Leveraging eBPF to Make TCP Path-Aware

Mathieu Jadin*, Quentin De Coninck*§, Louis Navarre*, Michael Schapira† and Olivier Bonaventure*
*ICTEAM, UCLouvain, Belgium, firstname.lastname@uclouvain.be
†Hebrew University of Jerusalem, Israel, schapiram@cs.huji.ac.il

***Abstract*—The Transmission Control Protocol (TCP) is one of the key Internet protocols. It is used by a broad range of applications. TCP was designed when there was typically a single path between a client and a server. Today's networks provide higher path diversity, yet TCP still only uses the single path selected by the network layer. This limits the ability of TCP to react to events such as interdomain failures or highly congested peering links.**

We propose the TCP Path Changer (TPC), a set of eBPF programs that are incorporated into the Linux TCP/IP stack to make it more agile. To illustrate the benefits of our approach, we first demonstrate that TPC can quickly reroute an ongoing TCP connection around a failure. We then show that TPC can also monitor the round-trip-time of active TCP connections and automatically reroute them if it becomes too high. Our evaluation of TPC in emulated networks evidences the significant performance benefits of a path-aware transport protocol.

***Index Terms*—TCP, eBPF, IPv6 Segment Routing,**

I. Introduction

The Transmission Control Protocol (TCP) was designed in the 1970s. Yet, despite its old age, TCP remains the dominant transport protocol in today's Internet, controlling more than 90% of the overall network traffic [1]. The protocol and its implementations have evolved during the last decades. While today's TCP implementations still use the original wire format [2], they include various improvements, including congestion control techniques [3], [4], support for larger windows [5], selective acknowledgments [6], and more.

TCP reflects the layering principle and considers the underlying network layer as a blackbox that exposes IP addresses to the transport layer in which TCP operates. A TCP connection is always bound to the IP addresses of the client and the server. TCP is agnostic to the network path between the communicating hosts that the packets traverse and, in particular, neither selects nor influences this path in any way. As TCP performance can be adversely affected by packet reordering, most networks avoid spreading packets belonging to the same flow across different paths [7] (even though recent advances such as RACK make TCP less sensitive to reordering [8]). Two main types of routing events induce path changes that can impact TCP connections: (i) link or node failures [9], [10] and (ii) changes due to in-network traffic engineering [11]. TCP reacts to path changes induced by such events by adjusting its round-trip time estimation, retransmitting lost packets, and possibly adjusting its congestion window.

When TCP was designed, networks supported one path for a given source-destination pair [12] and so TCP's role was to adapt its transmission rate to the traffic conditions on this path. Today's networks support a higher number of paths that can potentially be used by a TCP connection between two hosts [13]. In particular, many networks provide multiple equal cost paths towards internal destinations, and routers load-balance packets from different TCP connections across these paths [7]. Measurement studies show that although equal cost paths are supposed, in principle, to be comparable, performance-wise, this is not always the case [11], [14]. The specific path traversed by the packets of a TCP connection in such networks mainly depends on the client's selected source port (since the IP addresses and server port are fixed). Researchers have proposed to change the client's TCP port [15], [16] to influence a TCP connection's path, but this solution has not been widely adopted. A similar approach was proposed for the context of Multipath TCP [17] in datacenters [18].

Using the TCP client ports [15], [18] or other packet fields [16] to indirectly influence a TCP connection's path is fragile. For this reason, the designers of IPv4 and IPv6 proposed a more explicit solution with loose source routing [19], [20]. Unfortunately, this approach was deprecated due to security concerns [21], [22]. Recently, the Internet Engineering Task Force revisited this topic and adopted the Segment Routing architecture [23], [24]. Segment Routing is a modern variant of source routing and can be applied in MPLS and IPv6 networks. Specifically, IPv6 Segment Routing (SRv6) enables endhosts to select the path followed by their packets. With Segment Routing, a network path becomes a succession of shortest paths that are encoded as a loose source route in the packet header. Segment Routing has already been applied to address a variety of networking problems, ranging from traffic engineering [25]–[27] to fast restoration [28].

We leverage Segment Routing and the expressiveness afforded by extended Berkeley Packet Filter (eBPF) programming [29] to demonstrate how TCP can become path-aware, i.e., making TCP able to react to network-level events not only by modulating the transmission rate but also by selecting an alternate path. eBPF is a virtual machine included in the Linux kernel[1] that can be used to tune the Linux TCP/IP stack. We take advantage of this expressiveness to devise the TCP Path Changer (TPC) and implement it as a set of eBPF programs. Using eBPF provides extensive flexibility for an application to *customize* TPC to its specific requirements. For instance, some applications might require very low network latency while others might be more sensitive to route instability.

§FNRS Post-doctoral Researcher.

[1]Although we focus on Linux in this paper, we note that there are ongoing efforts to also port eBPF into Microsoft Windows [30] and FreeBSD [31].

To demonstrate the usefulness and flexibility of the TPC, we evaluate it on two different use-cases: (1) recovery from distant link failures, and (2) dynamic selection of lowest-delay paths. Our results for the first use-case show that TPC can quickly react to link failures in a distant network, enabling TCP connections to continue operating without being severely affected. This should be contrasted with the time needed to converge to a new global routing configuration, which may require several seconds, if not much longer. Our second use-case is motivated by highly interactive applications such as web services or request-response applications, whose performance is heavily affected by network delays. We propose a TPC that monitors the round-trip-time of TCP connections and reroutes them to a shorter-delay path. This TPC employs an online learning algorithm to make good decisions despite uncertainty regarding its environment, and provides significant improvements in performance over path-oblivious transport.

This paper is organized as follows. Section II presents the motivation for our work and some necessary background. In Section III we propose the TCP Path Changer (TPC) and demonstrate how it enables the Linux TCP/IP stack to detect distant link failures and react by rerouting the affected connections. Section IV demonstrates how our proposed TPC monitors the round-trip-time of connections and reacts by rerouting the flows that suffer from excessive delays. We discuss related work in Section V and conclude in Section VI.

II. Motivation

Today's hyperscale datacenters, such as Amazon's AWS, Google Cloud and Microsoft's Azure, host hundreds of thousands of servers that are interconnected via a private globe-spanning network that is also connected to a large number of Internet Service Providers. In addition to these hyperscalers, there is a myriad of smaller datacenter networks that host up to tens of thousands of servers. These smaller datacenters are frequently used by smaller companies to host internal servers and also provide public services such as web sites, file and backup servers, game servers, etc. Little public information is available about these datacenters and their networks.

Interestingly, OVH, a French hosting company, provides public information about the topology of its datacenters and backbone infrastructure[2]. As of April 2022, OVH maintains 28 datacenters located in 19 countries and host more than 300,000 servers. Most enterprises deploy servers in multiple geographic locations to cope with datacenter failures [32]. Content providers also often need to maintain multiple servers at different locations to offer low latency services.

Examining the topology of OVH's backbone and the locations of its datacenters, Fig. 1 reveals the number of *distinct* peering links between OVH and any external peer network. We filter parallel links and regard an external peer as having x *distinct* connections to OVH when that peer is directly connected to x different OVH routers. Fig. 1 shows that roughly $40\,\%$ of the peers in OVH's Europe backbone have two or more distinct connections with OVH. Note that the number of peering links is indicative of the importance of a peer, both in terms of traffic volume and of business considerations.

[2]See http://weathermap.ovh.net and http://peering.ovh.net.

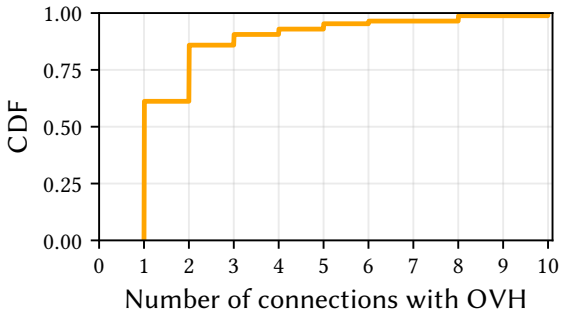

Fig. 1. Distribution of the number of *distinct* connections between OVH and external peers in the Europe backbone.

