Performance Analysis of TCP Variants

Neeki Hushyar
UMass Amherst
nhushyar@cs.umass.edu

Angela Upreti
UMass Amherst
aupreti@cs.umass.edu

Keywords

TCP variants; Comparison; SACK; Reno; New Reno, Vegas; Tahoe

1. INTRODUCTION

Web traffic runs over TCP and given the scale at which the internet has grown and how much we rely on the web, it is important that the underlying transport protocol is able to operate at a reasonably low latency and provide high throughput. Network congestion causes packet loss which increases latency and hence, reduces throughput. Using the right TCP variant saves both money and time. Amazon found that every 100ms latency cost then 1 percent in sales [1]. Higher latency/wait-time in web applications discourages clients from revisiting the cite. This is specially true for video applications such as youtube. Clients usually close the page if the latency is high. Fairness in a the underlying transport layer is equally important. One flow should not hog most of the network bandwidth and throttle other flows. Otherwise, different customers paying the same amount for internet service would get widely different performance. Metrics such as throughput, average and end-to-end latency, packet drop rate, fairness are different across different tcp variants and the congestion control algorithms they use.

Congestion control algorithm for TCP was proposed by Van Jacobson in response to 1986's 'congestion collapse'. The original TCP proposed by Van Jacobson contained elements such as dynamic window sizing, slow start, additive increase and multiplicative deacrease (AIMD). Since then, several variants of TCP have been proposed. TCP Reno adds fast recovery and fast retransmit to TCP Tahoe. TCP Reno's fast recovery and fast retransmit suffer when multiple packets are dropped from the same window. In presence of multiple packet loss, the TCP pipe often gets drained. NewReno has to wait for retransmit timeout to be able to send data again. Consequently, multiple packet loss leads to low throughput. TCP NewReno was introduced to solve this. When in fast recovery, TCP NewReno interprets a single partial Ack as a signal to retransmit another packet rather. NewReno does not wait for 3 duplicate ACKs before retransmitting. TCP SACK is another TCP variant that solves the multiple packet drop problem in Reno. SACK allows the receiver to acknowledge any non sequential packets it has received. NewReno without SACK is unable to tell exactly which packets are missing. While NewReno can retransmit only one lost packet per RTT, TCP SACK can transmit as many lost packets as the congestion window allows. Variant Vegas uses RTT as an estimate of congestion rather than using packet loss.

We conduct three different simulation studies in this paper:

1. Compare performances of Tahoe, Reno, NewReno, and Vegas under congestion.
2. Compare fairness between different TCP variants.
3. Study influence of queuing disciplines (1)RED and (2) Drop Tail on TCP Sack and TCP Reno.

2. METHODOLOGY

2.1 The Setup

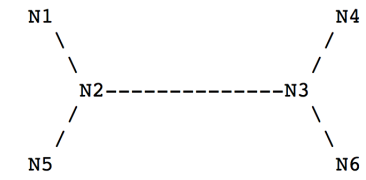

Figure 1: We attach TCP agent with FTP as the application, UDP with CBR to this topology.

We use ns2 to create the topology in Figure 1. The setup consists of five different nodes, N1- N5. The link between each pair of nodes is a duplex link with a capacity of 10 Mb.

2.2 Calculation

NS2 trace files were used to calculate drop rate, average throughput, average latency and latency over time for combinations of different TCP variants and different queuing algorithms. The trace files were parsed using a python script because of the ease of text parsing with python. We used a

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

python library called the Matplotlib to generate the plots. We use a bash script to automate a series of experiments with varying TCP variant, CBR rate, CBR packet size and queuing algorithms.

1. **Average Throughput:** To calculate the average throughput of the tcp stream, we divide the sum of packet sizes of all TCP packets that reached the destination by the total time of the tcp flow.
2. **Latency:** To calculate latency over time, we recorded the time it took for each packet to get from the source to desination over time. We took the average of latencies over time to get the average latency.
3. **Drop Rate:** The ns trace clearly identified the dropped packets. To calculate the drop rate of a flow, we divided the total number of drops by the total number of packets in that flow.

2.3 TCP Performance Under Congestion

This experiment was designed to test the performance of different TCP variants Tahoe, Reno, NewReno and Vegas under various levels of congestion. TCP agent was attached to start at N1 and sink at N4. FTP application was attached to the TCP agent to simulate traffic that is typical of FTP. CBR flow was added to go from N2 to N3 to create congestion in the link between N2 and N3. We varied the CBR rate from 1 Mbps to 10 Mbps to create different levels of congestion. We had the liberty of choosing the packet size to maintain the CBR. To study the effects of varying CBR packet sizes, we also simulated varying packet sizes, 500B to 10,000B, for all four TCP variants. We did not change the default ns2 buffer size of 20 packets.

