A good title

Your N. Here
Your Institution

Second Name
Second Institution

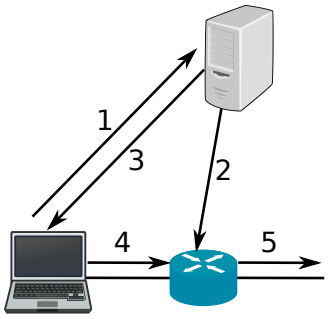


Figure 1: Endhost path query to the controller.

Abstract

Abstract

1 Introduction

A paragraph of text goes here.

2 Architecture

2.1 Path injections

This paper proposes to integrate TCP with SRv6 to optimise congestion. We reuse a similar controller as in Software Resolved Networks [7]. As shown in Figure 1, an application on the endhost can query (1) the controller for a path to reach a particular destination. The SRN query is implemented by extending DNS requests. The controller computes a path, build its SRH and assign it a binding segment, that we call Path ID. After compuation, the mapping between the binding segment and the SRH is inserted in the access router (2). Then, the controller sends the Path-ID inside the DNS reply back to the endhost (3). The application can then build an SRH containing only the Path-ID and insert it in all its packets (4). The access router reads the Path-ID and encapsulate the packet in an outer IPv6 packet with an SRH that steer the packet through the actual path.

While this first prototype shows that we can build a SDN solution with SRv6, this prototype has two issues. First, it requires to modify applications to talk to a controller. This raises deployment issues if you want to control the path taken by all your applications. Second, it did not consider reaction to congestion, only to failures.

In this paper, we offload the choice of the path to the endhost. The TCP stack of endhosts have access to the congestion state of each connection. This state, along with information from the controller enables the endhost to choose between paths that it received from the controller.

We extend the solution by enabling a daemon, the Path Daemon**TODO: name ?**, in endhosts to request for a set of dijsoint paths. The path are disjoint so that we know that a congestion occuring on a path does not influence the others. The endhost can remain agnostic of the actual topology of the network and it will not waste time hitting paths that suffer from congestion on the same link.

We use **TODO: insert algo name ???** from [1] to find these disjoint paths. This algorithm is specific to SRv6 because it compute disjoint paths between two endpoints and checks that these paths can be expressed in fewer segments than a given limit. Some hardware components in their network do not support arbitrarily long SRHs as documented in [**?**].

The controller computes a set of disjoint paths for every set of pair of source prefix/destination prefix. Each endhost daemon receives a list of Path IDs for pairs matching its access router as source. Along with the Path ID, the controller sends the bandwidth capacity, its current usage and its delay. We scales up the number of controllers with the number of endhost **TODO: inaccurate, it is actually the database but it might be simpler to explain that**.

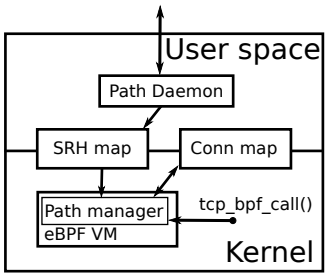


Figure 2: Path management code talks through eBPF maps to the Path Daemon.

2.2 Application-independent path management

The previous section describes the path injection to a daemon that runs on the endhost. However this endhost needs to enable other applications to manage these paths without actually modifying them. A solution could be to modify the kernel directly to include this path management. However, this would require operators to modify the kernel of all the endhosts. An alternative is to inject code in the kernel thanks to eBPF hooks in the kernel. Most of these hooks exists since Linux v.**TODO: version ???**. **TODO: Quid eBPF windows?**

In [7], applications had to be modified to inject their Path IDs in their sockets so that they are inserted in the packets. This limits the deployment of the solution. We need a way to inject the Path ID that is application-indenpendent. Another different with [7] is that we want to react from congestion events inside the endhost. We want to inject a path management behavior inside TCP which is quick to react to congestion in the network. Modifying the kernel directly would require operators to update the kernel of all their machines. Therefore, relying on kernel modification is not the best solution.

