Safe and Flexible Controller Upgrades for SDNs

Karla Saur
Intel Labs*
karla.saur@intel.com

Joseph Collard
UMass Amherst
jcollard@cs.umass.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.edu

Arjun Guha
UMass Amherst
arjun@cs.umass.edu

Laurent Vanbever
ETH Zurich
lvanbever@ethz.ch

Michael Hicks
University of Maryland
mwh@cs.umd.edu

ABSTRACT

SDN controllers must be periodically upgraded to add features, improve performance, and fix bugs, but current techniques for implementing *dynamic updates*—i.e., without disrupting ongoing network functions—are inadequate. Simply halting the old controller and bringing up the new one can cause state to be lost, leading to incorrect behavior. For example, if the state represents flows blacklisted by a firewall, then traffic that should be blocked may be allowed to pass through. Techniques based on record and replay can reconstruct controller state automatically, but they are expensive to deploy and do not work in all scenarios.

This paper presents a new approach to implementing dynamic updates for SDN controllers. We present the design and implementation of a new controller platform called Morpheus that uses *explicit state transfer* to implement dynamic updates. Morpheus enables programmers to directly initialize the upgraded controller's state as a function of its existing state, using a domain-specific language that is designed to be easy to use. Morpheus also offers a distributed protocol for safely deploying updates across multiple nodes. Experiments confirm that Morpheus provides correct behavior and good performance.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; C.2.3 [**Network Operations**]: Network management

Keywords

Software-Defined Network; Dynamic Software Updating

1. INTRODUCTION

SDN controllers are complex software systems that must simultaneously implement a range of interacting services including discovery, routing, monitoring, load balancing, authentication, access control, and others. Like any large software system, SDN controllers must be periodically upgraded to add features, improve performance, and fix bugs. However, in most networks any downtime is unacceptable, so controller upgrades must be deployed *dynamically*, while the network is running and in a way that minimizes disruptions.

Dynamic upgrades differ from static upgrades in that while modifying the *program code* they must also be concerned with the current *execution state*. In SDN, this state can be divided into the *internal state* stored on controllers (e.g., in memory, file systems, or databases) and the *external state* stored on switches (e.g., in forwarding rules). A key challenge is that upgraded code may make different assumptions about state—e.g., using different data structures on the controller or installing different rules on switches.

Existing approaches. SDN controllers today typically employ one of three strategies for performing controller upgrades, distinguished by how they attempt to ensure correct, post-upgrade execution state.

- In *simple restart*, the default on open-source SDN platforms such as POX [5] and Floodlight [2], the system halts the old controller and begins executing a fresh copy of the new controller. Any existing internal state not explicitly designed to be persistent is lost, so for consistency, rules on the switches are wiped at startup, precipitating state reconstruction.
- In *record and replay*, provided by HotSwap [34], OpenNF [15] and related systems [31], the system maintains a trace of network events received by the old controller. During an upgrade, the system first replays the logged events to the new controller to "warm up" its internal state and then swaps in the new controller for the old one, leaving existing switch state in place.
- In *rule-sourced reconstruction*, provided by some industrial controllers, the new controller's internal state is initialized based on service-specific code that first queries the current rules on switches, which provide clues about active flows, configurations, etc.

Unfortunately, none of these approaches constitutes a complete general-purpose solution to the controller upgrade problem. Simple restarts discard internal state, and wiping rules to regenerate it floods the new controller with packet-in events, leading to performance disruptions. In addition,

*Work performed while at the University of Maryland.

SOSR '16, March 14-15, 2016, Santa Clara, CA, USA

DOI: http://dx.doi.org/10.1145/2890955.2890966

simple restarts offer no guarantees that the reconstructed controller state will be harmonious with assumptions being made by end hosts—e.g., existing traffic flows may be dropped or misrouted. Record and replay can reproduce a harmonious controller state in some cases, but it requires a complex logging system that can be expensive to run. Moreover, record and replay goes wrong in cases where the new controller would have induced a different set of events than the old controller did. Reconstructing controller state from forwarding rules is laborious and error prone, and it works only when all necessary controller state is encoded into switch rules. It is also risky in the face of inevitable switch failures. Section 2 discusses the limitations of current approaches in more detail.

Morpheus: Controller upgrades by state transfer. This paper proposes a more general and flexible solution to the controller upgrade problem that we call *controller upgrade by state transfer*. We have implemented this approach in a new controller platform called Morpheus, which is described in detail in Section 3.

Upgrade by state transfer has three key ingredients:

- First, in contrast with existing techniques, which *indirectly* reconstruct controller state, Morpheus provides programmers with *direct* access. This functionality is enabled by storing critical internal state in a *network information base* (NIB). Morpheus's design is modular, with separate modules implemented by processes providing distinct services (like forwarding, traffic shaping, etc.), each of which can coordinate and share information via the NIB (like network performance data or information about active flows).[1] Importantly, the NIB is itself a separate module that persists between upgrades; when we upgrade a module, then the old version's state is still available in the NIB for subsequent use (§3).

- Second, Morpheus provides a way to easily *transform* the old controller state to make it consistent with the new controller, ensuring that it is harmonious with the network—e.g., preserving active flows. We expect the programmer of an upgrade (or perhaps the operator applying it) to write any necessary transformations. To make this easier we define a domain-specific language (DSL) for doing so (§4.2). In our experience, and based on our investigation of in-the-wild controller upgrades, writing such transformations is often straightforward (§5.4).

- Third, Morpheus provides a protocol for *coordinating* the deployment of controller upgrades. Because multiple modules may share access to the same NIB state (e.g., a topology discovery module and a routing module might both read/write network graph and performance data), changes to one module might affect the state used by the other, necessitating an upgrade. Hence, one must be careful that state upgrades take effect in an orderly fashion, so that new code does not access old state and vice versa. For this purpose we have designed a simple three-phase protocol that *quiesces* the affected modules, *installs* the relevant transformation at the NIB, and then *reloads* the modules at their new versions (§4.1).

Upgrade by state transfer directly addresses the performance and correctness problems of prior approaches. There is no need to log events, and there is no need to process many events at the controller, whether by replaying old events or by inducing the delivery of new ones by wiping rules. Moreover, state transfer gives the operator complete control over the post-upgrade network state, affording greater flexibility and ease of programming. Finally, Morpheus's modular design should allow its techniques to scale to even more distributed architectures.

Using Morpheus we have written several modules, and several versions of each, including a stateful firewall, topology discovery, routing, and load balancing. Through a series of experiments we demonstrate the advantages of upgrade by state transfer, compared to simple restarts and record-and-replay: there is far less disruption and no incorrect behavior. We also confirm that the state transformer functions are relatively simple to write.

