A Highly Available Software Defined Fabric

Aditya Akella
UW-Madison
akella@cs.wisc.edu

Arvind Krishnamurthy
University of Washington
arvind@cs.washington.edu

ABSTRACT

Existing SDNs rely on a collection of intricate, mutually-dependent mechanisms to implement a logically centralized control plane. These cyclical dependencies and lack of clean separation of concerns can impact the availability of SDNs, such that a handful of link failures could render entire portions of an SDN non-functional. This paper shows why and when this could happen, and makes the case for taking a fresh look at architecting SDNs for robustness to faults from the ground up. Our approach carefully synthesizes various key distributed systems ideas – in particular, reliable flooding, global snapshots, and replicated controllers. We argue informally that it can offer high availability in the face of a variety of network failures, but much work needs to be done to make our approach scalable and general. Thus, our paper represents a starting point for a broader discussion on approaches for building highly available SDNs.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Distributed networks

Keywords

software defined networks; distributed control

1. INTRODUCTION

By enabling logically-central, software-based control of a network, Software Defined Networking (SDN) improves the agility of network management, lowers costs, and helps operators roll out novel network control services. To date, a number of research and development efforts have shed light on how best to leverage SDN to fully realize these benefits in operational settings big and small [6, 7, 8, 2]. The early promise shown by these efforts has spurred rapid adoption of SDN in many production settings [6, 7, 11].

However, a critical examination of the SDN architecture in its entirety has not been conducted to date. In particular, important *availability-centric* questions such as "can today's SDNs always guarantee that routes can be established between every pair of connected nodes in a network?", and more generally, "how can an SDN cope with link and node failures?" are not as well understood as SDN's flexibility and performance benefits.

A clear understanding of the availability issues induced by SDN is paramount as it helps put SDN's other benefits in context: for instance, if the management flexibility afforded by SDN comes at a steep, and fundamental, availability cost, then perhaps it does not make sense to adopt the SDN architecture in the first place.

In this paper, we examine the different components of existing SDN systems, and their reliance on a variety of networking and distributed systems protocols. We argue that SDNs today *cannot* offer high network availability, as in guarantee that *routes can be established between a pair of nodes as long as there is a physical path connecting the nodes*. Luckily, we find that it is possible to *re-architect* today's SDNs in a more principled fashion to be fault-tolerant from the ground up.

SDN Availability Issues. We start by noting that existing SDNs do not have a sufficiently clean separation of concerns among their internal modules. In particular, three key modules of SDNs – distributed consensus protocols, mechanisms for supporting switch-controller or controller to controller communication, and transport protocols for reliable message exchange – can have cyclical dependencies on each other.

These inter-dependencies imply that the failure of a handful of network links may temporarily disconnect switches from controllers or controller instances from one another. We advocate using reliable flooding to eliminate some of these dependencies.

However, existing SDNs' availability issues are rather intrinsic in nature, such that even with reliable flooding, it may not be possible to achieve high availability. For example, two switches in a connected minority component of a net-

HotNets '14, October 27–28, 2014, Los Angeles, CA, USA.

http://dx.doi.org/10.1145/2670518.2673884.

work may not be able to communicate with each other even though they have a physical path connecting them.

Why do these issues arise? Fundamentally, by extricating the control plane out of network devices and implementing it using a distributed system, SDNs inherit the weaknesses associated with reliable distributed services. Furthermore, because the distributed services are used to control the underlying network, the weaknesses become magnified.

For instance, requiring controllers to assemble a majority quorum to process network events could be impossible under partitions [3], a consequence of which is that switches in a minority partition can no longer be controlled (see Figure 1 – covered in detail in §2). Such partitions are likely to happen in practice [10]. One could potentially fix this by reconfiguring the controller set and associating with each partition its own controller set. However, this falls short in other situations, e.g., where asynchronous delivery of network events leads to inconsistent topology views at different controllers, leading some controllers to incorrectly initiate reconfiguration and attempt to exercise independent control over a subset of switches (see Figure 2 – discussed in §3).

