In-Band Synchronization for Distributed SDN Control Planes*

Liron Schiff[1] Stefan Schmid[2] Petr Kuznetsov[3]
[1] Tel Aviv University, Israel [2] Aalborg University, Denmark [3] Télécom ParisTech, France

ABSTRACT

Control planes of forthcoming Software-Defined Networks (SDNs) will be *distributed*: to ensure availability and fault-tolerance, to improve load-balancing, and to reduce overheads, modules of the control plane should be physically distributed. However, in order to guarantee *consistency* of network operation, actions performed on the data plane by different controllers may need to be *synchronized*, which is a nontrivial task. In this paper, we propose a synchronization framework for control planes based on atomic transactions, implemented *in-band*, on the data-plane switches. We argue that this in-band approach is attractive as it keeps the failure scope local and does not require additional out-of-band coordination mechanisms. It allows us to realize fundamental consensus primitives in the presence of controller failures, and we discuss their applications for consistent policy composition and fault-tolerant control-planes. Interestingly, by using part of the data plane configuration space as a shared memory and leveraging the match-action paradigm, we can implement our synchronization framework in today's standard OpenFlow protocol, and we report on our proof-of-concept implementation.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network Management

1. INTRODUCTION

By consolidating and outsourcing the control over the data-plane switches to a logically centralized controller, Software-Defined Networks (SDNs) simplify network management and facilitate faster innovations: the (software) control plane can evolve independently from the (hardware) data plane. While the perspective of a *logically centralized* control plane offered by SDN is intuitive and attractive, there is a wide consensus that the control plane should be *physically distributed*. First, in order to provide high availability, controllers should be redundant [1,5,11]: a failure of one controller can be masked by other controllers. Second, it has also been proposed to distribute controllers *spatially*, in order to handle latency-sensitive and communication-intensive data plane events close to their origin [3,9]. Third, larger SDNs are likely to be operated by multiple administrators [2] or may even offer participatory interfaces where different users can install and trigger policy changes concurrently [6].

Today, we do not have a good understanding yet of how to realize such distributed control planes. The problem is essentially a distributed-systems one: multiple controllers may simultaneously try to install conflicting updates and we want to resolve these conflicts *consistently* (no undesired behavior is observed on the data plane) and *efficiently* (no undesired delays are imposed on the control application). *Synchronizing* the distributed controllers and manipulating the network state *consistently* are non-trivial tasks [19].

Consider, for example, the problem of consistent installation of new forwarding policies, stipulating routes that packets of different header spaces should follow across the network [14, 16, 20]. Installing *conflicting* forwarding rules, e.g., rules of the same priority defined over non-disjoint flow spaces may lead to pathological network behavior (loops, blackholes, routes bypassing a firewall, etc.) [15, 16]. Similarly, installing diverging load-balancing policies may, when combined, *increase* the load. To render things more difficult, controllers may also fail, even before their updates have been completed.

The concurrent computing literature offers a wide range of synchronization abstractions to realize consistent distributed systems. In particular, the popular *transactional memory* abstraction (cf. the survey in [8]) provides a collection of concurrent processes with the ability of aggregating sequences of shared-memory operations in *atomic transactions* with all-or-nothing semantics.

Contributions.

This paper applies the principle of atomicity in concurrent computing to distributed SDN control planes. In particular, we propose synchronization constructs which allow a controller to represent multiple configuration commands on the data plane as an atomic transaction. If none of the transaction's commands *conflicts* with the current configuration, where a conflict can be defined in a general, application-specific manner, the transaction appears to be executed atomically. Otherwise, the transaction is *aborted* in its entirety and affects neither traffic nor other controllers.

