Ground Control to Major Faults: Towards Fault Tolerant and Adaptive SDN Control Network

Liron Schiff[†] Stefan Schmid[‡] Marco Canini[*]
[†]Tel Aviv University [‡]TU Berlin & T-Labs [*]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 to be done in-band, without dedicated control network, and without relying on legacy protocols. This paper promotes a distributed systems approach to build and maintain connectivity between a distributed control plane and the data plane. In particular, we make the case for a *self-stabilizing* distributed control plane, where from any initial configuration, controllers self-organize, and quickly establish a communication channel among themselves. Given the resulting managed control plane, arbitrary network services can be implemented on top. Based on these concepts, we develop Medieval, a plug & play distributed control plane which supports automatic topology discovery and management, as well as flexible controller membership: controllers can be added and removed dynamically. Interestingly, Medieval is also self-reliant, in the sense that it is based on OpenFlow only, and does not require any legacy protocol for the bootstrap. Medieval also comes with interesting security features.

1. INTRODUCTION

In traditional networks, the control software is distributed across all devices, which run routing protocols to compute forwarding state. An advantage of this design is that legacy networks can use in-band control: that is, control plane packets are carried over the same data plane network as with the regular traffic. Because legacy routing protocols are designed to exchange data between neighboring routers and do not require global state, they are self-stabilizing in the sense that they can automatically bootstrap the network and converge to a valid operating state from any initial conditions.

In contrast, Software Defined Networking (SDN) advocates for software-based control of a network based on a logically-centralized global network view. A number of research and development efforts have illustrated that this approach provides several benefits in terms of improved flexibility and performance, easy of management and decreased operational complexity, and lower costs [4, 9, 10, 16, 18]. While the literature has well articulated several benefits of the separation between control and data planes, the question of how connectivity between these planes is maintained – that is, paths for supporting switch-controller or controller to controller communication – has not received much attention. In practice, most SDN deployments [9–11] use out-of-band control, where control plane packets are carried by a dedicated management network. The management network runs its own routing system, which typically is realized using traditional routing protocols such as STP or OSPF.

In-band control is desirable for several reasons including its economical benefits (in certain contexts, such as carrier networks, out-of-band control would be prohibitively expensive). In essence, with in-band control there is no reason to build, operate, and ensure the reliability of a separate network. Also, out-of-band networks are typically underprovisioned and have limited redundancy. In these conditions, congestion losses are possible and link failures can lead to a partitioned SDN control plane even though the data plane network is not disconnected. This raises several concerns regarding the availability of the SDN architecture — for instance, can we guarantee that, if the data plane network is a connected graph, the SDN control plane will always establish a route between every pair of nodes in the network?

Moreover, in-band control in SDN is an appealing prospective for transitioning from legacy networks [13]. However it is also a challenging one, as others have noted, *“by extricating the control plane out of network devices and implementing it using a distributed system, SDN inherits the weaknesses associated with reliable distributed services”* [1]. These challenges are further amplified by the fact that the logically-centralized SDN control plane actually means physically distributed, for a plethora of reasons, including reliability, availability, scalability and latency [3, 8, 12].

This paper makes a first step towards the design of a fault-tolerant (namely self-stabilizing), distributed SDN control plane. We first present a model which captures the underlying fundamental challenges of self-stabilizing control planes, and the model also allows us to reason about and develop basic self-stabilizing mechanisms.

We then show that the mechanisms derived from our model can readily be implemented in today’s Openflow. Our main contribution is Medieval, an in-band control system that coordinates distributed controllers in a self-organizing manner, to bootstrap a connectivity service between controllers and switches. Medieval is based on a plug&play paradigm: it supports a dynamic membership of controllers, in the sense that new controllers can be added to the network, or old controllers removed from the network, at any time and without explicit notice; despite these changes, Medieval will ensure that unmanaged switches are assigned again to controllers, and a communication network between controllers is established. Once this "bootstrap problem" has been solved, SDN control planes can be realized on top of Medieval. Medieval is self-reliant and represents an "SDN-only" approach: it is based on OpenFlow only, and does not depend on any legacy protocols.

