When Raft Meets SDN: How to Elect a Leader and Reach Consensus in an Unruly Network

Yang Zhang, Eman Ramadan, Hesham Mekky, Zhi-Li Zhang
University of Minnesota, Twin Cities
Minneapolis, Minnesota
yazhang,eman,hesham,zhzhang@cs.umn.edu

ABSTRACT

In SDN, the logically centralized control plane ("network OS") are often realized via multiple SDN controllers for scalability and reliability. ONOS is such an example, where it employs *Raft* – a new consensus protocol developed recently – for state replication and consistency among the distributed SDN controllers. The reliance of network OS on consensus protocols to maintain *consistent* network state introduces an intricate *inter-dependency* between the network OS and the network under its control, thereby creating new kinds of fault scenarios or instabilities. In this paper, we use Raft to illustrate the problems that this inter-dependency may introduce in the design of distributed SDN controllers and discuss possible solutions to circumvent these issues.

CCS CONCEPTS

• **Networks** → **Network control algorithms**; *Network protocol design*; *Routing protocols*;

KEYWORDS

Consensus, Raft Algorithm, SDN, Resilient Routing

1 INTRODUCTION

Software-defined networking (SDN) simplifies network devices by moving control plane functions to a logically centralized control plane; therefore data plane devices become simple programmable forwarding elements. SDN controllers use OpenFlow APIs [16] to set up forwarding rules and collect statistics at the data plane, which enables controller software and data plane hardware to evolve independently. Under SDN, physical connectivity between two end points do not guarantee they can communicate with each other – the underlying (logical) communication graph depends on the network policies reflected by the flow entries installed by the controller. For scalability and reliability, the logically centralized control plane ("network OS") is often realized via multiple SDN controllers (see Figure 1), forming a distributed system. Open Network Operating System (ONOS) [2] and OpenDayLight (ODL) [17] are two such Network OS examples supporting multiple SDN controllers for high availability.

In distributed network OS such as ONOS and ODL, the replicated controllers rely on conventional distributed system mechanisms such as consensus protocols for state replication and consistency. Paxos [14, 15] is a widely used distributed consensus protocol in production software [4, 5, 9, 13] to ensure liveness and safety. Unfortunately, Paxos is very difficult to understand and implement in practical systems [20]. Raft [20] attempts to address these complexities by decomposing the consensus problem into relatively independent sub-problems: leader election, log replication, and safety. It implements a more "easy-to-understand" consensus protocol that manages a replicated log to provide a building block for building practical distributed systems. Both ONOS and ODL use certain implementations of Raft to ensure consistency among replicated network states. For example, ONOS maintains a global network view to SDN control programs that is logically centralized, but physically distributed among multiple controllers. It employs Raft to manage the switch-to-controller mastership and to provide distributed primitives to control programs such as `ConsistentMap`, which guarantees strong consistency for a key-value store.

The reliance of distributed network OS on consensus protocols to maintain *consistent* network state introduces an intricate *inter-dependency* between the network OS (as a distributed system) and the network it attempts to control. This inter-dependency may create new kinds of fault scenarios or instabilities that have neither been addressed in distributed systems nor in networking. In particular, it may severely affect the correct or efficient operations of consensus protocols, as will be expounded in this paper. The key issue lies in the fact that the design of fault-tolerant distributed system mechanisms such as consensus algorithms typically focus on

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APNET '17, August 3–4, 2017, Hong Kong, China
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5244-4/17/08…$15.00
https://doi.org/10.1145/3106989.3106999

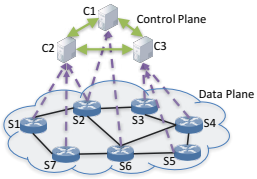


Figure 1: SDN Control Plane Setup.

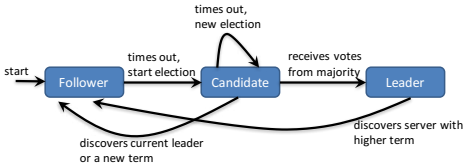


Figure 2: Raft States.

