Master of Thesis Academic Year 2013

Improving transmission performance with one-sided datacenter TCP

Midori Kato

Graduate School of Media and Governance (Cyber Informatics)
Keio University

Master's Thesis Academic Year 2013
Improving transmission performance with one-sided datacenter TCP

Summary

Data centers host diverse applications, mixing flows that require short predictable latency with ones requiring high sustained throughput. In this network environment, the existing TCP (i.e., SACK TCP) does not achieve both goals. The existing TCP over a network with FIFO queues fills these buffers by background flows and this results in critically high latency for short flows. DCTCP is a proposed variant of TCP which solves the performance impairments in the data center network. DCTCP leverages Explicit Congestion Notification (ECN) in the network and identifies the depth of congestion from the proportion of ECN feedbacks. It provides high burst tolerance, low latency and high throughput for individual flows.

In a data center, some applications prevent from easily upgrading the kernel of servers due to the limitation of the software and hardware support. A new protocol should be incrementally deployable considering the deployment into such a network, however, DCTCP behaves poorly in a mixed network environment with other TCP flows due to lacks of the compatibilities against ECN.

This thesis proposes ODCTCP (One-sided DCTCP) which can be deployed on one endpoint only as long as the other implements SACK TCP with ECN. ODCTCP eliminates small but critical performance impairments in the incremental deployment path by being compatible with the standard ECN and rebalances total transmission performance in the data center network.

DCTCP and ODCTCP were implemented done as the modular TCP congestion control of the FreeBSD kernel and the performance was evaluated with the implementation. The results show that the data transfer time using ODCTCP is reasonably rebalanced for all the scenarios that have been evaluated while keeping a fair share of bandwidth between non-ODCTCP and DCTCP flows.

keywords
1. Datacenter network, 2. ECN, 3. TCP, 4. DCTCP,
5. incremental protocol deployment

Media and Governance
Keio University
Midori Kato

Acknowledgment

The main part of this thesis has been done at the NetApp GmbH Research Lab in Munich, as the Master thesis in Media and Governance (Cyber Informatics) of Keio University. I would like to express my sincere acknowledgment to my NetApp supervisor, Dr. Lars Eggert for giving me a invaluable internship experience, guiding for an interesting TCP topic and reviewing my thesis in detail. He also gives me helpful job recommendation and university information for my Ph.D. I would also like to thank my colleague at NetApp, Dr. Alexander Zimmermann who provided me with his support and technical discussion for my research. He also leads me to develop software with other researchers. Furthermore, I feel very thankful to intern friends, Naman Muley, Aris Angelogiannopoulos and David Leib. I am very happy to meet them. Thanks to Naman, I gain a new hobby, watching good movies. As Aris has studied TCP researches like me, we could exchange helpful information for TCP research. David helps my experiments for the further evaluation by splitting his working time. Without his sincere help, I cannot finish the evaluation in my thesis. Besides, I never forget many people kindness during my internship. I appreciate to Lucien Amoverlov, an engineer of a Cisco product, Nexus 3548. Thanks to his helpful comment, I could correctly configure ECN. I feel a thankful to Mirja Kuhlewind, Richard Scheffenegger and Michio Honda. They kindly lead me to my current topic and gave information for correct understanding of ECN and DCTCP algorithms.

I would like to express acknowledgment to my adviser, Hideyuki Tokuda. He gave me many chances to join international communities and take a great care of me. I also express my acknowledgment to my committees, Kenjiro Cho and Rod Van Meter. I could never learn how to do good research without them. In addition, Kenjiro had a great support to me since the first year of my master. My implementation skill wouldn't improve without working with him. Rod gives me a great support for researches and his help to revise poor my writing texts. In addition, I honor taking his classes through my student life. His awesome class about operating system and networking programming class leads me to this research area. I personally appreciate Akimichi Ogawa and Jin Nakazawa to give advice for good technical writing. Their advice is always simple but powerful to revise my poor texts.

Table of Contents

Chapter 1

Introduction

This chapter describes the background of this research and gives a brief overview of the problem of the existing proposal.

1.1 Background

Data centers are emerging as essential computing platforms for IT enterprises. A large service provider such as Google, Facebook or Amazon builds data centers for itself to operate its services.

In a data center, up to a few thousand servers are interconnected via switches to form the network infrastructure. A typical data center network topology is the two-level tree shown in Figure 1.1. In the topology, a rack includes dozens of servers and a switch at the top. The servers are connected to a rack switch via 1Gbps Ethernet link and the rack switch is connected to an aggregation switch via 10Gbps Ethernet link. Internet bound traffic passes throughput and aggregation switches, then, a core switch.

A data center is designed to build highly available, highly performant computing and

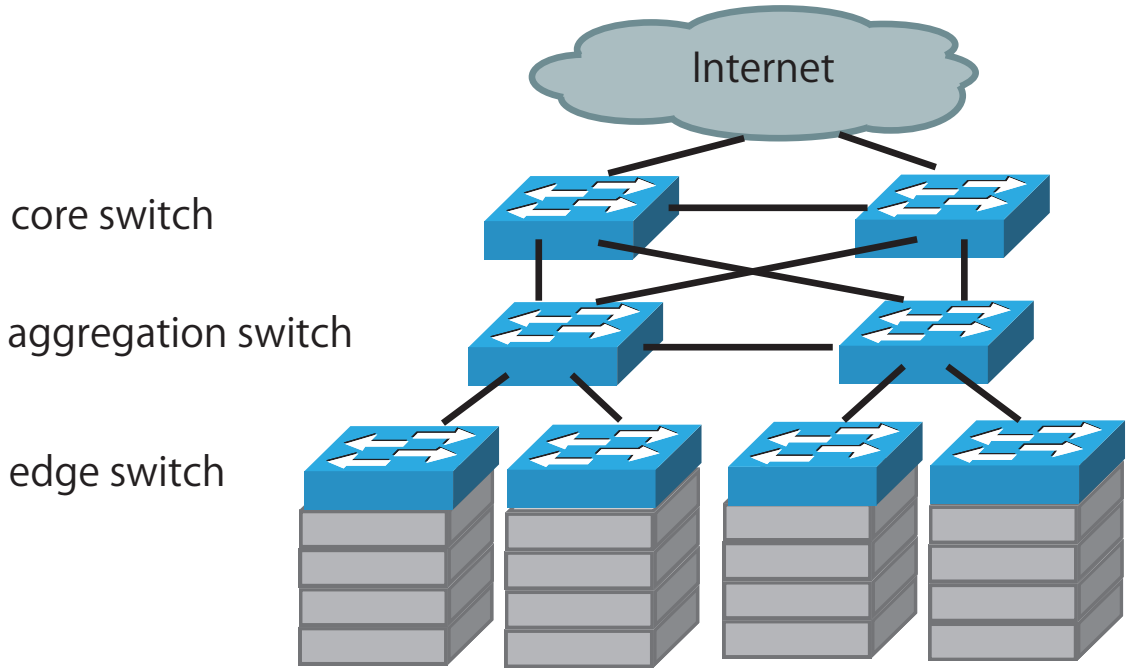

Figure 1.1: Example of data center design: Fat-tree topology

storage infrastructure using low cost, commodity components. In particular, rack switches are available at a low price, providing up to 48 ports at 1Gbps. On the other hand, how well such commodity switches handle the traffic of real data center applications is important for total network performance.

The concrete requirements for packet processing at rack switches depend on the services running on servers. This thesis focuses on soft real-time applications such as online web search, online retail and advertisement. These applications generate a diverse mix of short and long flows and require three things from the data center network: short latency for short flows, high burst tolerance and high-sustained throughput for long flows.

The first two requirements stem from the need to supply high quality of information to users. Soft real-time applications are tightly scheduled until the data transmission to users is complete. Although each data transmission corresponds to a short query and response, a small latency for individual task directly affects the quality of results and the revenue of the service provider. According to Amazon's report, 100 milliseconds increase in network latency results in a loss of 1% of the customers [20]. Reducing network latency for short flows and high burst tolerance improve the ability to provide results of such services.

The third requirement stems from the need to refresh, analyze and update internal data for soft real-time applications. A response to a user assembles multiple components, such as the result of large data analysis and big data mining. These components affect the quality of the result, therefore, high throughput for long flows is as essential as short latency for short flows and high burst tolerance.

A traffic measurement in a data center [5] for soft real-time applications reveal performance impairments when using the existing TCP, namely, SACK TCP. SACK TCP is an enhanced TCP with SACK (Selective Acknowledgment) [18]. SACK helps the sender to learn which received packets have been received safely. This mechanism provides perfor-

mance improvement in the presence of multiple packet losses during one RTT (Round Trip Time). However, SACK TCP flows experience high application latency in the data center. If many flows converge on the same queue of a switch, SACK TCP induces burst traffic. When short and long flows are mixed in the traffic, long flows consume the FIFO buffer in the switches.

Using SACK TCP with ECN (Explicit Congestion Notification) [23] is one of the solutions for reducing the FIFO buffer occupancy. ECN provides a means for end hosts to detect incipient congestion before intermediates (i.e., routers and switches) in a network are forced to discard packets. In addition, ECN has been already implemented in the TCP stack and vendors implements ECN in L3 switches. Therefore, leveraging ECN in the network is low cost. However, the existing ECN mechanism is leads to buffer underflows and low throughput for long flows because it reacts too aggressively to the available bandwidth.

DCTCP (Data Center TCP) is a proposed TCP which modifies the existing ECN mechanism. It reacts to congestion in proportion to the extent of ECN marked packets. This mechanism matches the input rate and the available bandwidth in the network and achieves all three requirements for the data center network.

1.2 Problem statement

It is hard to deploy a new TCP variant such as DCTCP all at once in a data center network in practice. The kernel on some servers cannot be upgraded even when all of the machines are under centralized control. For example, storage appliances are constrained to not use the latest kernel due to the requirements of a wide range of hardware and software. Suppose a network environment that includes both storage and front-end application servers. On one hand, storage servers use the existing TCP or TCP-like protocol because of their reliance of the out-dated kernel. On the other hand, front-end servers such as web services and database

applications use a new TCP variant because they can upgrade their kernels. In the result, flows inside a storage network use a new TCP variant and an existing TCP serves traffic in the storage network. Flows that exchange data between a storage server and a front-end server use different TCP variant on each endpoint.

DCTCP is not designed to deploy on one endpoint only. Therefore, this thesis considers the incremental DCTCP deployment in a network operated with multiple protocols and addresses problems which occurs on one endpoint using DCTCP.

When considering the partial deployment of DCTCP, what protocol has been deployed is important. It is hard to share the fair use of a bandwidth between DCTCP and SACK TCP flows. When both DCTCP and SACK flows are running, DCTCP flows get much lower throughput than SACK TCP flows. This limitation is due to ECN. DCTCP detects an incipient congestion while SACK TCP does not detect it. Therefore, DCTCP behaves more conservatively than SACK TCP does and a partial DCTCP deployment into the network operated with SACK TCP results in less than expected performance gains.

Instead, we can use SACK TCP with ECN to deploy DCTCP partially. This thesis abbreviates SACK TCP with TCP as standard ECN as below. Since both standard ECN and DCTCP use ECN, DCTCP flows must be reasonably friendly with standard ECN flows. However, DCTCP is not friendly to standard ECN.

The unfriendliness between one DCTCP and one standard ECN endpoint occurs under the following two situations. When DCTCP is used on a sender only, the DCTCP sender declines throughput to the minimum. This is because of a lack of the compatibility support to the standard ECN receiver. When DCTCP is used on a receiver only, the standard ECN sender transmits a smaller amount of data than expected. This is because the DCTCP receiver ignores the existing TCP technique and the standard ECN sender misbehaves against the DCTCP receiver. Thus, if we can solve the problems with DCTCP interacting poorly with

standard ECN, we can improve the ease of deploying DCTCP and improve data center network performance.

1.3 Contributions

This thesis proposes ODCTCP (One-sided DCTCP) which allows the partial or incremental deployment in a data center network where servers and switches already use standard ECN. ODCTCP eliminates small but critical impairments for transmission performance by applying two modifications in the DCTCP algorithm. One of them applies the DCTCP sender algorithm and the other applies the DCTCP receiver algorithm. In the evaluations, ODCTCP shows reasonable transmission performance in a network where some servers use standard ECN. When the ODCTCP sender algorithm is used for short flows only, ODCTCP short flows show up to 1.2 times as short data transfer time as standard ECN short flows in the traffic mixed short and long flows. When the ODCTCP receiver algorithm is used for short flows only, ODCTCP short flows show up to 1.2 times as short data transfer time as standard ECN short flows in the traffic mixed short and long flows.

This thesis includes two additional contributions. One is an implementation of DCTCP in the FreeBSD kernel as a modular congestion control algorithm in the kernel. The other is to apply three modifications to the DCTCP algorithm. These modifications improve DCTCP transmission performance.

1.4 Thesis outline

This chapter defines the problem statement and the contributions of this thesis. The next chapter describes related work about standard ECN and DCTCP. Chapter 3 describes the DCTCP algorithm in detail. Then, Chapter 4 describes the DCTCP implementation in the FreeBSD kernel. This chapter also contains the DCTCP modifications that improve the

transmission performance. Chapter 5 presents the ODCTCP algorithm. Chapter 6 describes the experimental setup and scenarios for the evaluation. Then, Chapter 7 shows the result. The conclusion and future work are described in Chapter 8.

Chapter 2

Related work

This chapter introduces SACK TCP with ECN and related work about advanced congestion control using ECN.

2.1 Standard ECN

TCP provides best-effort data delivery that is not sensitive for delay or packet losses. Although TCP has Fast Recovery and Fast Retransmit algorithm [19] to recover from packet losses quickly, these mechanisms are not sufficient enough for applications which is sensitive to latency or losses of individual packets.

ECN [23, 4] provides a means for an endpoint to detect incipient congestion over networks. Endpoints using standard ECN (i.e., SACK TCP with ECN) can avoid the excessive queuing latency and identify congestion without dropping packets.

ECN never works without an AQM (Active Queuing Management) [10] implemented switch (or routers). One of the AQM schemes which can use ECN is RED (Random Early Detection) [17]. RED provides a congestion indication to servers by dropping packets before a queue of a switch builds up. When RED is used for ECN, the switch marks packets instead of dropping packets.

A ECN implemented switch marks a packet based on the averaged queue length exceeding a threshold. The switch configures two levels of thresholds, min_th and max_th in advance and maintains the queue length using weighting parameter w. If the queue length is between min_th and max_th, the switch marks packets randomly with the probability p. Beyond maximum threshold, it marks all incoming packets.

The switch uses two bits in the IP header when it marks a packet. Figure 2.1 highlight the two bits in the IP header. An ECN capable sender sets either of two bits. The set bit is called ECT (ECN Capable Transport), which corresponds to 01 or 10. A switch sets the other bit when the queue length exceeds a threshold. The bit corresponds to a CE (Congestion Experience) bit. Unless a sender sets an ECT bit, the switch is not allowed to set a CE bit.

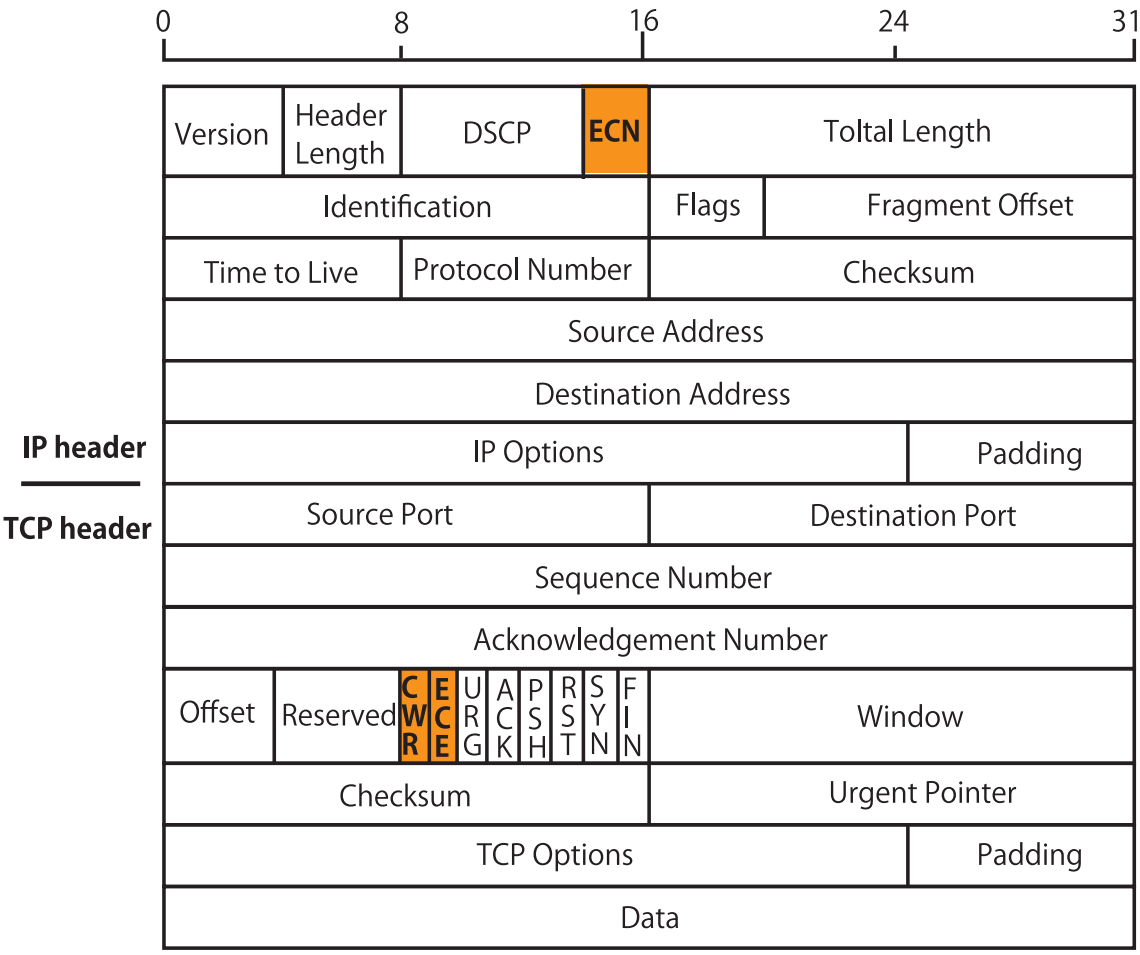

Figure 2.1: Reserved 4 bits for ECN in IP and TCP headers

ECN capable endpoints use two bits in the TCP header in addition to two bits in the IP header. Figure 2.1 highlights the two bits in the TCP header. The two bits are used for ECN setup and subsequent actions to be taken a CE bit in the network. In this section, the explanation for ECN setup is skipped. See [23] if needed.

