Enhancing TCP Performance for Dedicated Clusters and Grids

Eric Weigle and Wu-chun Feng

Research and Development in Advanced Network Technology
Computer and Computational Sciences Division
Los Alamos National Laboratory
Los Alamos, NM 87545

{ehw,feng}@lanl.gov

Abstract

TCP congestion control methods seriously and unnecessarily harm performance of network transmissions when used in dedicated clusters and grids. We present a simple method in which congestion control can be disabled under appropriate circumstances while still addressing fairness issues and avoiding congestion collapse. We discuss a Linux-based implementation of this "Rude TCP"[1] *and demonstrate the performance benefits this change provides over one and ten gigabit Ethernet links in Linux – showing a factor of 20 improvement in some cases.*

Keywords: TCP, congestion control, flow control, high-performance networking, wide-area network, Ethernet.

1. Introduction

The Transmission Control Protocol (TCP) [2, 4, 9] contains provisions for congestion control; that is, it perceives certain events as evidence that the network is overloaded, and reduces its send rate to compensate. This method was devised after the well-publicized congestion collapse in the early 1980s [23].

Unfortunately, in most cluster environments and some grid environments, the methods used to detect and respond to congestion are not necessarily appropriate. This paper describes a method to disable congestion control in limited circumstances; while retaining fairness and avoiding the possibility of congestion collapse it improves performance by significant amounts. A real-world Linux implementation (to be released under the GPL) is used to provide experimental results.

Our tests were conducted on high-end cluster hardware with one and ten gigabit Ethernet technology. Though the focus of this paper is to demonstrate the relative performance increase of Rude TCP (a modified TCP Reno) over stock TCP Reno, the absolute performance results using this hardware will be of interest to the community as well. In addition to Ethernet being a ubiquitous legacy technology that allows new 10-Gigabit Ethernet (10GigE) hardware to be "dropped in" without any substantial infrastructure changes, the performance of 10GigE compares well to Quadrics [25], Myrinet [10], and InfiniBand [20].

In Section 2 we discuss TCP congestion control and its effects in more detail to motivate the rest of this paper. In Section 3 we discuss the Linux implementation and use of Rude TCP. We then discuss our experiments and results in Sections 4 and 5. Lastly we discuss related work in Section 6 before concluding in Section 7.

2. Motivation

In today's high-performance computing & networking, clusters and grids are everywhere. To minimize complexity, many of these use TCP/IP to communicate. This is particularly true when communicating between nodes over wide-area networks (WANs) as when doing remote visualization or sharing large data sets between collaborators.

These network links are often characterized by being dedicated (unshared) and by having some quality of service guarantees. The network transmissions are typically long term, bulk data transfers. In this case, TCP congestion control is not required and will hurt performance by overreacting to packet loss. In the following circumstances, it should be obvious that congestion control is not required:

- when on a dedicated link
- when the "fair share" of the link is known and we transmit at that rate
- when an upper or lower layer protocol provides quality of service (such as ATM) or congestion controlled (such as RealNetworks [8]) transmissions

[1] We originally called this "Evil TCP" but chose to rename it after the April 1st definition [3] of the Evil Bit in the IP header.

We propose giving users the ability to turn off TCP congestion control in constrained circumstances. If done properly (in the circumstances listed here), it will not lead to congestion collapse and will be perfectly fair to the network. More importantly, we discuss a few simple and efficient ways in which malicious users of this functionality (those trying to turn off congestion control inappropriately in shared networks) can be kept from harming the network.

2.1. Benefits of TCP/IP and Ethernet

Ethernet is the most widely used networking technology in the world due to its low cost, simplicity, ease of use, and forward/backward compatibility. Similarly, as the price of new 10GigE hardware falls and adoption grows, it may be used instead of proprietary interconnects when very low latency (below 20 microseconds) is not required. The development of remote direct memory access (RDMA) over Ethernet and TCP/IP [31] will further decrease user experienced latencies and ease programming challenges.

One of the main benefits of using TCP/IP with Ethernet is the standard sockets interface. It is easy to program with, and virtually all applications rely on it. With a high-performance TCP (such as Rude TCP), applications will not need to be rewritten to use proprietary network APIs as found with other OS-bypass capable protocols such as Quadrics/Elan, Myrinet/FM, Myrinet/GM, InfiniBand, and so forth. Furthermore, while installation and configuration of Ethernet is trivial (and rude TCP/IP is readily available, see Appendix A), the aforementioned OS-bypass protocols and hardware are more difficult to work with.

2.2. TCP With Congestion Control

The most prevalent TCP congestion control method, that found in "TCP Reno," simply maintains a flow window (`fwnd`, the amount of buffering on the local host) and a congestion window (`cwnd`, the maximum amount of data out in the network). Successfully acknowledged transmissions increase `cwnd` to probe the network. Losses (or other congestion events such as explicit congestion notification) decrease `cwnd` under the assumption the network is congested. Reno simply maintains the invariant:

```
packets_out <= min(cwnd, fwnd)
```

to avoid overloading the network or overrunning the receiver.

The congestion window starts at 1 packet and ideally doubles each round-trip time (RTT) during the start of a connection ("slow-start" phase). After the first loss occurs, the congestion window is cut in half and Reno enters "congestion avoidance" phase. Here, Reno increases the congestion window by one packet per RTT until another loss occurs. Then Reno halves the congestion window and continues in congestion avoidance. This additive increase, multiplicative decrease (AIMD) behavior has provable mathematical properties in terms of long-term fairness among flows and has served well in communication networks for the past twenty years.

Unfortunately, this mechanism suffers from several well-known problems including:

- Unfairness between flows with differing round-trip times and window sizes. [13]
- Multifractal behavior in terms of buffer lengths, rates, etc. on the network. [28]
- Sudden, dramatic network rate changes that are problematic for visualization tools, or any real-time audio or video transmission. [19]
- Incompatibility with other congestion control mechanisms. [22]
- Congestion inferred from loss that is not always appropriate, such as with wireless networks. [1]
- Difficulty in tuning connection parameters. [26]
- Inappropriateness on high-delay networks (if you had a loss 100ms ago, should you still cut your window?) [4]

These problems have led researchers to constantly reinvent the wheel with reliable UDP protocols (such as in [7, 8, 30]). It has also proven to be a great way to keep network researchers busy inventing a plethora of other congestion control methods (see the Related Work, Section 6) or jury-rigged patches to avoid some subset of the problems above. Many have been proposed and implemented, but few have been put into widespread use. We assert that a more rational approach is simply to turn off congestion control when one of the extenuating circumstances described above applies. Many scientists have been clamoring for a simple way to "just turn it off"– this work provides an appropriate method for doing so.

