# An Investigation into Data Center Congestion Control with ECN

Randall R. Stewart[*] Michael Tüxen[†]
George V. Neville-Neil[‡]

February 24, 2011

**Abstract**

Data centers pose a unique set of demands on any transport protocol being used within them. It has been noted in Griffen et.al. [1] that common datacenter communication patterns virtually guarantee incidents of incast. In Vasudevan et.al. [2] solving the incast problem involved reducing the RTO.min (the minimum retransmission timeout) of the transport protocol, but this failed to alleviate the root cause of the problem, switch buffer overflow. Alizadeh et.al (DC-TCP) [3] address the same problem with thought given to using ECN [4] with a new algorithm to not only eliminate incast but to also reduce switch buffer occupancy, thus improving both elephant and mice flows within the datacenter. In this paper we attempt to revisit some of the DC-TCP work with a few differences, namely:

1. Instead of using only TCP [5] we separate the external transport protocol from the internal datacenter protocol. To achieve this separation, we use SCTP [6] instead of TCP for the internal datacenter communication, giving us more flexibility in the feature

[*]Huawei Inc., 2330 Central Expressway, Santa Clara, Ca, 95050, USA. e-mail: randall@lakerest.net.

[†]Münster University of Applied Sciences, Dep. of Electrical Engineering & Computer Science, Stegerwaldstr. 39, 48565 Steinfurt, Germany. e-mail: tuexen@fh-muenster.de.

[‡]Neville-Neil Consulting, 822 10th Avenue, New York, NY, 10019, USA. e-mail: gnn@neville-neil.com.

set available to our internal datacenter, and at the same time assuring that changes within the transport stack internally will not adversely effect external communications on the Internet itself.

2. When attempting to reproduce some of DC-TCP findings, we will use existing switch products to provide the appropriate ECN marking.
3. Instead of using the DC-TCP algorithm we have defined a less compute intensive modification to ECN we call Data Center Congestion Control (DCCC), implementing it within the FreeBSD SCTP stack.
4. We compare four variants of SCTP: standard SCTP, SCTP with ECN, SCTP with DCCC and an alternate form of DCCC we call Dynamic DCCC. This version of DCCC is capable of switching between regular ECN and DCCC based on the initial Round Trip Time (RTT) of the path.

## Keywords

Data Center Congestion Control (DCCC); Stream Control Transmission Protocol (SCTP); ECN; Transmission Control Protocol (TCP).

## 1 Introduction

At the IETF in Beijing in the fall of 2010 M. Sridharan presented some of the DC-TCP results. This was the original impetus that led to the development of this paper. In reviewing their material, several questions seemed unanswered by their work and encouraged this investigation.

1. Why was plain ECN not used in the DC-TCP work? ECN itself was mentioned but no real measurements were performed contrasting DC-TCP and plain TCP with ECN.
2. The algorithm defined by DC-TCP seemed somewhat complex, using both floating point numbers in its calculations and also introducing another state machine to track acknowledgement state. Could there be a simpler method than prescribed that would obtain similar results?

3. Could existing hardware, readily available from vendors, be used to reproduce these results? The DC-TCP paper explicitly mentions a Broadcom Triumph, Broadcom Scorpion, and Cisco CAT4948 [1] switch, but what is the real availability of a switch off the shelf to support ECN?

4. Could it be beneficial to use one protocol within the datacenter and a different one for accessing the datacenter? This would allow isolating any congestion control changes within the datacenter while not effecting internet traffic.

5. SCTP seems to be a perfect candidate for use within a datacenter.

    - It provides a rich set of additional features not available in TCP (for example it can automatically discard old data after a specified time).
    - SCTP also provides multi-homing and CMT [7] which could prove useful inside a datacenter.
    - SCTP's ECN implementation is more conducive to knowing when each packet is marked without changing any internal ECN mechanism or maintaining a small state machine to track acknowledgment state. This would make implementation both simpler and more efficient.

After considering these facts, it was decided to put together a 'mini-datacenter' to simulate the kinds of traffic highlighted in the DC-TCP work.

# 2 The Testnetwork

## 2.1 Gathering the Network Equipment

At first glance one would think gathering the necessary network equipment in order to do DC-TCP like experiments would be easy:

1. Find several switches that can perform ECN and configure them to do ECN on an instantaneous basis.[2]

[1]The CAT4948 was explicitly mentioned as NOT supporting ECN.

[2]Normally ECN is set to use average queue size for its decision to mark.

2. Setup the appropriate set of switch parameters as to when to start marking.

Finding a switch that supports ECN proved surprisingly difficult. There are chipsets like the Broadcom BCM56820 and BCM56630 that natively support ECN in hardware, but the software available from vendors that use these chipsets did not have the the knobs to enable ECN. After looking at various CLI configuration guides and talking with switch manufacturers, the only switch with documented ECN support to be found was the Cisco 4500 series. Thus, instead of having several switches to use for experimentation, only one was available. A Cat 4500 is really a "big buffer" switch. This posed additional problems in setting up a configuration that would allow us to do DC-TCP like tests.

Our final lab configuration can be seen in Figure 1.

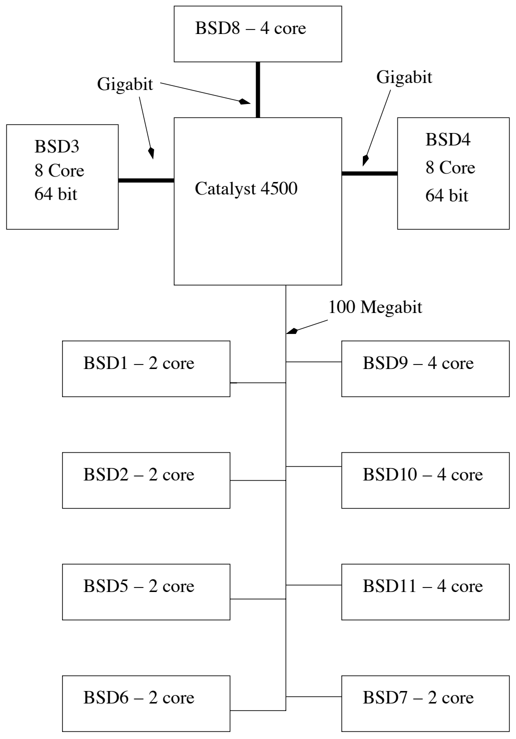

Figure 1: Final Laboratory Configuration

## 2.2 Caveats when using the Cisco 4500

We configured the switch in a way that would generate either drops or ECN with using the Data Buffer Limiting (DBL) feature. DBL allows ECN or Random Early Drop (RED) like features to be enabled on transmit queues within the switch.

After some initial problems finding a software version that would support the features we wished, we finally settled on an older 12.2(31)SGA9 – Enterprise Release. This version was the only one of those tested that would generate ECN marked packets.

The DBL feature on a CAT 4500 proved less configurable than desired. There are four basic settings that control the ECN feature.

1. Maximum Credits - this value represents how many credits to issue to a flow.[3] The value was configurable between 1–15. A credit comes in to play only when the switch decides it is congested. At the point the switch does decide it is congested, it then issues this number of credits to all identified flows[4]. Every additional packet a flow adds to the outgoing transmit queue decrements the number of credits it has available. If the flow does *not* respond by reducing its rate of sending, then it may reach the point where it reaches the 'Aggressive Credits' threshold, where it will be treated as an aggressive (non-responsive) flow.

