NC STATE UNIVERSITY

Network Assisted Congestion Control

A Report submitted in partial fulfillment of the requirements
for the Coursework of

Internet Protocols
North Carolina State University
Fall, 2010

Special thanks to,
Dr. Rudra Dutta

Submitted by,
Abhishek Maroo (amaroo, 000993449)
Jitesh Shah (jhshah, 000999459)
Neel Sheth (nvsheth, 000999216)
Shobhank Sharma (ssharma5, 000993400)

Abstract

The traditional TCP congestion control protocol (Reno or CUBIC) is well suited for slow-speed low-latency networks. However, for high-speed high-latency links, the algorithm cannot utilize the available bandwidth efficiently. This problem has been well researched and two notable solutions are: Scalable TCP and HighSpeed TCP. Both of them try to increase the congestion window size as rapidly as they can for full utilization of the link. However, in the absence of any feedback from the network, both of them have to be relatively conservative to avoid network doom. This project proposes a new TCP congestion control algorithm for high-speed high-latency networks which uses network feedback to quickly ramp up the sending rate when the network is free and quickly reduce the rate in the face of congestion. We provide some experiments and tests in the later section of the report. We use the generic name for the technique: NAC (Network Assisted Congestion Control).

1. Introduction

In 1988-89 Van Jacobson saved the internet from collapse due to heavy traffic. His contribution was a congestion control algorithm to reduce the sender's transmission rates in the face of congestion. A congestion control algorithm is elemental in avoiding a complete network meltdown. Algorithms like Reno have been over-conservative in the past, always assuming the worst, dropping the congestion window size to minimum in case of a loss. There was little these algorithms could do without any feedback from the network.

A TCP connection starts with a three-way handshake. After the connection has been established, sender initializes the congestion window size to 2 MSS and starts increasing exponentially. When it encounters a packet loss, the congestion window size is reduced to half (assuming fast retransmit) and a linear probe for the available bandwidth is started. This reduced value is known as ssthresh (Slow Start threshold). Let us abbreviate the congestion window size by cwnd. The rate at which data is transmitted is usually the minimum of the flow control window and cwnd. Since, the flow control window is usually huge, the typical case is that cwnd limits the transmit rate. As such, choosing the right cwnd is elemental to high performance of a connection.

Now consider how this algorithm will perform on a link with a bandwidth of 1Gbps and RTT of 200ms. Assuming an MTU of size 1500, the approximate cwnd is given by:

$$cwnd = \frac{T_r * RTT}{MTU}$$

Tr is the transmission rate in bytes/sec.
In the given situation, cwnd comes out to be 18000. If there is a packet loss, the new ssthresh will be 9000. cwnd is now set to 9000 and we start cautiously increasing cwnd. cwnd is increased by one per RTT. So assuming no more losses, we'll take 9000 RTTs to get back to the original transmit rate. This corresponds to almost 30 minutes!! This is clearly a very slow and unacceptable rate of increase.

Some solutions like scalable TCP and HighSpeed TCP have explored ways to increase the cwnd faster. However this can prove to be conservative if the network is relatively free or too aggressive in the face of congestion. A better decision can be arrived at if we knew how congested the network is. In this project, we propose a method to provide exactly this feedback and adjust our cwnd accordingly.

Feedback is provided in the form of each intermediate router adding a metric of how busy it is into every IP packet (technically only all packets that use the NAC algorithm) it sends out. There can be many choices of the metric to be used. Input queue size, Output queue size, CPU load, Memory usage, Average latency, etc. This metric is interpreted by the end hosts to adjust their cwnds appropriately. We explain more about how the adjustment is done in section 4.

More insight into the metric is provided in the section 3.1. A specification of how the routers should update the metric is written in section 3.2. The routers should also allow other TCP connections which do not use the metric to co-

exist. Such a mechanism is also provided. The actual algorithm is discussed in section 4 and we round up the report with the description of the test-bed we used and some preliminary numbers we gathered.

2. Design and Analysis

The scheme requires each intermediate router to insert a metric of how busy it is in each IP packet it sends out. We could reserve a 16-bit/32-bit space in the IP header for this metric. Since TCP is a two-way connection, we'll need one metric for the forward path and another for the reverse path. The IP header doesn't have 32-bits of free space in its header. So, we define a new IP Option called OPT_NAC which reserves 32-bit/64-bit space in the extended IP header for each router to fill in its data. Note that after the packet has traversed the network and reached its destination, the metric field will only have the information about the busiest router on the path. This value is echoed back in the next packet to the sender. More details about the IP Option are provided in the section 3.2.

