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 the 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 and 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 (see Section II), 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 regarding the SDN layering, and we eventually conclude.

II. SDN layering, the key used to an automated 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. Tools to find the code layer concerned by the issue and tools which operate inside the code layer, to find the cause of the bug.

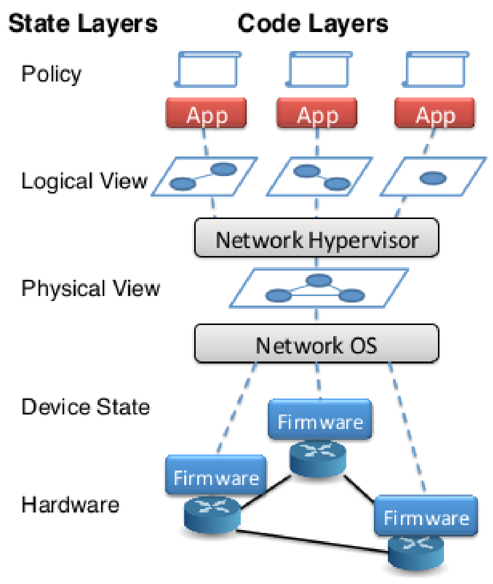

Fig. 1. SDN architecture[1]

III. Network Troubleshooting - Tools

A. Network Debugger (NDB)

SDNs bring a new way of building networks, like a programmer build a software, a network administrator will be able to program its network. The software programmer has tools to debug its program, like the well-known **gdb**. **ndb** is a tool inspired from **gdb**, it aims at finding network error by identifying uncorrect sequence of events. **ndb** has the ability to do a sort of *breakpoint*, which is in fact a packet filter. When packet are matched by this filter, the breakpoint is triggered and the entire trace of that packet is displayed. The authors mention that the term *tracepoint* should be more appropriate but the term *breakpoint* is more familiar. **ndb** uses its *backtrace* primitive to show the sequence of events for a packet. This trace is simply the sequence of forwarding actions applied on the packet by the switches through which it passed. For example of **ndb** utilisation, please refer to its paper [?].
In our layering SDN architecture, **ndb** give us the opportunity to build a tool to confirm the right action of the policies implemented in the control application. For example, if a routing bug occurs in the network, a tool of top of **ndb** could observe paths taken by packets and complete switches states in order to conclude if the policies are correctly matched or not. If the policies are correctely matched, it means that the network administrator has written policies that dont't fit his needs. In this case, **ndb** has found the *code layer* where the issue happen. Now if the policies are not correctly matched, then the issue does not come from an uncorrect implementation. **ndb** has thus exclude the implementation to be the root of the bug and the search can continue among other layers, with other tools such as **SOFT** [?].

[1]Source : [?]

B. VeriFlow

If the actual behavior of the network does not correspond to the policy, then the issue can potentially be found in the erroneous correspondance between the *device state layer* and the *policy layer*. There exist different tools to verify this hypothesis. The one 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 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 and this proxy verifies some common network invariants at each forwarding rule installation with a very high speed. In order to achieve a 7ms delay inflation with regards to traditional TCP connections[2], 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.
As limitations, we can note that presently, VeriFlow only checks for reachability invariants, and that it is not suitable for multiple/distributed controller [?].

C. SOFT - Systematic OpenFlow Testing

SOFT is a tool which aims to test the interoperability between OpenFlow switches. On our layering architecure, the OpenFlow protocol appears in the Physical View. SOFT has been designed to identify wrong behaviors of different switches running different implementation of the OpenFlow protocol. Thus, SOFT compare different Physical View of different switches. But SOFT can do more than that. It can also help to find inconsistency in the layers below the Physical Layers. SOFT can be used to check the consistency between the *state layer* of Device State and the *state layer* of Hardware

D. NICE - No Bugs in Controller Execution

Following the SDN layering technique, once the erroneous 2-layers correspondance has been determined, another tool can be used to spot the bug more precisely, either in the controller software or in the firmware running on the switches.
The NICE tool is used to produce traces leading to a bug, by systematically exploring the set of possible states of the system (including controller, switches and hosts) and checking them regarding to some network invariants [?]. What makes this contribution to OpenFlow applications

[2]Test configuration : 20 nodes OpenFlow network sing Mininet and a NOX controller [?]

verification a real asset is its ability to deal with a very large space state. The real challenge for testing openFlow applications is the scalability related to its wide environment : there is a potentially unbound state space, due to a large space of switch state, of inputs packets and of event orderings [**?**]. To handle that situation, developers can create abstractions of their application and use traditional model checking techniques to prove the properties about the system. Example tools are SPIN and JavaPathFinder, whose the mains limitations are respectively the complexity or writing the model due to the domain-specific information and a significant performance slackening [**?**]. Moreover, both are suffering from some state-space explosion.

NICE extends traditional model checking with symbolic execution of the event-handler. It identifies therefore equivalence classes of packets that cover all code paths of the application. NICE combines also specific search strategies allowing it to reduce the state space by up to 20 times [**?**].

E. FlowChecker

IV. Conclusion