Passive Detection of TCP Congestion Events

Shane Alcock and Richard Nelson
University of Waikato, Hamilton, New Zealand
Email: {salcock, richardn}@cs.waikato.ac.nz

***Abstract*—Detailed passive analysis of TCP sender behaviour requires accurate identification of congestion events. Previous tools that attempt to provide such information do not incorporate the behaviour of recent operating systems and TCP features and are therefore of little use to researchers analysing contemporary TCP traffic. In this paper, we present a new tool for identifying and classifying TCP congestion events from a passive packet trace, called *tcpcsm*, which understands modern operating system TCP behaviour. We discuss the major problems that occur when passively identifying TCP congestion events and describe how *tcpcsm* solves them. We also show that *tcpcsm* is more accurate than previous tools using a series of controlled experiments involving a variety of operating systems.**

I. Introduction

The development and improvement of TCP congestion control algorithms continues to be a significant and popular topic within the network research community. Algorithms and techniques are typically validated using both simulation and experimental test-bed networks, but there has been little emphasis on measuring and evaluating the congestion behaviour of TCP senders using passive network measurements.

One important parameter when evaluating the performance of a TCP sender is the size of the congestion window (*cwnd*), which determines the amount of data that the sender can have outstanding (i.e. unacknowledged) at any given time. Typically, the value of *cwnd* increases gradually as the sender observes acknowledgements that indicate the successful receipt of a segment and sharply decreases whenever a packet is deemed to have been lost and must therefore be retransmitted. Congestion control algorithms are defined by the manner in which *cwnd* is adjusted in response to these events. Being able to passively estimate the value of the congestion window at any point during a TCP connection would therefore be very helpful to researchers evaluating the performance and behaviour of congestion control algorithms in a real-world setting.

The first step in passively estimating the congestion window for a TCP sender is to develop a system for identifying and classifying congestion events. In this context, the term 'congestion event' refers to not only indications of congestion, such as the retransmission of lost packets, but any event that may affect the value of the congestion window. This includes the end of a fast recovery phase where the congestion window is deflated according to the algorithm defined in [1], or the detection of a spurious retransmit using the F-RTO algorithm [2] that causes the window to be reverted to a previous value.

The main difficulty in categorising congestion events is dealing with the variations present in TCP implementations for different operating systems. For example, Windows XP only requires two duplicate acknowledgements to trigger a fast retransmit event instead of the three defined in [1]. Many previous congestion window analysis tools, such as [3] and [4], developed algorithms that are based solely on the IETF standards for TCP, which significantly decreased their accuracy when used with real-world TCP traffic. By contrast, [5] designed a tool that did account for operating system differences, but this tool has not been updated since 2004 and is therefore of limited use when analysing recent data.

In this paper, we present *tcpcsm*, a tool that identifies and classifies congestion events in a TCP packet trace, which is based on the libtrace [6] trace processing library. *tcpcsm* is based on a new algorithm that successfully accounts for operating system variation using only a single state machine. This approach ensures that *tcpcsm* can be updated to support new congestion behaviour without needing to develop an entirely new state machine for each operating system release. The algorithm used by *tcpcsm* is also less vulnerable to timing variations than previous approaches. The tool supports TCP features that did not exist or were uncommon when previous passive congestion analysis tools were developed, such as duplicate SACK and F-RTO. *tcpcsm* has been validated using packet traces captured from both recent and older operating systems and can analyse both contemporary and historical traffic traces from a variety of sources.

II. Related Work

There have been several previous measurement studies that have attempted to passively identify and classify congestion events for TCP connections. The first prominent study was [3], where the authors implemented a tool called *tcpflows* that analysed the ACK stream to detect the occurrence of congestion events and update the congestion state for each flow accordingly. It also estimated the value of the congestion window and suggested the congestion control algorithm being used by the TCP sender. However, the state machine implemented within *tcpflows* was based solely on the RFC specifications and did not account for the differences between individual TCP implementations. Also, many new congestion control algorithms, such as CUBIC [7] and Compound TCP [8], have been developed and deployed since *tcpflows* was released. As a result, *tcpflows* is unable to accurately estimate the congestion window for most TCP senders.

