Canaries in the Network

Danyang Zhuo, Qiao Zhang, Xin Yang, Vincent Liu[†]
University of Washington, University of Pennsylvania[†]
{danyangz, qiao, yx1992, vincent}@cs.washington.edu

ABSTRACT

Updating a large network deployment is a dangerous process. Regardless of whether the operation is a switch BGP configuration change or a network-wide SDN controller upgrade, misconfigurations and bugs can potentially cause network outages and downtime for critical cloud services. Many cloud applications have adopted a useful strategy that networks have tried to emulate: phased rollouts, in which a small fraction of users are redirected to the updated version while most users continue unassailed. Unfortunately, this analogy is fundamentally flawed. This paper explores the limits of phased rollouts in networks and shows when and why they can fail. We also go on to propose two preliminary designs for approximating the benefits of phased rollouts. Although preliminary, we argue that our designs can achieve a useful level of isolation between a 'known correct' and 'new' control plane.

1. INTRODUCTION

In any large, complex system, nothing is riskier than change. Change in the form of new hardware, software, and configurations can introduce untested code or trigger latent bugs. This was true three decades ago when Jim Gray published one of the first papers with failure statistics from a production system[1], and by many accounts, it is even more true of the large-scale networks found at the center of today's cloud service providers [4, 9, 12].

Part of what makes this problem difficult is the fact that these networks, by their very nature, are connected—a change in one part of the network can necessitate changes elsewhere. For instance, when a data center operator adds/removes an IP prefix from a rack of servers, the network may need to distribute new BGP updates to the rest of the network to ensure that the new prefix is routable. If she mistakenly added a prefix that is too large, a great deal of traffic can be incorrectly diverted to the rack. If the routing updates trigged latent bugs in the BGP implementation, the entire network can go down.

More generally, control plane bugs and misconfigurations can be triggered by either operator intervention or the arrival of previously-untested inputs (e.g., a routing update). Errors can lead to bad switch configurations that either drop or redirect packets. Redirected packets can, in turn, negatively impact other switches effectively performing a Denial-of-Service (DoS) attack.

While these types of errors may seem trivial to check for, but the reality is that the size and complexity of these systems make them inevitable [2, 3, 15, 17]. This is despite extensive testing [10, 31] and verification [8, 14, 20, 21, 26] of new software, hardware, and configurations. As we move toward more complex routing policies and logically-centralized control planes like the ones envisioned by Software-Defined Networking (SDN), the problem becomes even more difficult because the SDN platform and control applications have a great deal of control over the network and can be a single point of failure.

Rather than trying to prevent mistakes, we ask a different question: how can we do a better job of rolling out changes safely? Many large-scale applications accomplish a similar objective through *phased rollouts* in which a small fraction of users are redirected to an updated version of the software, and the rest of the population is incrementally added to the set. Using this technique, the updated version is strongly isolated: infrastructure that was not chosen as part of the rollout never interacts with infrastructure that was. Is there an analogue in networks? Such a mechanism would allow operators to roll out new switch configurations or SDN control software to a fraction of the network with a guarantee that the effect is completely isolated from the rest of the network.

We make the observation that, in general, such a mechanism is impossible—the connected nature of networks means that, outside of duplicating the entire deployment, networks cannot have the degree of isolation enjoyed by phased application rollouts. This is true for networks with in-network control planes (e.g., BGP) and is also true for those with logically-

[1]30% of failures were of new hardware/software, 29% were due to system administration (e.g., config changes, maintenance, etc.).

HotNets-XV, November 9–10, 2016, Atlanta, Georgia, USA.
© 2016 ACM. ISBN 978-1-4503-4661-0/16/11...$15.00
DOI: http://dx.doi.org/10.1145/3005745.3005767

centralized control planes (e.g., OpenFlow [28]/ONIX [22]).

Our primary contribution is to identify several different levels of canary isolation and discuss their application to computer networks. As part of this, we introduce two approaches to approximating a phased rollout for data center networks. In both approaches, the underlying objective is to divide the network into a 'known correct' and a 'new' partition such that (1) the old partition's operation does not need to rely on the correct behavior of the new one, and (2) the partitioning can be adjusted dynamically. This type of flexible isolation allows operators to limit the impact of network control plane bugs and misconfigurations. For both, we briefly outline how they can be implemented using readily available hardware features and protocols. We leave a more detailed exposition and analysis to future work.

