Sibyl: a Framework for Evaluating the Implementation of Routing Protocols in Fat-Trees

Tommaso Caiazzi*, Mariano Scazzariello*, Leonardo Alberro†, Lorenzo Ariemma*
Alberto Castro†, Eduardo Grampin†, and Giuseppe Di Battista*
*Roma Tre University – Rome, Italy †Universidad de la Republica – Montevideo, Uruguay

***Abstract*—Several data centers adopt fat-tree topologies, where high bisection bandwidth is achieved by interconnecting commodity hardware and by using specific routing solutions. These solutions, which include protocol implementations and configurations, are difficult to evaluate and test both for the density of fat-trees and for the complexity of the protocols. Also, since most issues show up only when a fault happens, it is unfeasible to perform such tests in a production environment. Additionally, the lack of standard testing procedures motivates an effort in developing solutions for such a critical task. In this paper, we propose a methodology devised for testing fat-tree routing protocol implementations. It adopts a wall-clock independent method to establish metrics, which permits normalizing the results of different routing protocol implementations independently from the execution environment. The methodology is implemented by Sibyl, a software framework developed to perform repeatable tests on arbitrary fat-tree topologies automatically. Sibyl also provides a set of tools to analyze the results and investigate implementation behaviors. We evaluate the methodology and Sibyl in three use cases. Such use cases witness a wide spectrum of situations where Sibyl is effective for analyzing, comparing, developing, and debugging routing protocol implementations.**

***Index Terms*—Routing protocols, Testing, Data center, Fat-tree**

I. Introduction

Massive Scale Data Center (MSDC) architectures have evolved towards topologies that seek to guarantee a large bandwidth among all servers. The MSDCs traffic is fundamentally East-West (among servers) and does not respond to the typical statistical multiplexing of the internet, as a consequence of the applications that run in the data center, where distributed computation and replication are fundamental elements. Hence, data center design seeks to guarantee constant bisection bandwidth. This requirement led to the widespread of *fat-trees* [1], [2], a particular case of a Clos network [3], where high bisection bandwidth is achieved by interconnecting commodity switches. The adoption of fat-trees by several OTTs [4]–[6], combined with the intrinsic complexity of the network, the wide variety of routing protocols with heterogeneous implementations, and the fact that a good portion of failures is caused by software bugs [7]–[9], justify the development of methodologies and tools for assessing routing protocol implementations in fat-trees. This paper gives two main contributions. First, we propose a methodology. Namely, we developed a wall-clock independent method to establish metrics, which permits normalizing the results of different routing protocols and implementations, independently from the execution environment (i.e., the underlying hardware). Second, we present Sibyl, an open-source framework that implements the methodology and that has the following features: (1) it runs real routing daemon implementations to hunt bugs in the software; (2) it allows to perform interoperability tests between different implementations; (3) it allows to test implementations on real-scale networks since some bugs show up only in large topologies; (4) it allows to build an automatic testing pipeline to evaluate implementations and support their development.

Finally, we present three use cases to show the effectiveness of the methodology and Sibyl. In UC1, we analyze a well-known BGP (Border Gateway Protocol) implementation. In UC2, we show how the framework can be exploited to implement a feature in a new protocol proposal, considering the case of RIFT (Routing in Fat Trees) [10]. Also, we show how Sibyl can be exploited for evaluating prototypical implementations and, as a byproduct, we highlight the potential of RIFT. UC3 focuses on the debugging of a vendor implementation of RIFT. All the proposed experiments are fully reproducible.

The paper is organized as follows. A description of fat-tree topologies is in Sec. II. In Sec. III, we describe our methodology, and in Sec. IV, we present Sibyl along with our testing environment. Sec. V shows three different use cases witnessing a wide spectrum of situations where the methodology and Sibyl show their effectiveness. Sec. VI is on related work. Concluding remarks and future work are in Sec. VII.

II. Fat-Tree Data Center Topologies

A *fat-tree* is a hierarchical topology where nodes are assigned to levels, and nodes of adjacent levels are connected to form a tree with redundant links. Since in recent data centers, the fat-trees have typically three *levels* [4], [5], we concentrate on this case. We adopt the terminology used in the RIFT protocol draft [10]. For a more comprehensive description, we refer readers to [11] and [12] (and the references therein).

A *Leaf* is a node at level 0 that is connected to servers and has northbound adjacencies with Spines. We denote by K_{LEAF} the number of its ports pointing north. A *Spine* is a node at level 1. Also called *Top of PoD* (*ToP*), it is connected southbound with Leaves and northbound with ToFs. We denote

by K_{TOP} the number of its ports pointing north or south. A *Point of Delivery* (*PoD*) is a set of Leaves that are connected to the servers, plus a set of Spines fully interconnected to the Leaves. A *Top of Fabric* (*ToF*) is a node at level 2 that provides inter-PoD communication and has no northbound adjacencies. A ToF is connected to at least one Spine per PoD. We denote by R the *redundancy factor*, i.e., the number of links from a ToF to a PoD. We assume $K_{LEAF} = K_{TOP} = K$. Hence, we denote a fat-tree with a pair (K, R).

