Title Congestion Control in Datacenter Based on Both RTT and ECN

Student Name Liang Junpeng

Student No. 2014050690

Major Computer Science and Technology

Supervisor Cui Lin

Date(dd/mm/yyyy) 18/05/2018

暨 南 大 学

本科生毕业论文

论文题目 基于往返时延及显式拥塞标记的数据中心拥塞控制算法

学　　院 国际学院

学　　系

专　　业 计算机科学与技术

姓　　名 梁俊鹏

学　　号 2014050690

指导教师 崔林

2008 年 **5** 月 **18** 日

Statement of Originality

I hereby declare that the thesis presented is the result of research performed by me personally, under guidance from my supervisor. This thesis does not contain any content (other than those cited with references) that has been previously published or written by others, nor does it contain any material previously presented to other educational institutions for degree or certificate purpose to the best of my knowledge. I promise that all facts presented in this thesis are true and creditable.

Signed: _____ Date：18/05/2018

Congestion Control in Datacenter Based on Both RTT and ECN

Abstract: Cloud computing is thriving, and the amount of data is also growing rapidly. Datacenter, an important base of cloud computing, should be more aggressive in providing higher throughput, lower latency and low flow completion time that cater to application demand. Traditional TCP fails to satisfy these requirements, so plenty of congestion control algorithms are issued to complement the flaws of traditional TCP in datacenter network. Most previous works take ECN and RTT independently as congestion indicator. However, ECN is not good at handling packet burst while RTT is not precise enough on controlling congestion window. This thesis uses both ECN and RTT to plot the congestion level and modifies senders and receivers. T-DCTCP, a congestion control algorithm based on both RTT and ECN, is designed, and it aims at coexisting with traditional TCP flow. Therefore T-DCTCP utilizes congestion window to control sending rate. An experiment with 16 hosts in Fat-tree topology is carried on Mininet. The experiment result shows that T-DCTCP achieves 12% lower average RTT, 30% lower tail latency and lower small flow completion time comparing to DCTCP. The small flow completion times of different flow sizes vary and the decrease of 2KB flow is the most which is about 1.4X lower comparing to DCTCP.

Key Words: ECN, RTT, Congestion control, Datacenter network, TCP

基于往返时延及显式拥塞标记的数据中心拥塞控制算法

摘　要: 云计算正蓬勃地发展中，越来越多的数据被生成了。数据中心作为云计算的重要基础设施，它应该在计算吞吐量、延迟和数据流完成时间方面有更突出的表现以满足计算需求。传统的传输层控制协议无法满足这些需求，所以很多种拥塞控制算法被提出来解决传统传输层控制协议与数据中心的不兼容问题。这些新提出的拥塞控制算法分别单独地使用显式拥塞标记和往返时延作为拥塞程度指示信号。比如显式拥塞标记不善于处理包爆发传输情况，而在控制拥塞窗口方面往返时延不能够提供精准的控制。本文将显式拥塞标记和往返时延同时应用于描述拥塞情况，并在发送端和接收端进行修改。T-DCTCP 是一个基于往返时延及显式拥塞标记的控制算法，并且能够与传统传输层控制协议的数据流相兼容。为了兼容传统的数据流，T-DCTCP 通过控制拥塞窗口来间接地控制发送速率。我们在 Mininet 上做了一个网络实验，其中有十六个虚拟主机在一个 Fat-tree 的网络拓扑。实验结果表明 T-DCTCP 提供低于 DCTCP 9.4%的平均往返时延以及小于 DCTCP 37.7%的尾延迟，还有在小数据流的完成时间方面 T-DCTCP 比 DCTCP 要小。每种不同大小的小数据流的完成时间都有不同降幅，降幅最大的是 2KB 数据流，比 DCTCP 小了约 1.4 倍。

关键词:显式拥塞标记；往返时延；拥塞控制；数据中心网络；传输层控制协议

Contents

1. Introduction

In recent years, with the robust development in IT, more and more different access points to Internet appear, so people do not have to access Internet via PC and they can use smart phones or tablets to go online, among whom average customer has 3.64 devices (Tellas, 2017) for accessing Internet. Due to these changes, about 2.5 quintillion bytes of data are created per year according to IBM (Kleinberg, 2013), beneath which plenty of valuable information is concealed, so service providers would like to analyze data. In addition, lots of applications which are based on cloud computing turn up, and they are requiring more and more computation resources; and on the other hand, it will be bad if these applications suffer from high delay in response. As a result, data center should provide high throughput and low latency computation service.

However, traditional TCP fails to satisfy such a demand (Alizadeh et al., 2010), and it suffers from the problems of incast issue and outcast issue, which cause severe decrement in throughput and sharp increment in latency. Actually, these problems are also related to the consistent scheme of deploying infrastructures in data center, which is adopted to build a high computing performance data center using low cost commodity components (Alizadeh et al., 2010). Low cost switches are common at the top of rack, and they have a common drawback that is providing a shallow and shared buffer. It is not realistic to replace the current switches for every rack of servers with the ones which offer huge buffer space and independent buffer for different ports. Another source of these problems are caused by incompatibility between computation model and traditional TCP (Alizadeh et al., 2010), which will be illustrated in Section 2. Therefore, we need to improve TCP congestion control algorithm in order to conquer the challenges that data center is facing.

There are varieties of congestion control algorithms for datacenter published (Sreekumari et al., 2015). Some algorithms are delay-based, some are ECN-based, and some are deadline-aware, but two of them attract us. The first one is DCTCP (Alizadeh et al., 2010), and it utilizes Explicit Congestion Notification (Floyd, 1994) to reveal the congestion level of the network. Every switch has a threshold K on queue length to set Congestion Encountered signal, and the congestion severity will be reported to source so that sources can react to congestion based on its severity. The second one is TIMELY, and it uses package delay as a metric of congestion level. TIMELY depends on precise latency detection mechanism with accuracy up to micro

second (Mittal et al., 2015), and it control the sending rate directly instead of congestion window. TIMELY measures precise RTT with the support of network interface card. To use RTT as a metric of congestion level, it is necessary to have a RTT measurement with high accuracy.

ECN is a single bit notification while RTT is a multi-bit notification which enables RTT to perform better than ECN in handling bursty link and describing network status, and we will give examples in detail in Section 2. However, some problems of RTT should be considered as well, such as precision on modifying congestion window and whether measurement is accurate enough. We are motivated to design a novel congestion control combining ECN and RTT to complement the flaws of each one.

So our scheme of congestion control algorithm compatible with traditional TCP flow comes. It will judge congestion level with the *alpha* value implied from ECN. T-DCTCP uses RTT to indicate whether the flow is suffering from high delay and the modification on congestion window. When it needs to decrease the congestion window, T-DCTCP modifies congestion window according to ECN and RTT. What's more, we obtain RTT from Linux kernel, because Linux improves its accuracy of RTT measurement and the accuracy can be up to microsecond which can satisfy the demand of T-DCTCP. T-DCTCP mainly focuses on the problems ECN brings, and it should be able to revise some misjudgments ECN makes in order to improve latency performance. The goal to achieve is to reduce the latency of DCTCP with increased or consistent throughput. The experiment carried on Mininet with Fat-tree topology shows us that comparing to DCTCP, T-DCTCP provides 9.4% lower average RTT and 37.7% lower tail latency. In addition, it shows that FCT of small flows of T-DCTCP is less than that of DCTCP. This experiment result roughly matches up with our expectation.

In Section 2, we will introduce the background in details and define the motivation of T-DCTCP; in Section 3, we will describe T-DCTCP algorithm in detail; and in Section 4, the environment and evaluation of experiment will be shown. Discussion and Conclusion will be presented in Section 5 and Section 6. So let us begin with the background in of this thesis and our motivation.

2. Background & Motivation

In this section, we mainly discuss the problems that may encounter in data center network and

the problems of using ECN and RTT independently. To get better understanding of these problems, it is necessary to illustrate the whole view of how computation works in datacenter and the critical influence of latency. We will identify the performance impairment issues caused by these problems. At the end of this section the motivation of T-DCTCP will be described.

2.1 TCP Basics in Datacenter

Data center network is different from wide area network. It usually provides network links with high bandwidth and low packet loss rate, so latency in data center network is very low and is usually within microsecond. TCP flows transmitting in data center vary from their lengths and priorities. The result of applying TCP congestion control algorithm to data center network reveals that TCP congestion control scheme is not compatible with data center network and actually bring severe performance impairment to it. In the following part, these performance impairment issues will be illustrated in detail. In order to illustrate the performance impairment issues well, we should begin with computation model in data center network, which is the source of TCP incompatible issues.

