Software Transactional Networking: Concurrent and Consistent Policy Composition

Marco Canini[1] Petr Kuznetsov[1,2] Dan Levin[1] Stefan Schmid[1]
[1] TU Berlin / T-Labs [2] Télécom ParisTech
<first name>@net.t-labs.tu-berlin.de

ABSTRACT

It seems natural to imagine that SDN policy specification and control is distributed, and this paper focuses on the resulting concurrency issues. Indeed, conflicts among concurrent policy updates may result in serious inconsistencies on the data plane, even when each update is installed with per-packet consistent update semantics. This paper introduces the problem of *consistent* composition of concurrent policy updates. Intuitively, consistent concurrent policy composition must *appear* as though there is no concurrency neither between any policy updates, nor between a policy update and in-flight packets on the data plane.

We propose an elegant policy composition abstraction based on a *transactional interface* with *all-or-nothing* semantics: a policy update is either *committed*, in which case the policy is guaranteed to compose consistently over the entire network and the update is installed in its entirety, or *aborted*, in which case, no packet is affected by it. Consequently, the control application logic is relieved from the cumbersome and potentially error-prone synchronization and locking tasks, and control applications are kept lightweight. In this paper, we also sketch a simple implementation of the transactional synchronization: our approach is based on fine-grained locking on network components and avoids complex state machine replication.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network operating systems

Keywords

Software Defined Network; Control plane; Network policy

1. INTRODUCTION

The raison d'être of Software Defined Networking (SDN) is to enable the specification of the network-wide policy, that is, the desired high-level behavior of the network. While SDN makes this possible, there are several other requirements that are not immediately accommodated by this model. First, the network-wide policy unlikely comes from a single, monolithic specification, but rather, it is composed of *policy updates* submitted by different functional modules (*e.g.*, access control, routing, *etc.*). It may ultimately come from multiple human operators, each of which is only responsible for a part of it. Second, while control in SDN is logically centralized, important considerations such as high availability, responsiveness, and scalability dictate that a robust control platform be realized as a distributed system.

To support modular network control logic and multi-authorship of network policy, we face the issues of policy *composition* and *conflict resolution*. In previous work, Foster *et al.* [3] and Ferguson *et al.* [2] have addressed these two issues to a good extent in centralized (or *sequential*) settings—namely, in which there exists a central point for resolving policy conflicts and serializing policy composition. In this work, we bring existing approaches one step further, and present a solution for distributed composition of network policies that supports *concurrency* among the network control modules.

To satisfy commercial deployment requirements, Onix [7] realizes a control platform that relies on existing distributed systems techniques and builds upon the notion of a Network Information Base (NIB), *i.e.*, a data structure that maintains a copy of the network state. Different control modules can operate concurrently by reading from and writing to the NIB. While Onix handles the replication and distribution of the NIB, it does not provide policy synchronization semantics. Instead, Onix expects developers to provide the logic that is necessary to detect and resolve conflicts of policy specification. In this paper, we argue that *synchronized policy composition* must be an indispensable element of the distributed control platform.

We illustrate in Section 2 that a concurrent execution of *overlapping* policy updates (*i.e.*, affecting overlapping sets of traffic flows), may result in serious inconsistencies on the data plane (*e.g.*, in violation to the policy, traffic not being forwarded to an IDS middlebox). To fix this, we need to ensure that concurrent overlapping policy updates are *consistently* synchronized. Informally, we propose to define consistency by reducing it to the equivalence to a sequential policy composition, in the spirit of how correctness is defined for concurrent data structures [6]. We distinguish between strong and weak consistency levels for policy composition. Informally, we say that strongly consistent policy composition results in all data plane traffic experiencing the same globally ordered sequence of composed policy updates. Weakly consistent policy composition allows different traffic flows to experience differently ordered sequences of composed policy updates. In the absence of permanently pending policy updates, both strongly and weakly consistent concurrent policy compositions eventually lead to the same network-wide forwarding state.

Of course, we envision that some policy updates cannot be installed either because their composition is not defined [2,3] (*e.g.*, mutually exclusive forwarding actions) or due to unforeseen network conditions (*e.g.*, failures or scarce network resources). Therefore, we stipulate that the synchronization interface should be *transactional*. A policy update submitted by a control module either *commits*, *i.e.*, the update is successfully installed, or *aborts*, *i.e.*, it does not affect the data plane. We term our approach "Software Transactional Networking", inspired by the software transactional memory (STM) abstraction for optimistic concurrency control [12].

