High Fidelity Off-Path Round-Trip Time Measurement via TCP/IP Side Channels with Duplicate SYNs

Xu Zhang
Department of Computer Science
University of New Mexico
xuzhang@cs.unm.edu

Jeffrey Knockel
Department of Computer Science
University of New Mexico
jeffk@cs.unm.edu

Jedidiah R. Crandall
Department of Computer Science
University of New Mexico
crandall@cs.unm.edu

***Abstract*—Off-path round-trip time (RTT) measurement has many potential applications, including: improved geolocation capabilities, measuring the performance of parts of the Internet where there is not much measurement infrastructure (*e.g.*, PlanetLab), and providing data plane measurements to better understand global Internet routing. Off-path means that the measurement machine is not on the path being measured. More specifically, we can measure the RTT between essentially any two machines (A and B) on the Internet without having special access to A or B or having any presence in the path between A and B.**

Alexander and Crandall proposed a new technique for off-path RTT measurements that made fewer assumptions than previous techniques, such as King (based on DNS). Alexander and Crandall's technique assumed only that one of A or B was a standard Linux machine with at least one open port and the other replied to unsolicited SYN-ACKs with RSTs. Thus, their technique is widely applicable across many parts of the Internet. However, their technique's accuracy was severely impacted by short RTTs or high packet loss rates. In this paper, we propose an improved technique that overcomes both of these limitations.

Our new technique is shown to have 82.95% of the RTT measurement results within 10% of the actual RTT, and 91.18% of the results within 20% of the actual RTT; while the previous technique by Alexander and Crandall only had 60.7% of the results within 10% and 81.33% of the results within 20%.

I. Introduction and Related Work

Round-trip time (RTT) measurement is an important staple of Internet measurements and sees application in everything from IP geolocation to performance analysis [1]. The ability to measure RTTs completely off-path, *i.e.*, to know the RTT between virtually any given machines A and B on the Internet without having any special access to A or B, would enable measurements in parts of the world where there is not good measurement infrastructure (such as Africa, South America, and other regions where there are not many machines to measure from such as PlanetLab [2] nodes).

In 2002, Gummadi *et al.* [3] proposed King, a method to measure RTT off-path. Their technique was based on the DNS system and therefore was relatively limited in terms of what machines or networks it could measure between. Recently, Alexander and Crandall [4] proposed a technique that is based on TCP/IP side channels and therefore can be applied between any Linux server and any client that responds to SYN-ACKs with RSTs. However, the accuracy of their technique was greatly reduced when either the RTT being measured was low or the packet loss rate during the measurement was high. In this paper, we improve Alexander and Crandall's technique by overcoming both of these limitations.

We summarize our major contributions as follows:

1) We propose a significantly improved off-path TCP/IP side channel RTT measurement technique *via* active probing. Our technique is more robust to packet loss and more accurate across different RTT ranges compared to previous off-path RTT measurement techniques. Overall, 91.18% of our RTT measurement results are within 20% of the actual RTT, while the previous techniques of Alexander and Crandall [4] and Gummadi *et al.* [3] achieved 81.33% and 75%, respectively.
2) We perform a detailed analysis of sources of error across different RTT ranges when using our TCP/IP side channel off-path RTT measurement technique, and discuss certain challenges in our work as well as directions for future improvement.

Traditional RTT measurements use distributed servers to perform direct measurements. See, for example, IDMaps [5] or iPlane [6]. King [3] estimates off-path round trip time by using recursive DNS queries to DNS servers topologically close to each end point. Queen [7] uses a similar technique to measure packet loss rate. By contrast, the technique we present in this paper, and the technique it is based on [4], assume only that one machine is a Linux server with an open port and the other responds to SYN/ACKs with RSTs.

Spoofed return IP addresses and side channel inferences [8], [9], [10] have been shown to be very useful for Internet measurement, see, *e.g.*, Chen *et al.* [11]'s inferences based on IPIDs, reverse traceroute [12], or PoiRoot [13].