2.1.1 Partition/Aggregate

To our understanding, computation work is simple for electronic devices, and they can run fast enough to deal with all kinds of work flows. However, as raw data increases, it becomes not as straightforward as we think, and the input data is usually large and the servers cannot handle such a heavy computation job by themselves. Therefore, great engineers figure out a novel solution: partition/aggregate (Alizadeh et al., 2010), which is a use of divide and conquer. This means the computations will be distributed across hundreds or thousands of machines in order to finish in a reasonable scope of time. After workers finish their work, they will send the results back to the aggregator, and the aggregator should collect these result and aggregate them to generate a new result to send back to the back-end device or the upper layer aggregator.

Figure 2-1 illustrates the Partition/Aggregate pattern in detail, and this figure is quoted from the paper of DCTCP. As is shown in figure 2-1, a computation request is sent to a server, and then this server becomes a main aggregator in charge of dividing the computation task into pieces. After division job is done, all of the tasks will be sent to next level; and if the raw task is too large to be computed by a rack of server, the next level of servers will act as aggregators as well and assign subtasks to the workers in other rack. Each subtask has its deadline to meet,

before when it should be finished and the result needs to be sent back to the aggregator. If one of subtasks misses its deadline, the aggregator has to aggregate the results collected from workers and produces result back to backend device without waiting for its arrival, which will affect the quality of the final result produced by the aggregator. More and more subtasks miss their deadline, less and less accurate the result is (Alizadeh et al., 2010).

In effect, most of the web online real time applications are driven by the computation support from datacenter side, which are highly related to response time. Even a short delay will have bad influence on users' experience of the real time application. In addition, users also care about whether the application can provide a precise and correct answer or result after a period of waiting. In one sentence, Partition/Aggregate pattern cannot tolerate the situation of deadline missing. For example, when you ask Siri on iPhone to operate a command, Siri will first send an audio request to Apple Severs, and the audio request might be partitioned into dozens of pieces to be parsed by Natural Language Process module in different workers and aggregators (Pierce, 2017). Based on the replies from lower level of workers, the aggregator refines valuable information and send it back to Siri, and then Siri will operate as is told by Apple Server side. You may remember how inaccurate and stupid Siri was when its first version was issued in 2011, and Siri was mocked as a little toy published by Apple which can be only used for entertainment. However, cloud computing was rising rapidly at that time, and DCTCP was published in 2011 to improve the throughput and latency performance of Partition/Aggregate pattern. The following versions of Siri becomes smarter and smarter comparing to its first version. A good congestion control designed for data center is one of the important reasons, because it makes it possible for more and more subtasks to meet their deadlines.

Average latency is the most concerning thing, but 99th percentile latency may be the most important thing related to the quality of result (Mittal et al., 2015). As is mentioned above, the number of work flows which successfully meet the deadline determines the quality of result extracted from the replies from workers (Vamanan et al., 2012). 99th percentile latency, also called tail latency, describes the latency of the last replies of work flows. In other word, if 99th percentile latency is less than deadline, the result refined from the work flow replies will be accurate, because it means almost all of the packets do not exceed their deadline.

It seems that packet delay plays an important role in data center network performance, and

it will be great if we can figure out an approach to control it within a tolerable range. This requires the congestion control algorithm to be concern about not only average latency but also tail latency. Actually, there is a model called Map/Reduce (Dean et al., 2008), which can also enable data center to provide a more precise result. Map/Reduce is a programming model similar to Partition/Aggregate, and Map/Reduce is improved by reducing response time (Venkatesh et al., 2018). So what needs to be concerned is latency, and it is proven that there are lots of benefits accompanied by low tail latency.

Partition/Aggregate computation model is described in the previous context and in the following parts the performance impairment issues due to the incompatibility between TCP and Partition/Aggregate computation model in data center will be introduced.

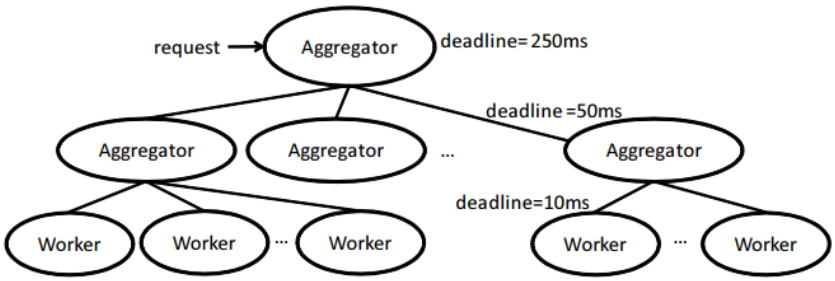

Figure 2-1 The Partition/Aggregate computation model (Alizadeh et al., 2010)

2.1.2 TCP Incast

TCP incast is one of the crucial performance collapse issue in data center network. Unfortunately, it is a common phenomenon in data center network. Network throughput will drastically decrease when incast issue shows up (Haitao et al, 2013). It is the side effect of Partition/Aggregate model (Alizadeh et al., 2010), and the most probable situation when incast issue appear is that workers simultaneously send replies back to their shared aggregator on the upper layer (Qin et al., 2016). The switch on the top of the rack receives packets from different ports and puts them to the queue of a single port which is linked to the aggregator (Haitao et al., 2013). As a result, the queue length will increase multiplicatively and the switch buffer will drain out quickly, so the packets at the end of the queue may be dropped or suffering from high delay (Goel, 2017). As packets are dropped, TCP retransmission mechanism will be on its service and retransmit the packets that have been dropped by the switch (Sreekumari et al., 2015). However, this will not help solve incast problem, instead it will do heavy harm to the network throughput and aggravate the congestion severity. Because new coming packets that

are retransmitted by senders cannot reduce the queue length of the port to receiver, and on the other hand, they compete with the old packets waiting in the queue. TCP incast issue is plotted in Figure 2-2.

In addition, receiver will return a small synchronized data requests using a large logical block size to multiple senders in order to assure the quality of replies. Unfortunately, these requests is sent to the port of switch which is in heavy congestion. At this time an unpredictable performance reduction happens, and the network throughput should experience severe collision.

Actually packet timeout not only increases congestion severity in switch, but also increases the end-to-end latency. After packet loss, sender should wait for a relatively period of time to retransmit the lost packet, but retransmission timeout (RTO) set by TCP is mainly for wide area network which has much larger latency than that data center network has. However, due to the design of TCP, RTO is an important metric for packet transmission. When incast issue occurs, packet loss is unavoidable. Some application developers try to decrease RTO in order to reduce the impact of timeouts, but this proposal cannot cater to other problems caused by the incompatibility between TCP congestion control algorithm and data center network not to mention the difficulty of finding a suitable value of RTO which is suitable for low latency network and normal retransmission frequency at the same time (Alizadeh et al., 2010).

As is shown above, TCP cannot adapt itself to this many-to-one pattern, Partition/Aggregate, and its reaction to congestion is not sensitive enough to handle this issue (Wang et al., 2018). TCP suffers from late congestion notifying issue and the timeout problems caused by packet loss. In summary, TCP incast issue is based on three principal preconditions (Sreekumari et al., 2015) as listed below:

- High bandwidth and low latency network
- Shallow-buffer and share-buffer switch installed
- Partition/Aggregate pattern

After the above analysis, TCP incast issue does degrade the performance of data center network heavily, and it will be the main issue to be solved or be optimized while it is concerned with network throughput and latency.

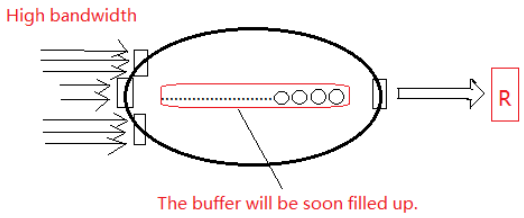

Figure 2-2 TCP incast issue

2.1.3 TCP Outcast

TCP Outcast problem, also called Queue Buildup, is another important issue affecting the latency in data center network. When a set of short flows share the same switch with another set of long flows, the set of short flows always suffer from low throughput issue with TCP congestion control algorithm, while the set of long flows is not affected too much in this situation (Qin et al., 2016). This unfairness will also increase the latency of short flows, which is definitely a great harm to short flows (Sreekumari et al., 2015). In addition, in data center network, short flow is usually a time-critical flow (Alizadeh et al., 2010), and even a little increase in latency can degrade the quality of computation.

