Improving TCP congestion control in datacenters

By
Taimur Hafeez
00000172340

Supervisor
Dr. Nadeem Ahmed
Department of Computing

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science in Information Technology (MS IT)

In
School of Electrical Engineering and Computer Science,
National University of Sciences and Technology (NUST),
Islamabad, Pakistan.

(May 2018)

Approval

It is certified that the contents and form of the thesis entitled "**Improving TCP congestion control in datacenters**" submitted by **Taimur Hafeez** have been found satisfactory for the requirement of the degree.

Advisor: **Dr. Nadeem Ahmed**
Signature: ____________________

Date: ____________________

Committee Member 1: **Dr. Muazzam Ali Khan Khattak**

Signature: ____________________
Date: ____________________

Committee Member 2: **Dr. Syed Taha Ali**

Signature: ____________________
Date: ____________________

Committee Member 3: **Dr. Arsalan Ahmad**

Signature: ____________________
Date: ____________________

Author's Publications

1. T. Hafeez, N. Ahmed, B. Ahmed and A. W. Malik, "Detection and Mitigation of Congestion in SDN Enabled Data Center Networks: A Survey," in IEEE Access, vol. 6, pp. 1730-1740, 2018.
doi: 10.1109/ACCESS.2017.2780122
This publication forms the bulk of Chapters 2 and 3.

Abstract

These day interactive and social media applications demand huge computation and storage capacity. To cope with the issue, technology giants such as Google and many others have deployed large number of datacenters. Inside these datacenters, where queries are highly time sensitive, about 80% of the traffic is East-West (remaining within the datacenter). TCP restricts the performance of the datacenter due to its various parameters designed for wide area networks. In this regard, research community proposed many modification to TCP that could improve its performance in datacenter environment. Google has recently proposed increase in TCP's Initial window (IW) to 10MSS. In this work, we would be investigating performance of TCP with IW=10 in datacenter environment. In this simulation-based study, we would evaluate the performance issues related to IW=10 proposition if implemented in datacenter environment. We propose SDN controller based application to mitigate the incast issue with the usage of IW-10. This application uses network to avoid incast issue and does not require any modification on end hosts.

Dedication

To my parents,
without whom it was almost impossible for me to accomplish.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Taimur Hafeez**
Signature: __________________

Acknowledgment

I am absolutely grateful for my thesis adviser, Dr. Nadeem Ahmed for his leadership in ensuring my successful development through academic achievement and for polishing my skills. The door to his office was always open whenever I ran into a trouble spot or had a question about my research or writing.

Finally, I must express my very profound gratitude to my family members and friends for providing me with unfailing support and continuous encouragement throughout my years of life.

Table of Contents

List of Figures

List of Tables

List of Symbols

Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
MAC	Media Access Control
MPLS	Multiprotocol Label Switching
ONF	Open Networking Foundation
RTO	Retransmittion Timeout
RTT	Round Trip Time
SDN	Software Defined Networking
IP	Internet Protocol
TCP	Transmission Control Protocol
MSS	Maximum Segment Size
IW	Initial Window
NS	North South
EW	East West
Cwnd	Congestion window
Rwnd	Receiver window
QoS	Quality of Service
VM	Virtual Machine
ECMP	Equal-Cost MultiPath
BDP	Bandwidth Delay Product
OFDP	Openflow Discovery Protocol
RSTP	Rapid Spanning Tree Protocol
SRU	Server Requested Unit
ToR	Top of Rack
FCT	Flow Completion Time
RCT	Request Completion Time
ML	Machine Learning

Nomenclature

$Th1_{buffer}$	Soft threshold
$Th2_{buffer}$	Hard threshold
n	Number of servers
$rwnd$	TCP receiver window
Qx	Queue for more than 1 MSS Bytes
$Q1$	Queue for 1 MSS Bytes
$Q0$	Queue for 0 Bytes

Chapter 1

Introduction

In this chapter, we first describe the motivation behind this work. We then move to problem statement. This is followed by objective and goal of this research. Finally, we provide a roadmap of this thesis.

1.1 Motivation

Data centers have become the part of enterprises of all scale. Technology Giants such as Youtube, Amazon, Google or Facebook have built large scale data centers to provide services. A data centers consists of thousands of computer servers where they are connected in different architecture (topologies) with the help of switches. Traffic inside a data center is mainly divided into to categories, Elephant; throughput sensitive, and the mice; latency sensitive. Since data center is meant to provide services with the fundamental requirement of reliability. In communication networks, reliable communication conventionally make us choose no other than Transmission Control Protocol(TCP). TCP was not designed for data centers, thus, its performance deteriorates in data centers considerably. Recently, Software Defined Networks (SDNs) have gained pace from academia and industry. Google's adoption to SDNs [1] is the practical example of this. Next generation network architecture, the SDN, provides more centralized control in the hands of network administrators, making the network devices dumb, therefore cheap . Such centralized management feature of the SDNs makes its usage ideal for data centers alike networks where each host and device is in control of service provider. Though a number of data centers issues (security, load balancing, routing, to name a few) can be enhanced with capabilities of SDN, however, our this work focuses on congestion control caused by TCP inside data centers using SDNs approach.

1.2 Problem definition

TCP after connection establishment, sends different number of Maximum Segment Size (MSS). This window of TCP is called Initial Window (IW). Its value varies from one operating system to the other. Its value has direct impact on the performance of the connection. For example if large value is used, a fewer number of Round Trip Time (RTTs) are required to complete the data transfer and vice versa. On the other hand, large value would result in triggering of burst of flow right after final ACK of the three-way handshake process. This would not be case with small value of IW. In this regard, Google has made effort for the standardization of IW=10 in WAN services. Does this IW-10 is suitable for data centers like networks where large number of concurrent servers communicate with one server and switches in between are shallow buffered? We show that this value could have drastic impact on the throughput of data centers flows. SDNs can make this proposition of IW-10 realistic because SDN controllers have knowledge of entire network and various approaches can be used to either limit or throttle the sender according to dynamic network conditions.

1.3 Objectives and goals

In this thesis, Our two-fold goals are below:

- We unveil the effects of IW-10 on traffic in data centers environment. More specifically, we target relation of IW-10 with incast problem; concurrent servers competing for one output port. We explore how IW-10 can increase severeness of the incast in the presence of shallow buffer switches.

- We propose how SDN can help make the adoption of IW-10 possible. We explain the usability of any IW value on end hosts. Our method neither proposes new version of TCP, nor does it require any changes on end host TCP implementation.

1.4 Thesis roadmap

Our thesis organization is as follows: In Chapter .2, we first provide fundamental concepts of conventional networking, TCP and SDNs. These concepts provide base knowledge to grasp the advanced concept in later chapters. Chapter .3 describes different approaches from literature which are proposed to mitigate the effects of congestion in data centers. This chapter is followed by Chapter .4, in which we describe in detail our contribution; A controller application for avoiding TCP incast in data centers. Chapter .5 gives results of the simulation. We conclude and provide future research directions in .6

Chapter 2

Background

In this chapter, we provide the background knowledge essential to follow the research work discussed in subsequent chapters.

2.1 Data Centers

Technology giants Twitter, Youtube, Amazon, Google, manage large scale data centers to provide services in real time. Data centers typically supports link of Gbps while switches with low, shallow buffer are commonly employed. Due to this, data center traffic faces very low latency. All this setup is networked mostly in the form of a tree topology [2, 3], where multiple paths exist from one host to the other.

2.1.1 Traffic Characteristics

Based on the origin and destination, the traffic inside a data center can be classified in two types namely North-South (NS) and East-West (EW). NS traffic arrives from outside the data center e.g., application query and leaves after required processing within the data center in the form of a response to the query. EW traffic is intra-data center that flows between the servers inside the data center to complete either computation or storage related tasks. NS and EW traffic is shown to be distributed in the ratio of 20:80 i.e. almost 80% of the traffic inside a data center is EW [3, 4]. This implies two important things, first that EW traffic requires careful traffic engineering because of its volume and velocity. Second, dealing with EW traffic is easier than the NS traffic as both ends of communication lies within the administrative domain of the data center. For example, it is trivial to deploy custom/replacement protocols that can outperform the standard protocols.

Traffic inside a data center carries a mix of flows with varying characteristics and demands. These flows are broadly classified as Elephant and Mice flows. Elephant flows are typically TCP flows having large size (from few MBs to even GBs) that persist for a duration generally from a few seconds to hours. These flows demand high throughput and are non-latency-sensitive. Applications such as Hadoop, MapReduce [5], VM migration and cloning generate Elephant flows. Mice flows, on the other hand, are short flows that are highly latency-sensitive. These are typically bursty in nature and are normally generated by gaming, voice and web traffic. The ratio of mice to elephants in a data center is 80% to 20% with elephant flows carrying 80% of the total bytes and mice flows carrying 20% of rest of the bytes in the data center [6, 7]. Therefore, although less in number, a large volume of traffic inside a data center is TCP-based.

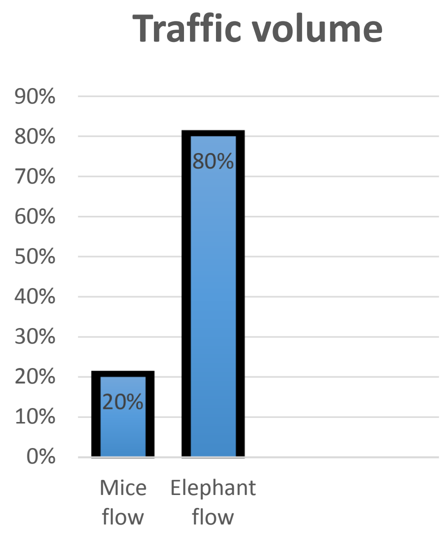

Figure 2.1: Comparison of Traffic volume

2.1.2 Problems and challenges

TCP exercises congestion control at senders so that it can adapt to the network conditions. The problem is that TCP was originally designed keeping the network dynamics for the wide area network in mind. Standard TCP struggles to perform when it is deployed as the transport protocol in the unique environment of a data center that supports high-bandwidth and low-latency. Changes are thus warranted in the standard TCP to adapt it to the

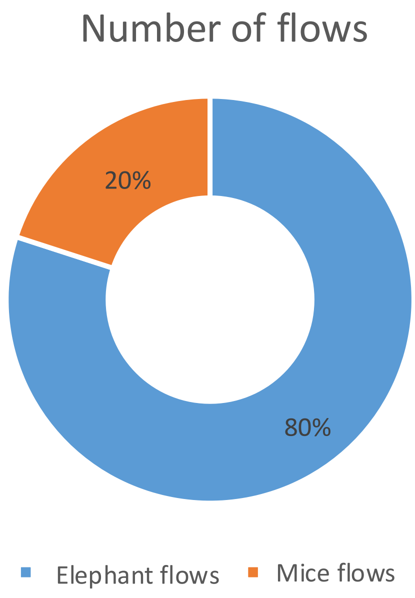

Figure 2.2: Share of Mice and Elephant flows in data center

dynamics of the data center traffic. TCP faces issues such as TCP incast, TCP outcast and queueing delay etc. [8, 9] inside a data center.

TCP incast [10] is a kind of congestion collapse due to simultaneous transmission of data by multiple synchronized servers to the same receiver/aggregator server resulting in the overwhelming of the shallow buffer at the bottleneck switch. This results in severe throughput degradation up to 90%. This problem is very common with storage traffic in data centers operating on partition/aggregate model [11]. In partition/aggregate model, the aggregator partitions a single application request and sends concurrent sub-requests to multiple servers for fulfilment of the original request. The response from multiple servers (see Figure 2.3) arrives simultaneously (following many-to-one communication pattern) resulting in exhaustion of the shallow switch buffer leading towards large number of packet drops. TCP congestion control mechanism tries to recover from this situation through timeouts and retransmissions. This ultimately results in severe throughput collapse [12]. TCP outcast [13] is another issue that is primarily caused by different types of flows competing for the same output port at a switch. For example, consider Elephant and Mice flows arriving through different input ports of a switch while competing for the same output port. The elephant flows easily saturate the output port queue and leads towards starvation of the mice flows. Mice flow may even face a total port blackout depriving it of any bandwidth utilization. Similarly, Elephant flows consume high throughput for a long period of time, especially in periods of congestion, leaving mice flows with non-trivial queuing delays thus affecting their delay sensitiveness drastically.