Each TCP connection might traverse a different set of routers. Also, forward and reverse flows of a single TCP connection might traverse different paths. So, we maintain a metric for each flow. Now, a flow might change the set of routers it traverses dynamically since each packet is routed separately. We'll start getting the new metric values in one RTT time and hence we'll converge quickly.

Now that we have communicated the metric value to both the ends of the TCP connection, the problems remains of how to adjust the congestion window based on this metric and the Acks/losses. A short answer is provided here. Details of the algorithm follow in section 4.

For every ACK in which we are not in the slow start phase, we can increase cwnd once for anywhere between 50 consecutive ACKs (Network is free) to once every 200 consecutive ACKs (Network is relatively busy). Also, for every non-congestion related packet loss, we can reduce the window size to 0.9 of the original (Network is free) and 0.5 of the original (Network is congested. Go back to the original conservative scheme). The discussion of how the decision is made is deferred to Section 4 where we discuss the actual algorithm.

3. Implementation Details

3.1 Metric

As we have seen before, the metric is inserted by each router in each IP packet (which has our option) to indicate how much more traffic it can take. The metric should have certain properties.
* It should be an instantaneous value. We do NOT need the metric to carry a history because then it won't reflect the current condition of the network and we will adapt very slowly which is not desirable.
* The metric should be independent of the router capacities since each router in the path will have different capacities. Percentages would work well with the algorithm.
* Lower values of the metric indicate a congested router. Higher values indicate a relatively free router.
* Each router can ideally define its own parameters of how busy it is and represent it in percentage or a unified scale. Thus, we achieve a certain independence from the router design. eg. One router might choose CPU load to represent how busy it is whereas another router might use queue lengths.

There are various choices for the metric (not limited to these)
* Output queue length
* Input queue length
* CPU load
* Memory consumption
* Maximum latency of a packet from the point when it was queued to the point when it is put on wire for transmission

Of these, CPU load and memory consumption have the property that we can get instantaneous values for them, however spikes in either are not uncommon and they are dependent on the

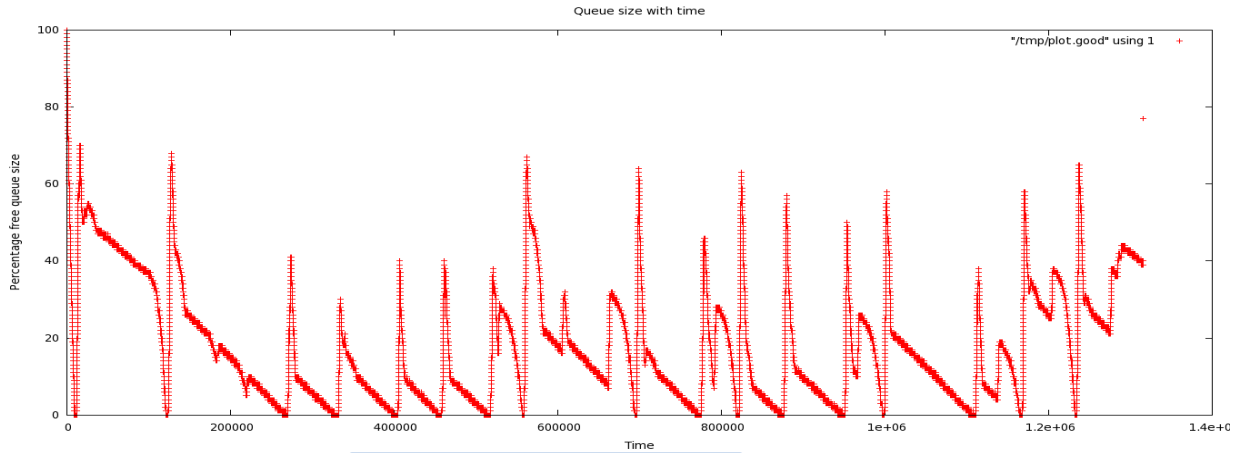

Figure 1. Analysis of Free queue size over time

hardware and OS characteristics. Latency of a packet is a good parameter, but cannot be represented in terms of a unified metric like percentage since we do not know a maximum value for latency. Input and Output queue length together can be a good choice as a metric. We know the maximum queue lengths, so we can represent them as percentages and we can learn their instantaneous values too. A higher queue length means higher latency for packet routing, thus, it is a good choice for a metric. There might be spikes where the queue length might increase because of bursty traffic, but in this case we want to ask senders to slow down to accommodate the burst in traffic, so that is not a problem.