Another interesting observation regarding the OVH network relates to the distance between a given datacenter and these multi-connected peers. We measured the difference in the number of hops between each datacenter i and external peer j, restricting our attention to external peers with at least two *distinct* peering links. We computed the shortest path from datacenter i to all *distinct* connections of peer j and quantified the length difference between the different paths. Fig. 2 plots the CDF of these results when aggregating over all i's and j's. In almost $60\,\%$ of the samples, the *shortest* shortest-path and the *second shortest* shortest-path have the same number of hops. This suggests that there may be many multiple available paths with comparable performance interconnecting a datacenter and an external peer.

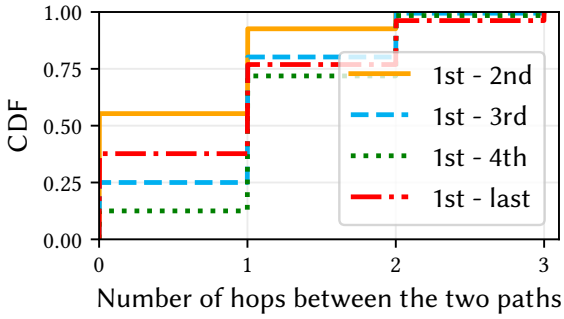

Fig. 2. For each external peer - OVH datacenter pair, we compute all shortest paths from the datacenter to a *distinct* connection with the peer. The figure shows the difference in the number of hops between all these paths for each pair in the Europe backbone. The paths are sorted by the number of hops.

A. Path-aware datacenter servers

While datacenter networks provide multiple paths to reach most Internet destinations, in practice, most servers are oblivious to this diversity and treat the underlying network as a blackbox. Specifically, each server uses the route selected for it by the first-hop router, entrusting the network with the responsibility to choose the best path towards any destination.

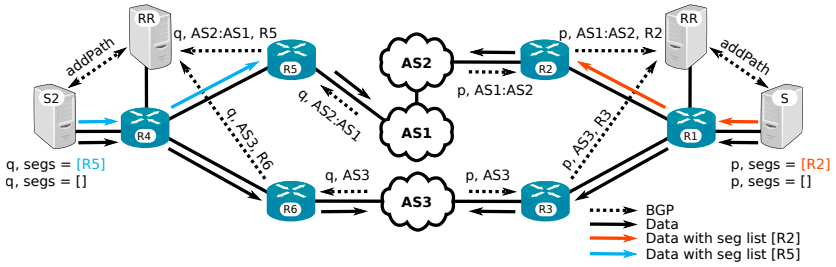

Fig. 3. These servers learn several paths to important prefix using BGP Add-path and use them to send packets with the IPv6 Segment Routing Header.

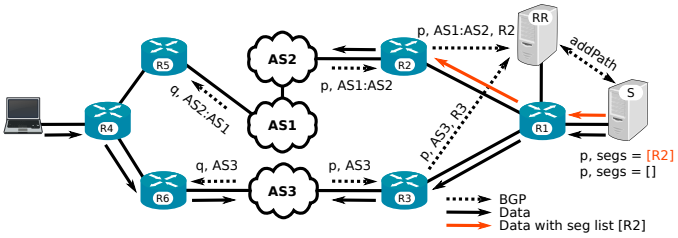

Fig. 4. These servers learn several paths to each important prefix using BGP Add-path and use them to forward packets with the IPv6 Segment Routing Header (SRH), while clients only use one path.

We propose a new service that datacenter providers can offer to customers needing higher performance (e.g., towards specific destinations) than the one provided by the default paths, as well as faster recovery from link failures. In our scheme, instead of blindly using the routes selected by first-hop routers, these servers learn additional routes themselves and actively select among them. For this, we install a BGP daemon on each of these servers and connect it to one or two BGP route reflectors. We configure these route reflectors to only send routes to these servers and never accept routes from them. To distribute alternate paths, we leverage the BGP Add-Path extension [33] enabling a BGP router to advertise multiple routes towards the same prefix over iBGP sessions. We point out that it is possible to configure BGP filters so as to only advertise multiple paths for the most important prefixes.

Fig. 3 and Fig. 4 illustrate this utilization of BGP Add-Path in a simple datacenter network connected to two different BGP peers. The server `S` uses BGP Add-Path on its iBGP session with a local route reflector. This route reflector learns the two routes from `AS2` and `AS3` through the `R2` and `R3` BGP next-hops. The server learns the two paths from the route reflector. If the server sends a regular IP packet, it will follow the default route, e.g., via `R3`. If the server prefers, for any reason, the path via `R2`, it simply needs to add an SRv6 header to forward those packets via `R2`. `R2` removes the Segment Routing Header (SRH) and forwards it to its destination, through `AS2`. This is illustrated via the plain arrows in the figures.

To offer fast recovery from failures, the alternative routes are injected to both sides of the connection since a failure can affect both sides, e.g., in the context of inter-datacenter communication, as depicted in Fig. 3. However, to offer a best path discovery service, we only need to control one side of the connection. Fig. 4 illustrates an asymmetric scenario where the server benefits from path diversity but not the client.

B. eBPF makes the TCP/IP stack programmable

Historically, TCP implementations are structured as monolithic code that can be configured at the granularity of individual connections using system-wide parameters [34], [35] and socket options. The Linux TCP/IP stack, which is dominant on servers, can also be tuned by leveraging eBPF [29]. eBPF is a virtual machine included in the Linux kernel that supports limited RISC-like assembly language. System administrators can attach eBPF programs using different hooks inside the kernel. A static verifier, packaged with the kernel, checks memory calls and program termination of the code before injection to guarantee that the code does not make the kernel crash. Different types of eBPF programs have been developed [36]; some collect statistics about the utilization of the kernel data structures, while others monitor the operation of system calls, etc. Brakmo [37] extended the Linux TCP stack to enable it to execute eBPF programs when specific events occur. Researchers have used this framework to support new TCP options [38]. Here, we propose eBPF programs that enable the Linux TCP stack to change the path of an established TCP connection in response to network failures or when the path is not providing the expected delay. We realize such path changes using the IPv6 Segment Routing implementation within the Linux kernel [39]. This scheme could also be realized via multi-topology routing [40] in IPv4 networks.

C. The TCP Path Changer (TPC)

By putting together the different ingredients described above, we demonstrate how TCP can become path-aware. We propose the TCP Path Changer (TPC) and implement it as a set of eBPF programs in the Linux TCP/IP stack. In the next two sections, we present two flavors of the TPC, each targeting a specific use-case. In Section III we show how the TPC enables the detection of distant link failures and rerouting of affected connections. In Section IV, we present an online-learning-based TPC that monitors connections' round-trip-times and automatically reroutes flows that suffer from excessive delays.

III. Recovering from Distant Failures

Datacenter networks must preserve connectivity with peering networks in the presence of various types of link and router failures, which have been shown to be frequent in ISP [9] and datacenter networks [41]. To address this, network operators have deployed various fast reroute techniques [42], [43], enabling transition to new routes within less than a few

tens of milliseconds for wide area links. When a peering link [44] or a distant network fails, affected prefixes might be unreachable for several seconds or more. For this, researchers have proposed techniques such as Blink [45], where routers monitor the TCP flows and automatically reroute the flows when their destination suddenly appears unresponsive.

Here, we explore a simpler approach for coping with these distant failures: when such a failure occurs, the servers that exchange data with the affected destinations quickly detect that these became unreachable and leverage knowledge of alternate BGP next-hops to reroute the affected TCP connections.

A. Triggering rerouting upon path failure

Consider the network shown in Fig. 3. The server sends packets towards a destination that belongs to prefix p learned from both `AS2` and `AS3`. How can the server detect a transient failure on this path? A first approach would be to wait until the reception of an ICMP destination unreachable message from an intermediate router. This is unlikely to succeed since routers strictly limit the rate of transmission of ICMP messages. Instead, we leverage the fact that our server will stop receiving acknowledgments for the data sent towards this prefix. Its retransmission timer will expire and the unacknowledged data will be retransmitted. We use these retransmissions as triggers for executing the eBPF code that selects an alternate path. We present two strategies for detecting path failures: NRTOChanger(N,M) and TimeoutChanger(T,M).

NRTOChanger(N,M) selects a path during the handshake (i.e., when sending the SYN or the SYN+ACK). Then, NRTOChanger(N,M) monitors the retransmission timer and the reception of duplicated data for each TCP connection, and selects an alternate path after N successive expirations or M retransmissions from the other end.

