A Network-State Management Service

Peng Sun (Princeton) Ratul Mahajan Jennifer Rexford (Princeton)
Lihua Yuan Ming Zhang Ahsan Arefin
Microsoft

Abstract— We present Statesman, a network-state management service that allows multiple network management applications to operate independently, while maintaining network-wide safety and performance invariants. Network state captures various aspects of the network such as which links are alive and how switches are forwarding traffic. Statesman uses three views of the network state. In *observed state*, it maintains an up-to-date view of the actual network state. Applications read this state and propose state changes based on their individual goals. Using a model of dependencies among state variables, Statesman merges these *proposed states* into a *target state* that is guaranteed to maintain the safety and performance invariants. It then updates the network to the target state. Statesman has been deployed in ten Microsoft Azure datacenters for several months, and three distinct applications have been built on it. We use the experience from this deployment to demonstrate how Statesman enables each application to meet its goals, while maintaining network-wide invariants.

Categories and Subject Descriptors:
C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design;
C.2.3 [**Computer-Communication Networks**]: Network Operations—*network management*

Keywords:
Network state; software-defined networking; datacenter network

1. Introduction

Today's planetary-scale services (e.g., search, social networking, and cloud computing) depend on large datacenter networks (DCNs). Keeping these networks running smoothly is difficult, due to the sheer number of physical devices, and the dynamic nature of the environment. At any given moment, multiple switches experience component failures, are brought down for planned maintenance or saving energy, are upgraded with new firmware, or are reconfigured to adapt to prevailing traffic demand. In response, DCN operators have developed an array of automated systems for managing the *traffic* (e.g., traffic engineering [12, 13], server load balancing [25], and network virtualization [17]) and the *infrastructure* (e.g., hardware power control for failure mitigation or energy saving [10, 30], switch firmware upgrade, and switch configuration management [4, 5]).

SIGCOMM'14, August 17–22, 2014, Chicago, Illinois, USA.

http://dx.doi.org/10.1145/2619239.2626298.

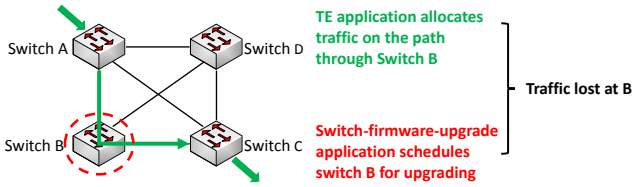

Figure 1: Example of an application conflict

Each management application is highly sophisticated in its own right, usually requiring several years to design, develop, and deploy. It typically takes the form of a "control loop" that measures the current state of the network, performs a computation, and then reconfigures the network. For example, a traffic engineering (TE) application measures the current traffic demand and network topology, solves an optimization problem, and then changes the routing configuration to match demand. These applications are complicated because they must work correctly even in the presence of failures, variable delays in communicating with a distributed set of devices, and frequent changes in network conditions.

Designing and running a single network management application is challenging enough; large DCNs must simultaneously run multiple applications—created by different teams, each reading and writing some part of the network state. For instance, both a TE application and an application to mitigate link failures need to run continuously to, respectively, adjust the routing configuration continuously and detect and resolve failures quickly.

These applications can conflict with each other, even if they interact with the network at different levels, such as establishing network paths, assigning IP addresses to interfaces, or installing firmware on switches. One application can inadvertently affect the operation of another. As an example in Figure 1, suppose a TE application wants to create a tunnel through the switch B, while a firmware-upgrade application wants to upgrade B. Depending on which action happens first, either the TE application fails to create the tunnel (because B is already down), or the already-established tunnel ultimately drops traffic during the firmware upgrade.

Running multiple management applications also raises the risk of network-wide failures because their complex interactions make it hard to reason about their combined effect. Figure 2 shows an example where one application wants to shut down switch $AggB$ to upgrade its firmware, while another wants to shut down switch $AggA$ to mitigate packet corruption. While each application acting alone is fine, their joint actions would disconnect the ToRs (top-of-rack). To prevent such disasters, it is imperative to ensure that the collective actions of the applications do not violate certain network-wide invariants, which specify basic safety and performance requirements for the network. For instance, a pod of servers must not be disconnected from the rest of the datacenter, and there must be some minimum bandwidth between each pair of pods.

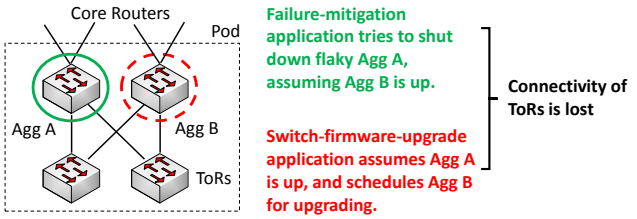

Figure 2: Example of a safety violation

DCN operators could conceivably circumvent the problems that stem from running multiple applications by developing a single application that performs *all* functions, e.g., combining TE, firmware upgrade, and failure mitigation. However, this monolithic application would be highly complex, and worse, it would need to be extended repeatedly as new needs arise. Thus, DCN operators need a way to keep the applications separate.

Another option is to have explicit coordination among applications. Corybantic [23] is one recent proposal that follows this approach. While coordination may be useful for a subset of applications, using it as a general solution to the problem of multi-application co-existence imposes high overhead on applications. It would require each application to understand the intended network changes of all others. To make matters worse, every time an application is changed or a new one is developed, DCN operators would need to test again, and potentially retrofit some existing applications, in order to ensure that all of them continue to co-exist safely.

We argue that network management applications should be built and run in a loosely coupled manner, without explicit or implicit dependencies on each other, and conflict resolution and invariant enforcement should be handled by a separate management system. This architecture would simplify application development, and its simplicity would boost network stability and predictability. It may forgo some performance gains possible through tight coupling and joint optimization. However, as noted above, such coupling greatly increases application complexity. Further, since applications may have competing objectives, a management system to resolve conflicts and maintain invariants would be needed anyway. We thus believe that loose coupling of applications is a worthwhile tradeoff in exchange for significant reduction in complexity.

In this paper, we present Statesman, a network-state management service that supports multiple loosely-coupled management applications in large DCNs. Each application operates by reading and writing some part of the network state at its own pace, and Statesman functions as the conflict resolver and invariant guardian. Our design introduces two main ideas that simplify the design and deployment of network management applications:

Three views of network state (observed, proposed, target): In order to prevent conflicts and invariant violations, applications cannot change the state of the network directly. Instead, each application applies its own logic to the network's *observed* state to generate *proposed* states that may change one or more state variables. Statesman merges all proposed states into one *target* state. In the merging process, it examines all proposed states to resolve conflicts and ensures that the target state satisfies an extensible set of network-wide invariants. Our design is inspired by version control systems like `git`. Each application corresponds to a different `git` user and (i) the observed state corresponds to the code each user "pulls", (ii) the proposed state corresponds to the code the user wants to "push", and (iii) the target state corresponds to the merged code that is ultimately stored back in the shared repository.

Dependency model of state variables: Prior work on abstracting network state for applications models the state as independent variable-value pairs [18, 19]. However, this model does not contain enough semantic knowledge about how various state variables are related, which hinders detection of conflicts and potential invariant violations. For example, a tunnel cannot direct traffic over a path that includes a switch that is administratively down. To ensure safe merging of proposed states, Statesman uses a *dependency model* to capture the domain-specific dependencies among state variables.

Statesman has been deployed in ten datacenters of Microsoft Azure for several months. It currently manages over 1.5 million state variables from links and switches across the globe. We have also deployed two management applications—switch firmware upgrade and link failure mitigation, and a third one—inter-datacenter TE—is undergoing pre-deployment testing. The diverse functionalities of these applications showcase how Statesman can safely support multiple applications, without hurting each other or the network. We also show that these benefits come with reasonably low overhead. For instance, the latency for conflict resolution and invariant checking is under 10 seconds even in the largest DCN with 394K state variables. We believe that our experience with Statesman can inform the design of future management systems for large production networks.