There are two types of fat-trees: single plane and multi-plane. In a *single plane* topology, each ToF is connected to all the Spines. This topology has the maximum value of redundancy factor, with $R = K$. In a *multi-plane* topology, ToFs are partitioned into $N = K/R$ sets (R being a divisor of K), each with the same number of nodes, called *planes*. The Spines of a PoD are partitioned into N sets of R nodes. All the Spines of the same set are connected to all the ToFs of the same plane, and vice versa. In this configuration, the network's redundancy is reduced to increase the maximum number of supported PoDs (and hence the number of servers). Notice that, with $R = 1$, if a link between a ToF and a Spine fails, the ToF loses the connectivity to the PoD of that Spine.

III. The Methodology

This Section describes the methodology we propose for evaluating fat-tree data center routing protocol implementations (in what follows, just *implementations*). It is based on the following milestones. 1) Fat-trees have regular topologies. This allows performing tests focused on specific topology elements to obtain results with a general value. 2) Performing tests that measure actual times can be misleading since timings heavily depend on the used hardware. Also, assuming that the system clocks of nodes are perfectly synchronized is not realistic. Hence, we evaluate the implementations being oblivious as much as possible concerning the actual timings. 3) Routing protocols can be conceptually very different. For this reason, we adopt a black-box method, disregarding, as much as possible, the internals of the implementations.

Tests. The methodology includes *failure* and *recovery* tests. Failure-tests induce many types of common faults in data centers [13], and measure how implementations react to those faults. Recovery-tests measure how implementations react when the network is restored. For each failure test, we: 1) start the fabric, waiting for convergence; 2) cause the failure; 3) capture all the Protocol Data Units (PDUs) until protocol convergence. We perform the recovery tests analogously, focusing on restoring the regular operation. The methodology only considers single node failures to track and analyze the basic behaviors of implementations. More specifically, we consider the ability of an implementation to converge in the following cases. **Node Failure:** A single node failure. We consider the failure of a Leaf, a Spine, and a ToF. **Node Recovery:** Return to normal operation when a Leaf, a Spine, or a ToF is restored after a failure. **Link Failure:** A single link failure. We consider failures of a Leaf-Spine link and a Spine-ToF link. **Link Recovery:** Return to normal operation when a Leaf-Spine or a Spine-ToF link is restored after a failure. **Partitioned Fabric:** Occurs when a ToF node is completely severed from access prefixes of an entire PoD by multiple link failures. The aim is to verify the ability of an implementation to react to a Partitioned Fabric.

Coordinates of the Analysis. For each test: 1) We check the *Convergence* to verify that, when an event occurs, the fabric reaches the expected state. 2) We measure the *Messaging Load* injected into the network. 3) We focus on *Locality* to verify how "local" is such an overhead. 4) We focus on the *Number of Rounds* used by the implementation to converge. To be agnostic to the internal operations of the implementations, Messaging Load, Locality, and Number of Rounds analyses rely only on the signaling packets captured during the experiments. Such packets have different features. First, they are either injected into the network as a consequence of an event (e.g., BGP Update packets) or exchanged independently on events in the network (e.g., keep-alive packets). To analyze the reaction to a specific event, the methodology only considers the packets that carry information related to it. Second, they may travel on different levels of the network stack (e.g., on TCP or directly on Ethernet). Lower-level packets that are useless to track the reaction to a specific event (e.g., TCP SYNs or ACKs) are not considered.

Convergence. This aspect aims to check that, after an event, the nodes' data plane forwarding tables reach the expected state. If the check fails after a maximum number of attempts, the test is aborted. Observe that the convergence check can be implemented in several ways, e.g., dumping the forwarding tables [14] or using formal verification [6].

Messaging Load. The purpose of this part of the analysis is to determine if the overhead grows smoothly with respect to the main features of the topology: the values of K and R, the number of nodes and interfaces. The methodology adopts two different metrics to measure the overhead injected by the protocol implementation. Namely, it *counts the packets* originated by a test and *computes their aggregate size*. In computing the size, it only considers the protocol payload.

Locality. To evaluate the ability of an implementation to limit the "blast radius" of a failure or a recovery, the methodology adopts the following algorithm. First, for each link of the network, the number of packets received by the interfaces of that link is computed. Second, a *topological distance* is associated with each link of the network, computed as the distance from the event (failure or recovery). Namely, if the event involves a node, all its incident links have a topological distance of 0. If the event involves a link between node u and node v, where node u is the lower level, then all the links incident to u, including the failed/recovered link, have topological distance 0. In both cases, for all the other links, the topological distance is the minimum number of links to be traversed to reach a link with topological distance 0. Third, the total number of packets received from all the interfaces having a certain topological distance is computed. The result is a vector L, where $L[i]$ with $i = 0, 1, 2, 3$ is the number of packets captured at distance i. To summarize L

into a scalar value (called *blast radius*), the scalar product is computed between L and vector W where $W[i] = i + 1$ with $i = 0, 1, 2, 3$. This gives a larger weight to packets that are far from the event and allows capturing how the "wave" of packets originated by an event spreads into the fabric.