Figure 2.2 shows a series of ECN actions between a standard ECN sender and a standard ECN receiver. Suppose that the standard ECN sender transmits four packets with the ECT bit set and a switch set a CE bit in the second packet (1st-4th segments). When the receiver

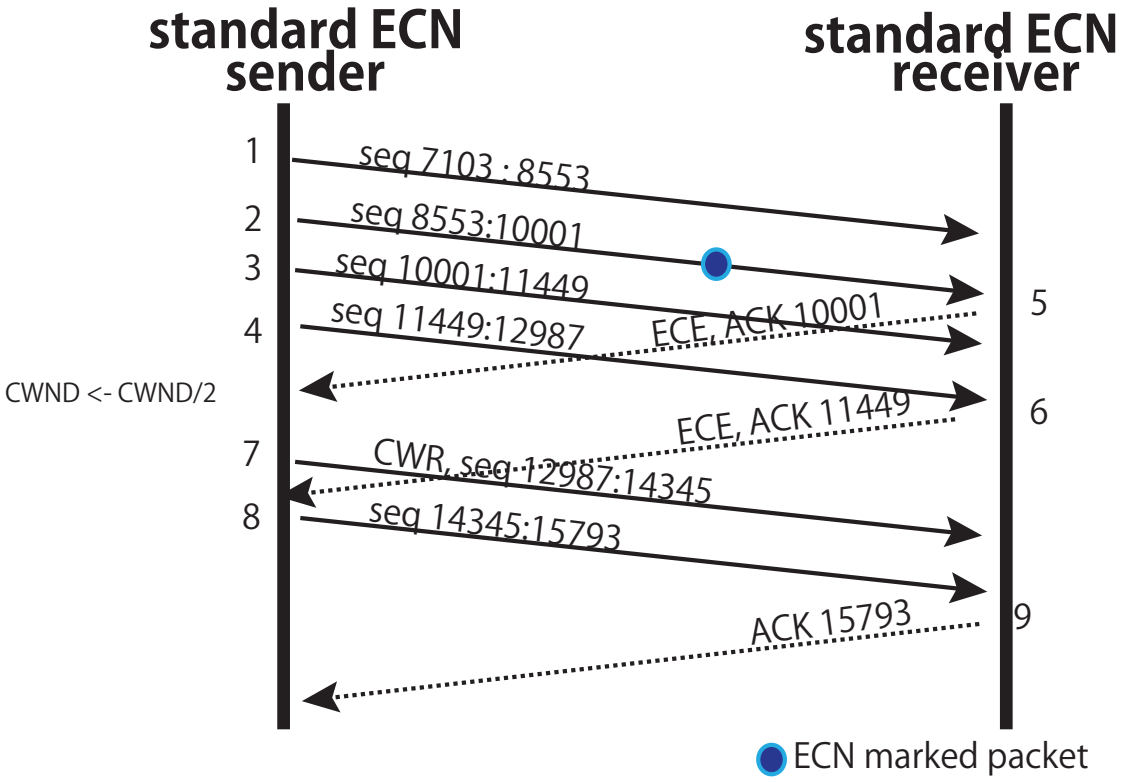

Figure 2.2: A series of ECN actions between a sender and a receiver using standard ECN

identifies a CE bit in an arriving packet, it sets the ECE (ECN-Echo) flag in the subsequent ACK packets (5th-6th segments). Upon the reception of a packet with the ECE flag, the sender halves `CWND`. Then, to confirm the receipt of the ECE flag to the receiver, it sets the CWR (Congestion Window Reduction) flag for the outgoing packet (7th segment). When the receiver identifies the CWR flag, it stops to set the ECE flag (9th segment).

2.2 Advanced TCP stack using ECN

New uses of ECN in the TCP stack are intensively researched for data center networks. D^2TCP [31] and D^3 [32] highlight the prioritized flow handling considering to use a data center network with multi-tenants. However, prioritizing individual flows is high cost for

applications. To deploy these protocol, all applications operated in the data center explicitly are prioritized and identifies the priority by itself when generating flows. Therefore, these protocols are hard to deploy rather than DCTCP.

VCP [33], RCP [15] and Re-ECN [12] correspond to researches that study ECN feedback methodologies for high-speed networks. However, these protocols require complex AQM operations at switches. In addition, other TCP flows are impacted by the complex AQM scheme. These negative points let us abandon their partial deployment.

The most similar research to this thesis is AccrECN [25]. It proposes an advanced ECN feedback methodology to distinguish DCTCP and standard ECN. This proposal is useful for a network mixed multiple protocols. However, because AccrECN needs to support its own negotiation, this proposal does not allow to use in the network environment where the out-dated kernel exists. Thus, the network where this thesis is target does not allow to use AccrECN.

Chapter 3

DCTCP algorithm

This chapter describes the desirable AQM scheme for DCTCP flows at a switch and the DCTCP algorithm.

3.1 Simple ECN marking mechanism at switches

DCTCP employs a very simple AQM scheme at switches. There is only one parameter to be set, which corresponds to the ECN marking threshold, K. A switch sets a CE bit for an arriving packet if the instantaneous queue occupancy is greater than K. Otherwise, it simply forwards the packet. Note that this simple ECN marking mechanism works with the RED configuration by setting parameters to $w = 1.0$, $min\_th = max\_th = K$ and $p = 0.0$. According to [5], the K value is recommended to set 20 packets for a 1Gbps link and 65 for a 10Gbps link.

3.2 Congestion control at DCTCP senders

Unlike a standard ECN sender, a DCTCP sender controls `CWND` per RTT in the extent of congestion. The following equation is used for the `CWND` calculation.

$$CWND \leftarrow CWND(1 - \frac{\alpha}{2}) \tag{3.1}$$

where α is a moving average over the fraction of ECE marked segments during the last RTT. In this equation, α close to zero indicates light congestion and α close to one indicates heavy congestion. When α is 1, the DCTCP sender behaves just like the standard ECN sender. The α value is given by

$$\alpha \leftarrow (1 - g)\alpha + g * F \tag{3.2}$$

where F is the fraction of marked segments in the last RTT and g is a weight factor. The F value is updated every RTT.

The g value is recommended to be set to $1/2^4$. The DCTCP analysis [6] describes the trade-off between small g and large g. The small g value gives the slow convergence between

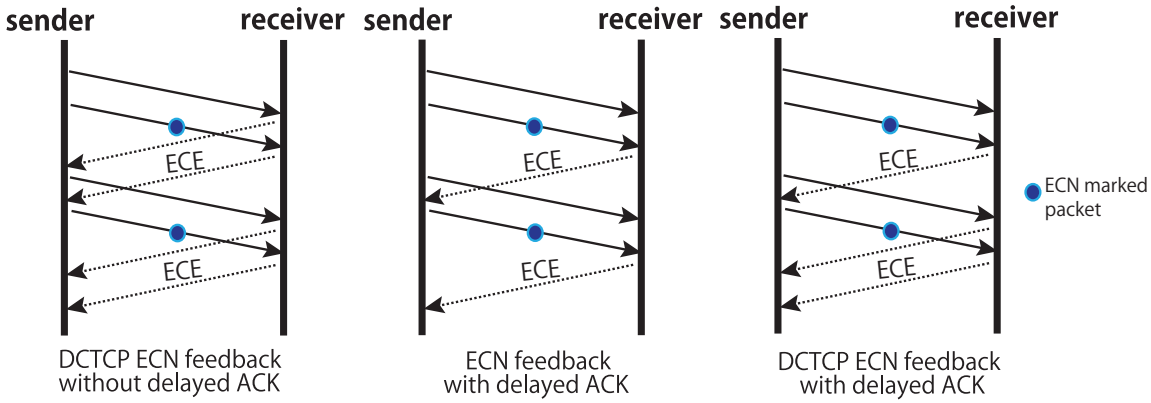

Figure 3.1: ECN feedback with/without delayed ACK

competitive DCTCP flows. On one hand, the large g value impacts the low utilization of bandwidth at switches. The selected g value is said to be reasonable when considering the trade-off.

The difference between the standard ECN and DCTCP sender algorithm is `CWND` calculation per RTT. Otherwise, the DCTCP sender applies Fast Recovery, Timeout reaction, Fast Retransmit from the standard ECN sender algorithm.

3.3 ECN feedback at DCTCP receivers

A DCTCP receiver uses a different ECN feedback methodology from a standard ECN receiver. A standard ECN receiver sets the ECE flag for outgoing packets until it receives the confirmation, which is a packet with an CWR flag, from the sender. On the other hand, a DCTCP receiver responds the ECE flag only when it identifies an ECN marked packet from a switch.

This algorithm is very simple, but it is problematic when the DCTCP receiver uses delayed ACK [11]. Delayed ACK allows the server to ack every m packets [1]. The consideration of

[1]The m value depends on operating systems. Note that the latest FreeBSD kernel (11.0) set two as m value.

delayed ACK is important because most of operating systems supports the delayed ACK to reduce the overhead of the packet processing at the sender. Figure 3.1 shows three TCP sequence diagrams for ECN feedback that a DCTCP receiver can use. The left one shows a TCP diagram when DCTCP does not implement delayed ACK at the receiver. The DCTCP receiver responds every packet and sets an ECE flag for the second and fourth ACK packets. Using this ECN feedback gives high overhead at the sender due to the large number of ACK packets.

The middle one shows a TCP diagram when the receiver enables delayed ACK. The receiver responds every two packet and sets an ECE flag for the first and second ACK packets. This reduces the overhead at the sender, but the sender incorrectly understands the number of ECN marked packets. Although the correct number is two, the sender will estimate the number at three. Therefore, the DCTCP receiver does not use this ECN feedback methodology.

Instead, the DCTCP receiver enabling delayed ACK uses the right diagram. The DCTCP receiver responds an ACK packet when the boarder of marked and not marked packets corresponds to the ACK packet being delayed. Thereby, the receiver responds the third packet in the right figure. This ECN feedback at the DCTCP receiver makes it possible to exactly reconstructs that a CE bit were seen at the DCTCP sender.

Chapter 4

DCTCP implementation in FreeBSD

This chapter introduces a necessary extension to the the modular congestion control framework in the FreeBSD kernel to implement DCTCP. Then, it investigates the design space for the detailed DCTCP algorithm to improve the transmission performance and the fairness.

4.1 ECN processing in the FreeBSD congestion control framework

A desirable DCTCP implementation is transparent to applications and system configuration variables. To achieve this, a DCTCP implementation is added as a module in the congestion control framework [9].

The FreeBSD kernel supports a modularized TCP congestion control framework [27]. The framework aim to facilitate implementation of new congestion control algorithms. Implementing a congestion control algorithm using the framework allows to configure the congestion control algorithm as a kernel module.

The modular congestion control framework defines two fields in TCP control block [1]: `cc_var` and `cc_algo` structures. The `cc_var` structure is used to store a set of specific variables to control a congestion for a TCP connection. The `cc_algo` structure accommodates basic housekeeping functions for congestion control.

The existing framework is insufficient support for non-standard ECN processing. The latest FreeBSD kernel processes three things for standard ECN.

1. the kernel decides whether an ECE flag should be set in the next outgoing TCP segment by snooping reserved ECN bits in IP and TCP headers.
2. The kernel controls a congestion if an ECE flag is set in an arriving TCP segment.
3. After the outgoing TCP segment is generated, the kernel decides whether an ECT bit should be set in an ECN field of IP header in the outgoing packet.

However, the first and third processing are apart from the modular congestion control framework. Thus, the kernel needs new functions for the two ECN processing.

[1] TCP control block is a data structure which maintains a TCP connection state [30].

function	usage
int ecnpkt_handler(struct cc_var *ccv, uint8_t iptos, int cwr, int is_delayack)	Execute an additional ECN processing using ECN field in IP header and the CWR bit in TCP header. As an option, input a flag which shows this packet is in delayed ACK.
int ect_handler(struct cc_var *ccv)	Check whether this packet should set an ECT bit in IP header.

Table 4.1: Additional two functions defined in the structure `cc_algo`. Both allows different ECN processing from the standard ECN without interfering in the standard ECN processing.

This thesis appends two additional functions to make up for the first and third ECN processing. Table 4.1 shows functions and the usage. The `ecnpkt_handler()` function is called after the first ECN processing. Therefore, a kernel can track an ECN field and the CWR flag of the received packet in different way from the standard ECN algorithm. This function allows an additional input as arguments. In the current implementation, a flag which tells that this packet is in the delayed ACK is defined as an argument. If the return value is set to non-zero, the kernel disables delayed ACK for the packet. The other function is `ect_handler()`, which is called after the third ECN processing. According to [23], standard ECN restricts to mark ECE flag in retransmitting packets. Using this function enables the sender to change the condition to mark ECE. If the return value is set to non-zero, the kernel sets an ECT bit for the packet.

The DCTCP implementation uses both appended functions as follows. The function `ecnpkt_handler()` is used to set the ECE flag only when the received packet sets a CE bit. The function `ect_handler()` is used to set an ECT bit to all outgoing packets to estimate the accurate extent of the congestion.

4.2 Implementation choices

Several design options remain when implementing the DCTCP algorithm. This section discusses them.

4.2.1 ECN handling during congestion recovery mode

The FreeBSD kernel defines two congestion signals, which invoke congestion control at the sender. One is an ACK packet which contains an out of sequence number. The other is an ACK packet with the ECE flag set. The former is named fast recovery, and the latter is named congestion recovery. Once the sender identifies either of two signals, it stops the increase of `CWND` for roughly one RTT.

From this background, the ECN capable sender processes packets using the congestion recovery mode when it identifies a ECE flag for an RTT. However, this is exactly a part of the ECN processing defined in the standard ECN algorithm. When considering the DCTCP implementation, the necessity of this mode is an open question.

The DCTCP implementation in the FreeBSD kernel chooses the invalidation of the congestion recovery. This is because the DCTCP algorithm is motivated to the available bandwidth utilization by reacting to the extent of congestion. DCTCP does not use ECN as a congestion event but uses ECN as a metric to understand the extent of a congestion.

A simple test is conducted in order to figure out the influence of implementation choices. The network environment is built by three servers. One of them is a software switch that uses `dummynet` [24] to mark ECN. `Dummynet` is a traffic shaper to filter packets by matching parameters in the IP and TCP headers. It creates a bottleneck link with a pipe configuration. A pipe can configure a limited bandwidth, queue type, queue length and RTT. For this test, there are two pipes. Each pipe has a bandwidth set to 2Mbps, RTT set to 40 milliseconds,

the queue length set to 20 packets and the ECN threshold set to 6 packets. Unfortunately, the existing `dummynet` has no ECN option for a pipe configuration. Therefore, the ECN support for `dummynet` has been extended through this thesis. The pipe configuration is common for the experiments described in this section.

The other two servers are used as a sender and a receiver. The sender generates a TCP connection using modified DCTCP (or DCTCP) and transfers data for 10 seconds.

During the test, the sender keeps packet trace records by using `tcpdump` [3] and `siftr` [2]. The data can be used to generate four plots in the visualization. `Tcpdump` data is a source for TCP time-sequence, and throughput. `Siftr` data is used to see the processing of internal TCP parameters such as `CWND`, `ssthresh`, `SRTT (Smoothed RTT)` and `RTO (Retransmission Time Out)`. The first two TCP parameters are important to understand how the congestion control algorithm sets the amount of data to be sent. The `SRTT` value visualizes TCP's estimate of the network delay. The `RTO` value shows the longest delay when a packet encounters a time-out.

Figure 4.1 and 4.2 correspond to the original DCTCP and modified DCTCP behavior observed from 0 to 5 seconds. There are two differences between the two figures. The first difference is seen after the first slow start around 0.5 seconds in the first plot. The modified DCTCP encounters a packet loss although the original DCTCP does not. This is the influence of no congestion recovery mode. The modified DCTCP sender continues to increase `CWND` as shown in the second plot. In the result, as shown in the fourth plot, modified DCTCP makes the queue full sooner than the original DCTCP does.

The other difference is the `CWND` behavior after 0.5 seconds in the second plot. The modified DCTCP sender always advances the timing of `CWND` growth because it has no congestion recovery mode. In the result, a modified DCTCP sender observes a smoother throughput than DCTCP as seen in the fourth plot. As noted, modified DCTCP has a CWR flag after

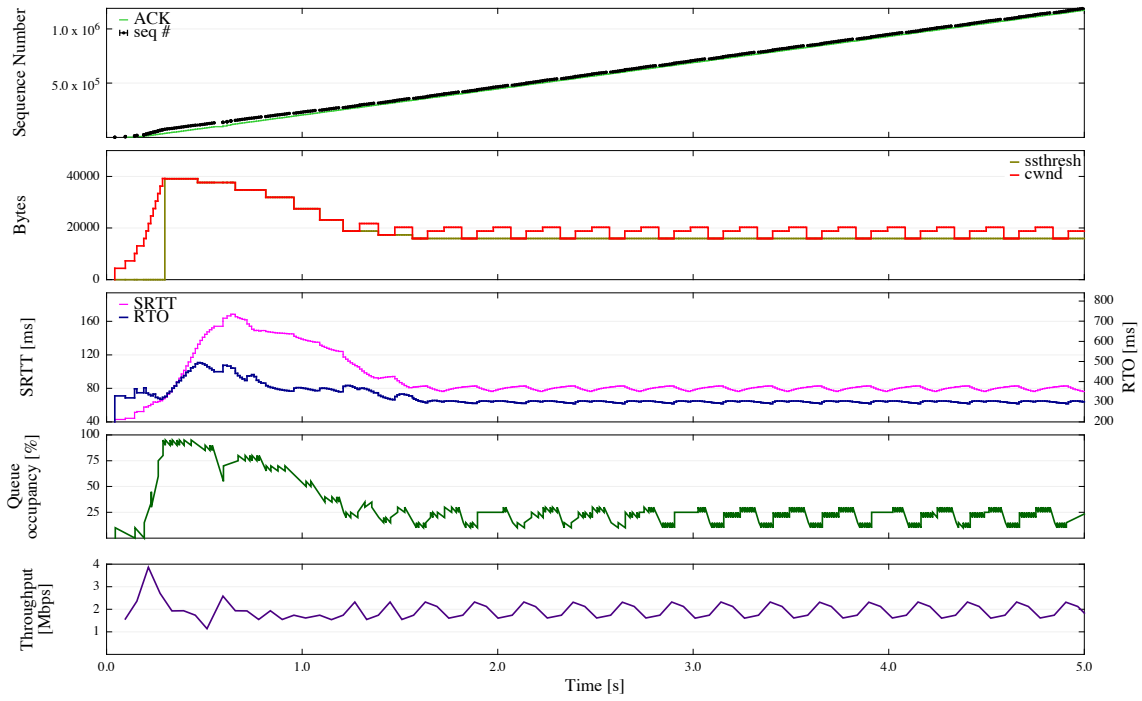

Figure 4.1: Fine-grained DCTCP behavior. The sender uses the congestion recovery mode when an arriving packet sets the ECE flag.

`CWND` reduction although DCTCP does not use the CWR flag in the first plot. The reason for this difference will be discussed in the next chapter.

4.2.2 α value initialization

α is an important parameter which indicates the extent of the congestion in the network. A DCTCP sender estimates the next `CWND` using this parameter to utilize as high bandwidth as the queue of switches allows. However, the initial α value does not correspond to the extent of a congestion because the sender has no way to estimate the extent of congestion in the network before it starts. In other words, the initial α impacts `CWND` behavior for the duration until the value convergence.

There are two choices for the initial α value: $\alpha = 0$ and $\alpha = 1$. When α is set to zero at first, the DCTCP sender transfers as much of data as possible until the α value converges.

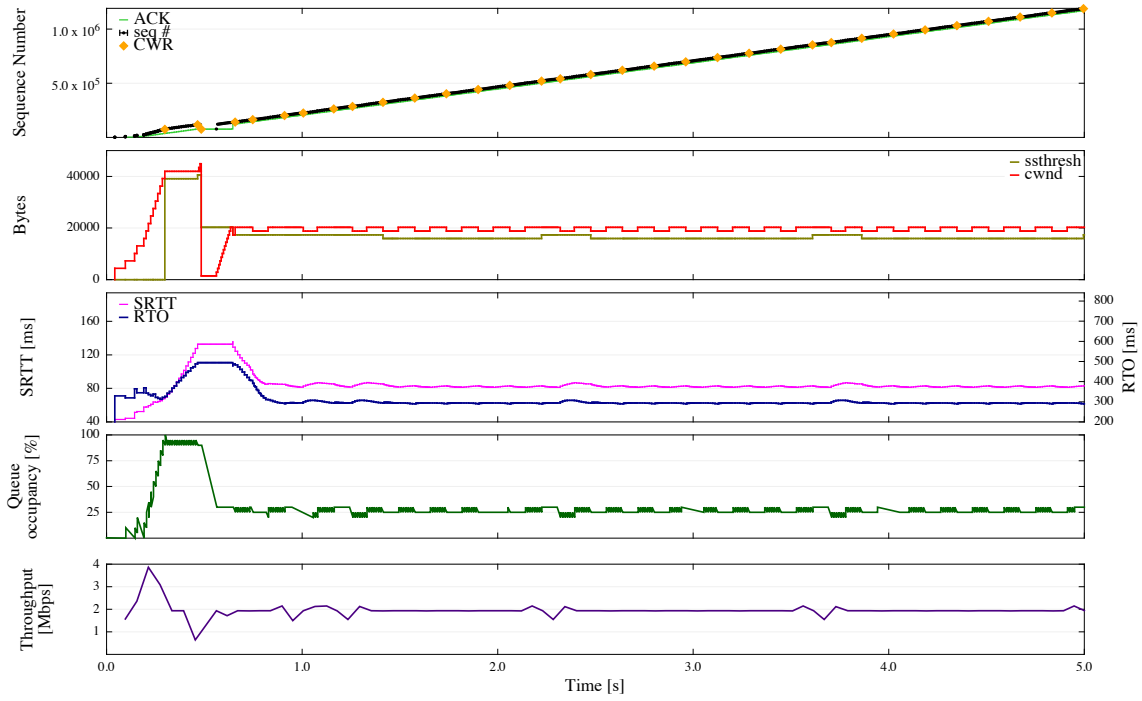

Figure 4.2: Fine-grained modified DCTCP behavior. The sender does not use the congestion recovery mode.