2. Aggressive Credits — This value represents when a flow that is not responding begins to be considered as a non-responsive flow. The aggressive credits value can be between 1–15.

3. Aggressive Buffers — This is the maximum number of packets a classified non-responsive flow may have outstanding in the transmit queue. This value was configurable between 0–255. Any flow having more than this many packets in a queue that is considered aggressive will have new packets dropped.

4. Exceeds Action — This item can be set to either use ECN or not. When not using ECN, the switch signals with a packet drop. With ECN on, flows supporting ECN will mark the packet instead of dropping it.

[3]Either identified via IP addresses or IP addresses and layer 4 ports.

[4]In the case where layer 4 ports are not used then a flow is considered any packets between a source and destination IP address.

5. Exceeds Action Probability — This is a percentage of random times that the Exceed Action will happen (i.e. a packet will be dropped or marked).

When first examining the switch configuration, it appeared that just by setting some reasonable values in these control fields we would obtain ECN marking or drops. A first attempt was made on a Gigabit link using TCP[5], but our initial incast experiments with the eleven servers we had resulted in no DBL activity drops or ECN markings. Further investigation into the switch showed us that each Gigabit link has 1920 packet buffers dedicated to it on the transmit side[6], and the receiver side is completely non-blocking. Evidently, the threshold to obtain enough activity inside any given transmit queue was high enough that many more flows would be necessary than our small datacenter could provide with its eleven multi-core FreeBSD servers.

As an alternative, most of our servers were moved to the 100 Megabit interfaces that were also available on the switch. These interfaces only have 240 packets in their transmit queues, giving a more easily overtaken set of transmit buffers. Three of the servers, two eight core and one four core, were left on Gigabit interfaces to provide higher amounts of traffic into the remaining eight servers. With this configuration we could finally obtain DBL markings and drops on a reliable basis. With the help of wireshark[8] it was established that the switch would not enter a "congested state" and start marking until at least 120–130 packets were queued by a single flow. Two flows appeared to get the switch marking around 70–80 packets from each flow.

We were now left with a dilemma. In the DC-TCP experiments, the switch was configured to mark whenever a flow took more than N packets: N was set to 20 for Gigabit links and 65 for 10 Gigabit links. Our switch would not mark packets until more than 100 packets were in the queue. Without the proper controls over when the switch would start marking packets, how could the switch be configured to approximate the circumstances used in the previous work?

The solution was to configure the switch to have a maximum number

[5]We used TCP in our initial tests to validate that the switch would do ECN since we were unsure, at first, if SCTP would be properly marked. Later we switched to SCTP to do all of our measurements after confirming that the switch would correctly mark SCTP packets.

[6]There is no configuration setting that allows us to lower this value.

of credits (15), the fall-over to an aggressive flow at a low value (5), the probability of exceeds action to 100% and the exceeds switch buffer to the maximum to 255. With this configuration, marking would happen as noted earlier around 120–130 packets, and in the worst case (with two flows) around 75 packets, or 150 total packets outstanding. This then would leave somewhere between 90–120 packets available before the transmit queue would overflow.

It was felt that this configuration would be closer to what could be obtained with a more configurable switch with less buffer space (i.e. somewhere in the range of a switch with 100 packet buffers available configured to start marking packets around 10). The only downside to this configuration was that with 130 or more packets in queue[7] the round trip time would increase to up to 14–15ms. Though not as flexible as the configuration presented in the DC-TCP papers, it was definitely a more realistic configuration with the only off-the shelf switch we could find supporting ECN.

## 3 Targeted Communication Pattern

In examining the results of DC-TCP it was decided to focus specifically on the interactions between the elephant flows; in a datacenter those flows updating databases on the order of a constant one or two flows sized randomly at about 100–500 Megabyte per flow, and the mice; partition/aggregation flows that typically can cause incast. In the following subsections we describe each client and server that was created for our experiments and which is available online at `http://www.freebsd.org/~rrs/dccc.tar.gz`.

### 3.1 Partition/Aggregation Flows

Incast occurs when simultaneous sends by a number of hosts cause buffer overflow within a switch. For a detailed look at incast see R. Griffith et.all [1]. A partition/aggregation flow is one that sends a request to multiple servers, and all of the servers respond as rapidly as possible, usually with a small number of packets. The requestor then aggregates this data and sends it back to the external requestor. Since most of the aggregate servers will respond at almost the same time the partition/aggregation work model precisely fits a description of incast.

[7]Before marking began the round trip time normally around 115us.

To simulate incast on our testbed we built a client (`incast_client.c`) and a server (`incast_server.c`), which do the following:

1. Open a socket and bind to a specific well known port and address[8]. After creating and binding the socket issue a listen setting the backlog to X[9].

2. Create N[10] additional threads, besides the main thread.

3. Each thread, including the main thread, loops forever accepting from the main listening socket.

4. When a connection is accepted, a simple structure is read from the client connection that includes two fields, the number of packets (or sends) to make, and the size of each send.

5. After receiving the request, the thread then sends the requested number of packets at the set size and then closes the socket, returning to the accept call.

The server was always bound to the single address that our CAT 4500 was connected to the hosts over[11] see Figure 1. The server also accepted one other option, whether to run over SCTP or TCP, allowing us to use the same server in both our early testing, validating the switches configuration for ECN, and our final tests using SCTP.

The clients were a bit more complicated. It was decided to build a universal configuration file that could be reused to describe all of the peers that any particular machine had. An example of the file from 'bsd3' is shown below:

```
sctp:on
peer:10.1.5.24:0:bsd1:4:
peer:10.1.5.23:0:bsd2:4:
peer:10.1.5.22:0:bsd5:4:
peer:10.1.5.21:0:bsd6:4:
```

[8]The multi-homing behavior of SCTP was specifically disabled by binding to only a single address.

[9]We used a value of 4 for the backlog value X in our tests.

[10]N in our testbed was set to 10.

[11]In our case 10.1.5.X.

```
peer:10.1.5.20:0:bsd7:4:
peer:10.1.5.19:0:bsd8:4:
peer:10.1.5.18:0:bsd9:4:
peer:10.1.5.17:0:bsd10:4:
peer:10.1.5.16:0:bsd11:4:
peer:10.1.5.15:0:bsd4:8:
times:0
sends:1448
sendc:2
bind:10.1.5.14:0:bsd3:8:
```

All fields within the file are separated by the `:` character. The first line in the example contains either a `sctp:on` or `tcp:on`, which controls the transport protocol the incast_client will use.

The peer line is made up of the keyword `peer` followed by the IP address of a machine running the server. The next field indicates the port number the server is listening on, where 0 indicates that the default value for the server should be used[12]. The next field represents the hostname of the peer, a value that is used only by some of the data processing routines. The final field represents how many bytes a long holds; and was used by our data processing routines to determine what record format to read[13].

The line containing the keyword `times` indicates how many times to run the test, a value of 0 means run forever.

The keyword `sends` indicates how many bytes to send. We always elected 1448 since this is the number of bytes that will fit in a typical timestamped TCP packet, SCTP is actually capable of fitting 1452 bytes, but it was decided to keep the same exact configuration without changing the number of bytes in each packet.

The keyword `sendc` specifies how many packets the client should request of the incast_server.

