STN: A Robust and Distributed SDN Control Plane

Marco Canini[†], Daniele De Cicco[†], Petr Kuznetsov[‡], Dan Levin[•], Stefan Schmid[⋆] and Stefano Vissicchio[†*]
[†] *Université catholique de Louvain* [‡] *Télécom ParisTech* [•] *TU Berlin* [⋆] *TU Berlin / T-Labs*

In Software Defined Networking (SDN), control applications operate on a global network view. This view enables simplified programming models to define at a high-level the intended operational behavior of the network. While a fully centralized system is a natural solution to create such a global view, it may not adequately or cost-effectively provide the required levels of availability, responsiveness and scalability. Instead, commercial SDN deployments [9] resort to redundant and distributed systems running on multiple physical nodes. However physical distribution comes with hard challenges, *i.e.*, "the designers of such systems have to face the fundamental trade-offs between the different consistency models, the need to guarantee acceptable application performance, and the necessity to have a highly available system" [1].

An appealing approach to deal with these challenges consists of creating sophisticated applications by composing multiple, simpler modules, in the same spirit as Pyretic [6], an SDN programming language that enables policy composition. Beyond enabling a divide-and-conquer approach to networking problems, modularity also enables multi-authored policies that may come from different administrators or even end-host applications [4]. In the following, we describe our progresses in developing a distributed SDN control plane, called **Software Transactional Networking (STN)**, which is inspired by Software Transactional Memory principles and is provably robust to a fixed number f of controller node failures. In particular, this paper focuses on how to solve concurrency issues that arise from the concurrent execution of control modules on different physical nodes. Indeed, conflicts among concurrent policy updates to network state may result in serious inconsistencies on the data plane, even when each update is installed with per-packet consistent update semantics (see [3] for an example). Generally, a policy update herein refers to a collection of state modifications spanning one or more switches.

STN Design

An overview of the STN architecture is conveyed in Figure 1. We designed STN to ensure: (1) *all-or-nothing semantics*, *i.e.*, every policy update is either fully installed, or it will never affect any single packet, (2) updates which generate no conflicts (*e.g.*, defined over independent flow spaces), are *eventually* installed, (3) a packet is *never delayed* due to an ongoing policy update, and (4) *per-packet consistent forwarding* [7]. To this end, the STN design is based on the following concepts.

The Transactional Interface. A possible approach to deal with update conflicts is to expose conflicts to the application developer and let her handle their resolution. To the best of our knowledge, the state-of-the art distributed control planes Onix and ON.Lab's ONOS, adopt this approach. As both use a replicated data store to maintain an up-to-date view of network state, they can detect conflicts at that level. In contrast with existing solutions, we propose an elegant policy composition abstraction based on a *transactional interface* with *all-or-nothing* semantics, inspired by the concept of Software Transactional Memory (STM) [8]. This interface ensures that a policy update required by one control module is either *committed*, *i.e.*, applied and consistently composed over the entire network, or *aborted*, in which case no packet is affected by it. The transactional interface relieves control modules from cumbersome and error-prone synchronization and locking tasks, and enables easier and more lightweight control applications. We envision scenarios in which multiple concurrent updates modify the policy by introducing possibly conflicting rules, and we expect that only policy updates that *compose* (induce a *correct* network behavior, which we next define) are allowed to affect the traffic. To define the correct network behavior, we adopt a solution similar to PANE's global share tree [4], which we refer to as the *global network policy*.

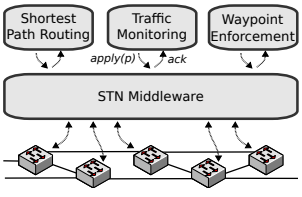

Figure 1: STN architecture.

Our correctness property informally requires that the STN abstraction, regardless of the actual interleaving of policy updates and packet arrivals, *appears* sequential to every packet, as though all the committed policy updates (and possibly a subset of incomplete ones) are applied atomically and no packet is in flight while an update is being installed. Note that our approach is not concerned with how an update is computed, and we can utilize a data store to obtain a view of network state. Such a data store must be accessed by modules in a read-only fashion, however, as all updates (i.e., writes) must execute via our abstraction to guarantee that they are composed and installed

*Stefano Vissicchio is a postdoctoral researcher of the Belgian fund for scientific research (F.R.S.-FNRS).

correctly. Also, note that our notion of consistency applies to the data plane, similarly to consistent network updates [5, 7]. In this paper, we are not concerned with the consistency model of the network state data store, if any is used.

