MPLS-Kit: An MPLS Data Plane Toolkit

Juan Vanerio
Faculty of Computer Science
University of Vienna
Austria

Stefan Schmid
TU Berlin, Germany &
University of Vienna, Austria

Morten Konggaard Schou
Dept. of Computer Science
Aalborg University
Denmark

Jiří Srba
Dept. of Computer Science
Aalborg University
Denmark

***Abstract*—Networking research often requires a means to quickly generate different realistic networks for evaluating the practical relevance. This is especially the case for emerging fields related to the automated verification of network configurations ("what-if analysis") or to AI-driven network operations ("self-driving networks"). Unfortunately, the data of real world network deployments are often scarce. In particular, while the topologies of many real communication networks have been made available online, this data typically does not include the routers' forwarding tables, e.g., by Internet Service Providers (ISPs). This introduces a dilemma, as generating arbitrary forwarding rules for these topologies may not adequately mimic network behavior.**

We present MPLS-Kit, a tool for the automated generation of realistic MPLS data planes. In particular, the tool supports an efficient generation of MPLS data planes following widely-deployed industry-standard control protocols on top of arbitrary network topologies. Notably, MPLS-Kit supports the instantiation of MPLS Fast Reroute and VPN services. It further supports packet-level simulations providing a rich set of statistics about the simulated data plane which can be used for numerous applications, like congestion, latency, and resilience analysis. The generated data planes can be further exported in standard exchange formats and analyzed by formal verification tools.

I. Introduction

Modern communication networks are often very complex, and hence, modeling and analyzing their behavior can be difficult. Given that communication networks have become a critical infrastructure of our digital society in general and ISP networks in particular, and their dependable operation is crucial, this is worrisome.

In order to identify performance bottlenecks or to try out new innovative protocols, and verify their practical relevance, researchers need a means to generate realistic networks which mimic real and complex behavior. For example, in order to evaluate emerging automated network verification and what-if tools such as AalWiNes [1] and DeepMPLS [2], a way to generate realistic data planes (DP) is required. The generation of network configurations is also particularly important for emerging AI-driven approaches to improve the dependability and performance of networks, e.g., [2] or [3].

However, the data of real world network deployments are often scarce [4]. In particular, while the topologies of existing real communication networks have been made available online, e.g., [5], this data does not include the routers' forwarding tables; the latter however are required to model data planes. Introducing arbitrary forwarding rules may seem like a reasonable workaround, but the resulting data plane may vastly differ from the one found on a real network. Research based on such data planes may give results far from what can be observed in practice.

Even researchers who manage to obtain such data, e.g., through a collaboration with industry, typically only have access to one or two complete network configurations. They are likely also not allowed to share their data with other researchers. Furthermore, while many network protocols are based on open standards and RFCs, many implementations of these protocols are either not open-source or not fully featured.

This paper presents MPLS-Kit, a toolset that allows the generation of synthetic data planes for the popular Multiprotocol Label Switching (MPLS) [6] system, which is widely deployed by ISPs. Concretely, given a weighted network topology the tool directly computes the MPLS data plane that would have been obtained after running commonly deployed MPLS protocols as Label Distribution Protocol (LDP [7]) and Resource Reservation Protocol with Traffic Engineering extensions (RSVP-TE [8]) until convergence.

Our Contributions. Our main contribution is MPLS-Kit, a tool and library to quickly generate MPLS data planes. The tool provides researchers and operators with fine control over the configuration of each network element, from fine path-level to automatic creation of a given number of tunnels, supporting the evaluation and analysis of MPLS configurations and their behavior under various conditions.

It also enables the study of the impact of different design choices on the network performance. MPLS-Kit includes utility tools that provide easy-to-use, simple interfaces for evaluation and automation of the execution.

The library is easy to extend and re-usable, allowing for fast prototyping of new concepts. In particular, MPLS-Kit supports RSVP-TE with Fast ReRoute protections and VPN services, features rarely supported by other tools and forwarding stacks. MPLS-Kit further supports forwarding simulations at packet-level and provides easy-to-use, simple interfaces for evaluation and automation of the execution.

Together with this publication, MPLS-Kit is released as open source software at https://github.com/juartinv/mplskit, along with further examples of library usage. It can also be used online at https://demo.AalWiNes.cs.aau.dk.

