DeSRTO: an effective algorithm for SRTO detection in TCP connections

Antonio Barbuzzi, Gennaro Boggia, and Luigi Alfredo Grieco
email:{a.barbuzzi; g.boggia; a.grieco}@poliba.it

DEE - Politecnico di Bari - V. Orabona, 4 - 70125, Bari, Italy
Ph. +39 080 5963301; Fax +39 080 5963410

Abstract. Spurious Retransmission Timeouts in TCP connections have been extensively studied in the scientific literature, particularly for their relevance in cellular mobile networks. At the present, while several algorithms have been conceived to identify them during the lifetime of a TCP connection (e.g., Forward-RTO or Eifel), there is not any tool able to accomplish the task with high accuracy by processing off-line traces. The only off-line existing tool is designed to analyze a great amount of traces taken from a single point of observation. In order to achieve a higher accuracy, this paper proposes a new algorithm and a tool able to identify Spurious Retransmission Timeouts in a TCP connection, using the dumps of each peer of the connection. The main strengths of the approach are the great accuracy and the absence of assumptions on the characteristics of TCP protocol. In fact, except for rare cases that are not classifiable with absolute certainty at all, the algorithm shows no ambiguous nor erroneous detections. Moreover, the tool is also able to deal with reordering, small windows, and other cases where competitors fail. Our aim is to provide to the community a very reliable tool to: (i) test the working behavior of cellular wireless networks, which are more prone to Spurious Retransmission Timeouts with respect to other technologies; (ii) validate run-time Spurious Retransmission Timeout detection algorithms.

1 Introduction

TCP congestion control [1,2,3] is fundamental to ensure Internet stability. Its main rationale is to control the number of in-flight segments in a connection (i.e., segments sent, but not yet acknowledged) using a sliding window mechanism. In particular, TCP sets an upper bound on the number of in-flights segments via the congestion window (*cwnd*) variable. As well known, the value of *cwnd* is progressively increased over the time to discover new available bandwidth, until a congestion episode happens, i.e., 3 Duplicated Acknowledgements (DUPACKs) are received or a Retransmission Timeout (RTO) expires. After a congestion episode, *cwnd* is suddenly shrinked to avoid a network collapse. TCP congestion control has demonstrated its robustness and effectiveness over the last two decades, especially in wired networks.

Recently, the literature has questioned the effectiveness of the basic RTO mechanism originally proposed in [1] and refined in [4]. Such a mechanism measures the Smoothed Round Trip Time (SRTT) and its standard deviation (DEV) and then sets $RTO = SRTT + 4DEV$. The rationale is that, if the RTT exhibits a stationary behavior, RTO can be considered as a safe upper bound for the RTT. Unfortunately, in cellular mobile networks this is not ever true. In fact, delay spikes due to retransmissions, fading, and handover can trigger Spurious Retransmission Timeouts (SRTOs) that lead to unnecessary segment retransmissions and useless reductions of the transmission rate [5,6].

Despite of the importance of SRTOs, there is not any tool able to accomplish the task of properly identifying them with high accuracy by processing off-line traces. The only off-line existing tool is designed to analyze a great amount of traces taken from a single point of observation [7]. To bridge this gap, this paper proposes a new algorithm and a tool able to Detect SRTOs (which will be referred to as *DeSRTO*) in a TCP connection, using the dumps of each peer of the connection. The main strengths of the approach are the great accuracy and the absence of assumptions on the characteristics of TCP protocol. In fact, except for rare cases that are not classifiable with absolute certainty at all as SRTO, the algorithm shows no ambiguous nor erroneous detections. Moreover, the tool is also able to deal with reordering, small windows, and other cases where competitors fail. Our aim is to provide to the community a very reliable tool to: (i) test the working behavior of cellular wireless networks, which are more prone to Spurious Retransmission Timeouts with respect to other technologies (but it is well suited also for any other kind of networks with traffic flows using TCP); (ii) validate run-time Spurious Retransmission Timeout detection algorithms. To provide a preliminary demonstration of the capabilities of *DeSRTO*, some results derived by processing real traffic traces collected over a real 3G network have been reported. Moreover, to test its effectiveness, a comparison with the Vacirca detection tool [7] is reported.

The rest of the paper is organized as follows. In Sec. 2 a summary of related works is reported. Sec. 3 describes our algorithm, reporting also some examples on its behavior. Sec. 4 shows the experimental results. Finally, conclusions are drawn in Sec. 5.