One of the main reasons of the TCP outcast problem is 'Port Blackout'. This problem is in fact stemmed from shared-buffer switch. That is, a series of packets from different input ports compete for a single output port. As is mentioned in the previous subsection, the packets which are at the end of queue suffer from high latency and when the queue is close to maximum value (Qin et al., 2016), some packets will be dropped and retransmission is needed to start. The previous one is TCP Outcast problem (Queue Buildup) that is going to be discussed, and the latter one is TCP incast issue mentioned above.

Figure 2-3 show us a simple example of queue buildup problem which happen in a switch. There are two flows arriving at the switch one after another, and flow A is enqueued right before flow B. As the figure reveals, flow A contains more packets than flow B. However, the sending rate of Port C heading to the receiver is limited, which means it takes time to forward the packets within flow A. As a result, flow B has to wait until all of the packets in flow A are forwarded. Consequently, the round trip time of flow B should contain the forwarding time of flow A. Although data center is deployed with high bandwidth and forwarding delay is very low even for a long flow, TCP congestion control is not good at controlling the queue length of switch,

which can bring trouble to the short flow at the end of the queue because it has to wait until numbers of long flows finish forwarding. That will be a disaster for a time-critical data flow when it encounters such a long delay. What's more, ACK packet should be the first packet forwarded at its arrival on the queue, but due to its tiny packet size, its forwarding time can be ignored and will not generate negative influence on flow latency.

In effect, short flows are the most affected ones in TCP outcast problem. It can degrade the throughput of short flows sharply, and can even decrease the accuracy of outcomes. Short flow is important in data center, and according to the paper of DCTCP, Queue Traffic is usually as the form of short flow which serves as partition message between high level aggregator and midlevel aggregator (Alizadeh et al., 2010). Therefore, outcast problem is also caused by a feature of Partition/Aggregate pattern, and it is clear that some changes should be made to TCP congestion control mechanism.

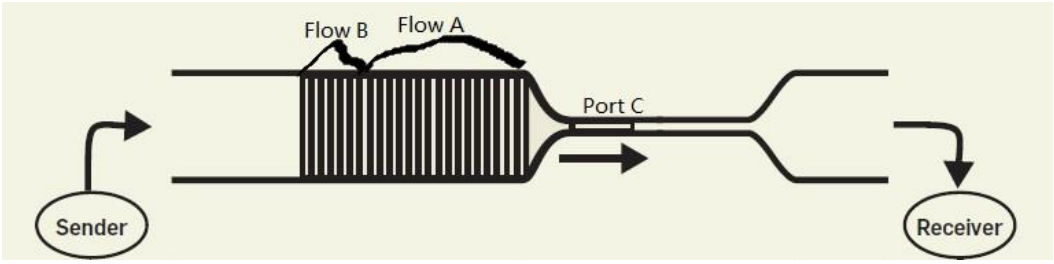

Figure 2-3 TCP Outcast

2.2 Existing Congestion Control Algorithms in Datacenter

In effect, there are lots of congestion control algorithm using ECN or RTT as congestion level indicator issued, such as DCTCP, DT-DCTCP, L^2DCT, TCP BOLT, MTCP, D^2TCP, TIMELY, TCP-Illinois, EAR, and so on. Among these congestion control algorithms, DCTCP, L^2DCT, TCP BOLT, MTCP are able to reduce flow completion time (Sreekumari et al., 2015). TIMELY are proven that it can reduce latency greatly comparing to DCTCP (Mittal et al., 2015). We will talk about congestion control based on different congestion signal with examples.

2.2.1 Congestion Control Based on ECN

The most famous congestion control algorithm using ECN to judge congestion severity is DCTCP. To use DCTCP, ECN marking threshold *K* should be set in switches. When the sequence number of the packet in queue exceeds *K*, the packet will be marked with an ECN flag. The sender needs to count the number of marked packets among the acked packets. The

fraction of marked packets will be utilized to calculate new congestion window. There is also a congestion control algorithm called DT-DCTCP, which is an improved version of DCTCP. It sets double marking thresholds in the switch to have a better illustration on congestion status (Chen et al., 2013).

In addition, MTCP (Fang et al., 2012), TCP BOLT (Stephens et al., 2014) and L^2DCT are also based on ECN. L^2DCT takes flow size into consideration (Munir et al., 2013), and it gives higher bandwidth to short flows than long flows. These congestion control algorithm based on ECN aim at improving flow completion time. Actually they do reduce the flow completion time in datacenter significantly. They reveal that ECN is indeed a good congestion signal.

2.2.2 Congestion Control Based on RTT

Due to inaccurate RTT measurement mechanism, RTT was not considered as a good congestion indicator until TIMELY was proposed. TIMELY utilizes hardware support to measure RTT with high accuracy and it controls sending rate directly to control congestion. RTT is used to judge network congestion status and new sending rate is implied from RTT difference between the current one and the previous one. TIMELY controls sending rate via a mechanism called RDMA. In effect, TIMELY reduces latency tremendously comparing to DCTCP and it reduces tail latency by 13X (Mittal et al., 2015). TIMELY shows that RTT is also capable of revealing congestion status.

2.2.3 Congestion Control Based on Other Congestion Signal

D^2TCP is a congestion control algorithm which differs from DCTCP and TIMELY. It utilizes deadline as a metric to control the forwarding priority of data flows in datacenter. The flows in queue that are near their deadlines will be forwarded first by the switch (Vamanan et al., 2012). It does decrease the number of flows that miss their deadline comparing to DCTCP. However, sometimes short flows may suffer from high latency when a long flow near its deadline is in the same queue.

What's more, EAR is also a congestion control algorithm which combines both ECN and RTT as congestion signal, but the way that it applies ECN and RTT to congestion control differs from T-DCTCP. EAR defines RTT as a threshold like ECN marking threshold *K* and the packets exceeds these two thresholds will be marked as congested (Zeng et al., 2017). Then EAR will decrease congestion window according to the fraction of the marked packets. However, T-

DCTCP defines RTT as a congestion level indicator which offer assistances to ECN to judge the congestion level and RTT also needs the assistance from ECN to control congestion window.

There are still lots of congestion control algorithms utilizing other congestion indicators. However some of them have obvious flaws, so they are not in actual deployment. We will not discuss them here.

2.3 Discussion on RTT & ECN

2.3.1 The Limitation of ECN & RTT

Two main sources causing performance degradation have been described in previous context, and DCTCP was proposed to repair these performance impairments. It is well known that DCTCP uses ECN to retrieve network status and react to upcoming congestion situation through the fraction of packets that are marked with ECN-Echo by receiver. It is informed that ECN can help plot a whole picture of how network traffic is going and the information it reveals is valuable (Black, 2018) so that DCTCP can control congestion well and bring good performance to data center network. However, we should notice that ECN is single-bit signal (Salim et al., 2000; Saeed, 2016), which means the information within ECN is limited. For example, ECN cannot reflect whether the flow is experiencing high delay or queue buildup problem, because only the packets exceeding the threshold in the queue of the switch will be marked with ECN.

The limitation of ECN can lead to bad judgment on network status. When a long flow and a short flow is enqueued one after another, it is possible that most of the packets in short flow are marked while most of the packets in long flow are not marked. Then the sender of long flow is not informed about there is a queue buildup issue happening in the switch, but the sender of short flow is informed that there is a queue buildup issue so it reduces its sending rate. At this situation, it will be unfair for the sender of short flow (Ito et al., 2018). Although, the network can keep high throughput in sum, but unfairness still exists.

RTT cannot provide precise control on congestion window when the latency gets high, because even if RTT is normalized it is still a decimal value while congestion window is counted in packet. The reason why RTT can provide precise control on sending rate (Mittal et al., 2015) is that sending rate is also a decimal value; but congestion window can only be an integer, and consequently the control precision of RTT will be reduced. However, when T-DCTCP is required to be compatible with TCP, using congestion window to control sending rate is a must.