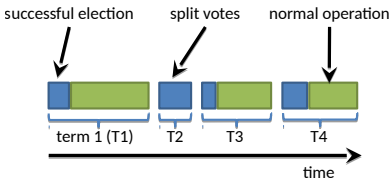


Figure 3: Raft Terms.

server failures alone, while assuming the underlying network will handle connectivity issues on its own. For example, the design of Paxos or Raft assumes that the network may arbitrarily delay or drop messages; however, *as long as the network is not partitioned*, messages from one end point will *eventually* be delivered to another end point. Such assumptions about the network hold true in classical IP networks, where distributed routing algorithms running on routers cooperate with each other to establish new paths after failures. SDN now creates *cyclic dependencies* among *control network connectivity*, *consensus protocols*, and *control logic managing the network*, where the control logic managing the network is built on top of a distributed system (e.g., ONOS) which relies on consensus protocols for consistency and control network connectivity for communication, whereas the network data plane (and control network) hinges on this distributed system to set up rules to control and enforce "who can talk to whom" among networking elements. Consequently, new failure scenarios can arise in SDN.

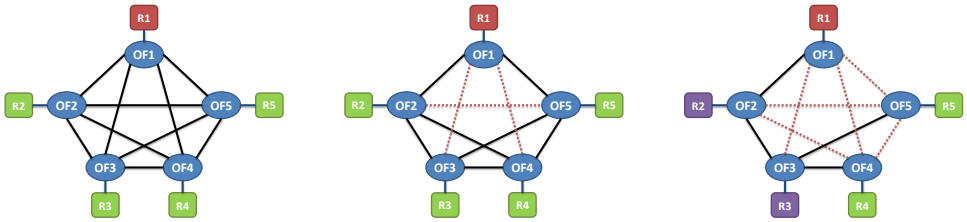
In this paper, we first provide a brief overview of Raft in Section 2. We then illustrate a few network failure scenarios that may arise when applying Raft to a distributed SDN control cluster (see Section 3). We demonstrate how these failure scenarios can severely affect the *correct* or *efficient* operations of Raft: in the best case they significantly reduce the available "normal" operation time of Raft; and in the worst case, they render Raft unable to reach consensus by failing to elect a consistent leader. It is worth noting that the problems highlighted here are different from those addressed by, e.g., the celebrated CAP Theorem in distributed systems [3, 7], which establishes impossibility results regarding simultaneously ensuring availability and (strong) consistency under network partitions. This result has been recently generalized in [21] to SDN networks in terms of impossibility results regarding *ensuring network policy consistency under network partitions*. In contrast, we argue that thanks to the inter-dependency between the network OS as a distributed system and the network it attempts to control, *SDN introduces new network failure scenarios that are not explicitly handled by existing consensus algorithms such as Raft, thereby severely affecting their correct or efficient operations.* In Section 4, we discuss possible "fixes" to circumvent these problems. In particular, we argue that in order to fundamentally break this inter-dependency, it is crucial to equip the SDN control network with a resilient routing mechanism such as *PrOG* [24] that guarantees connectivity among (non-partitioned) SDN controllers under arbitrary failures. Using a vanilla Raft implementation [19] and *PrOG*, we provide preliminary evaluation results in Section 5. Finally, we discuss the related work in Section 6 and conclude the paper in Section 7.

2 RAFT OVERVIEW

Raft [20] is a consensus algorithm designed as an alternative to (multi-)Paxos [14, 15], to be easier to understand, with formal proof of its correctness. Raft is as efficient as Paxos, but its structure is different. It provides a better foundation for building practical systems. Raft separates consensus into the following subproblems: (1) Leader election: when the current leader fails; (2) Log replication: the leader accepts log entries from clients and replicates them, forcing other logs to be consistent with its own log; and (3) Safety: a few restrictions on leader election are enforced to ensure safety, i.e., if any member applied a particular command to its state machine, then no other member may apply a different command for the same entry. Raft starts by electing a *leader*, then it gives the leader full responsibility for managing the replicated log. When it is safe to apply log entries to state machines, the leader instructs servers to apply them to their local state machines.

Raft States. Raft clusters typically contain odd number of members. As in Figure 2, a server can be in one of three states: follower, candidate, or leader. Each cluster has one leader, and others are just followers passively receiving RPCs from the leader or candidates. **Raft Terms.** As shown in Figure 3, time is divided in terms of arbitrary length. Terms are monotonically increasing integers, where each term begins with an election. If a candidate wins an election, it serves as the leader for the rest of the term. Terms allow Raft servers to detect obsolete information such as stale leaders. Current terms are exchanged whenever servers communicate. When a leader or a candidate learns that its current term is out of date (i.e., there exists a higher term number), then it immediately reverts to the follower state. Servers reject vote requests and replicated log entry with a stale term number from the leader.

