ACKRUE

CSC 564 Fall 2006

Jason Rickwald, Shawn Tice, Kyle Leveque
{jrickwald, stice, kleveque}@calpoly.edu

Abstract

TCP congestion control contains many implicit assumptions that can be exploited without actually violating the TCP protocol. We demonstrate that a receiver can easily exploit these assumptions and in most cases achieve much higher download rates than a well behaved receiver. Our results show that ACKRUE, an application acting as a "miss-behaved" receiver, can out preform the *wget* application in most cases.

1 Introduction

The Internet today requires a great deal of cooperation in order to operate correctly and allow fair access. In this paper, we demonstrate that, particularly for TCP, a client can choose to be selfish yet still operate within the boundaries of the protocol. This, in turn, can put other parties at a disadvantage, as they are operating under the assumption that everybody is acting in a similar and cooperative manner (as outlined by TCP congestion control)[5].

In this paper we describe ACKRUE, a proof of concept application that downloads files over HTTP while ignoring normal TCP congestion control. In section 2 we briefly describe the work that we base our application on. Section 3 describes, at a high level, how ACKRUE was built and what it does to "cheat" the server into sending data at a higher rate. We describe the implementation details in section 4 and describe some of the limitations and problems with the current version of ACKRUE in section 5. In section 6 we present our evaluation and testing of ACKRUE. In section 7 we outline some potential future work. Finally, section 8 concludes.

2 Background and Related Work

Prior work has pointed out that TCP congestion control can be detrimentally affected by a misbehaving receiver due to the behavior specified in RFC 2581 rather than implementation bugs [1]. The authors of the paper "TCP Congestion Control with a Misbehaving Receiver" identified three different ways that a greedy receiver might misbehave in order to trick a sender into sending data at a higher rate than if the receiver was behaving properly [1]. The three different "attacks" that the authors identify are ACK division, duplicate ACK spoofing, and optimistic ACKing. The ACK division attack exploits the fact that TCP sequence numbers are defined in bytes, but TCP congestion control is implicitly defined in terms of segments. The receiver simply sends more than one ACK per segment received, but each ACK acknowledges different (non-overlapping) byte ranges. This causes the sender's window to increase as each ACK is received. This attack simply takes advantage of a poor implementation decision. Hence, a more correct (or fixed) implementation of TCP would not be susceptible to this technique. The second attack, duplicate ACK spoofing, causes the sender to enter fast recovery. Careful use of duplicate ACKing of a received packet can allow the sender to artificially inflate the sender's window by tacking advantage of the window inflation rules for fast recovery (3*SMSS, plus one for each additional duplicate ack). The final attack, optimistic ACKing, compensates for the fact that TCP congestion performs poorly when sending data across high-latency links. By optimistically ACKing packets that have not been received yet the sender is able to make the link seem much shorter. Thus, the ACKs are received by the sender earlier and drive its window open more quickly [1]. Our project is most closely related to this third form of attack. However, instead of trying to guess which packets will arrive, our receiver simply conceals any lost packets. Just like the optimistic ACKing attack, the sender's window is artificially increased -- it never knows about lost packets, so it never has a reason to shrink its window. Finally, our project requires a higher level protocol (such as HTTP range requests) in order to preserve end-to-end reliability, again like optimistic ACKing. We will show that even our more simple technique can provide significantly increased download speeds in most cases when compared to a normally behaving receiver.

Our project code is based off of the Sting project [3][4], which developed a basic user-level TCP stack implementation in order to measure network anomalies like packet reordering. This allows us to manipulate TCP packets and behavior from user-space, without having to modify the kernel. The implementation is further discussed in section 4. The Daytona project was also explored as a potential code base for a user-level TCP stack [2]. The Daytona project actually implements a fully functional TCP stack at the user-level that mimics the behavior of a real Linux TCP stack. However, we found that the Daytona project contained much more functionality than what was necessary for our experiment in TCP congestion control. Additionally, Daytona was not very portable and would have taken a lot of work in order to get running on the systems available to us.

