Central Control Over Distributed Routing (Extended Version)

http://fibbing.net

Stefano Vissicchio*, Olivier Tilmans*, Laurent Vanbever†, Jennifer Rexford‡

* Université catholique de Louvain, † ETH Zurich, ‡ Princeton University
* name.surname@uclouvain.be, †lvanbever@ethz.ch, ‡jrex@cs.princeton.edu

ABSTRACT

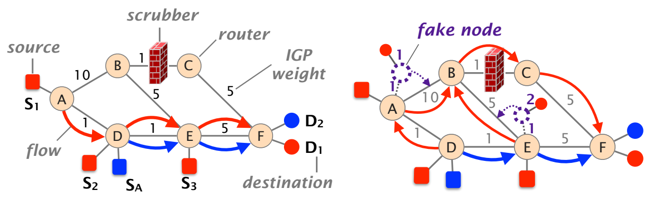
Centralizing routing decisions offers tremendous flexibility, but sacrifices the robustness of distributed protocols. In this paper, we present *Fibbing*, an architecture that achieves both flexibility and robustness through central control over distributed routing. Fibbing introduces fake nodes and links into an underlying link-state routing protocol, so that routers compute their own forwarding tables based on the augmented topology. Fibbing is expressive, and readily supports flexible load balancing, traffic engineering, and backup routes. Based on high-level forwarding requirements, the Fibbing controller computes a compact augmented topology and injects the fake components through standard routing-protocol messages. Fibbing works with any unmodified commercial routers speaking OSPF. Our experiments also show that it can scale to large networks with many forwarding requirements, introduces minimal overhead, and quickly reacts to network and controller failures.

1. INTRODUCTION

Consider a large IP network with hundreds of devices, including the components shown in Fig. 1a. A set of IP addresses (D_1) see a sudden surge of traffic, from multiple entry points (A, D, and E), that congests a part of the network. As a network operator, you suspect a denial-of-service attack (DoS), but cannot know for sure without inspecting the traffic as it could also be a flash crowd. Your goal is therefore to: (i) isolate the flows destined to these IP addresses, (ii) direct them to a scrubber connected between B and C, in order to "clean" them if needed, and (iii) reduce congestion by load-balancing the traffic on unused links, like (B, E).

Performing this routine task is very difficult in traditional networks. First, since the middlebox and the destinations are not adjacent to each other, the configuration of multiple devices needs to change. Also, since intra-domain routing is typically based on shortest path algorithms, modifying the routing configuration is likely to impact many other flows not involved in the attack. In Fig. 1a, any attempt to reroute flows

*S. Vissicchio is a postdoctoral researcher of F.R.S.-FNRS.



(a) Initial topology (b) Augmented topology

Figure 1: Fibbing can steer the initial forwarding paths (see (a)) for D_1 through a scrubber by adding fake nodes and links (see (b)).

to D_1 would also reroute flows to D_2 since they home to the same router. Advertising D_1 from the middlebox would attract the right traffic, but would not necessarily alleviate the congestion, because *all* D_1 traffic would traverse (and congest) path (A, D, E, B), leaving (A, B) unused. Well-known Traffic-Engineering (TE) protocols (*e.g.*, MPLS RSVP-TE [1]) could help. Unfortunately, since D_1 traffic enters the network from multiple points, many tunnels (three, on A, D, and E, in our tiny example) would need to be configured and signaled. This increases both control-plane and data-plane overhead.

Software Defined Networking (SDN) could easily solve the problem as it enables centralized and direct control of the forwarding behavior. However, moving away from distributed routing protocols comes at a cost. Indeed, IGPs like OSPF and IS-IS are scalable (support networks with hundreds of nodes), robust, and quickly react to failures. Building a SDN controller with comparable scalability and reliability is challenging. It must compute and install forwarding rules for all the switches, and respond quickly to topology changes. Even the simple task of updating the switch rule tables can then become a major bottleneck for a central controller managing hundreds of thousands of rules in hundreds of switches. In contrast, distributed routing protocols naturally parallelize this work. For reliability and scalability, a SDN controller should also be replicated and geographically distributed, leading to additional challenges in managing controller state. Finally, the de-

	centralized/SDN OpenFlow [2], PCE [3], SR [4]	*distributed/traditional* IGP [5, 6], RSVP-TE [1]	***hybrid*** **Fibbing**
forwarding paths:			
- *configuration*	simple (declarative & global)	complex (indirect & per-device)	**simple** (declarative & global)
- *manageability*	high (direct control)	low [7, 8] (need for coordination)	**high** (direct control)
- *path installation*	slow (by controller, per-device)	fast (by device, distributed)	**fast** (by device, distributed)
robustness:			
- *network failures*	slow (by controller)	fast (local)	**fast** (local)
- *controller failures*	hard (ad-hoc synch)	native (distributed)	**easy** (synch via IGP)
- *partitions*	hard (uncontrollable devices)	best (distributed)	**best** (fallback on distributed)
routing policies:	highest (any path)	- low for IGP (shortest paths) - highest for RSVP (any path)	**high** (any non-loopy paths)

Table 1: Fibbing combines the advantages of existing control planes, avoiding the main drawbacks.

ployment of SDN as a whole is a major hurdle as many networks have a huge installed base of devices, management tools, and human operators that are not familiar with the technology. As a result, existing SDN deployments are limited in scope, *e.g.*, new deployments of private backbones [8, 9] and software deployments at the network edge [10].

This paper introduces *Fibbing*, a technique that offers *direct control* over the routers' forwarding information base (FIB) by manipulating the input of a distributed routing protocol. Fibbing relies on traditional link-state protocols such as OSPF [5] and IS-IS [6], where routers compute shortest paths over a synchronized view of the topology. Fibbing controls routers by carefully *lying* to them, removing the need to configure them. It coaxes the routers into computing the target forwarding entries by presenting them with a carefully constructed augmented topology that includes fake nodes (providing fake announcements of destination address blocks) and fake links (with fake weights). In essence, Fibbing inverts the routing function: given the forwarding entries (*i.e.*, the desired output) and the routing protocol (*i.e.*, the function), Fibbing automatically computes the routing messages to send to the routers (*i.e.*, the input).

Fibbing can solve the problem in Fig. 1a adding two fake nodes (Fig. 1b), connected to A and E with the depicted weights. Both fake nodes advertise that they can reach D_1 directly. Based on the augmented topology, D starts to use A to reach D_1, as the new cost (3) is lower than the original one (6). A and E also select different paths. Since the fake nodes do not really exist, packets forwarded by A or E actually flow through B. Routers B and C do not change their forwarding decisions.

Table 1 gives an overview of how Fibbing improves flexibility and manageability by adopting a SDN-like approach while keeping the advantages of distributed protocols (*e.g.*, robustness and fast FIB modifications).

Fibbing is expressive. Fibbing can steer flows along *any* set of per-destination loop-free paths. In other words, it can exert full control at a per-destination granularity. For this reason, Fibbing readily supports advanced forwarding applications such as: (a) traffic engineering, (b) load balancing, (c) fast failover, and (d) traffic steering through middleboxes. By relying on destination-based routing protocols, Fibbing does not support finer-grained routing and forwarding policies such as matching on port numbers. Though, those policies can easily be supported via middleboxes.

Fibbing scales and is robust to failures. Lying to routers is powerful but challenging. Indeed, Fibbing must be *fast* in computing augmented topologies to avoid loops and blackholes upon network failures. At the same time, Fibbing must compute *small* augmented topologies since routers have limited resources. Finally, Fibbing must be *reliable* and gracefully handle controller failures. We address all three challenges.

Fibbing differs from previous approaches that rely on routing protocols to program routers. Prior approaches like the Routing Control Platform [11] rely on BGP as a "poor man's" SDN protocol to install a forwarding rule for each destination prefix on each router. In contrast, Fibbing leverages the routing-protocol implementation on the routers. Doing so, Fibbing can adapt the forwarding behavior of many routers at once, while allowing them to compute forwarding-table entries and converge on their own. That is, while the controller computes the routing *input* centrally, the routing *output* is still computed in a distributed fashion.

Fibbing works on existing routers. We implemented a fully-functional Fibbing prototype and used it to program real routers (both Cisco and Juniper). Based on an augmented topology, these routers can install hundreds of thousands of forwarding entries with an average installation time of less than 1ms per entry. This offers much greater scale and faster convergence than is possible with state-of-the-art SDN switches [12, 13], without requiring the deployment of new equipment and per-device actions from the controller. This also means that Fibbing can implement recent SDN proposals, like Google's B4 [8] and Microsoft's SWAN [9]—on top of existing networks.

Our earlier work showed that Fibbing can enforce any set of forwarding DAGs [14]. This paper goes fur-

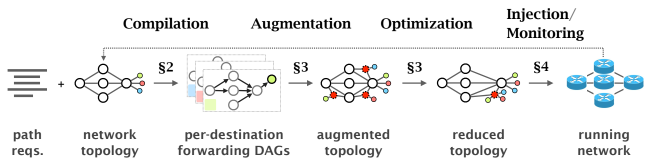


Figure 2: The four-staged Fibbing workflow.

pol	$::=$	$(s_1; \ldots ; s_n)$	Fibbing Policy
s	$::=$	$p \mid b$	Requirement
r	$::=$	p_1 and p_2 $\mid$ p_1 or p_2 $\mid$ p	Path Req.
p	$::=$	Path(n^+)	Path Expr.
n	$::=$	$id \mid * \mid n_1$ and n_2 $\mid$ n_1 or n_2	Node Expr.
b	$::=$	r as backupof $((id_1, id_2)^+)$	Backup Req.

Figure 3: Syntax of Fibbing high-level language.

ther by describing the complete design, implementation, and evaluation of a Fibbing controller managing intra-domain routing. Rather than focusing on specific use cases (like traffic engineering), we describe its support for different higher-level approaches (*e.g.*, [8, 9]). We make the following contributions:

- **Abstraction**: We show how to express and realize high-level forwarding requirements by manipulating a distributed link-state routing protocol (§2).
- **Algorithms**: We propose new, efficient algorithms to compute compact augmented topologies (§3).
- **Implementation**: We describe a complete Fibbing implementation which works with unmodified Cisco and Juniper routers (§4).
- **Evaluation**: We show that our Fibbing controller quickly generates small augmented topologies, inducing minimal load on routers (§5).

2. FLEXIBLE FIBBING

Fibbing workflow proceeds in four consecutive stages based on two inputs: the desired forwarding graphs (one Directed Acyclic Graph, or DAG, per destination) and the IGP topology (Fig. 2). The forwarding DAGs can either be provided directly or expressed indirectly, at a high-level, using a simple path-based language. In the latter case, the **Compilation** (§2) stage starts by compiling the requirements into concrete forwarding DAGs. Then, the **Topology Augmentation** (§3) stage computes an augmented topology that achieves these forwarding DAGs. While computing an augmented topology is fast, the resulting topology can be large. As such, the job of the next **Topology Optimization** (§3) stage is to reduce the augmented topology while preserving the forwarding paths. Finally, the **Injection & Monitoring** (§4) stage turns fake information into actual "lies" that the controller injects into the network.

In this section, we present the high-level language and compilation process (§2.1), and show that Fibbing can express a wide range of forwarding behaviors (§2.2).

2.1 Fibbing high-level language

Fibbing language (Fig. 3) provides operators with a succinct and easy way to specify their forwarding requirements. A Fibbing policy is a collection of *requirements*, naturally expressed as regular expressions on paths. Each requirement is either a *path requirement* which specifies how traffic should flow to a given destination, or a *backup requirement* which specifies how traffic should flow if the IGP topology changes. Each path requirement is recursively defined as a composition of path requirements through logical AND and OR. Operators can express load-balancing requirements using a conjunction of n requirements. Similarly, they can ensure that traffic to a specific destination will take one of n paths (*e.g.*, containing a firewall) using disjunction. Path requirements are composed of a sequence of node requirements. A node requirement is either a node (router) identifier or the wildcard *, representing any sequence of nodes. Like path requirements, node requirements can be combined using logical AND and OR. Whenever no path requirement is stated, the original IGP paths should be used. This way, operators do not have to express all the unmodified paths, only deviations from the IGP shortest paths.

