Virtual DataCenter Network using Mininet

CSL724 Project Report

Sudarshan Maurya
2013MCS8365
mcs138365@iitd.ac.in

Naveen Kumar Tiwari
2013MCS2566
mcs132566@iitd.ac.in

Alind
2013MCS2542
mcs132542@iitd.ac.in

February 20, 2015

1 Abstract

As per the technology grows there is requirement of large datacenters to serve but these datacenters are very huge and highly complex. There is some typical problems in datacenters of Incast which is inverse of broadcast, where large number of hosts are requesting to the same network. A lot of research is going on to simplify and manage datacenters more efficiently to get more reliable and efficient datacenters. Mininet is an emulator which is used to emulate network on a single computer system which can help researcher to emulate network of heterogeneous node and customize as per need.

In the project we are going to emulate datacenters using Mininet over a single system and demonstrate the Incast problem, where lots of hosts are sending data packet to a single node And due to congestion some data packet are dropped on time out. Because at the receiver node lot of hosts are sending data packet, and they are forming long queues at the node buffer and some packets TTL is expired before they are removed from the queue buffer. Therefore they are dropped from the buffer.

We have tried various solutions proposed in the literature, mainly categorized as transport layer and application layer solutions. Besides these solutions we have also perform some other experiments in order to eliminate or at least compensate Incast, like varying RTO_{min} and increasing buffer size, but we couldnt get satisfactory results. In the end we try to implement DCTCP as one of the transport layer solutions but we are facing some unresolved issues with it. Finally we implemented an application layer approach and try to schedule the responses to the client in order to avoid losses at bottleneck link.

2 Introduction

2.1 Incast

Our Project deals with TCP Incast problem and its solution for High speed low latency DataCenter networks. In order to emulate Incast, our DataCenter networks should have:

- High bandwidth (e.g. 1Gbps)
- Low latency
- Small switch buffer

Along with above prerequisite, request-response mechanism should be as following:

- There should be barrier synchronization between requests. Aggregators splits the requests and pass them to servers also dont proceed until they get responses from all the servers.
- Number of incoming packets at a particular time instant grows with number of servers, therefore servers return a relatively small amount of data per request

There are myriads of servers responding to client at the same time, which can lead to buffer overflow in switches (since buffer size is very small). Also some server may not respond within the time constraints in that case aggregators will have to wait for default TCP timeout 200ms. This will reduce goodput i.e. throughput at the application level.

2.2 DataCenter: Design

DataCenter is a physical or logical centralized repository for storage, management and processing of data for a particular knowledge body or business purpose.

Our DataCenter have following design

1. Basic design exploits the concept of aggregation, which are in form of clusters.
2. Each cluster have 4 switches they are vertically and cross connected to each other.
3. There are n number of hosts for each lower switches in the cluster, each of the host is again connected to both lower switches. Here were taking $n = 2$.
4. Each of the cluster is upwardly connected to upper layer switches, called The Core.
5. Each of the upper left switch in the cluster is connected to N(modulo Core_Size) and $N + 1$(modulo Core_Size) similarly upper right switch is connected to $N + 2$ (modulo Core_Size) and $N + 3$(modulo Core_Size).

