Towards Verified Self-Driving Infrastructure

Bingzhe Liu*
University of Illinois at Urbana-Champaign

Ali Kheradmand*
University of Illinois at Urbana-Champaign

Matthew Caesar
University of Illinois at Urbana-Champaign

P. Brighten Godfrey
University of Illinois at Urbana-Champaign and VMware

ABSTRACT

Modern "self-driving" service infrastructures consist of a diverse collection of distributed control components providing a broad spectrum of application- and network-centric functions. The complex and non-deterministic nature of these interactions leads to failures, ranging from subtle gray failures to catastrophic service outages, that are difficult to anticipate and repair.

Our goal is to call attention to the need for formal understanding of dynamic service infrastructure control. We provide an overview of several incidents reported by large service providers as well as issues in a popular orchestration system, identifying key characteristics of the systems and their failures. We then propose a verification approach in which we treat abstract models of control components and the environment as parametric transition systems and leverage symbolic model checking to verify safety and liveness properties, or propose safe configuration parameters. Our preliminary experiments show that our approach is effective in analyzing complex failure scenarios with acceptable performance overhead.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Availability**; Cloud computing; • **Networks** → *Network reliability*; • **Software and its engineering** → *Formal software verification.*

KEYWORDS

Self-driving infrastructure, Service infrastructure control, Verification, Parameter synthesis, Symbolic model checking

ACM Reference Format:
Bingzhe Liu, Ali Kheradmand, Matthew Caesar, and P. Brighten Godfrey. 2020. Towards Verified Self-Driving Infrastructure. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20), November 4–6, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3422604.3425949

1 INTRODUCTION

A modern service infrastructure typically consists of multiple automated or semi-automated dynamic control components working at various layers providing a broad spectrum of service- and network-centric functions. Most of these components continuously monitor various system metrics (e.g. end-to-end latency, server resources) and events (e.g. failures, traffic spikes, maintenance), and then perform actions to manipulate the system according to configured policies. For example, the scheduler in an orchestration system such as Kubernetes [24] or Docker Swarm [37] controls the placement of application containers according to server resources, while an application load balancer (e.g. NGINX, HAProxy) manages the amount of traffic sent to each application instance according to end-to-end request latency. In the meantime, various routing and traffic engineering mechanisms (e.g. BGP, ECMP, MPLS-TE) manage network connectivity and performance. As the commoditization of Internet services continues to drive down budgets, and as machine learning and other technologies continue to make automation easier (having already been applied to fault remediation [7, 33], network repair [39, 43] and database management systems [26, 30]), future services are likely to become ever closer to the vision of "self-driving" infrastructure.

Non-trivial effects emerge from such a diverse range of control components interacting with each other and with the environment. For example, a network traffic engineering component may modify routes to optimize global bandwidth, unintentionally increasing an application's traffic latency. This in turn might trigger a load balancer to re-distribute an application's incoming traffic based on the observed latency change that again affects bandwidth allocation. This complexity paves the way for a range of failures from subtle performance degradation to catastrophic outages. Moreover, these failures may only manifest under a certain combination of non-deterministic interactions, making them hard to detect before deployment. For instance, as in a Google incident (discussed in § 3.1), a software rollout may cause server overload and service outage, but only with certain configurations and if combined with network partition at a certain point of execution.

To the best of our knowledge, existing infrastructure verification and network verification tools are a poor fit for this emerging space. They either solely focus on the networking layer, ignoring higher-level control components like orchestration systems [2, 5, 15, 16, 19, 23, 31, 32, 40–42]; only consider logical properties like network reachability rather than quantitative ones (e.g., load, latency) [5, 31, 40, 41]; focus on static snapshots rather than dynamic control [19, 42]; target specific protocols (e.g., BGP, ECMP) [36]; or focus on low-level system details rather than inter-component interactions (e.g. idempotency of provisioning scripts) [34].

Given the increasing complexity of the modern service infrastructure, we believe it is time to call attention to the need for a formal understanding of dynamic service infrastructure control. As a first step, we identify several key characteristics of dynamic

*Authors contributed equally to the paper. Order determined by a coin toss.

HotNets '20, November 4–6, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8145-1/20/11…$15.00
https://doi.org/10.1145/3422604.3425949

infrastructure control components that complicate pre-deployment problem detection. We demonstrate the role of these characteristics in real-world failures by examining incident reports from large service providers and issues reported for Kubernetes, a popular open-source orchestration system. We then describe a proof-of-concept verification approach for systems of multiple dynamic control components. Our approach treats abstract models of control components and their environment as a parametric transition system, and employs symbolic model checking for verification of safety and liveness properties expressed using a temporal logic such as LTL (linear temporal logic) and CTL (computation tree logic). In case the model checker finds a problem, it produces parameters and concrete execution traces of the system that demonstrate the property violation. Our approach can also be used to find configuration parameter values that ensure the desired properties are never violated. We experiment with our prototype on two example scenarios inspired by real-world incidents and problems, showing effectiveness in finding complex failure scenarios, and promising scalability. Finally, we discuss future challenges to achieve a formal understanding of self-driving infrastructure.

