New Methods for Passive Estimation of TCP Round-Trip Times

Bryan Veal, Kang Li, and David Lowenthal

Department of Computer Science
The University of Georgia
Athens, GA 30602, USA
{veal,kangli,dkl}@cs.uga.edu

Abstract. We propose two methods to passively measure and monitor changes in round-trip times (RTTs) throughout the lifetime of a TCP connection. Our first method associates data segments with the acknowledgments (ACKs) that trigger them by leveraging the TCP timestamp option. Our second method infers TCP RTT by observing the repeating patterns of segment clusters where the pattern is caused by TCP self-clocking. We evaluate the two methods using both emulated and real Internet tests.

1 Introduction

Round-trip time (RTT) is an important metric in determining the behavior of a TCP connection. Passively estimating RTT is useful in measuring the the congestion window size and retransmission timeout of a connection, as well as the available bandwidth on a path [1]. This information can help determine factors that limit data flow rates and cause congestion [2]. When known at a network link along the path, RTT can also aid efficient queue management and buffer provisioning. Additionally, RTT can be used to improve node distribution in peer-to-peer and overlay networks [3].

Our work contributes two new methods to passively measure RTT at an interior measurement point. The first method works for bidirectional traffic through a measurement point. It associates segments from the sending host with the ACK segments that triggered their release from the sender. Our method uses TCP timestamps to associate data segments with the acknowledgments that trigger them. Since the other direction is easy—associating acknowledgments with the data segments they acknowledge—we can obtain a three-way segment association. Thus, we have a direct and simple solution that can collect many RTT samples throughout the lifetime of the connection.

There is no guarantee that the network route is symmetric, so only one direction of flow may be available to the measurement point. We introduce a second method to monitor a data stream and detect cyclical patterns caused by TCP's self-clocking mechanism. Because of self-clocking, a TCP connection's segment arrival pattern within one RTT is very likely to repeat in the next RTT.

We use algorithms that employ autocorrelation to find the period of the segment arrival pattern, which is the RTT. As with our previous method, we can take samples throughout the lifetime of a TCP session.

We show both methods to be accurate by evaluating them using both emulated and real network traces. For the emulated traces, we tested RTT estimates with network delays ranging from 15ms to 240ms, as well as with competing traffic over a bottleneck link using 0–1200 emulated Web users. The average RTT estimate for each delay tested was always within 1ms of the average RTT reported by the server. The maximum coefficients of variation (standard deviation/mean) were 3.79% for the timestamp based method and 6.69% for the self-clocking based method. Average RTT estimates for the tests with competing traffic were all within 1ms for the timestamp based method and 5ms for the self-clocking based method.

We also tested our RTT estimation methods with downloads from Internet FTP servers. Out of seven servers, the maximum coefficient of variation was 0.11% for the timestamp based method. For five of those servers, all RTT estimates for each server were within 1ms of each other using the self-clocking based method, and their average estimates were within 2.2ms of the average estimates from the timestamp based method.

2 Related Work

The method [4] uses segment association during the three-way handshake that initiates a TCP connection, as well as during the slow start phase. This takes advantage of the fact that the number of data segments sent can be easily predicted in advance. However, during the congestion avoidance phase, it is hard to predict the RTT based on the number of segments. Our method can associate a data segment with the ACK that triggered it, and thus it can follow changes in the RTT throughout the lifetime of a TCP session.

There is a method [5] to associate, throughout the lifetime of a session (including during congestion avoidance), a data segment with the ACK segment that triggered it. This method first generates a set of all possible candidate sequences of ACKs followed by data segments. Sequences that can be determined to violate basic TCP properties are discarded. The method then uses maximum-likelihood estimation to choose from the remaining possible sequences. This method is complex and would be cumbersome to implement as a passive estimation method at a device such as a router. Our method of using TCP timestamps to associate segments is simpler and more direct.

A previous work [6] introduces a method to passively measure RTT by mimicking changes in the sender's congestion window size. The measurement point must accurately predict the type of congestion control used: Tahoe, Reno, or NewReno. The accuracy of the estimate is affected by packet loss, the TCP window scaling option, and buggy TCP implementations. Our method avoids these difficulties by directly detecting the associations between segments.