Summary. This paper's contributions are as follows:

- We study the problem of performing dynamic upgrades to SDN controllers and identify fundamental limitations of current approaches.
- We propose a new, general-purpose solution to the dynamic upgrade problem for SDN controllers—*controller upgrade by state transfer*. With this solution, the programmer explicitly transforms old state to be used with the new controller, and an accompanying protocol coordinates the upgrade across distributed nodes.
- We describe a prototype implementation of these ideas in the new Morpheus controller.
- We present several controller evolutions as case studies as well as experiments showing that Morpheus implements upgrades correctly and with far less disruption than current approaches.

Next, we present the controller upgrade problem in detail (§2), the design and implementation of Morpheus (§3-4), our experimental evaluation (§5), and a discussion of related work and conclusion (§6-7).

2. OVERVIEW

This section explains why existing approaches for handling dynamic upgrades to SDN controllers are inadequate in general, and provides detailed motivation for our approach based on *state transfer*.

2.1 Simple Restart

In *simple restart*, the system halts the old controller and begins executing a fresh copy of the new controller, thereby discarding the old controller's internal state. While this approach can work for proactive controllers (assuming update primitives are used [27]), in general, restarting the controller can lead to incorrect handling of existing flows.

[1]This design is based on the distributed architectures used in industrial controllers such as Onix [21] and ONOS [10]. We considered retrofitting a simpler open-source controller such as POX or Floodlight [5, 2], but decided to build a new controller to explore issues related to upgrades in controllers in distributed settings.

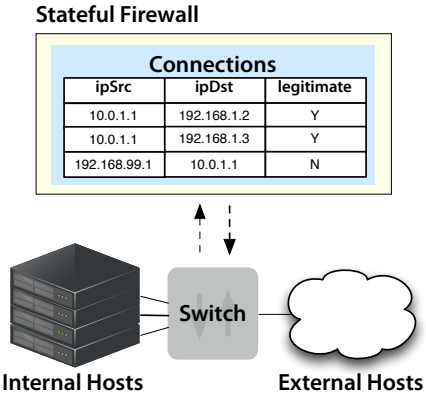

Figure 1: Example network service: stateful firewall.

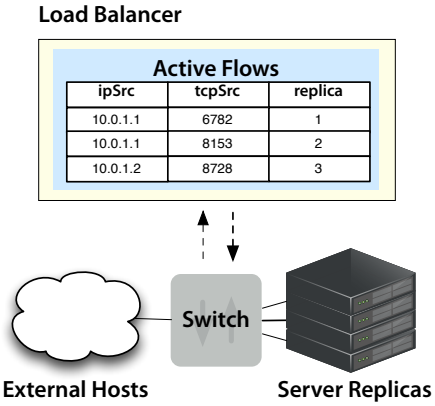

Figure 2: Example network service: load balancer.

Example service: Stateful Firewall. Suppose the SDN controller implements a stateful firewall, as depicted in Figure 1. The topology consists of a single switch connected to trusted internal hosts and untrusted external hosts. Initially the switch has no forwarding rules, so it diverts all packets to the controller. When the controller receives a packet from a trusted internal host, it records the internal-external host pair in its (internal) state and punches a hole in the firewall so the hosts can communicate. Conversely, if the controller receives a packet from an external host first, it logs the connection attempt and drops the packet.

Problem: Dropped state causes disruption. Now suppose the programmer wishes to upgrade the firewall so that if an external host tries to initiate more than n connections, then it is blacklisted from all future communications. With simple restart, the old controller would be swapped out for a new controller that has no knowledge of the old controller's internal state—i.e., the record of connections initiated by internal and external hosts. In the absence of any other information about the state prior to the upgrade, the controller would delete the rules installed on the switch to match its own internal state, which is empty. This leads to a correctness problem:[2] If the external host of an active connection sends the first few packets after the rules are wiped, then those packets will be interpreted as unsolicited connection attempts. The host could easily be blacklisted even though it is merely participating in a connection initiated previously by an internal host.

2.2 Record and Replay

The *record and replay* approach aims to "warm up" the internal state of the new controller by replaying events seen by the old one. The HotSwap (HS) system [34] (an extension of FlowVisor [32]) is a noteworthy example of this approach. At first glance, record and replay seems to offer a fully automatic solution to dynamic controller upgrades. For the stateful firewall, HS would replay the network events for each connection initiated by an internal host and so would easily reconstruct the set of existing connections, avoiding the problems with the simple restart approach. In addition, record and replay can be controller-independent because it works with network events and is indifferent to the internals of the old and new controllers.

But record and replay has two main limitations that prevent it from being a complete solution to the controller upgrade problem.

Problem: Run-time overhead. In general, a record and replay system would need to store all events that might contribute to the state of the new controller. Without prior knowledge of the new controller's functionality (which the system would lack, in general) it is impossible to determine whether a given event might contribute its state or not. Hence, the system would need to log and replay a large number of events. Doing this could be prohibitively expensive, especially in a large and long-running network.

Problem: Reconstructed state may be incorrect. A more serious issue is the recorded trace may not make sense for the new controller, so replaying it may result in an incorrect state. If the behavior of the new controller is different, the events that would have been generated using the new controller and the ones that were actually recorded using the old controller may also be different. For example, installing different forwarding rules on switches causes different packet-in events to be sent to the controller. Note that such events could be induced *directly* (e.g., because the new rules handle fewer packets compared to the old rules) or *indirectly* (e.g., because the new rules elicit different responses from end hosts). In either case, establishing the correctness of the internal state reconstituted by replaying old events would be difficult.

Example failure: Load balancer upgrade. To illustrate, consider the example of a server load balancer, as depicted in Figure 2. The topology consists of a single switch with one port connected to a network of external hosts and another n ports connected to back-end server replicas. Initially, the switch has no rules, so all packets are diverted to the controller. Upon receiving a new connection from an external host, the controller picks a server replica (e.g., uniformly at random) and installs rules that forward traffic in both directions between the host and the selected server. The controller also records the selected server in its internal state

[2]It may also be disruptive: if unmatched traffic is sent to the controller, then the new controller will essentially induce a DDoS attack against itself as a flood of packets stream in.

(e.g., so it can correctly repopulate the forwarding rules if the switch drops out and later reconnects).

Now suppose the programmer wishes to dynamically deploy a new version of the controller where the selection function selects the least loaded server and bounds the number of open connections to any given server, and refuses connections that would cause a servers to exceed that cap. During replay, the new controller would receive a network event for each existing connection request. However, it would remap those connections to the least loaded server instead of the server previously selected by the old controller. The discrepancy between these two strategies could easily break connection affinity—another server replica may receive the ith packet in an existing flow and reset the connection.