Based on the observations above, we argue that thoroughly addressing availability requires changes both to the design of reliable distributed services as well as the SDN support in the underlying network. To this end, we present a systematic rearchitecting of SDNs and present the design for a highly available software defined fabric that addresses the above weaknesses.

We borrow classical algorithms from the distributed systems community and presents a co-design of the network layer protocols and the distributed management services that provides a high degree of availability without compromising on safety or controller consistency even under arbitrary network partitions. To complete this design, we need to address issues related to optimizing performance and interactions with network management protocols, and we leave this to future work.

2. BACKGROUND AND MOTIVATION

SDN is a popular approach because the "clean" separation it advocates between the control and data planes enables rapid prototyping of network control plane functions. We observe that it is non-trivial to realize this separation in practice without sacrificing availability today. We now substantiate this observation, starting first with an overview of state-of-the-art SDN designs, followed by a discussion of the problems that are inherent to them.

A typical SDN architecture comprises of:

- A simple data plane consisting of the forwarding state at network elements, which perform match-action functions for each incoming packet (i.e., find matching forwarding state and take the appropriate action).
- A logically central control plane, often implemented by a distributed collection of physical controller machines for availability. Based on events reported by data plane elements (e.g., a link or switch failing), the current view of network topology, and the current data plane state, the control plane computes a new forwarding state and updates the data plane accordingly. In many SDN deployments, the controllers run distributed protocols (e.g., distributed consensus protocols such as Paxos [14]) to ensure they have a consistent view of network topology and data plane state even in the presence of controller failures or disconnections.
- A control channel for communication between the data plane and control plane elements. Implemented as a point-to-point TCP connection, the control channel is used by switches to communicate events of interest to the controllers, for controllers to communicate with each other for coordination purposes, and for controllers to push forwarding state updates down to the switches. The control channel could be out-of-band: many SDNs rely on legacy distributed protocols (e.g., VLANs/MST, BGP) to install paths for implementing such out-of-band control channels [6, 7, 11]. The control channel can also be in-band, i.e., the paths computed by the control plane are themselves used to support the control channel.

2.1 Impact on Availability

The set of network and distributed systems protocols that come into play in an SDN system have complex/critical dependencies. Consider in-band control channels, where intertwined dependencies between the transport, forwarding, and distributed control plane modules become immediately apparent: distributed control plane protocols rely on forwarding and transport protocols for basic message delivery; in turn, forwarding depends on transport (for switch-controller and inter-controller communication) and on the distributed control plane for route set-up. These dependencies not only increase system complexity but also introduce complex failure modes; for example, the system might not be able to react to a switch failure if the event notification is not reliably delivered to the controllers or if the controllers themselves cannot communicate with each other (due to the same failure or other related failures) in order to achieve consensus regarding distributed control. In the case of out-of-band control channels, distributed consensus protocols are forced to depend on a separate suite of *legacy protocols*, the very protocols that SDN solutions are supposed to replace and simplify.

A second aspect of SDNs is that they are susceptible to availability issues. One can imagine such availability issues stemming from reliance on legacy protocols and the interdependencies referred to above. For example, failure of one or a handful of links can make a BGP-based network control channel dysfunctional as parts of the network may not be routable from one another for extended periods of time due to BGP reconvergence [13, 20], even though the physical network has a live path connecting them.

However, the availability issues in SDNs are more deep rooted than that and arise even in the presence of ideal control channels that don't impose any dependencies.

Consider the network depicted in Figure 1. When the $S4 - S6$ and $S4 - S8$ links fail and the network is partitioned, the data plane for the partition on the right cannot be updated since the switches there can contact only a minority group of a replicated control plane (i.e., $C4$ and $C5$). Crucially, $S6$ and $S8$ cannot communicate with each other if the previously configured path between them is not wholly contained in the new partition. In §3, we argue that simple fixes to this only address part of the availability issue.

