Improvements for Congestion Control Algorithms in Large-Scale Datacenter Network Algorithm Adjustment for DCQCN undergraduate dissertation

Yuchen Yang

May 29, 2018

141242048
Kuang Yaming Honors School, Nanjing University
ranchyang96@gmail.com
Instructor: Professor Chen Tian

Abstract

Modern datacenters are experiencing a fierce increase in scale and traffic bandwidth. Remote Direct Memory Access (RDMA) permits high-throughput, low-latency networking, which is especially useful in such a large-scale scenario. The most primitive method (like TCP/IP stack) for congestion control is to drop packets when the receiver buffer is full. Later we have Piriority-based Flow Control (PFC) to generate congestion information in ACKs. Before this paper, we have Datacenter Quantized Congestion Notification (DCQCN) which uses the state-of-the-art scheme for congestion control. However, we may find DCQCN incapable of some large-scale traffic scenarios. In this paper, we analyze the drawbacks of DCQCN. At the same time, we present our improvements to DCQCN and name it DCQCN+. Improvements mainly focus on the adaptive increasing step and intervals. We have implemented them on testbeds and NS3 simulation. Our method has 10 times smaller latency than DCQCN under large-scale conditions (incast of over 400:1) and 4 times larger flow capability than DCQCN. While in small incast cases, we have similar performance with DCQCN. This paper additionally focuses on my own work, about the testbed implementation and configurations of the method. It actually includes many detailed methods used for execution of experiments and detailed information about testbed experiment results.

1 Introduction

Modern datacenters are experiencing a fierce increase in both scale and bandwidth. To be more specific, we find following features of datacenters nowadays:

1. Small latency: $< 100\mu s$
2. High bandwidth: 10/40 $100Gbps$
3. Shallow buffer: $< 300MB$ for ToR
4. Large scale: > 10000 machines

As we know, datacenters need small RTTs (some may be even tens of microseconds), so abilities to handle burst flows for incasts and support the equality and balancing for concurrent flows matter a lot. Traditional TCP/IP stack surely can't do the job. When encountering a large number of concurrent flows, TCP may choose to drop packets when the receiver buffer is full. That's not tolerable for sure in datacenters since datacenters require lossless network to ensure the security and accuracy in large-scale traffic. Then the first idea coming is to find a way to predict the congestion of the receiver end. We hope to let the sender get some information about the receiver.

Then we get Explicit Congestion Notification (ECN) [1] at the beginning. This method uses a mark in the IP header of the ACK. If the receiver dropped a packet, it echoes the congestion indication to the sender, so that the sender can reduce its sending rate. Such ECN flags indicate the existance of congestion from receiver side after congestions have already happened. ECN somehow relieves the congestion and makes the receiver side drop fewer packets, but this still can't be lossless.

Later Quantized Congestion Notification (QCN) [2] is developed. QCN enables the switch to control the packet sending rate of an Ethernet source whose packets are traversing the switch. This maintains a stable queue occupancy and is easy to be implemented on hardwares. QCN is also applicable for multiple flows in a same port.

Priority-based Flow Control (PFC) [3] applies the PAUSE frame to make things better. PAUSE is used by the receiver to send feedback to the sender about the remaining buffer space. PFC actually divide services into 8 classes to make the feedback information more accurate. Since the PAUSE frame carries more information, sender can know the exact remaining space in receiver buffer. Thus the reaction on sending rate can be more accurate. However this method can't be specific on flows but only on ports, which is usually combined with ECN to make it work better.

Before this paper, we have Datacenter QCN (DCQCN) [4] which handles congestion better and is the basic point of our work. DCQCN is the application of

QCN on datacenters, which divides the overall algorithms into three parts. It's kind of combination of Datacenter TCP (DCTCP) [5] and PFC. A brandnew concept mentioned here is Congestion Notification Packet (CNP), which functions similarly as PAUSE frame in PFC but carries more informations. CNP packets are actually sent by NP which is the receiver end. Each time when a packet with ECN mark is got and there are no CNP packets sent during the last interval period, a CNP is sent to notify the sender. Here the interval period mentioned is usually a interval set up in advance by hardware to make sure that there won't be a CNP burst on RP side. This interval is also greatly limited by hardware ability. Detailed information is also be mentioned in later sections.

In the end I briefly introduce the most important points about our new method DCQCN+. DCQCN+ deploys dynamic rate control mechanisms to adapt to incast of different scales instead of the fixed rate control for DCQCN. What I'm in charge of is the testbed configuration and experiments.

The paper is organized as follows. In Section 2 we introduce the background of the project. In Section 3 we present limitations of DCQCN and necessity for improvements. In Section 4 we show possible improvements for DCQCN. In Section 5 I display detailed methods used for testbed experiments. In Section 6 we show the results of experiments. In Section 7 there are some discussions and future work.

2 Background

2.1 Mellanox switch and ConnectX-4

In the testbed experiments, we are using Mellanox SN2700 switch and ConnectX-4 Network Interface Cards. Mellanox SN2700 carries a huge throughput of 6.4Tb/s, 32 ports at 100GbE. The port speed can actually vary from 10Gb/s, 25Gb/s, 40Gb/s and 100Gb/s. All 32 ports are connected to 16 machines inside our testbed with 2 each, but only 9 ports on 9 separate machines are used inside our testbed experiments.

ConnectX-4 EN adapter can support 100Gbps Ethernet Connectivity. Its Virtual Protocol Interconnect also supports EDR 100Gbps InfiniBand traffic. We have ConnectX-4 adapters on all machines in our testbed. Additionally we have tested the ability of ConnectX-5 adapters on a new testbed about the Congestion Notification Packet generation ability. This is described in later sections.

2.2 Remote Direct Memory Access

Remote Direct Memory Access almost satisfies all the demands for datacenter networks. It permits high throughput and low latency. Similar to Direct Memory Access, RDMA actually allows user-space applications to directly read or write without the operation from any operating systems. Such network feature omits the possible copies inside systems, thus performs better inside datacenters.

2.3 Priority-based Flow Control

Priority-based Flow Control ensures the lossless feature for RDMA. No buffer packet overflow is a must for datacenter networks.

However, PFC actually does nothing to make the latency low. When the receiver buffer is almost full, the packets inside the buffer queue up for a long time which makes the queueing latency extremely high. Such unfairness and high latency are fatal to datacenters and we hope to make up the drawbacks.