2.3 Rule-sourced reconstruction

Another possible strategy is to attempt to reconstruct the state of the new controller by deriving it from rules deployed (by the old controller) on existing switches. Like record and replay, this approach does work under certain assumptions. For example, if the old controller is a proactive application that implements destination-based forwarding along shortest paths, the new controller could "read off" the controller state from the current set of rules installed on switches. Similarly, for the load balancing upgrade in Figure 2, the new controller could extract the connection affinity information from the rules, and use this information to initialize its state.

Problems: Laborious and incomplete. Writing a controller to retrieve information from forwarding rules is laborious, error-prone work for the programmer. It is upgrade specific and requires disentangling the logic of the potentially many applications that installed the rules. Moreover, the information needed to properly initialize the new controller state may simply not be available—e.g., the message history needed for stateful firewalling is absent.

2.4 Solution: Upgrade by state transfer

This paper proposes a general-purpose solution to the controller upgrade problem that attacks the fundamental issue: *dynamically updating the controller's state.* The above approaches attempt to *indirectly* construct a reasonable state, but they lack sufficient precision and performance to fully solve the problem.

Our approach, which we call *upgrade by state transfer*, has three ingredients. First, the controller must be architected to provide *direct* access to its critical state. One way to do this is to store that state in a persistent *network information base* (NIB), as is done in controllers such as Onix [21]. Using a NIB also enables a distributed and fault tolerant architecture. Second, the programmer provides a *state transformer function*, call it μ, that initializes the new controller's state, call it σ', given the old controller's state, call it σ. That is, σ' can be computed as $\mu(\sigma)$. Third, the upgrading service must provide a protocol to signal the controller's components that an upgrade is available so that it can *quiesce* prior to performing the upgrade. Doing so ensures that σ is consistent (e.g., is not in the middle of being changed), before using μ to compute σ'.

Consider the problematic examples discussed thus far. For both the firewall upgrade and the load balancing upgrade, the state transfer approach is trivial and effective: setting the transformer function μ to a no-op (i.e., the identity function) grandfathers in existing connections and the new semantics is applied to new connections. For the load-balancing upgrade, any newly added replicas will receive all new connection requests until the load balances out.

Another feature of upgrade by state transfer is that it permits the developer to more easily address upgrades that are backward-incompatible, such as the load balancer with connection caps discussed above. In these situations, the current network conditions may not constitute ones that could ever be reached had the new controller been started from scratch. With state transfer, the operator can either allow this situation temporarily by preserving the existing state, with the new policy effectively enforced once the number goes below the cap. Or she can choose to kill some connections, to immediately respect the cap. The choice is hers. By contrast, prior approaches will have unpredictable effects: some connections may be reset while others may be unseen by the controller but inadvertently grandfathered in.

In addition to its expressiveness benefits, upgrade by state transfer has benefits to performance: it adds no overhead to normal operation (no logging), and is far less disruptive at upgrade-time (only the time to quiesce the controller and upgrade the state). The main cost is that the network service developer needs to write μ, which can be non-trivial. For example, if we upgraded a routing algorithm from using link counts to using current bandwidth measurements, the controller state would have to change to include additional bandwidth information. Fortunately, according to our experience and an investigation of upgrades to open-source controllers, μ can be relatively simple. In fact, it can be made even simpler by using a domain-specific language (DSL) that makes typical choices the default, and localizes attention on the most interesting aspects. We provide such a DSL with our prototype controller platform, Morpheus, discussed in the next section.

3. MORPHEUS CONTROLLER

To provide a concrete setting for experimenting with dynamic controller upgrades, we have implemented a new distributed controller called Morpheus, implemented in Python and based on the Frenetic libraries [14, 9, 27]. Our design follows the basic structure used in a number of industrial controllers including Onix [21] and ONOS [10], but adds a number of features designed to facilitate dynamic controller upgrades. Morpheus's modular design ensures that nearly all of the controller is dynamically updatable; only a few bits of functionality (involving the updating system itself) cannot be changed.

3.1 Architecture

Morpheus's architecture is shown in Figure 3. The controller is structured as a distributed system in which nodes communicate by message-passing. Morpheus provides four types of nodes:

- platform nodes (PLATFORM), which are responsible for managing low-level interactions with SDN switches and interfacing with network service modules,
- a network information base (NIB), which provides persistent storage for module state,

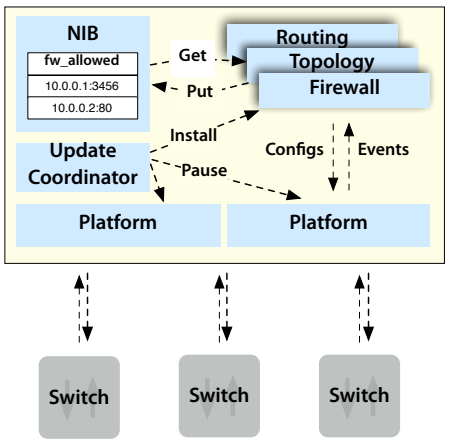

Figure 3: Morpheus architecture.

- an upgrade coordinator (UPDC), which implements distributed protocols for staging and deploying upgrades, and
- service modules (TOPOLOGY, ROUTING, etc.), which implement specific kinds of functionality, such as discovering topology or computing shortest paths through the topology.

Each node executes as a separate OS-level process, supporting concurrent execution and isolation. Processes also make it easy to use existing OS tools to safely spawn, execute, replicate, kill, and restart nodes.

3.2 Components

We now describe Morpheus's components in detail.

Platform. The basic components of Morpheus are PLATFORM nodes, which implement basic controller functionality: accepting connections from switches, negotiating features, responding to keep-alive messages, installing forwarding rules, etc. The PLATFORM nodes implement a simple interface that provides commands that other components use to interact with switches:

- `event()` returns the next network event,
- `update(pol)` sets the network configuration to `pol`, specified using NetKAT [9],
- `pkt_out(sw,pkt,pt)` injects packet `pkt` into the network at `sw` and `pt`,

as well as commands for synchronizing with the UPDC during dynamic controller upgrades:

- `pause()` temporarily stops propagating configurations to the network, and
- `resume()` resumes propagating configurations.

When multiple Morpheus modules are operating, the PLATFORM nodes make every network event available to each module by default. If needed, filtering can be applied to prevent some modules from seeing some network events. Likewise, the policies provided by each module are combined into a single network-wide policy using NetKAT's modular composition operators [9]. For scalability and fault tolerance, Morpheus would typically use several PLATFORM nodes that each manage a subset of the switches. These nodes would communicate with each other to merge their separate event streams into a single stream, and similarly for NetKAT policies. For simplicity, our current implementation uses a single PLATFORM node to manage all of the switches in the network.