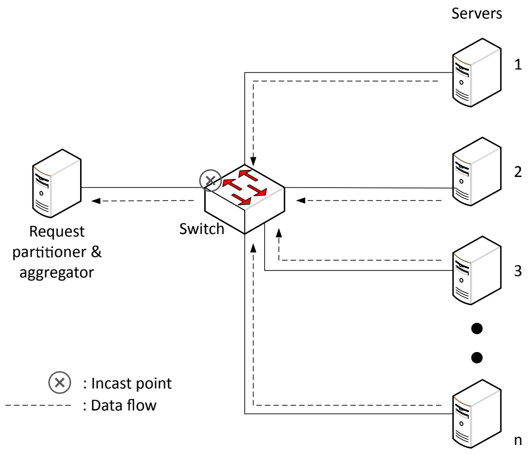

Figure 2.3: TCP Incast Scenario

2.1.3 TCP's Sending Rate

The main feature of TCP is its default congestion control mechanism that enables senders to react to the perceived congestion in the network without requiring any explicit support from the network. The sender can infer the congestion in the network based on indirect measures such as measurements of RTTs (Round trip times) and segment losses. It is a window-based, end-to-end mechanism where the sender maintains a dynamic sliding window named Congestion Window (Cwnd) that controls the sender's sending rate. Cwnd grows and shrinks as a function of the perceived network congestion. TCP also maintains another window, Receive/Advertised Window (Rwnd/Awnd) that is an indication of the current buffer capacity for that TCP connection at the receiver. Rwnd is used for flow control between the sender and receiver. The receiver advertises this window (in Acks) to prevent the sender overflowing its current buffer space. Collectively, the sender limits the amount of un-acknowledged sent packets in the network to the minimum of Cwnd and Rwnd as given in following,

$$TCP_{sendingrate} = min(cwnd, rwnd) \tag{2.1}$$

TCP's IW and MSS

The Maximum Segment Size (MSS) is the size that can be handled by the link layer (MTU minus the upper layer headers). It is shared among two

TCP ends at start of a TCP connection (SYN and SYN/ACKs). MTU for Ethernet is 1500 bytes, while MSS works out to 1460 Bytes accounting for 20 Bytes of IP and that of TCP. The TCP Initial Window (IW) represent the number of segments a TCP sender can send when the connection is initially established. The size of the IW varies from one operating system to the other. For instance, current kernel implementation of TCP in Linux has IW=10.

2.2 Non-SDN Networking

Normal network architecture is deployed in distributed fashion. Network consists of intelligent as well as dumb devices. All devices communicate with each other to provide services. Every device has own independent operating system which runs on hardware. Similarly all feature and applications like routing, security (firewalls and Access control lists), Quality of Service (QoS) to mention a few, are implemented on the top of operating system of devices. This is shown in figure .2.4. There is no central entity controlling all of them. There are following drawbacks of traditional networking

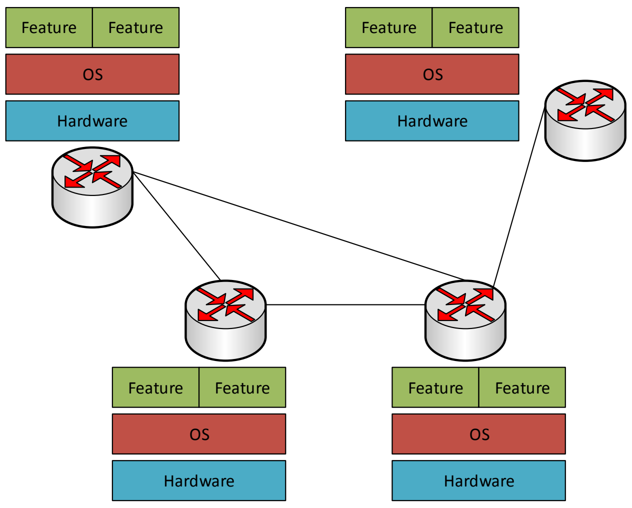

Figure 2.4: Traditional Networking overview

- Every device and protocol is proprietary. One can not build its own application and implement on network devices.
- In case of modification to application, each device must be configured explicitly.
- Using more rich applications on devices requires cost to be paid to vendor.

- Networks are configured from command line, no programming feature can be provided once device is installed with OS and deployed in the network.

2.3 Software Defined Networks

Software Defined Networking (SDN) has recently become subject of much research attention. The centralized nature of SDN coupled with its capabilities to provide programmability and dynamic flow management makes it a suitable paradigm to handle the congestion issues in data centers. Open networking foundation (ONF) [14], a non profit organization involved in development and standardization of SDN, has defined SDN as a network that offers separation of control plane from data plane and programmability of the networks [15]. Three major components of SDN include the centralized SDN controller, SDN enabled switches, and the end hosts. The controller manages the task of controlling the control plane and is responsible to enforce policy parameters in the form of flow rules in the data planes of the network devices. The controller being centralized is able to probe for network statistics from all devices under its control to oversee the current state of the network.

The SDN architecture describes two main APIs: i) Northbound API defines the application programming interface exposed by the controller to the network applications to configure the controller and its policy parameters. ii) The Southbound API defines an interface between a SDN controller and the network devices that it controls (e.g., SDN switches). OpenFlow [16], the most readily recognized protocol associated with SDN, is an example of the Southbound API.

2.3.1 OpenFlow

OpenFlow is a de-facto standard Southbound API which connects SDN controller with OpenFlow enabled switches. The controller issues commands to the switches and receive statistics and error messages in return using OpenFlow commands. OpenFlow has been continuously improved since its emergence. For example, its first version (1.0) only had 12 match fields. Though version 1.6 is in development phase, however, state-of-the-art version is 1.5 which provides more than 40 match fields and plenty of other rich features like metering, Pipeline processing and Group table etc.

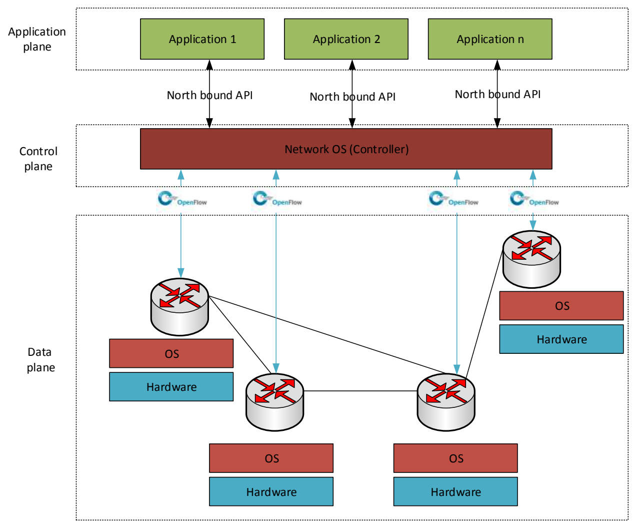

Figure 2.5: High level view of SDN architecture

2.3.2 How it Works?

In contrast with conventional networks which are distributed, a SDN is controlled in a centralized fashion. Controller has entire network intelligence and all forwarding devices like switches are dumb and depend on policy defined in flow rules installed by controller. Every switch contains one or more flow tables. These flow table(s) contain called match rules, called flow rules hereafter. On arrival of packet on any switch port, switch checks if any rule exists for this particular packet. This match depends on match fields used by the SDN controller at the time of flow rule installation. If packet does not match any rule in any table, it is forwarded to SDN controller for decision about it. The SDN controller computes next hop and actions for this packet and may install flow rule for subsequent packets of the same flow. If it only sends packet out message, switch simply forwards packet and does not install rule. If message is Flow-mod, then rule is installed in switch's table and all subsequent packet matching these fields are forwarded by switch without consulting controller. Depending on version of OpenFlow, controller can install any combination of match fields e.g(source/destination Mac addresses, source/destination IP addresses etc). There is a little bit of restriction here that is worth mentioning. One cannot simply use upper layer without defining low layer protocols in match fields. For instance we cannot install flow rule based on match fields TCP source/destination ports without defining protocol type (IP and TCP) in match fields.

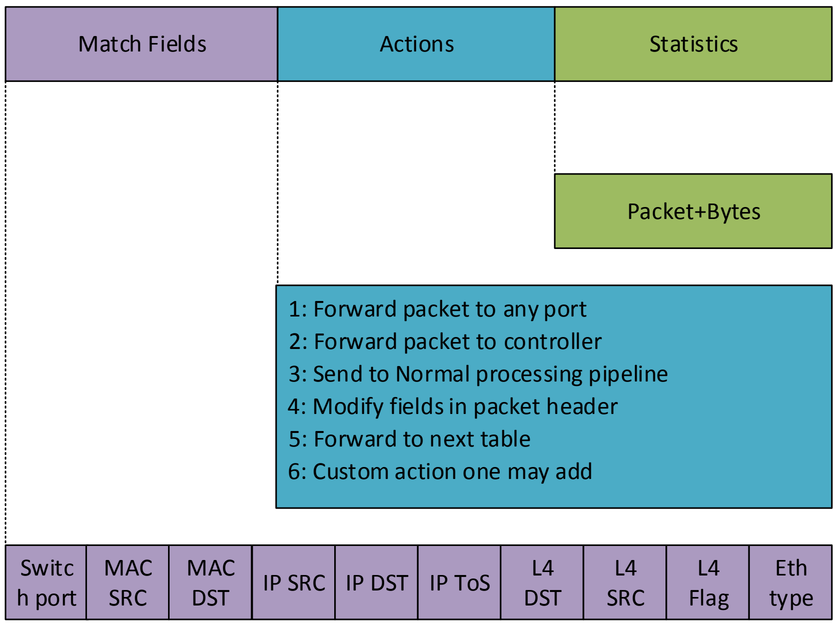
Match Fields
Actions
Statistics
Packet+Bytes
1: Forward packet to any port
2: Forward packet to controller
3: Send to Normal processing pipeline
4: Modify fields in packet header
5: Forward to next table
6: Custom action one may add
Switch port
MAC SRC
MAC DST
IP SRC
IP DST
IP ToS
L4 DST
L4 SRC
L4 Flag
Eth type

Figure 2.6: SDN flow rule components

2.3.3 Advantages

SDN have several advantages over traditional networking and are mentioned below:

- Since forwarding function is decoupled from control, we can easily program the network according to our requirements.
- It provides more control to network administrators to implement the policies dynamically.
- Since intelligence of the network is embedded in controller, all remaining devices(switches) can be dumb and hence cheap.
- Its programs are not proprietary. One can use opensource controller to develop his program and implement on SDN switches. Thus it eradicates proprietary protocols, standards and devices.

Chapter 3

Related Work

In this section we discuss various related works in the domain of congestion control in SDN enabled data center networks.

3.1 Congestion Control Techniques using SDN

Congestion control measures in SDN enabled data centers can be classified in two broad categories: i) *In-Network Congestion Control* covers solutions that rely on support from the network devices (such as switches and the SDN controller) to mitigate the effect of congestion in the data center. These solutions does not require support from either the end-host TCP/IP stack or the standard hypervisor (for virtualized data centers). These solutions can thus work with any OS and TCP variant deployed by the service provider or the tenant in their allocated Virtual Machines (VM). ii) *Hybrid Congestion Control* on the other hand involves both the end-hosts/hypervisor and the network elements to detect and mitigate the effect of congestion in the data center. End-hosts have greater computational power and access to complete TCP/IP stack as compared with the network devices. Similarly, a hypervisor is an optimal location to identify, detect and handle flows associated with each VM in a virtualised environment.

3.1.1 In-Network Congestion Control

Based on avoidance method of congestion, we further sub categorized this main category and describe as follows.

3.1.2 Providing larger switch buffer

Switches are normally shallow, meaning only having limited capacity of 100s of KBs. The obvious solution, although not related to SDN is to increase this size per port to cater the burst of packets. This could result in less number of packet being lost in the network and thus reduced retransmissions. It is shown in [17] that using large buffer capacity could delay the incast event and number of concurrent server can be increased with large buffer capacity. At European Organization for Nuclear Research, CERN, to provide large bandwidth, [18] is proposed which works on the principle of software switch. Unfortunately, that particular switch with large buffer capacity is only well suited for their network, not for generic data center environments. There is another issue that might occur with the increased buffer sizes; due to large queue, there can be much of variation in round trip times that could eventually affect congestion control mechanism in practice at end hosts. Moreover, long buffer delay may result in excessive TCP timeout event, worsening the situation more. Also, the increased buffer comes at increased cost.