We must note at this point that the current Internet trusts the end hosts implicitly. Any user could already be using the mechanism from Rude TCP (turning off congestion control) without changing the bits in the header. The only way to detect this is via work in "smart routers" (see Section 6).

2.3. TCP Without Congestion Control

If we were to disable congestion control in TCP, the sender would be able to transmit as much data as their flow control window could hold. Assuming the application always has data to send (not "application limited") this means that at any time during the connection, they would be only flow-window limited.

Therefore, if "bandwidth" is the bandwidth of the bottleneck link, the fraction of the link utilized is given by:

$$\frac{flow\ window}{bandwidth \times delay}.$$

And the overall bandwidth a flow experiences would be simply the bandwidth times this fraction, which reduces to:

$$\frac{flow\ window}{delay}.$$

With the application of work in TCP buffer autotuning such as with DRS [12, 15, 34, 35], ENABLE [32], Linux Autotuning [33], and other autotuning approaches [24, 29] the flow window can fluctuate based on the delay; allowing maximal performance on all networks.

What Rude TCP does is recover the bandwidth wasted by congestion control in circumstances where it is not required, as in dedicated clusters. Graphically, this wasted bandwidth can be seen as the gray area in Figure 1. This figure shows the bandwidth wasted for a typical TCP Reno flow over a network with a large $bandwidth \times delay$ product; for lower bandwidth or delay the figure would be compressed horizontally.

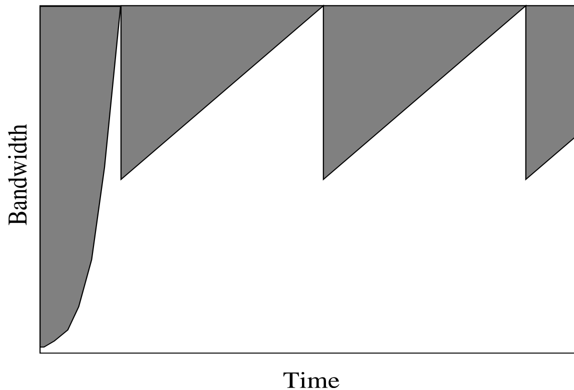

Figure 1. Bandwidth Reclaimed by Rude TCP

3. Implementation

In Linux [33] and other "real-world" operating systems the TCP stack is part of the kernel, so we must make some kernel-level modifications. However, since all we are doing is adding code to control the congestion control mechanism, the changes required are minimal.

3.1. Software Details

First we add a `sysctl` call (also exported via the `proc` file `/proc/sys/net/ipv4/tcp_rude`) that enables or disables rude flows on a system-wide basis. This allows the system administrator to decide whether or not users may create rude flows. Rude flows are by default disabled, so unless the administrator explicitly decides to enable them Rude TCP has no effect. Sample commands and user-level code for this section can be found in Appendix A.

Then, we add a TCP-level socket option called `TCP_RUDE`. This enables or disables congestion control on a flow-by-flow basis if it is allowed on the system. It also allows a user to check whether a socket is currently being polite or rude.

Finally, after a flow is set to be rude, the kernel marks all packets as rude using one of the reserved flag bits in the TCP packet header. Figure 2 shows the TCP header for reference (vertical gray lines indicate one bit; each row is one 32-bit word).

Source Port		Destination Port	
Sequence Number			
Acknowledgement Number			
HLen	Flags	Window	
Checksum		Urgent Pointer	
Options			Padding

Figure 2. TCP Headers

Marking packets from rude flows is required to avoid congestion collapse in today's Internet– routers will have the ability to kill Rude flows by dropping *all* packets or sending a `RESET` packet, giving inappropriate users no reward. Furthermore, as evidenced by the roll-out of ECN over the past few years, many routers already drop packets if their "reserved" bits are not zero. Thus rude flows will have problems traversing shared public links even if those links do not know about rude TCP.

Eight bits of flags have already been defined, with four bits remaining. We use the lowest of these to indicate a Rude TCP flow and we refer to it as the `rude` bit. Figure 3 shows the flags field in the TCP header, with the rude bit highlighted. For completeness, the other flags stand for congestion window reduced, explicit congestion echo, urgent, acknowledgement, push, reset, synchronize, and finish [9, 13]. Further details are outside the scope of this paper.

Reserved	RUD	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN

Figure 3. TCP Header Flags

	Additive Increase		Multiplicative Decrease	
Windows	$R_1 = R_0 + 1 \quad F_1 = F_0 + 1$		$R_1 = \frac{1}{2}R_0 \quad \frac{1}{2}F_1 = F_0$	
Absolute Difference	Constant	$(R_1 - F_1) = (R_0 - F_0)$	Decreases	$(R_1 - F_1) < (R_0 - F_0)$
	Proof:	$(R_1 - F_1) =$ $((R_0 + 1) - (F_0 + 1)) =$ $(R_0 - F_0)$.	Proof:	$(R_1 - F_1) =$ $((R_0/2) - (F_0/2)) =$ $(R_0 - F_0)/2$. $(R_0 - F_0)/2 < (R_0 - F_0)$
Relative Difference	Decreases	$(R_1 - F_1) < (R_0 - F_0)$.	Constant	$(R_1 - F_1)/F_1 = (R_0 - F_0)/F_0$.
	Proof:	$(R_1 - F_1)/F_1 =$ $((R_0 + 1) - (F_0 + 1))/(F_0 + 1) =$ $(R_0 - F_0)/(F_0 + 1)$. $(R_0 - F_0)/(F_0 + 1) < (R_0 - F_0)/F_0$	Proof:	$(R_1 - F_1)/F_1 =$ $((R_0/2) - (F_0/2))/(F_0/2) =$ $((R_0 - F_0)/2)/(F_0/2) =$ $(R_0 - F_0)/F_0$.

