Simulation of a Distributed Invariant Verification Scheme for Software Defined Networks

Rakesh Kumar
Department of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign
Email: kumar19@illinois.edu

***Abstract*—We perform a simulation study of a distributed system's performance. The purpose of distributed system is to verify that a large network's forwarding-plane state always conforms to certain invariants such as conformance to a security policy, or absence of loops etc. These invariants ensure the correct, secure operation of the system. Hence this verification needs to be performed in *near* real-time. This verification is performed on system state that spans across multiple controllers and aggregation nodes in the system. We first describe the abstract elements of the design and following that study the scalability properties of the proposed design using a process-oriented simulation package SimPy [7].**

I. Introduction

In the Software Defined Network [1] paradigm, the network comprises of a set of *dumb* switches that rely upon a central entity called the *controller* for all switching logic. The controller pushes the logic in the form of simplified *rules* to the switches. Each individual rule instructs the switch about the fate of traffic arriving at its ports. The functional output of a switch depends completely on the collection of rules that it holds. Hence only the controller has the logically centralized view of network's data-plane state. The controller then makes this logically centralized state available to network applications that orchestrate the state to perform desired operations such as access-control, load-balancing, virtualization by updating the rules that are applied on the individual switches.

The centralization of network state allows one to verify the real-time state snapshot at controller and check for presence of certain invariant properites, such as: loop-freeness, absence of black-holes, conformance to access-control policy etc. As the individual updates arrive at the controller, a verification engine can take as input the current state of network and the desired rule update and ascertain whether the update violates any pre-defined invariants. Two approaches to solving this problem real-time network invariant verification were recently explored. [2], [3]. For a single controller, both of these approaches have been demonstrated to provide answer to queries within 100s of milliseconds. However it is not clear if these appraoches scale in a setting where the network size is large and each the entire state of network is distributed across more than one controller for scalability and administrative reasons. It is this question that this project attempts to explore. In the next section, we describe the distributed invariant verification problem in detail and prescribe a design to solve the problem. Following that, we describe the model used to study the properties of the prescribed design. Following that we present results and conclude.

II. Problem Description

Consider a large software-defined network with multiple parts called slices. The network can span across multiple administrative/organizational domains (e.g. multiple branches of a large corporate network, autonomous systems in the internet). Each slice contains switches that are controlled by its controller. The switches are connected to other switches within the slice and some of them form peering links with switches across slices. Assume that a communication channel exists between peering slices' corresponding controllers via which they can share state about their network with each other. Rule updates can be applied on each slice's controller. In order to verify any invariant, the impact of a newly arriving rule on the current state of the network needs to be computed.

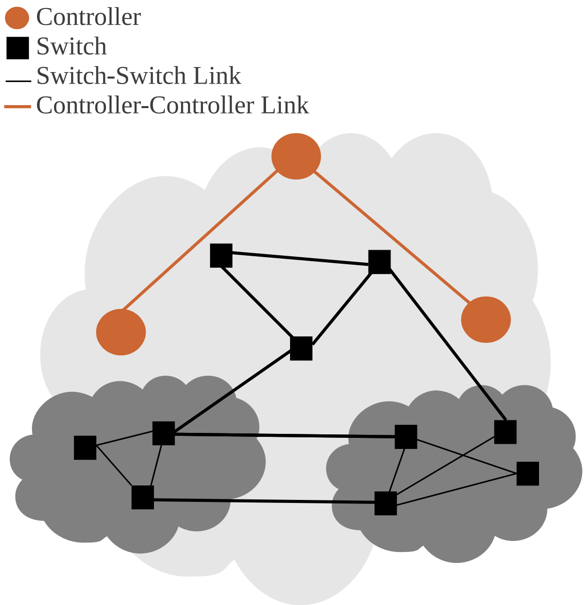

Fig. 1. A two-layer software defined network. The larger network contains two smaller networks and its controllers.

However, the state of the entire network is distributed across different controllers. Hence, the controllers need to collaborate to perform invariant verification. Each update arriving on a slice' controller can have effects that are entirely local to the slice or not. Given a slice of network and a rule update that has effect only on that slice, verifying whether the update would cause any invariant violatations is a problem that has been studied. [2], [3] What is not obvious is how to determine whether the rule update affects any other slices in the network.

In the next few sub-sections we present the outline of a design that answers this question.

A. *Network Slices are Switches*

The key idea is to think of a network slice as an *equivalent switch* with all its peering links as ports. The entire state contained by the controller can be expressed as rules on a switch expressing just what enters and what portion of the entering traffic exits various other ports while abstracting details of state related to intra-slice forwarding done by switches. This is accomplished by borrowing the technique presented by Peyman et. al. in NetPlumber [3]. The trick is to attach sources and sinks at all the peering interfaces to compute just what enters/exits that particular peer. Once that is computed, simplistic forwarding rules expressing the relationship between input and output traffic at each peering port can be computed. Hence, each controller becomes just another switch containing these equivalent rules. This switch representation is initially formed and subsequently updated as verified rules are applied to the local slice' network.

