

Relentless Congestion Control

Matthew Mathis
Pittsburgh Supercomputing Center
mathis@psc.edu

ABSTRACT

Relentless congestion control is a simple modification to AIMD congestion control: instead of halving $cwnd$ after a loss, $cwnd$ is reduced by the number of lost segments. It can be modeled as a strict implementation of Van Jacobson's Packet Conservation Principle.

Relentless congestion control has exactly unity gain, which is expected to make it much easier for network devices to accurately control traffic.

Relentless Congestion Control is not TCP-friendly. It requires that the network allocates capacity through some technique that segregates the traffic into flows or classes and can send different congestion signals to each.

Our goals are to illustrate some new protocol features and properties might be possible if we relax the TCP-friendly paradigm, and to create opportunities for deploying for these new algorithm in the Internet.

1. INTRODUCTION

Relentless congestion control is a simple sender side modification that can be applied to almost any AIMD style congestion control: instead of applying a multiplicative reduction to $cwnd$ after a loss, $cwnd$ is reduced by the number of lost segments. The resulting $cwnd$ is normally the same as the actual window of data that was successfully delivered during the lossy RTT. If the application is keeping up with the network and other traffic does not change, this is exactly the correct $cwnd$ to fill the network on subsequent RTTs.

Relentless Congestion Control conforms to neither the details nor the philosophy of current congestion control standards[1]. These standards are based on the so called "TCP-friendly" principle, that the Internet can attain sufficient fairness by having relatively simple network devices send uniform congestion signals to all flows, and mandating that all protocols have equivalent responses to these congestion signals.

To function optimally in a shared heterogeneous environment, Relentless Congestion Control requires that the network implements some form of traffic control that can send different congestion signals (segment losses, ECN marks or queuing delay) to each flow or class of flows. This gives the Internet Service Provider the ability manage how its limited resources (data capacity and buffer space) are allocated to different flows, independent of the details of the specific congestion control algorithms that may be in use. This alternative congestion control paradigm is described in a separate

document under consideration by the ICCRG[2].

Example algorithms that can be used to do this include Fair Queuing[3], Approximate Fair Dropping[4], etc, all of which allocate network capacity on the basis of data rate, or economic fairness mechanisms, such as those proposed by Kelly[5] and Briscoe[6]. The economic fairness mechanisms can potentially control traffic on the basis of the total congestion that it causes. In the short term the existing QoS mechanism is provides sufficient traffic segregation to deploy Relentless TCP. The salient features of all of these algorithms are that they can segregate the traffic into distinct flows or classes of flows, and can send different congestion signals to each such that the network has ultimate control over how its own capacity is allocated.

Although it would be natural to assume that requiring network support would be a barrier to deploying Relentless TCP, quite the opposite is true. Failing to have the proper traffic controls in the network gives Relentless TCP an unfair advantage over other congestion control algorithms. As a consequence users who perceive that they are suffering from performance problems might be motivated to install Relentless TCP irregardless of their ISP's ability to control their traffic. The real issue is what should (or will) the ISPs do in response to this situation.

Relentless Congestion Control offers an especially valuable new property: the TCP portion of the control loop has exactly unity gain. This property is ideal from the standpoint of the traffic controllers within the network – if some flow is using 1% too much bandwidth, then dropping 1% of its segments will bring it back to the proper data rate one RTT later. Likewise if a flow is occupying 10 segments too much queue space, then dropping 10 segments will bring it back to the proper queue occupancy on the next RTT. This unity gain property should make it vastly easier to implement traffic controllers in that accurately manage utilization and queue occupancy across a huge range of scales.

Note the duality here, Relentless Congestion Control both requires that the network controls traffic while at the same time, it makes it easier to do so.

Although Relentless TCP offers a number of interesting and potentially valuable new properties, we are not making any claims that our particular algorithm or implementation is optimal. The results here are very preliminary, and can certainly be improved over time.

Our goal is to create an entrée for a bigger set of issue: how should the Internet allocate resources? What is the proper role for the TCP-friendly paradigm, and how can we move beyond it's traditional interpretation and enforcement? We

want to demonstrate some of the scaling problems intrinsic to AIMD congestion control, and expose the burden that AIMD compatibility imposes on the Internet by showing an alternative. The deployment strategy is designed to cause the deployment of some minimal form of traffic segregation via QoS and to establish an installed base of new congestion control algorithms that are not at all compatible with standard TCP.