Rewaskar et al. [5] developed a tool to characterise out-of-sequence segments in a TCP packet trace. The primary contribution of this work was that the authors recognised that there are many subtle differences in TCP implementations,

978-1-4577-0024-8/11/$26.00 ©2011 IEEE

particularly in how the RTO timer is calculated, that must be considered when identifying and classifying out-of-sequence segments. The authors showed that *tcpflows*, which does not account for the differences, is inaccurate when analysing real-world TCP traffic. Their approach was to implement separate state machines for four prominent operating systems. An analysis of a TCP connection would then be performed using each of the state machines and the best explanation for the observed behaviour would be selected and presented as the end result. The authors used a series of controlled experiments to demonstrate that their tool was more accurate and reliable than previous efforts.

This tool was released publicly but has only limited utility for analysing recently captured TCP traffic. The software has not been updated since late 2004 and has no support for modern operating systems, including Windows Vista and Windows 7. In addition, the tool produces many erroneous classifications even when tested against operating systems for which the authors had implemented a state machine and uses the textual output from an outdated version of the *tcpdump* tool [9] as input, which is very difficult for contemporary researchers to acquire and use. The tool also attempts to determine whether a given retransmit is necessary, but we feel this is not relevant for congestion window analysis. This is because whenever a segment is retransmitted, the effect on the congestion window is the same regardless of whether the retransmit was necessary.

More recently, [4] conducted an analysis of out of order TCP segments in several traffic traces. The authors disregarded the need to account for operating system variations and implemented a conservative algorithm based strictly on the IETF standards for TCP, similar to [3]. The reasoning for this approach was that it would reduce the likelihood of an incorrect classification, instead preferring to treat any retransmit that cannot be described by any of the loss detection techniques described in an RFC as “Unknown”. Therefore, many SACK-based and Windows XP fast retransmits identified by *tcpcsm* would not be classified correctly by their tool. For example, [4] classed over 20% of out of order segments from one trace set into the Unknown category.

Furthermore, none of the aforementioned tools recognise recent TCP features, such as duplicate SACK [10], Forward-RTO recovery [2] and SACK fast retransmit [11], which also affect the value of the congestion window.

In summary, several tools already exist that claim to detect congestion events, but none are particularly suitable for analysing contemporary TCP traffic, either due to being outdated or failing to account for operating system variability.

III. Methodology

In this section, we first present the underlying theory behind the approach implemented in *tcpcsm* and then discuss the complications and challenging scenarios that had to be addressed during the development process. It should be noted that a bidirectional traffic trace is required to detect congestion events, as both the outgoing data packets and the resulting acknowledgements must be observed.

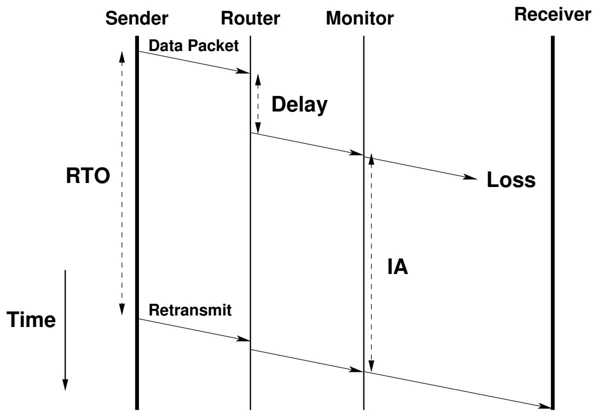

Fig. 1. An example of how delay between the sender and monitor can affect passive detection of RTO events. The original segment is delayed at the router before reaching the monitor. The retransmission is not delayed, however, and therefore the inter-arrival between the two segments (shown as IA in the diagram) is much smaller than the value of the RTO timer.

Because of space constraints, the basic concepts of TCP loss detection and recovery will not be discussed in detail here, as they are well-known and documented elsewhere. Instead, concepts such as retransmission timeout (RTO) [12], fast retransmit and recovery [1] [13], selective acknowledgements (SACKs) [11] [14], duplicate SACK (DSACK) [10] and Forward-RTO recovery (F-RTO) [2] are described in the referenced IETF RFCs.

