FlowTrace: A Framework for Active Bandwidth Measurements using In-band Packet Trains

Adnan Ahmed[1], Ricky K. P. Mok[2], and Zubair Shafiq[1]

[1] The University of Iowa
adnan-ahmed@uiowa.edu zubair-shafiq@uiowa.edu
[2] CAIDA/UC San Diego
cskpmok@caida.org

Abstract. Active measurement tools are important to understand and diagnose performance bottlenecks on the Internet. However, their overhead is a concern because a high number of additional measurement packets can congest the network they try to measure. To address this issue, prior work has proposed in-band approaches that piggyback application traffic for active measurements. However, prior approaches are hard to deploy because they require either specialized hardware or modifications in the Linux kernel. In this paper, we propose FlowTrace–a readily deployable user-space active measurement framework that leverages application TCP flows to carry out in-band network measurements. Our implementation of pathneck using FlowTrace creates recursive packet trains to locate bandwidth bottlenecks. The experimental evaluation on a testbed shows that FlowTrace is able to locate bandwidth bottlenecks as accurately as pathneck with significantly less overhead.

1 Introduction

Background. Internet performance measurement plays an important role in diagnosing network paths, improving web application performance, and inferring quality of experience (QoE). A notable example is the use of available bandwidth measurement in adaptive video streaming [20,15]. ISPs and content providers are motivated to build web-based measurement tests (e.g., M-Lab NDT [19], Ookla speed test, and Netflix fast.com [8]) to provide throughput measurement services for end-users. These platforms estimate access link capacity by flooding the network with one or more concurrent TCP flows [2], which result in very high overhead [9]. Note that while such tools can be used by the end-users to measure network performance, large scale deployment (e.g. by CDNs) to conduct Internet-wide network measurements poses scalability concerns due to high overheads.
Limitations of Prior Work. Over the last decade, many light-weight end-to-end active network measurement methods have been proposed to accurately measure network path metrics, including latency [17], packet loss rate [26], available bandwidth [28,13,25], and capacity [3,14,5]. These tools inject crafted probe

packets into the network with a specific packet sending pattern, and analyze the timing or events of responses to compute the network metrics. However, these tools are not widely adopted for measuring web service performance for two main reasons.

1) *Out-of-band.* The measurement probes often used different flow tuples (types of packets, source/destination ports) to user traffic [21]. The network path traversed by the measurement flow could be different from the user traffic, and thus the results may not be representative. In addition, some measurement tools (e.g., PATHLOAD [13], PATHCHIRP [25]) typically generate a significant amount of traffic—carrying no useful data—to interact with and measure the network.
2) *Prior solutions are hard to deploy.* Various solutions such as MGRP [24] and MINPROBE [29] have been proposed to mitigate the impact of these measurement tools on the network, by leveraging application traffic to conduct measurements. Ideally, such tools can be deployed at the server-side to leverage ongoing downstream traffic to conduct end-to-end measurements to the client-side. However, these solutions are limited in terms of feasibility of deployment. For instance, MGRP requires modifications in the Linux kernel, making it OS-specific, while MINPROBE requires dedicated FPGA-based SoNIC hardware.

Proposed Approach. In this paper, we propose FLOWTRACE, a user-space measurement framework to deploy in-band network measurement systems. FLOWTRACE overcomes the limitations of prior work as follows. First, it conducts in-band measurements by intercepting and rescheduling application data packets. Second, it only uses commodity Linux utilities such as `iptables` and `NFQUEUE`, thereby avoiding the need to patch the kernel or additional hardware, making it feasible to deploy across large scale infrastructures such as Content Delivery Networks or measurement platforms such as M-Lab. Overall, FLOWTRACE intercepts packets from the application flows and shapes them so as to implement different measurement algorithms.

Evaluation. We have implemented a prototype of FLOWTRACE and evaluated it using Emulab. Specifically, we demonstrate the effectiveness of FLOWTRACE by implementing a well-known measurement tool PATHNECK [10] over FLOWTRACE, and comparing the measurements done using both implementations. Note that PATHNECK uses recursive packet trains (RPTs) to locate the bottleneck by analyzing the packet dispersion of the ICMP TTL exceeded messages returned by the intermediate hops. We show in our evaluation that measurements done using PATHNECK implemented on FLOWTRACE closely follow the measurements done using PATHNECK. Lastly, we show that using FLOWTRACE only increases the application-perceived latency by at most 1.44 milliseconds.

We remark that FLOWTRACE can be used by various measurement platforms to efficiently implement a vast array of measurement algorithms, that would otherwise be infeasible to deploy at large scale.