The final line indicates the actual address and port to bind. The format is the same as that of a `peer` entry with the only difference being a `bind` keyword.

Once the configuration file is loaded (passed to the incast_client program with the required `-c` option), the process creates a kqueue, which it uses

[12]For the incast client the default server port was 9902.

[13]32 bit machines write the value for a time as two 4 byte longs where as 64 bit machines write the value as two 8 byte longs.

to multiplex responses from servers. It then enters a loop for the requested number of times (which may be infinite) and does the following:

1. Record the start time in a special header record using the precise real-time clock.

2. Build a connection to each of the peers listed in its peer list (loaded from the configuration file). Building a connection includes opening a socket to each peer and setting NODELAY[14] on it, connecting to each peer, and adding an entry into the kqueue to watch for read events from the peer.

3. After all the connection were built, another precise real time clock measurement was taken.

4. Every server is now sent a request for data, and then its state is set to 'request sent'.

5. After all the servers were sent the request, the kqueue was watched until all of the servers responded with the prescribed number of requested bytes. When the first byte of a message is read[15], a precise monotonic clock time is taken and stored with the peer entry, followed by its state being set to 'reading'. When the last byte of a request arrives another precise monotonic clock is again taken and stored with the peer entry.

6. Once all of the peers have responded, a final precise realtime clock is taken and stored in the header record with the number of peers.

7. The results are either displayed or stored based on whether the user specified an output file on the command line. If the results are being displayed, then only peers that responded in more than 300ms are shown. If the results are being stored, the header record is written first, followed by a record for each of the responding peers with their timings.

8. This sequence is repeated until the specified number of iterations is reached.

[14]NODELAY turns any Nagle[9] like algorithm off.

[15]Indicated by the state being 'request sent'.

## 3.2 Elephant Flows

To simulate datacenter like traffic, a small number of large flows are also needed. In a real datacenter, these flows would provide updates to the various databases that all the query traffic is attempting to access.

We realized these flows using a specific server (`elephant_sink.c`) and a corresponding client (`elephant_source.c`). In these measurements, each machine ran a server; but only two machines were chosen to run a client. The hosts that ran the elephant_source (bsd3 and bsd4) were our two, eight core, 64 bit machines, connected to the switch via a Gigabit ethernet port. This allowed the elephant flows to always push at filling the transmit queues of the 100 Megabit clients. One additional host (bsd8) was also connected via Gigabit ethernet to provide an additional push on transmit queues while running the incast_client. Note that the elephant flows that were sent to this additional host (bsd8) were not used in measuring overall bandwidth obtained in our datacenter simulation.

The elephant_sink worked much like our incast_server, creating a number of threads, defaulting to 2, that would do the following after binding the specified address and port numbers:

1. Accept a connection from a peer.
2. Record both the precise monotonic time and the precise realtime clock.
3. Read all data, discarding it but counting the number of bytes read, until the connection closed.
4. Record again the precise monotonic time and the precise realtime clock.
5. Subtract the two monotonic clock values to come up with the time of the transfer.
6. Display the results either to the user, if no output file was configured, or save it to the output file. The results include the size of the transfer, the number of seconds and nanoseconds that the transfer took, and the bandwidth. Note that when writing to an output file a raw binary record was used that also included the realtime clock start and stop values.
7. Repeat this sequence indefinitely.

The elephant_source used the same common configuration files that were used by the incast_client program. After loading the configuration our elephant_source would first seed the random number generator (by using the monotonic precise clock), and then do the following:

1. Choose a random number of bytes in the range of 100,000,000 and 536,870,912.[16]
2. Record in a header the precise realtime of the pass start.
3. Distribute the data to each peer, to do this we would connect to each peer, turning on NODELAY[17], record the precise monotonic clock with the peer record, send the specified number of bytes and close the connection. After closing the connection again, record the end monotonic precise time.
4. After distributing the data to all peers we would again obtain the realtime precise clock end time for our header record.
5. Next the results would be displayed to the user or stored. When storing the results, a header record would be written out with all of the timestamps followed by the precise times of each of the peers.
6. These steps were then repeated until the iteration count was reached (which was also allowed to be infinite).

When configuring the two elephant sources, we reversed the client list order so that on a random basis they met somewhere in their distribution attempting to send data to the same elephant_sink at various times during their transfers. During data analysis the elephant source file was used to first read the results, but the recorded timestamps of the elephant_sink files were used to obtain the precise time of transfer (first byte read to last byte received). Any data transfer bandwidth to either bsd3, bsd4, or bsd8 was not included in our bandwidth graphs, since these three hosts had Gigabit interfaces which would unduly skew the transfers.

With our pairs of clients and servers completed and tested we were almost ready to begin our experiments. But first we needed to examine SCTP

[16]This number was arrived at as a mask to bound the random number since when you subtract one the number becomes 0x1fffffff.

[17]NODELAY turns any Nagle[9] like algorithm off.

congestion control when handling ECN events and modify it according to our simplified algorithm.

## 4 Our Data Center Congestion Control Algorithm

The algorithm used by SCTP for ECN provides a lot more information than the standard TCP ECN. This is due to the fact that SCTP is not limited to two bits in a header to indicate ECN and so provides a much richer environment for its implementation. The normal response of SCTP to an ECN Echo event is the same as TCP in any loss within a single window of data:

```
ssthresh = cwnd / 2;
if (ssthresh < mtu) {
  ssthresh = mtu;
  RTO <<= 1;
}
cwnd = ssthresh;
```

Figure 2: Algorithm 1

When one or more losses occur in a single window of data, the cwnd is halved and the ssthresh is set to the same value. When an ecn_echo is received, the congestion control algorithm is given two extra pieces of information besides the stcb and the net structures[18].

These two extra pieces of information are the keys to our new DCCC algorithm, they are the in_window flag as well as the number of packets marked. The in_window flag is used by the current algorithm to know if it should perform the algorithm shown in Figure 2, and the number of packets marked is ignored by the existing algorithm. For our new algorithm we use both of these values combined with another value passed to us via the network structure (net).

[18]The stcb and net structures are internal structure used within the FreeBSD SCTP stack to track the specific SCTP association (stcb) and the specific details of a peers destination address (net).

Each time a SCTP chunk is sent in a packet, we also record the congestion window at the time the transmission was made[19]. When an ECN Echo arrives, before calling the modular congestion control, an attempt to find the TSN[20] being reported is made. The TSN will always be found if the ECN Echo is in a new window of data. If the TSN is found, the previous cwnd at the time that the TSN was sent[21], is recorded on the net structure in a special field for ECN (ecn_prev_cwnd). These three values are then used as shown in Figure 3.

```
if (in_window == 0) {
  cwnd = ecn_prev_cwnd - (mtu * num_pkt_marked);
  ssthresh = cwnd - (mtu * num_pkt_marked);
} else {
  cwnd -= mtu * num_pkt_marked;
  ssthresh -= mtu * num_pkt_marked;
}
```

Figure 3: Algorithm 2

It is important to remember when looking at SCTP congestion control parameters that there is a separate set of parameters for each destination address (RTO, cwnd, ssthresh and mtu). In our experiments this was never an issue since SCTP was kept in strict single-homed mode by binding explicitly a single address.