The name of Medieval is due to its intelligent embedding of a set of per-controller spanning trees, whose control region is continuously extended towards neighboring unmanaged switches — reminiscent of a medieval, feudal warfare.

2. MOTIVATION

The SDN control plane, that is, a collection of network-attached servers, must have connectivity to the data plane. The question we explore in this paper can be stated as: is it possible to rely on the same and already existing primitives by which SDN applications govern the network to bootstrap and provide connectivity between control and data planes? We find a positive answer by building upon the concept of self-stabilization. A distributed system that is self-stabilizing will end up in a correct state independently of its initial state [6, 17]. That correct state is reached after a finite number of steps (the convergence time).

We advocate for a software-driven, in-band control mechanism to support the broader goal of building distributed SDN control plane. To make this case, let us consider what services are required by a logically-centralized control plane (such as Onix [12] or STN [3]).

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.

Security: to ensure controller and switch authentication as well as session encryption. In addition to a secure bootstrap, control traffic may have to be given priority over other traffic, *e.g.*, to prevent some types of DoS attacks.

Self-stabilization is a natural approach to meet these goals while coping with dynamic conditions, such as arrival and departure of controllers [7], arbitrary topology changes (switch or link failures), and communication errors (temporary packet losses or delays). We emphasize that, as usual in the framework of self-stabilizing algorithms, anytime before, during, and after a self-stabilization, additional changes may occur: Medieval will simply reconverge starting from the current state.

3. MODEL AND CHALLENGES

The design of a self-stabilizing control plane which assigns switches to controllers raises some fundamental challenges. This section introduces a model which captues some of the particularities of such a system, and which serves as a basis for reasoning about self-stabilizing mechanisms. As we will see, the postulated mechanisms can be implemented in today's OpenFlow protocol, and lie at the heart of Medieval, the main contribution of this paper.

Essentially, we consider a network connecting two kinds of components, *controllers* and *switches*, henceforth called *nodes*:

1. **Controllers:** We assume that each controller has a unique address, but the controllers also share a multicast address. Controllers must be able to solve consensus. In principle, a controller can perform arbitrary local computations (i.e., it is a Turing machine).

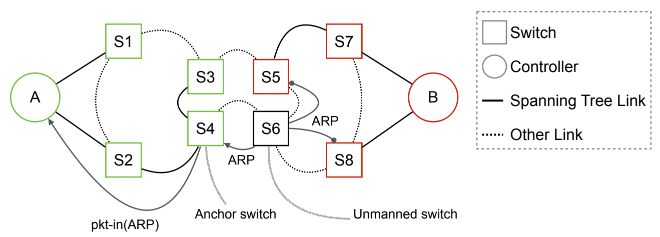
2. **Switches:** Switches also have unique addresses, and are simple and "passive" devices: they cannot perform any computations on their own. In principle, a switch can be in one of two possible states: *managed* or *unmanaged*. The configuration of a switch is defined by a set of match-action rules with priorities and (possibly infinite) time-out values. The configuration can only be changed (1) either by a controller or (2) through a time-out. We want to minimize the knowledge a switch has about the controllers, and assume that the switch initially only knows the multicast address shared by all controllers.

A first fundamental challenge of a self-stabilizing system regards the detection of inactivity: a switch must notice when it is unmanaged. As switches cannot perform path computations or establish paths through other switches, the responsibility to establish communication paths to switches lies at the controller: A challenge here concerns the fact that a controller may also have to manage "remote switches", switches which are not directly connected to the controller. In order to establish a route to remote switches, a controller must be able to install forwarding rules on "relay switches". Moreover, switches relaying the control traffic to remote switches must be able to classify traffic, e.g., differentiating between regular traffic, control traffic from or to different controllers, etc.