Crucially, legacy distributed protocols such as OSPF do not suffer from this issue: when partitions happen, OSPF routers re-converge to new intra-partition routes. In other words, current SDN designs fail to provide important fault-tolerance properties, which renders SDNs *less available* than traditional networks in some situations!

3. DESIGN

In this section, we outline the design of a fault-tolerant SDN fabric that can provide a high degree of availability in the presence of various kinds of network failures. Our design presents a rethinking of the SDN architecture, from the control channel up to the control plane, to address dependencies and avoid pitfalls intrinsic to SDNs today. We present the design as a series of refinements starting from a baseline design, with each refinement addressing some aspect of availability or controller consistency.

3.1 Baseline Design

We start with a baseline design that guarantees consistency of controller state and allows for switches/controllers to communicate reliably but does not necessarily provide high availability. It incorporates two sets of mechanisms.

Reliable Flooding: To completely free the control channel from all dependencies, we advocate a control channel based on reliable flooding. Such a control channel can be established without depending on routing or transport protocols.

Replicated Controllers: Controllers are made robust to failures using *state machine replication* and reliable flooding. In particular, controllers use a consensus protocol such as Paxos [14] to obtain consensus from a quorum (typically a simple majority of controllers) before responding to changes in topology and system configuration. For example, whenever there is a topology change (e.g., a link down or a link recovery event), the controllers come to a consensus that the corresponding event will be the next event that will be handled by the controllers and communicate to the switches the new set of forwarding rules based on the updated topology. All communications between controllers in order to reach consensus and communications from controllers to switches are handled using reliable flooding.

This baseline design is robust to controller failures (as long as a majority are alive) and can also reliably handle communications between controllers and switches even if the point-to-point forwarding rules are stale with respect to the current topology.

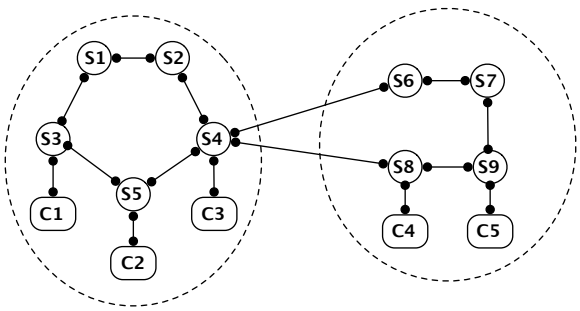

Figure 1: Impact of network partitions.

Pitfalls: While the baseline design provides resilience towards certain types of failures (e.g., the network is operable even if some of the controllers fail or if there is a failure in some of the paths between controllers and switches), it is not robust to failures that partition the controller set. Consider for example the network depicted in Figure 1. If the $S4 - S6$ and $S4 - S8$ links fail, then the network is partitioned. In this case, the routing and network policies for the partition on the right cannot be updated. This is particularly debilitating if the existing routes connecting pairs of switches in the right hand side partition are not fully contained inside the partition. For example, if the route from $S6$ to $S8$ is $S6 \rightarrow S4 \rightarrow S8$ when the network becomes partitioned, then the route cannot be updated to the working path $S6 \rightarrow S7 \rightarrow S9 \rightarrow S8$ since the switches $S6$ and $S8$ can contact only two of the five controllers (i.e., $C4$ and $C5$) and would therefore not be able to put together a majority quorum to process the topology update. Even if the switches are configured to use fast-failover in the form of a pre-configured backup path, connectivity between $S6$ to $S8$ is not guaranteed as the backup path might also not be fully contained inside the partition.

Thus, the baseline design ensures consistent handling of dynamic changes to network topology, but it does not provide high availability in the case of network partitions. In retrospect, it is apparent that the baseline design inherits the strengths and weaknesses associated with reliable distributed services; this is just a direct consequence of software defined networking and its separation of the control plane into a logically centralized controller service. The ability to perform network management operations is contingent on the controller service being able to assemble quorums to process network events. The availability limits associated with distributed services (e.g., the CAP theorem [3]) can be overcome only with some form of support from the underlying network, which is the focus of the remainder of this section.

