Traffic duplication through segmentable disjoint paths

François Aubry, David Lebrun, Yves Deville, Olivier Bonaventure

ICTEAM, Université Catholique de Louvain
Louvain-La-Neuve – Belgium
firstname.name@uclouvain.be

Ultra-low latency is a key component of safety-critical operations such as robot-assisted remote surgery or financial applications where every single millisecond counts. In this paper, we show how network operators can build upon the recently proposed Segment Routing architecture to provide a traffic duplication service to better serve the users of such demanding applications. We propose the first implementation of Segment Routing in the Linux kernel and leverage it to provide a traffic duplication service that sends packets over disjoint paths. Our experiments show that with such a service existing TCP stacks can preserve latency in the presence of packet losses. We also propose and evaluate an efficient algorithm that computes disjoint paths that can be realised by using segments. Our evaluation with real and synthetic network topologies shows that our proposed algorithms perform well in large networks.

1. INTRODUCTION

A wide range of Internet applications require both low latency and reliability. In the TCP/IP protocol suite, reliability has always been provided by the Transmission Control Protocol (TCP) whose implementation has been heavily tuned over the years [**?**]. Despite all these optimisations, short TCP flows can be severely affected by packet losses that are not a rare event. A measurement study performed on the google front end servers revealed TCP retransmit rates in the range 0.8% - 2.4% [**?**].

Various entreprise applications also require reliable low latency services. Since the advent of electronic trading, the financial world has been actively researching, developing and applying techniques to reduce the latency of their systems, hence allowing for a larger number of transactions per unit of time, yielding a higher profit. A typical use case is high frequency trading [**?**]. The desperate need for information to arrive as fast as possible has motivated some institutions to provide special low latency services for a fee. For example, Thomson Reuters supplies to its premium customers numbers such as Consumer Confidence two seconds before it is released to the general public for a $2,000/month fee [**?**]. Cable providers install new optical fibers or microwave links over shorter paths [**?**] to better server financial institutions. For high-frequency trading, every millisecond counts. For example, a 1-millisecond advantage for a major brokerage firm can yield a loss or a profit of $100 million a year [**?**].

These financial applications are only one example among a long list of applications [**?**, **?**] that require both low latency and reliable delivery. Other examples include requests for web objects in a CDN network, telesurgery and even online games. These applications typically exchange short amounts of data. The latency that affects such an application is spread over different systems, i.e. processing by APIs, applications, propagation between internal systems, and finally network processing and propagation. We focus on the networking part of the chain and propose a novel technique that network operators can deploy to ensure low-latency transmission of information between a source and sink in an IP network.

We build upon the *Low Latency with Redundancy* proposal [**?**] and extend it to replicate packets over disjoint paths between the communicating nodes. We implement this duplication service in the Linux kernel and demonstrate experimentally that this solution is compatible with existing Internet protocols and applications including TCP. We also leverage the recently proposed IPv6 Segment Routing technique to realise these disjoint paths and propose efficient algorithms to compute disjoint paths that are compatible with Segment Routing. Our work shows that traffic duplication can be offered as a service to demanding applications in entreprise or ISP networks.

This paper is organised as follows. We first describe in section 2 how the recently proposed Segment Routing architecture can enable an ISP to deploy such a duplication service. In section 2.1, we implement a traffic duplicator in the Linux kernel and evaluate its performance with regular TCP traffic. We explain in section 3 how segmentable disjoint paths can be used to realize the traffic duplication service. We then propose in section 3.3 an efficient algorithm that computes such disjoint paths. We evaluate its performance on several

network topologies in section 4. We discuss related work in section 5.

2. ROUTER-LEVEL TRAFFIC DUPLICATION

In this section, we show how an ingress router can duplicate the traffic towards a given destination in a pure IPv6 network. Our traffic duplicator builds upon Segment Routing. Segment Routing (SR) is a new forwarding architecture that is being developed within the Internet Engineering Task Force. Segment Routing changes the way packets are forwarded inside a network to enable network operators to have better control on the path followed by the packets. In pure IP networks, packets follow the shortest path towards their destination. In the control plane, this is achieved by using routing protocols such as OSPF, IS-IS and BGP. The selection of the shortest paths depends on the weights associated to each link. In the data plane, each IP packet contains the source and destination addresses.

With SR, the path followed by a packet does not need to be the shortest path towards its destination. The proposed SR architecture [7] modifies the control and the data planes to support such non-shortest paths. Two variants of the data plane exist : MPLS-based and IPv6-based [17]. In this paper, we focus on the IPv6-based data plane and its modifications to support segment routing [15]. With Segment Routing, the path between a source and a destination is composed of one or more segments. Segment Routing supports two types of segments: node segments and adjacency segments. To understand these two types of segments, let us consider the network shown in figure 1 where all links have the same weight. Consider the paths from `a` to `h`. With shortest path routing, the path `a-b-c-d-h` is used. With segment routing, we can force the utilization of other paths. A first solution is to use a node segment towards `f`. The packets sent by `a` will first follow the shortest path towards `f` and from there the shortest path towards the final destination, i.e. `h`. Another possibility is to use an adjacency segment. In this case, the source node includes in the packet a specific link that needs to be used by the path. For example, if link `c-f` is chosen, then the packets sent by `a` will reach `f` via the single path `a-b-c-f`. `f` will then forward the packets towards `h` over the shortest path. By combining node and/or adjacency segments, network operators have proposed various new innovative services [8].

