Accelerating short transfers in 802.11 networks

Andrés Arcia-Moret , T/ICT4D Lab, International Centre for Theoretical Physics (ICTP), aarcia_m@icpt.it
Nicolas Montavont , IMT/Telecom Bretagne, UEB, Irisa, nicolas.montavont@telecom-bretagne.eu
German Castignani , University of Luxembourg, german.castignani@uni.lu

***Abstract*—The legacy bandwidth discovery phase of TCP spends an unnecessary number of RTTs for reaching the fair share of the network. In this article we introduce a simple modification at the receiver that splits the TCP ACKs in a controlled manner. This mechanism allows to fast ramp-up the TCP congestion window. Our experiments performed in a real testbed show benefits not only in the increased data throughput but also in a non-congested uplink (Acknowledgement) path in an 802.11 access.**

I. Introduction

The available bandwidth in the Internet has been constantly growing. Yet the Transmission Control Protocol (TCP) has kept the same conservative initial phase to discover the available bandwidth. The slow start phase makes TCP to spend several RTTs to adapt the congestion window growth to the available bandwidth. In particular small to medium transfer make as many RTTs as $\log_2 N$, being N the number of segments to transfer.

The Transmission Control Protocol (TCP) depends on both, the uplink and the downlink, paths to effectively convey data. When data packets are sent, the other end sends ACKs creating a dynamic called the ACK-Clocking. The timely reception of ACKs reflects the congestion conditions of the network. And at other times, they simply reflect the nature of the media. For example, in IEEE 802.11 networks, ACKs get spaced by the distributed and random access to the media. And so do the data packets, they get sometimes delayed because ACKs need to be transfered.

Our focus is on the 802.11 networks which has reached an amazing popularity. Billions of IEEE 802.11 client and access devices are nowadays deployed world-wide. These devices implement the Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) mechanism for Medium Access Control (MAC). The shared nature of the MAC mechanism may create an artificial limit to increase the data throughput because of the regular TCP-ACK policy [1]. On the other hand, the initial fast ramp-up of the TCP congestion window has been the subject of several works, trying even to completely eliminate the initial phase of TCP [2].

In this paper, we propose a novel end-to-end approach to fast ramp-up a TCP connection by introducing a modification to the TCP ACK-clocking algorithm from the receiver. By sending an appropriate number of control packets, called *divacks* (regular valid ACKs renamed for the purpose of the work as in [1], [3]), during an initial time window of the connection, we obtain an increased initial rate that favours short transfers (living in slow start) and the recovery of the connections after a disconnection or a handover in an 802.11 mobile scenario.

The reminder of this paper is organized as follows. In Section II we introduce the related work. The *divacks* mechanism and its configuration parameters is proposed in Section III. Then in Section IV we evaluate the proposed mechanism in a Linux-based implementation of *divacks* over different scenarios. In Section V we analyse the potential of *divacks* for fast recovery after handovers. Finally, in Section VI, we conclude the paper.

II. Previous Work

There has been a recent interest on accelerating the initial ramp-up of the transfer for short lived flows, and even to eliminate altogether the initial conservative bandwidth discovery on TCP. In our particular interest, the initial ramp-up may also be useful after a handover in 802.11 networks. In this case, a faster ramp-up than legacy TCP, is intended to recover the previous data rate.

As suggested by Liu et al. [2], there are three broad categories classifying the different approaches for bandwidth discovery. First, methods for predicting the available bandwidth such as the well-known packet pair technique, in which TCP Westwood relies on [4]. Second, mechanisms for sharing the discovered capacity information among connections. For example, by sharing the discovered capacity between parallel connections living within a common path [5]. Third, mechanisms that relies on the network for obtaining a precise information about the availability of buffer space to send large bursts of data [6].

Alternatively, we approach the problem in a different perspective. There has been a recent evaluation of the performance of a finer ack-clocking control called *divacks*. Although the original report by Savage et al. [7] described it as a potential threat as a possible security hole, recent evaluations show that *divacks* effect is not straight forward to predict [1], [3]. The increasing of the ACK flow if well tuned, can derive the same benefits as other fast-start mechanisms with the additional gain of pacing the data packets controlled from the clients [8].

III. ACK rate adaptation

We propose to use the ACK rate adaptation to enhance wireless station performance. By using multiple ACK (called *divack*) per TCP segment a mobile station can strongly reduce the time to discover the available bandwidth, and accelerate short transfers over a wireless link. This mechanism is used during the slow start period, i.e., when a mobile station is initiating a file download, or when it performs a handover between two access points while downloading a file. In the

case of a handover, the TCP connection is interrupted and once the mobile station has connected to the new point of attachment, it uses slow start again to discover the available bandwidth. *Divack* will allow reaching the new capacity at the new point of attachment as fast as possible.

In order to implement the *divack* mechanism and study the trade-off between an appropriate *divack* rate and the admission control imposed by the CSMA/CA, we modified the regular ACK sending algorithm. These modifications only concern the kernel that will be deployed in the client computer. We have added just few lines of code to control the total number of ACKs sent during the earlier RTTs of the slow start.