2 Related Works

So far, research on SRTOs has produced schemes that can be classified in to two families: (i) real-time detection schemes for TCP stacks; (ii) off-line processing tools for SRTO identification. The Eifel Detection [5], the F-RTO [6], and the DSACK [8] schemes belong to the first family. Instead, the tool proposed in [7] and the DeSRTO algorithm herein presented belong to the second family.

The goal of the Eifel detection algorithm [5] is to avoid the go-back-N retransmits that usually follows a spurious timeout. It exploits the information provided by the TCP Timestamps option in order to distinguish if coming ACKs are related to a retransmitted segment or not.

The algorithm described in [8] exploits DSACK (Duplicate Selective ACK) informations to identify SRTOs. DSACK is a TCP extension used to report the receipt of duplicate segments to the sender. The receipt of a duplicate segment implies either the packet is replicated by the network, or both the original and the retransmitted packet arrive at the receiver. If all retransmitted segments are acknowledged and recognized as duplicated using DSACK information, the algorithm can conclude that all retransmissions in the previous window of data were spurious and no loss occurred.

The F-RTO algorithm [6] modifies the TCP behavior in response to RTOs: in general terms, when a RTO expires, F-RTO retransmits the first unacknowledged segment and waits for subsequent ACKs. If it receives an acknowledgment for a segment that was not retransmitted due to the timeout, the F-RTO algorithm declares a spurious timeout.

Possible responses to a spurious timeout are specified in [9], namely the Eifel Response Algorithm. Basically, three actions are specified: (i) the TCP sender sends new data instead of retransmitting further segments; (ii) the TCP stack tries to reverse the congestion control state prior to the SRTO; (iii) the RTO estimator is re-initialized by taking in account the round-trip time spike that caused the SRTO (a slightly different approach has been also proposed in [10]).

To the best of our knowledge, till now, the only SRTO detection algorithm aimed at the analysis of collected TCP traces is the one proposed by Vacirca et al. in [7]. This algorithm is conceived to process a large amount of traces from different TCP connections. Such traces are collected by a single monitoring interface, placed in the middle of a path traversed by many connections. The design philosophy of the algorithm targets strict constraints on execution speed and simplicity. Anyway, being the algorithm based on a monitoring point placed in the middle of the network, it cannot exploit fundamental information available on dumps collected at connection endpoints that could improve estimation accuracy. The rationale of the algorithm of Vacirca is to analyze ACKs stream to identify SRTOs. In case of a Normal RTO (NRTO), a loss involves the transmission of duplicate ACKs by the TCP data receiver, that indicates the presence of a hole in the TCP data stream. On the contrary, in case of a SRTO, no duplicate ACKs are expected, since there is no loss. It is well known that this algorithm does not work properly in the following conditions: (i) packet loss before the monitoring interface; (ii) presence of packet reordering; (iii) small windows; (iv) no segment is sent between the original transmission that caused the RTO and its first retransmission; (v) loss of all the segments transmitted between the original transmission that caused the RTO and its first retransmission; (vi) loss of all ACKs.

Some of these cases lead to erroneous conclusions, while others lead to ambiguity. Furthermore, the absence of packet reordering is a fundamental hypothesis for the validity of the Vacirca detection scheme. Our algorithm is instead aimed to the analysis of a single TCP flow, without making any assumption on the traffic characteristic.

3 Spurious Timeout Identification Algorithm: DeSRTO

In this section, we will examine closely the SRTO concept, reporting some example of SRTOs. Then, we will explain our algorithm.

3.1 What is a SRTO?

As well known, every time a data packet is sent, TCP starts a timer and waits, till its expiration, for a feedback indicating the delivery of the segment. The length of this timer is just the retransmission timeout, i.e., the RTO. The expiration of a RTO timeout is interpreted by the TCP as an indication of packet losses. The computation of RTO value is specified in [4]; it is based on the estimated RTT and its variance. The proper setting of the RTO is a tricky operation: if it is too long, TCP would waste a lot of time before it realizes that a segment is lost; if it is too short, unnecessary segments would be retransmitted, with a useless waste of bandwidth.