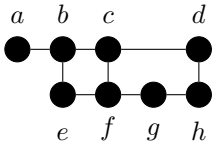


Figure 1: Simple network

In a regular IPv6 network, the intradomain routing protocol (i.e. OSPF or IS-IS) is configured to advertise the IPv6 loopback address of each router and the IPv6 addresses of each of its physical interfaces. BGP is used to advertise the external destinations. The Segment Routing architecture reuses this control plane. To enable the utilization of any path, SR defines a new IPv6 hop-by-hop header [17]. This header is a revision of the Type 0 IPv6 Routing Header that was deprecated a few years ago due to security problems.. The SR header contains a list of IPv6 addresses that specify a path through the network. To encode a node segment, the IPv6 loopback address of the corresponding router will be inserted in the SR header. For an adjacency segment, the IPv6 address of the outgoing interface will be used. It also includes a HMAC, which did not exist in the Type 0 Routing Header that solves the security problems that caused the deprecation of RH0 [17, 15].

As with the Type 0 Routing Header, routers forward the packets based on their destination address. A router only checks the SR header when it receives a packet destined to itself. In this case, it looksup the next address in the SR header, updates the `Last Segment` field and uses this address as the destination address of the forwarded packet.

2.1 Evaluation

We have implemented Segment Routing in the Linux kernel (3.14.x branch) through an IPv6 header extension (SR-IPv6). Our extension [15] implements the current draft [17] as much as possible. Our implementation is able to forward packets containing an SR header and to inject the SR header in a forwarded packet that matches a given destination prefix. Our implementation contains about 2,000 lines of code. The processing of an IPv6 packet containing an SR header is defined in algorithm 1.

Algorithm 1 SR header processing

```
1: if DA = myself (segment endpoint) then
2:    if Next Segment <> Last Segment then
3:       Update DA with Next Segment
4:       Increment Next Segment
5:    else if Last Segment <> DA then
6:       Update DA with Next Segment
7:       if Clean-up bit is set then
8:          Remove the SR header
9:       end if
10:   else
11:      Locally deliver the packet
12:   end if
13: end if
14: Forward the packet out
```

The processing of the SR header happens when the

packet is considered for local delivery, after the `PREROUTING` and right after the `INPUT` hooks (as the destination address of the packet belongs to the router when it is a segment endpoint), during the normal processing of IPv6 header extensions. If the packet passes the normal checks (HMAC, correct header format, etc.) and the router is not the last segment then the destination address of the packet is changed to the next segment and the packet is sent through the forwarding mechanism of the kernel. If the processing node is the last segment, then the packet is delivered to the local application or to the corresponding kernel routine. See figure 2 for an illustration of the Linux routing mechanism [18].

Figure 2: Linux routing mechanism

Our implementation is also able to add an SR header when forwarding a regular IPv6 packet whose destination matches a given prefix. This injection is performed just before the `PREROUTING` hook because we need to modify the original destination address of the packet to the address of the first segment. We provide a userland/kernel API that allows the construction of a segment table. This table contains a list of destination prefixes. For each destination prefix, there is an indexed set of segments lists. If there is only one segment list, then the kernel uses this list for the SR header. If there is more than one list, then the kernel can perform two different actions: in one of them (`SPLIT_RR`), the kernel will act as a round/robin over the available lists. In the other action (`MIRROR`), the kernel will forward n times the original packet, where n is the number of lists available. This `MIRROR` mode is the one we use in our experiments.

The ability to provide more than one segments list per destination prefix enables the kernel to act as a per-packet load balancer in order to spread the traffic across the available paths. Due to space limitations, we refer the reader to [15] for a detailed description of this implementation.

2.2 Impact of duplication on TCP performance

Most applications that require a reliable delivery rely on TCP. In this section, we experimentally evaluate whether TCP is compatible with traffic duplication. The answer to this question is not immediate since traffic duplication may generate multiple duplicate acknowledgments and adversely impact TCP performance by triggering fast retransmits.

Due to space limitations, we focus our evaluation on the impact of two main parameters that affect link performance : delay d and packet loss l. We created a virtual topology[1] within the Mininet framework [12] (see figure 3). Our reliable application is a web server. We requested 100 KBytes files from an HTTP server (the `lighttpd` dameon) to simulate low-latency requests. To measure the impact of d and l, we used the following methodology. Given a set of base RTTs $brtt = \{1, 5, 10\}$ milliseconds, a set of delta RTTs $drtt = \{2, 4, 6, 8, 10\}$, for each b in $brtt$ and for each d in $drtt$, we set the delay of the link `L1` to b ms and the delay of the link `L2` to $b + d$ ms. Then, we measure the total HTTP request time with an increasing packet loss percentage on link `L1`. We used `tc` with the `netem` module to set the delay of the links and the `htb` module to set the bandwidth of links `L1` and `L2` to 100 Mbps. The bandwidth of all other links is not limited.

Figure 3: Test network

We performed 1,000 requests of 100KB between `hostA` and `hostB` for each combination in the sets. Figure 4 reports the request reponse times measured over 1,000 requests generated across link `L1` with a loss of 10% and a delay of 10 ms; across link `L2` with a loss of 0% and a delay of 20 ms; and finally across both links using our traffic duplicator. We can clearly see that using both links at the same time yields a shorter response time than when using either link individually.

Tables 1 and 2 show the latency measured over 1,000 requests using traffic duplication by setting link L1 to resp. 5ms and 10ms and link L2 to resp. $5 + \Delta d$ ms and $10 + \Delta d$ ms. We can clearly see that on average, the fastest link wins. We can also see that the latency is more stable when the ratio of the latency of the slowest link over the fastest link is smaller.

