Boombox: A Dynamically-Updateable Network OS

Karla Saur
University of Maryland

Joseph Collard
University of Massachusetts Amherst

Nate Foster
Cornell University

Arjun Guha
University of Massachusetts Amherst

Laurent Vanbever
ETH Zurich

Michael Hicks
University of Maryland

Abstract

Like any non-trivial software system, SDNs periodically need to be updated to fix bugs, add features, and improve performance. A strawman approach would be to stop the controller, wipe the rules on the switches, and restart the controller, which then repopulates the switches with appropriate rules. But this approach loses any state maintained by the controller (and encoded in the switch rules), and worse, may cause the SDN to behave incorrectly; e.g., for stateful firewalls (whose policies are based in part on observed behavior), a previously blocked hosts packets may be let through.

This paper presents Boombox, a new SDN controller designed with network updates in mind. Unlike prior approaches based on recording controller events, and replaying them post-update to restore the controller state, Boombox allows the programmer to update the state directly, as kept in a persistent network information base (NIB). This approach, inspired by work on dynamic software updates, is both flexible and efficient. We demonstrate Boombox's effectiveness with experiments that show that existing flows' operation is only marginally disrupted during updates to several services, include a stateful firewall, a topology discovery engine, and a router.

1. Introduction

Software-defined network (SDN) controllers are highly-complex software. Among other things, they need to implement network services like topology discovery, routing, traffic shaping and do so in a highly-asynchronous environments. Like any non-trivial software program, a controller will have to be updated over time to fix bugs, add features, and improve performance. The question is: *How should we update controllers managing live networks?*

As discussed in detail in Section 2, existing approaches supporting live updates are inadequate. Simply wiping the rules on existing switches and restarting the controller, which would then repopulate the rules, is highly disruptive. Simply restarting the controller while leaving the switches alone is also problematic. The new controller may be unaware of the current switch state, and end up deploying new rules that conflict with existing ones, leading to routing loops and other signs of broken invariants. More advanced techniques, such as starting the new controller and replaying previously recorded events to it in an attempt to initialize its state [10] work in some cases but fail just as often, e.g., when recorded events do not match those the new controller would have actually seen, and when using distributed controllers.

This paper presents Boombox, a new SDN controller whose design draws inspiration from prior work on *dynamic software updating* (DSU) [7, 9]. As discussed in Section 3, Boombox implements network services as OS-level processes, and these services store their important state in a persistent *network information base* (NIB). Services may cooperate and communicate via state stored in the NIB. For example, a TOPOLOGY discovery service can store information about switches, links, and hosts it discovers, while a ROUTING service can use that information to compute (and also store) efficient routes. Importantly, if TOPOLOGY is shut down, the information in the NIB is retained, and is available to the service when it restarts.

As we describe in Section 4, this means that an update to a Boombox service simply requires shutting down the old version and starting the new one. One wrinkle is that the new version may expect a new representation of the persistent state. For example, the new TOPOLOGY may store current usage information along with each of its links. In this case, the update is installed along with a *transformation function* μ that takes as its argument an old representation and, from it, produces a new one. In Boombox, as with some prior work on DSU [9], this function is applied as needed, when the data is retrieved from the NIB. At present we expect a service programmer to write this function, but our general experience in Boombox and in prior work [7, 9] is that such functions are easy to construct.

We have written several services in Boombox, and several versions of each, implementing a stateful firewall, topology discovery, and routing. In Section 5 we show updates to Boombox services result in far less disruption than the equivalent updates, applied naively. Note that several of the updates we performed would not have been possible using record-and-replay techniques (as discussed in the next section). As such, we believe that Boombox represents a promising idea that should be considered in the design of future network controllers.

2. The problem with on-line updates

Network service updates are inevitable, but without care they will produce disruption, misbehavior, or both, as we explain in this section, motivating our approach.

2.1 Naive restart does not work

To perform an update via a naive restart, the network operator simply stops the old controller and starts the new one instead. While simple, this way of proceeding could be very disruptive, leading to both *service disruptions* and *correctness issues* [10]. In an attempt to avoid the latter, the first thing a controller usually does after establishing an OpenFlow connection with a switch is to send a command to delete the switch's flow entries. This is disruptive: In a network in which unmatched traffic is sent to the controller (the default in many OpenFlow switches), the newly started controller essentially induces a DDoS attack against itself.

Even if the disruption were acceptable, deleting the switches' existing flow entries does not completely avoid correctness issues. This is because important state kept by the old controller is lost on restart, and it may not be accurately recreated during the restart. For instance, depending on the ordering events seen by the new controller, perfectly legitimate traffic, allowed by the old controller, can end up being blocked by the new one. Or, even worse, forbidden traffic might end up being allowed [10].