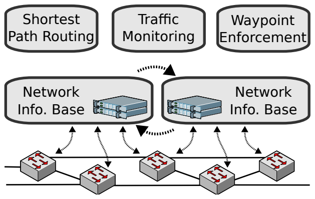
Implementing strongly consistent transactional synchronization can be achieved by centrally serializing the requests at a single master controller or using state-machine replication. However, we believe such heavy-weight solutions are not required, since weakly consistent policy composition is good enough for many cases. We describe a surprisingly simple light-weight implementation of it that only requires low-level, per-switch port conflict resolution, which can be implemented by, *e.g.*, maintaining one lock per hardware switch. Our implementation assumes that rules for composition be specified using the earlier work [2, 3], and ensures that the data plane is never affected by partially installed policies using a variation of the *two-phase update* protocol of Reitblatt *et al.* [11].

Overall, we believe this paper is the first to argue that policy synchronization protocols must be first-class citizens in software-defined networking by presenting counter examples and describing solutions. The rest of the paper is organized as follows. In Section 2, we demonstrate scenarios in which naïve NIB-based approaches to implementing concurrent policy updates result in inconsistencies in the data plane. In Section 3, we briefly describe the details of our model and state the problem of consistent policy composition. In Section 4, we sketch our implementation of weakly consistent policy composition. We overview the related work in Section 5. We conclude by discussing limitations of our results and speculating on future work in Section 6.

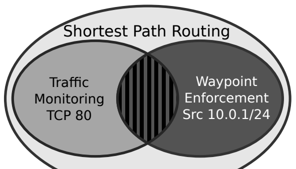
2. CONCURRENT COMPOSITION

We now discuss two simple examples that illustrate the need to extend previous work on SDN policy composition by addressing the concurrency issues in the distributed controller platform setting. Figure 1(a), logically illustrates our setting, an Onix-like SDN architecture in which two distributed controller instances run three functional modules—a packet repeater (hereafter called shortest path routing), a web traffic monitor, and a waypoint enforcement module coupled to an Intrusion Detection System (IDS).

In the first example, we begin with the policy composition scenario introduced in *Frenetic* [3], a network programming language and run-time system to automatically and correctly "compose" policies from separate control modules, and compile them to low-level forwarding rules. Assume the initial network policy comes from just the shortest path routing and web traffic monitor. Because these modules generate



(a) 3 control modules in an Onix-like architecture.



(b) Per-policy flow-space.

Figure 1: Three composed policies and their respective flow-space overlaps.

overlapping policy updates, they must compose their forwarding rules before any rule may be installed. The crux of the composition is that the correct forwarding rule priorities must be chosen such that the more specific web-monitoring rule matches packets with higher priority than the overlapping shortest path forwarding rule. As we illustrate in the flow-space diagram in Figure 1(b), the policy whose action applies to the striped region of flow-space, for example, must be defined consistently over the entire network.

Let us first assume that the monitoring and forwarding policies are generated sequentially in time. Even in the absence of concurrency, some synchronization step would be necessary for the latter controller instance to detect and resolve which policy should apply to any overlapping region of flow-space. For example, the composition of a monitor update request to count HTTP packets (`tcp_port=80` $\rightarrow$ `count`)[1] with a forwarding update request to forward packets arriving from address `10.0/16` to port 2 (`src=10.0.*` $\rightarrow$ `fwd`(2)) should result in the following forwarding rules regardless of the order in which the requests are processed:

Priority	Match	Actions
0	`tcp_port=80`	`count`
0	`src=10.0.*`	`fwd`(2)
1	`src=10.0.*` $\wedge$ `tcp_port=80`	`count`; `fwd`(2)

In the absence of such an agreement, conflicting rule priorities may be selected and different paths through the network would yield different policies, by virtue of the order in which the respective controller instance completed its update. For this specific case, the resulting data-plane inconsistency at the end of both updates would be that some paths through the network monitor web traffic volume while some do not. Note that even when both controllers utilize per-packet consistent network update semantics [11], the matching rule priority issue is not guaranteed to be resolved.

Next, we demonstrate how an inconsistent policy composition arises when multiple controller instances execute asynchronously and concurrently. Consider again two separate controller instances executing the three functional modules as depicted in Figure 1(a) over some time period based on their view of the network. Each controller instance executes with the goal of installing its resulting composed policy, in the form of new or modified flow-table entries into the underlying switches, using per-packet consistent updates.

