Automated troubleshooting toolset for SDNs

Knop Thibaut, Rochet Florentin
EPL, UCL
Louvain-la-Neuve, Belgique
{thibaut.knop,florentin.rochet}@student.uclouvain.be

Abstract—**TODO**

Index Terms—**TODO**

I. INTRODUCTION

For many years, the debugging has always been one of the major concerns in network maintenance. In order to localize problems into the network, operators usually use a narrow toolset, composed of traditional tools as `ping`, `traceroute` and `SNMP` agents [**?**].
Hopefully, this difficult and time-consuming [**?**] process could change, given the deployment of Software-Defined Networks. In a nutshell, SDNs are based on a separation between control- and data planes, which offer the opportunity of programmable networks [**?**]. Those SDNs are composed of a network of switches managed by a logically-centralized controller, whose role is to (un)install rules into the flow table of the switches, to read traffic statistics and respond to the network activity.
However, and because SDNs allows different operators, developers to dynamically program the same network, the complexity of software will increase [**?**] and potentially the numbers of bugs. To minimise the trade-off between introducing new functionality and increase the number of bugs in the network, there is a serious need for a complete and effective automated testing toolset, allowing the admins to focus on fixing the issues instead of localizing them. In traditional network architecture, it is almost impossible to create such an automated test suite, due to the complexity of *knowing the operator's intent* and *checking network behavior against intent* [**?**] (for more information about how traditional networks could be extended to support automated troubleshooting, please refer to [**?**]).
As we will explain, we use the different layers of the SDN stack to review the available tools for automated troubleshooting. Note that the methodology and the structure of this paper is influenced from the one used in [**?**]. It indeed seems the better way to articulate and present the different tools destined to localize the problems and their cause in an automated way. The paper is structured as follow : first we recall the different layers of the SDN stack, and how it can be leveraged to provide automated troubleshooting for SDNs, then we present different existing tools by positioning them regards to the SDN layering, and we eventually conclude.

II. SDN LAYERING, THE KEY USED TO A BETTER TROUBLESHOOTING

Finding and solving network bugs are not the aim of SDN, but we can use it to re-think the way we troubleshoot networks.
The SDN architecture is decomposed into layers, those layers can be represented in a two dimensionnal array. As you can see on Figure II, we have the two main layers called *State layers* and *Code layers*. The state layers hold a representation of the network's configuration for each parts of the network architecture. The code layers implement logic to maintain the mapping between two state layers. Each states layers should verify the equivalence properties, which means that each of them should correctly mapping every other state layer. The idea is that, for each policy, if the state layers are correctly mapped among each other, then the policy is set and acts like it should.
On the Figure II, you have the following elements.
Policy - Policies are set up by the network administrator to configure the Logical View. These policies can be routing, access control or QoS policies. They are written inside a control application. See [**?**] for exemple.
Logical View - Abstract representation of the network which aims to make things easier for the control application to create policies. A mapping between the Logical View and one or more Physical View are done by Network Hypervisor.
Physical View - This is a correspondance with the real network element, a representation of it handled by the network OS. The protocol used to configure the network element is one like OpenFlow [**?**]. A physical View has thus a one-to-one mapping with a real network element.
Device State - State of the network element maintened by its firmware.
Hardware - Network element.

Thanks to the SDN stack, one can first build a tool to check consistency between state layers in order to identify on which part of the network architecture a bug is happening (in which *code layer*). Then, when the layer is identified, an other tool take over to localize the issue inside the code layer. We will see in section III which kind of tools could be used to handle this bug hunting inside the code layer.
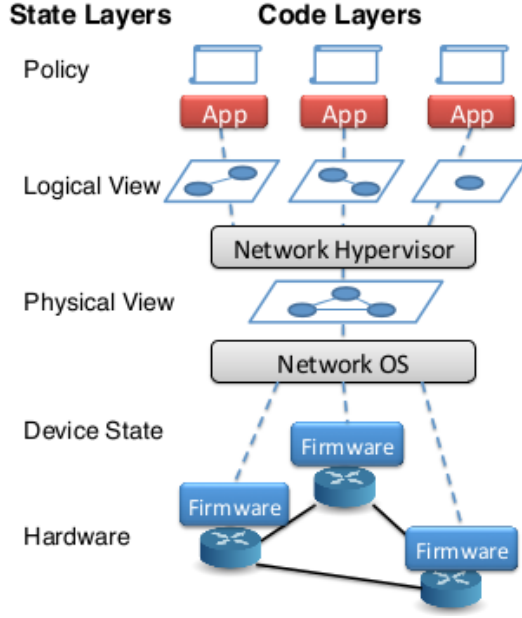
Fig. 1. SDN architecture - This image come from [**?**]

III. Network Troubleshooting - Tools

A. Network Debugger (NDB)

B. VeriFlow

If the actual behavior of the network does not correspond to the policy, then the issue is potentially the correspondance between the device state and the policy. There exist different tools to verify this hypothesis. The tools we discuss here have the advantage to verify some network invariants in real-time. Other tools as Anteater and Header Space Analysis (HSA) that are also used in the same purpose carry out the static analysis of snapshots of the network data-plane state [**?**] [**?**]. Since SDN controllers are capable of installing around 30.000 flows per second while maintaining less than 10 ms as delay for the installation, it is absolutely not enough to have tools that check for network invariant with a latency of the order of seconds [**?**].

VeriFlow acts as a proxy in between the SDN controller and the switches that verifies some common network invariants at each forwarding rule installation at very high speed. In order to achieve a 7ms delay inflation with regards to traditional TCP connections, VeriFlow slices the network into equivalences classes, then builds a virtual forwarding graph for each ones using a trie structure, and finally checks invariants by traversing those graph with a depth-first search approach. Note that VeriFlow allows to spotting network issues before they reach the network, which is very valuable.

C. SOFT - Systematic OpenFlow Testing

D. NICE - No Bugs in Controller Execution

E. FlowChecker

IV. Conclusion