Safe and Flexible Controller Upgrades for SDNs

Karla Saur
University of Maryland
ksaur@cs.umd.edu

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* changes—i.e., without compromising ongoing network functions—are inadequate. Simply halting old controllers and bringing up new ones can cause state to be lost, which often leads to incorrect behavior; e.g., if the state represents hosts blacklisted by a firewall, then traffic that should be blocked may be allowed to pass through. Techniques based on record and replay can reconstruct state automatically, but they are expensive to deploy and do not work in several common scenarios.

This paper presents a new approach to implementing dynamic controller upgrades. Based on what we see as the space of upgrades that deployed systems are likely to face, we believe that controller upgrades need to be a fundamental part of the controller's design. As such, we present a controller design and implementation called Morpheus that uses a technique called *explicit state transfer* to effect controller upgrades. Morpheus enables programmers to directly (and easily, in most cases) initialize the upgraded controller's state as a function of its existing state. Morpheus provides a domain-specific language for this purpose, as well as a protocol for safely deploying an upgrade across distributed and coordinating controller modules. Experiments confirm that explicit state transfer with Morpheus ensures good performance and correct behavior during upgrades, while prior approaches can do much worse.

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 such a way as to minimize disruption.

In general, dynamic upgrades differ from static ones 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 formats to represent internal data structures 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 is lost, so for consistency, rules on the switches are wiped at startup (precipitating state reconstruction).

• In *record and replay*, provided by HotSwap [31], OpenNF [14] and related systems [28], 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 switch rules alone).

• 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 general-purpose solution to the controller upgrade problem. We present a detailed argument in Section 2, but the gist of that argument is the following. Simple restarts discard internal state, but wiping rules to regenerate it causes performance disruption (e.g., packet-ins suddenly flood the controller) and makes no guarantee that the reconstructed state will be harmonious with the assumptions being made by end hosts (e.g., existing flows may be dropped or misrouted, breaking connections used by applications). Record and replay can reproduce a harmonious 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 in switch rules. It is also risky in the face of inevitable switch failures.

Morpheus: Controller upgrades by state transfer. This paper proposes a more general and flexible solution to the controller upgrade problem, which we call *controller upgrade by state transfer.* We have implemented this approach in a new controller called Morpheus, 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 *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.), and each can coordinate and share information via the NIB (like network performance data, active connection info, etc.).[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.

• Second, Morpheus provides a way to easily *transform* the old controller state so as to make it consistent with the assumptions of the new controller's code, ensuring that it is harmonious with the network—e.g., preserving existing flows and policies. We expect the programmer to write any necessary transformation, and to make this easier we define a domain-specific language (DSL) for doing so, described in section 4.2. In our experience, and based on our investigation of in-the-wild controller upgrades, writing such transformations is often straightforward.

• Third, Morpheus provides a protocol for *coordinating* the deployment of a controller upgrade, described in Section 4.1. In particular, 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), and changes to one module might affect the state's representation, necessitating an upgrade to the other module. We 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.

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, the operator has complete control over the post-upgrade network state, affording greater flexibility and ease of programming. Finally, Morpheus's modular design means its techniques should 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, described in Section 5, we demonstrate the advantages of upgrade by state transfer, compared to simple restarts and record-and-replay. In essence, there is far less disruption, and no incorrect behavior. We also confirm that the state transformation functions are relatively simple to write.

Summary. This paper's contributions are as follows:

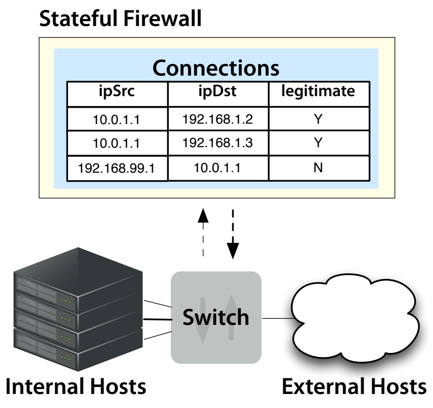
- We study the problem of performing dynamic upgrades to SDN controllers and identify fundamental limitations of current approaches.
- We propose a new, general-purpose solution to the dynamic upgrade problem for SDN controllers—*controller upgrade by state transfer.* With this solution, the programmer 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