Relentless TCP was optimized to foster these longer term goals rather than more traditional goals of optimizing some metrics of performance or fairness. We have deliberately foregone some enhancements to algorithm in order to avoid introducing some distracting side discussions. These improvements can be introduced later, after we as a community, have addressed the larger questions about deployability and the desirability of changing the Internet congestion control paradigm.

2. IMPLEMENTATION

The simplest mental model of Relentless Congestion Control is a strict implementation of Van Jacobson’s Packet Conservation Principle[7]. During recovery, new segments are injected into the network in exact accordance with the segments that are reported to have been delivered to the receiver by the returning ACKs. As long as the network does not lose all segments or ACKs, and the SACK scoreboard[8] chooses appropriate data to retransmit, this will clearly converge in one round trip time to exactly the window successfully delivered during the lossy round trip. Since this quantity of data was successfully delivered during a lossy round trip it is both the largest justifiable estimate as well as the most accurate estimate of the maximum capacity of the network, and a good starting point for subsequent RTTs.

Algorithms that open the window, Slow-Start and Congestion Avoidance, are only permitted when the receiver is reporting contiguous data, which implies that there has been at least one full loss-less RTT since the last retransmission.

Our Relentless Congestion Control is implemented as dynamically loadable kernel module that is easily installed in a stock Linux kernel. Unfortunately, there are some *cwnd* adjustments built into the common loss recovery code that is shared by all congestion control modules. The Relentless congestion control module overcomes these adjustments by setting *ssthresh* to the old *cwnd* minus losses, such that any non-packet-conserving *cwnd* adjustments are offset by a (hopefully) short interval of slowstart following recovery.

We strive to understand and eliminate these situations where the mainline code differs from the ideal packet conservation principle. In the future we may have to resort to an optional kernel patch to suppress some of the *cwnd* adjustments when Relentless congestion control is in effect. (The *ssthresh* workaround in the dlkm would still work if the kernel patch was not applied.) For implementation details and source code, see the web page[9].

To foster deployment and provide some network safety, our implementation requires that its traffic be marked with the “Lower Effort” (LE) DSCP[10]. Routers and other network devices are supposed to schedule LE traffic at a lower priority than the default “Best Effort” service. To the extent that LE service is supported it permits Relentless TCP to co-exist with standard AIMD TCP using default Best Effort service, without significantly perturbing the latter.

There may be concerns about the consequences of publish-

ing our code. However, any competent kernel programmer can reconstruct this work from this paper. The Relentless algorithm is substantially simpler than the carefully tweaked algorithms that it replaces. The majority of the complexity of our implementation comes from not making any changes to the shared recovery code in order to preserving compatibility with other dynamically loadable congestion control algorithms.

3. QUEUE CONTROLLERS

There has been a lot research into how to optimally manage queues in network devices, starting with Sally Floyd’s paper on Random Early Detection[11]. None of the proposed solutions is entirely satisfactory. The difficulty is that to optimally control traffic from AIMD congestion control requires that the queue controller estimate some parameters of the traffic, such as effective *RTT* or *cwnd*. These parameters are needed to estimate the drop rate required to attain a specific data rate or queue occupancy. Furthermore, in high speed devices these calculations have to run at line rate in silicon. So far all known solutions are, at best, engineering compromises, and are suboptimal in environments that are significantly different than their design point.

We claim that simplifying TCP’s response to congestion will make it much easier to implement simple lightweight queue controllers that can accurately manage queue lengths across a huge range of network scales.

For example we define a simple “Baseline Queue Controller” as follows: it monitors the minimum queue length over discrete time intervals, and in each interval it randomly discard as many packets as the minimum queue length was above the set point during the previous interval. This is exactly the correct loss to bring the minimum queue length down to the set point on the next RTT. The periodic queue management interval can be any convenient interval somewhat longer than one RTT. (e.g. 1 to 10 times the RTT). Although the accuracy of this controller is better when it’s polling interval is matched to the actual RTT, it clearly is not very sensitive to mismatch. When the polling interval is 10 times the RTT, its average error would only be 5 packets, as long as the traffic was from a single Relentless TCP flow.