Since Linux **TODO: version ???**, programmer can inject code to the kernel thanks to the eBPF VM [**?**]. We can inject code at various locations, called hooks, each calling a `tcp_bpf_call()` function. We use four of the hooks in our prototype. `BPF_SOCK_OPS_TCP_CONNECT_CB` is call before sending the SYN packet. This means we can still choose the SRH of the SYN packet in our path management code. We also use `BPF_SOCK_OPS_STATE_CB` that is called each time the state of TCP changes. We use it to know when the connection is closed to clean the per-connection state that we maintain. `BPF_SOCK_OPS_RTO_CB` is called after the expiration of the retransmission timer which is an important congestion signal. `BPF_SOCK_OPS_ECN_CE` is called before sending the segment setting CWR bit in response to packets with ECN bit set. The hook is the only one that we add to the kernel. We choose to place it when sending the CWR instead of the reception of the actual ECN because the CWR bit is sent only once by RTT. This is better done there than in the path management code.

Figure 2 shows the interactions of the different components inside the endhost. The Path manager code runs in an the isolated eBPF VM inside the kernel space. A verifier [**?**], packaged with the kernel, checks memory calls and program termination of the code before injection it so that we are guaranteed that the code does not make the kernel crash **TODO: Comments on other verifications ?**. For injected code in the TCP stack, the injected code receives a structure containing the code of the hook that triggered the code, the five tuple, and some variables of the TCP connection, such as the minimum RTT or the congestion window. Moreover, it can read and writes to memory chunks, called eBPF maps, that can be written to or read by a user space application. In this architecture, our daemon will fill a first eBPF map, the SRH map with the Path IDs received from the controller. This map is a hashmap that maps the destination prefix to a list of a structures containing a Path ID, the path bandwidth, its latency,... (i.e., all the pieces of information transmitted by the controller). **TODO: Discuss how the eBPF map can support multiple size of prefixes?** Since eBPF injected code does not have global variable or heap, we use a second eBPF map also oragnized as hashmap, the connection map. The key is the five tuple and it maps to a structure containing data about the connection.

TODO: Explain cgroups?

```
1  int handle_sockop(struct bpf_sock_ops *skops)
2  {
3      struct five_tuple five_tuple;
4      get_five_tuple(&five_tuple, skops);
5
6      struct conn *conn = bpf_map_lookup_elem(&
       conn_map, &five_tuple);
7      if (!conn) { // New connection
8          struct connection new_conn;
9          memset(new_conn, 0, sizeof(new_conn));
10         new_conn.srh = get_best_path(srh_map, &
       five_tuple.remote);
11         bpf_setsockopt(skops, SOL_IPV6, IPV6_RTHDR
       , new_conn.srh, sizeof(*new_conn.srh));
12         init_exponential_backoff(&new_conn);
13         bpf_map_update_elem(conn_map, &five_tuple,
        &new_conn, BPF_ANY);
14         *conn = new_conn;
15     }
16
17     switch (skops->op) {
18         case BPF_SOCK_OPS_STATE_CB:
19             if (skops->args[1] == BPF_TCP_CLOSE)
20                 bpf_map_delete_elem(conn_map, &
       five_tuple);
21             break;
22         case BPF_SOCK_OPS_ECN_CE:
23             // TCP segment with CWR bit set
24             react_to_congestion(five_tuple, conn);
25             break;
26         case BPF_SOCK_OPS_RTO_CB:
27             // Retransmission timer expiration
28             react_to_congestion(five_tuple, conn);
29             break;
30     }
31     socks->reply = rv;
```

```
32      return 0;
33  }
```

Listing 1: Path management

Listing 1 shows the pseudo code of the path management. When an application starts a connection to a server, the eBPF hook ??? is triggered. We start by retreiving the five tuple and check if an entry for this connection exists. Since it is the new connection, it won't find it. Therefore, we create this entry, choose the best path for the destination and change set the SRH with the Path ID in the socket so that it is inserted in every packet sent. After that, we identify the hook that triggered the call. When the TCP state changes, and when the connection is closed, we clean the connection state of the eBPF map. When we receive a congestion signal, either triggered by ECN or by the retransmission timer expiration, we call another routine to choose the new path.