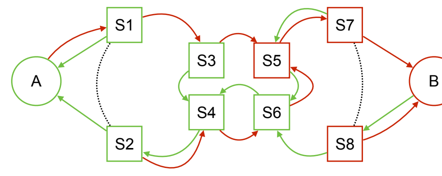
Last but not least, there is the challenge of providing fault-tolerance: all the above mechanisms need to tolerate a number of different failures: link failures, controller failures, TCP timeouts, congestion, etc. may all lead switches to become unmanaged.

4. SELF-STABILIZING MEDIEVAL

Given our model and conceptual discussion, we now show that many of the discussed concepts can actually be implemented in today's OpenFlow protocol. In particular, in this section, we present the Medieval system.



(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 exploring the network, taking control over unmanaged switches (a) and building per-controller spanning trees (b).

4.1 Overview and Building Blocks

Before going into the implementation details, we provide an overview of how Medieval addresses the fundamental challenges identified above.

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

In order to detect inactivity, Medieval leverages *rule timeouts* at the switches. In particular, when rules time out, a switch falls back to a set of "safe rules": these rules can be implemented as low priority rules which are hidden as long as other, managed rules are installed (and regularly refreshed) on top by a controller. Indeed, timeouts combined with low priority rules are a vital trick to render a system self-stabilizing: these low priority rules will eventually appear once all other rules time out, and prevent the switch from being permanently cut off from the network.

Concretely, Medieval defines an unmanaged switch configuration with statically installed rules which make sure that neighboring switches learn about the unmanaged switch. While the switch is not able to take actions toward finding and reconnecting to a controller, by informing the neighboring, managed switches, a controller will eventually learn about inactive switches. Medieval does not rely on any statically pre-configure controller IP addresses (this approach would not satisfy the controller plug-and-play requirement), and hence switches are pre-configured with IP anycast rules: a logical IP addressed shared by *all controllers*.

In order to manage a remote switch, the Medieval controller installs rules on its neighboring switches, which can then be used to install rules on switches two hops away, etc., iteratively expanding the controller domain. This iterative extension of controller domains is also the reason for our system's name: $Medieval$.

A standard and efficient approach to provide connectivity between controllers and managed switches are spanning trees, and indeed, spanning trees are useful tool also in our setting. In our case, it is not sufficient to use standard Layer-2 STPs, as such trees do not give us the control to match switches to controllers. In fact, for our self-stabilizing algorithm $Medieval$ we even use two trees: the first tree type is used by each controller to communicate with its switches, and the second type is constructed by each controller to allow all other controllers to reach it.

4.2 Medieval

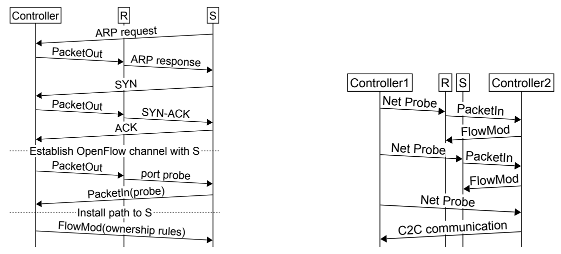
With these concepts in mind, we can now provide more details on the internals of Medieval. As we will see, Medieval can readily be implemented in OpenFlow.

As discussed, Medieval establishes connectivity in the control network by creating and maintaining two distinct spanning trees: (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.

At any moment in time, a switch is either managed or unmanaged (see Table **??**). When a switch is unmanaged, it broadcasts (only) its connection attempts (i.e., ARP requests for the controller multicast address) to all ports. Controller responses are forwarded from all ports to the switch management stack. When the switch becomes managed, the controller installs a set of match-action rules binding the communication to the controller tree. This configuration has higher priority than the unmanaged configuration but it has timeouts, so in case the switch becomes unmanaged, the managed configuration expires.