On the other hand, the sender may expands the queueing delay at the switch. In the worst case, it causes packet losses. When α is set to 1, the queuing delay is kept small. This results in the quick α convergence in the best case. In the worst case, the amount of data to be transferred during the α becomes much smaller than expected. Under the condition, the sender takes long time to convergence α.

The DCTCP implementation in the FreeBSD kernel prepares selective initial α because the appropriate initial α value depends on applications. When DCTCP is used for throughput-sensitive applications, the α value set to zero is preferred. When latency-sensitive applications uses DCTCP, the α value set to one is good choice because it prevents from an unnecessary queuing delay in any network environment.

Figure 4.2 and 4.3 show the fine-grained modified DCTCP behavior that set an initial α value to zero and one, respectively. When compared between the two figures, the modified

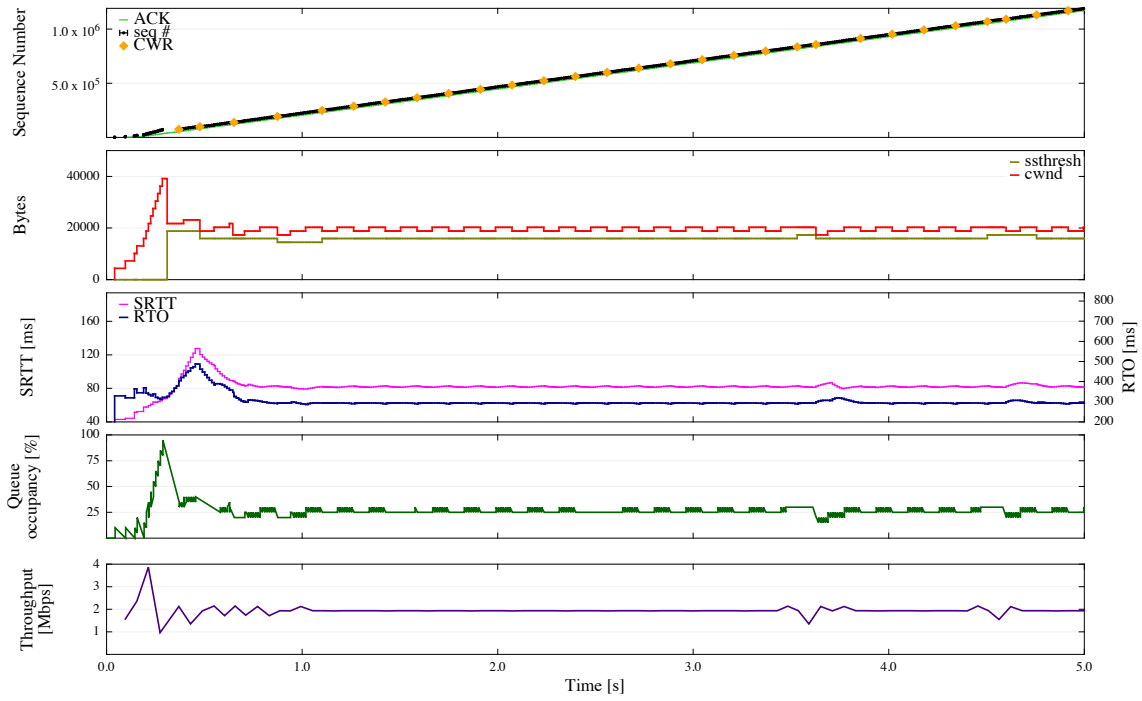

Figure 4.3: Fine-grained modified DCTCP behavior. A DCTCP sender has no congestion recovery mode and sets an initial α to one.

DCTCP using large initial α value has no packet loss around 0.5 seconds in the first figure. A modified DCTCP sender using a large initial α value refrains large `SRTT` and the high queuing occupancy in the third and fourth plots. These are exactly the trade-off between large and small initial α values described above.

4.2.3 α value handling after idle time

Idle time is rarely observed in a TCP connection, but it occurs when a receive buffer is full while the server transfers large data in the high speed network. Although the FreeBSD kernel allows to tune a receive buffer automatically, it takes long time to set an appropriate receive buffer in the high speed network.

For DCTCP senders, how to handle α under such conditions is an open question. There are two choices for the DCTCP implementation: one is to reset α, and the other is to keep

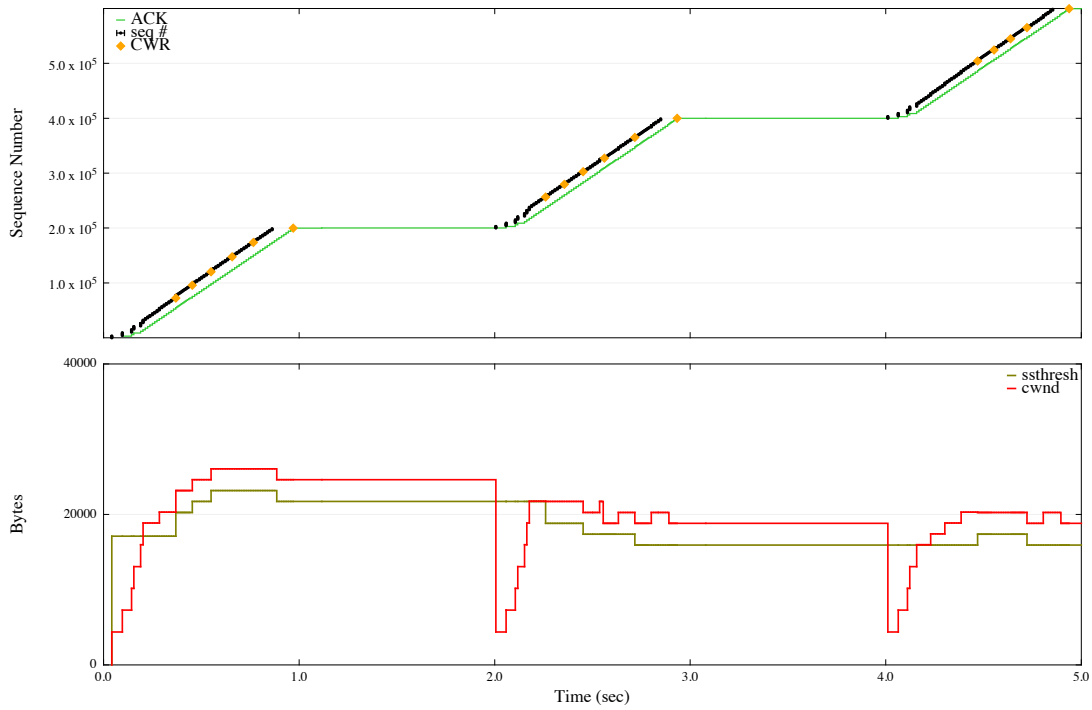

Figure 4.4: Fine-grained modified DCTCP behavior after idle time. A DCTCP sender sets initial α to zero. After idle time, the sender resets α.

α. The decision of the two choices depends on when the next congestion event is likely to occur. When the next congestion is induced by the previous one, keeping the current α value is reasonable. Otherwise, it is reasonable to reset α.

The DCTCP implementation in the FreeSBD kernel selects the reset of α because each congestion event occurs independently in the network. By applying this choice, a DCTCP sender begins the congestion control from the beginning after idle time.

Figure 4.4 shows a fine-grained modified DCTCP behavior after idle time. The experiments setup is same but a modified DCTCP sender suspends for 1 second after every 200KB data transmission. The sender configures initial α to zero for this test.

In Figure 4.4, there are two idle periods as seen in the first plot: one is from 1 to 2 seconds and the other is from 3 to 4 seconds. When the modified DCTCP sender starts data

transmission again after idle time, it transmits from 4 segments [2] when seeing the second plot. The sender increases `CWND` exponentially until `CWND` reaches `sshthresh`. Upon the first reception of an ACK packet with the ECE flag, The sender never reduces `CWND`. This behavior is safe because α set to zero means no `CWND` reduction.

[2]RFC5681 [8] defines `CWND` after idle time

Chapter 5

One-sided DCTCP algorithm

This chapter proposes the ODCTCP algorithm which allows a server to use the DCTCP algorithm on one endpoint in the network operated with standard ECN.

5.1 ODCTCP sender algorithm

5.1.1 Compatibility issue at DCTCP senders

When a sender uses DCTCP and the receiver uses standard ECN, throughput declines to the minimum value supported by the kernel [1]. Figure 5.1 reproduces the situation. (The setup for the test is explained in section 4.2). When examining the TCP time-sequence processing shown in the first plot, the number of TCP segments becomes two from 1.7 seconds. The main cause is the `CWND` processing shown in the second plot. `CWND` degrades to two segments after 1.7 seconds. The sender sets an unnecessarily small `CWND` rather than an appropriate larger value.

The result was unexpected but in retrospect easily explained. Figure 5.2 illustrates a series

[1] All congestion control algorithms implemented in the TCP stack sets at least 2 segments as `CWND` [8]

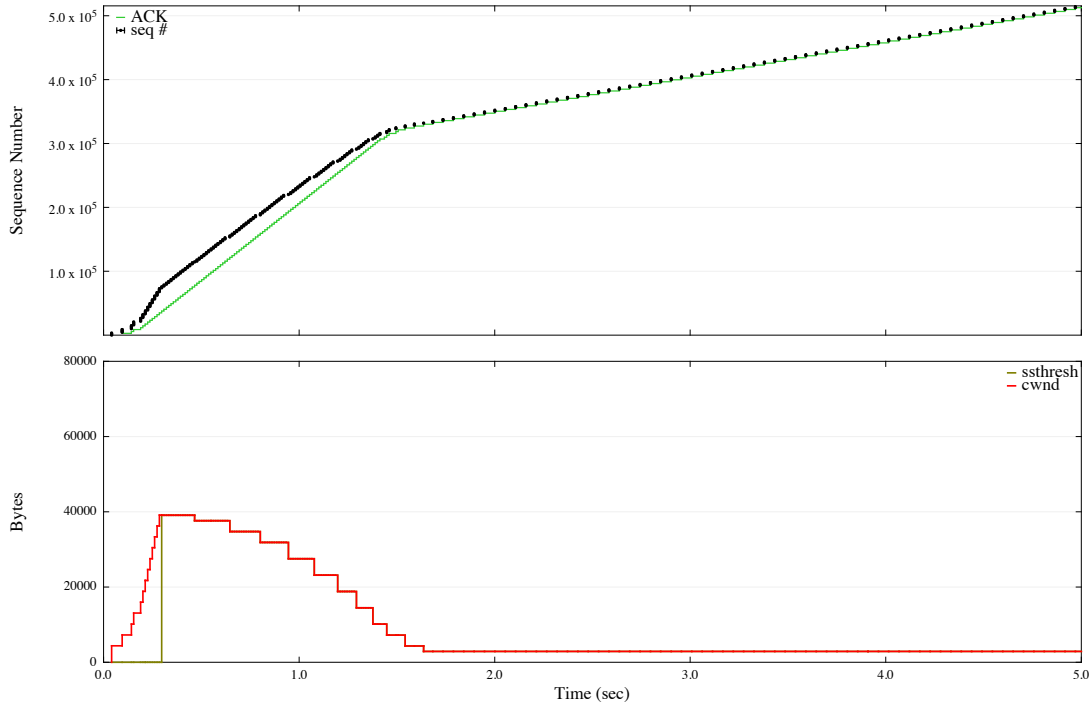

Figure 5.1: Fine grained DCTCP sender behavior: The sender uses DCTCP and the receiver uses standard ECN. `CWND` declines to the minimum supported value in the kernel and stays there.

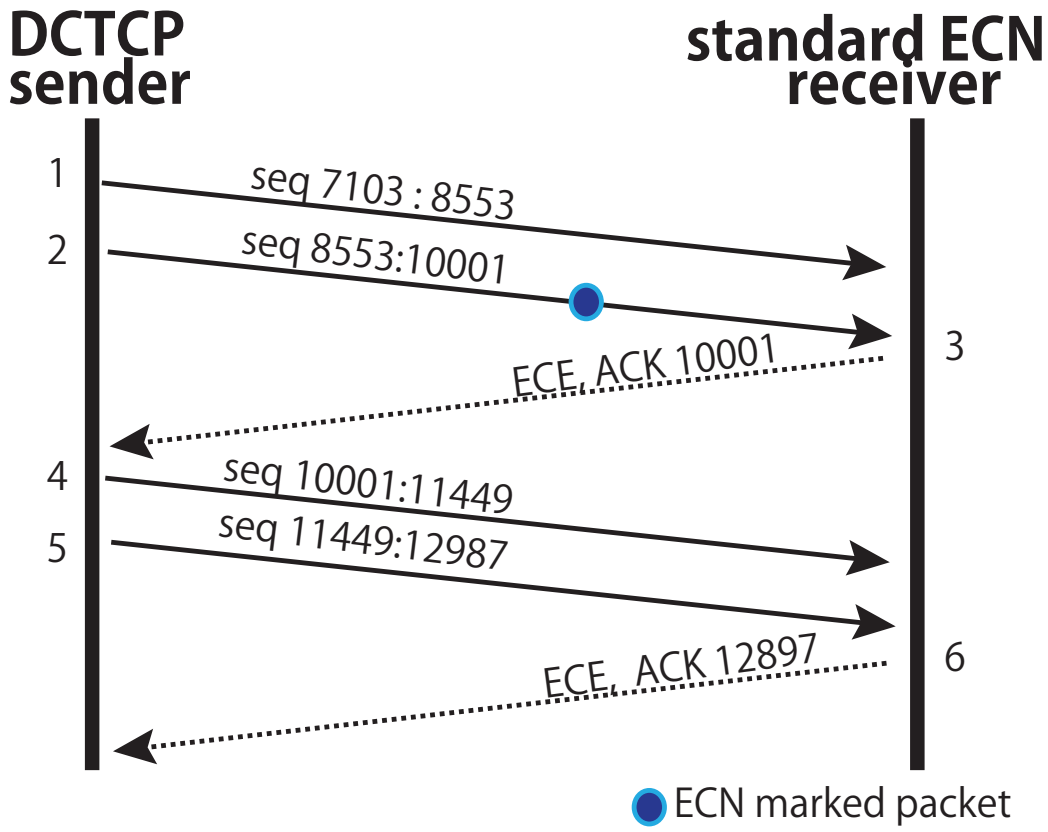

Figure 5.2: A series of the ECN processing between a DCTCP sender and a standard ECN receiver.

of ECN action between a DCTCP sender and a standard ECN receiver to explain the issue. Suppose that a packet is set a CE bit (2nd segment). When the standard ECN receiver recognizes a CE bit in an arriving packets, it feedbacks an ACK packet with an ECE flag (3rd segment). Uopn the reception of an ACK packet with the ECE flag, The DCTCP sender updates `CWND`, then, transmits two segments (4 - 5th segment).

The standard ECN receiver responds ACK packets with the ECE flag until the peer transmit a packet with the CWR flag (See section 2.1). However, the DCTCP sender never sets the CWR flag because the CWR flag is not used in the DCTCP algorithm. Because of this difference between the standard ECN and DCTCP sender behavior, the DCTCP sender

continues to receive ACK packets with the ECE flag till the end of this TCP connection (6th segment). In the end, the DCTCP sender converges `CWND` to the minimum value supported by the kernel.

5.1.2 Compatibility support at ODCTCP senders

An ODCTCP sender sets a `CWR` flag to avoid an unexpected degradation of `CWND` values. Figure 5.3 shows a series of the ECN action between an ODCTCP sender and the standard ECN receiver. Unlike a DCTCP sender, an ODCTCP sender sets the CWR flag in the outgoing packet after the receipt of an ECE flag (4th segment). Therefore, the standard ECN receiver stops to set ECE flag in subsequent ACK packets and the ODCTCP sender returns the first process as expected.

Figure 5.4 shows a fine-grained ODCTCP sender behavior. The same test is conducted with Figure 5.1 although the sender uses ODCTCP. As expected, the TCP time-sequence diagram shows a straight line shown in the first plot. In addition, `CWND` is kept reasonable value in the second plot.

Note that the ODCTCP sender algorithm compromises to count the number of ECE flags accurately. In the DCTCP algorithm, a sender estimates the number of ECN marked packet as accurately as possible. However, this policy cannot be applicable for an ODCTCP sender. This is because a standard ECN receiver continues to set the ECE flag in ACK packets for a RTT once it identifies a CE bit in an arriving packet. The standard ECN receiver responds with the larger number of packets with ECE flag than the DCTCP receiver does. Thus, the ODCTCP sender estimates a larger number of ECN marked packets than expected in the end.

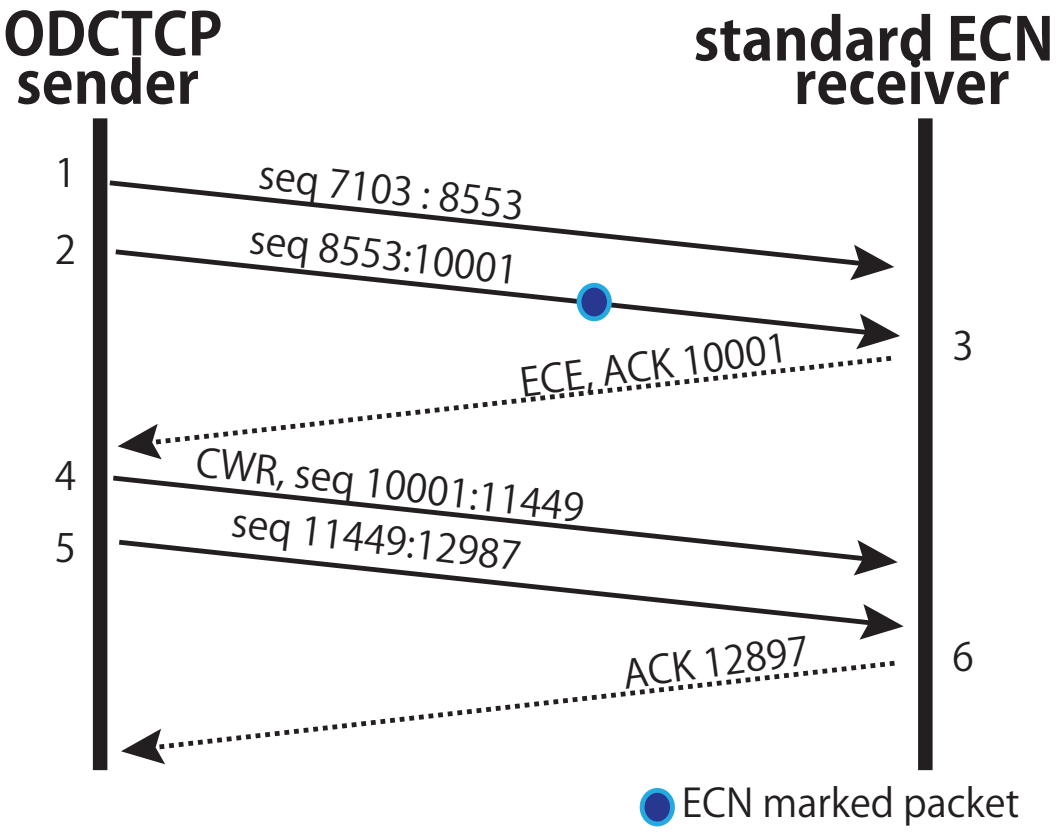

Figure 5.3: A series of the ECN processing between an ODCTCP sender and a standard ECN receiver.

5.2 ODCTCP receiver algorithm

5.2.1 Problematic ECN feedback at DCTCP receivers

A receiver algorithm rarely has a possibility to improve data transmission performance because the sender algorithm controls the `CWND`. However, when the sender uses standard ECN and the receiver uses DCTCP, the DCTCP receiver can trigger problematic behavior on a standard ECN sender due to the different interpretation of the ECN bits and their translation.