A. Fast ramp-up algorithms (using divacks)

We used an Ubuntu distribution (10.04) with the kernel 2.6.32.21 for our implementation. The legacy TCP, that we used as a reference in the performance evaluation, proposes a regular ACK policy which does not correspond exactly to acknowledging every other packet, neither to one ACK-per-packet, it falls in between. So an approximate number of 1.5 ACKs per data packet are regularly sent from the chosen Linux kernel. This may be induced by the delayed ACK timer being triggered when in presence of long RTTs.

In order to control the low-throughput effect produced by a high number of *divacks* [9], [10], [3], we reduce the uplink traffic by limiting the total number of *divacks* in the Controlled *divacks* algorithm. This reduction is achieved by sending a fixed amount of *divacks* at the beginning of the transfer. Two parameters were defined. *tcp_divack_count* keeps track of the number of *divacks* already sent and *tcp_divack_max_count* retains the maximum number of *divacks*. By sending a limited number of *divacks* not only is the available bandwidth rapidly discovered but also it liberates the wireless link to be competed by *divacks*, allowing the data packets to be transferred with a minimal uplink congestion.

IV. Experimentation

A. Experimental Platform

In order to evaluate the impact of the proposed *divacks* mechanisms (i.e., Brute-Force and Controlled), we have set up an experimental platform as depicted in Fig. 1. This platform consists of a (wired) TCP server *A*, a *divacks*-enabled client *C*, the network emulator *netem*, named *B* and the IEEE 802.11 access point (AP). *A* is connected to *B* via an Ethernet cable, and *B* is also connected to the AP via a second Ethernet cable. The client connects via a wireless link to the AP. In this testbed, the client *C* will be downloading data form *A*, and therefore, it will acknowledge every packet received with several *divacks*, speeding up the transfer. In our proposed experimental platform the bottleneck of the transmission is in the wireless link, that reaches a maximum throughput of $19.5\,Mbps$ while the wired connexion offers a maximum throughput of 100 Mbps.

The ramp-up algorithms using *divacks* were implemented in the client. For the server as it has been recently reported, a significant share of the well-known servers react favourable to *divacks* [11].

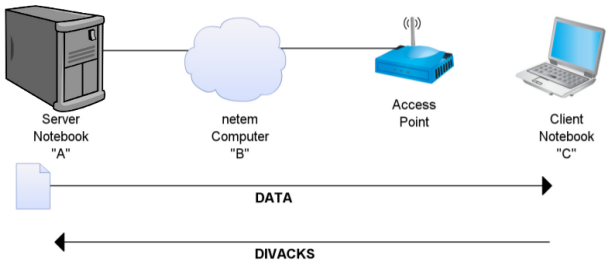

Fig. 1. Testbed platform

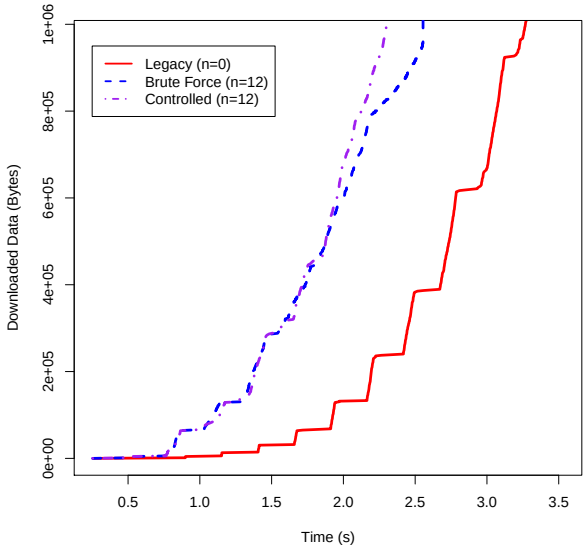

Fig. 2. Controlled method shows an improved performance

B. Results

In Fig. 2, we show the effect of the Controlled *divacks* algorithm in terms of the throughput. In particular, we consider a number of *divacks* $n = 12$ (between 7 and 14) and we compare legacy TCP ($n = 0$) with the Brute-Force and the Controlled *divack* algorithm. We observed that the Controlled algorithm corrects the misbehaviour showed by the Brute-Force algorithm when the number of divacks exceeds a threslhold ($n = 12$ in our particular case). Conversely, the Controlled algorithm allows accelerating the congestion window growth without exhibiting the known null-effect when using divacks for long time periods [1]. Morover, in the short-term there is a relevant throughput improvement compared to the legacy TCP implementation.

Fig. 3 shows the uplink traffic in packets (ACKs) per time unit. Observe that the controlled method of *divacks* noticeable decrease the uplink traffic and better accelerate the increasing of the congestion window. Another implicit benefit is that, compared to the legacy TCP (the red curve in Fig. 3), the Controlled *divacks* algorithm can use a slightly large amount of *divacks* at the beginning of the connection but at the end, the connection finishes first than Brute-Force and Legacy TCP, releasing the channel sooner.