Table 1. Mathematical Convergence of Rude (R) and Fair (F) cwnd

3.2. Semantics of Turning Off The Rude Bit

Flows which have been rude may later wish to "become good network citizens" by turning congestion control back on (because, for example, a router may only filter rude flows during peak hours or because our process is migrating to another node). This requires that the accounting for the flow's packets in the network be brought in line with its current congestion window. Our Linux implementation decouples the congestion window calculations from the actual sending code, so Rude TCP knows the fair congestion window even if it is not used.

There are three cases to deal with, depending on the number of packets actually in the network, the current actual `cwnd`, and the urgency of the flow to stop being rude.

In the first case, the flow has no more packets in the network than it should. Here we simply stop marking packets "rude" and continue in the normal congestion avoidance phase.

In the second case, the flow has more packets in the network than it should and wishes to immediately stop marking packets rude. Here we stop marking packets "rude" and start using the appropriate `cwnd` as though a serious congestion event has just occurred.

In the third case, the flow has more packets in the network than it should but does not want to immediately cut its sending rate. Here we enter the congestion avoidance phase using an artificial `cwnd` based on the number of packets currently in the network and continue marking packets rude. We perform all operations affecting the congestion window on both the artificial and actual `cwnd` variables. These two disparate values will eventually converge to a single value reflecting the fair share of bandwidth the flow should have in the network. When this occurs we stop marking packets "rude" and continue using the single correct `cwnd`.

Mathematically, you have two congestion windows, R (for *Rude*, the artificial window) and F (for *Fair*, the actual window), where $R > F$. Each change happens to both R and F simultaneously. We consider the two primary Reno events of additive increase and multiplicative decrease, and consider the absolute difference $(R - F)$ and proportional difference $(R - F)/F$ between the two windows.

Table 1 shows mathematically how the congestion windows change. For each additive increase, the relative difference between the rude and fair windows falls. Over time, it becomes insignificant compared to the sizes of the windows themselves. Similarly, for each multiplicative decrease that occurs, the absolute difference in the windows is halved. Together, these drive both the windows to the same value.

Figure 4 gives an example of this behavior. It represents a 1Gb link with 100ms RTT and 1500 byte MTU, giving an ideal `cwnd` of 8333 packets. Initially, Rude TCP utilizes 90% of the link while two other flows (not shown) share the remaining 10%. At time zero, Rude TCP decides to be fair and begins maintaining a artificial congestion window. It then proceeds through several additive-increase, multiplicative decrease cycles until the rude and fair windows converge to within 1% (an arbitrary value) at which point the flow is no longer considered Rude.

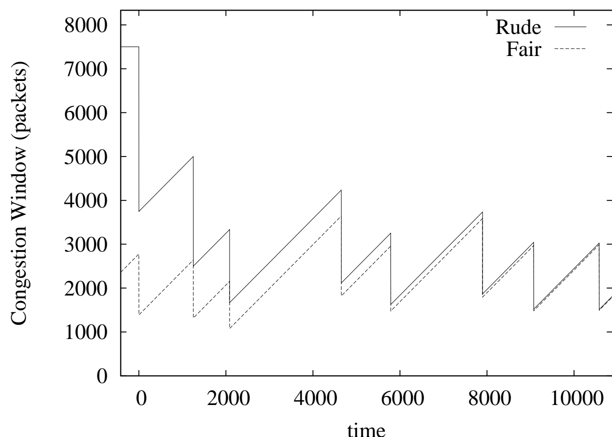

Figure 4. Congestion Window Convergence

4. Experiments

As a minor modification to the real-world Linux 2.4 network stack, Rude TCP is more than just the original congestion control free TCP of two decades ago. It includes all the tweaks and changes required to move from the Internet Engineering Task Force's RFCs [2, 9] into the real world. Our tests exhibit behaviors one could expect to see in live networks.

Our tests were performed on three high-end, Intel-based machines. Each has two 3.06GHz Intel Xeon processors with Hyperthreading enabled, 2GB memory, a 120GB serial ATA hard drive, two PCI-X slots running at 100MHz, and two on-board copper Ethernet ports at 100Mbps and 1Gbps speeds. We used the on-board Gigabit Ethernet and a 10-Gigabit Ethernet card in one PCI-X slot for the following tests. Two of these machines were used as a traffic source and a traffic sink, respectively, while the third was used as a wide-area network emulator (WAN) in some tests. We believe this hardware is representative of the type that will be used for clustering in the near future.

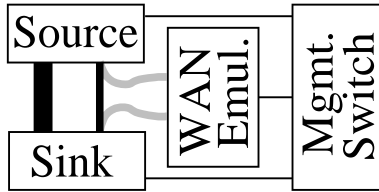

Figure 5. Network Topology

Figure 5 shows the connections between these machines. The thin black lines represent the 100Mb (Intel e100) copper Ethernet connections used for management of the machines. The thick black line represents the 10Gb (Intel ixgb) fiber Ethernet connection between the two main testing machines. The intermediate black line represents the 1Gb (Intel e1000) copper Ethernet connection between the two main testing machines. The two gray lines represent an alternate wiring of the 1Gb Ethernet connections through the third machine for WAN emulation; a second 1Gb Ethernet card was added to that machine for this purpose. The routing mode of the kernel-level tool TICKET [36] performs this emulation.

Tests were run using the benchmarking programs `nttcp` and `iperf`. Management was accomplished with `ssh`. We consider the following parameters:

- TCP congestion control mechanism: Normal TCP Reno versus Rude TCP Reno
- Transfer sizes: 1MB, 4MB, 32MB, 64MB, 128MB, and 256MB
- Network speed: 1Gb or 10Gb
- Loss rate: 5%, 2%, 1%, 0.1%, or 0.01%
- Delay: back-back (sub-millisecond) or emulated WAN (100ms)

The delay varies depending on the interface (see Table 2), and the WAN emulator can induce an additional 100ms delay (one thousand times greater than the back-to-back tests). Unfortunately we are only able to test the 1Gb (or slower) Ethernet in this WAN configuration as the emulator does not currently support the newly released 10GigE cards, and Los Alamos National Laboratory does not have sufficient connectivity to test in a "live" network.