Raft Leader Election. A leader in Raft sends periodical heartbeats to all followers. If a follower receives no heartbeat messages over a predefined period of time (*election timeout*), it assumes there is no leader and starts a new election. It increments its current term, votes for itself, and moves to candidate



(a) Control Cluster under Normal Operations. (b) Oscillating Leadership. (c) No Leader Exists.

Figure 4: Motivating Examples.

state. Then, it sends *request-to-vote* RPCs to other servers, for three possible outcomes. *Win Election:* if it receives votes from a majority, it sends heartbeats to all servers to prevent new elections and establish its authority for its term. *Lose Election:* While waiting for votes, the candidate server may receive a heartbeat message from another server claiming to be the leader. If the received term number is at least as large as the candidate's current term, then it surrenders as a follower. *Split Votes:* If no candidate server receives majority of votes, then one of the servers will timeout for not receiving heartbeat messages from any leader and start a new election. Raft uses randomized timeouts to ensure split votes is a rare event. Raft enforces restrictions on elected leaders e.g., a server votes to a candidate if its term is higher, and the candidate's log is at least as up-to-date as its own, otherwise the server rejects the vote request. Therefore, receiving a majority of votes means that the new leader log contains all committed entries.

3 RAFT MEETS SDN

In distributed network OS such as ONOS and ODL, multiple controllers must maintain a *consistent* global view of the network. This is achieved by employing a consensus protocol such as Raft to ensure consistency among the replicated network states maintained by each controller. The connectivity among these controllers can be provided either via a dedicated *control* network ("out-of-band") or via "in-band" (virtual) control network through the data plane under their control [2, 12, 17, 21]. In either case, we refer to the (dedicated or virtual) network connecting the controllers as the *control network*. We assume that it consists of OpenFlow switches with flow rules installed by the *same* controller cluster to which it provides connectivity.

Figure 4(a) shows an SDN control cluster with 5 controllers in a full-meshed control network with five OF switches. The controllers run Raft to ensure consistency among the replicated (critical) network states they maintain. We will use a few toy (contrived) examples to illustrate the new failure scenarios that may arise when applying Raft to a distributed SDN control cluster for consensus. In the following scenarios, we assume that initially R1 is the leader (in red) and green indicates that the logs of a member (controller) is up-to-date. The current term is T1 as seen by all members. Up links are black, while down link are red, and the cluster is not partitioned.

Scenario 1: Oscillating Leaders. Figure 4(b) shows a Raft cluster, where the links (R1, R3), (R1, R4), and (R2, R5) fail. Either R3 or R4 will time out for not receiving heartbeats. Assuming R3 times out first, then it will increment its term number to T2, vote for itself, and request votes from R2, R4 and R5. After getting the votes, R3 will be the leader, and the current term vector becomes (R1=1,R2=2,R3=2,R4=2,R5=2). After that, R1 will step down after learning about the higher term T2 from R2 or R5 through heartbeat messages, and update its term to T2. R1 and R3 cannot communicate, because the link (R1, R3) is down. R1 will time out, and increase its term number. Thus, it can get R2, and R5 votes to become a leader for term T3, and force R3 to step down and snatches the leadership, because its term number is larger than T2. The term vector becomes (R1=3,R2=3,R3=2,R4=2,R5=3). Then, we are back to the initial settings, and the whole scenario can be repeated.

Assuming after R3 became the leader again for T4, it receives some requests and updates the logs for all nodes except R1. Thus, when R1 tries to become the leader for T5, it will not receive votes from R2 and R5, because their logs are more recent than R1's log, and the current term vector will be (R1=5,R2=5,R3=4,R4=4,R5=5). Thus, R3 will step down. Currently, there is no leader, so whoever times out first can be the leader except R1. If R3 or R4 time out first, what we just discussed will be repeated.

Assuming R2 times out first, increments its term to T6, and becomes a leader. In this case, R5 will not receive heartbeats ((R2, R5) is down), and try to become a leader. Thus, we can notice the leadership will be oscillating between either (R2, R5) or (R1, R3, and R4). In the worst case, the cluster can be dead, since clients sending requests to the current leader will be redirected to the new leader. By the time they contact the new leader, it will change again.

Condition. Up-to-date nodes have a quorum, but they cannot communicate with each other.