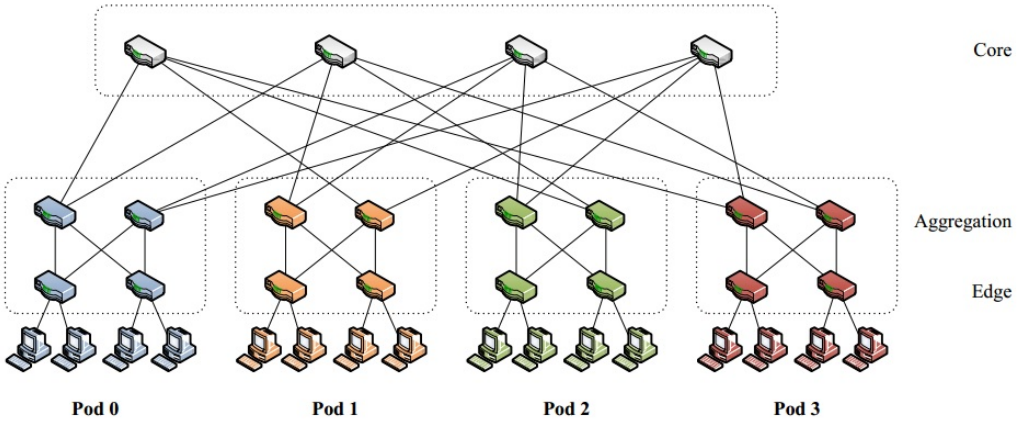

Figure 1: Our DataCenter fat tree topology.

3 Software Requirements

1. **Mininet:** It will be used to emulate datacenter network over the single computer system.
2. **SSH enabled terminal:** It will be used to define customization in the network.
3. **Virtualization Software:** We have used Oracle VirtualBox as a virtualization software for installing Mininet.

3.1 Mininet: installation and hands-on

Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking.

3.1.1 Installation:

Mininet is available for public use and benefit on http://mininet.org/download/. Now follow these steps to install mininet.

1. We downloaded the Mininet VM image V2.1.0 from the above link and extracted.
2. We installed a virtualization software, in our case we are using VirtualBox, since it is free and available for both windows and Linux.
3. Then we created a virtual machine of Linux type in VirtualBox and use the existing VM image of mininet.

4. Later on we started mininet from the VirtualBox by providing username and password as "*mininet* ".
5. To start mininet type command "sudo mn, it will create default network which contain one controller, one switch and 2 hosts.
6. Now we had hand-on this default network created by mininet. Commands can be run with following format < *server-name*> command <*arguments*>. Few oh them are listed below:
   (a) **ping:** it will send ICMP packet from source host to targete host.
   (b) **pingall:** it will force all the machine to ping other machines.
   (c) **net:** it will provide details about created networks, i.e. hosts and links between them.
   (d) **nodes:** it tells about all available nodes in the network.
   (e) **dump:** it provides all the details about networks, i.e. IPs and process-ids.
   (f) **exit:** it brings back from mininet interface.
7. To clear all virtual networks from the memory which have been created, use following commands, "sudo mn -c".

3.1.2 DataCenter: Deployment

To deploy our DataCenter, we have used a python script which create network over the mininet using commands, and we're listing few of them:

1. **addSwitch(Switch_Name):** This function takes switch name as argument and create a switch in the network.
2. **addHost(Host_Name):** Works same as addSwitch but add a host instead of switch.
3. **addLink(node-1, node-2):** This function creates a link between given nodes, here nodes may either be switch or host.

We are using flood light controller for controlling the traffic in the network.

4 Incast Emulation

DataCenter topology design: In contrast to our previous work, we have modified our DataCenter to have a special switch called load balancer and a host attached to it, to see Incast phenomenon.

First of all client issue a request which is passed to aggregators via load balancers. Now aggregators will partition the request and pass them to server. While collecting responses from servers, due to small buffer, Incast is observed at the load balancer switch.