2 Background

There is a long line of research on active network measurements to measure different network performance metrics such as round-trip-time (RTT), packet loss rate, and bandwidth [18,25,12,13,28,11,10]. These active measurement tools provide useful insights for network performance diagnosis, management, and even protocol design. For instance, `ping` [16] is a simple yet effective tool to measure RTT and packet loss between two hosts by constructing specially-crafted `ICMP` messages that, when received by a receiver, are echoed back to the sender. iPerf [6] is commonly used to measure bandwidth between two hosts by measuring the time it takes to complete a bulk transfer between the two hosts. These tools and their variants are used to conduct Internet-scale measurements using dedicated measurement platforms such as M-Lab [18].

Prior work has proposed more sophisticated active measurement tools such as pathload [12] and pathChirp [25] to measure available bandwidth as well as pathneck [10] to localize the bandwidth bottleneck between two hosts. Instead of relying on bulk data transfers, these tools probe the network and measure the timing information of the responses to estimate the bandwidth characteristics. More specifically, these tools craft *probe packets* that traverses the end-to-end path between a source and a destination host and interacts with the underlying network along the path. As a result, the underlying network modulates the probe traffic (such as packet transmission rates at the links) and generates a “response” (such as inter-packet gaps) as the *probe packets* move forward through the links along the path. The tools then analyze this timing information to estimate the bandwidth characteristics of the underlying network.

Even though these more sophisticated bandwidth measurement tools generate relatively less traffic as compared to iPerf, they still introduce non-trivial probe traffic that can cause congestion in the very network they are trying to measure. For instance, pathneck identifies the location of the bottleneck along the path by constructing recursive packet trains (RPTs), consisting of large payload packets wrapped around with small probe packets. Note that even though the probe packets in the RPTs are negligible in size, payload packets are typically much larger and carry dummy payload that can congest the network. Such non-trivial overheads make it infeasible to deploy these bandwidth measurement tools on a large-scale.

To address this issue, prior work has proposed methods that allow these measurement tools to piggyback *useful* application traffic onto the measurement traffic [24,29]. More specifically, MGRP [24] was designed to mitigate the overheads of measurement tools such as pathload and pathChirp by piggybacking payload data from all application flows destined to the remote host—to which the measurement is to be done—into probe packets. These probe packets are received and demultiplexed into the constituent application flows by the remote host, while the measurement is done by observing the arrival times of the MGRP probes. In the same vein, minProbe was proposed to leverage application traffic in a middlebox environment for Gigabit-speed networks. Specifically, minProbe [29] intercepts application flows destined to the target host at middleboxes and modulates (and measures) the transmission (and arrival) times of these packets with nanosecond precision to allow for high-speed network measurements.

While existing methods such as MGRP [24] and minProbe [29] do minimize measurement overheads by leveraging application traffic for probing, their deployment requires specialized hardware or kernel-level modifications at the hosts. minProbe requires specialized hardware such as FPGA pluggable boards and Software-defined Network Interface Cards (SoNIC) making it infeasible for Internet-scale deployment. MGRP requires changes to the Linux Kernel, making it OS- and Kernel-specific and severely limiting its deployability as acknowledged by [24].

3 FlowTrace

In this section, we first discuss some design goals of FlowTrace (§3.1), and then describe the technical challenges we tackled in implementing FlowTrace (§3.2).

3.1 Overview

The design of FlowTrace revolves around two main goals. First, FlowTrace leverages ongoing TCP flows to conduct *in-band* network measurement. By embedding measurement probes into the flows nwe make sure that the measurement traffic follows the same path as the application traffic, thereby enabling measurements along the paths undertaken by the application traffic. In addition, leveraging application traffic to conduct measurements can significantly reduce the measurement overheads. Second, FlowTrace can be feasibly deployed by various server-end entities such as content-providers and measurement platforms (such as M-lab) to measure the application of web services and conduct Internet-wide measurements, without requiring significant changes to the Linux Kernel and additional hardware respectively.

We use FlowTrace to perform pathneck-like measurements to locate network bottlenecks. Identifying under-provisioned links is useful for load-balancing traffic and improving the service quality. FlowTrace is implemented as an in-band, user-space tool that leverages ongoing TCP flows for measurement.

FlowTrace monitors traffic to identify new TCP flows and decides which flows to be measurement flows, and then intercepts packets from measurement flows to construct `RPTs`, which comprise of large *payload packets* wrapped around with TTL-limited *probe packets*. The routers on the path subsequently drop the first and the last *probe packets* and generate TTL-exceeded `ICMP` response messages [10]. FlowTrace captures these response messages to infer the location of the bottleneck. To this end, FlowTrace treats data packets in the flow as *payload packets*, and inserts leading and trailing *probe packets*—called head packets and tail packets, respectively. FlowTrace conducts bottleneck identification and localization by analyzing the arrival time of the response packets triggered by the dropped hand and tail packets. FlowTrace does not manipulate the

TTL of data packets, allowing them to be received by the remote host without any disruptions. Note that a large amount of data in the constructed `RPT` is the original *payload packets* carrying useful application data, with FlowTrace inserting only a number of small *probe packets* to conduct measurement.

