Nimble: Fast and Safe Migration of Network Functions

Sheng Liu
Microsoft Corporation
liusheng@microsoft.com

Michael K. Reiter
Duke University
michael.reiter@duke.edu

Theophilus A. Benson
Brown University
tab@cs.brown.edu

***Abstract*—Network function (NF) migration alongside (and possibly because of) routing policy updates is a delicate task, making it difficult to ensure that all traffic is processed by its required network functions, in order. Indeed, all previous solutions to this problem adapt routing policy only after NFs have been migrated, in a serial fashion. This paper proposes a design called Nimble for interleaving these tasks to complete both more efficiently while ensuring complete processing of traffic by the required NFs, provided that the route-update protocol enforces a specific property that we define. We demonstrate the benefits of the Nimble design using an implementation in Open vSwitch and the Ryu controller, building on both known routing update protocols and a new protocol of our design that implements specifically the needed property.**

I. Introduction

Stateful network functions (NFs) are a staple of modern networks. For example, network intrusion detection/prevention systems (NIDS/NIPS) are central to supporting secure and efficient operation of networks. For such NFs, the consequences of missing packets can be significant, and methods to sneak attacks past NIDS/NIPS to destination targets have a long history (e.g., [1], [2], [3]).

Network operators reconfigure network routing-policies daily, e.g., up to 20 times on average per day in a Tier-1 ISP [4]. The risk of traffic sidestepping NFs is particularly acute during routing-policy updates. Even if NFs remain in place during the update, packets that transition from a point upstream of the NF on the old routing path to a point downstream from the NF on the new routing path can result in an NF missing these packets. Routing-policy update algorithms that ensure *consistent update* (e.g., [5], [6], [7]) can guarantee that all traffic gets processed (again, when the NF does not move), for example by ensuring that each packet traverses either its old path in its entirety or its new path in its entirety.

If the NF migrates alongside the routing-policy update (e.g., the need to change one is caused by the need to change the other), though, then consistent update cannot, by itself, ensure complete processing of packets. Rather, ensuring that all packets get processed by their intended NFs requires additional coordination to both migrate NFs to their new locations and adjust routing policies in a way that ensures that all packets get processed. To our knowledge, existing works proposed to do so (e.g., [8], [9], [10], [11], [12]) coordinate these actions by performing network forwarding-state update strictly after NF migration is finished. While these techniques are capable of being used with arbitrary route-update protocols, their generality delays flow redistribution longer than necessary, possibly delaying rectification of the issue that required the routing-policy update in the first place.

In this paper we provide a method and its implementation, called Nimble, for interleaving routing-policy update and NF migration in a software-defined network (SDN), in a way that significantly reduces the latency to achieve both and without permitting packets to evade processing by NFs. Our technique works with any route-update protocol that implements a property we call *relaxed waypoint correctness*, which includes consistent-update protocols like CU [7] and SCC [5]. We confirm the sufficiency of relaxed waypoint correctness using model checking. Moreover, we design a route-update protocol "RWC" that enforces *only* relaxed waypoint correctness to show that Nimble can indeed exploit the gap between this property and traditional consistent-update semantics to more efficiently update the network in some cases.

As we will show, permitting both routing updates and NF migrations to proceed concurrently is a delicate endeavor. To synchronize these tasks as little as possible, Nimble leverages targeted buffering and packet marking in the network to coordinate packet processing with NF migration. The benefits to this approach are myriad, however, including lower latency for completion of both tasks and, depending on the routing-update protocol with which NF migration is being deployed and the circumstances requiring their update, reduced packet loss and/or reduced rule overhead in switches.

We have implemented Nimble on Open vSwitch [13] and the Ryu controller [14]. We evaluate implementations of Nimble building on CU [7], SCC [5], and the RWC protocol of our own design. We empirically compare Nimble to OpenNF [9] and SwingState [10], and demonstrate the benefits of our design in both FatTree and ISP topologies. To summarize, our contributions are as follows.

- We describe an algorithm for integrating NF migration with routing-policy updates in SDN networks to accomplish both with greater speed than performing them serially, as previous solutions do (Sec. IV).
- We define a condition of a routing-update protocol called *relaxed waypoint correctness* that, together with our NF migration algorithm, ensures that packets traverse their NF waypoints correctly. We confirm the sufficiency of this condition using model checking and provide an example route-update algorithm that implements (only) relaxed waypoint correctness without traditional consistent-update semantics (Sec. V).
- We demonstrate the benefits of our NF migration algorithm in empirical comparisons to alternatives (Sec. VII).

II. Related Work

NF migration in SDNs requires careful coordination between efficient routing-policy updates and NF migration. To ensure that all packets are processed correctly during NF migration, existing works [8], [9], [10], [11], [12] update network forwarding state strictly after NF migration is done. While some approaches [9], [11] use a centralized controller to reroute affected traffic from the old to new NF position, other works [8], [10], [12] tunnel traffic directly to the new NF position to reduce latency. An example of the former class is OpenNF [9], which uses a centralized SDN controller to coordinate NF migration with packet redistribution. The affected incoming packets arriving at old NF positions are buffered at the controller during NF migration to avoid packet loss. An example of the latter class is SwingState [10], which creates a tunnel between the old and new position of each NF. NF states are prepended to packets arriving at the old location and are forwarded to the new location. All of these works change network routing policy after NF migration is finished, which slows down traffic redistribution. Other works [15], [16] that deal with dynamic scaling of NFs either do not address the consistency issues that arise during routing-policy updates, or do not ensure NF *chain-wide* correctness (e.g., [17]).

Works offering consistent routing-policy update using SDN (e.g., [5], [6], [7], [18], [19], [20], [21]) do not consider NF migration. They assume the location of each NF is fixed. Most works provide either strong consistency that ensures that packets traverse either the old path or the new path [5], [7], [21] or guarantee specific properties (e.g., loop freedom) via weaker consistency [6], [18], [19], [20]. In this paper, we use SCC and CU as examples to update network forwarding state. SCC implements a property called *suffix causal consistency* [5] to ensure that each packet traverses either its old or new path. CU uses two-phase commit to perform rule updates atomically across the network.

Numerous works focus on virtual machine (VM) migration [22], [23], [24], [25]. Some works [22], [23] clone VM instances in their entirety, but they do not coordinate VM migration with the change of routing policies. Other works [24], [25] propose to migrate VMs along with the underlying virtual network, though they suffer from service disruption. Compared with these works, our algorithm achieves seamless reconfiguration by carefully coordinating routing-policy update and NF migration.

In our paper, NFs move along with their states. Some works [26], [27], [28] maintain an external data store such that states do not need to migrate during NF migration. StatelessNF [26] keeps all NF states in a standalone centralized store, while other works [27], [28] maintain states both locally and remotely for performance.

III. Framework and Goals

A. SDN Model

We adopt an SDN model, in which a *controller* deploys *rules* to distributed *switches* to implement routing policy.

a) Flows: As is standard, we define a *flow* to consist of packets with the same addressing information, i.e., IP 5-tuple. We denote the space of all possible such 5-tuples, and so the space of all flows, by $\mathbb{F}$, and the space of flows between switches or between a switch and the controller by $\mathbb{F}^* \subset \mathbb{F}$. When convenient, we treat a flow *f* as a set of all possible packets with addressing information defined by *f* and use $pkt \in f$ to denote a packet pkt with the addressing information of *f*.

b) Controller: The network has a logically centralized controller that is responsible for configuring the switches to update the route of each flow. The controller executes an SDN application consisting of a *route generator* and an *update scheduler* (see Fig. 1).

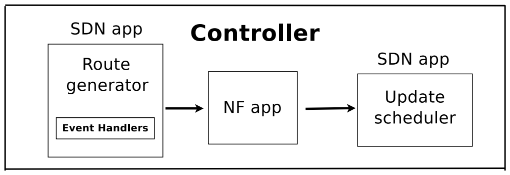


Fig. 1: Typical components of a network controller

The route generator decides whether to change the routes of flows by monitoring the network conditions and topology changes. The output of the route generator is the routing path of each flow *f*.

The SDN update scheduler produces *rules* (defined below) to be deployed on each switch, and schedules rule deployments to switches to preserve certain network routing properties when transitioning from the old routing policies to the new. Specifically, the update scheduler outputs a schedule of rule deployment in ℓ steps, denoted as $U_1, U_2, ..., U_\ell$. Each U_r includes a set of flowadd and flowdel commands to add or remove rules on switches; we discuss these commands and the structure of rules below. The commands in U_r are issued simultaneously to switches in the network, and updates of U_r are finished before U_{r+1} begins.