Failures can be asymmetric, that is, a failure can affect one direction of the communication and not the other. If the failure occurs on the path from the sender to the receiver (see Fig. 5), NRTOChanger(N,M) waits for N retransmission timeouts (RTO) before selecting another path. Assuming that only the current path fails and that TCP uses standard exponential backoff [46], which doubles the RTO each time, it takes the sender $\sum_{i=0}^{N-1} 2^i \cdot RTO_{sender}$ to switch to a new path.

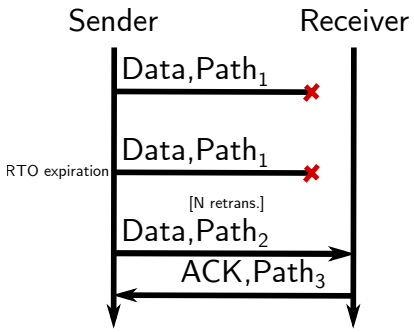

Fig. 5. Detecting failures on the forward path.

The failure could also affect the return path. Fig. 6 shows its detection. NRTOChanger(N,M) also monitors the sending of duplicated acknowledgments. These duplicated acknowledgements are sent upon receiving duplicated data. Since the return path fails, the duplicated acknowledgments are lost. The sender thus resend its data which will trigger other duplicated acknowledgments. NRTOChanger(N,M) waits for M duplicate acknowledgments sent and therefore the number of retransmissions received from the sender (see Fig. 6) before selecting another path. Under the same assumptions as before, this will involve a time duration of $\sum_{i=0}^{M} 2^i \cdot RTO_{sender}$ from the reception of the data and the sending of an acknowledgment that will reach the sender.

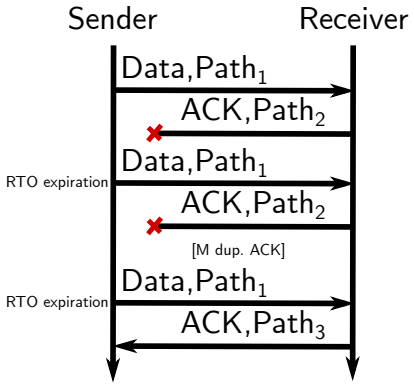

Fig. 6. Detecting failures on the return path.

Coordination between both entities would be useful from a performance viewpoint. Custom TCP options defined in eBPF [38] can carry these data. Indeed, we want M, the number of duplicated acknowledgments before changing the receiver's path, to be lower than N, the number of retransmission timer expirations, before changing the sender's path. Still, the NRTOChanger(N,M) on both endhosts will eventually converge to a working path if they have access to at least one path reaching the peer. The receiver's path only changes when the sender uses a working path. So, eventually, the receiver will use the right path. The sender might loop over its paths several times to make the receiver switch to a correct path.

Note that the lowest acceptable values for M and N are respectively 3 and 2. Operators should increase them if they expect many transient link flaps and very heavy congestion on all the paths. However, it is not advised to decrease M and N below these values. Indeed, to ensure a quicker recovery time, M should be strictly smaller than N, and setting M to 1 means changing path after seeing a single retransmission.

TimeoutChanger(T,M) works differently. When encountering retransmissions, it selects an alternate path when some data remains unacknowledged for more than a predetermined time interval of length T. This time value could be selected by the application developer, e.g., $200\,\mathrm{ms}$ for interactive applications. We keep M, the number of duplicated acknowledgments sent before changing the path. TimeoutChanger strategy is more convenient for the interactive applications that require delivery within some deadline. It provides a more concrete parameter for these applications than the number of RTOs.

B. Implementation

We implement TPC as a set of eBPF programs attached to the socket operations because such programs can access the state of the TCP socket in Linux kernel 5.3. We extended the Linux kernel to add new eBPF helpers for the presented use-cases. Our eBPF programs are triggered upon a series of events like the establishment of a connection, a retransmission,... This is more efficient from a performance viewpoint than attaching eBPF to every packet transmission [38].

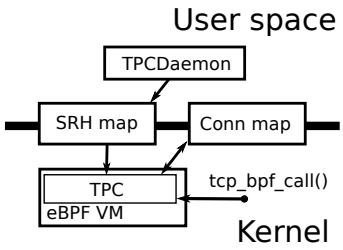

Fig. 7. Path management code talks to the Path Daemon through eBPF maps.

Fig. 7 shows the interactions between the different components inside the endhost. The TPC code runs in an isolated eBPF VM within the kernel. In the TCP stack, the injected code receives a structure containing the opcode of the hook that triggered it, as well as the five-tuple and some additional variables of the TCP connection, such as the minimum RTT or the congestion window. Moreover, it can read and write to memory chunks, called eBPF maps, which can be accessed by a user-space application. In this architecture, the TPC daemon fills a first eBPF map, the *SRH map* storing the IPv6 Segment Routing Headers (SRH) towards the different BGP next-hops. Since eBPF injected code does not have global variables or heap, we use a second eBPF map, the *conn map*, mapping a five-tuple to a structure containing data about the connection.

Listing 1 TPC: initialization

```
int handle_sockop(struct bpf_sock_ops *skops) {
  struct connection new_conn;
  struct conn *conn = bpf_map_lookup_elem(&conn_map,
                      &five_tuple);
  if (!conn) { // New connection
    new_conn = new_connection_state();
    new_srh = get_srh(0);
    move_to_path(skops, new_srh);
    bpf_map_update_elem(&conn_map, &five_tuple,
                        &new_conn);
    conn = &new_conn;
  }

  switch (skops->op) {
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
      // End of three-way handshake
      new_srh = get_srh(0);
      move_path(&dest_map, flow_id.remote_addr,
                flow_info->srh_id, skops);
      bpf_map_update_elem(&conn_map, &five_tuple,
                          conn);
      break;
    case BPF_SOCK_OPS_STATE_CB:
      // End of a connection
      if (skops->args[1] == BPF_TCP_CLOSE)
        bpf_map_delete_elem(conn_map, &five_tuple);
      break;
    // [...]
  }
  return 0;
}
```

Listing 1 shows the pseudo code for the path management initialization. Our TPC initializes its connection structure and sets the path upon actively starting the connection on the client-side. This way, the `SYN` follows the chosen path. However, we cannot do the same for the `SYN+ACK`. This is because the Linux kernel uses a specific structure to represent a non-established TCP connection, called a request socket. This structure is more lightweight than regular sockets to prevent `SYN` flooding attacks [47]. In particular, it does not support the socket options that are required to specify a specific path for the `SYN+ACK`. The kernel copies the contents of the request socket in a regular socket at the end of the TCP three-way handshake with the `BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB` hook. The `BPF_SOCK_OPS_STATE_CB` hook is used to cleanup the state of the connection. It is possible to circumvent the issue by adding an SRH to the `SYN+ACK` segments in eBPF program hooked to a TC on egress. This is left for future work.

Listing 2 TPC: reacting to failure

```
int handle_sockop(struct bpf_sock_ops *skops) {
  // [Initialisation presented before]

  int new_id = (conn->srh_id + 1) % nbr_srhs;
  switch (skops->op) {
    // [Other cases presented before]

    case BPF_SOCK_OPS_DUPACK:
      // Duplicated Acknowledgement is going to be sent
      if (conn->last_rcv_nxt != skops->rcv_nxt) {
        flow_info->last_rcv_nxt = skops->rcv_nxt;
        conn->remote_retransmission_count = 1;
        return 0;
      }
      conn->remote_retransmission_count += 1;
      if (flow_info->remote_retransmission_count < M)
        return 0;
      struct srh new_srh = get_srh(new_id);
      move_to_path(skops, new_srh);
      break;
    case BPF_SOCK_OPS_RTO_CB:
      // Retransmission timer expiration
      if (conn->last_snd_una != skops->snd_una) {
        flow_info->last_snd_una = skops->snd_una;
        conn->local_retransmission_count = 1;
        return 0;
      }
      conn->local_retransmission_count += 1;
      if (flow_info->local_retransmission_count < N)
        return 0; // Not enough retransmissions

      struct srh *new_srh = get_srh(new_id);
      move_to_path(skops, new_srh);

      break;
  }
  return 0;
}
```

As shown in Listing 2, during the transfer, the eBPF program is triggered by expirations of the local retransmission timer (`BFP_SOCKS_OPS_RTO_CB`) or the transmission of duplicate acknowledgments (`BPF_SOCKS_OPS_DUPACK`), the consequence of a retransmission from the other endhost.

