Demystifying and Mitigating TCP Stalls at the Server Side

Jianer Zhou[†§*], Qinghua Wu[†§*], Zhenyu Li[†*], Steve Uhlig[‡], Peter Steenkiste[♯], Jian Chen[♭], Gaogang Xie[†]

[†]ICT-CAS, [§]Uni. of CAS, [‡]QMUL, [♯]CMU, [♭]Qihoo 360

{zhoujianer, wuqinghua, zyli, xie}@ict.ac.cn, steve.uhlig@qmul.ac.uk, prs@cs.cmu.edu, chenjian@360.cn

ABSTRACT

TCP is an important factor affecting user-perceived performance of Internet applications. Diagnosing the causes behind TCP performance issues in the wild is essential for better understanding the current shortcomings in TCP. This paper presents a TCP flow performance analysis framework that classifies causes of TCP stalls. The framework forms the basis of a tool that is publicly available to the research community. We use our tool to analyze packet-level traces of three services (cloud storage, software download and web search) deployed by a popular Chinese service provider. We find that as many as 20% of the flows are stalled for half of their lifetime. Network-related causes, especially timeout retransmission, dominate the stalls. A breakdown of the causes for timeout retransmission stalls reveals that double retransmission and tail retransmission are among the top contributors. The importance of these causes depends however on the specific service. We also propose S-RTO, a mechanism that mitigates timeout retransmission stalls. S-RTO has been deployed on production front-end servers and results show that it is effective at improving TCP performance, especially for short flows.

*The first three authors contributed equally to this work.

CoNEXT '15, December 01-04, 2015, Heidelberg, Germany

 ISBN 978-1-4503-3412-9/15/12$15.00 .

DOI: http://dx.doi.org/10.1145/2716281.2836094

CCS Concepts

•**Networks** → **Transport protocols; Network measurement;** *Network experimentation;*

Keywords

TCP; Performance Troubleshooting; Measurement; Congestion Control

1. INTRODUCTION

Today's Internet users are increasingly concerned about the perceived performance (i.e., throughput and latency). Current popular applications rely heavily on TCP. Therefore, large service providers, e.g., Google and Amazon, are trying to improve TCP performance, especially at the server side where they have better control.

The effectiveness of congestion control and error recovery has a significant impact on TCP performance. A TCP sender transmits segments as its sending window allows, which is limited by both its congestion window and the receiver's advertised window. The congestion window is adjusted based on acknowledgments (in short ACKs), packet loss and delay events. The sender infers segment loss from a combination of duplicate ACKs and timeout heuristics, and retransmits segments if its sending window permits. While mechanisms such as fast retransmit [4] allow fast loss recovery in many cases, timeout retransmissions still have a significant impact. For example, a recent study [8] found that 77% of segment losses are recovered by timeout retransmissions.

There is a significant body of research on TCP performance optimization. Some works improve performance from the perspective of congestion control, such as FastTCP [22], Compound TCP [19], TCP Cubic [9], and RemyCC [23]. Other research proposes to improve TCP transfer performance by optimizing retransmissions so as to react to packet loss more quickly, e.g., tail loss probe [8], limited transmit [2], and adaptive re-

ordering threshold [25]. While many of these results have had significant impact, timeout retransmissions remain a problem and their causes are still not well understood. This motivated us to investigate the factors that cause timeout retransmissions in the wild and server-side techniques that alleviate the negative impact of timeout retransmissions.

To this end, we first analyze and categorize the factors that may cause TCP performance issues. We structure the analysis into a decision tree classification framework that identifies the cause of observed TCP performance issues. Based on this framework, we build a TCP performance diagnosis tool used to analyze a dataset made of 6 million complete flows from three representative services (i.e., cloud storage, software download and web search) of Qihoo 360, a platform serving hundreds of millions of users. We are particularly interested in the TCP performance issues caused by timeout retransmissions as they stall flows for hundreds of ms, and sometimes for several seconds. We find that more than 20% of the studied flows are stalled for the majority of their lifetime. Specifically, we make the following major observations about the causes of stalls:

- While all possible causes related to sender, receiver and network can impair TCP performance, timeout retransmissions are the most significant factor in terms of stalled time because of the long timeout. In particular, we observe that timeout retransmissions account for 30∼60% of the stalled time.
- A breakdown of the timeout retransmission stalls shows that double retransmissions and tail retransmissions are among the dominant factors. While tail retransmissions can be alleviated by mechanisms such as TLP [8], mitigating double retransmission stalls requires new server-side mechanisms.
- The main causes of TCP stalls are dependent on the properties of the application as well as the behavior of the client software. For example, web search flows are more likely to be influenced by tail retransmissions, which have limited impact on the other two considered applications. In addition, the small initial receive window (i.e., 2 MSS) that is used by some of the software installed on clients results in a high likelihood of suffering from ACK delay related retransmissions.

Based on these observations, we propose Smart-RTO (S-RTO), a mechanism aimed at reducing timeout retransmissions by retransmitting unacknowledged packets slightly more aggressively. S-RTO has been deployed on some servers hosting cloud storage and web search services in Qihoo 360's production network for daily use. In summary, the contribution of this work is twofold:

- We build a tool that classifies the causes for TCP stalls. We use this tool to analyze stalls in packet-level traces from front-end servers of a popular Chinese service provider. We consider stalls caused by server, client and network features and events, and we focus on timeout retransmission stalls because of their significant impact on performance.
- We propose S-RTO to mitigate timeout retransmission stalls. By deploying it in production network, we observe S-RTO can effectively improve the TCP performance, especially for short flows. For example, S-RTO decreases the 90th percentile of the latency of short flows in the cloud storage service by 45% compared with native TCP, which is significantly better than the improvement (14%) obtained by TLP [8].

Table 1: Flow-level statistics of the dataset.

	# flows	**avg. speed** (B/s)	**avg. flow size**	**pkt loss**	**avg. RTT**	**avg. RTO**
cloud stor.	2.2M	540K	1.7MB	3.9%	143ms	1.2s
soft. down.	0.9M	413K	129KB	4.1%	147ms	1.6s
web search	3.3M	644K	14KB	2.1%	106ms	0.9s

The rest of the paper is structured as follows. Section 2 describes the dataset used. Section 3 presents the TCP stall classification tool and high-level statistics of stalls. Section 4 details causes for timeout retransmission stalls. We present S-RTO and evaluate its performance in Section 5, followed by the discussion of related work in Section 6. Section 7 concludes this paper.

2. DATASET

The dataset we used in our TCP performance analysis was collected from front-end servers of Qihoo 360[1]. As a leading Internet and mobile platform company in China, Qihoo 360 provides a variety of Internet services, including Internet security, web search, mobile assistant and cloud storage, to over 600 million users. In this section, we first describe our TCP dataset and then provide general statistics about TCP stalls.