Tool	Type	Direct DP Computation	MPLS	LDP	RSVP FRR	MPLS VPN	Open-source MPLS Code	Parametrized Configuration
GNS3 [9]	Emulator		√	√	√	√		
ns-3 [10]	Simulator		Limited				√	
Mininet [11]	Emulator		√	Limited			√	
Batfish [12]	Configuration analyzer	√					N/A	
OMNet++ [13]	Simulator		√	√			√	
MPLS-Kit	Generator and Simulator	√	√	√	√	√	√	√

TABLE I: Related work feature comparison.

Novelty and Related Work. We are not aware of any open-source tool which allows generation of realistic MPLS data planes. MPLS-Kit complements many existing works such as [1]–[3], [14] which so far relied on ad-hoc methodologies to generate data planes.

The lack of network data and protocol implementations, e.g., control planes, is not solved by emulators. GNS3 [9] allows running closed-source MPLS implementations that may use non-standard features and which cannot be independently reproduced for usage on research evaluations. Tools like ns-3 [10] and Mininet [11] can run open-source networking stacks but these may not implement all the expected protocol features. Besides, emulation of an entire network may also be time-consuming, as it requires the protocols to run until convergence, eventually providing the data plane as a by-product of the emulation.

In fact, realistic control plane emulators and/or generators of data planes are scarce in general. Batfish [12] is capable of building an internal vendor-agnostic network model (including the forwading table) from complete and correct vendor configurations. Some useful features for research purposes, like using parametrized configurations or stating that some features should be created randomly are outside the scope of Batfish. Also, Batfish does not support MPLS networks.

MPLS-Kit may also be compared with OMNeT++ [13], a C++-based general discrete event simulation environment. OMNeT++ is often used for communication networks simulations. Although it provides limited support for MPLS, some required functionality is missing, like support for VPN services and MPLS failure-protection mechanisms. OMNeT++ is efficient for detailed, full-scale, cross-layer simulations and not for data plane generation and prototyping.

See Table I for a concise feature comparison among these tools.

II. MPLS Network Operation

We model an MPLS networks as a graph composed of *routers* interconnected through bidirectional *links*, forming a *topology*. MPLS networks are designed to transport packets from an ingress to an egress router.

Ingress packets. Packets are inspected by the ingress router on the edge of the MPLS domain that finds their *Forwarding Equivalence Class (FEC)*. FECs are used to represent a network resource or group of resources, such as traffic engineering tunnels or virtual private networks (VPNs). All packets that request the same resource must be forwarded the same way. Each router hosts a (control-plane) table called Label Information Base (LIB) mapping each FEC to a unique local label. After identifying the packet's FEC, the router initializes the packet's MPLS label stack with the corresponding label from the LIB, and then further processes it like an internal MPLS packet.

Internal MPLS packets. Each router has a Label Forwarding Information Base (LFIB) table that registers the prioritized forwarding instructions for each top of stack label. These instructions describe the outgoing interface and the operations (pop, swap, or push) to be performed on the packet's label stack. If a match is found, the router uses the highest priority forwarding rules such that the outgoing interface is up. This behavior enables the implementation of failure protections. If no match or no acceptable forwarding rule is found, the packet is dropped.

In a real MPLS network, the routers compute their LFIBs after exchanging FEC and label information through *MPLS control plane protocols* specialized in different FEC types. Additionally, the routers gain information about the topology through a Link-State Interior Gateway Protocol (IGP), typically OSPF or IS-IS. After exchanging messages, each router populates its local Link-State Database (LSDB) and the Traffic Engineering Database (TEDB) for traffic engineering functionalities.

We use the term *flow* to generalize over the reachability requirements of different FEC types. The flow specifies an initial router and header along with the destination routers allowed by the FEC.

An MPLS data plane is a network topology including its routers and links, together with the LFIB of each router. The primary purpose of MPLS-Kit is to generate such a data plane and to provide simulation capabilities on top of it.

III. MPLS-Kit Overview

MPLS-Kit is a modular Python library supporting MPLS data plane generation and packet-level simulation, following closely the operations described above.