Interface	Min.	Avg.	Max.
100Mb Copper	87	88	196
1Gb Copper	66	129	360
10Gb Fiber	28	30	100

Table 2. Interface Ping times (microseconds)

We perform 10 experiments with each set of parameters and present the arithmetic mean of the results. We wait at least five seconds between each test and flush the route cache of slow-start and congestion windows that Linux maintains each time; this is necessary for each experiment to be independent without actually rebooting the machine between tests.

These experiments had only minimal tuning for performance, so the results presented are the results one could expect from this hardware under normal circumstances (i.e., "out of the box") rather than the best-case or optimal circumstances. The only tuning that was performed was to set send and receive buffer sizes; the kernel was set to an 8MB default and 32MB maximum buffer size, nttcp allocated a 16MB (large) send buffer, and iperf allocated a 64KB (default) send buffer. These sizes were used in all tests.

Note that the ideal buffer sizes are based on the actual $bandwidth \times delay$ product of a given link. Using the data given in Table 2 we can calculate the ideal sizes, if we were to statically tune buffers.

Interface	Min.	Avg.	Max.
100Mb Copper	1.06 KB	1.07 KB	2.39 KB
1Gb Copper	8.06 KB	15.7 KB	43.9 KB
1Gb Copper/WAN	12.2 MB	12.2 MB	12.2 MB
10Gb Fiber	34.2 KB	36.6 KB	122 KB

Table 3. Ideal Static Buffer Sizes

Allowing iperf to use the large buffers in all cases, we never need to worry about being flow window limited. On the other hand, with nttcp using small buffers it does not permit large traffic bursts or reduction in the send window by more than 32KB at a time. We can think of the iperf results as the "dumb" approach; just allocate a lot of memory and let things run. The nttcp results are arguably the "smart" approach; if you know the bandwidth delay product tune your buffers appropriately. (See the discussion of buffer tuning in Section 2.3).

5. Results and Analysis

First we discuss the back-to-back tests over one- and ten-gigabit links with negligible loss. Then we consider the effect of loss on these cases. Last we consider the one-gigabit WAN link under various loss conditions.

5.1. One Gigabit, Local Configuration

Figures 6 and 7 give the iperf and nttcp performance for stock flows. The nttcp graph is more horizontal with respect to transfer sizes than the iperf graph, but qualitatively the results are generally the same. The difference in results between the two programs (in particular, the stability in the nttcp results) is due mostly to the small buffer nttcp is using, which we discussed at the end of Section 4. Nttcp is also dramatically simpler (single process, no call-back timers, and so forth; this simplifies the issue), and performs timing slightly differently.

Figures 8 and 9 give the iperf and nttcp performance for Rude TCP flows. Rude TCP is not as sensitive to loss. When packets are dropped, they are simply retransmitted (we use a kernel patch on the sender that drops the packet immediately before it is transmitted to induce these losses).