3.1.3 Flows Scheduling/Re-routing

Fat-tree alike topologies, mostly used in data centers, provide multiple links from source to destination with the help of interconnection which are cross layered. With multiple path, congestion can still not be avoided. Because mostly data center is designed with much over-subscription. Multiple path can be used to divert traffic towards a single destination. All congestion techniques discussed in this category use Elephant and Mice flow to detect the congestion. They used such traffic types to re-route and scheduling the flow towards a destination.

Depending on hashing and fields of the packet header, Equal-cost multipath (ECMP) [19] is famous multipath routing that can be used to route the traffic for the same destination on multiple paths. However, ECMP does not consider current network condition, statistics and flow type in the scheduling process. Also, same header fields might result in hash collision that may eventually result in forwarding of two different elephant flows on the same path, even if there are other path which are not being used and are idle.

Flow re-routing works as follows. The SDN controller gather the network conditions(statistics) and detects congestion if many elephant flows are traversing the same link. As a result, future or coming flow can be forwarded to other links for the same destination [20–24]. Based on current work load; bit rate ratio of current and maximum of each port of the switch, Congestion Avoidance Algorithm (ACAA) is proposed in [21] where SDN controller detects the congestion. To mitigate the congestion, based on threshold of 70%, the SDN controller checks workload on any port of the switch and installs more flow rules to divert the flows that result in avoidance of congested port. One threshold is used to detect the congestion and one 50% is used to recover from congestion event and recovering of flow rules that were in practice when no congestion was present.

And SDN based and dynamic scheduling of flows is given in [23] for data centers. For todays data centers, it uses Fat-tree topology. Mainly three steps are involved. In the first step, Elephant flows are detected from edge switches in the first step based on global information. Secondly, computation of available path that are non-conflicting is done with placement algorithm. This makes sure that all combined flows would not exceed a particular link capacity. Final step involves all the relevant switches accordingly.

Solution given in [20,21,24] do not differentiate between flow types; mice and elephant. This result in degrading of mice flows due to increased latency. Also, this re-scheduled might not be effective for mice flows. Until flows are re-scheduled, mice flows would have been terminated. Moreover, switch location limit the effectiveness of the approaches. For instance, if ToR switch is congested, which happens in incast case specially, there exist no route that can be used for re-routing. The benefit of such technique include no modification of TCP stack on end hosts.

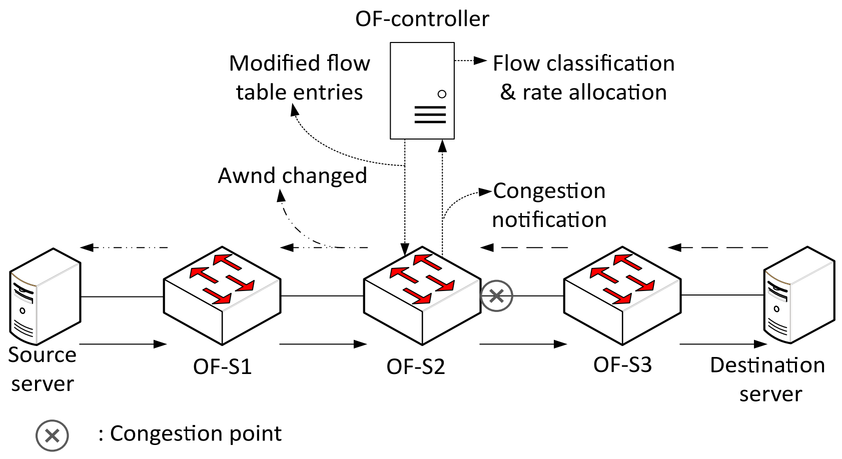

Figure 3.1: with SDTCP, ACK being modified for senders

3.1.4 Modifying TCP Parameters in the Network

In this section, we discuss solution which tweak parameters in TCP's header during the journey of the data packet from source to destination. In TCP's header, the most effective and common parameter available for tweaking is awnd/rwnd used in acknowledgments sent from one host to other host. Since sending rate of the TCP sender is chosen minimum from congestion window(computed from congestion algorithm in practice) and rwnd signaled from receiver side in acknowledgement. Therefore, by replacing rwnd value in the network according to share on the bottleneck link from source to destination, the sender can be throttled.

On such technique presented in [25] is Scalable Congestion Control Protocol (SCCP) that extends OpenFlow to detect the congestion and measures fair share of all flows competing for one output port of the switch. Computed fair share is placed in TCP header in the field of rwnd in ACKs packets in receiver to sender direction.

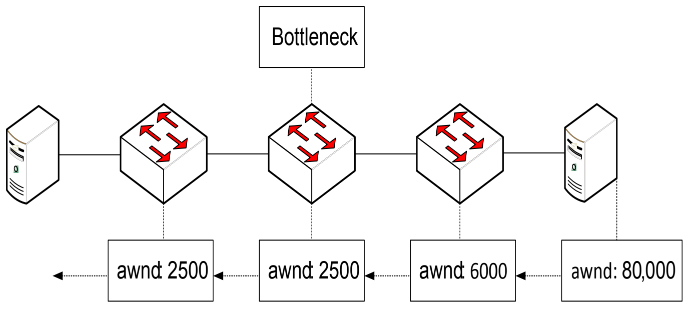

Figure 3.2: An example to demonstrate how bottleneck share is replaced in rwnd using SCCP

To keep record and count the number of flows on a particular switch, SCCP uses packet inspection for TCP flags SYN and FIN. SYN increments and FIN decrement the counter. Using value of number of flows and Bandwidth Delay Product (BDP), the fair share is computed. BDP is defined as product of capacity of the link and RTT(common value flows present in network). Once share is computed, switch compares this value with the rwnd value of every packet. In case rwnd value in ACK packet is more than computed share capacity, switch replaces this value with the computed one. From source to destination, all switch in the path perform this action and eventually when packet reaches sender, it contains minium value of the share of the path from sender to receiver.

The SCCP works well when connection termination take place gracefully, not with the timeout events. Because, connection not being terminated gracefully would require timeout to detect FIN and then algorithm would react, resulting in wastage of resources(bandwidth, buffer capacity). Moreover, it is assumed that ACK would follow the same path as data packet, which

Table 3.1: A Brief Overview of SDN based Congestion Control Schemes

Protocol	**Congestion Detection Method**	**Congestion Detector Location**	**Reactor Location**	**Changes Required**	**Performance Evaluation**
Re-routing [20]	Threshold on transmitted bytes	Controller	Switches	Controller	Simulations (Mininet)
ACAA [21]	Threshold on link utilization	Controller	Switches	Controller	Simulations (Mininet)
Local re-routing [22]	Threshold on link utilization	Controller	Switches	Controller	Simulations (Omnet++)
Hedera [23]	Detection of Elephant flows at edge switches	Centralized scheduler	Switches	Scheduler	Simulations (customized) & Real testbed
DLB [24]	Byte counting at port	Controller	Switches	Controller	Simulations (Mininet)
SCCP [25]	Flow counting at port	Switches	Switches & End hosts	Switches	Simulations (Ns3) & Real testbed
SDTCP [26]	Threshold on queue	Switches	Switches & End hosts	Controller & Switches	Simulations (Mininet)
SED [27]	Threshold on queue	Switches	Controller	Switches	Simulations (Mininet)
RWNDQ [28]	Flow counting and queue monitoring	Switches	Switches & End hosts	Switches	Simulations (Ns2)
OTCP [29]	Network statistics (queue, link rate & latency)	Controller	End hosts	Controller & End hosts	Simulations (Mininet)
OpenTCP [30]	Link utilization	Controller	End hosts	Controller & End hosts	Real Testbed
NSCA [31]	Flow characteristics	Controller	End hosts	Controller & End hosts	None
eSDN [32]	Queue monitoring at Switches	Slave Controller at End hosts	End hosts	End hosts	Simulations (Ns2)
SDN-GCC [33]	Queue monitoring at Switches	Controller	Hypervisor	Controller & Hypervisor	Simulations (Ns2) & Real Testbed
SICCQ [34]	Threshold on queues at Switches	Controller	Hypervisor	Controller & Hypervisor	Simulations (Ns2) & Real Testbed

is not the case in multipath Fat-tree alike topologies. The SCCP can be deployed on switches that are OpenFlow 1.5.0 [35] supported and requires no change of end host(s) TCP stack or network applications.

Prioritzing the mice flows, Software Defined TCP (SDTCP) [26, 27] limits the rate of background elephant flows by modifying TCP rwnd value in switches in the path from source to destination, just like SCCP. However, connection record method and behavior of switch differs from SCCP. Controller is responsible for flow regulation actions in the network; which flows to limit and which ones to throttle.

Another in-network solution that employs the receive window modification is RWNDQ given in [28]. The authors proposed to overwrite the Rwnd values (and the window scale option, if used) in the packet ACKs with bottleneck fair share value of the bandwidth for the flow. The Rwnd is modified if the calculated fair share is less than the current value of the Rwnd. The fair share is calculated by sampling the number of outgoing flows and the target queue occupancy on the forward data path.

3.1.5 Hybrid Congestion Control

In this section, we discuss proposed techniques that require support from the network as well as the end-hosts/hypervisor for their functioning. We have classified protocols of this category in two further sub-categories; *Host-based solutions* and *hypervisor-based solutions*. Host-based solutions include techniques that enable the SDN controller to directly communicate with the host for enforcing rate-limitations. The hosts thus exhibit a modified TCP congestion control behaviour, enabling it to accept notifications from the controller. Hypervisor based solutions leverage the virtualization capabilities of the hypervisor to identify flows and rate control the individual VMs operating on the server.

Involving hosts/hypervisor in exercising congestion control is beneficial because these possess more computational powers. Additionally, hosts have access to complete TCP/IP stack, while the hypervisor has visibility in to VM based flow information that becomes obfuscated once within the network.

3.1.6 Host-based Solutions

Omniscient TCP (OTCP) [29] uses SDN's centralized management capabilities to compute granular TCP congestion control parameters for dealing with TCP incast. These computed parameters are then distributed to all hosts for utilization. The authors suggest that suitable fine-grained TCP retransmission timers matching the network latency should be employed to trigger an early response to packet drops. Also, the initial and maximum congestion window used for that connection should depend on the BDP for that specific host pair. This would prevent the buffer overflow caused by the synchronized flows leading to TCP incast.

OTCP discovers the topology and switch fabric's latency with Openflow Discovery Protocol (OFDP). After latencies in the network are known, controller computes RTT among hosts and set a value for TCP Retransmission Time Out (RTO_{min}) to the minimum value. The upper bound of retransmission time is constrained by switch's queue delay; computed by dividing value with rate of the link, and TCP RTO_{min}. Following equation is used to compute the RTO_{max} between source destination pair.

$$RTO_{max}(H_1 \rightarrow H_2) = RTO_{min}(H_1 \rightarrow H_2) + \sum_{s \in R} \frac{Q_s}{T_s}. \tag{3.1}$$

Q and T demonstrate size of the buffer and rate of the link respectively in the equation. On any route R, s shows a switch between host H_1 and host H_2. Between any two host, lowest link available in route is multiplied with RTT to obtain BDP. This BDP between two hosts, is upper bound for maximum congestion window denoted with ($Cwnd_{max}$). In the presence of more flows, share of each flow should be equal to ($Cwnd_{max}$) divided by number of flows present at any time. Every user would be running a daemon for connecting with the API of the controller and fetching the fine tuned congestion parameters calculated at controller for end hosts. Controller uses a northbound famous API, the JSON/REST. Flow completion time is reduced by mitigating effect of TCP incast with OTCP. However, modifications at kerne level, and time involved in measurement of latency of fabric of switches are costs of technique and significant overhead.

To fin tune the TCP according to network dynamics, authors in [30] have presented a technique called OpenTCP. Application running on SDN controller, the Oracle, gather networks statistics from SDN switches . Based on parameters set by administrator, if congestion is about to happen, Oracle notifies the end hosts which are running a Congestion Control Agent (CCA) about this. This means that when to react, administrator can define certain values and threshold on the controller application, Oracle. While initiating a new TCP session, CCA at end host might opt entire new flavour of TCP according to current network conditions. For performance of TCP, CCA is capable of tuning TCP parameters for the established sessions. This technique requires installation of CCA on all end hosts but requires no changes in hardware switches in the network. Also, statistics collection period, though larger than RTT, bring a lot of overhead.