Note that when a switch becomes unmanaged, it no longer forwards any packets of other switches up/down the region tree, thereby causing them to become unmanaged as well. While this may seem suboptimal, many failures (e.g., of links) on the path between the controller and a switch will also affect the children and descendants of that switch (with respect to the tree topology). Allowing an unmanaged switch to serve other switches, conflicts with the requirement of contiguous and rooted management trees, and renders it hard to ensure self stabilization.

Algorithm 1 presents a pseudo-code version of the algorithm run by each Medieval controller. This algorithm – in combination with the switch behavior defined by the OpenFlow forwarding model – seeks to create two spanning trees to support connectivity for two classes of communications: (1) between switches and controllers, and (2) among con-



(a) **(b)**

Figure 2: (a) The management process of newly detected switch S through already managed switch R. (b) The creation of network wide spanning trees and controller to controller channels.

trollers.

Example. 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 virtual controller IP. Figure 1b shows the two fully established network-wide spanning trees as colored arrows that indicate the path towards the two color-coded controllers.

Building the Region Spanning Tree. 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.

A switch initiates a connection to a controller by resolving the pre-configured controller IP address via an ARP request that is broadcasted to all ports, except the mgmt port (rule *flood_mgmt*). In the absence of failures, and with common ARP table cache implementations, the switch connects to the first controller whose ARP reply it receives. When the switch connects to a controller, we say that the switch becomes connected. Informally, the switch becomes managed once the controller has taken ownership of it (see details below; § 4.3 gives a formal definition). A switch is unmanaged if it is neither connected nor managed.

Once the controller establishes the OpenFlow session (a TCP connection) with the switch (see details below), the controller proceeds as a first step to install into the connected switch the ownership rules described in Table 1. These rules override the a priori ones and are associated with an activity timeout; upon timer expiration (*e.g.*, due to failures) the switch returns to the a priori configuration from which the region spanning tree can be reconstructed. We discuss failures below.

Note how this mechanism maps to region growth: 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.

```
1  process event at ctrl :
       // Region Spanning Tree Events
2      on ARP request from directly-connected switch $S$:
3          - send ARP reply to $S$
4          - accept the TCP connection initiated by $S$
5      on ARP request from switch $S$ (via packet-in):
6          - save $S$'s anchor switch $R$ and anchor port
7          - configure all packets from / to $S$ to be relayed through packet-in /
           packet-out messages via $R$
8          - send ARP reply to $S$
9          - accept the TCP connection initiated by $S$
10     on OpenFlow session established with switch $S$:
11         - send a port probe packet to $S$
12         - install the $ctrl \rightarrow S$ path by sending downstream rules to every
           switch on the path
13         - periodically send a path probe packet to $S$ – but not via packet-out
           – until a packet-in with the probe is received
14     on port probe answer from $S$:
15         - send the ownership rules to $S$
16     on path probe answer from $S$:
17         - configure all packets from / to $S$ to use the normal path (don not
           tunnel using packet-in and packet-out messages)
18     on OpenFlow session terminated with switch $S$:
19         - remove downstream rules from switches on the $ctrl \rightarrow S$ path

       // Network-Wide Spanning Tree Events
20     on $C$'s network-wide probe as packet-in from $S$:
21         - install network-wide spanning tree rules for $C$ on $S$
22     on $C$'s network-wide probe:
23         - periodically attempt establishing a connection with $C$, until it
           succeeds or network-wide probes are no longer seen for a certain
           amount of time
24     on timeout of rule $net\_wide\_flood$ on switch $S$:
25         - remove rule $net\_wide\_block$ from $S$

       // Periodic Events
26     on periodically:
27         refresh the ownership rules and downstream rules on the switches we
           control
28     on periodically:
29         send a network-wide probe to all directly-connected switches we
           control
```

Algorithm 1: Pseudo-code executed by controller $ctrl$. C and S refer to a controller and switch, respectively.

To establish its region spanning tree, a controller needs to discover how switches in the tree are connected to one another, down to the port numbers being used. When a controller takes ownership of a switch, the *arp_to_ctrl* rule installed on the switch instructs it to send any ARP request it receives to the controller encapsulated in an OpenFlow packet-in message. The packet-in meta-data contains the input port number on which the ARP packet was received.

