Reproducing Compound TCP

CS244 Project, Spring 2018

Jaspreet Kaur
jaspreetkaur@stanford.edu

Tucker Leavitt
tuckerl@stanford.edu

Abstract

As the world becomes more interconnected, the importance of high speed and long distance networks is increasing. There is more focus on scalability than ever before. The standard TCP is unable to utilize the network capacity fully due to its conservative "Additive Increase Multiplicative Decrease"(AIMD) algorithm. This project is based on reproducing the results of the paper on Compound TCP algorithm. Compound TCP overcomes the conservative nature of the TCP algorithm by combining the loss and delay based congestion control approaches. Hence, it can perform much better for high speed and long distance networks. This can be achieved by adding a scalable delay based component to the standard TCP.

In this reproduction paper, we reproduce Table 1 and Figure 8 from the CTCP paper [8]. These figures describe the throughput of CTCP under different network conditions, and compare its performance to TCP. Our results are qualitatively consistent with the original paper, though our throughput measurements for both CTCP and TCP are lower.

1 Introduction

Transmission Control Protocol (TCP) defines how computers send packets of data to each other and maintain the network. Congestion occurs when system load is greater than the system resources. Congestion control is a very interesting topic from a network engineer point of view and we were very interested in exploring this area in more detail. Hence, we decided to reproduce this paper on CTCP congestion control.

TCP has embedded congestion control algorithm with additive increase of window size when there is no congestion and multiplicative decrease when there congestion. This has been described by [1] with multiplicative decrease factor of 0.5. TCP considers packet loss as a measure of congestion, hence it is considered a loss based congestion control algorithm. Conventionally, three duplicate ACK's or a retransmission timeout is considered as packet loss.

The TCP Congestion Control algorithm is considered very conservative for high speed and long distance networks. This is because for TCP to effectively utilize available bandwidth, it requires a window size roughly equal to the Bandwidth Delay Product (BDP). Even in an ideal link with no packet loss, it will take TCP a long time to increase it's window size for full utilization of the bandwidth. In a real world situation, TCP may never achieve a high enough window size. This is because TCP only increases its window size by one every RTT time. It has been shown that the average TCP window is inversely proportional to the square root of the packet loss rate[7].

There have been some approaches to overcome this behavior by introducing TCP slow start and higher multiplicative factors for more aggressive link utilization. The algorithms which utilize packet loss as a measure of congestion are known as "loss based" Congestion Control(CC) algorithms . There are also certain algorithms that used a delay based approach to CC and reduce transmission rate based on RTT variations. There are drawbacks and advantages to both these approaches. Hence, we decided to implement the algorithm that uses the best of both worlds. CTCP uses a combination of loss based and delay based approaches and provides us the benefit of efficiency, RTT fairness and TCP fairness. The delay based component in CTCP can efficiently use link capacity as well as detect congestion by sensing changes to the

RTT.

Our goal is to reproduce Table 1 and Figure 8 from [8]. Table 1 shows the throughput of CTCP and TCP Reno under burst background traffic, and Figure 8 shows the utilization of CTCP and TCP and different link loss rates.

2 Related Work

While designing a congestion control algorithm for wired, high-speed, long-distance connections, there are three primary performance metrics to consider: throughput/utilization, RTT fairness and TCP fairness. A new congestion control algorithm must efficiently utilize the network bandwidth, must have good intra protocol fairness for competing flows with different RTT's and ensure TCP fairness by not stealing bandwidth from other TCP flows.

CTCP was first proposed in 2005 [8], along with several other new congestion control algorithms designed to utilize high-speed links more efficiently. Some algorithms, like HSTCP[2], STCP[4] and BIC-TCP[10], were primarily loss based. These algorithms achieve high efficiency but lead to RTT and TCP unfairness. HSTCP needs larger amount of background traffic and more variable traffic than other protocols to achieve convergence [3]. On the other hand, delay based algorithms such as FAST [9], achieve high efficiency and RTT fairness but suffer from TCP unfairness especially if most flows are loss based. This is because delay based flows will reduce sending rate when queue is built to prevent self induced packet losses. This will trigger loss based flows to increase sending rate since they now observe lower packet loss rates, hence causing TCP unfairness. The delay based approaches try to maintain a fixed buffer occupancy. FAST suffers from unfairness especially in networks with small buffer sizes or networks with long delay as shown experimentally in [3].