2.4 Explicit Congestion Notification

Explicit Congestion Notification is designed to indicate congestion to the sender. An ECN-aware router may set a mark in the IP header instead of dropping a packet in order to signal impending congestion. The receiver of the packet echoes the congestion indication to the sender, which reduces its transmission rate if it detects a dropped packet.

2.5 DCQCN

Datacenter Quantized Congestion Notification is explained in detail by [4]. Here I wil generally describe the mechanisms used in DCQCN.

Three parts of algorithms are developed. The parts are Congestion Point (CP), Reaction Point (RP) and Notification Point (NP), which are switches, senders and receivers respectively. Both the switch and end points participate in to make things right.

For NP, i.e. receivers, they need to generate and send CNP when a packet with ECN mark and there is no CNP sent during the last interval. CNPs are generated by receivers, indicating the remaining size of the receiving buffer.

For CP, a function of ECN marking probability P is related with Egress Queueing size S:

for $0 < S < K_{min}$, $P = 0$

for $K_{min} < S < K_{max}$, $P = (S - K_{min})/(K_{max} - K_{min}) * P_{max}$

for $S > K_{max}$, $P = 1$

which is shown in Figure 1.

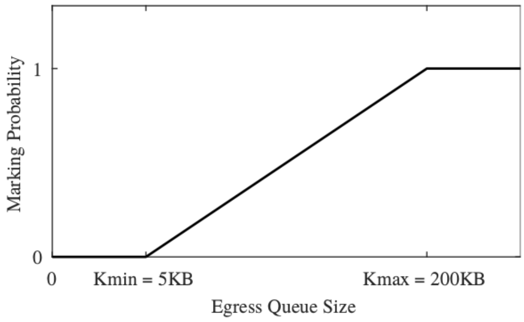

Figure 1: ECN Marking Probability for CP in DCQCN

Such marking probability is a brandnew idea from QCN. Previously marking ECN or not is a fixed thing without possibility. Such 0/1 resolution for ECN is not proper because that may cause the late reaction from sender.

3 Problems of DCQCN

The major problem for DCQCN is the failure of dealing with large-scale incast traffic. Both in simulation [6] and testbed experiments show that the switch buffer queue length remains high after a large burst at the beginning.

In testbed experiments, we actually find that when the flow number is under 400, the queue length will go down after 2 or 3 seconds. Here the flows are continuous InfiniBand traffic lasting for about 80 seconds and they are started at approximately the same time. There could be some difference in starting times but within 1 second. This means that when the flow number is not large, convergence should be reached within 2 seconds.

However, when we start over 400 flows, the high queue length lasts till the end of all flows. Convergence fails to be reached when the flow number gets really large.

Nowadays the tendency is that scale of datacenters grow larger and larger which gives much pressure on datacenter networks. Such a long queue length surely lead to high network latency and unfairness. Thus the demand of datacenter network is not met.

In simulation results, we find even worse convergence effect. Here from Figure 2, we can see that 160 flows fail to be converged under 10Gbps links and 80 flows fail to be converged under 40Gbps links. All the parameters used here are default parameters on Mellanox official forum [7].

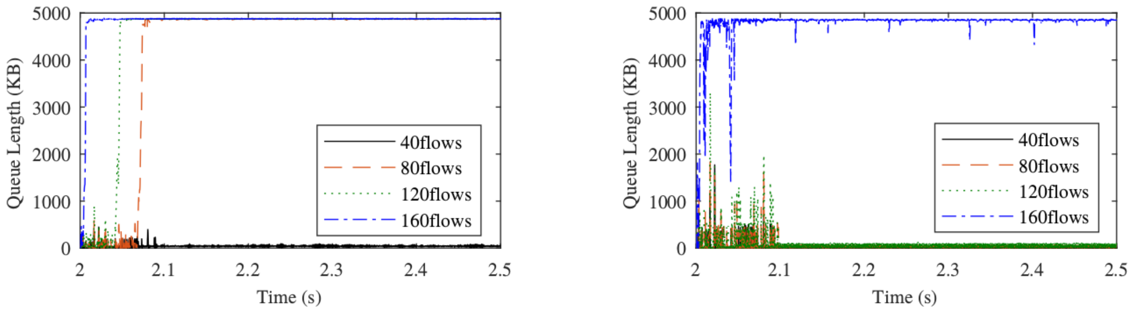

Figure 2: DCQCN fails to converge

4 Improvements for DCQCN

The key factor of DCQCN's failure is DCQCN's increase step, recovery speed and CNP supplies. These information used are out of date for the growing scalability for data centers. Those fixed increase steps, recovery speed and CNP intervals are not adaptive for large-scale traffic. What we do is to find factors which actually influence the congestion and can predict the congestion level better.

Our work, DCQCN+ uses dynamic parameters to better analyze the congestion and therefore relieve it. DCQCN never cares about the incast scale and thus uses fixed parameters to do the job. That surely won't work for large-scale incast since the reaction and adjustments of sending rate are not very accurate.

DCQCN+ is designed to use the CNP period carried in CNPs and the flow rate to reflect the condition of the incast scale. There are some remaining available fields to carry the period information of CNP. Also, since larger flows are more probable to be marked with ECN, its possibility to trigger a CNP is larger. Thus all congested flows can get similar rates and be balanced.

To be more specific, improvements are discussed from three points (Congestion Point, Notification Point and Reaction Point).

4.1 Congestion Point

This part remains almost the same. What we can do at Congestion Point is really limited since CP is responsible for ECN marking. The probability for ECN marks are very resonable so we leave this part the same.

We also need to mention here that PFC is still enabled to ensure lossless feature of datacenters. Although our method can make sure that convergence can be reached in most cases, we can't ensure lossless feature at the starting burst if PFC is disabled.

4.2 Notification Point

Receivers are in charge of generating CNPs. This is the most important part of the algorithm.

At NP, two factors decide the CNP interval, the hardware ability of generating CNP and the demand for CNP at Reaction Point. We hope to supply enough CNPs with shorter intervals for RP to properly adjust the sending rate. At the same time we don't want to cause other problems like CNP burst or CNP congestion if we set the interval which is too short. Such problems may cause unnecessary bandwidth cost. The ability of hardware may influence the minimal interval and demands for CNPs decide the maximal interval. We prefer the maximal interval to relieve the possibility of bandwidth cost.