TPC allows detecting complete path failures or extreme network congestion. It maintains two distinct counters for this purpose: `local_retransmission_count` and `remote_retransmission_count`. This counter is reset when the data transmission advances. If the local counter reaches N or the remote one reaches M, TPC calls `get_srh`

to find a new path. Note that the value of the N and M thresholds can be specified when the eBPF program is loaded.

Listing 3 Path migration

```
void move_to_path(struct bpf_sock_ops *skops,
                  struct srh *new_srh) {

  // Actual move
  bpf_setsockopt(skops, SOL_IPV6, IPV6_RTHDR, new_srh,
                 sizeof(*new_srh));

  // Reset TCP metrics
  bpf_setsockopt(skops, SOL_TCP, TCP_PATH_CHANGED,
                 &val, sizeof(val));
}
```

TPC changes the path of the TCP connection by calling two new socket options with `bpf_setsockopt` as shown in Listing 3. The first one sets the SRH on the path. As TCP does not have any information about the level of congestion on the new path, the second socket option resets the retransmission timer. Indeed, the RTO is doubled at each retransmission [46] to prevent the connection from worsening an already congested path. When TPC moves from one path to another disjoint of the first one, it is pointless to use high RTO values.

C. Evaluation

We experiment on networks emulated using Mininet [48] on a Linux kernel 5.3 running on a virtual server emulated by QEMU, equipped with 20 virtual CPUs and 16 GB of RAM. To emulate links, we use `tc-netem` to set their delays and `tc-htb` to set their bandwidth. The other parameters of these tools are the default ones. For our evaluation, we use the topology shown in Fig. 3. This network consists of 100 Mbps paths with 6 ms RTT. In the figure, the bottom interdomain path corresponds to the preferred next-hop and the top to the alternate one. We disable BGP failure convergence to demonstrate TPC's failure recovery without emulating a network with more ASes to delay BGP convergence.

The laptop initiates a given number of connections throttled at 10 Mbps using `iperf3` towards the server. The default initial congestion window in Linux is 10 packets. We fix N (i.e., successive timer expirations) to 3 and M (i.e., the maximum number of retransmissions from the peer) to 2. After 10 seconds, we emulate a failure of the primary path by introducing 100 % packet loss with `tc-netem` so that no ICMP messages are generated. Then, we observe the time needed to send traffic to the working path for each side of the connection. We repeat this experiment 100 times and provide a cumulative distribution function (CDF) of the recovery time.

NRTOChanger(N=3,M=2) can quickly reroute TCP connections after failures. Fig. 8 shows a CDF of the recovery times of the sender and the receiver. In the median experiment, the sender recovers first after 1600 ms. This is the time duration required by the sender to experience three successive retransmission timeouts.

The Linux kernel computes the RTO slightly differently from what is advised by the IETF [46]. The IETF advises using the following formula:

$$RTO = max(H, SRTT + max(G, K * RTT_{VAR}))$$

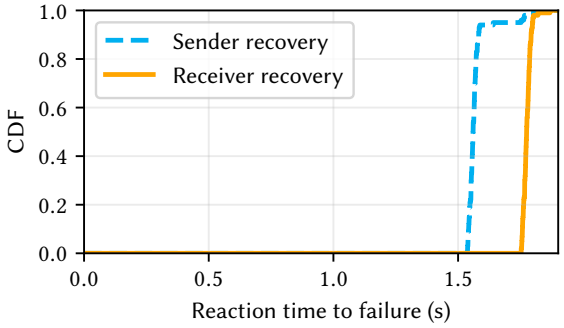

Fig. 8. CDF of the recovery time with one connection with NRTOChanger(N=3,M=2).

with H set to 1 second, K set to 4, G being the clock granularity, $SRTT$ being the smoothed RTT and RTT_{VAR} being an estimation of the RTT variance. However, the Linux kernel sets H to 0, K to 1, and G to 200 ms. The RTT being 6 ms, having only one connection in the network, the base RTO will always be close to 206 ms. This base RTO is doubled at each consecutive timeout. For this reason, the sender using TPC waits for 3 timeouts before changing its path. Therefore, it waits at least $206 + 2 \cdot 206 + 4 \cdot 206 = 7 \cdot 206 = 1442$ ms.

The difference between our lower bound, 1442 ms, and the experienced one, 1560 ms, can be explained by three factors. First, the smoothed RTT can be slightly inaccurate because of delayed acknowledgments, which increase the RTT by a few milliseconds. Second, the rate at which the retransmission timer is checked is at a maximum resolution of 4 ms. The difference of 116 ms (for the median) is the time required to resend the window. The retransmission timer is started when the last packet is sent. This last source of delay is the main reason behind the variation that we observe in the CDF.

The receiver needs 225 ms more in the median case to change its path. Indeed, TPC changes its path after 2 retransmissions of the same acknowledgment. The receiver sent the initial acknowledgment during the failure. The first retransmission is triggered by the retransmission of the data on the working path. The receiver will wait a last retransmission from the server, at least 206 ms later. Because of the time needed to send the window, we observe an increase of around 225 ms.

TimeoutChanger(T,M) also detects failures. Instead of relying on a given number of retransmissions, TPC can redirect if the same bytes are in flight for more than a given time that we arbitrarily set at 700 milliseconds. Fig. 9 shows the result with the same setup as before except that we use TimeoutChanger(T=700 ms,M=2) instead of NRTOChanger(N=3,M=2). TPC on the sender will change its path if the bytes are unacknowledged for more than 700 ms.

It measures this delay from the last call to a hook called at each RTT estimation. It is called upon receiving an ack for the window, at least once per RTT. The number of times depends on the delayed ack implementation on the receiver. If there was no byte in flight during the failure, this time will be measured from the first retransmission timeout. 700 ms is slightly above the time needed for two retransmissions, i.e., $206 + 2 \cdot 206 =$

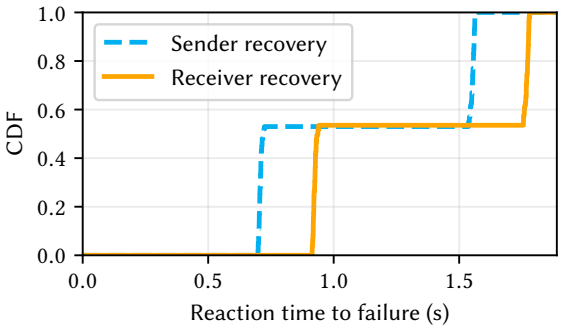

Fig. 9. CDF of the recovery time with one connection with TimeoutChanger(T=700 ms,M=2).

618 ms. The change is triggered after either 2 (for more than half of the connections) or 3 retransmissions. This depends on the state of the congestion window when the failure occurs. If the congestion window is almost full, the retransmission timeout occurs quickly and TimeoutChanger(T=700 ms,M=2) needs 2 retransmissions, otherwise, it needs 3 of them.

The receiver also waits for two retransmissions of the same acknowledgment before changing its path: hence the same delay difference of around 225 ms between the sender and receiver path changes.

Higher RTTs yield slower recovery. We use the NRTOChanger(N=3,M=2) and repeat the experiment by increasing the delays of all the links between routers. Increasing the RTT delay from 6 ms to 24 ms yields a recovery 400 ms slower. This is consistent between 24 ms and 84 ms: the recovery is 500 ms slower. This increase is explained by two factors: the RTO and the size of the congestion window. The RTO increases with the RTTs and this slows down the recovery. Moreover, a higher RTT increases the congestion window, raising the number of packets to send before it is filled and therefore, before starting the retransmission timer.

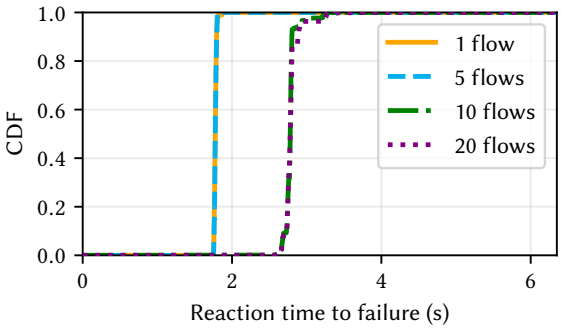

Fig. 10. CDF of the recovery time with increasing numbers of parallel flows with NRTOChanger(N=3,M=2).