We propose to implement our synchronization constructs *in-band*, on the data-plane switch. An in-band implementation allows us to efficiently alleviate the problems related to coordinating controllers via a control-plane out-of-band network, in the presence of asynchrony and failures. Indeed, the inherent costs of such conventional *out-of-band* protocols are often considered too high, both in terms of the necessary

*Supported by the European Research Council (ERC) Starting Grant no. 259085, the European Institute of Innovation & Technology (EIT) project 13153 (Software Defined Networking) and the FP7 EU project UNIFY.

computability assumptions about the underlying system [7], and the high communication complexity [13]. In contrast, our *in-band* solution allows the controllers to solve fundamental agreement tasks in just one message exchange with the data plane, tolerating asynchrony and failures of any number of controllers.

The key idea of our approach is to use the data-plane switch configuration space as a *transactional shared memory*. In addition to the actual data-plane configuration, the memory can store information about contention and conflicts between the controllers. A transaction can then contain standard control and update operations as well as *synchronization primitives* operating on this shared memory. The synchronization primitives allow the controllers to define general notions of conflicts between configuration updates, covering simple routing conflicts as well as more sophisticated multi-flow dependencies appearing in load-balancing applications.

As a case study, and to demostrate the feasibility of our conceptual approach, we consider the standard *OpenFlow* protocol (version 1.4 [17]). We show that OpenFlow's match-action paradigm can be instrumented to realize a simple transactional abstraction that allows the controllers to solve several important synchronization problems. For example, we describe an OpenFlow implementation of the fundamental *compare-and-set* (CAS) primitive. The atomic CAS operation is known to provide an infinite consensus number, and can, for instance, be used to implement a generic consistent and fault-tolerant replicated service [10], a mainstream building block in modern distributed systems (Chubby, Google Spanner, Amazon Webservices, etc.). We discuss a simple CAS-based concurrent policy-update mechanism, and present a generic template to implement concurrent policy compositions of a user-specific update functions. Our synchronization abstraction also provides the missing link for the read-modify-write object postulated in STN [2].

Motivation and Example.

A transactional distributed system offers all-or-nothing semantics, and aborts transactions in case of conflicts. For instance, in the context of SDNs, a transaction trying to add a forwarding rule whose domain overlaps with an existing rule of the same priority can constitute a conflict. However, as we will argue, there exist many applications with less obvious conflicts, which go beyond simple range overlaps. Moreover, sometimes it is desirable to react to conflicts in smarter ways than by simply aborting an operation, e.g., by supporting *conditional modifications*.

Let us consider an example. Imagine two controllers in charge of *load-balancing*, see Figure 1. In such a scenario, seemingly independent actions, defined over completely independent logical flow spaces (say, two different TCP microflows), may actually be dependent: the flows share the underlying physical network. Accordingly, if multiple controllers concurrently and independently update forwarding rules according to a naïve load-balancing algorithm, they may involuntarily unbalance the flow allocation. Simple flow space overlap checks cannot be used to detect such conflicts.

This paper proposes mechanisms to solve these and more general synchronization problems.

2. IN-BAND SYNCHRONIZATION

In order to synchronize controllers, we propose to use a

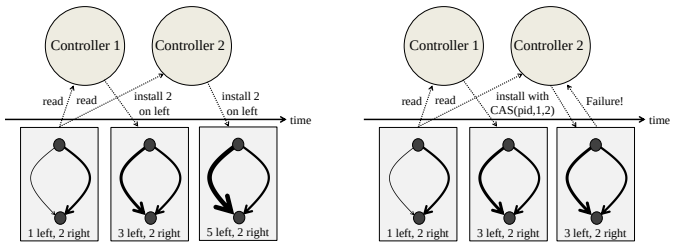

Figure 1: ***Left:*** **Without synchronization, the two controllers naturally choose to install their flows on the left link, which results in an undesirable unbalanced state.** ***Right:*** **With synchronization, i.e., by bundling the flow installation with a Compare-and-Set (CAS) primitive depending on the policy id (*pol-id*), this problem is avoided.**

part of the configuration space of the data-plane switch for storing a "meta-data", containing information on contention and conflicts between concurrently installed policy updates. Controllers can read and modify this meta-data by applying synchronization primitives. In the following section, we describe how this space can be organized and consistently maintained by multiple controllers despite of concurrency and conflicts.