[1]Although in OpenFlow each rule is implicitly associated with packet and byte counters, for presentation sake we represent "count" as an explicit action.

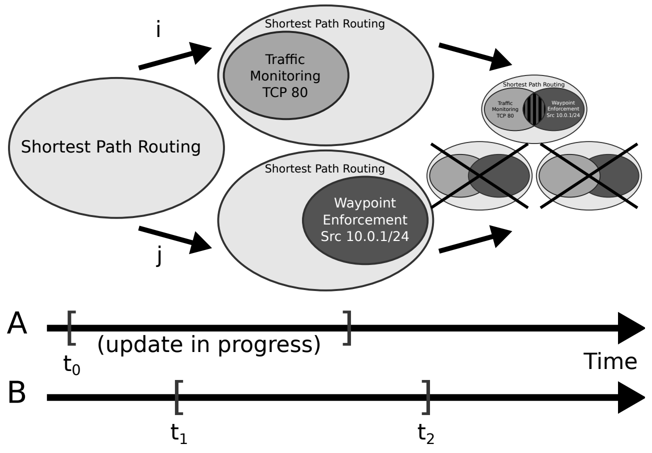


Figure 2: Possible policy composition outcomes from two concurrent policy updates. Conflicting compositions (crossed out) must be avoided.

Note that whenever policy updates compose trivially, *i.e.*, they affect disjoint network-wide slices of the flow-space, concurrency issues resolve automatically (as in the case of the FlowVisor [13]). In contrast, when composition is non-trivial, given concurrent execution, the previous policy composition problem becomes more difficult to detect and resolve. Without synchronization it is impossible at any single point in time for a particular controller instance to ensure that its policy specification does not conflict with that of any other's (see the companion technical report [1]).

We illustrate an asynchronous policy composition example using two controller instances A and B running concurrently in Figure 2. At time t_0, controller A begins a per-packet consistent network update to begin collecting web traffic statistics. From the perspective of controller A, it begins a transition along path i. Soon afterward, at time t_1, an influx of scanning traffic from source network 10.0.1/24 triggers the IDS module of controller instance B to execute a waypoint forwarding policy update to capture the scanning traffic for analysis. As the consistent update from controller A is still in progress, controller B computes an update as though it were transitioning along edge j. By time t_2, once the update execution of A and B have completed, at least three rules with different matching priorities will have been installed at each switch for the overlapping region of flow-space depicted in stripes in Figure 1(b). In the absence of a policy synchronization semantic, any of the three illustrated end-states may be reachable in the network. Each crossed-out flow-space diagram violates either the monitoring or waypoint enforcement policy, respectively. The goal of concurrent consistent policy composition is to ensure that only one such final state is reached eventually, everywhere in the network, and that the final state is not in violation of the policies which lead to its installation.

3. THE STN ABSTRACTION

This section gives an overview of the *Software Transactional Networking (STN)* architecture we propose. This architecture relies on a middleware offering a simple transactional interface where policy updates can be applied; the middleware implements the updates correctly and efficiently. We define what *correctly* means below. The discussion of efficient implementations is postponed to the subsequent section.

3.1 Interface

Concurrent policy updates may *conflict* in the sense that, intuitively, they may apply mutually exclusive actions to overlapping sets of packets. The goal of the synchronization layer of the network control platform is to make sure that all requests are composed in a "meaningful" manner, so that every packet in the network does not experience an inconsistent policy composition. On the other hand, if some conflicting policies cannot be automatically composed,[2] the STN layer may reject some requests. In that case, the corresponding control modules receive a *nack* response equipped with some meta-information explaining what kind of conflict was witnessed.

Formally, a controller module tries to install a new policy p (a collection of rules to be installed at different switches) by invoking *apply*(p). The invocation may return *ack* meaning that the policy has been installed or *nack*(e) meaning that the policy has been rejected, where e is the explanation why.

3.2 Preliminaries

Before we can introduce our notions of update consistency, some formalism is needed. We call the sequence of invocations and responses of policy-update requests plus all data-plane events (such as data packet arrivals and departures at the ports of the switches), a *history* H. We say that request r_1 *precedes* request r_2 in H, and we write $r_1 \prec_H r_2$, if the response of r_1 precedes the invocation of r_2. Note that, in a concurrent execution, $\prec_H$ is a partial order on the set of requests in H. A request r is *complete* in H if H contains a response of r. A complete request is said to be *committed* in H if its response is *ack*, or *aborted* otherwise.