3.2 Partitioned Consensus

A simple modification to the above design is to allow for

dynamic reconfiguration of the controller set. When a partition takes place, we can try to split up the controller set into two or more groups of controllers and associate with each partition its own controller set. Topology updates to a given network partition need only be handled by the controllers residing inside of the partition and need not require a majority quorum over all of the controllers in the network. This allows each individual partition to independently tear down old routes, install new routes, and be reactive to local network changes. We briefly outline below a refinement to the baseline design that allows for partitioned operation.

Link down: When a link between two network elements X and Y fails, both X and Y flood the topology update through the network. When this topology update is received by a controller, it computes the impact of the topology update and identifies other controllers that reside in the same network partition as itself. The controller then uses the consensus protocol to have the corresponding controller set process this link failure as the next update to the network topology. If the link failure results in a partition, then the controller sets in the two partitions will process the link failure and subsequent topology changes independently.

Link recovery: When a link between two network elements X and Y becomes available, the new link might restore connectivity between two network partitions. In such cases, we might have to reconcile the state associated with the controller sets of the two partitions. This requires the following (more elaborate) protocol that can be concisely characterized as a two-phase commit protocol layered over consensus. Let C_X and C_Y be the controller sets associated with the two partitions containing X and Y. We first pass a consensus round in each of X and Y regarding the introduction of the new link (i.e., we require a separate quorum consensus within C_X and C_Y). If both of these consensus operations succeed, we view the two partitions to be *prepared* for the merger of the two network partitions, and we then pass an additional consensus round on each of the controller sets to *commit* the merged topology. If either of the initial prepare rounds don't succeed (say due to a concurrent failure/network event), then the consensus operations are rolled back and retried.

Pitfalls: While this refinement allows for dynamic partitioning of controller sets and independent management of network partitions, there are consistency pitfalls especially in the presence of concurrent network events. Consider the network topology depicted in Figure 2 and let the network link $(S2, S6)$ fail just as the link $(S4, S8)$ comes back online. The corresponding network events will be flooded to the controller sets, and there might be race conditions in how they are handled. One possible outcome is that the link failure event is received first by controllers $C4$ and $C5$, who will then proceed to process this event as the onset of a network partition and elect themselves as the sole controllers managing switches $S6$, $S7$, $S8$, and $S9$. At the same time, let the link recovery event be the first event processed by the controllers $C1$, $C2$, and $C3$. From their perspective, they view this link recovery as a normal topology update (i.e., one which doesn't repair network partitions) and can rely on a simple majority quorum of $C1$, $C2$, and $C3$ to process this update. We now have two different controller sets who can independently perform network management operations on switches $S6$, $S7$, $S8$, and $S9$, which can result in conflicting management operations, inconsistent routing, and degraded availability.

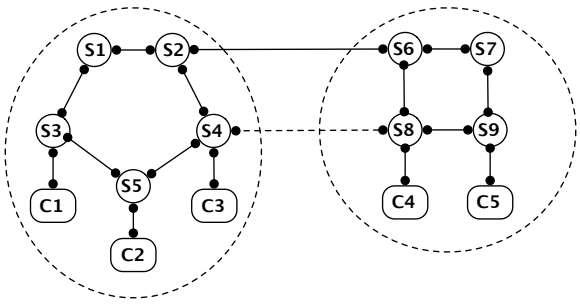

Figure 2: Handling of concurrent network events.

3.3 Whole Quorum Controller Consensus

The incorrect operation of controller nodes in the example outlined above illustrates the challenges associated with splintering the controller set especially when a link failure is erroneously flagged as a network partition event; the larger controller set might still be able to manage all of the switches since they have a sufficient quorum with respect to original controller set, while the smaller splinter group operates under the assumption that it is managing its own separate network partition.