Its main strengths are being stand-alone and extensible, producing MPLS data planes based on controlable industry-standard protocols. In terms of features, MPLS-Kit provides:

- per-platform label space,
- direct computation of converged data planes,
- support for Penultimate Hop Popping (PHP) ([15]),

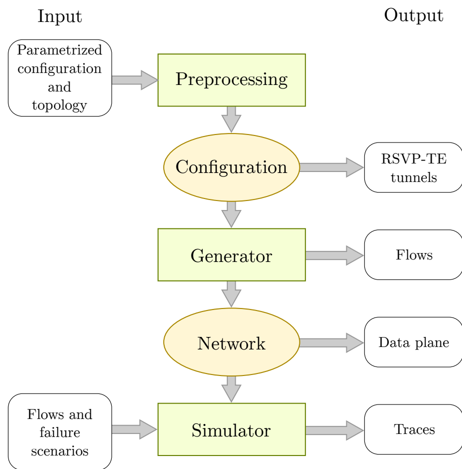

Fig. 1: MPLS-Kit interfaces and modules.

- computation of Fast ReRoute (FRR) protection paths,
- instantiation of VPN services,
- deterministic and non-deterministic forwarding rules (supporting e.g., ECMP),
- multiple supported control-plane protocols,
- printing data planes and flows to files, and
- easy automation through command line interface and external configuration files.

The main overview of MPLS-Kit is depicted in Figure 1. Initially, a user provides a *parametrized configuration* including a topology (either a NetworkX [16] graph, an external file, or random generation instructions) and the set of enabled control plane protocols along with their parameters. For instance, to create traffic engineering tunnels, allowed values are a list of *(tunnel start, tunnel end)* tuples or a number of different tunnels to be randomly created. The parametrized configuration can be provided as YAML files, python variables, or command-line arguments.

The Preprocessing module receives a parametrized configuration and returns a concrete one, i.e., a configuration with no undetermined elements. On the tunnels example, a concrete configuration has an explicit topology (a NetworkX graph) and an explicit list of tuples specifying the requested tunnels.

As shown in Figure 1, the tool has two core modules; the Generator and the Simulator. The *Generator* is responsible for computing the data plane, i.e. the topology and the MPLS forwarding tables, according to the specification of the concrete configuration. Its operation reproduces the basic functionalities of the control plane protocols including keeping tables as the LIB.

The Generator returns a *Network* object including the data plane and control plane components. The list of valid flows in the network can be extracted by examining the LIBs and exporting them to a file. The data plane can be exported to a JSON file.

The *Simulator* module provides packet-level simulation functionality. Its inputs are a Network object, a list of flows to reproduce and a file describing failure scenarios. Each failure scenario consists of a list of failed links, such that forwarding instructions using them become unavailable. For each flow, the Simulator instantiates and forwards a packet while recording the trace of traversed links and the final result (e.g., succesful delivery at destination, detection of a forwarding loop, etc.). Results and traces can be exported to a file.

IV. MPLS Dataplane Generation

MPLS-Kit uses high-level abstractions of the control plane components. Instead of thoughtfully mimicking control protocols and their subtleties, MPLS-Kit reproduces their essential functionalities in order to generate forwarding entries.

In a real MPLS network the routers engage in the distributed execution of an IGP protocol to obtain a consistent local view of the network topology to use in path computations. Such a process is time-consuming and prone to transient effects. Hence MPLS-Kit does not simulate any IGP, yet it does provide their essential features; i.e., providing a view of the topology and shortest path computations to every router, under the assumption of a single level, single area (OSPF or IS-IS) topology by default. This consideration covers most basic deployments, and the tool can be extended to multi-area deployments.

The communication involved in MPLS control protocols is also abstracted away; MPLS-Kit provides the protocol's client processes direct access to the memory content of their peer processes running on other routers. This simplification allows direct computation the same data plane that a real network achieves after convergence while avoiding delays and corner cases that arise due to the interleaving of communication processes and protocol computations. The following protocols are implemented in MPLS-Kit.