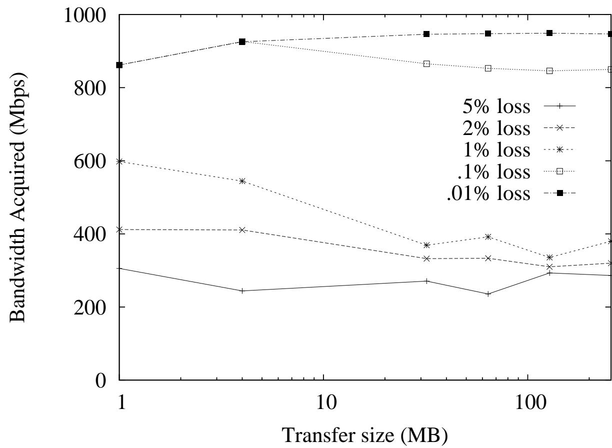

Figure 6. Iperf 1GigE Back-to-Back Reno Tests

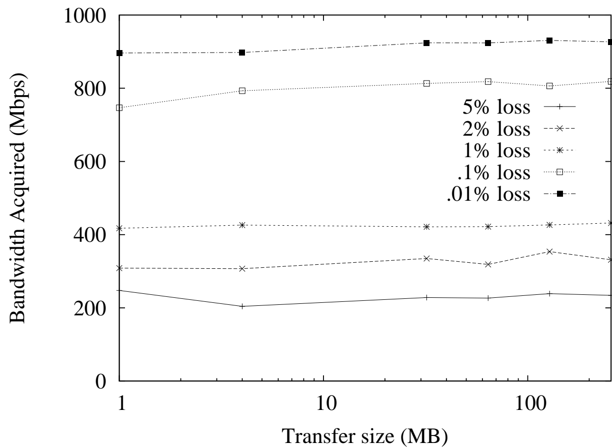

Figure 7. Nttcp 1GigE Back-to-Back Reno Tests

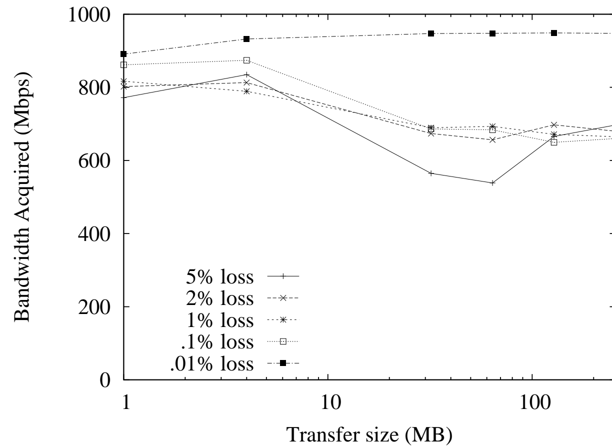

Figure 8. Iperf 1GigE Back-to-Back Rude Tests

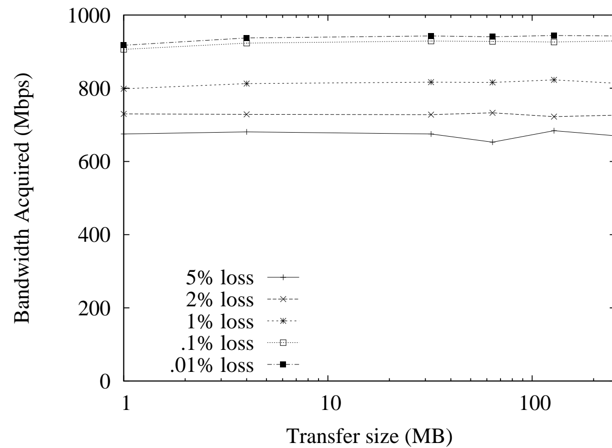

Figure 9. Nttcp 1GigE Back-to-Back Rude Tests

Stock Reno attempts to be “a good network citizen” in graphs 6 and 7, even though there are no other flows to be good citizens toward; as the loss rate increases it cuts back its sending rate. Performance falls from 95% utilization of the link to under 30% utilization, under the assumption that losses are caused by congestion rather than other possibilities such as a wireless connection, flaky media or malfunctioning router. We have seen the latter two due to overheating when a cooling fan fails, for example.

For the stock iperf graph (Fig. 6), the 0.01% and 0.1% loss cases are identical for short transfers, as it is likely that they will “complete” (that is, send all data to the kernel buffers) before a loss is induced. The 0.01% loss case is effectively zero loss when selective acknowledgements are enabled and the RTT is so low.

With Rude TCP, iperf and large windows (Fig. 8) there is enough buffer space that that we can avoid bubbles in the transmission stream when losses occur. The more data we send, the less the loss ratio matters– for 256MB transfers a loss of one in twenty packets is about the same as a loss of one in a thousand. When the loss rate is only 0.01% (effectively zero) Rude TCP and Reno TCP perform almost identically. There is a slight performance improvement in Rude TCP due to avoiding the slow-start phase, but this would only be significant if a user sends very small messages (under 4MB), each over a separate TCP connection. The drop in bandwidth acquired with the increase in transfer size is counterintuitive, but is due to several factors.

First is the increased probability that the benchmarking program will not be running on the processors on both source and sink for the full duration of the transfer. The Linux scheduler runs every 10ms, so for transfers above 1MB the benchmarking program will be forced to wait; for the largest transfers packets may be dropped while the benchmarking process is waiting to be rescheduled.

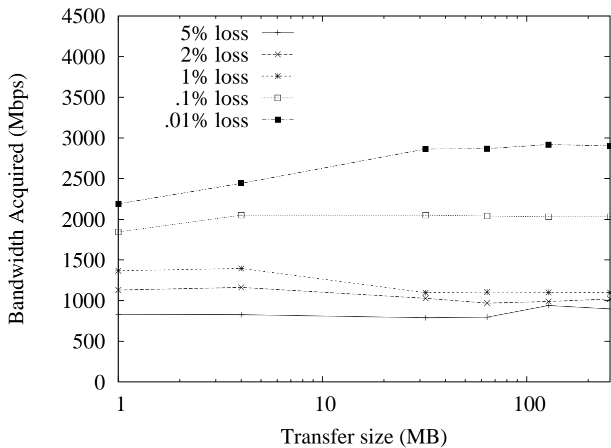

Figure 10. Iperf 10GigE Back-to-Back Reno Tests (1500B MTU)

Second is inefficiencies in the kernel’s memory management, the larger the transfer the more memory must be allocated and handled in the kernel.

Third is that larger transfers may trigger heuristics in the cards that slow them down; that is, when they see a sustained burst of packets for a given period they are more likely to transmit link-layer “source quench” flow-control packets which force the sender to slow down. Were we to test very large transfers (multi-gigabyte) the effect of these issues would be amortized and performance would again improve.

With Rude TCP we see a 65-70% utilization of the link under the most extreme loss (Fig. 8 and 9), as compared to stock TCP which attains about half the performance with only 30% utilization (Fig. 6 and 7). The only overhead of all these losses are the resources wasted preparing and transmitting the packet that did not reach its destination. This means a slightly higher CPU utilization (proportional to the number of losses, at most 5% in our tests), but packet losses here do not affect resources in the network because drops are induced by kernel code on sender.

5.2. Ten Gigabit, Local Configuration

Figures 10 and 11 give the iperf performance for stock and Rude TCP over the ten gigabit Ethernet connection and various loss parameters. The maximum value on the Y-axis has been increased from 1Gbps to 4.5Gbps.

At a high level, we see the same pattern as we hope to see in all of our graphs (and that we did indeed see in Section 5.1): performance increases with the length of the transfer and decreases with the increased probability of loss. Ideally longer transfers mean:

1. A better chance to reach the maximum, a steady streaming state

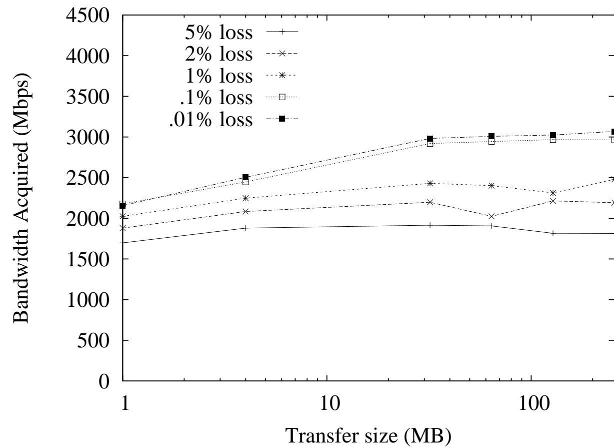

Figure 11. Iperf 10GigE Back-to-Back Rude Tests (1500B MTU)

2. Better acknowledgement clocking
3. Lower overhead per packet
4. Time to recover from minor losses

Streaming state is when our flow and congestion window are fully opened and we are limited only by the hardware; that is, TCP Reno has finished limiting itself with slow start and data flows over the network as fast as possible. This goes hand-in-hand with good acknowledgement clocking, which is when acknowledgements come evenly spaced with our sending rate. This allows more data to be sent out in well-spaced intervals, which improves performance [16].