3 Technique

This section describes the technique that we use to "trick" a TCP sender into opening up its congestion window quickly and keeping it open for the duration of a transfer. The sender relies on the receiver to acknowledge each received packet with an ACK for the next packet that the receiver is expecting. If a packet is dropped due to congestion, the sender wont receive an ACK for that packet, and will take one of two actions. If the sender's congestion window is still open, it will simply send the next packet; otherwise, the sender will wait for an outstanding ACK to arrive, or it will time out. In the former case, the sender will continue to send packets out until its window closes, or it receives an ACK requesting a packet that it has already sent, at which point it recognizes a loss. In the latter case, the sender is unable to do anything until it receives an ACK for another packet, at which time it will recognize a loss, or it times out, at which point it will return to slow start, reducing the window down to one.

The key observation is that the sender relies on the receiver to report congestion via an ACK for a previously-sent packet. The only other way the sender recognizes congestion is through a complete absence of communication from the receiver, or (in TCP Reno) reception of three duplicate ACKs. Thus, if the receiver always ACKs the latest packet received, and no major loss occurs such that the sender times out waiting for ACKs or receives three duplicate ACKs, the sender will believe that there is no congestion. Consequently, the sender will continue to send out packets as quickly as it can, widening its congestion window until it hits the maximum advertised size. In order to maximize the amount of data in flight under this system, we advertise both our window size and our maximum segment size (MSS) at their maximum values, 65535 and 1460 respectively. The window size tells the sender the maximum number of packets it should put on the line without receiving an ACK, and the MSS communicates the maximum number of bytes per TCP packet. Notice that both values are maximums, and that, while the sender will never use values greater than these, it may use lower ones.

