Service traceroute: Tracing Paths of Application Flows

Ivan Morandi

▶ **To cite this version:**

Ivan Morandi. Service traceroute: Tracing Paths of Application Flows. Networking and Internet Architecture [cs.NI]. 2018. hal-01888618

HAL Id: hal-01888618
https://inria.hal.science/hal-01888618

Submitted on 5 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Service traceroute: Tracing Paths of Application Flows

Ivan Morandi
Supervisors: Renata Teixeira, Timur Friedman
Advisor: Francesco Bronzino
Inria / UPMC Sorbonne University, Paris

ABSTRACT

Traceroute is often used to help diagnose when users experience issues with Internet applications or services. Unfortunately, probes issued by classic traceroute tools differ from application traffic and hence can be treated differently by middleboxes within the network. This paper proposes a new traceroute tool, called *Service traceroute*. Service traceroute leverages the idea from paratrace, which passively listens to application traffic to then issue traceroute probes that pretend to be part of the application flow. We extend this idea to work for modern Internet services with support for automatically identifying application flows, for tracing of multiple concurrent flows as well as for UDP flows. We implement command-line and library versions of Service traceroute, which we release as open source. This paper also presents a calibration and an evaluation of Service traceroute when tracing paths traversed by Web downloads from the top-1000 Alexa websites and by video sessions from Twitch and Youtube. The goal of the calibration is to find the best parameters of Service traceroute for each application. Our evaluation shows that Service traceroute has no negative side effect on the vast majority of downloads, but that in some rare cases it can cause application flows to abort or increase flow completion times. In addition, the evaluation shows that Service traceroute obtains different paths in at least 30% of paths when compared with a standard traceroute. Using the same source and destination ports as the target application flow to analyze for standard traceroute's probes, Service traceroute still obtains different paths in at least 7% of analyzed paths.

1. INTRODUCTION

Internet services and applications rely on highly distributed infrastructures to deliver content. When applications stop working or when their performance degrades, service providers and more sophisticated users often resort to traceroute to narrow down the likely location of the problem. Traceroute issues probes with increasing TTL to force routers along the path towards a destination to issue an ICMP time exceeded message back to the source, which iteratively reveals the IP addresses of routers in the path [2].

Traceroute, however, may fail to reveal the exact path that a given application flow traverses. For example, Luckie et al. [7] have shown that depending on the traceroute probing method (ICMP, UDP, and TCP) the set of reached destinations and discovered links differ. The authors explain these differences by the presence of middleboxes in the path such as load balancers and firewalls that make forwarding decisions based on flow characteristics. These results imply that diagnosing issues on application flows must ensure that traceroute probes have the same characteristics as the application's packets.

This paper develops a traceroute tool, called *Service traceroute*, to allow discovering the paths of individual application flows. Service traceroute passively listens to application traffic to then issue probes that pretend to be part of the application flow. Some traceroute tools (for instance, paratrace [4], TCP sidecar [10], and 0trace [3]) already enable probes to piggyback on TCP connections. These tools observe an active TCP connection to then insert traceroute probes that resemble retransmitted packets. TCP sidecar was developed for topology mapping, whereas paratrace and 0trace for tracing pass a firewall. As such, they lack of two important features in application debugging. The identification of the application flows to trace as they require as input the destination IP address and the destination port to detect the target application flow, and the support for tracing paths of modern application sessions, which fetch content over multiple flows that change dynamically over time. In addition, these tools lack the support for tracing application flows using UDP as transport protocol, which are increasing thanks to the adoption of QUIC protocol [5] in new services as well as in popular services like Youtube. Our work makes the following contributions. First, we develop and implement Service traceroute (§2), which we will release as open source software. Service traceroute is capable of identifying application flows to probe and tracing the paths of multiple concurrent flows of both TCP and UDP flows. For example, a user may simply specify trace 'Youtube' and Service traceroute will identify Youtube flows and then trace all of their paths. Service traceroute is configurable to cover a large variety of Internet services.

Our second contribution is the calibration of Service

traceroute for two popular Internet services: video and web pages, in particular, the top-1000 Alexa websites as well as Twitch and Youtube video streaming. The goal of the calibration is to find the parameters of Service traceroute with the best trade-off between the probing network overhead and the amount of information gathered. The problem of Service traceroute is that application flows may close before Service traceroute ends the tracing, and hence middleboxes along the path may discard Service traceroute's probes.

We then use the configurations with the best trade-off for the third contribution: evaluate Service traceroute. We focus the evaluation on the same Internet services used during the calibration. One issue with piggybacking probes with application traffic is that we may hurt application performance. Our evaluation shows that in the vast majority of cases, Service traceroute has no side-effect on the target application (§5). For a few websites, however, application flows get reset when running with Service traceroute. This result indicates that piggybacking traceroute probes within application flows requires careful calibration to avoid hurting user experience. Finally, we compare Service traceroute with 0Trace, which also embeds probes within a target application flow, and with Paris Traceroute, which launches a new flow for probing (§6). Our comparison with Paris traceroute shows differences in 37% of paths with Twitch, 9% with Youtube and 7% with web pages, even when running Paris traceroute with the same ports as the target application flow. We noticed that in some paths with web pages and Youtube videos, middleboxes discard UDP and TCP probes of Paris Traceroute before they reach the destination.