A. Basic Approach

A TCP sender uses the stream of acknowledgement packets transmitted by the receiving endpoint to infer the occurrence of congestion events. Two common events are the presence of congestion along the sending path and the end of a period of loss recovery. The former is determined through the loss of transmitted segments while the latter occurs when all segments that were outstanding when the loss recovery phase began are successfully acknowledged. Given a time-ordered sequence of the data segments and acknowledgements (henceforth referred to as ACKs) sent by both TCP endpoints, such as a packet trace, it should therefore be possible to passively replicate the congestion state for a TCP sender.

For instance, a packet loss can be detected by the retransmission of a previously observed segment. Using the knowledge of the ACK stream leading up to the retransmit, it is possible to determine whether the retransmit is a fast retransmit or was caused by the expiry of the RTO timer. The congestion state can then be adjusted accordingly. If sufficient duplicate ACKs were observed prior to the retransmit, it is determined to be a fast retransmit and the congestion state machine moves into a fast recovery state. If not, the RTO timer must have expired and the state machine should instead move into an RTO loss recovery state.

Another example is an ACK that acknowledges all the segments that were outstanding when the fast recovery state was entered. That packet can be used to determine that fast recovery has ended and the replicated state machine can be returned to the base state.

Each time the congestion state machine moves into a new state or an event occurs that would have an effect on the value of the congestion window, e.g. a duplicate ACK during the fast recovery state, an appropriate congestion event is reported. In the case of *tcpcsm*, this event is simply written to an output file for later analysis, but the events could instead be used as input into a congestion window estimation tool, for example.

B. Identifying RTO Events

One common congestion event is the retransmission of a packet due to the expiration of the RTO timer. Previous tools, such as [5] and [3], attempt to estimate the value of the RTO timer for the connection. If the time elapsed between an original transmission and the subsequent retransmit exceeds the estimated RTO timer, these tools will determine the retransmit to be an RTO event.

We believe that this is a flawed approach. First, estimation of the RTO timer requires an accurate and current estimate of the round-trip time (RTT). However, RTT estimation from an observation point in the middle of a connection is an inexact process based on potentially incomplete information. This is especially true if the data-flow is largely unidirectional, preventing the estimator from gathering any recent timing data for one half of the connection. By comparison, the sender will always know the exact value of the RTO timer and can retransmit a lost segment as soon as the timer expires, so a high level of estimation accuracy is necessary to correctly identify RTO events.

In addition, delay variations along the path prior to the passive monitor may result in the gap between packets observed at the monitor differing greatly from what is observed by the sender, as illustrated in Figure 1. Combined with a potentially inaccurate estimate of the RTO timer, such delays will certainly lead to the misclassification of RTO events.

Furthermore, the TCP stacks for each of the major operating systems use differing algorithms and parameters for calculating the value of the RTO timer. To make matters worse, some operating systems maintain a separate timer for each outstanding segment whereas others utilise a single timer that is reset every time a new segment is sent or acknowledged. Without knowledge of the operating system being used by a TCP sender, it is very difficult to determine how the RTO timer should be estimated for any given connection. This was one of the principal reasons for [5] developing separate analysis programs for each major operating system.

By contrast, our approach is simple, albeit counterintuitive: do not use an estimate of the RTO timer to identify RTO events. The reasoning is that all retransmissions that are not caused by the expiry of the RTO timer are preceded by obvious signs in the ACK stream indicating that the packet was lost, such as duplicate ACKs. If all other causes of retransmission can be ruled out, the only logical conclusion is that the RTO timer must have expired for that segment. Careful validation, as shown in Section IV, has shown that this assumption works well in practice.

C. Variation in TCP Stacks

As described in [5], there are many differences between the TCP implementations developed by operating system vendors. These variations can be caused by a differing interpretation of the specifications for TCP or they may be the result of the developers attempting to optimise the performance of their particular TCP implementation. As a result, a sender utilising the Linux 2.6 TCP stack, for example, can behave very differently to a Windows Vista sender, given the same ACK responses from the receiver.