2. Statesman Model

In this section, we provide more details on the model of network state underlying Statesman, and how applications use that model.

2.1 Three Views of Network State

Although management applications have different functionalities, they typically follow a control loop of reading some aspects of the network state, running some computation on the state, and accordingly changing the network. One could imagine that each application reads and writes states to the network devices directly.

However, direct interaction between the network and applications is undesirable for two reasons. First, it cannot ensure that individual applications or their collective actions will not violate network-wide invariants. Second, reliably reading and writing network state is complex because of response-time variances and link or switch failures. When a command to a switch takes a long time, the application has to decide when to retry, how many times, and when to give up. When a command fails, the application has to parse the error code and decide how to react.

Given issues above, Statesman abstracts the network state as multiple variable-value pairs. Further, it maintains three different types of views of network state. Two of these are *observed state* (OS) and *target state* (TS). The OS is (a latest view of) the actual state of the network, which Statesman keeps up-to-date. Applications read the OS to learn about current network state. The TS is the desired state of the network, and Statesman is responsible for updating the network to match the TS. Any success or failure of updating the network towards the TS will be (eventually) reflected into the OS, from where the applications will learn about the prevailing network conditions.

The OS and TS views are not sufficient for resolving conflicts and enforcing invariants. If applications directly write to the TS, the examples in Figure 1 and 2 can still happen. We thus introduce the third type of view called *proposed state* (PS) that captures the state desired by applications. Each application writes its own PS.

Statesman examines the various PSes and detects conflicts among them and with the TS. It also validates them against a set of network-wide invariants. The invariants capture the basic safety and performance requirements for the network (e.g., the network should be physically connected and each pair of server pods should be able to survive a single-switch failure). The invariants are independent of what applications are running. Only non-conflicting and invariant-compliant PSes are accepted and merged into the TS.

2.2 Dependency Model of State Variables

Applications read and write different state variables of the network, e.g., hardware power, switch configuration, switch routing, and multi-switch tunneling. Statesman thus provides the state variables at multiple levels of granularity for applications' needs (more details are covered in §4, and Table 2 lists some examples).

But state variables are not independent. The "writability" of one state variable can depend on the values of other state variables. For example, when a link interface is configured to use the traditional control-plane protocol (e.g., BGP or OSPF), OpenFlow rules cannot be installed on that interface. In another example, when the firmware of a switch is being upgraded, its configuration cannot be changed and tunnels cannot be established through it. Thus, when proposing new values of state variables, conflicts can arise because a state variable in one application's PS may become unchangeable due to some *dependent* state variables in another application's PS. Requiring applications to understand the complex cross-variable dependency will go against our goal of running them in a loosely coupled manner. For instance, it will be difficult for a TE application to have to consider how one specific switch configuration affects its operation.

Therefore, Statesman does not treat network state as a collection of independent variables but includes a model of dependencies among them. These dependencies are used when checking for conflicts and invariant violations. Based on the dependency model, Statesman also exposes to applications the "controllability" of each state variable as an additional logical variable. Thus, an application can read only the state variables of interest and their controllability variables to decide whether it is able to propose new values for those variables of interest. For example, a TE application can read the "path" variable (i.e., a tunnel through multiple switches) and whether it is currently controllable, which is computed by Statesman based on various hardware and routing configurations of the switches along the path; the TE application does not need to reason about the related hardware and routing configurations itself.

2.3 Application Workflow

In the observed-proposed-target model, the workflow of management applications is simple. Each application reads the OS, runs its computational logic, and writes a newly generated PS. Statesman generates the new TS after resolving conflicts and invariant violations in the PSes and merging the accepted ones.

In this model, some application proposals may be rejected. Handling this rejection does not impose extra overhead on applications; even if interactions with the network were not mediated by Statesman, applications have to be prepared to be unable to update the network to the desired state (e.g., due to failures during the update process). When Statesman rejects a proposal, applications get detailed feedback on the reason for rejection (§5), at which point applications can propose a new PS in an informed manner.

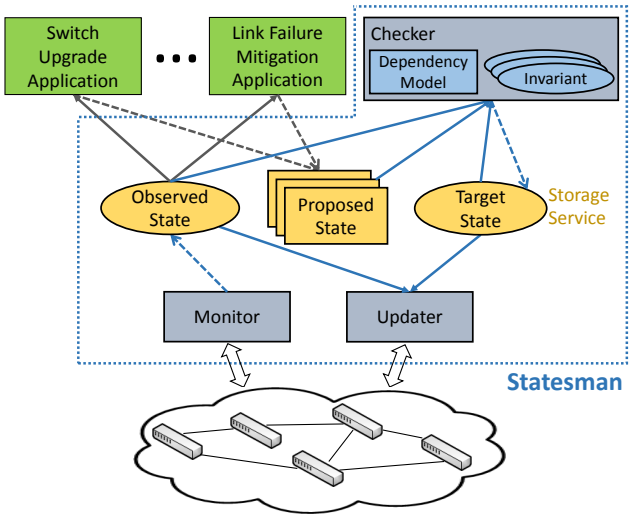

Figure 3: Statesman architecture overview

Component	Input	Output
Monitor	Switch/link data	OS
Checker	OS PSes TS	TS
Updater	OS TS	Switch update commands

Table 1: Input and output of each component in Statesman

Per our desire for loose coupling, applications make proposals independently. It is thus possible that PSes of two applications frequently conflict or we do not compute a network state that satisfies all applications (even though such a state exists). In our experience with Statesman (so far), such cases are not common. Thus, we believe that the simplicity of loose coupling outweighs the complexity of tight coupling. In cases where coordination among two applications is highly beneficial, it may be done out of band such that applications make proposals after consulting each other. Done this way, most applications and the system as a whole stay simple, and the complexity cost of coordination is borne only in parts that benefit the most from the coordination.

3. System Overview

Figure 3 shows the architecture of Statesman. It has four components: storage service, checker, monitor, and updater. We outline the role of each below. The following sections present more details.

Storage service is at the center of the system. It persistently stores the state variables of OS, PS, and TS and offers a highly-available, read-write interface for other components and applications. It also handles all data availability and consistency issues, which allows all other components to be completely stateless—in each round of their operations, they just read the latest values of the needed state variables. This stateless mode of operation simplifies the design of the other components.

The checker, monitor, and updater independently interact with the storage service, and the latter two also interact with the network. Table 1 summarizes the input and output of each component.

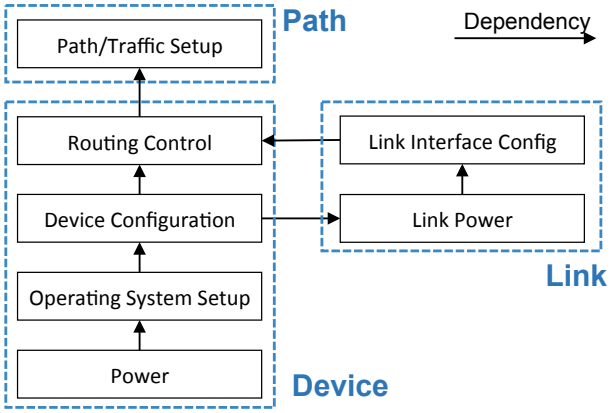

Figure 4: Network state dependency model

Checker plays a pivotal role of generating the TS. After reading the OS, PSes, and TS from the storage service, the checker first examines whether some PSes are applicable with respect to the latest OS (e.g., the proposed change may have already been made or cannot be made at all due to a failure). It then detects conflicts among PSes with the state dependency model and resolves them with one of two configurable mechanisms: last-writer-wins or priority-based locking. After merging the valid and non-conflicting PSes into the TS, the checker examines the TS for operator-specified invariants. It writes the TS to the storage service only if the TS complies with the invariants. It also writes the acceptance or rejection results of the PSes to the storage service, so applications can learn about the outcomes and react accordingly.