2. TOOL DESIGN AND IMPLEMENTATION

Service traceroute follows the same high-level logic as paratrace or 0trace. Given a *target application flow*, which we define as the application flow whose path we aim to trace, Service traceroute proceeds with two main phases. The first phase is the passive observation of a target application flow to define the content of the probes. Then, the active injection of TTL-limited probes within the application flow. The main difference is that Service traceroute identifies the flows to trace automatically and supports tracing paths traversed by multiple application flows concurrently. The user can either directly specify the set of target application flows or simply describe a high-level service (e.g., Youtube). Service traceroute will then trace paths traversed by all the flows related to the target service. This section first describes the two phases focusing on the new aspects of Service traceroute to allow per service tracing and then presents our implementation.[1]

2.1 Observation of target application flow

Service traceroute passively observes traffic traversing a network interface to search for packets with the flow-id of the target application flows[2]. One novelty of Service traceroute is that it takes a set of target application flows as input, in contrast with previous tools which can only trace the path traversed by one single application flow. Users can either explicitly specify one or more target application flows or they can simply specify a service. Service traceroute uses a database of signatures of known services to inspect DNS packets in real-time and identify flows that match the target service. We release the DB as open source, so users can contribute to add or update the signatures in the database. We define as signature the set of IP addresses and domains corresponding to a specific service. For instance, 'google.com' or the corresponding IP addresses can be used in the signature to detect Google services. Our current database has signatures for popular video streaming services such as Netflix, Youtube, and Twitch. Web pages are not included in the database as Service traceroute can identify web flows simply from the domain or the host name given as input. For additional flexibility, it is possible to easily add ad-hoc domains and IP addresses via command line parameters or through the library API.

2.2 Path tracing

Only once it identifies a packet belonging to the target application flow, Service traceroute will start the tracing phase. This phase works as classic traceroute implementations sending probes with increasing TTL, but Service traceroute creates a probe that takes the form of an empty TCP acknowledgement that copies the 5-tuple of the flow as well as its sequence number and acknowledgement number (similar to paratrace and 0trace). We rely on the flow-id plus the IPID field to match issued probes with the corresponding ICMP responses. We note this is sufficient to correctly identify probes even when tracing multiple concurrent target application flows. The maximum number of concurrent target application flows varies based on the used configuration as the IPID field is dynamically sliced based on the number of probes that have to be generated. For example, with traceroute standard parameters, i.e. maximum distance of 32 and 3 packets per hop, Service traceroute can trace paths of 682 target application flows.

Service traceroute stops tracing when the target application flow closes to avoid any issues with middleboxes (which may interpret probes after the end of the connection as an attack) and also to reduce any net-

[1]The library and command-line version of Service traceroute is publicly-available and open source at https://github.com/wontoniii/tracetcp/tree/servicetraceroute.

[2]We use the traditional 5-tuple definition of a flow (protocol, source and destination IP, as well as source and destination port).

work and server overhead. In contrast to prior tools that only support TCP, we add support for UDP. In this case, we create probes with empty UDP payload, but with the same 5-tuple flow-id as the target application flow. Given UDP has no explicit signal of the end of the flow (like the FIN in TCP), we stop tracing if the flow produces no further packet (either received or sent) after a configurable time interval.

2.3 Implementation

We implement Service traceroute in *Go* and release command-line and library versions. The command-line version is useful for ad-hoc diagnosis, whereas the library allows easy integration within monitoring systems.

The library version of Service traceroute outputs a json data structure that contains the discovered interfaces with the observed round trip time values. For the command line version, Service traceroute shows the results of each trace in the traceroute format, i.e. the list of hops with the corresponding round trips times.

Service traceroute is configurable to adapt to different applications. It includes three types of probing algorithms that capture the tradeoff between tracing speed and network overhead. The first, *PackeyByPacket*, sends only one packet at a time. The second, *HopByHop*, sends a configurable number of packets with the same TTL at a time (3 by default). The third, *Concurrent*, sends all packets at once. Given that Service traceroute requires the target application flow to be active during tracing, some applications with short flows (e.g., Web) require the higher overhead of the third algorithm to complete all the probes within the flow duration. Service traceroute also allows configuring the number of probes for each TTL, the inter-probe time, and inter-iteration time (i.e. the time between packets with different TTL) to further control this tradeoff between tracing speed and overhead. Finally, Service traceroute allows to specify three types of stop conditions: the maximum distance from the source, the maximum number of non replying-hops, like Paris Traceroute, or explicit stop points in the form of IP addresses. The last stopping condition is useful when one wants to focus the tracing. For example, one could use bdrmap [6] to identify the borders of the origin AS and then stop the trace at the border routers.

3. EVALUATION METHOD

We design the calibration to find the best configuration of Service traceroute for two popular applications: video and web pages. Our goal is to identify the optimal parameters setting to balance the tradeoff between completing the discovered paths and network probing load.

We then use the configurations from the calibration, i.e. the best trade-off for web and video, for our evaluation of Service traceroute and we design the evaluation around two questions. First, *does Service traceroute affect the target application flows?* Service traceroute injects new packets within the application flow. Although the majority of these packets will be discarded before they reach the servers, a few probe packets will reach the end-host and can potentially affect the target application flows. Second, *do paths discovered with Service traceroute differ from those discovered with other traceroute methods?* One assumption of our work is that paths taken by classic traceroute probes may not follow the same paths as the packets of the target application flows. We present an evaluation to help answer these questions, where we compare our results with that of Paris traceroute [1].

We conduct all experiments for the calibration and evaluation from 30 PlanetLab Europe nodes. These were the only ones working at the time of the analysis, PlanetLab US nodes were not compatible due to the lack of updated dependencies, like glibc.

Web. We select the top-1000 Alexa webpages on April 14 2018 as target web flows.

Video. We focus on two popular video streaming services: Twitch and YouTube. We select twitch videos on their homepage where Twitch shows dynamically a popular live streaming video. While for YouTube, we select 20 random videos from the catalog obtained after searching with the keyword "4K UHD". With YouTube, we evaluate both TCP and UDP by forcing Google Chrome to use or not QUIC.

Comparison with Paris traceroute. We select Paris traceroute because its Multipath Detection Algorithm (MDA) [12] can discover with high probability all paths between the source and the destination in case there is a load balancer in the path. This allows us to disambiguate whether the differences we may encounter between Paris traceroute and Service traceroute are because of load balancing or some other type of differential treatment. We evaluate Paris traceroute using two different versions. The first is Paris traceroute with MDA enabled using the three protocols ICMP, UDP, and TCP. We let MDA select the ports for Paris traceroute to discover multiple paths between the two end hosts. The second is the standard Paris traceroute using the same 5 tuple as the target application flow.