[5] attempted to solve this problem by implementing a separate state machine for each of the major operating systems. A TCP connection would be processed using all of the state machines and the analysis with the least unexplained events would be selected and presented as the final result. While effective, this is an inefficient approach; each connection must be processed multiple times to produce a single result. Also, additional state machines must be developed as new operating systems are released (or old ones are changed) which further increases the amount of processing required per connection.

Another problem is that many operating systems have a number of optional features that can be enabled or disabled by the user, creating further variation between instances of the same operating system. For example, F-RTO is implemented and supported within the Linux kernel but the feature is not enabled by default. This means that there are at least two viable profiles for a Linux sender: a default system that does not perform F-RTO and a custom system where the user has enabled the F-RTO feature.

As discussed earlier, one major source of operating system variation is the calculation of the RTO timer. Upon deciding to avoid estimating the timer, it became apparent that most remaining differences between operating systems can be defined in terms of the TCP extensions and features that have been implemented by the vendor. Examples of these features include F-RTO, partial ACKs and SACK fast retransmit. Each operating system supports a different subset of the TCP features, but the implementation of each feature typically does not differ markedly between TCP stacks.

As a result, we have implemented *tcpcsm* as a single state machine that is augmented with additional rules and states to detect the use of different TCP features and react accordingly. This approach has several major advantages. Firstly, there is no need to develop and maintain state machines for each operating system. Secondly, variation between different instances of the same operating system, e.g. due to the user enabling TCP features, is much less likely to invalidate the analysis. Finally, the analysis can be conducted using a single pass over the TCP traffic, which has obvious performance advantages and allows the analysis to be easily performed against live traffic.

However, there are still events that are difficult to identify unambiguously using the ACK stream alone. The most prominent example is fast retransmit under Windows XP. Despite the

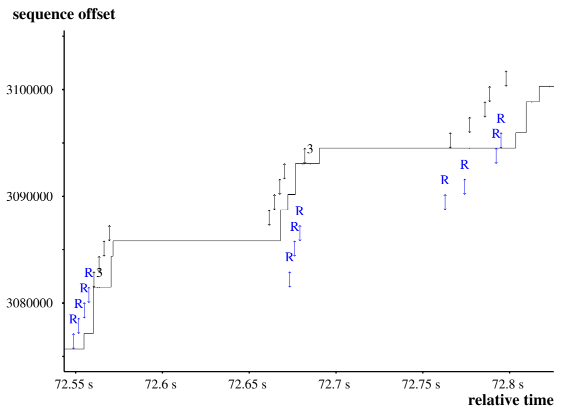

Fig. 2. A *tcptrace* graph for a real TCP connection showing how the relative ordering of packets travelling in opposite directions can be unreliable when the passive monitor is located in the "middle" of a path. Retransmitted segments (designated by the 'R') appear below the cumulative ACK (the solid line) whereas the actual cause is the series of duplicate ACKs (signified by the '3') that were observed much earlier by the monitor.

congestion control RFC [1] clearly stating that a fast retransmit can only be triggered after three duplicate ACKs, the default threshold in Windows XP is two duplicate ACKs. As a result, any retransmit preceded by two duplicate ACKs can have two possible causes: a fast retransmit for Windows XP senders and an RTO for other operating systems. To compensate, *tcpcsm* also incorporates some basic operating system fingerprinting, similar to that used by the p0f [15] tool, to disambiguate such special cases. The TTL, IP ID and receive window size are retained from the initial SYN packet sent by each endpoint and used to try and identify the sending operating system.

D. *Location of the Monitor Point*

Passive monitors are typically located in the core of a network so that they can capture and record a large quantity of traffic that is representative of all the users on that network. Packets can experience delay, loss or reordering between the sender and the passive monitor. Therefore, the sequencing and timing of the segments may be different in the packet trace compared to what occurred at each of the endpoints. The problem of additional delays has already been addressed in our discussion of RTT estimation; to summarise, such delays create uncertainty with regard to both RTT estimation and RTO event detection.

Another significant problem that arises from the location of the monitor point is that the relative ordering and timing between incoming and outgoing segments in the packet trace can be altered because the captured segments are only half-way to their destination. This is illustrated in the *tcptrace* [16] graph in Figure 2, where the retransmits are observed after a cumulative acknowledgement that should have prevented them being sent in the first place. As a result, one cannot rely solely on the most recent acknowledgements to determine the cause of a retransmit; historical information must be retained as well.