TPC failure detection works in presence of multiple flows competing for the network bandwidth. We start, on the same path, 1, 5, 10 or 20 connections in parallel, each using 10 Mbps. The bandwidth of the links being 100 Mbps, starting 10 or 20 flows triggers congestion and the data transfer becomes network limited. This explains the gap observed in Fig. 10. With 10 or 20 flows, the window is larger because application-limited transfer do not increase their congestion window as much. A larger window means a longer time to send it on the wire. Other than that, the observed variance is slightly larger for 20 flows but the difference is not significant.

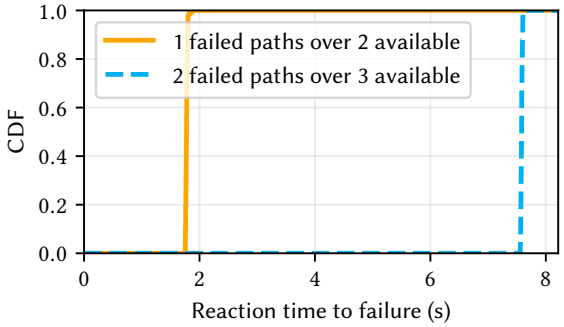

Fig. 11. CDF of the recovery time with different number of failed paths with NRTOChanger(N=3,M=2).

TPC failure detection works even if more than one path has failed. We add one alternative interdomain path to the topology shown in Fig. 3 and we introduce, after 10 seconds, a 100 % loss rate on two of the three paths. Even if this extreme scenario is unlikely to happen, it shows the resilience of TPC. Fig. 11 shows that the impact of two failing paths out of three is larger than doubling the recovery time. Indeed, both the sender and the receiver need to converge on the correct path.

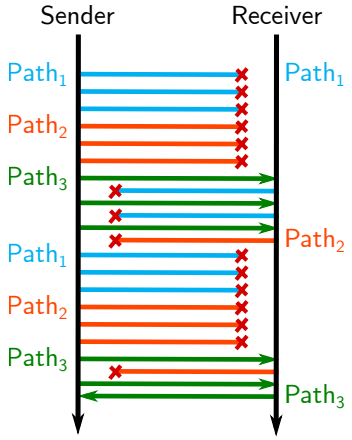

Fig. 12. Recovery of a TCP connection when two of three paths failed with NRTOChanger(N=3,M=2).

Fig. 12 explains this increase. One line represents a retransmission from the sender or a duplicate acknowledgment from the receiver. The color of a line represents the path it is sent on. Here, the first two paths (i.e., the red and blue ones) failed. Only the green one works. The sender needs to change its path 6 times to make the receiver move to the third path.

TPC failure detection converges faster on asymmetric failures. In the previous experiments, we considered that both endhosts had the same set of paths and that each path failed in both directions but this might not always be the case. Either

endhost might be on a valid path while the path used by the other endhost fails. To emulate asymmetric failures in our topology (see Fig. 3), we introduce `tc-netem` on only one side of a link. In case of sender path failure, the time required to converge is similar to the sender reaction time on symmetric failures (see Fig. 10). Indeed, upon symmetric failure, the sender is the first endhost to change its path. The receiver failure recovery in asymmetric design takes two retransmissions and is, thus, faster.

IV. Dynamically selecting lowest-delay paths

The previous section has shown that by leveraging eBPF a TCP connection can react to network failures by changing its path. In practice, eBPF programs can be executed at different places in the TCP stack [37]. In this section, we exploit this flexibility to enable TCP to autonomously find lowest-delay paths. Measurements have shown that different paths to the same destination can have different delay characteristics. Delays measured using `ping` vary even across equal-cost paths to a given destination [14]. Measurements performed within a large cloud provider network [11] show that the delays between servers located in different datacenters vary from one TCP connection to another. These two examples relate to the use of Equal Cost MultiPath (ECMP) for balancing packets across different paths on a per-flow basis. In Section II, we discussed the existence of multiple peering links between a cloud provider and many of its external peers. Measurements on such peering links [49], [50] reveal that some of them suffer from congestion. When the load on such a link increases,it first results in increased delays and then in packet losses (as link buffers become saturated). At any given time, different peering links will thus exhibit different delays, with the lowest delay path changing as the load on peering links changes.

Our aim is to accommodate the mapping of transport-layer connections to low-delay paths. Doing so entails contending with three main challenges: (i) gaining host visibility into the delays of different available paths, (ii) determining which path each connection should be mapped to at any point in time, and (iii) enabling hosts to realize their choices of paths. This last challenge is solved by the architecture presented in Section II. We next discuss the first two challenges, and how these are addressed by our schemes.

Selecting among paths via online learning. A simple approach for choosing among different paths is to pick the path exhibiting the best performance at that point in time. Such a greedy approach, however, has significant drawbacks. The performance of the chosen path might quickly deteriorate as additional connections are rerouted to it. For instance, a path that currently exhibits low latency because its bandwidth is not fully utilized, can become congested once additional connections are mapped to it, leading to unacceptably long latencies. This is even worse when decisions are made simultaneously and in an uncoordinated manner by different hosts, or for different connections from the same host. A too frequent rerouting of connections might lead to traffic instability.

We observe that ideas from online learning theory (and, more specifically, multi-armed bandit (MAB) theory [51]) are naturally applicable to this challenge. In the MAB setting, a decision maker (i.e. agent) repeatedly selects between different actions and observes, in hindsight, the implications of its chosen action on performance. In our context, the decision maker is the host and the actions correspond to different choices of paths to which a connection could be mapped. We show that using a classical MAB algorithm with provable guarantees can yield both high performance and stability. Here, we focus on passive measurements. Section IV-B details our solution. An alternative measurement strategy is the active probing of a path without moving the TCP connection on the path. This is not possible because the eBPF interface does not allow sending packets on every path at the same time and measure their smoothed RTT without disturbing the statistics (e.g., smoothed RTT) collected by TCP itself.

A. Path performance monitoring

To guide path selection, each host collects performance-related statistics for paths that carry its data traffic and stores these in a key-value cache. We next discuss our choices of the key of the cache, data to store, and data expiration time.

As in routing tables, data in the cache is aggregated by destination IP prefixes. To facilitate the aggregation of data by applications, the ports can also be included in the key or, alternatively, some applications can be assigned with unique identifiers, included in the key. The way data is aggregated has important implications; a very specific key will not be associated with sufficiently many connections to infer meaningful statistics while wasting memory, whereas a too broad key might encompass very different types of traffic, which would better be considered independently of each other.

The data itself consists of two elements: (i) the list of available paths (towards a certain destination IP prefix), and (ii) collected performance-related statistics for each available path. In SRv6, for instance, each path on the list will be represented as a sequence of segments, indicating the SRv6-capable routers the path traverses. Three natural path performance metrics to consider are throughput, packet loss ratio, and RTT. To assign more weight to recent, and so more informative, measurements than to older ones, while not being overly sensitive to variance in samples, we opt for keeping track of the exponential moving average of the monitored performance indicators (this approach is notably used in the Linux TCP stack to compute the connection's smoothed RTT [52]).

B. Online Learning Path Selection

We now explain our approach for mapping a new connection to an available path. Recall that a greedy scheme, while simple, can induce undesirable effects for performance and stability, and so we adopt a different approach: employing an online learning algorithm, namely, EXP3 [53].

EXP3, described in Algorithm 1, is designed for the MAB setting. An *agent* is repeatedly faced with a choice between different *actions*. At each discrete time step $t = 1, 2, 3 \ldots$ the agent selects an action a_t from a fixed set of actions $\mathcal{A}$ and then learns, afterwards, the implications of selecting a_t, captured by an observed *utility value* u_t. Importantly, the

agent only has visibility on the consequences of its chosen action a_t. Therefore, only its weight is updated. It cannot tell how *other* choices of actions would have impacted its derived utility value. The more rewarded the action is, the higher its weight increases, and the higher its selection probability increases. Another important feature of the MAB setting is that no assumptions whatsoever are made regarding how actions are associated with utility values. In fact, the utility values derived from selecting different actions can even be assumed to be chosen *adversarially*. Despite these limitations on the provided feedback and the unpredictability of the environment, algorithms like EXP3 provably provide meaningful benefits to the agent ("no regret") and convergence to equilibrium when multiple interacting agents employ EXP3 [53].