3 TCP Timestamps

Both our RTT estimation methods use the TCP timestamp option. The original purpose of the option was to estimate the RTT at the sender for the purpose of deriving the retransmission timeout. The option adds two fields to the TCP header: timestamp value (*TSval*) and timestamp echo reply (*TSecr*). *TSval* is filled with the time at which the segment was sent, and *TSecr* is filled with the *TSval* of most recently received segment, with some exceptions. If a segment is received before a segment previous to it in the sequence arrives, leaving a hole, then the timestamp of the segment previous to the hole in the sequence is echoed. When this hole is filled by an out-of-order segment or a retransmission, the timestamp of the segment that fills the hole is echoed rather than the timestamp of a segment later in the sequence.

3.1 Timestamp Deployment

For timestamps to be useful for passive RTT measurement, the option should have a wide deployment and its implementation should be consistent across different hosts. We have developed a tool that can test the timestamp option on remote Web servers. This tool was run on 500 servers taken from the Alexa Global 500 list [7]. Of these, 475 servers responded to HTTP requests from our tool.

The tool tests for timestamp deployment by sending SYN segments with the timestamp option enabled and checking the SYN/ACK response for timestamps. Of the 475 responding servers, 76.4% support the TCP timestamp option. We expect timestamp deployment to increase over time. Furthermore, the self-clocking based RTT estimation method does not have to rely on TCP timestamps as the time unit used to associate segments into clusters. Other time units are possible, such as arrival time at the measurement point. We will address this possibility in future work.

3.2 Implementation Consistency

The tool also tests for implementation consistency. It tests the exceptions to echoing the most recent timestamp, described above. The tool sends three data segments with the last two out of order in sequence. The server should indicate the hole by sending a duplicate ACK with the timestamp of the first segment. When the client sends the last segment that fills the hole, the server should echo its timestamp. Of the servers tested that support TCP timestamps, 100% echoed the correct timestamp in both cases.

Another possible implementation error is to echo the timestamps of only data segments, disregarding ACKs that carry no data. Our tool tests for this possibility by sending an HTTP request to the server, receiving a data segment, sending an acknowledgment, and receiving more data. The congestion window is throttled to one byte to ensure that one segment is sent at a time. The second data segment from the server should echo the timestamp of the ACK and not

the timestamp of the HTTP request. Of the servers tested, 99.4% correctly echo the timestamp of the the ACK.

3.3 Timestamp Granularity

The granularity chosen for TCP timestamps is implementation dependent. A fine granularity increases the accuracy and usefulness of both our RTT estimation methods, as shall be explained in later sections. Our tool tests granularity by sending data segments to the server at a known interval and then measuring the difference between the timestamps of the ACKs the server sends in response. Table 1 shows the distribution of timestamp granularity across the servers tested that support the timestamp option.

Table 1. Distribution of timestamp granularity

Granularity	Percent of Servers
500ms	0.6%
476ms	0.6%
100ms	36.9%
10ms	54.8%
1ms	7.2%

4 RTT Estimation Using Timestamps

Our first RTT estimation method method requires finding *associations* between TCP segments at an interior point along the route between the sender and receiver. The first segment in an association is a data segment from the sending end of a TCP connection. The second is the ACK segment from the receiving end that acknowledges receipt of the data segment. The third segment in the association is the next data segment from the sender, which is triggered when it receives the ACK. This assumes that the sender always has enough data ready to fill the congestion window as soon as more room becomes available.

Since multiple data and ACK segments may be in transmission concurrently, it is not obvious at an interior point which segments from one host have been triggered by the other. For the interior point to recognize an association, a segment must carry identification of the segment that triggered it. For the case of a data segment triggering an ACK, the acknowledgment number carried by the ACK is derived from the sequence number of the data segment. Thus the interior point can associate the two segments. However, the sequence numbers of ACK segments remain constant as long as the receiver sends no data. Because of this, it is impossible to use the acknowledgment number of a data segment to identify the ACK that triggered it.

The measurement point may use TCP timestamps instead of sequence numbers to associate segments. Timestamps are used only for association and not

for calculating the RTT. Both the sender and receiver of a TCP session echo the most recently received timestamp, with minor exceptions in the cases of loss and segment reordering. The measurement point records the timestamps, their echoes, and arrival times of segments in each direction to estimate the RTT.