Scenario 2: No Leader Exists. Figure 4(c) shows a Raft cluster, where the leader R1 successfully updated R4 and R5 logs, but failed to update R2 and R3 logs (in purple), due to

link failure, packet drop or network congestion. Assuming R2 times out first for not receiving heartbeats, it will increase its term to T2, forcing R1 to step down. In this case, even though R2 and R3 have a quorum (R2 connected to R1 and R3; R3 connected to R2, R4, and R5), they will not be able to become a leader, because their logs are not up-to-date. R1, R4, and R5 cannot be the leader also, because they do not have a quorum. Therefore, the cluster is not live anymore, even though the underlying network is *not partitioned*.

Condition. Nodes have a quorum, but they have obsolete logs, and nodes having up-to-date logs, do not have a quorum.

In summary, consensus distributed systems are designed agnostic from underlying network, with the assumption of all-to-all communication between network entities, as long as the underlying network is not partitioned. In SDN, these consensus systems are used to manage the underlying network. Therefore, making progress depends on updating the OpenFlow rules, which depends on the connectivity between servers (chicken and egg situation.) Hence, a novel approach needs to be designed to solve this issue in SDN networks.

4 POSSIBLE SOLUTIONS

In this section, we illustrate the solution requirements and discuss the limitations of some possible solutions. Then, we briefly introduce a prospective solution for this problem.

Solution Requirements. The problem with Raft, and distributed protocols in general, is the assumption of *all-to-all connectivity among cluster members as long as the network is not partitioned*. In SDN, switches in the data plane forward traffic based on decisions made by the control plane, which uses a consensus protocol like Raft to ensure the state is replicated correctly. Upon failures, controllers may lose connection to each other and to switches in the network. Moreover, in SDN failures can be *physical*, i.e., physical link/node failures, or *logical*, e.g., two servers are physically connected, but there are no corresponding rules installed, thus preventing them from communicating. Therefore, with unpredictable network failures, the solution should be resilient against arbitrary link/node failures and ensure that controllers retain reachability among them whenever the underlying network is physically connected.

4.1 Gossiping

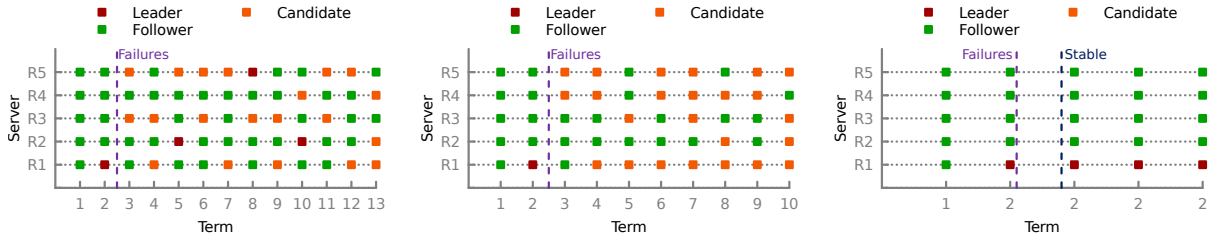
One common distributed systems solution to restore connectivity is via gossiping. Thus, to achieve connectivity, Raft can be extended to enable servers to gossip and forward heartbeats through other servers to overcome the failures affecting their direct communication. This solution may help avoid the scenarios mentioned in Section 3 by probabilistically forwarding Raft messages (heartbeat, replicated logs, ... etc.) through some other servers, which may have a path to the original message's destination. If the number of these servers are large enough, there is a high probability one of them is able to forward the message to its destination. As long as followers receive these messages from the leader, the cluster can still be live, as they will renew their heartbeatTime, and avoid starting a new election process.

However, the problem with such a solution is that it may work in some cases only, as it depends on the underlying network connectivity and which servers are selected to forward messages. As it assumes uncorrelated link failures and might be affected by new link failures and Raft timeouts, it is not guaranteed to work in all cases and scenarios. Alternatively, flooding can be used instead to ensure message delivery. However, it may lead to network congestion, which may not only increase packet delay and affect data plane flows, but create additional packet losses/network failures. We believe that *built-in resiliency* in the (control network/data plane) is essential for high-availability of SDN controllers. The control network (data plane) should not rely on the control plane to recover from failures, and should have pre-installed rules in switches for automatic failure recoveries. Fortunately, this can be achieved via a new routing paradigm proposed in [24], known as "routing via preorders", which provides adaptive resilient routing to ensure all-to-all communications among servers regardless of network failures, as long as the underlying network is not partitioned. We briefly discuss it as a prospective solution next.

