Estimating TCP Latency Approximately with Passive Measurements

Sriharsha Gangam[1], Jaideep Chandrashekar[2], Ítalo Cunha[3], and Jim Kurose[4]

[1] Purdue University
sgangam@purdue.edu
[2] Technicolor Research
jaideep.chandrashekar@technicolor.com
[3] UFMG, Brazil
cunha@dcc.ufmg.br
[4] Univ. of Massachussetts, Amherst
kurose@cs.umass.edu

Abstract. Estimating per-flow performance characteristics such as latency, loss, and jitter from a location other than the connection end-points can help locate performance problems affecting end-to-end flows. However, doing this accurately in real-time is challenging and requires tracking extensive amounts of TCP state and is thus infeasible on nodes that process large volumes of traffic. In this paper, we propose an *approximate* and *scalable* method to estimate TCP flow latency in the network. Our method scales with the number of flows by keeping approximate TCP state in a compressed, probabilistic data structure that requires less memory and compute, but sacrifices a small amount of accuracy. We validate our method using backbone link traces and compare it against an exact, baseline approach. In our approximate method, 99% of the reported latencies are within 10.3 ms of the baseline reported value, while taking an order of magnitude less memory.

1 Introduction

Latency is a key determinant in the performance of a network flow and large values can adversely affect bulk transfers, increase buffering, and make interactive sessions unresponsive. Thus, tracking flow latency is a critical tool in monitoring the performance of TCP-based applications; these form the bulk of Internet traffic today. While estimating this latency is an intrinsic part of TCP and thus trivial at the end-points of a connection, it is extremely challenging in the middle of the network, *i.e.,* at a network node along the path connecting the end-points. At the same time, the ability to infer the flow latency at such locations would be extremely valuable to users and network operators. Consider a typical WiFi-enabled home network with DSL broadband connectivity. Today, when applications underperform or latencies to destinations are larger than usual, it is extremely difficult to reason about where the bottleneck is. Is the increased latency occurring inside the wireless network? or is the server slow to respond? Answering this seemingly simple question directly is quite difficult (even if we could query the end-points for their estimates). This question is easy to answer if the home gateway could estimate latencies of the connections on the wireless link. Similar applications could be imagined at data center borders or egress routers in enterprise networks.

This area has attracted a lot of attention in the past and several methods have been proposed; these broadly fall into two categories—active or passive. Active methods rely on probing the destination(s) independent of the TCP flow (various flavors of ping exist), or else by inserting a transparent TCP proxy on the path of the TCP flow [4]. This has the effect of terminating one half of the connection, and creating a new connection from the midpoint. Clearly, this is impractical for large numbers of concurrent flows. Moreover, terminating the flow in the middle of the network *alters* the flow and may not be acceptable. The other set of methods is based on passive observations of the traffic. A single RTT estimate can be obtained by matching SYNs and ACKs in the beginning of a TCP connection [11]; this is useful, but of limited utility since latency can change considerably over the connection duration. Another idea is to infer the RTT by computing the delay between the transmission of two consecutive congestion windows [10,12,15]. In [2, 10], the TCP state machine is emulated *offline*, using passively recorded traces, to infer RTT estimates by matching ACKs and TCP sequence numbers. Some of these methods can track latency over the entire duration of a connection in the middle of the network; however, they are not scalable and are not designed to be run in real-time.

The challenge in *accurately* estimating TCP latency *in the network* centers on the amount of state that needs to be maintained. Packets going in one direction need to be stored and matched with acknowledgements coming back the other way. In measurement points that handle a large number of flows (routers in ISP networks, data center switches) or embedded devices that are resource constrained, it is generally infeasible to store sufficient TCP state information, or to process it fast enough to support very accurate TCP latency estimates in real-time. The key observation we make is that when estimating latency, a strict accuracy constraint limits how well a solution can scale. There are applications that need to measure latency accurately and with a high degree of precision (electronic trading systems, B2B applications). Correspondingly, there are particular solutions that target these markets, relying on specialized hardware and multiple vantage points (see [13, 14]). However, most other applications, particularly those that focus on troubleshooting or performance diagnosis, are more interested in tracking whether latencies are within a specified range or if they have exceeded a threshold. Importantly, such applications can tolerate approximate answers and a certain amount of error. Take for example an application that monitors VoIP call quality; acceptable quality might require that accuracy not exceed 150ms [9]. Similar latency thresholds are associated with other applications: 100 ms for online first person shooter and racing games [6], and in the same region for video streaming applications [16]. In such applications, tracking approximate latencies is good enough.