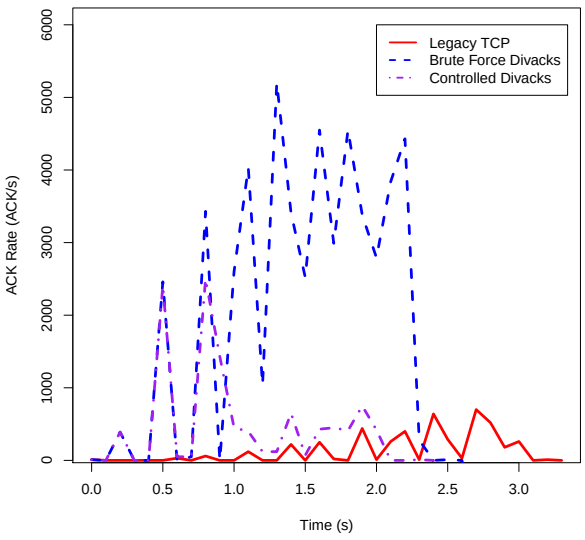

Fig. 3. ACK rate for different approaches

C. The role of the reception window

The client reception buffer plays an important role in the performance of the *divacks* mechanism. This buffer is allocated for each TCP connection and stores all incoming packets before delivering to the client. So, if the buffer size is increased, the throughput of the transmission is also increased due to a larger announced window, performing more relaxed flow control.

We have also observed that the data packet bursting oscillates when there is a need for a bigger buffer value. When the reception window saturates (close to applying a flow control), there is an increase of the announced window. This is announcing in turn the scaling of the reception buffer.

V. Use of divacks for dealing with handovers

In order to evaluate the performance of *divacks* in a handover scenario, we have set up a testbed in which the mobile device changes the point of attachment and recovers the previously-established TCP connection. We have considered three experiments: legacy TCP ($n = 0$, *divacks* disabled), brute force *divacks* algorithm ($n = 12$) and the controlled *divacks* algorithm ($n = 12$).

In Table I we show the throughput obtained before and after the handover was executed. As expected, when the *divacks* mechanism is executed in a controlled fashion, it overtakes the default TCP behaviour not only before a handover occurs ($0.86\,Mbps$ versus $1.38\,Mbps$) but also afterwards ($0.23\,Mbps$ versus $0.90\,Mbps$).

Configuration	**Throughput before** handover ($Mbps$)	**Throughput after** ($Mbps$)
Legacy (n=0)	0.86	0.23
Brute Force (n=12)	1.34	0.18
Controlled (n=12)	1.38	0.90

TABLE I
Measured throughput in a handover scenario

VI. Conclusions

The discovery of the available bandwidth in a new connection is handled by the slow start TCP algorithm. Even if this process allows the congestion window to grow in an exponential fashion, this is not enough after a handover has taken place. Thus, we proposed a couple of algorithms using *divacks*, i.e., sending multiple ACKs when a data packet has been received. Moreover, we showed the gains obtained by controlling the number of *divacks*.

In emulated handovers we showed that the *divacks* controlled mechanism not only perform better than the TCP default mechanism, but also after the transmission is re-established, with a throughput up to four times larger. These results make even clearer the need for limiting the number of *divacks* sent per data packet to avoid the uplink congestion when the Brute Force variation is executed. Finally, there is a *divacks* limit that we empirically discovered for which the maximum throughput is achieved. Over that limit the ACK traffic impairs the TCP throughput.

References

[1] 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.

[2] 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.

[3] A. Arcia-Moret, O. Díaz, and N. Montavont, "A Tunable Slow Start for TCP," in *IEEE 4th Global Information Infrastructure and Networking Symposium (GIIS)*, December 2012.

[4] L. A. Grieco and S. Mascolo, "End-to-end bandwidth estimation for congestion control in packet networks," in *Proceedings of QoS-IP 2003*, no. 2601. Milano: Springer, Feb. 2003, pp. 645–658.

[5] H. Balakrishnan, H. S. Rahul, and S. Seshan, "An integrated congestion management architecture for internet hosts," in *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM '99. New York, NY, USA: ACM, 1999, pp. 175–187. [Online]. Available in: http://doi.acm.org/10.1145/316188.316220

[6] P. Sarolahti, M. Allman, and S. Floyd, "Determining an appropriate sending rate over an underutilized network path," *Computer Networks*, vol. 51, no. 7, pp. 1815–1832, 2007.

[7] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP congestion control with a misbehaving receiver," vol. 29, no. 5, Oct. 1999. [Online]. Available in: http://www.acm.org/sigcomm/ccr/archive/1999/oct99/savage.pdf

[8] J. Aweya, M. Ouellette, and D. Y. Montuno, "A self-regulating TCP acknowledgement (ACK) pacing scheme," *International Journal of Network Management*, vol. 12, no. 3, pp. 145–163, 2002.

[9] 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.

[10] A. Arcia-Moret, N. Montavont, J.-M. Bonnin, and D. Ros, "TCP ACK Division Revisited," in *4th Ibero-american conference of electrical engineering. CIBELEC*, 2010.

[11] 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.