Network Assisted Congestion Avoidance (NSCA) is another host based solution presented in [31]. The end hosts communicate with network to get the information; occupied and available capacity, with the help of controller. Then end host applies rate control method to limit the rate. Since controller has recent global network statistics, thus, it can tell available data rate for a particular link and flows. Thus, end hosts become aware of the maximum available capacity in the path from source to destination to mitigate congestion event in then network. Controller computes all possible path on which flow can be accommodated with maximum capacity for the desired data rate. This controller to end host communication is UDP based, thus, when no UDP datagram is received, end hosts adapts to traditional mechanism of congestion avoidance.

Supplementing the SDN controller, eSDN framework is proposed in [32] is another approach based on endhost-controller communication which are light-weight and deployed on end host stack. These mini controllers gather network statistics from switches and communicate as slave with master controller residing in network. These mini controller can only query network statistics and are unable to install any flow rule on SDN enabled switches.

Though master controller's load is minimized; no congestion parameter gathering and detection, however, periodic gathering of all network end host mini controller from switches incur a lot of network message sharing and overhead.

3.1.7 Hypervisor-based Solutions

For SDN capable virtualized data centers, Abdelmoniem et al. in [33] proposed a SDN Generic Congestion Control framework (SDN-GCC). Shim layer is used to control the hypervisor in this framework. SDN controller is responsible for signaling the congestion to hypervisor of end host. As a result, hypervisor for achieving throughput, controls the rate of the sender VM. This framework allows using different flavours of TCP in different OS in VM TCP stack. The core idea of this framework is to let the controller take decision on congestion, and hypervisor is there to apply changes required notified from controller. Considering capacity of NIC card, shim layer in the hypervisor allocates bandwidth to all VMs. Allocation of bandwidth is repeated at every message of SDN controller to hypervisor, whenever congestion takes place on any switch. The benefit is that it requires no changing at TCP stack of end hosts.

To predict the incast, technique SDN based Incast Congestion Control through Queue based monitoring (SICCQ) proposed in [34] let the controller to monitor switch resources like queues. TCP flags such as SYN and FIN are counted for a specific time interval on switch. When defined threshold is crossed, it is predicted that incast is going to happen. At this stage, the controller notifies the senders(end hosts) about limiting rate to only 1 MSS. Hypervisor of Virtual Machine (VM), on reception of message of Incast ON from controller, enables the marking of receiving window fields in TCP ACKs packet for a particular VM. This lets senders to limit their rate and consequently, queue at switch starts draining. When controller, periodically with the help of statistics, checks that buffer occupied is less than 20%, it again notifies hypervisor(Incast OFF) to stop marking rwnd in ACK and restores sending rate that was set before Incast ON message. Marking stops as well on expiry of timer which is set to common flow completion time of mice flows.

In contrast with other such SDN based approaches, SICCQ requires a channel for communication between the controller and hypervisor for message of Incast ON and OFF and rwnd value to be modified at hypervisor. It neither requires modification of TCP stack working on end host, nor it requires any changing in the hardware of the switch. To match the TCP flag, it extended OpenFlow. In comparison with SDTCP; which degrades elephant flows, SICCQ does throttle all flows equally which share the same link and switch in the path. Thus, elephant flows are not effected for the throughput.

Chapter 4

Methodology

In this section, we describe proposed algorithm for avoiding TCP incast in data centers using SDN. We describe every major step in form of subsection to better explain it.

4.1 Application for Avoiding TCP Incast

As depicted in figure 4.6, our algorithm consists of several steps and decision steps that are part of controller application. But as pre-requisite, lets elaborate TCP's connection establishment steps depicted in figure 4.1 and briefly described below:

4.1.1 Proactive Flows Installation

Whenever switch connects with any SDN controller, it shares capabilities with SDN controller. In response, controller installs atleast one entry called missed entry. This results in every missed packet sent to controller. Using this approach, controller firstly installs another flow(in addition to default) rule based on TCP flag SYN. It sets only flag matching in this flow rule. The action given in this rule is NORMAL, which means there will be no packet-in for initial packet. Controller installed this rule to make communication faster since we are only interested in ACK of this SYN. Our goal is to get number of servers a particular server is going to connect. This should be done with minimum communication between switch and the controller. If we would have installed flow rule for getting SYN packet at controller, we still are required to get subsequent packet to run our algorithm. Hence, SYN packet is not of our interest and forwarded with NORMAL pipeline processing, demonstrated in figure 4.2.

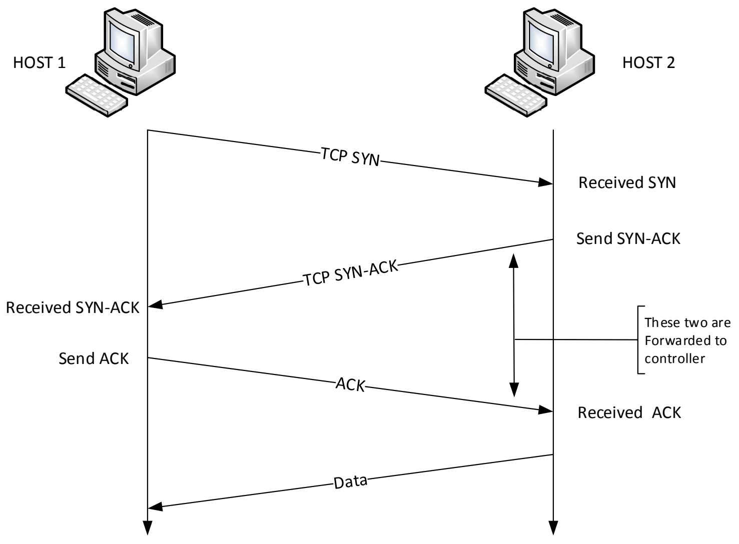

Figure 4.1: Three way handshake

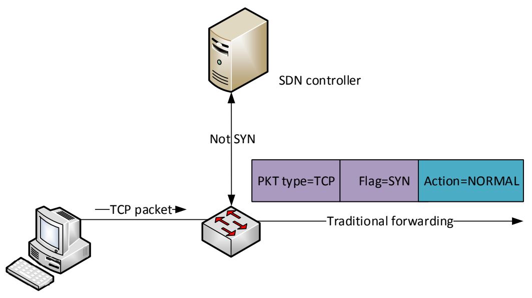

Figure 4.2: Flag with normal processing

4.1.2 Server Count with SYN-ACK

When host(s) get SYN packet forwarded from NORMAL processing, they reply to requesting server with SYN-ACK as depicted in figure 4.1. Now, no rule exists that could match SYN-ACK packet, hence this packet would be forwarded to SDN controller. To make our algorithm computationally efficient, this is done only on Top of Rack switch (ToR). All remaining switches forward and install flow rule based on application of normal layer-two forwarding mechanism. Note that, any other routing mechanism can also be used instead of normal layer-two forwarding. When switch is ToR, controller increases number of server to requesting host to 1. This record of connection will later provide us help in computing and share buffer capacity among all SYN-ACK senders.

4.1.3 Final ACK In reply to SYN-ACK

Now the requester would send final ACK to sender of TCP SYN-ACK. Again there exist no flow rule for this side, hence this packet would be forwarded to controller by ToR switch.The algorithm from previous packet has learned that there exist on expected sender which is going to send data to requesting server. At this time, there exist only one sender and no modification is done to window, so as to throttle it at full rate.

If there were more than zero senders and less than threshold(, say n, controller would have assigned $\frac{Th1_{buffer}}{n}$ of the buffer value to each sender. $Th1_{buffer}$ is limit where buffer is assigned any other value except 1 and 0. Once value is computed, it is installed in flow rule. Now every packet that has these match field, its Rwnd field is marked at switch. It is worth mentioning that this marking of Rwnd only takes place at ToR switch of requester. Remaining switch and final receiver of this modified packet would not know that ToR switch has modified the rwnd value in TCP ACK header.

4.1.4 Installation of FIN rule

When controller computes and installs flow rule for rwnd modification, it also installs flow rule based on TCP FIN flag for the same direction, as depicted in figure 4.3. Resultantly, when requester is finished with data receiving, It would send FIN-ACK for the sender server. Again note that this rule is installed on only ToR switch which would also minimize communication time. Once FIN packet leaves ToR switch it would be forwarded by data plane in all subsequent switches until it reaches sending server.

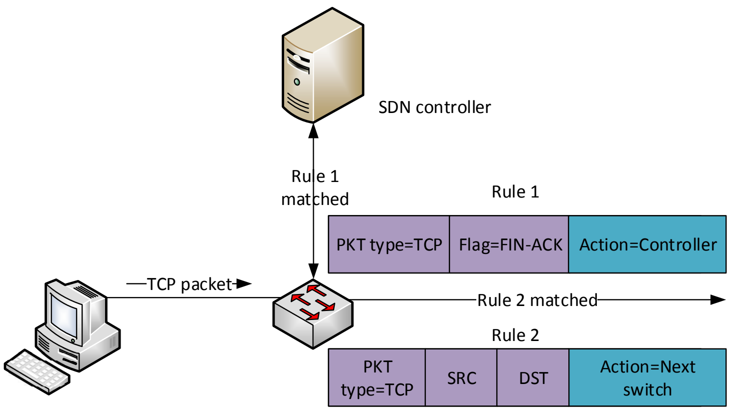

Figure 4.3: FIN as a match field

4.1.5 Marking of rwnd at ToR of requesting server

Once both rules are installed i.e, one with rwnd and one with FIN, TCP header field(rwnd) would be modified on ToR switch only. As demonstrated in figure 4.4, host A is requesting server and the switch in circle is its ToR switch. TCP window tweaking only takes place at this switch even it is communicating with the host E, 5 hops away.

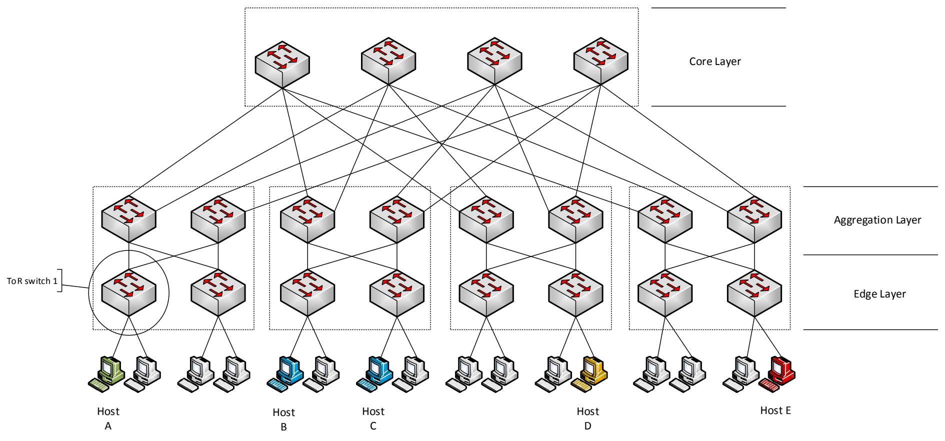

Figure 4.4: Typical Fat tree topology

4.1.6 Division of Buffer

Now let's assume there comes another final ACK from data requesting sender. At this time, buffer of is totally reserved and assigned to server to whom we assigned on first ACK reception. Now we have to divide assigned buffer among new server and the first one. For this reason, buffer value assigned to first server will be retrieved back from local database, and again share is computed. Now share becomes half of that value which was assigned before. Now rule for new ACK would be installed with computed share of buffer and relevant FIN rule would be installed as described in section 4.1.4

4.1.7 Single Flow modification

One of the key feature of OpenFlow is it provides functionality to modify the flow which has been installed earlier. One can easily modify action as well. This is done by providing match fields in the FLOW-MOD(flow modification) message. The matching flow rule in the table being modified would be changed according to new actions. Since we have to change rwnd of the previously installed flow, thus, we have to use distinguish match fields that would match only one single flow rule we desire to modify.

