1 INTRODUCTION
Centralized routing decisions provide enormous flexibility that often comes at the cost of the robustness provided by distributed protocols. While centralized control, such as Software Defined Networking (SDN), enables easy configuration and has high manageability, distributed routing protocols tend to have faster path installation and failure response rates. In their 2015 paper, "Central Control Over Distributed Routing", Vissicchio, Tilmans, Vanbever, and Rexford introduce 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 table based on the augmented topology.

Here, we aim to reproduce a subset of the results presented in their paper[6], namely the time it takes for a certain percentage of nodes to change next-hops (Figure 8a); number of fake nodes created for a certain percentage of nodes to change next-hops (Figure 8b); throughput evolution of Case Study presented (Figure 9); throughput evolution in relation to failure responses (Figure 10). To reproduce the Figures 8a and 8b, we generate random forwarding requirement DAGs, and apply the author’s implementation of the algorithms within a VM. For reproducing the case studies (Figure 9 and Figure 10), we set up a testbed in mininet.

While we’re able to reproduce certain aspects of Figures 8a, and 8b as done in the paper from Simple, we ran into additional friction on Merger, and our results differ from the paper. For Merger, our reproduction has a constant augmentation of fake nodes, while the runtime of the algorithm seems to correlate with the % of nodes changing their next hop.

We are able to replicate the results of the Case Study in Figure 9b and 10. While the trend are demonstrated to be largely similar, we ran into small discrepancies that are believed to arise from the way we collected our data.

2 BACKGROUND
Traditionally, the problem of inter-network communication was resolved using a distributed protocol such as OSPF[3] or IS-IS[4]. In these protocols, each router would ‘discover’ the topology itself. The software-defined network (SDN) trend, really began with Ethane[1], which focused on securing enterprise networks. With SDN, there’s a centralized control coming from nodes that understand the network topology. These controller nodes then install forwarding rules / routes on the switches in the network. SDN particularly works best in single administrative domain networks since a single entity owns and controls the network. One of the pitfalls of SDN is scalability concerns given the increasing network demands. For example, one of the earlier SDN controllers, NOX, could only serve 30k flow request per second with < 10ms response time.[2] Controller failure is also an issue faced in SDN since the remaining nodes in the network will only follow the exact rules programmed while the controller was up.

3 FIBBING
While SDN is promising in its goal to drastically improve network manageability by providing direct and centralized control over the network forwarding state via straightforward APIs, it has its weaknesses too. Using a network with SDN with high scalability and reliability is difficult. First, it poses challenges for network operators as it often requires numerous modifications to their current network architecture, such as i) equipment upgrade to enable their installed base of network equipment to support SDN protocols, ii) new management to provision system, and iii) the need to train operators to debug and manage a SDN network. Furthermore, in practice, SDN controllers are often difficult to scale and to be reliable. It must compute and install forwarding rules and for all the switches, and respond rapidly to topology changes. The simple task of updating switch rule tables can become a major bottleneck for a controller managing hundreds of thousands of rules in hundreds of switches.

As a result, for SDN-networks to be efficient and scalable, characteristics of distributed routing protocols are essential. Fibbing relies on traditional link-state protocols, i.e. OSPF and IS-IS, where routers compute shortest paths over a synchronized view of the topology. Fibbing also controls routers by "carefully lying" to them, removing the need for direct configuration. It coaxes the routers into computing the target forwarding entries by presenting them with a carefully constructed augmented topology that includes fake nodes and fake links (with fake weights). In the paper, the authors claim that through "careful lying", Fibbing is capable of providing
simple configuration, high manageability, fast response to network failures, and high robustness. Essentially, the best of both worlds.

The Fibbing workflow utilizes two inputs, the desired forwarding graphs (one DAG per destination) and the IGP topology, and has four consecutive phases: Compilation, Topology Augmentation, Optimization, and Injection & Monitoring. In the Compilation stage, Fibbing compiles the requirements into concrete forwarding DAGs. In Augmentation, small augmented topologies are quickly computed from a set of forwarding DAGs using a divide-and-conquer approach. Then, the Optimization stage reduces the augmented topologies while maintaining the forwarding paths. Lastly, the Injection & Monitoring stage transforms fake information into actual "lies" that the controller injects into the network.