Comparison with 0Trace. We select 0Trace as it implements the idea of tracing paths of a target application flow and it has a working implementation. However, since 0Trace does not implement the DNS resolution, we used Service traceroute as a mediator to detect the target application flows to probe. Since the version of 0Trace we considered was released in 2007, we had to

update the library for the transmission of probes from dnet to scapy, because it was crashing with the segmentation fault error on all PlanetLab servers while it was trying to send probes.

Experiment setup. Experiments for video and Web are similar. We first launch Service traceroute, then we start streaming a video or downloading a webpage, once that is done we run the three versions of Paris traceroute MDA back-to-back. Then, we stream again the same video or download the same webpage without Service traceroute. We have run a total of 459 videos, 153 for Twitch and 306 equally split between YouTube with TCP and with QUIC, and 1000 Web experiments during 30 days in July 2018.

Small Scale Measurement. We ran a preliminary small scale evaluation of Service traceroute during 14 days in May 2018. In this small scale analysis, we evaluated Service traceroute with Netflix, Youtube and Web pages. We used 7 PlanetLab Europe nodes as vantage points for web pages. While, for videos we used one laptop in a college located in Paris (France) because Netflix only allows from one to four concurrent videos with the same account. In this paper we omit the results of Netflix and Youtube as they are similar to the latest evaluation. While, we include some particular exceptions we encountered during the evaluation of Service traceroute with web pages.

Data representativeness. Our European-scale evaluation is useful to determine whether or not Service traceroute affects the application flows of popular services (top-1000 Alexa as well as Twitch/Youtube). It is also useful to shed some light on whether there are differences between paths discovered with Service traceroute and more traditional traceroute paths.

4. CALIBRATION OF Service traceroute

This section evaluates Service traceroute using different parameters to find the configuration with the optimal trade-off between the completeness of discovered paths and the network probing load.

We vary two parameters. The probing algorithm (PacketByPacket, HopByHop and Concurrent) and the number of probes per hop from 1 to 9. While, we set the other parameters to fixed values to simplify the calibration analysis. In particular, we set the maximum distance to 32, the inter-probe time to a negligible value of 1 micro second and the timeout to wait ICMP replies to 2 seconds.

Calibration Setup. The setup of the calibration is similar to the experiment setup. For each page and video, we run all configurations back to back to preserve the same network characteristics across all configurations.

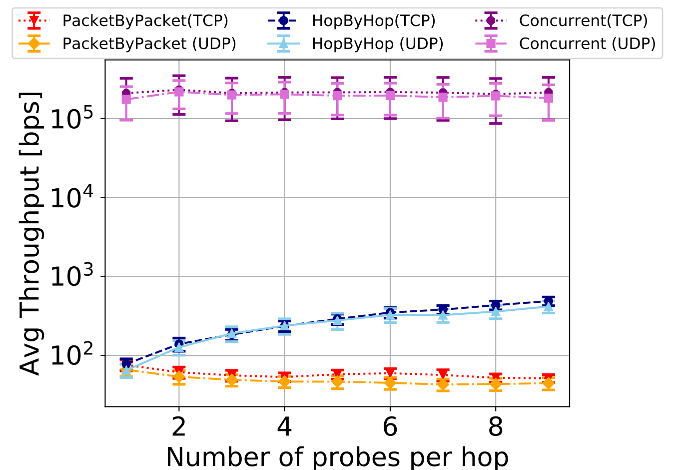


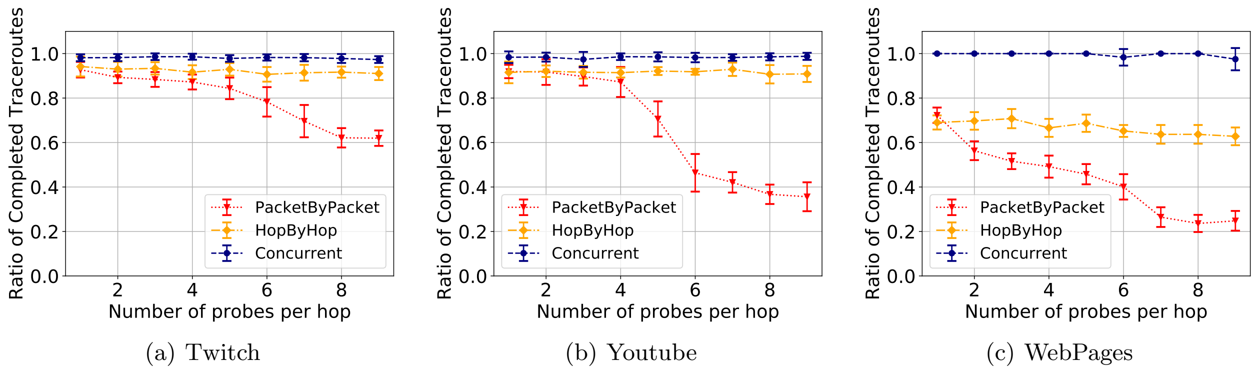
Figure 1: Probe overhead of Service traceroute

4.1 Probe Overhead

We define *probe overhead* as the throughput of probes injected into the network. We compute the probe overhead by summing the total number of bytes transmitted divided by the time between the first and last probe. This metric helps us to evaluate the load that Service traceroute is adding to the network under different probing algorithm and number of probes per hop. Figure 1 shows the probe overhead for all probing algorithm when we vary the number of probes per hop. The result shows that Concurrent has a constant probe overhead of more than 100kbps irrespective of the number of probes sent per hop. Concurrent issues all probes without waiting for ICMP. Thus, Concurrent's probe overhead is only limited by the capacity of the source host. While, for HopByHop algorithm, the probe overhead increases from 100bps to 1kbps when we increase from 1 to 9 probes per hop. The reason of this positive trend is that HopByHop waits ICMP replies at the end of the transmission of all probes with the same TTL. For PacketByPacket, it has a slight negative trend. The reason is that the number of times PacketByPacket waits ICMP replies and timeouts is linearly increasing with the number of probes per hop and the time to receive the ICMP reply increases with the distance from Service traceroute.