2.4 Fairness between TCP variants

This experiment simulated two TCP flows at a time. The variant pairs that were simulated are (1) Tahoe/Tahoe (2) Reno/Reno (3) NewReno/Reno (4) Vegas/Vegas (5) NewReno/Veg Like before, to create varying levels of congestion, a CBR flow was added to originate at N2 and end at N3. The first TCP stream originated at added N1 and ended at N4. The second TCP stream originated at N5 and ended at N6. For each pair of the variants, we ran simulations for CBR ranging from 1 Mbps to 10 Mbps. The packet size was 500, constant across all the experiments. We chose this packet size because an average IP packet size is around 600. Accounting for the IP headers, we wanted a smaller packet size for UDP. We also started the two competing variants at different times and ran the simulations for a bit longer (15s) to check if fair throughput could be achieved over time if one of the TCP streams was delayed.

2.5 Influence of Queuing

In this experiment, we study the influence of queuing algorithms RED and DropTail on TCP Reno and SACK. The set up for this experiment is the same as experiment 1. We conduct the experiments for two different buffer sizes, 20 and a 100. To compare the performance of RED and DropTail in the two TCP variants, we compare the differences in throughput, latency and drop rate between the queuing disciplines.

3. EXPERIMENT 1: ANALYZING PERFORMANCE OF TCP VARIANTS

This experiment analyzed the performance of different TCP variants over a setup consisting of a single TCP stream and a single CBR flow. The performance of the TCP variants were analyzed over a range of different packet sizes and transmission rates assigned to the CBR. We analyze the results of this experiment in two parts. First, by looking at the performance of the variants over an increasing range of CBR packet sizes, while the rate of the CBR is constant at 2Mbps. Second, we look at the performance of the variants over an increasing range of CBR flows - from 1 Mbps to 10 Mbps, while the size of the packet is held constant at 3,000 bytes.

When holding the CBR steady, at 2Mbps, none of the variants experienced any packet loss over the range of increasing packet sizes, from 500 bytes to 8,000 bytes. The average and end-to-end latency as well as the throughput for TCPs , Reno, NewReno and Tahoe were all the same. The throughput hovered around 1 Mbps as the size of packets increased and the average latency hovered around .02 seconds. The average and end-to-end latency of TCP Vegas was slightly smaller, likely because Vegas never saturates the full capacity of the link. Doing so, would imply congestion and the throughput would decrease. The throughput for Vegas hovered at just under .99 Mbps as the size of the packets increased and the latency hovered around .005 seconds.

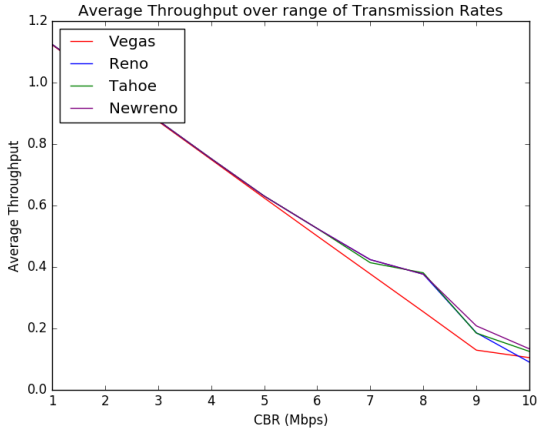

Figure 2: This figure plots the average throughput of each TCP variant over a range of CBR flow rates. The packet size is initialized to 3,000 bytes.

When the packet size is held steady, at 3,000 bytes and the CBR flow is increased from 1 Mbps to 10 Mbps, all four variants suffer packet loss. Reflected in Figure 3, TCPs Reno, NewReno and Tahoe suffer their first packet drops at a CBR of 7 Mbps. TCP Vegas does not suffer any packet drops, but the throughput in consistently lower than the other variants, as it is reducing its congestion window size to reflect the increasing RTT of the packets. TCP Vegas takes preventative measures to avoid packet loss. The other variants make changes only in the aftermath of packet loss. To prevent congestion, Vegas records the round trip time (RTT) of packets and adjusts its congestion window according to any delay. Vegas does not experience packet loss until the CBR rate is 9Mbps.