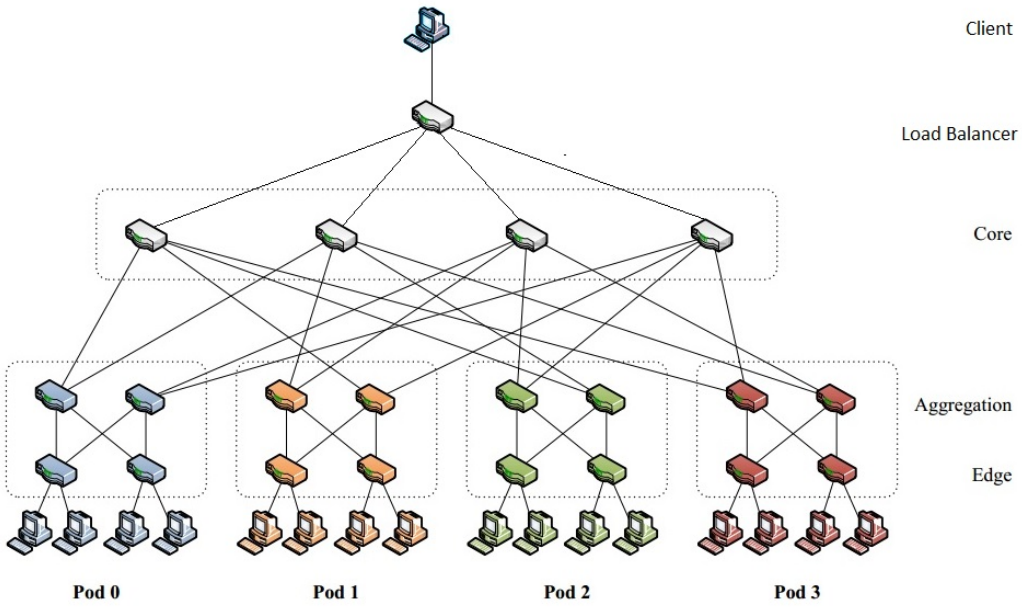

Figure 2: Modified DataCenter fat tree topology.

4.1 Implementation: Specification

- Link Bandwidth: 1 Gbps
- Host and switch output queue size: 64 KB
- Request Size per server: 1MB/n, where n is the number of servers.

4.2 Implementation

1. Platform is python, implementation have been carried out via various python scripts.
2. Client send request by creating a socket, asking data from servers. To make sure response data exceeds switch buffer, we have ensure that packet size will be 1MB/n. We have used python program for flow generation and other flow related issues.
3. Client will wait until all the responses have been gathered. This causes underutilization of links.

Fig(3) shows the observed Incast, for RTOmin=200ms, for different number of servers. There is clear decline of goodput.

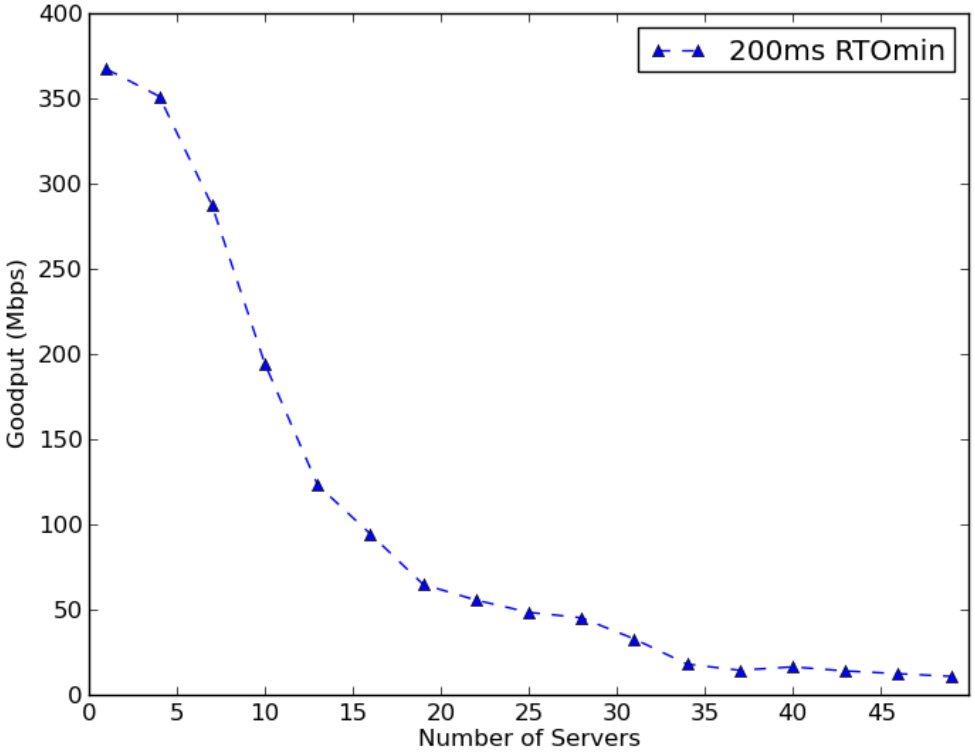

Figure 3: Incast observed

5 Solution To Incast

We had simulated the Incast problem on our Mininet virtual network on the fat tree topology. We tried to get an efficient solution for Incast by implementing the already proposed techniques for Incast handling in literature.