Label Distribution Protocol (LDP) [7]. Provides connectivity when traffic engineering is not required. It works by establishing Label Switched Paths (LSPs) along the existing IP paths. The routers broadcast label mappings for each IP prefix to all of their neighbors. In turn, these neighbors allocate a local label to the prefix and broadcast the information further. MPLS-Kit assumes the implicit existence of an IP prefix for each link and node in the topology. LDP is a "best effort" protocol; if a router fails, all LSPs through it will fail.

Resource Reservation Protocol with Traffic Engineering extensions (RSVP-TE) [8]. Used between ingress and egress routers to establish tunnels implemented as LSPs, with desirable traffic engineering properties. Examples include waypointing, ensuring a given bandwidth and avoiding some network links. MPLS-Kit supports the *facility backup* protection method standardized on MPLS for protection of traffic engineering tunnels [17]. Here a backup LSP is established to protect a set of primary LSPs sharing a path segment by intersecting at the closest possible common downstream node [18].

VPN. MPLS-Kit also provides an MPLS client for instantiating a generalization of industry-standard MPLS VPN services such as Pseudo Wires [19], VPLS [20], [21] and VPRNs [22].

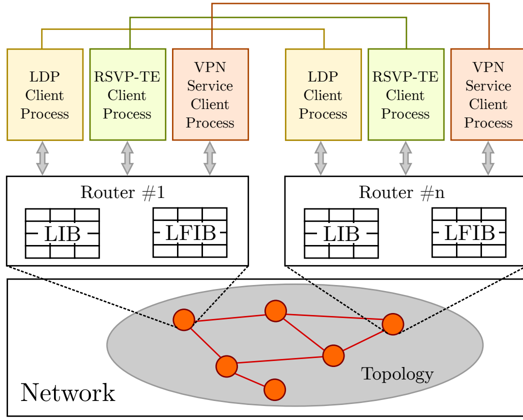

Fig. 2: Internal structure of the MPLS data plane generator.

A high-level view of MPLS-Kit's generator internal architecture and its components is shown in Figure 2. Descriptions are provided in the following sections. Protocol-related parameters are adjustable.

Router. Supports multiple concurrent MPLS client processes implementing control plane functionalities. As each router object has direct access to the network topology (in the same way a real router has access to its local LSDB or TEDB), it also provides path computation functionalities.

It keeps the following local tables:

- Label Information Base (LIB): allocates a local label to each FEC. Each FEC is managed by a single MPLS client process on each router.
- Label Forwarding Information Base (LFIB): keeps routing entries for each local label. The routing entries are computed by the respective MPLS client.

Network. A network object is composed by a given topology, pointers to the routers and global functions. In MPLS-Kit, a topology is implemented as an undirected weighted graph whose nodes are routers, and its edges are links connecting the routers.

MPLS Client Process. Represents an actual process running on a router to participate in the MPLS control plane. There is a specialized client type per protocol, each responsible for:

- creating FEC objects for the network resources related to its control plane protocol (e.g., IP prefixes for LDP and TE tunnels for RSVP),
- requesting labels for its FECs on the router's LIB table, and
- providing functions to compute appropriate routing entries for LFIB building.

Performance Examples. We use MPLS-Kit to generate 100 data planes (one per topology) in 14.16s on an Ubuntu 20.04 system with Intel Core i9 and 32GiB RAM. The selected topologies are taken alphabetically from the Topology Zoo [5], accounting for 36.16 nodes 45.34 edges on average. and ranging between 5 and 197 nodes.

The following parameter values are used:

- PHP enabled,
- LDP enabled,
- RSVP-TE enabled; 20 random tunnels with facility protection FRR, and
- VPN services enabled; 15 instances spanning 5 nodes each.

Additionally, we generate data planes for random topologies of different sizes, using the same configuration as above, except with $3n$ RSVP-TE tunnels and $2n$ VPN services with n being the number of nodes in the topology. For each n we generate 100 different topologies, one data plane per topology.

The results are shown in Figure 3. We can see that in the order of seconds, MPLS-Kit is capable of providing data planes fast enough for most interesting applications. These results are in line with MPLS-Kit's goal of providing a large variety of data to other tools in a time efficient way.