TCP Africa[5] combines both aggressive increase in window size (implemented in loss based CC's) and additive increase(implemented in TCP). It uses delay information to switch between both these modes.

CTCP was developed by Microsoft and is part of the Windows Vista and Window Server 2008 TCP stack. In the decade since CTCP's release, several other sophisticated congestion control algorithms have been developed, such as CUBIC, BBR, Remy, and PCC.

3 System Design

Our system implementation can be found here:
`https://github.com/tleavitt/sourdough`[1]

Our goal was to reproduce the TCP and CTCP throughput measurements in Table 1 and Figure 8 of [8]. Table 1 measures the throughput of a congestion control sender and receiver under "burst background traffic," which they define as constant-rate traffic that toggles on and off every 10 seconds. The authors did not specify what network topology they used to create this traffic. Other works (e.g. [3]) have used a bow-tie topology for performing throughput measurements in the presence of background traffic, so we did the same. Figure 1 shows the topology. Send and Recv are the CTCP sender and receiver, X1 and X2 are hosts exchanging UDP traffic, and the blank nodes are switches.

We simulated the bow-tie topology using the Mahimahi link emulator. We mostly used the mm-delay and mm-loss attributes implemented in the Mahimahi emulator [6]. The loss introduced using mm-loss was used to introduce packet loss rate from 10^{-6} to 10^{-2}. This was used in the reproduction of Figure 8 in the CTCP paper. The mm-delay attribute was used to specify the propagation delay in one direction. The CTCP paper uses a delay of 100ms but we used a delay of 40ms for our implementation.

We implement a socket sender and receiver framework that transmits UDP packets and has a pluggable "congestion controller" that modulates the sender window size. Our initial thought was to use Mininet to build this topology, but we could not get both Mininet and Sourdough to work on the same machine. We could only get Mininet to install on the pre-built VirtualBox Ubuntu 14.04 image, and we received socket errors when trying to run the sender from Programming Assignment 1 of CS 244 on this machine.

We decided to modify the topology so that we could run the experiments on a single Mahimahi link. Essentially, we collapsed the CTCP and UDP clients into a single host

[1]The accompanying data analysis notebook can be found at `https://github.com/tleavitt/ctcp-data-analysis`

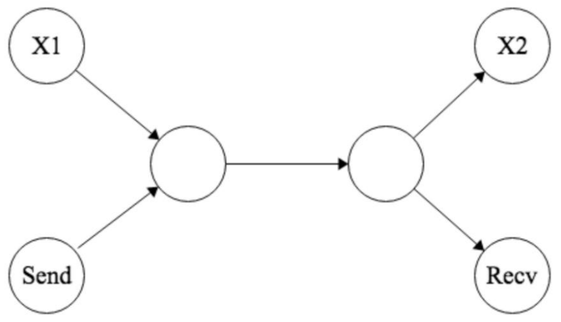

Figure 1: Simulated network topology used for throughput measurements. Send and Recv are the congestion control nodes. X1 and X2 are the cross traffic nodes.

that multiplexes the two streams over the bottleneck link. The final topology is shown in the Figure 2 below.

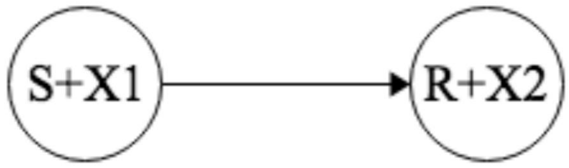

Figure 2: Simplified network topology used for throughput measurements.

We initially planned on using Iperf for generating the UDP traffic but changed it considering that Iperf uses TCP and not UDP. We implemented background traffic using a simple socket client that injects packets into the network at constant time intervals. Additionally, it turns on and off for 10 seconds, as in the CTCP paper.

We implemented CTCP according to the specifications in [8] and TCP Reno according to RFC 2581. Our implementation lacks many of the features that a production CC algorithm would have, but it has the correct long-term and qualitative behavior. The implementation is available at our github repo.

