"Traffic duplication through segmentable disjoint paths"

Aubry, François ; Lebrun, David ; Deville, Yves ; Bonaventure, Olivier

Abstract

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.

Document type : *Communication à un colloque (Conference Paper)*

Traffic duplication through segmentable disjoint paths

François Aubry, David Lebrun, Yves Deville, Olivier Bonaventure
ICTEAM, Université catholique de Louvain

Louvain-La-Neuve – Belgium
Email: 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.

I. Introduction

Various entreprise applications 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 [1]. 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 [2]. 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 [3]. These financial applications are only one example among a long list of applications [4], [5] that require both low latency and reliable delivery. Other examples include telesurgery, distributed simulations, ...

In circuit-switched networks, a classical solution to preserve mission-critical traffic in case of failures or losses is to send redundant information over disjoint paths. Optical networks often support two types of protection schemes : *1:1* and *1+1*. In both cases, two disjoint paths are used between the communicating nodes. With *1:1* protection, data is sent over the primary path and the secondary path is reserved so that data traffic can be quickly switched to the secondary path in case of failure on the primary path. Some deployments allow to send low-priority traffic over the secondary path. With *1+1* protection, data is sent over both paths and the destination can easily switch to the secondary path if the primary fails [6].

ISBN 978-3-901882-68-5

In TCP/IP networks, such techniques are rarely used. Most applications, including the latency sensitive applications mentioned above rely on acknowledgements and retransmissions to deal with losses. The Transmission Control Protocol (TCP) plays a central role in the performance of many latency-sensitive applications [4], [5]. The TCP protocol and its implementations have been heavily tuned over the years [7]. Despite all these optimisations, short TCP flows can be severely affected by packet losses that are not a rare event [8].

In this paper, we show how an IPv6 network can provide *1+1* protection on end-to-end paths. Our solution is composed of two key elements : *(i)* efficient algorithms that compute disjoint paths, *(ii)* the recently proposed IPv6 Segment Routing [9] architecture that we leverage to realise these disjoint paths. Furthermore, we demonstrate the applicability of this solution by evaluating how latency-sensitive TCP applications could benefit from such a service.

This paper is organised as follows. We first describe in section II how the recently proposed Segment Routing architecture can enable an ISP to deploy such a duplication service. In section II-A, we implement a traffic duplicator in the Linux kernel and demonstrate that it can be used by latency-sensitive TCP applications. We explain in section III how disjoint paths can be computed in a network supporting Segment Routing. We then propose in section III-C an efficient algorithm that computes such disjoint paths. We evaluate its performance on several network topologies in section IV. We discuss related work in section V.

II. 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 [10]. 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 traditional IP networks, packets follow the shortest path towards their destination. 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 [10] modifies the control and the data planes to support non-shortest paths. Two variants of the data plane exist : MPLS-based and IPv6-based [9]. In this paper, we focus on the IPv6-based data plane and its modifications to support segment routing [11]. 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**. In short, a node segment forces a node traversal and an adjacency segment forces a link traversal. To understand in more details these two types of segments, let us consider the network shown in figure 1 where all links have the same weight. Consider a packet sent by `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 possible path is to insert in each packet 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 outgoing interface that needs to be traversed. 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 [12].

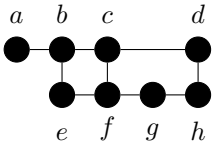

Fig. 1: Simple network

In a traditional IPv6 networks, 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 associated to 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 [9]. This header is a revision of the Type 0 IPv6 Routing Header that has been deprecated a few years ago due to security problems [13]. The SR header contains a list of IPv6 addresses that specifies 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. The SR header also includes a HMAC [14], which did not exist in the Type 0 Routing Header that solves the security problems that caused the deprecation of RH0 [9], [11].

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 `Next Segment` field and uses this address as the destination address of the forwarded packet.

A. Evaluation

We have implemented Segment Routing in the Linux kernel (3.14.x branch) through an IPv6 header extension (SR-IPv6). Our extension [11] implements the current drafts [9], [14] 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 2,500 lines of code and is publicly available from http://www.segment-routing.org. We focus on the key features of our implementation in this section. Additional technical details may be found in [11]. An IPv6 packet containing an SR header is processed as described in algorithm 1.