In this paper we investigate the problem of performing scalable and approximate latency estimation in real-time inside the network. We describe such a method, called ALE (Approximate Latency Estimator), present its key ideas and introduce two variants ALE-U (Uniform) and ALE-E (Exponential). These methods work by sacrificing accuracy, which requires (exactly) tracking a great deal of TCP state, and instead keeping approximate state, which uses far less memory, but have a certain inherent amount of error. Importantly, this loss of accuracy can be controlled by using more (or less) memory. These methods were implemented and compared against `tcptrace` [2], a well established, *offline* analysis tool for TCP. We carried out a validation study using two different traces obtained from CAIDA. On these traces, we show that ALE can achieve accuracy very close to tcptrace, while using far less memory and requiring less computation. In the best performing latency estimator, 99% of the reported latencies are within 10.3 ms of the actual value and over 97% of the median flow latencies reported are within 10.2 ms of the actual medians; all while taking about one thirtieth of the memory used by the baseline.

2 TCP Latency Estimation

TCP estimates RTT by matching ACKs against a set of *data segments* sent (but not yet acknowledged). For example, suppose host A is sending data to host B on a path that goes through M. At time t_1, A sends a data segment with k bytes of data to B. This segment contains a sequence number range $[s, s+k]$ (bytes are individually numbered). After B processes this segment, it sends back an acknowledgement to A which explicitly indicates the next byte in the stream it expects to receive, *i.e.,* $s+k+1$ (this is exactly one more than the last sequence number in the packet sent by A), and this reaches A at t_2. Since the acknowledgement can be matched with the segment sent previously, A estimates the RTT as $t_2 - t_1$. Now, node M can also perform a similar estimation by matching data segments with acknowledgements (ACKs, in short) seen in flight. The RTT estimate for the path segment $M \leftrightarrow B$ is $t_a - t_d$, where t_d and t_a are when the data segment and the acknowledgement were observed at M. Note that there is not enough information to estimate the RTT on the path segment $A \leftrightarrow M$; this requires B sending data to A and receiving ACKs back. To obtain accurate RTT estimates at M, for either side of the path, we need to remember all the unacknowledged data segments seen in one direction, and match them against ACKs coming back the other way. This makes straightforward RTT estimation infeasible at nodes that handle a large flow volume, or at memory constrained embedded devices of the type used in home and small business gateways. That being said, if we are willing to tolerate a small amount of error in the RTT estimates or a few missed RTT estimates, we significantly reduce the amount of memory required.

In our approach, we exploit the following two observations: (i) storing the exact timestamp associated with each TCP segments is overkill. It is sufficient to remember having seen it in a particular time interval, and (ii) we can avoid storing the sequence number *range* and just store a single sentinel value instead. Following the first observation, we can divide time into discrete intervals and just associate the segments with particular intervals. Thus, with each interval, we now associate a (possibly) large set of segments that arrived in that interval (specifically, the sequence number ranges and flowids). The second observation does away with having to store the sequence number *range* and lets us store a single number for each unacknowledged segment. Specifically, this number is just one larger than the end of the sequence range in the segment, *i.e.,* the number that is expected to be returned in the acknowledgement. We note that this is not guaranteed to always be the case; the TCP specification permits partial segments to be acknowledged. However, this is not the norm and when it does happen, it is an indication of a performance bottleneck at the receiver. If we overlook this corner case, we can simply record the *expected* acknowledgement number for each segment (this is exactly one larger than the last sequence number in the segment) and match this against incoming ACKs. By exploiting this "most likely behavior" in TCP, the problem of searching through a number of ranges or intervals now becomes that of set membership queries which can be done very efficiently with probabilistic data structures (such as Bloom filters).

