A Client-side Split-ACK Tool for TCP Slow Start Investigation

Michael Welzl and Rolf Erik Normann
Department of Informatics, University of Oslo, Norway
Email: michawe@ifi.uio.no, renorman@ifi.uio.no

***Abstract*—TCP's Slow Start phase has recently gotten much attention because Google proposes to use a larger starting point (Initial Window, IW). For their large-scale measurement study, they have applied a change to their well-connected web server; how the larger IW would affect other servers, or hosts that would use it to send data across a low-bandwidth or wireless connection, is less understood. We present a tool that, by splitting ACKs into multiple pieces at the beginning of a connection, can be used to trick a host into sending a larger number of packets than normally intended within one RTT in Slow Start. Test results indicate that, while many hosts do not react to our tool, some do – probably enough to use the tool for measurements.**

I. Introduction

It is well known that most web transactions are short-lived, and often terminate before TCP even exits the Slow Start phase. This makes this particular phase of TCP very important for the speed of the Internet as most of its users perceive it today. Understandably, as Slow Start was conceived in the late 1980s, several proposals for improving it have been brought forward. However, since Slow Start builds on the assumption that the host executing it starts with zero knowledge about the state of the network, it is hard to define a method that is faster, yet convincingly robust – and to make a real change, a proposal must be convincing enough to be taken up in some major Operating Systems and/or accepted as a standard.

Slow Start lets a TCP host increase its congestion window (cwnd) – an upper limit on the sending rate – grow exponentially. Since exponential growth is in fact very fast, a simple and easily implementable way to speed it up is to just use a larger starting point (i.e., do we begin with 1 or 20 packets before we let their number grow exponentially?). This starting point, called the Initial Window (IW), is currently defined to be up to 4 packets, depending on the packet size – 3 in case of the arguably most common packet size, the 1500 byte Ethernet MTU [1]. Google has proposed to increase this parameter to 10, and carried out an in-depth study to justify this choice [2]. This proposal has led to a significant amount of discussion in the IETF, and was adopted in Linux as of version 2.6.39, released on 18 May 2011. Google maintains a collection of links related to their proposal at [3].

The main argument for increasing the initial window is that Internet links have become faster since the window of 2-4 packets was defined. A large part of the IETF discussion therefore revolved around possible dangers for users with low-bandwidth connections – among other things, the proposers were asked to investigate the impact of their IW=10 proposal on users in regions where the average Internet access link rate is known to be smaller than in most western countries, causing them to study the performance seen by users in Africa and South America [4].

Since Google tested IW=10 with their own web server, the Google investigation is truly a large-scale experiment. However, their experiment was unidirectional: the sender, which employed the larger initial window, was their own server. Internet access links, which are frequently the bottlenecks of Internet paths, are often also asymmetric, with a much smaller upstream than downstream bandwidth. It would therefore be especially interesting to carry out a similarly large experiment where the sender is the end user, on a low bandwidth and possibly asymmetric Internet connection. One of the concerns, there, is latency-critical traffic such as Voice over IP (VoIP), or online interactive games – would such traffic suffer from the queuing delay caused by a larger initial window?

Google's web server is under Google's control, but the clients are not. How can we, then, make clients use a larger initial window? Now that Linux has adopted IW=10, many will already use it, but how could we, for instance, test if IW=8 or IW=12 would have been a better choice?

In this paper, we present a tool that makes this possible – albeit with a delay of one round-trip time (RTT) – by splitting ACKs into smaller pieces (N ACKs acknowledging X/N bytes each instead of one ACK acknowledging X bytes). This provokes some senders to send a larger number of packets in one go. ACK splitting is not new, and has been addressed by standards; it is still possible to apply it in Slow Start because the fix to the problem has long been regarded as experimental, in particular for Slow Start, and is not fully implemented in all current Operating Systems. We will give an overview of this situation combined with a look at related work in the next section. In Section III, we will describe our tool, and we will present our test results in Section IV. Section V concludes.