Algorithm 1 SR header processing

```
1:  if DA = myself (segment endpoint) then
2:    if Segments Left ¿ 0 then
3:      Decrement Segments Left
4:      Update DA with Segment List[Segments Left]
5:      if Segments Left == 0 AND Clean-Up bit set then
6:        Strip SRH
7:      end if
8:    else
9:      Give packet to next PID (application)
10:     End of processing
11:   end if
12: end if
13: Forward the packet out
```

The SR header is processed when the packet is considered for local delivery, after the `PREROUTING` and right after the `INPUT` hook (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 [15].

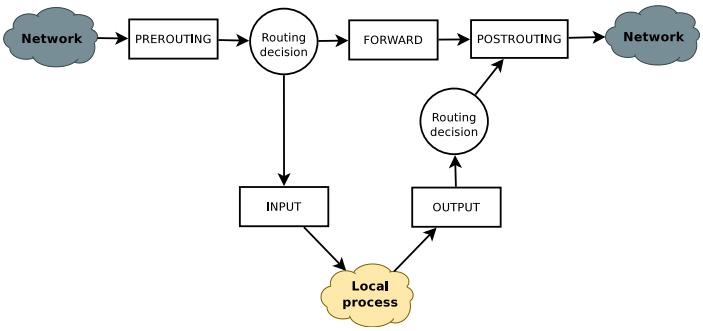

Fig. 2: Linux routing mechanism

Our implementation can also 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 where an indexed set of segments list is associated to each destination prefix. If there is only one segment list, then the kernel uses this list to build the SR header. If several segment lists are associated to a destination prefix, then we support two types of actions. With the SPLIT_RR action, the kernel load-balances the packets in a round-robin fashion among the associated segment lists. With the MIRROR action, the kernel forwards n times the original packet, where n is the number of segment lists available. We use this MIRROR action in our experiments.

B. Impact of duplication on TCP performance

Given its cost in terms of network utilization, traffic duplication should only be used for low-volume and delay sensitive applications. This includes several of the applications discussed in [4]. A detailed analysis of all these applications is outside the scope of this paper and is left for further work. Traffic duplication would clearly be beneficial for applications that use *1+1* protection in optical networks, but it could also be used by traditional data-oriented applications. As an illustration of the capabilities of our SR implementation in the Linux kernel, we experimentally evaluate whether traffic duplication can be used with TCP applications.

Due to space limitations, we focus our evaluation on the impact of two main parameters that usually affect the performance of request-response applications : link delay d and packet loss l. Our evaluation[1] was performed on the topology depicted in figure 3 that we created within the Mininet framework [16]. Our request-response application is a simple web client that interacts with a standard web server, the lighttpd daemon. We requested 100 KBytes files from the HTTP server 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 emulated links and the htb module to shape the bandwidth of links L1 and L2 at 100 Mbps. The bandwidth of all other links is not limited.

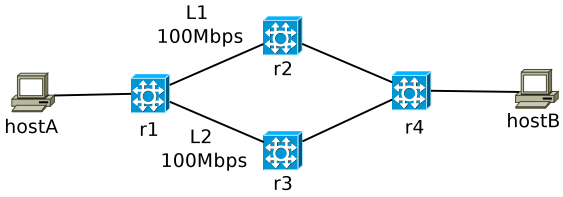

Fig. 3: Test network

We performed 1,000 requests of 100KB between hostA and hostB for each combination in the sets. Figure 4 reports the request-response 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 I and II show the latency measured over 1,000 requests using traffic duplication by setting link L1 to resp. 5 ms and 10 ms and link L2 to resp. 5 + Δd ms and 10 + Δd 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 I: Latency measured over 1,000 requests with link L1 having a delay of 5 ms and link L2 a delay of 5 + Δd ms

	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 II: Latency measured over 1,000 requests with link L1 having a delay of 10 ms and link L2 a delay of 10 + Δd ms

