Computer Communications 52 (2014) 82–88

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Transport layer reneging

Nasif Ekiz [a,*], Paul D. Amer [b]

[a] F5 Networks, Seattle, WA 98119, United States
[b] Computer and Information Sciences Department, University of Delaware, Newark, DE 19716, United States

ARTICLE INFO

Article history:
Received 21 June 2013
Received in revised form 16 April 2014
Accepted 29 May 2014
Available online 12 June 2014

Keywords:
OS fingerprinting
Reneging
SACK
TCP

ABSTRACT

Reneging occurs when a transport layer data receiver first selectively acks data, and later discards that data from its receiver buffer prior to delivery to the receiving application or socket buffer. Reliable transport protocols such as TCP (Transmission Control Protocol) and SCTP (Stream Control Transmission Protocol) are designed to tolerate reneging. We argue that this design should be changed because: (1) reneging is a rare event in practice, and the memory saved when reneging does occur is insignificant, and (2) by not tolerating reneging, transport protocols have the potential for improved performance as has been shown in the case of SCTP. To support our argument, we analyzed TCP traces from three different domains (Internet backbone, wireless, enterprise). We detected reneging in only 0.05% of the analyzed TCP flows. In each reneging case, the operating system was fingerprinted thus allowing the reneging behavior of Linux, FreeBSD and Windows to be more precisely characterized. The average main memory returned each time to the reneging operating system was on the order of two TCP segments. Reneging saves so little memory that it is not worth the trouble. Since reneging happens rarely and when it does happen, reneging returns insignificant memory, we recommend designing reliable transport protocols to not permit reneging.

1. Introduction

Transmission Control Protocol (TCP) [21] and the Stream Control Transmission Protocol (SCTP) [25] use sequence numbers and cumulative acks (ACKs) to achieve reliable data transfer. A data receiver uses sequence numbers to sort arrived data segments. Data arriving in expected order, i.e., *ordered data*, are cumulatively ACKed to the data sender. With receipt of an ACK, the data sender assumes the data receiver accepts responsibility for delivering ACKed data to the receiving application, and the data sender deletes all ACKed data from its send buffer, potentially before that data are delivered.

The Selective Acknowledgment Option, RFC2018 [16], extends TCP's (and SCTP's) cumulative ACK mechanism by allowing a data receiver to ack arrived *out-of-order data* using selective acks (SACKs). The intent is that SACKed data need not be retransmitted during loss recovery. SACKs improve throughput when multiple losses occur within the same window [1,4,9].

Deployment of the SACK option in TCP connections has been a slow, but steadily increasing trend. In 2001, 41% of the web servers tested were SACK-enabled [20]. In 2004, SACK-enabled web servers increased to 68% [17]. All inspected operating systems at the writing of the paper such as FreeBSD 8, Linux 2.6.31, Mac OS X 10.6, OpenBSD 4.8, OpenSolaris 2009, Solaris 11, Windows 7, Windows Vista negotiated SACKs by default.

Transport layer data reneging (simply, reneging) occurs when a data receiver first SACKs data, and later discards that data from its receiver buffer *prior* to delivery to the receiving application or socket buffer. TCP is designed to tolerate reneging. RFC2018 states: "The SACK option is advisory" and "the data receiver is permitted to later discard data which have been reported in a SACK option". Reneging might happen, for example, when an operating system needs to recapture previously allocated memory, say to avoid deadlock, or to protect the operating system against denial-of-service attacks (DoS). As will be discussed in detail in this paper, reneging is implemented in FreeBSD, Linux, Mac OS, and Windows.

Because TCP tolerates reneging, a TCP data sender must retain copies of all transmitted data in its send buffer, even SACKed data, until they are ACKed. Then, if reneging does occur, eventually the sender will (1) timeout on the reneged data, (2) delete all SACK information, and (3) retransmit the retained copies of the reneged data. The data transfer thus remains reliable. Unfortunately, if reneging does not happen, SACKed data is wastefully stored in the send buffer until ACKed.

* Corresponding author. Tel.: +1 302 2426831.
E-mail addresses: n.ekiz@f5.com (N. Ekiz), amer@udel.edu (P.D. Amer).

http://dx.doi.org/10.1016/j.comcom.2014.05.009
0140-3664/

A similar design to tolerate reneging is adopted by SCTP. The main difference is that an SCTP data sender is designed to identify a data receiver that reneges, whereas a TCP data sender is not. When previously SACKed data are not repeatedly SACKed in the successive ack, an SCTP data sender infers reneging and marks reneged data for retransmission [25].

We argue that the transport protocol design to allow reneging should be changed because: (1) reneging is a rare event in practice, and the memory saved when reneging does occur is insignificant, and (2) by not allowing reneging, reliable transport protocols have the potential for improved performance as has been shown in the case of SCTP [19,28].

This paper presents a thorough investigation into reneging to support (1). For that purpose, we herein extend an earlier model [6] to detect reneging instances in TCP traces. Then we use the extended model to analyze over 200,000 TCP connections from three different domains to report the frequency of reneging. For those connections that do renege, we fingerprint the OS to better understand how today's major OS deal with reneging. The amount of potential gain (i.e., item (2)) by designing TCP to not tolerate reneging is currently under study [2], and beyond the scope of this paper.