Approximate Latency Estimator (ALE). Looking into the recent past, we divide time into fixed size discrete intervals, $[w_0, w_1], [w_1, w_2], \ldots, [w_{n-1}, w_n]$, over a sliding window. Here, $[w_0, w_1]$ is always the most recently elapsed interval, the sliding window covers a span of $W = w_0 - w_n$ seconds, and each interval is of length $w = w_i - w_{i+1}$ (we use *interval* and *bucket* interchangeably). This time discretization is shown in Fig. 1. We denote by B_i the data structure *currently* associated with interval i. Apart from the buckets associated with the sliding window, we use another bucket B to hold state for the immediate present. At the end of every w seconds, we move B (to the left in the figure) into the past and into the sliding window. The data structures B_i and B are Counting Bloom Filters (CBF) [7], a variation that supports set member deletions.

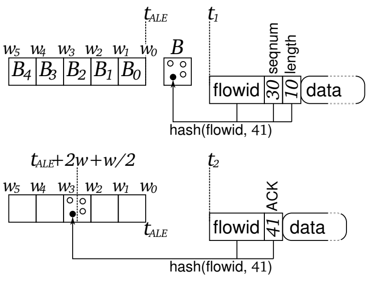


Fig. 1. Operation of the ALE Algorithm

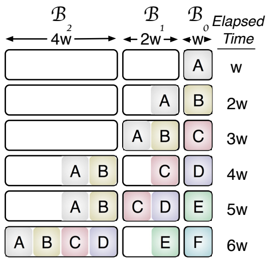


Fig. 2. Buckets Being Shifted and Merged in ALE-E

The TCP segment insertion operation in ALE records the flow identifier and the expected sequence number into a bucket by hashing the concatenation of the two and incrementing the appropriate counters (see [7] for details). Deletion proceeds the same way, but with the counters decremented. Set membership reads the counters indexed by the hash functions and reports 'yes' if all of them are non-zero, and 'no' otherwise.

We use the diagram in Fig. 1 to walk through an example. The upper half demonstrates a just arrived TCP segment being recorded. The TCP data segment that arrives at t_1 is recorded into B (with sequence number 30 and containing 10 data bytes). After every w seconds, the data structure associated with each interval is shifted to the left, *i.e.,* $B_{i+1} \leftarrow B_i$. Thus, the contents of the current window B move to B_0, the most recently elapsed interval in the sliding window, and B is reset. At the same time, B_n reaches the end of the sliding window and is discarded. Rather than keep track of each of the interval boundaries (which have fixed size), we use a single running timestamp t_{ALE} that points to the end of the sliding window, *i.e.,* corresponds to w_0. After every w seconds t_{ALE} is incremented by w. Thus, an entry stored in B_i would have arrived in the time interval $t_{\mathrm{ALE}} - wi$ and $t_{\mathrm{ALE}} - w(i+1)$.

Now, suppose an acknowledgement arrives (possibly piggybacked on a data packet) at time t_2 (see lower half of Fig. 1): we need to look backwards in time and find the (first) bucket where the corresponding segment was recorded. We first check B (and if there is a match, the RTT is just $(t_2 - t_{\text{ALE}})/2$). Then we check $B_0, B_1, \ldots, B_n$ until we find a match. In the figure, we find a match in the window $[w_2, w_3]$. We estimate the RTT as $t_2 - t_{\text{ALE}} + 2w + w/2$. More generally, when a match is located in B_i, the RTT is $t - t_{\text{ALE}} + (2i + 1)w/2$. If there is no match after checking the last bucket B_n, ALE does not return an RTT estimate.