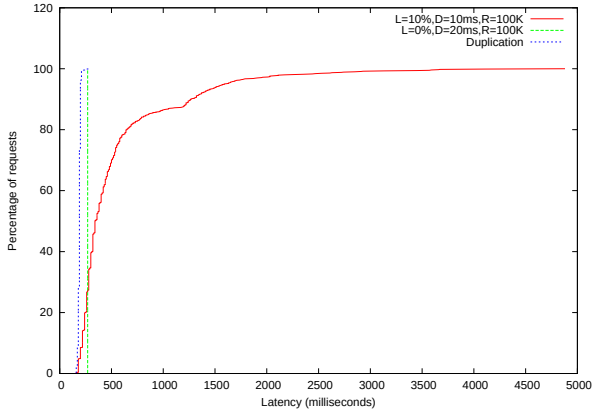

Fig. 4: Percentage of requests having a given latency over disparate links for duplicated and non-duplicated requests

In Figure 5 we show the evolution of the end-to-end request delay over two lossy links L1 and L2 and different packet loss ratios (same loss value for both links at the same time) and for variable Δd between the two links. We sent 1,000 duplicated 100KB requests over the two links for $\Delta d = 0, 10, 40, 90$ and for loss percentages on both link of $1, 2, 3, 4, 5, 10$. When $\Delta d = 0$, our measurements show a very low spread for packet

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

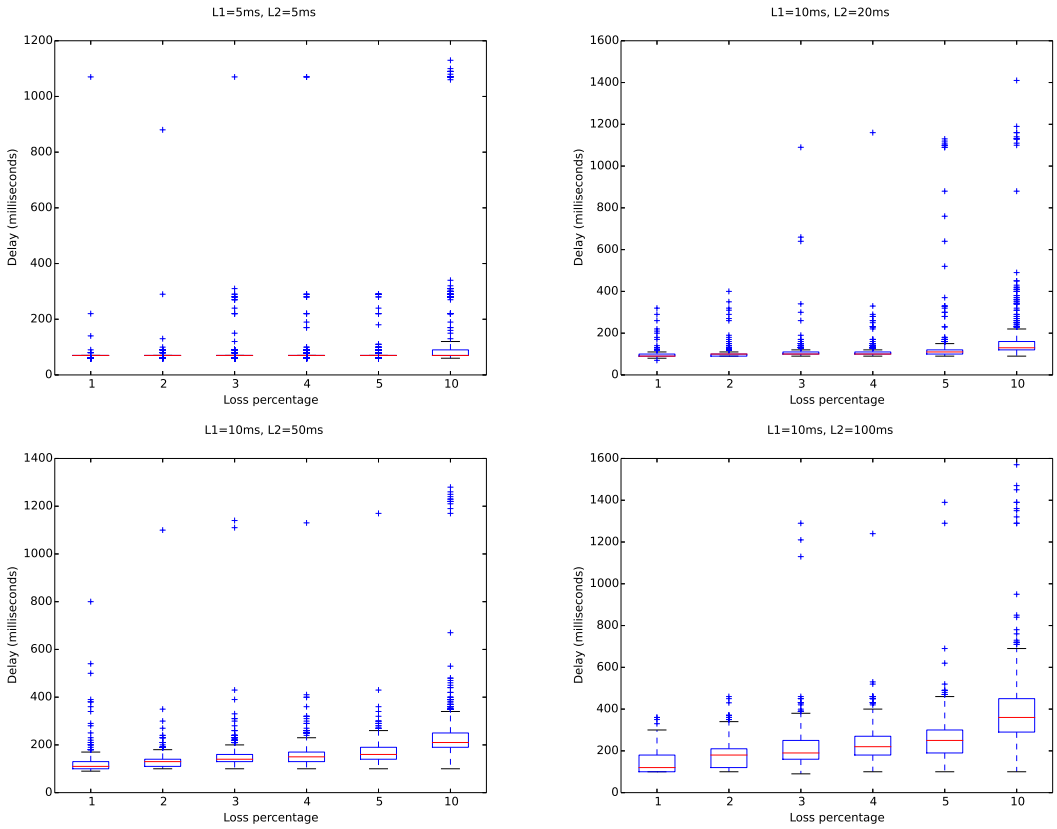

Fig. 5: Upper left plot shows request delay distribution for variable identical loss over two links with L1 set at 5ms delay and L2 set at 5ms. Upper right plot shows the same for L1 at 10ms and L2 at 20ms. Bottom left plot shows for L1 at 10ms and L2 at 50ms. Bottom right plot shows for L1 at 10ms and L2 at 100ms.