For our implementation we chose the Output queue length as our metric. Each time a packet arrives at an intermediate router, it will calculate the instantaneous value of the output queue length, divide it by the maximum allowed queue length, put that value in the pre-allocated space in the IP Options and send it to receiver. Note that a router will update the metric ONLY IF its value is lower than the current value of the metric (i.e. it is the busiest router seen uptil now). This metric will be echoed back to the sender in an acknowledgement packet and the sender will accordingly adjust its rate of transmission. If the queue utilization is low the router will put a high value which will indicate that the sender can increase its sending rate quickly. If the queue utilization is high, it will put a lower value which will indicate that the sender should reduce the sending rate as there is chance of congestion. The amount by which to reduce or increase cwnd is described in the algorithm.

Figure 1 shows the dump of instantaneous queue values captured over time. Note that when free size reaches zero, packets are possibly dropped and meanwhile queued packets are transmitted. This explains the sharp rises. Analysis of average queue sizes on real routers will greatly help in optimizing our scheme

3.2 IP Options : Implementation aspect of the metric

We need to reserve space in each IP packet for the routes to update their metric values. We define an IP Option for that purpose. Since "31" is the only remaining free Option number that is not being used for any other purpose, we choose this value to represent our Option. It is of 10 bytes - 1 byte Option Type (31), 1 byte option length (10), 4 bytes of metric value for forward path, 4 bytes of metric value for reverse path. Every router that recognizes this option should update the metric value. Every router that doesn't recognize this option can simply forward the Option as it is. Ideally, all routers in the path should recognize the option. At the end-points of the TCP connection, we need to communicate the value in the IP option to the TCP layer.

3.3 Flow-based actions during congestion

A philosophy we adopt here which we believe will complement our scheme is to prioritize certain flows while in congestion in the hope that we move forward and resolve the congestion soon. Flow can be determined by any field in the TCP or IP header.

In case of no congestion, the flow-based module is dormant i.e. the default action is not to do anything.

When a router is in congestion, flows that were already established (i.e. TCP connection in ESTABLISHED state) will be allowed to go through and new flows will be penalized. The philosophy is that we do not accept new flows when in congestion in the hope that the congestion is resolved soon and accommodating new flows will simply exacerbate the congestion.

When in congestion, the flow-based module will kick in. It will analyze all incoming packets looking at the SYN flag in the TCP header. If it encounters one with the flag set, the packet is dropped. Thus, we do not let new connections to be established as long as the network is in congestion.

We implemented this using a NETFILTER hook on the PRE-ROUTING stage. In case of a congestion-free network, the hook does nothing. In a congested network, if it encounters a packet with SYN flag set, we return NF_DROP. Else, if the packet belongs to an already established connection we return NF_ACCEPT.

4. Algorithm

In general a congestion control algorithm works as follows.
For every ACK received, cwnd is modified as

$$cwnd = cwnd + a$$

and for every non-congestion packet loss:

$$cwnd = (1 - b) * cwnd$$

For the Reno algorithm, "a" is 1 in slow-start and "a" is 1/cwnd in the congestion avoidance phase. Let us consider the congestion avoidance phase only. The fact that we increase cwnd by 1/cwnd means that we'll increase cwnd by 1, every RTT. This corresponds to linear increase. Reno terms it as "Additive increase".
"b" for Reno is cwnd/2. Thus, Reno reduces cwnd to half for every packet loss. This is "Multiplicative decrease". Reno is, thus, AIMD (Additive Increase Multiplicative Decrease).

Other values are possible for "a" and "b". Figure 3 shows the characteristics of cwnd for any values of "a" and "b" and Figure 2 shows the characteristics for Reno.

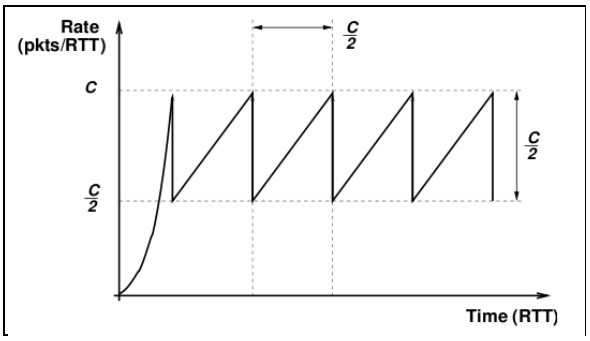

Figure 2. Reno TCP scaling properties (Figure reproduced from reference [1])

