EPOXIDE: A Modular Prototype for SDN Troubleshooting

Tamás Lévai, István Pelle*, Felicián Németh, András Gulyás†
HSN Lab, Budapest University of Technology and Economics
{levait,pelle,nemethf,gulyas}@tmit.bme.hu

ABSTRACT

SDN opens a new chapter in network troubleshooting as besides misconfigurations and firmware/hardware errors, software bugs can occur all over the SDN stack. As an answer to this challenge the networking community developed a wealth of piecemeal SDN troubleshooting tools aiming to track down misconfigurations or bugs of a specific nature (e.g. in a given SDN layer). In this demonstration we present EPOXIDE, an Emacs based modular framework, which can effectively combine existing network and software troubleshooting tools in a single platform and defines a possible way of integrated SDN troubleshooting.

Keywords

SDN, Network troubleshooting; Debugging; Emacs

1. INTRODUCTION

SDN takes away some parts of the forwarding logic from traditional switching devices, but the complexity is now shared among the different layers of a (logically) centralized controller and (virtual) network functions. Finding causes of malfunctions in this heterogeneous distributed system is even more challenging than before. To address this challenge, lots of SDN troubleshooting tools (e.g., OFRewind, NetSight, VeriFlow, NICE, OFTEN, etc.) have been created that are able to investigate one aspect of the SDN architecture for specific errors, failures, misconfigurations, or bugs. Additionally, Heller *et al.* proposed a holistic approach [1] that systematically searches for bugs through different layers of the SDN architecture. Their troubleshooting process narrows down the possible causes by a series of hypothesis testing. In the general case, however, conforming a hypothesis requires an application specific method.

In this paper we present EPOXIDE, an Emacs based modular framework, which can flexibly combine network and software troubleshooting tools in a single platform. EPOXIDE does not try to be a complex troubleshooting software that fully integrates all available tools, but rather a lightweight framework that allows the ad-hoc creation of tailor-made testing methods from predefined building blocks to test troubleshooting hypotheses [2]. In the demonstration, we define automated hypotheses testers as *troubleshooting graphs* (TSG, see Fig. 1b and 1c for an example) by flexible linking of simple wrappers around traditional troubleshooting tools (ping, ipref, ovs-ofctl, etc.), but the framework allows the users to write wrappers to more complex SDN tools listed above. Defining a TSG in the high-level language of EPOXIDE is less error phone than, for example, writing shell scripts for similar tasks. The TSG definition is also useful to share common practice between network engineers. Moreover, EPOXIDE makes convenient to troubleshoot the troubleshooting process itself by allowing to inspect or alter the data flow between wrappers.

2. ARCHITECTURE

EPOXIDE is an add-on package to the GNU Emacs text editor. Emacs is an ideal prototyping platform because of its many relevant features like efficient remote shell and powerful sub-process handling. Its main concept is the *buffer*, which is a universal data structure to store editable text and general variables. In Emacs everything is a buffer: opened files and interactive shells as well.

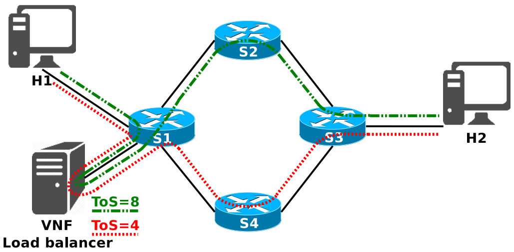
TSGs consist of nodes and links that can be defined using our Click-inspired language and stored in .tsg configuration files. Nodes are functional elements either implementing a wrapper for an external tool or written entirely in Emacs Lisp (like the Table-view node discussed later). Links connect nodes by relaying text data between them. The creation, execution and manipulation of TSGs are controlled by the framework that maps nodes and links to buffers and then helps the data distribution among nodes. Node buffers hold node specific attributes and potentially display state information; while link buffers are the inputs and the outputs of the nodes containing human readable text data (e.g., the output buffers of the `iperf` wrapper node contain the intact output of the wrapped tool). The framework with the help of its event scheduler is responsible to call the execution function of a node if one of its input buffers got modified, afterwards it is the responsibility of the execution function to process the input data, and to write into its output buffers.

EPOXIDE offers *easy extensibility*: framework and node functions are implemented in separate Emacs Lisp files and the framework provides a programming interface to node

*MTA-BME Future Internet Research Group
†MTA-BME Information Systems Research Group

SIGCOMM '15 August 17-21, 2015, London, United Kingdom

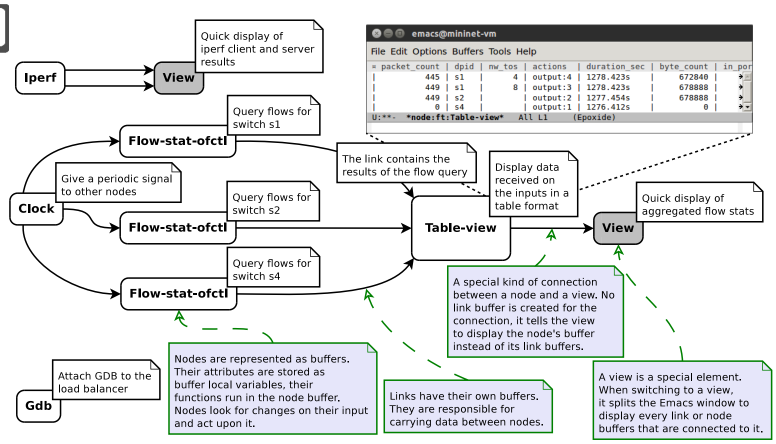
ACM ISBN 978-1-4503-3542-3/15/08.
DOI: http://dx.doi.org/10.1145/2785956.2790027