2.1 Dataset description

Our dataset consists of packet-level traces from the servers that host three popular Internet services, web search, cloud storage download, and security software (including patches) download, to users in several provinces in eastern China. Each server hosts only one type of service. The servers run CentOS Linux 6.2 with kernel version 2.6.32 [2]. Users on both mobile devices and fixed computers use a web browser to access the web search service. The cloud storage and software download services on the other hand are accessed from Qihoo

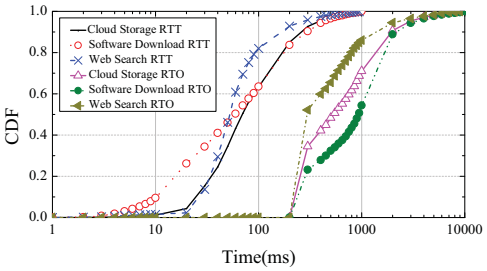
[1]http://www.360.cn/

[2]Early Retransmit [1] and Tail Loss Probe [8] are not available in this kernel version.

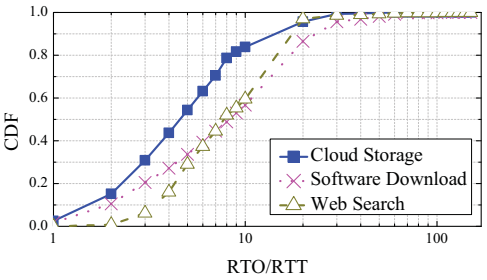
360 client software through standard HTTP requests. Cloud storage and software download services both use multiple TCP connections to download chunks of a file, but they do so in a different way. A TCP connection to the cloud storage service can transfer chunks belonging to multiple requested files (i.e., connections are shared among multiple file transferring.). In contrast, connections to the software download service are dedicated to a single file. If a front-end server does not have the requested content locally (e.g., dynamic objects for web search), it retrieves the content from back-end servers and then serves it to the client.

The dataset consists of daily one-hour long packet-level traces, collected during peak time in the evening over a 7-day period from Dec. 22, 2014 to Dec. 28, 2014. Overall, we collected 3.35 billion packets, corresponding to 6.4 million flows. Table 1 summarizes the flow-level statistics of the traces. Cloud storage flows are larger, on average one and two orders of magnitude larger than software download and web search flows respectively. We observe a packet loss rate of over 2% for web search, and of about 4% for cloud storage and software download.

Figure 1a plots the cumulative distribution of Round Trip Time (RTT) and Retransmission Time Out (RTO) for individual flows in the three services, using a log scale x-axis. The RTT is measured for each segment that is not retransmitted, while the RTO is recorded for each timeout retransmission. The three services experience different average RTTs, of 106ms for web search and around 140ms for the other two services. The RTO seems to be much higher than the RTT due to TCP's very conservative algorithm for RTO calculation [8]. Figure 1b shows the distribution of the RTO normalized by the RTT. We observe that the RTO is one order of magnitude larger than the RTT for over 40% of the software download and web search flows.



(a) Per-flow RTT and RTO.



(b) RTO / RTT

Figure 1: Distribution of RTT and RTO.

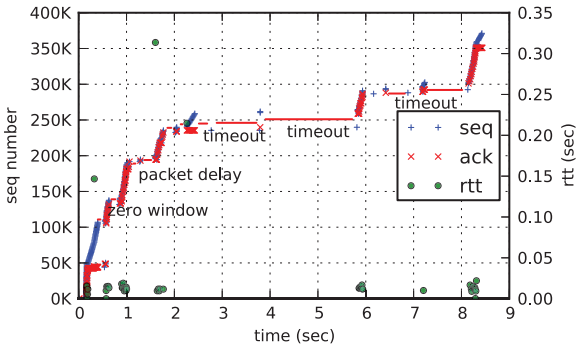


Figure 2: Illustrative example of TCP stalls within a flow.

2.2 TCP stalls

Many factors can degrade TCP performance, e.g., packet loss or a zero receive window, while combinations of events can exacerbate performance loss, e.g., packet loss when the number of outstanding packets is small. Figure 2 uses the trace of a TCP observed in Qihoo 360's cloud storage application to illustrate the impact stalls can have on performance; the graph shows the sequence number on the left y-axis and the RTT on the right y-axis. First, the flow is stalled due to the receiver's zero window for about 250ms. The next stall is the results of RTT variations for about 300ms and finally, the flow is stalled several times for more than 1 second due to timeouts. In total, it takes the sender 9 seconds to transmit 400KB of data, during which the transmission is stalled for more than 5 seconds.

This example clearly shows that TCP flow *stalls* can have a significant impact on the performance of Internet services. In this paper, a *TCP stall* is defined as an event where the duration between two consecutive packets received/sent by the sender is larger than $\min(\tau \cdot \text{SRTT}, \text{RTO})$. Here, the Smoothed RTT (SRTT) and RTO are calculated according to RFC 6298 [15] as implemented in the Linux kernel. Similar to [8], we set τ as 2, following the intuition that under normal circumstances, a TCP sender should be able to receive or send at least one packet during 2 RTTs.

In total, we observe 16M, 2M and 0.8M stalls in flows for the cloud storage, software download and web search services respectively. Figure 3 shows the CDF of the ratio of stalled time to flow transmission time for the three services. As much as 43% of the software download flows and 38% cloud storage flows experience at least one stall. For those experiencing stalls, we observe a significant impact of stalls on transfer time. For example, over 20% of the cloud storage and software

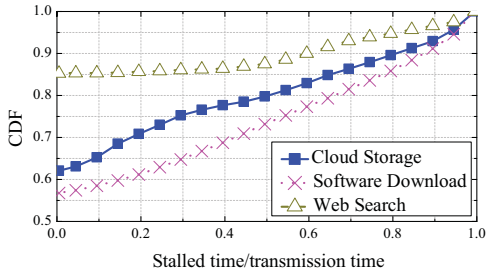


Figure 3: Distribution of ratio of stalled time to transmission time.

download flows spent more than half of their lifetime in stalls. Comparison among the three services shows that web search tends to be less affected by TCP stalls. However, given that web search is an interactive service and users are very sensitive to latency, TCP stall remains a serious concern from the web search service providers' point of view.

In summary, we find that TCP stalls heavily impair the performance of TCP and of the three services. This motivates the design of a diagnosis tool that classifies the causes of TCP stalls (see Section 3 and 4). Furthermore, the significant impact of packet loss on TCP stalled time motivates the need to mitigate timeout retransmission stalls (see Section 5).

3. DIAGNOSING TCP PERFORMANCE PROBLEMS

Monitoring and diagnosing TCP performance problems is a routine activity for Internet content providers that run thousands of servers. Designing a tool to detect TCP stalls is challenging in practice for at least two reasons. First, the statistics of TCP performance parameters have to be obtained by monitoring TCP flows. Even if many TCP variables could be obtained from the kernel of front-end servers, dumping kernel logs might impact the stability of the servers or increase the processing delay of applications. Second, server-side diagnosis tools do not have direct visibility into client-side events. In this section, we present a framework to infer the causes of TCP stalls. We have developed a TCP performance diagnosis tool, *TAPO*, that first uses the traces to extract a set of parameters for each flow (Sections 3.1 and 3.2), which are then used to identify the cause of stalls using a decision tree (Section 3.3).