4.2 Routing via Preorders

Main Ideas. Considering a network $G = (V, E)$ representing the control plane, and a flow F from a source s to a destination d, where s, d are SDN control cluster servers. A preorder is defined on a node set V' where $s, d \in V'$, and $V' \subseteq V$. This preorder specifies the relation between any two nodes u, v, where $v \rightarrow u$ means v is a child of u, and $v \leftrightarrow u$ means u, v are siblings. The result is a directed connected (sub)graph $G' = (V', E')$, where each edge in E' ($\subseteq E$) is oriented either uni-directional or bi-directional, and G' is called a preordered graph (*PrOG* in short.) At each node, packets can be forwarded to any of its parents or siblings, (i.e., follow any directed path from s to d without enumerating all paths between them.) Bi-directional links are only activated along one direction upon failures. Through its construction, *PrOG* includes all possible paths between s, d. Therefore, it ensures controllers retain connectivity, as long as the underlying graph is not partitioned, allowing Raft participants to exchange data even if direct communication links are unavailable. *PrOGs* are constructed for each source-destination pair, using a modified version of breadth-first search.

Upon failures, the affected parts of *PrOG* are deactivated, and traffic is routed along the remaining part, where each node



(a) Raft: leadership keeps oscillating between servers (unstable).

(b) Raft: no viable leader (liveness lost).

(c) PrOG: leadership is stable.

Figure 5: Results for simulating the motivation examples in Figure 4 using vanilla Raft and *PrOG*-assisted Raft.

uses its alternative outgoing links if they exist. Upon link/node recovery, relevant links are activated. The original *PrOG* is restored, when all links/nodes recover. *PrOG* also provides an additional feature, where the constructed graph can have a bounded threshold for the cost of the included paths from s to d (see [24] for more details.) Thus, a threshold can be defined for normal operations, and a relaxed threshold to be used upon failures. Therefore, Raft members can communicate within a predefined threshold even when there are failures. *PrOGs* are then converted to OpenFlow rules pre-installed in switches. Switches are provided with a small functionality required to maintain and update an internal state. For example, each switch maintains the state of its outgoing links whether they are active or not, and sends activation/deactivation messages upon link recovery/failure.

Summary. "Routing via Preorders" uses local data plane operations to achieve resiliency under arbitrary link/node failures, without any involvement from the control plane to recompute routes, since they are pre-computed and pre-installed. Thus, it avoids the cyclic dependency between control network connectivity and management, where controllers need to setup rules to recover from failures, but cannot reach switches because of failures. Therefore, it provides all-to-all communications among cluster members for a stable Raft leadership and enables the cluster to progress regardless of failures. Finally, it is a general solution as it does not require any modifications to Raft, and can be used by other distributed protocols as well. The correctness and overheads of *PrOG* are discussed [25]. We will expand its design for *control network resiliency* in a future paper.

5 PRELIMINARY RESULTS

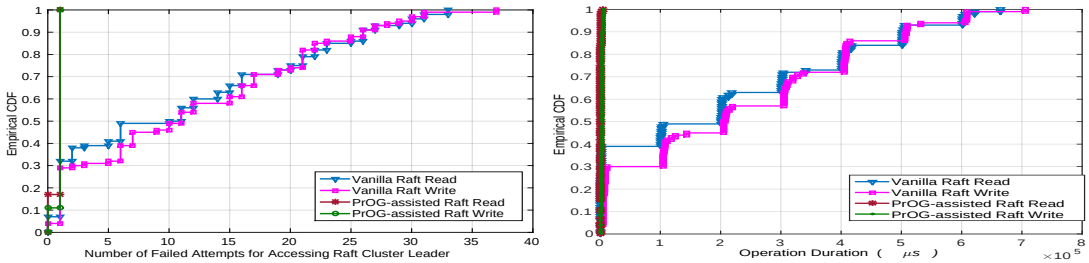
In this section, we compare the results of vanilla Raft and *PrOG*-assisted Raft to show that *PrOG* resolves the issues presented in Section 3 and enables a more resilient and robust SDN distributed control platform.