4.2 Completed Traceroutes

We define *completed traceroutes* as the number of traceroutes where Service traceroute was able to probe all hops in the path before the end of the target application flow. We consider that a traceroute reached the destination when Service traceroute does not receive replies from three consecutive hops. We compute the ratio of completed traceroutes over the total number of traceroutes done by Service traceroute for a spe-



(a) Twitch (b) Youtube (c) WebPages

Figure 2: Ratio of completed traceroutes

cific configuration. This metric helps us to evaluate whether Service traceroute is able to discover all hops in a path under different probing algorithm and number of probes per hop. Figure 2 shows the ratio of completed traceroutes for all probing algorithms when we vary the number of probes per hop. Concurrent always achieves around 100% of completed traceroutes for all analyzed services. As expected, Concurrent is very fast on discovering paths since there are no timeouts to wait for incoming ICMP replies. HopByHop is able to reach the distribution in almost 100% of cases when tracing paths of Twitch and Youtube flows. For web flows however, HopByHop can only complete tracing in 70% of traceroutes. This difference is caused by the short duration of web flows and the added delay to receive ICMP replies at the end of the transmission of packets with the same TTL. PacketByPacket has the same problem as HopByHop but for all services. Indeed, PacketByPacket waits the ICMP reply after the transmission of each packet, hence the time required to discover the path increases linearly with the number of probes per hop. PacketByPacket starts to have worse performance compared to HopByHop after 4 probes per hop for Youtube, 5 probes per hop for Twitch and 2 probes per hops for Web pages.

4.3 Results

The choice of the probing algorithm depends on different factors, like the application to probe and the added overhead on the network. For videos, like Youtube and Twitch, the best choice for the probing algorithm is HopByHop since the ratio of completed traceroutes is almost good as Concurrent but with a significant smaller probe overhead. While, for web pages, the best choice is Concurrent as the ratio of completed traceroutes is 30% more than the ratio with the other algorithms. The reason of this difference is that web flows have a shorter lifetime compared to video flows, so speed is crucial to probe all hops in the path before the end of the target application flow. Concerning the number of probes per hop, the best choice to maximize the information gathered by Service traceroute is 1 probe per hop. However, congested networks may drop probes and ICMP replies, so the packet concerning a specific hop is lost. Thus, we decided to use the default of 3 probes per hop to increase the probability to get a response from an interface.

Evaluation Settings. We decided to use two different configurations, one for videos and one for web pages. For videos, we use the HopByHop probing algorithm with a timeout of 2 seconds to wait ICMP replies. While, for web pages, we use the Concurrent probing algorithm. In both configurations, the maximum distance is 32 and the number of probes per hop is 3. The inter-probe time and inter-iteration time are negligible, i.e. 1 microsecond.

5. SIDE EFFECTS OF Service traceroute

This section evaluates whether Service traceroute affects target application flows as firewalls or servers may mistakenly interpret too many duplicated packets within a flow as an attack or losses, which in turn may cause application flows to be blocked or achieve lower throughput. Although the idea of piggybacking traceroute probes within application flows has been around for approximately a decade, there has been no prior evaluation of whether it can hurt target application flows. TCP sidecar evaluates the intrusiveness of their method, but only by measuring the number of abuse reports [10].

5.1 Metrics

We select different metrics to measure properties of target application flows. *Flow duration* refers to the time between the first and the last packet of a flow. For TCP, we measure the time from the server SYN to the first FIN or RST. For UDP, we measure the time from the first and the last packet coming from the server. We compute the *average throughput* of a target application flow as the total application bytes divided by the flow duration. In addition to these metrics, which we

can compute for both TCP and UDP flows, we have three other TCP specific metrics: the *number of resets*, which capture the number of target application flows closed by resets; *window size* is the difference between the minimum and the maximum TCP window size of the server for an application flow; and the *number of retransmissions* is the number of retransmission from the server per application flow.

5.2 Aborted flows

We first study whether Service traceroute causes flows to be aborted. We have seen no video sessions that ended with resets in our experiments with 153 streaming sessions of Twitch and 153 video sessions of Youtube with TCP. Even though our analysis is only from European vantage points, we believe that this result will hold more generally for both Twitch and Youtube as these large video providers deploy multiple versions of the same software across servers/caches [8]. Any differences will depend on middleboxes placed either close to the clients or in the path towards the service.

Our results for webpage downloads are also encouraging, we see no aborted flows. However, in the small scale measurement, we detected few exception. The IPs of the servers are different to the ones obtained in the European scale experiment. Five of the top-1000 websites (kompas.com, patria.org.ve, mobile01.com, costco.com and softonic.com) ended with resets only for downloads with Service traceroute. We experience the same behavior consistently across PlanetLab vantage points for three of the sites, whereas for kompas we only see resets for three of the vantage points. Our manual inspection of the traces shows that these three vantage points downloaded the kompas page from one IP, whereas the other four downloaded from another IP. This analysis suggests that some firewall close to the website or the web server itself is resetting the flows due to the duplicate packets. In the small scale experiment, we note that for 88% of the measured websites we see the content downloaded from different IPs depending on the vantage point, but only in one case we saw this difference in behavior across the IPs.

Takeaway. We conclude that for Twitch and Youtube sessions as well as for most of the top-1000 Alexa websites, Service traceroute causes no aborted flows. In a few cases, however, the target application flow is aborted when running with Service traceroute. It is hence key to evaluate the particular target service in a controlled setting before launching any large-scale experiments to trace paths of user's traffic.