Figure 1 provides an example. The sender transmits a segment at time *s1*. It arrives at the interior measurement point at time *m1*. The receiver responds with an ACK at time *r1* and echoes the sender's timestamp, *s1*. The measurement point recognizes *s1* in both segments and makes an association. Upon receiving the ACK, the sender transmits more data at time *s2* and echoes the receiver's timestamp, *r1*. The measurement point receives this segment at time *m2*. It recognizes *r1* in both segments and forms an association. Having associated all three segments, the measurement point estimates the RTT to be $m2 - m1$.

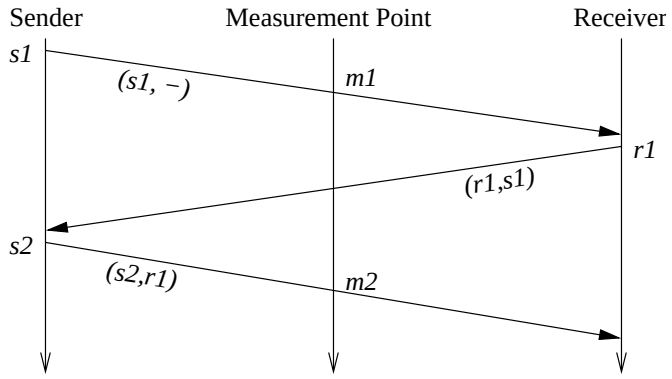

Fig. 1. Association of segments using TCP timestamps

4.1 Constraints

Timestamp Granularity. The granularity of timestamps depends upon the TCP implementation of the sender. Even with a granularity as fine as 1ms, a burst of segments sent in a short interval may carry the same timestamp. The receiver may acknowledge parts of the burst at different times, but all the ACKs would carry the same timestamp echo. It would be difficult for the interior point to determine which data segments caused which ACKs. Since the first segment carrying a timestamp may be associated safely with the first segment carrying its echo, the algorithm only considers the first arriving segment with a particular timestamp and others with identical timestamps are discarded. However, a coarser timestamp granularity increases the the number of segments with identical timestamps, and thus allows for fewer measurements to be taken.

A side effect of preventing associations with ACKs containing old timestamps is that later ACKs containing the same timestamp echo as the discarded segment

may be used to make an association, leading to an overestimate. To prevent this situation, only the first ACK with any particular timestamp echo is used to make associations.

Packet Loss. When the receiver is missing data due to packet loss, it sends duplicate ACKs. Since timestamp echoes are not updated when the receiver is missing data, this problem is automatically eliminated by discarding associations with ACKs that contain old timestamp echoes. However, when selective acknowledgments are enabled, overestimates can still occur. This problem is avoided by not considering selective ACK segments (which are only produced when loss is present), when making associations.

Interactive Sessions. This algorithm does not consider situations where the sender has no new data available when it receives and ACK. Such sessions are typically for interactive applications, such as ssh or telnet. Though not implemented here, it should be possible to obtain RTT estimates for interactive sessions based on some simple application heuristics. For example, in a typical session, when a user types a key, the character is sent to the server. Then the server echos the character back to the client to be displayed on the terminal. The client then responds with an ACK. An interior measurement point could take advantage of this to make an association for the three segments and estimate the RTT.

It is still possible that the sender has some delay in sending more data during a bulk transfer which could lead to an inflated RTT estimate at the measurement point. To filter such measurements, we have devised a method that tracks current maximum RTT for the session between the measurement point itself and each of the two hosts. These RTTs would be taken for only data-ACK pairs to avoid any possibility of sender delay. Any RTT estimates greater than the current sum of the two maximum delays would be discarded as an inflated estimate. We plan to evaluate this method as future work.

Asymmetric Routing. Though the RTT estimation algorithm requires both data and ACKs, there is no guarantee that both directions of traffic will follow the same route. However, it is still possible to obtain estimates using the second algorithm described in the next section.