2 BACKGROUND

Modern service infrastructures, ranging from large public cloud providers to small enterprise private infrastructure, are composed of multiple layers with several possible dynamic control components at each layer. To place our work in context, we briefly overview a few of these components.

Network layer. Service infrastructure networks leverage routing protocols (e.g. BGP, ECMP), security components (e.g. firewalls), and traffic engineering mechanisms (e.g. MPLS-TE), working together to connect physical servers and network devices, enforce security policies, and optimize network bandwidth and latency. Emerging self-driving networks may go further, transparently optimizing traffic, self-configuring routing, and even automating repairs. For example, hyper-scale cloud providers like Google have advanced online *traffic engineering (TE)* [18] that monitors network utilization of application classes and dynamically allocates routes to deliver prioritized max-min fair allocations.

Virtualization layer. Physical resources are typically divided into several logical slices (e.g. VMs, containers, virtual networks) that form the virtualization layer and are managed by orchestration systems like Kubernetes, Docker Swarm, or commercial products. We discuss a few controllers as examples, with a focus on the Kubernetes ecosystem (similar concepts can be found in other systems).

Deployment controller/ReplicaSet Controller controls the pod[1] configurations, update, runtime, and termination in a deployment to meet with the operator's expectations. A particularly interesting functionality is that it defines and maintains a certain number of pod replicas in the cluster for an application.

Scheduler places newly-created pods on the best nodes[2] according to certain rules. For example, it filters out nodes with insufficient resources and ranks those that remain with user-defined policies (e.g., favoring nodes with least requested resources).

[1]A *pod* is a basic execution unit that encapsulates an application's containers and their associated resources.
[2]A *node* is a virtual machine or a physical machine that runs pods.

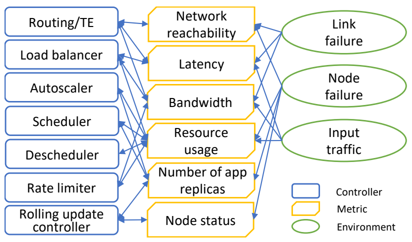

Figure 1: Examples of the complex relation between various controllers and environment in real world

Descheduler [25] evicts pods from a node according to user-defined strategies and node resource usage. `RemoveDuplicates`, for instance, evicts pods if there is more than one pod for an application on the same node.

Service/application layer. Controllers at the service or application layer manage service-level objectives related to performance and security. Examples include:

Load balancer (e.g., NGINX, HAProxy) distributes input traffic among application instances using various mechanisms, e.g., round-robin, hashing, least-connection-scheduling, or least load/latency.

Rate limiter limits the number of requests each server receives within a time period. It can be used to mitigate DDoS attacks.

Key characteristics. Summarizing the above, we identify four key characteristics that distinguish this space:

Dynamic control. The majority of these systems run as continuous loops running indefinitely (e.g. load balancer) or for a period of time (e.g. rolling update controller) during which they dynamically react to various system metrics and events.

Nontrivial interactions. These systems consist of multiple control components that may be built for separate roles but interact with each other, either through direct API calls, intertwined goals, or shared dependencies. Figure 1 illustrates some of these interactions.

Quantitative metrics. Many of these components monitor and react to quantitative metrics like end-to-end latency and load, rather than just Boolean properties such as network reachability.

Cross-layer. The control components collectively manipulate system elements at multiple logical layers. Therefore, reasoning about their behavior during deployment requires cross-layer knowledge.

3 WHAT CAN GO WRONG?

To see how the key characteristics identified in § 2 factor into real infrastructure failures, we study incident and issue reports from large service providers (§ 3.1) and Kubernetes (§ 3.2). We then describe additional hypothetical but plausible failure scenarios and demonstrate one experimentally (§ 3.3). We conclude with a few takeaways (§ 3.4).

3.1 Incident Report Study

We reviewed all incident reports made publicly available by Google Cloud between 2017-2019 [13], and by Amazon AWS between 2011-2019 [3]. Among the 242 total reports, we studied the 53 reports that had enough documented detail (42 of 230 from Google Cloud and 11 of 12 from AWS) to understand how they occurred.