In particular, the Augmentation stage introduces two separate algorithms, Simple and Merger, that are used to achieve difference purpose. Simple, the faster one, injects a dedicated fake node for every router that changes its next-hop and can compute augmented topologies within milliseconds. In contrast, Merger is slower but reduces the augmentation size by re-using the same fake nodes to program multiple routers. In the original paper, the authors run Merger in the background to progressively re-optimize augmented topologies. In situations where speed is the foremost concern, such as failure reaction, Simple is employed to achieve quick response rates. In the Figure 8a and 8b, the graphs not only demonstrate the effectiveness of Fibbing, but also the difference within Fibbing’s Simple and Merger algorithms on various metrics.

The original authors provide a prototype of Fibbing in Python (the algorithm part) and C (the interaction with OSPF part) by extending Quagga, a network routing software suite. The code can be found at https://github.com/Fibbing. The prototype consists of three main components: the fake topology generator, link-state translator, and event manager. Using the library, we reproduce figures from the paper.

4 EXPERIMENTS

4.1 Augmentation Algorithm

Figures 8a, and 8b are run on the biggest Rocketfuel topology AS1239 which represents an ISP’s backbone in the early 2000s[5]. The topology has over 300 nodes, and is a directed graph that is a single strongly connected component. Some nodes in the topology have 50 outgoing edges.

To generate Figure 8a and Figure 8b, we ran the Fibbing algorithms Simple and Merger on the Rocketfuel topology using the original fibbing authors implementation, with a tweak to a subroutine for Merger. Both of these algorithms require both a topology (the AS1239 graph), as well as a requirements forwarding DAG for it to fulfill. For Figures 8a, and 8b the forwarding DAG was randomly generated. The steps we used to generate a randomly generated forwarding DAG is as follows:

1. Randomly select a destination node, and then create a forwarding DAG.
2. Run a shortest path algorithm to this destination from every other node.
3. Apply the following mutation to vary the x axis on 8a and 8b:
   a. Select a node not yet modified with out-degree > 1, and randomly select a subset of its edges from the original base graph
   b. If the subset was different from the original shortest Path DAG, and this graph is still a DAG with these randomly selected subset of edges, then we will count that towards ‘% of nodes changing next-hop’

We ran these experiments within a VM similar to the VM specified by the original authors at https://github.com/Fibbing/virtual-machine (we had to change the Debian release from jessie to stretch due to jessie dependencies being broken). The VM runs with a single core with 1GB of memory. Our experiments were ran inside of a Juypter Notebook. We found generating the random forwarding DAG to be time consuming given that we didn’t have a great way of randomly exploring random subsets of edges for a given node. In the end we ended up trying 100 times to find a random subset that would work, otherwise, we’d skip modifying the next hops of that node.

4.2 Mininet

Figure 1: Figure 9 Topology

Figure 9 and 10 demonstrate Fibbing’s practicality through a case study based on a real-life topology (Figure 1). Due to the
The network consists of four routers (Cisco 3700 running IOS v12.4(3)), connected in a square formation where each link has a capacity of 1 Mpbs. Two sources (bottom left) and two destinations (bottom right) are introduced in this network. In this case study, the purple square sends traffic to the purple circle, and red square sends to red circle. The paths are assigned costs, notably (A, C) has the high cost of 10. In such a network, the two flows traverse through the same path, (C, X), and the upper path, (C, A, B, X), is never utilized.

A strict policy is enforced on flow 1 to only be able to traverse (C, X). Furthermore, flow 1 is configured to have a fail-close semantics, whereas flow 2 is to have a fail-open semantics.

5 RESULTS

5.1 Augmentation Algorithm

For Figure 8a, we notice that Simple algorithm runs fairly consistently around 2 seconds, with some occasional peaks, the peaks not seeming to relate to the % of nodes changing their next hop. For Merger, the various percentiles of start out with runtimes similar to Simple but as we increase the % of nodes changing next hop, we see the runtimes drastically diverge, with the 95th percentile Merger being almost 7x slower. For the median runtime of Merger, it is only around 2x slower than Simple. Surprisingly, the 5th percentile of Merger has around the same runtime as Simple.