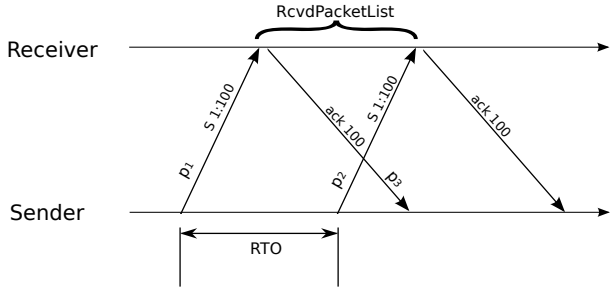
As already stated, it has been shown that the RTO estimation can be unreliable in some cases, especially when connections cross through cellular networks. Therefore, retransmission procedure can be unnecessarily triggered, even in absence of packet loss. The RFC 3522 names these RTOs as spurious (i.e., there is a SRTO), and defines them as follow: *a timeout is considered spurious if it would have been avoided had the sender waited longer for an acknowledgment to arrive.*

Let us clarify the SRTO concept through an example. In Fig. 1(a), it is reported a borderline case, representing the simplest possible case of SRTO. The segment in the packet p_1 was sent and received successfully, like the relative acknowledgment, i.e., the packet p_3. However, p_3 arrived after the RTO expiration; therefore, the sender uselessly retransmitted in packet p_2 the data contained in the payload of packet p_1. Note that if the sender had waited longer, it would have received the acknowledgment contained in packet p_3. Thus, as stated by the definition, here we are in presence of a SRTO.

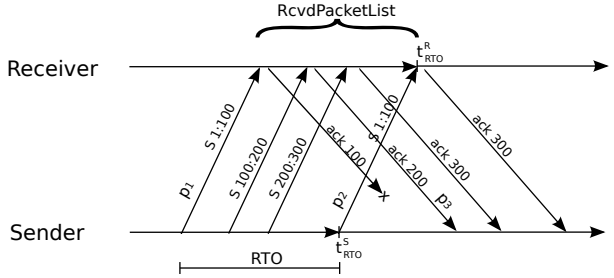
The presented example considers a very simple scenario with a small TCP transmission window. A more complex case is reported in Fig. 1(b): the first ACK segment (the one with *ack number* 100) is lost, but since ACKs are cumulative, data contained in packet p_1 can be acknowledged by any of the subsequent ACKs. Therefore, the correct application of SRTO definition requires to check that at least one ACK (among the ones sent between the reception of p_1 and the reception of the retransmitted segment) is delivered successfully to the sender.

3.2 DeSRTO

DeSRTO is an algorithm we developed to detect Spurious RTOs. The main difference between its counterparts is that it uses both TCP peer dumps, a feature that enhances the knowledge on the network behavior during a RTO event. *DeSRTO* discriminates SRTOs from NRTOs according to the RFC 3522, by



(a) A simple example of Spurious RTO.



(b) A more complicated example of Spurious RTO.

Fig. 1. Spurious RTO examples.

reconstructing the journey of the TCP segments involved into the RTO management.

For a specific RTO event, the algorithm needs to associate a packet containing a TCP segment with the packet(s) containing the relative acknowledgment that was(were) sent in reply to the segment. Note that the ACK number refers to a flow of data and not to a specific received packet. Instead, an IP datagram containing a TCP ACK is triggered by a specific TCP segment and thus can be associated to it. Hence, in the sequel, we will adopt the expression "packet A that acknowledges packet B" referring to the specific IP datagram A that contains a TCP ACK related to the reception of packet B. It is important to highlight that the algorithm needs unambiguous couple of packets on the sender and receiver dumps. Since we cannot rely only on the sequence number or on the ACK number of TCP segments, we make use of the identification field in the IP header, that, according to [11], is used to distinguish the fragments of one datagram from those of another.

Let us clarify the behavior of *DeSRTO* applying it to the SRTO cases described in the previous section. We will start from the simplest scenario in Fig. 1(a). In this instance, *DeSRTO* would proceed according to the following steps:

1. Identify the packet that caused the RTO (p_1) on the sender dump.
2. Find packet p_1 in the receiver dump.
3. If p_1 is lost, the RTO is declared *Normal*.

4. Otherwise, find the ACK "associated" to p_1 on the receiver dump (namely p_3). This packet should exist because it is transmitted by the receiver.
5. Find the packet p_3 on the sender dump.
6. If p_3 is not present in the sender dump (that is, it was lost), the RTO is declared *Normal*, otherwise it is declared as *Spurious*.

Now, let us analyze the more complicated case in Fig. 1(b). Of course, the loss of only the first ACK is not enough to come to a conclusion. Therefore, the algorithm would check if at least one ACK (sent between the reception of p_1 and the instant t^R_{RTO} of the reception of the retransmitted packet) was delivered correctly to the sender.

