A Tunable Slow Start for TCP

Andrés Arcia-Moret*†, Omar Diaz* and Nicolas Montavont†
*Universidad de Los Andes, Mérida, Venezuela
†Institut Telecom / Telecom Bretagne, Université Européene de Bretagne, France
Email: {andres.arcia, omardiaz}@ula.ve, nicolas@montavont.net

***Abstract*—Recently, there has been an interest in accelerating the initial bandwidth discovery phase of TCP. In this paper the feedback provided by TCP Acknowledgements (ACKs) does not only mean (as in legacy TCP) that a single data packet has gone out of the network, but also that new data packets can be sent in bursts as opportunistically indicated by the receiver. For this purpose, regular ACKs can be split so that an acknowledged portion of a segment instructs the sender to open more aggressively its congestion window. This article details the implementation and testing of ACK division which has been used so far to compensate the effect on TCP performance of random losses, spurious timeouts and handovers at transport level. We specifically show the benefits of using the ACK division in slow start for long delay networks.**

***Index Terms*—Split ACK, TCP, slow start, divack.**

I. Introduction

The advent of faster access and core networks yields to the need of efficiently using the available bandwidth [1], [2]. This efficiency has to consider at least three aspects: the rapid use of the available bandwidth, the fair use of the bandwidth and the effective increasing of the TCP goodput. At the same time it is well-known that the omnipresence of TCP and the increasing number of technologies for the Internet makes difficult proposing changes on TCP that can be widely adopted; and naturally TCP does not adapt well to all possible scenarios. For example, for high delay networks (e.g., satellite networks), TCP performance is poor unless there is a performance enhancing proxy (PEPs) to accelerate data transfers [3]. However, PEPs break the so-called Internet end-to-end principle by storing the state of TCP connections. On the other hand, long delays to reach the TCP fair-share of the bandwidth affect medium size transfers, such as web or interactive traffic.

In TCP congestion control, the sender controls the sending rate by manipulating a congestion window (*cwnd*) which limits the number of packets that can be sent without having received an acknowledgement (ACK). Each *new ACK* is taken as a signal of non congestion and the sender can safely increase the *cwnd*, so that the sender injects more data into the network without waiting for a confirmation. In this proposal, we profit from this semantic by splitting each regular ACKs in what we call *divacks* (and preserve its validity at the same time). In a regular scenario, TCP just ignores *divacks*, and middle-boxes takes them as regular traffic and thus we propose our own semantic for *divacks* without altering the regular interpretation from the network.

A. The Transmission Control Protocol (TCP)

TCP allows a reliable transmission, congestion control and reordering treatment. However, it does not provides a guarantee on the transmission rate, and applications must accept their fate rate found by the interaction of the TCP with different bottlenecks. When sending data, a sender divides the application messages into several segments, each of which should correspond to a Sender Maximum Segment Size (SMSS) determined during the connection establishment. Each segment is marked with a sequence number to guarantee the order and reliability of the transmission. Whenever the receiver gets a segment, it sends an ACK to notify that data preceding the received sequence number has been well received and can be passed to the application and removed from the reception buffer.

TCP uses some mechanisms to improve its performance and to fairly share the bandwidth on the bottlenecks. Particularly, the congestion window (*cwnd*) limits the number of segments that a sender puts in-flight without receiving an ACK. When a connection starts, TCP uses the algorithm *slow start* to ramp-up the *cwnd*. Once TCP discovers the capacity of the network, TCP changes the strategy to congestion avoidance to carefully test for the available bandwidth and to fairly share the bottleneck capacity. The objective of these mechanisms is to control the amount of data that enter into the network.

TCP implements reliability and advances its *cwnd* as follows. TCP identifies each segment by a unique sequence number for data segments, and an acknowledgement number for ACKs. The sequence number serves as a mark for the first byte transported by every segment. It could also show the total number of bytes already transferred if subtracted to the very first sequence number. On the other hand, the receiver uses the ACK number to indicate to the sender the amount of data already received. So when a receiver obtains segment with a sequence number s transporting n bytes, it will send an ACK numbered $s + n + 1$ indicating that it has correctly received $s+n$ bytes. This indication also means that a receiver is waiting for the next segment containing the byte $s + n + 1$.