Notice that there are two parameters, w and W, that control accuracy and coverage. For a given span W, increasing the accuracy requires smaller w; thus, more buckets and associated data structures. A similar argument holds if we were to hold accuracy fixed (*i.e.,* fix w) but increase W. Rather than require accuracy to be uniform in each bucket (necessitating a lot of buckets), one could also use non-uniform buckets. We now introduce a variation of the estimator just introduced that we call ALE-E, which employs exponentially increasing intervals. ALE-E attempts to support *relative accuracy* for the same number of buckets *i.e.,* better accuracy for smaller latency samples and lesser accuracy for larger latencies. To differentiate the two, we use ALE-U to denote the use of uniform sized buckets.

ALE with Exponential Buckets. ALE-E follows the same general idea of moving the contents of buckets to its older neighbor. However, the buckets follow a slightly different rule for the shift operation. Like before, bucket B_0 shifts its contents to B_1 after every w seconds. However, bucket B_i is shifted (and merged) into B_{i+1} every $2^i w$ seconds. This is illustrated in Fig. 2. Here, we see that every w seconds, B_0 is merged with B_1; B_1 is merged with B_2 every $2w$ seconds; B_2 is merged with B_3 every $4w$ seconds (not shown) and so on. Whenever the buckets are merged, we adjust their starting and ending times appropriately. The actual merging is trivial if B_i is maintained as a CBF: we simply add up the corresponding counters. Intuitively, the size of each interval is twice as long as the one preceding it. That is to say the i-th interval is of size $2^i w$. If the width of the smallest bucket is w, monitoring the span W requires $1 + \lfloor \log_2(W/w) \rfloor$ buckets. ALE-E can cover the same range using fewer buckets. However, this comes at a price: larger buckets cause larger errors in RTT estimate. Moreover, the merging of bloom filters causes them to attenuate with each merge, *i.e.,* the bitmaps get more "crowded" and prone to false positives. Thus, ALE-E makes the estimation of longer latencies inaccurate in return for parsimonious use of memory and better accuracy for smaller latencies.

Other Sources of Error. When multiple data segments are acknowledged by a single cumulative acknowledgement (*e.g,* a delayed ACK), we can only match (and remove) the last data segment and generate a single RTT sample. The other data segments are not removed from the CBFs until they drop out of B_n. The same phenomenon occurs if the ACKs are not flowing (or flowing fast enough) towards the sender. If this persists over time, the counting bloom filter becomes saturated and exhibits a high false positive rate. We can deal with this by increasing the size of the CBFs.

Since ALE does not maintain the state for the TCP segments, we cannot identify reordered packets. When this does happen, ALE will return an incorrect answer (rather than discarding the RTT sample). Retransmitted packets also pose a source of error. In general, retransmitted segments can be identified (and excluded from the RTT estimation) in ALE by first checking through all the buckets *before* recording the segment. If a copy is found, one of the segments being matched is a duplicate and they should both be discarded. However, this additional check makes the TCP segment insertion operation go to $\mathcal{O}(n)$ from $\mathcal{O}(1)$.

Accuracy and Overhead Bounds. ALE has simple discretization error bounds on the estimated latencies. Let w denote the width of the bucket B_0. For ALE-U, the worst case error is $w/2$ and the average case error is $w/4$. In ALE-E, when an ACK has a match in bucket i, the worst case error is $2^{i-1}w$ and the average case error is $(3w/16)2^i$. We omit the proofs due to a lack of space.

ALE with h hash functions takes $\mathcal{O}(h)$ time to insert an expected acknowledgement number in the CBF of bucket B. Matching an ACK takes $\mathcal{O}(hn)$ time (answering CBF membership queries on n buckets). For every time interval w, shifting buckets takes $\mathcal{O}(1)$ time, if the buckets are implemented as a linked list. Additionally, ALE-E takes $\mathcal{O}(C)$ time to add counters of two CBFs (merging). Finally, ALE takes $n \times C \times d$ bits of memory, where d is the number of bits in each counter of the CBFs.