In addition, packets can be lost as they traverse the path between an endpoint and the passive monitor. For example, if a data segment is lost prior to reaching the monitor, the retransmit will be the first occasion that the segment is seen by the tool. This means that the number of times a segment has been observed is not always a reliable indicator as to whether that segment has been retransmitted or not.

We use a similar approach to that described in [5] to ensure that retransmissions are identified correctly in such instances. Within *tcpcsm*, transition into a loss or fast recovery state can only occur once the retransmit itself is observed, rather than the loss indication in the ACK stream. An ordered list of potential fast retransmits is maintained for each TCP sender and an entry is added to the list whenever the duplicate ACK threshold is reached. The entry is also updated as further duplicate ACKs are observed. Whenever a segment is retransmitted, the list is searched for a corresponding entry that would indicate the sender had a reason to fast retransmit the segment. If found, the retransmit is classed as a fast retransmit.

Another problem is that packets can be duplicated or reordered as they traverse the path between the sender and the passive monitor. As the segments do not contain the expected TCP sequence number, these events can be easily mistaken for retransmissions. *tcpcsm* uses some simple heuristics to detect both reordering and duplication, where the IP ID of the unexpected segment is compared with the IP ID of the previous packet transmitted by the sender. If they are the same, a network duplication event is reported. Reordering is detected whenever the segment is being observed for the first time and the IP ID is below the value of the highest IP ID seen thus far. This assumes IP IDs are assigned sequentially, which is true for all prominent operating systems except for OpenBSD [17]. Our experiences from developing and testing *tcpcsm* suggest that this is still correct.

E. *SACK Fast Retransmit*

The Linux operating system implements a feature whereby the sender can enter fast recovery if it receives a selective acknowledgement for a segment that is ahead of the current cumulative ACK. This feature, which we refer to as SACK Fast Retransmit (SACK/FR), is not described or defined in any IETF standard. The closest we found was [11] which defines a similar method for detecting packet loss whenever three full segments of data are SACKed. However, we found that SACK/FR can occur even when only a single segment is SACKed.

Because SACK/FR can be triggered by only a single packet from the receiver and is not common to all operating systems, SACK/FR events are very difficult to detect reliably. Our approach first considers all other possible causes of retransmission that can be identified much more reliably than SACK/FR, e.g. fast retransmit due to duplicate ACKs. If the retransmit is still unexplained, it is deemed to be a SACK/FR if the following conditions are met:

- Some data has been SACKed since the most recent cumulative ACK.
- The sequence number of the retransmit matches the current cumulative ACK.

- The time elapsed between the retransmit and the last observed SACK is less than the time taken to complete the connection handshake.
- No new segments have been sent between the last observed SACK and the retransmit.

F. Summary

Taking the aforementioned problems into account, the key features of *tcpcsm* can be briefly described as follows.

- The implementation is based on a single congestion state machine using only three primary states: Open, Fast Recovery and Loss Recovery.
- Indications of loss in the ACK stream, i.e. duplicate ACKs, do not trigger state transitions, but are recorded in a list of potential retransmits to aid in classifying retransmissions when they do occur.
- As a result, transition into the recovery states only occurs when a retransmit is observed in the data stream. However, transition back into the Open state occurs as soon as the ACK stream indicates the end of recovery.
- Unlike other similar tools, *tcpcsm* does not estimate the value of the RTO timer. RTO events are identified solely by virtue of there being no other suitable explanation for a retransmit.
- Fast retransmits are identified by the presence of the segment in the list of potential retransmits described above or by satisfying the conditions for a SACK/FR.
- Reordered and duplicated packets are detected via analysis of the IP ID field. This also aids in identifying retransmits where the original segment was lost prior to the passive monitor.
- Simple OS fingerprinting based on data extracted from the SYN packet is used to classify events that would otherwise be ambiguous, i.e. a retransmit following two duplicate ACKs.
- Features that are not common to all TCP implementations, such as F-RTO and DSACK, are handled through additional logic and states that detect when they are in use and report any resulting congestion events.