	Min	Avg	Stddev
Δd 2ms	70ms	70ms	0ms
Δd 4ms	70ms	70ms	0ms
Δd 6ms	70ms	70.37ms	5.79ms
Δd 8ms	70ms	70.11ms	1.86ms
Δd 10ms	70ms	70.01ms	0.31ms

Table 1: Latency measured over 1,000 requests with link L1 set at 5ms and link L2 set at $5 + \Delta d$ ms

3. SEGMENTABLE DISJOINT PATHS

The previous section has shown how a router can send packets from a given flow over disjoint paths by inserting different SR headers. To implement our service, the router needs to know which SR header to be used for each destination. For this, we propose algorithms that compute link and node disjoint paths. These algorithms could be implemented in a centralized controller or directly on the router. Given that our service

[1]The virtual image used for the experiment is available from `http://www.segment-routing.org`.

	Min	Avg	Stddev
Δd 2ms	140ms	140.08ms	2.52ms
Δd 4ms	140ms	140ms	0ms
Δd 6ms	140ms	140ms	0ms
Δd 8ms	140ms	140ms	0ms
Δd 10ms	140ms	140.3ms	6.35ms

Table 2: Latency measured over 1,000 requests with link L1 set at 10ms and link L2 set at 10 + Δd ms

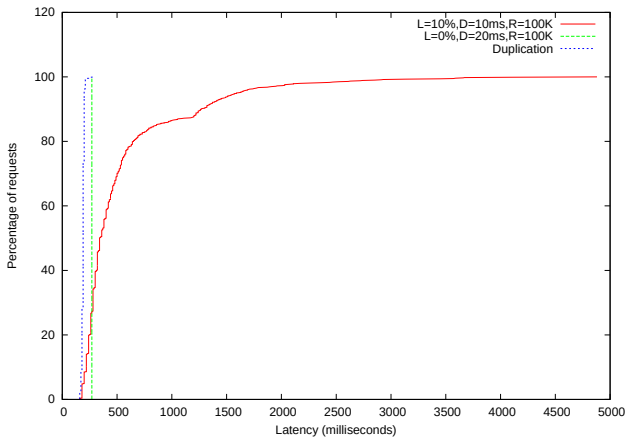


Figure 4: Percentage of requests having a given latency

builds upon the Segment Routing architecture, we need to ensure that the disjoint paths that are computed by our algorithm can be implemented as a sequence of segments. In general, any path could be implemented as a sequence of segments. However, this could require as many segments as there are links on a given path. This would lead to very long SR headers that would not be acceptable from a packet overhead viewpoint. Furthermore, some deployed routers have difficulties in forwarding packets that contain long extension headers. For this reason, we limit the number of segments that the SR header can contain. Due to space limitations, we focus on node segments, but a similar approach can be developed for adjacency segments. The proofs of the theorems may be found in [**?**].

3.1 Notations and definitions

We model the intradomain network as a directed weighted graph $G = (V, E, w)$ where V is a set of nodes, E is a subset of $V \times V$ that corresponds to the links and w is a function from E to $\mathbb{R}^+$. This function corresponds to the IGP costs configured on the links.

A *path* p is a sequence $(x_1, x_2, \ldots, x_h)$ such that $(x_i, x_{i+1}) \in E$ for all i and $x_i \neq x_j$ for $i \neq j$.

Let $c : E \to \mathbb{R}^+$. We define the *cost of a path* $p = (x_1, \ldots, x_h)$ relative to c, denoted $c(p)$, as the sum of the weights of its edges:

$$c(p) = \sum_{i=1}^{h-1} c(x_i, x_{i+1})$$

The *edge set of a path* $p = (x_1, \ldots, x_h)$, denoted $E(p)$, is defined as the set of edges that belong to p:

$$E(p) = \{(x_i, x_{i+1}) \mid i \in \{1, \ldots, h-1\}\}$$

Two paths p_1, p_2 from a node s to a node t are said to be *edge disjoint* if $E(p_1) \cap E(p_2) = \emptyset$.

Let $p_1 = (x_1, x_2, \ldots, x_{h_1})$ and $p_2 = (y_1, y_2, \ldots, y_{h_2})$ be two paths on a graph G. If $(x_{h_1}, y_1) \in E$ we denote by $p_1 + p_2$ path that result from appending p_2 to p_1

$$p_1 + p_2 = (x_1, x_2, \ldots, x_{h_1}, y_1, y_2, \ldots, y_{h_2})$$

If $x_{h_1} = y_1$ we denote by $p_1 \oplus p_2$ the concatenation of p_1 with p_2, that is,

$$p_1 \oplus p_2 = (x_1, x_2, \ldots, x_{h_1} = y_1, y_2, \ldots, y_{h_2})$$

Definition 1. *Given a graph $G = (V, E, w)$ and a path $p = (x_1, x_2, \ldots, x_h)$ in G we say that the path p is* k-segmentable *if there exist k shortest paths $s_1, s_2, \ldots, s_k$ in G such that $p = s_1 \oplus s_2 \oplus \ldots \oplus s_k$. The sequence $S = \langle s_1, s_2, \ldots, s_k \rangle$ is called a* k-segmentation *of p.*

A path is said to be *unsegmentable* if it is not k-segmentable for any k. A k-segmentation of a path p is said to be *minimal* if there exists no k'-segmentation of p such that $k' < k$. Note that there can exist several

minimal segmentations. For instance in the graph on figure 5 the path (a, b, c, d) has two minimal segmentations: $\langle (a, b), (b, c, d)\rangle$ and $\langle (a, b, c), (c, d)\rangle$.

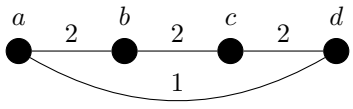


Figure 5: A graph G with a path (a, b, c, d) that has two minimal segmentations: $\langle (a, b), (b, c, d)\rangle$ and $\langle (a, b, c), (c, d)\rangle$.