We refer to an invocation of the SDN application to reconfigure routing policy as a new *epoch*. We assume that each routing policy change completes—i.e., its rules are deployed throughout the network—before the next epoch begins.

c) Switches: Each switch maintains a flow table that stores rules (see below) for flow management. We denote the set of rules in the flow table of switch S as S.ruleSet. The controller modifies this set by invoking the following interface, which is similar to that of OpenFlow [29]:

- S.flowadd(R_j) inserts rule R_j into S.ruleSet.
- S.flowdel(R_j) removes rule R_j from S.ruleSet.

We assume that switches support *bundling*, a feature provided with OpenFlow version 1.4., i.e., that a set of invocations from the controller will collectively be performed as a single atomic transaction with respect to packet processing by the switch.

d) Rules: The instructions for how a switch should treat packets are specified by *rules*. When a packet arrives at a switch, it can be *matched* to at most one rule installed on the switch, which determines what happens to the packet. Each rule R includes (at least) the following fields, all immutable:

- R.switch specifies the unique switch S where R can be installed.
- R.priority specifies the priority of this rule, with higher priorities indicated by larger numbers, and with a special priority ∞ to represent the maximum priority, which is used only by our algorithm;
- R.cover $\subseteq \mathbb{F}$ specifies the flows to which this rule can be matched, i.e., a packet pkt can be matched to R only if $f \in R$.cover for the flow *f* containing pkt.
- R.sendTo specifies the switch identifier (in practice, an outbound port) to which packets matched to this rule should be forwarded.

B. Network Functions

Our goal is to extend the SDN model described above to support NFs. Recall that the SDN route generator in the controller produces new routing policies, as shown in Fig. 1. The output is also provided to an NF application to determine for each flow *f* the switch at which *f* will be processed by each of its waypoint NFs. This information is further provided to the update scheduler to reconfigure the network.

A *network function* NF_i is an object with an immutable field NF_i.flowSpec $\subseteq \mathbb{F}$ and a method NF_i.processPkt that takes as input a packet pkt in some $f \in NF_i$.flowSpec and outputs a (possibly empty) set of packets P, also in f. We assume that NF_i is linearizable [30] with respect to a *sequential specification* that specifies its correct behavior when NF_i.processPkt is invoked sequentially. Let NF_1, ..., NF_n denote all NFs.

C. Goals

Our goal is to update routing policies efficiently while ensuring packets are processed by NFs correctly. Most prior works on SDN routing-policy updates that achieve waypoint enforcement (e.g., [6], [7], [31]) do so assuming that NFs remain at fixed locations of the network during the routing-policy update. Instead, here we allow NF locations to change from one epoch to the next, and our contribution lies in ensuring that waypoint enforcement continues to hold.

To understand the challenges in permitting migration alongside route updates, consider the example in Fig. 2 where a network function NF_i migrates from the old position S_i^{old} to the new position S_i^{new}. The flow f, which should be processed by NF_i, also needs its path to be updated from $S_1 \rightarrow S_i^{\text{old}} \rightarrow S_2$ to $S_1 \rightarrow S_i^{\text{new}} \rightarrow S_2$. The path change can be accomplished by updating the rule matched to f at S_1. Migrating NF_i and updating the path of f without coordination can be harmful, however. For example, if the controller sends commands to migrate NF_i and updates S_1 simultaneously, S_1 might be updated before NF_i migrates to S_i^{new}. Then, packets of f might start to arrive at S_i^{new} before NF_i can process packets there, which may cause problems since packets can bypass NF_i. Also, if S_1 has not been updated by the time NF_i leaves S_i^{old}, packets may arrive at S_i^{old} with NF_i no longer there; depending on how S_i^{old} handles these packets, this could result in packet loss or packets bypassing NF_i.

Several works (e.g., [8], [9], [10]) have explored the possibility of migrating network functions in concert with routing-policy updates. A strength of these approaches is that they make no assumptions regarding properties of the underlying route-update algorithm, except that it eventually deploys the rules to faithfully implement routing policies. However, because these NF-migration algorithms must tolerate any transitory behavior of the underlying route-update algorithm, they necessarily must be conservative in how they migrate NFs, to ensure that no packets bypass their NF waypoints while the routing policy is updated. In fact, for this reason, all of them permit routing-policy updates to proceed only after all NF migrations have completed.

Our contribution here is an algorithm that leverages consistency properties of underlying routing-update algorithms to permit NFs to be migrated *alongside* rule deployment for new routing policies, more efficiently than simply serializing routing-policy update after NF migration. In particular, our approach demonstrates that by leveraging an SDN routing-policy update algorithm that provides a property that we call *relaxed waypoint correctness* (see Sec. V-A), we can implement waypoint correctness (defined in the next paragraph) when NFs are allowed to change locations much more efficiently than known approaches to achieving both.

Waypoint correctness: Let $[w] = \{1,...,w\}$. For each flow f, there is an injective function $\text{wp}_f : [n_f] \rightarrow [n]$ where $n_f \geq 0$ is the number of network functions that should process packets of f sequentially and $\text{wp}_f(k)$ is the k-th network function that packets in f must traverse. We require that if $\text{wp}_f(k) = i$, then $f \in NF_i$.flowSpec. Our correctness condition is that the network enforces the waypoint property, i.e., for any f and any packet pkt in f that enters the network,

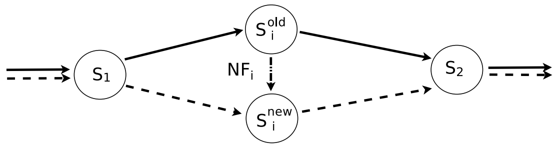


Fig. 2: Example of NF migration

- if $n_f > 0$, then NF_i.processPkt(pkt) is invoked for $i = \text{wp}_f(1)$;
- for each k, $1 \leq k < n_f$, if NF_i.processPkt(pkt') is invoked for $i = \text{wp}_f(k)$, outputting P, then $NF_{i'}$.processPkt(pkt'') is invoked for $i' = \text{wp}_f(k+1)$ and for every $pkt'' \in P$;
- no other invocations of any network function occur except by the above two rules; and
- if NF_i.processPkt(pkt') is invoked for $i = \text{wp}_f(n_f)$, producing output P, then every $pkt'' \in P$ is forwarded to its destination.

The first condition guarantees that if packets of f need to be processed by at least one network function, they must be processed by the first NF, $\text{wp}_f(1)$. Together with the first condition, the second condition ensures that packets of f are processed by network functions sequentially. The third condition prevents packets from being processed by network functions that are not specified. We use the last condition to guarantee the delivery of packets to the destination. Let $n_{\text{max}} = \max_f n_f$, i.e., n_{max} is the maximum number of waypoints that any flow can be required to traverse.

IV. Migratable Network Functions

Our framework uses an existing SDN route-update algorithm to generate rules and schedules to update routing policy.

A. Component Changes

To support NF migration, we require that the controller, switches, and rules be functionally enhanced in the following ways. Below we assume that each NF is hosted at a switch; this hosting could be implemented on the switch for a simple NF or at an attached middlebox for a more complex one.

a) Controller: The controller executes our algorithm that augments the SDN framework with two conceptual steps (elaborated in Sec. IV-B).

b) Switches: We add three new switch interfaces.

- S.export(i,j) marshals NF_i into a set P of packets with source address (the IP address) of S and destination address of S_j, and outputs P. S.export(i,j) executes only while no NF_i.processPkt invocations are underway at S, and S no longer permits invocations of NF_i.processPkt once S.export(i,j) completes.
- S.import(i,j) instructs switch S to await the arrival of packets P from S_j, from which to reconstitute function NF_i locally. This invocation causes S to allocate two buffers, an *inbound* buffer to hold packets to be processed in NF_i.processPkt invocations once NF_i is reconstituted locally, and an *outbound* buffer to hold packets output from NF_i.processPkt invocations. Starting from this invocation and until NF_i is reconstituted locally, packets matched to any $R \in S$.ruleSet for which R.sendTo $= NF_i$ (see

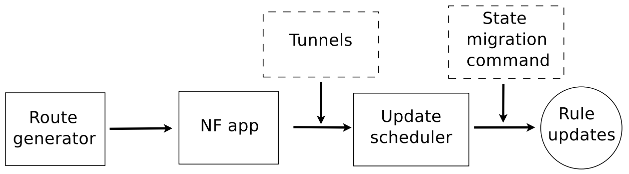


Fig. 3: Conceptual additions by our algorithm

below) are buffered in the inbound buffer for NF_i. Packets output from NF_i.processPkt invocations are buffered in the outbound buffer for NF_i, until a S.release(i) invocation.