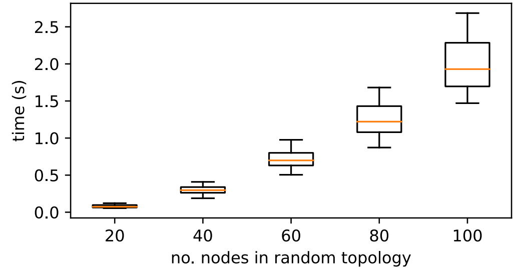

Fig. 3: Computation times for generating data planes.

Algorithm 1 Use of simulation to estimate link utilization.

Input: Flows $f \in F$, demands d_f, iterations n, capacity c_ℓ of each link ℓ.
Output: Utilization u_ℓ of each link ℓ

for each flow $f \in F$ **do**
  for i from 1 to n **do**
    $\texttt{trace} := \texttt{Simulator.run}(f)$
    for each link ℓ **do** $\texttt{use}_i(f,\ell) :=$ count ℓ in $\texttt{trace}$
  for each link ℓ **do** $\texttt{avg_use}(f,\ell) := \sum_{i=1}^{n} \texttt{use}_i(f,\ell)/n$
return $u_\ell := \sum_{f \in F} d_f \cdot \texttt{avg_use}(f,\ell)/c_\ell$ for all links ℓ

V. MPLS Forwarding Simulation

When a router forwards a packet in a real-world MPLS network, it uses the outmost label of the packet's stack to look up in its LFIB table. On a match, the router uses the forwarding rules with the highest priority such that the outgoing interface is up. If there are no acceptable forwarding rules at any priority level, or if at any point the time-to-live value reaches 0, the packet is dropped.

Consider the case in which the router's LFIB provides multiple equally preferable forwarding rules for a packet. Solving such nondeterminism is actually outside the scope of MPLS, and on actual routers this decision is made by the lower-layer Forwarding Information Base. MPLS-Kit resolves it by choosing uniformly among all available options.

Packet Forwarding Simulation Implementation. In MPLS-Kit, at the beginning of each execution step a packet lies inside a router's memory. It is then processed and forwarded

to the next hop (if any) moving to the next execution step, or finishing the simulation otherwise. As their real-world counterpart, MPLS-Kit's MPLS packets (instances of the MPLS packet class) have a label stack and a time-to-live field that decreases on each forwarding step. They also keep a pointer to the network object allowing access to all forwarding rules as well as to a list F of failed links. This information is used to filter out forwarding rules by priority when taking the forwarding decision.

The MPLS packet class implements the following forwarding methods:

- `step` simulates the next execution step as follows:
  1) In the current router's LFIB, identify the set of matching forwarding entries and their priorities.
  2) Filter out entries instructing to use a failed link, i.e. a link in F. If no entries remain, the packet is dropped.
  3) Select the highest priority rule. Break ties by randomly choosing with uniform distribution.
  4) Modify the packet's label stack, decrease its time-to-live, and send it to the next hop.
- `fwd` iteratively calls `step` until the packet depletes its label stack or its time-to-live expires.

As a packet is forwarded through the network, it keeps a record of its path (*traceroute*) for further analysis. Upon completion of the forwarding, the packet returns an exit code indicating its successful forwarding (0) or the specific type of error encountered.

Simulator Implementation. The Simulator class takes care of iteratively instantiating MPLS packets from user-specified flows on the network's routers with an adequate label stack and calling their `fwd` method.

Upon finalization, the Simulator gathers and aggregates statistics (exit code and traceroute) from all packet simulations and returns a summary of successful and failed cases. Results can be written to a CSV file or sent to the standard output.

VI. Use cases

We shall now provide examples of how MPLS-Kit can be used in practical applications.

Dataplane Verification. We can use the data plane output of MPLS-Kit to check properties using MPLS network data plane verification tools. The benefit of this use case is twofold. First, for testing and benchmarking a verifier tool, it is useful to have realistic data planes, since this is closer to what the tool will experience in real use. Second, the verifier allows us to check various properties of the generated data plane.

Formal verification. As a first example, we run AalWiNes [1] with a reachability query for each of the flows that the Generator outputs (see Figure 1). The query checks for a flow $(source, header, destinations)$ that a packet starting at the $source$ router with the given initial $header$ can reach one of the given $destinations$. For example for a flow (R_1, H, R_2), the query is $\langle H\rangle\ [\cdot\#R_1]\cdot^*\ [\cdot\#R_2]\ \langle\varepsilon\rangle$. This verifies that all the flows that MPLS-Kit claims to have created are in fact present in the output data plane.