5 RTT Estimation Using Self-Clocking Patterns

Our second algorithm detects patterns in a bulk data stream caused by a mechanism in TCP known as *self-clocking*. Capturing ACKs from the receiver is not required, so this algorithm maybe used for either asymmetric or symmetric routes. With self-clocking, the bulk data sender produces more data each time it receives an ACK, and the receiver sends an ACK each time it receives more

data. Because of this, the the spacing between bursts of segments is likely preserved from one round trip to the next. Although packet losses and competing traffic could change the spacing and cause bursts to split or merge, the changes do not always happen frequently, and the bursts tend to persevere for at least a few round trips after each change. There may be multiple bursts of segments per round trip, and their size and spacing generally repeat every RTT. This algorithm detects the repetition of these burst-gap patterns to find the RTT. An example of such a pattern is shown in Fig. 2.

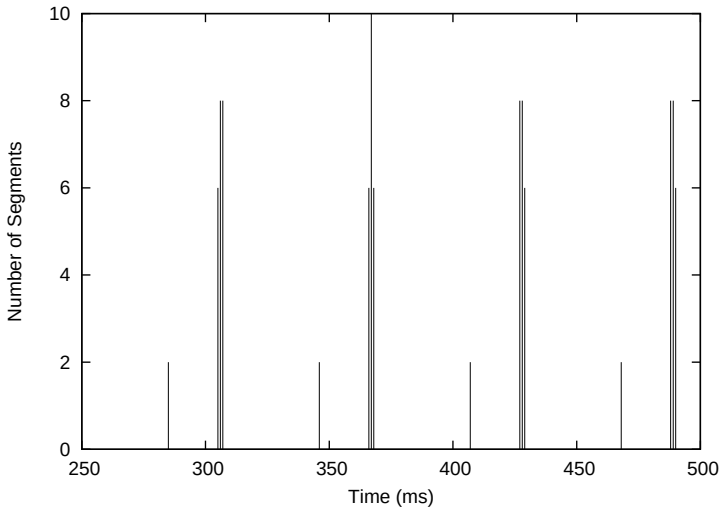

Fig. 2. Burst-gap pattern caused by self-clocking

Discrete autocorrelation measures how well a data set is correlated with itself at an offset determined by the lag (l). If the correlation is strong, then the data matches its offset closely. Figure 3 shows the autocorrelation strengths for the data in Fig. 2. The strong correlation at 61ms corresponds closely to the RTT (which is 60ms).

Our algorithm uses autocorrelation to make RTT estimates. The algorithm repeats the RTT estimation once per *measurement interval*, T, which is supplied as a parameter. During this interval, the number of packets that arrive with timestamp t is stored in array $P[t]$ ranging from 0 to $T-1$. Once the count is complete, the discrete autocorrelation $A[l]$ is computed for each lag l from 1 to $l/2$. The RTT estimate is computed as $\max(A)$.

This process is repeated to produce multiple estimates throughout the session. The number of estimates depends upon the duration of the measurement interval and the duration of the session. However, more estimates may be taken by allowing measurement intervals to overlap.

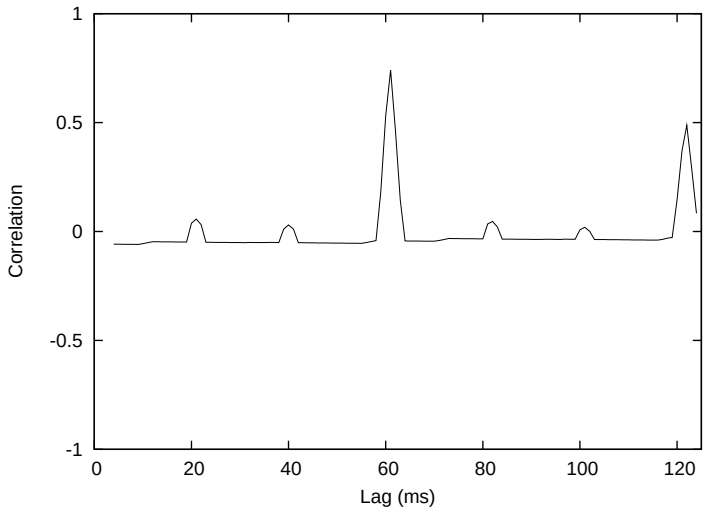

Fig. 3. Autocorrelation for self-clocking pattern

5.1 Constraints

Timestamp Granularity. According to a theoretical limit known as the *Nyquist period*, it is only possible to measure RTTs at least twice the TCP timestamp granularity. For instance, if the granularity is 10ms, we can only detect RTTs of at least 20ms. This is a problem with timestamp granularities of 100ms or more. Although we do not explore it in this paper, a possible solution is to use arrival times at the measurement point rather than TCP timestamps from the sender.