The following example illustrates the main features of the language. It states that traffic between E and D_1 should be load-balanced on two paths, traffic between A and D_2 should cross B or C and that traffic from F to D_3 should be rerouted via G if the link (F, H) fails.

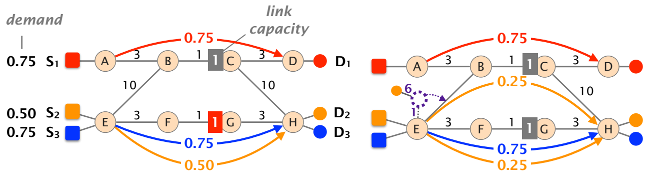
```
(  (E,C,D1) and (E,G,D1);
   ((A,*,B,*,D2) or (A,*,C,*,D2));
   (F,G,*,D3) as backupof ((F,H))); )
```

Fibbing policies are compiled into per-destination forwarding DAGs by finding convenient network paths (if any). Compilation works in two consecutive stages. First, the compiler expands any requirement with wildcards into paths. This step can be computationally expensive as, in general, a network can have a number of paths exponential in the number of nodes. While this is unlikely, especially for networks designed according to best practices, we bounded the number of paths that can be expanded out of a single requirement. We only expand again if no solution is found with the current set of paths. Once all requirements are expanded, the compiler groups them by destination and computes the Disjunctive Normal Form (DNF) of each requirement. To finally produce a forwarding DAG, the compiler iterates over the disjunction of path requirements and checks whether the resulting graph is loop-free.

2.2 Fibbing expressiveness

Beyond steering traffic along a given path (§1), we now show that Fibbing can also: (i) balance load over multiple paths and; (ii) provision backup paths.

Fibbing can forward any flow on any *set* of paths. Fibbing can load-balance traffic over multiple paths to maximize throughput, minimize response time, or increase reliability. For example, consider the network in Fig. 4a where three sources S_1, S_2, and S_3 send traffic to three corresponding destinations. Demands and link capacities are such that link (F, G) is congested. One way to alleviate congestion is to split traffic destined to D_2 over the top (via (B, C)) and bottom (via (F, G)) paths. Load-balancing traffic coming from E on multiple paths is possible under conventional link-state routing (*e.g.*, by re-weighting links (F, G) to 15. However, this would force the traffic from S_2 *and* S_3 to spread over both paths, creating congestion. More generally, it is impossible to route the traffic destined to D_2 and D_3 on different links under conventional link-state routing.



(a) Initial topology (b) Augmented topology

Figure 4: Fibbing supports multi-path forwarding. Here, it avoids the initial congestion (see (a)) by load-balancing traffic for D_2 (see (b)).

This simple requirement can easily be expressed as:

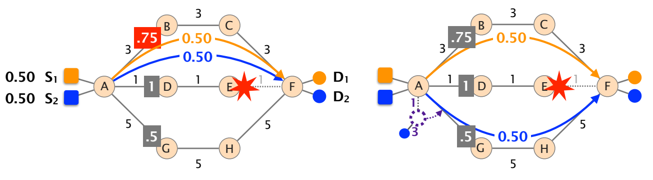
$((S_2, E, B, C, H, D_2)$ **and** $(S_2, E, F, G, H, D_2))$

Fig. 4b shows the augmented topology which realizes this requirement. A fake node announcing D_1 (with a weight of 6) is inserted between E and B. After introducing this node, E has two shortest paths (of cost 7) to reach D_2 and, hence, splits D_2 traffic over B and F. In this example, Fibbing enables maximum network efficiency as each link is used to its full capacity.

Fibbing can provision backup paths for any flow. Fibbing can provision backup paths to prevent congestion or increased delays after link and node failures. As an illustration, consider the network in Figure 5a. The failure of link (E, F) leads to congestion since traffic flows for D_1 and D_2 are both rerouted to the same path via link (A, B). To prevent congestion, traffic destined to D_1 and D_2 should be split over the two remaining disjoint paths but only upon failure on the path (A, D, E, F). This is another example of a requirement that is impossible to achieve with link-state routing, and would require significant control-plane overhead in MPLS. In contrast, it is easily done with Fibbing. Backup paths can be specified in our language as:

$(A, B, *, D_1)$ **and** $(A, G, *, D_2)$

as backupof $((A, D)$ **or** (D, E) **or** $(E, F))$



(a) Initial topology (b) Augmented topology

Figure 5: Fibbing can provision backup paths. Here, possible congestion upon a link failure (see (a)) is avoided by adding a fake node (see (b)).

Fig. 5b shows the corresponding augmented topology, which has a single fake node advertising D_2. The weights are set to prevent A from using the fake node to reach D_2 *unless* a failure occurs along the path (A, D, E, F). While successful for this example, Fibbing cannot satisfy all possible requirements for backup paths (§3.3).

3. AUGMENTING TOPOLOGY

In this section, we detail the augmentation problem (§3.1), and we show how the Fibbing controller quickly computes small augmented topologies from a set of forwarding DAGs. We rely on a *divide-and-conquer* approach based on three consecutive steps.

1) Topology initialization (§3.2): We modify the initial weights in the link-state protocol (if necessary), to guarantee that any set of forwarding DAGs can be enforced by Fibbing. If needed, this operation has to be done only once, when Fibbing is first deployed.

2) Per-destination augmentation (§3.3): Starting from an initialized topology, we compute a suitable augmentation, individually for every destination of an input forwarding DAG. We designed two algorithms for this step, achieving different trade-offs between computation time and augmentation size. The fastest one, **Simple**, can compute augmented topologies *within milliseconds*, and works by injecting a dedicated fake node for every router that changes its next-hop. The relatively slower one, **Merger**, reduces the augmentation size by re-using the same fake nodes to program multiple routers. Simple and Merger are suited for different goals. The speed of Simple is useful for quick failure reaction. In contrast, Merger can be run in background to progressively re-optimize the augmented topology. We evaluate the trade-offs achieved by each algorithm in §5.

3) Optimization across destinations (§3.4): We merge the augmentations obtained in the per-destination augmentation step to further reduce the number of fake nodes and edges. Namely, whenever safe, we replace multiple fake nodes announcing different destinations with a single fake node which either announces all the destinations or creates a new path (a shortcut) between routers in the augmented topology.

3.1 The Topology Augmentation Problem

We start the description of the topology augmentation algorithms by precisely defining the basic concepts on which they rely and the problem that they solve.

Fake nodes scoping. A Fibbing controller can generate both *locally-scoped* lies (targeted to a single router) and *globally-scoped* lies (targeted to all routers). Locally-scoped lies are useful as they enable local actions on one router without creating side effects on other routers. Globally-scoped lies affect the entire network. Hence, if carefully computed, they can reduce the size of the augmented topology. All of our previous examples used globally-scoped lies. We detail how to implement both kinds of lies in the current OSPF in §4.

Fake edges to forwarding next-hop mapping function. Fibbing can modify the routing path computed by any IGP router r. In particular, it can augment the IGP topology so that r's shortest path is no longer the one in the original topology but includes some fake sub-paths. Throughout the paper, we assume that a fake edge in the shortest path from any router r to any destination d corresponds to the ability to force the next-hop of r for d to be any of its neighbors. In the example in Fig. 1, for instance, the fake edge between the A and its adjacent fake node translates into A forwarding traffic to B. We discuss in §4 how to achieve this ability in the current OSPF protocol, as well as in future IGPs.

Topology augmentation problem. Since we assume an arbitrary mapping between fake edges and forwarding next-hops, the topology augmentation problem is defined as follows: Given an initial topology G and a set of forwarding DAGs, compute an augmented topology $G' \supset G$ such that for each path $(u, v, \dots, d)$ in the forwarding DAG for d, the next-hop of u in one of its shortest paths for d in G' is either v or a fake node.

3.2 Topology Initialization

In the topology initialization, we scale the link weights of the original IGP topology G to guarantee arbitrary per-destination control through Fibbing and help reduce the size of topology augmentations. In particular, we proportionally increase link weights (multiplying them by a constant factor) if they are too low in G. Moreover, we set very high announcement cost for any destination, at least equal to the length of the longest path in G times the maximum link weight.

Topology initialization enables full Fibbing expressivity. Indeed, it makes the IGP topology *Fibbing compliant*, which provably avoids cases in which a forwarding DAG cannot be implemented by Fibbing (see Appendix A). We say that a topology is Fibbing compliant if for every destination d, the cost of the shortest path from every router (including the ones announcing d) to d exceeds 2. In Fibbing compliant topologies, for any router r and destination d the controller can always compute a fake path P such that (i) P is shorter than the original shortest path from r to d; and (ii) P is longer than the original shortest path from any other router $v \neq r$ to d. As proved by the following theorem, this implies the ability of Fibbing to forward flows for the same destination on any set of loop-free paths.

THEOREM 1. *Any set of per-destination forwarding DAGs can always be enforced by augmenting a Fibbing-compliant topology even only with globally-scoped lies.*

PROOF. We prove the statement by showing a simple topology augmentation procedure. Let G be the initial topology. For every forwarding DAG with destination d, we add for each node r in the network a fake node f_r announcing d. This generates a new fake path (r, f_r, d) in the augmented topology. We set the total cost of this newly added fake path to 2. Since G is Fibbing compliant, then the cost of the shortest path from r to d in G is greater than 2. Hence, the shortest path of every node r in the augmented topology will be (r, f_r, d). The forwarding DAG is then implemented by mapping the fake link on the right physical link. $\square$

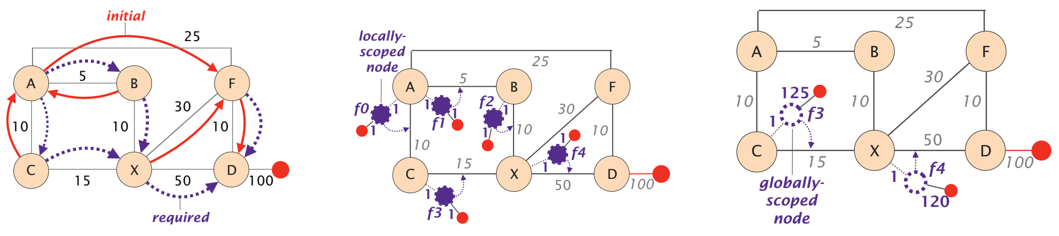
Note that Theorem 1 applies to destinations in the augmented topology. Those destinations do not need to match the destination prefixes announced in the original IGP. Hence, Fibbing allows to control flows for de-aggregation of the original IGP destinations (up to IP address granularity) or even non-overlapping prefixes.

Topology initialization is non-intrusive. Since it is based on the adaptation of a few configurable parameters, our initialization procedure can be applied to any link-state routing configuration, and *preserves the original forwarding paths.* It can be carried out in a running network, using known lossless reconfiguration techniques [15]. Moreover, it is strictly needed no more than once in the network lifetime. Indeed, since Fibbing compliance does not depend on the routing requirements or the presence of specific links, any topology remains Fibbing compliant independently of new requirements or the failures of nodes or links. Finally, topologies growing in size can be easily kept Fibbing compliant by ensuring that the new destinations are announced with high costs and the new links have weights consistent with pre-existing ones.

3.3 Per-destination augmentation

We now describe Simple and Merger. We use Figure 6 to illustrate the difference of the two algorithms.

3.3.1 Simple



(a) Requirements (b) Simple augmentation (c) Merger augmentation

Figure 6: Outcome of our per-destination augmentation algorithms. Simple produces larger topologies (with locally-scoped lies) in a very short time, while Merger reduces the size of the augmentation by relying on globally-scoped lies and a longer computation.