Network Information Base. Morpheus modules store critical state in the NIB. The information in the NIB is persistent, surviving a restart of the module(s) that placed it there.[3] The Morpheus NIB is implemented using Redis [6], a popular key-value store [12]. We selected Redis for its simple and efficient interface. At present we implement this NIB as a single node, but we could easily use a distributed implementation for better fault-tolerance (e.g., Redis cluster [7]).

Information stored in the NIB is organized into *namespaces*, which are asociated with a module. For example, a FIREWALL module might store information about which hosts are allowed to communicate under the `fw_allowed` namespace. A module may access data in multiple namespaces: it might be the producer of one piece of data and the consumer of another. For example, our TOPOLOGY module discovers the structure of the network by interacting with the PLATFORM nodes, and stores the topology persistently in the `topology` namespace. This data is then used by the ROUTING module. Redis does not support namespaces directly (some other NoSQL databases do) so we encode the namespace as a prefix of the keys used to store data values.

To handle frequently changing data, Morpheus modules can use Redis' publish-subscribe mechanisms. For example, the TOPOLOGY module publishes a notification to a channel if any of the keys in the topology namespace change, and ROUTING subscribes to this channel and updates its routing configuration when it receives a notification.

Modules. Morpheus modules are implemented using a common design pattern. Upon startup, they connect with the NIB to retrieve any relevant persistent state. The module then adds to, and retrieves from, the persistent store any other necessary data depending on its function. For example, TOPOLOGY discovers and stores hosts, switches, edges, and any additional topological information in the NIB. When ROUTING starts up it reads this information and then adds the least-cost paths to each destination. During normal operation, modules are *reactive*: they will process events from the PLATFORM and from other modules. In response, they will make changes to the NIB state and push out a new NetKAT program via the `update` function on the PLATFORM nodes, which will update in the switches.

Upgrade Coordinator. Because Morpheus has a distributed architecture, dynamic upgrades require coordination between nodes. Morpheus uses an upgrade coordinator (or UPDC) that manages interactions between nodes during an upgrade. These interactions are discussed in detail in the next section.

[3]Obviously, modules may also maintain in-memory state, e.g., for efficiency reasons. To support fault-tolerance (and upgrades), any critical state that cannot be reconstructed on restart should be stored persistently in the NIB.

4. UPGRADES WITH MORPHEUS

Morpheus's design supports controller upgrades by allowing important state to persist in the NIB between versions while providing a way to transform that state when required by an upgrade. To ensure consistent semantics, Morpheus's UPDC node organizes upgrades to the affected modules using a simple protocol. This section describes the update protocol, the domain-specific language Morpheus provides to write state transformations, and some example upgrades we have implemented using Morpheus.

4.1 Upgrade protocol

To deploy an upgrade, the operator provides UPDC with the following:

- new versions of the code for each module affected by the upgrade, and
- a state transformer function μ that maps the existing persistent state in affected namespaces into a suitable format for the new versions of the affected modules.

As a convenience, Morpheus programmers can write μ using a domain-specific language (DSL) we have developed for expressing tranformations on JSON values. This language is discussed in detail in the next subsection. Programs in this language are compiled to Python code that takes an old JSON value and produces an updated version of it.[4] Alternatively, the user can write μ using standard Python code.

Given the upgrade specification, UPDC then executes a distributed protocol that steps through four distinct phases: (i) quiescence, (ii) code installation and state transformation, (iii) restart, and (iv) resumption.

1. Quiesce the affected modules. UPDC begins by signaling the modules designated for an upgrade. The modules complete any ongoing work and shut down, signaling UPDC they have done so. A timeout is used to kill any unresponsive modules. At the same time, UPDC sends the list of modules to the PLATFORM, which temporarily suppresses any rules updates made by those modules, which could be stale. After all modules exit and the PLATFORM indicates it has begun blocking rule updates, Morpheus is said to have reached *quiescence*.

2. Install the upgrade in the NIB*.* Next, UPDC installs the administrator-provided μ functions at the NIB. The NIB verifies that these functions make sense, e.g., that if the request is to upgrade namespace `nodes` from `v3` to `v4`, then the current NIB should contain namespace `nodes` at version `v3`. The transformations are applied *lazily*, during the final step of the protocol.

3. Restart the upgraded modules. Now UPDC begins the process of resuming operation. UPDC signals the new versions of the affected modules to start up. These modules reconnect to the NIB, and the NIB ensures that the modules' requested version matches the version just installed in the NIB. The modules then retrieve relevant state stored in the NIB, and compute new rules to push to the PLATFORM. The PLATFORM receives and buffers the new rules: it propagates them to the network after it has received rules (or otherwise been signaled) from *all* upgraded modules, to ensure that the rules were generated from consistent software versions. After the PLATFORM has received rules from all upgraded modules, it removes the old rules previously created by the upgraded modules and installs the new rules on the switches.

4. Resume operation. At this point, the upgrade is fully loaded and the modules proceed as normal. As the modules access data in the NIB, any installed μ function is applied lazily. In particular, when a module queries a particular key, if that key's value has not yet been transformed, the transformer is invoked at that time and the data is updated. In the event that the data is several versions behind, Morpheus applies each upgrade's μ function in sequence to bring the data up to date.

Next, we describe the language we provide for writing state transformations. We conclude with some example upgrades we have implemented in Morpheus for a stateful firewall, and for TOPOLOGY and ROUTING modules.

4.2 Specifying State Transformations

To make it easier for programmers to migrate data that has changed format, we developed a domain-specific language (DSL) for specifying the required changes that will generate μ as an alternative to writing it as explicit Python code. Our language supports changes to key names, as well as modifications to the contents of JSON objects. For the latter we support adding, deleting, renaming, and updating fields. The API also permits querying the contents of data in other modules, e.g., for when a changed field depends on the contents of another module's objects.

To formulate an update, the programmer writes a simple program that describes *which* key-value pairs should be updated and *how* they should be modified:

```
for (regex) old_ver->new_ver {
  DIRECTIVE [json path] {action-code}
  ...
};
```

The first line contains a regular expression that specifies the namespace to be updated. This allows the update-writer to match all or part of the namespaces in Morpheus, such as using `fw_*` to match two namespaces called `fw_allowed` and `fw_pending`. The `old_ver` and `new_ver` fields are unique strings that the user must provide to assist Morpheus to track which data have been updated. The body of the `for{...}` contains one or more commands, each beginning with a `DIRECTIVE` that specifies the action to be performed. There are four directives to choose from, shown in Table 1: initializing a new field, updating the value contained in a field, deleting a field, or renaming the field.

After specifying the `DIRECTIVE`, the `json path` is an index into the JSON object (expressed as a list of JSON field names in the case of nested fields), indicating where to apply the corresponding `action-code`. This code consists of a mix of special DSL tokens and Python code. The `action-code` differs per directive, as shown in the third column of Table 1. The `INIT` and `UPD` directives are similar in that they both must specify the value that should be initialized or renamed. The `DEL` requires the code to return true or false, indicating whether a given path should, indeed, be deleted.