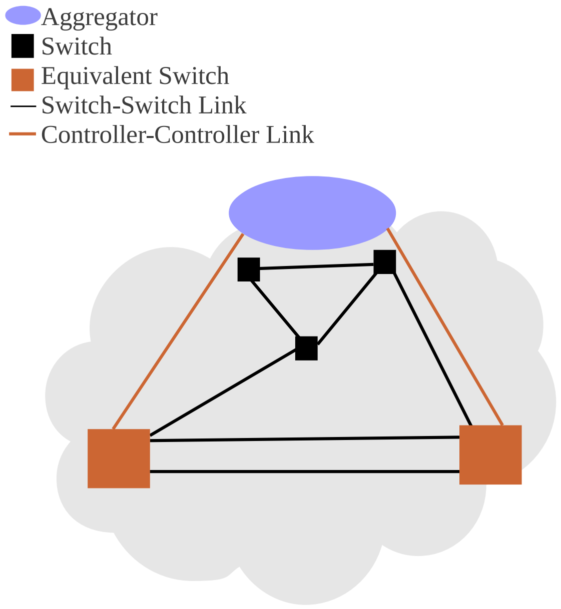

Fig. 2. An equivalent view of the network.

B. *Aggregation Nodes*

Each controller shares its switch representation with one aggregation node. Each aggregating node collects switch representations from multiple controllers. Hence the aggregation nodes are essentially looking at a collection of switches and they can further do what individual controllers did and compress the state again to form a single switch representing the state of a collection of controllers and further share it with other aggregation nodes in other parts of the network. This design choice follows the SDN philosophy of converging state - which comes at the cost of introducing central points of failures in the network while making computation model simple and in this case more amenable to analysis. Also, this enables one to model an aggregation node essentially as just another controller.

C. *Rule Update Propagation*

When a rule update arrives at a particular controller, it first checks whether the rule does not put the local slice's network in an incorrect state. If not, then the controller creates an *equivalent rule update* on its equivalent switch which represents the impact of applying the rule update on its network slice. It then forwards the equivalent rule update to the aggregation node which performs the same analysis *locally* on the state that now spans multiple controllers. Since the aggregation node is just another controller, it can further replicate the operation that happened on the individual controller, i.e. to check if the rule update is 'local' or whether it needs to be forwarded further up to a higher level aggregation node. Hence the process keeps repeating until the equivalent update has propagated to all the aggregation nodes that it can potentially affect.

D. *Rule Update Application*

When a rule update arrives at a controller, it has to wait for the update to propagate to all the aggregation nodes in the hierarchy and for the result to propagate back. Whether or not it violates any invariants can only be determined once all the results have been received by the entire hierarchy of aggregators. Our design choice is to wait until such determination is made before applying the update to the switches. This choice allows for the network to always be in the invariant conformant state at all times. However, we wish to study how long does *on average* for such decision to be taken.

III. Model Description

We simulate the model described above using the Python simulation package SimPy [7]. The network topology that we consider is that of hubs-and-spokes. Each spoke contains numerous switches and a controller. The controller in the spoke connects to a hub, where we assume an aggregation node is present. The aggregation node has multiple spokes' connecting to it. Furthermore, the aggregation node's network can then also behaves as a spoke to a larger network and so on. Hence the hub-and-spoke essentially reduces to a tree with multiple aggregation levels and an outdegree. All nodes except the leaves of this aggregation are aggregating controllers. The leaves are just simple controllers where updates are generated. For the purposes of this simulation the tree's outdegree was set to 2 with total 10 levels.

A. *Parameters*

- Exponentially Distributed Arrival Times: Each controller is responsible for producing updates. We use a single arrival generator to generate updates at a rate scaled by total number of controllers in the system and then splitting the arrivals using uniform distribution among all controllers.
- Exponentially Distributed Service Times: We assume the same service time distribution for all controllers and aggregators.
- Probability of Hopping: The probability that when the update arrives on the controller/aggregator, it will be sent to the next. It is uniformly chosen and this makes the total number of aggregation layers that are

involved in processing an update to be geometrically distributed.

B. Metrics

The key metric of interest is the Total Processing Time for any update. This is computed by keeping track of update's arrival on every hop and summing over processing times on all hops.

C. Policies

Our simulated system only has one policy of waiting before the arrival results come to apply the update.

IV. Experimental Design

We perform an 2^k study.

V. Conclusion

In the previous study using analytic methods, we were limited by the state-space explosion when it came to obtaining results for larger topologies. The results of this study are same for smaller networks that could be analyzed using analytic methods and hence trustworthy to be extrapolated.

References

[1] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[2] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Brighton Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, NSDI'13, 2013.

[3] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Whyte. Scott. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, NSDI'13, 2013.

[4] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 7–12, New York, NY, USA, 2012. ACM.

[5] Bruce Daniel McLeod. Performance analysis of n-processor time warp using stochastic activity networks. Master's thesis, University of Arizona, 1993.

[6] Kleinrock, Leonard Queueing systems. volume 1: Theory 1975, Wiley-Interscience

[7] Muller, Klaus Advanced systems simulation capabilities in SimPy Europython, 2004.