3.2 Path segmentation

In this section we explain how we can determine whether a path is k-segmentable and how to compute a minimal k-segmentation. Let $G = (V, E, w)$ be a graph. Let us denote by $\mathcal{D}_v$ the shortest path dag (directed acyclic graph) rooted at v. That is, $\mathcal{D}_v$ is the subgraph of G that contains all the edges that belong to a shortest path from v to some node. These n graphs can be computed in $O(n^3)$ using the Floyd-Warshall algorithm [3].

Naturally, k-segmentations of paths in G are intimately related to the shortest paths dag's of G. Consider a k-segmentable path $p = (x_1, x_2, \ldots, x_h)$ whose segmentation is $\langle s_1, s_2, \ldots, s_k\rangle$. By definition each s_i is a shortest path in G. Hence s_i is a path in $\mathcal{D}_{v_i}$ where v_i is the first vertex of s_i. This means that a path is k-segmentable if and only if it can be expressed as the concatenation of k paths each belonging to some of G's shortest paths dag's. The next lemma gives a characterization of segmentable paths in a graph.

LEMMA 3.1. *A path p in G is unsegmentable if and only if there exists some edge (x_1, x_2) in p such that for all $x \in V$, $(x_1, x_2) \notin \mathcal{D}_x$.*

PROOF. Let $p = (x_1, x_2, \ldots, x_h)$. If every edge (x_i, x_{i+1}) in p belongs to a shortest path dag then we have a trivial segmentation $s = \langle (x_1, x_2), (x_2, x_3), \ldots, (x_{h-1}, x_h)\rangle$ for p. Hence p is segmentable. □

In the previous lemma showed that for a path to be unsegmentable it must have at least one edge that does not belong to any shortest path dag. The next lemma shows that given an edge (u, v) we actually only have to check whether is belongs to $\mathcal{D}_u$ in order to determine if it belongs to some shortest path dag.

LEMMA 3.2. *If $(x_1, x_2) \notin \mathcal{D}_{x_1}$ then for each $x \in V$, $(x_1, x_2) \notin \mathcal{D}_x$*

PROOF. Suppose that $(x_1, x_2) \notin \mathcal{D}_{x_1}$ and that there exists $x \in V$ such that $(x_1, x_2) \in \mathcal{D}_x$. Then there exists a shortest path p from x to x_2 such that its last edge is (x_1, x_2). But because $(x_1, x_2) \notin \mathcal{D}_{x_1}$ the path consisting of a single edge (x_1, x_2) is not a shortest path. Since any subpath of a shortest path must also be a shortest path and (x_1, x_2) is a subpath of p, p cannot be a shortest path. This contradicts the optimality of p. □

In general, to obtain a minimal segmentation, we traverse the path p by walking on the shortest paths dag's of G. We start on $\mathcal{D}_{x_1}$ and follow the path until we reach an edge that does not belong to the current dag. Denote that edge by (x_i, x_{i+1}). Then if (x_i, x_{i+1}) belongs to $\mathcal{D}_{x_i}$ we continue our walk in $\mathcal{D}_{x_i}$. Otherwise, by Lemmas 3.1 and 3.2, the path cannot be segmented so we stop and report it.

3.3 Minimum latency K-segmentable path

In this section we describe an algorithm to find the minimal latency K-segmentable path between two vertices in a graph.

Given a graph $G = (V, E, w)$ and a latency function $l : E \to \mathbb{R}^+$ we define a graph $\overline{G}$ such that a path $\overline{p}$ in $\overline{G}$ corresponds a segmentation of a path p in G such that $l(\overline{p}) = l(p)$. More precisely, a node of $\overline{G}$ is a tuple $(v, \mathcal{D}_r, k)$ that is to be interpreted as a state with the following meaning: v is the current node in G, $\mathcal{D}_r$ is the current shortest path dag and k is the number of segments used so far. The edges in $\overline{G}$ are defined such that there is a path in $\overline{G}$ from $(v, \mathcal{D}_v, 1)$ to $(u, \mathcal{D}_r, k)$, for some u, r, k, if and only if there is a k-segmentable path from v to u in G.

Formally, we define $\overline{G} = (\overline{V}, \overline{E})$ with $\overline{V} = \{(v, \mathcal{D}_r, k) \mid v, r \in V \text{ and } 1 \leq k \leq K\}$. The edges $\overline{E}$ are defined as follows: $(v_1, \mathcal{D}_{r_1}, k_1)$ and $(v_2, \mathcal{D}_{r_2}, k_2)$ are connected by an edge of cost $l(v_1, v_2)$ if one of the following conditions holds: (i) $(v_1, v_2) \in \mathcal{D}_{r_1}$, $r_2 = r_1$ and $k_2 = k_1$ and (ii) $(v_1, v_2) \in \mathcal{D}_{v_1}$, $r_2 = v_1$, $k_2 = k_1 + 1 \leq K$.

We will denote the size of $\overline{V}$ by $\overline{n}$ and the size of $\overline{E}$ by $\overline{m}$.