For a history H, let $P_c(H)$ denote the set of policies resulting from sequential composition of all committed policies in H in the order respecting $\prec_H$, *i.e.*, no request r is (sequentially) processed before any request r' such that $r' \prec_H r$. Let $P_i(H)$ be the set of policies that result from a sequential composition of a *subset of* policies submitted by incomplete requests in H. Thus, a policy $p \circ p'$ such that $p \in P_c(H)$ and $p' \in P_i(H)$, is a composition of all already installed policies and a subset of policies that are being installed in H.

Following [11], we stipulate that every packet joining the network incurs a set of *traces*: sequences of causally related *located packets*, *i.e.*, (*packet*, *port*) pairs, indicating the order in which the packets traverse the network. Informally, a trace is *consistent with a policy* p [11] if all the events of the trace are triggered respecting p. For a trace π in a history H, let H_π denote the prefix of H up to the first event of π (*i.e.*, up to the moment when the corresponding packet enters the network).

3.3 Consistency

Intuitively, we want to ensure that every trace in H respects a policy resulting from a *consistent* composition of previously and concurrently installed policies. We distinguish between *strong* and *weak* consistent composition. In the strong version, policy-update requests are composed in

[2]Note that certain conflicts can be arbitrated, *e.g.*, by requiring a priori knowledge of a strict priority order across modules as in [2].

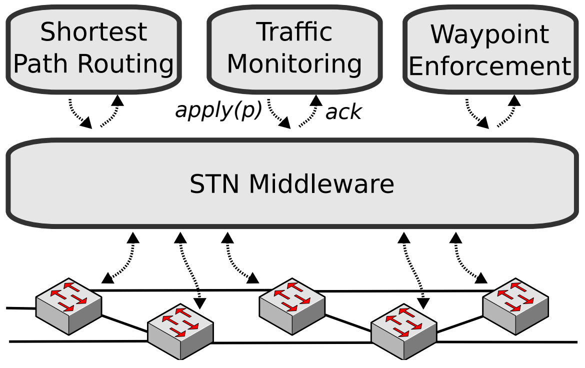


Figure 3: A logical representation of the STN middleware.

the same linear order, and in the weak one incomplete policies can be composed arbitrarily.

For example, weakly consistent composition allows the scenario depicted in Figure 2, where packets can be affected by any policy update that is not yet completed (corresponding to the policies along paths i and j in the figure). In contrast, strongly consistent policy may only take one of the paths, *i.e.*, before both updates complete, all the traffic is processed either according to the one of the concurrent policy updates or the other one, and then it is processed by the composition of the two.

Formally, we say that a history H provides *weakly consistent composition* if for every trace π in H, there exist policies $p \in P_c(H_\pi)$ and $p' \in P_i(H_\pi)$, such that π respects $p \circ p'$ (the composition of p and p').

Respectively, a history H provides *strongly consistent composition* if there exist a total order $S = (p_1, p_2, \ldots)$ of policies installed by requests in H such that (1) S respects $\prec_H$, and (2) every trace π in H respects a policy $p_1 \circ p_2 \cdots p_{i-1} \circ p_i$ that consists of all policies submitted by committed requests in H_π and a subset of policies submitted by a subset of incomplete requests in H_π. In other words, we require that there exists some global order in which policies evolve.

Note that we deliberately leave specifying the details of how exactly the *sequential* composition is done to the control logic since this is in general application-specific (and realizable, *e.g.*, through previous work in [3]). In particular, for our weaker notion of consistency, concurrently committed requests may be required to *commute*, *i.e.*, their sequential composition results in the same policy regardless of the order in which they are applied. Indeed, since we allow concurrently installed policies to be witnessed by different traces to be composed in arbitrary order, we may want this order to be effectively the same (*e.g.*, recall the composition of the monitor update and the shortest path routing update in Section 2).

4. THE STN MIDDLEWARE

In this section, we sketch a simple implementation of weakly consistent policy composition. We investigate a proposal for strongly consistent policy composition in a separate technical report [1]. On the progress side, we guarantee that every request eventually completes by returning *ack* or *nack*. To filter out a trivial solution that always returns *nack*, we require that no policy-update request aborts, unless it faces a *conflicting* policy with which it cannot be composed. The exact definition of a conflict is left to the control application; a practical definition comprises regarding mutually exclusive actions to overlapping sets of packets as conflicts.