As noted earlier we also used a third algorithm. We termed this version dynamic DCCC, because the algorithm would switch between normal ECN behavior and DCCC based on the initial round trip time measured on the first data packet. SCTP uses this value to try to determine if the destination address is on a local LAN. If it thinks that it is on a local LAN (i.e. having a round trip time of under 1.1ms), then it sets a flag. Our dynamic algorithm simply used that flag to switch between the algorithms shown in Figure 2 and Figure 3.

[19]The value is recorded on an internal structure used to track the chunk.

[20]A Transport Sequence Number is the unit of message transmission that SCTP uses when it sends data.

[21]Found on the chunk structure mentioned earlier.

## 5 Measurements

We first decided to examine incast and how the three algorithms (DCCC, DYN-DCCC and ECN) compared with running plain SCTP in our network. We started an incast_server and elephant_sink on each machine (bsd1–bsd11) and then started an incast_client passing it options to start collecting data. Once all incast servers were running, we started two elephant sources one on bsd3 and the other on bsd4.

### 5.1 Runtime

With the switch configured to drop, i.e. not do ecn, we then record all results and were able to obtain the plots shown in Figure 4[22].

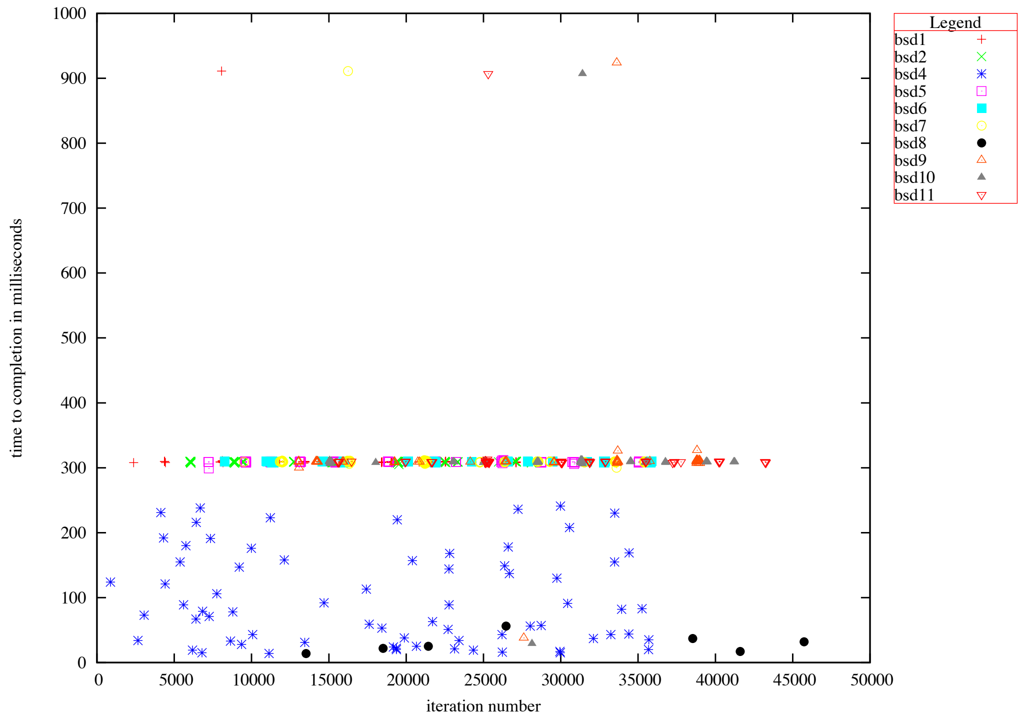

Figure 4: Normal SCTP transfers experiencing incast

In Figure 4 we see incast occurring in cases where the time to complete

[22]Time below 15ms excluded.

a transfer exceeds 300ms[23]. In some cases we see more than one retransmission. Clearly incast is occurring on a fairly regular basis. Counting the actual number of incast occurrences shows us 567 different incidents of incast represented in Figure 4.

Next it was decided to turn on ECN in the switch with no changes to the congestion control algorithm. This lead to the results seen in Figure 5[24].

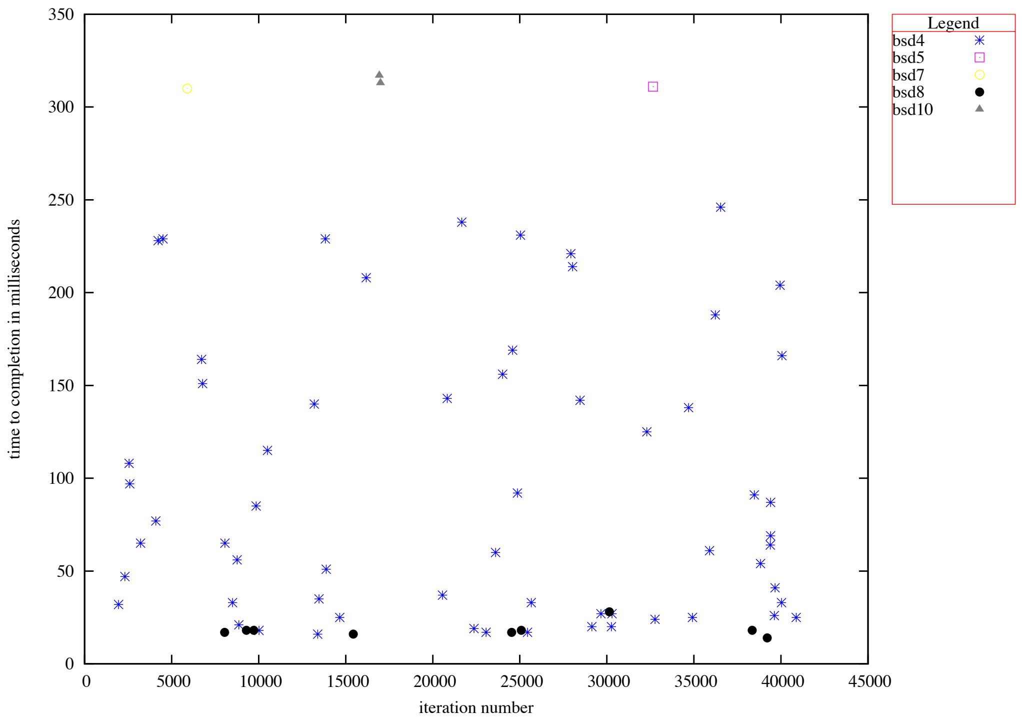

Figure 5: Normal SCTP with ECN experiencing incast

Note that simply enabling ECN on the switch with our settings reduces the incidents of incast to a minimal level. There are only 4 occurrences of incast that reach the 300ms level. This is a vast improvement over our previous figure.

Curiously, bsd4 shows quite a few delays in returning results, below the incast level, but still quite high. Since these measurements are taken from the perspective of the aggregator and bsd4 is connected via Gigabit ethernet,

[23]The default RTO.min timeout value for SCTP.

[24]Time below 15ms excluded.

clearly it must be caused by delays in the incast server receiving the initial request to transfer the two packets back to it. Most likely this is caused by a lost setup message during the connection startup.

Next, the same tests were run, but this time using our DCCC algorithm in place in the SCTP stack. The results from this run can be seen in Figure 6[25].