5.1 Reducing RTO_{min}

We first tried to alleviate the Incast problem by changing the value of RTO_{min} but observe that RTO_{min} does not impact any significant effect on the Goodput of the entire network. Since DataCenters have very less delay of the order of μ seconds, there one can misinterpret that decreasing the RTO_{min} may reduce the Incast as we wont have to wait for extra 199 μs for retransmission. But simulation fig(4) suggest that reducing RTO_{min} has almost no effect. Where fig(5) suggest that RTO_{min} has lesser effect for large number of servers, and it saturates for 16 servers in our simulation.

Then we practiced DCTCP for handling Incast. The elaborated procedure is mentioned further in the report.

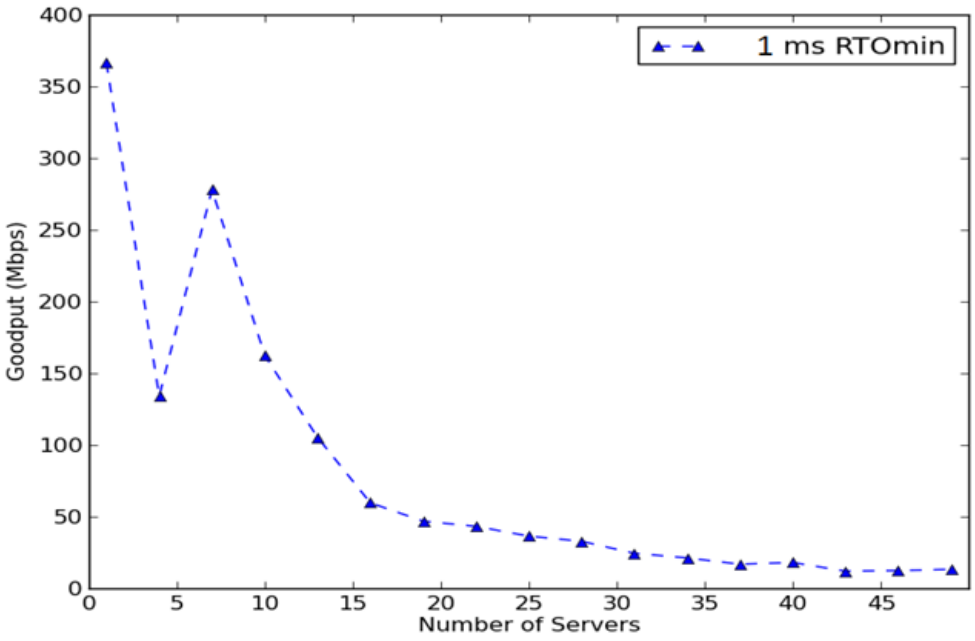

Figure 4: Impact of reducing RTO_{min} to 1 ms

5.2 DCTCP

5.2.1 Introduction

DCTCP stands for DataCenter TCP, which is used in datacenters to prevent Incast. The core strategy of the DCTCP is used to work with commodity switches with a thresholds, which certainly can be set to some appropriate value to exhibits the level of congestion. Further it employs active queue management with a single parameter i.e. marking threshold (K). Now switches mark the arriving packets with CE codepoint as it crosses the thresholds. Source is also reminded queue overshoot as soon as possible, by receiver marks the acknowledge packets by setting/resetting ECN flag. Marking decision is taken by a two state machine which looks back up to one step only, which send 1 Ack for every m packets with ECN=0 if CN was previously zero and vice-versa. Otherwise send immediate Ack with ECN to zero or one if CN was previously zero or one respectively.