ALE Parameters. The sliding window size, or span (W) should be chosen to ensure that most of the *normal* latencies observed fall inside of it. It is a limitation of ALE to use a preconceived estimate of the maximum latencies in the network. Interval width (w), the other ALE parameter, is mainly dictated by accuracy requirements. In an application like VoIP, which needs latencies lower than 150 ms, an error of 20 ms is acceptable to identify problematic scenarios. This requires setting w to 40 ms in ALE-U.

We use CBFs with 4 hash functions. For m entries and C counters, the optimal number of hash functions h is given by $h = (C/m)\ln 2$ and the corresponding false positive rate is $\approx (2^{-\ln 2})^{C/m}$ [5]. With 4 hash functions, this false positive probability is 0.0625. One can estimate C based on the traffic rate, the time interval w, and the number of hash functions h. The traffic rate R in our traces varies between $350,000$ and $600,000$ TCP packets per second [1]. For $w = 20$ ms, $h = 4$, and $m = R \times w$, the constraint for optimal h ($h = (C/m)\ln 2$) yields $C = 40396$. In practice, we require fewer counters (30000) as the matched ACK numbers are deleted from the CBFs. We use CBFs with 4-bit counters as they work well in practice [7].

3 Evaluation

We compare different ALE variants against `tcptrace` as the baseline solution. The terms tcptrace and baseline are interchangeable. We use two 60-second traces captured at a backbone link of a tier-1 ISP obtained from CAIDA. The traces capture headers of all packets in both directions of the link. Trace 1 starts on 07-21-2012 at 13:55 UTC, contains 2,115,802 TCP flows, and 50,022,761 TCP packets; Trace 2 starts on 02-17-2011 at 13:01 UTC, contains 2,423,461 TCP flows, and 54,089,453 packets. We find that only 1.8% and 2.1% of the captured flows are bidirectional, a result of most Internet paths being asymmetric at the core [8]. However, closer to the network edge (home gateways and even most access networks), traffic is bidirectional. Since we do not handle unidirectional flows in our current implementation, we pre-processed the traces to

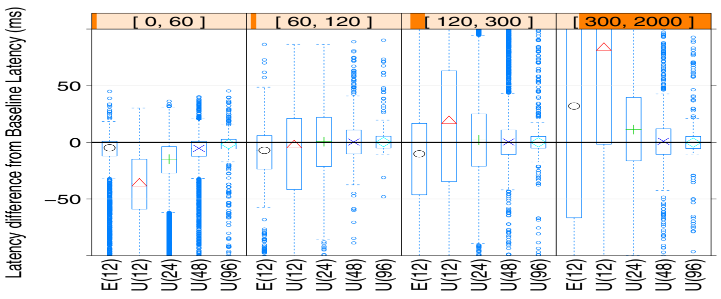


Fig. 3. Distribution of estimation error ($\mathrm{RTT}_{\mathrm{baseline}} - \mathrm{RTT}_{\mathrm{ALE}}$) over all samples

filter out unidirectional flows. We point out that it is indeed feasible to filter out such flows in an online fashion, but this discussion out of the scope of this paper. In the rest of the section, unless explicitly stated, we show results from Trace 1, but results for Trace 2 are qualitatively similar.

We compare 4 different configurations of ALE-U with $n = 12$, 24, 48, and 96 buckets; we refer to these configurations as ALE-U(n). Using `tcptrace`, we find that the majority of RTTs in Trace 1 are less than 100 ms and only 0.5% are larger than 2 s. We thus configure ALE with a window span of two seconds, *i.e.*, $W = 2$ s. This results in intervals w ranging from 167 ms (when $n = 12$) to 21 ms (when $n = 96$). Finally, we configure ALE-E with 12 intervals and we refer to this configuration as ALE-E(12). To accommodate the high traffic rates in our traces, we use CBFs with $C = 30000$ counters and 4 hash functions.

