Medieval: Towards A Self-Stabilizing, Plug & Play, In-Band SDN Control Network

Liron Schiff
Tel Aviv University

Stefan Schmid
TU Berlin &
Telekom Innovation Labs

Marco Canini
Université catholique
de Louvain

ABSTRACT

To provide high availability and fault-tolerance, SDN control planes should be distributed. However, distributed control planes are challenging to design and bootstrap, especially if this is done in-band, without a dedicated control network, and without relying on legacy protocols. We present Medieval, a plug & play, distributed control plane that supports automatic topology discovery and management, as well as flexible control plane membership: controllers can be added and removed dynamically. Medieval comes with interesting robustness guarantees and is provably self-stabilizing: from any initial topology, the controllers quickly self-organize and establish a communication channel among themselves. Given the resulting managed control plane, arbitrary network control services can be implemented on top. Interestingly, Medieval is also self-reliant, in the sense that it is based on OpenFlow only, and does not require any legacy protocol to bootstrap.

1. INTRODUCTION & CONTRIBUTION

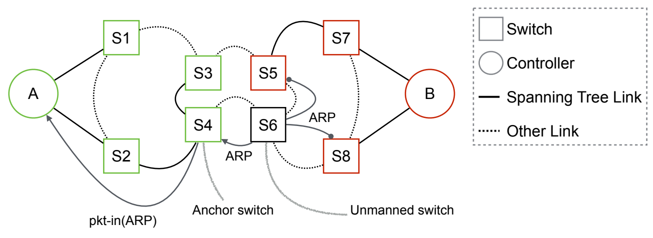
Software-Defined Networking (SDN) outsources control of data plane switches to a collection of network-attached servers. This paper explores the fundamental problem of how to provide connectivity between the control plane and the data plane. In particular, we advocate for a software-driven, in-band control mechanism to bootstrap and maintain connectivity, and more generally, to support a distributed SDN control plane. To make this case, let us consider what services are required by such a control plane:
Connectivity: to allow communication between controllers and switches, and between controllers.
Controller discovery: to allow individual controllers discover the existence of other controllers and to detect when some are no longer reachable.
Switch discovery: to detect switches that are not yet associated with a controller and establish a control channel with them. Also, to reestablish communication with switches in case of link failure, network partitions, or controller failures.

Self-stabilization is a natural approach to meet these goals while coping with dynamic conditions, such as arrival and departure of controllers, arbitrary topology changes (*e.g.*, switch or link failures), and communication errors (*e.g.*, temporary packet losses or delays). Recall that a distributed system that is self-stabilizing will end up in a correct state independently of its initial state [3].

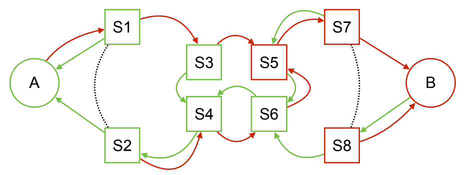
We present Medieval, a plug & play, in-band SDN control network thats supports the desired control plane services in a robust and self-stabilizing manner. Formally, we can prove the following theorem.

THEOREM 1.1. *Medieval is self-stabilizing: Given any initial configuration and set of controllers, Medieval will establish a communication network between controllers in any physically connected component.*

Medieval nicely complements ongoing related research, and can be used together with and support systems such as ONIX [6], ElastiCon [4], Beehive [7], Kandoo [5], STN [2], to just name a few. Our work also provides missing links for the interesting work by Akella and Krishnamurthy [1], whose switch-to-controller and controller-to-controller communication mechanisms rely on strong primitives such as consensus protocols, consistent snapshot and reliable flooding, which are not currently available in OpenFlow switches.



(a) Controllers "conquer" switches adjacent to their regions of control and build a spanning tree for controller-to-switch connectivity.



(b) Per-controller global spanning trees provide controller-to-controller connectivity.

Figure 1: Medieval controllers iteratively explore the network, take control over unmanaged switches (a) and build per-controller spanning trees (b).