Simple relies solely on locally-scoped lies to avoid having to compute any fake path cost. For every destination d and corresponding forwarding DAG D, the algorithm adds fake nodes to each router whose next-hops in the original topology differ from those in D. Precisely, for every router r that changes its next-hop for d, Simple adds a fake node $f_{r,d}$ and a fake link $(r, f_{r,d})$. Node $f_{r,d}$ announces d to r with a locally-scoped lie. We set the total cost of path $(r, f_{r,d}, d)$ to 2. Since the topology is Fibbing compliant, r is ensured to change its shortest path. Also, since the lie is locally-scoped, other routers are not affected by it.

Figure 6b shows the output of Simple for the example of Figure 6a. Nodes A, B, C and X are required to change their respective next-hops. Moreover, A needs to load-balance on B and C. Thus, Simple creates five locally-scoped nodes (two connected to A), all providing fake paths to the destination with a cost of 2.

3.3.2 *Merger*

To reduce the number of fake nodes, Merger relies on globally-scoped lies that can change the forwarding behavior of multiple routers at once. When applied to Figure 6a, Merger creates only two fakes nodes (see Fig. 6c) to change the next-hops of A, B, C and X, instead of the five used by Simple. The added fake nodes create load-balancing on A (cost 136) via B and C, as required.

Merger performs the topology augmentation for any destination d in two phases. First, it adds an excessive number of fake nodes, and computes the lower and upper bounds for their respective cost. Second, it merges fake nodes whenever possible, based on the value of the computed bounds. We now provide an intuitive description of those two phases. Additional details about them and Merger correctness proofs are reported in Appendix B.

Step 1. Fake bounds computation. Merger starts by adding fake nodes to every router r that is required to change the next-hop (according to the input forwarding DAG) for d. In this case, one new fake node $f_{r,d}$ is connected to r for every new r's next-hop in the input forwarding DAGs. However, to enable merging of globally-visible fake nodes, Merger calculates lower and upper bounds for every newly-added fake path $(r, f_{r,d}, d)$. Since the initial positioning of fake nodes is as in Simple, we illustrate bound computations referring to Figure 6b.

The *upper bound* $ub(f_{r,d})$ represents the maximum value of the fake path cost that changes r's shortest path to $(r, f_{r,d}, d)$. It is easy to compute by statically considering the non-augmented topology G. Indeed, it is equal to $dist(r, d, G) - 1$, where $dist(u, v, G)$ is the cost of the shortest path from u to v in G.

The *lower bound* $lb(f_{r,d})$ represents the minimal value of the fake path cost that does not change the shortest path of any real node different from r. To compute it, we divide nodes in two sets, depending on whether the input forwarding DAG prescribes to change their respective next-hops or not.

For the next-hop preserving nodes whose shortest path does not traverse r in the input topology, we impose that their original shortest path is not modified by $f_{r,d}$. For example, when computing $lb(f_0)$ in Figure 6b, we ensure that f_0 does not change the shortest path of F, by constraining $lb(f_0) + 25 > dist(F, d, G) = 110$, that is, $lb(f_0) = 86$. More generally, for every fake node $f_{r,d}$ and every next-hop preserving neighbor n of r, we impose that $lb(f_{r,d}) > dist(n, d, G) - dist(n, r, G)$.

For next-hop changing nodes (connected to other fake nodes), the final value of their shortest paths is not known in advance, but is determined by the augmentation itself. That is, the lower bound of any fake node generally depends on the lower bound of other fake nodes. For example, in Figure 6b, $f4$ changes the shortest path of X only if $lb(f4) < dist(X, C, G) + lb(f3)$. To avoid that real nodes pass through fake nodes not directly connected to them, Merger runs a *lower bound propagation procedure*. This procedure takes as input the lower bounds initialized with values from next-hop preserving nodes. It then fixes one lower bound at the time, following a specific order and adjusting the others to be consistent with the fixed one. This order guarantees that each lower bound must be considered only once. Sometimes, lower bounds cannot be made consistent. Indeed, Theorem 1 does not provide guarantees if fake nodes are connected only to next-hop changing nodes. We solve these cases by using locally-visible lie.

Step 2. Fake nodes merging. In this step, Merger tries to merge fake nodes together. More precisely, it iterates over every simple path from a source to d in the input forwarding DAG. For each of those paths, it merges pairs of fake nodes whenever safe.

To assess when it is safe to merge a fake nodes f' into f'', Merger sequentially performs three checks. We illustrate these checks by considering the required path (A, B, X, D) and the merge of f_1 into f_2 in Figure 6b.

First, Merger assesses whether the IGP shortest paths are compliant with the considered source-sink path in the DAG. In our example, it verifies that the shortest path from A and B (*i.e.*, (A, B)) is a sub-path of the required (A, B, X, D). If A had predecessors in the required path not connected to a fake node, this check would have been repeated for those predecessors as well.

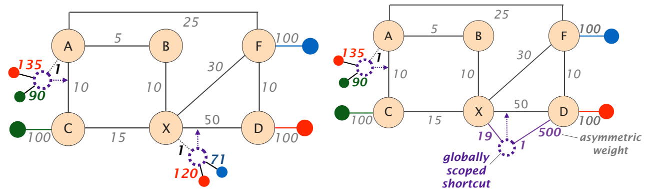
Second, Merger checks the possibility to use f'' as part of the shortest path of the real node connected to f' without changing the current next-hops of any node. To this end, the algorithm assess the existence of feasible post-merging bounds for f''. More precisely, it re-computes the modified lower bound of f'' as the minimum value (if any) that forces the new shortest path of the real node connected to f' and its next-hop preserving predecessors in the required path through f'', without affecting nodes previously not crossing f''. In our example, the lower bound of f_2 is modified to exclude the constraint of not changing A's next-hop, hence it is decreased to $lb(f_2) = 81$ (it was greater before, for A's next-hop to be f_1). The upper bound of f'' is also modified to ensure that the real node connected to f' and its next-hop preserving predecessors use the fake path via f''. In our example, $ub(f_2)$ is modified to 129, as the cost of the original shortest path from A to the destination is 135 and those between A and B is 5.

Third, Merger simulates the merge to assess whether all bounds can be consistently adjusted network-wide, given that the merging f' into f'' would change the bounds of f'' and remove one fake node. To this end, we re-run the lower bound propagation procedure, devoting special attention to fake nodes used for load-balancing. In our example, for instance, we constrain the lower bound of f_0 to be equal to the cost of path (A, B, f_2, d), meant to be used by A after the merging.

If all the three checks pass, then Merger actually perform the merge, by removing f' and updating the bounds of all other fake nodes (including f'') according to the values computed during the last check.

3.3.3 *Dealing with backup requirements*

Backup requirements can be specified by providing additional sets of (tagged) forwarding DAGs to our algorithms. Let G' be an augmented topology computed to accommodate primary requirements. To deal with backup ones, slight variants of Simple and Merger can be run after the computation of G'. The main modification of Simple consists in setting the cost of the fake paths added for backup requirements to 3 instead of 2. In contrast, backup requirements are supported in Merger by imposing that lower bounds are always greater than the cost of the shortest path in G'.



(a) Fake destination merging. (b) Fake shortcuts creation.

Figure 7: Cross-destination optimizations.

Contrary to primary requirements (see Theorem 1), Fibbing may not enforce backup requirements, even in a Fibbing-compliant topology (see Appendix C). Indeed, if the cost of the original shortest path of a node r is equal to a fake one (used for a primary requirement) in G', then backup paths different from the original shortest path cannot be implemented on r. In those cases, we notify the operator on the impossibility to implement the given backup paths.

3.4 Cross-Destination Optimization

Fake nodes computed on a per-destination basis may be redundant. We reduce such redundancy in two ways, namely, by (i) merging fake nodes connected to the same real node, and (ii) replacing fake destination announcements with fake paths connecting real nodes.

Cross-destination merging. After per-destination augmentations, two fake nodes f_1 and f_2 announcing different destinations d_1 and d_2 can be connected to the same node r and used to force traffic to the same real link (r, n). Those fake nodes can always be merged. Indeed, we can replace f_1 and f_2 with a new fake node f' such that (i) $cost(r, f') = min\{cost(r, f_1), cost(r, f_2)\}$, and (ii) f' announces both d_1 and d_2, with $cost(f', d_i) = cost(r, f_i, d_i) - cost(x, f')$ for $i = 1, 2$. For example, assume that in the network in Figure 6 additional destinations are attached to C and F as in Figure 7a. The result of the cross-destination merging is shown in Figure 7a, where both A and X have a single fake neighbor announcing multiple destinations (rather than multiple fake neighbors each announcing a single destination). This reduces the number of fake nodes from 4 to 2.

Creating shortcuts. One of the most appealing features (unmatched by competitor solutions) of Fibbing is that a single lie can change the paths for multiple destinations. To this end, we need however to replace fake destination announcements with fake paths connecting real nodes together, *i.e.*, *fake shortcuts*.

Currently, we use fake shortcuts only if a real link is never traversed in different directions. Consider, for example, the link between X and D in Figure 7a. It is traversed from X to D for the destinations attached to D and F, and never from D to X. In those cases, we try to transform X's fake neighbor into a fake shortcut, as in Figure 7b. Let u and v be the two real nodes at the endpoints of the shortcut. First, we check if a shortcut cost c exists such that all the shortest paths[1] are kept the same with and without the shortcut. If this condition is met, given a fake node f used to enforce the subpath $(u, \ldots, v)$ in the input forwarding DAGs, all fake destinations announced by f are replaced by a fake shortcut (u, f, v) and the cost of (u, f, v) is set to c. Figure 7b illustrates that we found such a value $c = 20$ for X in our example. Also, we use asymmetric weights to prevent the fake shortcut from being traversed in the opposite direction. To this end, the cost of path (v, f, u) is set to a very high value, *e.g.*, by setting a high weight of the directed link (v, f). In Figure 7b, we indeed set the weight of the link from D to the fake node in the shortcut to 500.

4. IMPLEMENTATION

We built a complete prototype of Fibbing in Python (algorithmic part) and C (interaction with OSPF) by extending Quagga [16]. Fibbing code base spans over 2300 (resp. 400) lines of Python (resp. C) code. It is available at `http://www.fibbing.net`. In this section, we present our prototype (§4.1), describe how Fibbing works with current OSPF routers (§4.2), and propose two small modifications to link-state protocols that would make Fibbing even more efficient (§4.3). Finally, we describe how to ensure controller reliability (§4.4).

4.1 Fibbing Controller

Our prototype consists of three main components:

Fake topology generator applies (i) the compilation algorithms (§2) to turn forwarding requirements into forwarding DAGs and (ii) the augmentation algorithms (§3) to convert the forwarding DAGs into fake nodes and links. The topology generator uses a JSON interface to register for update events produced by the event manager. Upon network updates, the generator automatically recomputes the augmented topology either using Simple to ensure fast convergence, or Merger to reduce the size of the topology. To ensure fast convergence *and* a small augmented topology, the generator also pre-computes augmentations with Merger, *e.g.*, those needed for any single link failure, and stores them in a *deltas database*.

[1] By definition of the shortest path, we only need to check the paths from v and from u neighbors to each destination.

Link-state translator interacts with the routers by establishing routing adjacencies to inject lies and track topology changes. Thanks to the flooding mechanism used by link-state protocols, a single adjacency is sufficient to send and receive all routing messages to/from all routers. Though, maintaining several adjacencies is useful for reliability. In this case, the translator simply injects the lies via all adjacencies. While this slightly increase the flooding load, doing so does *not* impact the routers memory as each message has a unique identifier and routers only maintain one copy per ID in memory.

Event manager maintains an update-to-date view of the network topology by (i) parsing the routing messages collected by the translator and (ii) constructing a network graph for the topology generator. The event manager checks whether each new event (*e.g.*, a node failure or addition) affects any of the forwarding requirements. If so, it first checks the deltas database for a pre-computed lie, and otherwise notifies the topology generator to request a new augmented topology.