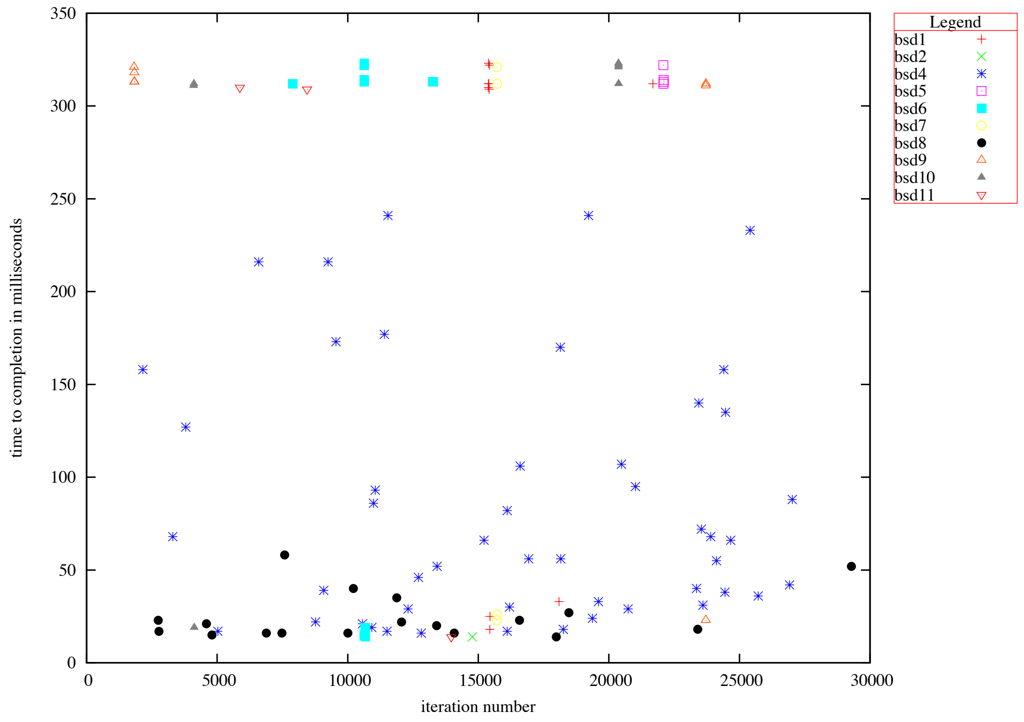

Figure 6: DCCC ECN experiencing incast

Surprisingly, the actually occurrence of incast increases in this graph to 34 separate incidents. This was unexpected, but with a look at some of our further graphs we will see possible reasons why so much unexpected incast is occuring.

Finally, we ran the tests again only this time we used our Dynamic DCCC algorithm. This allowed the SCTP stack to switch between normal ECN and DCCC ECN based on the round trip time seen when transfer of data first begins. The results of this test can be seen in Figure 7[26].

[25]Time below 15ms excluded.

[26]Time below 15ms excluded.

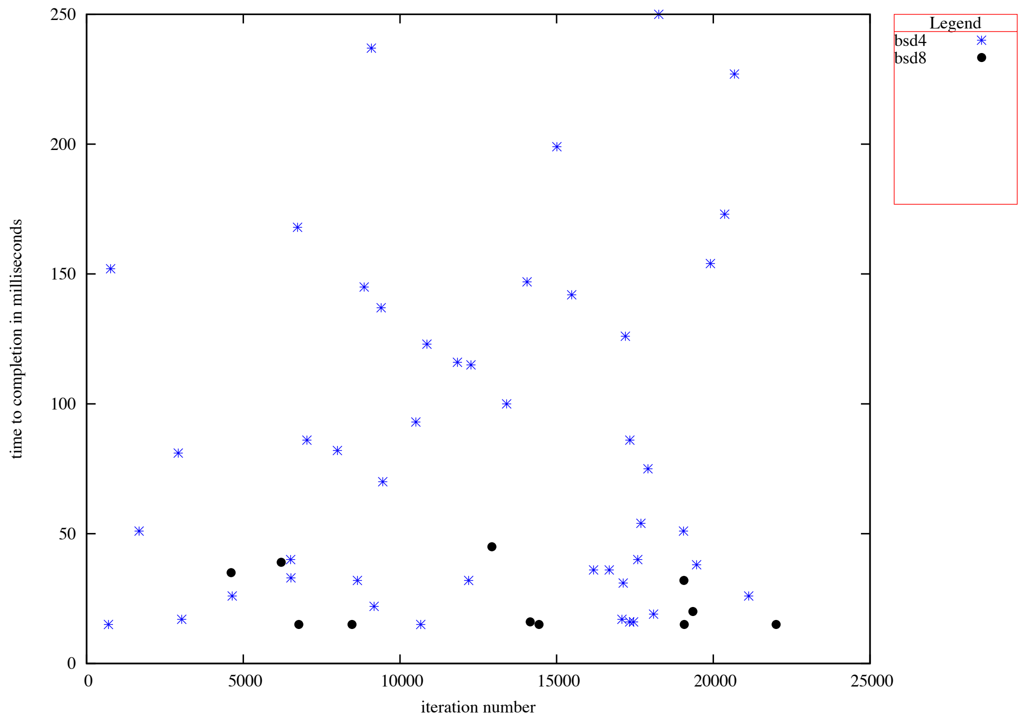

Figure 7: Dynamic DCCC ECN experiencing incast

Here we see no incidents of incast over 300ms, which is an unexpected outcome since we had anticipated no differences between DCCC (with its 34 incidents) and the dynamic version. If we turn, however, to the elephant graphs, we can reach a conclusion as to why these incidents are occurring.

## 5.2 Bandwidth of Elephant Flows

Lets look at the four graphs of the elephant transfers that were being run while all the incast was occurring.

Figure 8 shows a normal SCTP transfer with ECN disabled. Note the huge swings downward in throughput for both flows when the flows collide and share transfer to the same machine.

During these incidents, we can be sure that a large number of drops are occurring. A look at the switch information tells us that not only are DBL drops happening but also tail drops from transmit queue overflows are occurring as well. The switch reports 19,312 DBL drops and 46 Transmit queue drops. The SCTP statistics show a large number of T3-Timeouts[27] i.e. 1360. Any T3 timeout in the incast test would definitely cause an incident of incast since there is not enough data in the pipe to enable fast retransmits with only two packets being sent.