II. Related Work

We divide related work along two axes: ACK splitting and Slow Start improvement.

ACK splitting, also called "ACK division", was first documented in [5] as a means for a misbehaving receiver to trick a sender into sending data faster than it normally should. This was addressed with a fix to TCP called "Appropriate Byte Counting" (ABC) in [6], which is an Experimental RFC, i.e. it is not recommended for general use in all Internet hosts.

As its name suggests, ABC has the sender count bytes, not ACKs, as a basis for increasing cwnd. While this is a rather safe approach in Congestion Avoidance, its usage is a bit more critical in Slow Start, where the exponential growth could cause the sender to do an immediate rate jump if ACKs are lost or Delayed ACKs are applied. For instance, if a single ACK acknowledges 5 new packets worth of data, this could cause a sender to immediately send out 5 more packets without waiting for any extra ACKs. This is somewhat at odds with the "ACK-clocking" idea behind TCP congestion control, and was therefore recommended to be limited to a growth of at most two packets in [6]. Quite recently, ABC was incorporated in the general TCP congestion control specification [1], but only as a recommendation, not as a strict requirement.

A study published in 2005 [7] used an active tool called TBIT to find that many TCP implementations do not perform ABC. This tool is quite similar in spirit to ours, in that it changes the default behavior of a TCP client to deduce a certain server behavior. However, the goal of reliably detecting whether ABC is implemented (in TBIT) differs from provoking a different sender behavior (in our tool), making TBIT much more complex than our tool, and less appropriate for Slow Start experiments such as trying a larger initial window.

ACK splitting was successfully used in simulations in [8], which also refers to a handful of other works where this method was successfully applied for very specific scenarios (wireless handovers and satellite communication). All in all, there are not many implementations of ACK splitting, and while there are indications that some TCP implementations might, at least in Slow Start, quickly increase their cwnd in response to split ACKs, it is not clear how many hosts really do so today.

Regarding proposed improvements to Slow Start, they can roughly be divided into proposals focusing on the beginning or the end of the phase. The beginning, the initial window, was originally specified to be one packet [9]. It was raised to 2-4 packets, depending on the packet size, in [10]. This was later folded into the standard in [1]. As already discussed, Google are currently proposing to raise this value to 10 packets; alternative proposals were also made, mainly to avoid repeatedly discussing the IW "magic number" in the IETF, and they are available via [3].

Slow Start terminates when it reaches a threshold called ssthresh, which is set to half the number of packets in flight upon congestion during the Congestion Avoidance phase, and has an undefined initial value (a constant, often also retained from previous connections to the same host). At the beginning of a connection this value does not normally reflect network conditions – but given that the time available to determine it is short, proposals such as [11] and [12] to base it upon measurements are usually regarded as too unreliable to warrant standardization. The RFC that proposes an alternative behavior towards the end of Slow Start, "Limited Slow-Start" [13], is experimental, and is based on fixed numbers, reducing the increase factor as cwnd grows to avoid a too severe overshoot.

Finally, there are proposals to avoid Slow Start altogether:

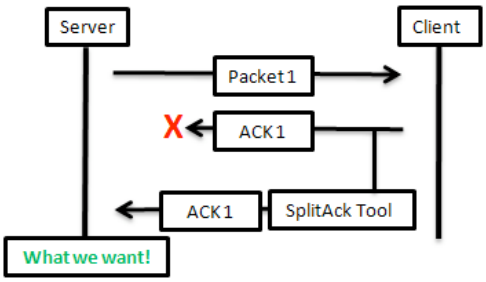

Fig. 1. Internal operation of the Split-ACK tool

Quick-Start [14] allows a sender to jump to a higher initial rate if explicit feedback from all routers along the path (carried in an IP option) is available, and Jump-Start [15] truly skips Slow Start, using an initial rate based on the receiver's advertised window and the amount of packets enqueued at the sender.