4.2 Fibbing with Unmodified OSPF

Our Fibbing prototype works with unmodified OSPF-speaking routers (tested on Cisco and Juniper). To creates lies, our prototype leverages the *Forwarding Address* (FA) [17] field of OSPF messages. Suppose the controller wants routers to think that destination d is directly attached to the router with IP address y. Then, the controller injects the route for d with a forwarding address of y and the desired cost for the fake edge from y to d. Router y ignores the message, and all other routers compute the cost of the route as the sum of their cost to y plus the cost in the message.

Locally-scoped lies in OSPF. To support locally-scoped lies, we reserve a set of IP addresses to be used as FAs. All those addresses are propagated network-wide in OSPF. However, every router is configured not to install routes to all those addresses. Consequently, only the directly-connected router can reach any of those addresses, and accept routes specifying that address as FA. Both allocation and configuration of FA-associated IP addresses can be done just once in the network lifetime.

Globally-scoped lies in OSPF, and limitations. OSPF readily supports globally-scoped lies by simply propagating OSPF messages with the FA set to an IP address announced in OSPF. However, some subtle constraints hold in an unmodified OSPF network due to how FAs are resolved on the router. Prominently, (i) OSPF fake nodes are actually fake routes to the specified FA, hence both their positioning and the path to be used for reaching them are constrained; and (ii) OSPF routers discard any OSPF message whose FA is one of its own IP address and computes its shortest path according to the topology without the fake node. The combination of these two constraints limit the power of globally-scoped lies in OSPF, making them insufficient to implement all possible forwarding DAG. An example of those cases is reported in Appendix D.

Overcoming OSPF limitations. To support the full expressiveness of Fibbing, our prototype controller uses an OSPF-compliant implementation of Merger with a combination of locally and globally-scoped lies. This implementation uses globally-scoped lies whenever possible, and falls back to locally-scoped lies for any requirements that cannot be met that way.

4.3 Proposed Protocol Enhancements

Other link-state protocols, like IS-IS [6], do not support forwarding addresses, and even the OSPF implementation of lies has limitations. However, minor protocol extensions can enable more flexible Fibbing in future routers. Fully-fledged Fibbing needs for the routing protocol to support two functions: (i) the creation of adjacencies (with fake nodes) on the basis of a received message; and (ii) a third-party next-hop mechanism which allows to specify in a route the forwarding next-hop to be used if that route is selected. Support for these functions can be added to protocol specifications (without impacting current functionalities), and can be deployed through router software updates.

Preliminary discussions with router vendors confirm that these changes are reasonable and could be integrated into current protocol implementations. Moreover, backwards compatibility is easily achieved as legacy routers would simply ignore any Fibbing-specific protocol features. Our algorithms can be modified to account for the fact that the shortest path of a legacy router is never changed by any fake node.

4.4 Controller Replication

As any network component, a Fibbing controller can fail at any time. Reliability can be ensured by running multiple copies the Fibbing controller in parallel and connecting them to different places in the network.

No state needs to be synchronized between the replicas besides the input forwarding requirements (mostly static anyway). Indeed, Fibbing algorithms are deterministic, hence all replicas will always compute exactly the same augmented topology. The only dynamic state maintained by a Fibbing replica is the network graph. This state, however, is implicitly synchronized through the shared topology offered by the underlying IGP: the link-state flooding mechanism keeps the network graph up to date and eventually consistent across all replicas.

The determinism of our algorithms enables all replicas to inject (the same) lies at the same time. However, this would increase the amount of flooded information.

To limit control-plane overhead, our implementation relies on a primary-backup architecture with a single active replica and an inexpensive election process. A pre-defined range of router IDs is reserved for controller replicas. The replica with the lowest router ID across all running ones is the active replica. It is the only one injecting lies, while the others only compute the topology augmentation. When the controller is booted or when an active replica fails, running replicas receive IGP messages on the current network topology. Based on those messages, every replica independently infers the possible presence of other replicas; it also checks whether it is the new active replica by comparing its router ID with the one of the other running replicas.

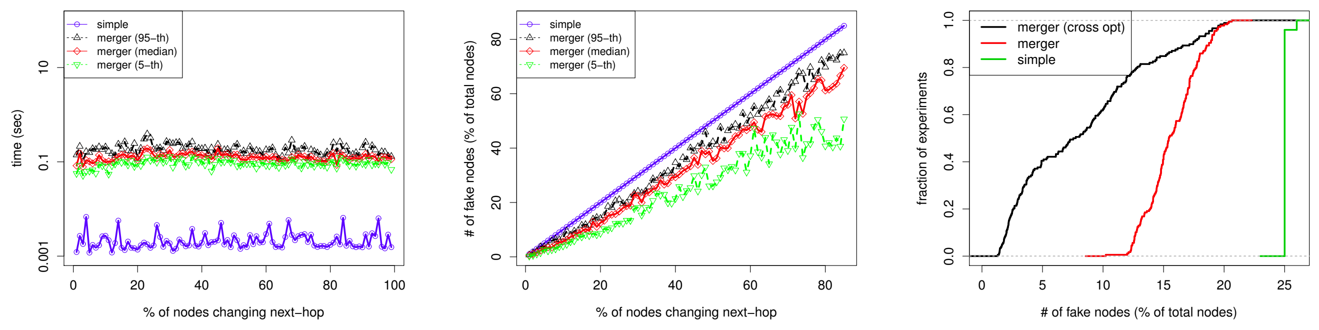
5. EVALUATION

We now evaluate Fibbing along three axis. First, we show that existing routers are perfectly capable of handling the extra load induced by Fibbing (§5.1). We then demonstrate the efficiency of Fibbing's augmentation algorithms in terms of speed and size of the topology (§5.2). Observe that Fibbing behaves as a plain IGP at the network level. Hence, given its negligible impact on single routers and the efficiency of our controller, current ISP networks can be seen as the best large-scale evaluation for Fibbing. We therefore complete the evaluation by illustrating how Fibbing can be used in a realistic case (§5.3).

5.1 Router measurements

By increasing the size of the link-state routing topology, Fibbing could increase the CPU and memory overhead on the routers, or slow down protocol convergence. Our experiments demonstrate that the impact on load and convergence time is negligible. All our measurements were performed using OSPF on a recent Cisco ASR9K running IOS XR v5.2.2 equipped with 12GB of DRAM assigned to the routing engine, as well as on a (7-year-old) Juniper M120 running JunOS v9.2, equipped with 2GB of DRAM. Both routers are representative of typical edge devices (*i.e.*, aggregation routers) found in commercial networks. We draw the same conclusions on both router platforms, and focus on measurements collected on the Cisco device in the following.

Fibbing induces very little CPU and memory overhead on routers. We first measured the memory increase caused by a growing number of fake nodes (Table 2). Two processes are impacted by the presence of fake nodes: (i) the RIB process, which maintains information about all the routes known to each destination, and (ii) the OSPF process which maintains the entire OSPF topology. Even with a huge number of fake nodes (100,000), the total overhead on both processes was only 154MB—a small fraction of the total memory available. We collected the CPU utilization on



(a) Per-destination time (b) Per-destination augmentation size (c) Cross-destination reduction

Figure 8: Evaluation of our augmentation algorithms.

the router every minute immediately after we started injecting fake nodes. The utilization was systematically low, at most 4%. This is easily explained as Fibbing relies on OSPF Type-5 LSAs which do not cause the routers to recompute their shortest paths to each other.

# fake nodes	RIB memory (MB)	OSPF memory (MB)
1,000	0.09	0.56
5,000	1.58	5.19
10,000	3.56	10.96
50,000	19.67	56.37
100,000	39.78	113.17

Table 2: Routers easily sustain Fibbing-induced load, even with huge augmented topologies.

Fibbing can quickly program forwarding entries. In a second experiment, we measured how long it took for a router to install a growing number of forwarding entries (Table 3). We injected a growing number of fake nodes, one per destination, and measured the total installation time, by tracking the time at which the router updated the last entry in its FIB. The time to process and install one entry was constant (around 900μs), independent of the number of entries. This result is several orders of magnitude better than any OpenFlow switches currently on the market [12, 13]. Since installation of forwarding entries is distributed, routers can install their entries in parallel, meaning *Fibbing can program thousands of network-wide entries within 1 second.*

# fake nodes	installation time (s)	avg time/entry (μs)
1,000	0.89	886.00
5,000	4.46	891.40
10,000	8.96	894.50
50,000	44.74	894.78
100,000	89.50	894.98

Table 3: Programming a forwarding entry in a router with Fibbing is fast, sub 1ms.

Fibbing does not have *any* impact on routing-protocol convergence time. Finally, we compared the total time for routers to converge with and without fake nodes. We failed a link and measured the time for the last FIB entry to be updated considering two cases: (i) no lie was injected and (ii) one lie per destination was injected. Similar to the previous experiments, we repeated the measurements for a growing number of destinations and lies, between 100 and 100,000. In all our experiments, the presence of lies did not have *any* visible impact. The total convergence times with or without lies were systematically within 4ms, with the router being even faster to converge in the presence of lies in some cases.

5.2 Topology Augmentation Evaluation

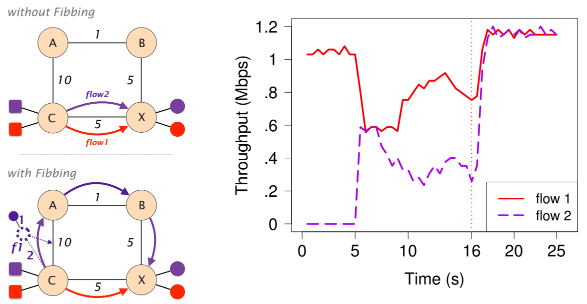
We now evaluate Simple and Merger (§ 3) according to: (i) the time they take to compute an augmented topology for a given requirement, and (ii) the size of the resulting augmented topology. Results are depicted in Fig. 8. Our evaluation is based on simulation performed on realistic ISP topologies [18], whose sizes range from 80 nodes to over 300. On these topologies, we generated forwarding requirements by randomly changing the next-hop of randomly selected nodes. Destinations of requirement DAGs were also randomly generated.

Fibbing augments network topologies within ms. Fig. 8a shows the time (on the y-axis) taken by Simple and Merger for an increasing number of nodes that must change their next-hop (on the x-axis). The plot refers to simulations we ran on the biggest Rocketfuel topology (AS1239). The time taken by Simple to compute the per-destination augmentation varies in the order of milliseconds, ranging from 0.5 ms to 8 ms. While Merger took more time (as expected), its performance is still one order of magnitude lower than the second. For both algorithms, the computation time does not vary much with the number of nodes changing their next-hops.

Merger and cross-optimization effectively reduce the size of the augmented topology. Fig. 8b plots the fake topology size (on the y-axis) when the number of nodes that have to change their next-hop increases (on the x-axis) for a single destination on all topologies. The plot shows that Merger reduces the number of fake nodes by about 25% in the average case and almost 50% in the best case. Fake topology reduction is further corroborated by our cross-destination optimization procedures (see §3.4). Fig. 8c shows a cumulative distribution function (CDF) of the topology augmentation size computed by Simple, Merger, and Merger with cross-destination optimization. The figure refers to simulations with a number of destinations varying between 1 and 100 with 26% of the nodes changing their nexthop. In more than 90% of our simulations, cross-destination optimization achieves a reduction of the augmented topology. Depending on the experiment, such a reduction is up to about 10% with respect to Merger without cross-destination optimization, and 20% with respect to Simple.

5.3 Case Study

We now show the practicality of Fibbing by improving the performance of a real network consisting of four routers (Cisco 3700 running IOS v12.4(3)) connected in a square with link of 1 Mbps capacity (see Fig. 9a). In this network, we introduce two sources (bottom left) that send traffic to two destinations (bottom right) using `iperf`. The first source is introduced at time $t = 0$, the second one at time $t = 5$. OSPF weights are configured such that all traffic flows along link (C, X).