2. BACKGROUND AND MOTIVATION

We begin our discussion by framing the types of errors that can result from changing a network deployment. This will lead into our discussion of the extensive steps network operators take to harden their networks against such changes and why these efforts cannot provide strong isolation.

At a conceptual level, networks are divided into two logical components:

- The *data plane* is responsible for forwarding packets from each switches' ingress ports to the target egress port. This involves a series of simple packet processing steps and table lookups based on local forwarding state.
- The *control plane* is then responsible for populating the forwarding state using computed routes. For a distributed control plane protocol like BGP, switches do this by disseminating reachability information across the network and iteratively computing the best path. For a centralized control plane protocol like OpenFlow, a logically centralized controller does this by gathering the topology of the network and computing the proper path for each flow.

In practice, errors can and do manifest in both of these components. For example, in the data plane, an unexpected/undertested packet format can cause the processing switch to crash entirely (this type of bug is often used in DoS attacks [1]). In the control plane, problems can result from human error as well as subtle bugs in the underlying switch.

In this paper, we focus on the latter component as the control plane tends to change much more frequently than the data plane. In this context, there are generally two types of events that can trigger network changes. The first is status modifications (e.g., hardware failures/recoveries and congestion), which change the control plane's view of the network. The second is operator intervention, in which an operator manually (or through a tool) changes the control plane software, its configuration, or its view of the network. In both cases, the state and/or configuration of directly affected switches will change. These state/configuration changes may trigger other changes throughout the network through routing updates, etc.

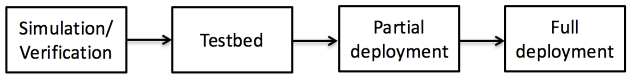

Figure 1: Rollout plan for a control plane change. Operators first simulate and verify the new software/configuration. They then deploy the changes on a separate testbed. If the previous two tests are successful, operators will start to do a partial deployment followed by a full deployment.

Our threat model considers cases where these network changes result in dropped or diverted packets. Diverted packets can potentially congest other parts of the network, effectively performing a resource-exhaustion attack. Note that we only consider drop/diversion errors that operators can easily detect, and we assume they can be found in a reasonable amount of time after deployment.

Operators use a wide variety of techniques to prevent and mitigate problems; however, to make our discussion concrete, we focus on the typical deployment process of data center configuration changes in a large cloud provider; when it comes to control plane changes, these operators are among the most strict due to the speed at which their networks grow and the large monetary penalty of downtime. Generally speaking, these data center networks are structured as a multilevel tree with racks of servers at the leaves of the tree (see Figure 4a). Their control plane disseminates routes in these large deployments through BGP, SDN, or a combination of both.

2.1 Phased Rollouts in a Typical Network

Reconfigurations and updates are inevitable in production networks and can range from adding a new route to updating the SDN control software that manages the entire network. A typical deployment flow is visually depicted in Figure 1.

Simulation and Verification. Before any potential update or configuration change touches any hardware, it can first be checked independently. At a minimum, this involves manual inspection and peer reviews of the resulting configurations/code. Other techniques used at this stage include software emulation of the resulting network and automatic validation of the new configurations to ensure properties like reachability and loop-freeness. While these approaches do prevent many errors, they also have their limitations. In particular, they make assumptions about the underlying operation of the network and therefore cannot catch all subtle errors that appear in practice.

Testbed The next step in checking a potential change to the network is to deploy the change on a separate testbed. Given the cost of a full data center, testbeds generally emulate an exceedingly small subset of the full network—perhaps only a few racks of servers. As such, the testbed cannot fully test a new network change as it does not replicate the same topology and often does not replicate the same workload.

Incremental Deployment The final step is to slowly roll out the change to the production network over the course of several hours to weeks. Operators start by deploying the change to a small subset of the network before pushing it to additional subsets and then to the entire data center. The granularity of deployment sets is typically either a single rack or some subtree of the data center topology. Because of the hierarchical fashion in which data centers are designed, if a problem arises in a single rack or subtree, it can sometimes be contained within that subset.