Although a polemical feature [4], congestion control algorithms of TCP deal with segments (and not bytes). These algorithms increase the *cwnd* in terms of SMSS bytes for every received ACK during the *slow start* phase. While in *congestion avoidance*, TCP increases by $1/cwnd$ for every received ACK that acknowledges new data, i.e., the *cwnd* grows at the approximate rate of 1 SMSS per round trip time (RTT) [5].

B. The ACK division

As described in previous section, TCP tracks the sending and reception of bytes, but manages congestion in segments. This characteristic allows varying the speed of the *cwnd* growth as follows: as shown in Fig. 1, instead of sending a single ACK for every received data packet, the receiver can send several ACKs to the sender (referred as *divacks* from now on), so that every divack confirms the reception of part of the segment. We propose that this partial confirmation signals the sender to put more packets in the network. Specifically, one data packet for every *divack* received in slow start.

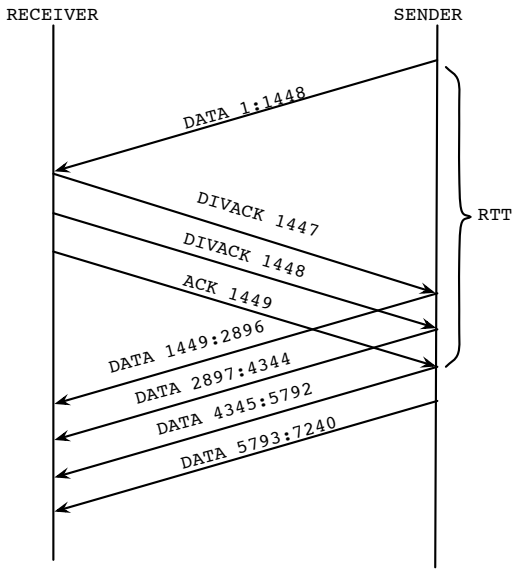

Fig. 1. A transmission using *divacks*.

This technique, known also as Split ACK, has been reported by Savage et al. [6] along with other possible modifications to the ACK clocking. Every ACK produced in this way, is perfectly valid because it acknowledges data that has been received and not confirmed. Besides, divack has a different effect on the cnwd growth depending on the congestion control algorithm (i.e., slow start or congestion avoidance). However, in *slow start* the growth effect results far more aggressive than in *congestion avoidance*. Savage et al. have described this kind of dynamic as a possible flaw of TCP, however Arcia-Moret [7] discuss, based on an extensive simulation campaign that a receiver abusing of this technique, would obtain an opposite effect as that described by Savage et al, i.e., the TCP performance could be rather impaired. And so our interest on studying this technique in a Linux kernel serving as a base of a tunable slow start for high delay networks.

Using divacks allows the TCP receiver to increase the TCP sender *cwnd* faster than regular 1 ACK per packet or 1 ACK every other packet version of TCP. The **receiver** thus, can control the *cwnd* growth rate through the arbitrary sending of *divacks* up until a maximum of 1 *divack* per byte in a segment. Using this frequency of divacks could make the *cwnd* to send in just few burst a complete file since this mechanism accentuates the well-known exponential growth dynamic. Nevertheless, several other factors such as the receiver window (*rwnd*), the available network bandwidth or the core-networks buffer sizes also limits the speed of transmissions.

The rest of the article is organized as follows. Section II discuss the state of the art on the use and evaluation of *divacks*. Section III describes the details for the implementation of *divacks* in a Linux Kernel v2.6. Section IV discusses about the TCP performance in a real testbed. Finally, in Section V we conclude the paper.

II. Applications for *divacks*

Several researchers have proposed to take advantage of *divacks* to improve TCP performance in wireless links, without having to modify the TCP sender. In such proposals, *divacks* are produced in a controlled manner, in order to help the sender in improving the recovering of the value of *cwnd* after it has been unduly reduced.