(a) Topology (b) Throughput evolution

Figure 9: Case study on how Fibbing can alleviate congestion. At $t = 16$, a fake node is added to shift one flow to the upper path in the network, increasing the total available bandwidth.

Such a network suffers from two inherent inefficiencies: (i) the upper path is never used and (ii) the two flows systematically traverse the same path, competing for bandwidth, no matter what the link weights are. Fig. 9b plots the throughput of each flow. Immediately after the introduction of the second flow at $t = 5s$, the two flows start competing for the available bandwidth. To improve network efficiency, the Fibbing controller injects a fake node f_1 connected to C and announces one destination at time $t = 16$. A few ms after the injection, we see that the throughput of both flows double as each of them now traverses a different path.

6. REACTION TO FAILURES

We now analyze Fibbing's reaction to different kinds of failures. We distinguish between network (affecting real router or router-to-router links) and controller (shutting down replicas or replica-to-router links) failures. Also, we separately deal with failures inducing network partitions and non-partitioning ones.

Fibbing quickly reacts to non-partitioning failures. Upon network failures, forwarded flows fall in one of the following three cases. First, some flows are not impacted by the failure as their pre-failure forwarding path is not disrupted. Second, flows for which no input requirements have been specified require only the IGP to establish a new path, but no action from the Fibbing controller. Reaction to failures is extremely fast in this case (sub-second even in large networks), thanks to fast convergence [19] and local fast re-route [20] features commonly supported by current IGP implementations. Third, the remaining flows are forwarded on paths modified by the Fibbing controller. They require reaction from the controller, both to remove possible blackholes or loops due to previously injected lies [14], and to avoid requirement violations due to new IGP paths. Theoretically, the total failure reaction time is equal to the sum of the notification time (for the controller to be notified of the failure), the processing time (for the controller to compute the new topology augmentation) and the IGP convergence time (for all routers to install the new lies). Our evaluation (§5) shows that the processing time is negligible, especially for the Simple algorithm. Moreover, the notification time is bounded by the IGP convergence time, as flooding is faster than re-convergence. Thus, in the worst case, the total reaction time is twice the IGP convergence time, that is, still below 2 seconds [19]. Also, in the average case, the notification time is smaller than IGP convergence, because the controller is notified about the failure before all other routers complete convergence; hence, the controller injects new lies *during* the IGP convergence, and the total failure reaction time is slightly higher than IGP convergence without Fibbing.

In addition, if one or more controller replicas fail but others are running, we have no impact on forwarded flows, unless the failed replica is the active one and some of its injected lies expire before the new active replica is elected. Even in the latter case, the new active replica is

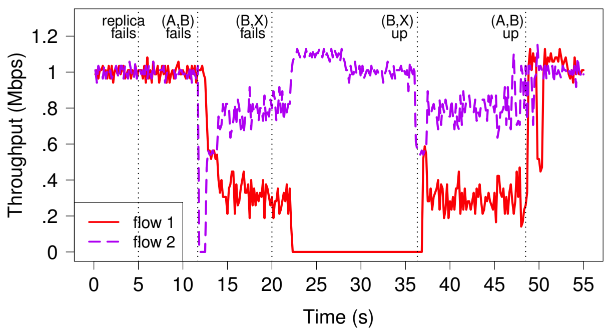


Figure 10: Case study on how Fibbing reacts to failures, and can successfully implement fail-close (flow 1) and fail-open (flow 2) semantics.

quickly elected, in a time which is approximately equal to the detection and flooding of the failure event by the IGP. The short election time makes it unlikely that lies expire before the new active replica is elected, and limits the period with possible disruptions.

Fibbing can implement both fail-open and fail-close semantics to deal with partitions. Even if unlikely, catastrophic events like a simultaneous failure of all the controller replicas or network partitions may happen. As for any centralized solution, a major risk in those cases is to leave the network uncontrolled. This happens, for example, if some routers are not reachable by a controller replica after a network partition. With respect to pure SDN solutions, Fibbing has the additional possibility to delegate control to the underlying IGP. This way, Fibbing can implement both the fail-open or fail-close semantics, on a per-destination basis. For non-critical (optimization) requirements like traffic engineering ones, the corresponding destinations can be injected in the IGP, so that connectivity can be preserved as long as the partition leaves at least one source-destination path. For stringent requirements like security ones (*e.g.*, firewall traversal), Fibbing can implement fail-close semantics by not announcing the corresponding destinations in the IGP. As such, the corresponding flows stop to be forwarded in the absence of the controller. To quickly reach this configuration, we can set a low validity time of the injected lies, making them rapidly expire if not refreshed. This then comes at the cost of additional control-plane overhead.

We confirmed Fibbing resilience. We consider again the topology in Figure 9a, and we connect two controller replicas respectively to routers A and B. The active replica is initially the one connected to A. We assume a strict policy on the red flow forcing it to cross the link (C, X). We then configure a fail-close semantics to it, and a fail-open to the other flow. Starting from a state in which both replicas and all links are up, we successively fail (i) the active replica at time $t = 5$; (ii) link (A, B) at $t = 12$; and (iii) link (B, X) at $t = 20$. Finally, we re-establish both failed links, one at the time (at $t = 36$ and $t = 48$).

The results of this experiment, collected via `iperf`, are reported in Figure 10. Concretely, the *failure of the active replica has no impact* on the forwarded flows. Indeed, the initially passive replica (connected to B) quickly detects the failure of the other replica, and start refreshing the injected lies by the failed controller. When (A, B) fails, the active replica needs to remove the fake node $f1$: Since the physical path (C, A, B, X) is not available anymore, this fake node is creating a loop between C and A for the violet flow. Upon failure detection, the controller then sends the LSA to remove $f1$, re-establishing the connectivity for the disrupted flow in approximately $1s$. Note that this time can be lowered by relying on fast failure detection mechanisms (like BFD). When (B, X) also fails, we create a partition that makes it impossible for the running replica to interact with routers A, C, and X. After about $1s$, the injected lies disappear, because they are not refreshed anymore by any controller. Consistent with the configured failure semantics, the red flow is blackholed (to avoid the IGP routing it over policy-violating paths) while the violet flow keeps using the IGP shortest path. Finally, re-adding the failed links allows the running replica to re-take control of the network: it re-builds a (safe) path for the red flow upon (B, X) restoration, and re-optimizes the distribution of both flow over the available paths when (A, B) is restored.

7. FREQUENTLY ASKED QUESTIONS

We now provide answers to high-level concerns often raised against Fibbing. Since empirical analyses are hardly applicable to those concerns (*e.g.*, debuggability), we describe qualitative considerations.

Is Fibbing a long-term solution? *Yes.* We believe Fibbing is here to stay. In the short run, Fibbing offers programmability and is easy to deploy, at very little cost. A network that ultimately needs even greater flexibility could deploy finer-grained SDN functionality at the edge, and solutions like Fibbing in the core, as advocated by major industry [21] and academic actors [10, 22]. By combining the best of centralized and distributed routing, Fibbing fits the needs of the network core (flexibility, robustness, low overhead) better than current forwarding paradigms.

Does Fibbing make networks harder to debug? *No.* Fibbing relies on “tried and true” protocols. This has several implications. First, Fibbing routing matches the current mental model of operators, a major advantage with respect to other SDN proposals. Moreover, Fibbing is compatible with any existing management, monitoring, and debugging tools. Finally, the Fibbing controller can expose a higher-level interface for debugging, including a mapping between the injected lies and their usage (matched requirements and how).

Does Fibbing sum the complexities of centralized and distributed approaches? *No.* Fibbing uses the underlying IGP in a very simple way. The IGP output is easy to predict and provides the controller with a powerful API to program routers. As a result, the design of the Fibbing controller is significantly simpler than for existing SDN controllers (*e.g.*, [23, 24, 25, 26]) since heavy tasks such as path computation and topology maintenance are offloaded to the routers. Even basic primitives for controller replication and replica consistency are mainly delegated to current distributed routing protocols (see §6).

Does Fibbing impact security? *No.* The lies introduced by the Fibbing controller can easily be authenticated, *e.g.*, using MD5-based authentication [27, 28].

Since Fibbing can only program loop-free paths, can it support middleboxes chaining? *Partially.* Forwarding loops can be encountered when steering traffic through a chain of middleboxes (*e.g.*, [29] and [30]). These requirements *can* be satisfied in Fibbing with local support from routers to break the loops. For instance, a router could match on the input interface in addition to the destination IP address using policy-based routing, a feature widely available on existing routers [31, 32] and provisioned centrally using BGP flowspec [33, 34]. Alternatively, middlebox traffic steering could be implemented through SDN functionality at the network edge, while still using Fibbing in the core.

8. RELATED WORK

Fibbing contributes to the larger debate about centralized and distributed control over routing by identifying a new point in the design space.

Centralized configuration of distributed routing protocols: A centralized management system can perform traffic engineering by optimizing the link weights in link-state routing protocols [35, 36]. Fibbing is more general, since it can implement *any* forwarding paths by injecting fake nodes and links into the link-state routing topology. The extra flexibility enables even better load balancing, as well as a wider range of functionality.

Centralized control using existing routing protocols as a control channel: RCP [11] is a logically-centralized platform that uses BGP to install forwarding entries into routers. RCP must install forwarding entries one-by-one, on each device. In contrast, Fibbing can adapt the forwarding behavior of many routers at once, with little input (*e.g.*, one fake node), and let them compute their own forwarding entries.

Centralized control over the routing/forwarding tables: In SDN, a central controller installs packet-processing rules directly in the switches, possibly reacting to the reception of specific packets. While more flexible (*e.g.*, enabling stateful control logic) than Fibbing, SDN requires updating the switch-level rules one-by-one, and forgoes the scalability and reliability benefits of distributed routing. Recently, the IETF developed I2RS [37] which offers a new management interface for centralized updates the routing information bases (RIBs) in the routers. Still, I2RS must push RIB entries individually to each router.

The Fibbing language for expressing requirements is similar in spirit to Merlin [38], but the mechanism for satisfying the requirements (*i.e.*, fake nodes/links) is entirely different. Our main contributions are the Fibbing techniques and algorithms, not the language.

For the networks which require the extra flexibility provided by OpenFlow, Fibbing helps during the transition by providing access to the FIBs of legacy routers to any SDN controller [39]. This contrasts to techniques like Panopticon [40], where programmability is only available in the SDN-enabled parts of the network.

9. CONCLUSIONS

The advent of SDN makes it clear that network operators want their networks to be more programmable and easier to manage centrally. In this paper, we show how Fibbing can achieve those objectives, by centrally and automatically controlling forwarding without forgoing the benefits of distributed routing protocols. Fibbing is expressive, scalable, and works with existing routers. In future work, we plan to look at extensions of IGP protocols (*e.g.*, for source-destination routing [41] or network service header awareness) to enable finer-grained control via Fibbing. Abstractly, Fibbing shows how centralized and distributed approaches can be profitably combined. We believe that new research can further explore this direction, for example, investigating an alternative division of tasks between centralized and distributed network components.

Acknowledgements

We are grateful to SIGCOMM anonymous reviewers and our shepherd, Teemu Koponen, for insightful comments. We thank Jo Segaert from BELNET and Dave Ward, Clarence Filsfils and Kris Michielsen from Cisco Systems for their support in testing Fibbing on real routers. This work has been partially supported by the EC Seventh Framework Programme (FP7/2007-2013) grant no. 317647 (Leone) and by the ARC grant 13/18-054 from Communauté française de Belgique.

10. REFERENCES

[1] D. Awduche *et al.*, “RSVP-TE: Extensions to RSVP for LSP Tunnels,” RFC 3209, 2001.

[2] N. McKeown *et al.*, “OpenFlow: enabling innovation in campus networks,” *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[3] A. Farrel, J.-P. Vasseur, and J. Ash, “A Path Computation Element (PCE)-Based Architecture,” RFC 4655, 2006.