Problems in this phase are the most problematic as they affect production traffic. They can occur because of latent issues in the incremental rollout code itself [2] or as a result of corner cases that only manifest in a true production environment.

2.2 Why Strong Isolation Is Impossible

The issue with current incremental deployment plans is twofold. First, there are many cases where problems do not manifest until they are deployed to production. Aside from building a duplicate network, this limitation is fundamental. Second, negative effects that do arise during the phased rollout have the potential to spread to the entire network.

Ideally, operators would be able to isolate network changes using two partitions: a 'known-correct' partition and a 'new' partition. The old partition should not, in any way, depend on the correct operation of the new partition so that failures in the latter will not spread to the rest of the network. Practically speaking, this means that the two partitions would not share any control plane state or routing information.

Previous work has suggested that, aside from duplicating the entire network, it is impossible to guarantee useful global properties like connectivity and load balancing without global information [6, 7, 24]. We formalize these claims below.

2.2.1 *Model*

As mentioned above, we model a phased deployment by extending the traditional single data/control plane model of the network to allow for multiple control planes (e.g, a 'known correct' control plane and a 'new' control plane).

Each control plane controls a set of physical devices; sets may overlap with each other or be entirely disjoint. To denote which switches a given control plane controls, we define a function, $control(c)$, that takes a control plane as input and outputs the set of switches that it controls. Said differently, if switch d is controlled by c, then $d \in control(c)$.

Each control plane may also communicate with a set of other control planes to disseminate routing information. The set of control planes with which c communicates is $comm(c)$. These control planes can be buggy or misconfigured. When that occurs, it can arbitrarily and recursively affect any device in $control(c)$ or control plane in $comm(c)$. The error recursion is similar to the model assumed by taint checking [13].

2.2.2 *Impossibility of Isolation*

Ideally, a phased rollout would provide three properties:

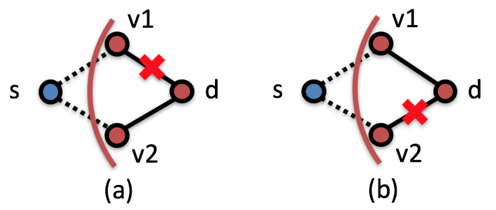

Figure 2: Key subgraph where the three properties do not simultaneously hold. The network is separated into two partitions (blue, red). Dotted black lines denote an arbitrary path. Solid black lines are similar, but must not include v_1 or v_2. The red arc is the partition boundary. (3) *connectivity* does not hold for a flow from s to d because (a) and (b) are indistinguishable from the perspective of the blue partition.

1. *Partitioned control:* There are multiple control planes and all are isolated, i.e., for every control plane c, $comm(c) = \varnothing$ and $control(c) \neq \varnothing$.
2. *Physical isolation:* Switches are controlled by exactly one control plane.
3. *Connectivity:* If there exists a path between two end hosts, there exists a route between them.

The first two properties ensure strong isolation between control planes. Failures in any control plane can affect the switches in $control(c)$, but will not spread any further. The third property ensures that the network fulfills its purpose.

Our proof takes the following approach. We first identify a simple subgraph and partitioning scheme that cannot satisfy all three properties simultaneously. Any network that contains this subgraph will suffer from the same problem. Figure 2 shows the subgraph, which borrows its basic structure from a one used in [7]. In Section 2.2.3, we go one step further and show that this subgraph is relatively common in current data center topologies.[2] We start by defining the operation of a routing algorithm on a partitioned network.

DEFINITION 1. *A routing algorithm, R, is a function that takes (1) a node v, (2) the control plane c that contains v, and (3) the destination d. Its output is an edge on node v. R fails if there exists a path from s to d, but we cannot reach d by recursively calling R starting from s.*

We also need to define the sufficient condition to determine whether a graph contains a sub-structure as in Figure 2. We call a graph "suitable" when it contains the base structure.

