In-Band Synchronization for Distributed SDN Control Planes

Liron Schiff[1], Stefan Schmid[2], Petr Kuznetsov[3]

[1] Tel Aviv University, Israel; [2] TU Berlin & T-Labs, Germany; [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 over-heads, 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 the well-known concept of *transactions*. A transaction aggregates a sequence of control messages and specific synchronization primitives, providing all-or-nothing semantics. We argue that a powerful transactional interface should and can be implemented *in-band*, using standard OpenFlow commands on the data plane configuration. We discuss how to use our synchronization framework to implement consistent policy composition and fault-tolerant control-plane applications.

1. INTRODUCTION

By consolidating and outsourcing the control over the dataplane 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 *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. [18]

Consider, for example, the problem of consistent instal-lation of new forwarding policies, stipulating routes that packets of different header spaces should follow across the network [13,15,19]. 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.). [14,15] 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. E.g., the popular *transactional memory* abstraction [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 conflicts 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 does not affect neither traffic nor other controllers.

We promote to implement our synchronization constructs *in-band*, on the 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 protocols are often considered too high, both in terms of the necessary computability assumptions about the underlying system [7], and the amount of communication complexity [12]. 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.

Interestingly, our mechanisms are simple and can be implemented using the standard *OpenFlow* protocol (version 1.4 [16]). Concretely, in our implementation, we use a part of the data-plane configuration space as a *shared memory* that stores information about contention and conflicts between the controllers. A transaction can then contain standard OpenFlow control operations as well as *synchronization primitives* operating on this shared memory. Our synchronization primitives allow the controllers to define general

notions of conflicts between configuration updates, not restricted to simple conflicts on the overlapping flow spaces. In particular, we can define dependencies between mappings of independent flows to the physical infrastructure, which is important, e.g., for load-balancing applications.

We then show that, using our abstraction, the controllers can solve several important synchronization problems. We describe an implementation of the fundamental *compare-and-set* (CAS) primitive, which in turn can be used to solve consensus and implement a generic replicated control plane service [10] in a consistent and fault-tolerant way: a mainstream building block in modern system design. We also discuss a simple CAS-based concurrent policy-update mechanism, and use our synchronization abstraction to provide the missing link for the read-modify-write object postulated in [2].

Organization The rest of the paper is organized as follows. In Section 2, we review the basics of the OpenFlow protocol and discuss the features which are relevant for our work. In Section 3, we highlight the limitations of the OpenFlow primitives and motivate the need for transactions. We then present our approach in detail in Section 4, and discuss use cases in Section 5. We conclude our paper in Section 6.

2. BACKGROUND: SDN AND OPENFLOW

Control and data planes. Software-defined networking decouples the *control plane*, software controllers that configure the network, from the *data plane*, the devices that implement the network. In OpenFlow, the *de facto* standard SDN protocol today, the data plane consists of *OpenFlow switches*. In a nutshell, 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.

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 a the `check_overlap` flag: if the switch maintains a flow entry with an overlapping but not identical match part with the same priority but a different action, and if the two rules have the same priority, then *FlowMod* will fail. The `check_overlap` flag in a *FlowMod* command helps resolving simple conflicts between controllers, and will be instrumental in implementing our synchronization abstractions.

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-config** 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 *error code* corresponding to 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 considered *controller-atomic* in the sense that all controllers can only observe switch configurations *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

Algorithm 1 *Naïve Flow Balancing*

Require: *new_flows*: flow entries of the new flows, *ports*: a list of switch output ports,
1: *policy* ← **read_config**
2: *update_cmds* ← LB_UPDATE(*policy*)
3: **send**{*update_cmds*}

    Function LB_UPDATE(*policy*)
4: *flows_per_port* ← [0, 0, . . . , 0]
5: **for all** *entry* ∈ *policy* **do**
6:     *port* ← *entry.output_port*
7:     *flows_per_port*[*port*] + +
8: **end for**
9: *new_flow_mod_cmds* ← {}
10: **for** *entry* ∈ *new_flows* **do**
11:     *port* ← arg min$_{p \in ports}$ *flows_per_port*[*p*]
12:     *entry.output_port* ← *port*
13:     *flows_per_port*[*port*] + +
14:     *cmd* ← *FlowMod*(*add*, *f*)
15:     *new_flow_mod_cmds.add*(*cmd*)
16: **end for**
17: **return** *new_flow_mod_cmds*

---

were added to the bundle, thereby respecting dependencies between bundle commands.