[4] C. Filsfils *et al.*, “Segment Routing Architecture,” Internet Draft, 2014.

[5] B. Clouston and B. Moore, “Definitions of Managed Objects for HPR using SMIv2,” RFC 2238, 1997.

[6] D. Oran, “OSI IS-IS Intra-domain Routing Protocol,” RFC 1142, 1990.

[7] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. A. Maltz, “Latency inflation with MPLS-based traffic engineering,” in *IMC*, 2011.

[8] S. Jain *et al.*, “B4: Experience with a Globally-Deployed Software Defined WAN,” in *SIGCOMM*, 2013.

[9] C.-Y. Hong *et al.*, “Achieving High Utilization with Software-Driven WAN,” in *SIGCOMM*, 2013.

[10] M. Casado *et al.*, “Fabric: A retrospective on evolving sdn,” in *HotSDN*, 2012.

[11] M. Caesar *et al.*, “Design and implementation of a routing control platform,” in *NSDI*, 2005.

[12] X. Jin *et al.*, “Dynamic scheduling of network updates,” in *SIGCOMM*, 2014.

[13] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, “OFLOPS: An Open Framework for Openflow Switch Evaluation,” in *PAM*, 2012.

[14] S. Vissicchio, L. Vanbever, and J. Rexford, “Sweet little lies: Fake topologies for flexible routing,” in *Hotnets*, 2014.

[15] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, “Seamless Network-Wide IGP Migrations,” in *SIGCOMM*, 2011.

[16] “Quagga routing suite,” www.nongnu.org/quagga.

[17] J. Moy, “OSPF Version 2,” RFC 2328, Apr. 1998.

[18] N. Spring, R. Mahajan, and D. Wetherall, “Measuring ISP topologies with Rocketfuel,” in *SIGCOMM*, 2002.

[19] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure, “Achieving Sub-second IGP Convergence in Large IP Networks,” *ACM SIGCOMM CCR*, vol. 35, no. 3, 2005.

[20] C. Filsfils, P. Francois, M. Shand, B. Decraene, J. Uttaro, N. Leymann, and M. Horneffer, “Loop-Free Alternate (LFA) Applicability in Service Provider (SP) Networks,” RFC 6571, 2012.

[21] T. Koponen *et al.*, “Network Virtualization in Multi-tenant Datacenters,” in *NSDI*, 2014.

[22] “Time for an SDN Sequel? Scott Shenker Preaches SDN Version 2,” www.sdxcentral.com/articles/news/scott-shenker-preaches-revised-sdn-sdnv2/2014/10/.

[23] “ONOS: Open Network Operating System,” http://onosproject.org/.

[24] T. Koponen *et al.*, “Onix: A distributed control platform for large-scale production networks,” in *OSDI*, 2010.

[25] “Project Floodlight,” http://www.projectfloodlight.org/floodlight/.

[26] N. Foster *et al.*, “Languages for software-defined networks,” *IEEE Comm. Mag.*, 2013.

[27] “Cisco OSPF MD5 Authentication,” http://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/13697-25.html.

[28] “Juniper OSPF MD5 Authentication,” http://www.juniper.net/documentation/en_US/junos14.2/topics/topic-map/ospf-authentication.html.

[29] Z. A. Qazi *et al.*, “Simple-fying middlebox policy enforcement using sdn,” in *SIGCOMM*, 2013.

[30] S. K. Fayazbakhsh *et al.*, “Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags,” in *NSDI*, 2014.

[31] “Cisco. Configuring Policy-Based Routing,” http://www.cisco.com/c/en/us/td/docs/ios/12_2/qos/configuration/guide/fqos_c/qcfpbr.html.

[32] “Juniper. Configuring Filter-Based Forwarding to a Specific Outgoing Interface or Destination IP Address,” http://www.juniper.net/techpubs/en_US/junos12.2/topics/topic-map/filter-based-forwarding-policy-based-routing.html.

[33] “Cisco. Implementing BGP Flowspec,” http://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/asr9k_r5-2/routing/configuration/guide/b_routing_cg52xasr9k/b_routing_cg52xasr9k_chapter_011.html.

[34] “Juniper. Enabling BGP to Carry Flow-Specification Routes,” https://www.juniper.net/documentation/en_US/junos12.3/topics/example/routing-bgp-flow-specification-routes.html.

[35] B. Fortz and M. Thorup, “Internet traffic engineering by optimizing OSPF weights,” in *INFOCOM*, 2000.

[36] B. Fortz, J. Rexford, and M. Thorup, “Traffic engineering with traditional IP routing protocols,” *IEEE Comm. Mag.*, vol. 40, no. 10, pp. 118–124, 2002.

[37] A. Atlas, J. Halpern, S. Hares, and D. Ward, “An Architecture for the Interface to the Routing System,” Internet Draft, 2013.

[38] R. Soulé *et al.*, “Merlin: A language for provisioning network resources,” in *CoNEXT*, 2014.

[39] S. Vissicchio, L. Vanbever, and O. Bonaventure, "Opportunities and research challenges of hybrid software defined networks," *ACM SIGCOMM CCR*, vol. 44, no. 2, pp. 70–75, 2014.

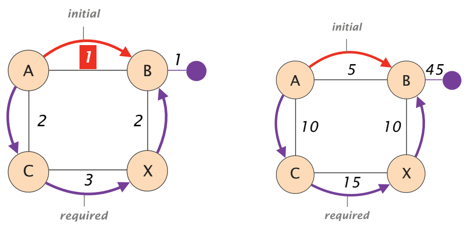
[40] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann, "Panopticon: Reaping the Benefits of Incremental SDN Deployment in Enterprise Networks," in *USENIX ATC*, 2014.

[41] F. Baker, "IPv6 Source/Destination Routing using OSPFv3," Internet Draft, 2013.

APPENDIX

A. TOPOLOGY INITIALIZATION

Figure 11a shows an example of a non-initialized topology where no feasible augmentation can implement a specific forwarding DAG. In the example, it is impossible to add fake nodes or links that would change A's shortest path. Any such fake path would, at a minimum, include a fake link to a fake node announcing the destination attached to B. Typically, routing protocols require that a positive cost is set on any link or destination announcement. Hence, the minimum cost of any fake path is 2, making it impossible to divert A away from its current shortest path (also with a cost of 2).



(a) Non compliant (b) Fibbing compliant

Figure 11: The left topology is not Fibbing compliant: A is at a distance of 1 from B preventing traffic from being attracted. The right topology is Fibbing compliant; traffic can be attracted from A by injecting a fake node with a cost less than 5. It is *always* possible to go from a non-compliant to a compliant topology.

Figure 11b shows a possible output of our initialization procedure for the topology in Figure 11a. Such an initialization makes the topology Fibbing compliant, which is a sufficient condition for implementing any per-destination forwarding DAG with Fibbing.

B. MERGER ALGORITHM

In this section, we provide additional details on the Merger algorithm, and we prove its correctness.

B.1 General Notation

Throughout this section, we use the following notation. We indicate a generic non-augmented topology provided in input to Merger as G, and augmentations of G iteratively computed by Merger as G', G'', etc. For any topology T, the shortest paths from s to d in T and their cost are respectively denoted as $sp(s, d, T)$ and $dist(s, d, T)$. Similarly, we denote the cost of a path P in T as $cost(P, T)$.

In addition, we always assume that fake path costs are *consistently assigned*, i.e., it exists an integer $k \geq 0$ such that the cost of any fake path (x, f_x, d) created by a globally-visible lie is equal to $lb(f_x) + k$. To prove Merger correctness, we also define *guarantees* in the augmented topology (e.g., about paths of traversing or not traversing a specific fake node) as assertions valid for any value of k used for consistently-assigned fake path cost.

B.2 Lower bound propagation procedure

The lower bound propagation procedure takes as input the non-augmented topology G, the target forwarding DAG for a destination d and initial fake node lower bounds (i.e., initialized according to the next-hop preserving neighbors as in §3). From this input, it first formalizes the constraints between lower bounds of next-hop changing neighbors as inequalities to be respected. Then, it fixes lower bounds satisfying the computed inequalities considering them one by one, in a precise order. In the following, we provide a more detailed description of those two phases.

Formalization of the constraints between lower bounds. This is a static computation, only based on the link weights in G. Consider any fake node $f_{r,d}$ connected to a node r. Let $n \neq r$ be another node in G, to which Merger has already connected another fake node $f_{n,d}$. We then want to impose that the path $(n \dots r\ f_{r,d}\ d)$ is longer than $(n\ f_{n,d}\ d)$. Under the assumption of consistently assigned fake path costs, this property holds if $lb(f_{n,d}) < dist(n, r, G) + lb(f_{r,d})$. We therefore use this inequality as the formalization of the dependency between lower bounds $lb(f_{r,d})$ and $lb(f_{n,d})$.

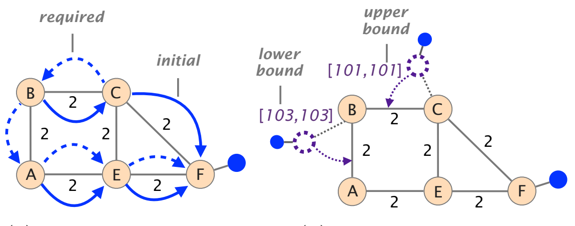
Computation of lower bound values. This phase is performed iterating over the lower bounds. At each iteration, we sort all not yet fixed lower bounds and we fix the value of the first lower bound in such a sorted set. Fixed values are not modified by successive iterations. We now detail a generic iteration i. We denote the set of lower bounds not yet fixed as $\mathcal{C}_i$, and the lower bound fixed at iteration i as $lb(f_{r,d})$.

1. **Sorting lower bounds**. Lower bounds are sorted according to the value of a specific function δ. The value of this function corresponds to the value of the considered lower bound minus the distance between the corresponding real node and its closest node connected to any other fake node. That is, the value of δ at iteration i for a given lower bound

$lb(f_{x,d})$ is $\delta_i(f_{x,d}) = lb_i(f_{x,d}) - dist(x, y, G)$ where $lb_i(f_{x,d})$ is the value of $lb(f_{x,d})$ at i and y is the closest (from x) real node connected to a fake one. In the case of lower bounds with the same δ value, any deterministic tie-breaker can be used to have a total order among lower bounds in $\mathcal{C}_i$.

2. **Decision of one lower bound**. We fix $lb(f_{r,d})$ to its current value, and we remove it from $\mathcal{C}_i$. In the following, we indicate this value as $\bar{l}$.

3. **Update of non-fixed lower bounds**. We ensure that path $(r\ f_{r,d}\ d)$ will be the shortest one from r to d in the augmented graph if $lb(f_{r,d}) = \bar{l}$. In particular, we impose that all constraint inequalities with $lb(f_{r,d})$ on the left side are satisfied. Consider any of those constraint inequalities $lb(f_{r,d}) < dist(r, j, G) + lb(f_{j,d})$, with $lb(f_{j,d})$ being any lower bound in $\mathcal{C}_i$. This would be satisfied if $lb(f_{j,d})$ ends up having a value strictly greater than $\bar{l} - dist(r, j, G)$. We then update the value of $lb(f_{j,d})$ to the maximum between its current value $lb_i(f_{j,d})$ and $\bar{l} - dist(r, j, G) + 1$. Note that this operation may increase $lb(f_{j,d})$ to a value higher than the corresponding upper bound $ub(f_{j,d})$, as in the example in Figure 12. In such cases, we use a locally-visible lie to create the fake path $(j\ f_{j,d}\ d)$, assign the minimum possible cost (2) in an IGP topology to it, and remove $f_{j,d}$, as for B in Figure 12. Then, we restart the computation of lower bounds from scratch, without considering j and $lb(f_{j,d})$ anymore.



(a) Initial and required flows. (b) Initial fake bounds.

Figure 12: A case in which Merger needs a locally-visible lie. Indeed, during the lower bound propagation procedure, $lb(f_A)$ is incremented to make A use the fake path (A, f_A, d) instead of the fake path via B; this way, it however becomes bigger than $ub(A)$.

