DCTCP:
Congestion Control In a Datacenter Network Structure

Jakob Horner and Mohammad Hadi
(Dated: May 13, 2020)

Abstract: In data center network topologies, there are often strict expectations on throughput and round trip time even when dealing with large amounts of data. With normal network message passing approaches, often the large and varied flows of a data center can break the implemented congestion control systems. We attempt to implement a TCP over UDP protocol that offers congestion control explicitly tailored to data center topologies based on previous data center TCP research. This protocol is tested on a virtual data center network made with Mininet.

The git repo can be accessed here: `https://github.com/jakobh7/CSCI_5550_DCTCP`

I. INTRODUCTION

In the past decade, with the increase of social networks, cloud computing, and consolidation of other data into single data centers, it has become extremely important to provide consistent and improved performance. While there have been different strategies employed to address this performance, the implementation of a specific data center TCP protocol is a lucrative prospect that can offer cost savings for data center owners by offering greater performance without upgrading hardware.

The topology of data centers is distinct from common network topologies. Often data centers are created with a "fat-tree" network topology to separate and balance workloads. This "fat-tree" topology has a top layer switch connected to distinct lower-level switches which connect to server racks to process data. These network topologies often use a Partition/Aggregate workflow pattern to split work across switches in a layer and receive information in a real-time fashion. Because of the layered structure of the tree, and the workflow dependency on the workers below low latency is required for every level of the tree so that the overall response can be in real-time. This difference in topology and performance expectation is the driving motivation for implementing a specialized method of network communication.

In TCP network communication, when a packet is dropped - commonly due to overflowed queues in a switch - TCP's reliable data transfer requires the packet to be resent which is extremely costly in terms of latency. Usually TCP's congestion control algorithm adequately limits dropped packets by adjusting how many packets are sent at a time, reducing that number after a packet is dropped. This method is reactive, however, and is not ideal for a system that requires low latency. With knowledge of the common flow patterns of the network you're designing, it is possible to create a more proactive method of congestion control to stop packet loss before a packet gets dropped. This idea of proactive congestion control is central to the previous research on data center TCP.

Most TCP implementations are build into modern operating systems and network ports. This often obfuscates and protects properties and functions of the TCP implementation, making them harder to access and work with. Our implementation of this previous work on Data Center TCP is based on a TCP over UDP model. This not only allows us for easy access to TCP members such as the cwnd structure and incoming and outgoing queues, it also allows us to implement a version of the TCP header that is pared down to the information needed by our specific protocol. Some of these implementation details might add overhead compared to a lower level implementation of the protocols, but having a basic TCP over UDP written in the same format gives us a good benchmark to test against.

Our tests rely on a Mininet virtual network to run benchmark testing. This allows us to create our own fat-tree topology and pass bursts of high message flows through the switches. This way we can replicate the message flow patterns present in real world data centers and provide benchmarks for this implementation without worrying about harming real world data.

II. CONGESTION CONTROL

To understand how the data center TCP improves upon the performance of a typical TCP implementation, we need to understand the importance of congestion control in a data center network and the differences in congestion control strategies.

a. Effects of Congestion Control In a data center network, messages sent to server racks get sent through layers of switches with limited queues to hold messages to forward on. In all cases, switches have a limited memory such that when there are too many messages being sent across the network to a particular switch filling the queue, which, when full, drops any new messages sent to it.

The message flows in a data center analyzed by the previous DCTCP research were shown to be a mixture of consistent smaller data flows with inconsistent bursts of large sized data. These varied data flow allow for a considerable amount of dropped packets from the bursts of large data. When these large flows are streamed out to the switches in the data center, the queues quickly fill and lead to dropped packets.

The common strategy of congestion control is lim-

iting the amount of packets sent across a network by implementing a "cwnd" value as a limit to how many packets can be sent without receiving acknowledgment. Once packets are received and subsequently removed from switch cues, the communication protocol sends an acknowledgment message to alert the sender that it is safe for more packets to be sent into the network. From this common framework, there are multiple strategies to address

b. TCP Congestion Control Strategies TCP's typical congestion control strategy has three main parts: Slow Start, Congestion Avoidance, and Fast Recovery. In Slow Start, cwnd is doubled every time an acknowledgment is received until a variable congestion threshold is reached. After this, cwnd is increased linearly with every ack. With its Congestion Avoidance, TCP waits until it receives an indication that one of its packets has been dropped: either the sender receives 3 duplicate acks or there is a complete timeout since no acks have been received in a reasonable amount of time $given the known RTT of packets for the connection$. If the sender has received 3 duplicate acknowledgments, this indicates that just one package has been lost - a less severe issue - and the congestion control algorithm enters a fast recovery mode. However, if there is a timeout, the congestion control algorithm reverts back to the slow start and reduces the congestion threshold by half so that the slow start is even slower. Fast Recovery allows the cwnd to expand even when it waits on acknowledgment. Until a timeout occurs, the cwnd increases every time a duplicate ack is received.

c. Data Center TCP Congestion Control Strategies Where Data Center TCP's congestion control strategy is significantly different is its active management to determine if a switch is congested before a packet is dropped by marking the TCP header both on the level of the switches and the receiver.