loss ratios loss between 1 and 5 percent. The spread starts to grow when the packet loss ratio becomes larger than 10 percent on both links. In practice, it is unlikely that links will be operated at such packet loss ratios. When the two links have different delays ($\Delta d = 10$), our measurements show slightly more spread in the response times, with noticeably more outliers. When the links have different latencies ($\Delta d = 40$ and $\Delta d = 90$), the spread in the measured response times clearly grows and the difference between a loss of 5 percent and a loss of 10 percent is substantial. We also performed additional tests for higher loss percentages $(20, 30, 40)$ but the performances rapidly plummet. The experiment with a packet loss ratio of 40% took nearly one hour to complete. These measurements indicate that traffic duplication can help to reduce the latency of request-response flows over lossy links provided that the two links have similar latencies.

III. 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 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 [17].

A. 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 $c : E \rightarrow \mathbb{R}^+$. 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 6 the path (a, b, c, d) has two minimal segmentations: $\langle (a, b), (b, c, d) \rangle$ and $\langle (a, b, c), (c, d) \rangle$.

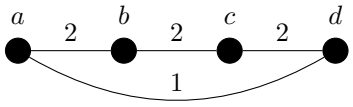

Fig. 6: A graph G with a path (a, b, c, d) that has two minimal segmentations: $\langle (a, b), (b, c, d) \rangle$ and $\langle (a, b, c), (c, d) \rangle$.

B. 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 [18].

Naturally, k-segmentations of paths in G are intimately related to the shortest path 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 relation between segmentable paths and shortest paths dag's is captured by the following lemma.

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 $x \in V$, $(x_1, x_2) \notin \mathcal{D}_x$.

The next lemma is a simple consequence of the fact that a subpath of a shortest path is also a shortest path. In conjonction with Lemma 3.1 it helps us decide whether a path is unsegmentable.

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

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.

C. 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 $k_2 = k_1$ or (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 7 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. 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.

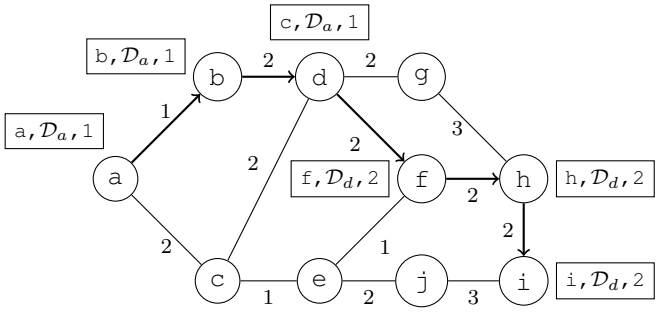

Fig. 7: Example for $\overline{G}$

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 $O(nmK)$.

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 $v \in V$.