DEFINITION 2. *A graph G is a suitable graph if, for some source s and some destination $d \in control(c)$:*

- $s \notin control(c)$ *and* $\exists v_1, v_2 \in control(c)$ *s.t.* $v_1 \neq v_2 \neq d$.
- *s has a path to v_1 and v_2 that does not cross any switch controlled by c. In other words, v_1 and v_2 are ingresses.*

[2]We note that some graphs allow for both strong isolation and connectivity (e.g., a strict, single-rooted tree). However, we show that most topologies found in practice do not have this property.

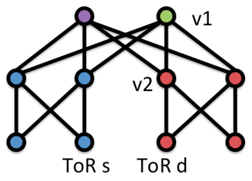

Figure 3: Our key subgraph can be found in any useful partitioning of a Clos network (the most common class of modern data center topology). This diagram shows one example partitioning with colored nodes, and annotates the key subgraph within it.

- *v_1 has a path to d that does not include v_2 and vice versa. Both paths are fully controlled by c.*

We can prove (3) *connectivity* cannot hold on such a graph.

THEOREM 1. *If G is suitable, for all routing algorithms A, we can fail a set of links L, so that A fails on the resulting graph $G' = G - L$.*

PROOF. By contradiction. Because G is suitable, we can find c, s, v_1, v_2, d that match the definition of a suitable graph. Fail all other paths between v_1/v_2 and d except the two required above (i.e., the path from v_1 to d that does not include v_2 and the path from v_2 to d that does not include v_1). s and d are still connected via at least 2 paths.

Consider any algorithm A that can successfully route from s to d. Let P be A's chosen path. Without loss of generality, assume P goes through v_1. Fail the link immediately after v_1 in P. Because the extra failure is not noticed by any node $\notin c$, A will continue to route packets to v_1. Therefore, the route produced by A fails even if an alternative path exists from s to d (the one going through v_2). □

2.2.3 Application to Today's Data Centers

The previous section identifies a simple subgraph whose presence in a network topology makes achieving all three of partitioned control, physical isolation, and connectivity impossible. The existence of this subgraph depends on both the network's topology and how we choose to partition it. What we show in this section is that for one common data center topology (the Clos network), *any* useful partitioning exhibits the subgraph. 'Useful', in this case, implies resilience to two things: (1) control plane errors such that a single control plane bug cannot take out the entire network and (2) hardware failures such that, regardless of the size of the network, two failures cannot cause the failure of the routing protocol. We sketch the proof here and leave a more formal treatment for future work.

Consider Figure 3, which depicts a Clos topology like the ones used in most of today's data centers. The bottom level of switches are the Top-of-Rack (ToR) switches, which serve as servers' gateways into the network. These are connected into small subtrees by a second level of switches, which are in turn connected together via the root switches. We can prove several things about the partitioning.

First, assuming ToRs have at least two different uplinks, *any partition that includes a ToR must include two of its parents*. This follows from the requirement that the partitioning be resilient to hardware failures. To see why this is true, consider a given ToR's partition. If it is connected to at most one of its parents, and its link to that parent fails, an additional failures can cause the routing protocol to fail.

Second, *there exists two ToRs that are in different partitions*. Imagine the negation—a design where all ToRs are in the same partition. If that partition fails, the entire network will go down—a violation of the need to be resilient to control plane failures.

Given the above two properties, we can prove that the subgraph exists. Let s and d be ToRs in different partitions. It can be shown that, in an sufficiently-large Clos network, s has a path to each of d's two parents in c such that those paths are node-disjoint. As s and d's parents are in different partitions, there must be an ingress into c on each of the two node-disjoint paths. These two ingress nodes, v_1 and v_2 satisfy $v_1 \neq v_2 \neq d$. Since the paths are node-disjoint, we also know that v_1 is not on v_2's path to d and vice versa. Thus, the Clos network is a suitable graph.

3. DESIGN

The previous section argues that if we want *partitioned control* in a network, we cannot simultaneously have *physical isolation* and *connectivity*. In this section, we outline two preliminary proposals that explore how we might relax each of these requirements in order to enable an approximation of an isolated, phased rollout for networks:

- *Physical partitioning:* The first proposal prioritizes physical isolation by dividing the network into two disjoint sets of switches—each independently controlled by a different controller.
- *Logical partitioning:* The second proposal allows for different controllers to manage the same physical switch in order to achieve useful global properties like connectivity. Instead, we logically isolate the two controllers by implementing simple checks at each switch.