2.3.2 The Reason of Choosing Both RTT and ECN

Analysis above reveals the limitation of ECN and RTT, but they can still provide useful information about network status. In order to utilize the valuable information from ECN and RTT, it is considerable to overcome their flaws. Introducing RTT to ECN is a possible solution to this problem. Actually we do have several reasons on choosing RTT as a congestion control signal for correcting possible misjudgment made by ECN, and ECN can help RTT improve the precision on congestion window control. Some features of RTT attract us, and it is reasonable to believe that RTT can have great influence on plotting network status in more details.

RTT reveals packet latency (Mittal et al., 2015). RTT is the abbreviation of round trip time, which represents time consumption from sending a packet to receiving the corresponding ACK. It is the end-to-end latency inflated by network queuing, which means it can convey queue status to us. Even if each flow has different priorities in forwarding sequence, RTT can reflect it and provide helpful information on resolving the congestion issue.

RTT contains multi-bit information (Mittal et al., 2015). After subtracting known propagation and serialization delays from a measured RTT, network queuing delay can be revealed. However, it is not possible to reveal network queuing delay with ECN fraction, or to say that there is no cohesive relationship between network queuing delay and ECN fraction. There is much richer and faster information about the state of network switches hiding behind RTT. RTT can convey multiple bits and collect the status of end to end queuing delay across multiple switches. When packet burst occurs, RTT can tell if the current flow is at the end of queue or is a large flow that may cause queue buildup problem to other flow. When current flow is a large flow, the delay of last packet in this flow will be much larger than the first packet in this flow, and control algorithm should take some measures to avoid queue buildup issue if the average RTT keeps increasing. If the queue is experiencing queue buildup issue, RTT and ECN fraction can plot the network status together in detail.

With these two features mentioned above, RTT contains some information that ECN cannot bring like whether the current flow is experiencing high latency. However, we should fix the inaccurate RTT measurement issue before using RTT as a congestion indicator. Fortunately, recently RTT measurement gets more accurate (Mittal et al., 2015). It has been mentioned that data center network is different from wide area network. With high bandwidth network, package

delay in data center is always counted with microsecond in measurement unit. The issue of accuracy keeps concerning congestion control algorithm designers. Fortunately, recent Linux kernel provide smooth RTT measurement mechanism to solve this problem and can accurately record the time of packet transmission and reception with accuracy up to microsecond. On the other hand, recent network interface cards also offer hardware support to measure RTT in microsecond. The development of software and hardware makes it possible to take precise RTT measurements to accurately track end-to-end network queues. We have observe the RTT values measured by Linux kernel and find they are measured in the metric of microsecond. What's more, in lossless tunnel the measured RTT does not vibrate sharply.

ECN has been proven that it can have excellent control on congestion window. The information contained in RTT can complement the shortcomings of ECN in plotting network congestion status. Therefore we choose both ECN and RTT as congestion indicators.

3. T-DCTCP Design

This algorithm aims at utilizing the information provided by both ECN and RTT. T-DCTCP consists of three parts: retrieving RTT and the mechanism of sending ACKs, T-DCTCP algorithm, and the approaches to control sending rate. Let us begin with the first part.

3.1 Retrieving RTT & the Mechanism of Sending ACKs

There are two different ways to retrieve RTT. RTT can be retrieved from Linux kernel with the accuracy up to microsecond, which caters to the need of T-DCTCP. Linux kernel provides an interface for us to get RTT value. To implement a congestion control algorithm, we have to follow the instruction of a structure called "tcp_congestion_ops", in which there is a function pointer called "pkts_acked". After reading the source code of "tcp_congestion_ops", we can see there are three parameters transferred to "pkts_acked" and the last one is "rtt_us". This function means when packets are ACKed, the round trip time will be transmitted to this function and we can retrieve RTT in microsecond by implementing this function.

The method of retrieving RTT above is provided by Linux kernel, and there also exists another method to measure RTT accurately which is proposed recently, but it requires supports from network interface cards. With the breakthrough in NIC, some latest modern NICs can

count delay in timestamp and generate ACK fast, which enable us to implement a round trip time detection mechanism. With the hardware support of NICs, it is possible to record serialization time of sender and sending time of NIC so that some tiny variation can be eliminated and the calculated RTT can clearly represent the status of network.

In effect, we adopt the previous proposal to measure RTT. However, some changes on sending ACKs are needed in order to enable ECN marks.

To accomplish this purpose, the function called “cwnd_event” in “tcp_congestion_ops” should be changed. The only difference between DCTCP receiver and a TCP receiver is the way that information in the CE codepoints is conveyed back to the sender (Alizadeh et al., 2010). The slightly changes are to ACK every packet and to set the ECN-Echo flag if and only if the packet has a marked CE codepoint. However, there exists a mechanism called Delayed ACKs in TCP, which can reduce the load on data links. To use Delayed ACKs, we can use the two-state machine proposed in DCTCP to send Delayed ACKs. Figure 3-1 shows how this two-state machine works in receiver. There are two states in the state machine, one is CE codepoint marked, and the other is unmarked. Each time when a transition of state occurs, an immediate ACK will be sent; and if there is no transition of state occurring, an ACK will be sent for every m packets. Whether the ECN-Echo flag is set depends on the situation of state transition. When the state keeps in CE=0, then ECN-Echo should be zero; when the state keeps in CE=1, then ECN-Echo should be one. When state is changed from 0 to 1, the ACK to be sent should be marked with ECN=0; otherwise, the ACK should be marked with ECN=1.

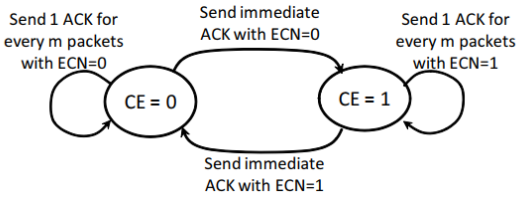

Figure 3-1 Two-state Machine (Alizadeh et al, 2010)

3.2 T-DCTCP Algorithm

T-DCTCP runs in both senders and receivers, and it controls the sending rate through increasing or decreasing congestion window in senders. It aims at resolving the flaws of TCP, ECN and RTT mentioned in Section 2 with the combination of RTT and ECN. Take a glance on T-DCTCP algorithm first.

T-DCTCP uses RTT and ECN fraction as both congestion indicators and parameters for controlling congestion window. We do believe that combining the information hidden within RTT and ECN can shape a better view of network congestion status than the one extracted from one of them. Pseudocode is shown in Figure 3-2. As is shown in pseudocode, T-DCTCP is driven by three parts: extracting information, judging network status, and controlling sending rate. Let us begin with the first part, extracting information.

The region before judgment region of pseudocode describes how to utilize RTT and ECN packet fraction. There are two EWMA functions in this region, and one is for calculating the difference of last RTT and current RTT and the other one is for implying *alpha* value for estimating the congestion severity. The reason why EWMA function is used is that there may occur some outliers transferred to this algorithm in practical application, and it can reduce the variation of the major parameters in case of rapid increase on them. A parameter called *normalized_gradient* is implied from new calculated *rtt_diff*, and it is a unitless value which means it is a scalar. This parameter is used to judge the trend of RTT and decrease sending rate. From the formula for implying it, it can be deduced that this parameter represents how RTT grows with the multiple rate of minimum RTT. The value of minimum RTT does not matter, and T-DCTCP only needs to know how RTT grows.

These valuable parameters allow us to judge how congested the network is, so it is very important and also difficult to use these valuable parameters well in next part: judging network status. Here comes three parameters: *alpha*, current RTT, and the difference of the previous RTT and current RTT. We choose a conservative scheme on network state judgment, which is using *alpha* as a delimiter between slight congestion level and severe congestion level and using RTT difference to determine how to control sending rate.

As is numbered in the pseudocode, there are 8 situations for sending rate control. In slightly congested network we control sending rate with *alpha*, scalars and *normalized_gradient*; and in severe congested network sending rate is controlled through *alpha* value. Before illustrating why we set up these 8 situations, it is necessary to explain why the flags T_{low} and T_{high} are needed. According to the statement in TIMELY, packet burst caused by a large flow can keep RTT increasing, so the RTT difference will keep increasing as well. However, packet burst happens as a large flow is being sent, and RTT increase at this situation is not usually

representing congestion occurring in network. Consequently, we need a flag to preclude this kind of situation and when RTT is below T_{low} the network is usually not congested. What's more, we cannot let RTT grow without limitation, and data center network should run in a tolerable latency. The situation when RTT exceeds the tolerable latency always means the congestion is highly severe. It is a must to define the upper bound of latency, so T_{high} is set.