2.3 Stable path management with EXP3

The choice of changing the path is left to the endhost instead of relying on the controller. This enables quicker reactions to congestion and better scalability of the network core. However letting endhost arbitrarly choose their own paths might lead to unstable solutions. Endhosts might decide to change the paths all at the same time and they could all choose the same path.

Listing 2 shows the pseudocode of this algorithm. We first check that we waited a sufficient amount of time before moving the connection. Indeed, we implement an exponential backoff to prevent stability issues. Without this, all connections in the network would switch paths at the same time and this might cause congestion in another part of the network. Then, they would all switch back to previous situation and continue flapping between both paths.

To choose the good paths, we use an algorithm of Adversarial Bandit Problem: EXP3 [**?**]. **TODO: Introduce the Bandit Problem** We associate a weight to each path. When we consider changing the path, we evaluate the current path quality and we update its weight depending of this metric. A large increase of the weight means a good quality while a small increase indicates the opposite. Then, we make a weighted random choice to select the new path. Note that this "new" path could be the old one. In this case, we do not change the path.

There are three design decisions to make to use this algorithm: the initial path weights, the reward computation and the value of `GAMMA`. We set the initial weight to the maximum amount of bandwidth available on the path. The controller gives this information with the SRH. The reward is computed based on the last congestion window observed. **TODO: Change to mean/median cwin** The value of `GAMMA` can change the emphasis on the path weights. If we set this value to 1, each path has the same likelihood to be selected. If we set this value to 0, paths are selected stricly based on their weight. **TODO: Explain why we don't want value 0?**

```
1  void react_to_congestion() {
2      // Wait exponential backoff
3      if (current_time - conn->last_move_time < conn
       ->wait_before_move)
4          return;
5      // We compute the reward and update EXP3
       weight
6      theReward = reward(choice, t);
7      estimatedReward = 1.0 * theReward /
       probabilityDistribution[current_path];
8      weights[current_path] *= math.exp(
       estimatedReward * gamma / numActions); #
       important that we use estimated reward here!
9      // Select the path with EXP3
10     probabilityDistribution = distr(weights, gamma
       );
11     best_srh = draw(probabilityDistribution);
12     // Set the SRH
13     bpf_setsockopt(skops, SOL_IPV6, IPV6_RTHDR,
       best_srh, sizeof(*best_srh));
14 }
15
16 void distr(weights, gamma) {
17     return [(1.0 - GAMMA) * (w / sum(weights)) + (
       GAMMA / len(weights)) for w in weights]
18 }
```

Listing 2: Reaction to congestion

3 Evaluation

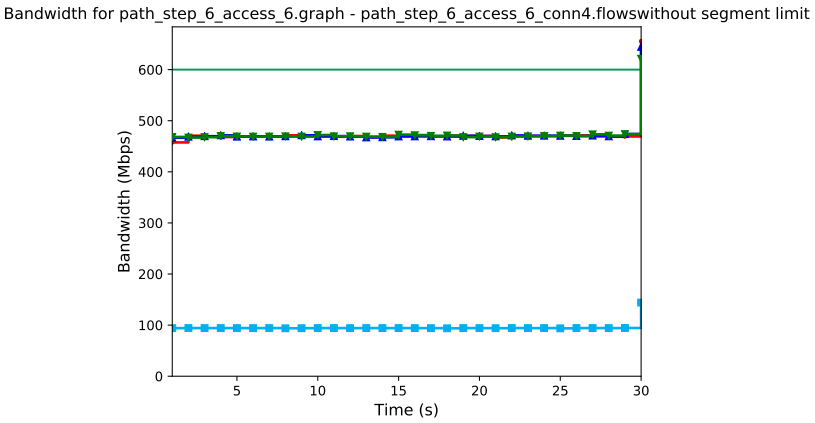
3.1 Simple example

We use the topology shown in Figure **??**. This simple network only has links with the same bandwidth (100 Mbps) and latency (1ms). We prevent ECMP from working by setting arbitrary IGP weights on the links.**TODO: Test with ECMP enabled** Each endhost starts 4 connections using iperf3 with the server in front of it the network. We emulate the network with Mininet [**?**] on a Linux kernel 5.3 running on machine with 20 CPU and 16 GB of RAM.