Rounds. The way Locality is considered allows to perceive the "blast radius" of an event but gives incomplete information on how the above-mentioned wave propagates. Namely, suppose that the links at a certain topological distance receive a total amount of m packets, the above-introduced Locality does not give any information about the number of "rounds" when this happens: the m packets could be received all at the same time or in different cycles of waves and "backwashes".

However, considering the succession of waves and backwashes, being oblivious as much as possible with respect to the actual timing is challenging. To do that, the methodology exploits a *node-state graph*, which describes how the states of nodes are synchronized by routing protocol packets.

Suppose to have a directed graph G, related to an experiment, such that a vertex of G is a pair $\langle v_i, s_j \rangle$ representing a node v_i of the fabric in a state s_j. An edge from a vertex $\langle v_i, s_j \rangle$ to a different vertex $\langle v_h, s_k \rangle$ represents the fact that a packet exiting from node v_i when it is in state s_j contributes to change the state of node v_h into s_k. Of course, if after the experiment the implementation converges, then G is acyclic. Graph G would give a clear description of the cause-effect relationships between state changes of nodes. Also, one could extract several useful information from G: e.g., computing the number of its vertices, one could estimate the number of state changes induced by an event. Further, computing the longest path of G (since G is acyclic, this is doable) would give the *number of rounds* an implementation took to converge after an event: a measure of efficiency that is independent of time.

Unfortunately, computing G requires entering the features of the protocols and their implementations, something that we want to avoid as much as possible. Hence, we compute G "from outside", observing the packets exchanged by nodes, with the following algorithm (see Fig. 1(a)).

Indeed, even if we are oblivious with respect to the global network time, we can use the local system clock of each node to order packets sent/received by/from that node. Let n be the number of nodes of a fabric. For each node v_i ($i = 1, \ldots, n$), we sort the packets entering or exiting v_i according to the local time they are recorded, obtaining an *ordered list* λ_i. Fig. 1(b) shows the ordered lists that led to the computation of the graph in Fig. 1(a). We call this type of diagram *node-state timeline*. Each list λ_i is a sequence of points along the horizontal timeline associated with the corresponding node v_i. Each packet p is represented by: i. a point on the line of the node that sends p, ii. a point on the line of the node that receives p, and iii. an arrow connecting those two points. We call $pred(p, \lambda_i)$ the predecessor of packet p in list λ_i, if any.

We partition each λ_i into lists of maximal sets of consecutive entering or exiting packets, called *receiving groups* and *sending groups* of v_i, respectively. Hence, we can write $\lambda_i = \mathcal{R}_{i,1}\mathcal{S}_{i,1} \ldots \mathcal{R}_{i,h_i}\mathcal{S}_{i,h_i}$, where the $\mathcal{R}_{i,j}$ and the $\mathcal{S}_{i,j}$ ($j = 1, \ldots, h_i$) are the receiving groups and the sending groups of v_i, respectively, and where $\mathcal{R}_{i,1}$ and $\mathcal{S}_{i,h_i}$ may be empty. E.g., in Fig. 1(b), `tof122` has a receiving group $\mathcal{R}_{\texttt{tof122},1}$ containing the first two packets, a sending group $\mathcal{S}_{\texttt{tof122},1}$ with the following four packets, and a receiving group $\mathcal{R}_{\texttt{tof122},2}$ containing the last two packets. Notice that $\mathcal{S}_{\texttt{tof122},2} = \emptyset$. An example of node whose first receiving group is empty is `spine111` that has $\mathcal{R}_{\texttt{spine111},1} = \emptyset$.

We associate with each list λ_i a sequence of states: (1) If the first group of λ_i is a sending group (namely, $\mathcal{S}_{i,1}$) we associate a state s_1 to v_i. (2) For each pair $\mathcal{R}_{i,j}$ and $\mathcal{S}_{i,j}$, we associate a state s_j to v_i. (3) If the last group of λ_i is a receiving group (namely, $\mathcal{R}_{i,h_i}$) we associate a state s_{h_i} to v_i. As an example, `tof122` has two states, s_1 associated with $\mathcal{R}_{\texttt{tof122},1}$ and $\mathcal{S}_{\texttt{tof122},1}$, and s_2 associated just with $\mathcal{R}_{\texttt{tof122},2}$.