II. Background

A. Preliminaries

Next, we briefly outline TCP behavior specified by RFC 793 [14] relevant to our technique. Then we describe Linux's TCP/IP implementation.

1) TCP behavior: Our technique utilizes the following TCP behavior:

1) When a client sends a SYN packet to a server, this creates a half-open connection in the SYN_SENT state on the client to the server.
2) When a SYN packet is received from a client on an open port on the server, it will create a half-open connection in the SYN_RCVD state on the server to the client, and a SYN-ACK will be sent in response.
3) A SYN-ACK packet received by a machine with no corresponding half-open connection will respond with a RST.
4) A RST received by a machine in response to a SYN-ACK will close the corresponding half-open (SYN_RCVD) connection.

2) Linux's TCP/IP implementation: On Linux, each listening socket has its own *SYN backlog*, a data structure that stores that socket's half-open (SYN_RCVD) connections. The SYN backlog can only store a finite number of entries, and its maximum capacity is determined by three variables:

1) The *backlog* argument of the listen() system call
2) The runtime kernel parameter *net.core.somaxconn*
3) The runtime kernel parameter *net.ipv4.tcp_max_syn_backlog*

We represent the above three variables by m_1, m_2, and m_3, respectively. Linux determines maximum backlog capacity m as

$$m = 2^{\lceil \log_2(\max(8, \min(m_1, m_2, m_3))+1) \rceil}.$$

Argument m_1 is large in most programs. Parameter m_2 is by default 128, and parameter m_3 is, by default, determined by the memory of the system but typically ≥ 128. Thus, m is commonly 256.

To compensate for packet loss, for each SYN still in the SYN backlog, Linux will send five SYN-ACK packet retransmissions, waiting $2^{i-1}t$ seconds after the previous transmission to send the ith retransmission, where t is the initial timeout for sending the first retransmission. On Linux kernels < 3.1, $t = 3$, and so these timeouts are 3, 6, 12, 24, and 48 seconds; and on Linux ≥ 3.1, $t = 1$, and so these timeouts are 1, 2, 4, 8, and 16 seconds.

To maintain service under high load or during a denial of service attack, Linux $\geq 2.3.41$ employs an additional mechanism to evict entries from the SYN backlog to prevent it from completely filling. Linux distinguishes between *young* entries, or entries whose SYN-ACKs have yet to be retransmitted, and *mature* entries, or entries whose SYN-ACKs have been retransmitted. Every 200ms, Linux checks if the backlog is at least half-full. If it is, it first determines a threshold number T as

$$T = \begin{cases} \max(2, 5 - \lfloor \log_2 \lfloor n/y \rfloor \rfloor) & \text{when } y > 0 \\ 2 & \text{otherwise} \end{cases}$$

where y is the number of young entries in the backlog and n is the number of entries currently stored in the backlog. It will then remove up to $\lfloor \frac{2m}{5t} \rfloor$ entries whose SYN-ACKs have been retransmitted at least T times, where t is again the timeout in seconds before retransmitting the first SYN-ACK and m is the size of the SYN backlog. The intuition here is that the more mature entries there are in the backlog, the more room Linux will make for young entries.

B. Previous Work

Below we summarize the previous off-path measurement technique by Alexander and Crandall [4]. As mentioned earlier, this technique assumes that the server is a standard Linux machine with an open port and the client responses to unsolicited SYN-ACKs with RSTs. In general, to estimate the round-trip time between the server and the client using an off-path measurement machine, they use a binary search algorithm. In each round, a midpoint in the binary search is selected as a new round trip time estimate (*eRTT*). Then, a result about whether the *eRTT* is too small or too large is obtained by running the below 3 steps.