To better understand this construction consider the graph in figure 6 and the path (a, b, d, f, h, i). The first two edges (a, b) and (b, d) belong to $\mathcal{D}_a$. However the third edge (d, f) does not because the shortest path from a to f is (a, c, e, f) of length 4 whereas the path (a, b, d, f) has length 5. This means that we need a segment at node d. Now we are not in the dag of a anymore but in the dag of d. One easily sees that all the remaining edges belong to $\mathcal{D}_d$ so no further segments are required. The decomposition of this path is $\langle (a, b, d), (d, f, h, i)\rangle$. This path corresponds in $\overline{G}$ to the path $(a, \mathcal{D}_a, 1)$, $(b, \mathcal{D}_a, 1)$, $(d, \mathcal{D}_a, 1)$, $(f, \mathcal{D}_d, 2)$, $(h, \mathcal{D}_d, 2)$, $(i, \mathcal{D}_d, 2)$. The first coordinates represent the corresponding node in G. The second coordinates represent the current shortest path dag root. We see that the first three nodes have second coordinate equal to a which means that that portion of the path is executed in $\mathcal{D}_a$. Then from $(d, \mathcal{D}_a, 1)$ to $(f, \mathcal{D}_d, 2)$ the second coordinate changes showing that we had to change the current shortest path dag in order to proceed. In this

case we changed from $\mathcal{D}_a$ to $\mathcal{D}_d$. The third coordinate shows the total number of segments required.

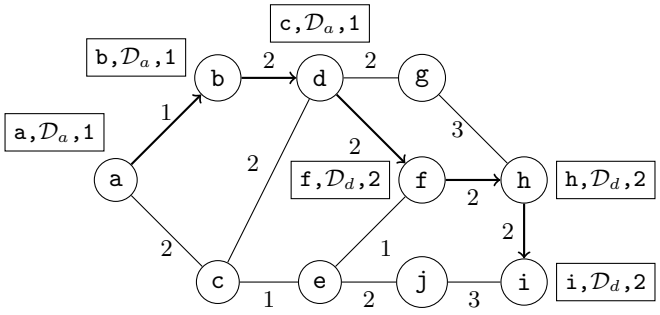


Figure 6: Example for $\overline{G}$

We now bound the size of $\overline{G}$ and propose an algorithm that computes it.

LEMMA 3.3. *The number of edges of $\overline{G}$ is $O(nmK)$.*

PROOF. A vertex (v, r, k) in $\overline{G}$ has at most $d^+(v)$ outgoing edges satisfying each one of the conditions. Thus

$$|\overline{E}| \leq 2 \sum_{v,r,k} d^+(v) = 2 \sum_{r,k} \sum_{v} d^+(v)$$
$$= 2 \sum_{r,k} m = 2nmK$$

□

Algorithm 2 builds the graph $\overline{G}$ given G, a latency function l and the shortest path dag's of G with respect to the IGP costs. To do so, for each vertex $(v, \mathcal{D}_r, k)$ it loops over all neighbors u of v in G and checks if either condition (1) or (2) of the definition of the edges of $\overline{G}$ applies. By Lemma 3.3 the algorithm runs in $O(nmK)$. The algorithm has a pre-processing cost of $O(n^3)$ because we need to compute the shortest path dag's $\mathcal{D}_v$ for all $v \in V$.

Now we establish formally the correspondence between paths in G and their segmentations and paths in $\overline{G}$.

PROPOSITION 3.1. *Let $G = (V, E, w)$ be a network, $l : E \to \mathbb{R}^+$ and $C \geq 0$.*

There exists a k-segmetable path $p = (x_1, x_2, \ldots, x_h)$ in G with $l(p) = C$ if and only if there exists in $\overline{G}$ a path $\overline{p}$ of the form $(x_1, \mathcal{D}_{x_1}, 1)$, $(x_2, \mathcal{D}_{r_2}, k_2)$, $(x_3, \mathcal{D}_{r_3}, k_3)$, $\ldots$, $(x_h, \mathcal{D}_{r_h}, k)$ with $l(\overline{p}) = C$.

PROOF. ($\Rightarrow$) Let $p = (x_1, x_2, \ldots, x_h)$ be a k-segmentable path in G with $l(p) = C$. Let $\langle s_1, \ldots, s_k \rangle$ be a k-segmentation of p and denote number of vertices of s_i by l_i and the j-th vertex of s_i by $s_i[j]$. Let $\mathcal{D}_i$ be a dag that contains s_i and let

$$\overline{s}_i = ((s_i[1], \mathcal{D}_i, i), (s_i[2], \mathcal{D}_i, i), \ldots, (s_i[l_i], \mathcal{D}_i, i)) .$$

Algorithm 2 Build-Graph

Input:

- A graph $G = (V, E)$
- $l : E \to \mathbb{R}^+$
- $\mathcal{D}_v$ for all $v \in V(G)$

Output:

- The graph $\overline{G}$ as defined above

```
1: Ḡ ← ({(v, D_r, k) | v, r ∈ V(G) ∧ k ∈ {1, ..., K}}, ∅)
2: for v, r ∈ V(G), k ∈ {1, ..., K} do
3:    for u ∈ neighbors(G, v) do
4:       if (v, u) ∈ D_r then
5:          connect(Ḡ, (v, D_r, k), (u, D_r, k), l(v, u))
6:       end if
7:       if k + 1 ≤ K and (v, u) ∈ D_v then
8:          connect(Ḡ, (v, D_r, k), (u, D_v, k + 1), l(v, u))
9:       end if
10:   end for
11: end for
12: return Ḡ
```

By the definition, each $\overline{s}_i$ is a path in $\overline{G}$ (all the edges satisfy the condition (i) of the definition of edges in $\overline{G}$). For each i we have that $s_i[l_i] = s_{i+1}[1]$ so that $(s_i[l_i], s_{i+1}[2]) = (s_{i+1}[1], s_{i+1}[2]) \in \mathcal{D}_{i+1}$. Therefore by condition (ii) of the definitions of edges in $\overline{G}$, we have that $(s_i[l_i], \mathcal{D}_i, i)$ is connected to $(s_{i+1}[1], \mathcal{D}_{i+1}, i+1)$ so $\overline{s}_1 + \overline{s}_2 + \ldots + \overline{s}_k$ is a path in $\overline{G}$. By the way costs are defined on $\overline{G}$ we have that $l(p) = l(\overline{p})$. Since

$$s_1 \oplus s_2 \oplus \ldots \oplus s_k = (x_1, x_2, \ldots, x_h)$$

we have what we wanted.