Configuration space. We distinguish between two parts of the configuration space of a switch: the "normal" configuration which determines the network policy and is used to process the data-plane traffic (e.g., forwarding rules), and the *meta-configuration* that contains information used by the controllers to synchronize their actions. Flow entries in the meta-configuration can be referred to as shared memory locations, each provided with a distinct *address*.

The meta-configuration is implemented as a set of flow entries designed in a way that it does not affect the processing of data plane traffic. We will discuss later how this can be achieved, for example, in today's OpenFlow.

Synchronization. We provide the controllers with the ability to execute sequences of *operations* on the switch configuration in an *atomic* or *non-atomic* mode. In the atomic mode, either *all* operations in the sequence, henceforth simply called *transaction*, should be executed, or none (the transaction is rejected). In the non-atomic mode, no atomicity guarantees are provided.

In particular, construct **execute**$\{op_1, \ \ldots \ op_k\}$ is used to perform a sequence of commands $op_1, \ldots, op_k$ in a conventional way. Construct **execute-transaction**$\{op_1, \ \ldots \ op_k\}$ ensures that either all commands are successfully performed, in which case it returns *ack*, or, if some of the commands cannot be performed, aborts the whole sequence and returns `error(xid, error-code)`, providing the identifier of the first command in the sequence that failed together with the corresponding error code.

Synchronization primitives. We aim to enforce a consistent use of *policy identifiers*, an important notion in consistent policy updates [2,20]. Intuitively, a controller should equip each to-be-installed policy with a distinct identifier, and the installation should succeed only if the identifier of the currently installed policy has not changed since the last time the controller read it.

Therefore, we provide the following transactional opera-

tions. A *write*($addr, k$) operation sets the content at memory location *addr* to k. A *compare*($addr, k$) operation *aborts* the transaction if and only if the content at *addr* is *not* k. Intuitively, by combining compare and write with a set of regular switch update commands (e.g., FlowMod in OpenFlow) in a transaction we can make sure that the configuration is changed and the current policy identifier is updated only if the check operation *succeeds* (i.e., does not abort).

To make sure that policy identifiers do not grow without bound [2], we also provide an *id-claimer* object. The object implements a weak type of locking by allowing a controller to claim a *resource identifier* in a given set, release an identifier, and check if a given identifier is currently claimed by any controller. More precisely, the id-claimer object exports three operations *claim*, *unclaim* and *check* with the following sequential semantics:

- With *claim*(k), a controller *claims* identifier k.
- With *unclaim*(k), the controller *unclaims* identifier k. An identifier k is called *claimed* at a given point of an execution if there is a controller that has performed *claim*(k) but has not yet performed *unclaim*(k).
- With *check*(k), the controller checks if k is currently claimed by any controller, and if so, the current transaction is aborted.

3. CASE STUDY: OPENFLOW

Before discussing applications in more detail, as a case study and in order to introduce some notation, we in this section show how to implement our transactional approach using today's standard SDN protocol, OpenFlow (version 1.4 [17]).[1]

Background.

The configuration of an OpenFlow switch is a set of *rules*. A rule is essentially a *match-action* pair: the match part of a rule identifies the space of packet headers that are subject to the rule (e.g., all packets to a specific destination) and the action part identifies how the switch should process the matched packets (e.g., forward them to a specific port). More precisely, the match part of a rule is a ternary pattern over packet *header* fields. The pattern is represented as a *value* and a *mask*. In the mask, certain header fields, e.g., TCP or UDP port numbers or destination and source addresses, can be *wildcarded*, stipulating that their content does not affect the match. Certain fields, such as *ipv4_src*, *ipv4_dst*, *metadata*, can be arbitrarily bitmasked. In our implementations, we are going to use bitmasking of the *metadata* field for performing conditional operations based on its content. For example, `ipv4_src=10.1.14.0/ff.ff.ff.f0` matches all packets with source IP addresses in the range `10.1.14.0`–`10.1.14.15`.