RTT Estimation Accuracy. We first report on the per-sample RTT estimation accuracy across the various methods. While this is not a very natural metric of comparison—most applications would be in interested in some statistic over these—it does serve to illustrate some of the intuition for why ALE performs comparably, and talks to its suitability for certain situations. Fig. 3 presents a box and whisker plot of the *differences* between the RTT reported by each method and `tcptrace`. In other words, it shows the distribution of $\mathrm{RTT}_{\mathrm{tcptrace}} - \mathrm{RTT}_{\mathrm{ALE}}$ over all RTT samples in Trace 1 for different ALE configurations. Recall that the size of the interval bounds the accuracy for ALE. To draw out how ALE might perform at different regimes, the plot is partitioned into four latency regions (unrelated to w): $(0, 60]$ ms, $(60, 120]$ ms, $(120, 300]$ ms, and $(300, 2000]$ ms. For example, the first group (left) plots the distribution for all samples where the baseline approach reported a latency between 0 and 60 ms. The horizontal line in the center of the figure marks the region where the difference is zero (the values reported by the baseline and ALE are identical). The height of the box spans the inter-quartile range of the differences in the RTT estimate and the point in the box is the median difference.

Not surprisingly, across all latency ranges, increasing the number of buckets improves accuracy. We also note that ALE-U(96), which almost completely agrees with the far more expensive baseline approach. There are a few rare large differences, but this may be quite acceptable considering the savings in memory; especially keeping

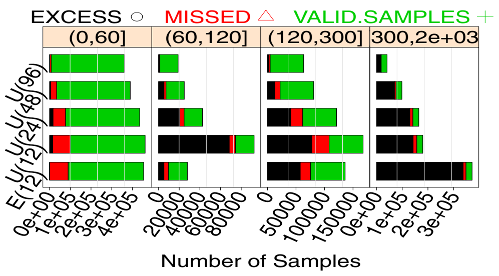


Fig. 4. Comparison of excess and missed samples across the different approaches

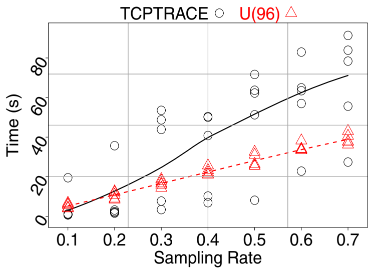


Fig. 5. Compute time for ALE-U(96) and tcptrace and for different thinned traces

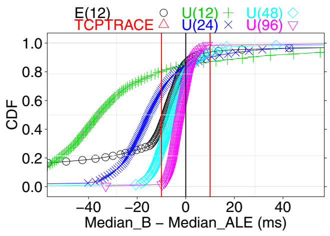
in mind that per-sample RTT estimates are rarely used directly. In Fig. 3, we note that ALE-E(12) does almost as well as ALE-U(96) for latencies less than 60 ms even though ALE-E(12) uses 8 times less memory. The price, however, is reduced accuracy for latencies above 60 ms.

Excess and Missed Samples. As discussed previously, the ALE algorithms sometimes miss RTT samples reported by the baseline approach (RTT *misses*, *e.g.,* when CBF false positives decrement counters prematurely); and sometimes report RTTs *not* reported by the baseline (*excess* RTTs, due to, *e.g.,* CBF false positives or reordered packets). We plot the absolute numbers of each, separated by latency regions in Fig. 4. We observe that adding more memory (buckets) reduces both missed and excess RTTs. With respect to ALE-E, we see that missed and excess RTTs are few when latencies are small, but the excess samples are common for large latency values. As the CBFs for each bucket in ALE-E are shifted and merged, they are increasingly attenuated and have higher false positive rates. Nevertheless, ALE-E is still accurate up to 120 ms, which may be enough for interactive applications.

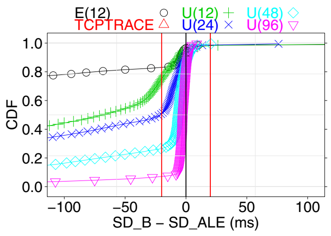
Fig. 4 can qualitatively explain the contribution of the different sources of error. For example, when ALE-U uses sufficiently large memory (*e.g.,* U(96)) the effects of false positives and negatives are mitigated. U(96) has few misses and excess values indicating that there are few retransmitted and reordered packets in the CAIDA traces. If the results in Fig. 3 and Fig. 4 do not improve with additional memory, one can conclude that the errors are due to re-ordered and retransmitted packets.