The total overhead per data packet when averaged over the lifetime of a connection depends on connection start-up and shut-down time, the time to generate packet headers and checksum, as well as the various copies to/from user space, kernel space, and to the driver. Larger transfers mean that start-up and shut-down time is amortized over more packets. They also mean that more data may be copied at one time from user space to kernel space, amortizing copy overhead.

In Figure 10 we see that while performance increases with transfer size for the lowest loss cases (0.01% and 0.1%) it stays about the same or decreases with higher loss. This is because under heavy loss conditions the benefits 1, 2, and 3 of longer transfers do not apply– Reno is continually cutting its window (dropping out of streaming state), losing acknowledgement packets (worse acknowledgment clocking), and suffering higher overhead per packet (due to retransmissions).

The difference between Figures 10 and 11 is just as pronounced as with the earlier 1-gigabit Ethernet results. While stock TCP's bandwidth falls by over two-thirds (from almost 3Gbps to 900Mbps), bandwidth of Rude TCP only falls to about 1.8Gbps under loss– about twice the performance. Again, the low-loss curves are almost identical between stock TCP and Rude TCP.

Iperf results, even with Rude TCP, are much more sensitive to loss at 10GigE speeds than at 1GigE speeds. This is because to achieve good performance it is imperative that large buffers be available to DMA to the NIC and that everything streams cleanly. As soon as we encounter a bubble (a loss) we have to re-build and retransmit individual packets. This is very inefficient and our performance begins to be limited by hardware memory and bus bottlenecks.

When we increase the Maximum Transmission Unit (MTU) to just under 8K, the results (Figures 12 and 13) are quite different. The larger packet size fits more cleanly with the underlying hardware page size (8160=8192-32; this is two 4KB pages of memory on x86 hardware, less 32 bytes for accounting and headers). The larger size also means that there is less overhead per copy, as discussed earlier.

However, the most important factor by far is that a larger Ethernet MTU allows a larger TCP maximum segment size. This in turn allows TCP congestion control operations, performed in units of packets, to adapt much more quickly ((8160-20(IP)-20(TCP))/(1500-20(IP)-20(TCP)) = 5.6 times as fast). This means the initial slow start phase is completed more quickly, and when a loss occurs it takes just over one-sixth the time to recover from it,

Because of these factors, there is very little difference between the Regular and Rude TCP graphs in Figures 12 and 13. Rude TCP still performs better for small transfers because it avoids slow start, but otherwise performance is nearly identical. This includes the strange fall at 128MB transfers and high loss.

We attribute this odd but consistent performance slump to the three factors discussed at the end of Section 5.1. It appears that the high loss rate causes havoc with the memory management subsystem in the kernel and makes packet retransmissions very expensive. Other factors such as bad heuristics in the TCP stack may also be involved.

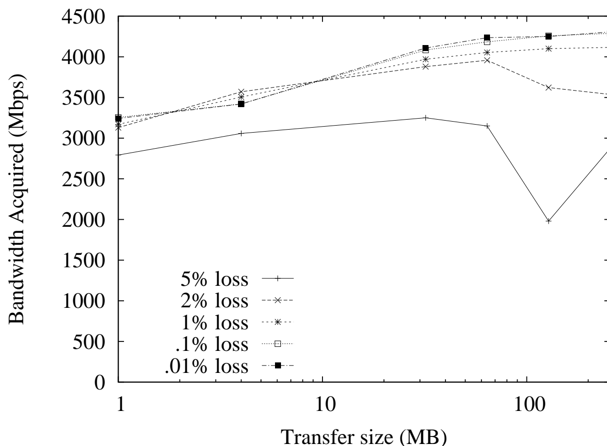

Figure 12. Iperf 10GigE Back-to-Back Reno Tests (8160B MTU)

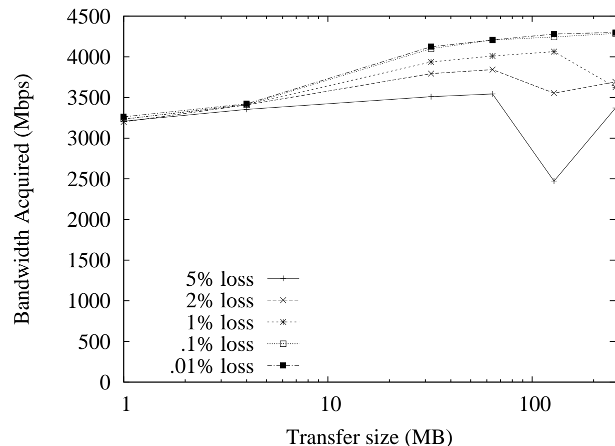

Figure 13. Iperf 10GigE Back-to-Back Rude Tests (8160B MTU)

5.3. One Gigabit, Wide-Area Configuration

This section returns to the 1GigE link, but increases the delay by 100ms. While the prior two sections were representative of message passing within a cluster, this section addresses dedicated grid environments– where the speed of light comes into play as we transfer information across a large country or an ocean.

Figure 14 gives the performance for stock TCP Reno over the emulated WAN link. Note that the y-axis has a maximum value of only 60 Mbps. For small sizes and the lowest drop probabilities, performance goes up to a meager 50Mbps. For higher drop probabilities, all the curves blend together with bandwidth on the order of a few hundred *Kilobits* per second.

The "no loss" line in Figure 14 means no programatically induced losses (which we control in the kernel). Losses may still occur due to link-layer bit errors or buffer overruns in the NIC's memory (which are beyond our control). This is likely what happened for the 32MB case; a congestion event occurred throwing the connection out of slow-start and forcing it into congestion avoidance phase.

The problem is the bandwidth-delay product of this network: about 12.5 MB of data. The "congestion avoidance" phase of TCP takes 4,280 round trip times: ($\frac{1}{2} \times 12,500,000$bytes$/1460$ (bytes/segment)). That's over seven minutes. Unfortunately, we inevitably suffer more losses before we get that far– thus the awful performance. In these kind of networks, it is imperative that loss be minimized or we take another approach (See Section 6).