Experiment Setup. Standard Raft C++ implementation in LogCabin [19] is used in the experiments. LogCabin is a distributed storage system which supports all major Raft features like log replication, membership changes, etc.. The basic setting of our experiment is to create six Docker containers [8] (Ubuntu 14.04) in which five containers serve together as a Raft cluster, while the other one serves as a client which reads logs from the cluster or writes logs to the cluster. Topology is setup as illustrated in Figure 4 and Open vSwitch [23] (OVS) instances are used as software switches. Data written to the Raft cluster is replicated across all cluster members (i.e., containers) by a Raft leader. We disable vote withhold [22] and simulate the two failure scenarios described in Section 3.

Results. To demonstrate the effectiveness of *PrOG*, we mainly present three types of results: 1) leadership shifting diagram. 2) statistics of clients' failed attempts when accessing cluster leader; 3) statistics of cluster availability time. Leadership shifting diagram is a straight-forward way of observing the states of Raft servers at each term before and after failure occurs. In Figure 5, the x-axis shows the current term number, and the y-axis shows the raft server. Different colors shows the different states: leader (red), follower (green), and candidate (orange), e.g., Figure 5(a) R1 starts as follower in term 1, then leader for term 2, then follower again for term 3, and so on. Note that for leaders, we don't show the transition to candidate since it is implied by successful transition to leader.

Figure 5(a) shows the results for Figure 4(b). It shows the leadership keeps oscillating and not stable. In our experiments, we noticed that this blocks the client from being able to read or write to the storage for a large number of trials (detailed later.) Figure 5(b) shows the results for Figure 4(c), in which no viable leader exists, since servers cannot *directly* communicate with each other, therefore the client cannot read or write to the storage anymore. Figure 5(c) shows that our extension resolves the issues in Figure 5(a), because servers can indirectly communicate with each other through other servers, therefore the client can read from and write to the storage quickly. The system is stable from term 2.

When a client performs read/write operations on the Raft cluster, it will randomly select one server in the cluster. If the selected server is not the current leader, it replies with the current leader's IP. However, if there is no leader at the moment, no suggestion is returned and the client will randomly select



(a) Client suffers much more failed attempts for accessing cluster leader in vanilla Raft.

(b) Latency of a request operation increases accordingly.

Figure 6: Vanilla Raft vs. *PrOG*-assisted Raft.

another server to contact. The client issues 100 read/write requests with 1-second interval under the failure scenario in Figure 4(b) for each round of experiments, and then we count the number of failed attempts before the client successfully reaches the current cluster leader as shown in Figure 6(a). Moreover, we also measure the duration of each read/write request as shown in Figure 6(b). The results demonstrate that *PrOG*-assisted Raft is more robust to network failures. In terms of why most durations in Vanilla Raft are close to $N \times 10^5 \mu s$ (N is positive integer), it is because the client is set by default in LogCabin to wait $10^5 \mu s$ before trying another server's IP.

We also carefully analyzed the log of the five Raft servers to sum up availability time of the whole Raft system under the oscillating leadership scenario. The availability time in our experiments is defined as the total time period in which a leader is available, because the distributed system can only serve clients when a leader exists. We perform five rounds of three-minute experiments and calculate system availability time. The average availability time is 75.67s (42.04% out of 180s) and 248 times of leadership shifting happen, even though the network is not partitioned.

6 RELATED WORK

SDN availability. SDN controllers may take advantage of established techniques from the distributed systems literature. For example, a control cluster with multiple controllers could use a distributed storage system for durable state replication. Distributed SDN controller designs rely on consensus algorithm such as Paxos used by ONIX [12] and Raft used by ONOS [2], and even stronger consistency guarantees are required by Ravana [10]. LegoSDN [6] focuses on controller crash failures caused by software bugs. Statesman [28] demonstrates incrementally mitigating up-to-date state to switch with obsolete state when a control master fails. Liron [26] proposes a model for designing distributed control plane which maintains connectivity between a distributed control plane and the data plane. In comparison, we study the availability issues in consensus algorithm, and propose *PrOG* to enhance Raft. HyperFlow [11] utilizes publish-subscribe messaging paradigm among controller instances to replicate network events, and local state is built solely by controller application based on subscribed event. We assume that Network OS employs consensus algorithms such as Raft to maintain the correctness of control logic managing the network, and enhaces it with a "self-healing" resilient control network.