We examine the role of each characteristic discussed in § 2 in causing, propagating, or complicating each incident and report the

Characteristic	Google Cloud	Amazon AWS	Total
Dynamic control	30 (71%)	8 (73%)	38 (72%)
Nontrivial interactions	12 (29%)	7 (64%)	19 (36%)
Quantitative metrics	20 (48%)	7 (64%)	27 (51%)
Cross-layer	21 (50%)	9 (82%)	30 (56%)

Table 1: System features involved in cloud incidents

results in Table 1. Generally, our study shows that these characterises play important roles in the incidents we studied.[3] We note that in some incidents, the root cause of the failure can be attributed to an environmental event, misconfigurations, bad component design, or maintenance events. However, it is the combination of these causes with the aforementioned characteristics that exacerbate the problem and result in a larger impact. We demonstrate the roles of the characteristics with two representative examples.

Google ticket #19007 [28]. An internal Publish/Subscribe messaging system (Pub/Sub), built using a replicated key-value store, was used to propagate control plane messages for many user-facing services. A routine software rollout of the key-value store restarted part of the key-value store. During the rollout, a network partition resulted in load shifting to a small number of replicas of the key-value store. Another issue caused a large number of clients to generate an unexpected amount of traffic to the replicas in the rollout region. The smaller number of replicas failed to handle these requests. Continued failures impacted the availability of Pub/Sub in the rollout region, which resulted in the cascading performance degradation of many user-facing services.

This incident involves all four characteristics in Table 1: the rollout software and load balancing are dynamic components. They interact by affecting the number of available key-value store replicas (a quantitative value). Moreover, the incident involves the service layer and the network layer.

Google ticket #18037 [27]. Unusually large requests were sent to the BigQuery "router server" (which proxies and directs requests from clients to back-end servers). This resulted in more memory allocated to process the requests. A garbage collector then consumed more CPU, which triggered a load balancer (LB) to treat the situation as potential abuse and reduce the router server's capacity. The insufficient capacity eventually resulted in the BigQuery service that depends on the router server rejecting user requests.

The router server, garbage collector, and LB are dynamic components involved in this failure. It occurred due to interactions among these components in specific, relatively complex conditions, and the LB manipulated a quantitative threshold to overly constrain the capacity of the router server. Thus, this incident illustrates all the key characteristics of § 2 except cross-layer interaction.

3.2 Issues with Kubernetes

We browsed the Kubernetes issue tracker and here illustrate two examples where the key characteristics played important roles.[4]

Kubernetes issue #75913 [12]. *Taints* allow nodes to repel a set of pods. In this issue, a Kubernetes deployment was set to place pods on a tainted node which did not accept the execution of such pods. These configurations caused two components to continuously modify the system – the deployment controller creating pods to maintain a certain number of pod replicas, and the taint manager terminating pods according to taint configuration – which then caused the deployment controller to act again, creating a loop.

Kubernetes issue #90461 [11]. A rolling update controller (RUC) was configured with `maxSurge = 1`, meaning that, to compensate for the pods that are brought down during an update rollout, at most one "additional" pod could be created beyond the "expected" number of replicas defined in the deployment spec. Meanwhile, a horizontal pod autoscaler (HPA) was deployed to dynamically adjust the number of pods based on pod resource usage. When the RUC temporarily incremented the number of pods, the HPA was triggered and falsely increased the number of "expected" pods, due to a defect in the HPA implementation (basically returning the "expected" number of pods as the "current" number of pods) – causing the RUC to create yet another "additional" pod, and so on. Note that the defect in HPA only manifests in unfortunate interactions with controllers like RUC.

3.3 What Else Could Go Wrong?

We expect the space of potential problems goes far beyond the publicly-disclosed incidents above, especially as further automation is adopted in enterprises. To explore what is possible, we describe how a few common infrastructure control components could even produce permanent oscillations (and hard-to-detect performance degradation), and we demonstrate one of these experimentally.

Oscillation caused by descheduler. The descheduler could cause permanent oscillation when it co-exists with a deployment controller or a scheduler and when they are configured inappropriately. The descheduler can set its strategy to `RemoveDuplicates` (§ 2), which could conflict with the deployment controller that requires more than one replica on the same node. In another case, the descheduler sets its strategy to be `LowNodeUtilization`, which evicts pods on a node when its CPU utilization is above a threshold. However, the scheduler may use a different threshold, which causes the pods to be descheduled and re-scheduled, and consequently moved back and forth indefinitely.

We experimentally demonstrate this oscillation problem in a Kubernetes cluster with 6 VMs consisting of 2 masters, 3 workers and 1 load balancer. The descheduler runs as a cronjob deployment [10] every 2 minutes. We deployed an app with a single pod that performed dummy CPU-intensive calculations. We set the app's requested CPU resource to 50%, and the eviction threshold for `LowNodeUtilization` policy (§ 3.3) to 45%. Figure 2 shows the oscillation in the pod placement between worker 2 and worker 3 caused by the interaction of the scheduler and the descheduler. Thus, even the interaction of related components within the same layer can lead to unforeseen problems. With the increase of control components developed or configured by uncoordinated teams, similar failure scenarios are quite plausible.