2.2 Record-and-replay does not (always) work

HotSwap (HS) [10] aimed at enabling seamless controller update using record-and-replay. Prior to the controller upgrade, HS intercepts all the messages exchanged between the network devices and the controller and records the relevant ones (according to a filtering query provided by the operator). Following an update, HS replays the recorded messages to the new controller, attempting to induce the new controller to construct the state it would have had, had it started from scratch. During replay, HS uses the controller's output to effect necessary changes to the network forwarding policy.

Unfortunately, HS suffers from at least three problems which greatly reduce its applicability. Namely, HS can: *(i)* miss events; *(ii)* replay too many of them; and *(iii)* fail to capture meaningful controller semantics not implied by the event trace. The first two issues are likely to happen in practice as HS filtering queries must be written without knowing anything of the new controller behavior.

Problem #1: Missing events. HS's replay presumes that the new version of the controller, had it started from scratch, would have seen the same events as the old version. But this is not necessarily the case. In particular, flow entries installed by the old version may suppress events that otherwise would have been visible to the new version. Consider a simple stateful firewall application (call it FIREWALL$_{\leftharpoondown}$) that allows bi-directional flows between internal and external hosts as long as the connection is initiated from the inside. When the controller sees the outbound packet, it installs a rule that routes all subsequent packets, so the controller will never see them. But now imagine a firewall that works in two steps (call it FIREWALL$_{\rightleftharpoons}$): it does not install the rule after seeing only the first packet, but instead waits to actually see return traffic. (It might do this to prevent attacks on the forwarding table originating from a compromised host within the network.) Using HS to update from FIREWALL$_{\leftharpoondown}$ to FIREWALL$_{\rightleftharpoons}$ will fail because the new controller will never see events due to the return traffic, during replay. As such, the new controller's state will not be correctly initialized, so there will be allowed flows it does not know about (which it may, therefore, fail to time out).

Problem #2: Replaying too many events. In contrast, coarse-grained filtering queries can cause HS to replay events that should *not* be seen by the new controller. One example of such a case is when the new controller deals with some traffic proactively while the old one was reactive (e.g., updating from FIREWALL$_{\rightleftharpoons}$ to FIREWALL$_{\leftharpoondown}$). In such case, HS can replay traffic that should *never* go to the controller, leading to potential bugs.

Problem #3: Missing semantics. Finally, HS fails to capture any controller semantics that is not directly encoded in the recorded messages. For instance, consider a controller running a load-balancing application that equally splits flows among two replicas A and B upon seeing the first packet. As network load increases, the operator installs a third replica C and wishes to use HS to upgrade its controller while preserving existing flows. But replaying all previous first packets is just wrong as those flows should never be forwarded to C—only new flows should consider it.

2.3 Our approach in a nutshell

In essence, the goal of an update is to start the new controller with state that is equivalent to the state that was on the old controller. For the example firewall update, we would like the new version to know about (and grandfather in) any existing flows so it can manage them properly. HS tries to generate this state *indirectly*, by replaying recorded events. Since replay is not particularly reliable, our proposal is to construct the new controller's state *directly*, as a function of the old version's current state.

This idea is inspired by work on *dynamic software updating* [7, 9] (DSU), which updates a program on-the-fly by updating its code and then transforming the existing state to

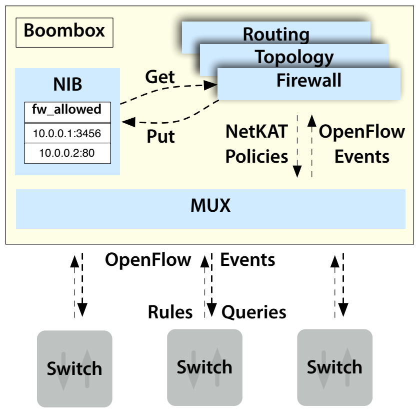

Figure 1. Example services: MUX, NIB, and FIREWALL.

be compatible with that code. As such, it suffers none of the problems of the replay-based approach: the new state can be constructed directly from the old state. For our firewall update, there is no issue when moving from the one-step procedure to the two-step one, or back again: the flows are known to the new version, and the new version can account for them when initializing its state. In the next two sections we explain Boombox, our network OS designed with DSU in mind.