3.1 Background: congestion avoidance

In the Linux implementation used by Qihoo 360, a TCP sender manages its congestion window (*cwnd*) through a state machine with 4 states, namely Open, Disorder, Recovery, and Loss. Figure 4 shows the transitions between these states. The *Open* state is the default one, in which there has been no recent duplicate ACK, SACK or congestion event. To simplify the discussion, both duplicate ACKs in newReno and SACKs are called *dupack* in this paper. The *Disorder* state happens when the sender receives dupacks but their number is less than *dupthres*. In this state, the sender does not adjust its cwnd, but transmits a new segment upon receiving each incoming ACK. When receiving *dupthres* dupacks, the sender enters the *Recovery* state, in which it reduces cwnd by one segment for each second incoming ACK, until cwnd is halved. The *dupthres* is initially set to 3, and is adjusted to the largest number of reordered packets in the flow. The sender enters the *Loss* state when the retransmission timer expires. In this state, the sender marks all outstanding packets as lost, and restarts to increase the congestion window from 1 MSS.

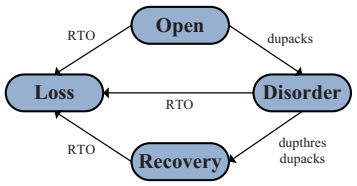


Figure 4: Transition between congestion states

3.2 TCP stalls and impacting factors

Table 2: Main parameters for TCP performance diagnosis.

Category	Parameter	Meaning
Congestion avoidance	*ca_state*	States: Open, Disorder, Recovery, Loss
Congestion window management	*in_flight*	# of in-flight packets
	sacked_out	# of packets SACKed
	holes	# of packets unACKed between ACK and SACK
	snd_una	Packet with the highest sequence number the receiver has ACKed
	snd_nxt	Next new segment the sender will transmit
	packets_out	$snd\_nxt - snd\_una$
	retran_out	# of retransmitted but not yet acknowledged packets
	cwnd	Sender's congestion window size
Retrans.	*retrans*	Times a packet is retransmitted
	lost_out	# of lost packets in the sending window
	spurious	Whether pkt is spuriously retransmitted
Receiver side	*rwnd*	receive window size
	init_rwnd	Initial *rwnd* in SYN
File transmission	*file_pos*	Position in file transfer
	pkt_dir	Direction of pkt after the stall

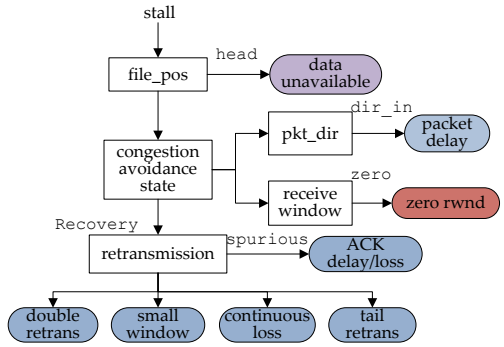


Figure 5: Tree-based classification of the causes of individual TCP stalls.

Table 2 lists the parameters that we identified as relevant to TCP stalls. They can all be obtained from the TCP traces collected at the server side and they fall into five categories shown in the left column in the table. The parameters are briefly explained in the right column of the table and we elaborate on some of the more critical parameters in the remainder of this section. The congestion avoidance state *ca_state* directly affects the growth of the congestion window. The *in_flight* size reflects the congestion window size, *sack_out* is the number of dupacks, *holes* is the number of packets that are reordered or dropped. These three parameters are relevant to congestion window management.

The next three parameters capture features related to packet retransmission. Parameter *lost_out* counts the number of lost packets in the sending window. The value is estimated by the sender through a combination of the fast retransmit threshold and the timeout retransmission heuristics [3, 13]. The estimation can be wrong due to ACK losses or packet delay. If a segment is retransmitted yet the previously transmitted one is not dropped, the receiver can transmit Duplicate SACK (DSACK) [3] to feed back on it. When a segment is retransmitted, we let *retrans* count how many times it is retransmitted. If a segment is received twice, the second one is called a *spurious* retransmission.

Receiver-side features can also affect TCP transfer performance. *rwnd* limits the window size that the server can use to transmit data. A small *init_rwnd*, e.g., of 4096 bytes, can significantly degrade TCP performance, where *init_rwnd* is the receive window size advertised by the receiver in the SYN packet. The position of the packet within the file being transferred $file\_pos$ can affect the transmission, as the server may have no data to transmit at the head or tail of a file.

3.3 TCP stall root cause analysis

We propose a tree-based classification method to determine the most likely cause of TCP stalls. Figure 5 shows the key decision points in the tree. The parameters from Table 2 that are used to make the decisions are shown as rectangles in the figure, while the inferred causes of the TCP stalls are shown as rectangles with rounded edges.

For each stall, the packet that is used for the analysis using the classification tree is the packet that ends the stall, i.e., *cur_pkt*, is the first packet that the server sends or receives after the stall. We first examine the position of this packet within the file transfer, which indicates whether there are outstanding segments to be transmitted. If the stall happens at the beginning of the file, the server may have no data to transmit, for example because the data is not available locally on the server. If it happens at the end of the file and there is a packet loss, the server may be unable to recover the packet loss due to too few dupacks.

Next, we examine the congestion avoidance state. If the state is not disorder or recovery, the stall is most likely caused by events unrelated to packet loss, such as a zero client receive window, server limitations, or packets delayed by the network. Otherwise, the reason for the stall is related to the inability of the server's TCP stack to handle the event fast enough, such as lost packets, or delayed ACK. To determine the exact cause, we need to check the number of in-flight packets and the number of times that the packet has been retransmitted.

If the current packet is a retransmission, the stall is caused by the fact that the server has to wait for a timeout. Otherwise, we examine the number of in-flight packets, which can be computed using the following formula:

$$in\_flight = packets\_out + retrans\_out - (sacked\_out + lost\_out) \ . \quad (1)$$

When the number of dupacks in TCP exceeds *dupthres*, the server retransmits the segments using fast retransmit. As such, if many packets are dropped in the sending window, the limited number of dupacks would cause a TCP stall.

TCP senders estimate the value of *lost_out* as *retrans_out*. However, since our tool can obtain information about the entire flow from the trace, it can calculate the real value of *lost_out* using DSACK. As such, we clarify the ambiguity between the timely detection of packet loss and packet reordering or delay [25] to infer the cause of stalls. If the real *lost_out* value is less than *retrans_out*, it means that packet delay was mistaken for a packet loss, and the stall was caused by the sender reducing its cwnd due to fast retransmit. If *lost_out* is greater than *retrans_out*, it means that the server misinterpreted the packet loss as a packet delay, and the stall was caused by the inability of the sender to recover from the packet loss using fast retransmit.

The number of in-flight packets *in_flight* reflects the maximum number of packets that the sender can trans-

mit. We assume that the server's sending buffer is large enough, and therefore *in_flight* is limited by either the server's *cwnd* or the client's advertised receive window *rwnd*,

$$in\_flight \approx \min(cwnd, rwnd) \ . \quad (2)$$

If *in_flight* is small (i.e., less than 4 MSS), the server cannot recover packet loss by fast retransmit due to insufficient dupacks. We then examine whether it is limited by *cwnd* or by *rwnd*. If it is limited by *rwnd*, we further examine the initial *rwnd* advertised in the receiver's SYN packet to figure out the causes behind the stalls.