[4]While the programmer must currently write μ, automated assistance is also possible [18, 26].

Directive	Path	Action code	Must Return
INIT	[json path, or empty for entire value]	Yes. (Assign value to `$out`.)	None
UPD	[json path, or empty for entire value]	Yes. (Assign value to `$out`.)	None
DEL	[json path, or empty for entire value]	Yes. (Code to determine what to delete.)	Bool
REN	[json path] → [json path]	No	None

Table 1: The DSL Directives

Token Meaning
`$out` - the value of the path inside the [ ] in the Directive *(this is the value to be updated, initialized, deleted, etc)*
`$in` - the original value stored in the key (same as `$out`, but not written to)
`$root` - the root of the JSON structure.
`$base` - the same JSON structure, used to address siblings
`$dbkey`- the database key name currently being processed

Table 2: The DSL Convenience Tokens

Table 2 shows the DSL tokens that may appear in action code, which are interpreted specially by our DSL compiler. All of the directives' action code may use any of the tokens, except `INIT` which may not use `$in` because an existing value does not exist for an initialized value. To update the *keys* rather than the JSON values, the programmer uses the `UPD` directive with an empty JSON path, and sets the `$dbkey` token to the desired new key name rather than setting `$out`.

DSL programs are translated into Python functions that are stored along with the upgrade specification in the database. During upgrade, these functions are called for each key-value pair whose key matches the regular expression.[5] We present two examples of writing upgrades with our DSL in the next two subsections.

4.3 Upgrade example: Firewall

We developed three different versions of a stateful firewall, and defined upgrades between them.

- FIREWALL$_{out}$ permits bidirectional flows between internal and external hosts as long as the connection is initiated by an outgoing request. The controller installs forwarding rules between internal host S and external host H when it sees S's outbound packet.

- FIREWALL$_{outin}$ acts like FIREWALL$_{out}$ but only installs the rules permitting bidirectional flows after seeing returning traffic following an internal connection request. It might do this to prevent attacks on the forwarding table originating from a compromised host within the network.

- FIREWALL$_{outinTO}$ adds to FIREWALL$_{outin}$ the ability to time out connections (and uninstall their forwarding rules) after some period of inactivity between the two hosts.

FIREWALL$_{out}$ defines a namespace `fw_allowed` that keeps track of connections initiated by trusted hosts, represented as JSON values:

```
{ "trusted_ip": "10.0.0.1",
  "trusted_port": 3456,
  "untrusted_ip": "10.0.0.2",
  "untrusted_port": 80 }
```

Updating from FIREWALL$_{out}$ to FIREWALL$_{outin}$ requires the addition of a new namespace, called `fw_pending`. The keys in this namespace track the internal hosts that have sent a packet to an external host but have not heard back yet. Once the return packet is received, the host pair is moved to the `fw_allowed` namespace. For this upgrade, no transformer function is needed: all connections established under the FIREWALL$_{out}$ regime can be allowed to persist, and new connections will go through the two-step process.[6]

Updating from FIREWALL$_{outin}$ to FIREWALL$_{outinTO}$ requires updating the data in the `fw_pending` and `fw_allowed` namespaces, by adding two fields to the JSON values they map to, `last_count` and `time_created`, where the former counts the number of packets exchanged between an internal and external host as of the time stored in the latter. Every n seconds, the firewall module will query the NIB to see if the packet count has changed. If so, it stores the new count and time. If not, it removes the (actual or pending) route.

In our DSL we can express the transformation needed to upgrade from FIREWALL$_{outin}$ to FIREWALL$_{outinTO}$ data for the `fw_allowed` namespace as follows:

```
for fw_allowed:* ns_v0->ns_v1 {
  INIT ["last_count"] {$out = 0}
  INIT ["time_created"] {$out = time.time()}
};
```

This program states that the JSON value stored under each key should be updated from version `ns_v0` (corresponding to FIREWALL$_{outin}$) to `ns_v1` (corresponding to FIREWALL$_{outinTO}$) by adding two additional fields. We can safely initialize the `last_count` field to 0 because this is a lower bound on the actual exchanged packets, and we can initialize `time_created` to the current time. Both values will be updated at the next timeout. The above DSL code will be transformed to Python code that is stored (as a string) in Redis.

[5]Programmers can test that these functions work properly as usual, e.g., by testing the upgrade on a private system. Techniques for testing dynamic software updates would also be relevant here [25, 17].

[6]We could also imagine moving all currently approved connections to the pending list, but the resulting removal of forwarding rules would be unnecessarily disruptive.

The existing data will be transformed as the new-version code accesses it via the NIB API. When the new version retrieves connection information from the NIB, the transformation adds the two new fields to the existing value:

```
key:   fw_allowed:10.0.0.1_3456_10.0.0.2_80
value: { "trusted_port": 3456,
         "untrusted_port": 80,
         "trusted_ip": "10.0.0.1",
         "untrusted_ip": "10.0.0.2",
         "last_count": 0,
         "time_created": 1426167581.566535 }
```

The new controller code need not be aware that it is being deployed as an upgrade, as the data values in the NIB are seamlessly transformed as they are accessed.

4.4 Coordination: Routing and Topology

In the above example, the firewall stores its own data in the NIB with no intention of sharing it with any other modules. As such, we could kill it, apply the upgrade, and start the new version without worrying about interactions with other modules. However, when multiple modules share the same data and its format changes in a backward-incompatible manner, then it is critical that we employ the upgrade protocol described in Section 4.1, which gracefully coordinates the upgrades to modules with shared data.

As an example coordinated upgrade, recall from Section 3 that our ROUTING and TOPOLOGY modules share topology information stored in the NIB. In its first version, TOPOLOGY merely stores information about hosts, switches, and the links that connect them. The ROUTING module computes per-source/destination routes, assuming nothing about the capacity or usage of links. In the next version, TOPOLOGY regularly queries the switches for port statistics and stores the moving average of the available capacity on each link in the NIB. This information is then used by the ROUTING module to compute paths. The result should be better load balancing when multiple paths exist between a given pair of hosts. However, performing this update correctly requires quiescing both ROUTING and TOPOLOGY using our update protocol. Note that if we did not do this, TOPOLOGY might be (unsafely) updated first, and then the non-updated ROUTING could see topology data in the wrong format.

To perform the update, the programmer must define a μ function that transforms the NIB state into the version used in the new controller. In this case, the μ function would add a field to represent the available capacity on each edge, initializing it with a default value—e.g., 1. The following code implements such a function in our DSL:

```
for edge:* ns_v0->ns_v1 {
  INIT ["weight"] {$out = 1}
};
```

Using this state transformer function, the new controller will initially use the same routes as the old controller, because the initial values are identical, ensuring stability. Subsequent ROUTING computations will account for and store available capacity information, thus balancing traffic better across all available routes.