3.1 Physical Partitioning

In physical partitioning, we separate the network into multiple node-disjoint regions, each managed by an independent control plane. Within a region, paths are discovered and computed just as they are today. Across region boundaries, we disallow the transmission of all control plane packets, though we continue to allow data packets and low-level failure detection protocols like Bidirectional Forwarding Detection (BFD). Instead, operators manually include adjacency information into the BGP configurations or SDN controllers using knowledge of the topology. Violations of this inter-region control message boundary can be checked for at the border switches.

A natural partitioning for today's multi-rooted data center topologies is to assign each subtree of the network its own region and to divide the root switches into these regions as

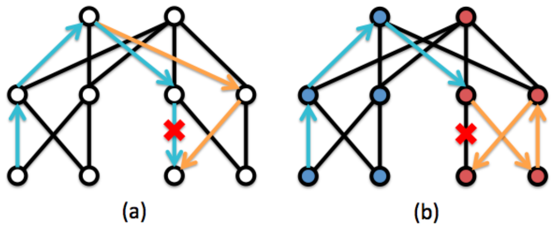

Figure 4: Rerouting with and without physical partitions. (a) is a traditional design. Blue arrows depict an example flow's path through the network. When the link under the red 'x' fails, the control plane will switch to use the orange path. (b) shows the same process in a network that is physically partitioned into red and blue regions. Here, the red region must fix the route independently.

well. Figure 4b shows an example of such a split where each cluster is a separate region. Other partitionings exist[3], but we choose this one because it creates the largest partitions such that any partition-level control plane failure will only take down a cluster-worth of servers/capacity.

Physical partitioning trivially satisfies *physical isolation* because the control planes never have access to other regions of the network. However, because we do not filter data packets, it is possible for regions to DoS one another. Moreover, this approach cannot guarantee any global properties like connectivity or balanced load. The primary challenge is therefore to be able to approximate these properties without leaking information from one region to another.

Approximating global properties. Consider a traditional SDN-based data center network. The controller will gather the statuses of all switches and links in the system and compute the best routes for the network. For the topology in Figure 4, one of the routes might be the blue path in Figure 4a. If a link fails, the controller will detect this and compute replacement routes (e.g., the orange path in Figure 4a).

In a physically partitioned deployment like the one in Figure 4b, the same process is not possible as the blue partition has no knowledge of the failure in the red partition. Instead, the blue partition must recover from the link failure without any outside help. Changes in the either partition's load balancing decisions are similarly challenging.

Our solution is to allow the blue partition to continue sending as usual and have the red partition compensate for misrouting using extra bandwidth and path dilation. For instance, in Figure 4b the red partition corrects for the blue partition's stale information by using the orange path to locally reroute packets around the failure. Essentially, red needs to be able to take any packet from the blue region at any blue/red border switch and deliver it (with best effort) to the correct destination. Previous work has shown that in modern data center topologies, the amount of path dilation for this type of local rerouting policy can be made small [24]. In a more general

[3]We leave a deeper exploration of optimal topologies and partitionings to future work.

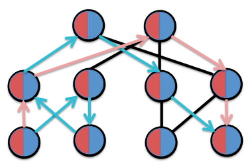

Figure 5: An example of logical partitioning. Both controllers (red, blue) control a slice of every switch. Traffic is assigned to a single partition and forwarded based on that partition's routing table entries.

network, partitions would need to be chosen with care to ensure sufficient redundancy within a region.

Rolling out changes. To roll out a new control plane update, operators would update one region at a time, providing sufficient time to monitor the network after each change. Because physical partitioning guarantees that bad regions can only affect the traffic that is explicitly forwarded to them, regions can, in principle, be updated in an arbitrary order without expanding the risk of catastrophic errors; however, other reasons may necessitate certain orders. For example, consider an address migration, where an operator wishes to move an IP prefix from one cluster to another. She may with to update the destination cluster first so that diverted traffic can be handled immediately rather than updating some other cluster.