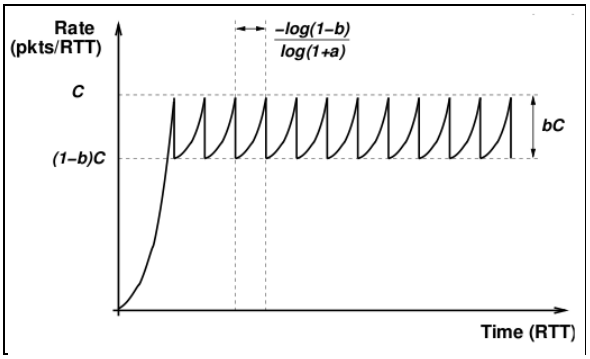

Figure 3. Scalable TCP scaling properties (Figure reproduced from reference [1])

The phase just after a packet loss is the fast recovery phase. The aim of this algorithm is to reduce the fast recovery phase to the minimum number of RTTs so that we achieve the full transmission rate sooner and hence, efficiently utilize the bandwidth.

Class	Comment	Values of a,b	Recovery time in RTTs
Class 0	Busy Network. 0% to 29% free	a=0.005, (1-b)=75%	57.6
Class 1	Approaching congestion, 30% to 44% free	a=0.008, (1-b)=82%	36.1
Class 2	Normal Operation, 45% to 64% free	a=0.01, (1-b)=87.5%	13.4
Class 3	Light loaded network, 65% to 79% free	a=0.015, (1-b)= 90%	7
Class 4	Almost free network, 80% to 100% free	a=0.02, (1-b)= 92.5%	4

Table1: Classes of how free the network is and the corresponding value of parameters "a" and "b"

On high-speed high-latency links, the fast recovery phase for Reno is long as we have already seen. Scalable TCP tries to improve this by fixing "a" to 0.01 and fixing "b" to "0.125". Thus, for scalable TCP, for every ACK

$$cwnd = cwnd + 0.01$$

And for every packet loss

$$cwnd = 0.875 * cwnd$$

[1] Shows that Scalable TCP is stable and converges. Our algorithm is based on the inferences from Scalable TCP. Note that it is an MIMD algorithm (Multiplicative Increase Multiplicative Decrease).

Looking at Figure 3, we can see that by adjusting parameter "a", we can control how rapidly the congestion window rises. By controlling the value for parameter "b", we can control how we react to packet loss events. By carefully controlling both, we can control the fast recovery time i.e. the time we will take to recover from a packet loss. The term we are looking to minimize is thus:

$$-\frac{\log(1-b)}{\log(1+a)}$$

If by looking at the value of the metric, we deduce that the network is relatively free, we can assume that the packet loss was due to a transient condition. Thus, there is no congestion in the network and we are safe to stay close to the current cwnd. Accordingly, we could set "b" such that we reduce cwnd to only, say, 0.9*cwnd. Also, we could set "a" such that we ramp up to the original speed quicker. Thus, we lose minimum efficiency.

If we deduce that the network is going to approach congestion soon, we can set "b" such that we reduce cwnd to, say, 0.6*cwnd and set "a" to, say, 0.005 i.e. we ramp up slowly and cautiously to probe whether we can get some more bandwidth. This is however still very much faster than linear probing in Reno.

Thus the first task of the algorithm is to define classes of how busy the network is based on the value of the metric. To start with we have defined five classes. These classes and the corresponding recovery times are listed in Table 1 above.

Class 0 is the class when the network is the busiest. Note that in case of a packet loss, we drop more and recover cautiously. We take about 57 RTTs to recover. Class 4 is the case when the network is almost free, hence, we try to recover quicker (almost 4 RTTs).

We define values for "a" and "b" in each class so as to utilize the network efficiently for the given congestion scenario as well as to avoid a complete network meltdown. Note that in cases of severe congestion, there will be several consecutive packet losses. In such a case, we play safe and reduce cwnd to 2 and start with the slow-start algorithm in conformance with Van Jacobson's proposal.

5. Convergence properties of NAC

How fast does the algorithm just described react to network changes?

If the network is not fully congested, some packets still go through. In this case, the sender will receive an ACK within one RTT. So, it will receive the state of the network within this period. It will then adjust the values of "a" and "b" to adjust to the network conditions.

If the network is congested and packets don't go through, then after three consecutive RTOs, we reset the cwnd to 2 MSS. Thus, the algorithm converges to congestion events pretty quickly.