Oscillation with load balancer. A load balancer (LB) may fail to converge even under steady load for various reasons. We illustrate an interesting such case resulting from unfortunate path selection combined with a latency-based LB. In Figure 3, three servers are connected to an LB through a network of four routers.

[3]Non-trivial interaction is less common in Google reports. This may be because Google reports do not contain as rich information as Amazon; in cases of uncertainty, we conservatively marked the incident as not involving a certain characteristic.

[4]We did not perform a quantitative study of these issues because the database also includes many software bugs irrelevant to our study, as it is not an incident database.

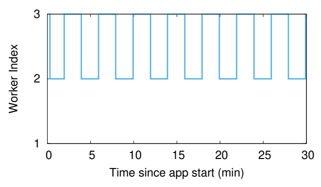

Figure 2: Oscillation in Kubernetes experiment

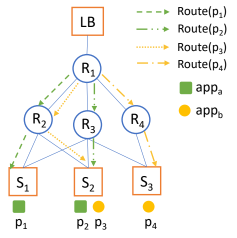

Figure 3: Load balancer oscillation example

The routers use ECMP with destination hashing, and the LB implements a weighted round-robin algorithm using service response times. Service response time depends on both server-side latency and latency of the links on the path between the server and the LB. The latency of each server and link depends on its load. There are two applications, each having 2 replicas. Figure 3 shows the placement of the replicas ($p_1 - p_4$) and the ECMP path choices.

Let w_j^i denote the percentage of app_i's traffic assigned to replica p_j by the LB. Oscillations can happen in the following situation: (1) Initially, $w_1^a > w_2^a$ and $w_4^b > w_3^b$, and the system is stable. (2) Sudden external traffic on link R_1-R_4 results in an increased response time for p_4. (3) The LB sets $w_3^b > w_4^b$, sending more of app_b's traffic towards p_3. Latency of link R_1-R_2 (shared by p_1 and p_3) increases due to the load increase, resulting in higher response time for p_1, which is sensitive to network latency. (4) The LB sets $w_2^a > w_1^a$, sending more of app_a's traffic towards p_2. This results in more load on s_2 (shared by p_2, p_3) and increased response time for p_3, which is sensitive to server latency. (5) The LB shifts app_b's traffic from p_3 back to p_4 by setting $w_4^b > w_3^b$. This decreases R_1-R_2's latency and p_1's response time. (6) The LB shifts app_a's traffic from p_2 back to p_1 by setting $w_1^a > w_2^a$, going to the same state as step (3) and failing to converge. Although a real-world LB might not oscillate infinitely, extended oscillation prior to convergence can still lead to performance degradation. We also note that this problem may be hard to catch as it depends on nondeterministic ECMP hashing.

3.4 Takeaway Points

The incident study presented here is not broad enough to produce confident quantitative estimates of frequency of certain types of failures. Instead, our goal is to show that the key characteristics that we identified do manifest in real incidents, and that their impact can be complicated, across multiple service providers and orchestration systems. Therefore, we believe such systems can benefit from verification that models the key characteristics we identified in § 2: continuous dynamic control, interactions among components (rather than verifying each component in isolation), quantitative variables (rather than just functional aspects like availability of a network path), and multiple system layers.

Existing network verification systems, e.g. [2, 4, 9, 15, 19, 23, 31, 32], do not consider one or more of the above characteristics. None consider control components above the network layer. Few consider quantitative properties, and then only in a limited way (§ 6). Network configuration verifiers [1, 4, 31] perhaps come closest: they model dynamic control, but are limited to specific routing protocols and primarily focused on reaching a converged state rather than continuous closed-loop control. They model interactions among routers, but limited to routing advertisements or network access control, rather than the richer space of interactions that exist (Fig. 1).

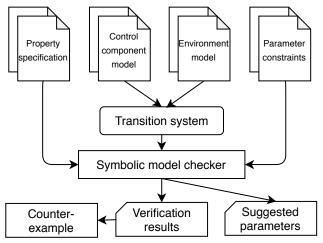

Figure 4: The workflow of our proposed proof-of-concept

The above four characteristics are, however, typical of modern distributed systems software. And not surprisingly, our solution approach (§ 4) is common in verification of other distributed systems. We apply this general technique to the novel domain of service infrastructure control.

4 TOWARDS VERIFIED INFRASTRUCTURE

As a first step toward alleviating the problems illustrated in the previous section, we propose a proof-of-concept verification approach for dynamic service infrastructure control. Figure 4 illustrates the overall workflow. We briefly describe our approach and then examine its use on two example scenarios inspired by incidents and problems mentioned in the previous section. We also examine the approach's scalability. We emphasize that the goal of this section is not to introduce a new formal technique. Rather, it is to demonstrate the use of a standard technique, namely symbolic model checking, in detecting and preventing the problems under consideration.