A *configuration* of an OpenFlow switch is represented as a collection of *flow tables*. A flow table is a set of *flow entries*, each containing a rule, with a match, an action, and a *priority* level, and flow entries are ordered according to their priorities. In the simplest setting, a switch maintains one flow table (table 0). A data-plane packet arriving at a switch is first checked against the rule with the highest priority and in table 0. If the header of the packet fits the match fields of that rule, a default *instruction* associates the packet with the corresponding action. Otherwise, the packet is checked against flow entries with lower priorities, and if no matching rule is found, the packet is dropped.

Interaction between the planes: FlowMod commands. The flow tables installed on the switches across the data plane determine the network *policy*. Controllers can change the policy by sending control messages containing *FlowMod commands*. In a nutshell, a *FlowMod* command either specifies a new flow entry or a modification to an existing flow entry. A FlowMod command can either add a flow entry or delete a flow entry. The standard processing of a FlowMod **add** command received by a switch is as follows. If the switch already maintains a flow entry with *exactly* the same match and priority level, then the new flow entry will simply *replace* it. Otherwise, the flow entry will be installed in addition to existing ones. A **delete** command simply removes an existing flow entry with the same match and priority, or does nothing if such an entry is not present.

In order to avoid inconsistencies caused by different rules with overlapping match fields, a *FlowMod* command can be equipped with the `check_overlap` flag which will be instrumental in implementing our synchronization abstractions: if the switch maintains a flow entry with an overlapping but not identical match part with the same priority, then the *FlowMod* command will fail.

We will use the following notation to create a *FlowMod* message: *FlowMod*(*match*, *op*, *action*, *flags*), where *op* is **add** or **delete**, and *flags* are used, e.g., to activate the `check_overlap` feature. For simplicity, we omit other standard parameters, assuming them to carry default values.

To read the current configuration of a switch (the existing flow entries), the controller should send the OpenFlow *add flow monitor* command with the ofpt_initial optional flag. In this paper, we write **read-state** for sending this message, receiving a response, and returning the received configuration.

Bundling. Another important feature in OpenFlow is *bundling*. A controller can send multiple *FlowMod* commands equipped with the same *bundle identifier*. These commands are buffered temporarily by the switch until a *bundle commit* message is received from the controller. The bundled commands are then performed with *all-or-nothing semantics*: either all of them are performed, or none of them is. In particular, if a configuration request contained in one of the bundled commands cannot be applied (e.g., the command is rejected because of the set `check_overlap` flag and a conflicting flow entry), all other commands in the bundle are rejected and an error message is sent to the controller that issued them. The error message contains `xid`, the identifier of the first failed command in the bundle, and `error-code`, the identifier of the failure.

A bundle begins with a `ofpt_bundle_control` message of type `ofpbct_open_request` (creating the bundle), wraps each of its *FlowMod* command in a message `ofpt_bundle_add_message` equipped with the *bundle identifier*, and ends with a `ofpt_bundle_control` message of type `ofpbct_commit_request` (committing the bundle).

The bundle execution is *controller-atomic* in the sense that all controllers can only observe switch configurations

[1] We note however that hardware OpenFlow switches do not always follow the spec as they should [12], and will report on a prototype implementation later in this paper.

before or *after* the bundle commands are executed. We set the flag `ofpbf_atomic`, making bundles *packet-atomic*, in the sense that every data-plane packet is processed by a configuration before or after the bundle, and not by a configuration resulting from an incomplete bundle execution. We also set the flag `ofpbf_ordered`, to make sure that the bundle commands are executed in the order they were added to the bundle, thereby respecting dependencies between bundle commands.

Implementation.