6. Testbed Configuration

6.1 Experimental Setup

The Experimental setup consists of three hosts and a router (laptop configured as a router). One host (A) acting as receiver is connected on a network different from the network on which the other two sending hosts (B and C) are on, as shown in Figure 3.
All machines run on Linux kernel 2.6.36, the latest stable version.
Machines have following configuration:
4 Intel® Core™ i3 CPU @ 2.27 Ghz
MemTotal: 3402276 kB
Cache size: 3072 kB

6.2 Traffic control

Router is further configured using a tremendously powerful feature of Linux called Traffic Control. Traffic control actually governs the queuing mechanisms by which packets are received and transmitted on a router. Traffic control in practical networks is necessitated by weakness of statelessness of Packet-switched circuits. Hence, this tool simulates behavior of a flow-based network by queuing packets differently based on some attributes of the packet.

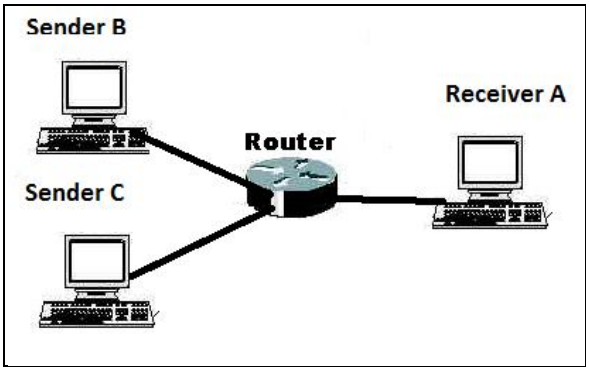

Figure 4. The Test Bed topology

Traffic control mechanisms frequently separate traffic into classes of flows that can be aggregated and transmitted. Shaping of data on routers to restrict hosts' sending rate can be done using queuing disciplines (qdisc). Qdisc can be classless in which case there is no configurable internal subdivision or it can be classful. Classful qdisc contains multiple classes or even qdisc itself. The classful qdisc needs to determine to which class it needs the packet to be sent to, this job is covered by classifier. This classification is done using filters that are like condition-matching-yes-no black box. To restrict traffic below a configured rate policing can be done that as a hard measure, drops packets if the rate exceeds.

As can be seen from Figure 5, the arrived packet is fed to Ingress qdisc that applies filter and drop if decided according to policing. This is done at an early stage so that useless consumption of CPU can be avoided. The packet if not dropped continues and can enter a forwarding state or can be locally delivered. If it needs to be locally delivered it passes to the user program for the ultimate delivery. There could be packet generated that enters kernel space through user program and continues to routing process.

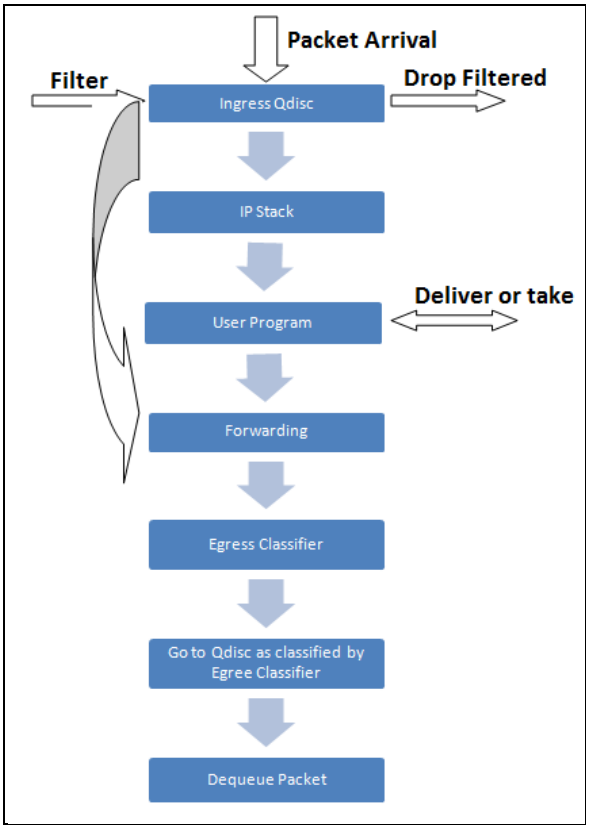

Figure 5. Traffic flow in Kernel

The packet is next delivered to Egress classifier which decides and puts the packet on the one of the qdisc if configured. In the case it is not configured, the packet is queued to the default qdisc which is pfifo_fast. The packet then waits for the kernel to transmit it over the network interface. Token Bucket Filter (TBF), Pfifo_Fast, and Stochastic Fairness Queue (SFQ) are all classless implementations. Class Based Queuing (CBQ), Hierarchical Token Bucket (TBQ) and PRIO are all examples of classful qdiscs.