Jin et al. [8] propose a method for coping with decreases in *cwnd* due to wireless random losses, in a wired-cum-wireless network configuration. They assume a TCP sender on a fixed host (FH) in the wired part of the network, and a TCP receiver on a host in the wireless part. The BS then sends a fixed number of *divacks* to the FH when the retransmitted data packet arrives. By means of simulations, the authors show how *cwnd* rapidly recovers its size prior to the loss, thereby improving TCP performance.

Matsushita et al. [9] use *divacks* to quickly adapt the *cwnd* size after an upward vertical handover in a heterogeneous wireless networks composed of a WAN service such as a 3G cellular network and a IEEE 802.11g-based wireless LAN. They assume that the mobile node is able to detect the available bandwidth in the new access network with a higher bandwidth-delay product (BDP). Once the node enters into the new network, it sends *divacks* in order to rapidly increase *cwnd* in congestion avoidance mode. This allows the TCP sender to adapt faster to the new BDP and, therefore, to improve the throughput.

Hasegawa et al. [10] developed a receiver-oriented, end-to-end solution to recover from undue *cwnd* decreases caused by wireless random losses, in a wired-cum-wireless network with an asymmetric UMTS access link. Their proposal considers three main aspects: a mechanism for differentiating between wireless losses and congestion losses; controlling the duration of the *divack-generation* interval; and controlling the *divack* sending rate. Ideally, *divacks* would only be sent when wireless losses take place. To control the duration of the ACK-division period, they keep at the mobile node an estimate of the sender's *cwnd* size. So, when a random loss is detected, they send as many *divacks* as needed to recover the *cwnd* achieved by the sender just before the loss. Since sending *divacks* increases the load on the uplink, they monitor the length of the upstream queue at the mobile node. As long as this queue is empty when a new data packet arrives, the number of *divacks* generated for every incoming data packet is additively increased; otherwise, n is decremented. The idea of this mechanism is to adapt the *divack* rate to the available uplink bandwidth.

Arcia-Moret [7] presents an extensive simulation campaign in ns-2, in which the *divack* mechanism is evaluated for

different transmission length and congestion scenarios. His results suggest that *divacks* may not represent a systematic harm to the network. The performance of a TCP connection using *divacks*, depends on the number of *divacks* and the level of congestion of the network. If the number of *divacks* is high enough, the number of losses experienced by the sender increase and therefore the TCP performance is likely to decrease. On the other hand with an appropriate number of *divacks*, the TCP performance improves.

Besides the research proposals described before, we are aware of one implementation of *divacks* in a commercial product. Cisco routers implement a so-called Rate Based Satellite Control Protocol (RBSCP) [11], which was designed as a solution for improving the performance of transport protocols in wireless satellite links. With this solution, the router at one end of the satellite link acts as a performance-enhancing proxy (PEP) [12]. Among the mechanisms implemented by the PEP, there is a *divack-generation* option: ACKs arriving over the satellite link are divided by the PEP and sent over the wired part of the network. The mechanism uses a conservative default configuration ($n = 4$), but a network administrator may statically set a different value of $n \in \{1, 2, ..., 32\}$.

Finally, Welzl and Normann [13] use *divacks* to investigate the impact of bigger TCP Initial Window (IW) considering the $IW = 10$ recently proposed by Google [14]. The issue is whether different IW sizes, say 8 or 12 may also improve the transmission. They base their experiments on short-lived transactions to emulate web traffic. Since it is well-known that this kind of traffic normally lives within the slow start phase of TCP. Their experimental set is based on sending *divacks* for IW of 3 or 4 found in common operating systems. And so, by sending 3 *divacks* per data packet they obtain up until 12 data packets from a variety of web-servers. Tests over 600 web-servers show that big IW makes retransmissions increase. Their implementation of *divacks*, although in a Linux platform, uses *libcap* to generate *divacks* at application-level.

III. Implementation on a Linux Kernel v2.6

One way of implementing this is by doing a loop in the TCP code section for sending *divacks* in bursts. However, we believe that different sending patterns impacts differently the TCP performance. The sending *divacks* should reside in this part of the code, since they will be interpreted as regular ACKs at the data-sender. So, the code for the generation of control ACKs (i.e., duplicated ACKs, SYN or FIN) should not be touched, otherwise mechanisms like fast recovery could be affected.

