

CSE 222 Final: Network Health Monitors

Federico David Sacerdoti

Daniel Marc Wittmer

March 20, 2001

1 Introduction

During the course of the quarter we have looked at several papers which analyzed instances of "bad behavior" in the current Internet. More specifically there have been several examples as well as solutions for identifying the behavior of TCP implementations but few that measure the congestion experienced by individual connections. We concentrate on client side metrics to make our results more applicable to web clients. We chose this strategy to help answer the often asked question: "Why is my Internet connection so slow?" Such questions can only be answered by examining the most prominent flow direction for a client: from the server to the browser.

We choose to measure TCP connections because they are used by both FTP and HTTP, which form a large portion of current Internet traffic. In addition, the TCP protocol intrinsically provides a rich amount of performance data that can be used to monitor performance [2]. We restrict ourselves to monitoring at the client node since getting access to popular web servers is usually not practical. Since most of the connections of interest involve much more downstream data than upstream, we concentrate on receiver side metrics such as the number of duplicate acks sent rather than figures only known by the sender such as the number of retransmitted packets. Although a typical client such as a web browser does perform some sender retransmits, they are usually insignificant since the upstream data consists of only small HTTP requests.

The rest of the paper is organized as follows. In section 2 we discuss the reasons for our choice of performance metrics, in section 3 we describe the Linux kernel modifications and define the metrics in more detail. Section 4 describes our experiment using the monitors, and section 5 discusses our results.

2 Motivation

We want to measure the speed of a client's TCP connections, and most of those connections are dominated by downstream traffic. Since we only have access to the client node and not the server, we must find performance data using only the information coming downstream to us. Savage showed that the TCP protocol maintains data that we can leverage to our advantage and we use that idea to monitor per-connection TCP health at the client.

This task is made difficult because of the TCP philosophy of "smart sender, dumb receiver". There is simply more information about the connection on the server side than on the client. To get around this we measure a connection from the receiver side using indirect means. We use duplicate ACK counters to measure lost and reordered packets, round trip time estimators calculated by the receiver to measure connection latency, and duplicate packets received counters to measure inefficiencies from not using the SACK TCP option. These

metrics are all available from careful instrumentation of the receiver's TCP stack and give a reasonable representation of the health and performance of a TCP connection.

3 Methodology

To monitor TCP performance we took several steps. First we instrumented the standard 2.2 Linux TCP stack to track select performance metrics. Then we extended the Linux *proc* filesystem to present these metrics to user space programs. Finally a set of perl scripts were run on the modified kernel to periodically do a set of common network tasks including web browsing and file transfers while we monitored their performance. Finally we created a monitor plugin for the popular GKrell Monitor to allow easy viewing of day-to-day network performance.

We have instrumented the Linux kernel to report the "health" of each established TCP connection, as defined by specific metrics we believe are indicative of real world performance. The modified Linux kernel keeps round trip time estimate, total acks sent, duplicate acks sent, total packets received, and duplicate packets received for each established TCP connection on the host machine. We chose these metrics because they give meaningful connection performance and can be measured on the receiver node as discussed in the previous sections. We later use these metrics to analyze congestion and search for bad behavior for connections to commonly used websites and file servers.

A duplicate ACK has the same sequence number as the one sent before it. The TCP protocol specifies that the receiver sends a duplicate ACK whenever it receives a packet whose sequence number is larger than the one expected. Therefore duplicate acks indicate that the network has lost or delivered packets out of order, and a new duplicate ACK will be sent by the receiver every time it sees a new out-of-order packet. Therefore we use duplicate ACKs to measure the quality of the downstream TCP connection, from the server to client. Since most common network tasks such as using email, web browsing, and file transfer are characterized by downstream data, the number of duplicate ACKs on a connection shows how *clean* it is, ie how often the receiver must announce out of order or lost packets to the sender.