1) Send SYN packets to the server, using the measurement machine's own IP address as the source IP address, without answering ACKs to complete the three way handshake. Each SYN packet uses different source ports to ensure that it is stored in a separate backlog entry. The total number of SYN packets sent is close to half of the backlog size in the server.
2) Wait until all the packets sent in Step 1 become *mature* (see Section II), then send spoofed SYN packets to the server, using the client's IP address as the source IP address, at a specific rate determined as a function of the *eRTT*. The server, after receiving these packets, will create an entry for each spoofed SYN and then reply with SYN-ACKs to the client. The client machine will answer with RSTs to the server to reset those spoofed SYNs.
3) Infer how many of the SYN packets in Step 1 were evicted by counting the SYN-ACK retransmission packets sent for them to the measurement machine. (If a SYN entry is evicted from the backlog prematurely, it will have made fewer transmissions.) Based on the number of evicted SYNs, the server's backlog status (whether half full or not) is inferred. The backlog status implies whether the *eRTT* is too small or too large.

III. Implementation

A. Pre-requisite

1) Selecting clients: To perform off-path RTT measurement between a server and a candidate client machine, we first need to determine whether the client machine meets the assumptions of our experiment. As laid out in Section II-A, we require that our client machines reply to unsolicited SYN-ACKs with RSTs as per RFC 793 [14]. To determine this behavior, we simply send SYN-ACKs to a candidate client machine using the server's open port as the source port (but with our measurement machine's address as the source address) to mimic as closely as possible the SYN-ACKs that will be

sent from server during the experiment. If we receive RSTs in response, we consider our technique applicable to the machine.

2) Determining SYN backlog size of a Server: Determining a Server's SYN backlog size is another critical step before the off-path RTT could be measured. We implement the technique as described by Zhang *et al.* [15].

B. Evaluating an estimated RTT

Now we will set out how we utilize the server's SYN backlog as a side channel and use it to measure the RTT between the server and the client. First, we will show three steps to evaluate whether the RTT between a server and a client is less than or greater than an estimated RTT. Then we will show how to use this evaluation to measure the actual RTT.

In the first step, we fill the server's backlog with SYNs (with the measurement machine's return IP address) to use as a side channel for testing an estimated RTT between the server and the client. For simplicity, we call these SYNs *canaries*. We nearly half-fill the backlog with $c = m/2 - (\log_2(m) - 2)$ canaries. We subtract out a $(\log_2(m) - 2)$ term as a buffer to make room for any connection requests from real clients. We want this term to be at least two in order to be as accommodating as possible no matter the size of the backlog, but we do not want the subtracted term to be too large in order to minimize packet rates in the next step. Each of the c canaries has a different source port and is sent three times to ameliorate possible packet loss. We send canary packets at 100 packets per second (pps).

In the second step, we wait to ensure the number of SYN-ACK retransmissions for each canary is at least 2. Then we send spoofed SYNs to the server, using the client machine's IP address as the source address, at a rate of $3(c-1)/eRTT$ packets per second for ten seconds. Each spoofed SYN has a unique source port and sequence number. We multiply the rate by three because each packet is sent three times to ameliorate packet loss. The server will reply to each SYN with a SYN-ACK to each client machine. The client machine, after receiving from the server each unsolicited SYN-ACK, will respond with a RST to the server. These RSTs will remove their corresponding entries in the backlog. The time it takes for a RST to remove each spoofed SYN from the backlog is the RTT between the server and the client machine. If more than $(\log_2(m) - 2)$ spoofed SYNs arrive before the first RST from the client machine arrives, then the server's backlog will become more than half full and Linux will begin evicting mature entries.