There are many further minor implementation details that will not be discussed here due to space constraints. These are typically required to handle the variety of unexpected and uncommon behaviour that can be found in real-world TCP traffic and are documented within the source code of *tcpcsm* itself, which can be downloaded from [18].

IV. Validation

To validate the performance of *tcpcsm*, we used the TCP Behavior Inference Tool (TBIT) [19] to conduct a series of controlled experiments where the ACK stream was modified to artificially create different congestion scenarios. The events reported by *tcpcsm* for each scenario were manually compared against the packet stream observed during the experiment to confirm that our tool was operating correctly and accurately. This was a similar approach to the one used by [5] to validate their own software except that we added further tests to evaluate the new TCP features supported by *tcpcsm*, creating a total of 13 validation scenarios. The scenarios are briefly described in Table I. We did not replicate the RTT and RTO estimation tests described in [5] as *tcpcsm* does not attempt to maintain an RTO timer, rendering these tests irrelevant.

TABLE I
Descriptions of each validation scenario

Scenario	Description
A	Detecting an RTO event without SACK
B	Detecting an RTO event with SACK
C	Detecting an RTO event following a single duplicate ACK
D	Detecting multiple RTO events for the same packet
E	Detecting an RTO event following a fast retransmit
F	Detecting an RTO event for a packet retransmitted during fast recovery
G	Detecting Windows XP fast retransmit
H	Detecting standard fast retransmit
I	Testing response to partial ACKs
J	Detecting a SACK fast retransmit event
K	Testing response to unnecessary retransmits
L	Detecting a spurious RTO event using F-RTO
M	Testing response to packet reordering

We validated *tcpcsm* against a wide range of operating systems, both new and old. These were Solaris 10, FreeBSD 5.3, FreeBSD 7.2, Linux 2.6.23, Windows XP SP2, an unpatched Windows Vista and Windows Vista SP2. We also used the traces captured from the Linux, Solaris, FreeBSD 5.3 and Windows XP servers to compare the performance of our tool against the Rewaskar tool [5], as it claimed to support those four operating systems. The validation process consisted of two principal steps for each individual experiment. Firstly, the packet trace captured during the experiment would be manually analysed using *tcptrace* [16] to establish how the server responded to the congestion scenario and identify the ground truth with regard to the congestion events that occurred. The trace was then processed using *tcpcsm* and the results compared with the events that were identified during the earlier *tcptrace* analysis.

The validation results are summarised in Table II. A tick symbol indicates that all congestion events were correctly identified for that scenario, whereas a cross indicates the opposite. Complete descriptions of each validation scenario and the results for the individual tests can be found at [20]. The results show that *tcpcsm* was able to correctly identify the congestion events for almost all of the scenarios, with two major exceptions. Firstly, there was a bug in the implementation of F-RTO in the initial release of Windows Vista that meant our tool was unable to correctly classify all congestion events following an RTO event, as the server behaviour did not conform to the F-RTO specification. Secondly, both Linux and FreeBSD 5 did not trigger the expected fast retransmit during scenario I due to the absence of SACK information, resulting in an RTO event that was incorrectly classified as the expected fast retransmit event by *tcpcsm*.

By contrast, the Rewaskar tool was unable to correctly identify many events for operating systems that it supposedly supported. In particular, the failure to identify Windows XP

TABLE II
VALIDATION TEST RESULTS

	tcpcsm							Rewaskar			
Scenario	Solaris	FreeBSD 5	FreeBSD 7	Linux	Win XP	Vista	Vista SP2	Solaris	FreeBSD 5	Linux	Win XP
A	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
B	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
C	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
D	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
E	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
F	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓
G	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
H	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
I	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✗
J	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
K	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
L	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
M	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓

fast retransmit events (scenario G) was surprising, given that these were explicitly noted in [5] as a major source of operating system variability. The Rewaskar tool also failed to identify any RTO events correctly for FreeBSD 5, which we believe is due to a change in the RTO timer algorithm since the Rewaskar tool was developed, and the Linux and Solaris state machines could not correctly classify congestion events in the partial ACK and reordering scenarios.