Once the flow rate collects enough information to decide the incast scale, the timer period should be decided.

4.3 Reaction Point

Here we first show the original DCQCN RP algorithm pseudocode in Figure 3.

We make some changes on the updates of parameters.

The reaction when receiving CNP is similar:

$$R_T = R_C$$

$$R_C = R_C(1 - \alpha/2)$$

$$\alpha = (1 - g)\alpha + g$$

Here α is a reduction factor, actually indicates the congestion level at the CP. R_C denotes current rate and R_T denotes target rate. g is a parameter to estimate the congestion level.

Instead of DCQCN's fixed rate increase timer $K = 55\mu s$, we make it flexible:

$$K = \lambda max(\tau, \frac{MTU}{R_C})$$

where MTU denotes Maximal Transmit Unit and τ denotes CNP interval.

Here we remove the byte counter and add the rate increase counter. When the rate increase counter times out, the state S increses 1.

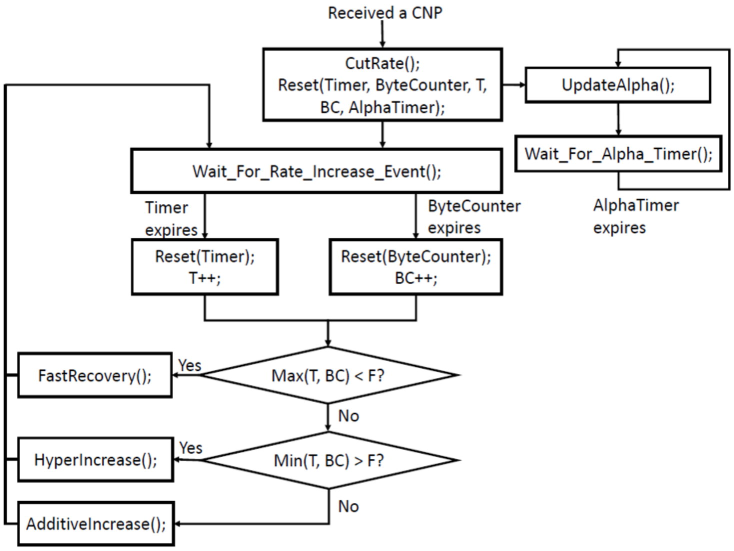

Figure 3: RP algorithm from DCQCN

The fast recovery scheme remains the same when S is less than the threshold $F = 5$:

$$R_C = \frac{R_C + R_T}{2}$$

When $F < S < 4F$,

$$R_T = R_T + min(\frac{1}{10}R_C, \frac{1}{100}R_l)$$

$$R_C = \frac{R_C + R_T}{2}$$

where R_l is a ratio used to bound the increase step for small incast cases.

When $F > 4F$, Hyper Increase is applied,

$$R_T = R_T + min(R_C, \frac{S - 4F}{100}R_l)$$

$$R_C = \frac{R_C + R_T}{2}$$

Additionally, when α timer runs out,

$$\alpha = (1 - g)\alpha$$

We can see a lot of changes to make the algorithm better. Such dynamic adjustments of parameters greatly improve the buffer estimation in advance and drain the congestion fast to maintain the buffer queue length low.

5 Implementation

We set up the testbed of testing for DCQCN+ on Mellanox SN2700 switch and 9 Ubuntu servers with ConnectX-4 adapter cards. Mellanox SN2700 provides the most predictable, highest density 100GbE switching platform for the growing demands of today's data centers, which satisfies our experiment environment for over 400 flows at a time. ConnectX-4 dual-port adapter cards with Virtual Protocol Interconnect (VPI) support 100Gb/s InfiniBand and 100Gb/s Ethernet connectivity. All the features including RoCEv2 [8], PFC and ECN are supported on Mellanox SN2700 and ConnectX-4 adapter card.

To generate enough number of flows for our experiment, the topology is shown in Figure 4. Each machine can create several flows simultaniously.

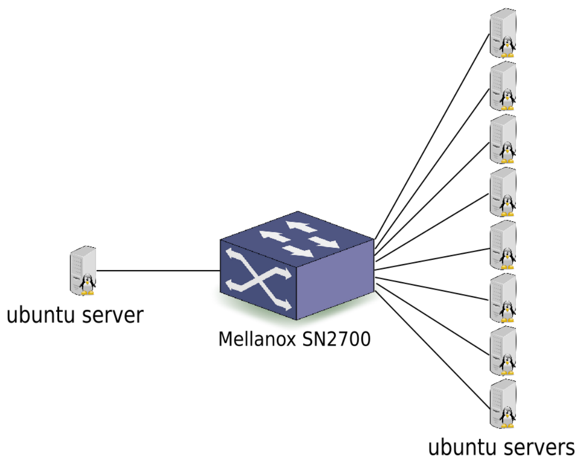

Figure 4: Topology for multi-flow tests

To be clear, we regard the server on the left (which is the receiver) as S0 and the servers on the right (which are senders) as S1 to S8. All the servers are configured into the same virtual local area network (VLAN) so that we can construct an incast traffic inside the network.

5.1 Switch Configuration

Mellanox 2700N has provided a lot of interfaces for users. Among them, we need to configure VLAN, PFC, ECN and buffer size to make things work.

5.1.1 VLAN

The whole expetiment environment should be separated from outside traffic including SSH connection and SFTP file transmission. VLAN is thus needed to create a separate topology.

Switch need to enable the VLAN feature and include all machines in the same VLAN session. This includes some work on distinguishing ports.

5.1.2 PFC

PFC is needed because what we want is a lossless network which is a must for data center networks. PFC makes receiver to send PAUSE frame to sender where sending ratio is limited upon received PAUSE. Default PFC configuration contains 8 priority classes and we only have 2 used here. The mapping is mentioned in the following subsection about ECN.

5.1.3 ECN

ECN priority is configured to make CNP function well. Higher priority is needed for higher traffic class. Inside our experiments, there are mostly two classes of traffic. RDMA traffic is in traffic class 3 and should be mapped to swith priority 3. Other traffic like TCP is in traffic class 0 and should be mapped to switch priority 0. The reason is that the traffic for setting up a flow shouldn't be blocked. If we don't set the priority difference, the flow setup traffic for a flow is blocked when there are really a large number of flows. The flow setup period should use higher priority so that more flows can be created. About this point, we explain it in later sections. And the probability to make a ECN mark should be configured the same as mentioned in DCQCN [4].