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
```

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 $(x_1, \mathcal{D}_{x_1}, 1)$, $(x_2, \mathcal{D}_{r_2}, k_2)$, $(x_3, \mathcal{D}_{r_3}, k_3)$, ..., $(x_h, \mathcal{D}_{r_h}, k)$ 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

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

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 $(x_1, \mathcal{D}_{x_1}, 1)$, $(x_2, \mathcal{D}_{r_2}, k_2)$, $(x_3, \mathcal{D}_{r_3}, k_3)$, ..., $(x_h, \mathcal{D}_{r_h}, k)$. 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. From this we can easily build a k-segmentation of p. ■

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.

Since $\overline{G}$ is acyclic shortest paths can be computed in $O(\overline{m} + \overline{n})$ using Dynamic Programming [18]. 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.

D. Computing disjoint K-segmentable paths with low latencies

In the literature, there are two common ways of computing disjoing paths: (i) iteratively calling a path finding algorithm, such as Dijkstra, to find a path and remove its edges or (ii) using a network flow algorithm [19]. The first approach is subject to finding sub-optimal number of paths since removing a path can take away paths of the optimal solution. In contrast, the network flow approach is guaranteed to be optimal in the sense that it will find the maximum number of disjoint path possible. In our setting we want our path set to minimize the difference of latency between the best path found and the

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

$(d, \pi) \leftarrow$ DAG-SP$(\overline{G}, (s, s, 1))$
$(t, \mathcal{D}_{r^*}, k^*) \leftarrow \arg\min_{r,k}\{(t, \mathcal{D}_r, k) \mid d(t, \mathcal{D}_r, k) < \infty\}$
if $(t, \mathcal{D}_{r^*}, k^*) = nil$ **then**
  return nil
end if
$path \leftarrow \emptyset$
$seg \leftarrow \emptyset$
$cur \leftarrow (t, r^*, k^*)$
while $cur \neq nil$ **do**
  $path \leftarrow path \cup \{cur.vertex\}$
  if $\pi[cur] \neq nil$ ***and*** $\pi[cur].k \neq cur.k$ **then**
    $seg \leftarrow seg \cup \{\pi[cur].vertex\}$
  end if
  $cur \leftarrow \pi[cur]$
end while
return $(path, seg)$

worst path found. With this additional constraint the problem becomes NP-hard [20]. Computing an exact solution is thus unrealistic. Therefore we propose a heuristic solution based on the successive shortest paths approach. We show in the evaluation that the results we obtain are close to optimal with respect to the number of paths found.

We use algorithm 3 iteratively to find a set of edge disjoint K-segmentable paths. 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.

Algorithm 3 requires a latency function l as input. One possibility is to use the propagation delay wich can be inferred from traceroute data. Since our algorithm forbids path with more than K-segments we give a priority to the segmentation constraint. Then among all K-segmentable path we try to optimize the path latencies.

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 $O(nmK + n^2K)$.

IV. 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 [21]. 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 III 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 III: 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 8 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.

As we shown in Section II-A, 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 9 shows for each Δd what is the percentage of pairs such that the difference between their worst path and their best path is

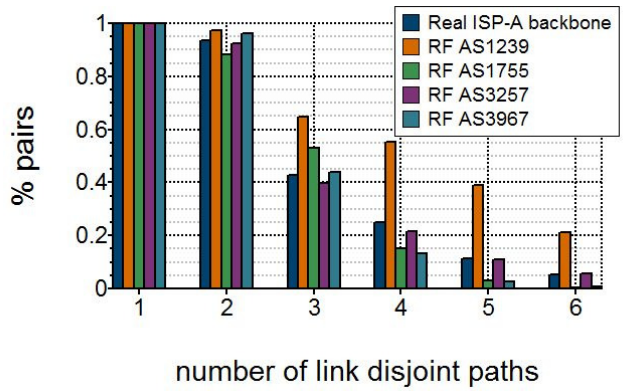

Fig. 8: For $P = 1, \ldots, 6$ the figure shows the percentage of pairs in each graph for which we can find at least P disjoint paths.

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.

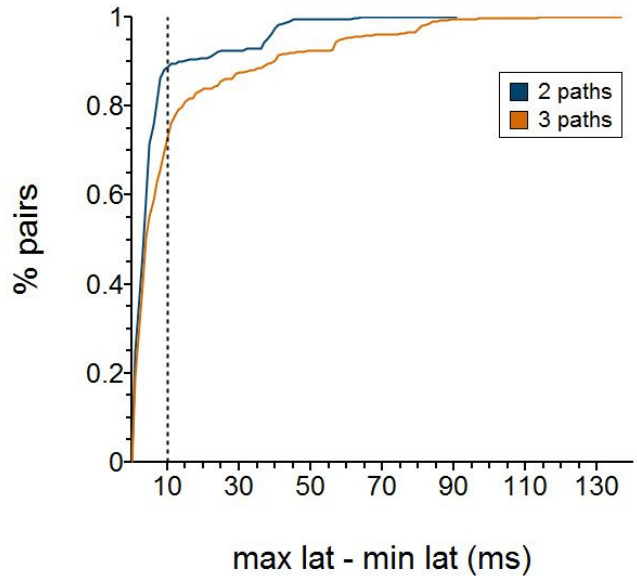

Fig. 9: For $P = 2, 3$ and Δd we show the percentage of pairs that have a difference of latency between the IGP path and the worst path in the path set found by our algorithm at most Δd.

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 [22].

V. Related work