4. PROPERTIES

These are preliminary results, which are expected to improve over time.

4.1 Performance Model

Clearly the traditional $1/\sqrt{p}$ performance model[12] will not apply. A simple analysis suggests that the steady-state data rate for Relentless TCP is $1/3p$. In an absolutely quiet network with drop tail queues you expect one loss per three round trips: when the bottleneck queue is full and congestion avoidance increments the window, it will take one RTT before the sender observes the subsequent loss, one RTT to repair the loss, and one RTT of congestion avoidance to reach the next *cwnd* increment. In real networks the losses are observed to be much more bursty: clusters of losses followed by multiple loss-less RTTs. Our implementation is also hampered by the need to offset *cwnd* adjustments that occur in the shared TCP recovery code.

4.2 Long Fat Networks

In our moderately large window tests (a Gigabit network with a 70 ms RTT) we were able to routinely attain better than 500 Mb/s even though the loss rate was in excess of 0.07%. Attaining this rate with standard AIMD congestion control requires a loss rate that is less than 0.00003%. Clearly Relentless TCP can tolerate many orders of magnitude higher loss rates than other TCPs.

However the packet traces reveal several anomalies not directly related to Relentless Congestion Control, including spurious retransmissions, intermittent massive packet reordering, and various unexpected limitations on data transmissions, when there seems to be plenty of available window.

This raises an interesting side issue: Relentless TCP exercises the SACK and recovery code several order of magnitude more vigorously than other congestion control algorithms. Getting Relentless TCP to the point where it really implements the packet conservation principle over a wide range of conditions may require raising the robustness of the recovery code to a new level.

4.3 Relentless TCP is not AIMD-friendly

Without traffic controls or some other mechanism to keep Relentless TCP in check, Relentless TCP has the potential to overwhelm standard TCP at any shared bottleneck. Consider the following thought experiment: two flows, one Relentless and one standard TCP having the same RTT but sharing a common bottleneck that is randomly dropping packets with uniform probability. Both flows increase *cwnd* at the same rate, but each loss has a larger effect on the standard TCP, such that it keeps forfeiting capacity to the Relentless TCP. It can be shown that when the two flows come into equilibrium, the window size of the Standard TCP in segments will be approximately the square root of the window size of the Relentless TCP.

Note that with small average windows (less than 4 segments) or when timeout driven, Relentless TCP and Standard TCP are expected to have identical behaviors.

5. REMOTE VIDEO UPLOAD

To help understand how Relentless TCP makes traffic control easier, consider the following thought experiment: You are trying to upload digital video from a remote site through a high delay satellite link, shared by some small amount of other traffic. With either Fair Queuing or Lower Effort service in the ground station it can maintain separate queues for the TCP video upload and the smaller flows. As long as the satellite link has fewer losses than one per several RTTs, Relentless TCP can maintain a standing queue of video data at the ground station. As long as the ground station has queued video data, it can optimally allocate the link to the small flows first (according to an explicit policy) and completely fill the rest of the link with video. The small flows see minimal delays while the link is 100% utilized.

To make this example more concrete, imagine that you are trying to fill 10 Mb/s satellite link with a 1500 Byte MTU and a 600 ms RTT. It would require about 500 packets in flight to fill this path. With Relentless TCP and a Baseline Queue Controller using a 1 second sample interval, you would expect on average to see about one drop every 2 sample intervals (roughly once per 3 RTTs, or about every 1500 packets).

To mask the noise caused by the small flows, the Baseline Queue Controller could either have an extra long sample

interval or slightly higher setpoint. Failing this only compromises the extent to which the system can compensate for the jitter introduced by the small flows, but not its ability to fill most of the link using a very straightforward configuration.

Contrast this to standard AIMD congestion control: the queue at the bottleneck needs to hold more than 500 packets, and TCP *cwnd* needs to sawtooth between 500 and 1000 packets. There needs to be one loss every 1000 RTTs (twice 500 due to delayed ACK), or about every 700,000 packets – 500 times¹ lower loss rate! To do this optimally, the queue controller has to estimate when the queue is holding slightly more than half of TCP’s total window. Otherwise after recovering from one drop or ECN mark, TCP’s window will be too small. If the queue controller gets it wrong, or if there are too many background losses (e.g. due a high Bit Error Rate), then a single standard TCP flow can not fill the link.