Errors in Flow Latency Properties. Typical flow performance monitoring applications track some *statistic* of the flow latencies, rather than use the RTT samples directly. Consider the example of VoIP quality tracking, very sensitive to jitter. This involves monitoring and tracking the variation in latencies of a flow, relating this to the user perceived quality of the session. We study the impact of the approximations native to ALE on two relevant flow statistics: (i) median latency of a flow, which impacts the quality of interactive applications (network games, web browsing), and (ii) jitter (latency variation) in a flow, which impacts the perceived quality of most real-time streaming applications (VoIP, video conferencing).

Fig. 6a plots the distribution of differences in the median latency computed by the baseline and ALE. Each curve is a distribution of the values of $\text{median}_{\text{baseline}} - \text{median}_{\text{ALE}}$



(a) Difference in median latency.



(b) Difference in latency std. dev.

Fig. 6. Accuracy for latency statistics over all flows

over all flows. The vertical black line in the middle represents perfect agreement with the baseline. In the figure, we note that ALE-U(96) performs best over the other instances: at least 97% of the flows have medians within 10 ms of the baseline (shown by the vertical red lines). To put this in context, the acceptable latency for VoIP is between $[20, 150]$ ms: being off by 10 ms does not affect the monitoring application to a large degree. The other ALE-U instances perform as expected: larger number of buckets tends to make the curve steeper (and more aligned with the center line). We also note that ALE-E performs reasonably well: about 65% of the flows have median latency estimate within 15 ms of the actual value. Since the median is robust to outliers, some of the large errors that ALE-E reports for individual samples are filtered out. Thus, along with ALE-U(96) and perhaps ALE-U(48), ALE-E might be effective as a lightweight method to monitor the quality of low latency sessions.

In a similar comparison, Fig. 6b plots the CDF of disagreement in the standard deviations of flow latencies, *i.e.,* $\sigma_{\text{baseline}} - \sigma_{\text{ALE}}$. The vertical black line (at $x = 0$) marks the region where the std. dev. reported by ALE matches that of the baseline perfectly. Firstly, we notice that the baseline approach in general reports lower variance than the ALE approaches (the cases when the difference is negative). This is because the latencies are dispersed over a time interval w. In the figure, the red vertical lines indicate the 20 ms boundary from zero. We again see that ALE-U(96) performs better than any of the others, and 95.7% of the flows have delay variance that differ from the baseline reported version by at most 20 ms. We also see that ALE-E performs poorly on this comparison. About 80% of the flows disagree with the baseline reading by at least 20 ms. This is most likely due to CBF attenuation in the larger intervals leading to a large number of false positives.

Memory and Compute Overhead. We thin out the trace by sampling flows uniformly at random at rates 0.1, 0.2, . . . 0.7, such that there are 5 pcap sub-traces at each rate. For a given rate, all the 5 pcap sub-traces have about the same number of flows. A sub-trace with higher sampling rate requires processing of more packets and flows per unit time. Using these sub-traces, we run ALE-U(96) and tcptrace (both implemented in GNU C), on an AMD quad core 512 KB cache, 2.6 GHz, 8 processor machine with 32 GB RAM to study the overhead. We use tstime [3] tool which leverages the GNU Linux taskstats API to get user time, system time, high water RSS (resident segment size) memory and high water VSS (virtual segment size) memory of a process.

As expected, ALE-U(96) takes constant high water RSS memory of 2.0 MB and high water VSS memory 9.8 MB for all sampling rates. In contrasting, tcptrace requires RSS memory ranging from ≈ 64 MB (at rate 0.1) to ≈ 460 MB (at rate 0.7). The VSS memory requirement ranges from ≈ 74 MB to ≈ 468 MB. These experiments confirm our hypothesis that ALE has significantly less memory overhead.