The aim of our algorithm is to try to detect all types of SRTOs. Thus, through a deeper analysis of the methodology needed to distinguish all the possible RTO types, we realized that the definition used by RFC 3522 does not allow the practical identification of all possible RTO episodes. In fact, according to the definition, in order to understand what would happened if we had "waited longer", it is possible to check what the TCP data sender would have done till the RTO event t^S_{RTO}, and what the TCP data receiver would have done till the reception of the first retransmission triggered by the RTO event, t^R_{RTO}. Everything happens after these two instants depends also on how the RTO event has been handled. From t^R_{RTO} on, the *storyline* diverges, and the check of the "what if" scenarios can result in significantly different outcomes, since the consequences of the RTO management cannot be really undo: it influences the following events in an unforeseeable way.

We can check what would have happened if we "waited longer" with certainty as long as we do not need to undo the TCP stack management of the RTO event. To deal also with these uncertain cases, we define the concept of *Butterfly-RTO* as follows: *a Butterfly-RTO is a RTO whose identification as SRTO or NRTO would require the check of packets at the receiver side after the instant* t^R_{RTO}.

In order to highlight the complexity involved in dealing with a Butterfly-RTO, we can consider the RTO example in Fig. 2. Packet p_1 is transmitted, but the RTO timer expires before the reception of any ACKs related to it (the ACK relative to packet p_1 is lost). Packet p_2 is the retransmitted packet, with the same sequence number interval as p_1. Note that the reception of packet p_2 marks the instant t^R_{RTO}, i.e., the instant of the reception by the sender of the retransmitted packet. The following packet (the one with sequence numbers $100 : 200$) is lost whereas packet p_B (the one with sequence numbers $200 : 300$) is delivered correctly. But, due to reordering, it arrives after the instant t^R_{RTO} and, consequently, p_3, the packet that acknowledges p_B, is sent after t^R_{RTO}. Apparently the case depicted in Fig. 2 is a Spurious RTO, because, if we had waited for p_3 to arrive, we would not have sent the retransmission p_2.

Examining more carefully the situation, we should note that the *story* of packet p_B could also be different if packet p_2 would have not been never transmitted. In other terms, the *storylines* of p_2 and p_B are coupled and there is no way to undo the effects of RTO management with absolute certainty. To further stress the concept, we remark that, if the TCP delayed ACK option is enabled,

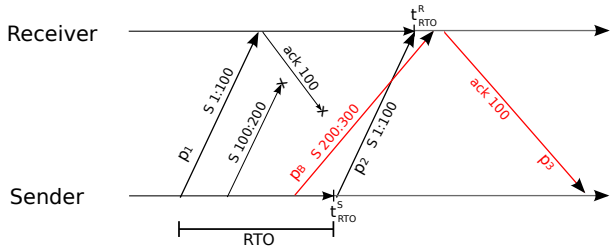


Fig. 2. An example of Butterfly RTO.

the time instant at which the ACK p_3 is transmitted depends also on p_2 and not only on p_B. Finally, TCP implementations running at different hosts often implements RFC specifications in slightly different ways, thus making even worse the problem.

Actually, Butterfly-RTOs are negligible in normal network conditions, since reordering is rare and the probability of joint RTO and packet reordering events (needed for the Butterfly-RTO occurrence) is very low if the RTO and reordering events are uncorrelated. Anyway, note that possible network malfunctions during handover or channel management (due, for example, to bugs or incorrect settings in network devices) could systematically cause the scenario happening.

To conclude this discussion, we describe how our algorithm tries to solve and/or classify the uncertain cases. If no reordering happens, *DeSRTO* searches for all ACKs that acknowledge p_1 and have been transmitted within t^R_{RTO}. If at least one of these ACKs has been received by the sender, then the RTO is classified as *spurious*. Otherwise, the RTO is *normal*.

If a reordering happens and none of the ACKs transmitted within t^R_{RTO} has been received, then the RTO is classified as *Butterfly*.

In Fig. 3, it is shown a case taken from [7], where the detection with the Vacirca tool would fails, as stated in [7] itself. The dashed line indicates the point where the monitoring interface used by the Vacirca algorithm is placed, and the sequence of packet passing through the line is the one seen by such a detection tool. In this case, the network experiences packet reordering, specifically packet with sequence numbers $100 : 200$ arrives after packet with sequence numbers $200 : 300$. As reported in [7], the Vacirca tool will erroneously classify the SRTO as normal. In fact, the algorithm will see an ACK (the packet with ACK number 300) after the retransmission of p_1 and p_3; therefore, it deduces that packet p_3 fills the hole in the data sequence. Our tool, instead, is not affected by packet reordering. In fact, by following the same steps (1 - 6) before outlined to explain the simpler SRTO example in Fig. 1(a), it is straightforward to show that *DeSRTO* is able to classify the RTO as spurious.