A transmission performance degradation occurs when a DCTCP receiver responds an

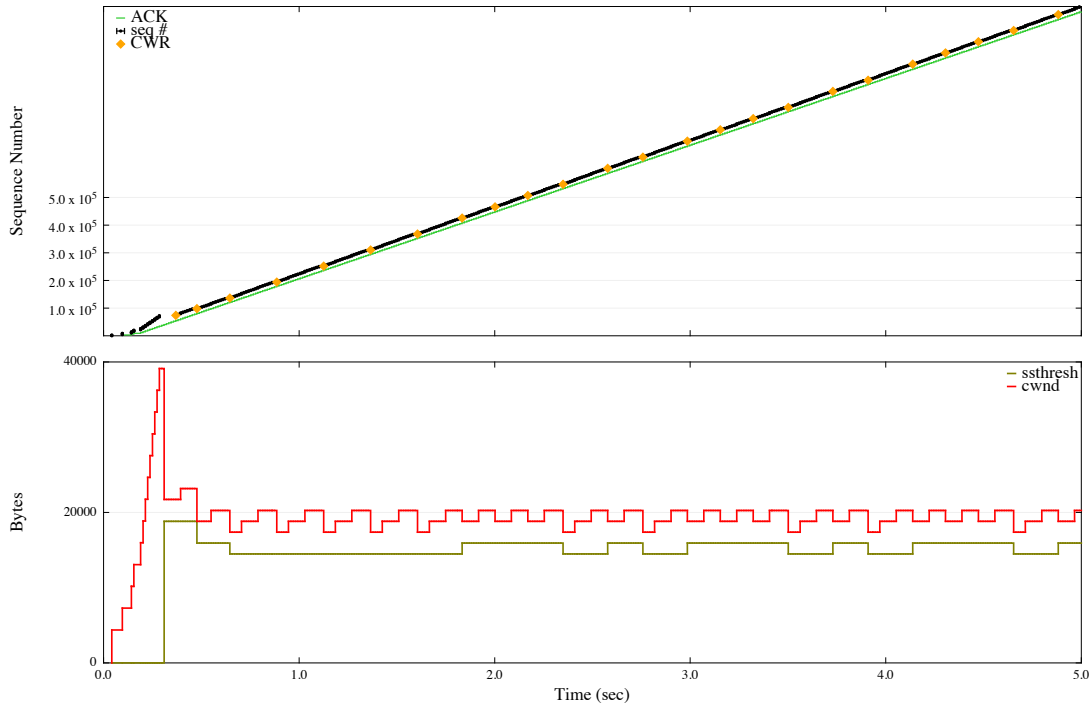

Figure 5.4: Fine grained ODCTCP behavior. When the sender uses ODCTCP and the receiver uses standard ECN, `CWND` stabilizes at a larger value, resulting in the stable TCP segments transmission.

ACK packet that triggers to increase `CWND` at the sender. This problem happens only when the receiver use delayed ACK. Figure 5.5 shows the TCP time-sequence diagram which reproduces the problematic situation. Because the situation is hard to explain with one figure, Figure 5.6 is used for the comparison. In Figure 5.6, both a sender and a receiver use standard ECN. Both figures are generated using the experimental setup described in Section 4.2.

A DCTCP (or standard ECN) receiver get a CE and an ECT bit by turns in arriving packets seven times on the way. Once the receiver detects a CE bit, it feedbacks an ACK packet with the ECE flag. Therefore, the TCP time sequence in both figures shows that the standard ECN sender sets the CWR flag. After then, the sender transmits 7 segments during one RTT (One RTT corresponds to the duration that an ACK number reaches the

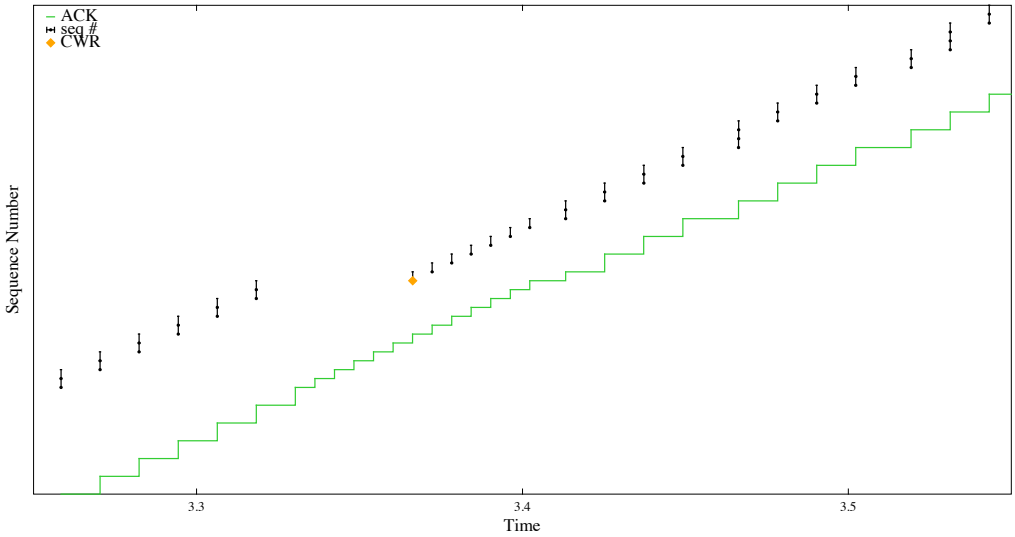

Figure 5.5: Fine grained TCP time-sequence diagram when a sender uses standard ECN and a receiver uses DCTCP. After the standard ECN sender transmits a packet with the CWR flag, it transmits six segments. Then it transmits two segments at 3.41 seconds. In the result, the total number of segments to be transmitted for roughly one RTT is eight.

corresponding TCP sequence number). The point is the number of segments to be transmitted next. The standard ECN sender transmits 2 segments at 3.41 seconds in Figure 5.5, but the sender transmits 4 segments at 3.49 seconds in Figure 5.5. A DCTCP receiver makes the standard ECN sender transmit the smaller number of packets than a standard ECN receiver does.

The main cause of this issue comes from the temporal disabling of delayed ACK at the DCTCP receiver. Figure 5.7 illustrates a series of the ECN action between the standard ECN sender and the DCTCP receiver. Once the standard ECN sender detects the ECE flag in an ACK packet, it reduces `CWND` and enters the congestion recovery mode which stops `CWND` increasing for a RTT from the next outgoing packet (3rd segment). The standard ECN sender set CWR flag in the next outgoing packet (4th segment). The DCTCP receiver

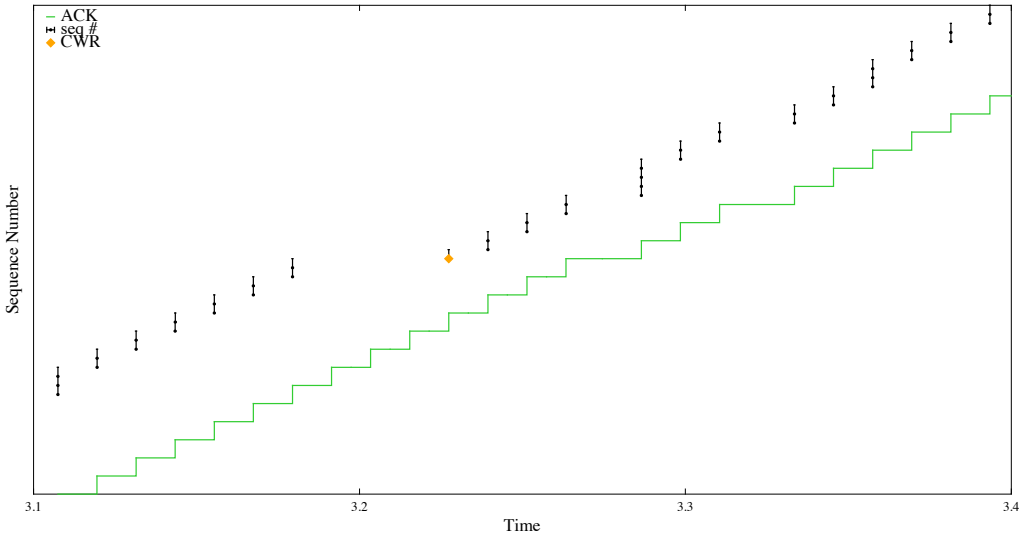

Figure 5.6: Fine grained TCP time-sequence diagram when a sender and a receiver use standard ECN. After the standard ECN sender transmits a packet with the CWR flag, it transmits six segments. Then it transmits four segments at 3.49 seconds. In the result, the total number of segments to be transmitted for roughly one RTT is twelve.

identifies the turns of CE bits and report it to the sender via an ACK packet without an ECE flag by disabling delayed ACK (6th segment). As this ACK packet includes the ACK number which exceeds the sequence number of the packet with the CWR flag, the sender exits congestion recovery mode (7th segment) and starts to increase `CWND` again.

The standard ECN sender adds one segment into `CWND` but this is not an expected behavior. According to RFC 3465 [7], the delayed ACK algorithm allows to increase `CWND` by two segments. Thus, the standard ECN sender should add two segments in theory. However, because the DCTCP receiver temporally disables delayed ACK, the standard ECN sender allows to increase `CWND` by one segment.

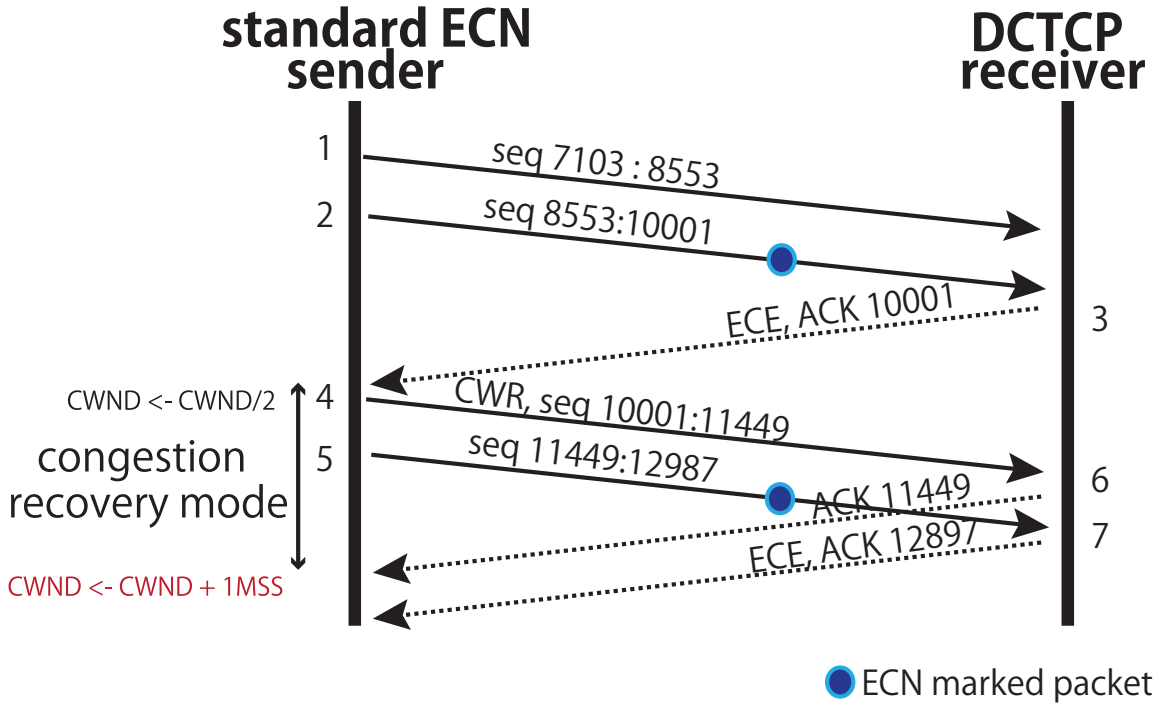

Figure 5.7: A series of the ECN processing between a standard ECN sender and a DCTCP receiver.

5.2.2 Modified ECN feedback at ODCTCP receivers

To avoid this situation, an ODCTCP receiver enables delayed ACK when an arriving packet sets the CWR flag. Figure 5.8 shows a series of the ECN action between a standard ECN sender and an ODCTCP receiver. Unlike a DCTCP receiver, an ODCTCP receiver enables delayed ACK when the CWR flag is set in the packet (6th segment). In the result, the standard ECN sender can increase `CWND` by two segments.

The ODCTCP receiver algorithm compromises the accurate estimation of ECN marked packets when the sender uses DCTCP. As the ODCTCP receiver enables delayed ACK for a packet with the CWR flag, the DCTCP receiver cannot report a turn of the CE bit to the DCTCP sender correctly. However, this is a minor problem to be considered. Unless the

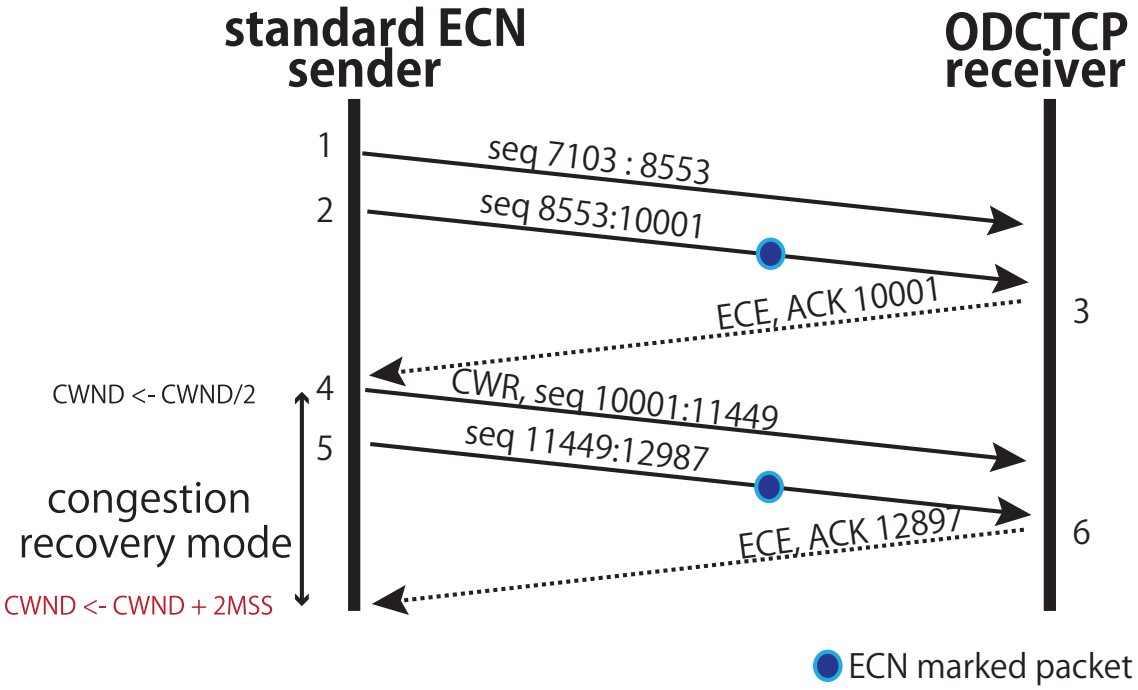

Figure 5.8: A series of the ECN processing between a standard ECN sender and an ODCTCP receiver.

`CWND` is quite small, the influence of incorrect ECN feedbacks is small, which can be ignored.

Chapter 6

Evaluation setup

This chapter describes metrics, methodologies, network topologies and kernel parameter choices in the evaluation.

6.1 Scenarios

The evaluation includes two performance investigations. One investigates modified DCTCP and original DCTCP. The other investigates the ODCTCP performance. They both are evaluated via microbenchmarks and a benchmark described as below. Microbenchmarks purposes to see the protocol transmission performance on the specific traffic patterns. A benchmark, on the other hand, emulates data center network traffic using a measurement insights for targeting service operation.

6.1.1 Microbenchmark

Microbenchmark contains three experimental scenarios.

(1) incast: Incast occurs when many flows converge in the same egress queue of a switch for short time. The burst of incoming packets fills the queue of a switch. As a result, packets are dropped the queue. This traffic pattern occurs regardless of packet size.

For this test, 10 flows [1] which transfer a specific amount of data are started at the same time. The amount of data varies from 10KB to 800KB. Each flow transfers data just once.

(2) Queue buildup: Queue buildup occurs when several long flows and many short flows share an egress queue of a switch. Because the long flows fills up the queue, short flows experience either long queuing delay or packet losses when they arrive. This condition affects the data transfer time of both long and short flows.

In order to generate this traffic pattern, many short flows are started while multiple long flows are already running. An initial delay before starting short flows is configured. The number of short flows is set to 10 and the number of long flows is set to either 2 or 8. The initial delay is set to 500 milliseconds. Each flow transfers data just once.

[1]The packet processing for the large number of flows with a NIC generate unexpected delay in a NIC because arriving packets are passed to a kernel sequentially. To avoid the possibility, the number of flows is set to the relatively small number.

(3) Fairness test: The final scenario is typical fairness test. This test is investigated to see the fairness between ODCTCP and standard ECN flows and the fairness between ODCTCP flows. The number of flows is set to 2, 4, 8, 16 and 32 and these flows are run for 20 seconds. The number of servers split into the number of flows in each test.

6.1.2 Benchmark

How DCTCP or ODCTCP perform under the traffic patterns found in production data centers is hard to conclude from the result of the microbenchmarks. This is because the microbenchmarks do not reproduce typical data center traffic but use simplified traffic patterns For example, in the queue buildup test, how much impact short flows see from long flows depend on the delay to start short flows.

To study these considerations, a benchmark test is conducted. Servers generate 560 short flows and 40 long flows [2] and transfer data between endpoints chosen randomly. The traffic generation rule is as follows. Flows are started to transfer a different amount of data from 1KB to 50MB. Once a server starts a flow, a server decides on an inter-arrival time according to the distribution shown in Figure 6.1. This distribution is constructed from a measurement result of a data center network which operate target applications. The data size distribution of flows is random because this distribution was impossible to reproduce from figure 4 in [5]. The concurrent number of flows is not considered because flows are generated according to the flow inter-arrival time distribution. Five different benchmark tests are conducted for this test. The duration of each test is around 30 seconds.

6.2 Metrics

The experimental scenarios described above are evaluated using the following set of metrics.

[2]Why this number is chosen because the performance evaluation tool, `flowgrind`, does not work well when the number of flows to be generated in a command is more than 600.

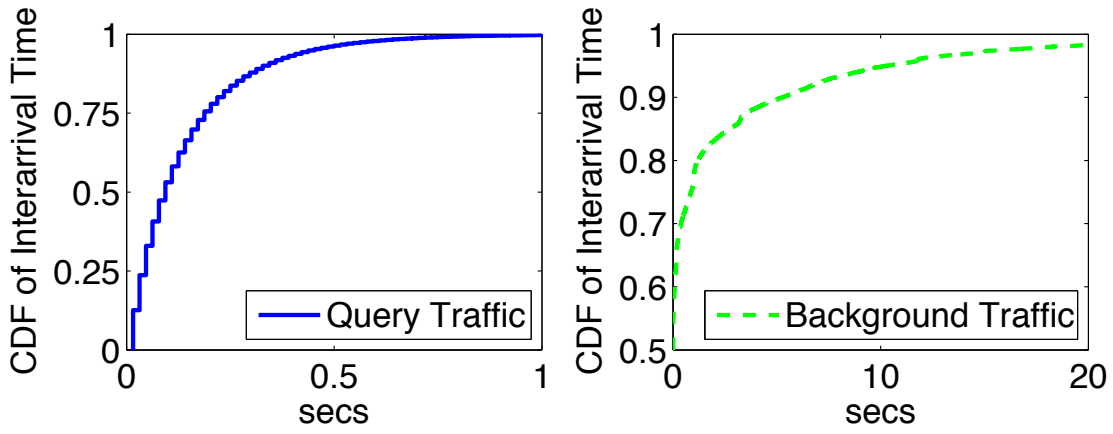

Figure 6.1: Inter-arrival time of short flows and long flows between servers referred from Figure 3 in [5].

(a) Average of data transfer time and `SRTT`: The data transfer time is equivalent to `goodput`, which is the application-level throughput. Because a data center frequently deals with time-sensitive flows, the data transfer time is appropriate to use in the evaluation. All the test scenarios (except for fairness) use data transfer time as a metric.

The `SRTT` value corresponds to the smoothed average of TCP's RTT estimate. Queuing delay can be used as a metric but it is not used because capturing the queuing delay increases overhead in a switch.

There is a problem with using the `SRTT` metric. Due to the TCP timer granularity in the FreeBSD kernel, 1 millisecond is the minimum value that the kernel can express. Because of this limitation, this metric is used only in the incast tests to roughly estimate the queuing delay of a switch.