In the final step, instead of counting SYN-ACK retransmissions for each canary as in [4], we send "duplicate SYNs" that we call *probes* to test for the canaries' presence in the backlog. These probes have the same TCP header values as the original canaries, except each of their sequence numbers is the sequence number of their corresponding canary minus one. Each probe is sent three times to ameliorate possible packet loss. We send probe packets at the same rate as in step one. We have two possible results, as illustrated in Figure 1:

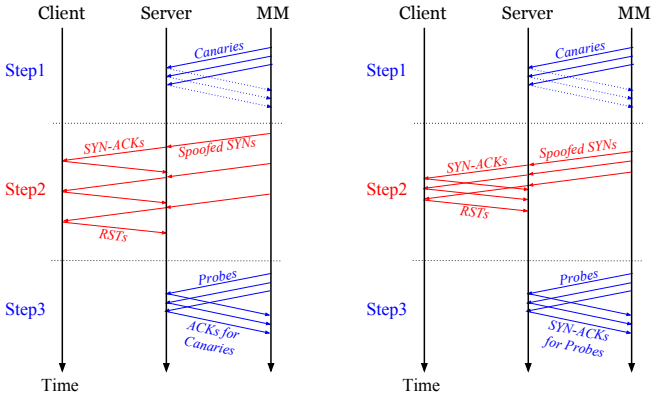

Fig. 1. Two cases in our estimated RTT evaluation.

1) In the case that, for every canary, we received ACKs in response to at least one of the probes sent to test its presence in the backlog, none of the canaries were evicted, and thus we conclude that the server's backlog was never at least half full because each spoofed SYN has been reset before up to $(\log_2(m)-2)$ spoofed SYNs arrived. Therefore, the chosen estimated RTT is larger than the actual RTT.
2) In the case that at least one canary's probes only responded with SYN-ACKs, at least one canary was evicted, and so the server's backlog was at some time at least half full because up to $(\log_2(m)-2)$ new spoofed SYNs arrived to the server's backlog before the previous had been reset. Therefore, the chosen estimated RTT is less than the actual RTT.

C. Determining the RTT

We will now show how to use our technique to evaluate an estimated RTT to determine the actual RTT between a server and a client. By performing binary search over the estimated RTT [4], we can converge on the actual RTT. However, during our experiments, we found that the actual RTT may vary. The problem of using binary search is that once a bisection decision is made, it cannot be revisited. If a wrong decision is made in early stage, the influence can be severe.

We replace the binary search with a heuristic search algorithm so that it can backtrack. Instead of testing the midpoint between the lower and upper bounds as an estimated RTT, we perform a ternary search such that we test two estimated RTTs, each evenly dividing the remaining search space into thirds. We also use two stacks to keep track of the trace points on both sides. For the left (lower) point, if the estimation is too small, the left point will be kept; if the estimation is too big, the previous left point will be returned as a new left point. The case for the right (upper) point is similar. This way we are able to backtrack when experiencing variations of actual RTT. We keep iterating this algorithm until the distance between left point and right point is less than 5ms.

IV. Experimental Setup

We duplicated the experimental setup used by Alexander and Crandall in order to make a direct comparison with their results. The measurement machine we used ran Ubuntu Server 14.04 with Linux kernel 3.13 installed. It was directly connected to an Internet backbone without any stateful firewall or egress filtering in between. We selected 15 PlanetLab nodes as our servers, evenly distributed in Asia, North America, South America, Europe, Australia, three nodes for each continent. We used PlanetLab nodes as servers in order to directly compare our off-path RTT measurement results with on-path RTTs recorded on the servers themselves; however, we did not use the servers to assist in our off-path RTT measurement technique–they are only used to verify results. The PlanetLab servers that we selected were all running Linux with kernel 2.6.x. We opened an unprivileged TCP port 15216 on every Planet lab node and confirmed that every backlog size was 256 before running our off-path RTT scan.

Once an off-path RTT measurement began, the measurement machine first copied necessary scripts to every PlanetLab server. We used these scripts to record on-path RTTs which serve as a ground truth to verify our off-path technique results. Then our measurement machine sent a remote ssh command to activate the server program to listen on port 15216 on each PlanetLab server. We used an unprivileged port 15216 to help setup tcpdump filters and also to avoid conflicts with other PlanetLab users. After that, the measurement machine created multiple threads, each using one of our PlanetLab servers to perform our off-path measurement.