The CTCP paper uses drop tail queues, but there is not an easy way to implement these using Mahimahi. Hence, to simulate the behavior of a drop tail queue at the bottleneck, we programmed the sender to label a packet that has been delayed by more than a threshold amount as "dropped." To simulate a drop-tail queue of size B, the delay threshold D_{queue} should be:

$$D_{queue} = \frac{B}{R} - D_{prop}$$

where R is the bottleneck link rate and D_{prop} is the round-trip propagation delay. In the paper, they used queues of size $B = 18$Mb. We had $R = 240$Mbps and $D_{prop} = 40$ms, which gives $D_{queue} = 35$ms The authors used a bottleneck link speed of 700 Mbps considering their router CPU bottleneck limitations and used burst UDP traffic rates of 50 Mbps, 100 Mbps, 150 Mbps and 200 Mbps for Table1. However, considering limitations of our system, we have kept the bottleneck link speed as 240 Mbps. Accordingly, we have scaled our burst UDP rates to 17 Mbps, 34 Mbps, 51 Mbps and 68 Mbps accordingly.

CTCP has several hyperparameters, as specified in [8]. We found empirically that the following hyperparameters worked well: $\alpha = 1, \beta = 0.3, \gamma = 30, \zeta = 0.02$ and $k = 0.1$.

CTCP falls back to the standard TCP implementation when the window size is small. However, one the delay based component is active, the window size increases by at least one MSS every RTT.

4 Evaluation

Table 1 compares CTCP and TCP Reno for different burst background traffic rates. We observe that TCP utilization drops significantly (from 60.21% to 43.13%) as the background traffic rate increases. However, the utilization of CTCP varies very less(between 58.09% to 63.60%). This behavior matches the qualitative behavior of TCP and CTCP in the original paper.

However, in the CTCP paper the link utilizations for TCP and CTCP were higher; CTCP had a utilization of almost 90% for all background rates. We suspect this has two causes: our implementation does not include most of the performance enhancements and optimizations that a real CC implementation would, and our algorithm hyper-parameters are probably not tuned optimally.

Cross-traffic Rate	17 Mbps	34 Mbps	51 Mbps	68 Mbps
TCP-Reno	139.33 60.21%	130.02 58.18%	92.49 43.13%	95.19 46.23%
CTCP	146.89 63.45%	138.16 59.59%	125.00 58.09%	130.62 63.60%

Table 1: Throughputs (in Mbps) and utilizations under constant-rate cross-traffic over a 240 Mbps bottleneck link.

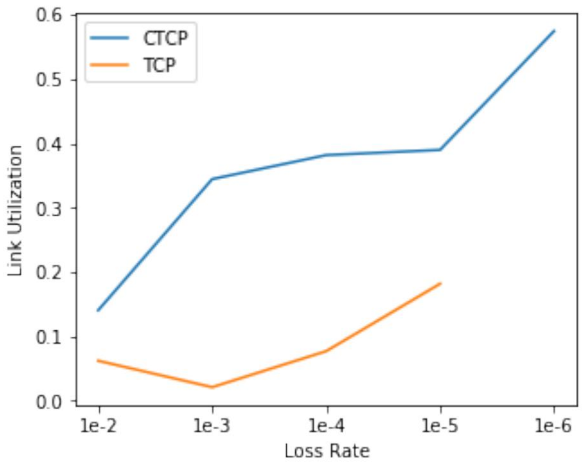

Figure 3: Percent utilization of the bottleneck link as a function of link loss rate.

We can observe from Figure 3 that CTCP performs better than normal TCP for varying loss rates. This is because CTCP has a more aggressive algorithm for increasing the window size after occurrence of packet loss. This behavior is also seen more clearly in Figures 4 and 5 below. While TCP takes 125s to rise to a throughput of 200 Mbps, CTCP is able to achieve the same throughput in around 50s.