Harmonic Frequencies. A consequence of a burst-gap pattern that repeats every RTT is a strong autocorrelation at multiples of the RTT that is sometimes stronger than that of the actual RTT. Rather than assuming that the strongest correlation corresponds to the RTT, the algorithm starts with the lag at the strongest correlation, s, and compares it to $A[\frac{s}{2}], A[\frac{s}{3}], A[\frac{s}{4}], \ldots$, until a certain limit is reached. If the correlation at the fractional lag is at least a certain percent of the actual lag, then that lag is considered the RTT instead. The limit of fractional lags and the percent of the maximum correlation are both provided as parameters to the algorithm.

Measurement Interval. The measurement interval chosen places an upper bound on the maximum RTT that can be measured. Autocorrelation becomes unreliable at a lag of half the measurement interval, since two complete round trips are needed to fully compare one round trip with its offset.

Delay Variation. While a smaller measurement interval may miss large RTTs, a larger measurement interval decreases the amount RTT variation that can be detected. If multiple strong correlations exist at different lags, this may indicate different RTTs at different times within a measurement interval. Our algorithm can report multiple candidate RTT estimates within an interval along with their correlation strengths.

Noise Effects on Self-Clocking Patterns. Congestion caused by competing traffic or other network conditions may disrupt the burst-gap pattern caused by self-clocking. One consequence of this is a high correlation at very small lags. This problem can be corrected by allowing a lower bound on the RTT to be specified as a parameter. We evaluate the effects of competing traffic on our algorithm in the next section.

6 Evaluation of RTT Estimation Methods

To evaluate these two passive RTT estimation methods, we have implemented them in a toolkit called TCPpet (TCP Passive Estimation Toolkit). Our implementations take traces captured by `tcpdump` and generate RTT estimates. The implementation of the timestamp method takes a trace with bidirectional TCP traffic and generates as many RTT estimates as the timestamp granularity will allow.

The implementation of the self-clocking method can use a trace with bidirectional or unidirectional traffic. Harmonic frequency detection is enabled for all experiments. If $\frac{1}{2}$, $\frac{1}{3}$, or $\frac{1}{4}$ of the lag having the strongest correlation has at least 75% of that correlation, it is taken as the RTT measurement. Note that 75% is measured from the minimum correlation in the measurement interval, which may be negative, instead of from zero. These values were chosen for good overall performance with the FTP downloads described later. Additionally, RTTs are assumed to be at least 10ms. Lags less than 10ms are not considered in order to correct for strong autocorrelations caused by noise effects. Note that if the Nyquist period is higher than 10ms, it becomes the minimum instead.

6.1 Emulation Test with a Single Flow

We have evaluated both RTT estimation methods with different network delays over an emulated network. The network consists of four machines: a sender, a receiver, and two routers, creating a three-hop route between the sender and receiver. Both the sender and receiver have timestamp granularities of 1ms. NIST Net [8] was used to add delay along the route by adding delay to both directions of traffic on each of the two routers. Thus, there are a total of four sources of delay along the route. If, for instance, a total round trip delay of 100ms is desired, then a delay of 25ms is added to each of the four points.

Traces were taken with `tcpdump` on the router closest to the sender. These traces were taken from the network interface that connects to the receiver's

router, so that segments from the sender are delayed both before and after being recorded by `tcpdump`.

TCP data transfers were generated by the `ttcp` utility, which has been instrumented to report RTT estimates from the server's TCP implementation. All transfers were 16MB of data generated by `ttcp`.