One possible refinement is to use *whole quorum* consensus over a controller group as opposed to a simple majority quorum consensus, i.e., actions performed by the controllers require consensus involving every member of a controller set. This would prevent the majority set from achieving consensus without requiring consent from the members of a splinter group, and thus we will not have scenarios where dueling controller sets are managing the same set of switches.

It is worth noting that whole quorum consensus is typically not meaningful for distributed systems since the purpose of replicating a service is to allow it to operate in the presence of failures of service nodes using simple majority quorums. Our setting deals with controller failures by refining the controller set to exclude failed nodes and requiring whole quorum consensus on the remaining controller nodes. This means that controller failures are explicitly handled, and this failure processing has to be performed before any other network management operations are effected by the controller set. Note that a temporarily unreachable controller node would be first removed from the controller set and then explicitly added later when connectivity to it is restored.

Pitfalls: While the use of whole quorum controller consensus can prevent conflicts in how switches are managed

by different controller groups, there are complications in how state reconciliation can be performed across controller groups. Consider again the scenario depicted in Figure 2 where the link failure event is handled by $C4$ and $C5$ (which would result in the formation of a splinter group), and the link recovery event is received by $C1$, $C2$, and $C3$ before the link failure event. In such a setting, $\{C1, C2, C3\}$ view the link recovery as an ordinary topology update while $\{C4, C5\}$ believe that it is a partition repair operation. Further, $\{C1, C2, C3\}$ and $\{C4, C5\}$ disagree on the existence of the $(S2, S6)$ link and might also have diverging views on other link failure/recovery events processed independently by $\{C4, C5\}$ if they had happened in the interim.

3.4 Network-wide Transactional Updates

We present a complete design that pulls together the mechanisms presented in the earlier iterations. In particular, we use a network-wide topology snapshot operation (heavily based on the consistent snapshot algorithm [4]) along with the mechanisms of reliable flooding, replicated controllers, and whole quorum consensus. The resulting design ensures consistent state management, preserves a high degree of availability, and facilitates reconciliation of partitions.

The network comprises of switches, end-hosts, and controllers. We will refer to these using the generic term *nodes*. Each node maintains the following local information: a *snapshot sequence number (ssn)* (initialized to zero when the node is first introduced into the network), the set of links to neighbors, and the most recent *ssn* value received from each of its neighbors (*nssn*). We will assume that at the beginning of an epoch node n_i has initialized its *nssn*(n_j) to *null* for each of its neighbors n_j. When a node detects a change in its connectivity state (e.g., failure or recovery of an adjoining link), it triggers a partition-wide snapshot operation by sending a *take snapshot* message to itself. The following actions are performed by a node n_i upon receiving the *take snapshot* from n_j for the first time in this snapshot epoch (i.e., when *nssn*(n_j) is *null* for all of the node's neighbors):

1. If the message is from a neighboring node (as opposed to a initial snapshot message sent by a node to itself), set *nssn*(n_j) to the neighbor's *ssn* value received with the message;
2. increment its own local *ssn*;
3. send a *take snapshot* message to each of its neighbors along with the incremented *ssn* value that replaces the neighbor's *ssn* value received with the original message;
4. set a timeout for receiving a *take snapshot* message from the other neighbors (except n_j);
5. upon timing out or after collecting all *nssn* values, compose a *link state* message that contains the following information: (a) local *ssn* value, (b) the neighbors from which it received a *take snapshot* message, and (c) the *nssn*() values received along with them;
6. flood this message to all of the controllers; and
7. reset all *nssn*() to *null*.

Controllers collect all of the *link state* messages from the nodes and perform the following local consistency check:

- If a controller receives a *link state* message from n_i indicating that n_i received a *take snapshot* message from n_j, then it should have received a *link state* message from n_j as well.
- Further, n_j should report that it received a *take snapshot* from n_i.
- Finally, the *nssn*(n_j) reported by n_i should be the same as the *ssn* value reported by n_j.