At first sight, it may seem that bundling *FlowMod* commands with the `check_overlap` flag already provides a useful synchronization mechanism. Indeed, its "mini-transaction" nature allows a controller to install multiple flow entries in an atomic way. However, as is, bundling has important limitations.

First, except for some limited constraint types (such as overlapping flow spaces or insufficient free space), the bundle construct *per se* does not provide any means to modify the switch configuration based on application-specific conditions. But, as we have seen in our load-balancing example, supporting such conditions might be essential. Second, it is sometimes desirable to react to conflicts in smarter ways than by simply aborting an operation, e.g., by providing *conditional modifications*.

We now show that the OpenFlow match-action paradigm can be used to realize a transactional synchronization abstraction with the desired functionality. The meta-configuration can be realized using low-priority rules. Alternatively, we can leverage the possibility to maintain multiple flow tables at an OpenFlow switch, and use a separate flow table for the meta-configuration, making sure that the *pipelining* mechanism will never involve this flow table.

Recall that a transaction consists of a sequence of operations, each of which can either be a regular FlowMod command or a synchronization primitive, which, as we show below, translates into a sequence of FlowMod commands operating on the switch meta-configuration.

The implementation of **execute**$\{op_1, \ldots, op_k\}$ and **execute-transaction**$\{op_1, \ldots, op_k\}$ constructs invoked by a control application is simple. First we translate $op_1, \ldots, op_k$ into a sequence of FlowMod commands. If op_i is already a FlowMod command, the translation is trivial. If op_i is a synchronization primitive, we employ Algorithm 1 for *claim*, Algorithm 2 for *checkclaim*, Algorithm 3 for *unclaim*, Algorithm 4 for *write* and Algorithm 5 for *compare*.

To implement **execute**$\{op_1, \ldots, op_k\}$, we simply send the commands in the resulting sequence, one by one, to the switch. To implement **execute-transaction**$\{op_1, \ldots, op_k\}$, we additionally create a bundle, wrap each of the resulting commands in a bundle message with the corresponding bundle identifier, and complete it with a bundle commit message. Then we wait until the corresponding responses are received. If an error message is returned, it is forwarded to the control application, otherwise it is *ack*ed.

We now describe Algorithms 1-5 in more detail. Each of the operations generates one or two FlowMod commands. The *match* part in these commands specifies only the 64bit meta-data field in the header space. In defining the value and the mask, we use the notation $a \cdot b$ to represent a concatenation of two integers a and b: $a \cdot b := (a << s) + b$, where s is the size of b in bits. We also write 1^s (0^s) for a s bit string of 1's (0's). The action part in these commands is defined as an integer, which in practice can be implemented by a *set meta-data* instruction where the written value is that integer.

Algorithm 1 *claim*(x)

Require: *self* as the calling controller id.
1: $value \leftarrow 0^{16} \cdot self \cdot x$
2: $mask \leftarrow 0^{16} \cdot 1^{16} \cdot 1^{32}$
3: $match \leftarrow (value, mask)$
4: $action \leftarrow CLAIM\_MAGIC$
5: $cmd \leftarrow FlowMod(match, op = \textbf{add}, action)$
6: **return** *cmd*

Algorithm 2 *check*(x)

Require: *self* as the calling controller identifier.
1: $value \leftarrow self \cdot 1^{16} \cdot x$
2: $mask \leftarrow 1^{16} \cdot 0^{16} \cdot 1^{32}$
3: $match \leftarrow (value, mask)$
4: $action \leftarrow CLAIM\_MAGIC$
5: $flag \leftarrow$ `ofpff_check_overlap`
6: $cmd1 \leftarrow FlowMod(match, op = \textbf{add}, flag, action)$
7: $cmd2 \leftarrow FlowMod(match, op = \textbf{delete})$
8: **return** *cmd1*,*cmd2*

The id-claimer. In Algorithms 1-3, the match part of the constructed commands represents a concatenation of claimed identifier with a distinct 16bit controller identifier. This allows multiple controllers to claim the same identifier without overriding existing flow entries (Algorithm 1). For example, for any two resource identifiers x_1 and x_2 and controller identifiers $c_1 \neq c_2$, the match patterns $m_1 = (*^{16} \cdot c_1 \cdot x_1)$ and $m_2 = (*^{16} \cdot c_2 \cdot x_2)$ are distinct, even if $x_1 = x_2$. Therefore, they can both be used to claim and unclaim identifiers without affecting each other. In order to distinguish between claim entries and other policy entries, we set the action part of the entries storing claimed ids to be a unique designated value, *CLAIM_MAGIC*.