Figure 6 illustrates the hierarchy followed in traffic control classful implementation. The filter attached to a qdisc returns with a decision, which is used to queue packet into one of the classes. Each disc and class is assigned a handle as shown in Figure 6 ; the handle consists of a major number separated by a ':' from a minor number. Major number act as parent class handle identifier while minor number identify the class itself. In the above figure filtering

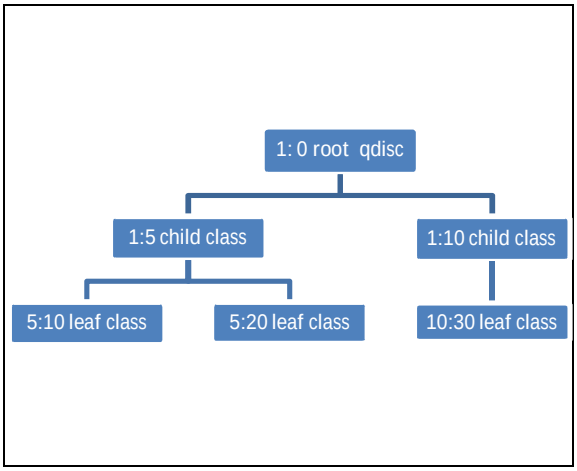

Figure 6. Hierarchy of traffic classes

action can force following path: 1:0->1:5->5:10, hence after the filtering action the packet gets queued up in the disc attached to 5:10 class.

a.Implementing Traffic Control

In our experiment we are implementing traffic control using HTB, a classful qdisc. Figure 6 shows the tc commands used to implement traffic control. $INTERFACE is the interface of router on which qdisc is configured. HTB gives us control on outbound bandwidth on a given link (or even for a protocol). The first command attaches a qdisc HTB to eth0 interface, and declares that any traffic that will otherwise be not classified would be assigned to the class 1:12 (1=qdisc and 12=class id). The second command creates a root class with class id 1:1, and assigns bandwidth of 4 mbps to this class of traffic. Since there is allowance for borrowing in HTB, the parameter ceil specifies the maximum bandwidth that a class can use. Child class of a Root class in HTB can borrow traffic from each other. Rate and ceil are actually not the instantaneous measures rather they are average quantities hence burst is given which acts as an instantaneous parameter controlling the amount of data that can be sent at maximum rate. The fourth command assigns flow to class, based on destination IP.

```
tc qdisc add dev $INTERFACE root handle 1: htb default 12
tc class add dev $INTERFACE parent 1: classid 1:1 htb rate 4mbps ceil 4mbps burst 8m
tc qdisc add dev $INTERFACE parent 1:1 handle 10: netem delay 100ms 10ms loss 2.5% 50%
tc filter add dev $INTERFACE parent 1:0 protocol ip prio 1 u32 match ip dst $DESTIP police
rate 32mbit burst 8m drop flowid 1:1
```

Figure 7. Traffic control implementation in action

6.3 WAN Emulation

In our project we desired to experiment on a network resembling the Internet. In order to do this we created a WAN scenario using WAN Emulation. We chose to emulate a WAN using 'netem', a tool of linux. It emulates properties of WAN like variable delay, loss, duplication and re-ordering. Netem module is controlled by tc. The third command in Figure 6 configures the network to have a delay of 100ms on router, with variability of 10ms. Also using this we configured the network to have different loss rate so that measurements for different test conditions could be taken.

6.4 Actual Measuring tool

The parameter we measure here is the send congestion window of sender denoted by snd_cwnd. In order to achieve this we have developed two user space programs, tcpsender and tcpreceiver. The tcpreceiver opens up a listening socket and when tcpsender is ready to send, the tcp connection is established on an ephemeral socket. At this point we measure the sender's send congestion window. All the tcp related parameters are stored in tcp_info structure which is defined in /usr/include/netinet/tcp.h header file. This structure is populated using getsockopt() function, which copies everything to the memory region indicated by the supplied pointer. The parameter of our interest is tcpi_snd_cwnd. But we do not want this value for every byte rather we record this variable for some aggregate data periodically. Before running the above described programs we set the linux boxes for two scenarios; one with traditional reno TCP and a second scenario with our proposed transport protocol "NAC". We plot graphs of time against tcpi_snd_cwnd using gnuplot utility of linux.