3.2 Technical Challenges

While the basic concept behind FlowTrace is intuitive, as we discuss below, it presents a unique set of technical challenges.

Lack of Kernel-level visibility. The first and foremost challenge in leveraging ongoing application traffic to deploy active measurement techniques in user-space is the lack of kernel-space visibility and packet-level control. Specifically, when an application generates data that is to be sent to a remote host, it passes the data down to the kernel where it is fragmented and formed into TCP/IP packets after filling all the corresponding packet header fields, and is finally sent over the wire. Prior approaches such as MGRP implemented an in-kernel solution to intercept and piggyback application layer packets in probe packets to implement various active measurement techniques. However, as mentioned before, MGRP requires changes in Linux kernel, and is OS-specific, which makes it difficult to deploy at a large scale.

On the contrary, we use commodity Linux-based utilities such as firewalls and basic user-space libraries to implement FlowTrace. More specifically, FlowTrace relies on utilities such as `iptables` and `NFQUEUE` to obtain fine-grained per-packet control *in user-space*. In this manner, FlowTrace intercepts packets from application traffic, modifies (or modulates) packet transmission times, and inserts *probe packets* to create `RPTs`. All in all, these commodity Linux-based utilities provide relatively fine-grained control and visibility over application layer traffic without compromising on the feasibility for large-scale deployment.

Interception vs. "Respawning". In addition to fine-grained control over application traffic, active measurement techniques require control over the transmission rate of the measurement traffic. Specifically, pathneck transmits a "well-packed" `RPTs` at line rate from the source host to effectively locate bottlenecks along the path. However, since FlowTrace leverages application layer traffic to construct `RPTs` and conduct measurement, it is limited by the traffic characteristics of the application. For instance, if the application generates payload data at a rate slower than the line rate, FlowTrace will be unable to construct "well-packed" `RPTs`, resulting in inaccurate network measurements.

To enable FlowTrace to send `RPTs` at line rate, we can *buffer* application packets in the kernel using `NFQUEUE`, and transmit a "well-packed" `RPT` when FlowTrace has received enough packets from the application layer. However, a limitation of using `NFQUEUE` is that FlowTrace only has visibility at the *head* of the queue, with no information about the number of packets in the queue. To solve this, FlowTrace "respawns" the application layer traffic in the user space.

FlowTrace retrieves and copies available packets in `NFQUEUE`, and notifies the kernel to discard the original packet by returning `NF_DROP`. Lastly, upon receiving the specified number of application layer packets, FlowTrace constructs a "well-packed" `RPT` using the buffered data packets, and (re-)transmits them using `pcap` at line rate.

Minimizing the impact of packet buffering. Lastly, FlowTrace largely depends on the traffic generation behavior of the application conduct measurements. Note that traffic generation patterns and volume of traffic can vary significantly across applications, ranging from small short-lived flows generated by web browsing to long-lived flows for video streaming services. As a result, FlowTrace may not be able to receive and intercept sufficient data packets to generate measurement traffic according to the specified measurement configuration. To this end, we can increase the time FlowTrace waits for the next packet to accumulate more data packets. However, application packets may perceive excessive buffering period before they are sent, thereby hurting the throughput and responsiveness of the application.

To reduce the impact on the application performance, FlowTrace employs an opportunistic approach to conduct measurement. FlowTrace continuously monitors the inter-arrival delays of the new data packets. Whenever the inter-arrival delay exceeds a certain time threshold, t_{ipa}, FlowTrace gives up on that round of measurement, and immediately sends out all the buffered packets over the wire using `pcap`, thereby resuming the flow. After resuming the flow, FlowTrace waits for the next chance to conduct the required measurement using subsequent application packets. Note that the value of t_{ipa} governs the tradeoff between the capability of FlowTrace to consistently conduct measurements and application layer performance.

Another alternative approach to minimize the impact of buffering is to introduce additional dummy data packets when the number of data packets from the application is not enough. However, this approach introduces additional congestion to the network which is against the design goal for FlowTrace. Therefore, in this paper, we use t_{ipa} to configure the wait time and resume application flows when FlowTrace receives insufficient data packets.

3.3 Implementation

FlowTrace employs `NFQUEUE` [22], which is a user-space library to intercept an ongoing network flow from the system. FlowTrace identifies a flow of interest based on the IP address provided by the operator, and sets up `iptables` to intercept the application flow. Specifically, FlowTrace sets up `iptables` rules inside the Linux kernel with `NFQUEUE` as the target, effectively redirecting packets from the flow of interest to `NFQUEUE`. Consequently, whenever a packet satisfies an `iptables` rule with an `NFQUEUE` target, the firewall enqueues the packet and its metadata (a Linux kernel skb) into a packet queue such that the decision regarding the corresponding packets can be delegated to a user-space