A. Congestion Window considerations

Fig. 2 shows the upper and lower limits for the starting and ending of every segment. In the Figure, the value of *snd_nxt* in the sender, should match the value of *rcv_nxt* at the receiver. So, the receiver tracks the next incoming segment by saving the next expected byte in *rcv_nxt*. When the receiver sends an ACK, its sequence number is set to *rcv_nxt*. In this way the receiver informs that every byte just before the one announced in the ACK has been correctly received. Recall that TCP implementation on Linux Kernels v2.6 ignore *divacks*[1].

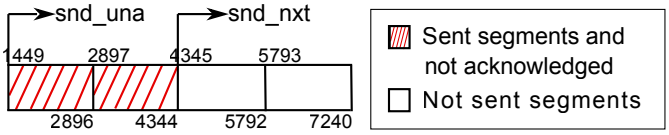

Fig. 2. The TCP congestion window. `snd_nxt` and `snd_una` at the sender.

On the other hand, sender *snd_nxt* and *snd_una* represent the first byte of the next outgoing segment, and the first byte of the oldest in-flight segment. We use these variables as *divack* limits so that the sequence number of every *divack* keeps between these two values. A *divack* should not be greater than *snd_nxt* because this segment has not been sent yet and should not be less than *snd_una* because these bytes have already been acknowledged. Moreover, a regular TCP implementation slides the *cwnd* when a full segment has been acknowledged, generating an extra packet in the burst.

We implemented a kernel variable called *sysctl_tcp_divack* to control the amount of *divacks* to send at any time. *sysctl_tcp_divack* controls either the rate of *divacks* or the deactivation of the mechanism. When experimenting with Linux Kernel, it is very common to change parameters from the console in run-time through `sysctl` interface. It allows to control the sending of *divacks* by setting in the following way:

- 0: Turns off the sending of *divacks*, it goes to the standard mechanism of one ACK per segment.
- n: Divides the outgoing ACK in $n + 1$ parts, that is, n *divacks* and the ending *full-ACK* to confirm the whole segment.

Note that all bytes up until $SMSS - 1$ could be profited with this technique. So, using the most aggressive strategy implies to send $SMSS - 1$ *divacks* and the final *full-ACK* to acknowledge every single byte of the segment. However, recall that the effective increase in the TCP throughput will depend on the congestion conditions [7].

B. Control ACKs versus data ACKs

ACKs has two fundamental rolls, either confirming the reception of data packets (as explained in Section III-A) to make the congestion window slide and grow, or controlling different TCP sub-mechanisms (i.e., establishment or tear down of a connection, data recovery, etc.). In the control case, sending *divacks* would impair the transmission. For example, since TCP is connection oriented, it is necessary to establish a connection. At the beginning the receiver acknowledges the connection-opening with a *SYN+ACK*. To end the data transmission, the receiver and the sender interchange *FIN* and *ACK* messages so that both extremes of the connection liberate allocated resources. In both cases, ACKs have a control semantic and *divacks* should not be used.

[1]Welzl and Norman [13] have found that a fair deal of OSs increase their throughput when sending divacks in slow start

When there are losses, the receiver uses the so-called duplicate ACKs to inform the sender that it should enter into the recovery mode. After three duplicate ACKs, lost packets are retransmitted before the expiration of the retransmission timeout. The sending of *divacks* during this phase would trigger spurious timeouts.

C. The ACK division algorithm

The *divack* mechanism has been implemented in a Linux OS V2.6.35.7. In Table I we show an algorithm to create and send *divacks*. This algorithm takes as an input the sequence number of an ACK and stores it into `last_byte` as a reference for assigning *divacks* numbers. The receiver can send *divacks* within a loop that repeats as many times as the number n of *divacks* to send. The first *divack* will have as a sequence number `last_byte - n`, and the following $n-1$ *divacks* will have a sequence number increased by 1 (line `12`), i.e., each one will confirm a new single byte. Finally, when the loop ends, at line `16` a *full-ACK* is sent confirming the whole segment. This *full-ACK* also signals for the next byte to receive with the sequence number *last_byte*. Recall that when the *divack* mechanism is deactivated (*sysctl_tcp_divack* $= 0$), the sequence number of the regular ACK will not be modified and the loop will not take place, therefore sending just the regular ACK (leaving TCP with the standard behaviour).