3. Boombox: A Network OS

The Boombox controller architecture is depicted in Figure 1, illustrating its three main elements: the MUX, which manages interactions between the controller services and the switches; the NIB, which maintains persistent network state; and general services, like ROUTING, TOPOLOGY, etc., that implement the controller's various service functions. Each blue box in the figure is implemented as a separate OS-level process, ensuring they can execute concurrently, and in isolation. This design allows individual services to be killed and restarted without harming other services, a feature that supports fault tolerance and, as discussed in the next section, is useful for dynamic updates. Boombox's architecture also closely resembles the design used in production controllers at Google, VMware and others today [8]. As such, we believe that our dynamic software update techniques will be broadly applicable. The rest of this section discusses Boombox in detail, focusing on the essential services that are used in many applications.

3.1 MUX

Boombox's most basic component is the MUX, which acts as a simple multiplexor for other services. To switches in the network, the MUX behaves like a controller, accepting connections, negotiating features, responding to keep-alive messages, requesting statistics, (un)installing forwarding rules, etc. To other services, the MUX exposes a simple interface for interacting with the switches. When a service connects to the MUX, it can invoke several functions: `event()` returns the next OpenFlow event ; `update(pol)` sets the network configuration to `pol`, specified as a NetKAT policy [4]; `pkt_out(sw,pkt,pt)` injects packet `pkt` into the network at `sw` and `pt`; and so on.

Note that if there are multiple services connected to a given MUX it broadcasts OpenFlow events to *all* services, and it combines the policies provided by each service into a single policy using NetKAT's modular composition operators, which provide a simple means to resolve conflicts between different services [4]. In practice, for scalability and fault tolerance, the controller would deploy several MUXes that each manage a subset of the switches; for simplicity we use a single MUX in our examples.

3.2 NIB

Services store persistent state that is important for network functions in the *network information base*, or NIB. This information is retained when a service shuts down and restarts, thus avoiding disruption. For example, FIREWALL might store information in the NIB about which connections are currently allowed. The NIB provides a simple interface to a NoSQL key-value store, which can be used to store persistent state. NoSQL databases are a natural fit for SDN because most services do not require complicated queries or concurrency controls, but merely need to periodically share data. Boombox's store is based on Redis [3], but could use any number of other implementations such as Cassandra [1], MongoDB [2], or HyperDex [5].

Because several services may store state in the NIB, each service has its own conceptual *namespace*. 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 service's data values. A service is allowed to access data in multiple namespaces, where it might be the data owner for one, but a consumer of another. For example, our TOPOLOGY service discovers the network's structure by interacting with it via the MUX, and stores the topology information persistently in the `topology` namespace. Other services can query the data stored in this namespace to be informed when the topology changes.

3.3 Other services

Boombox also provides a suite of common services; we have already mentioned TOPOLOGY, and there is also ROUTING, and others. The common design pattern for such a service is to have it, upon startup, connect with the NIB to retrieve any relevant state possibly left by a previous version of the service. The service then adds to, and retrieves from, the persistent store as necessary 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.

As an example to illustrate Boombox's features, consider how we implement FIREWALL$_{\leftarrow}$, introduced in Section 2.2. The service defines a namespace `fw_approved` that identifies the allowed connections initiated by trusted hosts. Each mapped-to value in the namespace (encoded as a JSON value) defines the particulars:

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

On startup, the service reads in the last version of this list from the NIB (if no data is available in the NIB, it simply initializes the list to the empty list) and generates a NetKAT program that implements the firewall functionality in the current state. In this instance, the firewall would forward packets bidirectionally between existing connections but redirect other packets from internal hosts to the FIREWALL$\leftharpoondown$ service. This means that most traffic will be handled in the data plane, but new connections from internal hosts will be received by FIREWALL$\leftharpoondown$, which can update its list of approved connections in the NIB, generate a new NetKAT program, and invoke the *update* function in the MUX to update program the switches.

4. Updating services with Boombox

Boombox's design supports dynamic updates by allowing important state to persist between versions, and providing a way to transform that state, when needed, according to an upgrade.

4.1 Basic service updates

Service updates proceed in three steps. The first step is to (gracefully) kill the service application.

The second step is to install the update at the NIB. This involves logically advancing the version of the namespace of the to-be-updated data, and (if necessary) registering a *transformer function* μ with the NIB. This function is used to transform persistent data as required by the new version of the service. For example, one of our updates to the stateful firewall, discussed further below, requires timeout and packet count information; the transformer function augments existing allowed-flow data with two fields to track this added information. The service programmer writes μ in a domain-specific language we developed, for writing transformers over JSON values (inspired by Kitsune's *xfgen* language [7]), described below. This language's programs are compiled to Python code that takes an old JSON value and produces an updated version of it.[1]

The final step is to start the new version, which connects to the NIB and declares the namespaces it would like to access, and along with the expected version of the namespace. The NIB ensures that the service code's requested version matches the actual version; if it does not, the application is terminated. Otherwise it may proceed. Persistent data is transformed *lazily*, as the service application accesses it. In particular, when an application 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. While laziness can, in principle, reduce the delay in starting the new version, in practice many applications will read all of their persistent state immediately, so as to configure switches under their purview.