Sender maintains a parameter α as how much of packets are marked. This α is updated for every window (roughly in each RTT):

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

Where F denotes fraction of packet marked in the previous window and 0¡ g¡ 1 used to control exponential average. Indirectly α is the measure for source to indicate the probability of queue size greater than K. It means α close to 0 means no congestion and

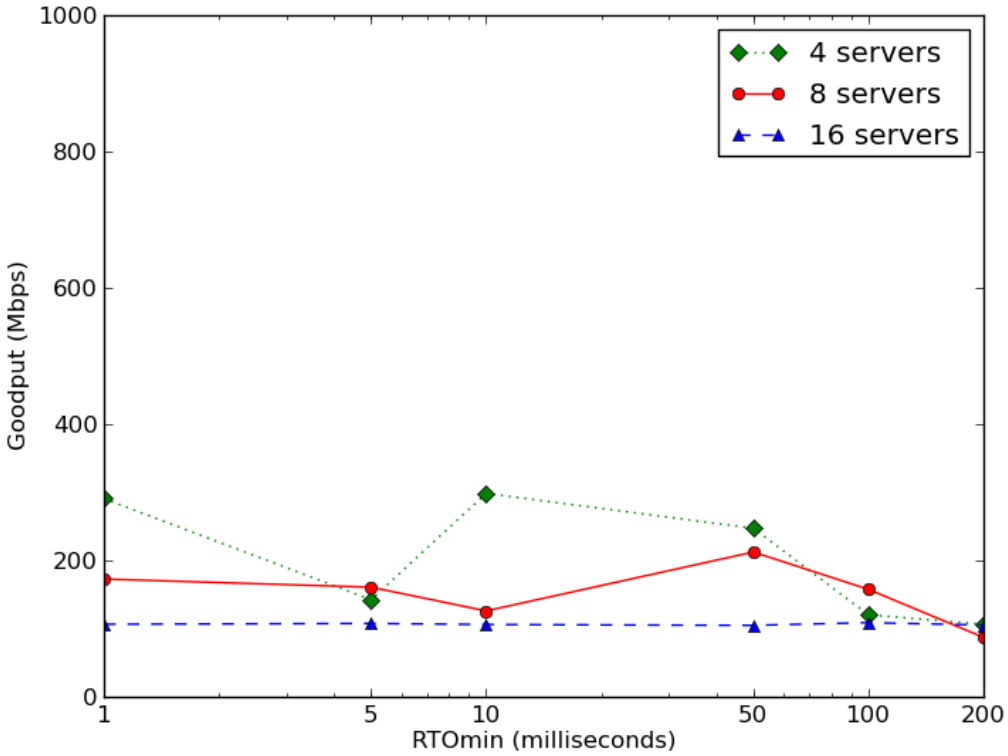

Figure 5: Impact of RTO_{min} on goodput for different number of servers

close to 1 means sever congestion. This α is used by the source to set the window size as:

$$cwnd \leftarrow cwnd * (1 - \alpha/2) \tag{2}$$

In our topology, at the end there are n number of hosts in which n-1 are workers are serving 1 server (aggregator) as shown in the figure(6).

Here query forwarded from the client is routed all the way down to the final aggregator which is passed to n-1 workers. They process the data and pass the result to server. This is exact place where Incast is observed in each core, satisfying all the conditions for Incast to occur as limited switch memory, barrier synchronized requests.

5.2.2 Implementation

In the implementation phase, we downloaded source code of the stock kernel which contains basic services for the linux envirnment. Stock kernel has Reno as default TCP protocol to communicate over the network. We have to make some modifications to introduce DCTCP in the stocked kernel by putting the DCTCP patch provided by Stanford University at github.com/. This patch integrates DCTCP protocol on the kernel source code. After that