Acknowledgments may be dropped or arrive at the sender after several RTTs due to jitter, which can induce a TCP stall. We classify such stalls into two types: *packet delay* and *ack delay*. Neither stall is associated with data loss. The difference between them is that a packet delay does not induce timeout retransmissions, while an ack delay causes retransmissions. Finally, it is possible that at the end of the analysis, some stalls have still not been associated with a specific cause. Such stalls are classified as "undetermined". Fortunately, they account for only 4% to 8% of the stalls. Classifying these stalls requires further research.

It is worth noting that the classification results using the above methodology are not affected by the order in which the conditions are examined. In other words, the results are deterministic. We implement the classification as the *TAPO* TCP performance diagnosis tool and released it publicly[3]. TAPO combines the three components described above: 1) reconstruction of the congestion state machine for each flow, 2) calculation of the parameters by mimicking the TCP stack and 3) classification of the stalls using the tree-based methodology. TAPO can analyze stalls in offline mode as we did. TAPO has been integrated into the TCP analysis platform in Qihoo 360 for daily maintenance of its network since Sept. 2014. Next, we leverage TAPO to analyze the TCP stalls for Qihoo 360's major services.

3.4 High-level statistics of overall stalls

We utilized TAPO to identify the causes of TCP stalls in our dataset and grouped them into three categories, based on whether they were caused by the server-side, the client-side, or network events. Table 3 presents the distribution of stalls in terms of volume and time across these categories.

Server-side stalls: Server-side stalls include those caused by the *data unavailable* and *resource constraint* causes. *Data unavailable* stalls happen at the beginning of file transfers when the requested content is not available locally on the front-end server, so it has to retrieve the content from back-end servers. *Data unavailable* accounts for about 13% of TCP stalls (in terms of time) for the software download service. Given that the software objects are static, this can be partly alleviated through improved cache management strategies and/or larger storage at the front-end server. Such stalls account for more than 20% for the cloud storage and web search services respectively, as servers have to retrieve client-specific content or dynamic search results from the back-end servers.

Resource constraint stalls (not shown in Figure 5) differ from *data unavailable* in that they happen in the middle of file transfers. They happen when the sending window is non-zero but the server was not able to provide new data to the TCP stack. Such kind of stalls are more prevalent (in terms of time) in the software download service, partially because software download requests tend to be synchronized when new software or patches are available, leading to a higher load on the servers.

Client-side stalls: Client-side stalls include those caused by the *client idle* and *zero receive window* causes. *Client idle* (not shown in Figure 5) means that the client does not request any content for a period of time after the three-way handshake or after a request has been completed, although the client does request content later, resulting in the end of the stall. Client idle stalls are more notable in cloud storage. The reason is that multiple objects might be requested via a single connection in cloud storage, which increases the possibility of stalls between two requests. Indeed, its effect on the other two services is very limited.

Zero receive window stalls prevent the server from transmitting new data. It contributes to 21.7% of the total stall time for software download, but it accounts for only a small portion of the stall time of the other two services. These stalls are actually related to a small initial receive window size (*rwnd*) advertised by receivers

Table 3: Percentage of stalls (%) in terms of volume (#) and time (T) for different causes.

		cloud stor.		soft. down.		web sear.	
category	**stall type**	#	T	#	T	#	T
server	data una.	8.5	**22.8**	7.1	13.6	65.9	**24.1**
	rsrc cons.	9.3	3.1	1.9	13.2	0.9	0.4
client	client idle	1.1	15.7	1.6	5.6	0.6	1.3
	zero wnd	7.4	7.0	26.7	**21.7**	1.6	2.2
net.	pkt delay	38.6	**17.4**	48.0	**14.9**	15.2	8.6
	retrans.	35.0	**36.3**	15.2	**31.2**	15.8	**63.4**

[3]https://github.com/tcp-tool/tapo

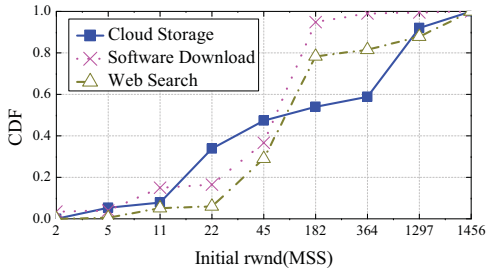


Figure 6: Distribution of initial receive windows.

in the SYN packets. Figure 6 plots the distribution of the initial receive window. We observe that 18% of the software download flows have a small initial receive window (less than 10 MSS), and some flows even experience a 2 MSS (4096 bytes) initial receive window size. A smaller initial receive window size is more likely to result in a zero receive window. To verify that small initial receive windows contribute significantly to zero receive window stalls, Table 4 presents the probability of a flow to suffer from a zero receive window when the initial receive window size ranges between 2 and 1297 MSS. We observe that flows with a smaller initial receive window tend to have a higher probability of experiencing zero receive window stalls. For example, more than half of software download flows experience zero receive window stalls when their initial window size is smaller than 11 MSS. In such a situation, the sender has to wait for the receive window to increase, and packet loss is more likely to result in timeouts because fast retransmit is not effective. We expect that the small initial *rwnd* problem is caused by the configuration of old client software.

Table 4: Percentage of flows suffering from zero rwnd as a function of the initial rwnd (MSS)(%).

init rwnd	2	11	45	182	648	1297
cloud stor.	–	–	11.5	9	7.5	1.9
soft. down.	56.5	54.2	28.4	3	–	–

Network-side stalls: The network itself degrades TCP performance by dropping or delaying packets. We distinguish network-related stalls by checking if packet loss or delay of an outstanding packet is triggering the stall. These are classified as *timeout retransmission* and *packet delay* stalls respectively. We observe from Table 3 that packet delay stalls contribute to 38.6% and 48% of the TCP stalls in terms of volume for the cloud storage and software download services respectively. However, as the durations of these stalls are shorter than an RTO, they account for only 17.4% and 14.9% of the stall time. The timeout retransmission stalls on the other hand are the most significant contributor for stall time in all three services. In particular, they account for as much as 63.4% of TCP stall time in web search. The reason is that web search flows

Table 5: Percentage of retransmission stalls (%) in terms of volume (#) and time (T) for different causes.

	cloud stor.		soft. down.		web search	
stall type	#	T	#	T	#	T
Double retr.	26.7	**45.4**	41.2	**60.8**	25.6	**41.9**
Tail retr.	4.8	5.0	0.4	0.4	44.4	**36.0**
Small cwnd	35.2	**27.3**	16.9	7.2	15.2	11.6
Small rwnd	0.4	0.3	10.6	3.7	0.87	0.3
Cont. loss	19.0	**10.1**	5.6	1.6	0.6	0.6
ACK delay/loss	6.3	6.5	14.9	**22.2**	2.1	1.8
Undeter.	7.4	6.1	10.3	4.4	11.1	7.8

tend to be short, so they suffer from more timeout retransmissions as packet loss usually happens at the tail of flows [8]. Network side stalls, especially the timeout retransmission stalls, are the only category that can be addressed by TCP, so we focus on them in the remainder of this paper.

