Retransmission Timeout in TCP
Reproducing Stephen Edge’s "An adaptive timeout algorithm for retransmission across a packet switching network"

Ryan Eberhardt
Stanford University
rebs@stanford.edu

Kathryn Rydberg
Stanford University
rydbergk@stanford.edu

1 ABSTRACT
In this paper, we compare the retransmission performance of various congestion control algorithms. Congestion control algorithms should seek to avoid unnecessary retransmission in order to minimize load on a network, but when packets are lost in a network, waiting too long to retransmit will increase latency and decrease throughput. We measure unnecessary retransmission rates and retransmission latency for TCP Reno, Vegas, Cubic, BBR, and Westwood in a simulated environment. We find that network bandwidth and latency have little effect on retransmission performance when held constant, but when the network has extremely small buffers or unpredictable loss, TODO: finish this sentence: which algorithms perform the best in these scenarios?

2 INTRODUCTION
Retransmission is a critical part of TCP, since packets can get lost or damaged in the network. However, determining when a packet should be retransmitted is difficult. If a sender decides to retransmit packets too soon, it may retransmit them unnecessarily, contributing to unneeded congestion. On the other hand, if a packet is dropped within the network and is retransmitted late, latency is increased. Deciding when to retransmit a packet is a crucial component of a congestion control strategy.

In 1984, early on in TCP, Stephen Edge proposed an algorithm for deciding packet timeout values based on round trip time estimates [3]. Rather than using a fixed retransmission timeout value, which does not account for changing network conditions, Edge argues that an adaptive timeout algorithm is necessary. Since end hosts have little visibility into packet buffers in the network, Edge proposed using acknowledgements to estimate RTT and RTT variance using an exponentially weighted moving average, and to determine timeouts based on these values. This technique ended up becoming the foundation of congestion control in TCP Tahoe.

This paper served as the motivation for our work. Because Edge’s work was in the beginning stages of TCP, there is little prior work it compares to, and the paper lacks an experimental results section. However, we were interested in the goal of the paper: How does one minimize unnecessary retransmissions while also minimizing retransmission latency when packets are lost in a network? We decided to compare a suite of TCP algorithms developed since Edge’s work to see how their retransmission techniques performed. Specifically, we evaluated the algorithms based on the following two metrics we devised:

(1) Retransmission overhead, defined to be the fraction of transmitted segments representing unnecessary retransmissions (retransmissions of segments that were not lost in the network and eventually reached the receiver)

(2) Time-to-retransmit, defined to be the amount of time the sender waited before retransmitting a segment that was lost within the network and would have otherwise never reached the receiver

The following paper is structured as follows: Section 3 reviews Edge’s work on retransmission, as well as the retransmission strategies that have followed it in newer TCP congestion control schemes. Section 4 describes the experiments we ran in more detail, and Section 5 presents the results. Section 6 presents ideas for future work, and Section 7 concludes the paper.

3 BACKGROUND
Edge’s paper about retransmission in TCP showed the importance of a retransmission algorithm. Retransmitting a packet too soon adds unnecessary traffic to the network, while retransmitting too late causes delay. Edge’s research illustrates that the retransmission algorithm is critical to the performance of TCP.

Since Edge’s paper, there have been many iterations of TCP. In our project, we look at Reno, Vegas, Cubic, BBR, and Westwood. Each algorithm identifies congestion and manages the congestion window in a different way.

Reno’s retransmission algorithm has two ways of estimating when a packet has been dropped. In both cases, since a dropped packet indicates too much congestion in the network, the congestion window is decreased. The first case is if 3 duplicate ACKS have been received; in this scenario, the
congestion window is halved. Reno uses duplicate ACKS to indicate loss because it receiving duplicate ACKS indicates that the receiver is successfully receiving packets, but that those packets followed a lost packet (otherwise the ACKNO would have increased). The second case is if the retransmission timeout (RTO) timer times out; in this case, the congestion window is reduced to 1 segment and the algorithm enters slow start. Reno uses Additive Increase, Multiplicative Decrease (AIMD) in order to slowly increase the size of the congestion window until it experiences a loss, at which point it quickly decreases the size of the congestion window [1].

The Vegas implementation of TCP has 37 to 71% higher throughput than Reno does because it is able to achieve one-fifth to one-half of the loss that Reno experiences. Vegas does not have an aggressive retransmission strategy that steals bandwidth from other flows, but instead uses bandwidth more efficiently. Specifically, Vegas extends Reno’s retransmission algorithm:

1. First, Vegas is able to obtain more accurate measures of RTT. It does so by reading the system clock every time a segment is sent and then when the ACK for the segment is received, allowing it to calculate the RTT for an individual packet.
2. Vegas then uses this more accurate RTT estimate to determine retransmission.
3. When a duplicate ACK is received, it checks to see if the current time and the time the segment was sent is greater than the timeout value. If so, it immediately resends the segment. With Vegas, the sender does not always have to wait to receive three duplicate ACKS before retransmitting.
4. When a non-duplicate ACK is received after retransmitting a segment, it checks again to see if the time since the segment was sent is greater than the timeout value. If so, it automatically resends the segment without waiting for a duplicate ACK.