It is worth to note that the four examples reported in Figs. 1-3 have been pictured to provide an idea of the extreme complexity associated to SRTO detection. More SRTO scenarios can happen, depending on reordering, loss of data

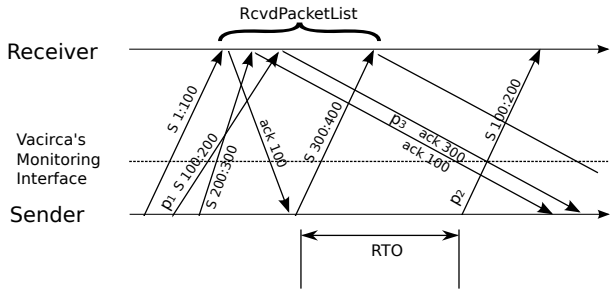


Fig. 3. Example of SRTO with reordering: the Vacirca tool fails considering it as normal.

packet, and so on. The details on *DeSRTO* behavior in a general setting are presented in the pseudocode description of the algorithm (see Sec. 3.3).

3.3 The algorithm in detail: the pseudocode

Hereafter, we discuss the *DeSRTO* pseudocode reported in Algotithm 1. We continue to refer to Fig. 1(b) all over the text. As general consideration, we recall that to unambiguously couple packets on the sender and receiver dumps, we use the identification field of the IP header [11]. To simplify the notation, we use the following conventional definitions:

SndDump is the dump of the TCP flow at the sender side;
RcvDump is the dump of the TCP flow at the receiver side;
RTOList is the list of all the generated RTOs.

Below, in the step-by-step description of the algorithm, numbers at the beginning of each line refer to the line numbers in the pseudocode of Alg. 1. This description will give further insight in the comprehension of *DeSRTO* behavior.

1:3 Initially, *RTOList* contains a list of RTOs, with timestamps of each RTO and the sequence number of the TCP segment that caused it. From these information, we can find in the *SndDump* the packet p_1 (see fig. 1(b)) that caused the RTO. The second step requires to find p_1 also on the receiver side in *RcvDump*.

4:5 If packet p_1 is lost, that is, if it is not present in the *RcvDump*, the RTO is declared *Normal*. Note that TCP "fast retransmit" is comprised in this case.

6:13 It is found the first retransmission of the segment encapsulated in the p_1 datagram, straight after the RTO event, namely p_2. The packet p_2 is also found at the receiver side. If p_2 is not present on the receiver dump, i.e., it has been lost, the algorithm looks for the first packet transmitted after p_2 that successfully arrives at the receiver.

14:21 In *SndDump*, all the packets sent between the transmission of p_1 and the transmission of p_2 are found. They are stored in the list *SentPktsList*. Then, *DeSRTO* stores in the list *RcvdPktsList* all the packets in *SentPktsList* that have been successfully received (this step requires an inspection of *RcvDump*). The first and the last packets in *RcvdPktsList* are called p_m and p_M, respectively. We will refer to t_m as the reception instant of p_m and to t_M as the transmission time instant of the first ACK transmitted after the reception of p_M. Note that p_m and p_M could be different from p_1 and p_2, respectively, in case of packet reordering.

22:27 Search on *RcvDump*, in the time interval $[t_m, t_M]$, all the ACKs that acknowledged p_1. Found ACKs are saved in the list *AckList*, according to their transmission order.

28:43 For each ACK saved in *AckList*, the algorithm checks if it was successfully received by the sender. The search stops as soon as the first ACK successfully received is found. If a received ACK is found, two cases are considered:

1. the ACK was sent before t^R_{RTO}.
2. the ACK was sent after t^R_{RTO}

In the first case, the sender received an ACK for p_1 after the RTO expiration, therefore the RTO is declared *Spurious*.

In the second case, we have a *Butterfly-RTO*.

Note that we account also for reordering of the ACKs on the reverse path; therefore, the check on the ACK packets is done in chronological order, starting from the first sent ACK packet till the last one.