Each variant is consistent, only in that they wait until

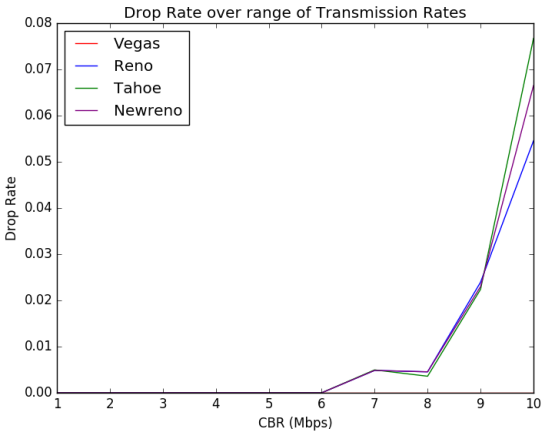

Figure 3: This figure plots the number of packets dropped by each TCP variant over a range of CBR flow rates. The packet size is initialized to 3,000 bytes.

receiving 3 duplicate-acknowledgements or retransmit timeouts before retransmitting the first lost packet.

Given low-to-moderate levels of congestion, TCP Vegas is able to efficiently decrease its congestion window size, before experiencing packet loss, and thus is able to maintain 0 packet loss while its counterparts are unable to. However, once some threshold is reached, Vegas will suffer packet loss and will no longer the most efficient variant in terms of dealing with multiple packet losses. NewReno, handles multiple packet losses better than its counterparts. After the first packet loss is identified by a triple duplicate acknowledgement, or by a retransmission timeout, NewReno assumes that any packet following a partial acknowledgment is lost. This makes the retransmission of packets much faster and accounts for the increase in throughput experienced by NewReno in Figure 2.

4. EXPERIMENT 2: ANALYZING FAIRNESS BETWEEN TCP VARIANTS

This experiment used two TCP flows and a single CBR flow, to analyze the fairness between the TCP flows among different pairs of TCP variants. We tested different combinations of variants, including assigning TCP Vegas to both flows, TCP Reno to both flows, NewReno to one flow and Reno to another, and NewReno to one flow and Vegas to another. Further, we ran two slightly different simulations. The first simulation analyzes fairness in a network with a CBR flow rate of 1Mbps and a packet size of 1000 Bytes. The second simulation analyzes fairness is a network with a CBR flow rate of 10Mbps and a packet size of 10,000 bytes. The first test simulates a network with little to no congestion, and the second test simulates a network with congestion and packet loss.

4.1 Simulation 1: Little-to-No Network Congestion

In the network with little congestion, the throughputs of the individual streams given the combinations of Reno/Reno and NewReno/Reno were all equal, at .56 Mbps. This is to be expected. There were no packet losses, the NewReno stream acted as a Reno stream. This is because the variants only differ in what they do after multiple packet drops. After the first packet drop, both variants use fast retransmit. Afterwards, Reno enters fast recovery and cuts its congestion window in half while NewReno inflates its congestion window to allow for outstanding packets between the source and destination to complete their transmissions. The throughputs of the two Vegas streams, were .55 and .6 Mbps. This can be a result of one stream calculating a higher than expected RTT and reducing the congestion window, allowing the other to maintain and increase the cwnd size. Unlike Reno and NewReno, Vegas is not deterministic given no packet drops. The largest discrepancy between any two streams we tested, was that of NewReno and Vegas. The average throughput of NewReno was .98 Mbps, and the throughput of Vegas was .15 Mpbs.

TCP Vegas senses congestion through larger than expected RTT values, and proceeds to decrease its congestion window. TCP NewReno continuously increases its window size until loss actually occurs. Even though there were no packet drops and little to no congestion, the aggressive nature of TCP NewReno monopolizes the bandwidth of the streams, reducing the throughput of the more cautious TCP Vegas.

4.2 Simulation 2: High Network Congestion

Each combination of variants in the second simulation saw high amounts of congestion. Although the combination of TCP Vegas, once again resulted in similar throughputs for individual streams - at about .1 Mbps each, we note that these throughputs are much smaller than what was sccen in th eprevious simulation. The throughput was much lower because each stream sensed the congestion caused by the other flow; however, there were no packet drops, instead the congestion window was continiously decreased. The combination of Reno/Reno also saw lower, but equal throughputs, with each stream maintaining an average throughput of about .2 Mbps. In this case, we saw about 23 packet drops. Still, the fast recovery and fast retransmit process is the same for both of these streams, so the similar throughputs are to be expected.