When an unmanaged switch attempts to connect to a controller, we call **anchor switch** the first switch of each controller's region that receives the ARP request of the connecting switch. Hence, there is at most one anchor switch per switch-controller pair. Unless an unmanaged switch is directly connected to the controller (lines 2-4), it is always the anchor switch that forwards the ARP request to the controller via a packet-in message. When a controller receives an ARP request via an anchor switch, the ARP reply is then delivered to the unmanaged switch via an OpenFlow packet-out message sent to the anchor switch (lines 5-9).

name	match	output	prio	overrides

arp_to_ctrl	eth_type = ARP
	target = CTRL_IP
		CTRL	2	/

us_to_ctrl	in_port = MGMT_PORT	*S's upstream port*	3	flood_mgmt
all_to_ctrl	in_port = MAC_C	*S's upstream port*	3	/

Table 1: Ownership rules. These rules are installed by a controller C on a switch S after C and S have established an OpenFlow session and C has received the port probe answer to learn S's upstream port (the port of S through which it is attached to C's region). *CTRL_IP* is the global virtual controller IP.

name	match	output	prio
forward_to_owned_switch	ip_dst = IP_{S2}	*$S2$'s anchor port on $S1$*	2

Table 2: The downstream rule. This rule is installed by a controller C on each switch $S1$ that lays on the path between C and a switch $S2$ with whom C has established an OpenFlow session.

At this point, the controller still does not have a direct path to send traffic to the unmanaged switch, *i.e.*, it must send all packets intended for the switch through the anchor switch's OpenFlow session. In the reverse direction (switch to controller), a path exists between the anchor switch and the controller, but the control traffic sent by the switch is still flooded on all its ports.

Note that even though the flooding is still ongoing, all switches other than those in the controller's region will drop these packets, as they carry the virtual controller IP address and the unique MAC address of the controller, and only switches in the controller's region can match on this combination. It is possible, however, that the packet will be relayed by multiple switches in the region, but this will be handled properly by the controller, using the TCP sequence number.

To complete the setup and make that switch part of the controller's region (hence allowing that switch to serve as anchor switch itself), the controller must perform two steps. First, to stop relaying traffic over the anchor switch, the controller must establish a path from itself to the switch. It does so by installing a rule on every switch on its path to the connecting switch in the region's spanning tree (line 12). This rule, called *forward_to_owned_switch*, is shown in Table 2.

To confirm that all such rules have been installed properly, the controller sends a *path probe packet* on the path to the switch, and makes the switch to this path once it receives an answer from the switch (lines 13 and 16-17). The answer from the switch is simply a packet-in encapsulating the original probe packet, resulting from the a priori rule called *path_probe*.

Second, to stop the now-managed switch from flooding all its control packets, the controller must learn the *upstream port* of the switch: the port number through which the connecting switch is connected to its anchor switch. Once this number is known, it can be used to route all packets going from the switch to the controller via the anchor switch (rule *us_to_controller*).

To learn the upstream port, the controller send a *port probe packet* to the switch, via the anchor port of the anchor switch (line 11). The a priori rule *port_probe* ensures this packet is sent back to the controller via packet-in. The controller can determine the upstream port by the in-port in the packet-in meta-data. Once the upstream port is known, the controller installs the *ownership rules* on the switch (lines 14-15 of the algorithm). These rules enable the switch to serve as an anchor switch itself (rule *arp_to_virtual_controller*); enable it to forward packets towards the controller (*all_to_controller*), and directs its control traffic towards the controller instead of flooding (*us_to_controller*).

name	match	output	prio	overrides

net_wide_block	ip_dst = NET_PROBE
	mac_src = MAC_{C2}
		/	2	net_wide_probe (part)

net_wide_flood	in_port = *parent port*
	ip_dst = NET_PROBE
	mac_src = MAC_{C2}
		FLOOD	3	net_wide_block (part)

ctrl_to_ctrl	ip_dst = IP_{C2}	*parent port*	2	/

Table 3: Network-wide spanning tree rules. These rules are installed by a controller $C1$ on a switch S it manages whenever the first network-wide probe packet from controller $C2$ is received by S. We call *parent port* the port of S on which the first network-wide probe from $C2$ was received. *NET_PROBE* is the IP address signaling a network-wide probe packet.