Figure 15 shows the performance of our Rude TCP under the same conditions, 100ms RTT and varied loss rates. The maximum on the y-axis has been increased to 400Mbps. While nowhere near the link's capacity, we are at least seeing performance on the order of megabits instead of kilobits. In fact, in cases where stock TCP achieves only 500Kbps Rude TCP achieves over 25Mbps; a factor of 50 improvement. In all but the 128MB transfer case, we achieve at least an order of magnitude improvement over stock TCP.

The main reason that Rude TCP does not attain higher performance is because very long transfers are required to adapt to such a high delay. This is not due to a requirement for the congestion window to open, but for the heuristics in the stack (which we explicitly flush between flows for independent tests) to adapt. High-delay networks will always be hard to deal with– it is not always clear what action to take when your information is already a tenth of a second old.

6. Related Work

There are many approaches to deal with problems in high bandwidth-delay networks. Here we simply discuss avoiding the problem by turning off congestion control. Others attempt to solve the issues with congestion control in various ways.

One idea is using segment sizes that scale with the delay, in this case by a thousand times: 1.5MB packets. This is not feasible because the switching is too difficult and there are blocking issues at high speeds. Others propose alternate (but Reno-compatible) congestion control methods such TCP Vegas [5] or the Vegas-derived FAST TCP [6], Scalable TCP [18], or HighSpeed TCP [14].

We have mentioned that a mechanism is necessary for controlling malicious flows, as embodied by smart router work [11, 17]. In this case, routers act as police– if a flow misbehaves, it is killed. One easy way to do this is to select packets at random and model the congestion control method of their connection; if it doesn't conform drop all its packets.

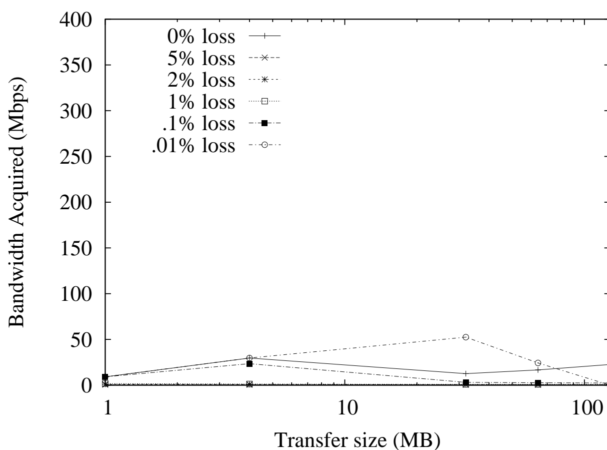

Figure 14. Iperf 1GigE WAN Reno Tests

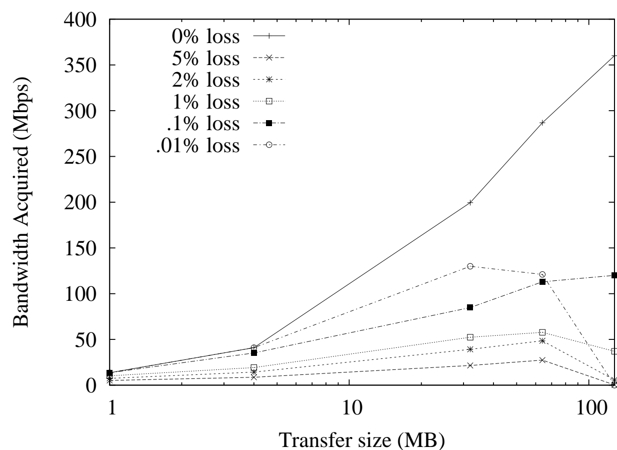

Figure 15. Iperf 1GigE WAN Rude Tests

One final benefit of a congestion control free TCP is that it is significantly easier to implement; one possible way to use this idea is to make an OS-bypass capable TCP that runs on a smart NIC's embedded processor or a memory-integrated network interface [21].

7. Conclusion

We shown that Rude TCP is more than just the congestion control free TCP of the 1980s, it is a real-world Linux implementation of TCP with all the improvements made in the past two decades. Our results exhibit the massive performance increases this can achieve in several circumstances relevant to dedicated cluster and grid computing, while still considering fairness issues for shared links. We have also discussed the benefits of TCP/IP, among them that application writers can achieve the performance of OS-bypass protocols without having to re-write their applications.

The network performance results on the high-end hardware used for these tests has been interesting in its own right. The results indicate that we are close to a complete, drop-in solution. All the pieces exist: inexpensive high-performance network hardware, automatic buffer tuning (as in DRS), improved congestion control, a plethora of performance tweaks (selective acknowledgements, fast retransmits, etc.) already in everyday stacks, and now a simple way to turn off congestion control. When these techniques become more widely adopted, truly high-performance TCP will be readily available for dedicated clusters and grids.

References

[1] M. Allman. Ongoing TCP Research Related to Satellites (RFC2760), February 2000.

[2] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control (RFC2581), April 1999.

[3] S. Bellovin. The Security Flag in the IPv4 Header (RFC3514), April 2003.

[4] D. Borman, R. Braden, and V. Jacobson. TCP extensions for high performance (RFC1323), May 1992.

[5] L. Brakmo and L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communication*, 13(8):1465–1480, October 1995.

[6] C. Jin et al. FAST kernel: Background theory and experimental results. In *Proceedings of the First International Workshop on Protocols for Fast Long-Distance Networks*, 2003. http://netlab.caltech.edu/pub/papers/pfldnet.pdf.

[7] Z. Chen, S. M. Tan, R. Campbell, and Y. Li. Real-time audio and video in the world wide web. *WWW Journal*, January 1996.

[8] J. Chung, M. Claypool, and Y. Zhu. Measurement of the congestion responsiveness of realplayer streaming video over udp. In *Proceedings of the Packet Video Workshop (PV)*, 2003.

[9] Defense Advanced Research Projects Agency. Transmission Control Protocol (RFC793), September 1981.

[10] N. B. et al. Myrinet: A gigabit-per-second local-area network. *IEEE Micro*, January-February 1995.