(b) Fairness index and instant throughput: The fairness itself is evaluated by Jain's fairness index [21]. The equation 6.1 shows the equation to compute a fairness index when n flows are running. X_i is i-th flow's normalized throughput. The range of fairness index is from 0 to 1. If we observe 1 as fairness index, each flow is given fair bandwidth.

$$\text{Jain}'\text{s fairness index} = \frac{\Sigma(x_i)^2}{n\Sigma x_i^2} \tag{6.1}$$

Throughput is important when studying fairness. Even when the fairness index is perfect, a low utilization of the bottleneck link is problematic. Therefore, the average of throughput is used as the metric to check the bandwidth utilization. This thesis defines good fairness when the fairness index is reasonably large (> 0.9) and when the utilization of the link achieves around 80% against the maximum bandwidth of a link [3]. Throughput values are captured every 50 milliseconds by `flowgrind` [34].

6.3 Performance measurement tool

The traffic generation for all the test scenarios is performed by `flowgrind` [34]. There are two benefits to use `flowgrind`. First, `flowgrind` allows to assign control traffic and test traffic separately. `Flowgrind` prepares a controller and daemons unlike other tools [22, 29]. The controller sends a command for traffic generation to daemons, and daemons starts to run test traffic according to the command. This is useful for simultaneous measurement between any hosts.

	NUTTCP	thrulay	flowgrind
Average	6.25 %	1.05 %	0.30 %
Median	3.43 %	0.66 %	0.22 %
Top Outliers	23.71 %	6.05 %	1.21 %
	23.20 %	4.84 %	1.11 %
	22.76 %	4.42 %	1.10 %
	22.18 %	3.72 %	1.01 %
	21.04 %	3.69 %	0.99 %

Table 6.1: Effects of process scheduler on variation between flows (ubuntu 9.04, Linux kernel 2.6.28, CFS scheduler)

[3] A packet contains 40bytes for TCP and IP headers. Considering the header field, 77% of a packet fills with data.

The other benefit is process scheduling of daemons. Different performance can be observed if the measurement tool forks either additional processes or threads. This problem occurs regardless of the number of flows due to the influence of task scheduling of operating systems. `Flowgrind` avoids this problem by adopting a single thread for all traffic. Table 6.1 displays the effect of `flowgrind`'s process scheduling, which is referred from [34]. The authors consecutively measure the TCP `goodput` between the same two machines over standard Fast Ethernet with the three different tools `NUTTCP` [16], `thrulay` [26] and `flowgrind`. For each reporting interval of 50 ms the absolute difference in `goodput` between the two flows relative to the total `goodput` in that interval, as measured by the source, has been calculated. The throughput difference between two flows using `flowgrind` is obviously small in the comparison with other tools.

The following command is an example of `flowgrind` usage.

```
$ flowgrind -n 1 -E -H s=host1,d=host2 -Z s=10000 -Y s=0.1
```

The command will generate a TCP connection (-n option) from `host1` to `host2` (-H option). `Host1` gets three-way handshake down before generating a test traffic. Then, `host1` starts to transfer 10KB bulk data (-Z option) to `host2` after 0.1 second initial delay (-Y option). Each packet contains randomized integers in the data segment (-E option). The completion time of data transmission is signaled from `host2` by an ACK packet which includes the arrival time of the last segment. The combination of these options performs all the experiment scenarios.

6.4 Experimental network environment

DCTCP and ODCTCP are evaluated in the real network testbed and in the emulated network environment with `dummynet`. The testbed is used to run a set of microbenchmarks.

There is a single-bottleneck on a path. The emulated network environment is used to evaluate realistic workloads. It reproduces traffic for the inter-rack and intra-rack. To achieve this network traffic, three switches are required, however, the testbed has a switch which supports ECN. As the emulated network environment does not restrict the absolute number of switches, the realistic workloads is evaluated in the emulated network environment. The CPU and memory of servers were never a bottleneck in both experiments.

6.4.1 Real network testbed with a Nexus 3548 switch

Figure 6.2 illustrates the experimental testbed network for microbenchmarks. The network environment is composed of four servers and one Cisco switch. Three servers are senders (S1 - S3) and one emulates receivers (R1 - R2). S1 transfers data to R1. S2 and S3 transfer data to R2. R1 and R2 are hosted in a server. Two receivers uses two different IP addresses configured on one interface. Each server is equipped with 4-core Intel Xeon processors 3.0 GHz CPU, 16 GB RAM and two Intel 1 Gigabit Ethernet controllers. The FreeBSD kernel version is 11.0.

The Cisco Nexus 3548 has three deep-buffers, which allows sharing a large buffer pool among neighboring 16 switch ports. The amount of a buffer is 6MB. The Cisco Nexus 3548 always enqueues a packet to an ingress queue which is prepared every switch port. The packet stored in the ingress queue is passed to one of shared buffers which resides the switch port connected to the destination. The ECN processing is applied at this moment. When the instantaneous queue length of a buffer pool has already exceeded the ECN threshold, the switch sets a CE bit of the packet. In all the microbenchmarks, the ECN threshold is set to 20 packets. Why this value is chosen as the ECN threshold is because the DCTCP paper concludes this value is small enough to prevent from the burst of packets at a switch [5].

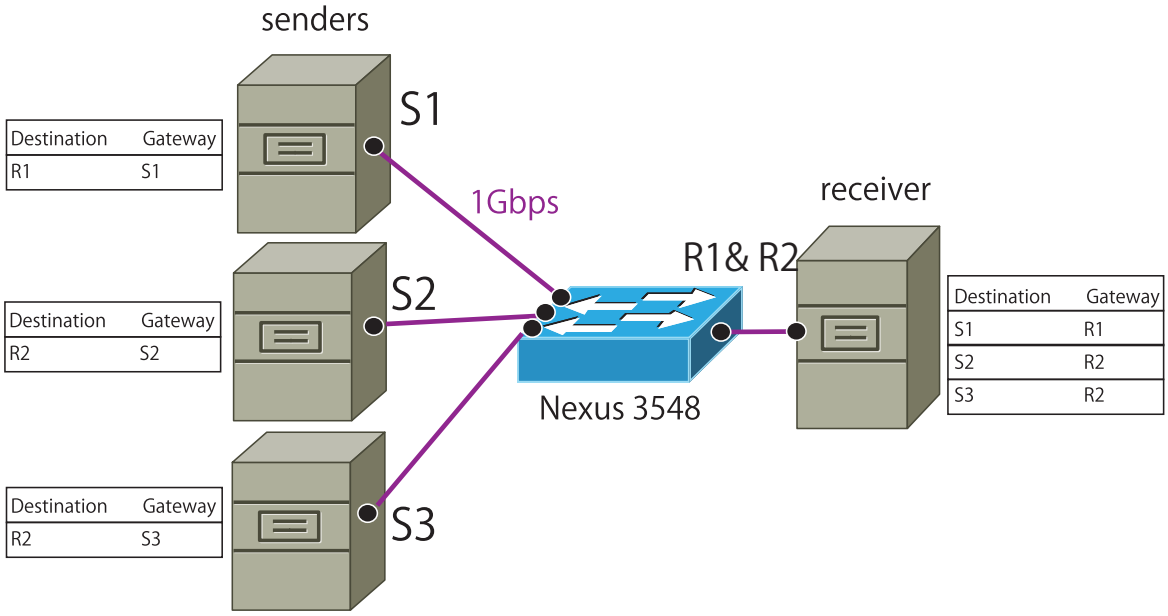

Figure 6.2: A single-bottleneck network topology with Nexus3548

6.4.2 A multi-hop network with `dummynet`

The emulated network for the benchmark consists of five servers shown in Figure 6.3. One of them is used as a `dummynet` machine. The flows grouped in S1 and S2 are served by one server. This server transmits packets over two different interfaces for S1 and S2. Another server serves the S2 group. The last server serves the R1 and S3 groups over two different interfaces. All servers have Intel 1 Gigabit Ethernet controllers, but CPUs and RAM size are differed. The `dummynet` machine has 6 Intel Core i7-3930K 3.2GHz CPUs and 4GB RAM. A server belonging to S1 and S2 groups is equipped with 8 Intel Core i7-2600 3.4GHz CPUs and 8GB RAM. A server of S3 group is equipped with 8 Intel Core i7-2600 2.67GHz CPUs and 20GB RAM. A server in R1 and R2 groups is equipped with 8 AMD FX processors and 4GB RAM.

The senders in the S1 and S3 groups all transfer data to a single receiver whose destination is R1. The senders in S3 each transfer data to an assigned receiver R2. The network traffic is

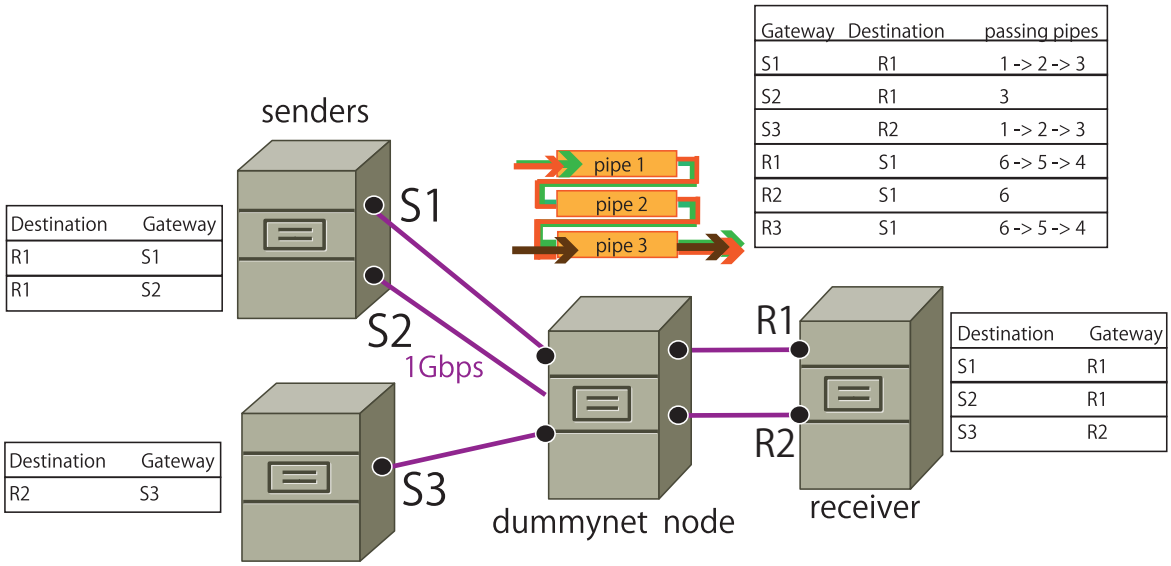

Figure 6.3: A multi-bottleneck network topology with `dummynet`

shaped by `dummynet` over six pipes, which emulate three switches. Three pipes shapes traffic from senders and three shapes traffic from receivers. Pipe 1 and 3 set the bandwidth to 100Mbps, RTT to 10 milliseconds, the queue length to 100 packets and the ECN threshold 30 packets. Pipe 2 sets bandwidth to 1Gbps, RTT to 1 milliseconds, the queue length to 100 packets and the ECN threshold 30 packets. The configured queue length is larger than BDP [4] because this thesis assumes the use of a Cisco Nexus 3548. The ECN threshold is chosen through the experience of many experiments. The bandwidth between Pipe 1 and 2 and the bandwidth between Pipe 2 and 3 are oversubscribed to reproduce network traffic in the inter-rack and intra-rack. As used in the actual data centers, the bandwidth which aggregates inter-rack network traffic is 10 times greater than the bandwidth for the inter-rack network traffic.

The network topology contains two bottlenecks: both the link between Pipe 1 and 2 and the link connecting Pipe 3 to R1. Flows from group S2 passes the second bottleneck and

[4] According to the BDP equation, the proper queue length is equivalent to 86 packets.

flows from group S3 passes the first bottleneck. Flows from the senders in group S1 encounter both these bottlenecks.

6.5 Kernel TCP Parameters

Setting kernel parameters for TCP is important for a performance evaluation of congestion control algorithms. Several parameters are set as below.

net.inet.tcp.initcwnd10: This parameter allows the initial window size to increase to 10 segments [13]. However, a large initial window tends to lose the last segment and triggers a timeout for low speed network [14]. Therefore, the emulated network conducts tests by disabling this parameter.

net.inet.tcp.tso: This parameter permits a server to use TSO (TCP Segmentation Offload), which reduces the CPU overhead for processing a large number of small packets on a high speed network. This parameter is disabled in the emulated network because TSO does not influence the result of test scenarios in a low speed network.

TCP host cache: The TCP host cache saves the TCP parameters observed for the last TCP connection instance. An entry in the cache includes the `RTT` estimate to the remote server, `CWND`, `ssthreshold`, `MTU` and bandwidth-delay product (BDP) [28]. Reusing the values can improve the performance of TCP connections between the same hosts. Both network environments disable the host cache and isolate individual TCP connection in a test to see the pure transmission performance.

Nagle algorithm: The Nagle algorithm is intended to reduce the number of small packets over the network. Applying this algorithm has a great benefit to the network and operating system. Suppose an application which generate small packets such as 1 byte at a time. Transmitting such small packets causes the network and an operating system to be overloaded. However, for applications in a data center network, this algorithm works harmful

because most flows transfer only a small amount of data. Flows without Nagle algorithm tend to get unnecessary latency before transmitting a last packet. Therefore, the Nagle algorithm is disabled to avoid such a situation in both network environments.

Chapter 7

Evaluation

This chapter evaluates DCTCP and ODCTCP performance using both microbenchmarks and benchmarks. The first section investigates DCTCP performance in the FreeBSD kernel. The next section evaluates the modifications applied the DCTCP implementation. The last section evaluates ODCTCP performance.

7.1 DCTCP performance validation

As switches tell incipient congestion by using ECN, DCTCP flows provide the short queueing delay if many DCTCP flows transfer data into a queue. In addition, as a queue length is kept short by using ECN, each DCTCP flow achieves the almost same data transfer time. On the other hand, SACK TCP flows have no way to identify incipient congestion. They increase the amount of transmitting data until they encounter a packet loss. In the result, if many SACK TCP flows transfer data into a queue, they induce the high queueing delay by transmitting the large number of packets at once.

Incast: Figure 7.1 shows the result of the incast test. When many flows converge in the same queue of a switch for a short period, the packets exhaust the queue and the switch generates the long queueing delay. The tendency is high when the amount of data that senders transfer is large as shown in the top plot of Figure 7.1. DCTCP flows refrain the increasing ratio of the data transfer time against the amount of data rather than SACK TCP flows. When

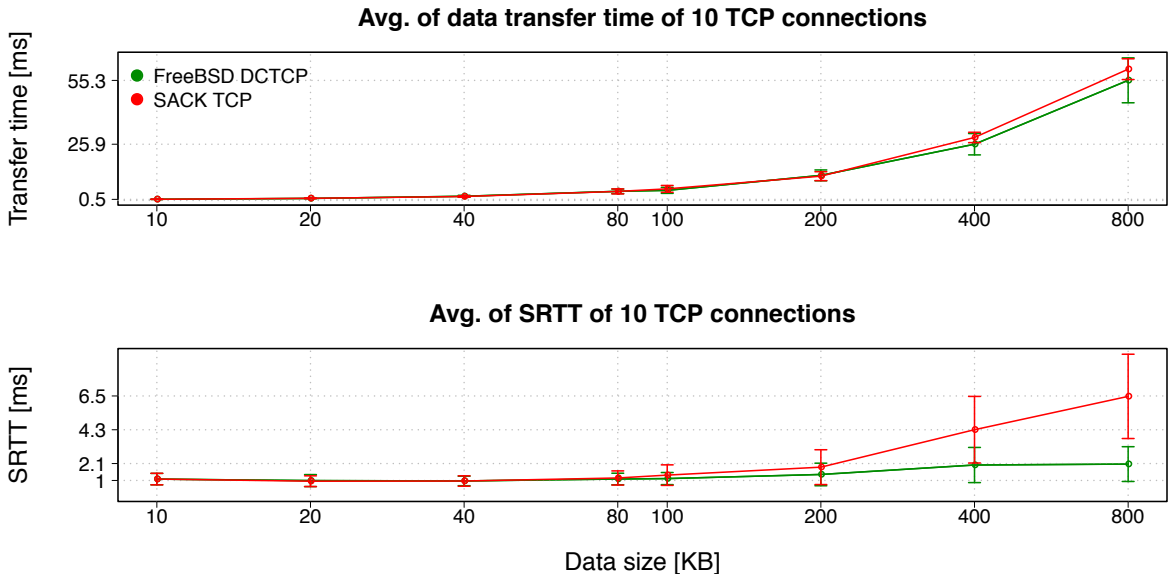

Figure 7.1: The result of the incast test. Because DCTCP flows maintain the short queueing latency, they achieve 1.1 times faster 800KB transmission than SACK TCP flows.

senders transfer 800KB, DCTCP flows achieve 1.1 times faster data transmission than SACK TCP flows.

The difference of the data transfer time between SACK TCP and DCTCP flows corresponds to the queueing delay. Because the switch limits the number of arriving packets in the queue by marking ECN to packets, DCTCP flows keep the short queueing delay. It result in the short data transfer time rather than SACK TCP flows.

The influence of the queueing delay is seen in the bottom plot of Figure 7.1. DCTCP flows sustain the small `SRTT` even when the amount of data is differed. DCTCP flows increase the `SRTT` to 1 millisecond when the amount of data to be transferred increases from 10KB to 800KB. On the other hand, SACK TCP flows grow the `SRTT` increasing to 3.1 times from 10KB to 800KB transmission.

The standard deviation of the `SRTT` should be highlight to see the queueing delay. When the short queue length is maintained, it should be small because any flows do not receive an impact from the queueing delay. This is true in the result. In transmitting 800KB, because DCTCP flows keep short queueing delay, they show 2.5 times as small standard deviation as SACK TCP flows.

Queue buildup: Figure 7.2 shows the result of the queue buildup test when two long flows are running. When long flows have exhausted a queue, SACK TCP short flows significantly delay to transfer the small amount of data due to the long queueing delay. On the other hand, DCTCP short flows do not delay unlike SACK TCP flows because using ECN prevents from the high queue occupancy. In transmitting 10KB, DCTCP short flows save 33 milliseconds for the data transfer time compared to SACK TCP flows as shown in the top plot of Figure 7.2. This corresponds to 33 times of small data transfer time for DCTCP flows. The increasing ratio of data transfer time is small for DCTCP flows when the amount of data to be transferred is large. When DCTCP flows transfer 800KB, they save 170.2 millisecond

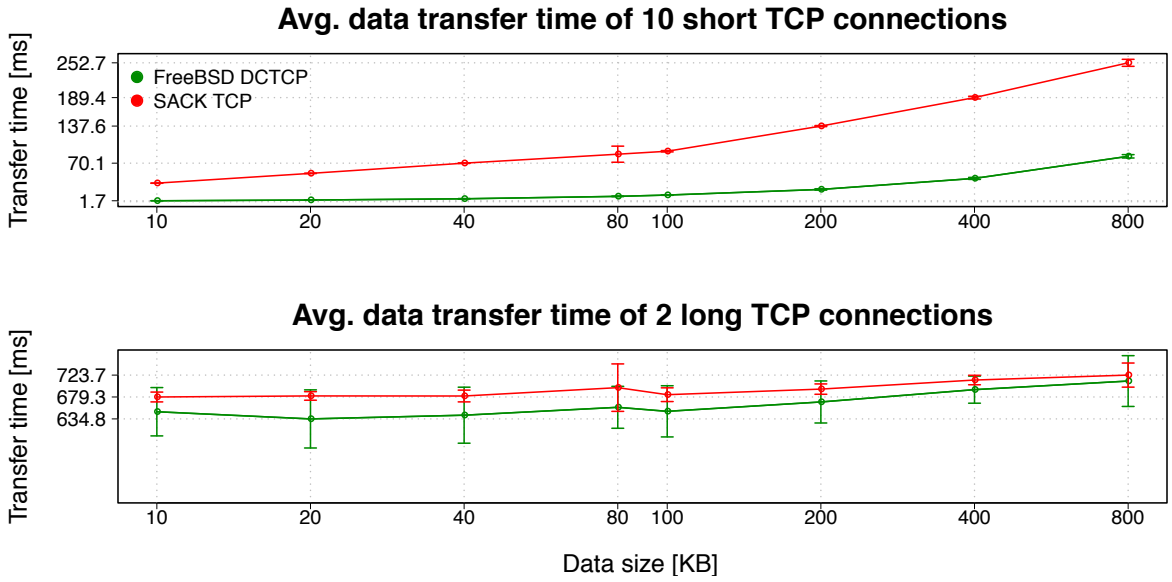

Figure 7.2: The result of the queue buildup test when two long flows are running. SACK TCP short flows receive an impact to the data transfer time by the presence of long flows but DCTCP flows keep the short data transfer time.

compared to SACK TCP flows, which corresponds to 3 times short data transfer time.