The *Low latency via Replication* proposal [4] is close to our work. Vulimiri et al. propose replicate packets across diverse resources and evaluate how replication affects response under load. However, their approach differs in the sense that they use replication to query different systems such as DNS, while our solution uses multiple disjoint paths.

Different source routing techniques have been proposed in the literature, e.g. [23], [24], [25], [26]. Savage *et al.* propose Detours in [23]. This overlay technique allows to exploit paths that are not used normally. This technique is mainly targeted at interdomain scenarios while ours is targeted at networks managed by a single company. Kaur *et al.* propose in [24] the BANANAS framework that expresses paths as a short hash (PathID) of a sequence of globally known identifiers that have global significance. They show that BANANAS allows to introduce explicit routing and multipath capabilities within existing routing protocols. Their framework moves control-plane complexity and state overheads to network edges. Our Segment Routing implementation is similar in the sense that the segment endpoints do not need to maintain states and can forward packets solely thanks to the SR header. Dixit *et al.* show in [26] the benefits of random packet spraying across multiple paths in datacenter networks. Their solution modifies the ECMP forwarding on the routers to spread the load. Our approach is more costly from a bandwidth utilisation viewpoint since packets are replicated.

Another type of approach is to rely on Forward Error Correction to protect reliable services from the impact of losses. Such techniques have been used in a variety of networks including ATM [27], wireless [28], multicast [29] and inter-datacenter networks [30]. The closest to our work is Balakrishnan *et al.* who propose in [30] a Forward Error Correction (FEC) mechanism to recover from bursty losses. They propose to install this FEC mechanism on wide-area links that interconnect datacenters. Compared to replication, a FEC approach has the benefit of a lower bandwidth consumption at the expense of a higher CPU load and possibly a higher latency.

In the literature there are essentially two approaches to compute disjoint paths between a source and a destination[19]. The most straightforward approach works in the same spirit as ours in the sense that it consists on successively computing shortest paths with the Dijkstra algorithm [31]. Then the paths are deleted until no more path can be found [31]. To our knowledge, these approaches have not 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, Dunn and his colleagues have shown that it is very rare in real networks [19]. 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 [32].

VI. Conclusion

In this paper, we have shown that it is possible to provide an *1+1* protection service by using segmentable disjoint paths between endhosts in an IPv6 network that supports the recently proposed Segment Routing architecture.

Our first contribution is an efficient algorithm that computes such disjoint paths that can be realised by using Segments in large networks. The evaluation of this algorithm in both real and inferred networks shows that it finds disjoint paths that have similar latencies and can be realised by using a small number of segments.

Our second contribution is an open-source implementation of the IPv6 version of Segment Routing inside the Linux kernel. This is the first complete implementation of this new protocol in an operating system kernel. We use it to implement a prototype of the traffic duplication service. Our measurements show that with this service, latency-sensitive TCP applications can better cope with random packet losses. This improved performance comes at the cost of a higher bandwidth utilisation which might not be acceptable in all environments and for all applications. Our further work will be to analyse in more details for which types of applications the proposed traffic duplication is really beneficial.

Acknowledgements

This work was partially supported by the ARC grant 13/18-054 (ARC-SDN) from Communauté française de Belgique. We want to thank Bui Quoc Trung for his help with the integer programming model. We also would like to thank the reviewers for their insightful comments.