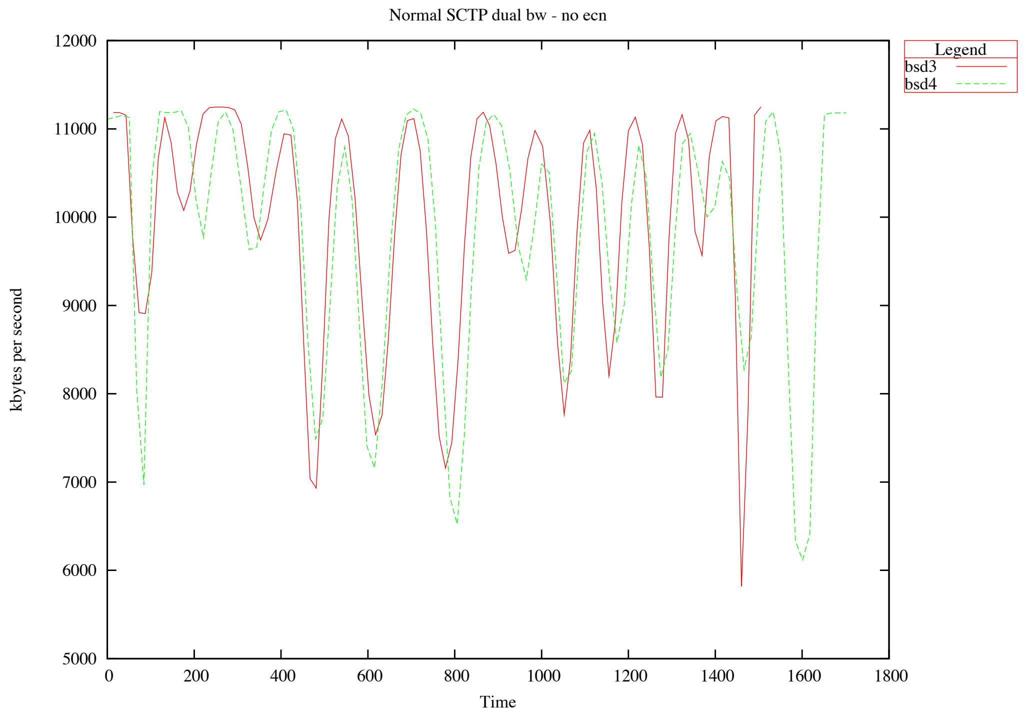

Figure 8: Bandwidth of two flows no ECN

[27]A T3-Timeout, in SCTP, is the firing of the SCTP data retransmission timer.

Figure 9 show the data transfer when ECN is enabled on the switch but no alternate congestion control is running. In this graph, we see fewer swings and a more orderly set of transfers. Checking the switch error counts we find that there occurred 1,079 DBL drops and 65 tail drops. SCTP's statistics also show 43 T3-Timeouts[28], but far from the larger number seen in the instance of no ECN.

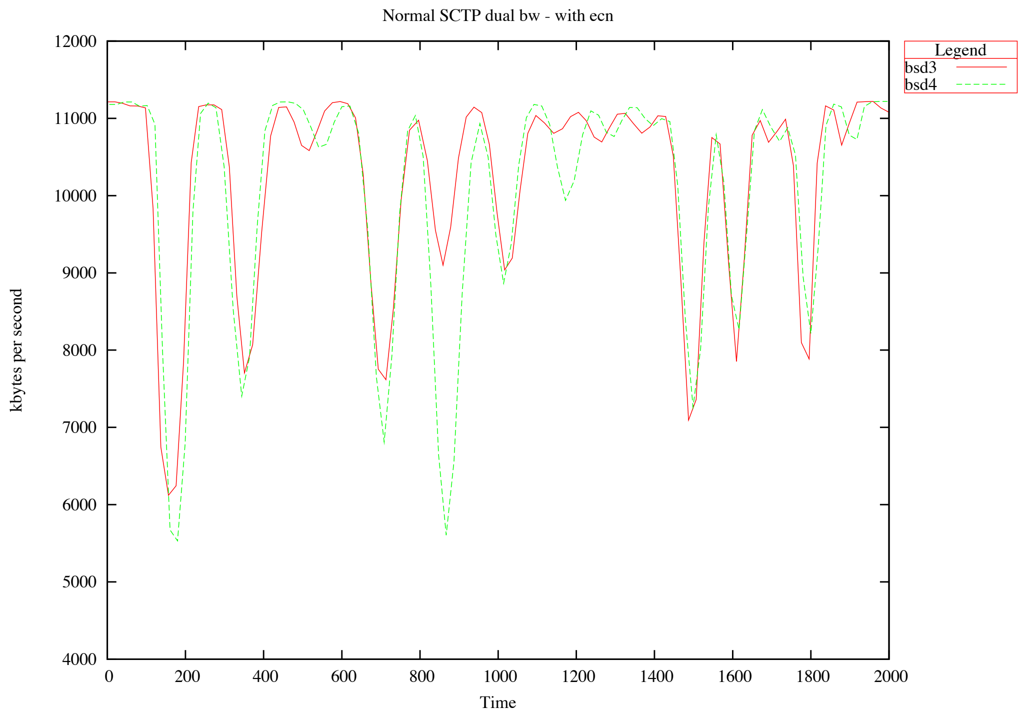

Figure 9: Bandwidth of two flows with ECN

We enable our modified congestion control with DCCC, and the bandwidth results are seen in Figure 10, showing similar behavior to ECN but with a bit better performance in overall bandwidth. The error counters on the switch, however, show 1,890 DBL drops and 1,441 tail drops. This tells us that the more aggressive congestion control, when the two large flows meet on the same host, end up overflowing the switch transmit queues. The SCTP statistics show similar results of 422 T3-Timeouts[29].

[28]A T3-Timeout, in SCTP, is the firing of the SCTP data retransmission timer.
[29]A T3-Timeout, in SCTP, is the firing of the SCTP data retransmission timer.

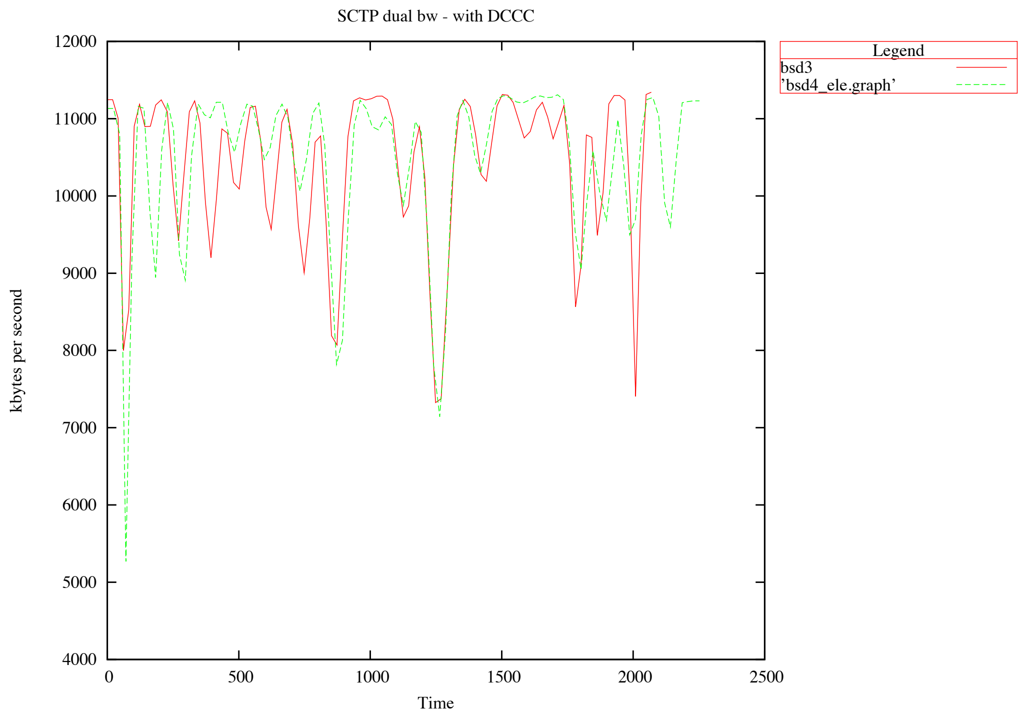

Figure 10: Bandwidth of two flows with DCCC

Finally, the bandwidth of our Dynamic DCCC is shown in Figure 11. Clearly the overall bandwidth is better than in any other of the tests. The switch error counters show that only 35 tail drops occurred and 1,028 DBL drops. SCTP statistics show only 42 T3-Timeouts[30]. This would explain the lack of incast on our earlier incast chart for Dynamic DCCC.