Figure 4 shows all the RTT estimates for a network trace with a 60ms delay. Although the trace is longer, we only show the first 2 seconds for clarity. It includes estimates made by the server as well those made by the two passive estimation methods. A 250ms measurement interval was heuristically chosen for the self-clocking method. We plan to find a general default measurement interval as future work. As shown in the figure, nearly all the RTT measurements are close to those reported by the server. Note that the first few estimates for the server were influenced by preexisting state in the TCP implementation. After the 1s mark, the server estimates level off throughout the duration of the trace.

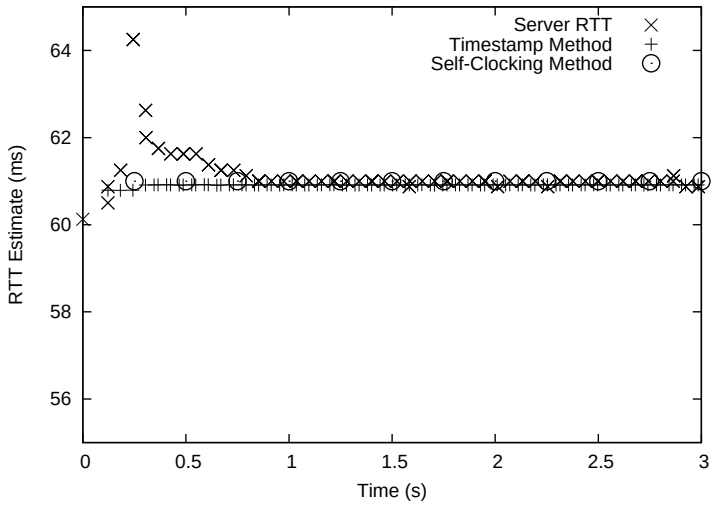

Fig. 4. RTT estimates for an emulated network trace with a 60ms delay

RTT estimates were taken for traces with 15, 30, 60, 120, and 240ms delays. Figure 5 shows the average of all the RTT estimates reported by each method for each trace. Here, a 500ms measurement interval was chosen for the self-clocking estimation method to accommodate possible RTTs up to 250ms. As shown in the figure, the averages are nearly identical. The largest coefficients of variation (standard deviation/mean) occurred for the 15ms delay trace, which were 3.79% for the timestamp based method and 6.69% for the self-clocking based method.

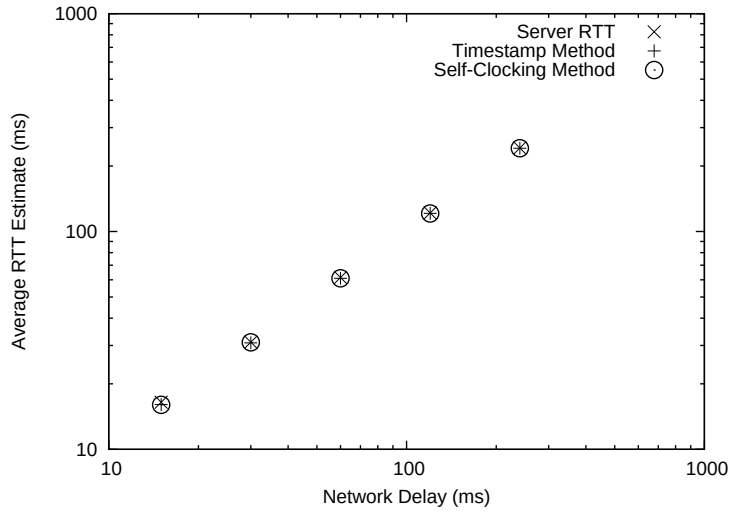

Fig. 5. Emulated network traces with varying delay

6.2 Emulation Test with Competing Flows

While the previous experiments show that our methods work well with different delays, real networks have conditions such as bottlenecks and competing traffic. The emulated network used previously was modified using NIST Net to limit the segments sent from the sender to the receiver to 10mb/s with a queue length of 13 packets. Delay was only added to the receiver side router, and it was set to 30ms for either direction of traffic. Thus, there was a total 60ms delay for the round trip.

To generate competing traffic, an Apache web server was run on the sender. Surge [9] was run on the receiver to generate HTTP requests to be served by the sender. Surge was configured with the default settings based on analyses in [10]. Traces were generated with `ttcp` while Surge was concurrently generating traffic. The traces were captured by `tcpdump` as described previously. A 250ms measurement interval was chosen for the self-clocking based method.