If these consistency checks fail or if the controller fails to receive a *link state* message from a node n_j that was included in some other node n_i's *link state* message within a given timeout, then the controller will initiate a new partition-wide snapshot operation by sending a *take snapshot* message to itself. This operation could be triggered by any of the controllers in the system. On the other hand, if the local consistency check succeeds, each controller has its local report regarding the topology of its network partition and the controllers contained inside of the partition. A controller then attempts to achieve whole quorum consensus across all of the controllers contained in the partition regarding the *link state* messages received from the nodes. This whole quorum consensus succeeds only if all of the controllers had received matching *link state* updates from the nodes. If whole quorum consensus cannot be reached, then any one of the controllers can initiate a new partition-wide snapshot to obtain more up-to-date and complete information from all of the nodes.

Finally, when controllers install new rules on switches, they send along with the update the most recent *ssn* obtained from the switch. If the *ssn* value is stale, the switch would disregard the update as it is likely based on stale network topology information. Rule installation happens only after the controllers have obtained the results of a partition-wide snapshot and ensured that this collected information is consistent across all controllers in the partition.

Observations: While a formal correctness and liveness analysis of the design is beyond the scope of this paper, we now outline a few observations regarding the design. Concurrent network events can trigger snapshots from different nodes at about the same time. The snapshot operation can gracefully handle this case as it is based on the Chandy-Lamport snapshot algorithm, which doesn't require a designated node to initiate snapshots. Concurrent changes in topology are also dealt with consistently by different controllers. Consider again the example depicted in Figure 2. The use of the snapshot mechanism ensures that the link failure and the link recovery events are either considered as part of the same snapshot or as part of two different snapshots that are ordered by the *ssn* values associated with the *link state* messages. If they are part of the same snapshot or if the link recovery event is captured in an earlier snapshot, then there is no splintering of the controller group. Otherwise, the link failure is handled

independently by $\{C1, C2, C3\}$ and $\{C4, C5\}$ and then the controller groups are merged when the link recovery event is identified as part of the subsequent snapshot. Additional node failures might prevent controllers from receiving the flooded *link state* updates or from achieving partition-wide consensus. In such cases, liveness but not safety is compromised. That is, a subsequent snapshot operation will result in more up-to-date information that is reliably delivered to the controllers. The liveness guarantee that is provided by the design is that controllers will be able to control switches in their partitions as long as there is a quiescent period in which no further node failures occur, and this is similar to the liveness guarantees associated with distributed protocols such as Paxos or BFT. Note however that controllers are able to eventually perform network management operations even when partitions occur (as in Figure 1), and this is possible only because the distributed management protocols are co-designed with the network layer.

4. RELATED WORK

Prior works have examined specific facets of the availability issues imposed by SDN. In particular, recent studies have considered how to leverage redundancy to improve the reliability of controller-switch communication [19]. Along the same lines, [9] focuses on availability issues due to flaky switch-side functionality, for example switches stalling on state updates, or crashing suddenly, and advocates dealing with such issues at the SDN application layer (e.g., by provisioning multiple node-disjoint paths). Others have considered ensuring availability of network state data [12, 21], with one popular idea being to use reliable distributed storage. However, none of these attempt to address the intrinsic availability issues afflicting SDN designs that we focus on.

In [17], the authors consider the inter-play between network policies and the ability of SDNs to offer partition tolerance. They show that arbitrary policies, e.g., isolation policies, cannot be implemented correctly on arbitrary topologies when the network is partitioned. In contrast, we show that even simple policies – i.e., reachability among network nodes – cannot be realized within current SDN designs; we offer foundational prescriptions on how to overcome this.

As indicated earlier, we draw inspiration from decades of rich distributed systems research. A key idea we employ - distributed snapshots - has been used in many distributed systems. Many of the follow-on works on making snapshots efficient [16] and scalable [5] are relevant to our work. Finally, our reliable flooding approach is inspired by OSPF's link-state flooding [1].