The basic building block of our solution is the *two-phase update* algorithm [11]. In the first phase, the algorithm first installs the new policy on the internal ports[3] of the network (in arbitrary order) for packets with a unique tag number. In the second phase, the algorithm updates the ingress ports of the network (in arbitrary order) to equip all incoming packets with the new tag number. This way, while the policy update is in progress, every packet is processed either only by the old policy or only by the new one, but never a mixture of both.

Several implementations are possible based on different granularity of *locking*. A trivial solution that provides consistency could maintain a global lock on the entire network: To install a new policy, a controller grabs the global lock and then runs the two-phase update algorithm in the absence of contention. Assuming that the global lock is implemented in the starvation-free manner (*e.g.*, by a fair central server), we obtain a solution providing strongly consistent composition.

Below we describe a more intelligent algorithm that only requires a weak form of local (*per-switch*) synchronization. Our algorithm assumes that each switch exports an atomic *read-modify-write* operation $rmw(x, m, f)$, where x is a switch, $m \subseteq States$ is a "mask" matching a subset of the forwarding rules in x's state, and $f : 2^{States} \rightarrow 2^{States}$ is a function that updates the part of the state of x matching m based on the read value. However, note that we do not necessarily require the switches to implement this atomic read-modify-write operation; in a NIB-based implementation of STN, this operation may be incorporated into the NIB itself.

To illustrate by example, when a control module wants to collect statistics on all packets with `tcp_port=80` arriving to switch x, it should check if x currently maintains rules that affect packets to TCP port 80. If this is the case, *e.g.*, x forwards all packets with `src=10.0.*` to port 2 (see the example in Section 2), then function f stipulates that the new rules to be added to the configuration of x are:

Priority	Match	Actions
0	`tcp_port=80`	`count`
1	`src=10.0.*` $\wedge$ `tcp_port=80`	`count`; `fwd`(2)

The pseudocode of our algorithm is sketched in Algorithm 1. A controller first chooses a unique tag number, and then goes through every switch in the network. In the first phase of the algorithm, the controller tries to atomically install the new policy using the read-modify-write (confined in the *atomic* block) at every switch. If, at a given switch, the currently installed policy p cannot be composed with existing policies, all previous updates are rolled back and *nack* is returned, together with the information about conflicting policies.

In the second phase, the controller updates the ingress ports of every switch to tag the incoming traffic with the new number, computed based on the tags of previously installed policies. Note that we hide some extra complexity

[3]Internal and ingress ports are defined as per [11].

```
procedure apply(p)
    tag := choose a unique id for the new policy;
    foreach switch x do
        atomic{
            if x can be composed with p (wrt f) then
                foreach subset s of tagged policies at x do
                    tag′ := the composition of tags of s and tag;
                    add the composed set of rules marked with
                    tag′ to x;
                end
            else
                remove all previously installed rules;
                return nack(e), where e is the reason why;
            end
        }
    end
    foreach switch x do
        foreach ingress port i of x do
            atomic{
                tag′ := the composition of tags at i and tag;
                add the rule to tag all traffic to i with tag′;
            }
        end
    end
    return ack;
```

Algorithm 1: Concurrent policy update algorithm: code for *apply*(p).

here: we need to make sure that all policies that may coexist in the network are represented at every affected switch. Therefore, we may need to concurrently maintain a separate tag number for each subset of previously and concurrently installed policies. This brings new interesting challenges related to consistent tag generation and garbage collection of obsolete policies (see [11] and the discussion in Section 6).

The correctness of the algorithm stems from the fact that when a packet that arrives at an ingress port is processed, it obtains a tag number of a policy that has been previously installed at all internal ports of the network. Moreover, from the moment when an update request for policy p is complete, every packet to arrive is processed according to a composed policy that contains p. Also, the composed policy trivially preserves the real-time order of non-overlapping requests. Thus, we implement weakly consistent policy composition. Finally, assuming atomic read-modify-write primitives, we observe every policy-update request commits, unless it witnesses a policy it cannot be sequentially composed with.

5. RELATED WORK

The need for a distributed control plane has already been motivated in different contexts, *e.g.*, to reduce the latency of reactive control [5], to place local functionality closer to the data plane [4], to improve Internet routing [8] by outsourcing the route selection, and so on. The focus of this paper is on how to design a distributed control plane which supports concurrency but yet consistency in policy composition.

Our work builds upon two recent threads of research: (1) *consistent network updates* [11] and (2) abstractions for horizontal and vertical *composition* [3, 10].