References

[1] J. Adler, "Raging bulls: How wall street got addicted to light-speed trading," *Wired*, Aug. 2012.

[2] B. Insider, "Elite traders are getting access to data before everyone else," http://www.businessinsider.com/latency-in-trading-2013-6.

[3] R. Martin, "Wall street's quest to process data at the speed of light," *Information Week*, 2007.

[4] A. Vulimiri, P. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low latency via redundancy," in *CoNEXT '13*, 2013.

[5] B. Briscoe *et al.*, "Reducing internet latency: A survey of techniques and their merits," *Communications Surveys Tutorials, IEEE*, 2014.

[6] J.-P. Vasseur, M. Pickavet, and P. Demeester, *Network recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS*. Elsevier, 2004.

[7] J. Chu, "Tuning tcp parameters for the 21st century," 2009, presented at IETF75.

[8] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: the virtue of gentle aggression," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 159–170.

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

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

[11] D. Lebrun, "Supporting IPv6 Segment Routing in the Linux kernel," Nov. 2014, http://www.segment-routing.org/.

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

[13] J. Abley, P. Savola, and G. Neville-Neil, "Deprecation of Type 0 Routing Headers in IPv6," RFC 5095 (Proposed Standard), Internet Engineering Task Force, Dec. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5095.txt

[14] E. Vyncke, S. Previdi, and D. Lebrun, "IPv6 Segment Routing Security Considerations," February 2015, internet draft, draft-vyncke-6man-segment-routing-security-02, work in progress.

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

[16] N. Handigol *et al.*, "Reproducible network experiments using container-based emulation," in *CoNEXT '12*, 2012, pp. 253–264.

[17] F. Aubry, D. Lebrun, Y. Deville, and O. Bonaventure, "Technical report: Traffic duplication using segmentable disjoint paths," 2014, http://hdl.handle.net/2078.1/152888.

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

[19] 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*, vol. 12, no. 1, pp. 88–99, 1994.

[20] C.-L. Li, S. McCormick, and D. Simchi-Levi, "The complexity of finding two disjoint paths with min-max objective function," *Discrete Applied Mathematics*, vol. 26, no. 1, pp. 105 – 115, 1990.

[21] N. Spring *et al.*, "Measuring isp topologies with rocketfuel," *IEEE/ACM Trans. Netw.*, vol. 12, no. 1, Feb. 2004.

[22] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure, "Achieving sub-second igp convergence in large ip networks," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 3, pp. 35–44, Jul. 2005.

[23] S. Savage, T. Anderson, A. Aggarwal, and D. Becker, "Detour: informed Internet routing and transport," *Micro, IEEE*, vol. 19, no. 1, January 1999.

[24] H. Kaur *et al.*, "BANANAS: an evolutionary framework for explicit and multipath routing in the internet," in *ACM SIGCOMM*, 2003.

[25] X. Yang, D. Clark, and A. Berger, "NIRA: A New Inter-Domain Routing Architecture," in *IEEE/ACM Trans. Networking*, 2007.

[26] A. Dixit, P. Prakash, Y. Hu, and R. Kompella, "On the impact of packet spraying in data center networks," in *IEEE INFOCOM*, 2013, pp. 2130–2138.

[27] E. W. Biersack, "Performance evaluation of forward error correction in an atm environment," *Selected Areas in Communications, IEEE Journal on*, vol. 11, no. 4, pp. 631–640, 1993.

[28] J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher, and J. Barros, "Network coding meets tcp: Theory and implementation," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 490–512, 2011.

[29] C. Perkins, O. Hodson, and V. Hardman, "A survey of packet loss recovery techniques for streaming audio," *Network, IEEE*, vol. 12, no. 5, pp. 40–48, 1998.

[30] M. Balakirshnan *et al.*, "Maelstrom: Transparent error correction for communication between data centers," in *IEEE/ACM Trans. on Networking*, 2011.

[31] W. Grover, *Mesh-based Survivable Networks*. Prentice Hall, 2004.

[32] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993.