44:48 If none of ACKs in *AckList* has been received by the sender, i.e., an entire window of ACKs is lost, two cases are considered:

- if the the greatest timestamp of packets in *AckList* is smaller than t^R_{RTO} (i.e., t_{p2}), the RTO is declared *Normal*,
- otherwise it is declared *Butterfly*.

3.4 Implementation Details

To verify the effectiveness of our algorithm, *DeSRTO* has been written in python programming language. The realized tool implements exactly the pseudocode described above (the actual version of the tool is v1.0-beta and it is freely available at `svn://telematics.poliba.it/desrto/tags/desrto_v1`).

The *DeSRTO* tool takes in input a list of RTOs (*RTOList* in the pseudocode), with timestamp and sequence number and the dumps related to the TCP connection of two peers. The list of RTOs is generated using a Linux kernel patch, included in the repository, that simply logs the sequence numbers and the timestamps of each RTO. Of course, other methods can be used to have a list of RTOs, such as a simple check of the presence of duplicate transmission without 3-DUPACK. We have planned to implement it as an option in the near future.

The dumps of each peer can be truncated in order to discard the TCP payload. Of course, *DeSRTO* requires that no packets are discarded by the kernel. In fact, if packets we look for are not found, the analysis would be wrong. An

Algorithm 1 Pseudocode of DeSRTO

```
for each rto in RTOList do
    FIND the packet p1 that causes the rto in SndDump
    FIND packet p1 in RcvDump
    if p1 is Lost then
        rto ← NRTO
    else
        FIND packet p2 in SndDump, the first retransmission of packet p1
        FIND p2 in RcvDump
        while p2 is Lost do
            FIND tmp the first packet transmitted after p2 in SndDump
            p2 ← tmp
            FIND p2 in RcvDump
        end while
        SET t_retr TO the timestamp of p2 on RcvDump
        GET all sent packets between p1 and p2 in SndDump, including p1 and not
        p2
        FIND the corresponding received packets in RcvDump
        STORE founded packets in RcvdPktsList
        SET p_m TO the first packet in RcvdPktsList in chronological order
        SET t_m TO the timestamp of p_m
        SET p_M TO the last packet in RcvdPktsList in chronological order
        SET t_M TO the timestamp of the first ACK transmitted after p_M
        for EACH sent packet p_a in RcvDump FROM t_m TO t_M do
            if p_a acknowledges p1 then
                STORE p_a IN AckList
            end if
        end for
        SET t_max to the greatest timestamp of the packets in AckList
        SET t_p2 TO the timestamp of p2 on RcvDump
        SET ACK_FOUND TO False
        for EACH ack packet a in AckList do
            if ACK_FOUND = False then
                FIND packet p_a in SndDump
                if p_a is not LOST then
                    SET ACK_FOUND TO True
                    SET t TO the timestamp of p_a on RcvDump
                    if t_3 > t_p2 then
                        rto ← ButterflyRTO
                    else
                        rto ← SRTO
                    end if
                end if
            end if
        end for
        if ACK_FOUND = False and t_max > t_p2 then
            rto ← ButterflyRTO
        else if ACK_FOUND = False then
            rto ← NRTO
        end if
    end if
end for
```

option to deal with flows that go through a NAT has been implemented. The aim of each option is to analyze most of the cases in background, without the presence of an operator.

4 Experimental Results

To test the effectiveness of the tool and its performance, we have considered a series of TCP flows generated using *iperf*, a commonly used network testing tool that can create TCP data streams (http://dast.nlanr.net/iperf/), in a real 3.5G cellular network (a UMTS network with the HSPA protocol) with concurrent real traffic . The testbed is presented in Fig. 4. There are two machine equipped with a Linux kernel patched with the Web100 patch (http://www.web100.org/), a tool that implements TCP instruments, defined in [12], able to log internal TCP kernel status, and with a kernel patch we developed that logs all the RTO events (see Sec. 3.4). The developed patch has been tested comparing the reported timeouts against Web100 output.

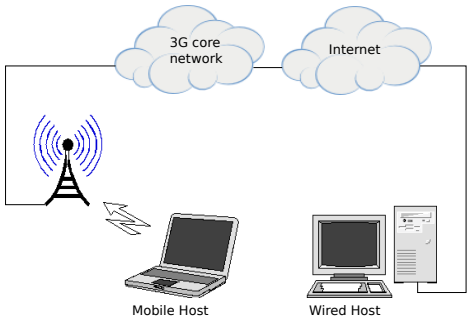


Fig. 4. Experimental testbed.