In order to check if an identifier x is claimed (Algorithm 2), a controller c tries to add a "tester" entry, where the match value field is $(c \cdot *^{16} \cdot x)$ and the flag `check_overlap` is set. Note that the controller identifier is shifted 16 bits to the left in the match field, which allows us to ensure that the entry is uniquely identified, while not conflicting with other entries. This attempt inflicts a failure in case another entry with claimed value x exists. If the check succeeds, i.e., no entry with claimed value x is present, the tester entry is deleted.

Note that the tester rule has no effect on claims and checks of other controllers. Hence the add and delete commands do not have to be executed atomically (e.g., within a bundle): the `check` operation "takes effect" at the moment it attempts to add the tester entry.

By employing the **read-state** command, we may easily implement an additional operation that returns all currently claimed ids: it is sufficient to go over the rules matching every controller identifier carrying action *CLAIM_MAGIC*.

Write and compare. The memory write and compare operations are implemented using similar techniques. To write a value k in a memory address *addr* (Algorithm 4), a

Algorithm 3 *unclaim*(x)

Require: *self* as the calling controller id.
1: $value \leftarrow 0^{16} \cdot self \cdot x$
2: $mask \leftarrow 0^{16} \cdot 1^{16} \cdot 1^{32}$
3: $match \leftarrow (value, mask)$
4: $cmd \leftarrow FlowMod(match, op = \textbf{delete})$
5: **return** *cmd*

Algorithm 4 *write*$(addr, k)$

1: $value \leftarrow 1^{32} \cdot addr$
2: $mask \leftarrow 0^{32} \cdot 1^{32}$
3: $match \leftarrow (value, mask)$
4: $cmd1 \leftarrow FlowMod(match, op = \textbf{delete})$
5: $value \leftarrow 1^{32} \cdot addr$
6: $mask \leftarrow k \cdot 1^{32}$
7: $match \leftarrow (value, mask)$
8: $action \leftarrow WRITE\_MAGIC$
9: $cmd2 \leftarrow FlowMod(match, op = \textbf{add}, action)$
10: **return** *cmd1*,*cmd2*

new flow entry is added, in which the value part of the match field contains *addr* and the mask part contains k. Therefore, the effective match string of the entry is $k' \cdot addr$, where k' is result of replacing every zero bit in k with a wildcard $(*)$. In order to rewrite a memory location, the old entry needs to be deleted and a new one need to be added atomically. Similarly to claim entries, we set the action to a magic value, *WRITE_MAGIC*, in order to distinguish them from policy entries.

To check whether a value k is written at a given address *addr* (Algorithm 5), a controller tries to add a new entry the same way as in the write operation, but additionally setting the `check_overlap` flag. If a different value k' is currently written, the entry inflicts a failure, as it overlaps but does not coincide with the existing entry. Indeed, the value parts of the two entries are identical, but the masks differ. In case the currently written value is k, the existing flow entry is replaced with an identical one. Before using an address *addr*, it is advised that a controller safely initializes it with a special initial value 0 by calling *compare*$(addr, 0)$: the first such command succeeds, and subsequent ones will do no harm.