When compared to the data transfer time observed in the incast test, we can see the impact of the presence of long flows. As SACK TCP long flows exhaust a queue, SACK TCP short flows require additional 33.1 milliseconds for data transfer time in transmitting 10KB when long flows exist. This corresponds to 67 times the increasing of the data transfer time. When SACK TCP flows transfer 800KB, the data transfer time in the queue buildup test is 4.2 times as long as one in the incast test. On the other hand, DCTCP flows get a small impact from the presence of long flows. Because DCTCP prevents from burst traffic by using ECN, it mitigates the impact of the presence of long flows. DCTCP short flows slow the data transfer time to 3.1 times in transmitting 10KB and to 1.5 times in transmitting 800KB.

When seeing the data transfer time of long flows as shown in the bottom plot of Figure 7.2, both SACK TCP and DCTCP flows observe similar data transfer within less than 40 milliseconds difference in any amount of data transmission, which corresponds to a 4%

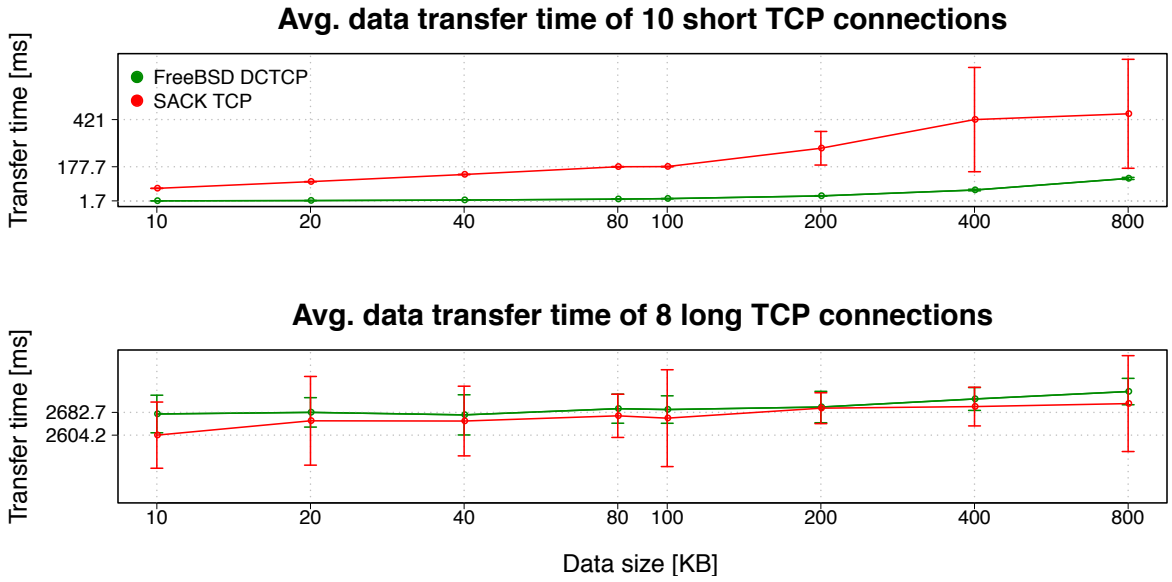

Figure 7.3: The result of the queue buildup test when eight long flows are running. SACK TCP flows increase the data transfer time to double in proportion to the number of long flows. On the other hand, DCTCP flows show less than 1.5 times of an increase of data transfer time even when the number of long flows is increased.

difference. The difference is not statistically significant.

Figure 7.3 shows the result of the queue buildup test when the number of long flows is increased to eight. In this test, we can see the impact of data transfer time of short flows when the number of long flows increases. When compared to the top plot in Figure 7.2, SACK TCP short flows take twice the data transfer time for 10KB and 800KB transmission. When seeing the standard deviation of the data transfer time of SACK TCP short flows, SACK TCP short flows observe more than 20 times large standard deviation in transmitting 800KB. This is the impact of the queueing delay induced by the large number of long flows. Because the large number of SACK TCP long flows exhaust a queue, SACK TCP short flows get an impact to the data transfer time compared to the result of the previous queue buildup test.

On the other hand, DCTCP flows avoid the expansion of the data transfer time. DCTCP

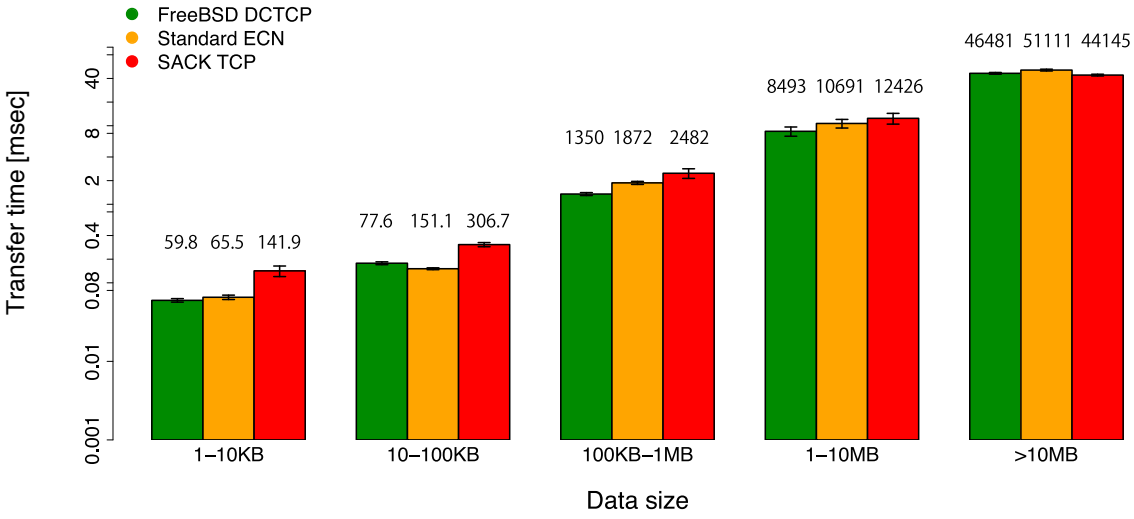

Figure 7.4: Benchmark result for DCTCP performance validation. DCTCP flows achieve short data transfer time in comparison with SACK TCP flows. When compared to standard ECN flows, DCTCP flows show short data transfer time especially for long flows.

flows show the same data transfer time for 10KB transmission and show 1.4 times long data transfer time for 800KB transmission when compared to the top plot in Figure 7.2. Even when the number of DCTCP long flows is increased, DCTCP long flows do not delay DCTCP short flows. Since any DCTCP flows receive the short queueing delay thanks to ECN, DCTCP flows achieve the short data transfer time.

Benchmark: Figure 7.4 shows the data transfer time of SACK TCP, standard ECN and DCTCP flows observed in the benchmark test [1]. In this figure, to simplify the result, the data transfer time is averaged regardless of the number of bottlenecks that flows pass. Thereby, the error bar shows 95% of the confidential interval instead of the standard deviation. The small confidential interval means that the average of the data transfer time converges. Note that the fine-grained range of data size is used for x-axis and the log scale of the data transfer time is used for y-axis.

[1]DCTCP senders set α to zero in the benchmark test.

DCTCP flows show the shortest data transfer time in mixed network traffic of short and long flows as shown in Figure 7.4. In comparison to SACK TCP flows, DCTCP flows show the improvement of the data transfer time in any amount of data transmission. In particular, DCTCP flows improves the data transfer time to 2.4 times for 1 - 10KB transmission, to 1.7 times for 10 - 100KB transmission, to 1.9 times 100KB - 1MB transmission and to 1.5 times for 1MB - 10MB transmission. This is because the switch maintains the low queue occupancy with ECN. Although SACK TCP flows show 5% of shorter data transfer time than DCTCP flows for more than 10MB transmission, the actual difference of the data transfer time is less than 2 milliseconds. This is a very small impact considering the data transfer time in transmitting more than 10MB (= around 45 seconds).

In the result of the benchmark test, we also verify the advantage of DCTCP flows by comparing to standard ECN flows. When seeing Figure 7.4, DCTCP flows improves the data transfer time to 1.1 times for 1 - 10KB transmission, to 1.4 times 100KB - 1MB transmission, to 1.3 times for 1MB - 10MB transmission and to 1.1 times for more than 10MB transmission. Standard ECN flows react to the available bandwidth too aggressively in the receipt of ECN while DCTCP flows try to utilize the available bandwidth as much as possible. The mechanism difference makes standard ECN flows slow compared to DCTCP flows. It results in the longer data transfer time compared to DCTCP flows.

7.2 Modified DCTCP performance validation

This section investigates transmission performance of DCTCP flows using modifications done in this thesis with microbenchmark tests. Each figure shows a microbenchmark result and each result evaluates two things: the first and second modifications described in Section 4.2. To evaluate the two things, each result prepares four labels in the legend. The first label is named “Linux DCTCP” which has no modification. Note that “Linux DCTCP” is

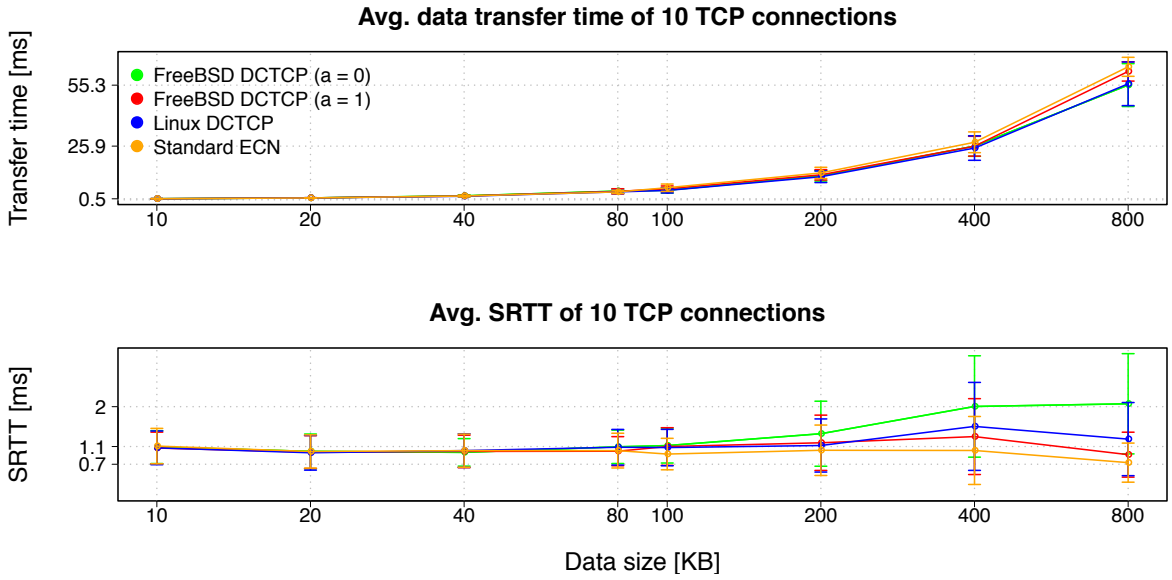

Figure 7.5: The result of the incast test. FreeBSD DCTCP flows using the first modification show shorter data transfer time than Linux DCTCP flows. When FreeBSD DCTCP flows set a large initial α, they get similar performance with standard ECN flows.

not equivalent to the Linux patch [1] but the FreeBSD implementation which applies the same internal parameters and procedures used in the Linux patch to the FreeBSD kernel. The second label named "FreeBSD DCTCP (a = 0)" and the third label named "FreeBSD DCTCP (a = 1)" are equivalent to modified DCTCP. They set different initial α values. The second label sets the initial α value to zero. The third label sets the initial α value to one. The fourth label is "Standard ECN".

To see the improvement of transmission performance using the first modification, the comparison of "FreeBSD DCTCP (a = 0)" with "Linux DCTCP" is used. To evaluate the second modification, We compare "FreeBSD DCTCP (a = 1)" to "Standard ECN". Because DCTCP flows with the large initial α value behave like standard ECN flows in the beginning, they should show the similar performance with standard ECN flows.

Incast: Figure 7.5 shows the result of the incast test. The difference of data transfer time between Linux DCTCP and FreeBSD DCTCP is less than 1 millisecond. On the other

hand, the difference of SRTT values is seen from over 100KB transmission. When FreeBSD DCTCP flows transfer 800KB, they show 0.8 milliseconds larger SRTT than Linux DCTCP flows. This is the influence of the first modification. As a FreeBSD DCTCP sender never enters congestion recovery mode, it sets large CWND aggressively while a Linux DCTCP sender stops the CWND growth. The aggressiveness of the CWND growth like DCTCP flows results in the SRTT.

When using a large initial α for FreeBSD DCTCP senders, DCTCP flows reduce 3% of the data transfer time in transmitting 800KB compared to the data transfer time of standard ECN. Although DCTCP flows with the second modification reacts to the receipt of ECN like standard ECN flows in the beginning of a TCP connection, they achieve the high utilization of a available bandwidth in the end. On the other hand, because of the aggressive DCTCP sender's behavior, DCTCP flows with a large initial α expand the SRTT to 1.2 times rather than standard ECN flows.

Queue buildup: Figure 7.6 shows the result of the queue buildup test when eight long flows are running. The advantage of the DCTCP modifications is also seen when short and long flows are mixed. DCTCP flows finish the data transmission around 1.1 times faster than standard ECN in the maximum. On the other hand, Linux DCTCP flows show 2% of shorter data transfer time for long flows than FreeBSD DCTCP flows in the maximum and show 1.4 times larger standard deviation than FreeBSD DCTCP flows. This is the influence of the first modification. The congestion recovery mode stops the CWND increasing for a RTT, thus, the competitive short flows get more bandwidth. As a side effect of the cumulative call of the congestion recovery mode, DCTCP long flows without the first modification make the large standard deviation.

Interestingly, the data transfer time of DCTCP flows using a large initial α value is 5% of longer than that of standard ECN flows in the maximum. This is because DCTCP short flows

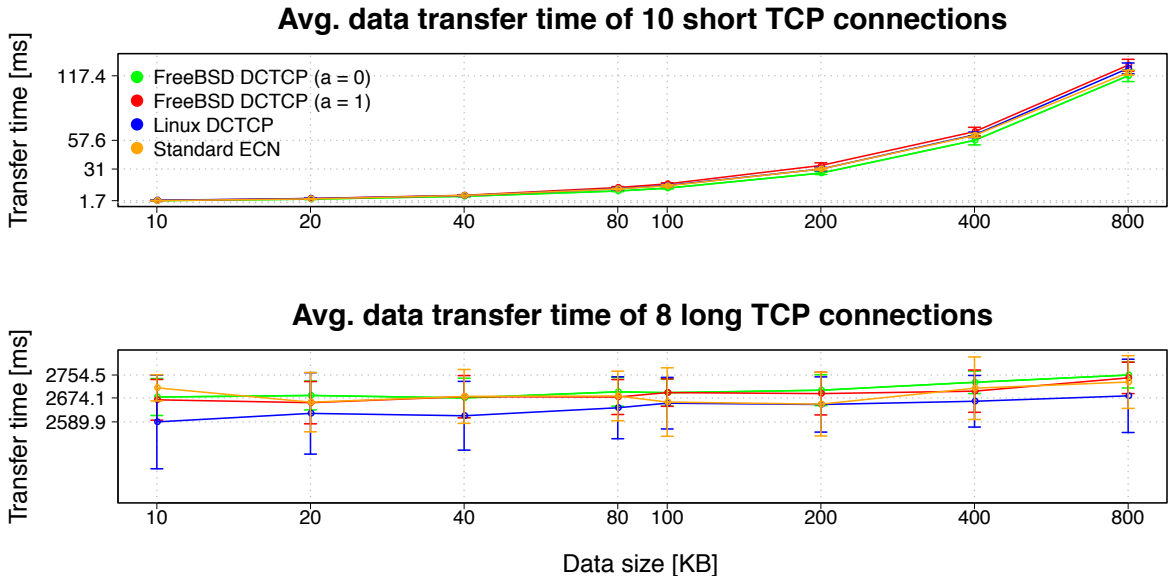

Figure 7.6: The result of the queue buildup test. FreeBSD DCTCP short flows using the first modification only show short data transfer time than Linux DCTCP short flows while keeping short data transfer time for long flows. DCTCP short flows using the large initial α show 2% of an increase in the data transfer time rather than standard ECN short flows.

cannot quickly get high throughput when DCTCP long flows are running. At the moment that short flows start, long flows have occupied an available bandwidth. When DCTCP is used for long flows, long flows has utilized as much bandwidth as possible. Therefore, it takes long time to get a bandwidth for standard ECN short flows. On the other hand, when standard ECN is used for long flows, it is easy to get a bandwidth for DCTCP short flows because standard ECN long flows save the input rate against an available bandwidth.

7.3 ODCTCP performance investigation

7.3.1 The combination of ODCTCP servers

There are many evaluation scenarios for ODCTCP when considering the combination of the ODCTCP deployment to a network. This thesis chooses the simplest situation from them.

Table 7.1: Evaluation scenarios for ODCTCP senders

	short flows		long flows	
label no.	senders	receivers	senders	receivers
1	Standard ECN	Standard ECN	Standard ECN	Standard ECN
2	ODCTCP	Standard ECN	Standard ECN	Standard ECN
3	Standard ECN	Standard ECN	ODCTCP	Standard ECN
4	DCTCP	DCTCP	DCTCP	DCTCP

Table 7.2: Evaluation scenarios for ODCTCP receivers

	short flows		long flows	
label no.	senders	receivers	senders	receivers
1	Standard ECN	Standard ECN	Standard ECN	Standard ECN
2	Standard ECN	ODCTCP	Standard ECN	Standard ECN
3	Standard ECN	Standard ECN	Standard ECN	ODCTCP
4	DCTCP	DCTCP	DCTCP	DCTCP

The ODCTCP performance evaluation is divided into ODCTCP senders and ODCTCP receivers. Table 7.1 shows all the protocol combination for the ODCTCP sender's evaluation . Senders use either standard ECN or ODCTCP and receivers use standard ECN. The table 7.2 shows all the protocol combination for the ODCTCP receiver's evaluation. Senders use standard ECN and receivers use either ODCTCP or standard ECN. The two tables separate into short and long flows that use ODCTCP. This is because applications that use ODCTCP depends on data center networks.

In order to quantify ODCTCP performance, choosing baselines is essential. This thesis defines the first and the fourth label shown in Table 7.1 and Table 7.2 as baselines. This is because the ODCTCP evaluation is motivated to see the performance in the incremental deployment path. The first label assumes the network which is operated with the existing kernel. The fourth label is equivalent to the ideal network which is operated with DCTCP [2]. The experimental result of the second and the third label are expected to improve transmission performance in the comparison with the result of the first label but are expected to

[2]DCTCP means modified DCTCP as below

underperform in the comparison with the result of the fourth label.

Label numbers shown in Table 7.1 and Table 7.2 are used to understand the protocol combination easily. The following figures of experimental results are represented by these label numbers. Note that how the incast test and the fairness test represent the labels. For the incast test, the experimental results are expressed by referring the congestion control algorithm of short flows. Therefore, the first, second and fourth labels in the two tables are used to show the result of the incast test. The fairness test does not use the result using labels because it evaluates fairness between ODCTCP and standard ECN flows, or the fairness between ODCTCP flows. Figures for the fairness test express each experimental condition instead of labels.

7.3.2 Performance investigation for ODCTCP senders

This subsection shows the transmission performance when sending servers that use ODCTCP share a bottleneck with standard ECN flows. This thesis abbreviates a flow that work with an ODCTCP sender on one endpoint as an ODCTCP flow in this subsection. Note that ODCTCP flows win the data transfer time when ODCTCP and standard ECN flows transfer data at the same time. This is because an ODCTCP sender always tries to use an available bandwidth as much as possible while a standard ECN sender reacts too aggressively to the available bandwidth.