Monitor periodically collects the current network state from the switches and links, transforms it into OS variables, and writes the variables to the storage service. In addition to making it easy for other components and applications to learn about current network state, the monitor also shields them from the heterogeneity among devices. Based on the switch vendor and the supported technologies, it uses the corresponding protocol (e.g., SNMP or OpenFlow) to collect the network statistics, and it translates protocol-specific data to protocol-agnostic state variables. Other components and applications use these abstract variables without worrying about the specifics of the underlying infrastructure.

Updater reads the OS and TS and translates their difference into update commands that are then sent to the network. The updater is memoryless—it applies the latest difference between the OS and TS without regard to what happened in the past. Like the monitor, the updater handles how to update heterogeneous devices with a command template pool, and allows other components and applications to work with device- and protocol-agnostic state variables.

4. Managing Network State

We now describe the various aspects of Statesman in more details, starting with the network-state data model. We use the examples in Table 2 to illustrate how we build the state dependency model, and how to use and extend the model.

4.1 The State Dependency Model

Managing a DCN involves multiple levels of control. To perform the final function of carrying traffic, the DCN needs to be properly powered and configured. Statesman aims to support operations in the complete process of bringing up a large DCN from scratch to normal operations. In order to capture the relationship among the state variables at different levels of the management process, we use the state dependency model of Figure 4. We use the process of bootstrapping a DCN as an example to explain this model.

Entity	Level in dependency	Example state variables	Permission
Path	Path/traffic setup	Switches on path MPLS or VLAN config	ReadWrite ReadWrite
Link	Link interface config	IP assignment Control plane setup	ReadWrite ReadWrite
	Link power	Interface admin status Interface oper status	ReadWrite ReadOnly
	N/A (counters)	Traffic load Packet drop rate	ReadOnly ReadOnly
Device	Routing control	Flow-link routing rules Link weight allocation	ReadWrite ReadWrite
	Device configuration	Mgmt. interface setup OpenFlow agent status	ReadWrite ReadWrite
	Operating system setup	Firmware version Boot image	ReadWrite ReadWrite
	Power	Admin power status Power unit reachability	ReadWrite ReadOnly
	N/A (counters)	CPU utilization Memory utilization	ReadOnly ReadOnly

Table 2: Example network state variables

At the bottom of the dependency model is the power state of network devices. With the power cable properly plugged in and electricity properly distributed to the switches, we then need to control which switch operating system (i.e., firmware) runs. Running a functioning firmware on a switch is a prerequisite for managing switch configuration, e.g., use switch vendor's API to configure the management interface, boot up compatible OpenFlow agent, etc.

With device configuration states ready, we are able to control the link interfaces on the switch now. The fundamental state variable of a link is its being up or down. The configuration of a link interface follows when the link is ready to be up. There are various link-interface configuration states, such as IP assignment, VLAN setup, ECMP-group assignment, etc. Consider an example of control plane setup where a link interface can be configured to use the OpenFlow protocol or traditional protocols like BGP. For the chosen option, we need to set it up: either an OpenFlow agent needs to boot and take control of the link, or the BGP session needs to start with proper policies. These control plane states of the link determine whether and how the switch's routing can be controlled.

We can manage the routing states of the switches when all the dependent states are correct. We represent the routing state in a data structure of the flow-link pairs, which is agnostic to the supported routing protocols. For example, the routing states can map to the routing rules in OpenFlow or the prefix-route announcement or withdrawal in BGP. When applications change the value of the routing state variable, Statesman (specifically the updater) automatically translates the value to appropriate control-plane commands.

One level higher is the *path* state which controls tunnels through multiple switches. Creating a tunnel and assigning traffic along the path depend on all switches on the path having their routing states ready to manage. Again, Statesman is responsible for translating the path's states into the routing states of all switches on the path, and the application only needs to read or write the path states.

4.2 Using and Extending the Model

For simplicity, applications should not be required to understand all the state variables and their complex dependencies. They should be able to simply work with the subset of variables that they need to manage based on their goals. For instance, a firmware-upgrade application should be able to focus on only the *DeviceFirmwareVersion* variable. However, this variable depends on lower-level variables such as switch power state whose impact cannot be completely ignored; firmware cannot be upgraded if the switch is down.

We find that it suffices to represent the impact of these dependencies by using a logical variable that we call *controllability* and exposing it to applications. This boolean-valued variable denotes whether the parent state variable is currently controllable, and its value is computed by Statesman based on lower-level dependencies. For instance, *DeviceFirmwareVersion* is controllable only if the values of variables such as switch's power and admin states are appropriate. Now, the firmware-upgrade application can simply work with *DeviceFirmwareVersion* and *DeviceFirmwareVersionIsControllable* variables to finish its job.

To give another concrete example, the variable *DeviceConfigIsControllable* tells whether applications can change various state variables of switch configuration, such as the management interface setup. The value of *DeviceConfigIsControllable* is calculated based on whether the switch is powered up, whether the switch can be reachable via SSH/Telnet from the management network (indicating the firmware is functioning), and whether the switch is healthy according to the health criterion (e.g., CPU/memory utilizations are not continuously 100% for certain amount of time). Similarly, links have *LinkAdminPowerIsControllable* calculated with the *DeviceConfigIsControllable* of the two switches on the link's ends.

Exposing just the controllability variables makes the dependency model extensible. The functions calculating the controllability are implemented in the storage service. When a new state variable is added to Statesman, we just need to place it in the dependency model, i.e., find what state variables will be dependent of the new one. Then we modify the controllability functions of corresponding state variables to consider its impact. For applications that are not interested in the new variable, no modifications are needed.

5. Checking Network State

The checker plays a pivotal role in Statesman. It needs to resolve conflicts among applications and enforce network-wide invariants. Its design also faces scalability challenges when handling large networks. We first explain the state checking process and then describe techniques for boosting scalability of the checker. Figure 5 outlines the checker's operation.

5.1 Resolving Conflicts

Multiple types of conflicts can occur in the network state due to the dynamic nature of DCNs and uncoordinated applications:

- **TS–OS.** The TS can conflict with the latest OS when changes in the network render some state variables uncontrollable, although they have new values in the TS. For instance, if the OpenFlow agent on a switch crashes, which is reflected as *DeviceAgentBootStatus=Down* in the OS, any routing changes in the TS cannot be applied.
- **PS–OS.** When the checker examines a PS, the OS may have changed from the time that the PS was generated, and some variables in the PS may not be controllable anymore. For example, a PS could contain the new value of a *LinkEndAddress* variable, but when the checker reads the PS, the link may have been shut down for maintenance.
- **PS–TS.** The TS is essentially the accumulation of all accepted PSes in the past. A PS can conflict with the TS due to an accepted PS from another application. For example, assume that a firmware-upgrade application wants to upgrade a switch and proposes to bring it down; its PS is accepted and merged into the TS. Now, when a TE application proposes to change the switch's routing state, it conflicts with the TS even though the switch is online (i.e., no conflict with the OS).

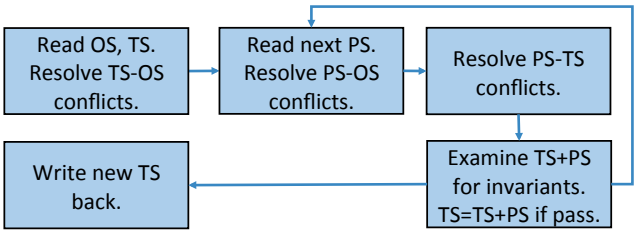

Figure 5: Flow of the checker's operation

The first two types of conflicts are because of the changing OS, which makes some variables in the PS or TS uncontrollable. To detect these conflicts, the checker reads the controllability values from the OS, which are calculated by the storage service based on the dependency model when relevant variable values are updated. It then locates the uncontrollable variables in the PS or TS. To resolve TS–OS conflicts, we introduce a flag called *SkipUpdate* for each variable in the TS. If set, the updater will bypass the network update of the state variable, thus temporarily ignoring the target value to resolve the conflict. The checker will clear the flag once the state variable is controllable again.