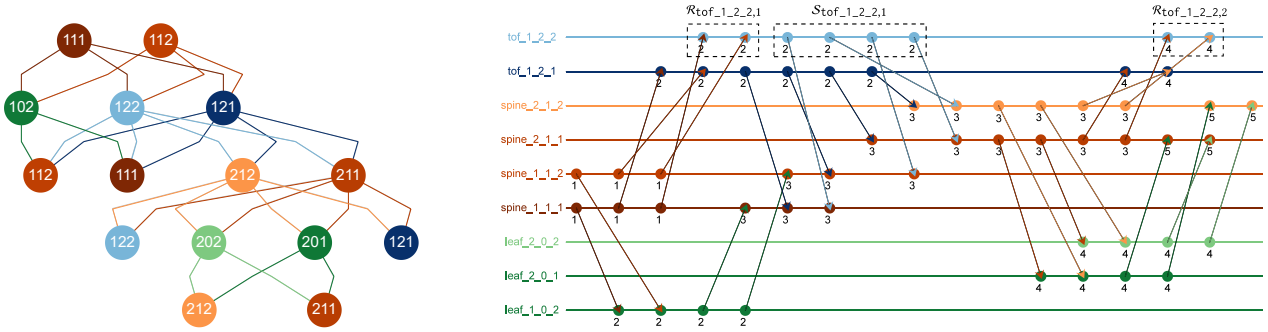
Exploiting the above-defined states, we construct a directed graph G as follows (see Fig. 1(a)). Each vertex of G is a pair $\langle v_i, s_j \rangle$ where the first element is a node of the fabric and the second element is a state. For each $\mathcal{S}_{i,j}$, consider each packet p sent from v_i to a node v_k of the fabric; G has an edge $(\langle v_i, s_j \rangle, \langle v_k, s_g \rangle)$ where g is the index of the corresponding receiving group $\mathcal{R}_{k,g}$ of v_k that contains p (i.e. $p \in \mathcal{R}_{k,g}$). As an example, the state s_1 associated with `tof122` corresponds to the vertex $\langle \texttt{tof122}, s_1 \rangle$, that has two incoming edges, representing the packets of $\mathcal{R}_{\texttt{tof122},1}$, and four outgoing edges, representing the packets of $\mathcal{S}_{\texttt{tof122},1}$. Furthermore, the vertex $\langle \texttt{tof122}, s_2 \rangle$ has two incoming edges and no outgoing edges, since the receiving group $\mathcal{R}_{\texttt{tof122},2}$ is the last group in $\lambda_{\texttt{tof122}}$. In Fig. 1(a), $\langle \texttt{tof122}, s_1 \rangle$ and $\langle \texttt{tof122}, s_2 \rangle$ correspond to the two light-blue vertices labelled with `122`. We call G *node-state graph*. For the node-state graphs figures, we follow the above convention, labeling vertices only using the node number, hiding the type of the node and the state.

At this point, we compute, for each vertex v of G, the number of rounds as the length of the longest path from a source of G to v. This is the number of state changes that led to v. We call it *state-round value* of v. Also, considering all the vertices of G, we can compute the maximum of the state-rounds of the vertices. We call it *state-round value* of G. In Fig. 1(a) the length of the longest path, and hence the state-round value of the graph, is 4. Fig. 1(b) depicts the result of the algorithm. The label of each packet p is the assigned round number. Notice that, the maximum round value is 5 since it is computed considering the number of vertices.

Notice that the steps an implementation takes to converge and the set of packets the nodes exchange may change depending on the network parameters, the processing time of nodes, and the presence of timers in the protocols. So, given an experiment, the node-state graph and the state-round values are not univocally defined. This implies that several executions of the same experiment are recommended to analyze an implementation behavior.

IV. The Sibyl Framework

Performing the tests described in Sec. III in a physical data center would be unfeasible due to the costs of the required



(a) Node-state graph.

(b) Node-state timeline. It shows the packets exchanged during the experiment.

Fig. 1. The reaction of a BGP implementation to `leaf101` failure in a $(2, 2)$ topology.

equipment [7] and the impact on the quality of the hosted services. Moreover, performing the experiments in a physical fabric would require manually changing the wiring at every test. This would limit the automation and reproducibility, leading to a more error-prone testing pipeline.

For these reasons, we use a virtual environment that gives the flexibility to scale the topology size with limited constraints and allows to automate the testing stage. The main drawback is the impossibility of analyzing the actual wall-clock time since it heavily depends on the virtualization environment [15]. However, as explained in Sec. III, our methodology aims to be agnostic from temporal aspects.

Sibyl is developed in order to perform a large number of experiments on several topology configurations. It exploits the tools described below to automatically deploy and run the tests proposed in Sec. III. During each experiment, Sibyl performs the following steps: 1) generates a topology; 2) deploys a set of containers and of virtual links representing the topology; 3) begins, in each container, to capture all the generated PDUs on all the interfaces; 4) starts the routing daemons on all the nodes with the suitable configuration; 5) performs the convergence check; 6) performs the required actions for a specific type of test; 7) performs the convergence check again; 8) ends the capture of PDUs and gathers the obtained `.pcap` files; 9) shutdowns the containers; and 10) performs the analysis and outputs the results. Notice that the analyses of Step 10 may be parallelized on all the results at a later stage.