5.1.4 Buffer

About the buffer size, we don't need to be very specific. The maximal buffer size inside Mellanox 2700N switch is 5.1MB. One important factor to be considered is the buffer usage during the experiment process. When a lot of flows are created, there must be a burst of packets in the receiver buffer at the beginning. After a short period of reaction time, the buffer usage should go down because the senders are limited by switch based on flows. In the results, what we are hoping to see is that the beginning of the buffer is large but after the short reaction time, the buffer is low. Thus the buffer here is only to handle the burst at the beginning.

5.1.5 Additional Configurations

To make things clear, we need to cut unnecessary parts of traffic.

We obviously don't contain loops inside our topology so that the spanning tree protocol is removed from the switch.

The interface speed is set to 100Gbps when testing the hardware ability of generating CNP. For other most experiments, interface speed is configured as 10Gbps (reason is explained in later sections).

5.2 NIC Configuration

At the very beginning, we need to install Mellanox OFED (driver for ConnectX-4) for every server.

After that, we have several steps (most corresponds with the upper switch configurations):

1. Enable PFC and set up the priority.
2. Enable ECN for both InfiniBand traffic and TCP traffic.
3. Set CNP priority and DSCP priority.
4. Set priority for RDMA traffic.
5. Open sniffer for Tcpdump [9] to capture InfiniBand packets.

5.3 Experiment Process

The general idea is creating an incast and observe the congestion point. To be more specific, we are generating flows from S1 S8 (tens of flows from each) and observe the bandwidth tendency, latency and buffer usage.

However before all of that, the first experiment is to test the CNP generation speed for ConnectX-4 and ConnectX-5.

5.4 Ability to Generate CNP

All NIC has limited ability. For ConnectX-4 and ConnectX-5, we hope to figure out the exact ability of doing this so that we can adjust the DCQCN efficiently.

The topology is simple. Two servers are connected to the switch, one as sender and one as receiver. For the receiver side, the interface speed is set to 10Gbps. The sender side has an interface speed of 100Gbps. Thus the congestion surely exists and we use Tcpdump to capture the packets during a short period of time.

Among the packets captured inside the Pcap file, we filter the traffic with right direction and right IP addresses out. According to the timestamps, we can know something about the ability of ConnectX-4 and ConnectX-5.

5.5 Bandwidth Testing

Now the topology is the one shown in Figure 4. We use the command ib_send_bw to start flows.

First let me introduce the key tools used here. The mostly used library is Python Paramiko which provides secure connection for remote command execution and file transmission. It's actually a Python implementation of SSHv2 protocol, providing functions of both server and client. To be more specific, I contracted them down to four functions, one for connection checking, one for remote command execution and the other two for file transmission between local and remote machines. Everything here is using SSH protocol and thus my functions can be implemented by Python Paramiko.

Another point necessary to be mentioned here is multithreading. In the original version, the time for running is previously estimated. Thus the execution time of the program is greatly increased or a command is run before its previous steps are finished. Function "join" can correctly sequence all the steps in the program.

Bandwidth testing needs several steps:

1. Refresh machine states.
2. Time synchronization for all servers.
3. Start Tcpdump simultaneously on S1 to S8.
4. Start ib_send_bw server on S0.
5. Start ib_send_bw clients on S1 to S8.
6. Send Pcap files to local.
7. Parse Pcap files.
8. Use parsed results to draw bandwidth figure.

The overall tests is done by scripts of many languages including Python, C, Bash, Awk and Gnuplot.

Let me describe the above steps one by one.

5.5.1 State Refresh

Because we start a large number of flows on each machine, the command of ib_send_bw may fail to complete correctly, leaving some processes continuing during the stages. We need to refresh the states for mahcines.

To kill the processes running from the last experiment, we use the command "pkill -f" to kill processes with the label of "ib_send".

5.5.2 Time Synchronization

We use Network Time Protocol (NTP) to synchronize the time for S0 to S8. The accuracy for NTP is about 0.01ms so that should work for servers on a rack. Choosing a server inside the same rack, we set up an NTP server on it.

Then we use Python Paramiko to execute the NTP command on S0 S8 for time synchronization with the NTP server.

Here we need to claim a fact that the receiver side can't tolerate the amount of packets we need to actually generate a bandwidth figure. Even an increased size of Tcpdump buffer can't hold the traffic of even 1 second which is obviously not enough to analyze the bandwidth tendency. Thus we are running Tcpdump on each machine which can last for approximately 10 seconds. With time synchronized, we can analyze packets on different machines with timestamps correct.

5.5.3 Tcpdump Start

Python Paramiko is still the tool used for remote command execution.

Different from normal Tcpdump, we need to specify several parameters here. "-B 900000" parameter is needed to increase the buffer size of Tcpdump. "-s 60" parameter is to capture only the first 60 bytes which contains just the packet headers.

It's hard to estimate the capturing time needed, but we can limit the packet counter of Tcpdump (which uses "-c"). Thus the file size is shrinked and it saves time for file transmission.

5.5.4 ib_send_bw Server Client Starter

This step contains concatenation work. What we are doing is to accept parameters from the command line execution of the program. This contains the concatenation of mark "&" and other strings. "&" is used for simultaneous command execution. Linux System usually has the ability to run over 1000 processes simultaneously.

Luckily string operations are easy for Python. I'm actually recording the same parts of the commands and adding loops with transformed string from numbers.

About the detailed information about the commands, we have several specific options set. The option "-R" is for RDMA connection setup. However we find that using RDMA traffic for a flow start causes the traffic congested with other already-started flows. Thus we may fail to start a lot of flows at the same time. So in real experiments, we are not using the option.

The option "-S 3" means the priority class for RDMA traffic. This can also be set for ROCEv2 protocol. We add it here just to be sure that the traffic class priority is working.

The option "-x" denotes the device ID we are using, which can be fetched through the command "show_gids". This displays the ROCE version we are using and the corresponding device ID and destination IP. "show_gids" is only needed once for each machine so that it's not actaully included as part of the automation tool.