Figure 3a shows the network bandwidth used by all the connections through time. In theory, this badnwidth could go as high as 600 Mbps according to the Maximum Flow approximation [**?**] since there are six disjoint paths available. In practice, the total bandwidth is slightly below that number but since this number is above 500Mbps, we can deduce that all of the six paths are used. Regular TCP connections, without ECMP, cannot achive more than 100Mbps because they can only use one path. **TODO: Test with ECMP enabled** We do not see significant changes with different values of `GAMMA`. **TODO: Discuss with Schapira**

3.2 Topology Zoo set

We used 233 topologies from the Topology Zoo [6]. We modified these topologies to remove part of the graphs that



(a) Sum of the used network bandwidth

Figure 3: Simple example with 6 disjoint paths and 4 clients communicating with 4 servers

lack link redundancy,... trees. The resulting topologies have only routers with at least two links. Communication between routers in a tree can only use a single path. Such trees cannot see any improvement from using Segment Routing. Moreover, such trees are likely artifacts of the topology collection because they cannot preserve connectivity even with a single link failure. Figure 4a and 4b shows the CDF of the number of routers and links in the resulting topologies. Figure 4c shows the distributions of routers percentage with a given number of attached links.

The set of topologies does not include which routers are actually linked to endhosts and which are core routers. We assume that routers with two links are edge routers and the others are core routers because they are more connected.

3.3 Maximum improvement

This sections evaluation the theoritical improvement of using multipath transport protocols and Segment Routing on the endhosts instead of a single path protocol with only shortest paths.

We used a linear programming formulation of the Max Flow problem. Given a set of paths linking a set of the edge routers, we compute the maximum amount of traffic that can go through the topology without exceeding the link capacity.

$$maxflow(\mathcal{P}) \quad \max_{f} \quad \sum_{p \in \mathcal{P}} f_p$$

$$\text{s.t.} \quad \sum_{p \in \mathcal{P}} r(l,p) \cdot f_p \leq c(l) \ \forall l \in \mathcal{L}$$

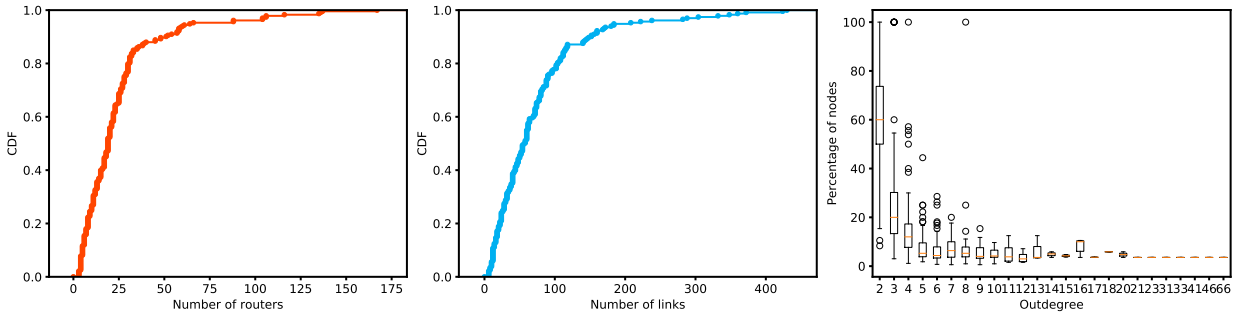
$\mathcal{P}$ represents the set of `sr-paths`, encoded as a list of intermediate routers. As for IPv6 SR, The real path taken between each intermediate router is the shortest one. $r(l,p)$ represents the ratio of flow that goes through a particular link. This ratio is used to model the ECMP splitting of the traffic.

To model a single-path transport protocol without SR we insert all the shortest sr-paths between each pair of edge router to $\mathcal{P}$. We assume that we have a high number of connections for each path and therefore that hash-based ECMP splits the traffic evenly. [2] shows that it is not the case in practise.