Algorithm 2 Use of simulation for latency analysis.

Input: Flow f, delay $\texttt{delay}(\ell)$ for each link ℓ, iterations n
Output: Expected latency $E(\texttt{latency})$ for the flow f
for i from 1 to n **do**
  $\texttt{trace} := \texttt{Simulator.run}(f)$
  $\ell_1\ell_2\ldots\ell_m :=$ the sequence links in $\texttt{trace}$
  $\texttt{latency}_i := \sum_{j=1}^{m}\texttt{delay}(\ell_j)$
return $E(\texttt{latency}) := \sum_{i=1}^{n}\texttt{latency}_i/n$

Algorithm 3 Use of simulation for resilience analysis.

Input: Flows F, links L, failure-bound k,
  probability of single link failure p.
Output: Average success rate, weighted by failure probability
let $S := \{X \subseteq L \mid |X| \leq k\}$ ▷ Failure scenarios
for each X in S **do**
  $r_X := \texttt{Simulator.run}(F, X)\texttt{.success_rate}()$
  $w_X := p^{|X|}\cdot(1-p)^{k-|X|}$
return $\sum_{X\in S} r_X \cdot w_X / \sum_{X\in S} w_X$ ▷ Normalized average

Machine learning assisted verification. While formal verification tools for the data plane provide guaranteed results, they can still be relatively slow in practice. DeepMPLS [2] is a highly accurate, low execution time machine learning-based verification tool that also needs MPLS data planes as input, hence also benefitting from MPLS-Kit generation capabilities. The same properties (i.e., queries) verified with AalWiNes can be checked with DeepMPLS, and while the former provides guarantees, the latter can synthesize new MPLS header-rewriting rules in case a network property is not satisfied. Combined usage of AalWiNes, DeepMPLS and MPLS-Kit suggest an opportunity for synthesizing arbitrary property compliant MPLS data planes.

Congestion Analysis. We can use the simulation component of MPLS-Kit together with flow demands of a bandwidth-limited network to estimate congestion on links. From the trace output of the simulation, we can count how many times each flow traverses a certain link. By running the simulation of each flow multiple times, we can average out the randomness introduced by multiple path. Now, given traffic demands for each flow and the capacity of each link, we can compute the link utilization as in Algorithm 1. All links ℓ with $u_\ell > 1$ can be congested given the traffic demands of the flows. The benefit of MPLS-Kit for this use case is the realistic packet-level simulation on an MPLS data plane that can be used e.g. in early stages of network design.

Latency Analysis. The trace output of the simulation in MPLS-Kit can be used to estimate latency of flows in the MPLS network. Measuring network latency in a simulated environment like the one provided by MPLS-Kit can be helpful in networks where the researcher or the operator has no access to trace collecting tools or the ability to inject test traffic into the network.

We can turn the trace into a sequence of links $\ell_1\ell_2\dots\ell_m$, where m is the hop count. Due to nondeterminism in the data plane, several packets of the same flow may traverse different paths, so we repeat many experiments for the given flow. By labeling each link $\ell \in L$ in the topology with a delay, we can estimate the path latency of the flow as in Algorithm 2.

Resiliency Analysis. To test the resilience of various data planes, we can use the simulator's capability of simulating link failures. For a given set of failing links, the simulator outputs for each flow, whether the packet is successfully forwarded to an intended destination. We can run multiple simulations and compute the average success rate of packets in each failure scenario.

Using the external script, `create_confs.py`, we can for a data plane with links L systematically generate all k-failure scenarios as all the subsets $X \subseteq L$ with size $|X| \leq k$. To speed up the simulation, MPLS-Kit supports batch processing of failure scenarios and allows for possible parallelization of this computationally heavy task.

We can now average the success rates over all k-failure scenarios. To take failure probabilities into account, a weighted average can be used, as in the example in Algorithm 3, where link failures are modelled with a uniform, independent probability p. This gives a measure of the resilience of the given data plane. We can use this resiliency measure to compare different data planes of the same topology, for instance with or without fast re-route (FRR) protection.