DCTCP's congestion control strategy start from the level of switches. At the switch level, TCP headers are marked as congested if the switch's cue is more full than a given percent. This is a separate cue than the ECE header bit.

On the DCTCP receiver end, each packet is assessed to see if it has been marked as going through a queue with a filled queue. The receiver's ack responding to a marked packet will flag the ECE flag in the TCP header. This gives an accurate representation of filled queues back to the sender.

The sender uses the information gained from the package acknowledgments to calculate an alpha value between zero and one - indicating a less congested network for values of zero and a fully congested network with a value of one.

This allows a DCTCP sender to accurately profile the network and minimize its cwnd before a packet is dropped and has to be resent.

d. Implementation Details and Problems Our implementation of this protocol ran into issues in development and test.

While our TCP over UDP implementation documented below was created for flexibility and usability, it took a considerable amount of development and testing time to create its baseline version. While it might have been necessary to allow for manipulation of the packet header, it was a roadblock to our implementation.

Another main failing of our implementation was an inability to deploy the protocol so that switches could flag the necessary congestion notifications. As we document later in our Benchmark Testing section, we were able to create a virtual network to test on, but failed to properly deploy and send messages with a unique protocol.

III. DATA CENTER TCP OVER UDP IMPLEMENTATION

Our implementation of DCTCP was planned around a TCP over UDP style connection. This was implemented in python to allow for flexibility in working with messages being sent across. This implementation did add overhead that most likely added processing time. To make up for any added processing added by this implementation of the base TCP over UDP, we could test our DCTCP protocol against the basic TCP over UDP we created first.

This implementation offers an easy to use, verbose header structure and parser, overloads for python sockets' send, recv, accept, and connect methods, and a distinct queue structure which groups packet data together with associated socket metadata.

IV. BENCHMARK TESTING

A. Mininet Setup

A typical Data Center topology has multiple layers of switches and has decreasing number of switches as you approach the most interconnected switch(es). We created a simple Data-Center topology based off of the Three-Tier topology. In the Three-Tier Data-Center topology you have 3 layers of switches: core, aggregate and access. The core layer is the highest layer, and its switches connect the topology together and connect the topology to other networks, like the internet. The lowest layer is the access layer which connects the hosts to the network. The middle layer is the aggregate layer and it works as a bridge to connect the access and core layer, connecting hosts to each other and to the outside. Our topology has a single core switch, connected to 2 aggregate switches, with each aggregate switch connecting to 2 access switches. Each access switch had a single host connected beneath it. We chose this topology because it seemed to be a simple Data-Center topology to mimic. We programmed our topology in python and verified that it was connected properly in Mininet by checking the links.

B. Benchmark Results

We were unable to execute the scripts on two separate hosts on Mininet, and trouble shooting did not yield any results. This means we were not able to get any measurements and results in which DCTCP was used between two hosts/ We opened two separate terminal windows connected to Mininet and once we connected a terminal to our topology, we could not get the other terminal to connect to the topology without killing the process of the first terminal. We DCTCP has been verified in linux terminal, and resulted in no errors, but we were not able to get it to run on hosts on our topology. I also tried using xterm to run the scripts on two hosts. Despite not having scripts running on the hosts, we already had benchmarking tools in place. Through the addition of a few lines of code, we added a round-trip time and throughput measurement. We used the python command perf_counter() to mark time the file was sent and the time the file was finished sending. The size of our messages were 1024, so we could find the throughput by dividing size by the time it took to send.

V. CONCLUSION

In this paper, we sought to compare the performance of DCTCP protocol to a more standard protocol, like TCP, to test whether it can offer better optimizations in Data-Centers. While we were unsuccessful in implementing the DCTCP protocol in our Mininet environment we produced a baseline TCP over UDP protocol that is easily modifiable and a Mininet topology matching a common data center structure allowing a baseline structure for future work. We also analyze the previous work on DCTCP and provide insight its overall structure. DCTCP fixes issues that TCP faces in Data-Centers, like Queues being overwhelmed by a sudden large number of small flows and queue buildup, by marking flows aggressively and offering more buffer space.