All these works underline the significance of Slow Start, and the demand for ways to test different approaches. For other initial windows and maybe also methods such as Jump-Start, the tool that we present in the next section could be the right means to carry out a large-scale measurement study.

III. The Split-ACK Tool

Our tool splits ACKs of a regular TCP client before they actually go back to the server. We implemented it in Linux, using kernel version 2.6.31-18. For convenience and portability, we decided to design our tool in such a way that it can run in user space (albeit requiring root privileges). This means that we have no means to actually modify packets – but we can access them, using the "libpcap" library, and then re-generate them, using raw sockets. The problem with this approach is that we then have a duplicate packet: the original ACK. This is sent back to the server in addition to our split ACKs, causing a different response than what we want to obtain. In particular, our own ACKs would mostly be ignored because they would usually arrive later than the original ACK due to our tool's extra processing overhead.

Our solution, shown in Figure 1, was to run our tool under a different user than the TCP application being tested (in our case "wget"), allowing us to use the "iptables" Linux firewall and have it operate on the original ACK only. The idea was to essentially have iptables drop the original ACK, while we create our split ACKs to take its place. Since, however, libpcap accesses the network interface to obtain packets, this approach does not work because the original ACK would not even reach far enough to be visible to libpcap. We therefore made iptables set the IP header's Time-To-Live (TTL) field to 1 in the original ACK instead of dropping it, meaning that the packet actually traverses the network interface, but is then dropped by the first hop (typically an access router). Upon creating split ACKs, our tool now also has to update the Time-To-Live field to a larger value.

The idea behind splitting ACKs is to create multiple ACKs that each acknowledge a subset of the amount of data that would normally have been acknowledged by the original ACK.

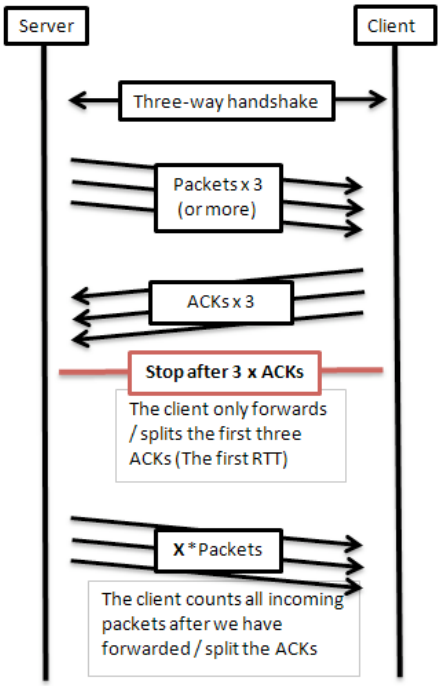

Fig. 2. On-the-wire operation of the Split-ACK tool

Our tool divides the amount of acknowledged data equally among the ACKs. For instance, when splitting a 1000-byte ACK carrying the ACK number 2001 into four ACKs, our tool would generate ACKs carrying the numbers 1251, 1501, 1751 and 2001, respectively. The amount of data acknowledged by an ACK is determined by looking at the ACK number used in the ACK just before the one that is split (we call this `prev_ack`).

We still have to let our tool decide which ACK packets to split. Since the idea is to only test the behavior of the server at the beginning of Slow Start, the tool only splits the first N (configured, default 3) ACKs after the connection handshake (i.e. not the SYN/ACK, which is only used to determine the initial value of `prev_ack`), and remains silent afterwards. Then, it counts the number of packets arriving from the server and stops after a timeout (3 seconds by default, to make sure that the tool would notice retransmitted packets after a timeout; note that this not related to the default RTO of 3 seconds, as the RTO is already updated during the handshake). Figure 2 shows the on-the-wire behavior of our tool with the mentioned default parameters.