4.1 Proof of Concept Design

Symbolic model checking. Our goal is to capture failures arising from a combination of bad design or configuration, environmental conditions, and unfortunate interactions of control components with each other and/or with the environment. A common technique for verification of concurrent systems is to model the behavior of all components as a non-deterministic transition system and employ model checking to efficiently search the space of possible executions. Furthermore, we may want certain parts of our models to be parametric (configuration parameters like number of instances to be simultaneously updated in an update rollout, or environmental conditions like link latency). The model checker should figure out the parameters, in addition to execution steps, that lead to failure (or suggest safe parameters preventing it). Symbolic model checkers are useful in such scenarios. A symbolic model checker works with symbolic states (i.e. sets of states represented together in a logical form) rather than enumerating individual explicit states. Beyond efficiency benefits, this lets the user define symbolic parameters in her models rather than concrete values.

In the current work we use NuXMV [6], a state-of-the-art symbolic model checker supporting BDD/SAT/SMT-based symbolic model checking for both finite domains and infinite domains (models containing integer or real values, useful for modeling quantitative metrics such as real time, load, and latency). The tool provides a modular but low-level modeling language for defining the transition system, and implements a collection of advanced symbolic model checking algorithms for verification of properties encoded in

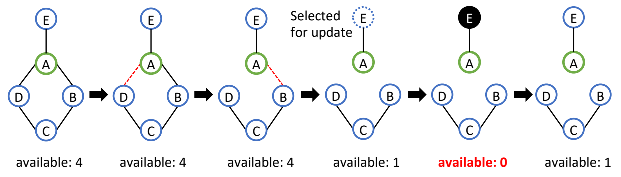

Figure 5: Counter-example for case study experiment 1

various formalism including LTL and CTL. It also provides limited parameter synthesis functionality (i.e. synthesizing parameters that ensure a desired property holds).

Modeling. We need two general categories of models to define our transition system: (1) Models of the various infrastructure elements, environment, metrics, events, and the rules that govern their evolution. Examples include network topology, failure events, input traffic, and the relation between load and latency or resource usage for each link or device. (2) Models of control components and how they react to the environment according to their configurations. This includes routing protocols, load balancer, autoscaler, scheduler, rolling update controller, etc.

We envision providing a high-level modeling language that facilitates modeling of control components and environment, accompanied by a library of common control system and environment models. We would then compile the model into the lower-level language used by the underlying model checker, possibility with domain-specific optimizations. In our current proof of concept, we directly model everything in NuXMV's language.

Specification. We are interested in properties about the dynamic aspects of an evolving system. This includes both safety properties stating nothing bad will ever happen (e.g. no server will ever get overloaded under certain input load), and liveness properties stating something good will eventually happen (e.g. system will stabilize in an stable environment). We usually encode these properties using a temporal logic such as LTL or CTL. For example, in LTL, $G(P)$ (always P) where P is a proposition about the system state (or another LTL formula) is true iff in all execution traces of the system (starting from the current state), P holds in all states. $E(P)$ (eventually P) holds iff in all execution traces of the system there is a state in which P holds. NuXMV supports multiple specification languages including CTL and LTL.

4.2 Case Study Experiments

Here we briefly describe our case study experiments. Full implementation details are publicly available at [21].

1. Update rollout + network partition (safety). We create a simplified scenario inspired by the root cause of the failure in the actual Google incident discussed in § 3.1. The infrastructure consists of a topology of several connected nodes. A service is running on the infrastructure using a subset of the nodes as *service nodes*, and one of the nodes as the *service front-end* that distributes incoming requests among service nodes. We model a rollout controller that takes service nodes down, updates them, and then brings them back up again, in a non-deterministic order. The rollout may bring up to p nodes down simultaneously. We also model link failures: up to k links may fail at non-deterministic points of execution. There is a loop that re-computes the reachability of the front-end to each service node after any change. We want to make sure that the

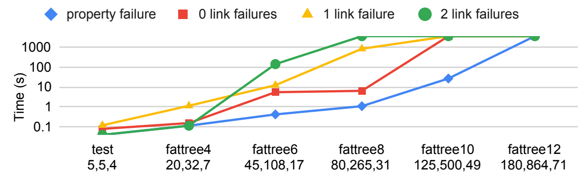

Figure 6: Performance results. Numbers below topology name indicate the number of nodes, links, and service nodes respectively. Numbers above 1000s line indicate timeout.

number of *available* (i.e up and reachable) service nodes never goes below a threshold m, otherwise the available service nodes may fail due to overload. We can encode this property as the LTL formula $G(converged \implies available >= m)$: *always whenever the reachability computation is converged, the number of available service nodes must be at least* m. Depending on the topology and values of p, k and m, the property may fail. Figure 5 shows a counter-example produced by the model checker for the parameters $p = m = 1, k = 2$.

