Aggressive TCP Slow Start

Soonho Kong  
soonhok@cs.cmu.edu

Qinsi Wang  
qinsiw@cs.cmu.edu

Gabriel Weisz  
gweisz@cs.cmu.edu

I. BACKGROUND

TCP Slow Start is a mechanism that is intended to allow a data transfer to start at a low bandwidth and quickly reach the peak bandwidth that it can use, subject to congestion control constraints that are enforced by intermediate network nodes. The current version is defined by RFC 5681 [2]. In general, the mechanism used for congestion control in TCP networks is that a congested router will randomly drop a packet as a signal to the sending system that it is using too much bandwidth and needs to slow down. Sending systems control the bandwidth that they are using by controlling the number of packets that can be in flight at once (the congestion window), linearly increasing this number until a packet is dropped, at which point it is halved. At steady state, TCP connections exhibit a sawtooth pattern where the bandwidth repeatedly slowly increases to some peak, at which point it drops and starts increasing again.

TCP Slow Start is meant to address the issue that connection must start with a pretty small window in order to support the slowest network connections, and it can take a very long time for the connection to ramp up from the initial congestion window size to the steady state congestion window size. During TCP Slow Start, the congestion window size increases exponentially rather than linearly, which in general allows the connection to reach the steady state bandwidth quickly. In practice, in addition to performing slow start at the beginning of a transfer, most modern TCP implementations also perform slow start periodically during a transfer in order to adjust to changing congestion conditions throughout the transfer.

While TCP slow start works well in general, a data transfer must not experience dropped packets during slow start, because if it does the transfer will move from exponential increase to linear increase very quickly. If this occurs, the transfer might never reach the steady state that will allow the transfer to use the maximum possible bandwidth, and network performance will suffer. In wireless network environments, which can see high packet losses due to the nature of the medium, and in data centers, where crowded networks can lead to routers that randomly drop packets if they are not configured for any sort of quality of service, dropped packets during slow start can be a real problem. Since a large number of the data transfers in today’s Internet is composed of web traffic, which is comprised mainly of text files and images that are small, transfers may never reach the peak transfer rate even with slow start to accelerate the transfer rate’s movement towards the peak, and if a single packet is lost during the transfer the overall transfer rate seen by a client can suffer tremendously. Figure 1 provides a simplified illustration of how TCP Slow Start works, and shows how a client experiencing a dropped packet even one round trip time might never reach peak congestion window size, which means that it will never reach peak bandwidth utilization. While previous research has investigated changes to TCP to support data center...
workloads \cite{1} and modifying TCP retransmission timeouts to prevent congestion collapse \cite{3}, we no of no recent efforts to improve the performance of TCP communications by preventing spurious slow start terminations.

II. PROPOSED SOLUTION

We propose Aggressive TCP Slow Start as a possible mechanism to assuage this problem. Aggressive TCP Slow Start allows the sender to preemptively retransmit packets during slow start in order to prevent packet loss during slow start from reducing overall transfer rates. We propose two different mechanisms to allow for this preemptive retransmission, and show that both of these mechanisms increase the overall transfer rate experienced by the client without adding undue overhead to the network. The intuition behind these results is that during slow start, a connection is sending much fewer packets than the network will allow the connection to send, and the additional packets that are sent only represents a small increase in the number of packets that standard TCP will send. In addition, the mechanisms that we propose only require a modification of the TCP stack in the sender - neither the receiver nor intermediate nodes in the network need to be modified in any way. We felt that limiting the number of changes to the network infrastructure as a whole was essential if the mechanisms that we propose is to be adopted. The two mechanisms that we propose are as follows:

- Mechanism A (Figure 2): This mechanism leaves the packet at the head of the send queue with some probability. This mechanism is interesting for two reasons:
  1) This mechanism actually allows packet to be preemptively retransmitted more than once. This could be helpful if a loss was due to a real medium loss, or in a data center that is really congested where a lot of packets are being dropped. Since the repeated packets are sent in succession, there will not be that much difference in the spacing of the packets with this mechanism when compared to the spacing of the packets with the first mechanism.
  2) Packets are only sent when the congestion window mechanism allows packets to be sent. Thus this mechanism is closer to the TCP congestion control semantics of using the congestion window size to limit the total number of packets that are in flight. While it is true that reducing the intermediate systems’ ability to signal that this connection should slow down by dropping a packet stretches the semantics of the congestion window algorithm a bit, this mechanism is not blatantly breaking the idea of the congestion window as the first mechanism does.

- Mechanism B (Figure 3): This mechanism is very simple: when a packet is sent during slow start, rather than sending the packet once, send it twice with some probability. This probability must be chosen carefully in order reduce the chance of negatively impacting the network. This mechanism may be well suited for data center networks where there is no medium induced packet loss, but packets are only lost due to congestion control. If packets are only lost due to congestion control, there is no real correlation between lost packets, so if a packet is sent twice in rapid succession, it is not more likely that the second instance would be lost if the first one was.

Both mechanisms take a single parameter for retransmission probability, and each method can work with a static or dynamic retransmission probability: with a static retransmission probability, the retransmission probability is always the same; with a dynamic retransmission probability, the retransmission probability is divided by the current congestion window size prior to determining whether or not to retransmit. In the dynamic mechanism, earlier phases of slow start are more likely to retransmit than later phases.