7. Results and Observations

In this phase of experiment we will be comparing different TCP variants with our proposed TCP. Comparison is done by plotting send congestion window against time. The comparison is done using different WAN emulation loss parameters particularly at high losses (1%) and low losses (0.1%) using the netem tool as described in the previous section. Figure 8 to 11 show the resulting graphs.

7.1 Reno TCP v/s NAC TCP

Reno as expected has a very slow ramp up in both high and low loss conditions. NAC has far superior performance due to exponential ramping. The linear rise of Reno as stated earlier is due to longer recover phase which in turn is due to increase of congestion window by 1/cwnd for every ACK. In NAC we set the parameter 'a' according to the metric that creates fast and safe ramp according to current network conditions.

The red line in the below diagrams is the congestion window size for NAC. Both the figures clearly depict how inefficient the linear increase of RENO is. NAC grabs the available bandwidth quickly and is in general much more efficient than RENO.

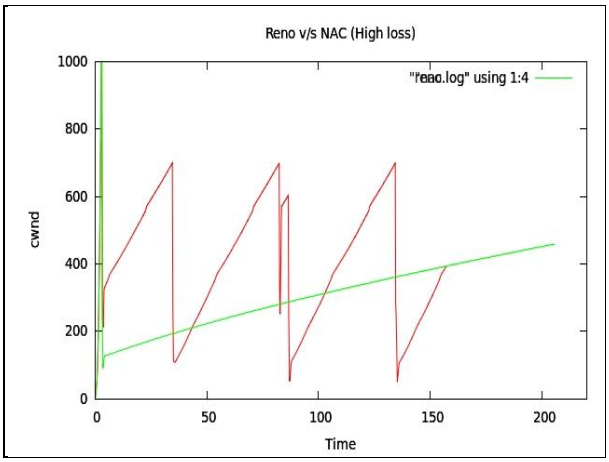

Figure 8. Reno v/s NAC at high loss

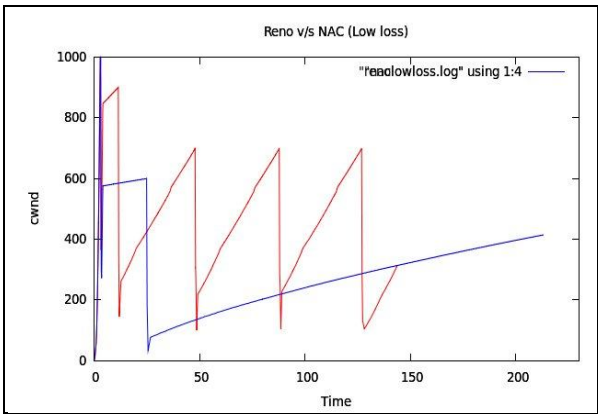

Figure 9. Reno v/s NAC at low loss

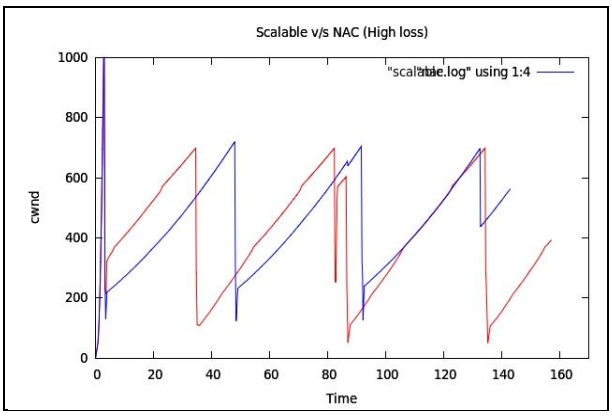

Figure 10. Scalable v/s NAC at high loss

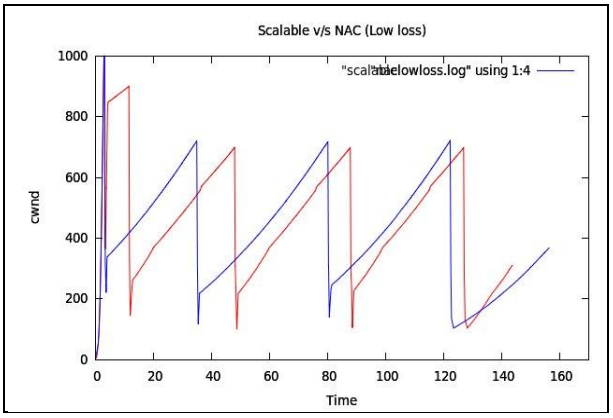

Figure 11. Scalable v/s NAC at low loss

7.2 Scalable TCP v/s NAC TCP

Figures 10 and 11 show that Scalable TCP and NAC TCP are comparable in performance. However, there are instances where NAC (red lines) grabs the bandwidth quickly because it is safe to do so. This test was done at a clamp of 4mbps. We believe that the effect of this slight improvement will be very much pronounced on a, say, 10Gbps network with high latencies.