4. APPLICATIONS

Our synchronization primitives can be used to implement powerful synchronization primitives such at *compare-and-set* (CAS). When executed within a transaction, a $CAS(addr, old, new)$ operation on a memory location *addr* checks if the content of *addr* is *old* and if so, replaces it with *new* (the CAS *succeeds*), otherwise it causes an abort of the invoking transaction (the CAS *fails*). It is straightforward to implement the CAS operation by executing *compare*$(addr, old)$ followed by *write*$(addr, new)$ within the **execute-transaction** construct.

Let us now revisit our load-balancing application discussed in Figure 1. A simple strategy to balance load in our 2-link example (Figure 1) fails in the presence of multiple controllers, even if bundling and `check_overlap` features are used: the flows are defined over "independent" flow spaces.

Using CAS, we can implement a generic template for

Algorithm 5 *compare*$(addr, k)$

1: $value \leftarrow 1^{32} \cdot addr$
2: $mask \leftarrow k \cdot 1^{32}$
3: $match \leftarrow (value, mask)$
4: $action \leftarrow WRITE\_MAGIC$
5: $flag \leftarrow$ `ofpff_check_overlap`
6: $cmd \leftarrow FlowMod(match, op = \textbf{add}, flag, action)$
7: **return** *cmd*

Algorithm 6 Policy update with only CAS

Require: policy update function UPDATE, policy id address *paddr*
Ensure: installed policy is consistent with previous one
1: **repeat**
2: *pol-id*, *policy* ← **read-state**
3: *update_cmds* ← UPDATE(*policy*)
4: **execute-transaction**{
5: $CAS(paddr, pol\text{-}id, pol\text{-}id + 1)$,
6: *update_cmds*
7: } → *res*
8: **until** *res* = *ack*
9: **return** *res*

concurrent policy composition via a user-specific UPDATE() function [2]. This can be, for example, a specific load-balancing function LB_UPDATE, which, when applied atomically, excludes the pathological scenario described in Figure 1. We use CAS to exactly achieve this atomicity.

Let us consider a simple policy update protocol that allows a set of controllers to concurrently update switch policies, i.e., sets of effective flow entries, under the condition that these rules can be *composed* with the currently installed policy. Algorithm 6 uses the bundle feature to update the switch configuration, where the configuration contains both the policy and a meta-configuration memory address *paddr* that holds the *policy identifier (pol-id)*. In the algorithm, the controller first reads the currently installed policy together with its *pol-id*, applies the UPDATE() function to the current policy which results in a set of update FlowMod commands and then tries to apply them atomically together with a CAS operation on *paddr* replacing *pol-id* with *pol-id*+1. The CAS operation ensures that if, concurrently, a new policy (with a different identifier) has been installed, the update fails and takes no effect. Note that, since all policy modifications are executed within a bundle providing controller-atomicity, the **read-state** command in Line 2 appears atomic. Algorithm 6 can therefore also be used to render a naïve, single-controller load-balancing algorithm consistent, without any modifications in the update function.

Algorithm 6 is correct assuming that policy identifiers may grow without bound. In practice, we can use a modulo scheme on the 32-bit field of a flow entry used for storing the policy identifiers, which, in principle, may lead to inconsistencies if controllers proceed in different speeds. Algorithm 7 adresses this issue by using our id-claimer abstraction. Here, the controller first reads the current policy id and claims it, which prevents another controller from using the id for a different policy (Lines 2-4). Then the controller chooses any *unused* id and computes the update commands to be executed (Line 10). Finally, within an atomic transaction (**execute-transaction**), it verifies whether the chosen pol-

Algorithm 7 Advanced policy update

Require: policy update function UPDATE, policy id address *paddr*, id space C.
Ensure: installed policy is consistent with previous one

repeat
  $pol\text{-}id, claims, policy \leftarrow$ **read-state**
  execute$\{claim(pol\text{-}id)\}$
  $pol\text{-}id2, claims, policy \leftarrow$ **read-state**
  if $pol\text{-}id \neq pol\text{-}id2$ **then**
    execute$\{unclaim(pol\text{-}id)\}$
    continue (**restart** loop)
  end if
  $my\_id \leftarrow$ choose a number from $\{C \setminus claims\}$
  $update\_cmds \leftarrow$ UPDATE$(policy)$
  execute-transaction$\{$
    $check(my\_id)$,
    $CAS(paddr, pol\text{-}id, my\_id)$,
    $update\_cmds$
  $\} \rightarrow res$
  execute$\{unclaim(pol\text{-}id)\}$
until $res = ack$
return res

icy id has not been claimed and that the policy has not been concurrently modified, and then tries to update the policy.