Figure 6 shows average RTT estimates when Surge is generating requests from 0, 200, 400, 600, 800, 1000, and 1200 emulated Web users. The error bars show the standard deviation of each set of estimates. Note that the initial 1s period of server RTT estimates for each trace is discarded because the initial estimates were influenced by preexisting state in the TCP implementation.

The average RTT estimates from the timestamp based method are all within 1ms of that of the server. They are all within 5ms for the self-clocking method. High loss and retransmission rates were the principle causes of variation in the estimates, especially in the traces with 200 and 400 emulated users. In fact, for the self-clocking based method, 9.7% of the 250ms measurement intervals could not produce RTT estimates because no segments arrived during that time. Other

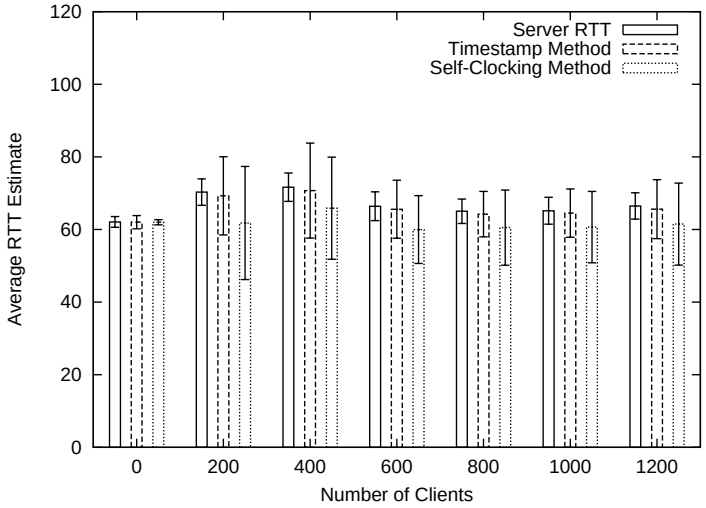

Fig. 6. RTT estimates with competing traffic from emulated Web users

intervals had one or two segments–too few to produce accurate estimates. We are currently investigating why traces with more than 600 emulated users had lower loss rates. Considering the severity of the effects of congestion, our average RTT estimates were very accurate.

6.3 Real Network Tests

To evaluate our RTT estimation methods on real networks, we performed FTP downloads from seven sites (Table 2). Traces were captured for the FTP data streams on the client with `tcpdump`. A measurement interval of 500ms was chosen for the self-clocking based method to accommodate possible RTTs up to 250ms. Note that RTT estimates for the servers' TCP implementations are not available since we do not have access to these machines. ICMP ping times were captured for reference at a later date than the traces. The RTT for the Sun server appears to have changed after the traces were taken. For the Intel server, the 100ms timestamp granularity was too coarse, so no valid RTT estimate could be produced. The Intel server also did not respond to ping requests.

The estimates for the self-clocking based method were very accurate. With the exception of one trace, the minimum and maximum estimates differed by at most 1ms. The maximum difference between the average self-clocking based estimate and the average timestamp based estimate was 2.2ms for the Sun trace, which had a high 10ms timestamp granularity. The 150ms estimate for JRiver trace from the self-clocking based method was caused by a single missed harmonic frequency. Similarly, the low 40ms was caused by two consecutive measurements during a series of half-RTT bursts.

Table 2. RTT estimates in ms for FTP downloads

Site	Ping	Timestamps Method			Self-Clocking Method		
		Avg.	Min.	Max.	Avg.	Min.	Max.
ftp10.us.freebsd.org	97.6	98.7	97.9	134.5	98.1	98.0	99.0
ftp.cs.washington.edu	59.7	62.0	60.7	87.0	60.0	60.0	60.0
ftp.cs.stanford.edu	62.4	65.3	64.0	99.9	65.1	65.0	66.0
ftp.jriver.com	60.5	75.3	60.6	101.6	75.7	40.0	150.0
docs-pdf.sun.com	90.7	52.2	51.7	61.4	50.0	50.0	50.0
ftp.cs.uiuc.edu	20.7	21.5	21.2	24.4	20.0	20.0	20.0
download.intel.com	—	110.6	98.2	127.1	—	—	—

Despite some of the high maximum estimates for the timestamp based method, the *number* of large estimates were few. The highest coefficient of variation was only 0.11% for the JRiver trace. While many of the other RTT estimates from the timestamp based method are affected by a few outliers, estimates for this trace have a more scattered distribution in both directions from the mean. This suggests that our method is detecting actual small-scale variations in RTT caused by conditions in the network route.