By choosing the values of p, k, and m, an operator can use the system to make sure their rollout config. is safe under assumptions about the number of failures. It can also be used to synthesize parameters that make the given property valid. Say we are interested in finding safe non-zero values for p, given the property and $k = 1$, $m = 1$. The system in this case suggests the values $p \in \{1, 2\}$.

2. Load balancer + ECMP (liveness). Next, we check a liveness property in a model of the load balancer example described in § 3.3. We model the topology in Figure 3. We hard-code ECMP path selections described in the example.[5] We model each application's input traffic as a positive real-valued parameter. We model each w^i_j as an integer (only allowing 0 or 1 values). The load on each server/link is the sum of the amount of traffic on that server/link. We also model a non-deterministic one-time external traffic increase on one of the links. E.g. $load_{R_2} = w^a_1 \cdot t^a + w^b_3 \cdot t^b + e_{R_2}$, where t^i is the input traffic to app_i and e_{R_2} is the external traffic on R_2. We assume the latency of each server (for each app) has a linear relationship with the load on that server. E.g., $latency^a_{S_2} = m^a \cdot load_{S_2} + l^a$ where m^a and l^a are (positive) real-valued parameters for app_a. The latency of each link is calculated in a similar way, except that the latency is the same for both apps. The load balancer takes turns setting the weights for app_a and app_b. For each app, it checks each replica's response time and sets the weights accordingly (either 0 or 1). We model a "smart" load balancer that considers the effect of weight changes on the response times in weight calculations.

We check the LTL formula $F(G(stable))$: *eventually the system becomes always stable*, where *stable* means the weight selections do not change. Interestingly, the model checker finds a counter-example where the system is unstable even before the sudden external traffic. To find a more interesting counter-example, we check $stable \implies F(G(stable))$: *if the system is initially stable, it eventually becomes always stable [again]*. The model checker finds a counter-example (values for input loads, latency parameters, and a lasso-shaped execution path) where the system is stable and starts oscillating after an external traffic increase on the link R_1-R_4.

[5]One could alternatively model ECMP's non-deterministic path selection and let the model checker find the unfortunate choices.

Scalability. As a preliminary assessment of the scalability of our approach, we repeat experiments in case study 1 (safety) with fat tree topologies with 32 to 864 links. In each topology one leaf is the front-end and all other leaves are service nodes. We set $p = m = 1$ and use various numbers for k. We terminate the model checker if it cannot make a decision within 1 hour. All experiments run inside a VM on a MacBook Air, 1.6GHz Intel Core i5, 8GB RAM.

Figure 6 shows the runtime results. The horizontal axis shows input topologies in ascending size order. The first topology (test) corresponds to Fig. 5 and the rest are fat trees. The blue line shows runtime for cases in which the property is not valid. This corresponds to setting k equal to $2, 2, 3, 4, 5, 6$ along the horizontal axis.

In all of our experiments the time to find a violation is significantly shorter than the time for verification. E.g., in fattree6 it only takes 0.5s to find a violation ($k = 3$), while it takes > 12,141s to verify the property for $k = 1, 2$ respectively. This is expected: finding a violation usually requires few steps of execution, but the model checker usually has to exhaust the whole state space to verify a property that holds. In addition, runtime generally increases exponentially with the size of the input. The model checker times out for any k on fattree12. Moreover, the runtime for the passing cases increases exponentially with k.[6]

5 DISCUSSION AND FUTURE WORK

Scalability. State-space explosion is a common problem with model checking based approaches. Still, given the size of the state space and our deliberate omission of any domain-specific optimizations, the observed performance in our experiment is promising. Inspired by the successes in network verification [5, 16, 31, 41], we are hopeful to devise domain-specific optimizations to scale our verification approach in future work. Future work should also evaluate and improve scalability with regard to the complexity of system under investigation (e.g., number of distinct types of control components).

Formal modeling of control components. This is one of the main challenges of our approach, and we expect it to be exacerbated as complex control components continue to emerge. Still, we believe there is hope. Standardization of orchestration systems like Kubernetes and Istio [17] can provide a unified understanding of control components. Extracting models from these system would become a one-time effort by experts. Besides, the push towards "infrastructure as code" provides standard languages for users to define their infrastructure (Terraform [38], AWS CloudFormation [8]) and a formal model may be derived from the code. Also, modelling all implementation details may not be necessary: we only need enough to verify certain classes of target properties. A future direction is to define high-level modeling languages that capture the control logic of these components at the right level of abstraction.