For uncontrollable state variables in a PS, the checker removes them from the PS, i.e., rejecting the part of PS that is inapplicable on the current network. The choice of partial rejection is a tradeoff between application's progress and potential side-effects of accepting a partial PS. The asynchrony between the application's and the checker's views of the OS is normal. By rejecting the whole PS due to a small fraction of conflicting variables, Statesman will be too conservative and will hinder application progress. We thus allow partial rejection. We have not yet observed any negative consequences in our deployment from this choice. In the future, we will extend the PS data structure such that applications can group variables. When a variable in a group is uncontrollable, the entire group is rejected, but other groups can still be accepted.

For PS–TS conflicts, which are caused by the conflicting application proposals, Statesman supports an extensible set of conflict resolution mechanisms. It currently offers two mechanisms. The basic one is last-writer-wins, in which the checker favors the value of state variable from the newer PS. The more advanced mechanism is priority-based locking. Statesman provides two levels of priorities of locks for each switch and link. Applications can acquire a low-priority or a high-priority lock before proposing a PS. In the presence of a lock, applications other than the lock holder cannot change the state variables of the switch or link; however, the high-priority lock can override the low-priority one. The choice of the conflict resolution mechanism is not system-wide and can be configured at the level of individual switches and links.

Although simple, these two conflict-resolution strategies have been proven sufficient for the applications that we have built and

deployed thus far. In fact, we find from our traces that last-writer-wins is good enough most of the time since it is rare for two applications to compete for the same state variable head-to-head. For our intra-datacenter infrastructure, we configure the checker with the last-writer-wins resolution; for our inter-datacenter network, we have enabled the priority-based locking. If needed, additional resolution strategies (e.g., based on application identities) can be easily added to Statesman.

5.2 Choosing and Checking Invariants

Network-wide invariants are intended to ensure the infrastructure's operational stability in the face of application bugs or undesired effects of collective actions of multiple applications. They should capture minimum safety and performance requirements, independent of which applications are currently running.

Choosing invariants: This choice must balance two criterion. First, the invariant should suffice to safeguard the basic operations of the network—as long as it is not violated, most services using the network would continue to function normally. Second, the invariant should not be so stringent that it interferes with application goals. For instance, an invariant that all switches should be operational is likely too strict and interferes with a firmware-upgrade application.

Following the criterion above, we currently use two invariants in the checker. Both are application-agnostic and relate to the topological properties of the network. The first invariant considers network connectivity. It requires that every pair of ToR switches in the same datacenter are (topologically) connected and that every ToR is connected to the border routers of its DCN (for WAN connectivity). Here, we ignore the routing configurations on the switches (which could be that two ToRs cannot communicate) because applications can intentionally partition a DCN at the routing level for multi-tenant isolation.

The second invariant considers network capacity. We define the capacity between two ToRs in the same datacenter as maximum possible traffic flow between them based on the network topology. The invariant is that the capacity between $p\%$ of ToR pairs should be at least $t\%$ of their baseline, when all links are functional. The values of p and t should be based on level of capacity redundancy in the DCN, tolerance of the hosted services to reduced capacity, and implications of blocking a management application that violates the invariant. We currently use $p = 99$ and $t = 50$, i.e., 99% of the ToR pairs must have at least 50% of the baseline capacity.

Although Statesman currently maintains only two invariants, the set of invariants is extensible. As explained below, the invariant checking is implemented as a boolean function over a graph data structure that has the network topology and all state variables. It is straightforward to add more invariants by implementing new functions with the same interface. For example, some networks may add an invariant that requires the absence of loops and blackholes, which can be checked using the routing states of the switches.

Checking invariants: When checking whether the TS obeys the invariants, the checker first creates a base network graph using variable values from the OS. Then, it translates the difference between a variable's observed and target values into an operation on the network graph, e.g., bringing a switch offline, changing the routing state to the target value, etc. Finally, the invariant checking functions are run with the new network graph.

The invariant checking is invoked in two places. The first is when the checker reads the TS from the storage service. The TS was compliant when written but can be in violation when read due to changes in the network (i.e., the OS). While running the invariant checking as described above, the checker clears or sets the *SkipUpdate* flags in the TS respective of its compliance status.

The second place is when the checker tries to merge a PS into the TS after conflict resolution. The checker analyzes TS+PS, as if the PS was merged into the TS. If TS+PS passes the check, the PS is accepted and merged into the TS. Otherwise, the PS is rejected.

The acceptance or rejection (and reasons) for each PS is recorded by the checker into the storage service. We categorize the rejection reasons into three groups: state variable became uncontrollable; invariant was violated; and TS was unreachable. The reasons are encoded as machine-readable status code as values of a special state variable called *ProposalStatus* for each PS. Applications use the same interface as for reading the OS to read their PSes' status. They can then react appropriately (e.g., generate a new PS that resolves the conflicts with the latest OS).

5.3 Partitioning by Impact Group

The checker needs to read all the state variables in the OS and TS to examine conflicts and invariants every round of operation. The number of such variables across our infrastructure—the datacenters and the WAN that connects them—poses a scalability threat.

To help scale, we observe that the impact of state changes for a given switch or link is limited. For instance, changes to an aggregation switch in one datacenter do not affect the connectivity and capacity of ToRs in another datacenter. Similarly, changes to border routers in a datacenter do not impact the capacity and connectivity within the datacenter, though they do impact other datacenters' WAN reachability.

Based on this observation, we partition the checker's responsibility into multiple impact groups. We set up one impact group per datacenter and one additional group with border routers of all datacenters and the WAN links. These groups are independent with respect to the state checking process. In our deployment of Statesman, we run one instance of checker per impact group, which has enabled the state checking to scale (§8).

6. System Design and Implementation

We now describe in more details the design and implementation of Statesman. We start with the storage service, followed by the updater and monitor; we skip the checker as it was covered in details in §5. Figure 6 provides a closer look at various components in a Statesman deployment. Our implementation of Statesman has roughly 50 thousand lines of C# and C++ code, plus a number of internal libraries. At its core, it is a highly-available RESTful web service with persistent storage. Below, we also describe Statesman's read-write APIs.

6.1 Globally Available and Distributed Storage Service

The storage service needs to persistently and consistently store the network states. We thus use a Paxos-based distributed file system. However, two challenges prevent us from using a single big Paxos ring to maintain all the states for all our datacenters. The first is the datacenter reachability. Due to WAN failures, one datacenter may lose connectivity to all others, or two datacenters may not be able to reach each other. To protect Statesman from such failures, the storage instances need to be located in multiple datacenters.