Tools. Sibyl assembles existing tools with new ones.

VFTGen [16] automatically generates and configures virtual fat-tree topologies. It takes as input the parameters of a fat-tree and generates as output a directory containing all the files needed to run the corresponding topology on Kathará.

Kathará [17] is a container-based network emulation system. It is the only available open-source emulator that supports Kubernetes as a container orchestration system [18], and it can leverage on pure L2 networks. Hence, it allows emulating very large virtual networks in a distributed environment faithfully.

Sibyl RT Calculator is a tool for generating the forwarding tables of a fat-tree. It takes as input a topology, a protocol, and a type of test, and returns the routing information for each node, containing the server farm prefixes and the number of next hops that are expected to be computed.

Sibyl Agent is a REST Service composed of a controller and several worker agents. An instance of worker is deployed on each node of the fat-tree. The controller agent can perform actions on the nodes through the worker agents.

Sibyl Analyzer is a tool to analyze and plot the results of the experiments. Its input is the set of `.pcap` files containing the packets exchanged by the nodes during an experiment. It computes the metrics of Sec. III, saving them in a `.json` file. It also generates three interactive `.html` files containing the topology labeled with the number of packets on each link, the node-state timeline, and the node-state graph. Sibyl Analyzer can also be used to analyze data captured on physical nodes.

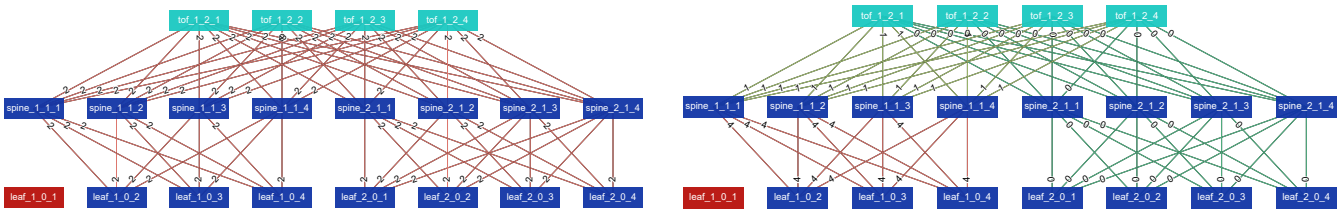
Execution Environment. We tested Sibyl and its scalability on the topologies in Fig. 3. To emulate topologies with up to 320 nodes and $\sim 4.5k$ interfaces (black border boxes), we used a local virtual cluster composed of 22 VMs, each with 2-core vCPUs and 8GB of vRAM. To emulate larger topologies (red border boxes), we used the Azure Cloud with a cluster composed of 160 VMs, each with 4-core vCPUs and 8GB vRAM. In these tests, we emulated topologies with up to $1,280$ nodes and $33k$ interfaces. None of the experiments took less than 2 mins and more than 2 hours of execution time.

V. Evaluation

In this section, we illustrate 3 use cases showing the Sibyl effectiveness. In all use cases, we performed all the tests of Sec. III on several topologies. Because of space limitations, we discuss only Leaf-node failures, which are the most disruptive since they imply the unreachability of a set of prefixes. The results of the other tests are briefly mentioned. We performed the described tests 5 times and computed the average of the results. The full set of results is available at [19].

Tested Implementations. We list here the protocols and the implementations tested in our use cases.

BGP is the de-facto standard for Clos-based data centers [5], so considering a use case involving BGP is mandatory. To work in fat-trees, BGP requires the tweaks defined in [20] and specified in [21]. We consider the open-source BGP implementation of FRRouting [22]–[24] (fork of Quagga [25]).



(a) Packets exchanged by FRRouting BGP.

(b) Packets exchanged by RIFT-Python.

Fig. 2. A $(4, 4)$ Leaf-node failure (`leaf101`). Each link is labeled with the number of packets captured on it.

RIFT [10] is a fat-tree-specific routing protocol under development. RIFT is a hybrid link-state/distance vector protocol that aims at minimizing configuration and operational complexity. It includes a novel mechanism of automatic disaggregation of prefixes in case of failures. There is only one open-source implementation, RIFT-Python [26], and one advertised vendor implementation by Juniper [27]. Since we have access to both implementations, we include them in our evaluation, using the Zero Touch Provisioning configuration mode.

UC1: An analysis of a well-known BGP implementation (FRRouting BGP). We first analyze the FRRouting BGP implementation, performing the tests on all the topologies in Fig. 3. The metrics were only computed on the Update packets, discarding keep-alives and all the TCP acknowledgments.

Fig. 3 shows the results. The x-axis represents R and the y-axis represents K. The color intensity of the blue heatmap in the background is proportional to the number of nodes in a topology with the corresponding K and R.