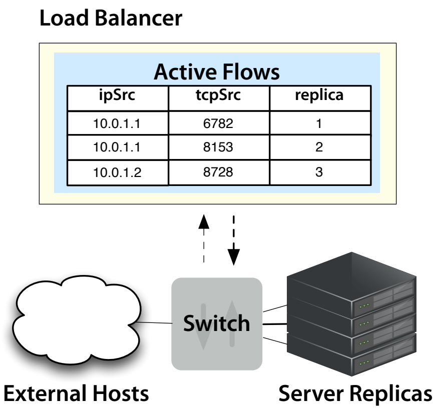
This section explains why existing approaches for handling dynamic upgrades to SDN controllers are inadequate in general, and provides detailed motivation for our approach based on *state transfer.*

[1]This design is reminiscent of that of the distributed architecture provided by industrial controllers such as Onix [19] and ONOS [9]. We considered retrofitting a simpler open-source controller such as POX or Floodlight [5, 2], but decided to build a new distributed controller to provide evidence that upgrade by state transfer scales to industrial architectures, which tend to be distributed.



ipSrc	ipDst	legitimate
10.0.1.1	192.168.1.2	Y
10.0.1.1	192.168.1.3	Y
192.168.99.1	10.0.1.1	N

Figure 1: Example network service: stateful firewall.



ipSrc	tcpSrc	replica
10.0.1.1	6782	1
10.0.1.1	8153	2
10.0.1.2	8728	3

Figure 2: Example network service: load balancer.

2.1 Simple Restart

In *simple restart*, the system halts the old controller and begins executing a fresh copy of the new controller, and thereby discards the old controller's internal state. Doing so can lead to incorrect handling of existing flows, as the following example shows.

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]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.

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 [31] (an extension of FlowVisor [29]) 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. Moreover, record and replay offers some generality because by focusing on events it 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 full solution to the controller upgrade problem.

Problem: Run-time overhead. In general, unless the system has prior knowledge of the new controller's functionality (which it will not, in general), the system will have to record (and replay) all relevant events that contributed to the network's current state. Doing this can be expensive in a large, 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. Why? The new controller may, in general, behave differently than the old one—e.g., it may install different switch forwarding rules. As such, if the new controller had been used from the start, these rules might have caused different network events to be generated than those that were actually recorded. Such events could have been induced *directly* (e.g., because the new rules handle fewer or more packets compared to the old rules) or *indirectly* (e.g., because the new rules elicit different responses from the hosts that are communicating via the network). Establishing the correctness of the internal state reconstituted by replaying old events is difficult in general.

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 (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 load balancing strategies could well break connection affinity—a different server replica may receive the ith packet in a flow and reset the connection.

2.3 Rule-sourced reconstruction

Another possible strategy would be 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 can work in some cases. For example, according to private discussions with SDN operators, this approach is often adapted by proactive controllers that do not make frequent changes to network state (e.g., destination-based forwarding along shortest paths). For the load balancing upgrade in Figure 2, the new controller could extract the connection affinity information from the current rules, and use this information to initialize its state.

Problems: Laborious and incomplete. Writing a controller to retrieve information from forwarding rules is potentially laborious, error-prone work for the programmer, because it is upgrade specific and requires disentangling the logic of the potentially many services 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 [19]; pleasantly, using a NIB also enables a distributed and fault tolerant architecture. Second, the programmer provides a function, call it μ, that initializes the new controller's state, call it σ, given the old controller's state, call it σ_0; i.e., $\sigma = \mu(\sigma_0)$. 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_0 is consistent (e.g., is not in the middle of being changed), parallel before using μ to compute σ.

Consider the problematic examples presented thus far. For both the firewall upgrade and the load balancing upgrade, the state transfer approach is trivial and effective: setting the μ function to a no-op (i.e., identity function) grandfathers in existing connections and the new semantics is applied to new connections. Pleasantly, 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

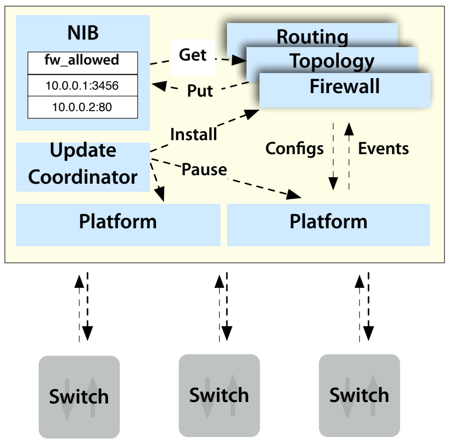


Figure 3: Morpheus architecture.