- S.release(i) releases the packets in the outbound buffer for NF_i to be matched against S.ruleSet, and disables buffering so that packets inbound to or outbound from NF_i are no longer buffered.

S.import, S.export, and S.release can be invoked by the controller, just like S.flowadd and S.flowdel.

c) Waypoint counters: We add to each packet a field, called its *waypoint counter*, that can hold any value in $[n_{\mathsf{max}}+1] = \{1, \ldots, n_{\mathsf{max}}+1\}$. Upon arrival in the network, a packet's ingress switch initializes the packet's waypoint counter to 1. In brief, this counter is incremented in the packet as it is submitted to each of its waypoints for processing (see below). In this way, rules can treat a packet differently depending on how many of its waypoints it has already traversed.

d) Rules: We extend rules to include a new field R.wpCtr that takes on a value in $[n_{\mathsf{max}}] \cup \{*\}$, and stipulate that a packet can be matched to this rule only if R.wpCtr $= *$ or the packet's waypoint counter equals R.wpCtr. As such, when packet pkt in flow f arrives at switch S, pkt is matched to the highest priority rule $R \in S$.ruleSet for which $f \in R$.cover and either R.wpCtr $= *$ or the packet's waypoint counter equals R.wpCtr; we denote this rule as $match(S, pkt)$. If there is no $R \in S$.ruleSet to which pkt can be matched, then pkt is dropped. We also extend rules to accommodate additional functionality related to the R.sendTo field.

- R.sendTo can be a network function NF_i, in which case for any packet pkt it matches to R, $S = R$.switch increments the pkt's waypoint counter and then submits pkt to NF_i in an NF_i.processPkt(pkt) invocation. If NF_i is not hosted locally at R.switch, then the packet must be buffered (in the inbound buffer for NF_i, see above) until it is. Any packets returned from the NF_i.processPkt(pkt) invocation are matched again to S.ruleSet. We do not require NF_i to process packets' waypoint counters, but we do require that any packets NF_i.processPkt(pkt) emits bear the same waypoint counter as pkt.
- R.sendTo can take on two more possible values, namely functions encap[f] and decap. If a switch S matches packet pkt to rule $R \in S$.ruleSet where R.sendTo $=$ encap[f], then the packet is encapsulated into a packet for flow f, which is then resubmitted for matching against S.ruleSet at this switch S. A packet matched to a rule $R \in S$.ruleSet where R.sendTo $=$ decap is decapsulated (i.e., the existing packet header is removed) and the packet contained therein is then resubmitted for matching against S.ruleSet.

B. Algorithm

Our algorithm augments the SDN framework outlined in Sec. III with two conceptual steps (see Fig. 3). The first instantiates routing policy for tunnels to migrate NFs from their old locations to their new locations and to relocate traffic that arrives at an NF's old location to its new location. Once these routes have been determined, the full routing policy (including these new routes and the location of NFs) is then submitted to the routing-policy update scheduler, which produces the schedule for deploying rules to switches. The second phase of our algorithm then augments this update schedule with commands to bridge traffic on/off tunnels as needed, to invoke each NF with packets destined for it at its new location, and to initiate migration of NFs. A later phase of our algorithm (not shown in Fig. 3) cleans up the bridging rules once they are no longer needed.

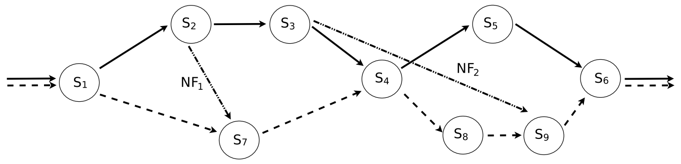


Fig. 4: Example for algorithm description

The first of these steps is implemented as follows, per NF_i that migrates from S_i^{old} to S_i^{new} in this epoch.

> **Migration routes**: The controller constructs a route from S_i^{old} to S_i^{new} to carry flows f_i^{mig}, f_i^{tun} with source S_i^{old} and destination S_i^{new}. f_i^{mig}, f_i^{tun} and their associated route are added to the routing policy that is input to the update scheduler.

f_i^{mig} will be used to migrate NF_i from S_i^{old} to S_i^{new}, and f_i^{tun} will be used to tunnel packets from S_i^{old} to S_i^{new} that should be processed by NF_i. Because we assume that the IP addresses of S_i^{old} and S_i^{new} are distinct from the source and destination addresses of flows routed according to the policies output from the route generator, the routes chosen to carry f_i^{mig}, f_i^{tun} cannot contradict the routes output from the route generator.

Fig. 4 shows an example for this step. Suppose a flow f, which is processed by NF_1 and NF_2, needs to be rerouted from the path $S_1 \to S_2 \to S_3 \to S_4 \to S_5 \to S_6$ (solid line) to the path $S_1 \to S_7 \to S_4 \to S_8 \to S_9 \to S_6$ (dashed line). Consequently, the controller decides to migrate NF_1 from S_2 $(= S_1^{\mathsf{old}})$ to S_7 $(= S_1^{\mathsf{new}})$ and NF_2 from S_3 $(= S_2^{\mathsf{old}})$ to S_9 $(= S_2^{\mathsf{new}})$. $S_2 \to S_1 \to S_7$ can be selected as the migration route between S_1^{old} and S_1^{new}. $S_3 \to S_4 \to S_8 \to S_9$ can be selected as the migration route between S_2^{old} and S_2^{new}.

Recall that the update generator now outputs a schedule for rule deployment in ℓ steps $U_1, U_2, \ldots, U_\ell$, where each U_r includes a set of flowadd and flowdel commands. Continuing with the example of Fig. 4, before adding migration routes by our algorithm, the update scheduler might formulate the following three-step schedule. In the first step, S_7, S_8, and S_9 install new rules; i.e., U_1 should include the flow modification commands for these three switches. In the second step (U_2), S_4 is updated to send packets to S_8. In the third step (U_3), S_1 is modified and packets are sent through the new path. By feeding migration routes, the first step U_1 should also contain extra commands on S_2, S_1 for migration of NF_1, and commands on S_3, S_4, S_8 for migration of NF_2.

Continuing our algorithm, it first sets R.wpCtr $\leftarrow *$ for any rule R in any flowadd or flowdel command in any step of the given update schedule $U_1, \ldots, U_\ell$, indicating that these rules ignore packet waypoint counters. Then, for each NF_i to be hosted at a switch

S_i^{new} in this epoch different from the switch S_i^{old} where it was hosted in the last, the controller performs the following steps.

Rules for routing to NF_i: The controller constructs additional rules as follows. First, the controller constructs a rule R_i^{enc} with the following fields:

- $R_i^{\mathsf{enc}}.\mathsf{switch} \leftarrow S_i^{\mathsf{old}}$
- $R_i^{\mathsf{enc}}.\mathsf{cover} \leftarrow \left\{ f \;\middle|\; \begin{array}{rr} \exists pkt, R: & pkt \in f \\ \wedge & R \in S_i^{\mathsf{old}}.\mathsf{ruleSet} \\ \wedge & R = match(S_i^{\mathsf{old}}, pkt) \\ \wedge & R.\mathsf{sendTo} = NF_i \end{array} \right\}$
- $R_i^{\mathsf{enc}}.\mathsf{wpCtr} \leftarrow *$
- $R_i^{\mathsf{enc}}.\mathsf{priority} \leftarrow \infty$
- $R_i^{\mathsf{enc}}.\mathsf{sendTo} \leftarrow \mathsf{encap}[f_i^{\mathsf{tun}}]$

In addition, the controller constructs the rule R^{dec} with the following fields:

- $R_i^{\mathsf{dec}}.\mathsf{switch} \leftarrow S_i^{\mathsf{new}}$
- $R_i^{\mathsf{dec}}.\mathsf{cover} \leftarrow \{f_i^{\mathsf{tun}}\}$
- $R_i^{\mathsf{dec}}.\mathsf{wpCtr} \leftarrow *$
- $R_i^{\mathsf{dec}}.\mathsf{priority} \leftarrow \infty$
- $R_i^{\mathsf{dec}}.\mathsf{sendTo} \leftarrow \mathsf{decap}$

Finally, for each $k \in [n_{\mathsf{max}}]$, the controller constructs a rule $R_{i,k}^{\mathsf{inv}}$ with the following fields:

- $R_{i,k}^{\mathsf{inv}}.\mathsf{switch} \leftarrow S_i^{\mathsf{new}}$
- $R_{i,k}^{\mathsf{inv}}.\mathsf{cover} \leftarrow \{f \mid \mathsf{wp}_f(k) = i\}$
- $R_{i,k}^{\mathsf{inv}}.\mathsf{wpCtr} \leftarrow k$
- $R_{i,k}^{\mathsf{inv}}.\mathsf{priority} \leftarrow \infty$
- $R_{i,k}^{\mathsf{inv}}.\mathsf{sendTo} \leftarrow NF_i$

In the example of Fig. 4, S_2 and S_3 should install rule R_1^{enc} and R_2^{enc}, respectively, to encapsulate packets of *f* onto f_i^{tun}. In this way, packets arriving at S_2 or S_3 can be relocated to new positions S_7 and S_9, respectively, during NF migration. Packets relocated through these tunnels to the new NF locations should be decapsulated back to the original flow *f* such that they can traverse the remainder of the new path after being processed by the appropriate NF. Therefore, S_7 and S_9 need rules R_1^{dec} and R_2^{dec}, respectively.