For Figure 8b, we notice that the Simple algorithm scales sub-linearly as the % of nodes changing their next hop increases. For Merger, across all percentiles, we see the algorithm is fairly flat as the % of nodes changing their next hop increases. The distribution for Merger looks fairly tight.

5.2 Mininet

We reproduced Figure 9b and 10 of the Case Study using mininet. We recreated the topology as seen in Fig. 2 using the setup provided by the original authors.

For Figure 9b, we reproduced the Case Study using the provided topology and tracked the network traffic using iperf. At t = 0, only flow 1 (blue) is introduced. We can see that flow 1 is fully utilizing the bandwidth of the link at around 1 Mbps. At t = 5, we inject flow 2 (orange). Due to the relatively high cost of (A, C), flow 1 and flow 2 share the same route of (C, X). It is clearly that the throughput of each flow is halved to around 0.5 Mbps since the capacity of (C, X) is artificially capped at 1 Mbps using iperf. The Fibbing controller is introduced to the network at t = 16. The controller injects a fake node f1 connected to C and broadcasts a destination for flow 2. This "coaxes" flow 2 to take the upper path (C, A, B, X) and diverts the number of flows on the original link. Within less than a second, we can observe that the throughput of both flows double and utilize each link’s full capacity.

Figure 10 presents an extension of the above Case Study topology to verify Fibbing’s resilience in the event of network failures. At t = 0, the Fibbing has intervened in coaxing flow 2 (orange) to take the upper path of (C, A, B, X) and diverts the number of flows on the original link. Within less than a second, we can observe that the throughput of both flows double and utilize each link’s full capacity.

Using the command line of mininet, different link failures and resurrections are evoked
at various time intervals. At t = 10, link (A,B) was failed. Since the path (C, A, B, X) is no longer viable, the controller removes the injected fake node f1. Flow 2 is disrupted for about 1 second before recovering to take (C, X). Such link failure drops the throughput of both flows. When (B,X) is failed at t = 20, flow 1 is blackholed as consistent with the failure semantics of Fibbing to prevent the IGP from routing it over paths that violate the prescribed policies. Flow 2 stays on because it is using IGP’s shortest path. It took about 1 to 2s for flow 2 (orange) to recover. When we up-ed (A,B) at t = 35, it took about 0.5 seconds for flow 1 to re-establish connection and re-optimize its routing. Similar event happened when (B,X) was brought back up at t = 50. We see that Fibbing controller has re-distributed the paths amongst the nodes, thus reaching a similar level as t from 0 and 10.

6 ANALYSIS OF RESULTS

6.1 Augmentation Algorithm

6.1.1 Figure 8a. Our reproduction of Figure 8a for the Simple runtime is similar to the results presented in the Fibbing paper[6]. The magnitude of time is fairly different, but that might be due to differing hardware and that our results are produced within a VM. For Merger, however, our results differ from those presented in the original paper.

Our reproduction only goes to around 80% of nodes changing their next hop, while the original paper goes up to 100% of nodes changing next-hop. On our end, this is due to us skipping some nodes hop changes if we cannot easily find a modification that abides by the algorithm for generating random forwarding DAGs. It’s interesting to note that in the original paper, Figure 8a and Figure 8b have different axes (i.e. differing % of nodes changing next-hop). Perhaps the original authors also faced difficulties generating random forwarding DAGs.

The other deviation from the original figure, is that our runtimes for Merger increase as the % of nodes changing next-hop increases. While we aren’t entirely sure what the exact reason for this deviation is, we suspect it might be related to minor tweaks to Merger as it was failing to run on larger topologies using our random forwarding DAG generation, but would work on smaller topologies (such as those used in the unit test which we also tested).

6.1.2 Figure 8b. Our reproduction for Figure 8b for the Simple algorithm somewhat matches the original figure. In the original paper, Simple was fairly linear, while in our reproduction it appears to be sub-linear.