Each thread first tested if a randomly generated IPv4 address replied to SYN-ACKs with RSTs. Then it activated previously copied scripts on its corresponding PlanetLab server to:

1) Start a tcpdump to record traffic between the server and the client
2) Run traceroute to the client during the experiment

The tcpdump output was used to capture the SYN-ACKs and RSTs between the server and the client. This traffic was created by our off-path technique, but the purpose of keeping it was to directly compare with our off-path result. The traceroutes were used to verify if there were any routing changes during our off-path RTT measurement. After all the environmental setup was done, we executed our off-path experiment. We used a search algorithm discussed in Section III to search the space between 0 and 3000ms. When a result RTT range was found, ssh commands were sent to terminate tcpdump and any running traceroute. Each off-path testing took about an hour to complete. We used tcpdump output to calculate an average RTT between the server and the client during our off-path measurement, and compared it with the mean of our off-path RTT result range.

V. Results

We ran off-path RTT measurement using a single measurement machine from 1 October 2015 to 6 October 2015. 728 round trip time estimates were collected during this period.

Dataset	Within 10%	Within 20%
Overall	82.95% (60.7%)	91.18% (81.33%)
RTT $>$ 25ms	83.36% (63.6%)	91.39% (83.7%)
RTT $<$ 25ms	42.86% (18.0%)	71.43% (46.1%)
RTT $>$ 100ms	86.05% (67.1%)	92.77% (87.2%)
RTT $<$ 100ms	63.92% (35.5%)	81.44% (58.06%)
25ms$<$ RTT $<$ 100ms	65.56% (43.5%)	82.22% (63.5%)

TABLE I
Percent of measurements within given percent of actual round trip time. Previous measurement results are in parentheses.

Dataset	Within 10%	Within 20%
Overall	89.74% (68.87%)	96.92% (90.84%)
RTT $>$ 25ms	90.6% (72.1%)	97.39% (92.7%)
RTT $<$ 25ms	42.86% (16%)	71.43% (60.0%)
RTT $>$ 100ms	93.15% (75.9%)	97.82% (96.0%)
RTT $<$ 100ms	73.91% (39.3%)	92.75% (69.05%)
25ms$<$ RTT $<$ 100ms	77.42% (49.3%)	95.16% (72.9%)

TABLE II
Percent of measurements within given percent of actual round trip time with no packet loss. Previous measurement results are in parentheses.

We determined by tcpdump output that 36 (4.5%) experiments showed no RST traffic from the client to the server, so we excluded these experiments from our analysis, leaving 692 data points. Using the method we discussed in Section III, we calculated the corresponding on-path RTT for each of our off-path RTT estimates. Figure 2 shows the CDF plot for the estimated RTTs divided by actual RTTs.

Our technique has higher accuracy than previous off-path RTT measurement techniques. 82.95% of RTT measurement results are within 10% of the actual RTT, and 91.18% of the results are within 20% of the actual RTT. In contrast, the previous technique by Alexander and Crandall has 616 round-trip time estimates, 60% of them within 10% and 81.33% of the results within 20%. Table I and Table II show a direct comparison of our result to the previous technique developed by Alexander and Crandall. King, another off-path latency estimation tool that used DNS, has less than 20% of error in over three quarters (75%) of estimates and 10% of error in two-thirds (67%) of the estimates. Below we investigate the accuracy of our technique as well as analyzing the sources of error.