3. LIMITATIONS AND MOTIVATION

At first sight, 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, as we will argue in the following.

First, except for some limited constraint types (such as overlapping flow spaces or insufficient free space), a bundle does not provide any means to modify the switch configuration based on application-specific conditions, which depend on the current configuration, However, as we will argue, a support for more generic conditions under which configuration updates should or should not take place, is often desirable. Second, 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 two examples.

Example 1: Handling complex dependencies. Let us consider two controllers in charge of *load-balancing*, see Figure 1 for a simplistic example with two links. In such a scenario, seemingly independent actions, defined over completely independent logical flow spaces (say, two different TCP micro-flows), 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 à la Algorithm 1, they may involuntarily unbalance the flow allocation. Simple flow space overlap checks cannot be used to detect such conflicts. Note that, assuming that all these flows are completely independent, bundling and `check_overlap` features do not really help here.

Example 2: Supporting modifications. Consider the arguably very basic problem of *modifying* previously installed forwarding rules in a consistent manner. In principle,
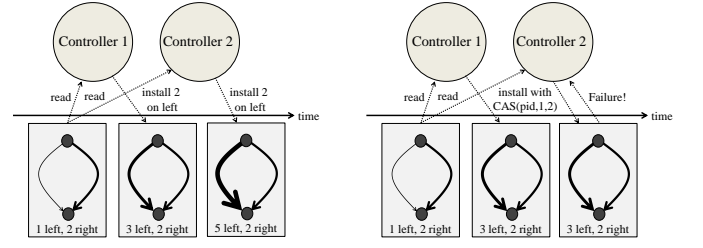

Figure 1: *Left:* **Without synchronization, the two controllers naturally choose to install their flows on the left link, which results in an undesirable unbalanced state.** *Right:* **With synchronization, i.e., by bundling the flow installation with a Compare-and-Set (CAS) primitive depending on the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

a bundle allows to conditionally add new rules or remove old rules, and, using the flow space overlap check feature `check_overlap`, to abort the transaction if potentially conflicting rules were already there. However, it is not obvious how to use the bundle to modify *existing* rules, especially if the modification depends on specific conditions. In other words, the bundle and overlap check only help for the *conditional creation* of rules, not their *modification*.

Our Approach. This paper proposes mechanisms to solve these two examples, by using a minimal "meta-configuration" space. That is, controllers store information on contention and conflicts between concurrently installed policy updates on the switch. 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.

We will later come back to these examples, and show that Algorithm 1 can actually be used without any modifications within our framework. Consistent load-balancing can also be achieved using our atomic Compare-and-Set instruction based on policy ids (pids), see Figure 1 (*right*).

4. SYNCHRONIZATION ABSTRACTIONS

We now describe how our meta-configuration space at an OpenFlow switch can be implemented and manipulated using our synchronization framework.

Configuration space. We distinguish 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, provided a total addressing scheme.

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. One way to achieve this is to set a low priority to the meta flow entries and ensure that the regular flow entries maintain higher priority levels. Alternatively, we could 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.

Synchronization. Our framework provides the controllers with the ability to execute sequences of *operations* on the switch configuration in an *atomic* (transactional) 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. An operation here can 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.

We propose the following constructs **execute**$\{op_1, \ldots op_k\}$ and **execute-atomic**$\{op_1, \ldots op_k\}$ which execute a sequence of commands $op_1, \ldots, op_k$ in a regular and atomic way, respectively. The construct returns *ack* if the sequence of operations is performed and all commands are performed successfully, and `error(xid, error-code)` received from the switch otherwise, providing the identifier of the first command in the sequence that failed together with the corresponding error code.

Synchronization primitives. We focus first on the consistent use of *policy identifiers*, an important notion in consistent policy updates [2, 19]. 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 operations. 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 FlowMod commands 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* (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*).

- If *k* is currently claimed by any controller, *check*(*k*) aborts the current transaction.

Implementation. We now describe how to implement our transactional abstractions using standard OpenFlow features. Essentially, we translate each of the transactional operations into a sequence of FlowMod commands.