III. METHOD

We tested our idea by modifying the NS-2 network simulator to implement Aggressive TCP

![Figure 2: Mechanism A](image-url)
Slow Start. We created a new TCP agent based upon the original NS-2 vanilla TCP agent, giving it a new protocol name “Agent/TCP/Agr”. This allowed us to test the original TCP implementation and our modified implementation side by side. We implemented both of the proposed Aggressive TCP Slow Start mechanisms, allowing the mechanism to be selected via environment variables. In order to control the level of aggressiveness, we also pass the retransmit probability in as an environment variable, and we also implemented a dynamic retransmit probability where the retransmit probability is scaled by dividing it by the current congestion window. We used two different experimental configurations to test out mechanisms, Figure 4 shows a single source, single receiver configuration, which we used to benchmark the performance of our mechanisms against each other and unmodified TCP, in order to see the raw performance capability. Figure 5 shows a configuration with two sources, one using unmodified TCP and one using a modified TCP mechanism, that share a single bottlenecked link to a single sink, that was used to see if our modified TCP implementation would negatively affect other, unmodified senders, and if so what the extent of that impact would be. These experiments were intended to see what the effect of our changes would be if they were incrementally deployed over the Internet.

IV. Experiments

Our first experiments were intended to see which mechanism worked the best. We used the single source configuration, and experimented with loss rates varying between 0-5% in increments of 1%, and going up to 25% in increments of 5%. For each loss rate, we tried all four methods (mechanism A or B, with static or dynamic retransmission probability), with retransmission probabilities varying between 0 and 95%. Our next experiments were intended to see how the best performing mechanism worked against standard TCP. We chose the best overall winner from the first experiment and tested it against unmodified TCP, both using the single source configuration. Again, we experimented with loss rates varying between 0-5% in increments of 1%, and going up to 25% in increments of 5%. For each loss rate, we varied the number of bytes sent by the client (at the user level, not the TCP level) from 50 to 500 kilobytes, and repeated each experiment 100 times. We collected overall average data for each trial, as well as data for the worst 5% of the trials between 0 and 5%, representing the most unlucky senders. Our final experiments were designed to determine whether or not our modified mechanism would negatively impact unmodified TCP clients using the same connections. We used the two source configuration with one unmodified TCP client and one modified TCP client. We varied the packet loss rate between the sending node and the shared intermediate node between 0-5% in increments of 1%, and going up to 25% in increments of 5%. Again, we varied the number of bytes sent by the client between 50 and 500 kilobytes, and repeated each experiment 100 times. We also used the
Figure 6: Transmission size vs Transmission Time for different mechanisms
Figure 7: Retransmission probability vs total transmission time for a single modified and unmodified sender
Figure 8: Packet Loss Rate vs Transmission Time for the unmodified sender in a two unmodified sender and mixed sender scenarios
two source configuration with two unmodified TCP clients in order to have control data for the impact of our modification. We collected overall average data for each trial, as well as data for the worst 5% of the trials between 0 and 5%, representing the most unlucky senders.

V. RESULTS

Figures in 6 shows the results of the first experiment. From the figures, it is clear that Method B with static retransmission probability worked the best. The best performance was seen with the highest retransmission probability (which is a fairly obvious result), so we used a retransmission probability of 95% for the remainder of the experiments. We believe that the static method was better than the dynamic method because of the monotonic increase in performance that was observed with increased retransmission probability - so decreasing the chance of retransmission with higher congestion window size would not increase performance over a static retransmission rate. We did anticipate that this high retransmission probability might negatively affect other clients, but as later results will show, this was not the case.

Figure 7 shows the results of the second experiment. As we expected, the modified mechanism was always faster than unmodified TCP. Figure 9 and 10 shows a summary of these results - for high loss rates the improvement was almost 45%. The worst case results between 0 and 5% improved by up to 10%, but out modification was worse with 0% loss, which makes sense because if there is no loss, then additional overhead is real.

Figures in 8 show the results of the third experiment. The results show that in many cases, the unmodified sender actually experiences increased performance when paired with a modified sender as opposed to two unmodified senders. Figure 11 and 12 show the results of the average change in performance seen by the unmodified sender in the mixed modified/unmodified sender scenario. The unmodified sender loses performance in some of our trials, but actually gains performance in others. When averaged over the worst case, the loss in performance was under 2%, and the best case increase was 1%.

VI. CONCLUSION

We have shown that Aggressive TCP Slow Start, our proposed modification to the TCP Slow Start mechanism, is an effective way to increase TCP transfer performance in networks with small to moderate packet losses. We tested two different mechanisms for Aggressive TCP Slow Start, and found that while both mechanisms were effective, Mechanism B was more effective than Mechanism A. We found that when a modified client shared a congesting link with with an unmodified client, the penalty that the unmodified client paid was
minimal, and in some cases the performance of the unmodified client was actually increased. We believe that this is because the modified client completed its transfer earlier, freeing up bandwidth for the unmodified client. Since our mechanism only requires modifying the TCP stack on the sending computer, we believe that it is realistic to think that our modified mechanism can be deployed to a large number of computers on the internet, particularly if we implement our change to the Linux TCP stack.

References