The option "-d" denotes the network devices used for the execution which is mlx5_1 for our machines.

The option "-D" shows the lasting time of our experiment. It counts in seconds of the continuous sending procedure. In our experiment it's usually 60 or 80 which shows almost a complete tendency of the bandwidth and flows.

The option "-p" is responsible for separating flows. In our experiment, we hope to create multiple flows from one machine. The number could be tens and even reach 90 so we need to use port number to start different flows. The actual implementation of port allocation is like this:

We are taking parameters of machine number and flow number of each machine at the beginning of the overall scripts. Assume that they are m and n. All port number for flows starts from 10011. There is a continuous range from 10011 to $10010 + m \times n$ for port numbers. The first machine takes the position of port number like 10011, $10011 + m$, ..., $10011 + m \times (n - 1)$. To be more general, the i_{th} flow on the j_{th} machine (where $i = 1, 2, ..., n$ and $j = 1, 2, ..., m$) has the port number of $10010 + i + m \times (j - 1)$. Thus all the port numbers don't collide with each other. This makes later steps easier to process. Those are mentioned in later sections.

5.5.5 File Transmission

To be clear, we are running experiments on 9 machines. Besides that, we have a middle machine with accounts to login, making the machines more secure. And we have a local machine to process the results we get. We are using a local machine to do this because we hope to keep the experiment machines clean and fewer new tools should be used there. I use Python Paramiko for remote login and file transmission. A tough point is that all file location should be absolute locations but that's OK since the amount needed is not very large.

Thus the file transmission has two parts. First step would be sending Pcap files from experiment machines to the middle machine. The second would be transmitting the files from the middle machine to the local machine.

5.5.6 Pcap Parsing

This contains several steps. We can't make sure that this is the easiest way to process these packets but it works fine.

1. Transform Pcap files to plain text.
2. Filter the packets to get the traffic with the receiver we want.
3. Divide the packets from all different flows in the same machine.
4. Collect number of packets in every millisecond.
5. Transform timestamps to relative time starting from the beginning of the flows.
6. Add counts of zeros for those milliseconds which contains no packets.

For step 1, I use the command of Tcpdump with the option of "-r" which means reading. With options "-rrrr" and "-nnq" we transfer the Pcap file into plain text with 8 terms. Among them we have the timestamp, source IP, source port, destination IP and destination port. Thus we can approximately calculate the bandwidth from such informations.

For step 2, I use Awk to do the filtering. By limiting the receiver as the only host we regard as incast receiver, we get all the receiver IP (which is the 6^{th} term of the plain text) out of all possible flow receivers.

For step 3, we can only devide all the flows on the same machine using the ports. TCP and InfiniBand has its unique port forwarding methods which would surely dismiss the ports I set for them in the "ib_send" commands. To separate all these flows, I write a C++ program to do the filtering work, collecting all the different port numbers and raise flags for new port numbers. Each new port number starts a new file for collection, containing the packets of a single flow. The file names are carefully calculated. Suppose we have m machines and each starts n flows, then the numbering of the file names are similar to the one previously mentioned in port nubmering. The only difference is that port numbers start from 10010 and the file names start from 85 (which is the machine index in our testbed).

For step 4, I specifically count the digits in every line (which contains a packet) of the plain text. Thus I can make sure which digits represent the condition of the timestamps. Then I decide in plain text which exact digit decides the millisecond and which digit is the starting point of the timestamp. To decide the amount of packets in one millisecond, we can find the ones with a sharing first 3 digits after the decimal point in the timestamp. To do the filtering and counting work, we have an easy commandline tool which is "cut". Using the option of "-c", we can have a count on the packet number with the same timetamp till millisecond digit. We therefore print the timestamps and the packet numbers. (which has two columns and therefore can be drawn into a figure.

Step 5 and step 6 are relatively easy. Record the earliest packet occurs and do subtractions. Then we need to add zeroes for those milliseconds which contains no packet to make the figure fluent and complete.

5.5.7 Figure Plotting

Here we use the tool of Gnuplot, which is a practical plotting tool under Linux systems.

Before we have figured out the timeline and packet count for each millisecond. For most packets, they have the same size of 1KB which is fixed by the command "ib_send". Thus we can indicate the bandwidth through packet count of each millisecond.

5.6 Latency and Buffer Usage

Most steps are similar to the last subsection about bandwidth testing. We are doing this part because as the flow number increases bandwidth collecting is super time-consuming.

Latency and buffer usage can reveal the effect of convergence more effectively. For latency, I'm using "ib_send_lat" on each machine which provides average latency and extreme latency during a time period. What I collect is simply the average latency which is shown in Figure 9 and Figure 10.

About buffer usages, I collect manually from interfaces provided in Mellanox switches. I collect buffer information every 10 seconds to get them collected in Figure 9 and Figure 10.

6 Results

6.1 NIC Ability to Generate CNPs

We test the ability of generating CNPs for both ConnectX-4 and ConnectX-5 NICs.

The experiment topology is really simple. Two hosts are connected to the Mellanox SN2700 switch, with the interface speed configured to 10Gbps and 100Gbps respectively. The 100Gbps one then sends continuous InfiniBand traffic to the 10Gbps one. When the switch is correctly configured with PFC and ECN enabled, we can find a burst of CNPs in a short period of time.

CNP can be distinguished by filtering the OP code of InfiniBand header which is 129. Then I find a period of continuous CNPs and can get the approximate interval between CNPs. With the parameter of CNP interval (min_time_between_cnps) is set to 0, the results collected are just what we want.

Figure 5 shows the interval of continuous CNPs under ConnectX-4 adapters.

7915	13.197500
7916	13.197500
7917	13.197500
7918	13.197501
7919	13.197501
7920	13.197501
7921	13.197502
7922	13.197502
7924	13.197503
7925	13.197503
7926	13.197503
7927	13.197504
8055	13.197556
8056	13.197556
8057	13.197556
8058	13.197557
8059	13.197557
8060	13.197557
8062	13.197558
8063	13.197558
8064	13.197559
8065	13.197559
8066	13.197559
8067	13.197560
8259	13.197636
8260	13.197637
8261	13.197637
8262	13.197638
8263	13.197638
8265	13.197639
8267	13.197639
8268	13.197640
8269	13.197640
8270	13.197640
8271	13.197641
8272	13.197641
8273	13.197641
8274	13.197642

Figure 5: CNP interval test

The first column marks the packet number (those ommitted are not CNPs) and the second column includes timestamps in seconds.