In [11], Reitblatt *et al.* introduce the notion of consistent network updates that guarantee that during transition from an initial configuration to a new one every packet (*per-packet* consistency) or every flow (*per-flow* consistency) is processed either by the initial configuration or by the new one, but never by a mixture of the two. In a centralized setting, they proposed a two-phase per-packet consistent update algorithm that first installs the new configuration on the internal ports and then updates the ingress ones.

Foster *et al.* [3] present the *Frenetic* language, which includes a run-time system to correctly compose policy modules to low-level rules on switches. Compared to the alternative OpenFlow interface, the language simplifies network programming as it circumvents the coupling of different tasks (like routing, access control, traffic monitoring, *etc.*). As we have shown, the modular composition concepts in [3] are a very useful abstraction also for the design of a distributed control plane.

We are not the first to explore SDN architecture as a distributed system. A study of SDN from a distributed systems perspective is given in Onix [7], a control plane platform designed to enable scalable control applications. Its contribution is to abstract away the task of network state distribution from control logic, allowing application developers to make their own trade-offs among consistency, durability, and scalability. However, Onix expects developers to provide the logic that is necessary to detect and resolve conflicts of network state due to concurrent control. In contrast, we study concurrent policy composition mechanisms that can be leveraged by any application in a general fashion. In [9], Levin *et al.* studied the implications of the logically centralized abstraction provided by SDN and the resulting design choices inherent to the strong or eventual network view consistency. They demonstrate certain consequences of this design choice on the performance of a distributed load-balancing control application.

6. DISCUSSION AND FUTURE WORK

Our sketch of an STN implementation hid many technical implementation details under the rug. Of course, we plan to address those in future work and below we speculate how.

The support of concurrent and consistent network updates features several additional challenges which have not been addressed so far. For example, while tags in the packet headers can be used to isolate different flows from each other and ensure that only one policy is applied throughout the packet paths, the number of required tags grows exponentially in the number of policies: given k policies, in the worst case, we may have to maintain the order of 2^k policy compositions. Adding these tags ensures consistent per-packet forwarding, but the solution is not scalable. Automatically reducing the number of used tags within the STN is however complicated. We envision some new and interesting trade-offs between the complexity of the state maintained at network components and the amount of synchronization required to ensure consistency across them.

Also the size of the Forwarding Information Base (FIB), *i.e.*, number of flow table entries that can be stored, may be a limiting factor for policy composition. While this problem is not specific to the distributed controller but general, we envision that an efficient STN middleware may be able to optimize the policy implementation by automatically aggregating multiple forwarding rules into a single one, and/or by distributing ("load-balancing") rules across multiple switches where possible.

The current definition of consistency of composition works on *per-packet* basis, but seems to be straightforward to extend it to the *per-flow* level (along the lines of [11]), as long as we know how to define flow delimiters.

There are many ways to implement the *read-modify-write* primitive giving access to the network switches. At a high level, it depends on how the rights of accessing a switch are distributed across the controllers. If there is a single controller that is allowed to modify the configuration of the switch, then we can implement the *rmw* primitive at the controller, so that it is responsible for the physical installation of the composed policy. An alternative solution is to provide a lock abstraction by the switch itself. For example, a weak form of locking can be achieved by maintaining a single *test-and-set* bit at a switch. Also, we believe that even more fine-grained locking mechanisms are possible, where, *e.g.*, instead of acquiring locks on entire switches, transactions synchronize on *ports* only. We leave the discussion of implementation details and further optimizations for future work.

Acknowledgements

This work was supported in part by the EU FP7 project OFELIA.

7. REFERENCES

[1] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. The Case for Reliable Software Transactional Networking. *CoRR*, abs/1305.7429, 2013. http://arxiv.org/abs/1305.7429.

[2] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Hierarchical Policies for Software Defined Networks. In *HotSDN*, 2012.

[3] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ACM ICFP*, 2011.

[4] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *HotSDN*, 2012.

[5] B. Heller, R. Sherwood, and N. McKeown. The Controller Placement Problem. In *HotSDN*, 2012.

[6] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[7] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[8] V. Kotronis, X. Dimitropoulos, and B. Ager. Outsourcing the Routing Control Logic: Better Internet Routing Based on SDN Principles. In *HotNets*, 2012.

[9] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically Centralized? State Distribution Trade-offs in Software Defined Networks. In *HotSDN*, 2012.

[10] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *NSDI*, 2013.

[11] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[12] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 1997.

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