TABLE I
divack ALGORITHM AT THE RECEIVER.

```
1  /* check: ACK answer to a data segment */
2  If (TCP_segment has data && TCP_segment is
       not corrupt) then
3     /* Save the regular ACK number */
4     last_byte = TCP_segment.rcv_nxt
5     /* ACK seqno in snd_nxt and snd_una */
6     TCP_segment.rcv_nxt -=
          sysctl_tcp_divack
7
8     While (TCP_segment.rcv_nxt < last_byte)
9        /* divack with a new seqno */
10       form_and_send_ack(TCP_segment)
11       /* next divack number inc by 1 */
12       TCP_segment.rcv_nxt++
13    End While
14
15    /* send full-ACK */
16    send_ack(TCP_segment)
17 End If
```

IV. EXPERIMENTATION IN A TESTBED

The algorithm explained in Table. I has been implemented following the details of Section III-B. Recall that we call n the number of *divacks* that will be sent for every data packet, that is, the value of the *sysctl_tcp_divack*.

Fig. 3 shows the topology of the testbed. The cloud emulates a variable delay implemented by a computer running *netem*, a network emulator that allows to introduce Wide Area Network properties [15]. This computer dispose of a couple of networks interface cards (NICs) to buffer, delay, and loss packets through an highly configurable API. The access network corresponds to an 802.11 access point. We decided to add an 802.11 access to show the ping-pong effect as observed in [16] for low end-to-end delay.

Fig. 4 shows a screen capture of a Wireshark trace of a TCP transfer using the *divack* mechanism, with $n = 3$ and a roundtrip time of $250ms$ for data transmission, emulated with *netem*. Note that the handshaking has not been affected, ACKs have been divided in four valid ACKs and the receiver performs the *three-way handshake* and then it completes the opening of the connection with a single control ACK. After the reception of the first segment it is possible to observe the *divacks* technique behaviour, in this case with $n = 3$ the first divack confirms up until the byte 1446 obtained as $snd\_nxt - n$. Then, the following 2 *divacks* acknowledge a new byte each time, up until the final *full-ACK* that confirms the whole segment. Note within the algorithm in Table I at Line `4`, that the *full-ACK* number is *snd_nxt*.

A. Dynamics of the Congestion Window in Slow Start

Using *tcp_probe* it is possible to capture the changes inside the kernel on the *cwnd*, and observe the *cwnd* dynamic when *divacks* arrive. Fig. 5 shows a comparison of the *cwnd* growth when activating the *divack* mechanism with regular sending of one ACK per data packet and divacks with $n = 15$. This test corresponds to the *netem* emulating an RTT of $250ms$ (easily observable within the figure). The *divack* burst effect at the receiver is clearly appreciated, it provokes a greater increase of the *cwnd* compared with the regular ACK frequency. The divack-*cwnd* grows 4 times more in half the time.

In Fig. 6 we observe the effect of the acceleration of the *cwnd* growth and the consequent greater throughput (as also shown in Fig. 10). In Table II, there is a summary of the total number of segment transfer per RTT, considering two different scenarios, regular TCP and *divacks* with $n = 15$. At the end of every RTT, the receiver obtains a greater data burst after the first sending with $IW = 3$. For the first RTT, in both cases TCP effectuates the 3-way-handshake, and after 250 ms

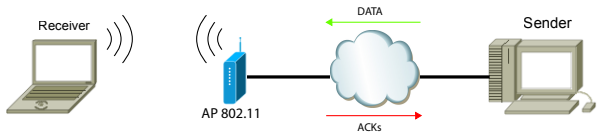

Fig. 3. Testbed for different end-to-end delay emulation.