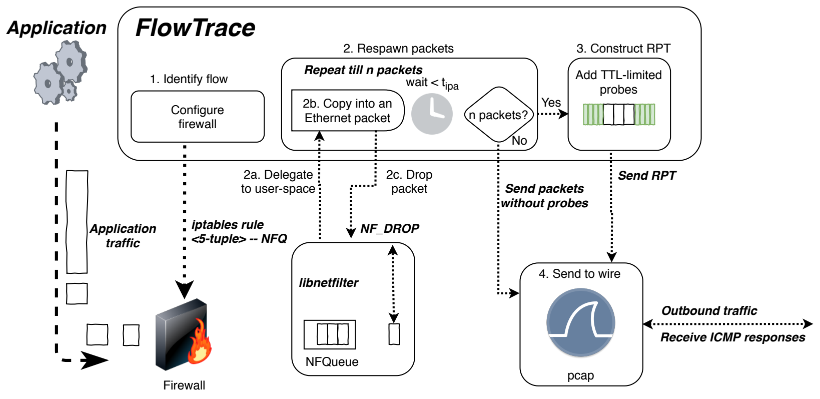


Fig. 1. High level work flow of FLOWTRACE. FLOWTRACE first identifies an application flow based on the 5-tuple. It then configures `iptables` rules such that the packets from the application flow are directed towards the `NFQUEUE` target. `NFQUEUE` delegates the decision of each packet to a user-space program spawned by FLOWTRACE, which buffers the payload from each packet in user-space and directs `NFQUEUE` to drop the packets. When FLOWTRACE receives enough application packets, it creates a "well-packed" `RPT` out of the buffered application packets and transmits them on the wire via `pcap`. Otherwise, FLOWTRACE transmits the buffered application packets without any additional probe packets.

program. To this end, the kernel then executes a callback registered by the user-space program, and sends the enqueued packet with the metadata using *nfnetlink* protocol. The user-space program then parses the packet information and payload, and can decide a verdict on dropping (`NF_DROP`) or releasing (`NF_ACCEPT`) the packet. At a high-level, FLOWTRACE intercepts the packets from the flow of interest, then handles these packets in the user-space and modulates traffic accordingly to implement the measurement algorithm specified by the user. We implemented a prototype of FLOWTRACE using `GO` programming language, that supports lightweight concurrency using `goroutine`. As a result, we can conduct measurement to multiple concurrent flows with small overheads.

Implementing pathneck. Based on the high-level idea described above, we now describe how FLOWTRACE can be used to implement PATHNECK, as illustrated in Fig. 1.

1. *Identifying the flow of interest.* To avoid process all incoming/outgoing packets, the operator provides the IP address and port information of the flows, directed towards the clients that we are interested in measuring (such as port 80 and 443 for web servers). Consequently, FLOWTRACE can initialize `iptables` rules and `pcap` filters to reduce workload. All the flows matched the specified IP and ports consider as the flow of interest.
2. *Intercepting flows.* Once the initialization completed, FLOWTRACE creates a map of flows using the 5-tuple (source IP, destination IP, source port, destination port, protocol), and starts to handle packets using `NFQUEUE` and

record the state of the flows. We do not perform any measurement in the beginning of the flows, because the TCP congestion window during the slow-start phase is too small for us to receive sufficient data packets to construct measurement probes. In our implementation, we do not manipulate the first 10 packets of the flows.

3. *Constructing measurement probes.* We use the flows that transferred more than 10 data packets to conduct measurements. FlowTrace waits for n load packets from the TCP flow to construct a "well-packed" RPT. Fig. 2 depicts the construction of one `RPT`. As the application data arrives, FlowTrace first buffers the load packets and drops the original packets (using `NF_DROP`) (§3.2). When FlowTrace has n payload packets from the application flow in the buffer, it first sends m TTL-limited zero-size measurement packets, followed by the n load packets, and m tail packets. This way, FlowTrace injects a "well-packed" `RPT` constructed from the application traffic, into the network.
4. *Capturing and analyzing response packets.* The final step is to monitor the incoming traffic using `pcap` for the `ICMP` messages triggered by the `RPT`. Based on the `ICMP` response messages from each hop, FlowTrace computes the per-hop packet train dispersion and determine the network bottleneck.