Multipath transport protocols with SR also uses shortest path routing. However, they can establish a big number of subflows for each connection and hope that they will cover all or a portion of the shortest paths. This hijacks ECMP that sees subflows as indepedent connections. Flow Bender [5] and SEMPER [4] used this solution to leverage some of the path diversity of a topology. We will assume for simplicity that this method works perfectly and that multipath protocols can always cover all the shortest paths. We insert a sr-path for each shortest path between each pair of edge routers instead of giving one sr-path representing all the shortest paths between a given pair.

Introducing Segment Routing means adding more paths to $\mathcal{P}$. Note that generating the set of all possible paths is in $O(|\mathcal{R}|^k)$ with $\mathcal{R}$ being the set of routers and k the maximum number of intermediate nodes. This is not practicle for big topologies. Therefore, we reuse the approach of [**?**] that iteratively adds relevant paths to the set until no interesting paths can be added to $\mathcal{P}$. When it reaches this point, computing the model on this restricted set or on the complete one gives the same solution. **TODO: Explain how the model is different from infocom, i.e., demands become unbounded or is it too detailed ?**

Adding IPv6 SR can significantly improve the maximum throughput. Figure 5a shows the relative maximum flow improvement of using single path or multiple path with SR. There is a boxplot for a given percentage of edge router pairs that have heavy hitters to route **TODO: Give the rational behind the choice of the values**. The number of heavy hitters has influence on the relative improvement. It is easier to unused alternative paths with fewer heavy hitters. With 1% of heavy hitter pairs, for half of the topologies, we can



(a) Number of routers (b) Number of links (c) Out degree of the routers

Figure 4: Features of the evaluation topologies

improve the maximum flow by 71%. For 10% of heavy hitter pairs, we can still improve it by 30%. Figure 5b shows the improvement with multiple path protocol without SR. The improvement is about 5% lower. This justifies the use of SR even with multipath protocols.

3.4 Largest set of disjoint paths

Finding the largest set of disjoint paths between two routers can be done with Edmonds–Karp algorithm [3] in $O(|\mathcal{R}||\mathcal{L}|^2)$. Even faster strategies can imply applying Dijkstra algorithm iteratively. Each time that the algorithm finds a path, we remove its links for the next computations. We stop it when it cannot find another path. This is solved in $O((|\mathcal{L}|+|\mathcal{R}|)n\log|\mathcal{R}|)$ with n being the number of iterations of dijkstra algorithm. $O(n)$ cannot be bigger than $O(|\mathcal{L}|)$ because at each iteration, if a path is found, the links if this path will be removed. In practise, source and destinations are behind routers with exactly two outgoing links. There will be at most two disjoint paths. Therefore, applying the Dijkstra algorithm iteratively is faster than using the Edmonds-Karp.

Figure 6a shows a CDF of the mean number of disjoint paths by pair of access router in each topology. We consider routers with exectly two links as access routers. This means that there are at least one path between two routers but at most two since there are only two outgoing links. Edmonds-Karp computes the largest set of disjoint paths that can be reached. We can reach the same sets for 70% of topologies by applying iteratively Dijkstra algorithm.

SRv6 is limited in the number of segments that it can apply in a packet. This limit can be due to hardware limitationsi of IPv6 SR implementations [8]. So we need to use algorithms that prevent the number of segments to grow beyond a given limit. Always finding the largest set of disjoint paths within the limit of segment is exponential and therefore not feasible in practise. [1] defines the SPS**TODO: Check with Francois for the name** algorithm that computes the shortest path between two routers under a given limit of segments. Each time that the algorithm finds a path, they remove all the links for the next computations. They stop when no more path can be found. Figure 6b shows that the number of disjoint paths increases with the number of authorized segments. With only one segment, we can only have one path because only the destination can be encoded in the segment list. Increasing to 4 segments, (i.e., 3 intermediate routers), signigicantly improves the number of disjoint paths. We see that increasing the limit beyond this does not improve significantly the number of disjoint paths.