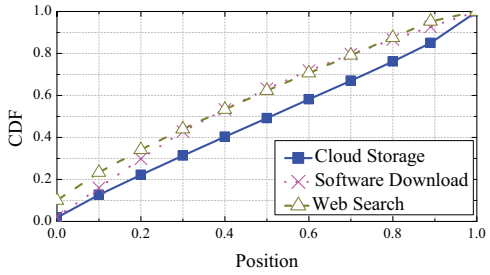
4. TIMEOUT RETRANSMISSION STALLS

Timeout retransmission stalls degrade TCP performance significantly because the TCP sender cannot transmit packets for a period equal to the RTO, which can be tens of RTTs (see Figure 1). It also forces the TCP sender to ramp up its congestion window from 1 MSS after the stall. We rely on TAPO to further break down the retransmission stalls; the results are shown in Table 5. Some stalls might meet the classification rules of several stall types. To address this issue and classify each stall into one single type, we examine the rules of stalls according to the order listed in Table 5, i.e., the rules of double retransmission stall are examined first and rules of ACK delay/loss are examined last. The results in Table 5 show that double retransmission stalls are the most expensive type for all 3 services and that tail retransmission stalls contribute almost as much for the web search service. We thus first focus these two types of stalls and then analyze the other types.

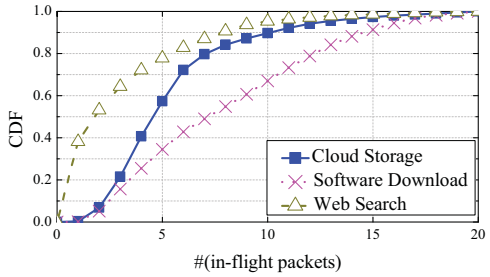
4.1 Double retransmission stalls

A double retransmission refers to a case where the retransmitted packet itself, either recovered by fast retransmit or a timeout retransmission, is dropped or delayed, causing a new timeout retransmission. From Table 5, we observe that double retransmission stalls are expensive: they account for 45%, 61%, and 42% of the stall time for the cloud storage, software download and web search services, respectively.

We first examine the context in which this type of stall happens. Figure 7a plots the CDF of the relative position where stalls happen. The relative position



(a) Relative position



(b) *in_flight* size

Figure 7: Context for double retransmission stalls.

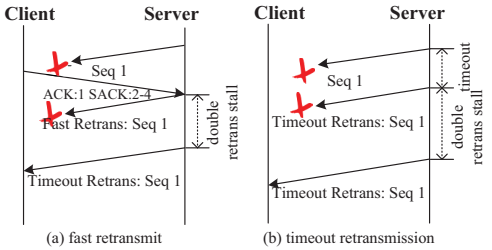


Figure 8: Two different scenarios of double retransmission stall.

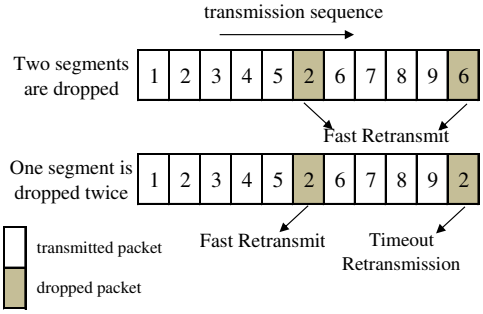


Figure 9: The transmission sequences in fast retransmit and double retransmission.

is measured as the index of the corresponding retransmitted packet divided by the number of data packets in the flow. We observe that the distribution is almost uniform, which one would expect if the stalls are caused by random packet drops. About 10% of stalls in web search happen for the first packet. The reason is that web search flows are very short – some flows even contain only one packet.

Figure 7b shows the distribution of the *in_flight* size when double retransmission stalls happens. The *in_flight* size is the number of packet that have been sent out but have not yet been acknowledged by either an ACK or SACKs. Figure 7b shows that web search tends to have a smaller *in_flight* size when double retransmission stalls happen. This is again not a surprise since web search flows are shorter and the *cwnd* often never ramps up to a high value. The other two services have a much large *in_flight* size: the median is 5 for the cloud storage and 8 for the software storage service; some stalls have *in_flight* values larger than 15. After a double retransmission stall, the sender will not only have to retransmit these packets, but it will also set the *cwnd* to 1. This is a significant drop in *cwnd* for both the cloud storage and software download services, further degrading performance.

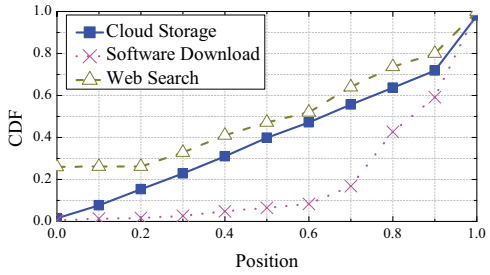
In a double retransmission stall, a data segment s is retransmitted at least twice. The first retransmission of s can be a fast retransmit or timeout driven, depending on whether there are enough dupacks as shown in Figure 8. We refer to these two stall types as *f-double stalls* and *t-double stalls* respectively. The most significant difference between the two types is that a *t-double* stall delays the data segment (and the entire flow) by two or more timeouts.

f-double stalls happen when a retransmitted packet that was triggered by a fast retransmit is lost again. Figure 9 illustrates why this result in a timeout retransmission. It shows two scenarios for a window of 9 transmissions, two of which are dropped. If two different segments are dropped, 2 and 6 in the top scenario, both of the lost packets can be recovered via fast retransmit. In the bottom scenario, segment 2 is dropped twice. The first loss is recovered by fast retransmit, but the second loss of segment 2 can only be recovered through a timeout retransmission (i.e. a double retransmission stall) in the current TCP as the TCP sender has already marked segment 2 as retransmitted. Both scenarios transmit exactly the same amount of data. We thus argue that *f-double* stalls could be eliminated through a slightly more aggressive and efficient retransmission strategies that avoid timeouts without adding further congestion to the network.

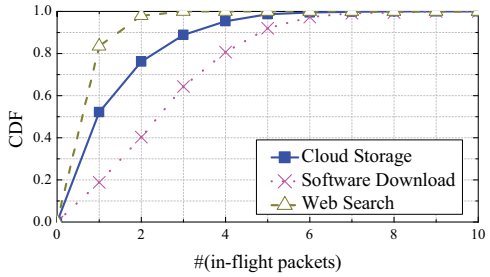
Table 6: Percentage of each type of double retransmission stalls in terms of stalled time.

	cloud s.	software d.	web search
f-double stall	62.3%	52.7%	55.6%
t-double stall	37.7%	47.3%	44.4%

Table 6 shows the percentage of each type of double retransmission stall in terms of stalled time. The three services experience similar distributions, with more than 50% of the stall time contributed by *f-*



(a) Relative position



(b) *in_flight* size

Figure 10: Context for tail retransmission stalls.

double retransmission stalls. This implies that mitigating *f-double* stalls can significantly reduce the impact of double retransmissions and improve performance. Recently proposed retransmission mechanisms like Early Retransmit [1], Tail Loss Probe [8] do not help here because they either only reduce the threshold of fast retransmit or require the TCP sender to be in the Open state. These observations motivate our proposed TCP mitigation mechanism, which will be described in Section 5.