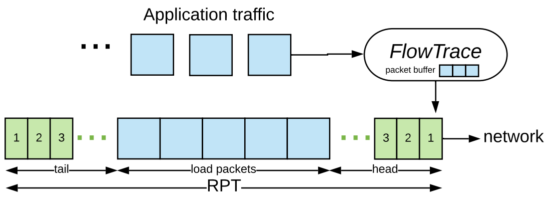
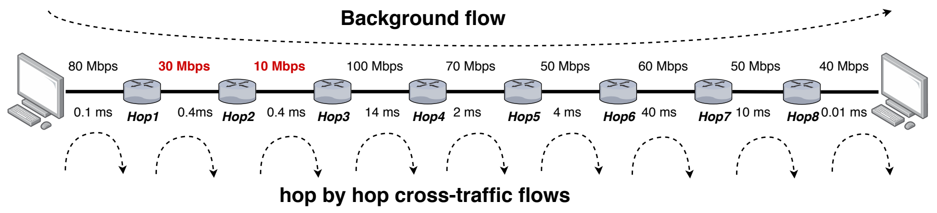


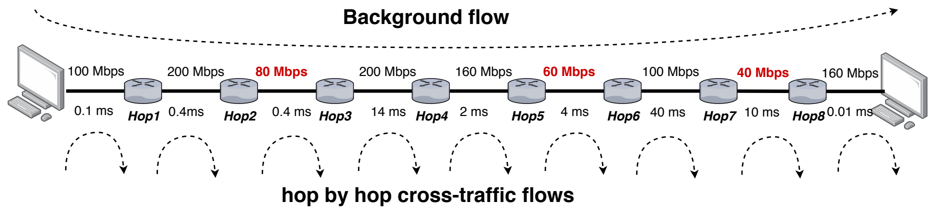
Fig. 2. Details of RPT generation. Light blue squares are data packets from the application, and green rectangles are probe packets. The number inside the green rectangles are the TTL values of the packet.

Minimizing the impact of packet buffering. In step 3, the flow is effectively stopped while we wait to receive n flow packets. Because the amount of application data is unpredictable, it is possible that FlowTrace intercepts less than n packets and keeps waiting for the next data packet to arrive. This can seriously affect the throughput, RTT, and congestion control mechanisms of the application. Therefore, we choose a small inter-packet-arrival timeout, $t_{ipa} = 1,000\mu s$, to ensure that FlowTrace does not wait and hold the flow packets for too long while constructing `RPTs`. When FlowTrace receives a flow packet in user-space, it sets up a timer of t_{ipa} for receiving the next flow packet. Upon the arrival of the next packet, FlowTrace resets the timer.

In the event that timer expired, FlowTrace decides to not generate the `RPT` and instead just reconstructs and sends the intercepted flow packets to recover the flow. This way FlowTrace ensures that the flow RTT and congestion



(a) Two bottlenecks



(b) Three bottlenecks

Fig. 3. Emulab topology configurations for our evaluation of the performance of both pathneck and FlowTrace. The bottleneck hops in both Fig. 3(a) and Fig. 3(b) are highlighted.

control mechanisms are not significantly affected as we will demonstrate in §4.3.

4 Evaluation

We now evaluate the implementation of pathneck on FlowTrace in a controlled testbed environment for different network conditions. Specifically, we want to see how closely the measurements done using pathneck implemented on FlowTrace agree with the measurements done using pathneck. Emulab [23] allows us to test and compare the measurements of the two against known traffic workloads in a controlled testbed environment.

To this end, we create a linear network topology in Emulab as shown in Fig. 3 similar to that studied in [10]. Our network consists of a sender and a receiver machine, connected to each other via a series of intermediate routers. In our evaluation, we evaluate the performance of both pathneck and FlowTrace in “two-bottlenecks scenario” and “three-bottlenecks scenario” as shown in Fig. 3(a) and Fig. 3(b) respectively.

In addition to the routers along the path from the sender to the receiver machine, we generate background traffic as well, across the network as shown in Figure 3. The background traffic comprises of two kinds of flows, 1) the background flow from the source machine to the destination machine, and 2) the hop-by-hop cross-traffic flows that traverse the links between the intermediate routers. In our setup, we use `iPerf` to set up bandwidth-constrained TCP flows as the hop-by-hop cross-traffic flows. On the other hand, we set up a large file transfer between a web server at the sender machine and a `wget` client at the receiver machine as the background flow. Note that since FlowTrace leverages