A second challenge stems from the volume of state data. In DCNs, there are hundreds of thousands of switches and links, each

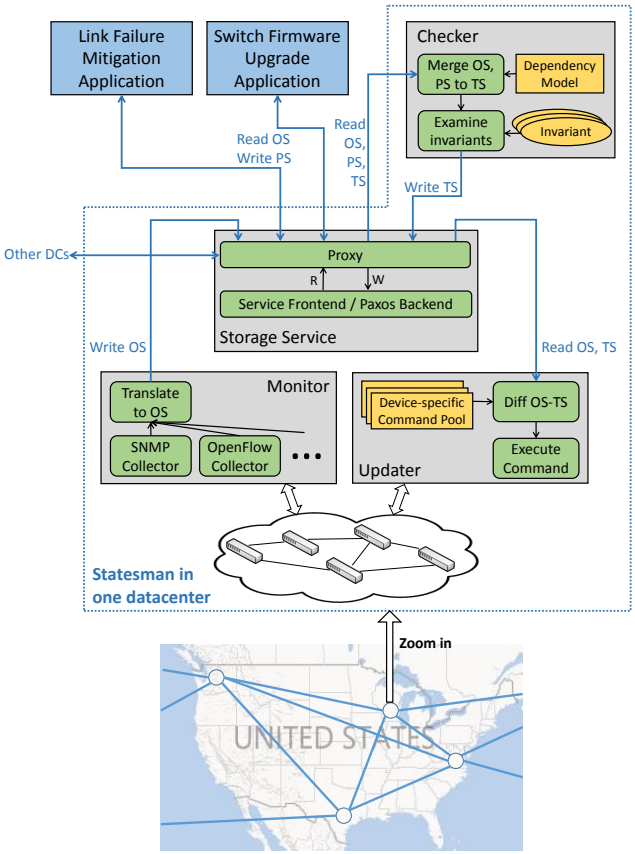

Figure 6: Statesman system design

with dozens of state variables. This scale results in millions of state variables (§8). Manipulating all variables in a single Paxos ring would impose a heavy message-exchange load on the file system to reach consensus over the data value. This impact worsens if the exchange happens over the WAN (as storage instances are located in multiple datacenters for reliability). WAN latencies will hurt the scalability and performance of Statesman.

Therefore, we break a big Paxos ring into independent smaller rings for each datacenter. One Paxos ring of storage instances is located in each datacenter, and it only stores the state data of the switches and links in that datacenter. In this way, Statesman reduces the scale of state storage to individual datacenter, and it lowers the impact of Paxos consensus by limiting message exchanges inside the datacenter.

Although the underlying storage is partitioned and distributed, we still want to provide a uniform and highly available interface for the applications and other Statesman components. These users of the storage service should not be required to understand where and how various state variables are stored.

We thus deploy a globally available proxy layer that provides uniform access to the network states. Users read or write network states of any datacenters from any proxy without knowing the exact locations of the storage service. Inside the proxy, we maintain an in-memory hash table of the switch and link names to corresponding datacenters for distributing the requests across the storage-service instances. The proxy instances sit behind a load balancer, which enables high availability and flexible capacity.

6.2 Stateless Update on Heterogeneous Devices

Network update is a challenging problem itself [12, 20, 28]. In the context of managing a large network, it becomes even more challenging for three reasons. First, the update process is device- and protocol-specific. Although OpenFlow provides a standard interface for changing the forwarding behaviors of switches, there is no standard interface today for management-related tasks such as changing the switch power, firmware, or interface configuration. Second, because of scale and dynamism, network failures during updates are inevitable. Finally, the device's response can be slow and dominate the application's control loop. Two aspects of the design of the Statesman updater help to address these challenges.

Command template pool for heterogeneous devices: The changes from applications (i.e., PSes) are device-agnostic network states. The updater translates the difference between a state variable's OS and TS values into device-specific commands. This translation is done using a pool of command templates that contains templates for each update action on each device model with supported control-plane protocol (e.g., OpenFlow or vendor-specific API). When the updater carries out an update action, it looks up the template from the pool based on the desired action and device details.

For instance, suppose we want to change the value of a switch's *DeviceRoutingState*. If the switch is an OpenFlow-enabled model, the updater looks up this model's OpenFlow command template to translate the routing state change into the insertion or deletion of OpenFlow rules, and issues rule update commands to the OpenFlow agent on the switch. Alternatively, if the switch is running a traditional routing protocol like BGP, the updater looks up the BGP command template to translate the routing state change into the BGP-route announcement or withdrawal.

Stateless and automatic failure handling: With all network states persistently stored by the storage service, the updater can be stateless and simply read the new values of OS and TS every round. This mode of operation makes the updater robust to failures in the network or in the updater itself. It can handle failures with an implicit and automatic retry. When any failure happens in one run of update, the state changes resulted by the failure reflect as a changed OS in the storage service. In the next run, the updater picks up the new OS which already includes the failure's impact, and it calculates new commands based on the new OS-TS difference. In this manner, the updater always brings the latest OS towards the TS, no matter what failures have happened in the process.

Being stateless also means that we can run as many updater instances as needed to scale, as long as we are able to coherently partition the work among them. In our current deployment, we run one instance per state variable per switch model. In this way, each updater instance is specialized for one task.

6.3 Network Monitors

The monitors collect the network states with various protocols from the switches, including SNMP and OpenFlow. The monitors then translate the protocol-specific data into the value of corresponding state variables, and write them into the storage service as the OS. We split the monitoring responsibility across many monitor instances, so each instance covers roughly 1,000 switches.

Currently the monitors run periodically to collect all switches' power states, firmware versions, device configurations, and various counters (and routing states for a subset of switches). For links, our monitors cover the link power, configuration, and counters like the packet drop rate and the traffic load.

GET	NetworkState/Read?Datacenter={*dc*}&Pool={*p*} &Freshness={*c*}&Entity={*e*}&Attribute={*a*}
POST	NetworkState/Write?Pool={*p*} (Body is list of NetworkState objects in JSON)

(a) HTTP Request

Datacenter	*dc*	Datacenter name
Pool	*p*	OS, PS, or TS
Freshness	*c*	Up-to-date or bounded-stale
Entity	*e*	Entity name (i.e., switch, link, or path)
Attribute	*a*	State variable name

(b) Parameters

Table 3: Read-write APIs of Statesman

6.4 Read-Write APIs

The storage service is implemented as a HTTP web service with RESTful APIs. The applications, monitors, updaters, and checkers use the APIs to read or write *NetworkState* objects from the storage service. A *NetworkState* object consists of the entity name (i.e., the switch, link, or path name), the state variable name, the variable value, and the last-update timestamp. The read-write APIs of Statesman are shown in Table 3.

There is a *freshness* parameter in the read API because Statesman offers different freshness modes for reading the network states. The up-to-date mode is for applications who are strict with the state staleness. For instance, the link-failure-mitigation application needs to detect link failures as soon as possible when the failures happen. Statesman also offers 5-minute bounded-staleness mode by reading from caches [27]. Many management applications do not need the most up-to-date network states and can safely tolerate some staleness in state data. For instance, the firmware-upgrade application needs to upgrade the switches within hours; it does not matter if the value of *DeviceFirmwareVersion* is slightly stale. By allowing such applications to read from caches, we boost the read throughput of Statesman. At the same time, applications that cannot tolerate staleness can use the up-to-date freshness mode.

7. Application Experiences

In this section, we present our experiences of running Statesman in Microsoft Azure. Statesman is now deployed in ten datacenters (DCs), and we have built two production and one pilot-stage applications on top of it. We first describe this deployment, and then use three representative scenarios to illustrate how Statesman facilitates the operations of management applications.

7.1 Statesman Deployment

Statesman currently manages ten geographically-distributed DCs of Microsoft Azure, covering all the switches and links in those DCs and the WAN connecting the DCs. The three applications that we have built leverage Statesman to manage different aspects (e.g., switches, links, and traffic flows) of our DCN.

- **Switch upgrade**: When a new version of firmware is released by a switch vendor, this application automatically schedules all the switches from the same vendor to upgrade by proposing a new value of *DeviceFirmwareVersion*. In the upgrade process, the checker of Statesman ensures that the DCN continues to meet the network-wide invariants.