5.3 Flow performance

We next evaluate whether Service traceroute affects flow performance in terms of flow duration, throughput, TCP window size, and retransmissions. Figure 3 presents the cumulative distribution function of the flow duration in seconds with and without Service traceroute. We present eight curves: two for video sessions over TCP both for Twitch and Youtube, two for Youtube sessions over UDP, and two for all web page downloads. We see that the distributions with and without Service traceroute are mostly the same. During the small scale experiment, for web pages we observe one exception for softonic.com, where with Service traceroute the flow duration takes 600 seconds more. In this case, we see that all packets trigger ICMP time exceeded. At the end, the application flow is closed with a reset. We assume that some middlebox in the path prevents packets with increasing TTL.

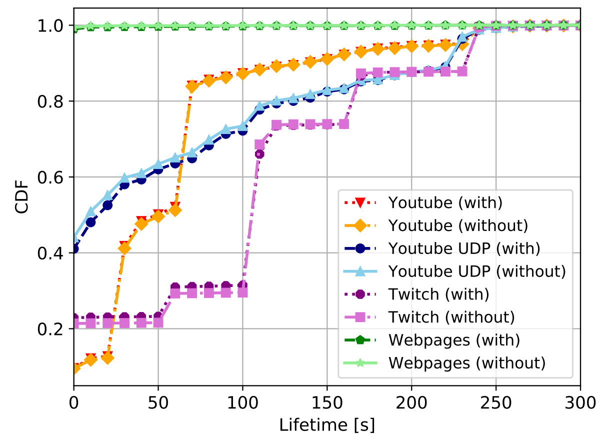


Figure 3: Flow duration distributions of target application flows with and without Service traceroute

Figure 4 presents the cumulative distribution of the average throughput of target application flows in our experiments for Web and video. Similar to flow duration, we observe that Service traceroute has no impact on target application flow throughput, since the evolution of distributions are the same without noticeable differences.

Figure 5 presents the cumulative distribution of the maximum difference of TCP windows size of target application flows for both web and video. The figure shows that Service traceroute does not affect the window size of the receiver, hence probes which are duplicate acknowledgments are not causing any reduction of the TCP window size.

Finally, Figure 6 shows the cumulative distribution of the number of TCP retransmissions of target application flows. In this case, web pages and twitch have the same distribution whether Service traceroute is used or not. For Youtube, there is a slight difference when Service traceroute is used but the number of cases is limited to 1% of Youtube application flows. The difference of the number of retransmissions is limited to

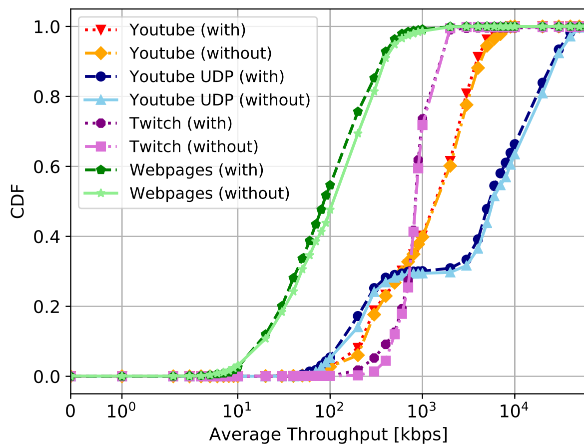


Figure 4: Throughput distributions of target application flows with and without Service traceroute

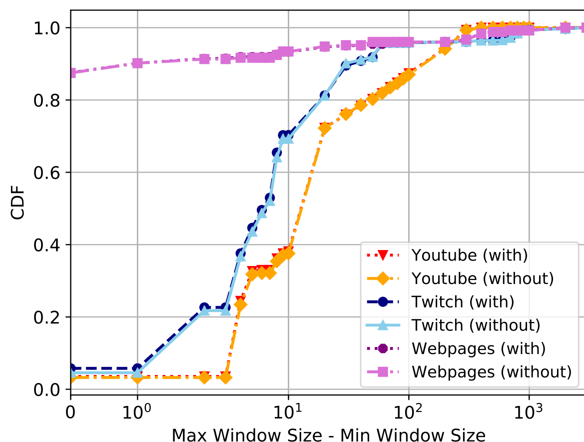


Figure 5: Difference of maximum and minimum window size of target application flows with and without Service traceroute

few application flows, so this issue may be due to some other factors.

6. COMPARISON WITH TRACEROUTE TOOLS

The key motivation for building Service traceroute is that we must send probes within the target application flow to discover the path of this flow. Although Luckie et al. [7] have observed different paths depending on the traceroute method (UDP, ICMP, or TCP), no prior work has studied how often piggybacking traceroute probes within application flows will discover different paths. This section compares Service traceroute with different traceroute probing methods using Paris traceroute. In addition, this section compares Service traceroute with 0Trace, which also piggybacks probes

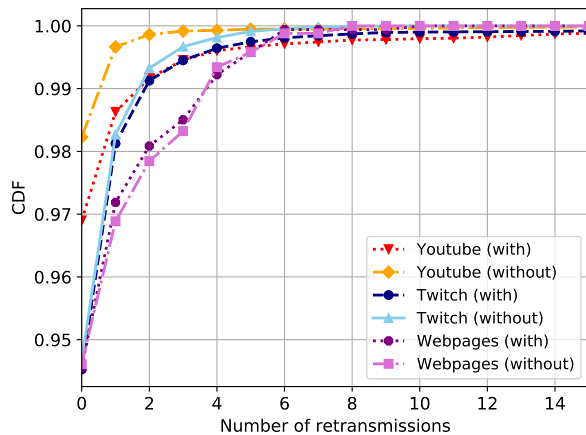


Figure 6: Number of TCP retransmissions of target application flows with and without Service traceroute

inside an application flow.