Robust message exchange. Robust message exchange in SDN is fundamental for controller availability. Webb *et al.* [29] propose a way of deploying tightly-coupled distributed system in wide area in a scalable way. It preserves efficient pairwise communication through an overlay network with gossip-based communication protocol. Schiff *et al.* [27] propose a synchronization framework for control planes based on atomic transactions, implemented in-band, on the data-plane switches. Akella *et al.* [1] architectures SDN for robustness to faults. They tackle the problem of in-band network availability and synthesize various distributed system ideas like flooding, global snapshots, etc. Muqaddas *et al.* [18] quantify the traffic exchanged amongst controller running Raft and summarize that the inter-controller traffic scales with the network size. *PrOG* provides a general robust and resilient message exchange mechanisms for both in-band and out-of-band control channels. network size.

7 CONCLUSION

SDN controllers use distributed consensus protocols like Raft to manage the network state and provide a highly available cluster to the underlying networking elements. Therefore, SDN controller liveness depends on all-to-all message delivery between cluster servers. In this paper, we use Raft to illustrate the problems which may be induced by this inter-dependency in the design of distributed SDN controllers. We also discuss possible solutions to circumvent these issues. Our preliminary results show the effectiveness of *PrOG* in improving the availability of leadership in Raft used by critical applications like SDN controller clusters.

Acknowledgement. This research was supported in part by DTRA grant HDTRA1-14-1-0040, DoD ARO MURI Award W911NF-12-1-0385 and NSF grants CNS-1618339, CNS-1618339 and CNS-1617729.

REFERENCES

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

[2] Pankaj Berde et al. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proc. HotSDN*.

[3] Eric A. Brewer. 2000. Towards Robust Distributed Systems.

[4] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proc. OSDI*.

[5] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: an Engineering Perspective. In *Proc. PODC*.

[6] Balakrishnan Chandrasekaran and Theophilus Benson. 2014. Tolerating SDN Application Failures with LegoSDN. In *Proc. HotNets*.

[7] Seth Gilbert et al. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*.

[8] Docker Inc. 2016. Docker Containerization Platform. https://www.docker.com/.

[9] Sushant Jain et al. 2013. B4: Experience with a Globally-deployed Software Defined WAN. *Proc. SIGCOMM CCR*.

[10] Naga Katta et al. 2015. Ravana: Controller Fault-tolerance in Software-defined Networking. In *Proc. SOSR*.

[11] Takayuki Dan Kimura. 1993. Hyperflow: A Uniform Visual Language for Different Levels of Programming. In *Proc. CSC*.

[12] Teemu Koponen et al. 2010. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. OSDI*.

[13] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a Decentralized Structured Storage System. *SIGOPS Operating Systems Review*.

[14] Leslie Lamport. 1998. The Part-time Parliament. *ACM Transactions on Computer Systems*.

[15] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News*.

[16] Nick McKeown et al. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*.

[17] Jan Medved et al. 2014. Opendaylight: Towards a Model-driven SDN Controller Architecture. In *Proc. WoWMoM*.

[18] Abubakar Siddique Muqaddas et al. 2016. Inter-controller Traffic in ONOS Clusters for SDN Networks.

[19] Diego Ongaro. 2016. LogCabin: A Distributed Storage using Raft. https://github.com/logcabin.

[20] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proc. USENIX ATC*.

[21] Aurojit Panda et al. 2013. CAP for Networks.

[22] Jehan-Francois Paris et al. 2015. Pirogue, a Lighter Dynamic Version of the Raft Distributed Consensus Algorithm. In *Proc. IPCCC*.

[23] Ben Pfaff et al. 2015. The Design and Implementation of Open vSwitch. In *Proc. NSDI*.

[24] Eman Ramadan, Hesham Mekky, Braulio Dumba, and Zhi-Li Zhang. 2016. Adaptive Resilient Routing via Preorders in SDN. In *Proc. DCC*.

[25] Eman Ramadan, Hesham Mekky, Cheng Jin, Braulio Dumba, and Zhi-Li Zhang. 2017. Provably Resilient Network Fabric with Bounded Latency. In *Under Submission*.

[26] L. Schiff, S. Schmid, and M. Canini. 2016. Ground Control to Major Faults: Towards a Fault Tolerant and Adaptive SDN Control Network. In *Proc. DSN-W on IFIP*.

[27] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. 2016. In-Band Synchronization for Distributed SDN Control Planes. *SIGCOMM CCR*.

[28] Peng Sun et al. 2014. A Network-state Management Service. In *Proc. SIGCOMM*.

[29] Kevin C. Webb et al. 2013. Scalable Coordination of a Tightly-coupled Service in the Wide Area. In *Proc. SOSP TRIOS*.