5. DISCUSSION AND OPEN ISSUES

We advocated using reliable flooding as one of the mechanisms to ensure controllers can always communicate with switches and amongst each other as long as a path is available. Reliable flooding can be expensive, and developing techniques to improve its efficiency and mitigate its impact on the network is a subject of future research. In practice, it makes most sense to employ reliable flooding in conjunction with traditional unicast, as a fall back when unicast cannot offer the desired reachability.

Our description of the protocol and arguments for the properties it offers have largely been informal. Formally proving correctness of our protocol is left for future work. A related issue is that the mechanisms we proposed appear to be *sufficient* to ensure availability in the face of partitions. However, we are as yet unsure if they are all *necessary*. This is an important issue because the mechanisms have a direct bearing on the complexity of switch-side implementation. It may be possible to just rely on a subset of mechanisms, e.g, just distributed snapshots and not whole quorum consensus, but we are yet to establish this formally.

Our protocol assumed that the network is implementing simple end-to-end routing. In practice, the network may wish to implement a different function, e.g., monitoring, isolation, traffic steering (i.e., middlebox traversal), or traffic engineering. Each function places a different requirement on the nature of consensus that the controllers must arrive at. For example, routing requires consensus over actual topology, but filtering implemented at network end-points requires controllers simply to agree on which partition an end-host belongs to. Thus the function in question may affect the mechanisms needed to ensure high availability; these could be simpler or more complex than what we proposed.

We must also consider how our protocol interacts with various consistent update schemes [18, 15]. Consider the protocol described in §3.3, where we stated that switches accept a state update from a controller only if the sequence number included therein matches the current local sequence number. It is quite possible that among multiple switches on a path, some may accept an update corresponding to the path, whereas others reject it (as they initiated new snapshots that are as yet unaccounted for at the controllers). The simplest way to accommodate this is to enhance the update mechanism to prevent the new path from taking effect until all switches have accepted an update and to use state version numbers (similar to [18]). However, this could inflate the latency for a path update to execute fully. Thus, an interesting avenue for research here is the design of faster staged update schemes satisfying global properties such as congestion- or drop-freedom [15].

In essence, our paper does not provide a final answer, but rather it forms the basis for a rich discussion of various issues surrounding availability in SDN.

Acknowledgments We thank the HotNets reviewers and our shepherd, Marco Canini, for their feedback. This research was supported by NSF grants CNS-0963754, CNS-1040757, CNS-1302041, CNS-1314363, and CNS-1318396.

6. REFERENCES

[1] OSPF Version 2: The Flooding Procedure. Request for Comments 1583, Internet Engineering Task Force.

[2] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.

[3] E. Brewer. Towards robust distributed systems. Invited talk at Priniples of Distributed Computing, 2000.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1), Feb. 1985.

[5] R. Garg, V. K. Garg, and Y. Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, 2006.

[6] 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.

[7] 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.

[8] X. Jin, L. Li, L. Vanbever, and J. Rexford. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *CoNEXT*, 2013.

[9] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, J. Rexford, R. Wattenhofer, and M. Zhang. Dionysus: Dynamic scheduling of network updates. In *SIGCOMM*, 2014.

[10] K. Kingsbury and P. Bailis. The network is reliable. http://aphyr.com/posts/288-the-network-is-reliable.

[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] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed internet routing convergence. In *SIGCOMM*, 2000.

[14] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, Dec. 2001.

[15] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. In *HotNets*, 2013.

[16] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *J. Parallel Distrib. Comput.*, 18(4), Aug. 1993.

[17] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker. Cap for networks. In *HotSDN*, 2013.

[18] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[19] F. Ros and P. Ruiz. Five nines of southbound reliability in software-defined networks. In *HotSDN*, 2014.

[20] A. Sahoo, K. Kant, and P. Mohapatra. Bgp convergence delay after multiple simultaneous router failures: Characterization and solutions. *Comput. Commun.*, 32(7-10), May 2009.

[21] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A network-state management service. In *SIGCOMM*, 2014.