You can see that the large flow that arrives first on a particular host is being more aggressive, while the later arriving large flow is less aggressive. This is due to the fact that our switch will not start marking until 130 packets. At about 115us per packet this means that the round trip time must be close to 14ms by the time the second flow arrives. This pushes that flow into standard ECN mode. The combination obviously keep both flows aggregate packet load offered much smaller and thus less drops occur.

One thing that is not illustrated in any Cisco documentation is what a DBL drop means. Considering that we have the exceeds action set at 255

[30]A T3-Timeout, in SCTP, is the firing of the SCTP data retransmission timer.

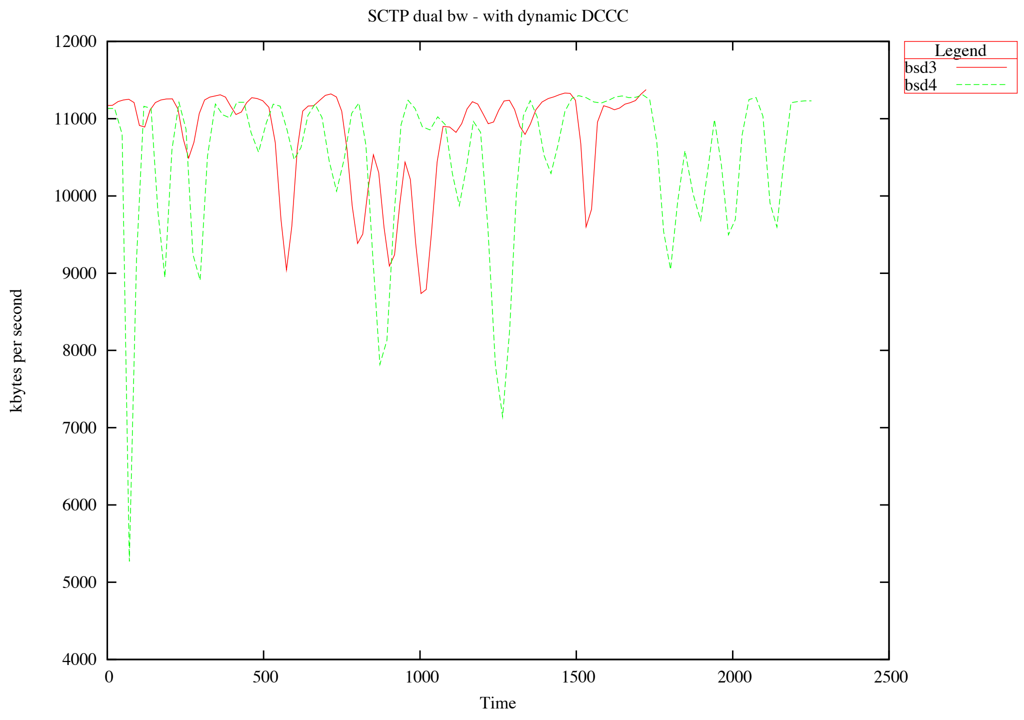

Figure 11: Bandwidth of two flows with Dynamic DCCC

packets, 15 packets above the transmit queue size, one would expect that you would only get tail drops occurring. It is important to also remember that a DBL drop would occur for any non-Data packet since the ECN code-point ECT0[31] would NOT be enabled on any packet that does not contain data. Thus our DBL drops could be lost SACK's or other connection setup packets. Without further information from Cisco, it is hard to tell if the DBL drop counter is strictly drops or if it also includes the number of ECN marked packets that would have been dropped.

## 5.3 Aggregate Bandwidth

Finally, Figure 12 shows us an aggregate bandwidth comparison of the overall throughput of the 4 different tests. Each of the two flows in the test are

[31]ECT0 is the mark used in the IP level header by a transport protocol to indicate ECN support to a switch or router.

summed together to give an overall aggregate and then these are summed to give us the average goodput of the combined flow on a per second basis.

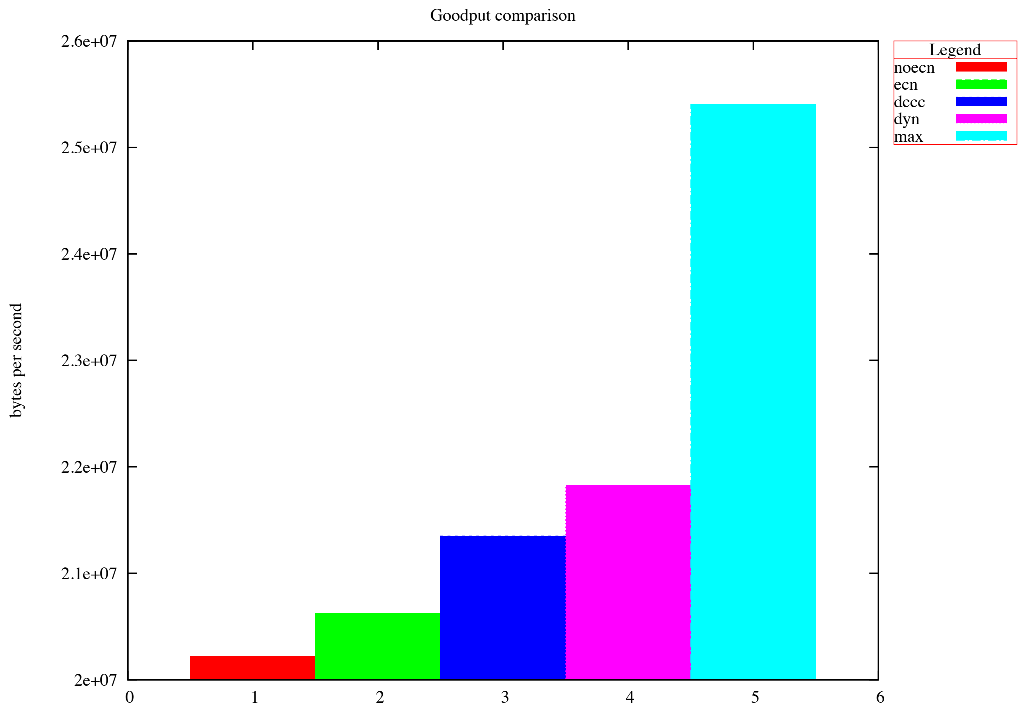

Figure 12: Aggregate Bandwidth Comparison

As expected the ECN flow was better than the non-ECN flow by approximately 2%. The pure DCCC flow increased its bandwidth by nearly 5.5% over the standard SCTP with no ECN. And as expected from our earlier graphs, the Dynamic DCCC yielded close to a 7.9% bandwidth improvement over standard SCTP. As can be seen from the graph, this is quite a distance from the theoretical maximum. Note that the theoretical maximum includes a key assumption i.e. that no two flows would be transferring to the same computer at the same time. This could only be achieved by flow coordination between the two elephant flows.