4.1.8 The threshold $th1_{buffer}$ and threshold $th2_{buffer}$

Entire buffer of any output port is managed with two different values of buffer. First one is $th1_{buffer}$ for which value is 75% of the entire buffer. Second buffer is $th2_{buffer}$, 95% of the entire buffer. Remaining is reserved for other protocols traffic, like ARP UDP etc, although 95% of the traffic in data center is TCP. If current value is less than $th1_{buffer}$, this means that congestion is not expected yet and window can be assigned. One the other hand, if value of current buffer of any output port is equal or greater than $th1_{buffer}$, we mark this point as medium level congestion and mark new sender's window as 1 MSS (1460 Bytes). We keep marking window as 1 MSS until $th2_{buffer}$ is reached. This triggers marking of 0 MSS in rwnd of all subsequent packets.

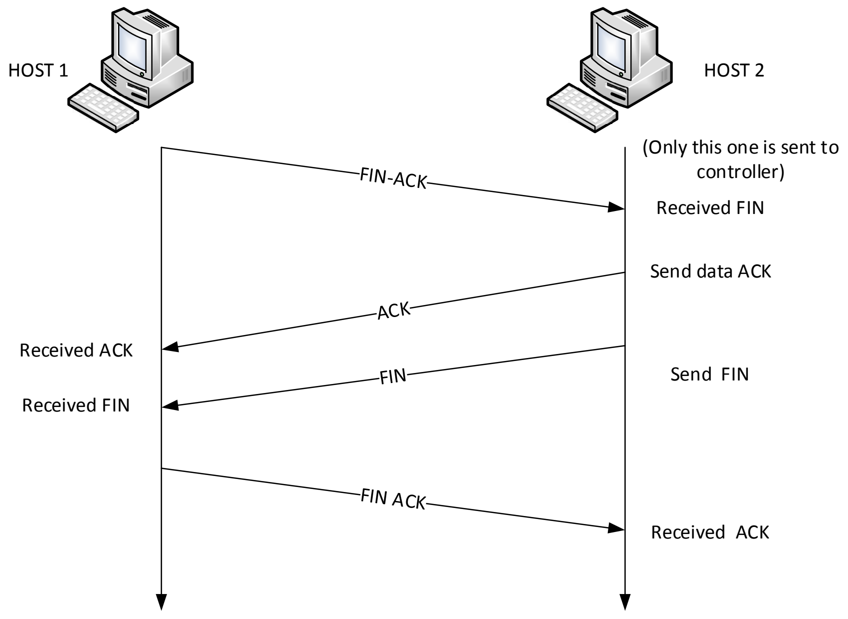

Figure 4.5: Connection termination

4.1.9 Reception of FIN-ACK packet

When requester is finished with the sender host, it sends FIN-ACK to terminate the connection, shown in figure 4.5. Since we have installed FIN rule and action that this particular packet must be sent to controller. This decision was to release the buffer say B_r, assigned for this particular host and divide among other waiting either in Q0, Q1, and Qx. If rwnd assigned to host with which connection is being terminated is more than 1 MSS, priority would be given to Q0 because host in Q0 are not able to transfer any data. Thus, we assign first host of Q0 as 1 MSS and put it to Q1. Now remaining buffer is assigned to first host of Q1(who's previous rwnd value was 1 MSS), now value has become B_r-1 MSS. Relevant flow modification messages are sent to only one ToR switch.

4.1.10 Combined flow modification

We have set maximum number of *senders* that would be assigned rwnd more than 1 MSS at any time. We assign all such sender same cookie value, say X. On reception of FIN, if Q1 and Q0 are empty, now we have to distribute released value among all competing flow in Qx. For this purpose we only have to send a single flow modification message irrespective of number of senders for those flow has been installed. We use cookie value X and new action(rwnd) to modify the flows.

4.2 TCP Flags and RWND in OpenFlow

Support of TCP flags is not directly provided in OpenFlow. TCP flags as match fields are provided by Nicira in their extension to OpenFlow. On the other hand, TCP rwnd modification is yet not supported by any means in OpenFlow itself and its implementation in OpenFlow switches. Therefore, changes in both are required to achieve this functionality. We modified Open vSwitch for the modification of TCP rwnd value.

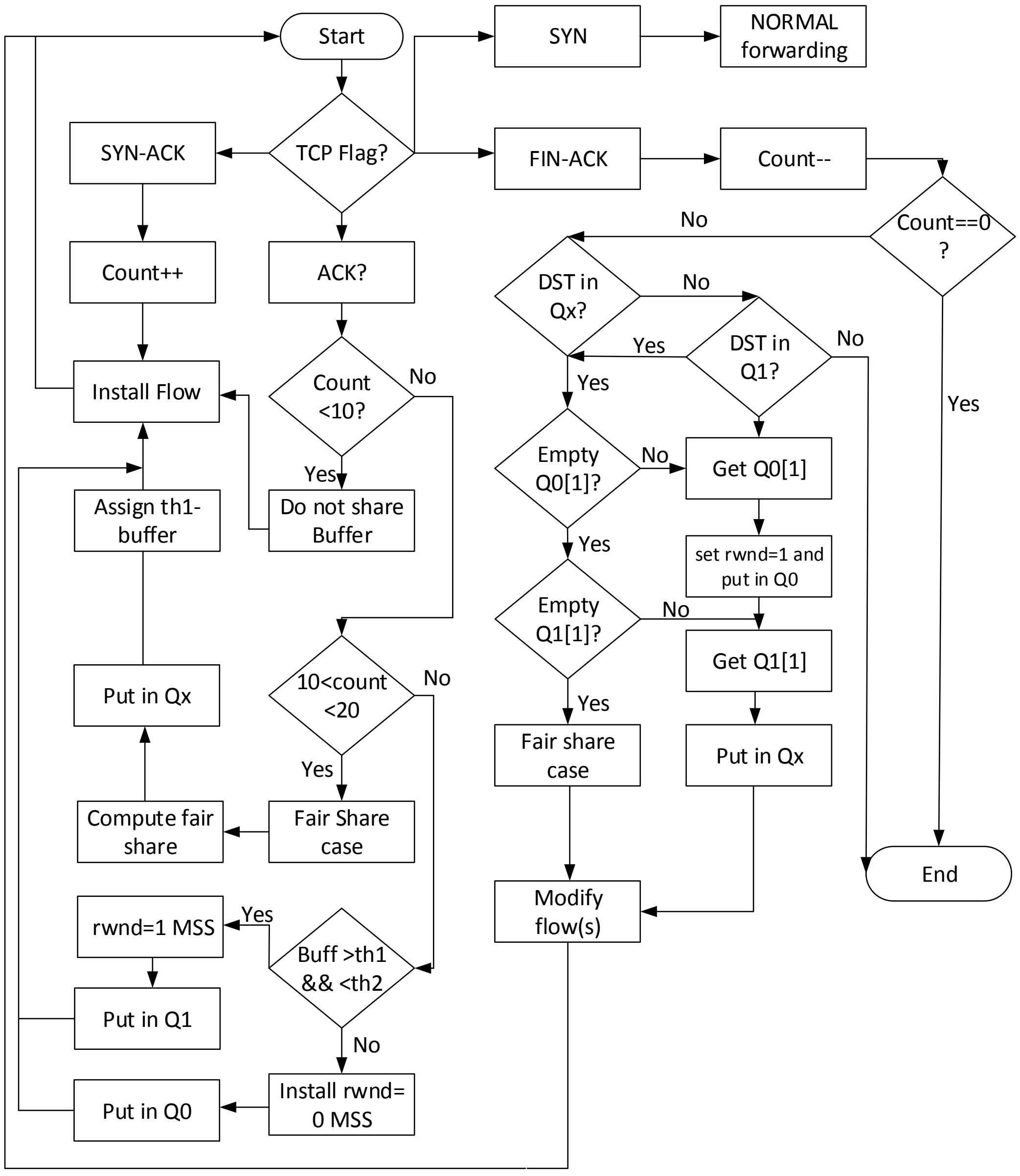

Figure 4.6: Flow chart of the proposed algorithm

Chapter 5

Evaluation

In this section we evaluate performance of proposed algorithm. We have evaluated two cases. First, when algorithm reacts from very first sender, mentioned as Case 1 hereafter. Second is when until 10 number of senders, algorithm does not work and starts working when servers exceed 10, mentioned as Case 2 hereafter. We have discussed Throughput, Flow Completion Time (FCT) and Request Completion Time (RCT) in evaluation, though number of retransmissions are depicted as well but are not discussed separately.

5.1 Why Mininet

We choose Mininet due to its following advantages:

- It allows creation of custom topologies and provides GUI that can export python scripts of generated topology automatically.
- Every host in this run actual Linux kernel under the hood which makes us easy to run program/applications on particular host(s).
- One can use either default switch(Open vSwitch) or modified switch easily
- It has support for SDN, for instance switches are capable of OpenFlow protocol and it has support for several controllers (POX, RYU, OpenDaylight etc.).

5.2 Data center topologies

There exist number of topologies for different types of data centers. However, most common which provides multiple paths from one host to other is Fat

Tree topology which is depicted in figure 5.1. There are mainly following reasons due to which this topology has gained attention and importance in deployments

- It consists of multiple paths from one host to other
- It uses cheap switches
- Oversubscription can be adopted

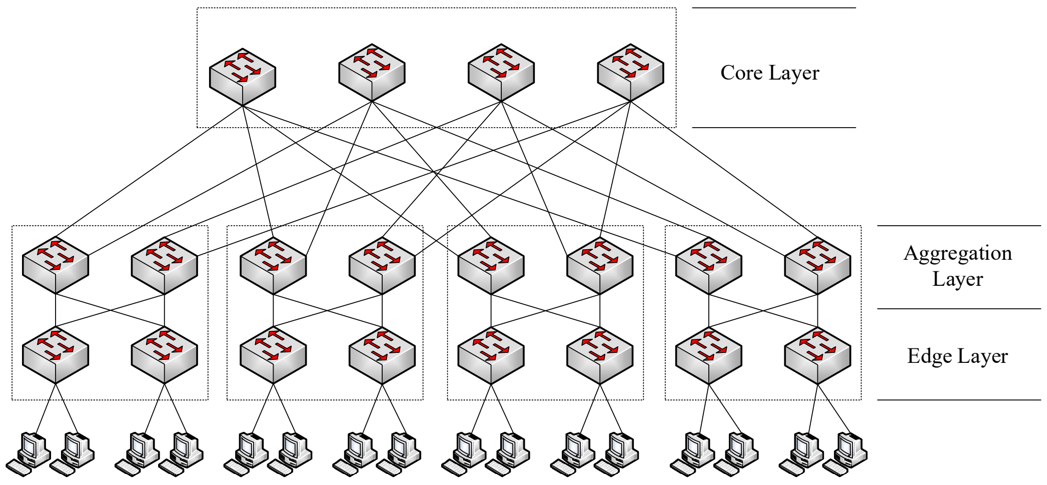

Figure 5.1: Fat tree topology

5.3 Emulation environment

We are using Linux OS, and Mininet emulator. Switch we are using is Open vSwitch which is software opensource available switch and has mostly all implementation of features of OpenFlow. Its new version are available but we are using version 2.6. Fat-tree forms redundant links, so we have enabled Rapid Spanning Tree Protocol (RSTP) to enable forwarding in loop topology. We have listed important environment setup parameters in table 5.1.

5.3.1 Incast experiment

We have generated traffic which could result in Incast like events. This setup is shown in table 5.2.