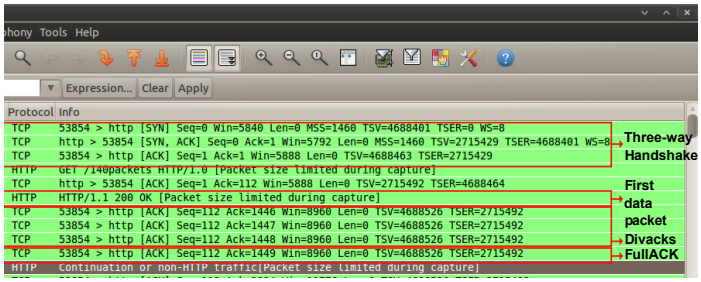

Fig. 4. Wireshark trace of TCP using 3 (three) *divacks*.

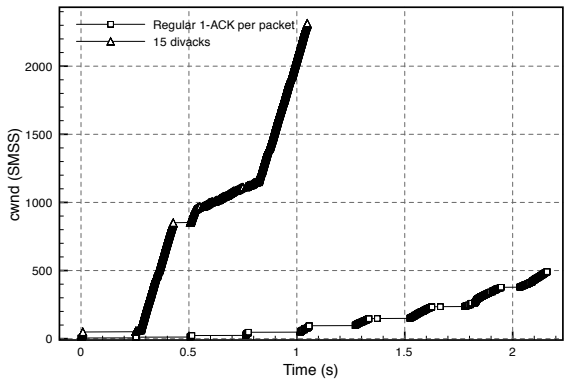

Fig. 5. *cwnd* dynamic using n=15 and n=0 (regular ACK frequency), RTT=250 ms and 990 KB file transfer.

(during the second RTT), in both cases, receivers obtain 3 segments. From this point on the difference in the amount of received packets increases as indicated by the number of *divacks*. The *divack* receiver finishes the transfer at the fourth RTT while the regular TCP needs 6 additional RTTs to finish the transfer.

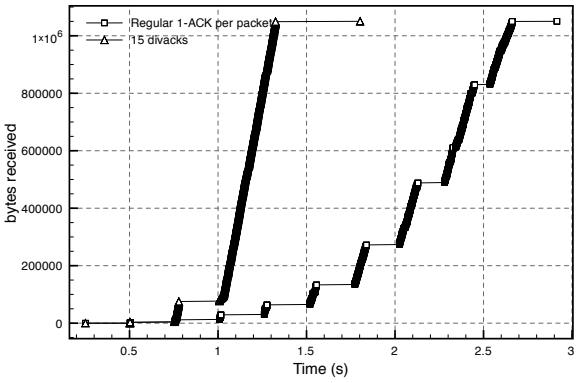

Fig. 6. Download process of 990 KB using n=15 and n=0. Both cases with 250 ms of RTT (trace at the receiver).

Note that there is a noticeable difference on the *cwnd* size showed in Fig. 5 and the amount segments in the two bursts shown in Fig. 6 for the divack flow. During the time period from 0.55 s and 0.9 s in Fig. 5 there is a slow-down on the TCP throughput for the divack flow. However, this is comparable to the throughput of the regular TCP (compare the slopes of both regular and divack flow withint the same figure). This dynamic obeys the ping-pong effect reported in [16] in which the competition for the 802.11 shared access to the media, limits the TCP throughput. There may also be other factors that can limit the TCP throughput such as the *rwnd* size, the available bandwidth or middle buffers sizes. So, with $n = 15$ we have obtained a 60% time save in the download. Observe also that the greater the end-to-end delay, the higher the increase obtained with *divacks*.

TABLE II
RECEIVED SEGMENTS PER RTT.

RTT	Regular TCP (pkts)	divacks *n=15* (pkts)
1	0 (handshaking)	0 (handshaking)
2	$IW = 3$	$IW = 3$
3	6	51
4	12	673
5	24	–
7	48	–
8	96	–
9	147	–
10	238	–
11	152	–
Total	**727**	**727**

B. Performance Evaluation

We show the TCP performance while varying the RTT, the number of *divacks* and the file size. In general, for downloading tests for several file sizes we obtained better performance for long RTTs. We can also observe that there is a maximum improvement that depends on the file size and the RTT. For example for a 198KB file (Fig. 8), the maximum improvement is around 80 ms one-way delay whereas for 990KB file, it is around 250 ms (Fig. 10). Moreover, for the case of 250 ms we have obtained savings in the download mean time of 49,3% for files of 99 KB, 58,6% for files of 198 KB (Fig. 8 and Fig. 7) and 64,9% for files of 990 KB (Fig. 10 and Fig. 9).