$R_{i,k}^{\mathsf{inv}}$ has two functions. First, it sends packets that need to be processed (i.e., with waypoint counter k where $\mathsf{wp}_f(k) = i$) to NF_i. Note that packets of *f* output from NF_i will have a waypoint counter of $k+1$ and thus will not be matched to $R_{i,k}^{\mathsf{inv}}$ again. Second, $R_{i,k}^{\mathsf{inv}}$ prevents packets from being processed twice by NF_i. Continuing with the example of Fig. 4, since at the beginning of U_3, S_4 has been updated (in U_2) but S_1 has not yet been changed, packets of *f* traversing a part of old path ($S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4$) and a part of new path ($S_4 \rightarrow S_8 \rightarrow S_9 \rightarrow S_6$) encounter both the old (S_3) and new position (S_9) of NF_2. $R_{2,2}^{\mathsf{inv}}$ at S_9 ensures packets carrying waypoint counter 2 can be processed by NF_2, but packets with waypoint counter 3 (NF_2 already processed these packets at S_3) are forwarded to next switch S_6 immediately (packets are matched by the rule enforcing the new routing policy).

Now that these rules have been generated, we need to integrate them into the update schedule. To do so, the algorithm initializes $U_{\ell+1}$ to be empty, i.e., $U_{\ell+1} \leftarrow \{\}$. Then, for each migrating network function NF_i, the controller performs the following.

Update schedule: To deploy R_i^{enc}, R_i^{dec}, $\{R_{i,k}^{\mathsf{inv}}\}_{k \in [n_{\mathsf{max}}]}$ in the update schedule U_1, U_2, ..., $U_{\ell+1}$, the controller performs the following steps.

- The controller adds $S_i^{\mathsf{new}}.\mathsf{flowadd}(R_i^{\mathsf{dec}})$, $S_i^{\mathsf{new}}.\mathsf{import}(i,j)$, and $S_i^{\mathsf{new}}.\mathsf{flowadd}(R_{i,k}^{\mathsf{inv}})$ for each $k \in [n_{\mathsf{max}}]$ to U_1, where $S_i^{\mathsf{old}} = S_j$.
- The controller searches for the last step U_r in which rules to route f_i^{mig}, f_i^{tun} are deployed. It adds $S_i^{\mathsf{old}}.\mathsf{flowadd}(R_i^{\mathsf{enc}})$ and $S_i^{\mathsf{old}}.\mathsf{export}(i,j)$ to U_{r+1} where $S_i^{\mathsf{new}} = S_j$.
- The controller adds $S_i^{\mathsf{new}}.\mathsf{release}(i)$ to $U_{\ell+1}$.

The first bullet incorporates commands to prepare switches at new positions for NF migration. In the example of Fig. 4, the controller issues $S_7.\mathsf{import}(1,2)$, $S_7.\mathsf{flowadd}(R_1^{\mathsf{dec}})$ and $S_7.\mathsf{flowadd}(R_{1,1}^{\mathsf{inv}})$ to S_7. The first of these commands instructs S_7 to wait for messages from S_2 and prepare to reconstruct NF_1 locally. The controller also performs similar operations on S_9.

The second bullet incorporates commands to migrate NF_i from its old to its new position. This should be done after the rules implementing the migration route have been deployed (i.e., after step U_r). In the example of Fig. 4, assume the controller deploys rules to create a tunnel $S_2 \rightarrow S_1 \rightarrow S_7$ to migrate NF_1 from S_2 to S_7 in step U_1. Then, in step U_2, the controller can use the interface $S_2.\mathsf{export}(1,7)$ to instruct S_2 to create a set of packets to marshal NF_1 and send these packets to S_7. Upon receiving packets from S_2, S_7 reconstructs NF_1 and starts to perform $NF_1.\mathsf{processPkt}$ invocations. Meanwhile, S_2 uses the rule R_1^{enc} to encapsulate the packets of *f* to packets of f_1^{tun}. Packets of f_1^{tun} are then forwarded to S_7 and S_7 uses the rule R_1^{dec} to decapsulate these packets back to packets of *f*. The packets have not been processed by S_2 and therefore should carry the waypoint counter 1. Thus, S_7 uses the rule $R_{1,1}^{\mathsf{inv}}$ to forward these packets to NF_1. After being processed by NF_1, the resulting packets are buffered at S_7 with the waypoint counter 2.

The last bullet ensures that packets released from switches at new positions can be matched to rules implementing new routing policy at all downstream switches. In the example of Fig. 4, the controller sends $S_7.\mathsf{release}(1)$ in step U_4. S_7 then releases the buffered packets to the network since S_4, S_8 and S_9 have installed rules to forward packets to the destination.

The last step of our algorithm cleans up migration-related rules once they will no longer be used. Specifically, for each migrated NF_i, the following is performed:

Bridging rules cleanup: After sufficient time passes to ensure that f_i^{mig} and f_i^{tun} will contain no more packets, the controller issues $S_i^{\mathsf{old}}.\mathsf{flowdel}(R_i^{\mathsf{enc}})$ and $S_i^{\mathsf{new}}.\mathsf{flowdel}(R_i^{\mathsf{dec}})$ commands.

In Fig. 4, this step causes the deletion of R_1^{enc} and R_1^{dec} from S_2 and S_7, respectively, and the deletion of R_2^{enc} and R_2^{dec} from S_3 and S_9, respectively. The rules to implement the migration routes $S_2 \rightarrow S_1 \rightarrow S_7$ and $S_3 \rightarrow S_4 \rightarrow S_8 \rightarrow S_9$ can also be removed, if doing so does not disrupt other routing policy.

V. Update Scheduling

The algorithm described in the previous section adapts a given update schedule to migrate NFs during path updates. In this section, we explore the requirements for the given update schedule that, when combined with the algorithm of the previous section, ensures waypoint correctness as defined in Sec. III-C. We define a sufficient condition in Sec. V-A, and then report on our use of model checking to confirm that this condition is sufficient in Sec. V-B. Finally, in Sec. V-C we provide an update scheduling algorithm that is tailored to implement specifically this condition.

A. A Sufficient Condition for Waypoint Correctness

In this section, we give a sufficient condition for the NF-migration algorithm of Sec. IV to ensure the waypoint correctness property defined in Sec. III-C. Recall that during a route change, each NF_i is migrated from its old location S_i^{old} to its new location S_i^{new}, while traffic to be processed by NF_i that arrives at S_i^{old} and matched to a rule R with $R.\mathsf{sendTo} = NF_i$ is transported from S_i^{old} to S_i^{new} to be processed once NF_i is reconstituted there. Whether traffic reaches S_i^{new} via this mechanism or by the new routing policy does not matter. Rather, all that really matters is that a packet on flow f with waypoint counter k reaches either $S^{\mathsf{old}}_{\mathsf{wp}_f(k)}$ or $S^{\mathsf{new}}_{\mathsf{wp}_f(k)}$. We call this property *relaxed waypoint correctness*:

Relaxed waypoint correctness: An update scheduling algorithm satisfies *relaxed waypoint correctness* if during any route update, it ensures that for each flow f and each $k \in [n_f]$, each packet on flow f with waypoint counter k reaches $S^{\mathsf{old}}_{\mathsf{wp}_f(k)}$ or $S^{\mathsf{new}}_{\mathsf{wp}_f(k)}$.