Tcpdump with RDMA packet sniffer on can capture enough packets to generate the results. ConnectX-4 NIC can generate 1 packet per microsecond. ConnectX-5 NIC can generate 3 packets per microsecond. We can find some small improvements in hardware ability between the NICs of the two generation. Both are capable of our DCQCN+ algorithm.

6.2 Bandwidth Condition for Small-Scale Incast

At the beginning, we are not sure about our switch's ability, so we conducted some small-scale incast to test the switch and our configurations. This can be called a duplicate of the original DCQCN. The topologies are similar to Figure 4 but the number of receiver hosts vary. All the interface speeds are set to 10Gbps.

Here are the examples of several experiments.

In Figure 6, it's a 2:1 incast with flows from 2 machines.

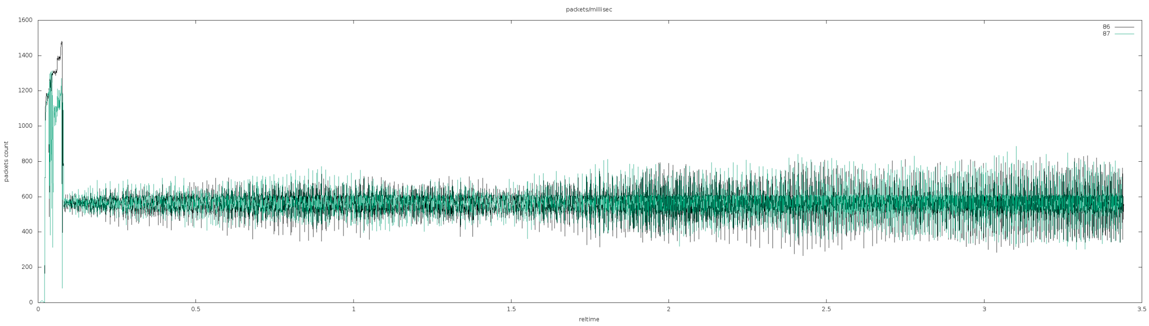

Figure 6: 2:1 incast bandwidth

We can see, at the beginning of the flows, there's a line-rate start and the speed swiftly climbs to over 10Gbps. After that, CNPs start working and both flows begin to limit sending rate and later within 0.5s, their speeds converge about 600 packets per millisecond, which is about 5Gbps.

In Figure 7, it's a 10:1 incast with 10 flows coming from 10 hosts.

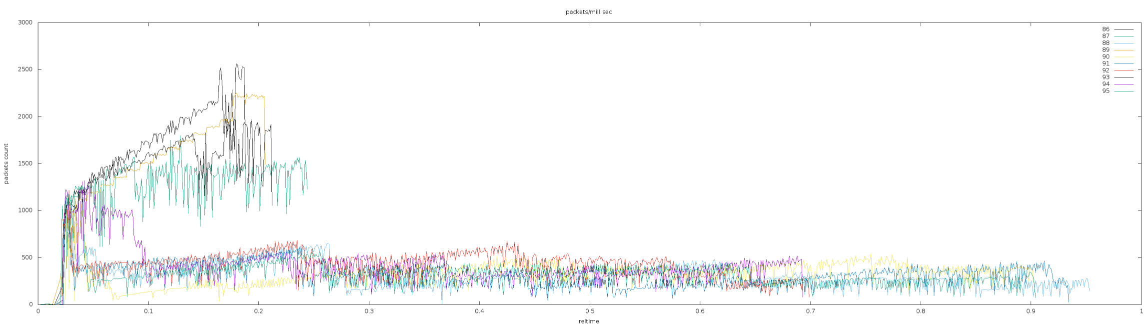

Figure 7: 10:1 incast bandwidth

As we can see, the beginning parts are similar with 2:1 incast, but several flows fails to converge. This is actually because of the Tcpdump covering range. I set a limit for number of packets captured by Tcpdump for each flow and that stops the figure of showing later bandwidth.

Later we try to draw bandwidth figures for more flows on the same machine. We use 3 machines with each sending 20 flows at the same time and we get this in Figure 8.

Figure 8: 60:1 incast bandwidth

Although I have extended the capturing range a lot, the information can be grabbed from this figure is not much. However, it's still obvious that the 60 flows converge within 0.5 second. We should admit that the original DCQCN works for small-scale incast.

6.3 Latency and Buffer Usage of Large-Scale Incast

As the flow number grows larger, we find it hard to depict the bandwidth tendency using pictures. There are mainly 2 reasons: too many lines override each other and we can't observe the tendency clearly, long capturing time makes the processing and figuring time extremely long (some may even last hours).

So we transfer to use latency and queue length to take a look at the large-scale incasts. We are sending traffic for 80 seconds and pick 7 samples for each running starting from 10 seconds after flows start. We execute 9 experiments, with each creating 80, 160, 240, ..., 720 flows. That is, with 8 senders, each of the senders creates 10, 20, 30, ..., 90 flows.

With default paremeters coming from DCQCN, we get Figure 9.

Those columns marked from 86 to 95 are average latency for flows on each machine. 86 to 95 are machine numbers. I also use scripts to collect number of flows that are successfully started to make sure that all flow start properly.

Then we apply some parameter changes to DCQCN+. To be more specific, they are listes as follows:

flownum	starting buffer	maximal buffer after 5s	86(us)	87(us)	88(us)	91(us)	92(us)	93(us)	94(us)	95(us)	avg lat(us)	flow started	buf1	buf2	buf3	buf4	buf5	buf6	buf7
8*10	457.5K	130.1K	24.63	24.83	24.89	24.9	24.86	24.81	24.71	25.16	24.84875	80	11.2K	39.1K	43.6K	0B	0B	1.0K	57.2K
8*20	4.9M	168.4K	31.81	31.75	31.75	31.74	31.79	31.69	31.97	31.78	31.785	160	22.1K	3.0K	0B	30.2K	2.1K	53.7K	61.4K
8*30	5.0M	195.4K	32.86	32.95	32.9	32.89	32.87	32.87	32.99	32.92	32.90625	240	43.5K	18.7K	105.7K	3.0K	71.9K	0B	105.2K
8*40	5.0M	267.4K	33.59	33.79	33.59	33.71	33.68	33.85	33.81	33.65	33.70875	320	864B	0B	6.4K	0B	14.4K	7.7K	77.6K
8*50	5.0M	388.9K	39.73	39.57	39.54	39.89	39.72	39.84	39.94	39.55	39.7225	400	6.6K	64.1K	176.6K	283.5K	277.7K	104.6K	86.4K
8*60	5.0M	591.4K	64.26	64.06	63.8	63.53	63.86	63.97	63.85	63.3	63.82875	480	469.1K	6.4K	248.6K	0B	0B	392.3K	96.8K
8*70	5.1M	858.0K	73.92	74.01	74.02	73.79	73.84	74.31	74.17	73.96	74.0025	560	410.6K	394.7K	135.0K	7.5K	0B	682.9K	58.7K
8*80	5.1M	1.2M	104	102.98	102.65	103.52	102.64	106.26	103.9	104.51	103.8075	640	232.9K	0B	961.9K	568.1K	15.8K	518.6K	702.0K
8*90	5.1M	5.1M	2300.02	1662.54	1612.22	1615.82	1613.06	1610.73	1620.38	2307.92	1792.83625	720	4.4M	4.9M	4.5M	1.3M	4.1M	2.0M	4.6M

Figure 9: DCQCN buffer usage and latency

1. DcQcnRateReduceMonitorPeriod, which the time period between rate reductions, is changed from $4\mu s$ to $40\mu s$.
2. DcQcnTimeReset, which is the time period between rate increase events, is changed from $300\mu s$ to $3.1 * flownum\mu s$.
3. DcQcnAiRate, which is the rate increase step in Active Increase phase, is changed from $5Mbps$ to $\frac{256}{flownum}Mbps$.

These are the only three parameters modifiable for our new algorithm and should contain more modifications if really applied to chips.

Thus we get Figure 10.

flownum	starting buffer	maximal buffer after 5s	86(us)	87(us)	88(us)	91(us)	92(us)	93(us)	94(us)	95(us)	avg lat(us)	flow started	buf1	buf2	buf3	buf4	buf5	buf6	buf7
8*10	1.0M	189.0K	27.56	27.06	27.24	26.92	26.92	27.39	27.26	27.27	27.2025	80	72.9K	90.0K	133.7K	88.9K	26.7K	27.8K	13.5K
8*20	4.7M	191.2K	31.08	31.12	31.11	31.04	31.4	31.16	30.93	30.93	31.09625	160	57.4K	116.6K	45.5K	91.9K	13.5K	43.4K	50.3K
8*30	4.8M	122.6K	29.96	29.94	29.89	29.95	29.9	29.9	29.88	29.89	29.91375	240	65.2K	13.3K	47.8K	35.2K	29.4K	87.3K	33.5K
8*40	4.7M	117.9K	30.05	30.02	29.92	29.97	29.93	29.95	30.17	29.94	29.99375	320	38.2K	15.5K	33.0K	0B	42.4K	35.2K	20.7K
8*50	4.9M	130.5K	30.04	30.09	30.44	30.31	30.04	30.04	30.18	30.09	30.15375	400	8.8K	48.9K	12.0K	33.2K	18.7K	35.9K	43.3K
8*60	4.9M	169.9K	30.32	30.34	30.69	30.73	30.3	30.29	30.35	30.52	30.4425	480	32.6K	60.1K	52.5K	61.5K	28.7K	15.9K	31.9K
8*70	5.0M	211.5K	30.79	30.75	39.73	31.03	30.94	30.94	30.87	30.76	31.97625	560	26.8K	56.0K	49.5K	70.9K	39.1K	43.7K	68.4K
8*80	5.0M	230.6K	32.19	31.64	31.65	31.95	31.66	31.63	31.58	31.62	31.74	640	8.6K	2.1K	86.6K	25.6K	71.8K	47.4K	121.5K
8*90	5.0M	681.8K	57.28	57.37	57.22	57.91	57.34	57.31	53.99	57.43	56.98125	720	99.0K	497.2K	0B	105.8K	399.4K	510.8K	517.5K

Figure 10: Simulated DCQCN+ buffer usage and latency

We can see obvious difference between Figure 10 and Figure 9. It's apparent that modified parameters provides shorter latency time and less buffer usage. That marks the success of convergence and also demands reached for datacenter networks.

To take a closer look at the difference between the original DCQCN and DCQCN+. We get the buffer usage difference in Figure 11 and latency difference in Figure 12.

To be clear, the buffer we collected is the maximal buffer usage 5 seconds after flows are started. And in Figure 12 the latency is depicted in exponential scaling to make the difference clearer.

We see that at the small scales, DCQCN+ seems to be worse than DCQCN. That's actually the reason caused by parameter scaling. The parameter scaling provided by Mellanox interface can't handle such accurate work, so I just pick the nearest integer, which possibly cause the small disadvantage.