The bigger deviation is with regards to Merger, unlike the original paper, our reproduction of Merger has roughly a constant percentage of \( \frac{\text{fake nodes}}{\text{total nodes}} \), which is surprising. We suspect that our slight modification of Merger changed how the algorithm collapses local lies into global lies, particularly the algorithm ends up using the same number of global lies, and has to ‘search’ longer as the % of nodes changing their next hop increases. This is why our reproduction of Figure 8a for Merger has an increase in runtime as % of nodes changing their next hop increases.

6.2 Mininet

6.2.1 Figure 9b. Given the straightforwardness of Figure 9b, we were able to easily replicate the same trend as exhibited in the original paper. It demonstrates that Fibbing does work and provides the remedy it promised in the case of congestion in the Case Study topology.
6.2.2 Figure 10. The overall trend of Figure 10 is reproduced successfully, demonstrating that Fibbing is reacting correctly to various forms of network failures. When the second link (B, X) is brought down at \( t = 20 \), the controller removes the injected lie and flow 1 is blackholed, causing the throughput to completely drop out. This behaviour is subject to change depending on what policies are enforced on the SDN; however, in this experiment, such behaviour should be anticipated. One of the biggest deviation from the original graph occurs in between 25s to 35s. In the reproduced result, flow 2 has a consistent throughput of around 1 Mbps, whereas flow drops about 0.3 to 0.4 Mbps midway through that range. Furthermore, from 45s to 50s, both flows demonstrate much less fluctuation in throughput compare to the original graphs. One speculates that such discrepancies arise as a result of the differences in the rates in which throughput data is collected. In the reproduced result, data is collected in intervals of 0.5 seconds. In contrast, the original seems to collect data much more frequently. Given more time, one would re-do both the case studies to collect data in more frequent intervals to verify that more fluctuations in throughput are present in the experiment.

7 LIMITATIONS

While Fibbing works on smaller topologies as our mininet case studies show, we expect it to be difficult to debug in a deployed setting with more nodes – particularly due to the additional fake nodes / ‘fibs’ which might increase cogitative load on network operators. Moreover, in order for fibbing to work at all, the network needs to have a minimum cost between hops that fibbing can undercut.

Regarding our reproduction, we acknowledge the following limitations:

- The original Fibbing ran the case studies on real routing equipment, particularly Cisco 3700, while all of our case studies were done within mininet.
- We skip over nodes for Figure 8a and Figure 8b if we cannot settle upon a random subset of outgoing edges when modifying the forwarding DAG. We suspect the authors of Fibbing did something similar (leading to Figure 8a and Figure 8b in the paper having different x-axes).
- We slightly modified a subroutine used in the Merger algorithm which might both allowed the algorithm to run on AS1239, but might have led to it diverging slightly from the original paper.

8 CONCLUSION AND FUTURE WORK

If we had additional time, we would have tried to explore how Fibbing works under different workloads as the Case Study presented in the original paper were small ‘toy’ examples, to prove that the concept could work. Additionally we would try to capture the difficulty of debugging a network controlled by Fibbing vs a network controlled by a traditional SDN with a centralized node. We believe the operation cost of Fibbing from that perspective could make it difficult to use in practice.

Furthermore, another extremely fascinating experiment that we would love to conduct is how Fibbing can be used to improve network performance and user experience in the case of video streaming. Such experiment is inspired by another paper by the original authors, titled Fibbing in action: On-demand load-balancing for better video delivery. In particular, we would like to replicate Figure 2[7], which demonstrate the throughput evolution with Fibbing during video streaming. Figure 2 is another case study on a slightly more complicated topology, but demonstrates that Fibbing is able to leverage is unique features to alleviate congestion.

Overall, Fibbing is a very interesting system that does not fib about its ability to reduce congestion and improve network performance. It maximizes the flexibility within both centralized and distributed systems to achieve better results. We are curious to see Fibbing being applied to larger networks in the future and to examine how it scales and reacts to the unpredictability of real-life.

REFERENCES