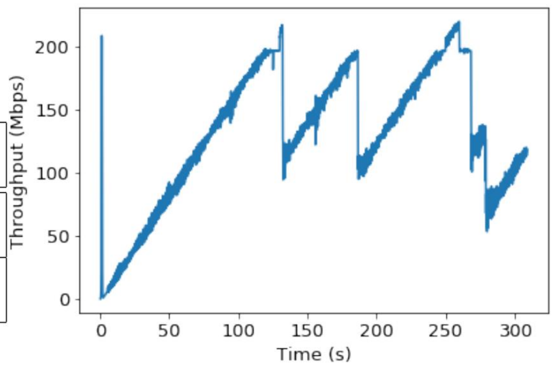

Figure 4: Instantaneous throughput for a long-lived TCP connection over a 240 Mbps link with 34 Mbps background cross traffic .

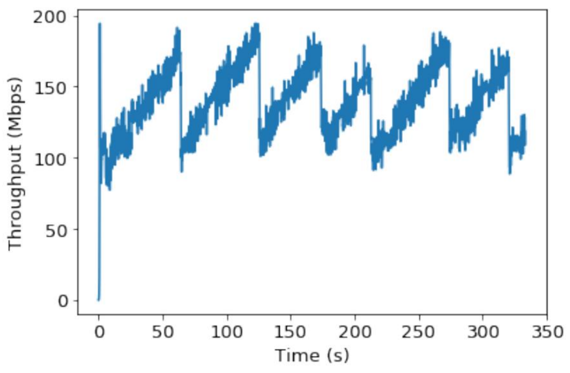

Figure 5: Instantaneous throughput for a long-lived CTCP connection with identical conditions to figure 4. The average throughput is higher, and the loss events are more frequent.

5 Conclusion

We were successfully able to implement Table 1 and Figure 8 from the CTCP paper and get similar qualitative results. Our results show that CTCP does indeed perform better than TCP for high speed and long distance links. Performance optimizations in our CTCP implementation to match the CTCP implementation in the paper would allow us to match the results quantitatively as well.

6 Future Work

Possible extensions of this project could include comparing CTCP with one loss based approach like HSTCP and one delay based approach like FAST. Additionally, it would be really interesting to observe TCP fairness and RTT fairness experimentally for CTCP and their absence in HSTCP or FAST. This would mean experimentally measuring the stolen bandwidth in all three cases. However, this would require more complicated network structure and might require the use of a router implementation.

Another possible extension of this project could be testing the Windows implementation of the CTCP and comparing observed throughputs against our implementation.

7 Acknowledgements

We would like to thank Professor Nick Mckeown, Professor Keith Winstein and TA's Saachi Jain and Emre Ombray for their valuable insights throughout the project. We thoroughly enjoyed working on this project and it was a great learning experience for us. (Special thanks to Keith for his work on the Sourdough congestion controller framework and the Mahimahi link emulator.)

References

[1] Mark Allman, Vern Paxson, and Ethan Blanton. *TCP congestion control*. Tech. rep. 2009.

[2] Sally Floyd. "HighSpeed TCP for large congestion windows". In: (2003).

[3] Sangtae Ha et al. "Impact of background traffic on performance of high-speed TCP variant protocols". In: *Computer Networks* 51.7 (2007), pp. 1748–1762.

[4] Tom Kelly. "Scalable TCP: Improving performance in highspeed wide area networks". In: *ACM SIGCOMM computer communication Review* 33.2 (2003), pp. 83–91.

[5] Ryan King, Richard Baraniuk, and Rudolf Riedi. "TCP-Africa: An adaptive and fair rapid increase rule for scalable TCP". In: *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*. Vol. 3. IEEE. 2005, pp. 1838–1848.

[6] Ravi Netravali et al. "Mahimahi: Accurate Record-and-Replay for HTTP." In:

[7] Jitendra Padhye et al. "Modeling TCP throughput: A simple model and its empirical validation". In: *ACM SIGCOMM Computer Communication Review* 28.4 (1998), pp. 303–314.

[8] Kun Tan et al. "A compound TCP approach for high-speed and long distance networks". In: *Proceedings-IEEE INFOCOM*. 2006.

[9] David X Wei et al. "FAST TCP: motivation, architecture, algorithms, performance". In: *IEEE/ACM transactions on Networking* 14.6 (2006), pp. 1246–1259.

[10] Lisong Xu, Khaled Harfoush, and Injong Rhee. "Binary increase congestion control (BIC) for fast long-distance networks". In: *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*. Vol. 4. IEEE. 2004, pp. 2514–2524.