Building the Network-Wide Spanning Tree. With the region's spanning tree, a controller can communicate with all the switches it controls. In order to be able to communicate with all other controllers, we make use of a per-controller network-wide spanning tree. Each controller constructs its network-wide spanning tree concurrently with the region's spanning tree.

A controller A constructs its network-wide spanning tree by periodically sending out a special packet: **network-wide probe** (lines 28-29). When a managed switch receives controller A's network-probe for the first time, it will encapsulate it in a packet-in message and send it to its owning controller (rule *net_wide_probe*, say B. Note that A and B can be the same controller. B will then install A's network-wide spanning tree rules (Table 3) on the switch.

The network-wide spanning tree rules ensure that further probes received from A *on the same port* as the first probe will be flooded to all other ports (rule *net_wide_flood*). Furthermore, any probe received from A on any other port is dropped (rule *net_wide_block*) — effectively, this prevents loops in the topology from causing broadcast storms.

Once these rules are installed, subsequent probes from A will reach one hop further in the network. Eventually, A's network-wide spanning tree will span the entire network. Whenever the spanning tree of a controller A extends to another controller, say B, B learns of a path to talk to A. This path results from all the instances of the *ctrl_to_ctrl* rule installed in the network-wide spanning tree. B will then periodically attempt to establish a connection with A (*e.g.*, TCP) (lines 22-23). These attempts will succeed as soon the spanning tree of B reaches A, hence supplying A with a return path to communicate with B.

Handling Failures. Informally, our failure model considers all the usual failures that ultimately lead to the termination of an OpenFlow session. Failure of an OpenFlow session is detected by failing to receive an answer to an OpenFlow echo-request message, which are sent periodically. To deal with a failed OpenFlow session, the affected switch reverts the flow tables to the a priori rules (using timeouts and rule priorities) and re-attempts to connect to a controller. By setting the rule timeout to be larger than the timeout of the echo-request message, we can ensure that the rules will never be dropped while the OpenFlow session is live.

Note that the way the switch handles data plane traffic, depends entirely on the controller. Medieval does not impose a pre-defined behavior for this. For instance, a controller application that needs to react to Packet In messages will be unavailable for a brief period of time. But existing data plane rules will still match the ongoing traffic.

To avoid the overhead of refreshing a large number of rules, the non-initial rules could be grouped into a separate secondary flow table.[1] Then, a single rule in the first flow table could be used to pass all matching packets to this secondary table; and so, only this catch-all rule needs to be refreshed.

Note that a controller failure is just a special case of an OpenFlow session failure. When that happens, the switch will still try to reconnect to a controller, it just will not be able to reconnect to the same as before. After its session is terminated, a switch will try to reconnect to a controller. The OpenFlow specification does not specify the delay. Assuming we can control it, it should be set higher than the rule refresh timeout; therefore ensuring we only try to reconnect after reverting to the a priori rules. The controller also detects the termination of the session with a switch S and uninstalls the downstream rules for S from switches on the controller-S path (lines 18-19).

Furthermore, due to failures, the network-wide spanning-tree may need to be altered. Unlike the region's spanning tree, we cannot rely on switches reverting to their a priori rules after a spanning tree experiences a partition. To solve this problem, we set an idle timeout on the *net_wide_flood* rule. As long as there are probe packets, the rule persists. After the probes stop, the rule eventually expires. The controller of the switch will be notified of the rule expiration and will also remove the *net_wide_block* rule from the switch, so that other controllers have a chance to re-include the switch in their network-wide spanning trees.