Algorithm 1 EXP3 Algorithm.

Given $\Gamma \in [0,1]$, set the weights: $w_a(1) = 1$ for all $a \in \mathcal{A}$
At each time step $t = 1, 2, 3 \ldots$,
  Set $p_a(t) = (1-\Gamma)\frac{w_a(t)}{\sum_{b\in\mathcal{A}} w_b(t)} + \frac{\Gamma}{|\mathcal{A}|}$ for each action a
  Draw the action a_t randomly according to the $p_a(t)$ distribution
  Observe reward u_t at the end of time step t
  Define the estimated reward $\hat{u}_t$ to be $\frac{u_t}{p_{a_t}(t)}$
  Set $w_{a_t}(t+1) = w_{a_t}(t) \cdot e^{\Gamma \hat{u}_t/|\mathcal{A}|}$
  Set $w_a(t+1) = w_a(t)$ for each action $a \neq a_t$

MAB algorithms are natural to apply to our context. A host can be modeled as an agent that, for each connection destined for the same IP prefix, selects between different paths to determine which path the connection should be mapped to. Selecting a path constitutes the action. The empirical implications of a certain choice of path are only observable to the host in hindsight, and the performance that would have resulted from a different choice of path for the connection is unknown to the host. In addition, since path performance is often affected by many dynamic factors that are external to the host (such as the existence and behavior of connections originating on other hosts whose paths intersect this path), avoiding assumptions regarding path performance (the utility from selecting the path) is prudent. Γ captures the extent to which alternative paths are probed. If Γ is set to 1, no exploration is made. If Γ is set to 0, EXP3 degenerates to a uniform probability distribution over paths.

To demonstrate the feasibility of our proposed scheme, we implement a path selection mechanism that monitors TCP connections that require low delays. We aggregate path performance by destination IP prefix and monitor the smoothed RTT. We use SRv6 to select paths and the EXP3 weights, with respect to the paths towards a specific destination IP prefix, are updated once a connection towards that prefix terminates.

Listing 4 shows the pseudocode of this path management. We implement the cache using two eBPF maps: one for the list of paths and one for the connections, as shown in Figure 7. Our TPC path manager initializes its connection structure and sets the path upon the end of the three-way handshake on server-side as the TPC presented in Section IV. We use the EXP3 algorithm to select the path at the `BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB` hook. Every RTT, using the `BPF_SOCK_OPS_RTT_CB` hook, we check if we transmitted more than a given hook activation rate. If we did, we reward the path and restart a new path choice. If the chosen path changes, we update the SRH. The hook activation rate is an amount of sent data after which TPC rewards its path. For short-lived connections, we reward the path once, at the end of a connection, using `BPF_SOCK_OPS_STATE_CB` before cleaning up their state.

Listing 4 Path management

```
int handle_sockop(struct bpf_sock_ops *skops) {
  conn = bpf_map_lookup_elem(&conn_map, &five_tuple);

  switch (skops->op) {
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
      // End of three-way handshake
      new_conn = new_connection_state();
      flow_info->srh_id = choose_srh_exp3();
      new_srh = get_srh(flow_info->srh_id);
      move_path(skops, new_srh);
      break;
    case BPF_SOCK_OPS_RTT_CB:
      // Every RTT
      if (skops->snd_nxt - conn->last_seq >= HOOK_RATE) {
        conn->last_seq = skops->snd_nxt;
        reward_path_exp3(flow_info->srh_id, skops->srtt,
                         flow_info);
        flow_info->srh_id = choose_srh_exp3();
        new_srh = get_srh(flow_info->srh_id);
        move_path(skops, new_srh);
      }
    case BPF_SOCK_OPS_STATE_CB:
      // End of a connection
      if (skops->args[1] == BPF_TCP_CLOSE) {
        reward_path_exp3(flow_info->srh_id, skops->srtt,
                         flow_info);
        bpf_map_delete_elem(conn_map, &five_tuple);
      }
      break;
    // [...]
  }
  return 0;
}
```

The overhead of Listing 4 might appear more consequent because it is called every RTT. However, only two conditions are usually checked. The hook activation rate controls the overhead of updating the paths's weights. Moreover, the overhead of eBPF programs running at every sent packet has been shown to be small [38]. At worst, they show an overhead of 10% for a 10-Gbps traffic.

The eBPF architecture [29] in the Linux kernel 5.3 only supports integer computations. However EXP3 and other machine learning techniques rely on floating numbers. To support them, we implement additional helpers in the Linux kernel. We define a floating point value as a structure with a mantissa and an exponent. We define a helper for every regular operator: addition, difference, product and division. As EXP3 needs to compute an exponential, we rely on the following Taylor series to compute it: $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$. The two last helpers that we need are to convert a float from and to two integers, one for its integer part and one for its decimal part. We implement these helpers with 400 lines of code in the Linux kernel.

Our implementation of EXP3 requires us to define the reward function. Providing high reward values makes EXP3 increases the probability of using a path. As we want to target the path with the lowest latency, its reward expression must be higher than the other ones. Knowing this, we define the reward as $SRTT_{max} - SRTT$. $SRTT_{max}$ is the highest observed SRTT value for the connection so that the reward is guaranteed to be positive. The pseudo-code of this reward computation is

Listing 5 Reward the path

```
void reward_path_exp3(int srh_id, int srtt,
          struct flow_info *flow_info) {

  flow_info->max_srtt = max(flow_info->max_srtt, srtt);
  observed_reward = flow_info->max_srtt - srtt;
  estimated_reward =
    observed_reward / flow_info->path_probability;

  // Update weights
  exp = exponent(e, GAMMA * estimated_reward
        / nbr_paths);
  weight = get_weight(srh_id) * exp;
  set_weight(weight, srh_id);
  normalize_weights();
}
```

detailed in Listing 5. For readability, we replace the floating point helpers by the regular arithmetic operator signs.

Listing 6 Weights's normalization

```
#define NBR_TOKENS 10000

void normalize_weights() {
  for (i = 0; i < weights_len; i++) {
    weight = get_weight(i) * NBR_TOKENS / sum_weights();
    if (weight < 1)
      weight = 1;
    set_weight(weight, i);
  }
}
```

By default, the EXP3 algorithm is designed to reach equilibrium. However, in networks, various events can disturb it. If we let EXP3's weights exponentially increase in stable times, they will quickly reach the maximum floating point number that we can represent. Moreover, the increase of the best path's weight will deepen the difference with the other weights even if it does not actually change much in the probabilities of using the path. If the best path becomes undesirable, it will take much time to reduce this difference and start using the other paths. For instance, TPC identifies a path as the optimum with a probability of $> 99.99\%$ but its RTT changes. If its weight is 99.99 (and the other path's one is 0.01), it will only take a few iterations/measurements to increase the other path's weight enough for it to be used. If the current path's weight was instead 2^{64}, it would have required many more iterations to let the other path being used. Both weight distributions will yield similar probabilities during stable phases. Therefore, we normalize paths's weights to optimize TPC's reaction time. We chose to limit it to $[1, 10000]$ so that it is at most possible to reach 99.99% of connections on the best path (if there are two alternative paths). On servers with millions of connections, it is interesting to increase the upper limit in order to go above 99.99% of probability during stable phases. Listing 6 shows how this is implemented. We normalize the paths's weights after rewarding any path, at the end of a connection.

This path selection mechanism could be combined to the remote failure detection ones (Section III). We could modify failure detection algorithms to mark paths that lost connectivity and filter out failed paths when targeting low latency.

C. Evaluation

We emulate networks with Mininet [48] on a Linux kernel 5.3 running on a virtual server emulated by QEMU, equipped with 20 virtual CPUs and 16 GB of RAM. To emulate links delays, we use `tc-netem`. The bandwidth is emulated with `tc-htb`. Their other parameters are the default ones.

We evaluate TPC for the interdomain use case shown in Fig. 4. A CDN server is queried by endhosts in distant ASes. The AS of the server provides two paths to TPC. Both paths have different latencies that will change over time. We use our TPC in the server to find the lowest RTT path. The client does not run TPC and thus, only the server can change its path.