($\Leftarrow$) Let $\overline{p}$ be a in $\overline{G}$ of the form $(x_1, \mathcal{D}_{x_1}, 1)$, $(x_2, \mathcal{D}_{r_2}, k_2)$, $(x_3, \mathcal{D}_{r_3}, k_3)$, $\ldots$, $(x_h, \mathcal{D}_{r_h}, k)$. By construction, $(x_1, x_2, \ldots, x_h)$ is a path in G with the same cost. Since the third coordinate of the last vertex is k is means that the second coordinate changes exactly k times. Thus p is k-segmentable. □

Then, given s and t, by Proposition 3.1, we can find the shortest s-t path on G that requires at most K segments by computing a shortest path on $\overline{G}$. This can be achieved with Dijkstra's algorithm. However the following lemma tells us that we can do this even faster.

LEMMA 3.4. *Given a graph G, the corresponding graph $\overline{G}$ is acyclic.*

PROOF. Suppose that $\overline{G}$ has a cycle. Let its vertices be $(v_1, \mathcal{D}_{r_1}, k_1)$, $(v_2, \mathcal{D}_{r_2}, k_2)$, $\ldots$, $(v_{h-1}, \mathcal{D}_{r_{h-1}}, k_{h-1})$, $(v_1, \mathcal{D}_{r_1}, k_1)$. By the definition of the edges of $\overline{G}$ we have that $k_i \leq k_{i+1}$. But then we have $k_1 \leq k_2 \leq \ldots \leq k_{h-1} \leq k_1$ so $k_1 = k_2 = \ldots = k_{h-1}$. Again by the definite of the edges of $\overline{G}$ we have that if an

edge maintains the value of the third coordinate then it maintains the second coordinate. Thus $\mathcal{D}_{r_1} = \mathcal{D}_{r_2} = \ldots = \mathcal{D}_{r_{h-1}}$. But this means that $(x_1, x_2, \ldots, x_{h-1}, x_1)$ is a cycle in D_{r_1}. This contradicts the fact that D_{r_1} is a dag. $\square$

Since $\overline{G}$ is acyclic shortest paths can be computed in $O(\overline{m}+\overline{n})$ using Dynamic Programming [3]. By Lemma 3.3, we can express the time complexity as $O(nmK + n^2K)$. The algorithm is specified as algorithm 3. In this algorithm we call DAG-SP to compute the shortest paths. We suppose that it outputs two vectors d and π, the first containing the shortest path distance to each vertex and the second containing the parent of each node in the shortest path tree. We take the convection that when no path exists to a given vertex the distance is infinite. Using π we easily reconstruct the path by taking the first coordinate of the nodes. The segments are reconstructed using the third coordinates which represent the number of segments. Whenever this changes it means that a new segment was used.

Notice that for a fixed source s we do not need to build the whole graph $\overline{G}$. Building the subgraph of nodes reachable from $(s, \mathcal{D}_s, 1)$ is sufficient and more efficient.

Algorithm 3 SPS

Input:

- The graph $\overline{G}$
- Two nodes $s, t \in V(G)$
- The maximum number of segments K

Output:

- Shortest K-segmentable s-t path

```
1: (d, π) ← DAG-SP(Ḡ, (s, s, 1))
2: (t, D_{r*}, k*) ← arg min_{r,k}{(t, D_r, k) | d(t, D_r, k) < ∞}
3: if (t, D_{r*}, k*) = nil then
4:    return nil
5: end if
6: path ← ∅
7: seg ← ∅
8: cur ← (t, r*, k*)
9: while cur ≠ nil do
10:   path ← path ∪ {cur.vertex}
11:   if π[cur] ≠ nil and π[cur].k ≠ cur.k then
12:      seg ← seg ∪ {π[cur].vertex}
13:   end if
14:   cur ← π[cur]
15: end while
16: return (path, seg)
```

3.4 Computing disjoint K-segmentable paths with low latencies

Now that we have an algorithm that computes K-segmentable paths, we can find a set of edge disjoint paths by iteratively calling algorithm 3. Each time the algorithm finds a path we remove all of its edges from the graph. We continue until no more path exists. If we want to provide traffic duplication over P paths and the algorithm finds more than $P' > P$ paths we can keep the worst $P' - P$ as secondary path that we use in case there is a failure in one of the paths we are using. This allows to maintain a set of P paths even in the case of a failure.

One thing that we have to be careful with is that an edge in G corresponds to many edges in $\overline{G}$. Formally an edge (v, u) in G corresponds to all the edges in $\overline{G}$ of the form $((x, \mathcal{D}_{r_1}, k_1), (y, \mathcal{D}_{r_2}, k_2))$ for $r_1, r_2 \in V(G)$ and $k \in \{1, \ldots, K\}$. We can avoid to explicitly remove all these n^2K^2 edges by using a n by n boolean matrix A such that $A[v][u]$ is true if the edges of the form

$$\{((v, r_1, k_1), (u, r_2, k_2)) \mid r_1, r_2 \in V \wedge k_1, k_2 \in \{1, \ldots, K\}\}$$

are active and false otherwise. Then, instead of removing the edges we set $A[v][u]$ to false and take the values of A into consideration when finding paths.

The time complexity of finding P paths is $O(P(nmK + n^2K))$ since the SPS algorithm runs in $O(nmK + n^2K)$.

