow configuration of restarting the congestion window (cwnd) of idle DCTCP connections as described in [[RFC5681](#)], since network conditions can change rapidly in datacenters.

[RFC3168] forbids the ECN-marking of pure ACK packets, because of the inability of TCP to mitigate ACK-path congestion and protocol-wise preferential treatment by routers. However, dropping pure ACKs - rather than ECN marking them - has disadvantages for typical datacenter traffic patterns. Because of the prevalence of bursty traffic patterns that feature transient congestion, dropping of ACKs

causes subsequent retransmissions. It is RECOMMENDED that an implementation provide a configuration knob that forces ECT to be set on pure ACKs.

The DCTCP.Alpha calculation 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 zero once it falls below the smallest shifted value (16 in the above example). At the other extreme, a scaled update MUST also 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:

$$\text{ScaledM} = \text{SCF} * \text{DCTCP.BytesMarked} / \text{DCTCP.BytesSent}$$

2. Update the congestion estimate:

$$\text{if } (\text{DCTCP.Alpha} \gg \text{SHF}) == 0 \text{ then DCTCP.Alpha} = 0$$
$$\text{DCTCP.Alpha} += (\text{ScaledM} \gg \text{SHF}) - (\text{DCTCP.Alpha} \gg \text{SHF})$$
$$\text{if DCTCP.Alpha} > \text{SCF} \text{ then DCTCP.Alpha} = \text{SCF}$$

5. Deployment Issues

DCTCP and conventional TCP congestion control do not coexist well in the same network. In DCTCP, the marking threshold 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 datacenter 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

DSCP bits to segregate the network such that AQM is applied to DCTCP traffic, whereas TCP traffic is managed via drop-tail queueing.

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. If the datacenter traffic consists of such traffic (including UDP), one possible mitigation would be to mark IP packets as ECT even when there is no transport that is reacting to the marking.

Since DCTCP relies on congestion marking by the switches, DCTCP can only be deployed in datacenters 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 RED. [[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 degrade to loss-based congestion control when transiting a congested drop-tail link.

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 only requires standard ECN behavior has been described in [[ODCTCP](#)][BSDCAN], but 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 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](#)][BSDCAN]. Furthermore, it is possible that other implementations may also modify [[RFC3168](#)] behavior without negotiation, causing further interoperability issues.

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

[7.](#) Implementation Status

This section documents the implementation status of the specification in this document, as recommended by [[RFC6982](#)].

This document describes DCTCP as implemented in Microsoft Windows Server 2012. Since publication of the first versions of this document, the Linux [[LINUX](#)] and FreeBSD [[FREEBSD](#)] operating systems have also implemented support for DCTCP in a way that is believed to follow this document.

[8.](#) Security Considerations

DCTCP enhances ECN and thus inherits the security considerations discussed in [[RFC3168](#)]. The processing changes introduced by DCTCP do not exacerbate these considerations 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.

[9.](#) IANA Considerations

This document has no actions for IANA.

[10.](#) Acknowledgements

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.

11. References

11.1. Normative References

- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, [RFC 793](#), DOI 10.17487/RFC0793, September 1981, <<http://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, <<http://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, <<http://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, <<http://www.rfc-editor.org/info/rfc3168>>.

11.2. Informative References

- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", [RFC 5681](#), DOI 10.17487/RFC5681, September 2009, <<http://www.rfc-editor.org/info/rfc5681>>.
- [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, <<http://www.rfc-editor.org/info/rfc5562>>.
- [RFC6982] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", [RFC 6982](#), DOI 10.17487/RFC6982, July 2013, <<http://www.rfc-editor.org/info/rfc6982>>.

- [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, Proc. ACM SIGCOMM 2010 Conference (SIGCOMM 10), August 2010, <<http://dl.acm.org/citation.cfm?doid=1851182.1851192>>.
- [ODCTCP] Kato, M., "Improving Transmission Performance with One-Sided Datacenter TCP", M.S. Thesis, Keio University, 2014, <<http://eggert.org/students/kato-thesis.pdf>>.
- [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>>.
- [ADCTCP] Alizadeh, M., Javanmard, A., and B. Prabhakar, "Analysis of DCTCP: Stability, Convergence, and Fairness", DOI 10.1145/1993744.1993753, Proc. ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 11), June 2011, <<http://dl.acm.org/citation.cfm?id=1993753>>.
- [LINUX] Borkmann, D. and F. Westphal, "Linux DCTCP patch", 2014, <<https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/commit/?id=e3118e8359bb7c59555aca60c725106e6d78c5ce>>.
- [FREEBSD] Kato, M. and H. Panchasara, "DCTCP (Data Center TCP) implementation", 2015, <<https://github.com/freebsd/freebsd/commit/8ad879445281027858a7fa706d13e458095b595f>>.
- [MORGANSTANLEY] Judd, G., "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter", Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), May 2015, <<https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/judd>>.
- [RFC3168-ERRATA3639] Scheffenegger, R., "RFC3168 Errata ID 3639", 2013, <http://www.rfc-editor.org/errata_search.php/doc/html/rfc3168&eid=3639>.

Authors' Addresses

Stephen Bensley
Microsoft
One Microsoft Way
Redmond, WA 98052
USA

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

Lars Eggert
NetApp
Sonnenallee 1
Kirchheim 85551
Germany

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

Dave Thaler
Microsoft

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

Praveen Balasubramanian
Microsoft

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

Glenn Judd
Morgan Stanley

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