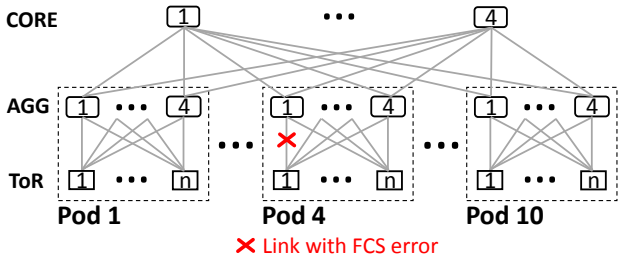

Figure 7: Network topology for the scenario in §7.2

- **Failure mitigation**: This application periodically reads the Frame-Check-Sequence (FCS) error rates on all the links. When detecting persistently high FCS error rates on certain links, it changes the *LinkAdminPower* state to shut down those faulty links to mitigate the impact of the failures [30]. The application also initiates an out-of-band repair process for those links, e.g., by creating a repair ticket for the on-site team.
- **Inter-DC TE**: As described in SWAN [12], Statesman collects the bandwidth demands from the *bandwidth brokers* sitting with the hosted services. In addition, the monitor of Statesman collects all the forwarding states, such as tunnel status and flow matching rules. Given this information, the TE application computes and proposes new forwarding states, which are then pushed to all the relevant routers by the Statesman updater.

The first two applications have been fully tested and deployed, while the third one is in pilot stage and undergoing testing.

We emphasize two issues on how applications interact with Statesman. First, they should understand that it takes time for Statesman to read and write a large amount of network states in large DCNs. Thus, their control loops should operate at the time scale of minutes, not seconds. Second, the applications should understand that their PSes may be rejected due to failures, conflicts, or invariant violations. Thus, they need to run iteratively to adapt to the latest OS and the acceptance or rejection of their previous PSes.

7.2 Maintaining Network-wide Invariants

We use a production trace to illustrate how Statesman helps two applications (switch-upgrade and failure-mitigation) safely coexist in intra-DC networks while maintaining the capacity invariant—99% of the ToR pairs in the DC should have at least 50% of their baseline capacity.

Figure 7 shows the topology for the scenario. It is a portion of one DC with 10 pods, where each pod has 4 Agg switches and a number of ToRs. The switch-upgrade application wants to upgrade all the 40 Aggs in a short amount of time. Specifically, it will upgrade the pods one by one. Within each pod, it will attempt to upgrade multiple Aggs in parallel by continuing to write a PS for one Agg upgrade until it gets rejected by Statesman.

In Figure 8, we pick one ToR from each pod, and organize the 10 ToRs from the 10 pods into 90 ToR pairs (10×9, excluding the originating ToR itself). On the Y-axis, we put the 9 ToR pairs originating from the same ToR/pod together. Essentially, Figure 8 shows how the network capacity between each ToR pair changes over time. Boxes A, B, and C illustrate how the network capacity temporarily drops when the switch-upgrade application upgrades the Aggs in Pod 1, 2, and 3 sequentially. To meet the 50%-capacity invariant, the switch-upgrade application can simultaneously upgrade up to 2 out of 4 Aggs, leaving at least 2 Aggs alive for traffic.

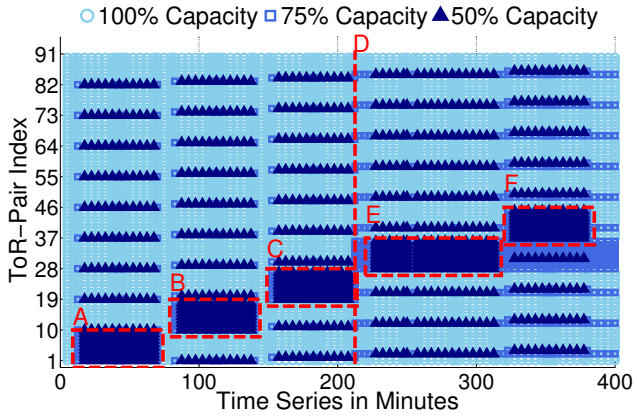

Figure 8: Illustration of how the switch-upgrade and failure-mitigation applications interact through the checker. Y-axis: One ToR from each one of 10 pods to form directional ToR pairs, indexed by the originating ToR/Pod. A, B, C: The switch-upgrade application upgrades Pod 1, 2, 3 normally while the checker maintains an invariant that each ToR pair has at least 50% of baseline capacity. D: The failure-mitigation application detects a failing link ToR_1-Agg_1 in Pod 4 and shuts the link down. E: Due to the down link, upgrade of Pod 4 is automatically slowed down by the checker to maintain the capacity invariant. F: Upgrade of Pod 5 resumes the normal pattern.

During the upgrade, the failure-mitigation application discovers persistently high FCS error rate on link ToR_1-Agg_1 in Pod_4. As a result, it shuts down this link at time D. Since one ToR-Agg link is down, the capacity of all Pod_4-related ToR pairs drops to 75%, which originate from Pod_4 (index # 28–36) or end at Pod_4 (index # 3, 12, 21, 40, 49, 58, 67, 76, & 85). When the switch-upgrade application starts to work on Pod_4, Statesman can only allow it to upgrade one Agg at a time to maintain the 50%-capacity invariant. Thus, as shown in box E, the switch-upgrade application automatically slows down when upgrading Pod_4. Its actual upgrade steps are Agg_1-Agg_2-together, then Agg_3, and finally Agg_4. Note that Agg_1 and Agg_2 can be upgraded in parallel, because link ToR_1-Agg_1 is already down and hence upgrading Agg_1 does not further reduce the ToR-pair capacity. The switch-upgrade application resumes normal speed when it upgrades Pod_5 in box F.

7.3 Resolving Application Conflicts

The inter-DC TE application is responsible for allocating inter-DC traffic along different WAN paths. Figure 9 shows the pilot WAN topology used in this experiment. This WAN inter-connects four DCs in a full mesh, and each DC has two border routers. The switch-upgrade application is also running on the WAN.

One recurring scenario is that we need to upgrade all the border routers while inter-DC traffic is on. This can lead to a conflict between the two applications: the switch-upgrade application wants to reboot a router for firmware upgrade, while the TE application wants the router to carry traffic. Without Statesman, the operators of the two applications have to manually coordinate, e.g., by setting up a maintenance window. During this window, the operators must carefully watch the upgrade process for any unexpected events.

With Statesman, the whole process becomes much simpler. When there is no upgrade, the TE application acquires the low-priority lock over each router, and changes the forwarding states as needed.

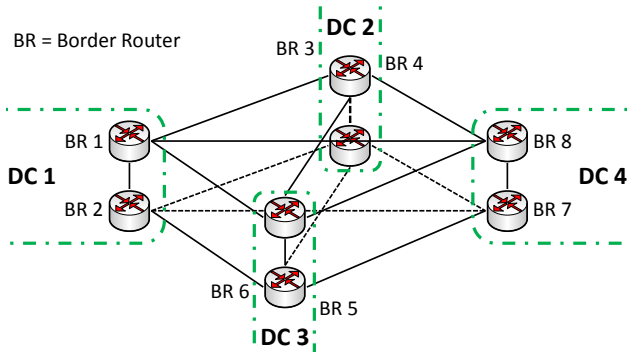

Figure 9: WAN topology for the scenario in §7.3

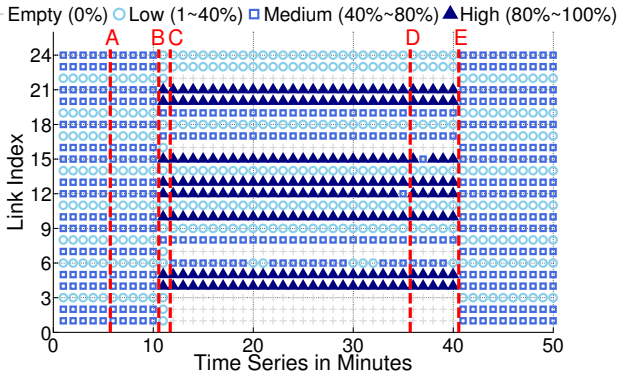

Figure 10: Illustration of how Statesman resolves conflicts between the inter-DC TE and switch-upgrade applications. A: The switch-upgrade application acquires the high-priority lock of BorderRouter$_1$ (BR_1). B: The TE application fails to acquire the low-priority lock and moves traffic away from BR_1 to other links. C: The switch-upgrade application starts to upgrade BR_1 since traffic is zero. D: Upgrade of BR_1 is done. The switch-upgrade application releases the high-priority lock. E: The TE application re-acquires the low-priority lock of BR_1 and moves traffic back.