Vegas has a more fine-grained way of measuring RTT, which allows it to anticipate loss much earlier than Reno [2]. This decreases delay and increases throughput for the packets that are actually lost, but also potentially retransmits packets that will eventually reach the receiver since it doesn’t wait as long to retransmit a segment.

Cubic modifies how the congestion window grows; rather than growing linearly, like AIMD used in other TCP congestion algorithms, Cubic instead uses a cubic function for window growth. The growth function is also not based on RTT, but instead a function of time since the last “congestion event”; this allows flows with different RTTs to increase their window size at the same rate, which improves fairness. Having a cubic function for window increase, rather than a linear function, improves stability [4].

Reno, Vegas, and Cubic are all loss based algorithms; they regulate congestion based on when loss is experienced. One disadvantage to this approach is that this causes high transmission delay, since loss must be experienced before retransmitting and regulating congestion; by the time loss is experienced, there’s already congestion in the network. BBR tries to address this issue instead uses bufferbloat to identify congestion. Like Vegas, BBR keeps a continuous estimate of a flow’s RTT.

Westwood makes modifications on TCP Reno. In Westwood, the sender attempts to estimate the end-to-end bandwidth to determine how many packets to send. Like Reno, Westwood still uses 3 duplicate ACKS and an RTO timer to determine when to retransmit; however, with Westwood, the bandwidth helps set the congestion window.

All of these algorithms use different metrics for determining when there is congestion and when to retransmit a packet, and each algorithm optimizes for something different. In our paper, we compare these algorithms in terms of the frequency with which they transmit a packet that didn’t need to be retransmitted, as well as how long it takes to retransmit a packet that should be retransmitted.

4 EXPERIMENTS

An ideal comparison would involve comparing outcomes for different packet retransmission strategies while holding the rest of the congestion control protocol constant. However, we looked at the Linux implementations of Reno, Vegas, Cubic, BBR, and Westwood, and the timeout logic seems tightly coupled to the rest of the implementation. To simplify our experiments, we decided to compare the retransmission performance of entire implementations, rather than of the isolated retransmission algorithm.

In order to compare different TCP retransmission algorithms, we measure the retransmission characteristics of different implementations under adverse network conditions. There are two sides to this evaluation: How often does an implementation unnecessarily retransmit packets that have not been lost? Additionally, how long does it take an implementation to retransmit when packets have been lost?

To do this, we first use `sysctl` with the `tcp_congestion_control` option to set the kernel’s congestion control algorithm prior to each round of data collection. We are currently testing Reno (which implements Edge’s algorithm proposed in [3]), Vegas, Cubic, BBR, and Westwood. Then, we run a full-throttle TCP flow over a simulated network in Mininet. We run a packet capture at both the sender and receiver. For each packet dropped by the network, we can use timestamps in the capture to determine how long it took the sender to retransmit the packets, and we can count the number of times
Figure 1: Topology for retransmission performance tests. A primary sender maintains a full-throttle TCP stream to the primary receiver using iperf. A series of additional sender/receiver pairs add load to the network.

4.1 Data collection with simulated flows

Our first challenge was to figure out how to operate the simulation environment. We wrote a Python script to generate a Mininet topology with a sender and receiver that use iperf to maintain a full-throttle TCP flow. The sender and receiver are separated by two switches, and additional pairs of senders/receivers are added to the network to add load to the switches (Figure 1), where each sender uses iperf to maintain a full-throttle flow with the corresponding receiver. We run tcpdump on both the primary sender and receiver to record the TCP segments that were sent and received.

Analyzing the recorded segments was unexpectedly challenging. We started by using tshark (a terminal-only version of Wireshark) to export data from the packet captures to JSON, which we could then work with in our Python script. However, we could not find a way to exclude the TCP payload from the export, since we are only interested in the timestamps and sequence numbers. This created a significant performance problem; since Python needs to parse the entire tshark output when deserializing from JSON, it consistently ran out of memory when deserializing any respectably-large packet capture. We tried using a native Python pcap-parsing library called pcapkit, but found it has an unwieldy interface, and we couldn’t figure out how to use it to extract any TCP segment information. We eventually found a tshark option intended for streaming to Elasticsearch databases that outputs each packet’s information as an individual JSON object on a separate line. Using this option, we were able to stream data from tshark to our Python script without running out of memory. Performance is still slow, and we would like to figure out how to properly use pcapkit (or alternatives, such as scapy) in the future.

4.2 Inducing packet loss

We sought to examine retransmission performance under two conditions:

1. Networks with a modest number of flows, in which a congestion control algorithm may be able to construct a model of the network in order to predict performance and loss
2. Networks with a large number of flows, in which loss is essentially a random process

To simulate the former condition, we experimented with modifying link bandwidth, buffer size, and latency. We also attempted to add jitter to the link latencies, as this would be much more representative of a real network and would make it more challenging for congestion control algorithms to model the network. However, we were not able to figure out how to do this in the time available. Whenever we set a jitter parameter on the Mininet links, we were not able to transmit any packets across them, and despite searching for other examples of how to do this, we could not find any that worked.