The first four situations are in slightly congested network, and the last four situations are in highly congested network. The ① situation represents the network is totally clear and there is no congestion in the network, and packet burst issue of a single large flow is precluded. The ② situation represents there may be queue buildup issue happening and these packets may be in a large flow for the reason why they are not marked with ECN flag but with high latency. The ③ & ④ situations should be the normal gradient phase when there is not too much packets marked with ECN flag, and in this phase the trend of RTT change is important. At these two situations data center network is in a steady phase. The ⑤ situation represents there is packet burst issue happening and the flow is much larger than situation ①. The ⑥ situation warns that the network is experiencing severe congestion, and it needs emergent measures to take. The ⑦ & ⑧ situation means there is incast issue or queue buildup issue in switch, and the flow may be at the latter half of the queue.

Eight situations are described above, and we should take different measures to resolve each situation separately for the reason that each situation stands for different of congestion status that the flow experiences. Let us look at these situations one by one. As is mentioned before, when the congestion level is low, RTT value has better effect on controlling congestion level than that ECN fraction has. Therefore, from situation ① to situation ④ RTT is used to increase or decrease congestion window. For the first situation, congestion is not happening in the network, so congestion window should be increased and the addition step is self-defined depending on whether the scheme is aggressive or conservative. For the second situation, it needs to decrease the sending rate and decreasing ratio is defined by new RTT value. If the new RTT value is much greater than T_{high}, then congestion window will be decreased rapidly. As usual there is a self-defined multiplicative factor for controlling the decreasing ratio, and it is a factor representing whether the decreasing scheme is aggressive. When the network is in the third and the fourth situation, it is in steady status in which performance is excellent, so we aim

at keeping the network in this status. If RTT value decreases, then sending rate should be increased to get higher throughput; and if RTT value increases, then sending rate should be decreased by a ratio determined by *normalized_gradient* in case that congestion status get worsened. When RTT value decreases for five consecutive times, sending rate should increase by 5 times *addstep* for a faster convergence.

When congestion level gets high, RTT may vibrate rapidly and it is not capable of controlling sending rate. Consequently, from situation ⑤ to situation ⑧ ECN fraction is used to decrease congestion window. For the fifth situation, it is most probable that a short flow experiencing queue buildup issue, and the sending rate should be decreased but not too hard, so we decrease congestion window slightly less than normal situation and it is determined by a decrement factor. When the network is in the sixth situation, sharp reduction should be applied to sending rate, and it is wise to cut congestion window to half in order to recover the network as fast as possible. In the seventh situation, it is informed that the network is getting better, so T-DCTCP decreases congestion window slightly less than normal one like situation ⑤. For the eighth situation, it is a good idea to follow the measure proposed by DCTCP, and in this situation it is fear to reduce congestion window by the ratio of half of *alpha* value which is deduced from the latest ECN fraction. When it comes to incast problem and the sender is informed with high ECN fraction, the current flow is probable at the latter half of the queue, so it do need a reduction on congestion window. When the *normalized_gradient* is equal to or less than 0, it means that the network is getting better and T-DCTCP will allow a smaller reduction on congestion window to keep sending rate steady; and when *normalized_gradient* is greater than 0, the network is getting worse in congestion severity, so it is necessary to have a great reduction on congestion window in case that congestion severity keeps worsen.

Although the flows at the former half of the queue may be responded with low ECN fraction, their RTT values will be not ignored and the corresponding senders will take measure which is in either situation ② or situation ④ (most probable one).

Algorithm: T-DCTCP

Input: new_rtt, F (the fraction of packets that were marked in the last window of data)

Output: cwnd

```
01:  new_rtt_diff = new_rtt - prev_rtt;
02:  prev_rtt = new_rtt;
03:  rtt_diff = (1-β) * rtt_diff + β * new_rtt_diff;      Δβ: EWMA weight parameter, beta
04:  normalized_gradient = rtt_diff / minRTT;
05:  α = (1-g) * α + g * F;    Δg: EWMA weight parameter, g
06:  if α is less than alpha_factor then:
07:        if new_rtt < T_low then:  ①
08:              cwnd = cwnd +σ;
                 Δ σ: additive increment factor, addstep
09:              return;
10:       end if
11:       if new_rtt > T_high then:  ②
12:             cwnd = cwnd * (1 - γ * (1 - T_high / new_rtt) - cwnd * α / 2;
                Δγ: multiplicative decrement factor, decre
13:             return;
14:       end if
15:       if normalized_gradient <= 0 then:  ③
16:             cwnd = cwnd + N * σ;
          Δ: N = 5 if gradient < 0 for five consecutive entrances; otherwise N=1
17:       else:  ④
18:             cwnd = MIN(cwnd * (1 - γ * normalized_gradient), cwnd - 1);
19:       end if
20:  else: /* α from 0.125 to 1 */
21:       if new_rtt < T_low then:  ⑤
22:             cwnd = cwnd * (1 - α / (2 * θ));
                Δθ: decrement factor for alpha, theta
23:             return;
24:       end if
25:       if new_rtt > T_high then:  ⑥
26:             cwnd = cwnd /2;
27:             return;
28:       end if
29:       if normalized_gradient <= 0 then:  ⑦
30:             cwnd = cwnd * (1 - α / (2 * θ));
31:       else:  ⑧
32:             cwnd = cwnd * (1 - α / 2);
33:       end if
34:  end if
```

Figure 3-2 T-DCTCP Algorithm

3.3 Approaches to Control Sending Rate

T-DCTCP controls sending rate via modifying the value of congestion window, but that is only the concept in pseudocode. In actual implementation, compatibility with TCP should be taken into consideration.

Slow start is a well-known mechanism to avoid congestion, and if we abandon this mechanism, T-DCTCP will be incompatible with TCP. In order to utilize slow start mechanism, we implement the function, "ssthreshold", in "tcp_congestion_ops", which will be called each time when slow start is enabled. The new reduced congestion window will be transferred to slow start mechanism through the function "ssthreshold"; and if an increased congestion window is deduced from T-DCTCP algorithm, it will be transferred to the sender immediately. The reasons why we treat reduction and addition differently are that packet loss is the starting signal of congestion and slow start so the reduction of congestion window is needed when slow start begins, and that increase on congestion window need a fast convergence to get high utilization of bandwidth so that the network can keep high throughput.

3.4 Choosing Suitable parameters

This subsection is to tell how to select a suitable value for different parameters in T-DCTCP. There are totally 8 self-defined parameters in T-DCTCP, and different combination can lead to different performance.

For *g*, a EWMA weight parameter for calculating new *alpha*, we follow the suggestion from the paper of DCTCP and set it to 1/16 combining with the experiment environment. In addition, *beta*, another EWMA weight parameter for calculating new *rtt_diff*, is assigned with 1/8. These two parameters are used to filter outliers that may appear suddenly, but *beta* cannot be as small as *g* is, because beta determines how much new RTT difference influences.

For *addstep*, additive increment factor, *decre*, multiplicative decrement factor, and *theta*, decrement factor for *alpha*, parameters, their values determines whether the reaction to the change of RTT is conservative or aggressive as is mentioned in previous context. We decide to adopt a conservative scheme, so they are set to 1, 1/4, and 1/2.

As for *alpha_factor*, it is an important factor, and this is highly related to the performance in latency control. It is suggested to adopt a value of 1/8, and it is proven that it can deal with congestion situation switching well. We carry a small scale incast experiment and the

experiment result shows that the throughput keeps rising when *alpha_factor* is increasing but it stops rising when *alpha_factor* is more than 1/8; and tail latency keeps decreasing when *alpha_factor* is decreasing but it does not decrease when *alpha_factor* is 1/32. The trends of throughput and tail latency are shown in Figure 3-3. In order to get highest performance, the *alpha_factor* value of 1/8 is advised. However, if the tasks are latency-sensitive, it is wise to choose 1/16 as the *alpha_factor*.