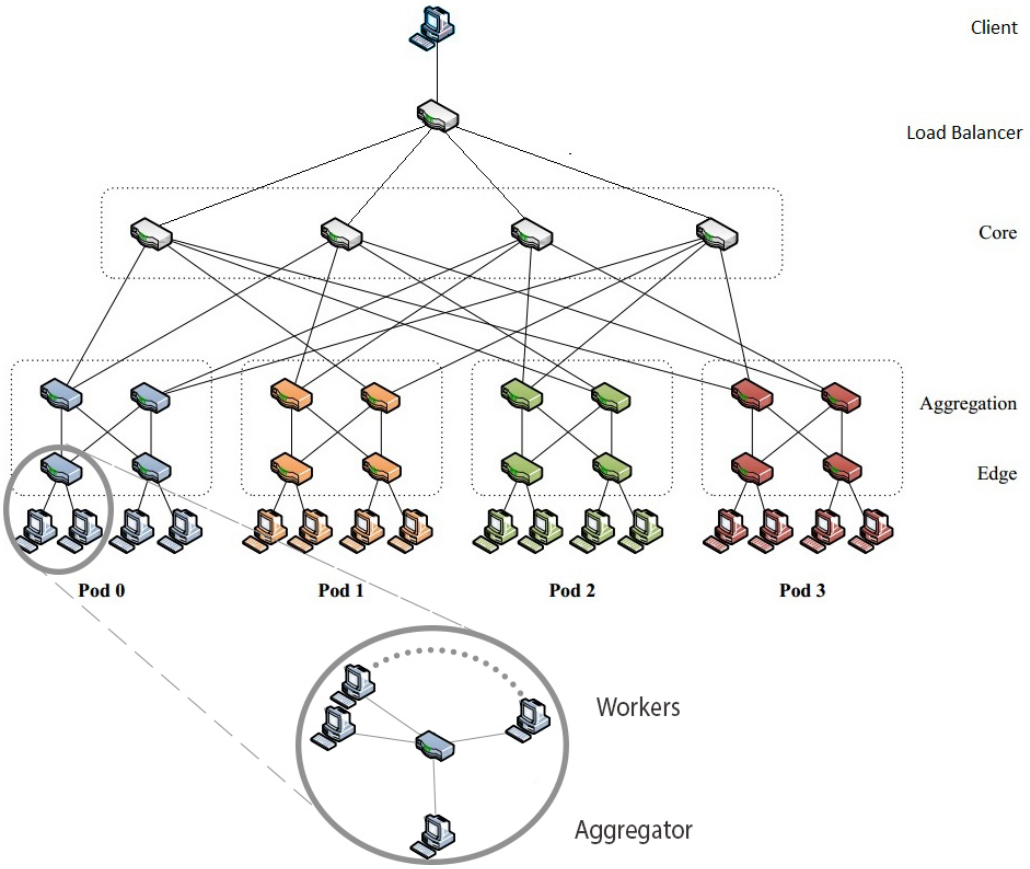

Figure 6: A more detailed view at cluster network

we have compiled and create ".deb" packages from the source code to get DCTCP enabled kernel. Then we installed these packages on our mininet virtual machine using commands below:

Linux kernel v2.6.38.3 can be found on *link*[1]. After downloading the kernel, unzip it and put it in the directory

```
/home/mininet/incast/dctcp/linux-2.6.38.3/
```

Then download the patch for DCTCP provided by *link*[3], now put the patch in the extracted kernel directory. Again move to the kernel directory "linux-2.6.38.3" and run the patch command.

```
$patch -p1 < dctcp-2.6.38.3-rev1.0.0.patch
```

Now compile the the kernel using following commands:

```
$ sudo make-kpkg clean
```

```
$ sudo time fakeroot make-kpkg --verbose --initrd --append-to-version=-dctcp
```

Now install the packages using following commands.

```
$ sudo dpkg -i linux-image-3.0.1-dctcp*.deb

$ sudo dpkg -i linux-headers-3.0.1-dctcp*.deb
```

Finally, we have two kernels installed on our mininet virtual machine. Then, we have to update grub loader to get the boot menu option for the newly installed kernel to boot it, restart to boot from the new kernel.

5.3 An Application Layer Approach

5.3.1 Introduction

The research question we study is how to schedule responses to client requests to avoid data losses at a bottleneck link. In particular, we model application-level scheduling, which does not require any changes in the TCP stack or the network switches. The client requests data using a large logical block size. In particular, 1 MB is a common read size in distributed file systems, such as Google File System (GFS). The actual data blocks are striped over many servers using a much smaller block size (e.g., 32 KB) called the Server Request Unit (SRU). A client issuing a request for a data block sends a request packet to each server that stores data for the requested data block. The request, which is served through TCP, is completed only after all SRUs from the requested data block have been successfully received by the client.