This network uses links with the same bandwidth (100 Mbps). The lower interdomain path has a RTT of 10 ms while the upper path has this RTT multiplied by a given factor (10 by default). The client always uses the lower path.

Our evaluation represents a set of clients that retrieve small files from a web server. The server includes our TPC with EXP3. The clients use Apache Benchmark to continuously download a 100 kB file from a `lighttpd` server during 50 seconds. We configured Apache Benchmark to use 4 parallel connections. Apache Benchmark creates one connection for each HTTP request and TPC is executed when the connection closes. After 10 seconds, we invert the delays of both paths in the direction from the server to the client. We use `tc-netem` to change the link delays. Then, we observe both the completion time of the requests and the time needed for EXP3 to converge to the working path at the initialization and after the inversion of the path delays. We consider that EXP3 has converged once the lowest-delay path carries over 90% of the weights' sum. We use a uniformly random selection of the paths as a baseline to compare the benefits of EXP3 in TPC. This baseline is implemented through static routes adding one of the SRH that are load balanced through ECMP. The default initial congestion window in Linux is 10 packets.

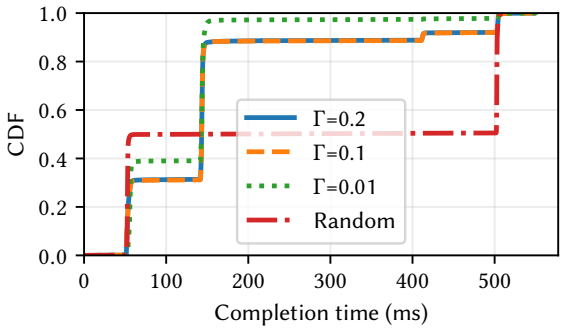

Fig. 13. Completion times of HTTP requests. TPC outperforms a purely random selection and a low value of Γ provides better results.

TPC uses the quickest path as opposed to a uniformly random solution. Fig. 13 shows the CDF of the completion time of each request with TPC either using a uniformly random choice or EXP3 with various Γ values. Without learning from experienced delays, the uniformly random heuristics cannot learn anything from the network and therefore half of the connections use the longest delay path. In contrast, TPC using EXP3 with $\Gamma \neq 1$ learns and quickly adapts to delay changes.

We observe roughly 4 different completion times in each TPC curve and 2 for a uniformly random heuristic. The

fastest and longest of these four completion times relate to connections whose traffic uses the lowest RTT path and the highest RTT path, respectively. The two completion times in between pertain to connections whose SYN+ACK is not sent along the path used to carry the data. Recall that since we do not control the path for the SYN+ACK packets in Linux, this path is always the lower (and default) path whose delay changes during the experiment. This accounts for the distance of the second completion time from the first, and of the third one from the fourth. In the uniformly random heuristics, all of the traffic (including SYN+ACK packet) uses the same path.

When the situation is stable, the lower the Γ, the better. Fig. 13 shows that the Γ value impacts the rate of connections using the quickest path. As described in Algorithm 1, every path has at least a probability of $\frac{\Gamma}{|Paths|}$ to be chosen. Intuitively, Γ is the percentage of uniformly random choices that TPC makes. In our example with two paths, at most $(1-\Gamma)+\frac{\Gamma}{2} = 1-\frac{\Gamma}{2}$ of the traffic can be routed on the right path if Γ is constant and the traffic is stable. With Γ set to 0.2, TPC cannot send more than $1-\frac{0.2}{2} = 90\,\%$ of the traffic on the best path. There are only two convergences in this experiment and therefore, probing too much for only one change hurts more connections than converging slower. For instance, setting Γ to 0.01 is a better choice in the big picture while it converges 6 times slower than setting Γ to 0.1. A real deployment of the TPC would help to refine the choice of the Γ over time. This is left as future work.

A higher Γ enables a faster convergence. Indeed, with a high Γ, we often measure the path delay changes because this parameter gives the rate of connections that are routed randomly. With Γ set to 0.1, $90\,\%$ of the convergence times is lower than $1.25\,\mathrm{s}$. If we set Γ to a tenth of that, $0.01\,\%$, the convergence time increases to $5\,\mathrm{s}$. The influence of Γ is similar because setting it to 0.1 is still better than to 0.2 or 0.01. Therefore, with unstable paths, when we expect frequent delay changes, Γ should be higher.

A larger difference in path delays only slightly speeds up convergence. We change the delay factor between the quickest and the slowest paths. We use two, five and ten times the quickest path RTT of $10\,\mathrm{ms}$. Increasing the delay factor beyond five times does not change the convergence time. Even with a delay factor of two, the increase of delay did not exceed $100\,\mathrm{ms}$ in $95\,\%$ of the runs. Indeed, the reward is more important in Listing 5 when the maximum smoothed RTT is higher. TPC works better in heterogeneous networks and this is in these networks that TPC's purpose is the most useful.

The higher the hook activation rate is, the quicker TPC converges. This hook activation rate is influenced by the size of the transfer since we use a hook at the end of the TCP connection. Logically, more information about the state of the paths yields faster convergence. With a hook activation rate of $1\,\mathrm{kB}$, 90 of the 100 runs are below $0.75\,\mathrm{s}$ while with a rate of $1\,\mathrm{MB}$, the 90^{th} percentile is $2\,\mathrm{s}$.

If we do not change the hook activation rate of each connection but we increase the number of simultaneous connections, paths will be more rewarded. When TPC runs on 2 parallel connections instead of 4, paths are rewarded twice less. Therefore, we observe a decrease of the convergence time of $0.5\,\mathrm{s}$. Leaving one connection adds $1\,\mathrm{s}$ of delay on average.

V. Related work

CPR [54], Blink [45] and INFLEX [55] are the closest related work to our first use case, as they support recovery from remote failures. CPR can be installed in Edge Networks of Content Distribution Networks. It monitors connection stalls in a similar way to TPC. When a failure or stall is detected, they use a combination of `fwmark` bits, MPLS, and ECMP to reroute the affected flows. INFLEX [55] and Blink [45] are network-level solutions. Both require programmable switches to monitor connections and reroute *all* the traffic if they notice many connections experiencing a stall. They will fail to notice remote failures affecting a few connections but, when they affect many connection, they will have more data than TPC to decide to change paths.

The closest related work to our second use case are Replex [56], CONGA [57] and Clove [58]. Replex [56] reroutes flows withing hundreds of milliseconds to meet traffic engineering objectives using ideas from game theory. Unlike TPC, Replex is primarily focused on improving bandwidth utilization. CONGA [57] is a distributed load balancer for datacenters intended to minimize flow completion times, and so addresses a different problem than ours. CONGA is installed on leaf switches, measures the congestion for each flow, and realizes load balancing decision using VXLAN when forwarding the first packet of a flow. Clove [58] tackles the same problem as CONGA and is implemented in the hypervisor of the end-host.

TPC, in contrast to the related work, is a transport-based solution that utilizes standardized IPv6 Segment Routing. In addition, TPC leverages eBPF programs, enabling the tuning of its behavior to the requirements of different applications (not only with respect to failure recovery but also with respect to route selection).

VI. Conclusion and Discussion

TCP was designed to depend on the network layer for selecting the path used to reach any given destination. This separation of functions between the transport and network layer implies that TCP can only react in two ways to network-level events: the retransmission of lost packets and the adaptation of its transmission rate.

We have proposed the TCP Path Changer (TPC). TPC makes TCP more agile by enabling the TCP stack to change the current network-layer path in reaction to different types of events. Thanks to eBPF, each application can inject its own TPC into the underlying Linux TCP/IP stack. To illustrate the benefits of TPC, we have applied it to two very different use-cases. First, we have shown that TPC can detect distant link failures and react quickly by rerouting the affected connections, and also monitor the health of a connection and reroute it when needed. Second, we demonstrated how servers can use our TPC to find small delay paths.

Our plans for future research include applying TPC to other use-cases (e.g., rerouting connections to better utilize network bandwidth) and to collect measurements from production traffic.

SOFTWARE ARTIFACTS

To encourage system administrators and network researchers to build upon our eBPF-based TPC, we release our eBPF programs at https://github.com/jadinm/tpc-ebpf and modifications to the Linux kernel at https://github.com/jadinm/tpc-kernel. TPC is composed of 1100 lines of eBPF code and 470 lines of kernel patch. We also release scripts to run our experiment environment at https://github.com/jadinm/tpc.