4.2 Tail retransmission stalls

A tail retransmission stall happens as a result of a retransmission at the tail of a flow. At the end of a flow, the receiver cannot generate the three dupacks needed for fast retransmit so a timeout is needed for recovery. We observe from Table 5 that the web search service suffers significantly most from tail retransmission, accounting for 36% of all stalls. The reason is that most of the web search flows are small, so packet loss is more likely to happen in the tail of the flow.

We examine the position where the tail retransmission stall happens in Figure 10a. We observe a uniform distribution for both the cloud storage and web search services. The reasons are as follows. With the cloud storage service, multiple files may be transmitted in a single flow. Tail retransmission can happen at the end of any file, which may be in the middle of the flow. With the web search service, many of the flows contain only several packets, which will make the tail retransmission position uniformly distributed. For example, if a flow has just one packet the tail retransmission position is 0 for this flow. With the software download service, the flow size is larger than that in web search and the flow often contains only one file. Thus most of the tail retransmission positions are close to the end of flow. Figure 10b shows the number of in-flight packets when a tail retransmission happens. For the web search service, as the flow is short, most of the flows contain only one in-flight packet when the tail retransmission happens. For the other two services, the in-flight size is often no more than 3.

Table 7: Percentage of each type of tail retransmission stalls in terms of stalled time.

	cloud s.	software d.	web search
Open state	60.1%	41.3%	10.0%
Recovery state	39.9%	58.7%	90.0%

The TCP sender can be in either Open or Recovery state when a tail retransmission happens. Table 7 shows the distribution of tail retransmission stalls in both states. The distribution varies across the three services. The web search service has the smallest fraction of tail stalls in Open state, indicating a higher likelihood that at least one retransmitted packet has not been ACKed when a tail stalls happen. The Open state corresponds to a better network condition than the Recovery state, and thus a tail retransmissions happening in the Open state can possibly be mitigated via carefully retransmitting unacknowledged packets, which is the basis for TLP [8].

4.3 Other types of stalls

Small in-flight retransmission stall: The in-flight size reflects the number of packets limited by either *cwnd* or *rwnd*. We say that the in-flight size is small when $in\_flight < 4\text{MSS}$. When a packet loss occurs while the in-flight size is small, fast retransmit cannot be triggered due to too few duplicate ACKs. The sender therefore has to wait until the retransmission timer expires.

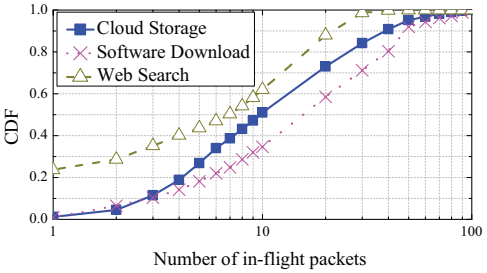


Figure 11: Distribution of in-flight size computed on each ACK.

Figure 11 shows the CDF of the number of in-flight packets for the three services. We record the *in_flight* value based on Equation 1 on *each ACK* (as opposed

to stalls). As many as 20% of in_flight values observed in the cloud storage and software download services are below 4. In other words, once an in-flight packet is dropped, the server has to wait for the retransmission timeout in 20% of the cases. For the web search, as some search results contain only one packet, about 23% of in-flight values are just 1. If the only one in-flight packet is lost, such a stall is classified as tail retransmission.

We further divide the small in-flight retransmission stalls by distinguishing which variable limits the in-flight size, *rwnd* or *cwnd*, as shown in Table 5. Small *rwnd* retransmission stalls contribute very little to the total stalls for the cloud storage and web search services. However, they account for 10.6% for the software download service. The cloud storage and web search services suffer less from this problem thanks to the relatively large initial receive window of their clients. In software download, 5% of the flows have their initial receive window set to 4096 bytes, leaving room for only 2 full segments. Any packet dropped or delayed will cause a retransmission timeout.

In contrast, small *cwnd* retransmission stalls are significant in all three services: they account for 35.2% of the retransmission stalls in terms of volume in the cloud storage service and twice as much as in the other two services. As shown in Table 3, cloud storage is more likely to suffer from timeout retransmissions that will reduce the *cwnd* to 1 MSS, leading to a higher probability to have small *cwnd* values when stalls happen. The small *cwnd* retransmission problem can possibly be solved using early retransmit [1]. However, early retransmit will not be triggered if there are two or more lost packets.

Continuous loss stall: Continuous loss means that all outstanding packets in the current window are lost. We only include cases when the number of outstanding packets is more than 3 (i.e., ≥ 4); if the window is less than four, the stall is identified as a small in-flight stalls. When all outstanding packets are lost, the server has to wait for a timeout, mark all the outstanding packets as lost, and retransmit all unacknowledged segments. Continuous loss stalls contribute to 19% of the retransmission stalls in cloud storage, but only 5.6% in software download. This type of stalls hardly happens in web search, possibly because of the small flow size. We also found that continuous loss stalls happen at any position in flows with similar likelihood.

Continuous loss is more likely to happen when a burst of packets passes through intermediate routers or middleboxes with full buffers [8]. Figure 12 plots the CDF of in-flight size with continuous loss. Surprisingly, the number of in-flight packets that are all dropped varies from 4 to as many as above 20, with a median value of 5. Sending such a large number of packets in a short time period can increase congestion and lead to the loss of all

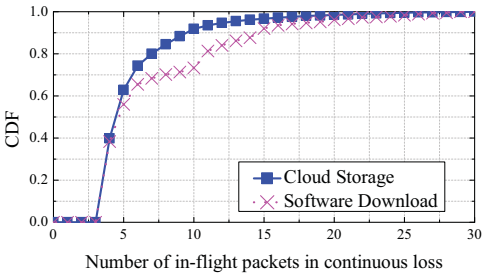


Figure 12: Distribution of in-flight size when continuous loss stalls happen.

in-flight packets. One possible way to mitigate continuous loss stalls is to reduce traffic burstiness by spacing out the transmission of packets in a window across one RTT [21].

ACK delay or loss stall: An ACK delay or loss stall happens when a sender does receive ACKs before the retransmission timer expires, while the segments are identified as not lost through DSACK. It is not possible to distinguish between an ACK delay and an ACK loss using only server-side information because in either case, the client sends a DSACK to the server. Table 5 shows that this type of stall accounts for 14.9% of the stalls in the software download service, more than twice as much than in the other two services.

ACK delay can be caused by the delayed-ACK mechanisms, which is widely used to reduce the number of ACKs that are not needed to help the sender to update its window. RFC 1122 specifies that there should be one ACK for at least every other segment and the delay must be less than $500ms$ [5]. The RTO value can be smaller than $500ms$, as the minimum RTO is $200ms$ in the Linux kernel implementation. In this case, when the in-flight size is 2 MSS, a delayed ACK would trigger a timeout retransmission because the delay is larger than a RTO. This also explains the observation that software download suffers from more ACK delay stalls. As we have seen in Figure 6, software download flows are more likely to have a small *rwnd*, so in-flight sizes of 2 MSS are more common.