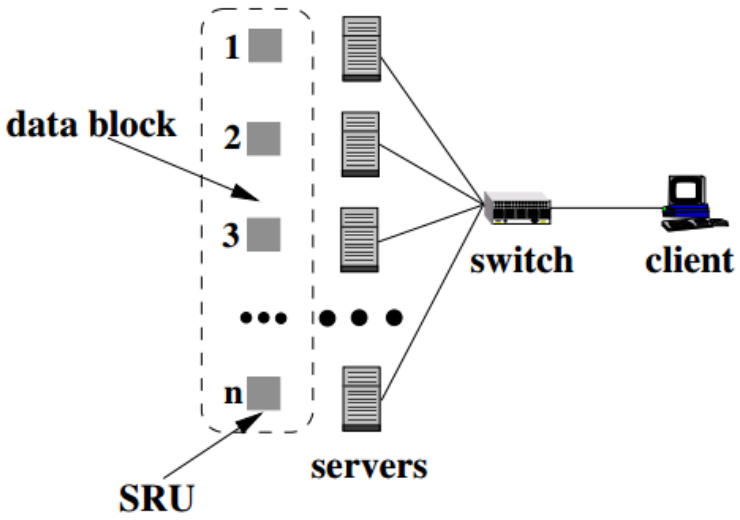

Figure 7: Cluster based storage system

In this model, a client requests a data block of size S (in bytes), which is striped over n servers. The bottleneck link has capacity C (in bits per second, bps) and buffer size B (in

bytes). The propagation delay between each server and the client is R. The size of data in each data packet is S_{DATA}, and the size of a data frame is S_f . The TCP timer granularity of the client OS is Δt (in seconds). The initial TCP congestion window is one packet. We assume that per-packet ACKs are used for a TCP flow.

This is an application-level solution that does not require any changes to the TCP stack or network switches. Since the main reason for TCP throughput collapse is data losses inducing a retransmission timeout, we explore how to schedule server responses to the same data block request so that there is no data loss at links.

5.3.2 Implementation

There are few protocol parameter which have been given in following fig(8).

MODEL NOTATION

Symbol	Description
S	size of a data block (in bytes)
S_{SRU}	size of an SRU (in packets)
n	number of servers
N	maximum number of concurrent flows (servers responding simultaneously)
k	number of groups of concurrent flows
C	bottleneck link capacity
R	propagation delay between a server and a client
B	buffer size (in bytes) at bottleneck link
S_{DATA}	size of a data packet
S_{ACK}	size of an acknowledgment packet
S_f	total size of a data frame
Δt	TCP timer granularity in client OS

Figure 8: Protocol Parameters and description

The key idea of the protocol is to scedule SRUs from the same data block without causing buffer overflow at the bottleneck link. Protocol claims that maximum goodput g, can be achieved in an application in a DataCenter with lossless scheduling is

$$g = \frac{S}{T'(\lceil \frac{n}{N} \rceil) + T} \tag{3}$$

where,

$$T' = \lceil \frac{T}{\Delta t} \rceil \Delta t \tag{4}$$

$$T = \frac{(S_f + S_{ACK})S_{SRU}}{C} + \lceil log_2(S_{SRU} + 1) \rceil (R + \frac{B}{C}) \tag{5}$$

$$N = \frac{B}{S_{SRU} wnd_{max}} \tag{6}$$

To avoid data losses, the responses of servers should be scheduled by groups, N flows in a group, at time moments 0, T',2T',3T',...,$T'(\lceil \frac{n}{N} \rceil - 1)$. If generalizing, the i^{th} server, where $1 \leq i \leq n$, starts responding at time ti and completes transmission at time t_i+T:

$$t_i = T'.mod(i - 1, \lceil \frac{n}{N} \rceil) \tag{7}$$

We define the number of groups of concurrent flows

$$k = \lceil \frac{n}{N} \rceil \tag{8}$$

Two important things about scheduling process first, the number of SRUs may not always pack perfectly into a rectangle, which is shown by the time slot of the response of the n^{th} server. Second, the number of concurrent SRU flows is not always equal to N.