[1] While the programmer currently must write μ, automated assistance is also possible [7, 9].

4.2 Update example: Firewall

We developed three different versions of a stateful firewall: FIREWALL$\leftharpoondown$, which allows any connection initiated by a trusted host; FIREWALL$\rightleftharpoons$, which (as described in Section 2.2) only blesses a connection after seeing a return packet; and FIREWALL$\rightleftharpoons\odot$, which adds the ability to time out connections (and uninstall their forwarding rules) after some period of inactivity between the two hosts.

Updating from FIREWALL$\leftharpoondown$ to FIREWALL$\rightleftharpoons$ 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 update, no transformer function is needed: all connections established under the FIREWALL$\leftharpoondown$ regime can be allowed to persist, and new connections will go through the two-step process.[2]

Updating from FIREWALL$\rightleftharpoons$ to FIREWALL$\rightleftharpoons\odot$ 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 service 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$\rightleftharpoons$ to FIREWALL$\rightleftharpoons\odot$ 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 states that for every key in the namespace, its corresponding JSON value is updated from version `ns_v0` (corresponding to FIREWALL$\rightleftharpoons$) to `ns_v1` (corresponding to FIREWALL$\rightleftharpoons\odot$) 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. In general, our DSL can express transformations that involve adding, renaming, deleting field

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

names, modifying any data stored in the fields, and also renaming the keys themselves.

The above code will be compiled to Python code that is stored (as a string) in Redis and associated with the new version. The existing data will be transformed as the new version accesses it. In our example from Section 3.3, 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:

```
key (string): 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,}
```

4.3 Coordinated updates: 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 services. As such, the approach of killing the application, installing the update, and starting the new version will work just fine. However, in some cases we may wish to update a service whose data is being used by other services. If the data changes in a backward-incompatible manner then these services will need to be restarted to a version that understands the new format.[3]

In this case, we could kill all of the affected applications before installing the update, and then restart them all. We also support a mode such performs a version check at each lookup and update to Redis. If the version of the data has changed (implying it is no longer compatible with the application), the application can gracefully shut down. Ideally, a new version of the application will be available that can be immediately restarted (perhaps using an automated process).

As an example coordinated update, recall from Section 3 that our ROUTING and TOPOLOGY services share topology information stored in the NIB. In their first versions, TOPOLOGY merely stores information about hosts, switches, and the links that connect them. The ROUTING service 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 calculates the moving average of the bitrate on each link, which it stores 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 Boombox requires adding a field to JSON object for edges, to add the measured bitrate. The transformer μ simply initializes this field to 0, indicating that no traffic is on the link; as such, the initial run of the routing algorithm will reproduce the existing routes, ensuring stability. Subsequent ROUTING computations will account for the added usage information, and thus better balance the routes.

[3] This problem often motivates preserving the schema of the data so as to not disrupt existing clients, but we make no value judgment about when or whether this is a good idea.

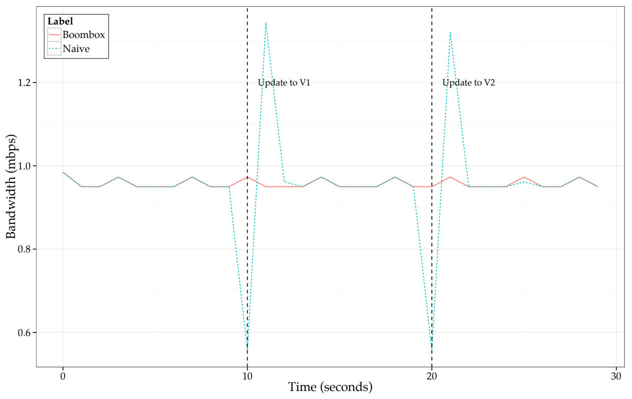

Figure 2. Firewall Update

5. Experiments and Evaluation

To demonstrate the benefits of Boombox, we performed experiments on our applications using both a naive approach of updating by killing and restarting (losing all state), and with the Boombox approach of persisting and updating state while restarting.