As for T_{low} and T_{high}, we should consider how big the bandwidth in data center network is. Let assume that the bandwidth is 100Mbps, and the maximum sending window is 64KB, so a sender would take 5000 microseconds mostly to send data. According to this inference, T_{low} can cover most of the uncongested situation when it is set to 5000 us, and to prove the inference, we carry an incast experiment. The experiment result shown in Figure 3-4 reveals that throughput keeps increasing when T_{low} is increasing but it drops if T_{low} exceeds 5000 us. It would be adaptive to set T_{high} to 10 times of T_{low}. Figure 3-5 tells that throughput stops growing when T_{high} exceeds 50000 us. Based on this result, T_{low} and T_{high} converge at the point of 5000 us and 50000 us, so it is reasonable to believe that the inference above is workable.

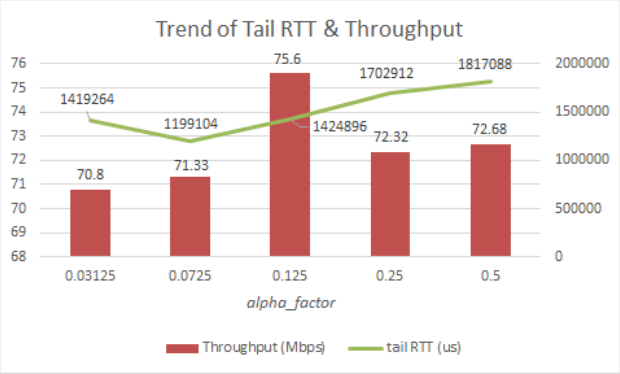

Figure 3-3 Trend of Tail RTT & Throughput when *alpha_factor* increases

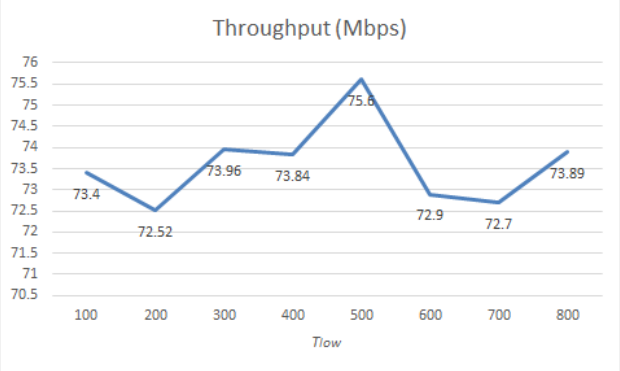

Figure 3-4 Trend of Throughput when T_{low} increases

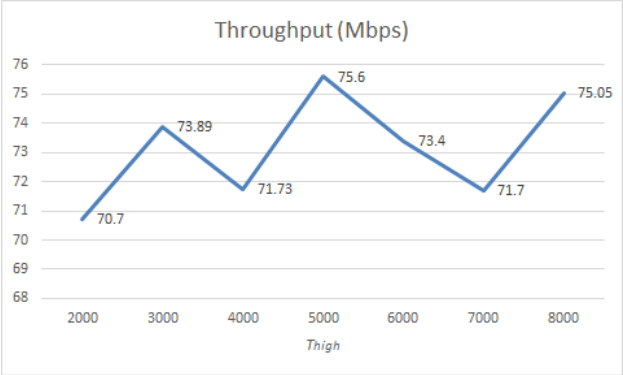

Figure 3-5 Trend of Throughput when T_{high} increases

4. Evaluation

4.1 Environment Setting

We have carried a small scale experiment for testing the performance of T-DCTCP comparing with the performance of DCTCP and TCP with CUBIC in a virtual network experiment, Mininet. Mininet provides a self-defined experiment environment for us, but due to the limitation on it, the experiment can be only run on 100Mpbs network environment. This experiment mainly focuses on the comparison of T-DCTCP, DCTCP and TCP with CUBIC, so this limitation will not have great impact on the experiment result.

Before testing the performance, all parameters are set to a reasonable valuable. In effect, we cannot find a way to change the value of ECN marking threshold in the ovsswitch of Mininet, so it is set to its default value. There is only one parameter needed to be determined in DCTCP, which is the EWMA value for calculating *alpha* value, and it is assigned with the value of 1/16. There exists 8 parameters in T-DCTCP, and the values set for them are listed as below: *g* is 1/16, *alpha_factor* is 1/8 (which determines if the network is in high congestion level), *beta* is 1/16 (EWMA value for calculating new *rtt_diff*), *addstep* is 1, *decre* is 1/4, *theta* is 1/2, T_{high} is 50000 us, and T_{low} is 5000 us. These parameters have been discussed in Section 3.4, so their functions will not be mentioned in this section. We simulate incast issue in Mininet, and there are fifteen servers simultaneously sending data to a client as fast as possible.

The experiment is carried on a spanning tree topology, called Fat-tree, and every two hosts belong to a field. Because of the complex structure of Fat-tree network topology, we have to apply a remote controller, ryu, to route flows to the destination. The topology is illustrated in

Figure 4-1. The first layer contains four core switches. There are 8 aggregation switches and 8 edge switches in second and third layer. Each switch in each layer has links to every switches in the next layer, so there exists loops in Fat-tree topology. Each edge switch connects to two hosts and it holds an independent network field. In other words, edge switch is the switch on the top of racks. Therefore there are 16 hosts in total in this network topology.

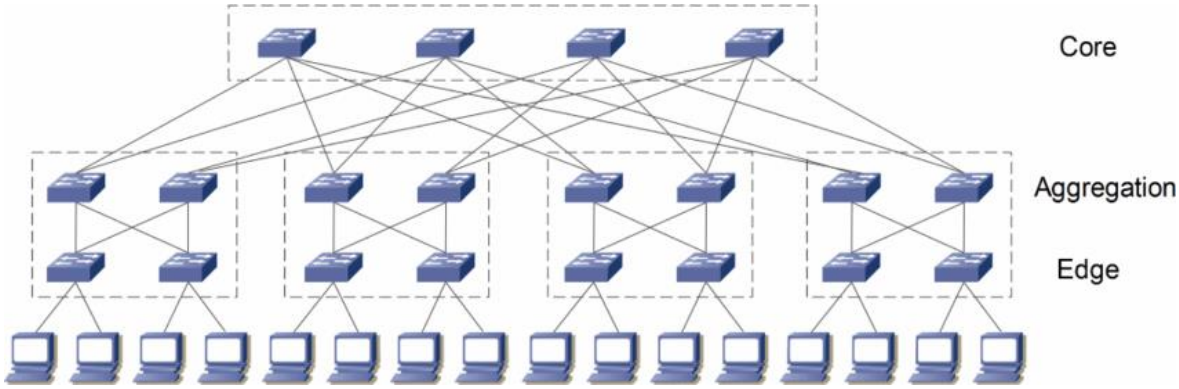

Figure 4-1 Fat-tree Topology

4.2 Evaluation Result & Analysis

Let us focus on several performance criteria for data center network, such as throughput, RTT, and flow completion time, and this section will list the data measured in this experiment of these criteria.

The first row of Table 4-1 compares how DCTCP and T-DCTCP performs in the aspect of throughput. As is shown, the throughput of T-DCTCP is smaller 2% than that of DCTCP, but it is close to the throughput of DCTCP, which means T-DCTCP can provide high throughput as DCTCP does. As for T-DCTCP, when incast issue happens, large flows will be assigned with lower bandwidth comparing to small flows because large flows are much possible to have higher latency than small flows. Therefore total throughput does not increase greatly until hardware bottleneck is solved like shallow buffer of switches.

As for latency, the second and third row of Table 4-1 compares how DCTCP, T-DCTCP and TCP with CUBIC performs in the perspectives of average RTT and 99^{th} percentile RTT. The average RTT of T-DCTCP is 9.4% smaller than the average RTT of DCTCP, and is 143.2% smaller than that of TCP with CUBIC. To the aspect of tail latency, T-DCTCP drops it by 37.7% and 123.2% comparing to DCTCP and TCP with CUBIC. The comparison reveals that T-DCTCP improves greatly the latency performance and it outperforms DCTCP and TCP with

CUBIC. Figure 4-2 plots the CDF diagram of DCTCP, T-DCTCP and TCP with CUBIC, which shows the difference among them clearly and the curve of T-DCTCP reaches the top the most quickly. We are informed from the CDF diagram that T-DCTCP enables most of the packets to be transmitted in lower latency than DCTCP and TCP with CUBIC do. For example, in 50^{th} percentile RTT of T-DCTCP achieves 185856 us, RTT of DCTCP achieves 261120 us and RTT of TCP achieves 1054208 us. T-DCTCP keeps good balance between small flows and large flows, and it can judge whether the current flow do suffer from high congestion. T-DCTCP will not reduce congestion window sharply until both RTT and ECN marked fraction are both high. Consequently, small flows experiencing incast issue will not be limited to an extremely low bandwidth, and T-DCTCP can have good performance in incast issue.