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 will not always be a no-op. 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, 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 [13, 8, 24]. Our design follows the basic structure used in a number of industrial controllers including Onix [19] and ONOS [9], 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 via well-defined message-passing interfaces. 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,
- 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 most basic components 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 [8],
- `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 [8]. 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 put it there.[3] The NIB is implemented using Redis [6], which a popular key-value store [11]; we selected it for its simple and efficient interface.

Data stored in the NIB is divided among conceptual *namespaces*, organized according to the modules that use it. For example, a firewall module might store information in the NIB in the `fw_allowed` namespace about which hosts are currently allowed. A module may access data in multiple namespaces, where it might be the conceptual data owner for one, but a 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 under which we store a module's data values. Many Morpheus modules also use Redis' built-in publish-subscribe mechanism to handle frequently changing data. For example, TOPOLOGY publishes a notification to a channel if any of the keys in the topology namespace changes, and ROUTING subscribes to this channel and updates its routing configuration appropriately when it receives a notification that the topology has changed.

Modules. Morpheus modules tend to be 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 additional information about the topology in the NIB, and 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 (e.g., via the pub-sub mechanism). 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. We discuss these interactions in detail in the next section.

[3]Obviously, modules may also maintain in-memory state for efficiency reasons, but this state is lost on restart.

4. CONTROLLER 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 this protocol, the language we provide to write state transformations, and then describes some example upgrades that we have performed.

4.1 Upgrade protocol

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

- New versions of the affected modules' code
- A state transformation function μ that maps the existing persistent state in affected namespaces into a format suited to the new module versions.

As a convenience, the module programmer can write μ in a domain-specific language (DSL) we developed for writing transformers over JSON values; we discuss the language in the next subsection. This language's programs 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 unresponsive modules.) At the same time, UPDC sends the list of modules to the PLATFORM, which will temporarily suppress any rules updates made by those modules, which could be stale. Once all modules have exited, and the PLATFORM has indicated it has begun blocking the rules, Morpheus has 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 versions `v3->v4`, then the current NIB should contain namespace `nodes` at version `v3`. All transformations will be applied *lazily*, as part of step 4.

[4]While the programmer currently must write μ, automated assistance is also possible [16, 23].

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 and push the new rules to the PLATFORM. The PLATFORM receives and holds the new rulesets. It will push them once it has received rules (or otherwise been signaled) from *all* of the upgraded modules, to ensure that the rules were generated from consistent software versions. Once the PLATFORM has received rules from all upgraded modules, it will remove the old rules previously created by the upgraded modules and install 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.

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 manually writing μ with 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.

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. 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 `$in` because an existing value does not exist for an initialized value. To update the *keys* rather than the JSON values, the upgrade-writer should use the `UPD` directive with an empty JSON path, and set the `$dbkey` token to the desired new key name rather than setting `$out`.

DSL programs are translated into Python functions (μ) 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. 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:

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] $\rightarrow$ [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

```
{ "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.[5]

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 (for some N, like 3), 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 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()}
  };
```

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

This states that for every key in the namespace, its corresponding JSON value is updated from version `ns_v0` (corresponding to FIREWALL$_{outin}$) to `ns_v1` (corresponding to FIREWALL$_{outinTO}$) by adding two JSON 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.

The existing data will be transformed as the new version accesses it via the NIB accessor API. When the new version of the program retrieves connection information from the NIB, the transformation would add the two new fields to the existing JSON value shown earlier in this section:

```
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 }
```

4.4 Coordination: Routing and Topology

In the above example, the firewall is storing its own data in the NIB with no intention of sharing it with any other modules. As such, we could have killed the module, applied the upgrade, and started the new version. However, when multiple modules share the same data and its format changes in a backward-incompatible manner, then it's 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 each link's bitrate in the NIB. This information is then used by ROUTING when computing paths. The result should be better load balancing when multiple paths exist, between hosts.

Updating from the first to the second version in Morpheus requires adding a field to the JSON object for edges, to add the measured bitrate. The transformer μ simply initializes this field to 1, indicating the default value for traffic on the link as follows:

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

As such, the initial run of the routing algorithm will reproduce the existing routes because all initial values will be the same, ensuring stability. Subsequent ROUTING computations will account for and store the added usage information and thus better balance the routes.