A delay sensing TCP such as FAST[13] or Compound TCP[14] might do better than Reno TCP in this environment, since it might be able to sense the queuing delay at slightly above 500 packets and use it to control TCP’s window without causing losses or a large queue, but it still requires a relatively low background loss rate. (But note that a relentless variant of a delay sensing TCP is likely to do even better).

Since some form of either Fair Queuing or Less Effort service is supported by many routers, this video upload example can be deployed today, on an as needed basis.

6. DEPLOYMENT

Relentless TCP has the potential to cause the widespread deployment of Less Effort service. In the absence of LE support, Relentless TCP will raise the performance of users who install it on their own end-systems. We plan to encourage a couple of specific user communities to adopt it, much the same way we marketed web100. As more people start using Relentless TCP it has the potential to be disruptive to other users. The most effective way for most ISP’s to respond would be to enable and properly configure the LE support already present in their networking equipment.

Once an ISP enables LE service, Relentless TCP traffic would effectively have lower priority than standard TCP running over “Best Effort” IP service. Since Relentless TCP performs better than standard TCP under high loss it has a better chance of filling available network capacity. Even though it would be at a lower priority, it might not do significantly worse than Standard TCP, especially when averaged over the long term. In portions of the Internet where users have more data than the available network capacity but standard TCP is failing to make full use of the capacity, this creates a substantial win-win situation for both the users and the ISP.

¹Note that the ratio of the steady state loss rates between Relentless TCP and Standard TCP is approximately the same as the pipe size in packets.

7. CONCLUSION

Relentless congestion control is a simple sender side modification TCP that is not “TCP-friendly.” It has a number of important properties, including a performance model that scales with $1/p$ instead of $1/\sqrt{p}$ and a one-to-one response to congestion signals, making its congestion response independent of scale.

Our implementation is easily installed in a Linux kernel. The packets it sends are marked for Less Effort DSCP to facilitate deployment. Over the medium term term, this has the potential to cause LE service to become widely enabled. If LE service is not enabled, Relentless TCP has the potential overwhelm Standard TCP.

Although this work does not address deep architectural questions about a future Internet congestion control paradigm, it has the potential to establish two essential precursors: first at least minimal traffic segregation within the operational Internet such that the network can send different congestion control signals to different classes of traffic using different congestion control algorithms. Second, to establish an installed base using a $1/p$ congestion control algorithm. Due to its better scaling properties, it is much more likely than any future standard congestion control algorithm will be more similar to $1/p$ than to $1/\sqrt{p}$.

8. ACKNOWLEDGMENTS

This work is supported by a generous gift from Cisco System, Inc. I want to thank Fred Baker, Nandita Dukkkipati and others at Cisco have provided valuable input and inspiration on how to best frame the problem. John Heffner contributed sanity to an earlier version of my ideas.

9. REFERENCES

- [1] S. Floyd. Congestion control principles, September 2000. RFC 2914.
- [2] M. Mathis. Rethinking TCP friendly, March 2009. <http://www.ietf.org/internet-drafts/draft-mathis-icrg-unfriendly-00.txt> (work in progress).
- [3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 1–12, 1989.
- [4] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker. Approximate fairness through differential dropping. *SIGCOMM Comput. Commun. Rev.*, 33(2), 2003.
- [5] Frank P. Kelly, Aman K. Maulloo, and David K. H. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49(3), 1998.
- [6] Bob Briscoe. A fairer, faster internet protocol. *IEEE Spectrum*, pages 38–43, December 2008.
- [7] V. Jacobson. Congestion avoidance and control. *SIGCOMM'88*, August 1988.
- [8] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options, October 1996. RFC 2018.
- [9] M. Mathis. Relentless congestion control, March 2009. Web site <http://staff.psc.edu/mathis/relentless/>.
- [10] R. Bless, K. Nichols, and K. Wehrle. A lower effort per-domain behavior (PDB) for differentiated services, December 2003. RFC 3662.
- [11] S. Floyd and V. Jacobson. Random early detection (RED) gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4), August 1993.
- [12] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, 1997.
- [13] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Trans. Networking.*, 14(6):1246–1259, 2006.
- [14] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A compound TCP approach for high-speed and long distance networks. (MSR-TR-2005-86), 2005.