The round trip time estimate tells how long it takes a packet of data to go from the sender to the receiver and back again; the TCP stack uses this figure to guess if a packet has been dropped by the network and needs to be retransmitted. The RTT is also a good measure of connection performance; a low RTT indicates low latency which leads to good response time for the user. A low RTT value is especially important for web browsing; a typical page contains many elements which must be individually retrieved using the request/response pattern. This leads to a serially executed set of element retrievals that are performance bound by the RTT.

The TCP stack calculates the RTT on both the sender and receiver nodes at the rate of once per packet in flight. To make this calculation the packet must originate at the node in question, so the sender estimates the RTT more often than the receiver due to the greater number of packets sent in that direction. However in the typical connections we examined the receiver sent enough packets to obtain a good RTT estimate.

The last metric we calculate relates to a perceived shortcoming of the standard TCP implementation. A TCP receiver can get duplicate packets from the sender because it cannot acknowledge packets that arrive out of order. These duplicates would happen when the sender mistakenly thinks some packets have been lost by the network because it does not receive acks for them but in reality they were successfully received out of order. Since the receiver has no way of letting the sender know about the receipt of these packets, they could potentially be re-sent and re-received at the receiver.

Not only do duplicate packets waste precious Internet bandwidth but they hurt performance because the sender mistakenly detects congestion from packet losses. The SACK TCP extension specifically addresses this

Table 1: Files downloaded and their sizes

File	Size
http://www.cnn.com/index.html	57 KB
http://www.nytimes.com/index.html	57 KB
http://www.webserver007.magnet.ch/index2.html	0.6 KB
http://www.slashdot.org/index.html	40 KB
http://wn-1.wired.com/index.html	44 KB
ftp://zeus.kernel.org/.../linux-2.0.34.tar.gz	6.8 MB

issue but is not yet deployed widely on the Internet. We would like to gauge the impact of this problem on the performance of baseline TCP implementations. A large number of duplicate packets received would indicate a significant benefit to the wide adoption of SACK. The *duplicatepacketsreceived* metric is computed at the receiver and counts these packets on a per-connection basis.

In order to present these metrics in a convenient way, we added a file to Linux's proc file system. The proc filesystem is a good fit for our needs since it allows user level programs to view kernel data structures with common file tools. We added the */proc/net/tcphealth* file to the Linux */proc* structure, which dynamically reports our chosen TCP metrics when the file is read. The file lists these separately for each TCP connection in the ESTABLISHED state, along with the source and destination socket of each.

4 Experimental Setup

The machine which was used to collect the experimental data was a IBM Pentium 233 laptop running Linux 2.2.14. Additionally the laptop was attached to the CSE ActiveWeb wireless LAN using an 802.11 compliant pcmcia network interface card.

In order to collect the data as well as generate traffic we wrote two Perl scripts, namely *collect_stats.pl* and *generate_traffic.pl*. The role of *collect_stats.pl* is to, on a per one second basis, poll the */proc/net/tcphealth* file and record the information therein. Due to the cumulative nature of the statistics given in the file, we simply used the final statistics for each connection and wrote these to a log file. More specifically for each run (as defined below) we recorded the local IP address, remote IP address, RTT estimate, number of ACKs sent, number of duplicate ACKs sent, number of packets received as well as the number of duplicate packets received for each TCP connection.