[11] W. Feng, A. Kapadia, and S. Thulasidasan. Green: Proactive queue management over a best-effort network. In *Proceedings of IEEE GLOBECOM*, November 2002.

[12] M. Fisk and W. Feng. Dynamic Adjustment of TCP Window Sizes. Technical Report Los Alamos Unclassified Report (LA-UR) 00-3221, Los Alamos National Laboratory, July 2000. See http://www.lanl.gov/radiant/website/pubs/hptcp/tcpwindow.pdf.

[13] S. Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communication Review*, 24(5):10–23, October 1994.

[14] S. Floyd. Highspeed tcp for large congestion windows, 2003.

[15] M. K. Gardner, W. Feng, and M. Fisk. Dynamic Right-Sizing in FTP (drsFTP): An Automatic Technique for Enhancing Grid Performance. In *Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing*, 2002. http://www.lanl.gov/radiant/pubs/drs/hpdc2002.ps.

[16] A. Kapadia, A. Feng, and W. Feng. The effects of inter-packet spacing on the delivery of multimedia content. In *Proceedings of 21st International Conference on Distributed Computing Systems*, 2001.

[17] A. Kapadia, W. Feng, and R. Campbell. Green: A practical solution for ensuring fairness in a best-effort network. In *Los Alamos National Laboratory Technical Report LA-UR-03-2372*, January 2003.

[18] T. Kelly. Scalable TCP: Improving performance in high-speed wide area networks.

[19] M. Li, M. Claypool, and R. Kinicki. Mediaplayer versus realplayer - a comparison of network turbulence. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, 2002.

[20] J. Liu, J. Wu, S. Kinis, D. Buntinas, W. Yu, B. Chandrasckaran, R. Noronha, P. Wyckoff, and D. K. Panda. Mpi over infiniband: Early experiences, January 2003. Ohio State University Technical Report OSU-CISRC-10/02-TR25.

[21] R. Minnich, D. Burns, and F. Hady. The memory-integrated network interface. *IEEE Micro*, February 1995.

[22] J. Mo, R. J. La, V. Anantharam, and J. Walrand. Analysis and Comparison of TCP Reno and Vegas. In *Proceedings of INFOCOM '99*, pages 1556–1563, March 1999.

[23] J. Nagle. Congestion Control in IP/TCP Internetworks (RFC896), January 1984.

[24] G. Navlakha and J. Ferguson. Automatic TCP Window Tuning and Applications. http://dast.nlanr.net/Projects/Autobuf/autotcp.html, April 2001.

[25] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The quadrics network: High-performance clustering technology. *IEEE Micro*, January-February 2002.

[26] Pittsburgh Supercomputing Center. Enabling High-Performance Data Transfers on Hosts. http://www.psc.edu/networking/perf_tune.html.

[27] RADIANT Team, CCS-1, Los Alamos National Laboratory. The RADIANT Team Website. http://www.lanl.gov/radiant/.

[28] R. Riedi and J. Levy-Vehel. Multifractal properties of tcp traffic: A numerical study, 1997.

[29] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. *ACM SIGCOMM 1998*, 28(4), October 1998.

[30] Sun Microsystems, Inc. NFS: Network File System Protocol Specification (RFC1094), March 1989.

[31] T. Talpey and S. Bailey. The architecture of direct data placement (ddp) and remote direct memory access (rdma) on internet protocols, March 2003.

[32] B. L. Tierney, D. Gunter, J. Lee, and M. Stoufer. Enabling Network-Aware Applications. In *Proceedings of IEEE International Symposium on High Performance Distrubted Computing*, August 2001.

[33] L. Torvalds and The Free Software Community. The Linux Kernel, September 1991. http://www.kernel.org/.

[34] E. Weigle and W. Feng. Dynamic Right-Sizing: A Simulation Study. In *Proceedings of IEEE International Conference on Computer Communications and Networks*, 2001. http://public.lanl.gov/ehw/papers/ICCCN-2001-DRS.ps.

[35] E. Weigle and W. Feng. A Comparison of TCP Autotuning Techniques for Distributed Computing. In *Proceedings of IEEE International Symposium on High-Performance Distributed Computing*, 2002. http://public.lanl.gov/ehw/papers/HPDC-2002-DRS.pdf.

[36] E. Weigle and W. Feng. TICKETing High-Speed Traffic with Commodity Hardware and Software. In *Proceedings of the Third Annual Passive and Active Measurement Workshop (PAM2002)*, March 2002.

A. Sample Use

This Appendix describes the commands and code that a user of rude TCP would use to enable and work with the Linux implementation.

The first step is to download and apply the kernel patch. In a web browser go to the Radiant website [27] and and look for the rude tcp patch for an appropriate kernel on the 'software' page. Go to the directory where you have the uncompressed kernel sources and apply the patch with:

```
patch -p1 < rude_tcp-2.4.20.patch
```

Then build and install the kernel as usual, and reboot into it.

Next, enable rude flows for this machine via the `/proc` filesystem:

```
echo 1 > /proc/sys/net/ipv4/tcp_rude
```

Any user may see if this is enabled via

```
cat /proc/sys/net/ipv4/tcp_rude
```

but only root may change its value (0=disabled, 1=enabled). Remember than *enabling* rude flows means allowing users to *disable* congestion control.

Finally, you must add a `setsockopt` call to your program with the file descriptor of the socket to make that specific flow rude.

```
int rude=1;
setsockopt(fd, IPPROTO_TCP, TCP_RUDE,
           &rude, sizeof(int));
```

You may also check if a given flow is rude with:

```
len = sizeof(int);
getsockopt(fd, IPPROTO_TCP, TCP_RUDE,
           &rude, &len);
```

At this point, you will have a reliable byte stream without congestion control that your application can use just like any other. If you want to turn congestion control back on later (turn off rude TCP), simply call `setsockopt` again with an appropriate parameter.

- `RUDE_ON` (=1): Turn on rude TCP
- `RUDE_OFF_NOW` (=0): Turn off rude TCP immediately, inducing a congestion event if necessary.
- `RUDE_OFF_NICELY` (=2): Turn off rude TCP as soon as the artificial and actual congestion windows converge.