As for flow completion time, the experiment fetches the flow completion time of T-DCTCP and DCTCP and does comparison on average flow completion time and CDF curves between these two algorithms. Figure 4-3 is a CDF diagram of the flow completion time of T-DCTCP and DCTCP. The CDF diagram clearly presents that the blue curve of T-DCTCP rises more rapidly than the orange curve of DCTCP does, and this means the flows transmitted with T-DCTCP congestion control algorithm complete earlier than the flows transmitted with DCTCP. For example, in 50^{th} percentile the flow completion time of T-DCTCP is about 800 millisecond and the one of DCTCP is nearly 1000 millisecond. Figure 4-4 is a bar diagram which compares the completion time of flows in different sizes between T-DCTCP and DCTCP. From this bar diagram, we should notice that as for small flows including 2KB, 50KB and 1MB, T-DCTCP allows them to suffer lower latency than DCTCP does which means the small flow completion time of T-DCTCP is smaller than that of DCTCP. Comparing to DCTCP, the completion time of 2KB flow drops by about 1.4X, the completion time of 50KB flow reduces by about 31%, and the flow completion time of 1MB flow falls by about 48.2%. However, DCTCP does slightly bit better in the flow with the size of 10MB than T-DCTCP does, and Figure 4-4 shows that 10MB flow transmitted with DCTCP suffers approximately 8.6% lower latency than that transmitted with T-DCTCP. As is mentioned before, T-DCTCP gives better bandwidth to small flows comparing to DCTCP, so T-DCTCP enables small flows to have a better completion time. However, the flow completion time of large flows may be influenced and it will be increased slightly especially in incast issue.

Table 4-1 Comparison on Throughput and RTT

Metric	DCTCP	T-DCTCP	TCP with CUBIC
Throughput (Mbps)	79.44	77.99	N/A
Avg. RTT (us)	362374	331272	805672
99th RTT (us)	1164288	845824	1888256

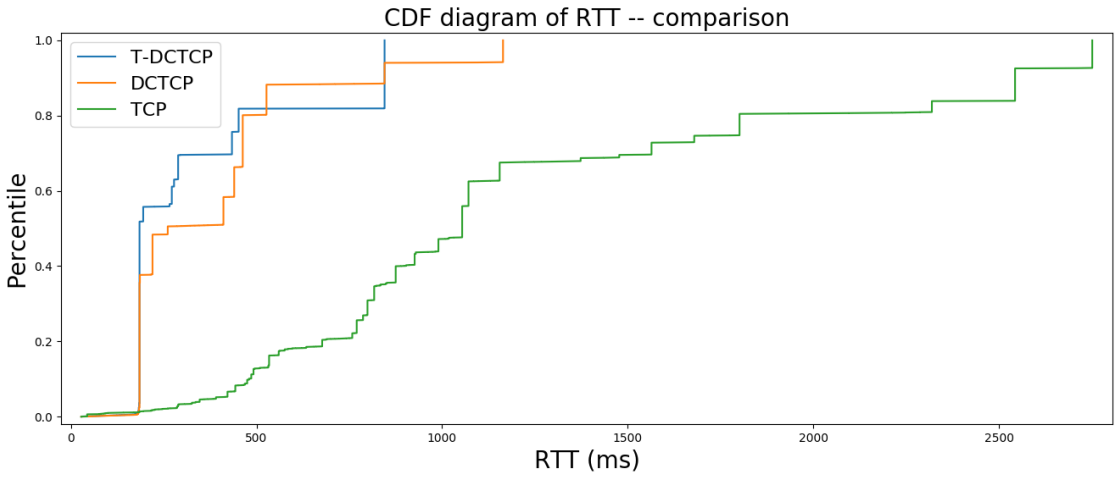

Figure 4-2 CDF Diagram of RTT, Comparison among DCTCP, T-DCTCP and TCP with CUBIC

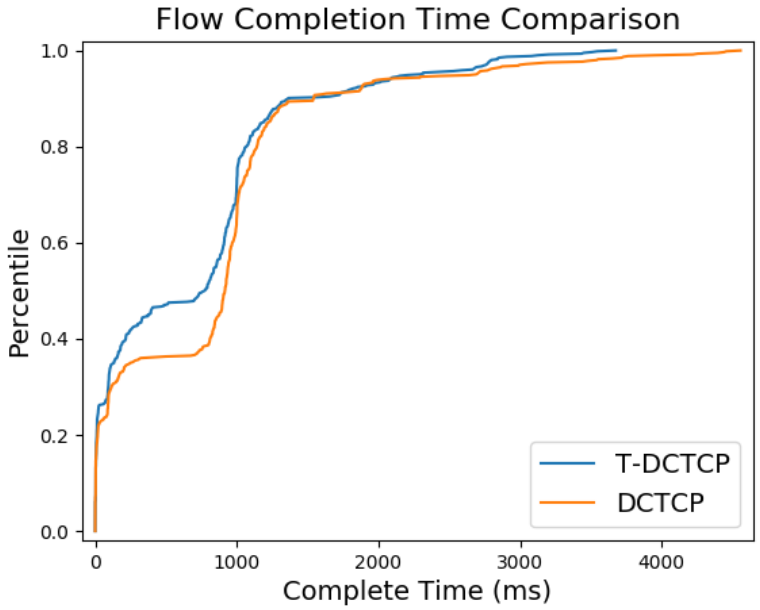

Figure 4-3 CDF diagram of Flow Completion Time, Comparison between DCTCP and T-DCTCP

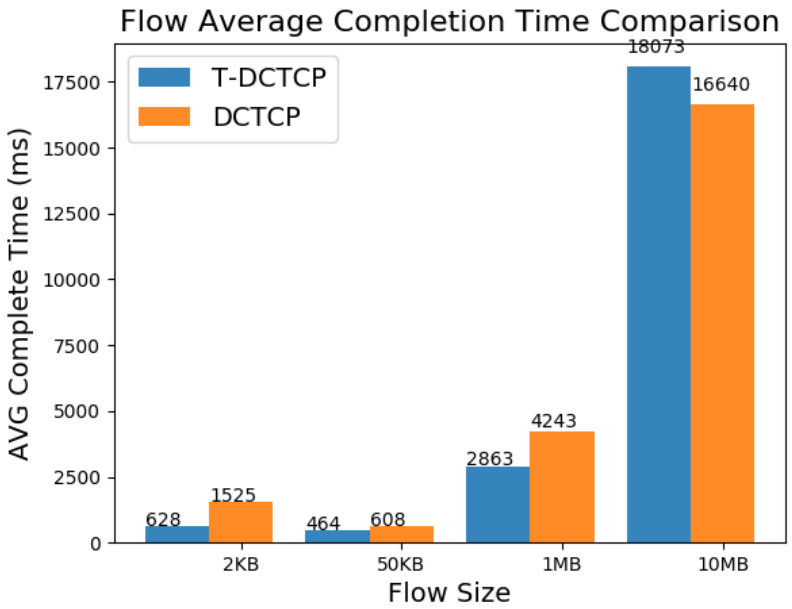

Figure 4-4 Bar Diagram of Average Flow Completion Time of Different Flow Sizes

5. Discussion

In previous section, this thesis shows some related performance indices of congestion control algorithm, which can describe live situation of data center network. From the result shown above, we make comparison among TCP with CUBIC, DCTCP and T-DCTCP. It is obvious that the performance of DCTCP and T-DCTCP is far beyond that of TCP with CUBIC, especially in latency control. Let us look back in Section 1, the initial purpose of designing T-DCTCP is to improve the performance of latency control and to make it better than DCTCP. According to the experiment result, T-DCTCP does perform slightly better than DCTCP in latency control, and it is exciting to have a similar throughput with latency reduction.

As for incast issue, T-DCTCP does not appear to have starvation situation and latency clamp when the incast simulation is performed. The experiment result shows that T-DCTCP keep tail latency far lower comparing to DCTCP, which actually indicates better performance on handling incast issue. As for outcast issue, comparing to DCTCP, T-DCTCP achieves lower completion time of small flows which are mainly influenced by outcast issue. However, a side effect of this is that the flow completion time of large flow increases. We have to optimize T-DCTCP performance on large flows without increasing the completion time of small flows.