IV. Test results

We carried out tests of our tool on a desktop computer, with a wired network card, Marvell Yukon 88E8056 PCI-E Gigabit Ethernet Controller, running a 100 MBit local connection to the router / ADSL modem. The modem was a Thomson TG585, running software version 8.2.3.10. The Internet connection was ADSL2+, also known as ITU G.992.5 annex B, provided by Powertech. The speed of the connection was 14.4 MBit downstream and 1.45 MBit upstream, giving the connection a 10-to-1 ratio between download and upload rate. Most of the software configuration was the same as the implementation software. The desktop computer was running Ubuntu Linux, with kernel version 2.6.31-18. The pcap library was running version 4.1.1. We utilized wget 1.12 to communicate with web servers, and all the tests were done against the HTTP protocol.

We have disabled delayed ACKs and used the default values for our tool, which means that we split the ACKs that were caused by the initial window (at least 3, given the packet sizes that we saw), and all our tests were therefore performed within the connection's second RTT (excluding the connection establishment handshake). To ensure consistency, every test was carried out 5 times, with a 3-second pause in between.

To get an idea of how hosts react to split ACKs in Slow Start, we first tested some individual servers running known Operating Systems. Then, to see how this differs from web sites that can be assumed to be configured for high efficiency, we also tested a handful of web servers of well-known companies, where we determined the OS with "nmap". Finally, to obtain a first rough overview of the general behavior of Internet web servers, we tested our tool against the top 600 web servers from Alexa [16].

A. Individual servers

In this section we present the preliminary results of the nine different hosts that we have tested at first. We have examined five different operating systems:

- Windows 2003 SP1 Server (host `connexion.at`, a friendly company who we know to run this OS)
- Linux 2.6.9 (our local host `heim.ifi.uio.no`)
- Solaris 10 (our local host `folk.uio.no`)
- FreeBSD 7.0 (host `freebsd.org`, which is known to run this OS)
- Windows 7 (host `s1010-0002.dsl.start.no`, which was a friend's desktop PC running a small web server for our tests)

The four different tested web servers of well-known companies were:

- Google (host `google.com`, running OpenBSD 4.3)
- Microsoft (host `microsoft.com`, running Windows 2003 SP3)
- Apple (host `apple.com`, running Linux 2.6.9)
- Oracle (host `oracle.com`, running Linux 2.6.9).

Before even beginning to split ACKs, we set the split parameter to 1 (i.e. the ACK was not really divided, just one ACK was generated) to investigate the default behavior of the hosts, as a baseline that we could compare the split results against. Given the recommended IW of 3, and answering with just 3 ACKs to these 3 first packets, the expected behavior was that 6 more packets would arrive and the connection would then stall, except for possibly retrying after a timeout. Table I shows what we actually saw.

Many servers sent less than the expected 9 packets. There may be various reasons for this – e.g. an IW that is *larger* than

Tests	*Original data packets (not counting duplicates)*
Windows 2003 SP1	**7**
Linux 2.6.9	**7**
Solaris 10	**8**
FreeBSD 7.0	**7**
Windows 7	**6**
google.com (OpenBSD 4.3)	14.6
microsoft.com (Windows 2003 SP3)	24.6
apple.com (Linux 2.6.9)	**9**
oracle.com (Linux 2.6.9)	**11**

TABLE I
BASELINE TEST RESULTS. NUMBERS IN BOLD ARE CONSISTENT RESULTS FROM ALL 5 TESTS, ALL OTHER NUMBERS ARE AVERAGES.

the proposed standard. Consider a sender that uses an IW of, say, 5 instead of 3 packets, but only receives ACKs for the first 3 of them; not receiving ACKs for the last two packets might cause this sender to experience a timeout rather soon, causing it to retransmit these packets instead of sending new ones. Retransmits are not shown in the table because they are not conclusive – they can have been caused as just described, or at the end of the procedure, after timing out for packets that were prompted by our 3 ACKs.

We did not investigate this issue further because this was not the goal of our study, but we note that Table I indicates that many servers did not use an IW of 3, but a smaller or, more probably (because this also correlated with a larger number of retransmits), a larger one. This finding is also in line with the results in [7] and [17]. The test results of the Google and Microsoft web servers varied widely; Microsoft sent as many as 36 packets in one of our tests (this strange behavior of the Microsoft web server has been reported before [18]). Such behavior may well be caused by an intermediate device and not by the host itself.