Messaging Load. Fig. 2(a) shows that the number of packets captured on each link in a $(4, 4)$ topology is 2. Analogous results have been obtained in all the other topologies. Fig. 3 confirms that, from the Messaging Load coordinate, the implementation has good scalability since it does not manifest any performance degradation varying K and R. Moreover, we found that the number of packets exchanged by nodes is not affected by the number of nodes in the network, but it only depends on the number of interfaces of each node.

Locality. As stated in the Messaging Load analysis, the BGP implementation floods 2 packets on all links, affecting all the nodes (Fig. 2(a)). First, BGP does not contain an event locally, spreading the information to the entire fabric. Second, the number of packets traversing a link is constant.

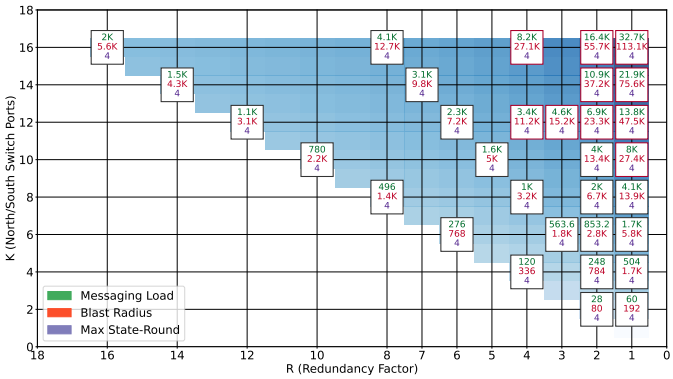


Fig. 3. FRRouting BGP analysis in Leaf-node failure tests.

Rounds. Fig. 3 shows the number of state-rounds in the node-state graph for each topology, highlighting that this implementation has a very stable behavior, with a value of 4 on all the topologies. Probably, the steadiness of the results is influenced by the regularity of fat-trees and the virtual environment, where latency is negligible.

Comparable results are obtained in the other type of tests (see [19]), demonstrating the stability of FRRouting BGP.

In conclusion, UC1 confirms that all the devised metrics give a different point of view that can be exploited for evaluating implementations. Also, tests on larger topologies prove that Sibyl can be useful for testing in virtual MSDCs.

UC2: Working on the PoC of a new routing protocol (RIFT-Python). UC2 shows how Sibyl can be used to support the implementation of new features and to evaluate prototypes.

We focus on RIFT-Python [26]. We found that it has critical scalability issues, converging only on topologies up to $(6, 3)$. The reason is that RIFT-Python has been developed to check the correctness of the IETF draft without having scalability as a goal. Second, we found that RIFT-Python was missing the negative disaggregation feature, one of the main novelties of RIFT, needed to handle failures in multi-planes (for details, refer to [10]). So, we implemented it, building an integration testing pipeline as follows: 1) we selected the scenarios where the negative disaggregation is needed, e.g., partitioned fabric; 2) we tested code changes in the above scenarios; 3) we analyzed the node-state timeline to verify the interactions between nodes and the expected properties of the node-state graph, to confirm that changes did not cause bugs.

As a byproduct, we analyzed how RIFT performs against the BGP implementation tested in UC1 on the topologies where RIFT-Python converges. The comparison is fair since our metrics do not take into account temporal-tied aspects, allowing normalization of the results that enable the comparison of implementations that are quite different from each other.

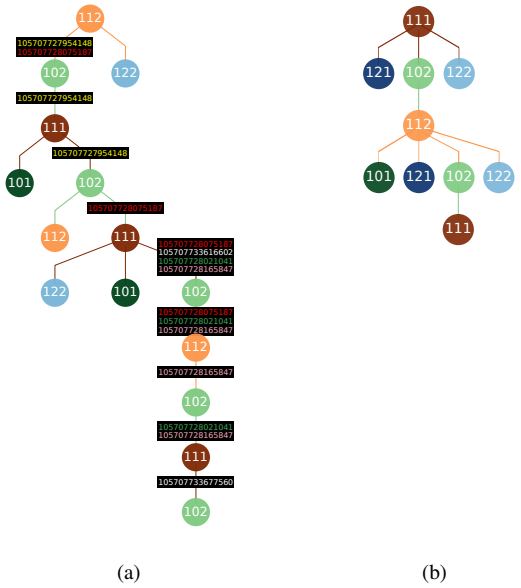
The analysis only considers TIEs (Topology Information Element) packets, discarding LIEs (Link Information Element), TIDEs (Topology Information Description Element), and TIREs (Topology Information Request Element), since they are periodic. We have that, even if it is a prototypical implementation, RIFT-Python exhibits good results, especially

in terms of packet number and blast radius (see Fig. 2(b)), showing the protocol potential. However, there is performance degradation, as the topology scales up, that cause a significant jump between the values obtained for $(2, 2)$ and $(6, 6)$ topologies. This happens in the state-round values that range from 3.4 on $(2, 2)$ to 13.8 on $(6, 6)$. Theoretically, RIFT should perform the same operations, generating the same state-round value. Detailed results can be found at [19].