As is discussed in Section 3.4, there are different ways to choose parameters. If aggressive

scheme is required, we can increase *addstep* and decrease *decre* and *theta*, which are the parameters determine how congestion window to be increased or decreased. The situation separator, *alpha_factor*, is proven that the value of 0.125 performs better in both throughput and latency control, but when low latency is first, it is suggested to choose 0.0625.

There are some drawbacks in T-DCTCP, like bad performance in large flows and no improvement in throughput. We will keep optimizing T-DCTCP algorithm and improve its accuracy of controlling congestion window and its precision on describing congestion status.

6. Conclusion

With so many cloud-based applications are thriving, data center is facing with congestion problem. Most of the communications are in TCP flows, but TCP cannot be compatible with Partition/Aggregate model used in data center because of several issues pointed out in Section 2. Fortunately, various congestion control algorithms for TCP are issued to make TCP suitable for data center network. Some typical algorithms are solving the incompatible problem well, such as DCTCP and TIMELY, which inspire us to design congestion control algorithm based on both RTT and ECN called T-DCTCP. EAR is also based on both RTT and ECN as congetion indicators like T-DCTCP, but it needs modification on switches. T-DCTCP does not fail to catch up with DCTCP, and it satisfies our expectation instead. T-DCTCP actually decreases latency and completion time of small flows in each link comparing to DCTCP. It is expecting to deploy T-DCTCP in large scale data center to test whether T-DCTCP can perform as well as its performance in our small scale experiment. There is much work to do in the future, and it is expecting to improve the performance of T-DCTCP in large flows. In conclusion, T-DCTCP proves that the approach of combining RTT and ECN is possible to improve the performance of congestion control, and there are more and more ways measuring packet latency accurately as is illustrated in Section 3 which can absolutely enable researchers to find more and more solutions to improve the performance of TCP in data center with the help of precise RTT.

Acknowledgements

I have to thank Doctor Cui Lin and Zhang Yuxiang sincerely for their assistance to me, which helps me find problems and improve T-DCTCP.

References

Abdelmoniem, A. M., Bensaou, B., & Abu, A. J. (2018). Mitigating incast-TCP congestion in data centers with SDN. *Springer Annals of Telecommunications*, 73(3-4), 263-277.

Alizadeh, M., Atikoglu, B., Kabbani, A., Lakshmikantha, A., Prabhakar, R. P. B., & Seaman, M. (2008). Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In Proceedings of Communication, Control, and Computing, 2008 46th Annual Allerton Conference (pp. 1270-1277). Urbana-Champaign, IL, USA: IEEE.

Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., & Sridharan, M. (2010). Data center TCP (DCTCP). *ACM SIGCOMM Computer Communication Review (SIGCOMM '10)*, 40(4), 63-74.

Black, D. (2018). Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation. Retrieved from: https://www.rfc-editor.org/rfc/pdfrfc/rfc8311.txt.pdf. (accessed 21 January, 2018)

Chen, W., Cheng, P., Ren, F., Shu, R., & Lin, C. (2013). Ease the queue oscillation: analysis and enhancement of DCTCP. In Proceedings of 2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS) (pp. 450-459). Philadelphia, Pennsylvania, United States: IEEE ICDCS.

Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107-113.

Fang, S., Foh, C.H., & Aung, K. M. M. (2012). Prompt congestion reaction scheme for data center network using multiple congestion points. In Proceedings of IEEE ICC 2012 (pp. 2679-2683). Ottawa, Canada: IEEE ICC.

Floyd, S. (1994). TCP and Explicit Congestion Notification. *ACM SIGCOMM Computer Communication Review*, 24(5), 8-23.

Goel, V. (2017). What is TCP Incast. Retrieved from: http://qr.ae/TU1Esz. (accessed 27 March 2018)

Haitao, W., Feng, Z., Guo, C., & Zhang, Y. (2013). ICTCP: Incast congestion control for TCP in data center networks. *IEEE/ACM Transaction on Networking*, 21(2), 345–358.

Ito, Y., Koga, H., & Iida, K. (2018). A Bandwidth Allocation Scheme to Improve Fairness and

Link Utilization in Data Center Networks. *IEICE Transactions on Communications*, 101(3), 679-687.

Kleinberg, A. (2013). 5 Small Ways to Use Big Data to Majorly Improve Business. Retrieved from: https://www.entrepreneur.com/article/227957. (accessed 25 April 2018)

Liu, S., Basar, T., & Srikant, R. (2007). TCP-Illinois: A loss- and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation*, 65(6-7), 417-440.

Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., & Zats, D. (2015). TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM Computer Communication Review (SIGCOMM '15)*, 45(4), 537-550.

Munir, A., Qazi, I.A., Uzmi, Z.A., Mushtaq, A., Ismail, S.N., Iqbal, M.S., & Khan, B. (2013). Minimizing flow completion times in data centers. In Proceedings of IEEE INFOCOM 2013 (pp. 2157-2165). Turin, Italy: IEEE INFORCOM.

Naman, A. T., Wang, Y., Gharakheili, H. H., Sivaraman, V., & Taubman, D. (2018). Responsive high throughput congestion control for interactive applications over SDN-enabled networks. *Computer Networks*, 134, 152-166.

Pierce, D. (2017). How Apple Finally Made Siri Sound More Human. Retrieved from: https://www.wired.com/story/how-apple-finally-made-siri-sound-more-human/. (accessed 21 April 2018)

Qin, Y., Yang, W., Ye, Y., & Shi, Y. (2016). Analysis for TCP in data center networks: Outcast and Incast. *Journal of Network and Computer Applications*, 68, 140-150.

Saeed, A. (2016). Congestion Control in Datacenters (PPT). Retrieved from: https://www.cc.gatech.edu/~amsmti3/files/6250_spring17_cc_in_dc.pdf. (accessed 12 January 2018)

Salim, J. H., & Ahmed, N. (2000). Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks. Retrieved from: https://www.ietf.org/rfc/rfc2884.txt. (accessed 23 April 2018)

Sreekumari, P. (2018). Rate Adjustment Mechanism for Controlling Incast Congestion in Data Center Networks. *Springer Information Technology-New Generations*, 558, 95-100.

Sreekumari, P., & Jung, J. (2015). Transport protocols for data center networks: a survey of

issues, solutions and challenges. *Photonic Network Communications*, 31(1), 112-128.

Stephens, B., Cox, A.L., Singla, A., Carter, J., Dixon, C., & Felter, W. (2014). Practical DCB for improved data center networks. In Proceedings of IEEE INFOCOM 2014 (pp. 1824-1832). Toronto, Canada: IEEE INFOCOM.

Tellas, R. (2017). Adding More Wireless Access Points: What It Means for Networks. Retrieved from: https://www.belden.com/blog/digital-building/adding-more-wireless-access-points-what-it-means-for-networks. (accessed 25 April 2018)

Vamanan, B., Hasan, J., & Vijaykumar, T. N. (2012). Deadline-aware datacenter TCP (D^2TCP). In Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication (pp. 115-126). Helsinki, Finland: SIGCOMM.

Venkatesh, G., & Arunesh, K. (2018). Map Reduce for big data processing based on traffic aware partition and aggregation. Retrieved from: https://link.springer.com/content/pdf/10.1007%2Fs10586-018-1799-6.pdf. (accessed 30 April 2018)

Wang, H., & Shen, H. (2018). Proactive Incast Congestion Control in a Datacenter Serving Web Applications. Retrieved from: http://www.cs.virginia.edu/~hs6ms/publishedPaper/Conference/2018/INFOCOM2018-PICC.pdf. (accessed 3 March 2018)

Zeng, G., Bai, W., Chen, G. Chen, K., Han, D., & Zhu, Y. (2017). Combining ECN and RTT for Datacenter Transport. In Proceedings of the First Asia-Pacific Workshop on Networking APNet'17 (pp. 36-42). Hong Kong, China: APNet.

Graduation Thesis Evaluation for Undergraduate Students

Supervisor's Comments：   Marks：_____/100 Signature： Date (dd/mm/yyyy):
Evaluator's Comments: Marks：_____/100 Signature： Date (dd/mm/yyyy):
Dean's Comments: Marks: _____/100 Signature： Date (dd/mm/yyyy):