We then split the three ACKs into 2, 4, 8 and 12 pieces, towards all of the servers above. The outcome was somewhat disappointing: splitting ACKs did not have a big effect on most sites. We only saw the expected cwnd growth with Windows 2003 SP1 and Solaris 10; the results for these two servers are shown in Figures 3 and 4, respectively. The Apple server showed a minor reduction of the number of packets when we split the ACKs into 12 pieces, and the Microsoft and Google web servers (i.e., the ones sending much more than the others in the baseline case) showed a minor / partial reduction too. All other hosts were ignorant to our ACK splitting tests, i.e. the outcome was exactly the same as in the baseline case.

B. Top 600 web sites

We then turned to investigating the behavior of the web, for which we picked the top 600 web sites from Alexa [16]. The idea was not to carry out an extensive measurement, but to obtain a rough idea about the general usefulness of our tool towards web servers. We carried out two sets of tests (as before, 5 runs each): one without splitting the ACK, to obtain a baseline result, and one where we split each of the three ACKs into 4, resulting in a total of 12 ACKs. Figures 5 and 6 show the original data packets (i.e., excluding retransmits) that our tool counted in these tests. The diagrams show the minimum,

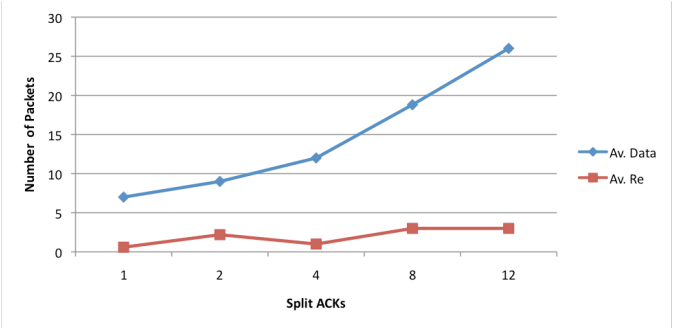

Fig. 3. Windows 2003 SP 1: average number of data packets and retransmits caused by splitting ACKs (1 Split ACKs = baseline, i.e. no split)

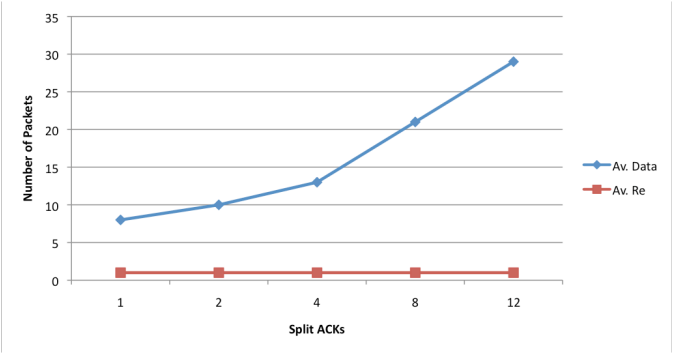

Fig. 4. Solaris 10: average number of data packets and retransmits caused by splitting ACKs (1 Split ACKs = baseline, i.e. no split)