V. CONCLUSION

This paper introduced a new tool for identifying and classifying congestion events in a passive TCP packet trace: *tcpcsm*. In doing so, we have improved upon the approach of existing tools that attempt a similar analysis. *tcpcsm* is implemented entirely within a single state machine, but is still able to compensate for operating system variability. This includes variation not only between different operating systems, but also in terms of optional TCP features, such as F-RTO, being utilised by the same operating system.

Recognising that RTO events can be identified by ruling out all other possible causes of retransmission enabled us to eliminate the need for separate state machines for each TCP stack, which was inefficient and difficult to maintain. *tcpcsm* is also less vulnerable to variations in delay between the sender and the passive monitor as a result, which was a major source of error in previous tools. Simple OS fingerprinting techniques have also been utilised to resolve any remaining ambiguities caused by differences between TCP implementations.

Furthermore, the previous tools have not been actively maintained for many years and are incapable of coping with modern operating systems and TCP features. Our validation experiments also found instances where an earlier tool failed to reliably identify even basic congestion events, such as the Windows XP fast retransmit. Therefore, the development of a tool that is up-to-date, accurate and comprehensive that can be used for processing recently captured data is a significant contribution in itself.

We also believe that the tool presented here can be further developed to analyse other aspects of TCP performance in real-world connections. For instance, the congestion events reported by *tcpcsm* can be used to develop a tool that can passively estimate the size of the congestion window for TCP senders. This will enable researchers to conduct an in-depth analysis of the performance of TCP congestion control algorithms in real-world environments and evaluate the potential effect of new congestion control techniques.

REFERENCES

[1] M. Allman, V. Paxson, and W. Stevens, "RFC 2581 - TCP Congestion Control," April 1999.

[2] P. Sarolahti, M. Kojo, K. Yamamoto, and M. Hata, "RFC 5682 - Forward RTO-Recovery (F-RTO)," September 2009.

[3] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Inferring TCP Connection Characteristics Through Passive Measurements," in *Proceedings of Infocom 2004, Hong Kong*, 2004.

[4] M. Mellia, M. Meo, L. Muscariello, and D. Rossi, "Passive Analysis of TCP Anomalies," *Comput. Netw.*, vol. 52, no. 14, pp. 2663–2676, 2008.

[5] S. Rewaskar, J. Kaur, and F. D. Smith, "A Passive State-Machine Approach for Accurate Analysis of TCP Out-of-Sequence Segments," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 3, pp. 51–64, 2006.

[6] WAND Network Research Group, "Libtrace," http://research.wand.net.nz/software/libtrace.php.

[7] S. Ha, I. Rhee, and L. Xu, "CUBIC: a New TCP-Friendly High-Speed TCP Variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[8] K. Tan and J. Song, "A Compound TCP Approach for High-Speed and Long Distance Networks," in *In Proc. IEEE INFOCOM*, 2006.

[9] http://www.tcpdump.org/.

[10] E. Blanton and M. Allman, "RFC 3078 - Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions," February 2004.

[11] E. Blanton, M. Allman, K. Fall, and L. Wang, "RFC 3517 - A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP," April 2003.

[12] V. Paxson and M. Allman, "RFC 2988 - Computing TCP's Retransmission Timer," November 2000.

[13] S. Floyd, T. Henderson, and A. Gurtov, "RFC 3782 - The NewReno Modification to TCP's Fast Recovery Algorithm," April 2004.

[14] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "RFC 2018 - TCP Selective Acknowledgment Options," October 1996.

[15] "p0f - a Versatile OS Fingerprinting Tool," http://lcamtuf.coredump.cx/p0f.shtml.

[16] "tcptrace," http://www.tcptrace.org/.

[17] J. Bellardo and S. Savage, "Measuring Packet Reordering," in *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*. New York, NY, USA: ACM, 2002, pp. 97–105.

[18] WAND Network Research Group, "libtcpcsm," http://research.wand.net.nz/software/tcpcsm.

[19] J. Pahdye and S. Floyd, "On Inferring TCP Behavior," in *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. New York, NY, USA: ACM, 2001, pp. 287–298.

[20] S. Alcock, "tcpcsm," http://www.wand.net.nz/~salcock/tcpcsm/.