The Shared Memory Analogy. To reason about correct and concurrent composition of network policy, we introduce a formal model describing the interaction between the network data plane and a distributed control plane, in which we consider failures under the fail-stop model. Our model views the network and its state as a distributed "shared memory" that is accessed by two kinds of processes: (i) the control modules, which concurrently read and update the network state in order to install different policies, and (ii) the data packet, whose paths across the network are determined by the forwarding rules installed by the controllers. The STN system is realized as a Distributed State Machine and can emulate both the 2-phase [7] and the dependency-based policy installation [5]. The control plane uses a tagging mechanism to (1) impose a global order on the modules and (b) ensure a coordinated use and reuse of the protocol tags. Concretely, the state machine exports, to each process, operations *push* and *pull* to obtain and release tags. Intuitively, the system invokes *pull* to fetch the next policy to be installed, based on policy update requests coming from different control modules. The invocation returns $\perp$ if all policies pushed so far are already installed, otherwise it returns a policy and a tag with which the controller should install that policy. The STN system minimizes the number of used tags. Indeed, we prove [2] that in our design, a controller's failure can block at most one tag, and that the STN system requires $\Theta(f)$ tags, where f is the number of fail-stop failures that the protocol can tolerate.

STN Prototype

We are building a prototype STN system upon Pyretic [6]. Our prototype leverages user-defined rules to compose some policies proposed by SDN control modules, and resorts to the STN interface for the other cases. Those composition rules use the sequential $>>$ and parallel $+$ composition operators already supported by Pyretic. Let p_1 and p_2 be two policies. The expression $p_1 >> p_2$ translates to first applying policy p_1 on each incoming packet and then policy p_2 on the output of p_1. In contrast, $p_1 + p_2$ applies the two policy functions on the same incoming packet and combines the results.

To enable the specification of realistic policy composition rules, we found the need to extend Pyretic. To describe our extensions, we use a running example of an STN system that coordinates actions between three modules respectively implementing (1) a routing policy R, installing shortest paths between sources and destinations for every traffic flow, (2) a waypointing policy W, forcing traffic from an IP prefix P to traverse a given middlebox, and (3) a monitoring policy M to be applied to all Web traffic. Additional modules are present in the system. We define the set of (located) packets to which a policy p applies as *domain* of the policy $dom(p)$.

Language Extension. As a first extension to Pyretic, we introduce a *precedence operator* $\succ$ to resolve a-priori some conflicts between policies. Concretely, the precedence operator enables STN system users (*e.g.*, network administrators) to indicate which policy has to be applied if multiple modules want to install rules for the same packets. Consider, e.g., the routing and waypointing policies with partially overlapping domains. If the latter policy induces an action different than shortest path routing, those policies generate conflicts, *i.e.*, mutually exclusive actions would be applied for packets originated from P. The $\succ$ operator provides a deterministic way to resolve such conflicts by setting the relative priority between policies. Given two policies p_1 and p_2, $p_1 \succ p_2$ translates to applying p_2 if and only if p_1 cannot be applied. Formally, for each incoming packet pkt, $p_1 \succ p_2$ is equivalent to **IF** $pkt \in dom(p_1)$ **THEN** p_1 **ELSE** p_2. Note that $pkt \in dom(p_2)$ is checked in the ELSE branch, by definition of policy.

Dynamic Module (De)Activation. Pyretic enables static definition of policies. However, STN needs to support dynamic activation and deactivation of modules, and to update accordingly the policy effectively applied by the switches. To this end, we define the *global policy* as the policy composition rule provided as input to the STN system. Moreover, we refer to the *effective policy* as the composition of policies proposed by currently active modules. Each time the set of active modules changes, the effective policy is updated according to the subexpression of the global policy only including the currently active modules. In addition, the effective policy can contain further policies not included in the global policy (if does not include all the policies implemented by modules in the system). Consider again our example with the R, M, and W policies, and let the global policy defined by the administrator be $(W \succ R) + M$. This means that monitoring is always applied in parallel either to the waypointing module or to the routing one (for traffic that has not to be waypointed). If only the routing module is initially active, then the effective global policy will be R. Then, if waypointing is activated, the effective policy is updated to $W \succ R$, and R is applied only for packets outside the domain of W. Otherwise, if the monitoring module is the second one to be activated, then the effective policy is updated to $R + M$. The global policy is eventually enforced when all the three modules are activated. Policies not appearing in the global policy are either accepted or rejected (if conflicting) according to the STN interface.

Acknowledgments. This work was (partially) supported by the ARC grant 13/18-054 from Communauté française de Belgique.

References

[1] F. Botelho, F. M. V. Ramos, D. Kreutz, and A. Bessani. On the feasibility of a consistent and fault-tolerant data store for SDNs. In *EWSDN*, 2013.

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

[3] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. Software Transactional Networking: Concurrent and Consistent Policy Composition. In *HotSDN*, 2013.

[4] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthy. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*, 2013.

[5] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.

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

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

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

[9] T. Koponen *et al.*. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.