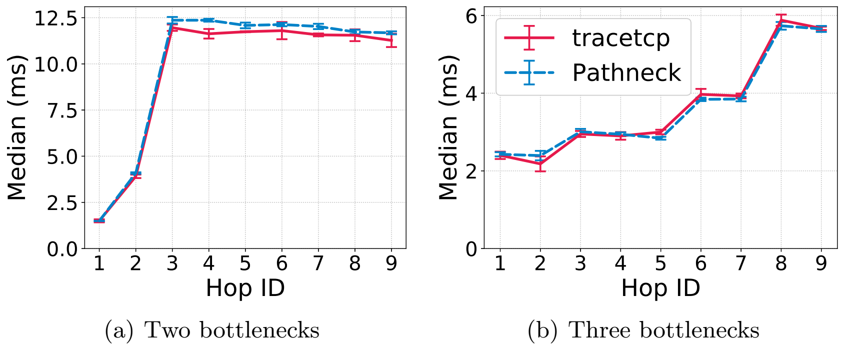


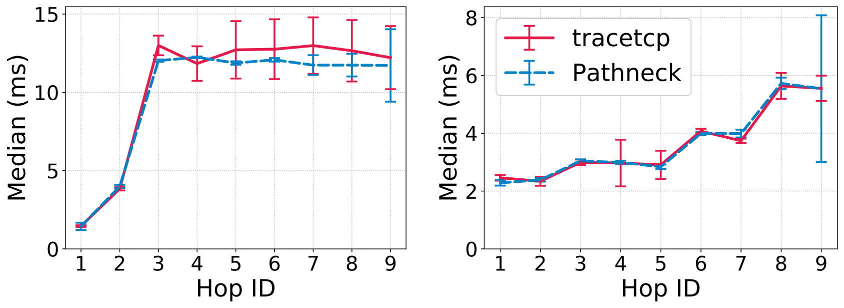
Fig. 4. Comparison of gap values reported by PATHNECK and FLOWTRACE across and end-to-end network without cross traffic in a controlled emulab testbed.

ongoing traffic to construct `RPTs` and conduct measurement, we configure it to leverage the background flow from source machine to the destination machine in our experiments. In particular, each `RPT` constructed by FLOWTRACE is composed of $n = 10$ *payload packets*, intercepted from the background flow, and $n = 15$ TTL-limited *probe packets*. Lastly, we configure the link characteristics such as link delays and link bandwidths, along the path, using commodity Linux utilities such as `tc` and `netem` to create the aforementioned testbed scenarios.

4.1 Minimal cross traffic

We first consider the "bare-bones" scenario where there is minimal cross traffic along the links in the network and the choke points and bottlenecks are primarily dictated by the change in link capacities along the end-to-end path. This scenario is ideal for PATHNECK, as the `RPTs` are affected only by the capacity of the links along the path without any interference from the cross traffic, making it easier to identify choke points and locate bottleneck. To this end, we configure the hop-by-hop cross traffic across all the links in Fig. 3 to be only 0.01 Mbps.

Fig. 4 plots the median gap values along with the standard deviation (for 15 runs each), across the hops in the end-to-end path reported by both PATHNECK and FLOWTRACE for both "two-bottlenecks" and "three-bottlenecks" scenarios. We note that both PATHNECK and FLOWTRACE report similar gap values across the hops for both scenarios. For instance, FLOWTRACE exhibits a gap value increase of 2.41 milliseconds and 8.05 milliseconds, whereas PATHNECK exhibits a gap value increase of 2.59 milliseconds and 8.27 milliseconds in gap values at hops 2 and 3, where the link capacities decrease by 50 Mbps and 20 Mbps respectively, in Fig. 4(a). On the other hand, FLOWTRACE exhibits a gap value increase of 0.77 milliseconds, 0.97 milliseconds, and 1.95 milliseconds, whereas PATHNECK exhibits a gap value increase of 0.62 milliseconds, 1.0 milliseconds, and 1.88 milliseconds in gap values at hops 3, 6, and 8, where the link capacities decrease from 100 Mbps to 80 Mbps, 60 Mbps, and 40 Mbps respectively, in Fig. 4(a). All in all, our results show that both PATHNECK and FLOWTRACE largely agree with one another in terms of reported gap values for various network



(a) Two bottlenecks (b) Three bottlenecks

Fig. 5. Comparison of gap values reported by pathneck and FlowTrace across and end-to-end network with cross traffic in a controlled emulab testbed.

configurations given that cross-traffic is minimal—which has been shown to be largely the case across the Internet in prior literature [10].

4.2 With cross-traffic

We now evaluate the impact of cross-traffic along the links in the end-to-end path, on the gap values reported by both pathneck and FlowTrace. To this end, we consider the impact of forward (upstream) cross-traffic—the downstream links do not experience any cross-traffic [3]. Note that ideally, in this scenario, the returning `ICMP` messages from each hop should be received by the sender machine without experiencing any interference from cross-traffic. In this case, we configure the hop-by-hop `iPerf` clients to generate cross-traffic equal to 5% of the corresponding link capacities.

Fig. 5 plots the median gap values along with the standard deviation (for 15 runs each), across each hop in the end-to-end path as reported by both pathneck and FlowTrace for both "two-bottlenecks" and "three-bottlenecks" scenario. We again note that both pathneck and FlowTrace report similar gap values across the hops in the end-to-end path. For instance, FlowTrace exhibits a gap value increase of 2.38 milliseconds and 9.16 milliseconds, whereas pathneck exhibits a gap value increase of 2.55 milliseconds and 8.04 milliseconds in gap values at hops 2 and 3, where the link capacities decrease by 50 Mbps and 20 Mbps respectively, in Fig. 5(a). We observe similar pattern for the "three-bottlenecks" scenario in Fig. 5(b). We do note that the forward cross-traffic results in significantly higher variance as compared to the scenarios without cross-traffic, especially for the hops farther away from the sender machine

[3] Hu et al. [10] reported that reverse path effects may impact the performance of pathneck as they may perturb the gaps between the `ICMP` response messages on the way back. Our goal is that FlowTrace performs well *when* pathneck performs well. Therefore, we do not evaluate the impact of reverse path effects on the performance of FlowTrace in this work for brevity.

in both scenarios in Fig. 5. For instance, the standard deviation for FlowTrace increased from 0.05 milliseconds at hop 1 to 2.01 milliseconds at the last hop, whereas the standard deviation for pathneck increased from 0.23 milliseconds at hop 1 to 2.31 milliseconds at the last hop in Fig. 5(a). This is because the error in gap values *accumulates* as the packets traverse the network.