5. S-RTO: TOWARDS MITIGATING TCP STALLS

We have shown in the previous section that some TCP stalls, like *f-double* and tail retransmission stalls in Open state, need to be mitigated by retransmitting unacknowledged packets slightly more aggressively, rather than waiting for an expensive timeout retransmission. While recently proposed recovery mechanisms such as TLP (Tail Loss Probe) [8] can reduce some tail retransmission and small *cwnd* retransmission stalls, it leaves other expensive types of stalls (e.g., *f-double* stalls) untouched. These observations motivate the design of our

S-RTO (Smart-RTO). We first introduce the design of S-RTO and then present our experience with S-RTO on production network servers.

5.1 Design

The negative influence of timeout retransmissions can be reduced by introducing more aggressive timers for retransmissions. However, simply shortening RTO may trigger spurious retransmissions and subsequently take the sender several RTTs to increase the congestion window from 1 MSS to the original value, during which the sender fails to fully utilize the available bandwidth. Thus, we explore an alternative design based on a second timer for a smart and slightly more aggressive retransmission before a timeout happens, and rely on RTO if the former fails to recover the lost packet.

S-RTO keeps a probe timer for each flow, but is only active when a timeout retransmission (as opposed to fast retransmit) is likely to happen. The first requirement for activating the timer is that the number of in-flight packets is relatively small, i.e. $packets\_out < T_1$, since this means that fast retransmit is likely to fail as the sender is unlikely to collect enough dupacks. The threshold T_1 is a design parameter and can be tuned according to the application properties. In the current implementation, it is set to 5 for the web search and 10 for the cloud storage service since the *cwnd* in web search tends to be smaller. The second requirement is that the current packet has not been retransmitted via a timeout retransmission, as an earlier timeout retransmission indicates a congested network. Next, we need to decide how large the value of the timer should be. If it is too long, lost packets will not be recovered in time, but if it is too short, spurious retransmissions can waste bandwidth and increase congestion. In our current implementation, we set the timer to $2 \cdot RTT$, the same as the threshold we used to identify TCP stalls.

When the probe timer is triggered, the TCP sender retransmits the first unacknowledged packet. One open question is how to manage the sender's TCP state machine. Since we assume that the unacknowledged packet is lost, the sender should enter the Recovery state. However, we must carefully manage the congestion window to avoid that *cwnd* falls below the estimated *pipe* value [4]. When $cwnd < pipe$, the sender has to wait for the arrival of acknowledgments for in-flight data. However, the estimated in-flight packets may have already been dropped by the network, causing further stalls. To this end, we introduce a new parameter T_2. If the TCP sender is not in Recovery state and the current *cwnd* value is larger than T_2, then *cwnd* is halved; otherwise the sender does not reduce the *cwnd* value. Then, the sender switches to the Recovery state. In our current implementation, T_2 is set to 5. S-RTO falls back to the native RTO mechanism for recovery if the packet retransmitted by S-RTO is dropped. The pseudo-code of *S-RTO* for handling packet loss is shown in Algorithm 1.

Algorithm 1 Packet loss handling by S-RTO.

```
1: procedure SET_SRTO
2:     if cur_pkt has not been retransmitted by native RTO
       and packets_out < T1 then
3:         timer ← 2 · RTT
4:     else
5:         timer ← native_rto
6:     end if
7: end procedure
8:
9: procedure TRIGGER_SRTO
10:    retransmit(packet)
11:    if cwnd > T2 and ca_state ≠ Recovery then
12:        cwnd ← cwnd/2
13:    end if
14:    ca_state ← Recovery
15:    timer ← native_rto
16: end procedure
```

Note that S-RTO is not limited to mitigating tail retransmission stalls. It can also help with stalls caused by double retransmission and ACK delay/loss. We have implemented the S-RTO algorithm in Linux kernel 2.6.32 with CentOS 6.2 using about one hundred lines of code. S-RTO has been deployed on several front-end servers hosting the web search and cloud storage services in Qihoo 360's production network. For evaluation purpose, we implemented both S-RTO and TLP on the same Linux kernel. The implementation enables the switch among native Linux, TLP and S-RTO through the `sysctl` command.

5.2 Performance evaluation

We used two servers hosting the web search service and two servers hosting the cloud storage service to evaluate our approach. On each server, we switched between native Linux, TLP and S-RTO using round robin with each mechanism working for 1 hour, over a period of 5 days. We captured packet-level traces during the experiments for analysis.

Table 8: Comparison of latency reduction between TLP and S-RTO.

	web search		cloud s. (short flows)	
Quantile	TLP	S-RTO	TLP	S-RTO
50	-1.2%	-1.2%	-7.3%	-19.3%
90	-0.7%	-1.3%	-13.6%	-45.0%
95	-4.7%	-2.9%	-14.4%	-21.4%
mean	-5.1%	-11.3%	-15.3%	-34.3%
#(flows)	880K	844K	56K	50K

Table 8 shows the performance of TLP and S-RTO relative to the native Linux. The latency of a flow is measured as the time between the client initiates a

request and all response packets have been acknowledged. For web search, the flows are short (less than 200KB) and flow latency is a good indicator of the user-perceived performance. Looking at the 50^{th}, 90^{th} and 95^{th} percentile latency values, we observe that TLP and S-RTO achieve comparable performance with a reduction rate less than 5%. This is because stalls in web search are dominated by tail retransmissions, of which only 10% are in the Open state (see Table 7) and can be mitigated by both mechanisms. However, S-RTO achieves a mean improvement of 11%, two times the one of TLP. The reasons is that a few (about 1%) web search flows experience very long latency in native Linux (as shown in Figure 3) due to double retransmission stalls, some of which are mitigated by S-RTO but not by TLP.

The cloud storage service has flow sizes ranging from tens of KB (e.g., control flows) to tens of MB (e.g., large file retrieval flows). We divide the cloud storage flows into short flows (less than 200KB) and large flows (larger than 200KB), and use latency and throughput as the performance metrics respectively. The latency reduction rates achieved by TLP and S-RTO for short flows are listed in Table 8. We observe a larger reduction rate by TLP in the cloud storage service compared with the web search service, because TLP can be effective for small *cwnd* stalls, contributing more on stalled time in cloud storage than in web search (see Table 5). S-RTO achieves an average latency reduction rate of 34%, more than two times the one achieved by TLP.

Our analysis of the throughput for large flows did not show a significant improvement by either mechanism: the mean throughput improvement by TLP is 2.6% and 3.7% by SRTO. One reason is that large flows last longer so some stalls, such as tail retransmission stalls, have less impact. Another reason is that in some cases, even slightly more aggressive retransmissions can increase the congestion of network, which is more likely to impact large flows than short flows.

Table 9: Retransmission packet ratio.