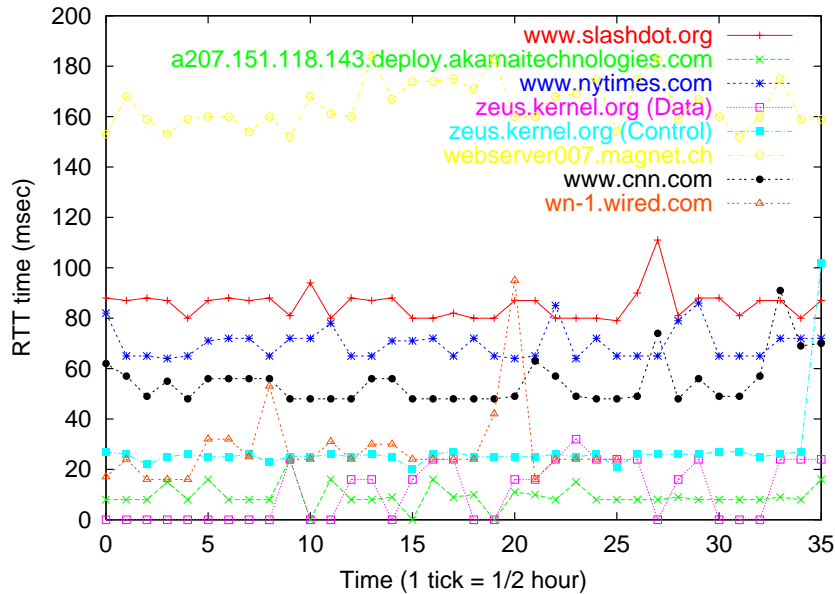
generate_traffic.pl is designed to connect to a set of given websites using wget [3] and either download the index.html page or file.

The script visited each of the above sites, and downloaded the associated files, in succession at a period of every half hour.¹

The sites visited represents only a small fraction of the Internet, and hence we do not expect our results to be very indicative of the Internet in general. However they do provide a basis for "local" observations as well as show the basis of our measurement infrastructure. Ideally we would like to perform a more extensive

¹Our logs show connections to a207.151.118.143.deploy.akamaitechnologies.com which were not included in our traffic generation script. These stem from visiting other websites during the monitoring process, but we include them due to interesting behaviors as noted in the Analysis section.

Figure 1: RTT time for each site over time



investigation of the overall health of TCP connections by connecting at differing geographic points in the network, using different mediums i.e. ethernet LAN, 56 K modems, ISDN, as well as extend the breath of sites visited and files downloaded. We do however include a site in Europe, namely www.webserver007.magnet.ch, in order to determine how traversing a trans-atlantic or satellite link affects the measured statistics.

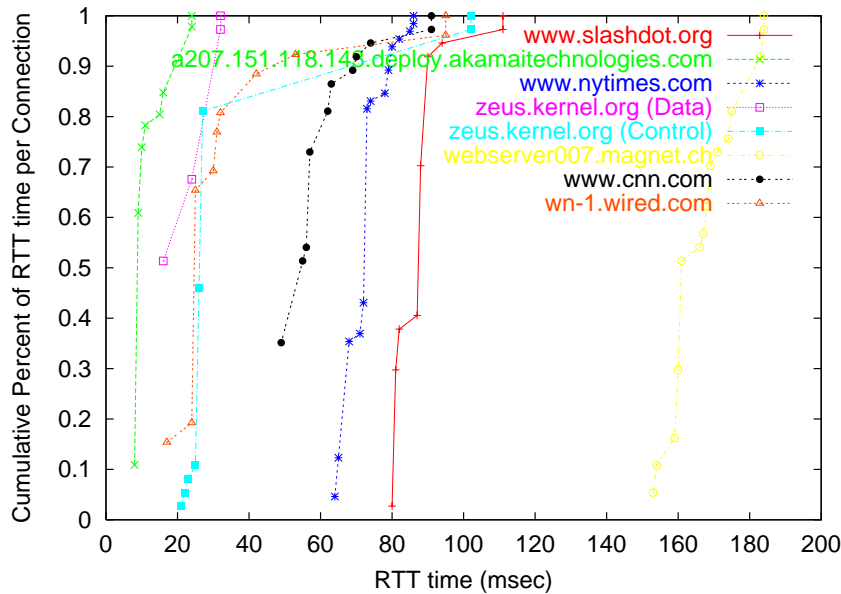
5 Analysis

Figure 1 shows the round trip time for each site visited over the time of the experiment. Of interest is the spike seen around 6:15 AM PST for the RTT time of wn-1.wired.com. This time corresponds to a range of 6:15 AM PST to 9:15 AM EST for the United States, which could be indicative of the daily morning spikes experienced due to users coming online. Furthermore, as expected, the RTT times for the most distant site, www.webserver007.magnet.ch, is nearly twice as large as those of the US based websites. Lastly of note is the fairly constant RTT time for the FTP control connection to zeus.kernel.org.