Fig. 5 shows the times taken to process at different sampling rates for ALE and tcptrace. As the data rates increase, tcptrace takes increasingly longer time than ALE. tcptrace has higher variability in compute times. ALE, by avoiding TCP state, has less variability and takes constant per-packet processing time (on average) at all traffic rates.

4 Discussion

Though our implementation does not incorporate computational optimizations, we hope that a performance-focused implementation (*e.g.,* parallelizing ACK lookups in the n buckets) would be even faster. An optimized implementation can fit the data required by a wide range of ALE configurations in the caches of low-end Atom and ARMv8 processors (currently between 256 KiB and 1 MiB). The evaluated configuration with 48 buckets and 30,000 4-bit counters per bucket would require about 700 KiB of memory for processing 10 Gbit/s links. We note that ALE lends itself well to implementation in hardware: ALE's basic building blocks are hashing functions, 4-bit accumulators, and 4-bit comparators.

Current home DSL gateways usually run local area networks that run at 100 Mbit/s and connect to the Internet with connections up to 28 Mbit/s. In an heavy-loaded scenario with an Internet download at 28 Mbit/s and a local transfer at 100 Mbit per second, the gateway would receive 11000 full-size (1500 B) packets/s (the absolute number of flows does not impact ALE in any way). Configuring ALE-U(12) with 12 buckets and $W = 200$ ms would require bucket sizes of 1058 counters per bucket, for a total memory utilization of $12 \times 1058 \times 4 \div 8 \div 1024 = 6.2$ KiB. This fits easily in the cache of current MIPS and ARM processors used in home DSL gateways.

References

1. CAIDA: Passive network monitors, `http://www.caida.org/data/realtime/passive/`
2. tcptrace, `http://www.tcptrace.org/`
3. tstime, `https://bitbucket.org/gsauthof/tstime/`
4. Web10G, `http://www.web10g.org`
5. Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., Varghese, G.: An Improved Construction for Counting Bloom Filters. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 684–695. Springer, Heidelberg (2006)
6. Dick, M., Wellnitz, O., Wolf, L.: Analysis of Factors Affecting Players' Performance and Perception in Multiplayer Games. In: Proc. ACM Netgames (2005)
7. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Netw. 8(3), 281–293 (2000)
8. He, Y., Faloutsos, M., Krishnamurthy, S., Huffaker, B.: On Routing Asymmetry in the Internet. In: Proc. IEEE GLOBECOM (2005)
9. ITU-T. Recommendation G.114: One-way Transmission Time (May 2000)

10. Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., Towsley, D.: Inferring TCP connection characteristics through passive measurements. In: Proc. IEEE INFOCOM (2004)
11. Jiang, H., Dovrolis, C.: Passive estimation of TCP round-trip times. SIGCOMM Comput. Commun. Rev. 32(3), 75–88 (2002)
12. Lance, R., Frommer, I.: Round-trip time inference via passive monitoring. SIGMETRICS Perform. Eval. Rev. 33(3), 32–38 (2005)
13. Lee, M., Duffield, N., Kompella, R.R.: Not all microseconds are equal: fine-grained per-flow measurements with reference latency interpolation. In: Proc. ACM SIGCOMM (2010)
14. Lee, M., Duffield, N., Kompella, R.R.: Leave them microseconds alone: Scalable architecture for maintaining packet latency measurements. Technical report, Purdue Univ. (2011)
15. Veal, B., Li, K., Lowenthal, D.: New Methods for Passive Estimation of TCP Round-Trip Times. In: Dovrolis, C. (ed.) PAM 2005. LNCS, vol. 3431, pp. 121–134. Springer, Heidelberg (2005)
16. Xiu, X., Cheung, G., Liang, J.: Delay-cognizant interactive streaming of multiview video with free viewpoint synthesis. IEEE Trans. on Multimedia 14(4), 1109–1126 (2012)