3.2 Logical Partitioning

The second proposal gives up physical isolation in favor of more power and flexibility in the control plane. In this proposal, we partition *every* switch in the network into two or more logical switches, where each logical switch is controlled by a different control plane. The end hosts or the edge of the network assign each flow to one of the control planes. Thus, every control plane operates on a 'slice' of the entire topology and can therefore operate exactly as they do today—at least in terms of routing and load balancing logic. At the same time, the different control planes operate side-by-side on the same hardware and topology.

With this approach, ensuring the third property, *connectivity*, is fairly straightforward. For each slice, the corresponding control plane can guarantee exactly the same properties as it does today. As we explain later, we can also protect against DoS attacks from other control planes. The tradeoff is that we do not have physical isolation—switches are shared amongst all control planes and so they can affect one another. For some types of errors (e.g., those that crash the underlying ASIC), failures triggered by one control plane can affect others. The primary challenge is thus to isolate each control plane in as many ways as possible.

Approximating physical isolation. Though logical partitioning cannot provide the 'air gap' between different control planes given by physical isolation, it can provide many of the benefits. We acknowledge that not all configuration options can be isolated in this fashion as modern switches were not designed for strong isolation. For instance, combining several physical ports into a single link aggregation group (LAG) will affect all traffic over those ports; our solution cannot protect against misconfigurations here, but we anticipate that switch virtualization could be a feature in future hardware. In this section, we look at three ways we can isolate different control planes on a single network switch.

The first is *forwarding isolation* in which packets should only ever utilize a single slice's routing tables—the presence or absence of other slices' entries should have no effect. We can implement this on current hardware using a technique like VLANs (or MPLS or partitioned IP address spaces), which allows operators to divide the network into several virtual networks. Packets can be tagged to use the routing table entries of a specific VLAN, thereby providing a tool to separate the operation of different slices.

The second form of isolation is *routing table isolation* where each control plane should only be able to modify a subset of the routing table. For this, we can again build on existing mechanisms. In Section 2.1, we explained that many network operators have simple verification procedures for new configurations. We propose to add a check on any configuration to ensure that it only operates on its own VLAN (or MPLS label or IP prefix) and that it does not occupy too much of each switch's routing table. It is possible for this check itself to be buggy, but it does have the advantage that it is simple and static as opposed to the more complex steps required to ensure correctness in current networks.

The last requirement is *load isolation*. As every physical switch is shared between multiple control plane components, each control plane must also share all of its bandwidth. A naive design of a logical partitioning could allow for a network-wide resource exhaustion attack. On the other hand, assigning static bandwidth allocation to each partition does not allow for efficient use of the network. Our solution is to use weighted fair queuing mechanisms available in most data center switches to balance traffic from different slices [32]. When a switch is congested, slices cannot take more than their fair share of the bandwidth. When a switch is not congested, all slices can efficiently use all of the available bandwidth.

Rolling out changes. Unlike the physical partitioning scheme, operators here roll out changes to the entire network, but only to a new slice. The untested slice starts with a very low amount of traffic and correspondingly low priority in the weighted fair queue. For instance, the operator could initially divert 1% of all data center traffic to the new slice. After a period of monitoring, she could change the amount to 5%, and so on.[4]

4. COMPARISON

Both physical and logical partitioning have their limitations. At a course granularity, the choice comes down to a trade-off between isolation and performance. Physical isolation ensures that, as long as the static checks at the border of each region hold, control plane errors cannot spread to other regions. On the other hand, there are routing policies that are not implementable with physically isolated control planes. In some cases, this may lead to loss of connectivity even when a physical path exists. Logical partitioning makes the opposite trade-off and gives control planes an unadulterated view of the network, provided changes do not affect shared configuration options on the switches. When those options must be changed, a non-protected upgrade is necessary. In addition, latent configuration errors and bugs in the switch implementation can still take out multiple slices at once.

5. RELATED WORK

Our work builds on a great deal of prior work, particularly in the realm of software-defined networks. In addition to the extensive literature on preventing and tolerating single-controller SDN failures [10, 11, 19, 22, 31], there are several other relevant directions.