The first PC is connected to the Internet through a wired connection, while the second one is equipped with a UMTS (3.5G) card and it is connected to the cellular network. We have generated a series of one hour greedy long flows between the two machines using iperf. We have conducted several experiments, with flows originating from both the machines, in order to test either the directions of the connection. The average transfer rate was 791 kbits/sec in download and 279 kbits/sec in upload. No experiments experienced packet reordering.

In the download case, where the UMTS equipped machine receives data, the number of detected SRTOs is negligible also due to the low number of RTOs (actually most RTOs are due to retransmission of SYN packets); whereas in the upload case, the SRTOs are more common, even if not prevalent. This behavior was expected due to the asymmetry between uplink and downlink in cellular

networks (downlink usually provides higher bandwidth, higher reliability, and smaller delays with respect to uplink [13]).

Tabs. 1 and 2 show the number of NRTOs and SRTOs detected by *DeSRTO* and by the Vacirca tool in the upload and in the download cases, respectively. Note that there are no Butterfly RTOs, since no reordering was experienced in performed experiments.

	DeSRTO Results			Vacirca tool results				
N.	SRTO	NRTO	% SRTO	SRTO	NRTO	Ambiguous	% SRTO	% Ambiguous
1	3	23	13,0%	5	535	4	0,9%	0,7%
2	5	27	18,5%	5	536	7	0,9%	1,3%
3	5	231	2,2%	15	711	31	2,0%	4,1%
4	4	305	1,3%	23	784	43	2,7%	5,1%
5	7	151	4,6%	24	637	19	3,5%	2,8%
6	5	48	10,4%	10	502	9	1,9%	1,7%
7	4	343	1,2%	28	749	69	3,3%	8,2%
8	3	83	3,6%	4	636	1619	0,2%	71,7%
9	2	9	22,2%	3	441	2	0,7%	0,4%
10	4	108	3,7%	9	629	22	1,4%	3,3%
11	1	2	50,0%	1	348	0	0,3%	0,0%
TOT	43	1330	3,2%	127	6508	1825	1,5%	21,6%

Table 1. Results reported by Vacirca Tool and DeSRTO for the upload case.

To validate the algorithm, we have manually inspected all the RTOs expired during the experiments and we have verified their correspondence with the ones revealed by *DeSRTO*. It is worth to highlight that no false positive or negative cases were found by DeSRTO. It was an expected results, since the algorithm behavior follows the human operational procedure to find SRTO.

To validate its own algorithm, [7] uses a patched kernel that logs the timeout sequence numbers on the sender side and, on the receiver side, logs the hole in the sequence number space left by the reception of an out-of-order segment. In that paper, it is claimed that an out-of-order segment point out a loss, i.e., a NRTO, and, therefore, all the remaining RTOs are spurious. Note that this technique is more accurate that the use of the Vacirca's tool, but it is not free from errors. In fact, besides the intuitive failure in case of reordering, where an out-of-order segments is not a lost packet, this validation technique does not consider RTOs due to lost ACKs. In fact, in case a whole ACK window is lost, no hole is logged on the receiver, and then a NRTO is wrongly believed to be spurious. Therefore, we think that the validation technique used by [7] was unfeasible for our algorithm; in fact, our algorithm claims to work even with cases where the validation technique used by [7] fails. Thus, the only possible validation technique is the manual inspection of all RTOs.

	DeSRTO Results			Vacirca tool results				
N.	SRTO	NRTO	% SRTO	SRTO	NRTO	Ambiguous	% SRTO	% Ambiguous
1	1	0	100,0%	0	11	0	0,0%	0,0%
2	2	0	100,0%	0	12	0	0,0%	0,0%
3	1	0	100,0%	0	18	0	0,0%	0,0%
4	2	0	100,0%	1	65	0	1,5%	0,0%
5	1	0	100,0%	0	10	0	0,0%	0,0%
6	1	0	100,0%	0	8	0	0,0%	0,0%
7	1	0	100,0%	0	10	0	0,0%	0,0%
8	1	0	100,0%	0	8	0	0,0%	0,0%
9	1	0	100,0%	0	13	0	0,0%	0,0%
10	1	0	100,0%	0	3	0	0,0%	0,0%
11	1	0	100,0%	0	6	0	0,0%	0,0%
TOT	13	0	100,0%	1	164	0	0,6%	0,0%

Table 2. Results reported by Vacirca Tool and DeSRTO for the download case.