A. Performance for low RTTs and high RTTs

Figure 3 shows the accuracy of our estimates for actual RTTs less than 25ms. In this RTT range, our measurement has 42.86% within 10% of actual RTTs, and 71.43% within 20% of actual RTTs. Comparing with previous results, we have a 24.86% and 25.33% increase of accuracy, respectively. Our estimates for small RTTs are bounded by 50% of actual RTTs, within an error less than 7.77ms. Meanwhile, our results show the capability of measuring off-path RTTs accurately even for a RTT less than 10ms, *e.g.*, the smallest RTT we measured had actual RTT 7.16ms, and our estimate was 7.01ms. We found that 85.7% of our low RTT results overestimates the actual RTTs. The reason for this is that the packet intervals in low RTT estimates are usually small, and any small delay of RSTs

(such as a packet loss of the SYN-ACK or RST) may cause the server's backlog to become more than half full unexpectedly. As a result, the search algorithm will falsely believe that the estimate is too small, and try a larger value in the next round.

Our technique performs better when the actual RTTs are greater than 100ms. Among our RTT estimates, 86.05% are within 10% and 92.77% are within 20%, compared to the previous results of 67.1% within 10% and 87.2% within 20%. Figure 4 shows the accuracy of our estimates for this RTT range. One major source of error in our results is from the variations of actual RTTs. Large RTTs typically have more variations than small RTTs. We found that for the 7.23% of results that are beyond 20% of actual RTTs, 58.14% of them have a standard deviation of RTTs 25ms or larger; while for the 92.77% results that are within 20%, only 11.96% of them have a standard deviation of RTTs 25ms or larger. For cases when a standard deviation of RTTs is 250ms or more, only 10% of results are within 20%; while for cases when a standard deviation of RTTs 25ms or less, 96.43% of results are within 20%. One possible reason for large variation in RTTs is route change. We found in our data that, although route changes existed in many of our experiments, it happened most frequently for large RTTs.

B. *Effects of packet loss*

We used active probing to ameliorate the influence of packet loss between the measurement machine and server, as previously discussed. For the purpose of this paper, we are interested in packet loss between the server and client. Figures 2, 3, 4, and 5 show a comparison between the full data set case and the case without packet loss. The sample for our low RTT estimates is small and did not experience packet loss, so in Figure 3, the no packet loss case shows the exact same pattern as a normal case. In other RTT ranges, the no packet loss case had an increase of accuracy from about 5% to 12%. To understand how packet loss between server and client influences our results, we consider a case where the server sent SYN-ACKs and the RSTs from client are lost. As a result of that, the server's backlog is more than half full and it evicts canaries. From the measurement machine's point of view, the estimate is less than the actual RTT.

VI. Discussion and Conclusion

For cases where actual RTTs are greater than 25ms and less than 100ms, our measurement performs better than the case of small RTTs, with 65.56% within 10% and 82.22% within 20%, as shown in Figure 5. For comparison, Alexander and Crandall had 43.5% within 10% and 63.5% within 20%. Note that about 10% of our results in this range underestimate the actual RTTs. The reason for this is that the backlog was completely filled by the first arrival of spoofed SYNs and refused any newly arriving spoofed SYNs. At the same time, RSTs from the client reset the spoofed SYNs currently in the backlog before the server's eviction happened. That is to say, this problem happens only when 1) The search algorithm makes an aggressive move to the left, causing the server's

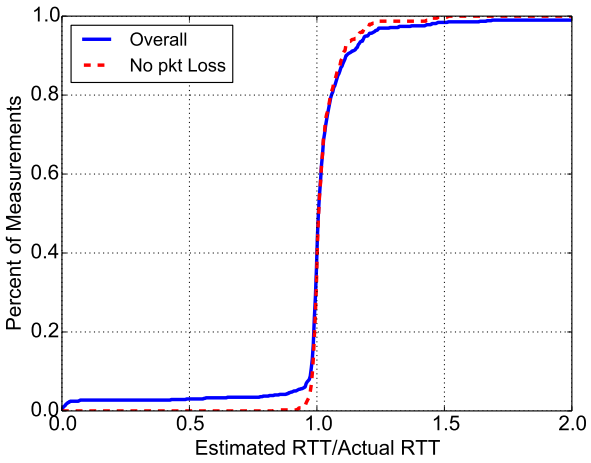

Fig. 2. Overall accuracy of Off-Path RTT estimates.