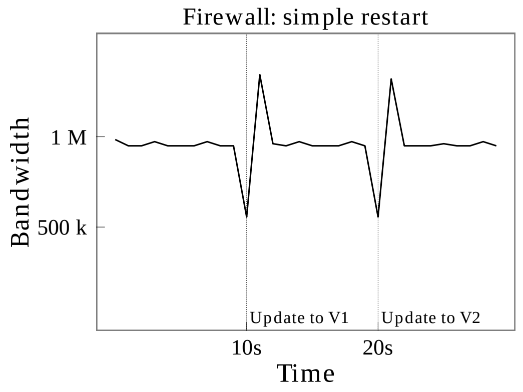
5. EXPERIMENTS AND EVALUATION

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 [15], 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 per Section 2.2).

For the experiment, we used a single switch with two ports (with 1 MBPS bandwidth) connected to two hosts. One host is designated the client inside the firewall and the other is the server outside the firewall. Using iperf, we establish a TCP connection from the client to the server. The figure plots the bandwidth reported



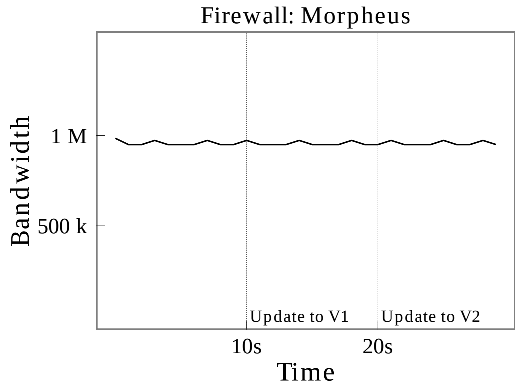


Figure 4: Firewall Upgrade

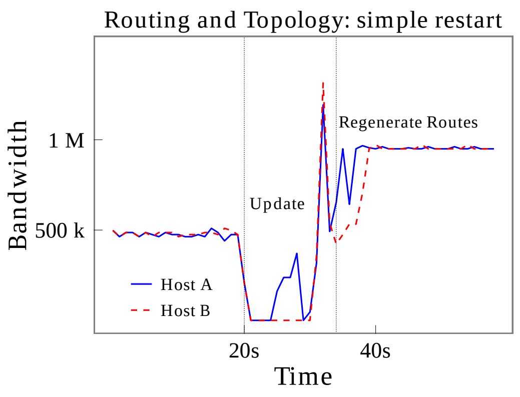
by iperf over time. In both experiments, we upgrade to FIREWALL$_{outin}$ after 10 seconds and FIREWALL$_{outinTO}$ after 20 seconds.

Using simple restart, the figure shows that bandwidth drops significantly during upgrades. This is unsurprising, since a newly started firewall doesn't 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 doesn't 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 routing and topology modules (described in section 4.4), where the initial version uses shortest paths and the final version takes current usage into account. The experiment uses four switches connected in a diamond-shaped topology with a client and server on either end. Therefore, there are two paths of equal length through the network. The client establishes two iperf TCP connections to the server.

Initially, both connections are routed along the same path because the first version of TOPOLOGY and ROUTING pick the same shortest path. The links along the



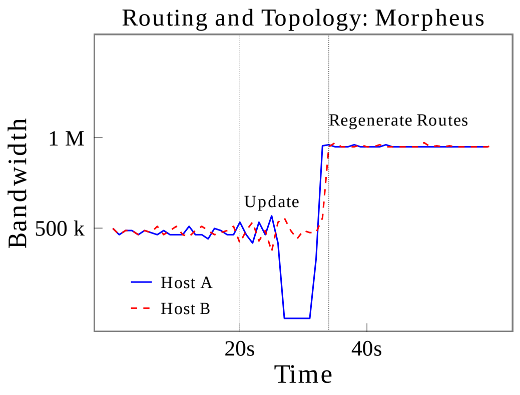


Figure 5: Routing and Topology Discovery Upgrade

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)

path are 1MBPS, therefore each connection gets 500K-BPS by fair-sharing. After 20 seconds elapse, we upgrade both modules: the new version of TOPOLOGY stores link-utilization information in the NIB and the new version of ROUTING using this information to balance traffic across links. After the upgrade, each connection should be mapped to a unique path, thus increasing link utilization and the bandwidth reported by iperf.

Using simple restart, both connections are disrupted for 10 seconds, which is how long TOPOLOGY takes to learn the network topology. Until the topology is learned, routing can't route traffic for either connection. Morpheus is much less disruptive. Since the state transfer 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. Even this disruption could be avoided using a consistent update [24].

Table 3 breaks down the time to run the upgrade protocol for this upgrade. It takes .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 .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 consists of 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 disconnecting clients, existing connections should not be moved after the upgrade.