6.1 Metrics

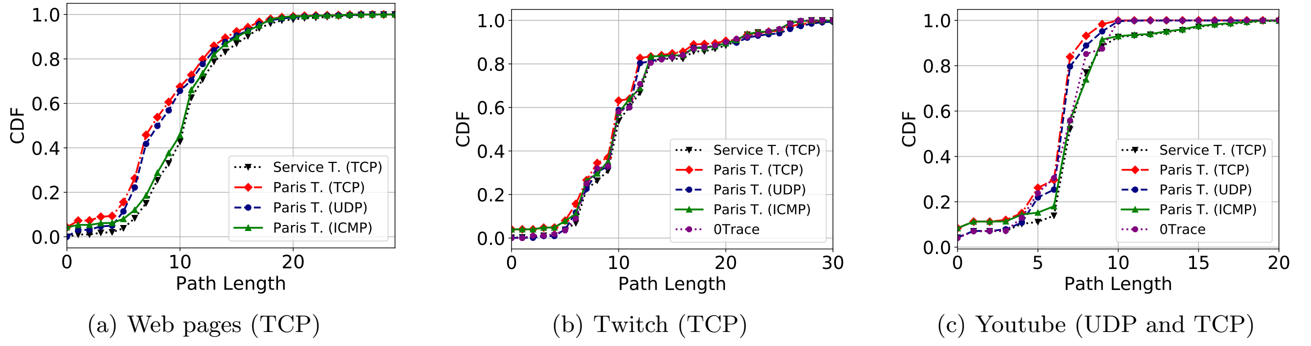
We select two metrics to compare the discovered paths. The *path length* captures the distance from the source to the last hop that replies to probes. For Paris traceroute, we take the length of the longest path in case of multiple paths. The *path edit distance* is the edit distance between the path discovered with Service traceroute and that discovered with 0Trace and with Paris traceroute (in case Paris traceroute returns multiple paths, we select the one with the smallest edit distance). The first metric captures whether one tool discovers more hops than the other, whereas the second captures how much the two outputs differ.

When we observe differences between paths, we analyze where the differences are in the path: origin AS, middle of the path, or destination AS. We map IPs to ASes using the RIPEstat Data API [9]. The location where the two paths diverge help us understand the placement of middleboxes.

6.2 Path differences with Paris Traceroute

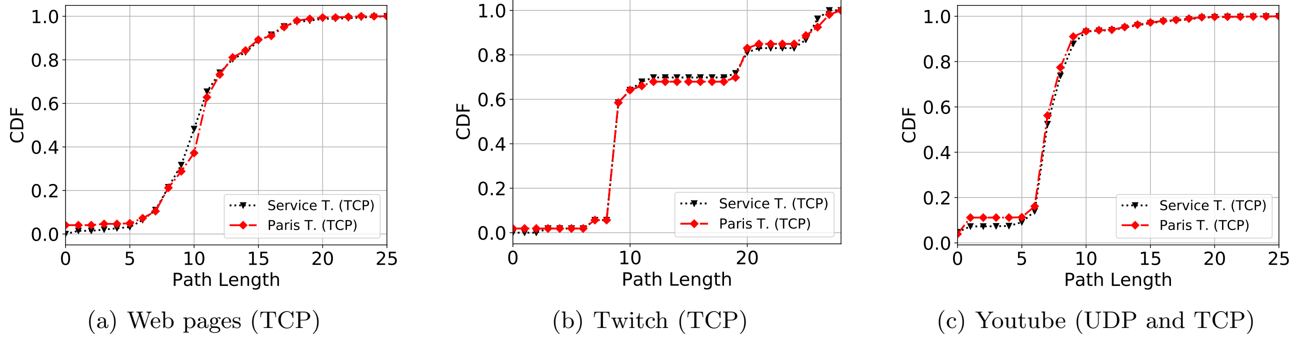
We compare Service traceroute with two versions of Paris traceroute. The multipath detection algorithm (MDA) enabled using TCP, ICMP, and UDP probes and the standard Paris traceroute using the same 5-tuple as the target application flow.

Figure 7 and Figure 8 present the cumulative distribution functions of path length for each service: Web, Twitch, and Youtube (UDP and TCP), using Paris traceroute MDA and standard Paris traceroute respectively. Figure 7 shows that for all three services, probing with TCP and UDP discovers less hops. While, Figure 8 shows that probing with the same 5-tuple of the target application flow discovers the same number of hops as



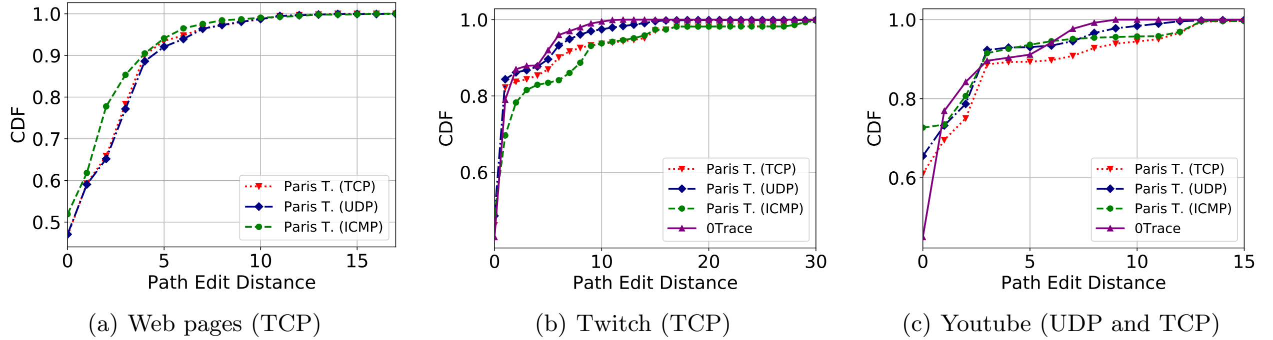
(a) Web pages (TCP) (b) Twitch (TCP) (c) Youtube (UDP and TCP)

Figure 7: Path length of paths discovered with Service traceroute, Paris Traceroute and 0Trace



(a) Web pages (TCP) (b) Twitch (TCP) (c) Youtube (UDP and TCP)

Figure 8: Path length of paths discovered with Service traceroute and standard Paris Traceroute



(a) Web pages (TCP) (b) Twitch (TCP) (c) Youtube (UDP and TCP)

Figure 9: Path edit distance between Service traceroute and Paris Traceroute, and between Service traceroute and 0Trace

Service traceroute. This result implies that firewalls close to the destination may prevent TCP and UDP traffic that corresponds to no active flow to reach the destination. In addition, this result confirms Luckie et al.[7]'s analysis from ten years ago, which showed that UDP probes cannot reach the top Alexa web sites as probes correspond to no active flow. ICMP and Service traceroute probes are able to obtain longer paths for all three services analyzed.