In Section 2, we further the motivation to detect reneging instances and present the only past study to investigate reneging in TCP. Then Section 3 presents our model to detect reneging instances in TCP trace files. Section 4 presents the TCP trace analysis and results. Finally, Section 5 presents our recommendation to change the design of reliable transport protocols.

2. Background and motivation

If a transport protocol were designed not to tolerate reneging (i.e., to be non-reneging), a data sender would no longer need to retain copies of SACKed data in its send buffer until ACKed. In that case, the main memory allocated for the send buffer could be utilized for other data or connections.

Natarajan et al. [19] present send buffer utilization results for data transfers using non-reneging vs. reneging SCTP under mild (~1–2%), medium (~3–4%) and heavy (~8–9%) loss rates. For the bandwidth-delay parameters studied, the memory wasted by assuming SACKed data could be reneged was on average ~10%, ~20% and ~30% for the given loss rates, respectively.

A non-reneging transport protocol also can improve end-to-end application throughput. To send new data, in TCP and SCTP, a data sender is constrained by three factors: a congestion window (congestion control), an advertised receive window (flow control) and a send buffer. When the send buffer is full, no new data can be transmitted even when congestion and flow control mechanisms allow. When SACKed data are removed from the send buffer in a non-reneging protocol, new application data can be read and potentially transmitted earlier increasing throughput.

Yilmaz et al. [28] investigate throughput improvements for non-reneging vs. reneging SCTP. The authors show that the throughput achieved with non-reneging SCTP is always $\geqslant$ the throughput observed with reneging SCTP. For example, the throughput for data transfer over SCTP is improved by ~14% for a data sender with 32 KB send buffer under low (~0–1%) loss rate with non-reneging SCTP.

In summary, it has been shown if SCTP were designed to not tolerate reneging, send buffer utilization would be always optimal, and application throughput could be improved for data transfers with constrained send buffers (send buffer < receive buffer). We believe that while significant differences exist between SCTP and TCP implementations, these SCTP results will apply to TCP as well following a modified handling of TCP's send buffer. SCTP results however are at best a predictor. A detailed evaluation of non-reneging TCP over a satellite link is an ongoing research [2].

The key issue for this paper is – in practice, does reneging occur or not? No one knows what percentage of connections renege. To the authors' best knowledge, the only prior study of reneging is an MS thesis not published elsewhere [3]. The author presents a reneging detection algorithm for a TCP data sender, and analyzes TCP traces using the detection algorithm to report frequency of reneging. The author hypothesized that discarding the SACK scoreboard at a timeout may have a detrimental impact on a connection's ability to recover loss without unnecessary retransmissions. To decrease unnecessary retransmissions, an algorithm to detect reneging at a TCP sender is proposed which clears the SACK scoreboard immediately upon detecting reneging instead of waiting until a timeout. The reneging detection algorithm compares existing SACK blocks (scoreboard) with incoming ACKs. When an ACK is advanced to the middle of a SACK block, reneging is detected. The author indicates reneging can be detected earlier when the TCP receiver skips previously SACKed data. In such a case, SACKs are used for reneging detection as in our model detailed in Section 3.

Using traces, the author analyzed TCP connections with SACKs to report frequency of reneging. Out of 1,306,646 connections analyzed, the author identified 227 connections (0.017%) as reneged. These results suggest that reneging is a rare event.

Our objective is to report the frequency of reneging in today's Internet. If we observe reneging occurs rarely or never, we will have evidence to change the basic assumptions of transport layer protocols. By designing non-reneging transport protocols, we hypothesize that few (if any) connections will be penalized, and the large majority of non-reneging connections will potentially benefit from better send buffer utilization and throughput.

3. A model to detect reneging

To empirically investigate the frequency of reneging, we present our extended model and its implementation, RenegDetectv2, to passively detect reneging instances occurring in TCP traces.

While TCP does not support detecting reneging at a data sender, SCTP does. In SCTP, when previously SACKed data are not repeatedly SACKed in successive acks as is specified, an SCTP data sender infers reneging. Our initial model to detect TCP reneging extends SCTP's reneging detection mechanism [6].

A state of the data receiver's receive buffer is constructed at an intermediate router and updated as new acks are observed. The state consists of a cumulative ACK value (stateACK) and a list of out-of-order data blocks (stateSACK blocks) known to be in the data receiver's buffer. When an inconsistency occurs between the state of the receive buffer and a new ack, reneging is detected. Our initial model was introduced in [6], and is now described so as to understand how and why we needed to extend it.

Fig. 1 illustrates an example reneging scenario, and how our initial model located at an intermediate router detects reneging. Three acks are monitored within a data transfer. For simplicity, data packets are not shown. Without loss of generality, the example assumes 1 byte of data is transmitted in each data packet. For each SACK X–Y, X and Y represent the left edge and right edge of the SACK, respectively.

On seeing ACK 1 SACK 3–4, our model deduces the state of receive buffer to be: ordered data 1 is delivered or deliverable to the receiving application (stateACK 1), and out-of-order data 3–4 are in the receive buffer (stateSACK 3–4). ACK 1 SACK 3–6 updates this state by adding out-of-order data 5–6 as SACKed (stateSACK 3–6). When ACK 2 SACK 7–7 is received and compared to the state of receive buffer (stateACK 1, stateSACK 3–6), an inconsistency is