## 6 Conclusion and Future Work

After examining the data presented here, packet traces, and respective graphs from those traces[32], the most clear conclusion reached is that with *proper* configuration, enabling ECN will virtually eliminate incidents of incast in the datacenter. This is, of course, provided that the SCTP or TCP stack properly implements ECN within it.

So what should switch vendors take away from these results?

1. They should have an implementation of ECN for their datacenter switches, and having one for non-datacenter switches could also be a benefit to their products. Many of the more modern chips (e.g. BCM56820) already support ECN in hardware and its just a matter of allowing its users access to those hardware features.

2. Vendors need to allow configuration for both 'network level settings' (i.e. those classically called for by ECN) and 'Datacenter settings'. Network level settings would allow the switch to use average queue size and hopefully set thresholds to start marking and dropping in terms of percentage of queue occupied by a flow. Datacenter settings would allow a switch to be configured for observing instantaneous queue size and allow control of the number of packets to start marking and dropping at, instead of percentages of queue size.

3. Vendors need to make sure that all transport protocols can be fully supported that use ECN. This means, if a vendor supports the concept of a flow, it needs to not be restricted to just TCP, but also should include SCTP. Using the same port numbers with SCTP would be acceptable, but innovative providers could also use SCTP v-tags for this purpose.[33]

Other conclusions from this work are much less clear. The performance gain of the various DCCC algorithms was evident but these gains were limited (5.5 and 7.9%). Clearly this is an advantage, but is it an advantage worth

[32]Not shown for brevity, but tools to generate them from a wireshark trace are included in the source code drop for those wanting to recreate our results.

[33]Our observations never determined if the CAT4500 considered all traffic between IP address or individual SCTP flows since there was no way to determine if the CAT4500 understood SCTP.

changing congestion control algorithms within the transport protocol? The dynamic version is obviously most attractive since if it was implemented, it could easily exist on the internet with no impact, since it would switch itself off in cases of a longer RTT time[34].

In thinking of our limited switch configuration, if this was changed so that marking could begin at 5–10 packets would we see the same results? Clearly the dynamic DCCC would become less dynamic and might look more like plain DCCC. But would plain DCCC do better since it would no longer be hitting tail drops by overrunning the switch transmit queues? Also the plain SCTP-ECN variant may well perform less well in a switch that was more aggressively marking ECN. These questions are some of the future work that is envisioned to attempt with a more controllable switch. The authors have been informed that Broadcom offers a development kit which hopefully can be acquired at a reasonable cost and configured in such a way as to provide insights into these questions.

## Acknowledgment

Thanks to Randall L. Stewart and Anne Stewart who have been inflicted upon by their father for their comments, and special thanks to our editor, Sandra Stewart.

## Appendix A - Switch Configuration

For those interested in repeating our experiments, we show the CLI configuration of the Cisco CAT4500 switch. Note that we varied the QOS DBL exceed action between ECN and no ECN.

#### 6.0.1 QOS DBL CONFIGURATION

Here we show the basic QOS DBL configuration used. It was also not ever certain that the CAT 4500 could actually distinguish SCTP ports in separate flows.[35]

[34]We used 1.1ms but would think something smaller, perhaps 700us would be better for general deployment.

[35]This may mean that the switch always saw traffic from each machine as a single flow, but this will have no impact on our results since all measurements would be impacted

```
no qos dbl flow include vlan
qos dbl exceed-action ecn
qos dbl exceed-action probability 100
qos dbl credits aggressive-flow 5
qos dbl buffers aggressive-flow 255
qos dbl
qos
```

A show qos dbl on our CAT 4500 displays the following:

```
QOS is enabled globally
DBL is enabled globally
DBL flow does not include vlan
DBL flow includes layer4-ports
DBL uses ecn to indicate congestion
DBL exceed-action probability: 100%
DBL max credits: 15
DBL aggressive credit limit: 5
DBL aggressive buffer limit: 255 packets
```

### 6.0.2 POLICY MAP AND CLASS MAP

Here we show the specific class map and policy map used. Notice the interfaces apply a transmit to shape traffic to one Gigabit. This was suggested by a former Cisco colleague to get ECN and DBL to work. The settings are high enough for both our 100 Megabit and Gigabit links so that the shaping action would never be applied to the interface. Only the QOS DBL actions would influence the transmit queue. Shaping actions on a Cisco 4500 happen before DBL actions.

```
class-map match-all c1
 match any

policy-map p2
 class c1
  dbl
  police 1000 mbps 100 kbyte conform-action transmit
         exceed-action drop
```

(ECN and non-ECN) in the same way.

### 6.0.3 Example Interface Configuration

Here is an example of our specific interface configurations. Notice that for Gigiabit interfaces we turned off flow control in the initial attempts at trying to use these interface to get ECN markings.

```
interface FastEthernet2/48
service-policy output p2

interface GigabitEthernet3/1
  flowcontrol receive off
  service-policy output p2
```

# Appendix B - Whats in the tarball?

Here is a brief list the program files you will find in the tarball, if you download it. Many of these utilities were used to generate data not presented here, but they may be of interest to those attempting to recreate the results. All software is released under BSD license.

**display_ele_client.c** This program understands how to access stored elephant results file from a client and will further look for timing from the stored elephant_sink output results. A common naming scheme is used to find the files, and the store option really takes a directory path prefix.

**display_ele_sink.c** This program knows how to read a single elephant sink file.

**display_inc_client.c** This program has the ability to read and interpret the incast client output.

**ele_aggregator.c** This utility was used to aggregate multi-machine elephant output.

**elephant_sink.c** This utility was described earlier and is the actual measurement tool for elephant flows.

**elephant_source.c** This utility was described earlier and contains the client code that drives the elephant flows.

**incast_client.c** This is the incast generation program described earlier.

**incast_lib.c** This is the common library utilities that most of these programs used.

**incast_server.c** This is the incast server described earlier.

**per_aggregator.c** This utility is yet another aggregator used in summing data.

**read_cap.c** This utility is a special pcap reader capable of reading a tcpdump[36] or tshark dump. It analyzes SCTP flow information and can create output for gnuplot to observe packets outstanding as well as packets outstanding when ECN Echo's occur.

**sum_aggregator.c** This is yet another aggregation utility.

## References

[1] R. Griffith et.al., "Understanding TCP incast throughput collapse in datacenter networks", In WREN Workshop, 2009.

[2] V. Vasudevan et. al., "Safe and effective fine-grained TCP retransmissions for datacenter communication." In SIGCOMM, 2009.

[3] M. Alizadeh et. al., "Data Center TCP (DCTCP)", In SIGCOMM, 2010.

[4] K. Ramakrishnan, S. Floyd, D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", *RFC 3168*, September 2001.

[5] J. Postel, "Transmission Control Protocol", *RFC 793*, September 1981.

[6] R. Stewart, "Stream Control Transmission Protocol", *RFC 4960*, September 2007.

[7] J. Iyengar et. al., "Concurrent Multipath Transfer Using SCTP Multihoming", In SPECTS, 2004.

[36]Providing the `-s0` option is used with the tcpdump.

[8] M. Scalisi, “Track down network problems with Wireshark”, Retrieved on February 24th, 2011, at http://www.computerworld.com/s/article/9144700/Track_down_network_problems_with_Wireshark.

[9] J. Nagle, “Congestion Control in IP/TCP Internetworks”, *RFC 896*, January 1984.