When the switch-upgrade application wants to upgrade a router, it first acquires the high-priority lock over that router. Soon after, the TE application realizes that it cannot acquire a low-priority lock over the router, and it starts to shift traffic away from that router. Meanwhile, the switch-upgrade application keeps reading the traffic load of the locked router until the load drops to zero. At this moment, it kicks off the upgrade by writing a PS with a new value of *DeviceFirmwareVersion*. Once the upgrade is done, the switch-upgrade application releases the high-priority lock of the router, and proceeds to the next candidate.

We collected the traffic load data during one upgrade event in off-peak hours. Since the load patterns of different routers are similar, we only illustrate the upgrade process of BorderRouter$_1$ (BR_1). Figure 10 shows the time series of the link load (note that both applications run every 5 minutes). The Y-axis shows the 24 links (12 physical links $\times$ 2 directions) indexed by the originating router of each link. At time B, the TE application fails to acquire the low-priority lock over BR_1, since the high-priority lock of BR_1 was acquired by the switch-upgrade application at time A. So the TE application moves traffic away from BR_1. At time C, the load drops

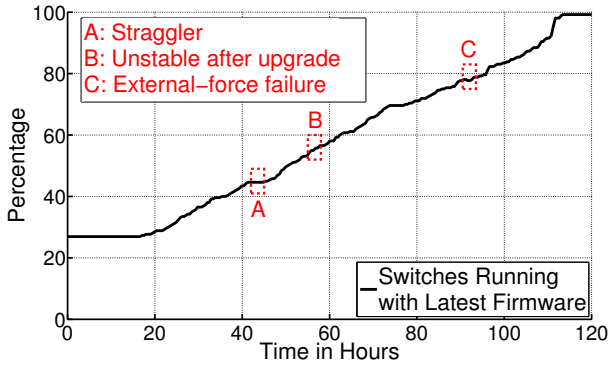

Figure 11: Time series of firmware upgrade at scale

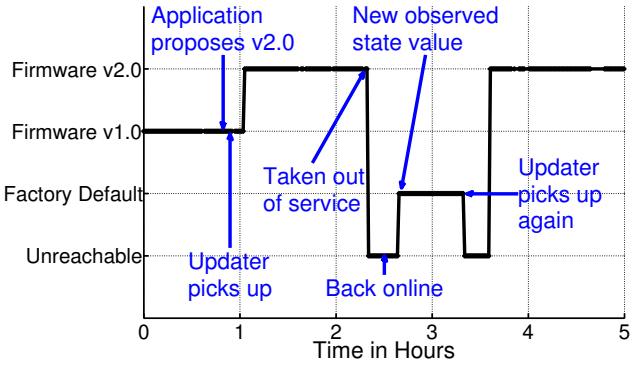

Figure 12: External-force failure in firmware upgrade

to zero on all the links originating from BR_1 (index # 1, 2, & 3) and ending at BR_1 (index # 7, 16, & 22). As expected, this increases the loads on the other links. After the switch-upgrade application finishes upgrading BR_1 and releases the high-priority lock at time D, the TE application successfully acquires the low-priority lock again at time E, and then moves traffic back to BR_1.

In this example, neither of the two applications needs to be aware of the other since conflict resolution and necessary coordination are automatically enabled by Statesman. We use locking as the conflict resolution strategy in the inter-DC case. So the TE application can move tunnels away from switches being upgraded before the upgrade process starts, rather than after (which would be the case with conflict resolution based on last-writer-wins). In the intra-DC case, we do not use tunnel-based TE, and neighboring routers can immediately reroute traffic when a switch is brought down for upgrade without warning.

7.4 Handling Operational Failures

In Statesman, an application outputs a PS instead of directly interacting with the network devices. Once the PS is accepted into the TS, Statesman takes the responsibility of (eventually) moving the network state to the TS. This simplifies the application design, especially in failure handling.

In this example, we show how the switch-upgrade application rolls out a new firmware to roughly 250 switches in several DCs in two stages. In the first stage, 25% of the switches are upgraded to test the new firmware. Once we are confident with the new firmware, the remaining 75% of the switches are upgraded in the second

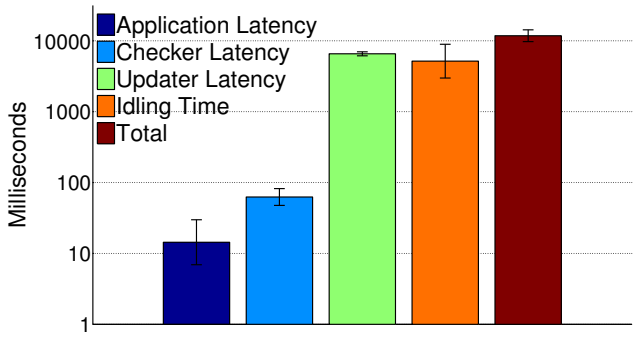

Figure 13: End-to-end latency breakdown

stage. Figure 11 illustrates the upgrade process, where the percentage of switches with new firmware gradually grows to 100% in about 100 hours. Here, the switch-upgrade application runs conservatively and upgrades one switch at a time.

During this process, Statesman automatically overcomes a variety of failures, which are highlighted in Figure 11. In box A, there is a straggling switch which takes 4 hours to upgrade; note the flat line in the figure. During the 4 hours, Statesman repeatedly tries to upgrade until it succeeds. This straggling happens because the switch has a full memory and cannot download the new firmware image. After some of the switch memory is freed up, the upgrade proceeds. In box B, a few switches become unstable after the upgrade, e.g., frequently flipping between on and off. This appears as a stair-shape line in Figure 11. Statesman automatically retries until the switches are stable with the new firmware.

Box C shows a failure case which involves human interventions. After a switch is upgraded, the operators take it out of service, and manually reset it to the factory-default firmware before returning it to production. This action causes the time-series line to slightly drop in Box C, and we zoom in the process in Figure 12. Once the switch is back to production, Statesman finds that the observed *DeviceFirmwareVersion* of one switch is the factory-default one. Since the firmware of the switch has been set to a new value in the TS, Statesman automatically redoes the firmware upgrade without any involvement from the switch-upgrade application.

8. System Evaluation

In this section, we quantify the latency and coverage of Statesman as well as the performance of checking, reading and writing network states.

End-to-end latency: Figure 13 shows Statesman's end-to-end latency measured by running the failure-mitigation application on a subset of our DCN. We manually introduce packet drops on a link, and let the failure-mitigation application discover the problem and then shut down the link. The end-to-end latency is measured from when the application reads the high packet drop rate of the link to when the faulty link is actually deactivated in the network. We break down the end-to-end latency into four portions:

- Application latency: from when the application reads the high packet drop rate to when it writes a PS for shutting down the faulty link.
- Checker latency: from when the checker reads the PS to when it finishes checking and writes a new TS.
- Updater latency: from when the updater reads the new TS to when it finishes shutting down the faulty link.