4.3 Latency overhead

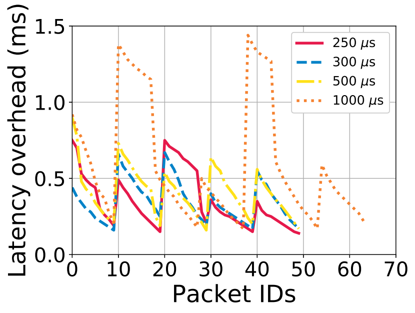


Fig. 6. Latency overheads involved in FlowTrace perceived by the application.

Since FlowTrace leverages application flows to construct `RPTs` and conduct measurement, we evaluate the impact of FlowTrace on the latency experienced by the application flows. Specifically, since FlowTrace buffers and "respawns" the application packets in the user-space, we report the additional latency each application packet experienced by the application flow for different values of the inter-arrival delay threshold, t_{ipa}, in Fig. 6. As expected, we observe a piece-wise linear pattern in Fig. 6. This is because the first packet generated by the application, when intercepted by FlowTrace, has to be buffered all the while FlowTrace waits for more application packets. On the other hand, as soon as FlowTrace received enough application packets, it creates a `RPT` and transmits it on the wire, thereby adding minimal latency for the last application packet in the `RPT`.

Note that this overhead is primarily dictated by t_{ipa} and traffic characteristics of the application as discussed in §3.2—the higher the value of t_{ipa}, the longer the packets can potentially be buffered and therefore, the higher the overheads in Fig. 6. This may result in FlowTrace affecting the latency characteristics—such as RTT and jitter—perceived by the application. In our evaluation, from Fig. 6, we observe that an application flow may experience an inflation of at most 1.44 milliseconds increase in application perceived latency, for $t_{ipa} = 1$ milliseconds, when leveraged by FlowTrace to conduct measurements. This overhead decreases to 0.75 milliseconds for $t_{ipa} = 0.25$ milliseconds, because FlowTrace waits for a shorter period of time for application packets before releasing the buffered packets without constructing the `RPT`. To summarize, these latency overheads are dictated by the traffic patterns of the underlying application—burst of packets generated by the application may result in FlowTrace having to wait for a lesser amount of time as compared to spaced out traffic patterns.

5 Related Work

Prior literature has proposed various tools and techniques to measure path performance in terms of metrics such as available bandwidth [4,13,28,11,25], bottleneck location [10,1], and loss rates [26]. However, most of these tools can only perform measurement out-of-band.

Few existing works adopted in-band measurement paradigm. Prior research has proposed different approaches such as paratrace [7], Service traceroute [21], and TCP Sidecar [27] to map Internet paths more effectively. These techniques rely on embedding TTL-limited measurement probes alongside the non-measurement application traffic, evading firewalls and NATs, and increasing the coverage of measurement systems across the Internet. Papageorge et al. proposed MGRP [24]—an in-kernel service that allows users to write measurement algorithms which are subsequently implemented by piggybacking application data inside probe traffic to minimize overheads and lower the impact of conducting measurements on competing application traffic. However, MGRP requires changes to the kernel at both the client- and the server-side machines, making it difficult to deploy at a large scale. In the similar vein, Wang et al. proposed minProbe [29]—a middlebox architecture that used application network traffic as probe traffic to conduct measurements such as available bandwidth by modulating packet transmissions with high fidelity. However minProbe requires specialized hardware and physical access to both end-points, which is often hard to deploy.

QDASH [20] integrates pathload [13] into adaptive streaming flows. It reshapes video data packets into different sending rates to detect the highest video bitrate the network can support. However, QDASH can only obtain end-to-end available bandwidth information. It cannot locate the bottleneck on the path. In this paper, we leverage application traffic to deploy pathneck, locating choke points along the path and facilitate the measurements of bandwidth characteristics of the network at a large scale.