Table 5.1: Experimental setup

Category Name	Specification
CPU	Intel Core (TM) i7-2600K CPU 3.4 GHz
RAM	8 GB - DDR4
OS	Ubuntu 14.04.01 LTS
Emulator	Mininet 2.3.0d1 [36]
South bound API	OpenFlow v1.3 [35]
Switch	Open vSwitch 2.6 [37]
Controller	RYU [38]
Traffic generator	Empirical traffic generator [39]
Packet analyzer	Wireshark [40]

Table 5.2: Incast traffic generation setup

Parameter	Values
SRU	1MB
Buffer	256KB
Servers	1-30
Packet size	1500B
Link	1Gbps
Number of Runs	4

5.3.2 Effect of Incast on throughput

Case:1

Major problem that incast incurs is throughput collapse. The fundamental reason lies behind the basic concept of concurrent sending behaviour of large number of TCP hosts to one host. Shallow buffer switches; which consist of KBs of buffer available per port, can not accommodate large number of TCP senders, especially when they use IW=10 MSS. Therefore, we plot TCP's throughput against number of concurrent server involved in communication. It can be seen clearly that, maximum throughput can be achieved if number of concurrent server are minimized. For instance, figure .5.2 reveals that if servers are upto 10, for 1 Gbps link can provide as much as 700-800 Mbps of throughput. However, following the trend also tells that at 15 servers, throughput has been degraded to 600 Mbps. The culprit for this downfall is

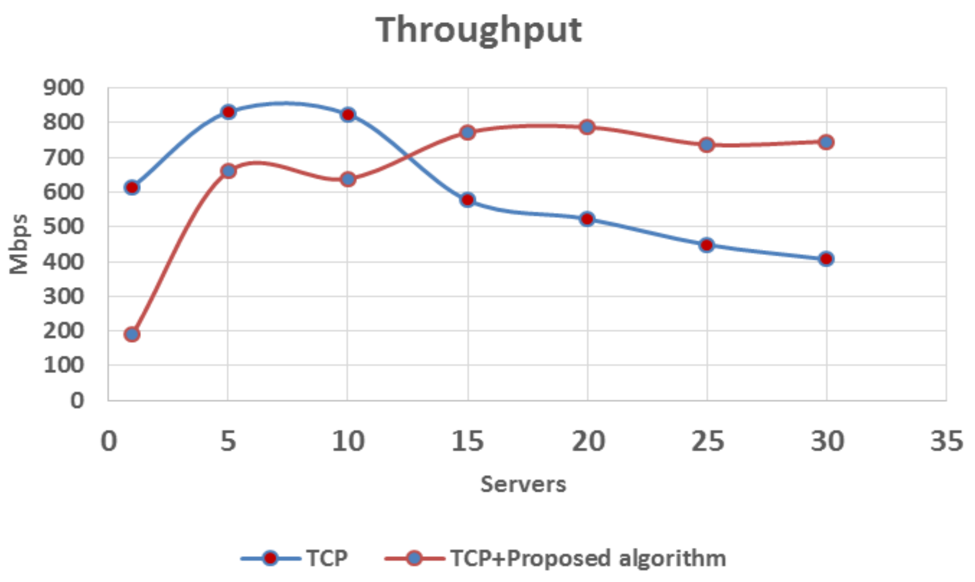

Figure 5.2: Throughput comparison for case 1

no other than entire window drop of TCP senders when incast takes place on port towards the receiver. Considering worst case situation, when number of senders are 30, throughput has been reached to less than 400 Mbps. This trend is being followed only in case of TCP. When proposed algorithm is being used in the network, it can be seen that even at 30 servers almost 750Mbps is achieved. Since we are avoiding incast in any case(number of servers), therefore, throughput of our algorithm is less than TCP alone. In that case TCP window is not being changed, hence it is throttled at full rate. If we see that in figure 5.4, TCP alone is therefore dropping more packets because it is throttled at full rate.

Case:2

In this case, we have set number of servers equal to 10 for which our algorithm would not affect ongoing transmissions. Therefore, it is shown in figure 5.3 throughput of our proposed algorithm is closed with throughput of TCP along. When number of server crosses 10, it is clear that algorithm outperforms the TCP. The reason behind is efficient allocation of buffer capacity marked by ToR switch where incast could occur. TCP is giving full IW=10 at all number of servers, hence throughput is drastically decreasing with the increase in number. On the other hans, our proposed algorithm is tweaking number of bytes in TCP's header according to number of servers involved in communication at any stage.

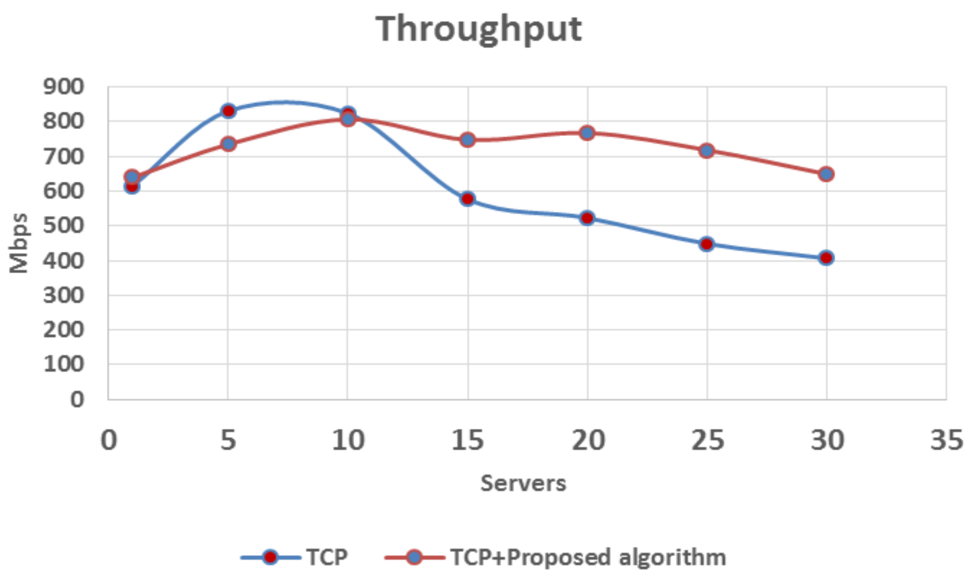

Figure 5.3: Throughput comparison for case 2

5.3.3 Flow Completion Time (FCT)

Case: 1

Flow completion time is defined as, the time aggregator takes for receiving data from the other server is called flow completion time. The data discussed in this context in literature is called Server Requested Unit (SRU). Most of the time, its size is from Bytes to KBs. We have set size of SRU as 1MB in this experimental setup. It can be seen from the figure .5.6, 5.5, 5.7 that number of senders involved in communication are directly proportional to flow completion time. This is obvious because, more senders means every senders will have less share on the Incast port. Consequently, more number of RTTs are required to complete the transmission of data. Consider that IW-10 is adopted and 1 MSS is 1500 Bytes. Therefor IW=10 would transmit 15KB of data in the first RTT. If 30 number of server would transmit this much data, 450KB of data would reach output port concurrently. Shallow buffer's capacity is by far less than this value. Flow's packet would eventually get dropped. Incast's severe case can also occur in which entire window gets drop. In this case, TCP's three duplicate ACK can not rescue the sender. TCP has no other option than waiting for RTO= 200 $\mu seconds$. If this has

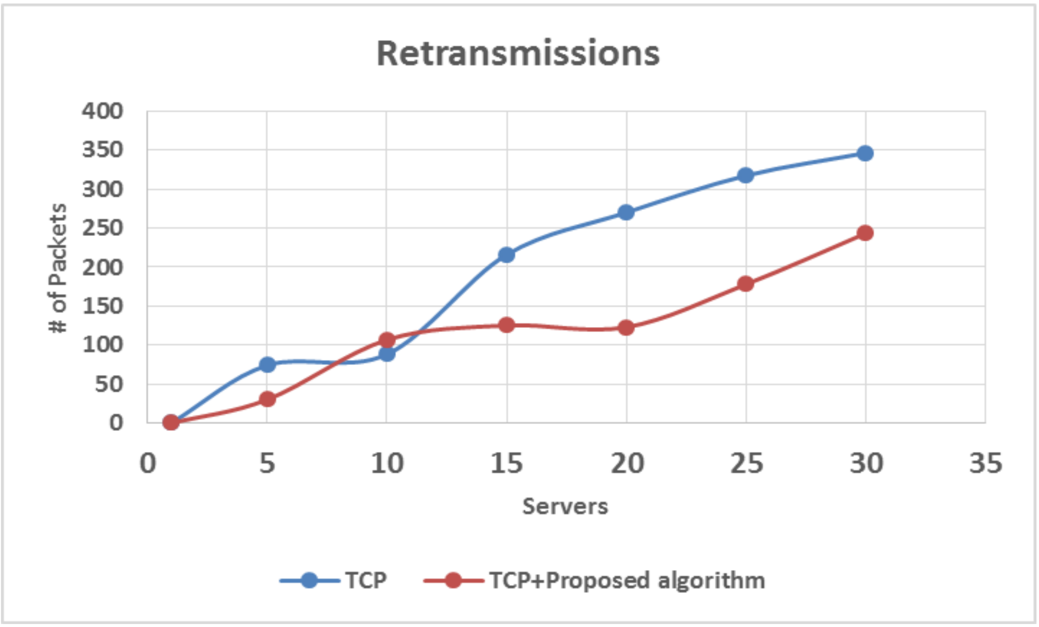

Figure 5.4: Number of retransmissions comparison for case 1

to be done in every window, it drastically increases flow completion time. So therefore, flow completion time either depends on number of servers involved or size of the SRU. It is seen that average FCT of proposed algorithm is almost equal to TCP alone. Due to entire window drop, it can be seen that maximum FCT is increased incase of TCP. On the other hand, proposed algorithm is not letting drop as much packets, resulting in less number of RTO and thus a few retransmissions. Hence, better maximum FCT.

Case: 2

In this case, since we do not act until number of senders are 10, therefore FCT in this is increased for our algorithm and has resulted in close value with TCP alone. But as long as number of servers are increased in 5.8, we see that FCT of our algorithm is better than TCP. Reason behind is same as better management of buffer capacity at ToR switch where all servers are competing for output port buffer towards aggregator.

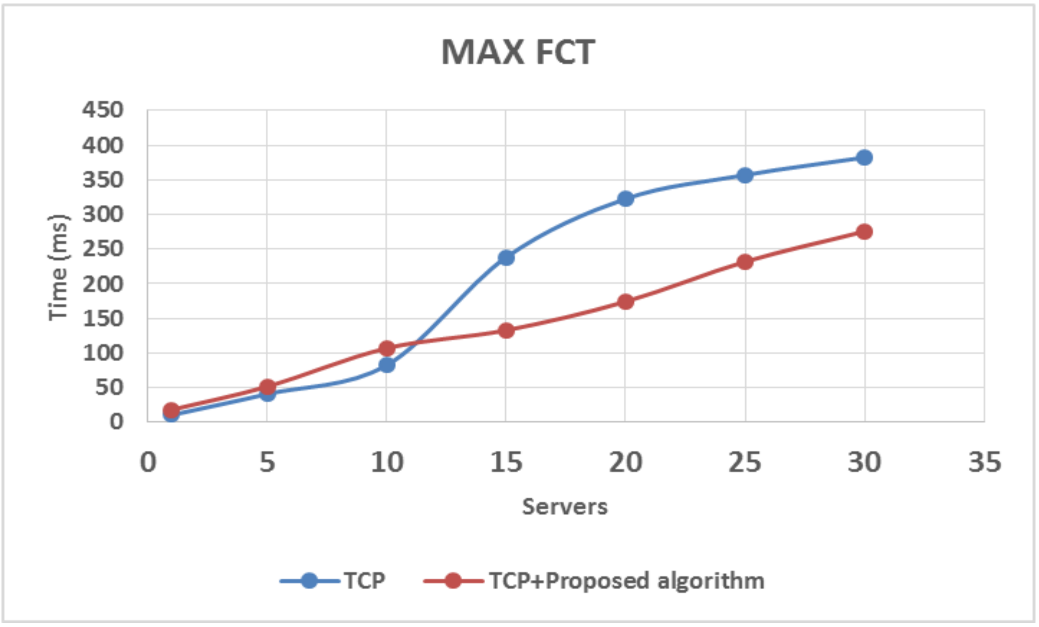

Figure 5.5: Maximum flow completion time for case 1

5.3.4 Request Completion Time (RCT)

Case: 1

RCT is defined as when all servers are finished with sending data to aggregator server. In other words, it is the value of last server finished its data sending process. It can be observed in figure 5.11 that in case 1, when we are reacting even at one sender, our RCT is slightly more than TCP. Because TCP is sending at full rate and results in small RCT value. But as long as servers are increased, we see that our algorithm efficiently allocating buffer capacity to servers minimizes RCT time with considerable margin.

Case: 2

As we have discussed in previous section that, our algorithm starts working incase 10 servers are communicating with servers, thus until 10, RCT is same as TCP in this particular case. But at increased number of senders, all server compete for sharing, therefore, all get affected and drop packets accordingly at buffer. This is also clear in figure 5.13. On the other hand, our algorithm solving incast issue, only allows that much number of bytes that can be accommodated in buffer, thus less number of retransmissions. This means that combined, all server would take less time finishing flows and hence, RCT is improved.