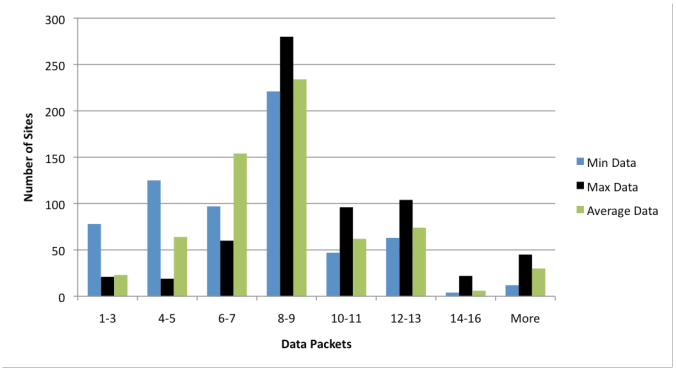

Fig. 5. Baseline result: original data packets

maximum and average data obtained in the five tests per host. In the baseline test, the average number of returning original data packets of all the sites was around 8-9 packets, and about 83% of all the tested sites were in the range of 6-13 packets. With an IW of 3-4 packets, we expected to receive around 9-12 packets within the first two RTTs, and thus the expectation correlated with the results.

The split ACK result is almost equal to the baseline result, but it shows a slight increase in the number of sites returning 10-16 packets. There were outliers, where the tool worked very well – e.g., the `aol.com` server returned 11 packets (with 2 retransmits) in the baseline test, but 23 packets (with 4 retransmits) with ACK splitting. On the whole, however, this result is rather disappointing. This situation changes when

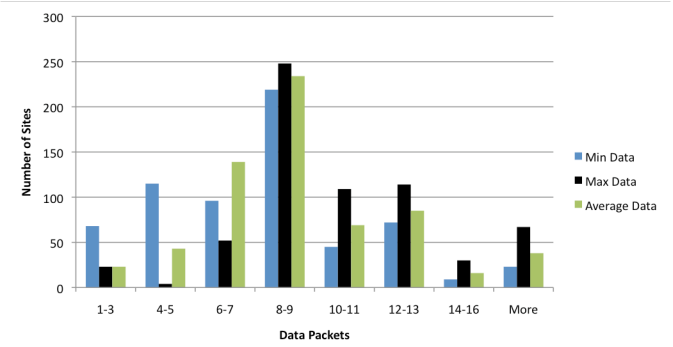

Fig. 6. ACK splitting result: original data packets

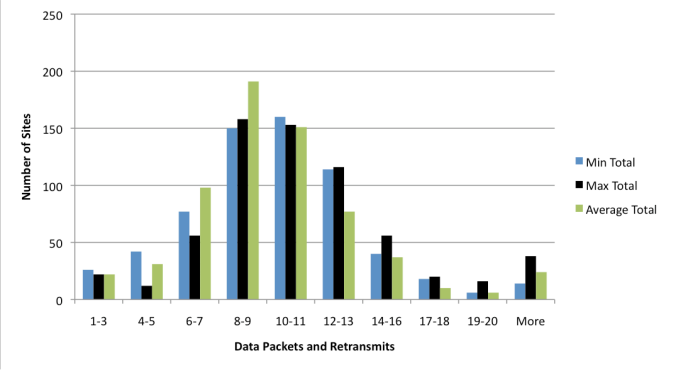

Fig. 7. Baseline result: all packets (including retransmits)

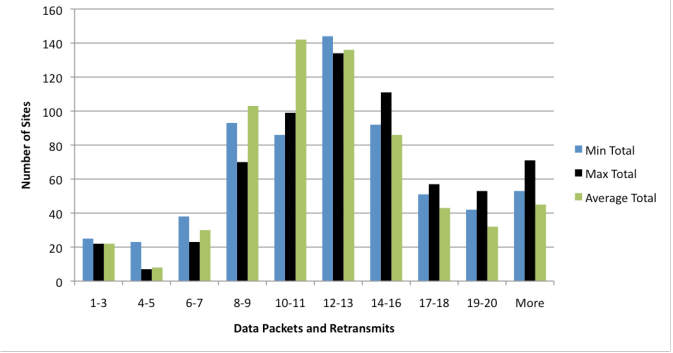

Fig. 8. ACK splitting result: all packets (including retransmits)

we look at the total number of packets (i.e. original data packets plus retransmissions). The corresponding baseline and split ACK results are shown in Figures 7 and 8. The baseline results are consistent with the baseline results for original data packets, because most of the sites were in the 6-13 packets range, but they are more spread out because of the larger total number of packets considered. The split ACK result showed a significantly increased amount of traffic, meaning that our tool was successful.