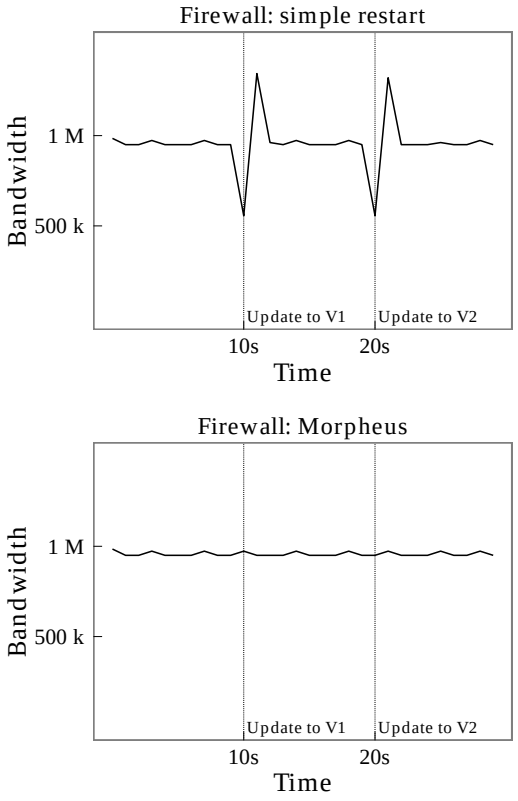

Figure 4: Firewall Upgrade

5. EXPERIMENTS AND EVALUATION

In this section, we report the results of experiments where we dynamically upgrade several canonical SDN services, implemented as Morpheus modules: a load balancer, a firewall, and a router. We demonstrate three controller upgrade mechanisms: state transfer using Morpheus, simple restart, and record and replay. In all cases, state transfer is fast and disruption-free, whereas the other techniques cause a variety of problems, from network churn to dropped connections. We ran all experiments using Mininet HiFi [16], on an Intel(R) Core(TM) i5-4250U CPU @ 1.30GHz with 8GB RAM. We report the average of 10 trials.

5.1 Firewall

Figure 4 illustrates a dynamic upgrade to the firewall, described in Section 4.3, from FIREWALL$_{out}$ to FIREWALL$_{outin}$ and then to FIREWALL$_{outinTO}$. The figure shows the result of simple restart (where all data is stored in memory and lost on restart) and state transfer (where data is stored in the NIB). We do not depict record and replay, which happens to perform as well as state transfer for this example (as discussed in §2.2).

For the experiment, we used a single switch with two ports, each connected to a host via a 1Mbps link. We designated one host as the client inside the firewall, and the other as the server outside the firewall. We used `iperf` to establish a TCP connection from the client to the server. The figure plots the bandwidth reported by `iperf` over time. In both experiments, we upgrade to FIREWALL$_{outin}$ after 10 seconds and to FIREWALL$_{outinTO}$ after 20 seconds.

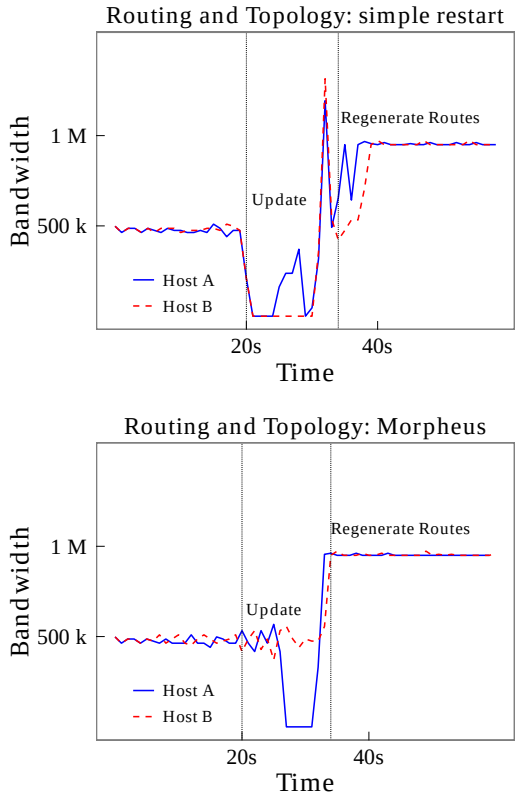

Figure 5: Routing and Topology Discovery Upgrade

start	*apps exit*	*restart begins*	*rout push*	*topo push*	*platform resume*
0.00s	0.05s	0.11s	1.67s	1.68s	1.70s

Table 3: Upgrade Quiescence Times for TOPOLOGY and ROUTING (Median of 11 trials)

The figure shows that the bandwidth drops significantly during an upgrade using simple restart. This is unsurprising, since a newly started firewall does not remember existing connections. Therefore, FIREWALL$_{outin}$ and FIREWALL$_{outinTO}$ first block all packets from the server to the client, until the client sends a packet, which restores firewall state. In contrast, Morpheus does not drop any packets because state is seamlessly transformed from one version to the next.

5.2 Routing and Topology

Figure 5 shows the effect of updating the routing and topology modules (section 4.4), where the initial version uses shortest paths and the final version takes network load into account. The experiment uses four switches connected in a diamond-shaped topology with a client and server on either end (i.e., with two equal-costs paths between them). The client establishes two TCP connections to the server using `iperf`.

Initially, both connections are routed along the same shortest path. Since the links along the path have a capacity of 1Mbps, each connection gets 500kbps on average. After 20 seconds, we upgrade both modules: the new version of