REFERENCES

[1] M. Trevisan, D. Giordano, I. Drago, M. M. Munafò, and M. Mellia, "Five years at the edge: Watching internet from the ISP network," *IEEE/ACM ToN*, vol. 28, no. 2, pp. 561–574, 2020.

[2] J. Postel, "Transmission Control Protocol," RFC 793, Sep. 1981.

[3] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM CCR*, vol. 18, no. 4, pp. 314–329, 1988.

[4] R. Al-Saadi, G. Armitage, J. But, and P. Branch, "A survey of delay-based and hybrid TCP congestion control algorithms," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3609–3638, 2019.

[5] D. A. Borman, R. T. Braden, and V. Jacobson, "TCP Extensions for High Performance," RFC 1323, May 1992.

[6] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018, Oct. 1996.

[7] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," RFC 2992, Nov. 2000.

[8] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha, "The RACK-TLP Loss Detection Algorithm for TCP," RFC 8985, Feb. 2021.

[9] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot, "Characterization of failures in an operational ip backbone network," *IEEE/ACM ToN*, vol. 16, no. 4, 2008.

[10] D. Watson, F. Jahanian, and C. Labovitz, "Experiences with monitoring ospf on a regional service provider network," in *IEEE ICDCS*, 2003.

[11] W. Reda, K. Bogdanov, A. Milolidakis, H. Ghasemirahni, M. Chiesa, G. Q. Maguire Jr, and D. Kostić, "Path persistence in the cloud: A study of the effects of inter-region traffic engineering in a large cloud provider's network," *ACM SIGCOMM CCR*, vol. 50, no. 2, 2020.

[12] V. Paxson, "End-to-end routing behavior in the internet," *IEEE/ACM ToN*, vol. 5, no. 5, pp. 601–615, 1997.

[13] B. Augustin, T. Friedman, and R. Teixeira, "Measuring multipath routing in the internet," *IEEE/ACM ToN*, vol. 19, no. 3, pp. 830–840, 2010.

[14] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush, "From paris to tokyo: On the suitability of ping to measure latency," in *ACM IMC*, 2013.

[15] G. Detal, C. Paasch, S. Van Der Linden, P. Merindol, G. Avoine, and O. Bonaventure, "Revisiting flow-based load balancing: Stateless path selection in data center networks," *Computer Networks*, vol. 57, no. 5, pp. 1204–1216, 2013.

[16] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks," in *ACM CoNEXT*, 2014.

[17] A. Ford, C. Raiciu, M. J. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824, Jan. 2013.

[18] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," *ACM SIGCOMM CCR*, vol. 41, no. 4, 2011.

[19] J. Postel, "Internet Protocol," RFC 791, Sep. 1981.

[20] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, Dec. 1998.

[21] S. M. Bellovin, "Security problems in the tcp/ip protocol suite," *ACM SIGCOMM CCR*, vol. 19, no. 2, pp. 32–48, 1989.

[22] G. Neville-Neil, P. Savola, and J. Abley, "Deprecation of Type 0 Routing Headers in IPv6," RFC 5095, Dec. 2007.

[23] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The segment routing architecture," in *IEEE GLOBECOM*, 2015.

[24] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing Architecture," RFC 8402, Jul. 2018.

[25] R. Hartert *et al.*, "Solving segment routing problems with hybrid constraint programming techniques," in *CP*. Springer, 2015.

[26] F. Aubry, D. Lebrun, Y. Deville, and O. Bonaventure, "Traffic duplication through segmentable disjoint paths," in *IFIP Networking*, 2015.

[27] T. Schüller, N. Aschenbruck, M. Chimani, M. Horneffer, and S. Schnitter, "Traffic engineering using segment routing and considering requirements of a carrier ip network," *IEEE/ACM ToN*, vol. 26, no. 4, pp. 1851–1864, 2018.

[28] P. L. Ventre, S. Salsano, M. Polverini, A. Cianfrani, A. Abdelsalam, C. Filsfils, P. Camarillo, and F. Clad, "Segment routing: a comprehensive survey of research activities, standardization efforts, and implementation results," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 1, pp. 182–221, 2020.

[29] The Linux Foundation, "eBPF," https://ebpf.io, October 2021.

[30] D. Thaler and P. Gaddehosur, "Making ebpf work on windows," https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/, May 2021.

[31] Y. Hayakawa, "ebpf implementation for freebsd," in *BSDCan*, 2018.

[32] S. Sarwood, "OVH data centre destroyed by fire in strasbourg – all services unavailable," the Register, March 10, 2021. [Online]. Available: https://www.theregister.com/2021/03/10/ovh_strasbourg_fire/

[33] D. Walton, A. Retana, E. Chen, and J. Scudder, "Advertisement of Multiple Paths in BGP," RFC 7911, Jul. 2016.

[34] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The design and implementation of the 4.4 BSD operating system*. Addison-Wesley Reading, MA, 1996, vol. 2.

[35] K. R. Fall and W. R. Stevens, *TCP/IP illustrated, volume 1: The protocols*. Addison-Wesley, 2011.

[36] B. Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[37] L. Brakmo, "Tcp-bpf: Programmatically tuning tcp behavior through bpf," in *NetDev 2.2*, 2017.

[38] V.-H. Tran and O. Bonaventure, "Beyond socket options: Towards fully extensible linux transport stacks," *Computer Communications*, vol. 162, pp. 118–138, 2020.

[39] D. Lebrun and O. Bonaventure, "Implementing ipv6 segment routing in the linux kernel," in *ACM ANRW*, 2017.

[40] S. Mirtorabi, L. Nguyen, A. Roy, P. Psenak, and P. Pillay-Esnault, "Multi-Topology (MT) Routing in OSPF," RFC 4915, Jun. 2007.

[41] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *ACM SIGCOMM*, 2011.

[42] M. Chiesa, G. Rétvári, and M. Schapira, "Lying your way to better traffic engineering," in *ACM CoNEXT*, 2016.

[43] M. Chiesa, A. Kamisiński, J. Rak, G. Rétvári, and S. Schmid, "A survey of fast-recovery mechanisms in packet-switched networks," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1253–1301, 2021.

[44] L. Wang, M. Saranu, J. M. Gottlieb, and D. Pei, "Understanding bgp session failures in a large isp," in *IEEE INFOCOM 2007*. IEEE, 2007.

[45] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *USENIX NSDI*, 2019.

[46] M. Sargent, J. Chu, D. V. Paxson, and M. Allman, "Computing TCP's Retransmission Timer," RFC 6298, Jun. 2011.

[47] W. Eddy, "TCP SYN Flooding Attacks and Common Mitigations," RFC 4987, Aug. 2007.

[48] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *ACM CoNEXT*, 2012.

[49] A. Dhamdhere, D. D. Clark, A. Gamero-Garrido, M. Luckie, R. K. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. C. Snoeren, and K. Claffy, "Inferring persistent interdomain congestion," in *ACM SIGCOMM*, 2018.

[50] R. Fanou, F. Valera, and A. Dhamdhere, "Investigating the causes of congestion on the african ixp substrate," in *ACM IMC*, 2017.

[51] S. S. Villar, J. Bowden, and J. Wason, "Multi-armed bandit models for the optimal design of clinical trials: benefits and challenges," *Statistical science: a review journal of the Institute of Mathematical Statistics*, vol. 30, no. 2, p. 199, 2015.

[52] D. V. Paxson and M. Allman, "Computing TCP's Retransmission Timer," RFC 2988, Nov. 2000.

[53] S. Bubeck and N. Cesa-Bianchi, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *arXiv preprint arXiv:1204.5721*, 2012.

[54] R. Landa, L. Saino, L. Buytenhek, and J. T. Araújo, "Staying alive: Connection path reselection at the edge." in *USENIX NSDI*, 2021.

[55] J. T. Araújo, R. Landa, R. G. Clegg, and G. Pavlou, "Software-defined network support for transport resilience," in *IEEE NOMS*, 2014.

[56] S. Fischer, N. Kammenhuber, and A. Feldmann, "Replex: dynamic traffic engineering based on wardrop routing policies," in *ACM CoNEXT*, 2006.

[57] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM*, 2014.

[58] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford, "Clove: Congestion-aware load balancing at the virtual edge," in *ACM CoNEXT*, 2017.