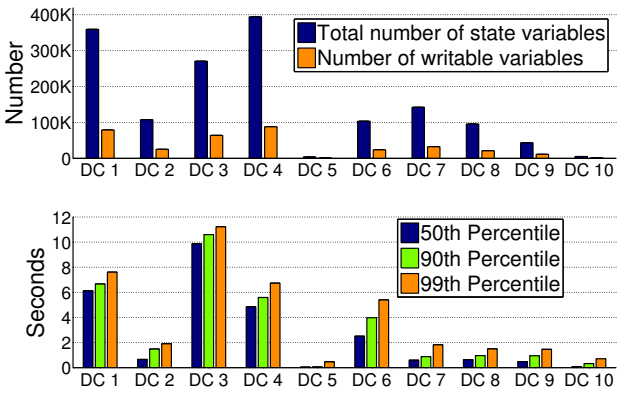

Figure 14: Network state scale & checker performance

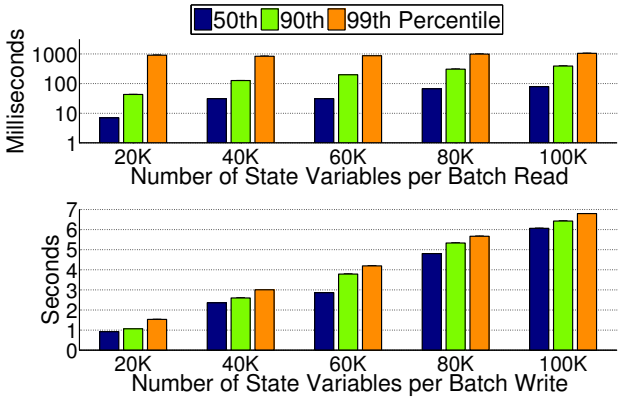

Figure 15: Read-write micro-benchmark performance

- Idling time: cumulative time when no one is running. It exists because the application, checker and updater run asynchronously (every 5 seconds in this experiment).

Figure 13 shows that the application and checker latencies are negligibly small, accounting for only 0.13% and 0.28% of the end-to-end latency on average. The main bottleneck is the time for the updater to apply the link-shutdown commands to the switch, which accounts for 57.7% of the end-to-end latency on average.

Coverage: Figure 14 shows the deployment scale and the checker latency across the ten DCs where Statesman is deployed. The number of state variables in each DC indicates its size. The largest DC (DC_4) has 394K variables and 7 out of 10 DCs have over 100K variables. The total number of variables managed by Statesman is currently 1.5 million and it continues to grow as Statesman expands to more DCs. Since only 23.4% of the variables are writable on average, the size of TS is correspondingly smaller than the OS.

Checker latency: Figure 14 also shows that one round of checking takes 0.5 to 7.6 seconds in most DCs. In the most complex DC (DC_3), the 99th-percentile of checking latency is 11.2 seconds.

Read-write performance: We stress-test the read-write performance of Statesman by randomly reading and writing varying numbers of state variables. Figure 15 shows that the 99th-percentile latency of reading 20K to 100K variables always stays under a second. Since the applications rarely read more than 100K variables at a time, Statesman's read speed is fast enough.

Figure 15 also shows that the write latency of Statesman grows linearly with the number of variables, and the 99th-percentile latency of writing 100K variables is under seven seconds. Since our largest DC has fewer than 100K variables in the TS, it takes less than 10 seconds to write the entire TS. In practice, the state-variable writers, e.g., applications, monitors, and checkers, rarely write more than 10K variables at a time. Hence, the write latency is usually around 500ms in our normal operations.

9. Related Work

Statesman descends from a long line of prior works on software-defined networking [1, 2, 3, 6, 8, 9, 22]. These works enable centralized control of traffic flow by directly manipulating the forwarding states of switches. In contrast, Statesman supports a wider range of network management functions (e.g., switch upgrade, link failure mitigation, elastic scaling, etc.).

Similar to Statesman, Onix [18, 19] and Hercules [16] provide a shared network-state platform for all applications. But these systems neither resolve application conflicts, in particular those caused by state variable dependency, nor enforce network-wide invariants.

Pyretic [24], PANE [7], and Maple [29] are recent proposals on making multiple applications coexist. Their target is again limited to traffic management applications. Their composition strategies are specific to such applications and do not generalize to the broader class of applications that we target at. Mesos [11] schedules competing applications using the cluster-resource abstraction, which is quite different from our network-state abstraction (e.g., no cross-variable dependency).

Corybantic [23] proposes a different way of resolving conflicts by letting each application evaluate other applications' proposals. As noted previously, such tight cross-application coordination, while sometimes beneficial, imposes enormous complexity on the application design and testing.

Another approach of hosting multiple applications is to partition the network into multiple isolated virtual slices as described in [17, 26]. Compared to the virtual topology model, our network-state model is more fine-grained and flexible. It allows multiple applications to manage different levels of states (e.g., power, configuration, and routing) on the same device.

There are also earlier works on invariant verification [14, 15, 21] for the network's forwarding state. In the future, we may incorporate some of these invariant checking algorithms into Statesman.

10. Conclusion

Statesman enables multiple loosely-coupled network management applications to run on a shared network infrastructure, while preserving network safety and performance. It safely composes uncoordinated and sometimes-conflicting application actions using three distinct views of network state, inspired by version control systems, and a model of dependencies among different parts of network state. Statesman is currently running in ten Microsoft Azure datacenters, along with three diverse applications.

Acknowledgments: We thank the Microsoft Azure Networking team, especially Murat Acikgoz, Jiaxin Cao, George Chen, Huoping Chen, Kamil Cudnik, Umesh Krishnaswamy, Guohan Lu, Shikhar Suri, Dong Xiang, and Chao Zhang, for helping develop Statesman and being early adopters. We also thank Chuanxiong Guo, Randy Kern, Dave Maltz, our shepherd Walter Willinger, and the SIGCOMM reviewers for feedback on the paper.

11. References

[1] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and Implementation of a Routing Control Platform. In *USENIX NSDI*, May 2005.

[2] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. *ACM SIGCOMM CCR*, 37(4):1–12, August 2007.

[3] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *USENIX Security Symposium*, July 2006.

[4] C.-C. Chen, P. Sun, L. Yuan, D. A. Maltz, C.-N. Chuah, and P. Mohapatra. SWiM: Switch Manager For Data Center Networks. *IEEE Internet Computing*, April 2014.

[5] K. Chen, C. Guo, H. Wu, J. Yuan, Z. Feng, Y. Chen, S. Lu, and W. Wu. Generic and Automatic Address Configuration for Data Center Networks. In *ACM SIGCOMM*, August 2010.

[6] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN. *ACM Queue*, 11(12):20:20–20:40, December 2013.

[7] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *ACM SIGCOMM*, August 2013.

[8] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR*, 35(5):41–54, October 2005.

[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM CCR*, 38(3):105–110, July 2008.

[10] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving Energy in Data Center Networks. In *USENIX NSDI*, April 2010.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *USENIX NSDI*, March 2011.

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

[13] 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 *ACM SIGCOMM*, August 2013.

[14] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*, April 2012.

[15] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *USENIX NSDI*, April 2013.

[16] W. Kim and P. Sharma. Hercules: Integrated Control Framework for Datacenter Traffic Management. In *IEEE Network Operations and Management Symposium*, April 2012.

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

[18] 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 *USENIX OSDI*, Vancouver, BC, Canada, October 2010.

[19] B. Lantz, B. O'Connor, J. Hart, P. Berde, P. Radoslavov, M. Kobayashi, T. Koide, Y. Higuchi, M. Gerola, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *ACM SIGCOMM HotSDN Workshop*, August 2014.

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

[21] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *ACM SIGCOMM*, August 2011.

[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR*, 38(2):69–74, March 2008.

[23] J. Mogul, A. AuYoung, S. Banerjee, J. Lee, J. Mudigonda, L. Popa, P. Sharma, and Y. Turner. Corybantic: Towards Modular Composition of SDN Control Programs. In *ACM HotNets Workshop*, November 2013.

[24] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-defined Networks. In *USENIX NSDI*, April 2013.

[25] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud Scale Load Balancing. In *ACM SIGCOMM*, August 2013.

[26] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *USENIX OSDI*, October 2010.

[27] D. Terry. Replicated Data Consistency Explained Through Baseball. Technical Report MSR-TR-2011-137, Microsoft Research, 2011.

[28] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Seamless Network-wide IGP Migrations. In *ACM SIGCOMM*, August 2011.

[29] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *ACM SIGCOMM*, August 2013.

[30] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. In *ACM SIGCOMM*, August 2012.