TODO: Conclusion

References

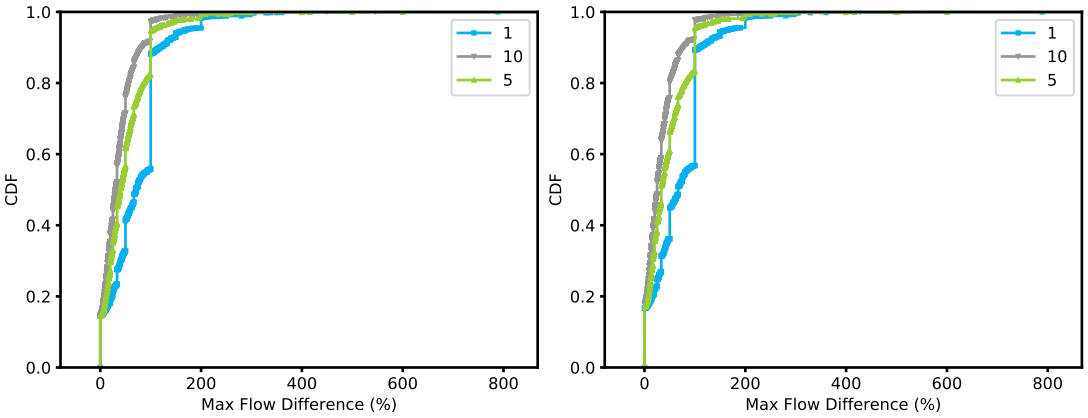
[1] François Aubry, David Lebrun, Yves Deville, and Olivier Bonaventure. Traffic duplication through segmentable disjoint paths. In *2015 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2015.

[2] Zhiruo Cao, Zheng Wang, and Ellen Zegura. Performance of hashing-based schemes for internet load balancing. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, volume 1, pages 332–341. IEEE, 2000.

[3] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[4] Gines Garcia-Aviles, Marco Gramaglia, Pablo Serrano, Marc Portoles, Albert Banchs, and Fabio Maino. Semper: a stateless traffic engineering solution for wan based on mp-tcp. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

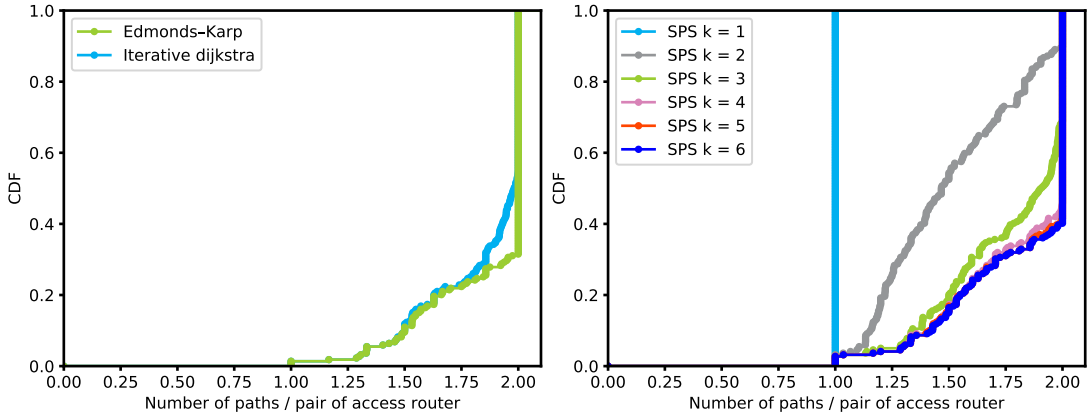
[5] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter



(a) Single path transport protocol

(b) Multiple path transport protocol

Figure 5: Relative Maxflow improvement of adding IPv6 SR



(a) Without a segment limit

(b) With a segment limit

Figure 6: Number of disjoint paths found by pair of access routers

networks. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 149–160. ACM, 2014.

[6] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[7] David Lebrun, Mathieu Jadin, François Clad, Clarence Filsfils, and Olivier Bonaventure. Software resolved networks: Rethinking enterprise networks with ipv6 segment routing. In *Proceedings of the Symposium on SDN Research*, page 6. ACM, 2018.

[8] J. Tantsura. The critical role of Maximum SID Depth (MSD) hardware limitations in Segment Routing ecosystem and how to work around those. In *NANOG71*, 2017.