4. EVALUATION

In the previous sections, we have proposed algorithms that enable routers to compute segmentable disjoint paths. In order to assess their possible application, we performed various experiments by considering several topologies of Internet Service Providers (ISP). Each topology is a graph with two attributes per link : the IGP metric and the latency. We use five topologies for our experiments. The first four ones were collected by the Rocketfuel project and are described in [20]. Both the IGP weights and the latencies were inferred from traceroute data in these topologies. The last topology includes the backbone routers of a large Tier-1 ISP with the real IGP weights. The link latencies were computed from the geographical distance between the cities. Table 3 provides some data about these topologies.

Name	# nodes	# edges
Rocketfuel : AS1239	153	1010
Rocketfuel : AS1755	67	248
Rocketfuel : AS3257	103	484
Rocketfuel : AS3967	57	208
Real backbone	approx 150	approx. 700

Table 3: ISP Topologies

We implemented the algorithms described in the previous sections in Java. Our code contains 3 classes and

600 lines of code. For our evaluation, we used an intel Core i7 on a laptop with 4 GBytes of RAM using Linux Ubuntu 14.04 LTS.

In all our experiments we set $K = 3$ meaning that we use at most 3 segments for each path. We performed experiments with other values and they show that we do not gain much by allowing more segments. Our first evaluation is to measure the number of link and node disjoint paths that exist in these networks. Figure 7 provides the proportion of the router pairs for which there are at least P link disjoint paths. This proportion was obtained by iteratively running algorithm 3 on each topology for all pairs of nodes until no more paths where found. The proportion of the number of link disjoint paths depends on several factors, notably the number of links attached to each router. We observe that for more than 90% of the pairs at least 2 disjoint paths exist in each topology, for 40% of the pairs we are able to find 3 paths and that for approximatively 20% of the pairs we are able to find 4 paths.

Using a brute force integer programming model we computed the exact maximum number of disjoint paths for the AS1755 and AS3967 topologies. This shows that our algorithm achieves the maximum number of paths for 96% of the pairs for the AS1755 topology and 98% for the AS3967 topology. This shows that, at least in these topologies, it would not be possible to find much more paths. This information could not be computed for the other topologies due to the exponential complexity of the brute force algorithm.

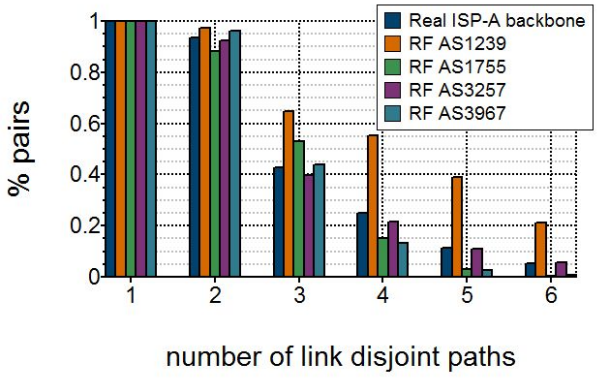


Figure 7: For $P = 1, \ldots, 6$ the figure shows the percentage of pairs in each graph for which we can find at least P disjoint paths.

As we shown in Section 2.1, when the difference of latency between the best path and the worst path in a path set increases we get a decrease of performance of TCP. Figure 8 shows for each Δd what is the percentage of pairs such that the difference between their worst path and their best path is at most Δd. We observe that for 2 paths more that 90% of the pairs have $\Delta d < 10ms$. For 3 paths we have 75% of the pairs.

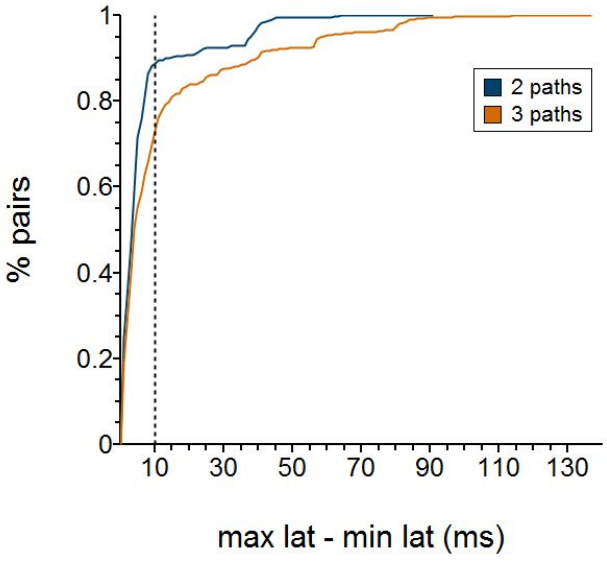


Figure 8: For $P = 2, 3$ and Δd we show the percentage of pairs that have a difference of latency between the IGP path and the worst path in the path set found by our algorithm at most Δd.

In a real network, the running time of such algorithms can be an important factor. Link and node failures are relatively frequent events in large ISP networks. After a remote failure, a router may need to recompute the disjoint paths that it uses to reach distant routers. As we said previously, we run our algortihm until no more paths are found. This means that we may find a lot more disjoint paths than the ones we use for traffic duplication. If this is the case we can instantly switch to another path upon a link failure. In the meantime we recompute a new path set from scratch to try to find a better set of paths. In the real ISP, our unoptimized Java code takes on average less than 315 milliseconds to find all the paths from a source to a destination. This remains a reasonable computation time compared with the various computations that the IGP already requires on routers [10].

5. RELATED WORK

The *Low latency via Replication* proposal [**?**] is the closest to our work. They also propose replicate packets across diverse resources then use the first answer. However, their approach differs in the sense that they use replication to query different systems such as DNS, while our solution use multiple disjoint paths to interact with one destination. Our approach is more network-based rather than system-based.