Figure 2 shows the cumulative distribution of the RTT times for each site. In general it is interesting to note that for all of the different sites, roughly 65% are fairly close to the minimum RTT for that respective site. This can clearly be seen based on Figure 1, in which most sites have a fairly constant RTT, marked with occasional spikes.

In order to measure the amount of out-of order packets and dropped packets, figure 3 shows the total number of duplicate ACKs sent for each connection during a run. As can be seen from the graph most sites suffer only from a small number of duplicate ACKs sent. But the data FTP connection to zeus.kernel.org, which is

Figure 2: CDF of the RTT times for each site



transferring 6.8 MB worth of data, clearly shows a large number thereof, peaking at nearly 180 duplicate ACKs sent in one run. The low number of duplicate ACKs sent for the other sites can be attributed to the fact that the file sizes downloaded from the respective sites is only a small fraction of the size for zeus.kernel.org (Data). Lastly we note an increase in the number of duplicate ACKs sent for the HTTP sites, stemming most likely from increased usage of the UCSD network since this occurs between the hours of 11:00 AM and 1:46 PM PST.

Of the information collected the ratio of the number of duplicate ACKs to the total number of ACKs holds the most surprises. Since we have already argued that duplicate ACKs are a measure of out-of-order or dropped packets, we would like to see the ratio as low as possible. However, as can be seen in figure 4, there are several sites for which we experienced a high percentage of duplicate ACKs. Most alarming is the fact that during one run the connection to a207.151.118.143.deploy.akamaitechnologies.com showed more than 60% duplicate ACKs as well as www.nytimes.com experience a similar spike in the number of duplicate ACKs sent. The time at which this spike occurs corresponds to 9:15 PM PST on Sunday March 18th, which one would generally not associate with a period of high user traffic. Thus this spike cannot be attributed to congestion, but may potentially stem from route changes etc. Also of interest is the observation that only a low percentage of ACKs were duplicates for the FTP data connection to zeus.kernel.org as well as consistently low, if not zero, percent for the connections to webserver007.magnet.ch.

In our limited experiment, the results indicated no duplicate packets were received on any connection in the 18 hour run. This leads us to several conclusions. Since duplicate ACKs were seen on many connections we know that some packets were lost or reordered, but unACKed reordered packets never caused a *coursegrainedtimeouts* on our connections. Only these timeouts will cause duplicate packets to be received since less severe out-of-order conditions will be resolved with fast retransmits. The lack of course timeouts may be due to the quality of UCSD's ActiveWeb network or the paucity of large gaps between received