Our NF-migration algorithm in Sec. IV guarantees the waypoint correctness property defined in Sec. III-C, if the underlying update scheduling algorithm (used by the update scheduler in the controller) satisfies relaxed waypoint correctness. An example of an update scheduling algorithm that implements this property is CU [7], which on its own ensures that each packet traverses either the old path in its entirety or the new path in its entirety. When conjoined with our NF migration algorithm, a packet that is being routed along its old path might be tunneled from S_i^{old} to S_i^{new} for processing by NF_i, after which it will be buffered until the route update is complete. From that point forward, it will be routed along its new path. A natural question is whether there are route update algorithms that satisfy relaxed waypoint correctness without enforcing a packet to traverse only the path of its old flow or of its new one. In Sec. V-C we answer this question in the affirmative.

B. Model Checking

We applied model checking to confirm that our algorithm of Sec. IV enforces waypoint correctness described in Sec. III-C. We used Z3Py [32], a Python API for the Z3 solver [33], to model the underlying route-update algorithm and packet redistribution, and let Z3 verify waypoint correctness. We constructed our model with fifteen switches in a mesh topology and with three flows. Each routing path was eight switches and each path contained three NFs. We modeled the effects of one new epoch that implemented a routing policy with NF migration (i.e., each NF was moved to a different position) using rules in the old and new configuration.

The underlying route-update algorithm deployed rule updates on switches (i.e., deleting unused rules and installing new rules) in at most $\ell = 15$ steps, with each switch being updated at most once. Each switch utilized either old or new rules to process packets based on the step during which it received those packets. For example, if S_1 was scheduled to be updated in step U_3, S_1 used an old rule R_1 to match f_1 before U_3 began and a new rule R_2 after U_3 completed. During U_3, to model unknown delays for switch updates to occur, either R_1 or R_2 was used to match f_1 nondeterministically. The step at which packets were received by each switch through the network was non-decreasing. Moreover, the update schedule generated by the underlying route-update algorithm ensured relaxed waypoint correctness. Z3 explored all possible rule-update schedules constrained by the above conditions to enforce new routing policy of each flow. As such, the model checked was not dependent on any specific route-update protocol, but rather permitted any route-update strategy as long as it satisfied these properties.

Our algorithm incorporated NF migration into the rule-update schedule generated by the underlying route-update algorithm and used tunnels to redistribute packets from old NF locations to new ones. For simplicity, the correctness of tunnels was assumed, and tunnels were not modeled explicitly. Specifically, when a packet on flow f arrived at S_i^{old} and was matched to rule R_i^{enc}, the packet was delivered to S_i^{new}, as if through the tunnel.

To model the delay caused by buffering packets at new NF locations, the specific step at which S_i^{new} forwarded packets using its new rule should not be earlier than the step at which $S_i^{\mathsf{new}}.\mathsf{release}(i)$ is invoked. For example, if a packet arrived at S_i^{old} for NF_i at step U_3 and then was forwarded through the tunnel to S_i^{new}, S_i^{new} could not release packets until $S_i^{\mathsf{new}}.\mathsf{release}(i)$ was deployed at step U_5. We let the Z3 solver explore all possible delays before S_i^{new} released packets and check for violations of waypoint correctness. In previous, incorrect versions of our algorithm, this model checking revealed corner cases that resulted in property violations. For the algorithm presented in the previous sections, however, after running about one day on a 32-core, 2.1GHz computer with 256GB of memory, the model checker successfully terminated and found no violations.

C. Update Scheduling for Relaxed Waypoint Correctness

In this section we provide an update scheduling algorithm, which we call RWC (for "relaxed waypoint correctness"), that is specifically designed to satisfy relaxed waypoint correctness, no more, no less, whenever it is possible to achieve this property while updating each switch only once during an epoch change. The algorithm is inspired by the TSU [6] routing update algorithm, though we have adapted it to accommodate NF migration and waypoint ordering.

RWC computes the update schedule U_1, U_2, ..., U_ℓ using a 0-1 integer program (IP). As we will show, the update schedule, once computed, enables Nimble to complete faster in some cases than schedules produced by consistent-update algorithms such as CU, thereby demonstrating a gap that Nimble can exploit between relaxed waypoint correctness and typical consistent-update properties. This IP can be solved (if it has a solution) using solvers like CPLEX or Gurobi, though not in time polynomial in the network size. As such, RWC serves primarily to reveal this gap or, in large networks, to produce an update schedule only as a precomputation step, off the critical path of updating the network.

As presented here, RWC assumes that the old and new routing policies differ only in a single path; i.e., a flow f (or set of flows) transitions from one old path to one new path. (Multiple path changes

Minimize ℓ' subject to:

$$
\begin{array}{rlr}
\ell' \geq r \cdot x_j^r & \forall r \in [m], S_j \in \mathbb{S}^{\pm} & (1) \\
1 = \sum_{r \in [m]} x_j^r & \forall S_j \in \mathbb{S}^{\pm} & (2) \\
y_{j,j'}^r = 1 - \sum_{r' \leq r} x_j^{r'} & \forall r \in [m], (S_j, S_{j'}) \in \mathbb{L}^- & (3) \\
y_{j,j'}^r = \sum_{r' \leq r} x_j^{r'} & \forall r \in [m], (S_j, S_{j'}) \in \mathbb{L}^+ & (4) \\
y_{j,j'}^r = 1 & \forall r \in [m], (S_j, S_{j'}) \in (\mathbb{P}^{\mathsf{old}} \cup \mathbb{P}^{\mathsf{new}}) \setminus (\mathbb{L}^+ \cup \mathbb{L}^-) & (5) \\
z_{j'}^{r,k} \geq z_j^{r,k} + y_{j,j'}^{r-1} - 1 & \forall r \in [m], k \in [n_f], S_{j'} \in \mathbb{S}, S_j \in \mathbb{S} \setminus \{S_{\mathsf{wp}_f(k)}^{\mathsf{old}}, S_{\mathsf{wp}_f(k)}^{\mathsf{new}}\} & (6) \\
z_{j'}^{r,k} \geq z_j^{r,k} + y_{j,j'}^{r} - 1 & \forall r \in [m], k \in [n_f], S_{j'} \in \mathbb{S}, S_j \in \mathbb{S} \setminus \{S_{\mathsf{wp}_f(k)}^{\mathsf{old}}, S_{\mathsf{wp}_f(k)}^{\mathsf{new}}\} & (7) \\
z_j^{r,k} = 1 & \forall r \in [m], k \in [n_f], S_j \in \{in(f)\} & (8) \\
z_j^{r,k} = 0 & \forall r \in [m], k \in [n_f], S_j \in \{out(f)\} & (9) \\
z_j^{r,k+1} \geq z_j^{r,k} & \forall r \in [m], k \in [n_f - 1], S_j \in \mathbb{S} & (10)
\end{array}
$$

Fig. 5: RWC integer program for generating update schedule

can be implemented one-by-one in multiple updates.) Moreover, our presentation assumes that both the old and new paths are loop-free.

1) Integer program: Let $\mathbb{S}^{\mathsf{old}}$ be the set of switches that appear on the old path; $\mathbb{S}^{\mathsf{new}}$ the set of switches that appear on the new path; $\mathbb{S} = \mathbb{S}^{\mathsf{old}} \cup \mathbb{S}^{\mathsf{new}}$; $\mathbb{P}^{\mathsf{old}} \subseteq \mathbb{S}^{\mathsf{old}} \times \mathbb{S}^{\mathsf{old}}$ the links comprising the old path; and $\mathbb{P}^{\mathsf{new}} \subseteq \mathbb{S}^{\mathsf{new}} \times \mathbb{S}^{\mathsf{new}}$ the links comprising the new path. Therefore, $\mathbb{P}^{\mathsf{old}} \setminus \mathbb{P}^{\mathsf{new}}$ is the set of links that will be *disabled* by the path change (i.e., that will no longer be traversed by the rerouted flow *f*), and $\mathbb{P}^{\mathsf{new}} \setminus \mathbb{P}^{\mathsf{old}}$ is the set of links that will be *enabled* by the path change. Let $\mathbb{S}^{\pm} \subseteq \mathbb{S}^{\mathsf{old}} \cap \mathbb{S}^{\mathsf{new}}$ contain the switches at which links to carry *f* must be both enabled and disabled, i.e., $S_j \in \mathbb{S}^{\pm}$ iff $S_j \in \mathbb{S}^{\mathsf{old}} \cap \mathbb{S}^{\mathsf{new}}$ and for some $S_{j'}$, $(S_j, S_{j'}) \in (\mathbb{P}^{\mathsf{old}} \setminus \mathbb{P}^{\mathsf{new}}) \cup (\mathbb{P}^{\mathsf{new}} \setminus \mathbb{P}^{\mathsf{old}})$. Let $\mathbb{L}^+$ be the new links enabled at the switches in $\mathbb{S}^{\pm}$, and let $\mathbb{L}^-$ bet the old links disabled at the switches in $\mathbb{S}^{\pm}$; i.e., $(S_j, S_{j'}) \in \mathbb{L}^+$ iff $S_j \in \mathbb{S}^{\pm}$ and $(S_j, S_{j'}) \in \mathbb{P}^{\mathsf{new}} \setminus \mathbb{P}^{\mathsf{old}}$, and $(S_j, S_{j'}) \in \mathbb{L}^-$ iff $S_j \in \mathbb{S}^{\pm}$ and $(S_j, S_{j'}) \in \mathbb{P}^{\mathsf{old}} \setminus \mathbb{P}^{\mathsf{new}}$. For a natural number w, let $[w] = \{1, ..., w\}$.