Beyond traditional verification. We discussed the possibility of synthesizing safe configuration parameters for controllers. We could also help with risk assessment by examining the blast radius of an operational event. A more ambitious goal would be to synthesize the entire infrastructure control from abstract models and expected properties. It would also be interesting to take the probability of events (e.g. link failures) into account and provide probabilistic reasoning capabilities similar to [9, 35]. Moreover, the support for infinite domains enables us to verify models and properties with real time values (e.g., the system should converge within 5s).

Ease of use. One of our future goals is to make our approach useful even for formal verification non-experts. Directions that could help include automatic extraction of models from standardized systems and/or configuration languages; a high-level domain-specific modeling and specification language; a library of common controller models and common properties of interest; and automatic model or property inference [20, 29].

[6]The apparent irregularity of test and fattree4 between $k = 2$ and $k = 0, 1$ is because the property does not hold for $k = 2$ in those cases while it holds for lower k.

6 RELATED WORK

Model checking in network verification. Our work is closely related to past network verifiers (e.g. [5, 22, 23, 31, 40, 41]), many of which effectively perform symbolic model checking for the control or data plane. We go beyond just network controllers or data plane elements and packet-level reachability properties. However, we believe inspiration from these systems' domain-specific optimizations may be relevant for scalability in our domain.

Quantitative/probabilistic network verification. [2, 19, 42] extend data plane verification with quantitative properties such as minimum available bandwidth. Unlike our approach, these works focus on static snapshots rather than dynamic control or only focus on network level components. [9, 35] allow modeling and reasoning about probabilistic network behavior (e.g., link failure), but do not consider dynamic control and temporal properties.

Recently, [36] checks if a control plane configuration may cause link overloads under failures. Unlike us, [36] is tied to fixed routing protocols (BGP, OSPF), load balancing (ECMP), events (link failure), metrics (link load), and state (convergence).

Infrastructure automation verification. There is progress [34] in configuration verification for provisioning tools like Puppet and Ansible. This line of work considers low-level effects of configurations on machine/VM state (e.g. determinancy, idempotency) rather than the dynamic control properties that we consider.

Distributed systems verification. We are not focusing on verification of specific distributed protocols or of distributed system implementations [14]. Although our approach is a common technique for distributed systems verification, we target a specific novel domain. We focus on the reliability of infrastructure automation rather than correctness of distributed apps running on the infrastructure. This may enable future domain-specific optimizations.

7 CONCLUSION

Our study of several real-world high-impact failures and problems shows the need for formal understanding of dynamic infrastructure control components and their interaction. As a first step towards verified self-driving infrastructure, we propose a verification approach that aims to go beyond the scope of what current network/infrastructure verification tools can verify. Our preliminary experiments with our proof-of-concept yield promising results.

Acknowledgements. We thank Bilal Anwer, Pramod Jamkhedkar, Rakesh Sinha, and Kostas Oikonomou of AT&T for helpful early discussions. We also thank Prof. Tianyin Xu and the anonymous reviewers of HotNets for their valuable feedback. This material is based upon work supported by the National Science Foundation under grant No. CNS-1513906 and by the Maryland Procurement Office under contract No. H98230-18-D-0007.

REFERENCES

[1] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast multilayer network verification. In R. Bhagwan and G. Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 201–219. USENIX Association, 2020.

[2] A. Abhashkumar, J. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu. Supporting diverse dynamic intent-based policies using janus. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017, Incheon, Republic of Korea, December 12 - 15, 2017*, pages 296–309. ACM, 2017.

[3] Amazon. Aws post-event summaries. https://aws.amazon.com/cn/premiumsupport/technology/pes/, June 2020.

[4] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 155–168. ACM, 2017.

[5] M. Canini, D. Venzano, P. Peresíni, D. Kostic, and J. Rexford. A NICE way to test openflow applications. In S. D. Gribble and D. Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 127–140. USENIX Association, 2012.

[6] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuxmv symbolic model checker. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[7] Cisco. Cisco automated fault management. https://www.cisco.com/c/dam/en/us/services/collateral/services/bcs-afm-aag.pdf, August 2018.

[8] CloudFormation. Aws cloud: formation model and provision all your cloud infrastructure resources. https://aws.amazon.com/cloudformation/, June 2020.

[9] T. Gehr, S. Misailovic, P. Tsankov, L. Vanbever, P. Wiesmann, and M. T. Vechev. Bayonet: probabilistic inference for networks. In J. S. Foster and D. Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 586–602. ACM, 2018.

[10] Github. Github of kubernete descheduler. https://github.com/kubernetes-sigs/descheduler, June 2020.

[11] Github. Hpa v2 scales up deployment during rolling updates 90461. https://github.com/kubernetes/kubernetes/issues/90461, June 2020.

[12] Github. Replicaset controller bug: continuously creating pod to tainted nodes 75913. https://github.com/kubernetes/kubernetes/issues/75913, June 2020.