Finally, we note that since the control traffic is sent in-band, steps should be taken to prevent congestion caused by regular traffic to negatively impact the control traffic. We have not implemented this yet; but we believe a solution based on strict priority queues could be made to work without major difficulties.

4.3 Evaluation and Discussion

Given our construction and algorithms, we can prove the following theorem.

THEOREM 4.1. *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.*

Due to space limit we omit our proof, which we provide in a companion technical report.

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

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

To further validate our approach, we also implemented a Medieval prototype as an emulator in Java. Our *unoptimized* prototype emulates OpenFlow switches and controllers in separate lightweight threads, while links are modeled by message queues. We run preliminary experiments (Figure 4) in which we use a Fat-Tree topology with $k = 4$. We use between 1 to 8 controllers to edge switches (each controller is connected to a single edge switch) and measured the time it took for all switches to become managed. We observe a roughly linear trend for this metric, which is what was expected. After investigating the small spike for 5 and 6 controllers, we suspect that it is an artifact of the scheduling algorithm of the Quasar threading framework [14] used by our prototype.

5. CONCLUSION

Interestingly, while the benefits of the separation between control and data planes have been studied intensively in the SDN literature, the important question of how to connect these planes has received much less attention.

Medieval nicely complements ongoing related research. In particular, Medieval is not limited to a specific technology, but offers flexibilities and can be configured with various additional robustness mechanisms, such as warm backups, local fast failover, or alternatives spanning trees [2, 15].

Moreover, Medieval can be used together with and support systems such as ONIX [12], ElastiCon [7], Beehive [19], Kandoo [8], STN [3], to just name a few. Our paper 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.

We also note that Medieval comes with interesting security features. First, Medieval can be used in parallel with other security measures (public key encryption) to bootstrap authentification and encrypted communication channels. Moreover, Medieval can be used to give control traffic a higher priority than other traffic, thereby preventing some types of DoS attacks. The global spanning trees can allow each switch to sense and communicate with all controllers in its connected component, which can be useful to bootstrap more advanced security protocols, such as SNBI [5], which needs to deliver encryption keys and NTP packets.

We believe that our work opens several interesting directions for future research. In particular, it would be interesting to generalize the robustness properties of the system to cover also Byzantine behavior.

[1]Flow table pipelining is available since OpenFlow 1.1.

6. REFERENCES

[1] A. Akella and A. Krishnamurthy. A Highly Available Software Defined Fabric. In *HotNets*, 2014.

[2] M. Borokhovich, L. Schiff, and S. Schmid. Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms. In *Proc. ACM SIGCOMM HotSDN*, 2014.

[3] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *INFOCOM*, 2015.

[4] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *SIGCOMM*, 2007.

[5] CISCO. Secure network bootstrapping infrastructure. In *https://wiki.opendaylight.org/view/Project_Proposals:Secure_Network_Bootstrapping_Infrastructure*, 2015.

[6] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, Nov. 1974.

[7] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an Elastic Distributed SDN Controller. In *HotSDN*, 2013.

[8] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *HotSDN*, 2012.

[9] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.

[10] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[11] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[12] 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.

[13] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann. Panopticon: Reaping the Benefits of Incremental SDN Deployment in Enterprise Networks. In *USENIX ATC*, 2014.

[14] Parallel Universe. Quasar: Lightweight threads and actors for the jvm. http://docs.paralleluniverse.co/quasar/.

[15] M. Parter. Dual failure resilient BFS structure. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 481–490, 2015.

[16] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[17] M. Schneider. Self-stabilization. *ACM Comput. Surv.*, 25(1):45–67, Mar. 1993.

[18] N. Vasić, D. Novaković, S. Shekhar, P. Bhurat, M. Canini, and D. Kostić. Identifying and Using Energy-Critical Paths. In *CoNEXT*, 2011.

[19] S. H. Yeganeh and Y. Ganjali. Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking. In *HotNets*, 2014.