Figure 9 and Figure 10 compare the hops in the discovered paths using the path edit distance. A path edit distance of zero corresponds to the case when the Paris traceroute output contains Service traceroute's path. Figure 9 shows that the path discovered with Service traceroute only matches with the best path discovered by Paris traceroute MDA in about 55% of the webpage downloads, 50% of the Twitch sessions, and almost 75% of the Youtube streaming sessions. For Twitch, UDP discovers paths that are the most similar to Service traceroute's paths, whereas for both Web and Youtube, ICMP leads to the most similar paths. While, Figure 10 shows that for all services, standard Paris traceroute obtains the same paths of Service traceroute more often than Paris traceroute MDA, in about 63% of the Twitch

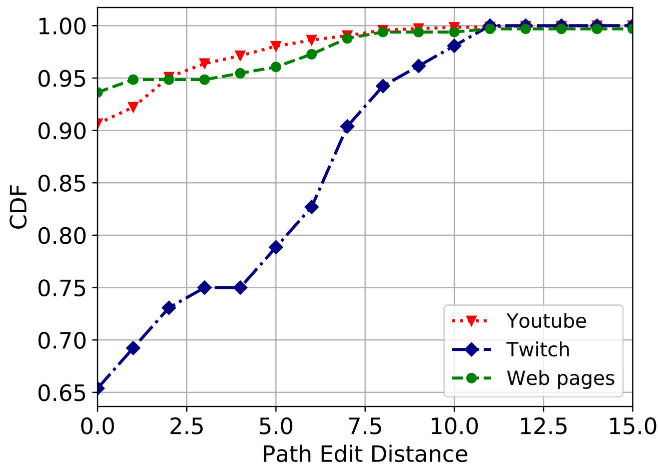


Figure 10: Path edit distance between Service traceroute and Paris Traceroute using the same source and destination ports

streaming sessions, 91% of the Youtube video sessions and 93% of the web downloads.

To help interpret these results we study the location of the points where Service traceroute's and Paris traceroute's paths diverge, which we call the *divergence point*. Table 1 shows the fraction of experiments with divergence points at the origin AS, the middle of the path, and the destination AS. We first consider the divergence points using Paris traceroute MDA. For Twitch, we observe a large number of divergence points in the middle of the path for all probing protocols. For ICMP, we also observe few divergence points on both origin and destination side. For Youtube, we see few divergence points when probing with UDP and TCP, with a peak of 2% on the destination side. This result, together with short path lengths and large path edit distances confirm that some hop in the path discards UDP and TCP probes. While, using ICMP probes, most of divergence points are on the destination side, and a few in the origin and in the middle of the path. Finally, for web pages, we notice a negligible numbers of divergence points when probing with TCP and UDP, mostly focused in the middle of the path. While, when probing with ICMP we observe a large number of divergence points on both middle (22%) and destination (20%) side. Similarly for Youtube, the low divergence points encountered when probing with TCP or UDP, the high path edit distance and the low path length confirms that sometimes probes are not able to reach the top Alexa web sites. For standard Paris traceroute, we observe more divergence points than Paris traceroute MDA. For web pages, the vast majority of divergence points are focused in the origin and in the middle of the path and fewer on the destination side. For Twitch, there is a significant number of divergence points in the middle of the paths, few on the origin side and a negligible numbers on the destination side. Finally, for youtube, there are few divergence points on both the origin and destination side. Using the same ports as the target application flow increases the probability to get the same path discovered with Service traceroute. Even if the number of divergence points is larger with the standard version of Paris traceroute, the fact that probes are not blocked as those of Paris traceroute MDA yields better results in terms of path edit distance, hence standard Paris traceroute's paths are the most similar to Service traceroute's paths when compared with Paris traceroute MDA.

Takeaway. Our analysis shows that the paths discovered by piggybacking traceroutes within the target application flow are often different by a few hops from paths discovered by typical traceroute tools. Although these results are for European vantage points and a small set of applications, they agree with previous large scale analysis of traceroute results with different methods [7]. As such, we argue that a tool like Service traceroute is crucial for accurately debugging Internet services.

6.3 Path differences with 0Trace

We compare Service traceroute with 0Trace. Similarly with Paris Traceroute, we studied the cumulative distribution functions of path length and path edit distance. Unfortunately, the download time for web pages is extremely short and our script was too slow to detect the target application flows and then run 0Trace, so for this comparison we focused only on Twitch and Youtube. This experience shows that simply running 0Trace for application debugging is not trivial and that integrating the identification of flows with path tracing as we do in Service traceroute is key for application debugging. In the case of path length for Twitch, we observe that the distributions of path length of 0Trace and Service traceroute are similar. While for Youtube, the path length reaches the maximum distance of 10 hops, which is the same as Paris Traceroute MDA with TCP and UDP. This is quite surprising, since Service traceroute and ICMP are able to discover longer paths. This result suggests that some middleboxes near the destination discards both Paris Traceroute and 0Trace probes. From the analysis of 0Trace code, the tool copies the 5-tuple identifier of the application flow, the sequence and acknowledgment numbers. However, during the execution it increases the sequence number by one for each probe and it does not update the sequence number when newer packets are transmitted. Thus, after the first transmitted probe from 0Trace, the other probes have a wrong sequence number compared to the target application flow and middleboxes might detect it and drop

	Web Pages (TCP)			Twitch (TCP)			Youtube (TCP and UDP)		
Configuration	Origin	Middle	Dest.	Origin	Middle	Dest.	Origin	Middle	Dest.
MDA UDP	0.54	5.22	1.86	0.34	18.88	0.26	0.10	0.18	1.15
MDA TCP	1.02	4.98	1.26	0.32	17.31	0.26	0.15	0.15	1.89
MDA ICMP	4.86	21.94	19.96	3.43	44.31	1.54	0.68	0.56	17.94
Standard PT	4.07	5.81	2.03	6.50	18.95	1.20	4.08	1.47	8.61
0Trace	-	-	-	6.18	39.34	9.87	3.52	48.43	8.59

Table 1: Location of divergence points [%]

the probes for security reasons.