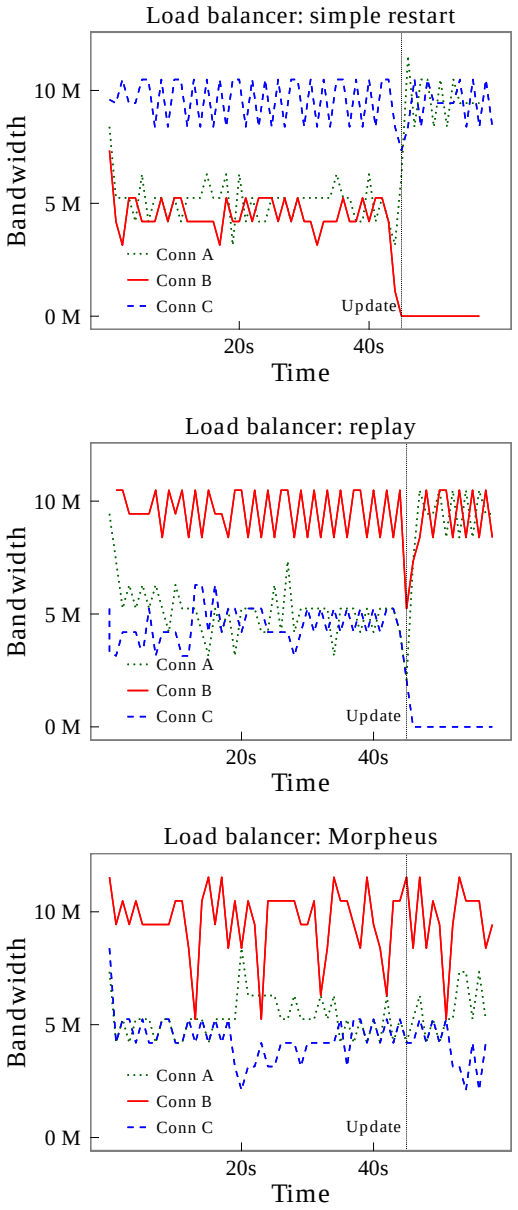

Figure 6: Load Balancer Results

TOPOLOGY stores link-utilization information in the NIB and the new version of ROUTING uses this information to load-balance traffic. After the upgrade, each connection should be mapped to a unique path, thus increasing link utilization.

Using simple restart, both connections are disrupted for 10 seconds, which is how long TOPOLOGY takes to learn the network topology. Morpheus is much less disruptive: since the μ function preserves topology information, the new ROUTING module maps each connection to a unique path. The connection that is not moved (Host B) suffers no disruption and gracefully jumps to use 1Mbps bandwidth. The connection that is moved (Host A) is briefly disrupted as several switch tables are updated.[7]

[7]This minor disruption could be avoided using a consistent update [27].

Table 3 breaks down the time to run the upgrade protocol for this upgrade. It takes 0.05s for both TOPOLOGY and ROUTING to receive the signal to exit at their quiescent points and shut down, and for the PLATFORM to also receive the signal and pause. At 0.11s, both modules restart, begin pulling from the NIB, and begin performing computations. At 1.67s and 1.68s respectively, the ROUTING and TOPOLOGY modules send their newly computed rules to the PLATFORM. The PLATFORM holds on to the rules until it ensures it has received the rules from both apps, and then PLATFORM pushes both sets of rules to the switches and unpauses. This entire process takes 1.70s, with most of the time taken by simply restarting the module (as would be required in the simple case anyway). In general, the amount of time to upgrade multiple modules safely will vary based on number of modules, the amount of state to restore, and the type computations to be performed to generate the rules, but the overhead (compared to a restart) seems acceptable.

5.3 Load Balancer

Figure 6 shows the effect of updating a load-balancer that maps incoming connections to a set of server replicas. For this experiment, in addition to the simple restart and Morpheus experiments, we also report the behavior of record-and-replay, which works by recording the packet-in events and replaying them after restart. After 40 seconds, we bring an additional server online and upgrade the module to also map connections to this server. To avoid disrupting clients, existing connections should not be moved after the upgrade.

As shown in the figure, simple restart and record-and-replay both cause some clients to be disconnected. As discussed in Section 2.2, replaying the recorded packet-ins will cause the three connections to be evenly distributed across the three servers. Similarly, for the simple restart, the connections will be evenly distributed when the clients attempt to reconnect. Therefore, one connection is erroneously mapped to the new server mid-stream, which terminates the connection. In contrast, Morpheus's state transfer causes no disruptions, since the relevant controller state is preserved.

5.4 Programmer Effort

A Morpheus programmer must implement two pieces of code to make a module upgradeable: (i) they must write code to quiesce the module prior to an update, and (ii) they must write a state transformer function μ to map the state of the old controller into the representation used by the new controller.

Quiescence. To check whether an upgrade is available, a Morpheus module can simply check for NIB notifications. Once notified, the module completes all outstanding tasks such as storing additional state in the NIB or sending external notifications, and then gracefully exits. In our examples, this functionality was extremely simple to implement, amounting to less than 10 lines of code in each module. Moreover, as this functionality is largely structural, it is likely to be a one-time effort—i.e., future changes to the module will likely not affect quiescence code [18].

Transforming the state. Writing the function μ to transform the state was also straightforward. For FIREWALL, as described in Section 4.3, we wrote 4 lines of DSL code to initialize new fields to desired values so that the fields could be read with the correct data. Similarly for our modules TOPOLOGY and ROUTING, as described in Section 4.4, we wrote 3 lines of DSL code to initialize the weight field to a default value. For the LOAD BALANCER, no μ function was necessary, as no state was transformed, only directly transferred to the new version of the program.

We also looked at the revision histories of other controllers to get a sense of how involved writing a μ function might be for controller upgrades that occur "in the wild." In particular, we looked at GitHub commits from 2012–2014 for OpenDaylight [4] and POX [5] controllers. We examined controller service modules such as a host tracker, a topology manager, a Dijkstra router, an L2 learning switch, a NAT, and a MAC blocker.

Several of the changes we observed only affected the service logic. This was the case for changes to POX's IP load balancer in 2013, for example. For code-only changes, no μ would be necessary. Many of the other changes involved adding new state, or making small modifications to existing state. For example, a change to OpenDaylight's host tracker on November 18, 2013 converted the representation of an `InetAddress` to a `IHostId` to allow for more flexibility and to store some additional state such as the data layer address. To support this change as a dynamic upgrade, the administrator would write μ to initialize the data layer address for all stored hosts, if known, or add some dummy value to indicate that the data layer address was not known. A change to POX's host tracker on June 2, 2013 added two booleans to the state to indicate if the host tracker should install flows or should suppress ARP replies. To make this change an upgrade, the administrator would write μ to initialize these two to `True` in the NIB. Of course, it is certainly possible that writing μ functions could be more complicated than this, but so far our investigation suggests that the effort to write μ would often be minimal.

6. RELATED WORK

Morpheus represents the first general-purpose solution to the problem of dynamically updating SDN controllers (and by extension, updating the networks they manage). We discussed the challenges related to dynamic updates extensively in Section 2, specifically comparing state transfer to alternative techniques involving controller restarts and record and replay (exemplified by the HotSwap system [34]). In this section we provide comparison to other work that provides some solution to the controller upgrade problem.

Graceful control-plane upgrades. Several previous works have looked at the problem of updating control-plane software. In-Service Software Upgrades (ISSU) [1, 3] minimize control-plane downtime in high-end routers upon an OS upgrade by installing the new control software on different blades, and synchronizing the state automatically. Other research proposals go even further and allow other routers to respond correctly to topology changes that affect packet forwarding, while waiting for a peer to restart its control plane [29, 30]. In general, most routing protocols have mechanisms to rebuild their state when the control software (re)starts (cf. [23, 28]), e.g., by querying the state of neighboring routers.