2. MEDIEVAL

The goal of Medieval is to place each switch in the network under the control of a single controller and to establish routes that support connectivity to the switches and connectivity between controllers. To do so, each controller runs the same algorithm continuously, reacting to any change in the

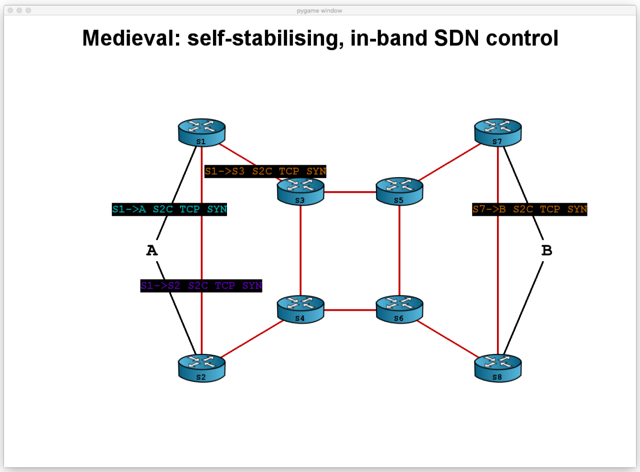


Figure 2: A screenshot from our graphical demonstration of Medieval.

# controllers	1	2	3	4	5	6	7	8
Time (ms)	9382	6983	6150	4224	6035	5104	3704	3680

Table 1: Time to conquer all switches in a Fat-Tree $k = 4$.

network (*e.g.*, due to failures or additions of switches, links, or controllers) in a self-stabilizing manner.

Medieval establishes connectivity in the control network by creating and maintaining two distinct spanning trees *for each controller*:
(1) A *per-region spanning tree* (Figure 1a) — a bi-directional spanning tree that spans over the *region* owned by the controller. The region owned by a controller is a connected graph containing the controller and switches it controls.
(2) A *network-wide spanning tree* (Figure 1b) — a spanning tree directed and rooted at the controller that spans over the whole network; *i.e.*, it supplies each switch and all other controllers with a path to reach the controller. The aim of this second spanning tree is precisely to enable each controller to reach any other controller.

Figure 1 illustrates at a high level an example instance of Medieval. Figure 1a shows the region's spanning trees of two controllers A and B. A's region comprises switches $S1-S4$, and B's region all other switches except $S6$. $S6$ has yet to connect to a controller, as denoted by the fact that this switch is broadcasting an ARP packet to its neighbors in order to resolve the controller's IP address. Figure 1b shows the two fully established network-wide spanning trees as colored arrows that indicate the path towards the two color-coded controllers.

Initially, the region of each controller only includes the controller itself. At this point, the switches that are directly connected to the controller can be added to that controller's region. When this is done, switches that are 1-hop away can be added to the controller's region, *etc.*

In Medieval, we pre-configure each switch with a virtual controller IP and a set of *a priori OpenFlow rules*, which apply to control packets. A switch initiates the connection with the controller by resolving this IP address via an ARP request that is broadcasted to all ports. In the absence of failures, the switch connects to the first controller whose ARP reply it receives. Once the controller establishes an OpenFlow session with the switch, the controller proceeds as a first step to install into the connected switch certain *ownership rules*. These rules override the a priori ones and are associated with an activity timeout.

This mechanism maps to region growth as follows. Initially, controllers can only answer the ARP request of switches directly connected to them. After taking ownership of a switch S, a controller installs rules on the switch. In particular, these rules enable ARP requests of switches connected to S to reach the controller.

3. PRELIMINARY EVALUATION

To validate our approach, we also implemented a Medieval prototype as an emulator in Java. Our unoptimized prototype emulates OpenFlow switches and controllers using separate lightweight threads, while links are modeled by message queues. We made a preliminary experiment in which we used a Fat-Tree topology with $k = 4$. We used between 1 to 8 controllers, each attached to a single edge switch and measured the time it took for all switches to become managed. Table 1 illustrates the results. We observed a roughly linear trend for this metric, which is what was expected.

4. DEMO

Figure 2 illustrates our graphical demonstration of Medieval in an example scenario. In this figure, switches $S1$ and $S7$ are establishing a TCP connection with the controller closest to them. In our demo, we present the complete execution of Medieval until its convergence.

Acknowledgments. We are thankful to Nicolas Laurent for his contributions to prior versions of Medieval.

5. REFERENCES

[1] A. Akella and A. Krishnamurthy. A Highly Available Software Defined Fabric. In *HotNets*, 2014.
[2] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *INFOCOM*, 2015.
[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, Nov. 1974.
[4] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an Elastic Distributed SDN Controller. In *HotSDN*, 2013.
[5] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *HotSDN*, 2012.
[6] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.
[7] S. H. Yeganeh and Y. Ganjali. Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking. In *HotNets*, 2014.