The combination of NewReno/Reno resulted in a larger throughput discrepency given the congested network. NewReno, as expected achieved the slightly higher throughput of .23 Mbps where as Reno's throughput was .185 Mbps. There were 30 total packets dropped. If the simulation had run for longer, we predict the number of dropped packets would increase and the discrepency between the throughputs of the NewReno and Reno streams would grow even larger. The reason for this discrepency is that NewReno will inflate its congestion window size by the number of duplicate acknowledgments it received, to facilitate the transmission of the remaining packets going between the source and destination. Ultimately, NewReno deflates its congestion window, but only after outstanding packets have been properly acknowledged by the receiver. Reno, on the other hand, will decrease its congestion window by two, after having completed fast retransmit. This is done with the assumption that size of packets entering the network is too big and decreasing the window size will reduce the amount of congestion.

This discrepency reappears more jurassicaly when assigning one stream to TCP NewReno and another to TCP Vegas. NewReno had an average throughput of .26 Mpbs and Vegas at .1 Mpbs. Only 8 packets were dropped in this simulation,

all by the TCP NewReno stream. NewReno utilizes an increasing amount of the link before and in the immediate aftermath of a packet loss occur. Vegas, on the other hand, continuously decreases its congestion window as it perceives the increased utilization by NewReno, as congestion. However, we notice the discrepency in throughputs here, is not as large as the discrepency we saw in the first simulation. This is because the packets dropped by NewReno, although inflating the congestion window intiially, ultimately deflated the window, reducing the size of the packets being released into the network.

5. EXPERIMENT 3: INFLUENCE OF QUEUING

TCP SACK aggresive about bandwidth Congestion window script RED vary buffer size This experiment analyzes the performance of TCP Reno and TCP Sack using the queuing algorithms: DropTail and RED. TCP Sack, increases and decreases the congestion window the same as TCP Reno does, when dealing with the first packet loss. Upon the first sign of packet loss, both variants cut the congestion window in half, and proceed to retransmit the lost packet.

In situations involving multiple packet losses, TCP Sack waits until the number of outstanding, or yet to be acknowledged packets, is less than the congestion window. While in fast recovery, TCP Sack keeps a record of outstanding packets as well as a corresponding counter. Reno only has the foresight to keep track of a single lost packet. Sack uses partial acknoweldgements to decrease the counter of outstanding packets at a rate which increases the congestion window faster than slowstart, which is why the performance of Sack benefits from a slightly higher throughput and a slightly lower latency than Reno.

DropTail fills up the queue, and drops any further packets when the queue is full. It is first come, first serve. Random Early Detection (RED) accepts all packets when the queue is below some threshhold, and randomly accepts packets when the queue is above the threshhold but still less then the maximum capacity. This allows it to reduce the power of large and fast packet burst which would otherwise monoplize the bandwidth and fill up the queue (in the case of DropTail).

Our experiment held constant a CBR flow rate of 8Mbps and a packet size of 4,000 bytes, which cause congestion and packet loss in the network. The results of the experiment showed an average throughput using RED in TCP Reno and SACK was .214 Mbps and .225 Mbps, respectively. The average throughput using the DropTail queuing algorithm in Reno and SACK, were .148 Mbps and .150 Mbps, respecitvely. The increased throughput using the RED queuing algorithm is because the network is able to maintain a more steady flow of traffic, and avoid being under or over utilized. Random dropping however, did double the drop rate - because packets were dropped when they were above some threshold to encourage fairness among the tcp traffic. TCPs Sack and Reno experienced 118 and 116 dropped packets using DropTail and experienced 216 and 267 dropped packets using RED.

The combination of TCP Sack using the RED queuing algorithm, consisted of the highest throughput, lowest average and end-to-end latencies of the four combinations of variants and queuing algorithm. As previously mentioned, it must be noted, there was a relatively high number of drop packets in comparison to results computed when using DropTail. It comes down to the amount of congestion on the network. If there is moderate congestion, it is efficient to use TCP Sack and RED together. If there is a rate of congestion, far exceeding the threshhold using in RED, at which point, packets are randomly dropped, it may be less efficient to use this queuing algorithm.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] R. Kohavi and R. Longbotham. Online Experiments: Lessons Learned. *Computer*, 40(9):103–105, Sept. 2007.