B.3 Merging procedure

This procedure iterates over source-destination paths in the input forwarding DAG R for destination d. For each source-destination path, it repeats the following operations.

Selection of candidates. Let $P = (r_1, r_2, \dots r_m)$ be a source-destination path in R, and let $F = (f_1, f_2, \dots f_k)$ be the sequence of fake nodes connected to nodes in P and used to implement P in the current augmented topology. Let F be ordered according to the sequence of routers in P. More formally, for any f_i and f_j in F, $i > j$ if and only if the routers r_i and r_j to which f_i and f_j are respectively connected are such that $i > j$ (that is, r_i is a successor of r_j in P).

Merger tries to merge fake nodes which are consecutive in F. More precisely, it iteratively tries to merge every f_i into f_{i+1}, with $i = 1, \dots, k-1$. It starts from f_1 and tries to merge f_1 into f_2 according to the subprocedure described in the next paragraph. Irrespectively of the outcome of this procedure, f_2 will remain (possibly, with modified lower bounds if f_1 has been merged into it). Hence, Merger applies a new merging attempt on the pair (f_2, f_3). The algorithm iterates merging attempts until f_k is reached.

Merging attempt. The core of the merging attempt is the *feasibility check*, used by Merger to assess whether the pair of input candidates (f_i, f_j) can be merged and with which lower bound adjustments. Assume that f_i and f_j are two fake nodes announcing destination d to the connected real nodes i and j in a source-destination path P in R. We now formally describe what are the checks performed by Merger in this phase.

1. **Shortest path compliance.** Let $all\_sps(i, j, G)$ be the set of shortest paths from i to j in the input topology G. Merger asserts whether every path in $all\_sps(i, j, G)$ is included in the input forwarding DAG R. A similar check is performed on all the paths from any node p whose shortest path for d includes i before the merging. In the following, we refer to the shortest path from i to j in G which is a sub-path of P as $sp(i, j, G)$.

2. **Candidate compatibility.** If shortest path are compliant according to the previous check, the algorithm assesses the existence of a cost of (j, f_j, d) such that (i) the shortest path of i in the post-merging augmented topology G' becomes the concatenation of $sp(i, j, G')$ and (j, f_j, d); and (ii) the current next-hops of any node do not change. To this end, we temporarily modify lower and upper bounds of f_j. The modified lower bound $\tilde{lb}(f_j)$ is the lower bound of f_j recomputed with the constraints that (i) $dist(i, j, G) + \tilde{lb}(f_j)$ must be strictly lower than $dist(i, d, G)$ (to let i use f_j) and than $dist(n, d, G)$ for any pre-merging predecessor n of i or j (to force pre-merging predecessors of i and j to use f_j); and (ii) $dist(i, j, G) + \tilde{lb}(f_j)$ must be strictly greater than $dist(m, d, G)$ for any other real node m (to avoid that another nodes change their shortest paths). The modified upper bound $\tilde{ub}(f_j)$ is the minimum between its unmodified upper bound $ub(f_j)$ and $ub(f_i) - dist(i, j, G)$. Merger asserts if

$\tilde{lb}(f_j)$ exists and $\tilde{lb}(f_j) \leq \tilde{ub}(f_j)$. If this is the case, it also stores $\tilde{lb}(f_j)$ and $\tilde{ub}(f_j)$, as they will be the new bounds of f_j if the merging happens.

3. **Network-wide feasibility.** Since the bounds of the merged node f_j can change and one fake node disappears in the merge, other pre-merging lower bound values may become inconsistent. To avoid inconsistencies, we re-run the lower bound propagation procedure of Merger, ignoring the bounds of f_i and using $\tilde{lb}(f_j)$ and $\tilde{ub}(f_j)$ as bounds for f_j. In this step, we also apply extra care for real nodes initially connected to multiple fake nodes, i.e., subject to load-balancing requirements. Consider any of those real nodes, l, initially connected to fake nodes $f_{l1} \dots f_{lk}$, with $k \geq 2$. During the lower bound propagation procedure, we impose additional constraints on l, such that the cost of all the current shortest paths from l to d (e.g., via fake nodes) have the same cost. For example, when Merger merges $f1$ into f_2 in the topology in Figure 6b, it needs to maintain the load-balancing requirement on A. To this end, it forces $lb(f0)$ to have the same lower bound value as $dist(A, B)+dist(B, d)$ (where d is the destination), that is $5 + lb(f2)$. Note that sometimes considering load-balancing requirements correctly prevents fake node merging. Considering again Figure 6b, Merger does not merge $f3$ into $f4$. Indeed, the lower bound of $f3$ has been raised to enable the merging of $f2$ into $f4$, hence it is now not compatible anymore with $f4$. This is correct as using only $f4$ forcedly breaks the load-balancing requirement on A. If the lower bound propagation succeeds without adding locally-visible lies, we finally store all the adjusted lower bound values, including the relaxed lower and upper bound of f_j.

If any of those checks fails, Merger immediately abort the merging attempt. Otherwise, it merges f_i into f_j: It removes f_i and sets the lower bounds of all other nodes to the ones computed during the third check.

B.4 Correctness Proofs

We prove the correctness of Merger in two steps. First, we show that the algorithm correctly implements any input forwarding DAG after the lower bound computation procedure (see Theorem 2). Second, we prove that merges performed during the merging procedure never change the forwarding paths implemented in the pre-merging augmented graph (see Theorem 3). The correctness of Merger then follows by noting that it sequentially runs the lower bound computation and merging procedures.

For the sake of simplicity, our proofs assume no equal-cost multipath in either the original topology or the required paths. Nevertheless, they can be generalized to cases of multiple shortest paths in the original IGP topology and load-balancing requirements.

We start by proving the correctness of the non-merged augmented topology. We denote with k an integer such that $k \geq 0$, with G the original topology, and with G' the topology as computed by Merger after the lower bound propagation procedure. Since we never change any link or link weight in G, for any pair of real nodes x and y in G, $dist(x, y, G') = dist(x, y, G)$.

Lemma 1. *If $lb(x)$ is fixed at an iteration i of the lower bound propagation procedure and $\delta_i(x) \geq \delta_i(y)$ at that iteration, then $\delta_j(x) \geq \delta_j(y)$ at any iteration $j > i$.*

Proof. Assume by contradiction that $\delta_i(x) \geq \delta_i(y)$ but $\delta_j(x) < \delta_j(y)$ for some iteration $j > i$ and some lower bound $lb(y) \neq lb(x)$. Let m be the closest iteration to i such that $\delta_m(x) < \delta_m(y)$. Since $lb(x)$ is fixed at i, it must be $\delta_l(x) = \delta_i(x)$ for any $l > i$, hence $\delta_i(x) = \delta_m(x)$. Moreover, let u and v be the real nodes respectively connected to x and y.

One of the following cases must apply.

- $m = i + 1$. In this case, the delta function of y must have been increased during the i-th iteration. By step 3 in the computation of lower bound values, $lb_m(y)$ must be equal to $max\{lb_i(y), lb_i(x) - dist(u, v, G) + 1\}$. We have two subcases.
  If $lb_m(y) = lb_i(y)$, then $\delta_m(y) = \delta_i(y)$. Since $\delta_i(x) \geq \delta_i(y)$ by hypothesis and $\delta_i(x) = \delta_m(x)$, we must have $\delta_m(x) \geq \delta_m(y)$.
  Otherwise, we have $lb_m(y) = lb_i(x) - dist(u, v, G) + 1$, that is, $lb_m(y) - 1 = lb_m(x) - dist(u, v, G)$. The value of $\delta_m(y)$ is defined as $lb_m(y) - cost(P, G)$, where P is a given path in G; hence, $\delta_m(y) \leq lb_m(y) - 1$, since the cost of any IGP path is strictly greater than zero. Moreover, $\delta_m(x) = lb_m(x) - dist(u, z, G)$ with z being the closest node to u also connected to a fake node; consequently, $\delta_m(x) \geq lb_m(x) - dist(x, y, G)$. Combining those inequalities, we have that $\delta_m(y) \leq lb_m(y) - 1 = lb_m(x) - dist(x, y, G) \leq \delta_m(x)$, that is, $\delta_m(x) \geq \delta_m(y)$.
  In both subcases, we generate a contradiction, as m is defined as an iteration in which $\delta_m(x) < \delta_m(y)$.
- $m > i + 1$. In this case, it must exist an iteration n, with $i > n \geq m$, such that $\delta_{n-1}(x) \geq \delta_{n-1}(y)$ and $\delta_n(x) < \delta_n(y)$. Since $\delta_{n-1}(x) = \delta_n(x) = \delta_i(x)$, $lb(y)$ cannot be fixed at $n-1$ and $\delta_n(y) > \delta_{n-1}(y)$. Consider the lower bound $lb(z) \neq lb(y)$ fixed at $n-1$. Fixing it at iteration $n-1$ must cause $\delta_n(y) > \delta_{n-1}(y)$. Since $lb(z)$ is fixed at $n-1$, we must have $\delta_{n-1}(z) \geq \delta_{n-1}(y)$. We have two subcases.
  If $\delta_{n-1}(z) \geq \delta_{n-1}(y)$ and $\delta_n(z) < \delta_n(y)$, we can consider z instead of x, and iterate our reasoning by contradiction on z and y. The first case of our proof applies since $n = (n-1)+1$, hence we directly generate a contradiction.

Otherwise, if $\delta_{n-1}(z) \geq \delta_{n-1}(y)$ and $\delta_n(z) \geq \delta_n(y)$, then $\delta_{n-1}(x) = \delta_n(x) < \delta_n(y) \leq \delta_n(z) = \delta_{n-1}(z)$. That is, $\delta_{n-1}(x) < \delta_{n-1}(z)$. However, since $lb(x)$ if fixed at i, we must also have $\delta_i(x) \geq \delta_i(z)$. Hence, we can iterate our reasoning by contradiction on x and z. Note that $n-1 < m$, which means that we cannot iterate indefinitely on this sub-case, but we eventually fall in another case and generate a contradiction.

In all the subcases, we eventually generate a contradiction, which proves the statement. □

LEMMA 2. *The shortest path from any real node r connected to a fake node $f_{r,d}$ to d in G' is $(r\ f_{r,d}\ d)$.*

PROOF. The statement holds if the fake node announces d using a locally-visible lie, because the cost (2) set for this fake path in the update of non-fixed lower bounds is guaranteed to be lower than the cost of any other cost in G'. We then focus on fake nodes announcing globally-visible lies.

Consider any pair of lower bounds $lb(x)$ and $lb(y)$. Let $i1$ and $i2$ be the iterations at which $lb(x)$ and $lb(y)$ are respectively fixed during the lower bound propagation procedure. We denote the value of $\delta(x)$ and $\delta(y)$ after the last iteration of the procedure as $\delta_f(x)$ and $\delta_f(y)$. Moreover, for brevity, we write F_x and F_y instead of $(r_x\ x\ d)$ and $(r_y\ y\ d)$, with r_x and r_y being the real nodes connected to x and y respectively.

By definition of the procedure, we have two cases.

- $lb(x)$ is fixed before $lb(y)$, that is, $i1 < i2$. Since we assume that fake path costs are consistently assigned, we have that $cost(F_y, G') = lb_f(y) + k$ with $k \geq 0$, which implies $cost(F_y, G') \geq lb_f(y)$. Moreover, as a consequence of step 3 in the computation of lower bound values, $lb_f(y) > lb_f(x) - dist(r_x, r_y, G')$. Since fake path costs are consistently assigned, we can rewrite the right side of the previous inequality as $lb_f(x) - dist(r_x, r_y, G') = cost(F_x, G') - k - dist(r_x, r_y, G')$, hence $lb_f(x) - dist(r_x, r_y, G') \geq cost(F_x, G') - dist(r_x, r_y, G')$ by definition of k. Concatenating all previous inequalities, we conclude that $cost(F_y, G') > cost(F_x, G') - dist(r_x, r_y, G')$.
- $lb(x)$ is fixed after $lb(y)$, that is, $i1 > i2$. It must then be $\delta_{i2}(x) \leq \delta_{i2}(y)$. This implies $\delta_f(x) \leq \delta_f(y)$, by Lemma 1. We now express $\delta_f(x)$ and $\delta_f(y)$ with respect to the cost of paths in G'.