5.3.3 Result

With 5 flows and 50 server and assuming clock granularity Δt=1 ms we found result provided in fig(9):

Simulation shows that Incast wasn't completely eliminated still it gives a bbetter result and thus a motivation to work on this protocol while keeping delay as low as possible.

6 Conclusion

We explored both transport layer and application layer solutions by varying RTO_{min} but it didn't gave us satisfactory results. We then try to install DCTCP, after so much efforts putting in it DCTCP was finally installed but while patching the installed kernel it gave some error regarding onpenVSwitch.

Finally we moved towards an application layer solution to address Incast. The main idea of the approach is to schedule the server responses to data requests so that no packet losses occur at the bottleneck link. The main result we derive is the achievable goodput of a data center application if using scheduling described above.

From the experimental results, we observe that we have been able to address the problem of incast signicantly by limiting the number of concurrent senders and introducing delays between them. We observe that the goodput remains high for most of the DataCenter scenarios having a high number of servers.

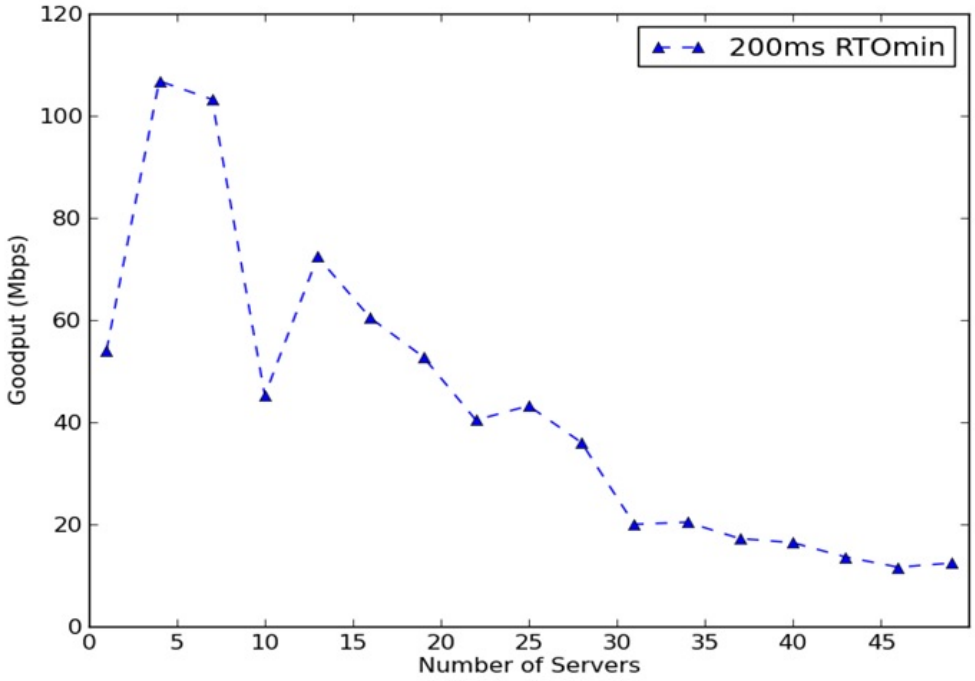

Figure 9: Moderate reduction in goodput with this application layer solution to Incast

7 Future Work

We have proposed an application layer solution to address the problem of TCP incast throughput collapse which has shown to reduce the throughput collapse signicantly. We aim to propose better means of nding out the number of concurrent senders and the delay that has to be introduced so that the throughput reduction is completely eliminated.

References

[1] http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.38.3.tar.bz2

[2] https://github.com/mininet/mininet/wiki/Documentation.

[3] https://github.com/myasuda/DCTCP-Linux

[4] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, Yongguang Zhang. ICTCP: Incast Congestion Control for TCP in Data-Center Networks.

[5] http://reproducingnetworkresearch.wordpress.com/

[6] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, Brian Mueller. In Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication.

[7] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. In PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric

[8] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, Anthony D. Joseph. In Understanding TCP Incast Throughput Collapse in Datacenter Networks.