The key difference between these works and Morpheus is that Morpheus aims to support unanticipated, semantic changes to control-plane software, possibly necessitating a

change in state representation, whereas ISSU and normal routing protocols cannot.[8] In addition, Morpheus is general-purpose (due to its focus on SDN), and not tied to a specific routing protocols.

Distributed Controllers. Distributed SDN controller architectures such as Onix [22], Hyperflow [33], ONOS [10] or Ravana [20] can create new controller instances and synchronize state among them using a consistent store. Morpheus's distributed design is inspired by the design of these controllers, which aim to provide scalability, fault-tolerance and reliability, and can support simple upgrades in which the shared state is unchanged between versions (and/or is backward compatible). However, to the best of our knowledge these systems have not looked closely at the controller upgrade problem when (parts of) the control program itself must be upgraded in a semantics-changing manner, especially when the new controller may use different data structures and algorithms than the old one. Morpheus handles this situation using the upgrade protocol defined in Section 4, which quiesces the controller instances, initiates a transformation of the shared store's data according to the programmer's specification (if needed), and then starts the new controller versions. We believe this same approach could be applied to these distributed controllers as well.

Record and Replay. Another approach to implementing dynamic updates is to record events received by the old controller and replay them to "warm up" the new system. This idea was explored in previous work on HotSwap [34]. Another line of work has explored using record and reply in the context of middleboxes [15, 31] and even in general-purpose operating systems [13]. Unfortunately, as described in the early sections of this paper, record and replay does not fully solve the controller update problem—in particular, when the new controller implements different functionality, simply replaying old events may not correctly reconstruct the internal state. By contrast, Morpheus offers tools that programmers can use to implement correct dynamic updates, including when functionality changes.

Dynamic Software Upgrades. The approach we take in Morpheus draws lessons from recent work on *dynamic software updating* (DSU) [19, 24, 18, 26], which focuses on updating a running software application without shutting it down. Most prior DSU work has focused on updating a single running process, which may involve transforming the contents of its heap to work with the new code. Upgrading in distributed systems is detailed in Ajmani et al. [8], where updates are preformed on distributed general purposed nodes communicating via remote procedure calls. By contrast, Morpheus is concerned with coordinating an upgrade to many processes that all share the same persistent state—transformation is on the persistent state, not each process's memory.[9] Our DSL for writing transformations draws inspiration from *xfgen*, a DSL provided by the Kitsune DSU system [18]; the different domain results in several differences, notably in how updateable values are specified (mapped to from their keys in particular namespaces) and on support for updating keys.

[8]Cisco only supports ISSU between releases within a rolling 18-month window [11]. Outside of this window, a hard-reset of the control-plane has to be done.

[9]Of course, DSU techniques could be applied to each Morpheus module, but in our experience this would add little value.

7. CONCLUSIONS

This paper proposes *controller upgrade by state transfer* as an approach for dynamically upgrading SDN controllers. This approach works by providing direct access to the relevant state in the running controller, and initializing the new controller's state as a function of the existing state. It is in contrast to alternatives that attempt to automatically reproduce the relevant state, but may not always succeed. We implemented the approach as part of Morpheus, a new SDN controller whose design is inspired by industrial-style controllers. Morpheus provides means to specify transformations in a persistent store, and employs an upgrade coordination protocol to safely deploy the transformation. Experiments with Morpheus show that upgrading by state transfer is both natural and effective: it supports seamless upgrades to live networks at low overhead and little programmer effort, while prior approaches would result in disruption, incorrect behavior, or both.

Acknowledgments. The authors wish to thank the SOSR reviewers for their helpful feedback. Our work is supported in part by the National Science Foundation under grants CNS-1111698, SHF-1253165, SHF-1408745, CNS-1413985, CNS-1413972, and ACI-1440744; the Office of Naval Research under grant N00014-15-1-2177; a Google Faculty Research Award; and gifts from Cisco, Facebook, and Fujitsu.

8. REFERENCES

[1] Cisco IOS In Service Software Upgrade. http://tinyurl.com/acjng7k.
[2] Floodlight. http://floodlight.openflowhub.org/.
[3] Juniper Networks. Unified ISSU Concepts. http://tinyurl.com/9wbjzhy.
[4] OpenDaylight. http://www.opendaylight.org.
[5] Pox. http://www.noxrepo.org/pox/about-pox/.
[6] Redis. http://redis.io/.
[7] Redis Cluster Specification. http://redis.io/topics/cluster-spec.
[8] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *ECOOP*, 2006.
[9] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *POPL*, 2014.
[10] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an open, distributed SDN OS. In *HotSDN*, 2014.
[11] Cisco Systems. Cisco IOS In-Service Software Upgrade. http://www.cisco.com/c/dam/en/us/products/collateral/ios-nx-os-software/high-availability/prod_qas0900aecd8044c333.pdf.
[12] Db-engines ranking. http://db-engines.com/en/ranking, 2015.
[13] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *OSDI*, 2014.

[14] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, 2011.

[15] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *SIGCOMM*, 2014.

[16] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT*, 2012.

[17] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Proceedings of the International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, pages 278–293, Jan. 2012.

[18] C. M. Hayden, K. Saur, E. K. Smith, M. Hicks, and J. S. Foster. Efficient, general-purpose dynamic software updating for c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):13, Oct. 2014.

[19] M. Hicks and S. M. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1049–1096, November 2005.

[20] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *SOSR*, 2015.

[21] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

[22] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

[23] J. Moy, P. Pillay-Esnault, and A. Lindem. Graceful OSPF Restart. RFC 3623, 2003.

[24] I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *PLDI*, 2009.

[25] L. Pina and M. Hicks. Tedsuto: A general framework for testing dynamic software updates. In *Proceedings of the International Conference on Software Testing (ICST)*, 2016.

[26] L. Pina, L. Veiga, and M. Hicks. Rubah: DSU for Java on a stock JVM. In *OOPSLA*, 2014.

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

[28] S. Sangli, E. Chen, R. Fernando, J. Scudder, and Y. Rekhter. Graceful Restart Mechanism for BGP. RFC 4724, Jan. 2007.

[29] A. Shaikh, R. Dube, and A. Varma. Avoiding instability during graceful shutdown of OSPF. In *INFOCOM*, 2002.

[30] A. Shaikh, R. Dube, and A. Varma. Avoiding instability during graceful shutdown of multiple OSPF routers. *IEEE/ACM Transactions on Networking*, 14(3):532 –542, june 2006.

[31] J. Sherry, P. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Macciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback recovery for middleboxes. In *SIGCOMM*, 2015.

[32] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *OSDI*, 2010.

[33] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, 2010.

[34] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford. Hotswap: Correct and efficient controller upgrades for software-defined networks. In *HotSDN*, 2013.