The optimization, shown in Fig. 5, minimizes the number ℓ' of update steps subject to constraints (1)–(10). x_j^r is a binary variable signaling if switch $S_j \in \mathbb{S}^{\pm}$ is updated in step r; i.e., if the solution to the integer program has $x_j^r = 1$, then the controller will include its updates to S_j in U_r. Constraint (2) ensures that each switch in $\mathbb{S}^{\pm}$ is updated exactly once. m is the number of switches.

The binary variable $y_{j,j'}^r$ for each $(S_j, S_{j'}) \in \mathbb{P}^{\mathsf{old}} \cup \mathbb{P}^{\mathsf{new}}$ indicates whether the rerouted flows will be forwarded directly from S_j to $S_{j'}$ as of the end of update U_r. Constraint (3) ensures that $y_{j,j'}^r = 0$ once link $(S_j, S_{j'}) \in \mathbb{L}^-$ has been disabled, and constraint (4) ensures that $y_{j,j'}^r = 1$ once link $(S_j, S_{j'}) \in \mathbb{L}^+$ has been enabled. Constraint (5) ensures that $y_{j,j'}^r = 1$ for any other link in $\mathbb{P}^{\mathsf{old}} \cup \mathbb{P}^{\mathsf{new}}$.

The binary variable $z_j^{r,k}$ indicates if a packet on the rerouted flow *f*, upon reaching switch j after the end of update $r - 1$ and before the end of update r, has yet to be processed by NF_i where $i = \mathsf{wp}_f(k)$. Constraints (6) and (7) ensure that if $y_{j,j'}^{r-1} = y_{j,j'}^r = 1$ and so the packet is forwarded directly from S_j to $S_{j'}$, and if the packet was not yet processed by NF_i upon reaching S_j (i.e., $z_j^{r,k} = 1$), then it still remains to be processed upon reaching $S_{j'}$ (i.e., $z_{j'}^{r,k} = 1$). Of course, this reasoning is valid only if $S_j \notin \{S_i^{\mathsf{old}}, S_i^{\mathsf{new}}\}$; if $S_j = S_i^{\mathsf{old}}$ then the packet will be processed by NF_i there, and if $S_j = S_i^{\mathsf{new}}$ then the packet will be buffered at S_j awaiting NF_i. Therefore, constraints (6) and (7) are included only for $S_j \notin \{S_i^{\mathsf{old}}, S_i^{\mathsf{new}}\}$. Constraints (8) and (9) indicate that the packets of flow *f* have yet to be processed by NF_i upon their arrival at their ingress *in*(*f*) and must be processed by NF_i upon departing the network at their egress *out*(*f*). Finally, constraint (10) ensures that if a packet has yet to be processed by NF_i for $i = \mathsf{wp}_f(k)$, then it also has yet to be processed by $NF_{i'}$ for $i' = \mathsf{wp}_f(k+1)$.

2) Generating the update schedule: Given a solution to the integer program of Fig. 5, the update scheduler generates the update schedule as follows. We assume that $\mathbb{S} \setminus ((\mathbb{S}^{\mathsf{old}} \cap \mathbb{S}^{\mathsf{new}}) \setminus \mathbb{S}^{\pm})$ is the set of switches at which the new rules $\mathcal{R}^{\mathsf{new}}$ to implement the new routing policy differ from the rules $\mathcal{R}^{\mathsf{old}}$ already deployed to the network.

- For each $S_j \in \mathbb{S}^{\mathsf{new}} \setminus \mathbb{S}^{\mathsf{old}}$, the update scheduler adds $S_j.\mathsf{flowadd}(R)$ to U_1 for each $R \in \mathcal{R}^{\mathsf{new}} \setminus \mathcal{R}^{\mathsf{old}}$ for which $R.\mathsf{switch} = S_j$.
- For $S_j \in \mathbb{S}^{\pm}$ for which $x_j^r = 1$, the update scheduler adds $S_j.\mathsf{flowadd}(R)$ to U_{r+1} for each $R \in \mathcal{R}^{\mathsf{new}} \setminus \mathcal{R}^{\mathsf{old}}$ for which $R.\mathsf{switch} = S_j$, and $S.\mathsf{flowdel}(R)$ to U_{r+1} for each $R \in \mathcal{R}^{\mathsf{old}} \setminus \mathcal{R}^{\mathsf{new}}$ for which $R.\mathsf{switch} = S_j$.
- For $S_j \in \mathbb{S}^{\mathsf{old}} \setminus \mathbb{S}^{\mathsf{new}}$, the scheduler adds $S_j.\mathsf{flowdel}(R)$ to $U_{\ell'+2}$ and for each $R \in \mathcal{R}^{\mathsf{old}} \setminus \mathcal{R}^{\mathsf{new}}$ for which $R.\mathsf{switch} = S_j$.

After we obtain the update schedule U_1, U_2, ..., U_ℓ ($\ell = \ell' + 2$), it can then be turned over to the algorithm of Sec. IV-B for adaptation as prescribed there.

VI. Implementation

We implemented our NF migration algorithm (Sec. IV) using Open vSwitch [13] and the Ryu controller [14]. Packets' waypoint counters were stored in six bits of the VLAN tag, permitting up to $n_{\mathsf{max}} = 2^6 - 2$ waypoints per flow. PRADS [34] was used as network functions and was modified to provide APIs for migration. We implemented three route-update algorithms, namely SCC [5], CU [7], and RWC, to generate rule-deployment schedules. We incorporated our state migration algorithm into the schedule updater as described in Sec. IV to achieve waypoint correctness.

A. Route-Update Algorithms

a) SCC: In SCC [5], each packet carries a packet timestamp that may be changed by the rules to which it is matched as it traverses switches in the network. Each switch receiving a packet finds the matching rule, checking that this rule is recent enough to match to this packet by comparing the packet's timestamp with the rule's timestamp. To implement relaxed waypoint correctness, the algorithm performs updates in two steps. First, each ingress switch is updated so that packets can be matched only to rules consistent with the new routing policy. Then the remaining switches are updated concurrently to forward packets through new paths. Packets are buffered as needed at switches awaiting new rules.

Our implementation leverages unused header bits, namely six bits of the VLAN tag, to store the timestamp in each packet. We modified Open vSwitch (OVS) to extract these bits from each packet, to compare the packet timestamp with rule timestamp, and to set these bits based on the action of the rule to which it is matched. If the timestamp of the matching rule is smaller than the packet timestamp, then this packet is buffered awaiting more up-to-date rules. To buffer packets, we connected each OVS with a local Ryu controller (running on the switch) that is in charge of updating rules for this switch. Instead of buffering packets itself, OVS forwards packets to the local controller. A globally centralized controller running our algorithm uses a RESTful API to issue rule modification commands to the local controller. Then the local controller updates OVS using OpenFlow and also sends buffered packets back to the switch when appropriate.

b) CU: Consistent Update (CU) first deploys, but does not yet enable, rules with a new timestamp at all switches. Then, after all the new rules have been installed, the controller updates the ingress switch to start tagging packets with the new timestamp to allow each downstream switch to apply the new configuration. Therefore, CU makes each packet traverse either its old or new path in its entirety, and in this way enforces relaxed waypoint correctness.