Figure 7 shows the passive RTT estimates for the FTP download from Stanford as an example. Note that the timestamp based method is able to take many samples, while samples from the self-clocking based method are limited by the size of the measurement interval. All the estimates from the self-clocking based method are either 65 or 66ms. Nearly all the estimates from the timestamps based method are within 5ms of the average self-clocking estimate. Only two estimates larger than 70ms exist, which were likely caused by sender delay.

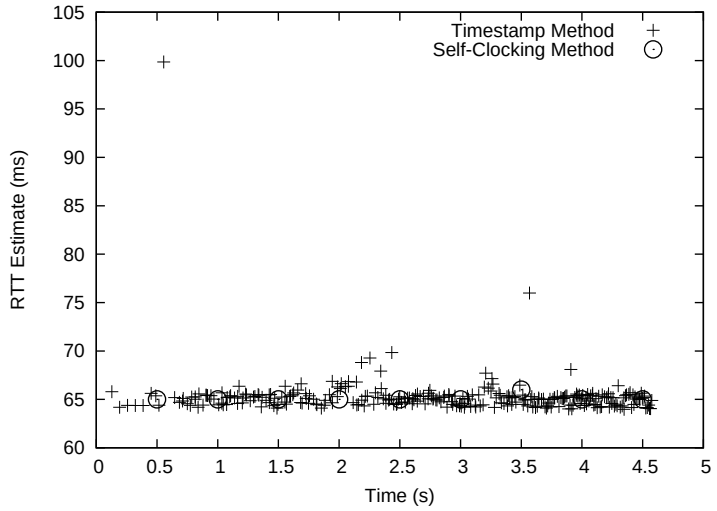

Fig. 7. RTT estimates for Stanford FTP download

7 Conclusions

We have presented two new methods for passive estimation of round-trip times for bulk TCP transfers. These RTT estimations can be made at an interior point along the network route. One method uses TCP timestamps to locate segments from a bulk data sender that arrive one RTT apart, while the other detects patterns caused by self-clocking that repeat every RTT. Both methods can be used throughout the lifetime of a TCP session. The timestamp based method can be used for symmetric routes, while the self-clocking based method works for both symmetric and asymmetric routes.

References

1. Jain, M., Dovrolis, C.: End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput. In: SIGCOMM, ACM (2002)
2. Zhang, Y., Breslau, L., Paxson, V., Shenker, S.: On the characteristics and origins of Internet flow rates. In: SIGCOMM, ACM (2002)
3. Ratnasamy, S., Handley, M., Karp, R., Shenker, S.: Topologically-aware overlay construction and server selection. In: INFOCOM, IEEE (2002)
4. Jiang, J., Dovrolis, C.: Passive estimation of TCP round-trip times. ACM Computer Communication Review **32** (2002)
5. Lu, G., Li, X.: On the correspondency between tcp acknowledgment packet and data packet. In: Internet Measurement Conference, ACM (2003)
6. Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., Towsley, D.: Inferring TCP connection characteristics through passive measurements. In: INFOCOM, IEEE (2004)
7. Alexa Internet: Alexa Web search—top 500. http://www.alexa.com/site/ds/top_sites?ts_mode=global&lang=none (2004)
8. Carson, M., Santay, D.: NIST Net: a Linux-based network emulation tool. ACM Computer Communication Review **33** (2003) 111–126
9. Barford, P., Crovella, M.: Generating representative web workloads for network and server performance evaluation. In: Measurement and Modeling of Computer Systems. (1998) 151–160
10. Barford, P., Bestavros, A., Bradley, A., Crovella, M.: Changes in web client access patterns: Characteristics and caching implications. World Wide Web **2** (1999)