VII. Conclusions

Motivated by the need to produce realistic data planes for networking research, we developed MPLS-Kit, a library with MPLS data plane generation capabilities (including its fast-rerouting features) which also supports packet-level simulations. MPLS-Kit is designed to faithfully mimick the control plane processes responsible for computing the forwarding tables in real networks, especially ISP networks, without engaging in time-consuming full convergence simulations. Our design and implementation are based on a hierarchical structure of classes closely following the internal architecture of a router, allowing easy development of new functionalities and prototyping. As demonstrated with our use cases, MPLS-Kit can be useful in many scenarios, ranging from verification of data plane properties with external tools to simulation-based studies like convergence, latency and resiliency analysis. In summary, our contribution provides an opportunity to alleviate the scarceness of data plane datasets, thus encouraging more reproducible research in networking.

As a future work, we plan to extend our current work with Segment Routing (SR) [23], addressing first MPLS-SR and afterward moving towards IPv6-SR.

References

[1] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba, "AalWiNes: A fast and quantitative what-if analysis tool for MPLS networks," in *Proc. ACM CoNEXT*. ACM, 2020, pp. 474–481.

[2] F. Geyer and S. Schmid, "DeepMPLS: Fast analysis of MPLS configurations using deep learning," in *Proc. IFIP Networking*, 2019.

[3] A. Blenk, P. Kalmbach, S. Schmid, and W. Kellerer, "O'Zapft is: Tap your network algorithm's Big Data!" in *Proc. ACM SIGCOMM 2017 Big-DAMA Workshop*, 2017.

[4] K. Claffy and D. Clark, "Comments on request for information on the american research environment," 2020-01.

[5] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[6] B. S. Davie and Y. Rekhter, *MPLS: technology and applications*. Morgan Kaufmann Publishers Inc., 2000.

[7] L. Andersson, I. Minei, and B. Thomas, "LDP Specification," Internet Requests for Comments, IETF, RFC 5036, Oct 2007.

[8] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow, "RSVP-TE: Extensions to RSVP for LSP tunnels," Internet Requests for Comments, IETF, RFC 3209, Dec 2001.

[9] "GNS3," https://gns3.com/, accessed: 2022-02-10.

[10] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[11] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proc. ACM SIGCOMM HotNets*, 2010, pp. 1–6.

[12] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *Proc. USENIX NSDI*. USENIX Association, 2015, pp. 469–483.

[13] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proc. of ICST SIMUTools Workshop*, 2008.

[14] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-Rex: Fast verification of MPLS networks with multiple link failures," in *Proc. ACM CoNEXT*, 2018, p. 217–227.

[15] E. Rosen, D. Tappan, G. Fedorkow, Y. Rekhter, D. Farinacci, T. Li, and A. Conta, "MPLS label stack encoding," Internet Requests for Comments, IETF, RFC 3032, Jan 2001.

[16] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proc. of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[17] P. Pan, G. Swallow, and A. Atlas, "Fast reroute extensions to RSVP-TE for LSP tunnels," Internet Requests for Comments, IETF, RFC 4090, May 2005.

[18] M. Chiesa, A. Kamisinski, J. Rak, G. Retvari, and S. Schmid, "A survey of fast-recovery mechanisms in packet-switched networks," *IEEE Communications Surveys and Tutorials (COMST)*, 2021.

[19] S. Bryant and P. Pate, "Pseudo wire emulation edge-to-edge (PWE3) architecture," Internet Requests for Comments, IETF, RFC 3985, Mar 2005.

[20] V. Kompella and Y. Rekhter, "Virtual private LAN service (VPLS) using BGP for auto-discovery and signaling," Internet Requests for Comments, IETF, RFC 4761, 01 2007.

[21] M. Lasserre and V. Kompella, "Virtual private LAN service (VPLS) using label distribution protocol (LDP) signaling," Internet Requests for Comments, IETF, RFC 4762, 01 2007.

[22] E. Rosen and Y. Rekhter, "BGP/MPLS IP virtual private networks (VPNs)," Internet Requests for Comments, IETF, RFC 4364, Feb 2006.

[23] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing Architecture," Internet Requests for Comments, IETF, Tech. Rep. 8402, Jul. 2018.