In conclusion, the use case shows that Sibyl is a valid support during the development and that the methodology can also be exploited to evaluate prototypical implementations.

UC3: Checking a vendor implementation (cRPD-RIFT). This use case proves that Sibyl can be used to find inefficiencies and bugs in production-grade implementations. In particular, we analyzed the RIFT implementation by Juniper [27], looking for anomalies that are not detected by unit tests. We had at disposal only the containerized implementation, called *cRPD-RIFT*, with no chance to inspect the source code.

Usually, when an inefficiency or a bug is detected, the root-cause analysis is done by examining logs, either manually or automatically. In large fat-trees, this could result in an error-prone task since information coming from all the nodes should be integrated to get a "global view" of the network interactions. Instead, the node-state graph already provides this "global view", resulting in a valuable support during debugging regarding the standard procedures. For example, we spotted a flooding inefficiency by analyzing a Leaf-node failure in a $(2, 2)$ topology. Analyzing the graph in Fig. 4(a), it is easy to notice a repeated exchange of messages between Leaf `102` and Spine `111`. This is evidence of a possible inefficiency of the flooding mechanism. Going deeper, analyzing the edges (labeled in Fig. 4(a)) between the two nodes, we found some packets with the same sequence number that are bounced by Leaf `102` to Spine `111`, and vice versa. The vendor confirmed the presence of the inefficiency and fixed it. Fig. 4(b) shows the graph of the fixed version.



(a) (b)

Fig. 4. cRPD-RIFT node-state graphs for a Leaf-node failure on $(2, 2)$. (a) A flooding inefficiency. Edges representing packets involved in the inefficiency are labeled with the corresponding sequence numbers. (b) After the fix.

VI. Related Work

Network control plane debugging may be performed using two main approaches: model-based verification and emulation-based testing. Several network control-plane verification systems have been developed. The interested reader can examine [28] for an exhaustive survey on network verification and testing with formal methods. Nevertheless, none of these tools can consider the impact of software bugs since they perform static configuration analysis and assume correct, bug-free implementations. Bugs have a non-negligible impact on data center failures; e.g., in [9] it is shown that 12% of failures in Facebook data centers are due to software or firmware bugs.

On the other hand, in emulation-based testing, network devices run unmodified network firmware, and therefore they are exposed to real software bugs. CrystalNet [7], proposed by Microsoft, highlights the need for a system to test and verify large-scale data center networks, and they leverage on emulation, which guarantees the execution of real routing daemons, scaling up like a real production network. Our approach has similarities with CrystalNet, building an emulation facility for real implementation testing. However, CrystalNet does not propose any methodology to evaluate protocol implementations. Also, it is mainly designed to validate the Microsoft Azure network configurations before applying them in the production environment. Finally, it is not open-source.

VII. Conclusions and Future Work

Currently, there aren't standard methodologies and tools for testing data center routing protocol implementations that do not depend on the wall-clock time, the execution environment, or the implementation internals. In this paper, we addressed this issue, presenting a methodology targeted to fat-trees, and Sibyl, a framework providing a flexible testing environment.

Additionally, Sibyl may support network management decisions, enabling to verify network configurations and automation, and to test management response to network events.

We evaluated the methodology and Sibyl, proving their effectiveness. The evaluation: (i) confirmed the stable behavior and the good scalability of the FRRouting BGP implementation; (ii) highlighted the usability of Sibyl for implementing protocol features; (iii) showed the potentiality of RIFT, even if the evaluated implementation is a prototype; (iv) proved that Sibyl could be useful during development and debugging.

Future works could concern the extension of the methodology and Sibyl to support different topologies (e.g., Jellyfish [29], BCube [30]) and routing solutions (e.g., SDN and programmable switches). Further, the methodology may consider tests involving multiple node and link failures.

Moreover, the analysis on the node-state graph and the node-state timeline could be extended by exploiting distributed systems theory [31] to verify formal properties.

Reproducibility. Sibyl is open source and available at [32]. The results of the evaluation are available at [19]. The figures of this paper have been automatically produced by Sibyl.

Acknowledgement. We thank Dr. Antoni Przygienda and Bruno Rijsman for their precious help during this work.

REFERENCES

[1] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, Oct 1985.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1402958.1402967

[3] C. Clos, "A study of non-blocking switching networks," *The Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, March 1953.