7.3 Inaccuracies in measurements

We have used a laptop machine as a router. We have simulated two virtual interfaces over a single physical interface. Even though this is theoretically sound, practically sharing two interfaces over one physical medium has its effects on the actual numbers. The graphs and congestion window values might be biased because of this.

Also, since we did not have access to a real WAN with real traffic, we have tried to simulate a WAN using the “tc” command as described in the previous sections. The packet loss properties of emulated WAN along with a dummy-router can be significantly different than real values. This might explain the sharp drops in all of the above graphs. These result out of continuous bursty losses when the router’s (laptop machine) buffers fill up.

8. Problems we faced

1.WAN emulation

The most challenging part was WAN emulation. Since we are working on a congestion control algorithm for high-speed high-latency links, it is important to recreate such an environment. It is clear that it is not possible to emulate the real congestion conditions or even link bandwidths, but we tried to reach as close as we can. Each one of us had a 100Mbps ethernet card and laid our hands on a gigabit ethernet switch, so that the switch won't be a bottleneck. Second, we learnt about how to set-up a Linux box as a router. We learnt how to control traffic based on flows using the tool "tc" (Traffic Control). We still couldn't emulate losses, reorderings, latencies and high congestion environments. We then learnt about the "netem" module of "tc" which lets us emulate losses, reorderings and latencies of high-speed high-latency environments. We also stumbled on the policing module of tc and learnt how to emulate congestion after a certain bandwidth limit has been reached. With all these modules, we were able to make a decent emulation of WAN.

2. Putting the metric value in the IP Option

We chose output queue length as our metric. This is updated by each router. When an ingress IP packet is processed, all the IP options are processed, the IP header is remade and then the packet is sent to the output queue for transmission. It is at this stage that we learn about in which queue it will we be put and how free it is. Note that we have already processed the IP Options, so knowing where to put the metric is tough. For now we are using only one queue, so we can read-ahead and determine how busy it is. Going forward we intend to introduce code for deferred processing of our option.

3. IP to TCP layer communication

We had to communicate our metric from IP layer to the TCP layer so that it can be used by our congestion control module. We used skbuff for the communication.

9. Conclusion

NAC TCP provides a framework over which one can design a Network-assisted scheme of congestion control. We demonstrated how this can be done by designing and implementing the NAC TCP congestion control algorithm which uses Network Feedback for adjusting the congestion window. Preliminary numbers show that it is way better than Reno TCP and slightly better than Scalable TCP. More work is needed to prove that this implementation is fair i.e. plays well with other TCP flavors. Also, work is needed to analyze its performance in real-life high-speed links and real routers. Defining traffic classes precisely will need empirical evidence of actual congestion scenarios. It will also be interesting to explore how the adaptation of HighSpeed TCP to this network-assisted framework will perform as compared to NAC TCP.

10. References:

[1] *Scalable TCP: Improving Performance in Highspeed Wide Area Networks*
Tom Kelly CERN/ University of Cambridge

[2] *Understanding Linux Network Internals*
By Christian Benvenuti Publisher:O'Reilly Media Released: December 2005

[3] *Congestion Control in Linux TCP*
Pasi Sarolahti University of Helsinki, Department of Computer Science
Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference

[4] *Congestion Avoidance and Control.*
Van Jacobson, Michael J. Karels. Proceedings of the Sigcomm '88 Symposium, vol.18(4): pp.314–329. Stanford, CA. August, 1988.

[5]http://www.linux.org/docs/ldp/howto/Traffic-Control-HOWTO/intro.html

[6]http://linux-ip.net/articles/Traffic-Control-HOWTO/classful-qdiscs.html

[7]http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[8]http://www.iana.org/assignments/ip-parameters

[9] RFC 791: *Internet Protocol*, September 1981.

[10] RFC 3649: *High-Speed TCP for Large Congestion Windows*, S. Floyd, December 2003.