6 Conclusion

We presented FlowTrace, an active measurement framework that conducts in-band network measurements by piggybacking application data. We showed that FlowTrace can transparently create recursive packet trains to locate bandwidth bottlenecks with minimal impact on application performance. FlowTrace not only significantly reduces the overhead of active measurements but can also be readily deployed in user-space without needing kernel modifications or specialized hardware. The experimental evaluation showed that pathneck's implementation of using FlowTrace as well as the original pathneck implementation can both accurately locate bandwidth bottlenecks. As part of our future work, we are interested in extending FlowTrace to implement other active bandwidth measurement techniques. Furthermore, we aim to study the impact of FlowTrace on the performance of different types of applications, such as realtime video and web. We are also interested in large-scale deployment of FlowTrace to conduct Internet measurements in the wild.

7 Acknowledgement

We thank the anonymous reviewers and our shepherd, Mirja Kühlewind, for helpful suggestions that improved the paper. We also thank Amogh Dhamdhere for his input in the early stage of this work. This work is supported in part by the National Science Foundation under grant numbers CNS-1617288, CNS-1750175, and CNS-1414177.

References

1. N. Baranasuriya, V. Navda, V. N. Padmanabhan, and S. Gilbert. QProbe: Locating the Bottleneck in Cellular Communication. In *CoNEXT*, 2015.
2. S. Bauer, D. Clark, and W. Lehr. Understanding broadband speed measurements. In *TPRC*, 2010.
3. E. Chan, A. Chen, X. Luo, R. Mok, W. Li, and R. Chang. TRIO: Measuring asymmetric capacity with three minimum round-trip times. In *ACM CoNEXT*, 2011.
4. D. Croce, M. Mellia, and E. Leonardi. The Quest for Bandwidth Estimation Techniques for Large-Scale Distributed Systems. In *ACM SIGMETRICS*, 2009.
5. C. Dovrolis, P. Ramanathan, and D. Moore. Packet dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Trans. Netw.*, 12(6):963–977, 2004.
6. J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu. IPerf.fr. `https://iperf.fr`.
7. Erich. Paratrace, 2018. `http://www.adeptus-mechanicus.com/codex/paratrc/paratrc.php`.
8. Fast.com. Internet speed test. `https://fast.com`.
9. O. Goga and R. Teixeira. Speed measurements of residential Internet access. In *PAM*, 2012.
10. N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang. Locating internet bottlenecks: Algorithms, measurements, and implications. In *Proc. ACM SIGCOMM*, 2004.
11. N. Hu and P. Steenkiste. Evaluation and Characterization of Available Bandwidth Probing Techniques. In *IEEE JSAC*, 2003.
12. M. Jain and C. Dovrolis. Pathload : A measurement tool for end-to-end available bandwidth. In *PAM*, 2002.
13. M. Jain and C. Dovrolis. End-to-End Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput. In *Transactions on Networking*, 2003.
14. R. Kapoor, L.-J. Chen, L. Lao, M. Gerla, and M. Y. Sanadidi. CapProbe: a simple and accurate capacity estimation technique. In *ACM SIGCOMM*, 2004.
15. Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, and D. Oran. Probe and adapt: Rate adaptation for HTTP video streaming at scale. *IEEE JSAC*, 32(4):719–733, 2014.
16. Linux. Ping. `https://linux.die.net/man/8/ping`.
17. X. Luo, E. Chan, and R. Chang. Design and implementation of TCP data probes for reliable and metric-rich network path monitoring. In *USENIX ATC*, 2009.
18. M-Lab. Internet measurement tests. `https://www.measurementlab.net/tests/`.
19. M-Lab. NDT (network diagnostic tool), 2017. `https://www.measurementlab.net/tests/ndt/`.

20. R. Mok, X. Luo, E. Chan, and R. Chang. QDASH: A QoE-aware DASH system. In *Proc. ACM MMSys*, 2012.
21. I. Morandi, F. Bronzino, R. Teixeira, and S. Sundaresan. Service traceroute: Tracing paths of application flows. In *PAM*, 2019.
22. netfiler. The `netfilter.org` "libnetfilter_queue" project. `https://netfilter.org/projects/libnetfilter_queue/`.
23. T. U. of Utah. emulab. `https://www.emulab.net/`.
24. P. Papageorge, J. McCann, and M. Hicks. Passive Aggressive Measurement with MGRP. In *SIGCOMM*, 2009.
25. V. Ribeiro, R. Riedi, R. Baraniuk, J. Navratil, and L. Cottrell. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *Passive and Active Measurement Workshop*, 2003.
26. S. Savage. Sting: A TCP-based network measurement tool. In *Proc. USENIX Symp. Internet Tech. and Sys.*, 1999.
27. R. Sherwood and N. Spring. Touring the internet in a tcp sidecar. In *IMC*, 2006.
28. J. Strauss, D. Katabi, and F. Kaashoek. A Measurement Study of Available Bandwidth Estimation Tools. In *IMC*, 2003.
29. H. Wang, K. S. Lee, E. Li, C. L. Lim, A. Tang, and H. Weatherspoon. Timing is Everything: Accurate, Minimum Overhead, Available Bandwidth Estimation in High-speed Wired Networks. In *IMC*, 2014.