Network upgrades with a single control plane. Upgrades and configuration changes create many challenges for network operators. Most of the work in this space focuses on ensuring policy consistency/safety/low congestion level for transient states [23, 25, 27, 30, 33] or decreasing the duration of network update [18, 34]. Our work is complementary as we focus on minimizing impact of faulty controller during upgrade on a multi-controller network.

Partition tolerance. An orthogonal line of work has looked at tolerating partitions in a software-defined network [5, 29]. We, on the other hand, wish to create them. As such, our work focuses on a much weaker model: we show even if the partitions are known ahead of time and are part of the design, it is not possible to achieve connectivity for current topologies.

DCN virtualization. DCN virtualization [16, 32] is a recent push to build virtual data planes and topologies on top of an underlying physical network. Some of these are also designed to enforce bandwidth caps in order to provide performance isolation. However, the primary purpose of network virtualization is to allow different control plane configurations to coexist. Our goal is instead to isolate their failures for the purpose of enabling phased rollouts.

6. CONCLUSION

We presented two approaches for providing phased rollouts in networks. In many ways, it is more difficult to safely roll out a new network configuration compared to the same operation on a cloud application. In fact, aside from building entirely separate networks, it is impossible to achieve a similar degree of isolation while still maintaining desirable global properties like connectivity. Our solutions therefore explore how to approximate phased rollout isolation in networks and the tradeoffs required in order to do so. An additional benefit of our designs is that both of them can be implemented on current hardware.

[4]Changing priorities might not be atomic, but we anticipate changes to come in small increments, which minimizes unfairness.

7. REFERENCES

[1] Cisco NX-OS Software Malformed DHCPv4 Packet Denial of Service Vulnerability. https://quickview.cloudapps.cisco.com/quickview/bug/CSCuq39250.

[2] Google Compute Engine Incident 16007. https://status.cloud.google.com/incident/compute/16007.

[3] Summary of the October 22, 2012 AWS Service Event in the US-East Region. https://aws.amazon.com/message/680342/.

[4] What's Behind Network Downtime? . http://www-05.ibm.com/uk/juniper/pdf/200249.pdf.

[5] A. Akella and A. Krishnamurthy. A highly available software defined fabric. In *HotNets*, 2014.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[7] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[8] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. In *POPL*, 2014.

[9] K. Behr, G. Kim, and G. Spafford. *The Visible Ops Handbook*. Information Technology Process Institute, 2005.

[10] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test Openflow Applications. In *NSDI*, 2012.

[11] B. Chandrasekaran and T. Benson. Tolerating SDN Application Failures with LegoSDN. In *HotNets*, 2014.

[12] R. J. Colville and G. Spafford. Top seven considerations for configuration management for virtual and cloud infrastructures. Technical Report G00208328, Gartner, Inc., Stamford, Connecticut, October 2010.

[13] D. E. Denning. A Lattice Model of Secure Information Flow. *CACM*, 1976.

[14] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A General Approach to Network Configuration Analysis. In *NSDI*, 2015.

[15] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *SIGCOMM*, pages 58–72, 2016.

[16] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNEXT*, 2010.

[17] 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.

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

[19] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller Fault-tolerance in Software-defined Networking. In *SOSR*, 2015.

[20] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[21] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable Dynamic Network Control. In *NSDI*, 2015.

[22] 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.

[23] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.

[24] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A Fault-tolerant Engineered Network. In *NSDI*, 2013.

[25] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[27] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient Synthesis of Network Updates. In *PLDI*, 2015.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 2008.

[29] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker. CAP for Networks. In *HotSDN*, 2013.

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

[31] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *SIGCOMM*, 2014.

[32] R. Sherwood, G. Gibb, K. kiong Yap, M. Casado, N. Mckeown, and G. Parulkar. FlowVisor: A Network Virtualization Layer. Technical report, 2009.

[33] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford. HotSwap: Correct and Efficient Controller Upgrades for Software-defined Networks. In *HotSDN*, 2013.

[34] S. Vissicchio and L. Cittadini. FLIP the (Flow) Table: Fast LIghtweight Policy-preserving SDN Updates. In *INFOCOM*, 2016.