  On one hand, by definition of δ, $\delta_f(x) = lb_f(x) - dist(r_x, n, G)$ with n being the closest node to r_x. Hence, $\delta_f(x) \geq lb_f(x) - dist(r_x, r_y, G')$, i.e., $\delta_f(x) \geq cost(F_x, G') - k - dist(r_x, r_y, G')$ by definition of consistently-assigned fake path costs. Since $k \geq 0$, it must be $\delta_f(x) \geq cost(F_x, G') - dist(r_x, r_y, G')$.

  On the other hand, by definition of δ, $\delta_f(y) = lb_f(y) - dist(r_y, z, G')$ for some real node z; that is, $\delta_f(y) = cost(F_y, G') - k - dist(r_y, z, G')$ by definition of consistently-assigned fake path costs. Since $dist(r_y, z, G') > 0$ and $k \geq 0$, we thus have $\delta_f(y) < cost(F_y, G')$.

  Combining the previous inequalities, we then have $cost(F_y, G') > \delta_f(y) \geq \delta_f(x) \geq cost(F_x, G') - dist(r_x, r_y, G')$.

In both cases, we have $cost(F_y, G') > cost(F_x, G') - dist(r_x, r_y, G')$, that is, $cost(F_x, G') < cost(F_y, G') + dist(r_x, r_y, G')$. This means that path F_x is shorter than reaching r_y from r_x and then using F_y. By applying the same argument to any lower bound $lb(y)$, we conclude that F_x is the shortest path from r_x to d in G'. The statement then follows by applying the same argument to all nodes r_x in G'. □

LEMMA 3. *For any real node n not directly connected to any fake node, if $sp(n, d, G)$ is guaranteed not to contain a fake node f before the lower bound propagation procedure, then $sp(n, d, G')$ is guaranteed not to contain f after the procedure too.*

PROOF. By definition of guarantee, $dist(n, r_f, G) + lb(f) > dist(n, d, G)$, where r_f is the real node connected to f. Note that $dist(n, r_f, G') = dist(n, r_f, G)$ and $dist(n, d, G') = dist(n, d, G)$ since the original topology is never modified by Merger (only new fake nodes and links are added to it). The statement then follows by noting that $lb(f)$ is never decreased by the lower bound propagation procedure, by construction (the procedure can only increase the value of some lower bounds during the update of non-fixed lower bounds). □

THEOREM 2. *If fake path costs are consistently assigned, Merger correctly implements the input forwarding DAG after the lower bound propagation procedure.*

PROOF. Consider any real node r. We have two cases.

If r is not connected to any fake node, the assignment of lower bound values before the propagation procedure ensures that paths including any fake node connected to real ones not in $sp(r, d, G)$ have a cost strictly greater than $dist(r, d, G)$. Hence, by Lemma 3, r's shortest path $sp(r, d, G')$ in the augmented topology is guaranteed not to traverse any fake node connected to a real one not in $sp(r, d, G)$. By the property of the shortest path, $sp(r, d, G')$ can actually be written as the concatenation of $P = sp(r, x, G')$ and $Q = (x, f_x, d)$, with either (i) $Q \neq \emptyset$ and f_x connected to a real node $x \in sp(r, d, G)$; or (ii) $Q = \emptyset$ and $x = d$ if no node in $sp(r, d, G)$ is connected to a fake one. In any case, r's next-hop is the same as in the original topology. This is consistent with the input forwarding DAG, given that Merger initially adds fake nodes to all next-hop changing real ones.

Otherwise, if r is connected to a fake node, then Lemma 2 holds. Hence, the required next-hop (as in the forwarding DAG) can be imposed via the fake path, which yields the statement. □

We now prove that Merger does not trigger violations of previously-enforced forwarding DAGs. To this end, we first prove a property preserved by the merging procedure (invariant).

LEMMA 4. *For any pair of real nodes (r, z) and any fake one f_l connected to z and announcing a given destination d, if $dist(r, d, G') < dist(r, z, G') + lb_1(f_l)$ in the topology G' provided as input to the merging procedure, then $dist(r, d, G^k) < dist(r, z, G^k) + lb_k(f_l)$ at any iteration k of the merging procedure.*

PROOF. Consider any iteration k of the merging procedure, such that $dist(r, d, G^{k-1}) < dist(r, z, G^{k-1}) + lb_{k-1}(f_l)$. During z, Merger picks two fake nodes f_1 and f_2, and assess the possibility to merge f_1 into f_2.

If the merging attempts fails, then all lower bounds and link weights remain as in the previous iteration. Hence, $dist(r, d, G^{k-1}) < dist(r, z, G^{k-1}) + lb_{k-1}(f_l)$ directly implies $dist(r, d, G^k) < dist(r, z, G^k) + lb_k(f_l)$.

Otherwise, f_1 is merged into f_2. By definition of the candidate compatibility step, the new lower bound of the merged node is initially set to a value $\tilde{lb}(f_2)$ such that $dist(r, d, G^{k-1}) < dist(r, z, G^{k-1}) + \tilde{lb}(f_2)$. Any other lower bound also satisfied the corresponding inequality by hypothesis. Moreover, by its definition, no lower bound is decreased during the lower bound propagation procedure run in the network-wide feasibility step. Finally, link weights in the original graph are never modified by the merging procedure, hence $dist(r, d, G^{k-1}) = dist(r, d, G^k)$ and $dist(r, z, G^{k-1}) = dist(r, z, G^k)$. This implies that for any fake node f_l $dist(r, d, G^k) < dist(r, z, G^k) + lb_k(f_l)$ at the end of the iteration k, which yields the statement. □

THEOREM 3. *Successful merging attempts in Merger do not affect the enforcement of the forwarding DAG implemented before the merging.*

PROOF. Let R be any forwarding DAG for a destination d, and let G' and G'' be the augmented topologies before and after merging a fake node f_1 into another f_2. We denote the real nodes connected to f_1 and f_2 as x and y respectively.

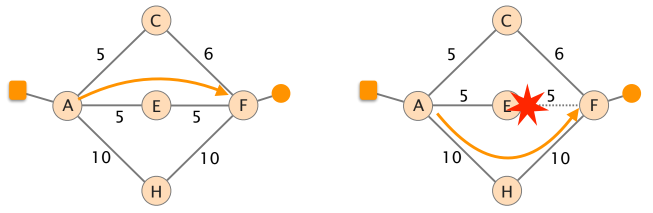
For any real node r, one of the following cases holds.

- r's shortest path includes f_1 before the merge (possibly $r = x$). By definition of the post-merging upper bound of f_2 in the candidate compatibility check, $dist(r, y, G') + \tilde{ub}(f_2) < dist(r, d, G')$, which guarantees that $sp(r, y, G') + \tilde{ub}(f_2)$ is shorter than the original shortest path from r to d. Moreover, by the candidate compatibility and network-wide feasibility steps, $dist(r, y, G'') + \tilde{lb}(f_2) < dist(r, d, G'')$. In contrast, consider any fake node $f_l \neq f_1, f_2$. By definition of the fake bound computation phase in Merger, we must have $dist(r, d, G^0) < dist(r, z, G^0) + lb(f_l)$ on the augmented topology G^0 provided as input to the merging procedure. Lemma 4 then implies that $dist(r, d, G'') < dist(r, z, G'') + lb(f_l)$. By combining the previous two inequalities, we conclude that $sp(r, y, G'') + (y, f_2, d)$ is guaranteed to be the shortest path of r after the merging. Finally, as a consequence of the shortest path compliance check, it must be $sp(r, y, G'') = sp(r, x, G'') + sp(x, y, G'')$, which implies that the next-hop of r is not change by the fake node merging.
- r's shortest path does not include f_1 nor f_2 before the merging. We have three sub-cases. If r is next-hop preserving, it is not directly connected to any fake node and $dist(r, y, G'') + \tilde{lb}(f_2) > dist(r, d, G'')$, by definition of $\tilde{lb}(f_2)$ in the candidate compatibility step. Otherwise, if r is next-hop changing and connected to a locally-visible fake node, then it uses the fake path via its connected fake node independently of the presence of any other fake node, because of the cost (2) set for this fake path in the update of non-fixed lower bounds. Finally, if r is next-hop changing and connected to a globally-visible fake node, the lower bound propagation procedure run in the network-wide feasibility check ensures that r keeps using in G'' the same shortest path as in G' (see Lemma 3).

In all the cases, r has the same next-hop in G' and G'', which proves the statement. □

C. UNSUPPORTED BACKUP

We now show a case of backup requirement not supported by our current graph augmentation algorithms. This case is represented in Figure 13.



(a) Primary requirement. (b) Backup requirement.

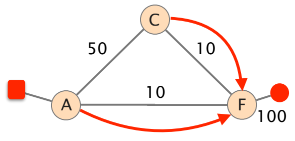
Figure 13: A backup requirement not supported by our algorithms.

In this example, the primary requirement is already fulfilled by the shortest path in the original topology (see Figure 13a). By construction, our algorithms do not add any fake node to the network, in order to minimize the control-plane overhead. However, this implies that the backup requirement depicted in Figure 13b cannot be enforced. Indeed, for path (A, H, F) to be used instead of (A, C, F) in an augmented topology, we need a fake node f_A connected to A such that the path $\mathcal{P}$ from A to the destination via the f_A is shorter than (A, C, F). This means that the cost of $\mathcal{P}$ must be 10 or less. However, such a cost is less or equal than the cost of the original shortest path, hence making $\mathcal{P}$ be used by A even in the absence of failures and disrupting the primary requirement illustrated in Figure 13a.

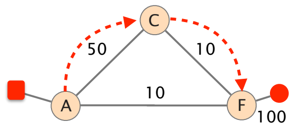
D. OSPF LIMITATIONS

As stated in §4, the current OSPF protocol does not allow full Fibbing expressiveness with global lies (i.e., it violates Theorem 1). The reason is that global lies are implemented in OSPF relying on the forwarding address (FA) field. Namely, in an OSPF LSA, we can specify an IP address x as FA. In this case, the router configured with x on a network interface discards the LSA. All the others compute the cost of the route included in the injected LSA as cost to reach x plus the cost specified in the LSA. Moreover, if this route is selected by a router u, u will forward the corresponding packets on its next-hop in the *OSPF shortest path to* x.

This latter observation is at the core of the example in Figure 14, where globally-visible lies cannot be used to enforce a simple requirement. In this example, for a given destination d, A is required to change its next-hop from F to C. Hence, we need to redirect A's next-hop on the link (A, C). We can then inject a OSPF globally-visible lie as an LSA specifying a new route with the closest interface of C as FA and cost $m < 90$. This would make the cost of this route, that is, $dist(A, C) + m < 20 + 90$, strictly lower than the cost of the original shortest path, i.e, 100. However, even in this case, A will use the next-hop on the shortest path to C as next-hop for the router announced in the lie. This means that A will keep sending traffic F which is on its shortest path (A, F, C) to C.



(a) Initial flows.



(b) Required flows.

Figure 14: A requirement impossible to support with global lies in the current OSPF implementation.

We rely on locally-visible lies to overcome the limitations of globally-visible ones. For example, a single locally-visible lie can be used to enforce the requirement illustrated problem shown in Figure 14. Indeed, an OSPF locally-visible lie uses an IP address which is announced network-wide but not installed in the OSPF routing table of any router (see §4). Hence, when resolving the FA, A will not find any OSPF shortest path to the FA. It will then fallback on the directly-connected route to the FA, that is (A, C), to forward the packet.