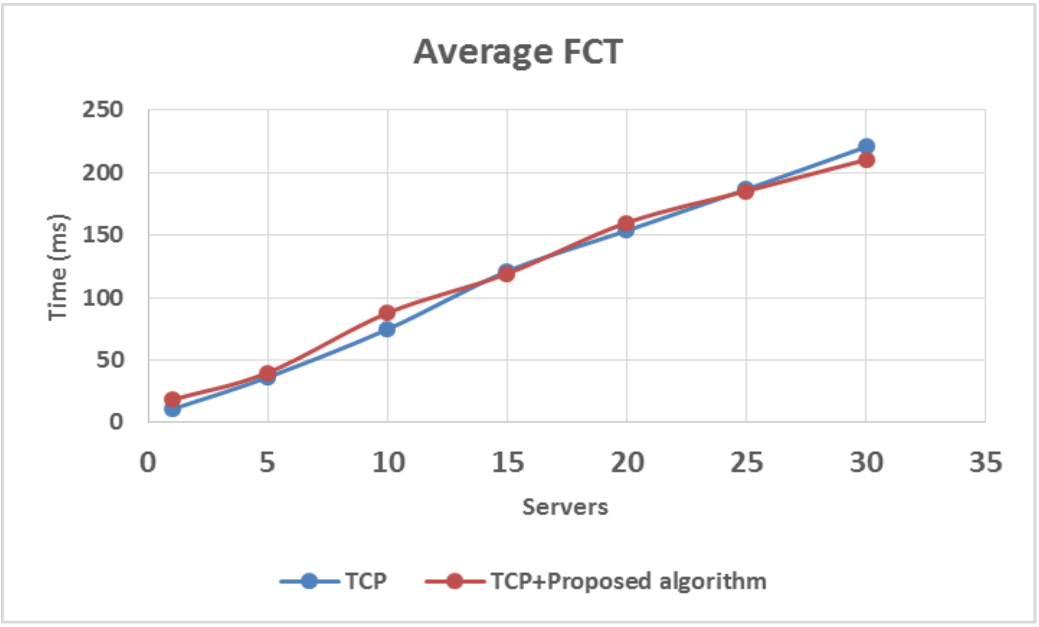

Figure 5.6: Comparison for average flow completion time case 1

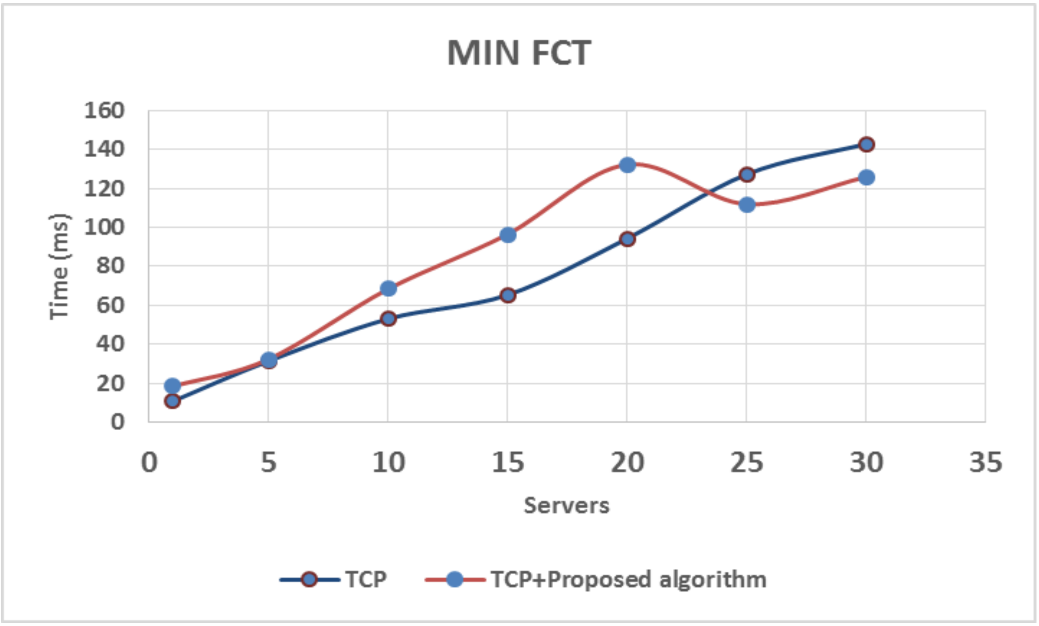

Figure 5.7: Minimum flow completion time comparison case 1

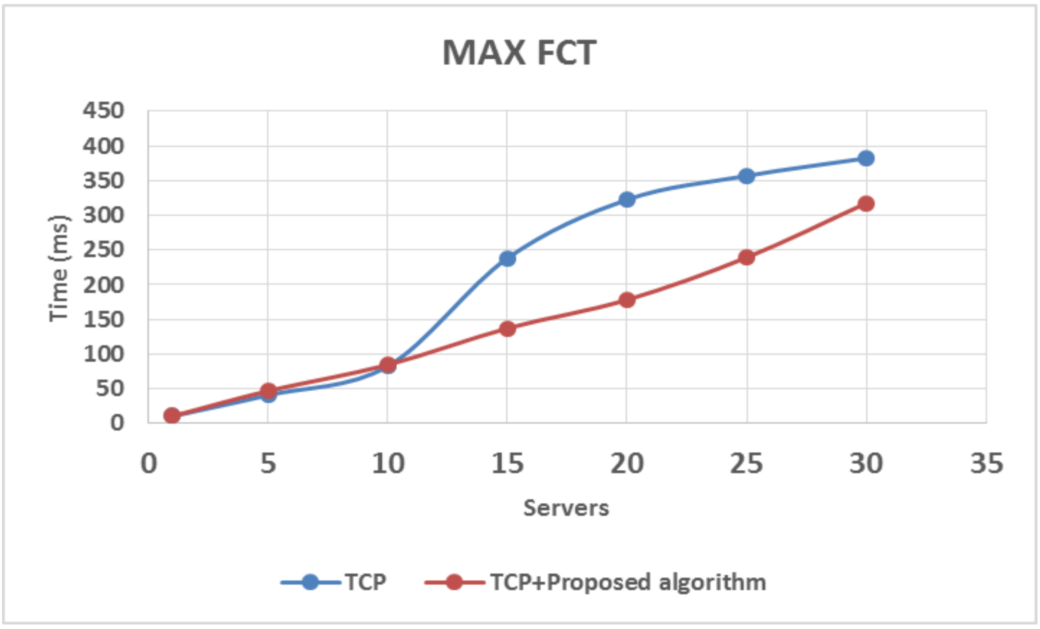

Figure 5.8: Maximum flow completion time for case 2

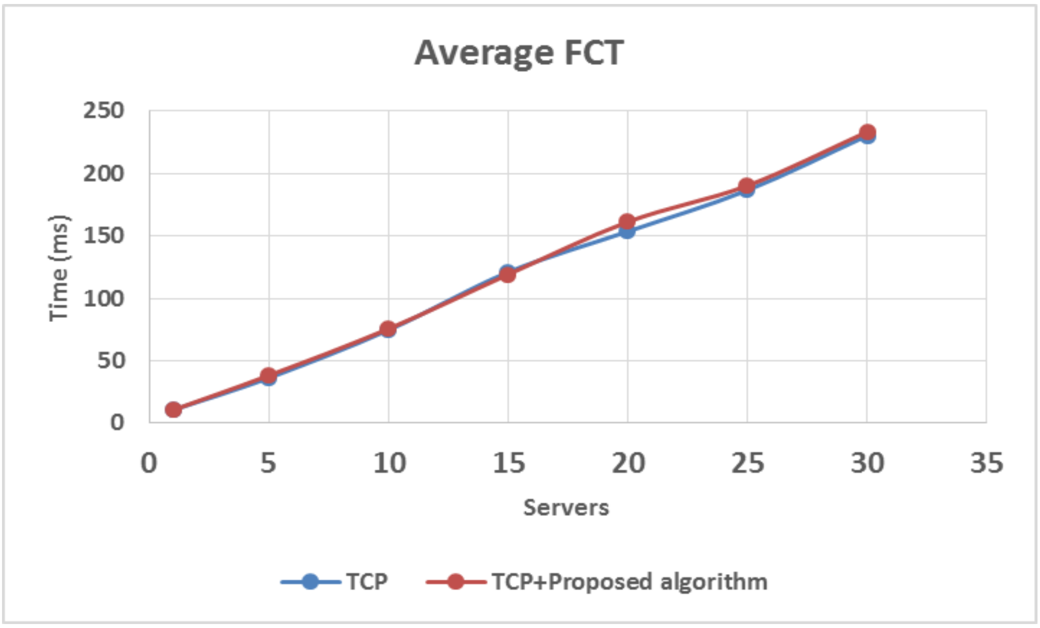

Figure 5.9: Comparison for average flow completion time case 2

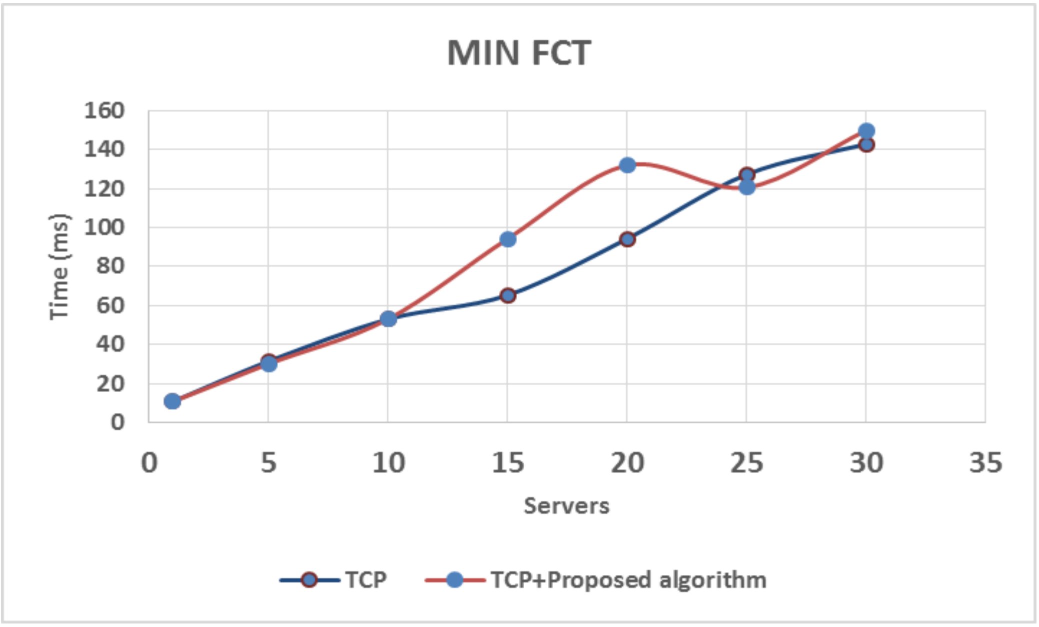

Figure 5.10: Minimum flow completion time comparison case 2

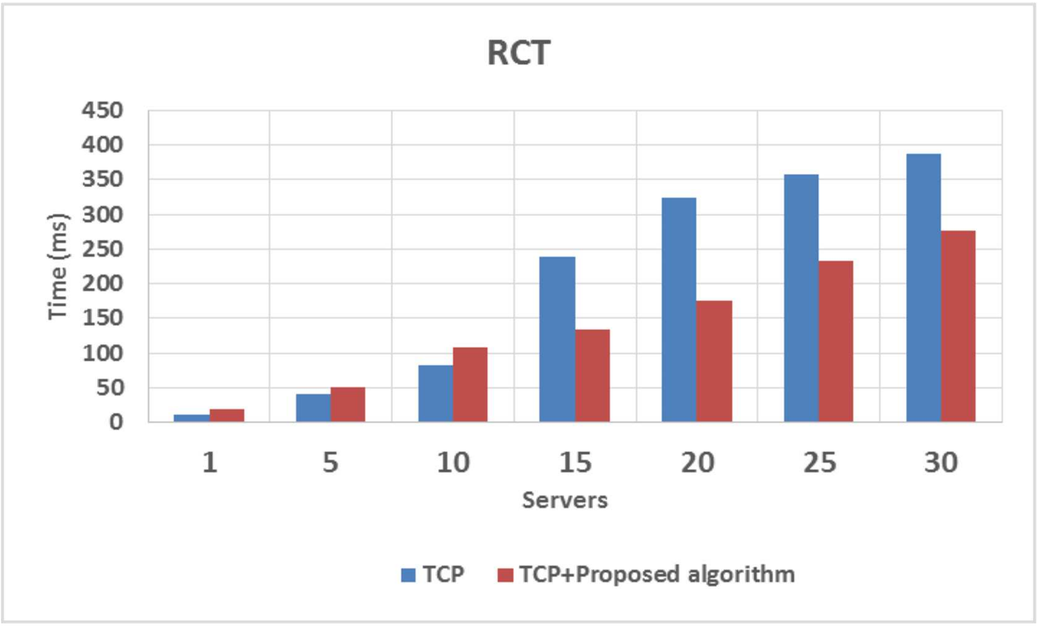

Figure 5.11: Request completion time comparison for case 1

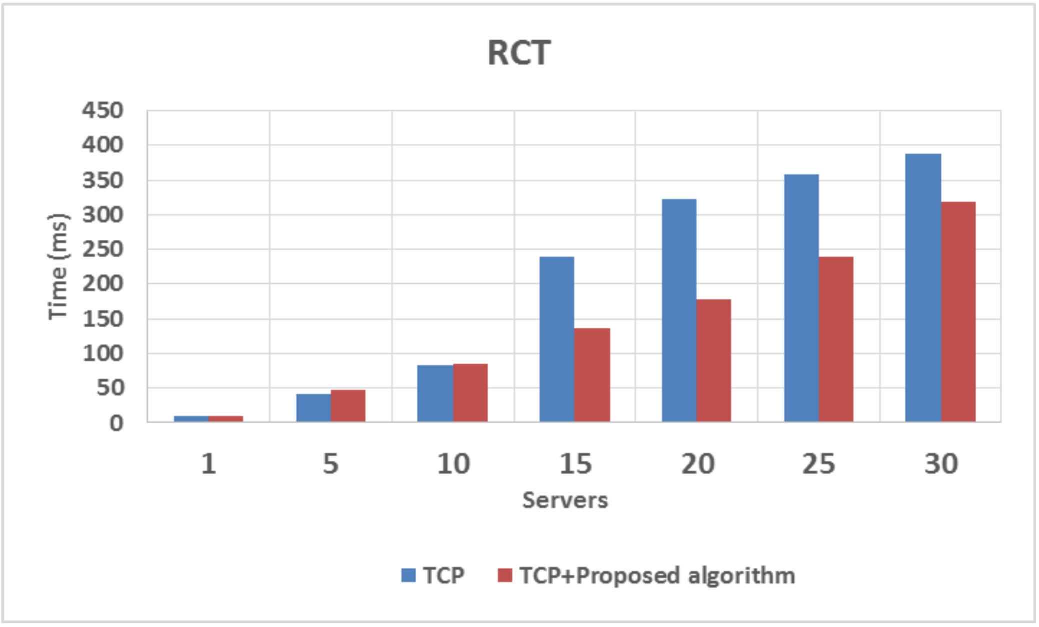

Figure 5.12: Request completion time comparison for case 2

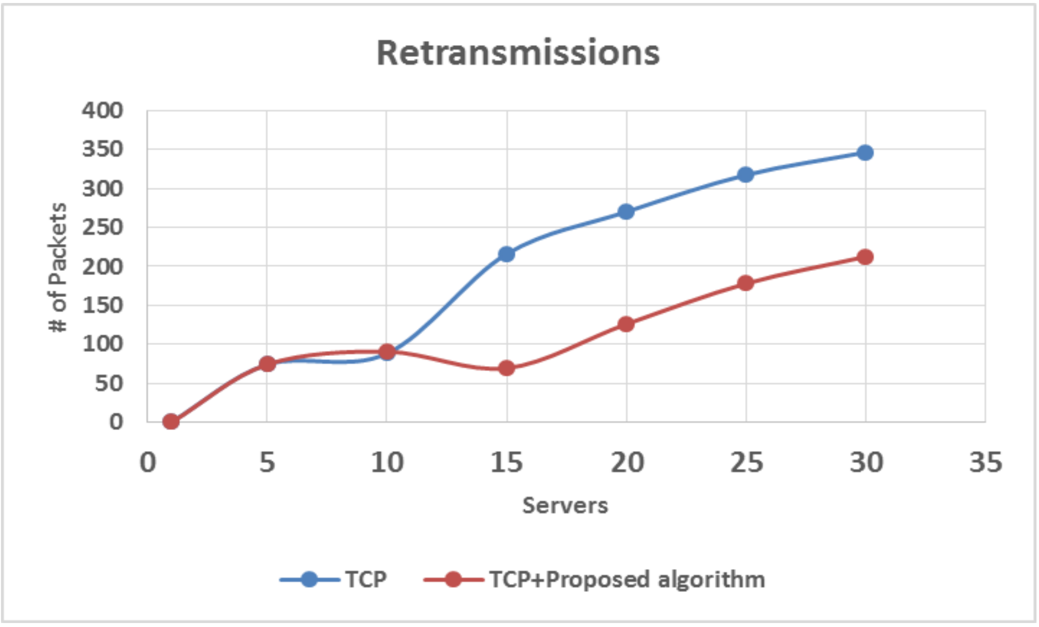

Figure 5.13: Number of retransmissions comparison for case 2

Chapter 6

Conclusion and future work

In this thesis, we unfolded the problem of TCP's IW-10 which takes place in data centers environments. We investigated its effects on TCP incast issue in data centers. In addition to this, we use SDN approach to mitigate this issue at the cost of controller processing. We argue that rather than buying number of intelligent, therefore expensive network devices, in data centers we could use cheap and dumb devices and we can run a number of applications (congestion avoidance in our case) on top of high computation capable SDN controllers, providing more control to network administrators to manage and modify the network configuration with minimum human intervention. Our incast avoiding SDN application manages buffer of the switch port and does the job on ToR switch only, further reducing the communication overhead between controller and switches. Note that we neither propose new TCP, nor we modify TCP's implementation on host. The only assumption we set is that switches are capable of modifying rwnd value in TCP's header. However, our this modifications is required only at ToR switches. Results of simulations show that approach is effective in managing buffer of output ports towards requesting server. Consequently, instead of recovering from incast, our scheduling and buffer management approach can avoid it occurrence.

Machine Learning (ML) techniques have recently being widely adopted in various fields. To the best of my knowledge, there is minimum work done on usage of ML for detection and mitigation of congestion in SDN enabled data centers. This is a reason for adoption of ML in data center environments; ML based method, based on dataset on historic basis, learn and predict the event that could occur. In data centers, their is alot of repition of flows or data being replicated on multiple servers. So ML can efficiently predict the flows based on previous and current flows and scheduler can schedule the flows accordingly. Such work is presented in [41] under the name of Knowledge-defined networking paradigm.

Bibliography

[1] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[2] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 183–197, 2015.

[3] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM computer communication review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[4] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 92–99, 2010.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference.* ACM, 2009, pp. 202–208.

[7] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement.* ACM, 2010, pp. 267–280.

[8] P. Sreekumari and J.-i. Jung, "Transport protocols for data center networks: a survey of issues, solutions and challenges," *Photonic Network Communications*, vol. 31, no. 1, pp. 112–128, 2016.

[9] G. Judd, "Attaining the promise and avoiding the pitfalls of tcp in the datacenter." in *NSDI*, 2015, pp. 145–157.

[10] D. Nagle, D. Serenyi, and A. Matthews, "The panasas activescale storage cluster: Delivering scalable high bandwidth storage," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 53.

[11] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *ACM SIGCOMM computer communication review*, vol. 40, no. 4. ACM, 2010, pp. 63–74.

[12] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding tcp incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*. ACM, 2009, pp. 73–82.

[13] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella, "The tcp outcast problem: Exposing unfairness in data center networks," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 30–30.