4 Implementation

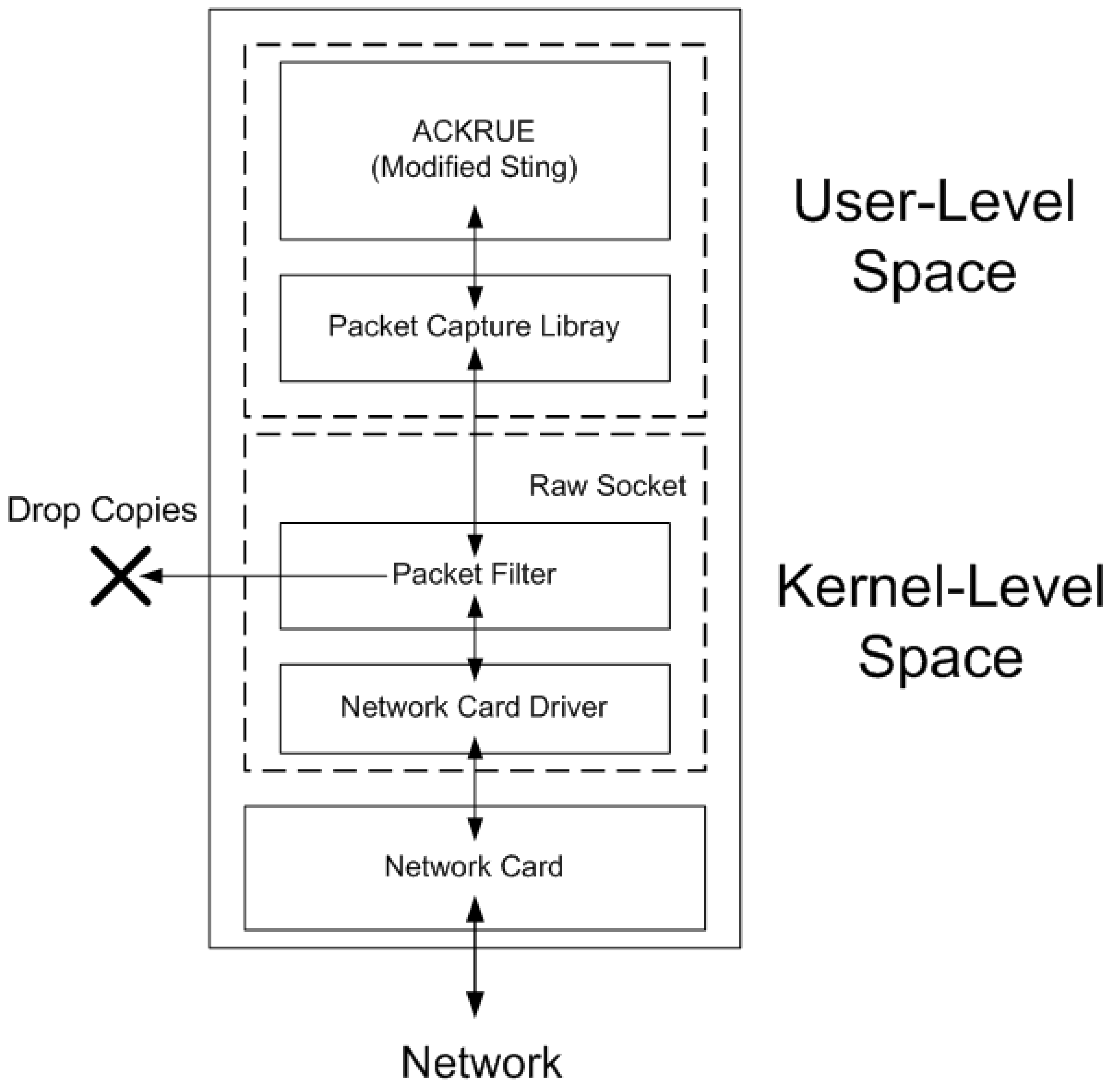

Figure 1: Overview of the ACKRUE system

The technique described above produces a TCP implementation that is no longer completely reliable on the receiver's end. When packets are lost, the receiver pretends that it has actually received them, and so creates a "hole" in its data. In order to go back and fill the hole without wasting work, we need to be able selectively request byte ranges from the sender. Fortunately, HTTP 1.1 provides such functionality via a range request. Range requests are intended to be used to resume downloads, and are just like normal HTTP requests, but with the addition of a header that instructs the server to send only a specific byte range in its response. Our application only works with HTTP 1.1 servers that support range requests.

Because our technique requires changing the behavior when a packet is received, we must either alter the kernel's TCP implementation, or use a raw socket to bypass the kernel completely. We decided to build our application on top of a user-space application that takes the latter approach, called Sting. Our basic architecture, then, is a command-line program, which we call ACKRUE, that gains access to the network interface via a raw socket, receives packets through libpcap, and stops those packets from reaching the operating system via a firewall rule. ACKRUE implements a very stripped-down version of a TCP receiver, and is limited to sending a single packet, which is just enough to make an HTTP GET request.

A connection to the target HTTP server is made via the standard three-way handshake, during which time a maximum window size and MSS are established. After sending a single GET request, ACKRUE goes into receiving mode and stays there until it receives a FIN from the server, at which point it tears down the connection normally. While in receiving mode, ACKRUE waits for packets to arrive, always ACKing the highest sequence number seen.

In order to handle the "holes" created by never admitting that a packet has been lost, ACKRUE manages a linked list of byte ranges that it will need to go back and get via an HTTP 1.1 range request. If a packet arrives out of order, ACKRUE will already have added a hole to its linked list for the bytes in that packet. Hence, the linked list supports a fill operation, in which part or all of a hole is filled, eliminating, shrinking, or splitting the hole into two as necessary. Because the received file may have holes in it until ACKRUE has completed all range requests, we keep the file in memory, in a linked list of five megabyte buffers which are allocated as needed.

Once ACKRUE closes its connection with the target server, it runs through each hole in its list making a normal connection (using the kernel's TCP stack) and requesting the appropriate byte range with an HTTP 1.1 range request. Once all of the range requests are complete, ACKRUE writes the file to disc and exits.

5. Limitations and Simplifications

There are a number of limitations and simplifications that have been allowed into ACKRUE due to time constraints. This section briefly outlines these problems, describes why they exist, and, when possible, proposes a way to remedy them.

First, ACKRUE will fail if a packet is lost that contains some or all of the HTTP response header. We cannot recover the lost data later, because we cannot do a range request for header data. It may be possible to detect that there is a hole where the header should be and do an HTTP HEAD request to get just the expected HTTP response header to fill the hole.