[13] Google. Google cloud incident reports. https://status.cloud.google.com/summary, June 2020.

[14] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In E. L. Miller and S. Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015.

[15] A. Horn, A. Kheradmand, and M. R. Prasad. Delta-net: Real-time network verification using atoms. In A. Akella and J. Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 735–749. USENIX Association, 2017.

[16] A. Horn, A. Kheradmand, and M. R. Prasad. A precise and expressive lattice-theoretical framework for efficient network verification. In *27th IEEE International Conference on Network Protocols, ICNP 2019, Chicago, IL, USA, October 8-10, 2019*, pages 1–12. IEEE, 2019.

[17] Istio. Istio: connect, secure, control, and observe services. https://istio.io/, June 2020.

[18] 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 D. M. Chiu, J. Wang, P. Barford, and S. Seshan, editors, *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013*, pages 3–14. ACM, 2013.

[19] G. Juniwal, N. Bjorner, R. Mahajan, S. Seshia, and G. Varghese. Quantitative network analysis. *Technical report*, 2016.

[20] A. Kheradmand. Automatic inference of high-level network intents by mining forwarding patterns. In *SOSR '20: Symposium on SDN Research, San Jose, CA, USA, March 3, 2020*, pages 27–33. ACM, 2020.

[21] A. Kheradmand. Case study implementation details. https://github.com/kheradmand/verdict-hotnets20, 2020.

[22] A. Kheradmand and G. Rosu. P4K: A formal semantics of P4 and applications. *CoRR*, abs/1804.01468, 2018.

[23] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. J. Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 59–72. USENIX Association, 2015.

[24] Kubernetes. Kubernetes: Production-grade container orchestration. https://kubernetes.io/, June 2020.

[25] Kubernetes-sigs. Descheduler. https://github.com/kubernetes-sigs/descheduler, June 2020.

[26] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 631–645. ACM, 2018.

[27] Google. Google bigquery incident 18037. https://status.cloud.google.com/incident/bigquery/18037, June 2020.

[28] Google. Google operations incident 19007. https://status.cloud.google.com/incident/google-stackdriver/19007, June 2020.

[29] S. Moon, J. Helt, Y. Yuan, Y. Bieri, S. Banerjee, V. Sekar, W. Wu, M. Yannakakis, and Y. Zhang. Alembic: Automated model inference for stateful network functions. In J. R. Lorch and M. Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 699–718. USENIX Association, 2019.

[30] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[31] S. Prabhu, K. Chou, A. Kheradmand, B. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In R. Bhagwan and G. Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 953–967. USENIX Association, 2020.

[32] S. Prabhu, A. Kheradmand, B. Godfrey, and M. Caesar. Predicting network futures with plankton. In K. Chen and J. Padhye, editors, *Proceedings of the First Asia-Pacific Workshop on Networking, APNet 2017, Hong Kong, China, August 3-4, 2017*, pages 92–98. ACM, 2017.

[33] E. Research. A look at automated fault management with machine learning. https://www.ericsson.com/en/blog/2019/6/automated-fault-management-machine-learning, June 2019.

[34] R. Shambaugh, A. Weiss, and A. Guha. Rehearsal: a configuration verification tool for puppet. In C. Krintz and E. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 416–430. ACM, 2016.

[35] S. Smolka, P. Kumar, N. Foster, D. Kozen, and A. Silva. Cantor meets scott: semantic foundations for probabilistic networks. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 557–571. ACM, 2017.

[36] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella. Detecting network load violations for distributed control planes. In A. F. Donaldson and E. Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 974–988. ACM, 2020.

[37] D. Swarm. Docker swarm: Swarm mode overview. https://docs.docker.com/engine/swarm/, June 2020.

[38] Terraform. Terraform: use infrastructure as code to provision and manage any cloud, infrastructure, or service. https://www.terraform.io/, June 2020.

[39] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated bug removal for software-defined networks. In A. Akella and J. Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 719–733. USENIX Association, 2017.

[40] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella. Liveness verification of stateful network functions. In R. Bhagwan and G. Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 257–272. USENIX Association, 2020.

[41] Y. Yuan, S. Moon, S. Uppal, L. Jia, and V. Sekar. Netsmc: A custom symbolic model checker for stateful network verification. In R. Bhagwan and G. Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 181–200. USENIX Association, 2020.

[42] Y. Zhang, W. Wu, S. Banerjee, J. Kang, and M. A. Sánchez. Sla-verifier: Stateful and quantitative verification for service chaining. In *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, pages 1–9. IEEE, 2017.

[43] W. Zhou, J. Croft, B. Liu, E. Ang, and M. Caesar. Automatically correcting networks with NEAt. In S. Banerjee and S. Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 595–608. USENIX Association, 2018.