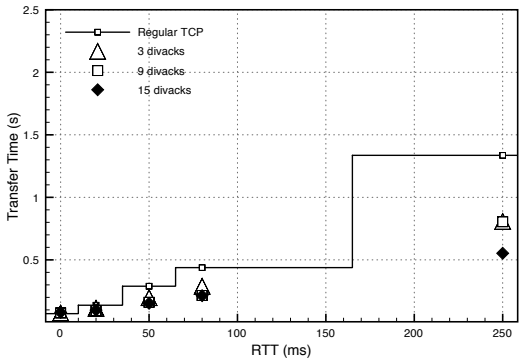

Fig. 7. Download time for 198 KB files with different delays.

As shown in Fig. 8 and Fig. 10, there is a crossing-point between the reference performance curve using 1 ACK per data packet and the group of *divacks*' curves. This corresponds to the minimum delay for experiencing a gain. Remark that this minimum delay increases for larger file sizes. This is due to the contention time spent by *divacks* and data packets (i.e., the so-called ping-pong effect [16] [2]). This phenomena has also been observed in [16] in congestion avoidance. So it justifies the interest on using *divacks* for short time-periods when there is a shared media along the path.

Finally, remark that before the minimum delay, *divacks* impairs data transmission. There are some packet level dynamics that attenuate the effect of *divacks*. For example, in Fig. 9

[2]The *divacks* and the data packet contend alternatively for the share media at the 802.11 access network.

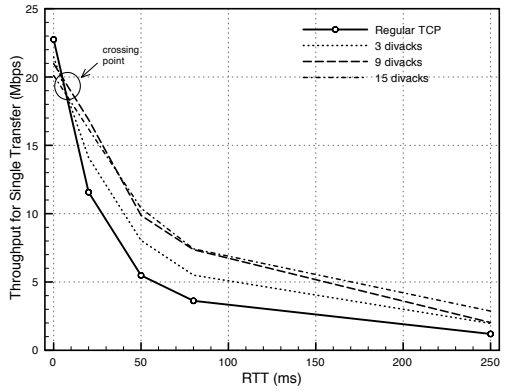

Fig. 8. Throughput comparison of file transfers of 198 KB with different delays.

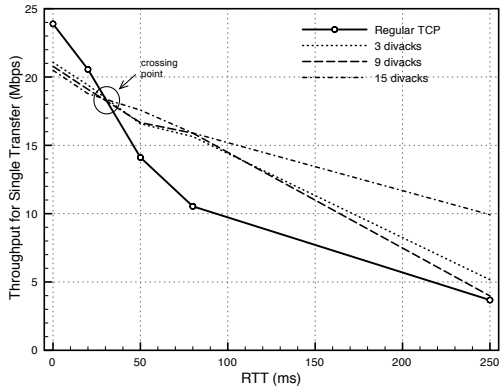

Fig. 10. Throughput comparison of file transfers of 990 KB with different delays.

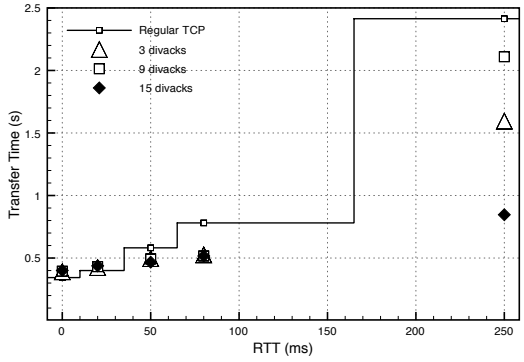

Fig. 9. Download time for 990 KB files with different delays.

using 9 *divacks* reports a smaller gain than 3 *divacks*. This is due to the effect of the contention for the media, which is intensified for certain quantities of *divacks*.