Different source routing techniques have been pro-

posed in the literature, e.g. [19, 13, 21, 5]. Segment Routing is a modern and standardised form of source routing. It inherits several of the features of the earlier proposals. On the other hand, Segment Routing has two advantages. First, its standardisation progresses well within the IETF, which implies that it will be useable in real networks in the near future. Our implementation in the Linux kernel will also contribute to this deployment on both servers and routers. Second, the IPv6 version of Segment Routing solves the security problems that affected the IPv6 Source Routing header and other proposals.

Another type of approach is to rely on Forward Error Correction to protect reliable services from the impact of losses. Balakrishnan *et al.* propose in [**?**] a Forward Error Correction (FEC) mechanism to recover from bursty losses. The goal of the authors is different in the sense that they want to recover from loss, while we focus on minimizing network delay.

In the literature there are essentially two approaches to compute disjoint paths between a source and a destination[6]. The most straightforward approach is in the same spirit as ours in the sense that it consists on successively computing shortest path with the Dijkstra algorithm. Then the paths are deleted until no more path can be found [11]. To our knowledge, none of these approaches have been extended to support Segment Routing. A frequent cited drawback of using successive shortest paths is that there are pathological cases where the algorithm is trapped in a path and cannot find the disjoint ones. This happens when a shortest path has edges in common with other potential paths. Although this problem exists, [6] has shown that it is very rare in real networks. The results obtained by our exact algorithm confirm this for the AS1755 and AS3967 topologies since we where able to find the maximum number of paths possible for more than 96% of the pairs. A more sophisticated approach consists of computing the maximum flow between the source and the destination [2].

6. CONCLUSION

In this paper, we have proposed three main contributions that allow network operators to offer a better service for mission critical applications that require both low latency and reliability. Our solution builds upon the Segment Routing architecture that is currently being discussed within the Internet Engineering Task Force.

Our first contribution is a complete and open-source implementation of the IPv6 version of Segment Routing inside the Linux kernel. This is the first complete implementation on this new protocol in an operating system kernel. Our second contribution is the implementation and of a traffic duplication service that uses disjoint paths by leveraging the capabilities of our Segment Routing implementation. We experimentally shown that this solution is compatible with existing TCP-based applications and does not require any change to the TCP stack. It can thus be deployed by network operators in access routers or be included in hypervisors in datacenters. Our third contribution is an efficient algorithm that computes disjoint segmentable paths in large networks. Its evaluation in real and inferred networks shows that it finds disjoint paths that have close latencies and do not require a large number of segments to be realised.

Acknowledgements

This work was partially supported by the ARC-SDN project. We want to thank Bui Quoc Trung for his help with the integer programming model.

7. REFERENCES

[1] J. Abley, P. Savola, and G. Neville-Neil. Deprecation of Type 0 Routing Headers in IPv6. RFC 5095 (Proposed Standard), Dec. 2007.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[3] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[4] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[5] A. Dixit, P. Prakash, Y. Hu, and R. Kompella. On the impact of packet spraying in data center networks. In *ACM INFOCOM*, pages 2130–2138. ACM, 2013.

[6] D. A. Dunn, W. D. Grover, and M. H. MacGregor. Comparison of k-shortest paths and maximum flow routing for network facility restoration. *IEEE Journal on Selected Areas in Communications*, 12(1):88–99, 1994.

[7] C. Filsfils et al. Segment Routing Architecture. Internet draft, draft-filsfils-spring-segment-routing-00, work in progress, Apr. 2014.

[8] C. Filsfils et al. Segment Routing Use Cases. Internet draft, draft-filsfils-spring-segment-routing-use-cases-00, work in progress, Mar. 2014.

[9] C. Filsfils et al. Segment Routing with MPLS data plane. Internet draft, draft-filsfils-spring-segment-routing-mpls-01, work in progress, Apr. 2014.

[10] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure. Achieving sub-second igp convergence in large ip networks. *SIGCOMM Comput. Commun. Rev.*, 35(3):35–44, July 2005.

[11] W. Grover. *Mesh-based Survivable Networks: Options and Strategies for Optical, MPLS, SONET, and ATM Networking*. Prentice Hall PTR, 2004.

[12] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[13] H. Kaur, S. Kalyanaraman, A. Weiss, S. Kanwar, and A. Gandhi. BANANAS: an evolutionary framework for explicit and multipath routing in the internet. In *ACM SIGCOMM*, pages 277–288. ACM, 2003.

[14] W. Kumari, J. Jaeggli, and R. Bonica. Operational Issues Associated With Long IPv6 Header Chains. Internet draft, draft-wkumari-long-headers-02, work in progress, Oct. 2013.

[15] D. Lebrun. Supporting ipv6 segment routing in the linux kernel. `http://www.segment-routing.org/sr6-doc.pdf`, November 2014.

[16] S. Lu, Y. Gao, F. Ye, P. Liu, O. Gunluk, T. Berman, S. Calo, and T. Abdelzaher. Robust Confidentiality Preserving Data Delivery in Federated Coalition Networks. In *IFIP Networking*, 2014.

[17] S. Previdi et al. IPv6 Segment Routing Header (SRH). Internet draft, draft-previdi-6man-segment-routing-header-03, work in progress, Mar. 2014.

[18] R. Russell and H. Welte. Netfilter architecture. `http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO-3.html`.

[19] S. Savage, T. Anderson, A. Aggarwal, and D. Becker. Detour: informed Internet routing and transport. *Micro, IEEE*, 19(1), January 1999.

[20] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, Feb. 2004.

[21] X. Yang, D. Clark, and A. Berger. NIRA: A New Inter-Domain Routing Architecture. In *ACM Transactions on Networking*, pages 775–788. ACM, 2007.