The implementation of **execute**$\{op_1, \ldots, op_k\}$ and **execute-atomic**$\{op_1, \ldots, op_k\}$ 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 2 for *claim*, Algorithm 3 for *checkclaim*, Algorithm 4 for *unclaim*, Algorithm 5 for *write* and Algorithm 6 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-atomic**$\{op_1, \ldots, op_k\}$, 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 return, it is forwarded to the control application, otherwise it is *ack*ed.

We now describe Algorithms 2-6 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, leaving the remaining fields at default values. In defining the value and the mask, we use the notation $a \cdot b$ to represent a concatenation of two 32bit integers a and b: $a \cdot b := (a << 32) + b$. We also write `1...1` for a 32bit string of 1's used in a mask. 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.

The id-claimer. In Algorithms 2-4, the match part of the constructed commands represents a concatenation of two claimed identifiers. In addition, every controller uses a unique mask part of the match defined as a distinct 32bit controller identifier concatenated with `1...1`. This allows multiple controllers to claim the same identifier without overriding existing flow entries. For example, for any two resource identifiers x_1 and x_2 and controller identifiers $c_1 \neq c_2$, the match patterns $m_1 = (x_1 \cdot x_1, c_1 \cdot 1...1)$ and $m_2 = (x_2 \cdot x_2, c_2 \cdot 1...1)$ 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 check if an identifier is claimed, a controller tries to add a new entry setting the flag `check_overlap`, and using a different action (2 instead of 1), thereby inflicting a failure in case an entry with overlapping match value exists. If the check succeeds, we delete the entry. Considering m_1 and m_2 from the example above, with respective actions $a_1 = 1$ and $a_2 = 2$, they overlap with different actions iff $x_1 = x_2$. Therefore, if m_1 is used in the check after m_2 was used in the claim operation, a conflict is detected iff $x_1 = x_2$, as desired.

Note that in order to save space, when checking for claims, also the deletion of the added tester flow entry is included, which however has no effect on claims and checks of other controllers; hence, the deletion command does not have to be sent as part of a bundle. Therefore, the **check** operation is atomic by nature, similar to the **claim** and **unclaim** operations, and is implemented as essentially one command.

By employing the **read-config** 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.

Write and compare. The memory write and compare operations can be implemented using similar techniques. As can be seen in Algorithms 5-6, the match part of the added flow entry is used as the memory address and the action as

Algorithm 2 $claim(x)$

Require: $self$ as the calling controller id.
1: $value \leftarrow x \cdot x$
2: $mask \leftarrow self \cdot 1 \ldots 1$
3: $match \leftarrow (value, mask)$
4: $action \leftarrow 1$
5: $flag \leftarrow 0$
6: $cmd \leftarrow FlowMod(match, op = \mathbf{add}, flag, action)$
7: **return** cmd

Algorithm 3 $check(x)$

Require: $self$ as the calling controller identifier.
1: $value \leftarrow x \cdot x$
2: $mask \leftarrow self \cdot 1 \ldots 1$
3: $match \leftarrow (value, mask)$
4: $action \leftarrow 2$
5: $flag \leftarrow \mathtt{ofpff\_check\_overlap}$
6: $cmd1 \leftarrow FlowMod(match, op = \mathbf{add}, flag, action)$
7: $cmd2 \leftarrow FlowMod(match, op = \mathbf{delete})$
8: **return** $cmd1, cmd2$

Algorithm 4 $unclaim(x)$

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

Algorithm 5 $write(addr, k)$

1: $value \leftarrow addr$
2: $mask \leftarrow 1 \ldots 1.1 \ldots 1$
3: $match \leftarrow (value, mask)$
4: $action \leftarrow k$
5: $flag \leftarrow 0$
6: $cmd \leftarrow FlowMod(match, op = \mathbf{add}, flag, action)$
7: **return** cmd

---

the written value. Values written to the same address will replace one another, following the command definition to replace the old flow entry with a new one in case they share the same match parts.

In order to check if a value is written at a given address, we try to add it the same way as in the write operation, but we also set the `check_overlap` flag. If the value is not there, our action differs from the action of the existing entry, thereby inflicting a failure. In case the value is there, our flow entry replaces the existing flow entry with an identical one.

5. APPLICATIONS

Our synchronization mechanisms can be useful in many situations. For example, we can implement the fundamental *Compare-and-Set* (CAS) primitive. 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-atomic** construct.

CAS can be used to solve the fundamental problem of fault-tolerant *consensus* [7] and allows us to implement any generic replicated state machine [10] in the control plane, in the *wait-free* manner, i.e., tolerating asynchrony and failure of any number of controllers. As such, CAS can also be used to implement STN [2], by providing the postulated *read-modify-write* object.