Another limitation to ACKRUE is that downloads through an actual bottleneck router will cause a large number of drops (at the router), forcing the program to have to redo most of its work as range requests at the end. This is an artifact of our current ACKing strategy. There are a number of solutions to this problem, most of which just involve observation of the download (are we getting holes, how often, how large), and adjustment of our ACKing to compensate. The ultimate goal is to lower the send rate from the server just enough that we no longer get (substantial) packet dropping. Another solution might be to fall back to normal TCP congestion control when these situations are detected, but we then lose the "cheating" aspects that make ACKRUE so worthwhile.

Next, file size can be a problem for ACKRUE. We are currently limited to a maximum of near 4 gigabyte files. This is due to both the use of unsigned 32-bit integers and the lack of logic to handle wrapping of sequence numbers around through the original sequence number. Fixes to these problems are achievable, but may not be worth the effort considering that few files on the Internet are that large.

Also, not all servers support HTTP range requests, so files downloaded from these servers will be left incomplete if they had any holes. This could be fixed by first checking to see if the server supports HTTP range requests. If it does not, we could use standard TCP sockets to establish our connection and perform a normal download.

Lastly, we still observe unusual behavior when connecting to some servers. This is due to the fact that neither Sting's TCP stack nor our HTTP support are all that robust. The obvious solution would be to flush out more of our HTTP support and handle more corner cases in the TCP stack.

6 Evaluation and Testing

For our testing machine setup we used an UltraSparc machine with 256 MB of RAM connected to the Cal Poly campus internet via a 100Mb switch. In the first type of test we compare the total download time of ACKRUE to the Linux application "wget". The "wget" download times are calculated using the "time" application. The ACKRUE download times are output from the program using the GetTime() library call.

Table 1 shows some sample download times for ACKRUE versus wget on the same machine. We are downloading a large file, an install for OpenOffice, from a server in England. ACKRUE performs very well for this example. We also tried compiling ACKRUE with and without compiler optimizations. The results show little difference for a large download, and we observed that it does improve download times for smaller files. This is likely because optimization means that we should be able to do user-space processing of incoming packets more quickly, so we can ACK more quickly and reach a higher send rate faster.

The performance of ACKRUE was not as favorable for another large download -- a copy of MySQL from a server in Finland. We do not need to present a table of results, but only need to describe the problem and its severity. First, we noticed that the download was taking considerably longer than wget. Closer inspection showed that there were large holes (dropped packets) in the data received during the first attempt at downloading the file, so the second step, falling back on range requests, had to download a great deal of the file. Our guess was that we were losing so many packets because of a bottleneck in our route to the server. Our ACKing strategy, by its definition, ignores congestion control. This example shows that ignoring congestion control might not always be a good idea -- particularly when there is congestion. Congestion at the bottleneck forced a large number of our packets to be dropped. A traceroute to the server showed a very high-latency hop that we attributed to be a satellite link. This was likely our bottleneck to Finland. Section 5 mentioned the bottleneck problem and proposed solutions.

Figure 2 plots the arrival time for most of the packets from the OpenOffice download. We can see that, for this link (without a bottleneck), we get a fairly consistent send rate. Figure 3 shows a zoomed-in section at the very beginning of our transfer. This figure shows two key things. One, the levels are likely due to the fact that we are receiving this packets in user-space, so there are jumps in arrival time based on when we don't have a time slice. Zooming in on any other section of this graph shows a similar leveling; packets seem to arrive in short bursts. Next, we can see the sender moving through slow start and expanding its window. We receive small, then larger and larger bursts of packets. Again, we get them at almost the same time due to our stack being in user-space. Figure 4 shows a slightly different view of the "bursts" of receives caused by time slicing. It plots the time difference between the arrival of a packet and the packet prior to it. The regularity of the jumps and their size suggest that they are due to process scheduling.