	Linux	TLP	S-RTO
web search	2.2%	2.3%	3.0%
cloud storage	2.7%	2.9%	3.9%

Both TLP and S-RTO can trigger unnecessary retransmissions, so we now show the ratio of retransmitted packets to all packets in Table 9. We indeed see an increased retransmission rate, probably caused by the retransmission of delayed (but not lost) packets, and possibly also as a result of having a slightly more congested network. This increase in retransmissions is however reasonable, and it does not hurt TCP fairness as the congestion window still follows the AIMD (Additive-Increase/Multiplicative-Decrease) principle of TCP. We leave the reduction of unnecessary retransmissions as future work.

6. RELATED WORK

TCP performance diagnosis has generated a lot of interest for a long time. Much of the literature [26, 11, 8, 10, 20] focuses on various aspects affecting TCP performance. Zhang et al. [26] inspected the receiver-side causes and found that the receive window size can severely limit the flow rate. Katabi et al. [11] studied the router's impact on TCP performance and found that few TCP flows exploit ECN due to lack of router support, which is still the case today. Flach et al. [8] discovered that tail losses in short flows can severely increase latency. Recent work [10, 20] also showed that middleboxes can impact TCP performance by changing or removing TCP options, or even by discarding packets with some TCP options. Packetdrill [6] is a debugging tool that aims to test the correctness and performance of TCP/IP implementations. Yu et al. [24, 18] designed a TCP flow-level monitoring and diagnosis tool to identify network inefficiencies. Compared to [18], our classification of TCP stalls is more fine-grained and we particularly focus on factors related to retransmissions.

Previous work has established that TCP performance is degraded by timeout retransmissions. In practice, most timeout retransmissions are not necessary, but can be partially replaced by FACK [13], limited transmit [2], TCP-NCL [12], or Tail Loss Probe (TLP) [8] [16]. Forward Acknowledgment (FACK) [13] estimates packet loss aggressively to better handle multiple packet losses in a window. Limited transmit [2] allows a sender to transmit a new segment after each of the first two dupacks, allowing a more efficient recovery through fast retransmit when the in-flight size is small. Early retransmit [1] proposed to explicitly reduce the fast retransmit threshold when the congestion window is small, to rapidly recover from single packet losses. TCP-NCL [12], which also recovers packet loss through an additional retransmission timer, mainly focuses on the recovery after non-congestion losses. TLP [8] on the other hand tries to transmit a probe packet if the packet in the flow tail is not ACKed within 2 RTTs. However, TLP is only active when the TCP sender is in the Open state, and thus it is unable to mitigate double retransmission stalls, which are very significant.

Recently, several new congestion control strategies [14, 23, 7] have been proposed to mitigate transmission efficiency problems. RemyCC [17, 23] relies on congestion control mechanism that is machine-generated based on the network topology and on application requirement, to achieve optimal transmission performance. PCC [7] is an adaptive congestion control that exploits empirically experienced performance to achieve higher performance. RC3 [14] proposed a technique to

use the available capacity from the first RTT to reduce flow completion time.

7. CONCLUSION

We aimed to shed more light on the causes behind the massive TCP stalls in the wild and mitigate the network-related stalls. To this end, we have developed a TCP stall diagnosis tool and applied it to packet-level traces of three popular services from a major Chinese service provider. The results show that the impact of TCP stalls and the causes behind the stalls vary across the three services. We examined timeout retransmission stalls in detail, including double retransmissions, tail retransmissions, and retransmissions under small in-flight size. Finally, we use the insights gained from our analysis to design S-RTO, an extension of TCP that helps mitigate timeout retransmission stalls. Experiments on production network servers show the effectiveness of S-RTO in improving TCP performance.

8. ACKNOWLEDGMENT

This work was in part supported by the National Basic Research Program of China No. 2012CB315801, the NSFC No. 61133015 and 61272473, the National High-tech R&D Program of China No. 2013AA013501, and CAS YZ201229. Zhenyu Li was also partly supported by NICTA Australia.

9. REFERENCES

[1] M. Allman, K. Avrachenkov, U. Ayesta, J. Blanton, and P. Hurtig. Early retransmit for tcp and stream control transmission protocol (sctp), 2010. RFC 5827.

[2] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing tcp's loss recovery using limited transmit, 2001. RFC 3042.

[3] E. Blanton and M. Allman. Using tcp duplicate selective acknowledgement (dsacks) and stream control transmission protocol (sctp) duplicate transmission sequence numbers (tsns) to detect spurious retransmissions, 2004. RFC 3078.

[4] E. Blanton, M. Allman, L. Wang, I. Jarvinen, M. Kojo, and Y. Nishida. A conservative loss recovery algorithm based on selective acknowledgment (sack) for tcp, 2012. RFC 6675 (Proposed Standard).

[5] R. Braden. Requirements for internet hosts, 1989. RFC 1122.

[6] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H.-k. J. Chu, A. Terzis, and T. Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX ATC*, 2013.

[7] M. Dong, Q. Li, D. Zarchy, B. Godfrey, and M. Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *NSDI*, 2015.

[8] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: the virtue of gentle aggression. In *ACM SIGCOMM*, 2013.

[9] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5), 2008.

[10] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In *ACM IMC*, 2011.

[11] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *ACM SIGCOMM*, 2002.

[12] C. Lai, K.-C. Leung, and V. O. Li. Tcp-ncl: a unified solution for tcp packet reordering and random loss. In *Personal, Indoor and Mobile Radio Communications, 2009 IEEE 20th International Symposium on*, pages 1093–1097. IEEE, 2009.

[13] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining tcp congestion control. In *ACM SIGCOMM CCR*, volume 26, 1996.

[14] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively cautious congestion control. In *NSDI*, 2014.

[15] V. Paxson, M. Allman, H. J. Chu, and M. Sargent. Computing tcp's retransmission timer, 2011. RFC 6298.

[16] M. Rajiullah, P. Hurtig, A. Brunstrom, A. Petlund, and M. Welzl. An evaluation of tail loss recovery mechanisms for tcp. *SIGCOMM CCR*, 45(1), 2015.

[17] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan. An experimental study of the learnability of congestion control. In *ACM SIGCOMM*, 2014.

[18] P. Sun, M. Yu, M. J. Freedman, and J. Rexford. Identifying performance bottlenecks in cdns through tcp-level monitoring. In *SIGCOMM WU-MUST Workshop*, 2011.

[19] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound tcp approach for high-speed and long distance networks. In *IEEE INFOCOM*, 2006.

[20] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *ACM SIGCOMM*, 2011.

[21] D. Wei, P. Cao, S. Low, and C. EAS. Tcp pacing revisited. In *IEEE INFOCOM*, 2006.

[22] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. Fast tcp: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (ToN)*, 14(6), 2006.

[23] K. Winstein and H. Balakrishnan. Tcp ex machina: computer-generated congestion control. In *ACM SIGCOMM*, 2013.

[24] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, 2011.

[25] M. Zhang, B. Karp, S. Floyd, and L. Peterson. Rr-tcp: a reordering-robust tcp with dsack. In *IEEE ICNP*, 2003.

[26] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. In *ACM SIGCOMM*, 2002.