5.1 Firewall

In testing the firewall, we performed an update from FIREWALL$\leftharpoondown$ to FIREWALL$\rightleftharpoons$ and from FIREWALL$\rightleftharpoons$ to FIREWALL$\rightleftharpoons\odot$, first using a naive approach of just killing the first version and starting the next version (necessitating flushing all of the state on the switches in addition to losing all application state), and then we also used Boombox to handle the updates.

For our experiments, we created a single switch topology with two connected hosts, using Mininet HiFi [6] with 1 MBPS, 5ms delayed links. We set up a TCP iperf server session on one of the hosts, and had an iperf client session running on the other host. We ran FIREWALL$\leftharpoondown$ for 10 seconds, killed that version and ran FIREWALL$\rightleftharpoons$ for 10 seconds, and then finally killed that version and ran FIREWALL$\rightleftharpoons\odot$ for 10 more seconds. We ran this experiment on an Intel(R) Core(TM) i5-4250U CPU @ 1.30GHz with 8GB RAM, and report the average of 10 trials.

Figure 2 shows the results of our experiment. In the naive case, the bandwidth of the connection between the pair of hosts running the TCP iperf session drops to zero during the updates. After the update, the bandwidth spikes temporarily due to the extra delays incurred by the packets, and then returns to normal. In the Boombox case, the connection is not affected by the update and the communication between the two hosts continues as normal.

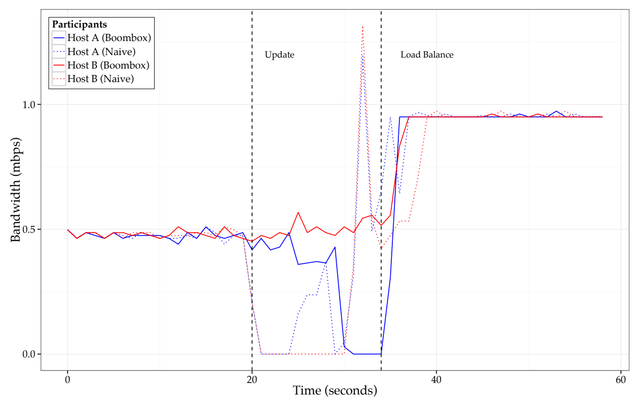

Figure 3. Routing and Topology Discovery Update

5.2 Routing and Topology

For this experiment, we set up four switches connected as a diamond using Mininet HiFi, with two hosts connected to one switch and a server connected to the switch on the opposite end. This topology creates the possibility for the server to communicate with hosts along multiple paths. All links in the topology are 1MBPS, except the link to the server is 10MBPS. We started a TCP iperf session at the server, and had the two clients connect to the iperf server. Using the initial version of TOPOLOGY and ROUTING, both hosts' traffic are directed along the *same* path, fairly sharing the 1MBPS available to them. We used the same machine specs for our routing and topology test as the firewall in the previous section.

Figure 3 shows both hosts' bandwidth during an update to TOPOLOGY and ROUTING. As described in Section 4.3, the new version of TOPOLOGY adds usage information to links, which ROUTING uses to compute more balanced routes.

In the naive case where we switch from the initial version of TOPOLOGY and ROUTING to the updated versions by wiping without maintaining state (and deleting rules on the switches), there is a big performance drop when the update takes place, at around the 20 second mark. The drop persists while the updated services work to rediscover internal links and the hosts. Once the routes are restored, TCP slowly restarts (over the course of about 10 seconds), followed by a brief burst of traffic before eventually settling at a higher bandwidth after load balancing, with Host A and Host B taking separate paths on the diamond to the server.

Boombox achieves the same effect of balanced load, but with far less disruption. During the load balancing phase around the 35 second mark, Host A drops to 0 bandwidth temporarily due to loss of connectivity during the reroute, before shifting to the higher bandwidth from the load balancing benefit. Host B experiences no drop in traffic during the update, and immediately benefits from the increased bandwidth due to Host A being on a different route.

6. Conclusions

This paper has presented Boombox, a new SDN controller designed to support non-disruptive dynamic updates to network services. The key idea in Boombox is to implement services as separate, restartable processes whose important information is kept in a separate persistent store. Updates can then be implemented by halting the existing process and starting the new one, which will transform the data to the needed format as needed. We have found that this approach is both natural and effective: it supports seamless updates to live networks, while prior approaches would result in disruption, incorrect behavior, or both.

References

[1] The apache cassandra project. `http://cassandra.apache.org/`.

[2] Mongodb. `http://www.mongodb.com/`.

[3] Redis. `http://redis.io/`.

[4] 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.

[5] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. In *SIGCOMM*, pages 25–36, 2012.

[6] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[7] 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.

[8] 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.

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

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