Local high-bandwidth test:
On campus: www.csc.calpoly.edu

Finland: mysql.tonnikala.org Downloads/MySQL-5.0/
mysql-standard-5.0.27-linux-i686-glibc23.tar.gz hope.tar.gz

Good performance, large download test:
England: download.mirror.ac.uk/sites/sunsite.dk/openoffice/stable/1.1.5/
OOo_1.1.5_LinuxIntel_install.tar.gz dld.tar.gz

Table 1:
[These tests were run without the compiler optimizations]
Test 1:
ACKRUE: 3min 42.887 seconds
WGET: 7min 54.351 seconds

Test 2:
ACKRUE: 3min 43.386 seconds
WGET: 7min 54.566 seconds

Test 3:
ACKRUE: 3min 43.167 seconds
WGET: 7min 16.750 seconds

[These tests were run with -O3]
Test 1:
ACKRUE: 3min 49.415 seconds
WGET: 8min 6.175 seconds

Test 2:
ACKRUE: 3min 49.263 seconds
WGET: 8min 2.980 seconds

Test 3:
ACKRUE: 3min 42.856 seconds
WGET: 7min 16.687 seconds

7. Future Work

There are some features that could be implemented as future work in order to make ACKRUE a more "useable" tool. Some of these features include dynamically determining if ACKRUE is speeding up a download or not, modifying ACKRUE such that it could be inserted directly into the kernel, and transforming ACKRUE into a library. As shown in section 5, for some cases ACKRUE actually performs worse than a well behaved receiver (wget). These cases mostly involve situations where there is some significant bottleneck in the network. In these cases, large "holes" in the file are created that later need to be filled. ACKRUE could utilize the number and size of holes that are being created in the file (due to packets being dropped at the bottleneck) and dynamically decide to switch to a well behaved receiver. Thus, ACKRUE would guarantee that a download would occur in no less time than normal, but potentially could be significantly speed-up. Currently ACKRUE runs in user-space, and context switches between kernel-space and user-space are required when both receiving TCP packets and when sending ACKs. There is potential for an increase in download speed if ACKRUE were able to be loaded directly into the kernel as this context switching between packet arrivals and packet (ACK) departures could be eliminated. Finally, ACKRUE is currently an application that is executed in a similar way to "wget". However, other applications that download HTTP files might wish to utilize ACKRUE in order to increase performance of the application. In this case, ACKRUE would be better suited as a library that future application developers could use instead of the traditional socket calls.

8. Conclusions

We were able to successfully demonstrate that in relatively small amount of time (only three masters students in only a few weeks) that an application can be developed that improves download speeds by ignoring TCP congestion control. Even though there is still significant future work left in order to implement a more "useful" application, our current version of ACKRUE can significantly speed up HTTP download rates. ACKRUE shows that many of the implicit assumptions in TCP congestion control can be exploited in simple manner that is very similar to what has been previously proposed [1].

9. References

[1] Savage, S. et al., TCP Congestion Control with a Misbehaving Receiver, ACM SIGCOMM Computer Communication Review, v.29 n.5, Oct 1999.
[2] Pradhan, P. et al., Daytona: A User-Level TCP Stack, http://nms.csail.mit.edu/~kandula/data/daytona.pdf, 2006.
[3] Savage, Stefan, Sting: a TCP-based Network Measurement Tool, *In proceedings USENIX Symposium on Internet Technologies and Systems,* Colorado, Oct 1999.
[4] Bellardo, John and Stefan Savage, Measuring Packet Reordering, *In proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*, Marseilles, France, Nov 2002.
[5] Allman, M, V. Paxson, and W. Stevens, TCP Congestion Control, RFC 2581, April 1999.
[6] Fielding, R. et al, Hypertext Transfer Protocol - HTTP/1.1, RFC 2616, June 1999.
[7] D.-M. Chiu and R. Jain, Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. Computer Networks and ISDN Systems, Vol. 17, pp. 1-14, 1989.
[8] Fall, K., and Floyd, S., Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. Computer Communication Review, V. 26 N. 3, July 1996, pp. 5-21.
[9] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. IEEE/ACM Transactions on Networking, 1(4):397-413, August 1993.
[10] L. S. Brakmo and L. L. Peterson, TCP Vegas: End to End Congestion Avoidance on a Global Internet . IEEE Journal of Selected Areas in Communication, Vol. 13, No. 8, pp. 1465-1480, October 1995.