As shown in the figure, both simple restart and record-and-replay cause disconnections, whereas state transfer causes no disruptions, since the state is preserved. 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.

5.4 Programmer Effort

Starting from a Morpheus module, there are two main additional tasks required to make it upgradeable: writing code to quiesce the module prior to an update, and writing a μ transformer function to change its state. In this subsection we discuss both tasks, showing that both are straightforward.

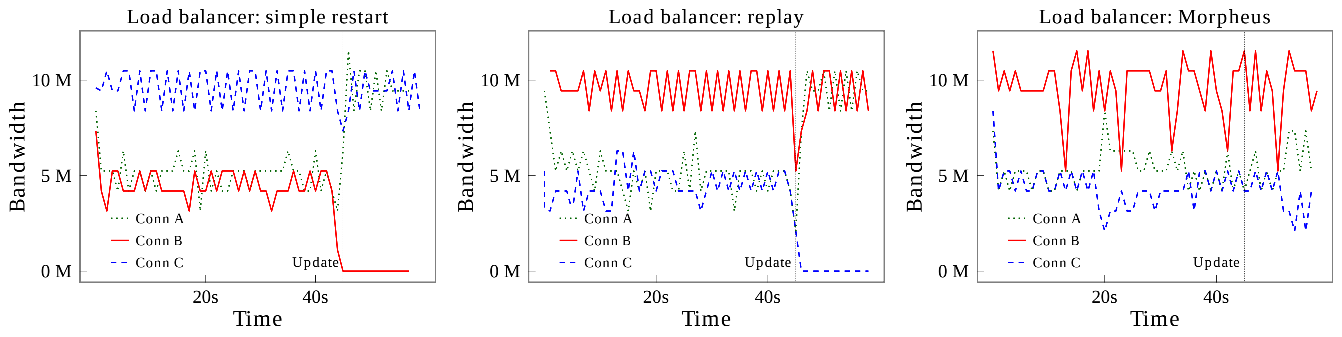


Figure 6: Load Balancer Results

Quiescence. The module developer must write code to check for notifications from the NIB that an upgrade is available, and if so to complete any outstanding tasks and gracefully exit. These tasks would include storing any additional state in the NIB and/or notifying external parties. For all of our examples, this work was quite simple, amounting to 8 lines of code.

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 consisted only of changes to the service logic, such as multiple changes to POX's IP load balancer in 2013. For them, no μ would be necessary. We also found that many of the changes involved adding state, or making small changes 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 to `True` in the NIB. To sum up, while the size of μ scales with the size of the change in state being made, in practice, we found that the effort to write μ is 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 argued this point extensively in Section 2, specifically comparing to alternative techniques involving controller restarts and record and replay (exemplified by the HotSwap system [31]). 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 in parallel with the old one, on different blade 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 [26, 27]. In general, most routing protocols have mechanisms to rebuild their state when the control software (re)starts (cf. [21, 25]), 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.[6] In addition, Morpheus is general-purpose (due to its focus on

[6]Cisco only supports ISSU between releases within a rolling 18-month window [10]. Outside of this window, a hard-reset

SDN), and not tied to a specific protocol design (e.g., a routing protocol).

Distributed Controllers. Distributed SDN controller architectures such as Onix [20], Hyperflow [30], ONOS [9] or Ravana [18] 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 [31]. Another line of work has explored using record and reply in the context of middleboxes [14, 28] and even in general-purpose operating systems [12]. 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) [17, 22, 16, 23], 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. [7], 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.[7] Our DSL for writing transformations draws inspiration from *xfgen*, a DSL provided by the Kitsune DSU system [16]; 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 of keys.

of the control-plane has to be done.

7. CONCLUSIONS

This paper has proposed *controller upgrade by state transfer* as a general-purpose approach to dynamically upgrade software-defined network controllers. The approach works by providing direct access to the relevant state in the running controller, and initializing the new controller's state as function of the existing state. This approach 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.

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] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *ECOOP*, July 2006.

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

[9] 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*, pages 1–6, 2014.

[10] 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`.

[11] Db-engines ranking. `http://db-engines.com/en/ranking`, 2015.

[12] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *USENIX OSDI*, pages 525–540, 2014.

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

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

[14] 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.

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

[16] 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.

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

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

[19] 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*, volume 10, pages 1–6, 2010.

[20] 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*. USENIX Association, Oct. 2010.

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

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

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

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

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

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