V. Conclusions

In this article we have discussed the impact of *divacks* in TCP slow start and its implementation in a Linux operating system. We have observed that *divacks* accelerate the growth of the congestion window at a rate proportional to the number of *divacks*. This property has been shown useful in scenarios such as recovery from random losses, spurious timeouts or handovers. Moreover, the acceleration on the congestion window growth exploits faster the available bandwidth. However, the effect of *divacks* is better appreciated in networks with high delay; such as satellite networks, for which data packets experience delays from 400 ms to 800 ms. Thus, we believe that we could improve the initial ramp-up of the congestion window in long-delay networks.

We have also observed that for large number of *divacks* (e.g., 15 or more) TCP performance may be limited by companion mechanisms. Large number of *divacks* provoke large bursts of data parckets, thus there are other factors that limit the performance improvement: inappropriate setting of TCP announced window (*rwnd*), short intermediate or final buffer space, or half-duplex links along the path.

As a future direction we are designing a mechanism to adapt transmission speed in terms congestion and the desired performance. We will also study the effect of divacks on the user experience when doing web surfing.

References

[1] D. Liu, M. Allman, S. Jin, and L. Wang, "Congestion control without a startup phase," *Fifth International Workshop on Protocols for FAST Long-Distance Networks (PFLDnet '07)*, 2007.

[2] S. Floyd, M. Allman, A. Jain, and P. Sarolahti, "Quick-Start for TCP and IP," Experimental RFC 4782, IETF, Jan. 2007.

[3] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby, "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations," RFC 3135, 2001.

[4] M. Welzl. (2006, March) [e2e] Since we're already learning TCP fundamentals..., Message ID: "http://www.postel.org/pipermail/end2end-interest/2006-March/005806.html".

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

[6] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP Congestion Control with a Misbehaving Receiver," *ACM SIGCOMM Computer Communications Review*, 1999.

[7] A. Arcia-Moret, "Modifying the TCP Acknowledgement Mechanism: Evaluation and Appication to Wires and Wireless Networks," Ph.D. dissertation, Institut TELECOM/TELECOM Bretagne, Rennes, France, December 2009.

[8] K. Jin, K. Kim, and J. Lee, "Spack: Rapid recovery of the tcp performance using split-ack in mobile communication environments," in *Proc. IEEE TENCON*, Cheju,South Korea, Sep. 1999, pp. 761–764.

[9] Y. Matsushita, T. Matsuda, and M. Yamamoto, "Tcp congestion control with ack-pacing for vertical handover," in *Proc. IEEE WCNC*, New Orleans, Mar. 2005, pp. 1497–1502.

[10] G. Hasegawa, M. Nakata, and H. Nakano, "Receiver-based ack splitting mechanism for tcp over wired/wireless heterogeneous networks," in *IEICE Transactions on Communications*, vol. E90-B(5), May 2007, pp. 1132–1141.

[11] *Cisco IOS Release 12.3(7)T, New Feature Documentation–Rate Based Satellite Control Protocol*, Cisco Systems, 2010.

[12] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby, "Performance enhancing proxies intended to mitigate link-related degradations," Informational RFC 3135, IETF, Jun. 2001.

[13] M. Welzl and R. Normann, "A client-side split-ACK tool for TCP Slow Start investigation," in *Computing, Networking and Communications (ICNC), 2012 International Conference on*, 30 2012-feb. 2 2012, pp. 804–808.

[14] 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 Comput. Commun. Rev.*, vol. 40, no. 3, pp. 26–33, Jun. 2010.

[15] (2011) netem. Linux Fundation. [Online]. Available in: www.linuxfoundation.org/collaborate/workgroups/networking/netem

[16] A. Arcia-Moret, D. Ros, and N. Montavont, "Auto-protection of 802.11 networks from TCP ACK division," in *CONEXT '08: Proceedings of the 2008 ACM CoNEXT Conference*. Madrid, Spain: ACM, 2008, pp. 1–2.