Even if Vacirca tool and DeSRTO are designed with different targets, a working comparison between the two tools is mandatory, although some differences in results are expected. For this purpose, we used an implementation of the Vacirca tool available (http://ccr.sigcomm.org/online/?q=node/220) as a patch for tcptrace v.6.6.7. The algorithm was applied using the traces captured on the sender side (the Ethernet interface in case of download, the UMTS interface in case of upload). Even if the location of the monitoring interface is unusual (it is not in the middle of the path), the placement is correct, since the only assumption done in [7] is that no loss is present between the sender side and the monitoring interface. The obtained results are reported in Tabs. 1 and 2. The comparison shows lots of differences. The number of RTOs detected by the Vacirca tool is substantially different from the ones reported by our kernel patch or, equally, by Web100. On average, the Vacirca patched version of tcptrace reports a number of RTOs about 6 times greater that the ones reported by the kernel, with peaks of 100 times. Instead, the number of SRTOs is more similar between the two tools, and, even if the results are significantly different, in some cases the reported values are comparable. It is worth to highlight that sometimes the number of ambiguous RTOs reported by the Vacirca tools is very high, although no packet reordering was experimented on the network in any experiments. Unfortunately, we were not able to make any reliable hypothesis on the causes of results obtained by the Vacirca tool. We found some issues in the use of such a tool and details about these problems can be found in [14].

5 Conclusions

In this paper, the new algorithm *DeSRTO* to find Spurious Retransmission Timeouts in TCP connections has been developed. Several examples have been re-

ported to illustrate its behavior in the presence of packet reordering, small windows, and other cases where competitors fail. Except for rare cases that are not classifiable with absolute certainty at all, the algorithm shows no ambiguous nor erroneous detections. Moreover, the effectiveness of the proposed algorithm has been highlighted with some results of its application on TCP traces collected in a real 3.5G cellular network and comparing its performance with respect to another detection tool available in literature. Future work will illustrate the application of *DeSRTO* to data traces in order to analyze the presence of SRTOs and their impact in several network environments.

Acknowledgement

Authors want to thank Dr. F. Ricciato and its team at FTW (Vienna) for suggestions and the valuable support during this work, which was funded by projects PS-121 and DIPIS (Apulia Region, Italy) as well as supported by TMA-COST action IC0703.

References

1. Jacobson, V.: Congestion avoidance and control. SIGCOMM Comput. Commun. Rev. **18**(4) (August 1988) 314–329
2. Allman, M., Paxson, V., Stevens, W.: RFC 2581: TCP congestion control (1999)
3. Floyd, S., Henderson, T., Gurtov, A.: RFC 3782: The NewReno modification to TCP's fast recovery algorithm (2004)
4. Paxson, V., Allman, M.: Computing TCP's retransmission timer (2000)
5. Ludwig, R., Meyer, M.: The Eifel detection algorithm for TCP. RFC 3522 (Experimental) (April 2003)
6. Sarolahti, P., Kojo, M.: Forward RTO-Recovery (F-RTO): An algorithm for detecting spurious retransmission timeouts with TCP and the stream control transmission protocol (SCTP). RFC 4138 (Experimental) (August 2005)
7. Vacirca, F., Ziegler, T., Hasenleithner, E.: An algorithm to detect TCP spurious timeouts and its application to operational UMTS/GPRS networks. Comput. Netw. **50**(16) (2006) 2981–3001
8. Blanton, E., Allman, M.: Using TCP duplicate selective acknowledgement (DSACKs) and stream control transmission protocol (SCTP) duplicate transmission sequence numbers (TSNs) to detect spurious retransmissions. RFC 3708 (Experimental) (February 2004)
9. Ludwig, R., Gurtov, A.: The eifel response algorithm for TCP. RFC 4015 (Proposed Standard) (February 2005)
10. Blanton, E., Allman, M.: Using spurious retransmissions to adapt the retransmission timeout (July 2007)
11. Postel, J.B.: Internet protocol. Internet RFC 791 (September 1981)
12. Mathis, M., Heffner, J., Raghunarayan, R.: TCP extended statistics MIB. RFC 4898 (Proposed Standard) (May 2007)
13. Bannister, J., Mather, P., Coope, S.: Convergence Technologies for 3G Networks: IP, UMTS, EGPRS and ATM. John Wiley & Sons (2004)
14. Barbuzzi, A.: Comparison measures between desrto and vacirca tool. Technical report (2009) available at http://telematics.poliba.it/DeSRTO_tech_rep.pdf.