Incast: Figure 7.7 shows the result of the incast test. When seeing the data transfer time as shown in the top plot of Figure 7.7, there is no difference between ODCTCP and DCTCP flows (See the result of the first and the third labels). Also, the `SRTT` in any amount of data transmission is matched more than 99% between ODCTCP and DCTCP flows. This means that ODCTCP flows behave like DCTCP flows in the incast traffic.

When comparison between (O)DCTCP and standard ECN flows, (O)DCTCP flows show

1.2 times short data transfer time in transmitting 800KB. This result is reasonable because (O)DCTCP flows try to use an available bandwidth as much as possible but a standard ECN flows react too aggressively to the available bandwidth.

This influence is represented in the `SRTT` as shown in the bottom plot of Figure 7.7. For example, standard ECN flows show 3% of smaller `SRTT` than (O)DCTCP flows in transmitting 800KB. Because standard ECN flows react too much against the available bandwidth, they achieve the shorter queueing delay than (O)DCTCP flows.

Queue buildup: Figure 7.8 shows the result of the queue buildup test when two long flows are running. When seeing data transfer time as shown in the top plot of Figure 7.8, in the maximum, ODCTCP short flows (the second label) reduce the data transfer time to 1.2 times compared to standard ECN short flows (the first label). This is reasonable because ODCTCP short flows try to use an available bandwidth as much as possible while standard ECN long flows generate a space against the available bandwidth.

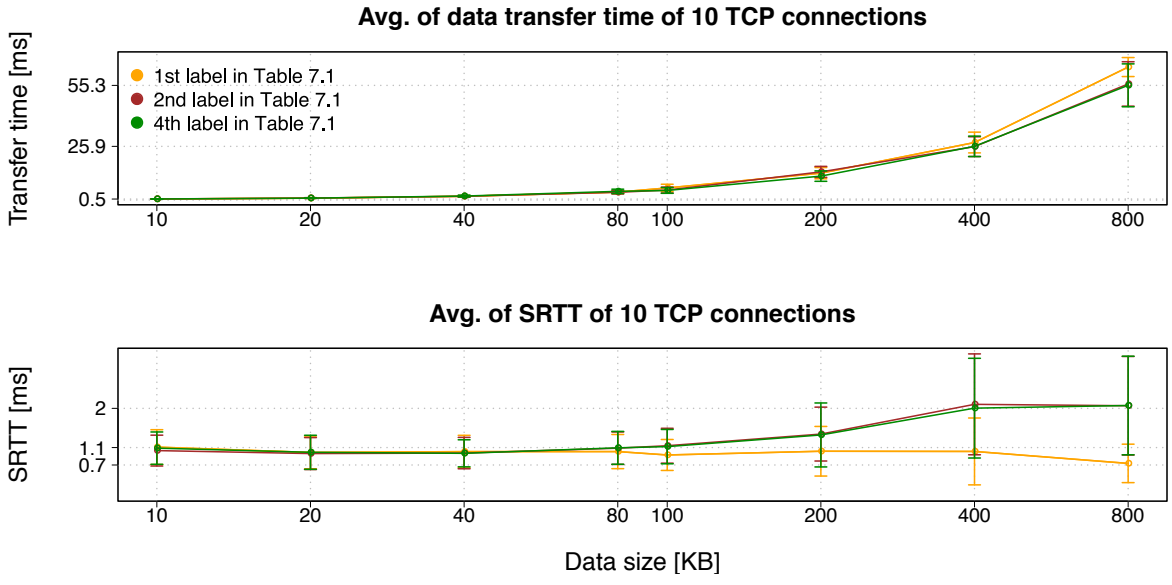

Figure 7.7: The result of the incast test. There is no difference of the data transfer time between ODCTCP flows and DCTCP flows.

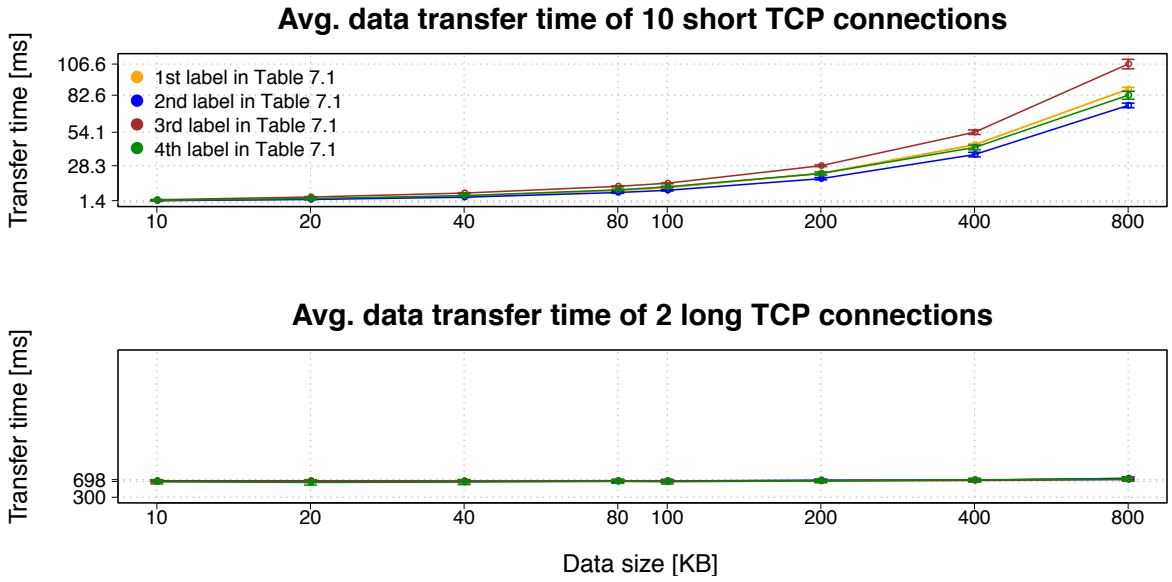

Figure 7.8: The result of the queue buildup test when two long flows are running. Using ODCTCP for short/long flows improves the data transfer time of short/long flows as expected. As a side effect, competitive standard ECN flows require long data transfer time, especially for short flows.

ODCTCP short flows (the second label) improve the data transfer time to 1.3 times compared to DCTCP short flows (the fourth label). This is because standard ECN long flows make a space to the bottleneck bandwidth rather than DCTCP long flows. In the result, ODCTCP short flows get an available bandwidth more quickly than DCTCP short flows and achieve short data transfer time.

When ODCTCP is used for long flows (the third label), the data transfer time of standard ECN short flows are 1.3 times longer than either that of standard ECN short flows (the first label) or that of DCTCP short flows (the fourth label) in the maximum, which is the inverse result of the second label. Standard ECN short flows take long time to get the bandwidth when ODCTCP long flows are running.

When seeing the data transfer time of long flows, any experimental scenarios in labels show less than 30 milliseconds difference, which is equivalent to a 4% difference. The difference is

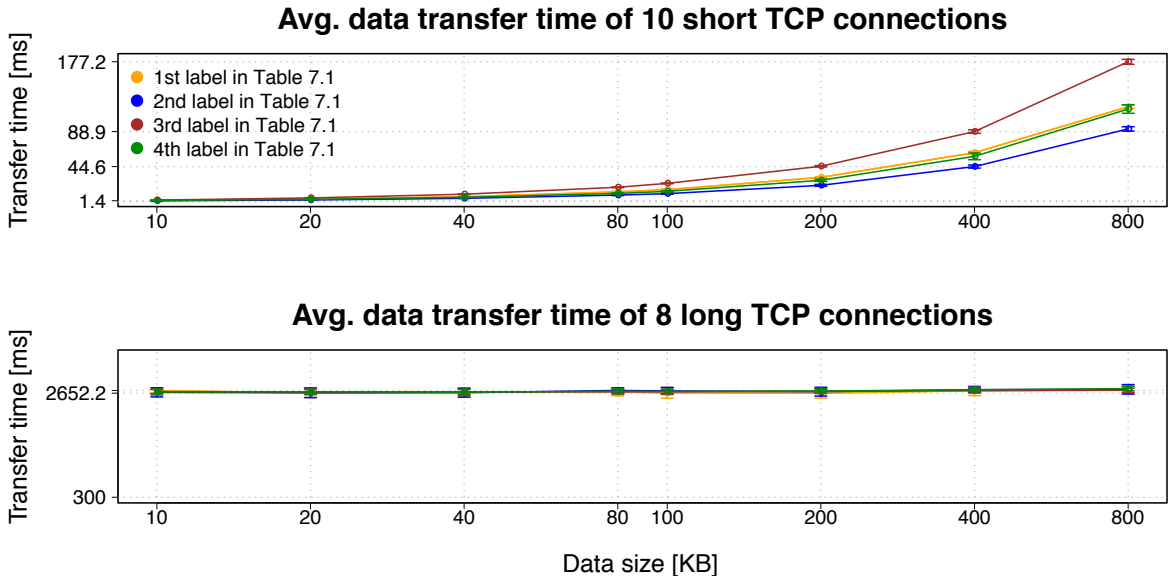

Figure 7.9: The result of the queue buildup test when eight long flows are running. Using ODCTCP for short/long flows improves the data transfer time of short/long flows as expected. As a side effect, competitive standard ECN flows require long data transfer time, especially for short flows.

not statically significant.

Figure 7.9 shows the result of the queue buildup test when the number of long flows is increased to eight. The features of the result is exactly same when the number of long flows is two.

Fairness: The result of the fairness test is shown in Figure 7.10. Fairness indices between ODCTCP flows show more than 0.99 regardless of the number of flows. In addition, the average throughput corresponds to around 80% of the bandwidth with small standard deviation. Thus, we can say that a fair utilization of the bottleneck bandwidth is achieved between competitive ODCTCP flows.

On the other hand, fairness indices between ODCTCP and standard ECN flows show small values. Especially when the competitive number of flows is two, the fairness index shows the smallest value, which corresponds to 0.7. This is because ODCTCP flows win the bandwidth

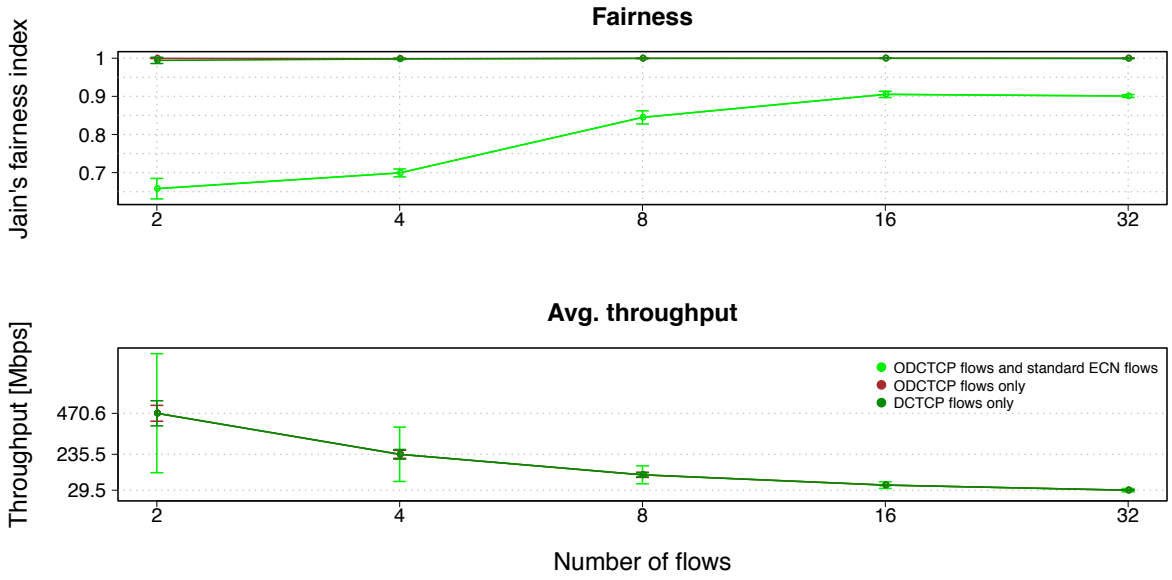

Figure 7.10: Fairness test result. When ODCTCP and standard ECN flows share a bandwidth, ODCTCP flows get more bandwidth than standard ECN flows. In the result, unfairness indices are observed. Otherwise, ODCTCP senders show good fairness.

utilization against standard ECN flows as described in the result of the queue buildup test. This influence is seen in the plot of the average throughput. When two flows are running, around 80% of the bandwidth utilization is achieved between ODCTCP and standard ECN flows but the standard deviation is pretty high, which is equivalent to 40% of the average throughput.

Benchmark: The benchmark result shown in Figure 7.11 makes sure the study of microbenchmarks. When ODCTCP is used for short flows (the second label), ODCTCP short flows improve the data transfer time compared to standard ECN short flows (the first label). ODCTCP flows show 1.2 times as short data transfer time as standard ECN for 1 - 10KB transmission and 1.2 times as short data transfer time as standard ECN for 10 - 100KB transmission. In addition, when seeing the data transfer time of long flows, standard ECN long flows in the second label almost improves the data transfer time compared to standard ECN long flows in the first label. This is a by-product from ODCTCP short flows. Since

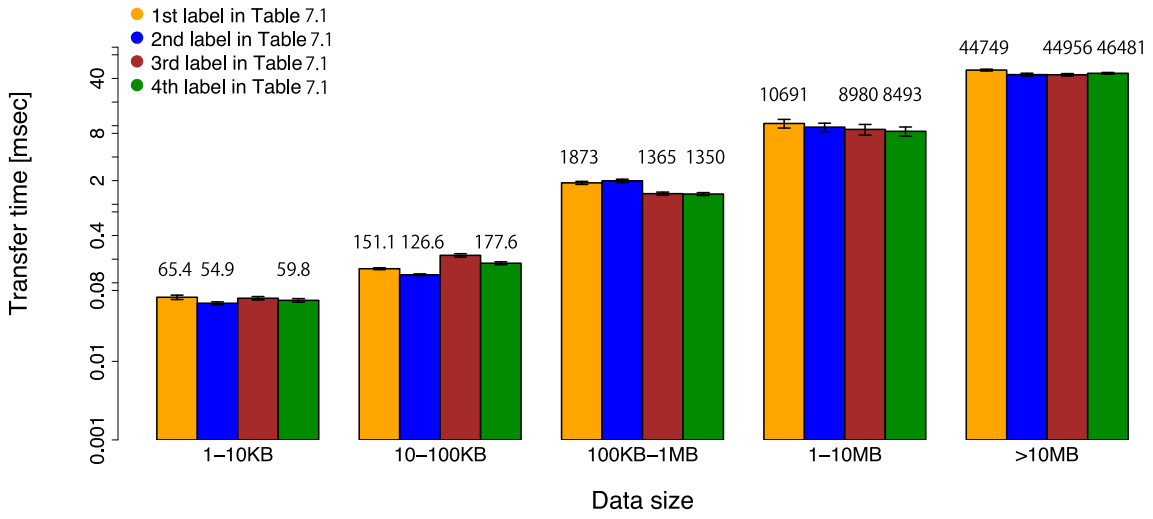

Figure 7.11: The result of the benchmark test for the ODCTCP sender evaluation. Using ODCTCP for short/long flows improves the data transfer time of short/long flows as expected. As a side effect, competitive standard ECN flows require longer data transfer time.

ODCTCP short flows complete data transfer time more quickly than standard ECN short flows, standard ECN long flows get the available bandwidth quickly when ODCTCP is used for short flows.

When ODCTCP short flows (the second label) are compared to DCTCP short flows (the fourth label), they improve data transfer time to 1.08 times in transmitting 1 - 10KB and to 1.4 times in transmitting 10 - 100KB. This is the influence of standard ECN long flows. Since standard ECN long flows reacts aggressively against the available bandwidth, ODCTCP short flows get more bandwidth than DCTCP short flows. In the result, in terms of transmission performance of short flows, ODCTCP short flows win. On the other hand, as expected, DCTCP long flows show shorter data transfer time than standard ECN long flows.

When ODCTCP is used for long flows (the third label), ODCTCP long flows fasten data

transfer time to 1.4 times in transmitting 100KB - 1MB, to 1.2 times in transmitting 1 - 10MB and to 1.1 times in transmitting more than 10MB compared to standard ECN long flows (the first label). This result is reasonable because standard ECN short flows refrain to utilize an available bandwidth although (O)DCTCP flows behave aggressively to the available bandwidth. As a side effect, the data transfer time of standard ECN short flows is larger than that of DCTCP short flows to 1.3 time in transmitting 10-100KB.

When compared to DCTCP long flows (the fourth label), ODCTCP long flows (the third label) rebalance transmission performance. ODCTCP long flows show 1.01 times as long data transfer time as DCTCP long flows for 100KB - 1MB transmission, 1.05 times as long data transfer time as DCTCP long flows for 1 - 10MB transmission and 1.04 times as long data transfer time as DCTCP long flows for more than 10MB transmission. For short flows, standard ECN short flows increase the data transfer time to 1.1 times in transmitting 1-10KB and to 1.3 time in transmitting 10-100KB.

7.3.3 Performance investigation for ODCTCP receivers

An ODCTCP receiver enables delayed ACK when the incoming packet sets the CWR flag. This modification avoids the situation which the standard ECN sender unnecessarily sets the small `CWND`. However, the probability that we can see the ODCTCP advantage at receivers is low because the situation is limited when the receiver sends an immediate ACK against a packet with the CWR flag. Therefore, in many experimental scenarios, the improvement of transmission performance would be very small. Note that this thesis abbreviates a flow that work with an ODCTCP receiver on one endpoint as an ODCTCP flow in this subsection.

Incast: Figure 7.12 shows the result of the incast test. When ODCTCP is used, ODCTCP flows show the same data transfer time with standard ECN flows. In addition, the observed `SRTT` values are same. This is the reasonable result because ODCTCP receiver has little

opportunity to use the algorithm.

Queue Buildup: Figure 7.13 shows the result of the queue buildup test when two long flows are running. When ODCTCP is used for short flows (the second label), ODCTCP

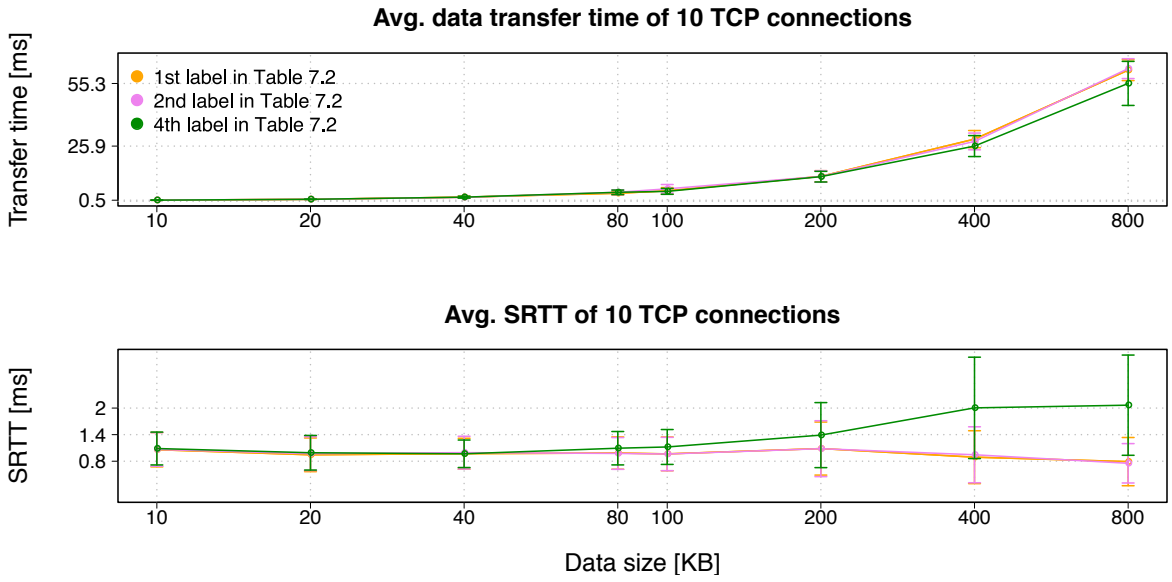

Figure 7.12: The result of the incast test. As expected, there is no difference between ODCTCP flows and standard ECN flows in terms of both data transfer time and `SRTT`.