Our mechanism can also be used as a generic template to implement concurrent policy compositions of a user-specific UPDATE() function [2]. This way, for example, by choosing our load-balancing function LB_UPDATE (Example 1 in Section 3), the pathological scenario described in Figure 1 can be resolved.

Let us first 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 7 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 (pid)*. In the algorithm, the controller first reads the currently installed policy together with its pid, 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 pid with $pid + 1$. The latter ensures that if, concurrently, a new policy (with a different identifier) has been installed, the update fails and takes no effect. Algorithm 7 can therefore also be used to render our naïve load-balancing Algorithm 1 consistent, without any modifications in the update function.

Algorithm 8 is a more efficient alternative to Algorithm 7. It uses our id-claimer abstraction to bound the number of used policy identifiers. Here, the controller first reads the current policy id and claims to prevent another controller from using it for a different policy (Lines 2-4). Then the controller claims the first *unused* id and computes the update commands to be executed (Line 10). Finally, within an atomic transaction (**execute-atomic**), it verifies whether the chosen policy id has not been claimed and that the policy has not been concurrently modified, and then updates the policy.

6. CONCLUSION

This paper initiated the study of mechanisms which allow distributed controllers to synchronize their possibly concurrent actions. We believe that our in-band approach is natural and attractive as it can avoid the complexities and limitations of distributed out-of-band agreement protocols. Moreover, our mechanisms 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 [20], as well as to useful high-level concurrency objects and language abstractions. [17] We also plan to release a small library with our synchronization primitives. Moreover, we are investigating possible generalizations of our transactional guarantees, across multiple switches [2], as well as opportunities for more fine-grained *partial* updates that concern subsets of flow entries, thus

Algorithm 6 $compare(addr, k)$
1: $value \leftarrow addr$
2: $mask \leftarrow 1\ldots1.1\ldots1$
3: $match \leftarrow (value, mask)$
4: $action \leftarrow k$
5: $flag \leftarrow \texttt{ofpff\_check\_overlap}$
6: $cmd \leftarrow FlowMod(match, op = \textbf{add}, flag, action)$
7: **return** cmd

Algorithm 7 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:     $pid, policy \leftarrow$ **read-config**
3:     $update\_cmds \leftarrow$ UPDATE$(policy)$
4:     **execute-atomic**{
5:         $CAS(paddr, pid, pid + 1)$,
6:         $update\_cmds$
7:     } $\rightarrow res$
8: **until** $res = ack$
9: **return** res

Algorithm 8 Advanced policy update
Require: policy update function UPDATE, policy id address $paddr$, id space C.
Ensure: installed policy is consistent with previous one
1: **repeat**
2:     $pid, claims, policy \leftarrow$ **read-config**
3:     **execute**{$claim(pid)$}
4:     $pid2, claims, policy \leftarrow$ **read-config**
5:     **if** $pid \neq pid2$ **then**
6:         **execute**{$unclaim(pid)$}
7:         **continue** (**restart** loop)
8:     **end if**
9:     $my\_id \leftarrow$ choose a number from $C \setminus claims$
10:    $update\_cmds \leftarrow$ UPDATE$(policy)$
11:    **execute-atomic**{
12:       $check(my\_id)$,
13:       $CAS(paddr, pid, my\_id)$,
14:       $update\_cmds$
15:    } $\rightarrow res$
16:    **execute**{$unclaim(pid)$}
17: **until** $res = ack$
18: **return** res

improving concurrency.

7. REFERENCES

[1] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. 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. Software transactional networking: Concurrent and consistent policy composition. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, August 2013.

[3] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. 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] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an Elastic Distributed SDN Controller. In *HotSDN*, 2013.

[6] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthy. 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] 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.

[12] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

[13] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It's good to relax! In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, 2015.

[14] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies. In *HotNets*, 2014.

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

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

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

[18] O. Padon, N. Immerman, A. Karbyshev, O. Lahav, M. Sagiv, and S. Shoham. Decentralizing SDN Policies. In *Proc. ACM POPL*, 2015.

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

[20] L. Schiff, M. Borokhovich, and S. Schmid. Reclaiming the brain: Useful openflow functions in the data plane. In *Proc. ACM HotNets*, 2014.