(a) A simple troubleshooting use-case.

```
iperf::Iperf(192.168.123.4, -u -i 1,
             192.168.123.2, -u -i 1
             ) -> view::View();
iperf[1]->[1]view;
clock::Clock()
  ->Flow-stat-ofctl(localhost, s1)
  ->ft::Table-view(packet_count, dpid, nw_tos)
  -->View();
clock->Flow-stat-ofctl(localhost, s2)->[1]ft;
clock->Flow-stat-ofctl(localhost, s4)->[2]ft;
Gdb(192.168.123.3, click);
```

(b) TSG definition in our Click-inspired language.



(c) The buffers of EPOXIDE after executing the .tsg config in Fig. 1b.

Figure 1: EPOXIDE through an example.

developers. This allows creating third party node implementations and also external node repositories. Developers only have to write node initialization, execution, and termination functions. Optionally, they can add documentation functions to provide information about node configuration parameters. *Context-aware buffer switching* keeps track of the created buffers and provides support to move among them in an orderly fashion by applying key combinations. When multiple branches of the TSG are available, possible choices are offered as a selectable list, or they can be selected by the same key combination prefixed with the output link's number. Users can also switch to buffers associated with the current EPOXIDE session by selecting from a list of buffer names grouped together based on their types and positions in the TSG. *TSG visualization* offers a graph representation of the interpreted .tsg file, navigation options for graph traversal, and quick access to node and link buffers. *Syntax highlighting and context-aware code completion* is provided for .tsg files. Additional, context-aware help is also shown with the short documentation of the current node, in which the currently typed parameter is highlighted. Intelligent code completion is offered: candidate lists are populated with node classes and phrases from other buffers. *Views* are special TSG elements that can create bundles of link and node buffers to be displayed together by splitting the Emacs window to different sub-windows. In a complex TSG, views provide quick access to a subset of the buffers. Moreover, in case of a long chain of nodes not only the final result is shown but every intermediate partial result is also available for inspection in the corresponding link buffer. *Run-time node creation, reconfiguration and re-linking* provides two possibilities to make modification on a TSG that has already been started. The first one allows reinterpreting a modified .tsg file and then reconfigures the TSG based on file modifications. The second one interactively guides the user to add a new node or reconfigure existing nodes or links between them.

During the course of the demonstration we present EPOXIDE via a simple troubleshooting use-case (Fig. 1a). In this mininet-based scenario we try to implement a load balancer in a VNF which is supposed to mark the packets randomly with ToS values 4 or 8. Flow entries are installed in $s1$ which ensure that the switch will forward the packets marked with ToS 4 and 8 to $s4$ and $s2$ respectively. But there is a software bug in the VNF, hence it marks all packets with ToS 4. A simple way to identify this bug would be (i) to run `iperf` between $H1$ and $H2$ (which works but is slow), (ii) inspect the flow tables of $s1$, $s2$ and $s4$ with `ovs-ofctl` (which are ok), (iii) attach `gdb` to the VNF process and finally identify the bug. Arguably going through this process in the standard way would require many shells (2 for the `iperfs`, 3 for the `ofctls` and 1 for `gdb`) to be opened and managed actively (e.g. ssh to the hosts, keeping in mind in which shell which particular process runs) by the troubleshooting person. EPOXIDE offers another way for doing this by letting the troubleshooter focus more on the troubleshooting process itself and less on the unnecessary details of implementing it. Instead of opening shells, the troubleshooter writes a simple `.tsg` config file incrementally (Fig. 1b). For `iperf` she simply specifies the server, the client and other extra arguments (if any) and redirect both the server and client outputs to an EPOXIDE view. The outputs of the server and client then can be seen together by switching to this view. Using the `ofctl` wrappers and a Table-view node, it is possible to direct the ordered (and possibly filtered) `ofctl` outputs to a shared table and examine them together (see the upper right corner of Fig. 1c). Finally, the `gdb` wrapper of EPOXIDE can connect to remote machines, grep for processes specified in the second argument (in our example this is a click process) and attach `gdb` to the process. The .tsg file then can be saved for future use or shared with another troubleshooter as a footprint of a process for tracking down a specific bug.

The source code is at: http://github.com/nemethf/epoxide.

Acknowledgements – The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement Nº 619609.

3. REFERENCES

[1] B. Heller et al. Leveraging SDN layering to systematically troubleshoot networks. In *ACM SIGCOMM*, HotSDN '13.

[2] I. Pelle et al. One tool to rule them all: A modular troubleshooting framework for SDN (and other) networks. In *ACM SIGCOMM*, SOSR '15, 2015.