To simulate the latter condition, we varied the loss parameter of the Mininet links. This parameter specifies a probability (as a percentage) that a packet should be dropped in transit to a switch. This may more effectively simulate networks with a large number of flows, where loss is essentially a random process, as simulating a large number of flows in Mininet is too computationally intensive.

In our experiments, we decided network parameters before running tests and did not modify them in the middle of the test. This is something we would like to do in future work, as it would allow us to see how algorithms adapt to changing network conditions.

4.3 Data analysis

Recall that we are interested in how many unnecessary retransmissions an implementation makes, as well as how long an algorithm takes to send necessary retransmissions. To do this, we devised two metrics with which to evaluate the congestion control algorithms:
Retransmission overhead is defined to be the fraction of transmitted segments that are unnecessary retransmissions (i.e., retransmissions of a segment that was not lost and eventually delivered). Bandwidth, buffer size, and delay are varied to test a congestion control algorithm’s ability to model the network, and loss is varied to test its ability to respond to random packet loss.

**Retransmission overhead** is the fraction of transmitted segments representing unnecessary retransmissions (retransmissions of segments that were not lost in the network and eventually reached the receiver). To compute this, for each transmitted sequence number, we count the number of times the sequence number was sent by the sender (denoted $n_{tx}$) as well as the number of times it was received by the receiver (denoted $n_{rx}$). The receiver only needs to receive a segment once, so the number of times a segment was unnecessarily retransmitted is $n_{rx} - 1$. We can divide the total number of unnecessary transmissions by the total number of transmissions in order to compute the fraction of transmitted segments representing unnecessary retransmissions.

**Time-to-retransmit** is the average amount of time the sender waited before retransmitting a segment that was lost within the network and would have otherwise never
reached the receiver. To compute this metric, we iterate over each sequence number and identify the number of necessary retransmissions $n_{\text{necessary}}$. This is computed as the total number of times the sequence number was transmitted minus the number of unnecessary retransmissions, i.e. $n_{\text{tx}} - (n_{\text{rx}} - 1)$. As a simplifying assumption, we assume that the first $n_{\text{necessary}} - 1$ transmissions were the transmissions that were lost in the network, and the $n_{\text{necessary}}$th transmission is the first one that reaches the receiver. Then, we compute time-to-retransmit as the time between each of the first $n_{\text{necessary}}$ transmissions.

5 RESULTS

Our results for measuring retransmission overhead are shown in Figure 2.

We found that bandwidth and delay have little effect on retransmission performance. This should be expected; with fixed bandwidth and delay, the network is easy to model, and congestion control algorithms can achieve minimal loss. It would have been more interesting to see how the algorithms perform with fluctuating bandwidth and delay, which we discuss in the Future Work section.

We found that buffer size does have a meaningful effect on retransmission performance (Figure 2b), as tiny buffers cause more frequent packet loss (although not as frequent as we expected). Unfortunately, because we did not see frequent packet loss via tiny buffers, the difference in performance between the congestion control algorithms is not significant, and we could not attribute the performance differences to the algorithms’ designs.

Increasing rates of random packet loss also appear to affect retransmission overhead (Figure 2d), although we did not have sufficient time to look into why the congestion control algorithms exhibit the behavior we observed.

Our results for measuring time-to-retransmit are shown in Figure 3.

6 FUTURE WORK

Due to time constraints in this hectic quarter, we were unable to complete all of the experimentation and analysis that we would have liked to have done. Here are some things that we would have worked on:

1. We would have like to have spent more time analyzing the performance of each algorithm to explain why it performs the way it does.

2. We also would have spent more time trying to emulate a “real” network using Mininet. Our network has no jitter in link delay, and even with buffers as small as two packets in size, we observe very little packet loss. Perhaps we could try using a different emulator, such as the Pantheon emulator.

3. We would like to see how algorithms handle changing network conditions. For example, we might see what happens when many more flows are added to the network while data collection is running.
(4) Ideally, we would also find a way to benchmark individual retransmission algorithms rather than entire congestion control implementations.

7 CONCLUSION

When it comes to segment retransmission, a congestion control algorithm must balance two contradicting goals: A good algorithm should minimize the number of retransmissions, but it must also not wait too long before retransmitting a segment that has been lost in the network. In this paper, we have looked at retransmission performance in terms of retransmission overhead, defined to be the fraction of transmitted segments that are unnecessary retransmissions, and time-to-retransmit, defined as the amount of time a sender waits before retransmitting a lost segment. We have created a test environment that uses a simulator to track these metrics under different network conditions. Unfortunately, more analysis is required to better understand how various retransmission approaches affect retransmission performance under these metrics.

REFERENCES