Like in SCC, our implementation leverages six bits of the VLAN tag to store the CU timestamp in each packet. However, unlike SCC, the value of the rule timestamp must be equal to the value of timestamp carried by the packet in order to match to this packet. Also, since CU does not need to buffer packets, local controllers are not required. A centralized controller running the CU algorithm uses OpenFlow to directly issue rule modification commands to OVS.

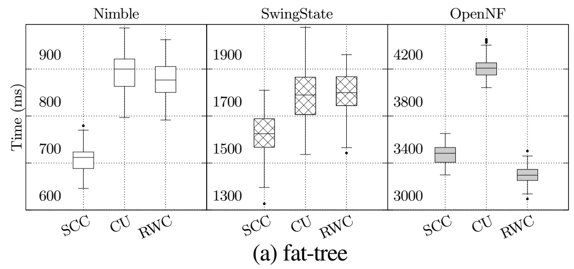
c) RWC: We solved the optimization in Fig. 5 with Gurobi [35]. The controller runs RWC and deploys updates in multiple steps.

B. NF Migration

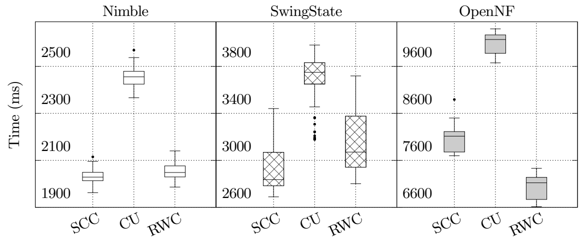
We instantiated network functions using the Passive Real-time Assets Detection System (PRADS). PRADS passively listens to network traffic and gathers information about hosts and services sending traffic. We modified PRADS to permit import/export of portions of its state (similar to OpenNF [9]), such as per-flow statistics. After receiving the S_i^{old}.export command, S_i^{old} exported the relevant PRADS state and crafted packets on f_i^{mig}. Each PRADS instance executed on a host directly connected to S_i^{old} or S_i^{new}. S_i^{old} and S_i^{new} were in charge of forwarding packets to the PRADS instance. To implement encapsulation and decapsulation, we used the IP tunnel command to configure Generic Routing Encapsulation (GRE) tunnels on each host. Moreover, to guarantee that packets carrying NF state are delivered to destination NF instances, a TCP connection was used. S_i^{new} used the local controller to buffer packets until receiving a $S.\mathsf{release}(i)$ invocation.

VII. Evaluation

Our experiments were conducted on topologies emulated in Mininet [36] on a 32-core, 2.1GHz computer with 256GB of memory. We used a fat-tree topology with $K=8$ ports per switch, and one ISP topology (Forthnet from Topology Zoo [37]) for these tests. The $K=8$ fat-tree contained 80 switches, and IP addresses were assigned as prescribed by Al-Fares et al. [38]. The Forthnet topology contained 62 switches and 62 links. To simulate the delay between the controller and switches, we randomly chose the position of the controller and computed the path from the controller to each switch



(a) fat-tree



(b) Forthnet

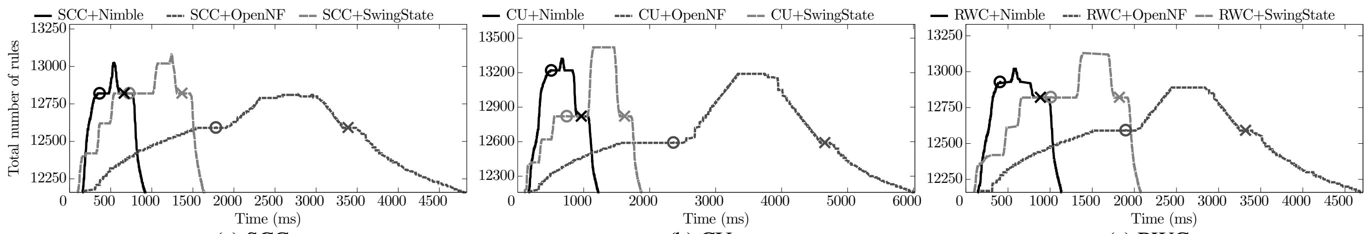
Fig. 6: NF migration and path change times

using a spanning tree protocol. The delay for each hop was measured using a simple topology with one OVS switch and two hosts sending ping packets through it. Specifically, the delay between each switch and the controller was computed as $db+dh\times h$ where db is the control path delay measured by Huang et al. [39] for a setting similar to ours, dh is the delay for one hop, and h is the number of hops. db was sampled from a normal distribution with mean 32ms and standard deviation 5.1ms and dh from normal distribution with mean 3ms and standard deviation 0.3ms. To create realistic path changes on the fat-tree networks, we replayed a log of route changes collected from Facebook's network [40]. For Forthnet, we used shortest-path routing and induced route changes by breaking links. In each case, NFs were reassigned from the old path to the new path randomly but constrained to appear on the new path in waypoint order.

We compared Nimble to OpenNF [9] and SwingState [10] over three route-update algorithms, namely SCC [5], CU [7] and RWC (Sec. V-C), based on our own implementation of each. SwingState migrates an NF over a tunnel between its old and new locations. OpenNF utilizes the centralized controller as a relay node to transfer NF states and redistribute incoming packets. Both SwingState and OpenNF separate state migration from path change. In each comparison, random values were sampled using the same random seed (or set of seeds) across each pair of state-migration and route-update algorithm; i.e., random link failures, packet sizes, NF locations were the same for each algorithm pair.

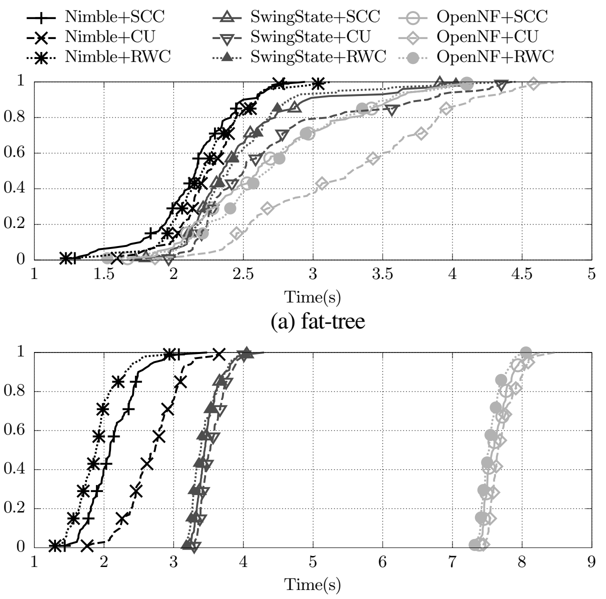
A. NF Migration and Path Change Time

We first measured the performance of these algorithms for NF migration and path change. Each evaluation involved 100 runs, in which hosts sent 100 packets per second for each flow. Fig. 6 demonstrates the times required to finish both NF migration and path changes; Fig. 6a shows times for 100 path changes with two NF migrations per path change in a fat-tree topology ($K=8$), and Fig. 6b shows times for 178 path changes with three NF migrations per path change in the Forthnet topology, induced by breaking its "busiest" link carrying the most flows. Each boxplot in Fig. 6a and Fig. 6b represents 100 points, i.e., one per run. The box marks the first, second



(a) SCC (b) CU (c) RWC

Fig. 7: Rules in the network during 100 path changes and accompanying NF migrations for fat-tree topology; markers show completion of path changes (×) and NF migrations (○)



(a) fat-tree

(b) Forthnet

Fig. 8: Times for receiving 1MB upon link failure

(median), and third quartiles, and whiskers extend to cover points within $1.5\times$ the interquartile range. Outliers are shown as dots.

As can be seen in these figures, Nimble performed much faster than SwingState and OpenNF, since NF migration and path change were executed simultaneously. OpenNF required much more time to perform the updates because it uses a single controller to buffer and redistribute packets. Upon receiving a large number of incoming packets, the controller consumed a lot of resources to process these packets, which significantly slowed down the rule-update process.