V. Conclusion

We have presented a simple and portable user-space tool that, by splitting ACKs, can cause a host to send out a larger number of packets than usual within one RTT in slow start. This can be useful to test the impact that, e.g., a larger Initial Window has on other traffic regarding delay or packet loss. Our test results have shown that our tool rarely succeeds in causing hosts to send out more original data packets, but it can cause them to send more retransmits. In other words, our tool can hardly be used to attain a speedup for the web by emulating a larger server initial window via a client-side trick; it does, however, succeed in making many servers send out a significantly larger total number of packets, which means that it can be used for doing large-scale measurement studies where, e.g., hosts using a P2P application are tricked into sending a data probe of sorts.

Our first set of tests mostly focused on server versions of Operating Systems, and the Internet tests only looked at web servers. We do not yet know how typical client-side OSs like Windows XP react. We believe that this would be interesting to investigate, as a basis for applying our tool to different applications such as BitTorrent, for example. Our tool is available for download from `http://heim.ifi.uio.no/michawe/research/tools/splitack`

References

[1] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," RFC 5681, Sep. 2009.

[2] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing TCP's initial congestion window," *SIGCOMM CCR*, vol. 40, no. 3, 2010.

[3] "TCP IW10 links," http://code.google.com/speed/protocols/tcpm-IW10.html, Mar. 2011.

[4] N. Dukkipati, J. Chu, and Y. Cheng, "Increasing TCP initial window," in *IRTF ICCRG meeting, IETF 78*, 2010. [Online]. Available: http://www.ietf.org/proceedings/78/slides/iccrg-3.pdf

[5] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP congestion control with a misbehaving receiver," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, pp. 71–78, 1999.

[6] M. Allman, "TCP Congestion Control with Appropriate Byte Counting (ABC)," RFC 3465, Feb. 2003.

[7] A. Medina, M. Allman, and S. Floyd, "Measuring the evolution of transport protocols in the internet," *SIGCOMM Comput. Commun. Rev.*, vol. 35, pp. 37–52, April 2005.

[8] A. Arcia, "Modifications du mcanisme d'acquittement du protocole tcp: valuation et application aux rsaux filaires et sans fils," Ph.D. dissertation, Telecom Bretagne, Rennes, France, December 2009.

[9] W. Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," RFC 2001, Jan. 1997.

[10] M. Allman, S. Floyd, and C. Partridge, "Increasing TCP's Initial Window," RFC 3390 (Proposed Standard), Internet Engineering Task Force, Oct. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3390.txt

[11] J. C. Hoe, "Improving the start-up behavior of a congestion control scheme for TCP," in *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '96. New York, NY, USA: ACM, 1996, pp. 270–280.

[12] S. Ha and I. Rhee, "Taming the elephants: New TCP slow start," *Computer Networks*, vol. In Press, Corrected Proof, 2011.

[13] S. Floyd, "Limited Slow-Start for TCP with Large Congestion Windows," RFC 3742, Mar. 2004.

[14] S. Floyd, M. Allman, A. Jain, and P. Sarolahti, "Quick-Start for TCP and IP," RFC 4782 (Experimental), Internet Engineering Task Force, Jan. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4782.txt

[15] D. Liu, M. Allman, S. Jin, and L. Wang, "Congestion control without a startup phase," in *Proc. PFLDnet*. Citeseer, 2007.

[16] "Alexa the web information company," http://s3.amazonaws.com/alexa-static/top-1m.csv.zip, Jun. 2011.

[17] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger, "TCP revisited: a fresh look at TCP in the wild," in *SIGCOMM IMC '09*. New York, NY, USA: ACM, 2009, pp. 76–89.

[18] "Google and microsoft cheat on slow-start. should you?" http://blog.benstrong.com/2010/11/google-and-microsoft-cheat-on-slow.html, May 2011.