A note on efficiency. Our implementations of synchronization primitives (Algorithms 1-5) require only a constant number of flow entries per memory address or claim. The number of used addresses is constant for Algorithm 6 and linear in the number of controllers for Algorithm 7.

5. PROOF-OF-CONCEPT

To investigate the feasibility of our approach, we implemented a proof-of-concept of our algorithms as a python library which wraps ovs-ofctl, a well-known command line tool for monitoring and administering OpenFlow switches. The code is made available online[2].

We experimented with the low-level abstractions as well as the transactional policy update algorithm of our proof-of-concept, using different workloads. The tests were conducted using a software switch, Open vSwitch (version 2.4.0) and a virtual network simulator, Mininet.

For instance, we experimented with a setting where three processes (controllers), concurrently issue 1000 concurrent policy updates each, where every update replaces a single rule with another one. We could confirm that despite the concurrency, 100% of the updates eventually completed in committing their transactions, possibly after several aborts caused by conflicts.

6. CONCLUSION

We believe that our centralized in-band synchronization approach is natural and attractive, as it avoids the complexities and limitations of distributed out-of-band agreement protocols, which may introduce additional points of potential failures. Moreover, the switch is the element implementing the controller policies anyway. The mechanisms presented in this paper are simple and can be implemented using the *standard OpenFlow* protocol: they do not require any protocol/hardware extensions as postulated in recent literature [2, 4]. Our work may hence also contribute to the ongoing discussion of what can be implemented in-band in today's OpenFlow protocol [21], as well as to useful high-level concurrency objects and language abstractions [18]. We see our paper as a first step, and more work is required to investigate and compare useful in-band synchronization abstractions, also in terms of performance. While our framework can be used as a synchronization library, we also work on integrating it into an existing SDN controller, e.g., Ryu.

[2]See `https://github.com/lironsc/of-sync-lib`.

7. REFERENCES

[1] Berde et al. ONOS: Towards an Open, Distributed SDN OS. In *Proc. ACM HotSDN*, pages 1–6, 2014.
[2] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust sdn control plane for transactional network updates. In *Proc. 34th IEEE INFOCOM*, 2015.
[3] Curtis et al. Devoflow: Scaling flow management for high-performance networks. In *Proc. SIGCOMM*, pages 254–265, 2011.
[4] H. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soule. Netpaxos: Consensus at network speed. In *Proc. ACM SOSR*, 2015.
[5] Dixit et al. Towards an Elastic Distributed SDN Controller. In *HotSDN*, 2013.
[6] Ferguson et al. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*, 2013.
[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2), 1985.
[8] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory,Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.
[9] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *HotSDN*, 2012.
[10] M. Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.
[11] Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.
[12] M. Kuniar, P. Peresni, and D. Kostic. What you need to know about sdn flow tables. In *Proc. Passive and Active Measurement (PAM)*. 2015.
[13] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.
[14] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It's good to relax! In *Proc. ACM PODC*, 2015.
[15] Ludwig et al. Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies. In *HotNets*, 2014.
[16] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.
[17] McKeown et al. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.
[18] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *NSDI*, 2013.
[19] Padon et al. Decentralizing SDN Policies. In *Proc. ACM POPL*, 2015.
[20] Reitblatt et al. Abstractions for Network Update. In *SIGCOMM*, 2012.
[21] Schiff et al. Reclaiming the brain: Useful openflow functions in the data plane. In *Proc. ACM HotNets*, 2014.