B. *Memory Overhead in Switches*

To evaluate the number of rules (including rules to build tunnels) imposed on the switches by each algorithm, we examined the per-switch logs of rule installations and deletions. We computed a time series of the total number of rules installed across all switches in the network, including the time cleaning up tunnels used to migrate NFs and to tunnel traffic. This time series for a representative run of each of SCC, CU, and RWC using Nimble, SwingState and OpenNF on the fat-free topology is shown in Fig. 7. (We obtained similar results on Forthnet [41], which are not shown due to space constraints.) Each curve is marked with the time when NF migration for all flows was done and the time when all path changes were completed. All three algorithms require installing rules per network function NF_i on S_i^{old} and S_i^{new} to deal with incoming packets. Unlike OpenNF, Nimble and SwingState also need to build tunnels to migrate NFs and to tunnel traffic which results in a larger number of rules. Though OpenNF required fewer extra rules on switches, it induced a significantly larger delay for NF migration and path change to be finished.

C. *Recovery from Link Failure*

We conducted two tests of each algorithm's ability to sustain networked applications across link failures. The first used the fat-tree topology ($K=8$) with the capacity of each link set to $300\text{MB}/\text{sec}$. We launched 448 TCP connections, each sending packets according to a Poisson process. The size of each packet was sampled from a distribution of packet sizes across all applications in a Google datacenter (the W3 workload shown in Montazeri et al. [42, Fig. 1]). The parameter of the Poisson process was then chosen so that the busiest link averaged 50% utilization. We broke one random link and selected all affected flows to update their routing policies to avoid that link. Fig. 8a shows the CDF of times for the destination to receive 1MB using each algorithm, averaged over the affected flows.

The second test used synthetic traffic matrices from a modulated gravity model [43] for the Forthnet topology. We set the total traffic volume to average $1.5\text{Gb}/\text{sec}$, spread (according to the gravity model) across TCP connections between all switch pairs. The capacity of each link was set to $800\text{MB}/\text{sec}$. We randomly broke one link and selected all affected flows to update their routing policies. Fig. 8b shows the CDF of times for the destination to receive 1MB using each algorithm, averaged over the affected flows.

Nimble outperformed SwingState and OpenNF in recovering from the link fault (Fig. 8). Though SwingState also uses tunnels to migrate NFs, it mirrors packets but still attempts to deliver packets through the old path (which contained a failed link in these tests) during NF migration. Moreover, for Forthnet topology, RWC outperformed SCC and CU when incorporated with Nimble, showing that Nimble can achieve even faster network recovery using RWC than when coupled with a traditional consistent update in some cases.

VIII. Conclusion

We presented an algorithm that accelerates NF migration and accompanying path changes in SDN networks over current solutions. Our design carefully interleaves NF migrations with path changes, and ensures correctness of traffic processing if the route-update protocol on which we build ensures a property that we call *relaxed waypoint correctness*. We provided a route-update protocol designed to achieve this property, without enforcing other properties typical of consistent-update protocols. We showed the sufficiency of this property through model checking and the improvements achieved by our algorithm in empirical comparisons to the state of the art.

References

[1] D. J. Chaboya, R. A. Raines, R. O. Baldwin, and B. E. Mullins, “Network intrusion detection: Automated and manual methods prone to attack and evasion,” *IEEE Security & Privacy*, vol. 4, no. 6, 2006.

[2] T.-H. Cheng, Y.-D. Lin, Y.-C. Lai, and P.-C. Lin, “Evasion techniques: Sneaking through your intrusion detection/prevention systems,” *IEEE Communications Surveys & Tutorials*, vol. 14, no. 4, 2012.

[3] I. Corona, G. Giacinto, and F. Roli, “Adversarial attacks against intrusion detection systems: Taxonomy, solutions, and open issues,” *Information Sciences*, vol. 239, Aug. 2013.

[4] T. Schneider, R. Birkner, and L. Vanbever, “Snowcap: Synthesizing network-wide configuration updates,” in *ACM SIGCOMM*, Aug. 2021, p. 33–49.

[5] S. Liu, T. A. Benson, and M. K. Reiter, “Efficient and safe network updates with suffix causal consistency,” in *14th ACM European Conference on Computer Systems*, Mar. 2019.

[6] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, “Transiently secure network updates,” in *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, 2016, pp. 273–284.

[7] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, “Abstractions for network update,” in *ACM SIGCOMM*, Aug. 2012.

[8] A. Gember-Jacobson and A. Akella, “Improving the safety, scalability, and efficiency of network function state transfers,” in *ACM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2015, pp. 43–48.

[9] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, “OpenNF: Enabling innovation in network function control,” in *ACM SIGCOMM*, 2014, pp. 163–174.

[10] S. Luo, H. Yu, and L. Vanbever, “Swing State: Consistent updates for stateful and programmable data planes,” in *3rd Symposium on SDN Research*, 2017, pp. 115–121.

[11] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, “Split/Merge: System support for elastic execution in virtual middleboxes,” in *10th USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2013.

[12] B. Kothandaraman, M. Du, and P. Sköldström, “Centrally controlled distributed vnf state management,” in *ACM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2015, p. 37–42.

[13] “Open vswitch,” http://openvswitch.org.

[14] “Ryu controller,” https://osrg.github.io/ryu.

[15] G. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. M. Jr., “Metron: NFV service chains at the true speed of the underlying hardware,” in *15th USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2018.

[16] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, “E2: A framework for nfv applications,” in *25th Symposium on Operating Systems Principles*, Oct. 2015.

[17] Y. Wang, G. Xie, Z. Li, P. He, and K. Salamation, “Transparent flow migration for NFV,” in *24th IEEE International Conference on Network Protocols*, 2016, pp. 1–10.

[18] K. Foerster, A. Ludwig, J. Marcinkowski, and S. Schmid, “Loop-free route updates for software-defined networks,” *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, pp. 328–341, 2018.

[19] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, “Dynamic scheduling of network updates,” in *ACM SIGCOMM*, Aug. 2014, pp. 539–550.

[20] F. Klaus-Tycho, M. Ratul, and W. Roger, “Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes,” in *IFIP Networking Conference and Workshops*, 2016.

[21] T. Mizrahi, E. Saat, and Y. Moses, “Timed consistent network updates,” in *1st ACM Symposium on Software Defined Networking Research*, 2015.

[22] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live migration of virtual machines,” in *2nd USENIX Symposium on Networked Systems Design and Implementation*, 2005, p. 273–286.

[23] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, “Remus: High availability via asynchronous virtual machine replication,” in *5th USENIX Symposium on Networked Systems Design and Implementation*, 2008, p. 161–174.

[24] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker, “Transparent, live migration of a software-defined network,” in *ACM Symposium on Cloud Computing*, 2014, pp. 1–14.

[25] D. M. F. Mattos and O. C. M. B. Duarte, “Xenflow: Seamless migration primitive and quality of service for virtual networks,” in *IEEE Global Communications Conference*, 2014, pp. 2326–2331.

[26] M. Kablan, A. Alsudais, E. Keller, and F. Le, “Stateless network functions: Breaking the tight coupling of state and processing,” in *14th USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2017.

[27] J. Khalid and A. Akella, “StreamNF: Performance and correctness for stateful chained NFs,” in *16th USENIX Symposium on Networked Systems Design and Implementation*, 2019.

[28] W. Shinae, S. Justine, H. Sangjin, M. Sue, R. Sylvia, and S. Scott, “Elastic scaling of stateful network functions,” in *15th USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2018.

[29] “Open Networking Foundation,” https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[30] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, Jul. 1990.

[31] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, “Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies,” in *13th ACM Workshop on Hot Topics in Networks*, 2014.

[32] “Z3py,” https://github.com/ericpony/z3py-tutorial.

[33] L. Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*, Mar. 2008, pp. 337–340.

[34] “Passive real-time asset detection system,” https://github.com/gamelinux/prads.

[35] “Gurobi solver,” http://www.gurobi.com.

[36] “Mininet,” http://mininet.org.

[37] “Topology zoo,” http://www.topology-zoo.org.

[38] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *ACM SIGCOMM*, Aug. 2008, pp. 63–74.

[39] D. Huang, K. Yocum, and A. Snoeren, “High-fidelity switch models for software-defined network emulation,” in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, Aug. 2013.

[40] H. Chen and T. Benson, “Hermes: Providing tight control over high-performance SDN switches,” in *13th International Conference on Emerging Networking Experiments and Technologies*, 2017, pp. 283–295.

[41] S. Liu, “Efficient and safe migration of network functions using software-defined networking,” Ph.D. dissertation, The University of North Carolina at Chapel Hill.

[42] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, “Homa: A receiver-driven low-latency transport protocol using network priorities,” in *ACM SIGCOMM*, Aug. 2018, p. 221–235.

[43] M. Roughan, “Simplifying the synthesis of internet traffic matrices,” in *SIGCOMM Computer Communication Review*, Oct. 2005, p. 93–96.