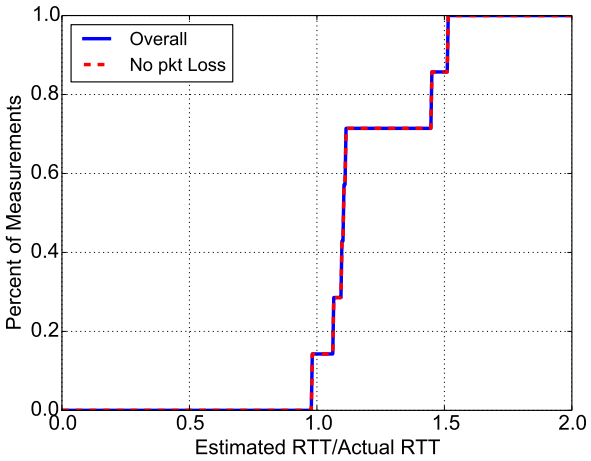

Fig. 3. Effects of packet loss on RTT estimates (actual RTT <25ms).

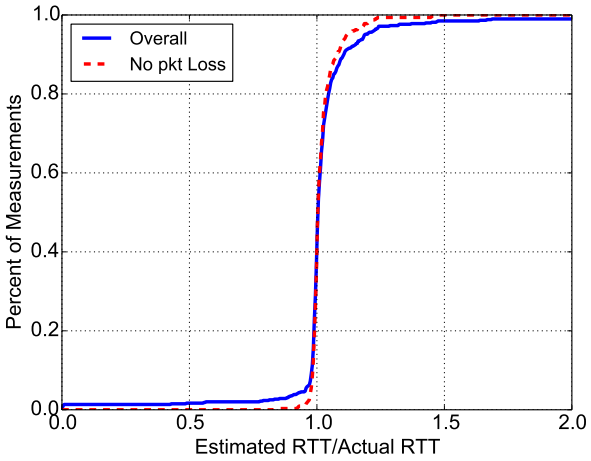

Fig. 4. Effects of packet loss on RTT estimates (actual RTT >100ms).

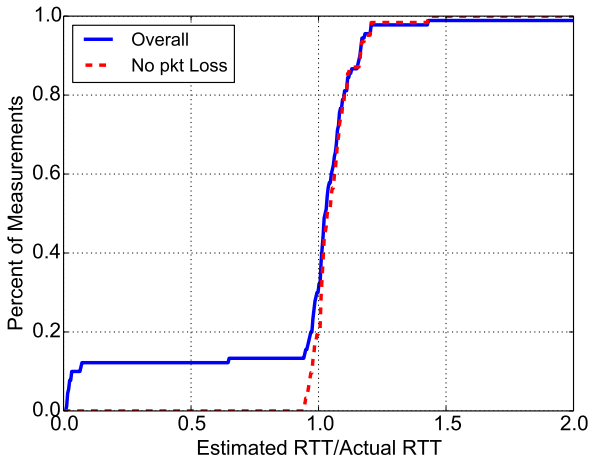

Fig. 5. Effects of packet loss on RTT estimates (25ms <actual RTT <100ms).

backlog to quickly fill; and, 2) The actual RTT is less than 200ms (the SYN-ACK timer), which means RSTs could reset spoofed SYNs before eviction happens. For large RTTs, the second requirement cannot be met because RSTs come back after the server starts the eviction; for small RTTs, the first requirement cannot be met because RSTs come back quickly enough and the server's backlog will not be full. Therefore, we saw these kinds of results in this range.

Our technique ameliorates packet loss by sending each packet three times. However, the two additional SYNs should not cause extra resource allocations since together they create at most one connection on the server. Each SYN packet's size is 60 bytes, and the network burst of traffic created by our scan is less than 150 kilobytes per second per server. During each round of the scan, the server's SYN backlog could only possibly be more than half full (but not completely full) for less than 200ms, and so it should not cause denial of service. However, in some rare cases, as we discussed above, the server's backlog was full due to the search algorithm making an aggressive move. To improve our technique, the search algorithm needs to be less aggressive when testing RTTs less than 100ms.