Considering the path edit distance, 0Trace obtains the same paths of Service traceroute in 45% of cases for both Youtube and Twitch. Comparing the distribution of path edit distance between 0Trace and Paris Traceroute, we observe that 0Trace obtains paths that are more similar to paths obtained from Service traceroute. Similarly as Paris Traceroute comparison, we study the location of points, where Service traceroute's and 0Trace's paths diverge, the divergence points. For both Twitch and Youtube, we observe a significant number of divergence points focused in the middle of the path (39% for Twitch and 48% for Youtube) and some on both the origin and destination side. The reason of these high values is that tracing paths of the same application flows but with different 5-tuple identifiers may yields different paths. Thus, even running the same script for Service traceroute and 0Trace, paths might differ at least one divergence point. Paris Traceroute MDA's paths have less divergence points because Paris Traceroute MDA returns multiple paths between two end hosts and one of these might be the one Service traceroute discovered.

Takeaway. Our analysis shows that 0Trace obtains paths that are more similar to Service traceroute's paths than comparing with Paris Traceroute's paths. However, some middleboxes detect the probes of 0Trace from the wrong sequence numbers and hence, for security reasons middleboxes discard the probes. Thus, preserving the sequence and acknowledgment numbers as Service traceroute does is key to reach the destination and obtain all replying hops when tracing the path of an application flow.

7. RELATED WORK

Since Jacobson's original traceroute tool [2], a number of new versions have emerged with different features and with new methods for constructing probes (e.g., Paris traceroute [1, 12] and tcptraceroute [11]). All these traceroute versions have a drawback for the goal of diagnosing a target application flow because they start a new flow to send probes. As such, middleboxes may treat them differently than the target application flow. Service traceroute avoids this issue by piggybacking traceroute probes within active application flows. This idea was first introduced in paratrace [4], which is no longer available, and then re-implemented in 0trace [3] with the goal of tracing through firewalls and in TCP sidecar [10] for reducing complaints of large-scale traceroute probing for topology mapping. Unfortunately, none of these tools is actively maintained. Service traceroute adds the capability of automatically identifying application flows to trace by a domain name, of tracing UDP flows as well as of tracing multiple concurrent flows that compose a service. We release both a command-line and a library version as open source. Furthermore, we present an evaluation of the side-effects of piggybacking traceroute probes within application traffic as well as of its benefit by comparing the differences with Paris traceroute and with 0Trace. Our characterization reappraises some of the findings from Luckie et al. [7], which show that the discovered paths depend on the protocol used in the probes. Their study, however, includes no traceroute tools that piggyback on application flows.

8. CONCLUSION

In this paper we present Service traceroute, a tool aimed at tracing paths from multiple concurrent flows generated by modern Internet services. Results from an European-scale evaluation performed using popular web and video services show that in the vast majority of cases, Service traceroute has no side-effect on the target application. It obtains paths not discovered by Paris Traceroute MDA in more than 40% of the traced paths, up to more than 70% with Youtube. While, it obtains paths not discovered by standard Paris traceroute using probes with the same 5 tuple as the target application flow in more than 7% of the traced paths, up to 37% with Twitch. 0Trace has a lower path edit distance compared with the path edit distance of Paris traceroute MDA, but the path length is generally lower than Service traceroute. In future work, we plan to add the support of IPv6 to Service traceroute. We further plan to perform a large-scale characterization of results of Service traceroute, across a wide-variety of services and global distributed vantage points. As part of this, we will evaluate possible side effects on different type of services different from video.

Acknowledgments

This work was supported by the French ANR Project no. ANR-15-CE25- 0013-01 (BottleNet), by the Inria Project Lab BetterNet, and by a Google Faculty Award.

9. REFERENCES

[1] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding Traceroute Anomalies with Paris Traceroute. In *Proc. IMC*, 2006.

[2] V. Jacobson. traceroute, Feb 1989.

[3] Jake Edge. Tracing behind the firewall, 2007. `https://lwn.net/Articles/217076/`.

[4] D. Kaminsky. Parasitic Traceroute via Established TCP Flows & IPID Hopcount. `https://man.cx/paratrace`.

[5] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196. ACM, 2017.

[6] M. Luckie, A. Dhamdhere, B. Huffaker, D. Clark, et al. Bdrmap: Inference of borders between ip networks. In *Proceedings of the 2016 Internet Measurement Conference*, pages 381–396. ACM, 2016.

[7] M. Luckie, Y. Hyun, and B. Huffaker. Traceroute Probe Method and Forward IP Path Inference. In *Proc. IMC*, Vouliagmeni, Greece, 2008.

[8] Netflix Open Connect OVerview. `https://openconnect.netflix.com/Open-Connect-Overview.pdf`.

[9] RIPEstat Data API. `https://stat.ripe.net/docs/data_api`.

[10] R. Sherwood and N. Spring. Touring the internet in a tcp sidecar. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 339–344. ACM, 2006.

[11] M. Torren. Tcptraceroute-a traceroute implementation using tcp packets. man page, unix (2001). See source code: `http://michael.toren.net/code/tcptraceroute`.

[12] D. Veitch, B. Augustin, T. Friedman, and R. Teixeira. Failure Control in Multipath Route Tracing. In *Proc. IEEE INFOCOM*, 2009.