[4] LinkedIn Engineering. (2016) The LinkedIn Data Center 100G Transformation. [Online]. Available: https://engineering.linkedin.com/blog/2016/03/the-linkedin-data-center-100g-transformation

[5] A. Abhashkumar, K. Subramanian, A. Andreyev, H. Kim, N. K. Salem, J. Yang, P. Lapukhov, A. Akella, and H. Zeng, "Running BGP in Data Centers at Scale," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 65–81. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/abhashkumar

[6] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," *SIGPLAN Not.*, vol. 51, no. 1, p. 69–83, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2914770.2837657

[7] H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "CrystalNet: Faithfully Emulating Large Production Networks," in *SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, October 2017, pp. 599–613. [Online]. Available: https://www.microsoft.com/en-us/research/publication/crystalnet-faithfully-emulating-large-production-networks/

[8] R. Singh, M. Mukhtar, A. Krishna, A. Parkhi, J. Padhye, and D. Maltz, "Surviving Switch Failures in Cloud Datacenters," *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 2, p. 2–9, May 2021. [Online]. Available: https://doi.org/10.1145/3464994.3464996

[9] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu, "A large scale study of data center network reliability," in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 393–407. [Online]. Available: https://doi.org/10.1145/3278532.3278566

[10] A. Sharma, P. Thubert, B. Rijsman, D. Afanasiev, and T. Przygienda, "RIFT: Routing in Fat Trees," Internet Engineering Task Force, Internet-Draft draft-ietf-rift-rift-15, Jan. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-rift-rift-15

[11] D. Medhi and K. Ramasamy, *Network Routing, Second Edition: Algorithms, Protocols, and Architectures*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[12] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data Center Network Virtualization: A Survey," *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 909–928, Second 2013.

[13] P. Gill, N. Jain, and N. Nagappan, "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 350–361. [Online]. Available: https://doi.org/10.1145/2018436.2018477

[14] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma, "Validating datacenters at scale," ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 200–213. [Online]. Available: https://doi.org/10.1145/3341302.3342094

[15] V. Babu and D. Nicol, "Precise virtual time advancement for network emulation," in *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 175–186. [Online]. Available: https://doi.org/10.1145/3384441.3395978

[16] T. Caiazzi, M. Scazzariello, and L. Ariemma, "VFTGen: a Tool to Perform Experiments in Virtual Fat Tree Topologies," in *IM 2021 - 2021 IFIP/IEEE International Symposium on Integrated Network Management*, 2021.

[17] M. Scazzariello, L. Ariemma, and T. Caiazzi, "Kathará: A Lightweight Network Emulation System," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020.

[18] M. Scazzariello, L. Ariemma, G. D. Battista, and M. Patrignani, "Megalos: A Scalable Architecture for the Virtualization of Network Scenarios," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020.

[19] T. Caiazzi, M. Scazzariello, L. Alberro Zimmermann, L. Ariemma, A. Castro, E. Grampin and G. Di Battista. (2021) Sibyl Results. [Online]. Available: https://gitlab.com/uniroma3/compunet/networks/sibyl-framework/sibyl-results

[20] P. Lapukhov, A. Premji, and J. Mitchell, "Use of BGP for Routing in Large-Scale Data Centers," IETF, RFC 7938, Aug. 2016. [Online]. Available: http://tools.ietf.org/rfc/rfc7938.txt

[21] D. G. Dutt, *BGP in the Data Center*. O'Reilly, 2017.

[22] FRRouting. (2020) Frrouting. [Online]. Available: https://frrouting.org/

[23] The Next Platform. (2021) FRR: The Most Popular Network Router You've Never Heard Of. [Online]. Available: https://www.nextplatform.com/2020/10/26/frr-the-most-popular-network-router-youve-never-heard-of

[24] Insider Pro. (2019) The modern data center and the rise in open-source IP routing suites. [Online]. Available: https://www.idginsiderpro.com/article/3396136/the-modern-data-center-and-the-rise-in-open-source-ip-routing-suites.html

[25] Quagga. Quagga Routing Suite. [Online]. Available: https://www.quagga.net/

[26] B. Rijsman. (2020) RIFT-Python. [Online]. Available: https://github.com/brunorijsman/rift-python

[27] Juniper Networks. (2019) RIFT Overview. [Online]. Available: https://www.juniper.net/documentation/en_US/junos/topics/topic-map/rift-in-junos.html

[28] Y. Li, X. Yin, Z. Wang, J. Yao, X. Shi, J. Wu, H. Zhang, and Q. Wang, "A survey on network verification and testing with formal methods: Approaches and challenges," *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 940–969, 2019.

[29] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 225–238. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/singla

[30] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, S. Lu, and G. Lv, "Bcube: A high performance, server-centric network architecture for modular data centers," in *ACM SIGCOMM*. Association for Computing Machinery, Inc., August 2009. [Online]. Available: https://www.microsoft.com/en-us/research/publication/bcube-a-high-performance-server-centric-network-architecture-for-modular-data-centers/

[31] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[32] T. Caiazzi, M. Scazzariello, L. Alberro Zimmermann, L. Ariemma, A. Castro, E. Grampin and G. Di Battista. (2021) Sibyl Framework. [Online]. Available: https://gitlab.com/uniroma3/compunet/networks/sibyl-framework