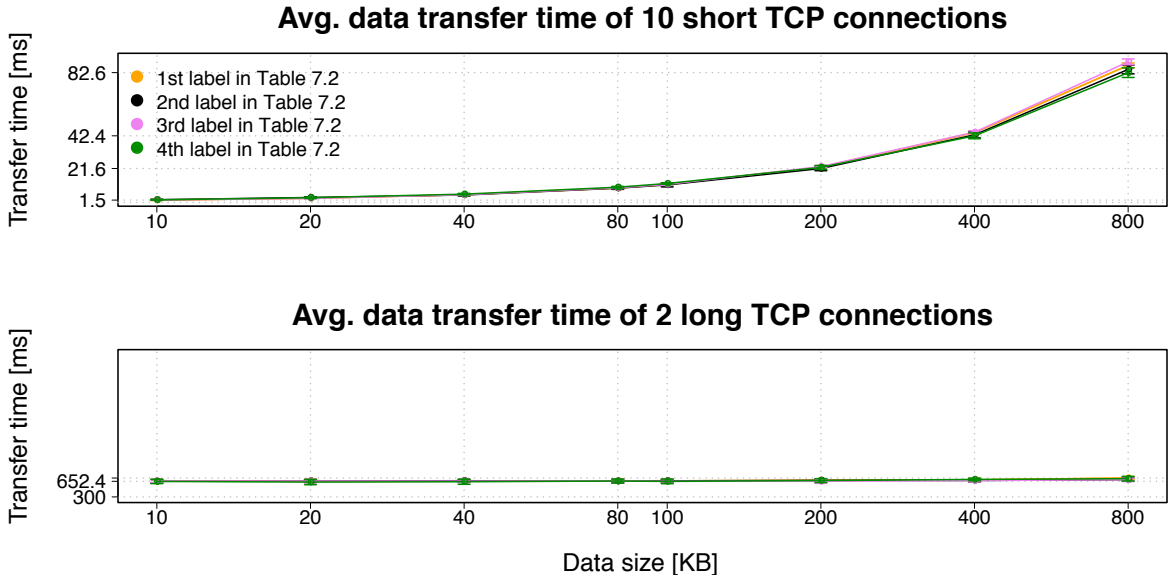

Figure 7.13: Queue buildup test result when two long flows are running. As expected, ODCTCP short flows show the same data transfer time with standard ECN short flows.

short flows show 2.4 milliseconds shorter data transfer time than standard ECN short flows (the first label) and 2.1 milliseconds longer data transfer time than DCTCP short flows (the fourth label) in transmitting 800KB. However, the difference corresponds less than 1% against the data transfer time so that it can be ignored. This is the reasonable result because ODCTCP receiver has little opportunity to use the algorithm.

When ODCTCP is used for long flows, standard ECN short flows show 2.1 milliseconds shorter data transfer time than standard ECN short flows (the first label) and show 6.9 milliseconds longer data transfer time than DCTCP short flows (the fourth label) in transmitting 800KB. However, the difference corresponds less than 1% against the data transfer time so that it can be ignored. This is also the reasonable result because ODCTCP receiver has little opportunity to use the algorithm.

When seeing the data transfer time of long flows, the difference between any two labels is less than 30 milliseconds, which is equivalent to 6% difference. The difference is not statically significant.

Figure 7.14 shows the result of the queue buildup test when the number of long flows is increased to eight. When ODCTCP is used for short flows (the second label), ODCTCP short flows show less than 1% of the difference of the data transfer time in transmitting less than 100KB compared to standard ECN short flows (the first label). However, the difference of data transfer time becomes long for large amount of data transmission. When transmitting 800KB, ODCTCP short flows show 1.2 times longer data transfer time than standard ECN short flows.

This is unexpected because ODCTCP receiver algorithm has eliminated a transmission performance impairment. A considerable cause is the temporal disabling of the delayed ACK algorithm in other situation. When the ODCTCP receiver gets the successive turns of a CE bit, it acks per packet. Then, the standard ECN sender transmits the smaller

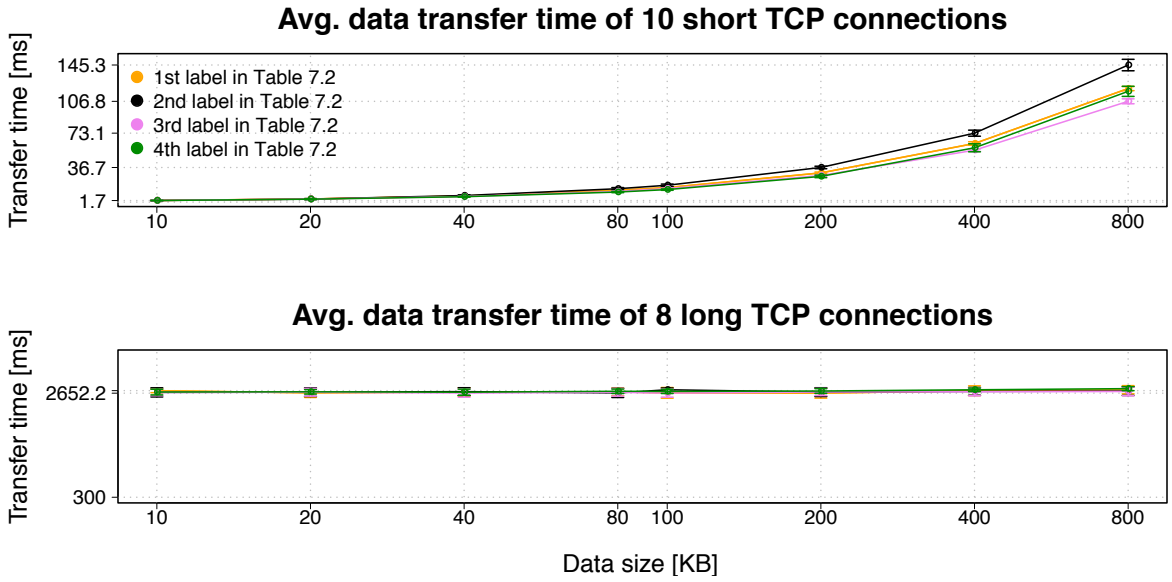

Figure 7.14: Queue buildup test result when eight long flows are running. Surprisingly, ODCTCP short flows require longer data transfer time than standard ECN short flows. It may be the negative influence of delayed ACK disabling on the ODCTCP receiver.

number of packets at once in that case. See the difference between Figure 5.7 and Figure 5.8. While the DCTCP receiver responses ACKs against the successive turns of a CE bit, the standard ECN sender transmits packets with fine-grained granularity although the total amount of transmitting packets does not differ. When the frequent disabling of the delayed ACK algorithm occurs, the sender gives a space to the available bandwidth for a short period for the competitive flows and standard ECN long flows may behave aggressively in the result. As the ground is not sure, this is one of future work to be investigated.

When ODCTCP is used for long flows, standard ECN short flows (the third label) improve the data transfer time to 1.1 times in transmitting 800KB compared to standard ECN short flows (the first label). Standard ECN short flows also shows 1.1 times shorter data transfer time than DCTCP flows (the fourth label). This is the inverse result when ODCTCP is used for short flows. Because ODCTCP long flows gives a space to the bandwidth by disabling delayed ACK, the competitive short flows may transfer data quickly.

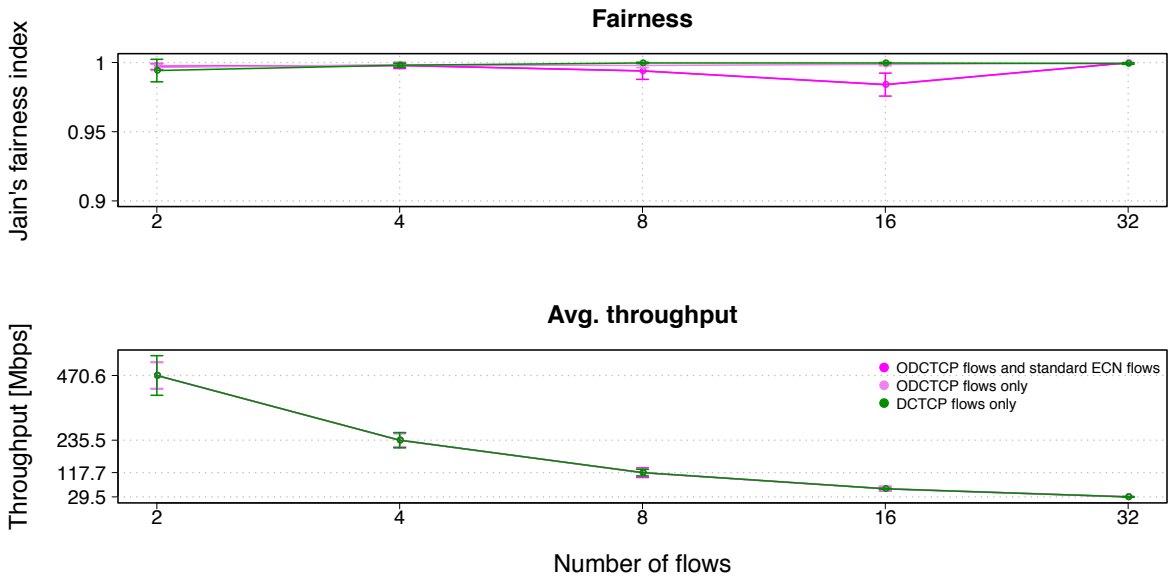

Figure 7.15: The result of the Fairness test. ODCTCP flows show good fairness as expected.

When seeing the data transfer time of long flows, less than 40 milliseconds difference is seen, which is equivalent to 2% of the difference. The difference is not statically significant.

Fairness: Figure 7.15 shows the result of the fairness test. When ODCTCP and standard ECN flows share a bottleneck link, the fairness indices show more than 0.98 regardless of the number of competitive flows. In addition, the average of throughput shows around 80% of the maximum bandwidth. Therefore, we can say that ODCTCP flows fairly share a bandwidth with standard ECN flows.

When a bottleneck link is shared among ODCTCP flows only, the fairness indices are more than 0.99 regardless of the number of competitive flows. In addition, the average of throughput shows around 80% of the maximum bandwidth. Therefore, we can say that the fairness between ODCTCP flows is also pretty good.

Benchmark: Figure 7.16 shows the benchmark result. Although a strange result is observed in the queue buildup test shown in Figure 7.14, the tendency has not seen in the benchmark test.

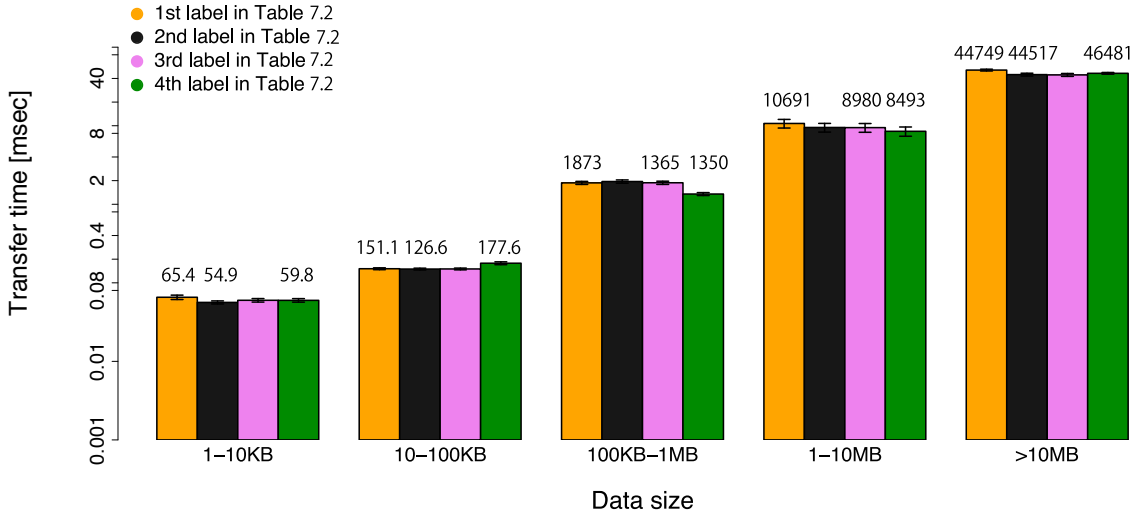

Figure 7.16: Benchmark result for the ODCTCP receiver evaluation. Using ODCTCP for short/long flows slightly improves the data transfer time of short/long flows and rebalances the transmission performance.

When ODCTCP is used for short flows, ODCTCP short flows show 1.2 times shorter data transfer time than standard ECN flows in transmitting 1 - 10KB and same data transfer time than standard ECN short flows in transmitting 10 - 100KB. In addition, ODCTCP short flows save 6% of shorter data transfer time than DCTCP flows in transmitting 1 - 10KB and data transfer time than DCTCP flows to 1.2 times in transmitting 10 - 100KB. However, ODCTCP short flows makes standard ECN long flows delayed as the side effect. Standard ECN long flows require 1.4 times longer data time than DCTCP long flows in the maximum. This result is expected because ODCTCP flows work better than standard ECN flows but worse than DCTCP flows. From this result, we can say that using ODCTCP for short flows improves the data transfer time and rebalances transmission performance in the network which deploys ODCTCP partially.

When ODCTCP is used for long flows, the data transfer time of ODCTCP flows show the same data transfer time with that of standard ECN flows in transmitting 100KB - 1MB, 1.1

times as small as that of standard ECN flows in transmitting 1 - 10MB and 1.2 times as small as that of standard ECN flows in transmitting more than 10MB. In addition, DCTCP long flows require longer data transfer time than standard ECN flows to 1.5 times in transmitting 100KB - 1MB, longer data transfer time than DCTCP flows to 1.1 times in transmitting 1 - 10MB and the almost same data transfer time than DCTCP flows in transmitting more than 10MB. Therefore, we can say that using ODCTCP receivers for long flows also rebalances transmission performance in the partially ODCTCP deployed network as expected.

Chapter 8

Conclusion and Future work

A data center network for soft real-time applications has three requirements: high burst tolerance, short latency for short flows and high throughput for long flows. To meet these requirements, SACK TCP is an inappropriate protocol because large FIFO buffers in the presence of bulk transfers impacts latency-sensitive flows. DCTCP is an advanced TCP which leverages ECN for high burst tolerance. However, a downside remains in the incremental deployment path of DCTCP. Due to the DCTCP compatibility issues, using DCTCP in the partial servers or flows degrades transmission performance.

This thesis proposes ODCTCP, which allows us to use DCTCP on one endpoint in the network where servers and switches use standard ECN. ODCTCP modifies two things in the DCTCP algorithm: one applies in the DCTCP sender algorithm to support a compatibility with standard ECN and the other applies in the DCTCP receiver algorithm to eliminate the possibility which the peer underestimates window size.

The simple experiment results show that the partial ODCTCP deployment eliminates performance impairments and improves data transmission performance in the network in comparison with the network where all servers use standard ECN only. When the ODCTCP sender algorithm is used for short flows only, ODCTCP short flows show up to 1.2 times as short data transfer time as standard ECN short flows in the traffic mixed short and long

flows. When the ODCTCP receiver algorithm is used for short flows only, ODCTCP short flows show up to 1.2 times as short data transfer time as standard ECN short flows in the traffic mixed short and long flows.

The DCTCP implementation described in this thesis will be merged into the main stream of the FreeBSD kernel in near future. Therefore, All FreeBSD users can easily deploy ODCTCP in the data center network. This activity will also provide a positive impact to IT enterprise. This thesis remains future work which relates with the DCTCP implementation choices. One is the DCTCP behavior when the receiver buffer is tuned by applications. In the real world, many applications control data transmission rate by tuning the receiver buffer. Neither ODCTCP nor DCTCP considers the usefulness of this condition. They assume all the applications controls congestion in the transport layer. The other is α update interval at the DCTCP sender. Although DCTCP paper defines an α update interval as every `CWND`, why this interval is proper is big question. Instead of it, for example, we can update α every `ACK`. Using frequent α update grabs a prompt congestion change. In addition, when a RTT is large, the frequency of an α update interval helps to identify the current congestion. The investigation of the transmission impairments for the two must be considered in the future.

Bibliography

[1] Dctcp implementation for linux. `http://simula.stanford.edu/~alizade/Site/DCTCP.html`.

[2] Siftr (statistical information for tcp research). http://caia.swin.edu.au/urp/newtcp/tools/siftr-readme-1.2.3.txt.

[3] Tcpdump project. http://www.tcpdump.org/.

[4] Kuzmanovic Aleksandar, Mondal Amit, Floyd Sally, and K. Ramakrishnan K. Rfc5562: Adding explicit congestion notification capability to tcp's syn/ack packets., June 2009. http://tools.ietf.org/html/rfc5562.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[6] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of dctcp: Stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, pages 73–84, New York, NY, USA, 2011. ACM.

[7] Mark Allman. Tcp congestion control with appropriate byte counting (abc).

[8] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. rfc 5681 (draft standard), 2009.

[9] Grenville Armitage, David Hayes, and Lawrence Stewart. Five new tcp congestion control algorithms for freebsd. http://caia.swin.edu.au/freebsd/5cc/.

[10] Bob Braden, David Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. Ramakrishnan, Scott Shenker, John Wroclawski, and Lixia Zhang. Recommendations on queue management and congestion avoidance in the internet. rfc 2309 (informational), April 1998.

[11] Robert Braden. Requirements for internet hosts – communication layers. rfc1122 (internet standard).

[12] Bob Briscoe, Arnaud Jacquet, Toby Moncaster, and Alan Smith. Re-ecn: Framework. (draft standard), October 2010. http://tools.ietf.org/html/draft-briscoe-tsvwg-re-ecn-tcp-motivation-02.

[13] Jerry Chu, Nandita Dukkipati, Yuchung Cheng, and Matt Mathis. Increasing tcp's initial window. rfc 6928 (experimental), April 2013. http://tools.ietf.org/html/rfc6928.

[14] Nandita Dukkipati, Neal Cardwell, Yuchung Cheng, and Matt Mathis. Tail loss probe (tlp): An algorithm for fast recovery of tail losses (draft standard), February 2013. http://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[15] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-Shen, and Nick McKeown. Processor sharing flows in the internet. In *Proceedings of the 13th International Conference on Quality of Service*, IWQoS'05, pages 271–285, Berlin, Heidelberg, 2005.

[16] Bill Fink, Rob Scott, Mike Muuss, and Phil Dykstra. Nuttcp, 2009.

[17] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, August 1993.

[18] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Matthew Podolsky. Rfc 2883: An extension to the selective acknowledgement (sack) option for tcp, July 2000. http://www.ietf.org/rfc/rfc2883.txt.

[19] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. The newreno modification to tcp's fast recovery algorithm. rfc 6582 (proposed standard), April 2012.

[20] Todd Hoff. Latency is everywhere and it costs you sales - how to crush it.

[21] Rajendra Jain, Dah-Ming Chiu, and William Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *DEC Research Report*, TR-301, September 1984.

[22] Rick Jones. Netperf, 2009. http://www.netperf.org/netperf/.

[23] K Ramakrishnan, Sally Floyd, and David Black. The addition of explicit congestion notification (ecn) to ip. rfc 3168 (proposed standard), September 2001. http://www.ietf.org/rfc/rfc3168.txt.

[24] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27:31–41, 1997.

[25] Richard Scheffenegger and Mirja Kuehlewind. More accurate ecn feedback in tcp (draft standard). http://tools.ietf.org/html/draft-kuehlewind-tcpm-accurate-ecn-02.

[26] Stanislav Shalunov. thrulay, network capacity tester, 2009.

[27] Lawrence Stewart and James Healy. Light-weight modular tcp congestion control for freebsd 7. December 2007.

[28] Lawrence Stewart and James Healy. Tuning and testing the freebsd 6 tcp stack, July 2007.

[29] Iperf team. Iperf, 2009. http://sourceforge.net/projects/iperf/.

[30] Joe Touch. Tcp control block interdependence, April 1997.

[31] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 115–126, New York, NY, USA, 2012. ACM.

[32] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 50–61, New York, NY, USA, 2011. ACM.

[33] Yong Xia, Lakshminarayanan Subramanian, Ion Stoica, and Shivkumar Kalyanaraman. One more bit is enough. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '05, pages 37–48, New York, NY, USA, 2005. ACM.

[34] Alexander Zimmermann, Arnd Hannemann, and Tim Kosse. Flowgrind - a new performance measurement tool. In *GLOBECOM*, pages 1–6. IEEE, December 2010.