We have presented an improved method for off-path RTT measurement that is more robust to packet loss and more accurate for low RTTs than Alexander and Crandall's method. Opportunities for improvement to the technique include accounting for Linux's SYN backlog pruning timing, adapting the technique to a broader set of server operating systems, and decreasing the packet rate so that the traffic is not perceived as invasive. Our results presented in this paper show that managing SYN backlog entries can enable significant improvements in accuracy over Alexander and Crandall's technique.

VII. Acknowledgments

We would like to thank the anonymous reviewers and Antonio Espinoza for valuable feedback. We would like to thank Ben Mixon-Baca and Geoff Alexander for helpful discussions about previous off-path RTT measurement techniques. This material is based upon work supported by the U.S. National Science Foundation under Grant Nos. #1314297, #1420716, #1518523, and #1518878.

References

[1] R. Landa, J. Araujo, R. Clegg, E. Mykoniati, D. Griffin, and M. Rio, "The large-scale geography of Internet round trip times," in *IFIP Networking Conference, 2013*, 2013, pp. 1–9.

[2] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: An overlay testbed for broad-coverage services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, pp. 3–12, Jul. 2003. [Online]. Available: http://doi.acm.org/10.1145/956993.956995

[3] K. P. Gummadi, S. Saroiu, and S. D. Gribble, "King: Estimating latency between arbitrary Internet end hosts," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 2002, pp. 5–18.

[4] G. Alexander and J. R. Crandall, "Off-path round trip time measurement via TCP/IP side channels," in *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2015, pp. 1589–1597.

[5] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang, "IDMaps: A global Internet host distance estimation service," *Networking, IEEE/ACM Transactions on*, vol. 9, no. 5, pp. 525–540, 2001.

[6] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "iPlane: An information plane for distributed services," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 367–380.

[7] Y. A. Wang, C. Huang, J. Li, and K. W. Ross, "Queen: Estimating packet loss rate between arbitrary Internet hosts," in *Passive and Active Network Measurement*. Springer, 2009, pp. 57–66.

[8] R. Ensafi, J. C. Park, D. Kapur, and J. R. Crandall, "Idle port scanning and non-interference analysis of network protocol stacks using model checking." in *USENIX Security Symposium*, 2010, pp. 257–272.

[9] Z. Qian and Z. M. Mao, "Off-path TCP sequence number inference attack-how firewall middleboxes reduce security," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 347–361.

[10] R. Ensafi, J. Knockel, G. Alexander, and J. R. Crandall, "Detecting intentional packet drops on the Internet via TCP/IP side channels," in *Passive and Active Measurement*. Springer, 2014, pp. 109–118.

[11] W. Chen, Y. Huang, B. F. Ribeiro, K. Suh, H. Zhang, E. d. S. e Silva, J. Kurose, and D. Towsley, "Exploiting the IPID field to infer network path and end-system characteristics," in *Passive and Active Network Measurement*. Springer, 2005, pp. 108–120.

[12] E. Katz-Bassett, H. V. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. Van Wesep, T. Anderson, and A. Krishnamurthy, "Reverse traceroute," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855726

[13] U. Javed, I. Cunha, D. Choffnes, E. Katz-Bassett, T. Anderson, and A. Krishnamurthy, "PoiRoot: Investigating the root cause of interdomain path changes," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 183–194.

[14] J. Postel, "Transmission Control Protocol," Internet Requests for Comments, RFC Editor, RFC 793, September 1981. [Online]. Available: http://tools.ietf.org/html/rfc793

[15] X. Zhang, J. Knockel, and J. R. Crandall, "Original SYN: Finding machines hidden behind firewalls," in *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2015, pp. 720–728.