Figure 3: Number of duplicate ACKS per site

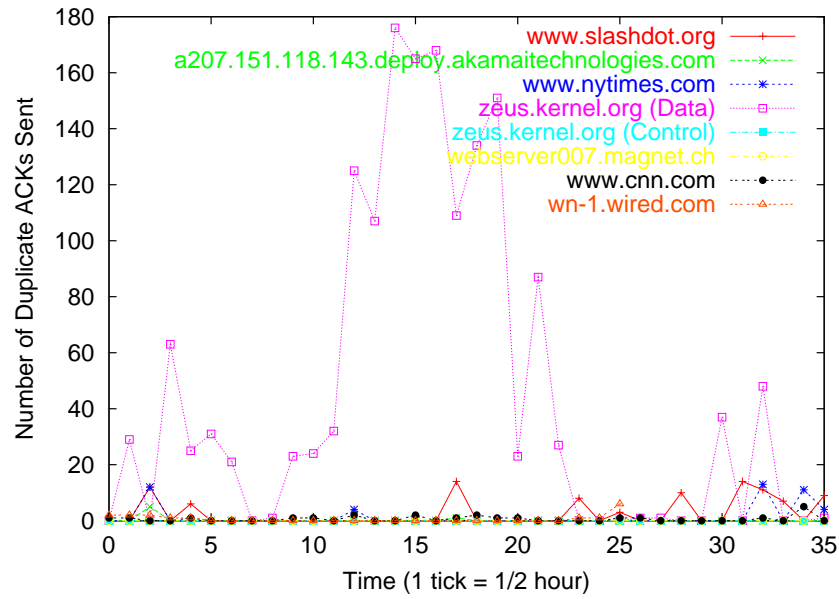
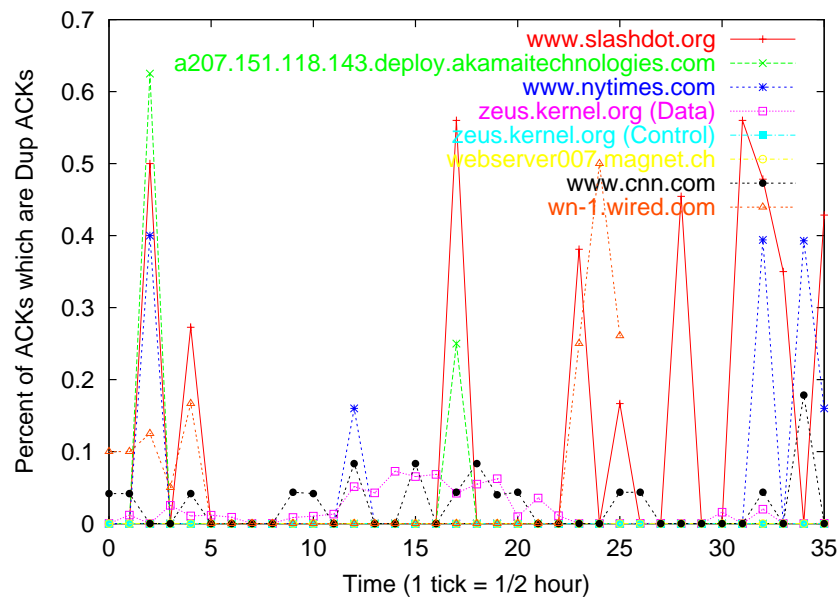


Figure 4: Percent of ACKs which are Duplicate ACKs



packet groups. It should be noted that Linux 2.2 implements fast retransmits for up to two packet gaps², thus reducing the need for coarse grained timeouts due to the lack of SACK.

The second conclusion we may make from the lack of duplicate received packets is that the SACK TCP feature may not be critical to everyday TCP performance. SACK makes duplicate packets impossible by allowing the receiver to acknowledge out-of-order packets and thereby letting the sender know they were not lost. If this case happened often in normal use we would expect to see more duplicate packets, which do not appear in our limited tests. It could be that SACK's advantage lies in other areas such as very large downloads or when using slow and unreliable network links.

6 Conclusion

Our search for the causes of network sluggishness led us to the TCP stack and using information contained in it we identified and explored some likely performance factors. Through the experiment we discovered some surprising things such as the high percentage of lost or reordered packets from supposedly highly reliable and fast services like the Akamai [1] network. We also found that the number of unnecessary duplicate packets were quite small potentially indicating that the SACK addition to TCP is not critical.

In the future we would like to extend our experiments to include more sites and run them from end nodes with a variety of connection speeds. With a longer running experiment a detailed time-traffic analysis would become possible and may lead to additional insights. Finally we plan to submit our Linux kernel modifications for inclusion in the official Linux source tree.

References

- [1] Akamai. <http://www.akamai.com/>.
- [2] Stefan Savage. Sting: a tcp-based network measurement tool. In *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems*, pages 71–79, October 1999.
- [3] GNU wget. <http://www.gnu.org/software/wget/wget.html>.

²The 2.2.14 TCP stack identifies this feature as "Hoe style secondary fast retransmits". (*tcp_input.c* : 533)