[14] "Open networking foundation (onf)," https://www.opennetworking.org/, accessed: 2017-10-09.

[15] Y. Kiriha and M. Nishihara, "Software-defined networking: The new norm for networks, 2012," *IEICE transactions on communications*, vol. 96, no. 3, pp. 713–721, 2013.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[17] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "Measurement and analysis of tcp throughput collapse in cluster-based storage systems." in *FAST*, vol. 8, 2008, pp. 1–14.

[18] G. Jereczek, G. L. Miotto, D. Malone, and M. Walukiewicz, "A lossless switch for data acquisition networks," in *Local Computer Networks (LCN), 2015 IEEE 40th Conference on.* IEEE, 2015, pp. 552–560.

[19] C. E. Hopps, "Analysis of an equal-cost multi-path algorithm," 2000.

[20] M. Gholami and B. Akbari, "Congestion control in software defined data center networks through flow rerouting," in *Electrical Engineering (ICEE), 2015 23rd Iranian Conference on.* IEEE, 2015, pp. 654–657.

[21] S. Song, J. Lee, K. Son, H. Jung, and J. Lee, "A congestion avoidance algorithm in sdn environment," in *Information Networking (ICOIN), 2016 International Conference on.* IEEE, 2016, pp. 420–423.

[22] R. Kanagevlu and K. M. M. Aung, "Sdn controlled local re-routing to reduce congestion in cloud data center," in *Cloud Computing Research and Innovation (ICCCRI), 2015 International Conference on.* IEEE, 2015, pp. 80–88.

[23] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *NSDI*, vol. 10, 2010, pp. 19–19.

[24] Y. Li and D. Pan, "Openflow based load balancing for fat-tree networks with multipath support," in *Proc. 12th IEEE International Conference on Communications (ICC13), Budapest, Hungary*, 2013, pp. 1–5.

[25] J. Hwang, J. Yoo, S.-H. Lee, and H.-W. Jin, "Scalable congestion control protocol based on sdn in data center networks," in *Global Communications Conference (GLOBECOM), 2015 IEEE.* IEEE, 2015, pp. 1–6.

[26] Y. Lu, Z. Ling, S. Zhu, and L. Tang, "Sdtcp: Towards datacenter tcp congestion control with sdn for iot applications," *Sensors*, vol. 17, no. 1, p. 109, 2017.

[27] Y. Lu, "Sed: An sdn-based explicit-deadline-aware tcp for cloud data center networks," *Tsinghua Science and Technology*, vol. 21, no. 5, pp. 491–499, 2016.

[28] A. M. Abdelmoniem and B. Bensaou, "Reconciling mice and elephants in data center networks," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on.* IEEE, 2015, pp. 119–124.

[29] S. Jouet, C. Perkins, and D. Pezaros, "Otcp: Sdn-managed congestion control for data center networks," in *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*. IEEE, 2016, pp. 171–179.

[30] M. Ghobadi, S. H. Yeganeh, and Y. Ganjali, "Rethinking end-to-end congestion control in software-defined networks," in *Proceedings of the 11th ACM Workshop on Hot Topics in networks*. ACM, 2012, pp. 61–66.

[31] J. Gruen, M. Karl, and T. Herfet, "Network supported congestion avoidance in software-defined networks," in *Networks (ICON), 2013 19th IEEE International Conference on*. IEEE, 2013, pp. 1–6.

[32] H. A. Pirzada, M. R. Mahboob, and I. A. Qazi, "esdn: Rethinking datacenter transports using end-host sdn controllers," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 605–606.

[33] A. M. Abdelmoniem and B. Bensaou, "Sdn-based generic congestion control mechanism for data centers: Implementation and evaluation," *CSE Dept, HKUST, Tech. Rep. HKUST-CS16-02*, 2016.

[34] ——, "Sdn-based incast congestion control framework for data centers: Implementation and evaluation," *CSE Dept, HKUST, Tech. Rep. HKUST-CS16-01*, 2016.

[35] "Openflow switch specification version 1.5.0," https://www.opennetworking.org/, accessed: 2017-10-09.

[36] "Mininet network emulator," http://www.mininet.org, accessed: 2018-05-26.

[37] "Open vswitch," http://www.openvswitch.org/, accessed: 2018-05-26.

[38] "Ryu," http://ryu.readthedocs.io/en/latest/, accessed: 2018-05-26.

[39] "Empirical traffic generator," http://www.github.com/datacenter/empirical-traffic-gen, accessed: 2018-05-26.

[40] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[41] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett *et al.*, "Knowledge-defined networking," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, pp. 2–10, 2017.