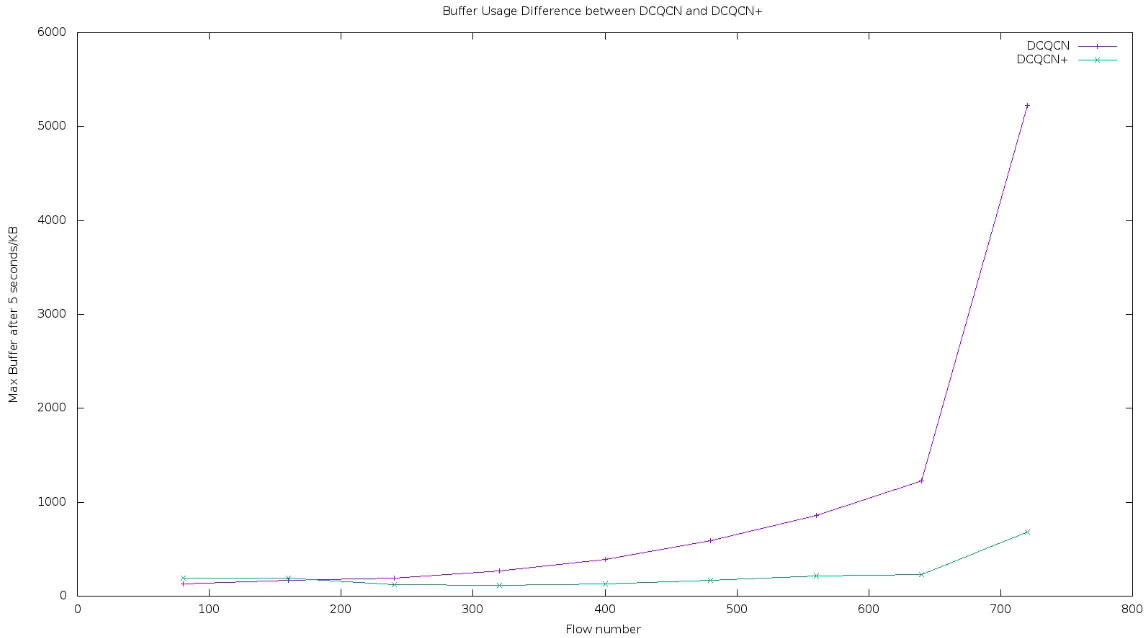

Figure 11: Buffer usage difference between DCQCN and DCQCN+

From Figure 11 we can know that the maximal buffer allocated is 5.1MB and when the flow number reaches 720, the convergence completely fails. Actually for most cases if the buffer usage maintains over 200KB, that's a failure of convergence. Most large-scale incast is over the ability of DCQCN but can be controlled by DCQCN+.

7 Conclusion and Future Work

We have displayed the drawbacks of DCQCN. When the network encounters large-scale traffic like an incast, DCQCN is forced to trigger PFC for a long time. Such congestion control causes problems of high latency and buffer occupation. We also conducted experiments to observe its features. Then we develop DCQCN+, which does a better job in handling large-scale incasts and has similar performance with small incasts. With adaptive parameter adjustments, DCQCN+ actually dynamicly adjust congestion control scheme according to the incoming traffic. We can see that DCQCN+ really does much better in latency and buffer usage.

Next steps of similar work should focus more on credit-based method of congestion control schemes like [10]. State-of-the-art methods do have better performance at the starting point. However as the scale of traffic and data storage grows larger,

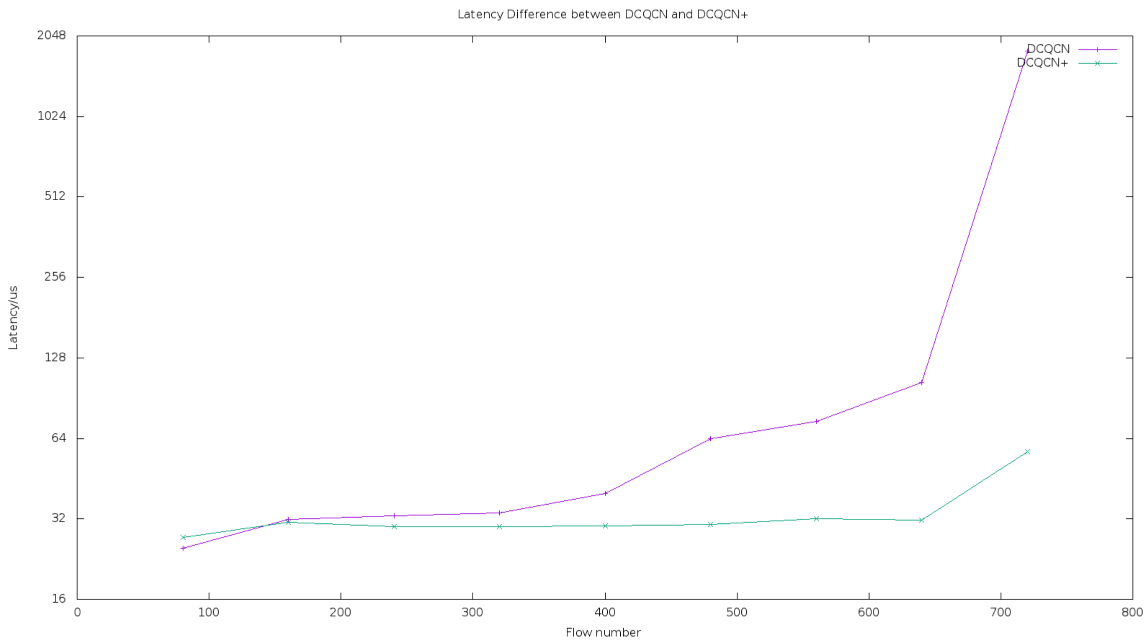

Figure 12: Latency difference between DCQCN and DCQCN+

bandwidth never becomes a problem. People care more about latency and user experience. The general case is that, the growth of bandwidth is much faster than that of processing ability and buffer size. What we are pursuing in the future is to explore in different schemes hoping to find a mixture of method to relieve or even eliminate congestion.

Aknowledgement

I greatly thank my advisor Prof. Chen Tian for his instructions and help all these years. I also thank Dr. Yixiao Gao for instructing me on experiments and his great efforts on NS3 simulations. Additionally I need to thank Bo Li and other members in NASA group for their help in switch configurations and experiment executions. I also appreciate Huawei Co. Ltd and Mellanox for their help during the process of the project. Thanks for the support from my family, friends and other people who give me instructions and suggestions.

References

[1] K. Ramakrishnan, S. Floyd, and D. Black, "The addition of explicit congestion notification (ecn) to ip," *RFC, no. 3168*, no. 2, p. 238, 2001.

[2] *802.1Qaz - Quantized Congestion Notificaction*, http://www.ieee802.org/1/pages/802.1au.html, IEEE DCB.

[3] *802.1Qbb - Priority-based Flow Control*, http://www.ieee802.org/1/pages/802.1bb.html, IEEE DCB.

[4] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," *Acm Sigcomm Computer Communication Review*, vol. 45, no. 4, pp. 523–536, 2015.

[5] 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 2010 Conference*, 2010, pp. 63–74.

[6] "The ns3 simulator." [Online]. Available: https://www.nsnam.org/

[7] "Mellanox dcqcn parameters." [Online]. Available: https://community.mellanox.com/docs/DOC-2790

[8] *RoCEv2*, https://cw.infiniband.org/document/dl/7781, Infiniband Trade Association, 2014.

[9] "Tcpdump." [Online]. Available: http://www.tcpdump.org/

[10] D. Han, D. Han, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 239–252.