On the Synchronization Power of OpenFlow

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 modern Software-Defined Networks (SDNs) are typically *distributed systems*: to achieve availability, load balancing and responsiveness, control modules are installed on multiple machines. Unfortunately, these goals are hard to combine with consistent network manipulation and certain synchronization challenges among the distributed controllers must be addressed. In this paper, we show that powerful synchronization mechanisms can be implemented *in band*, simply using today's Openflow switches. We show that standard OpenFlow features allow for providing simple but general-purpose transactions with all-or-nothing semantics. We then show that these transactions can be used to reach consensus among software controllers and, even more surprizingly, to resolve *on-the-fly* conflicts between concurrent policy updates in an effectively atomic way, without affecting the ongoing traffic.

1. INTRODUCTION

By consolidating and outsourcing the control over the dataplane switches to a logically centralized controller, the concept Software-Defined Networking (SDN) simplifies network management and facilitates faster innovations, as the (software) control plane is decoupled from the (hardware) data plane and can evolve independently. The distinct network control plane has the potential to simplify network management and opens new opportunities for innovation, but implementing it in a scalable and consistent manner is highly non-trivial.

Indeed, there is a wide consensus that to achieve availability, responsiveness, and balancing of network management tasks, the *logically* centralized SDN control plane must be physically *distributed*. First, in order to provide a high availability as well as a minimal degree of fault-tolerance, controllers should be redundant [2, 3, 16]: 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 control plane events close to their origin. [5, 12, 24, 26] Third, larger SDNs are likely to be operated by multiple administrators or may even offer participatory interfaces where different users can install and trigger policy changes concurrently [3,7]. All this, however, comes with the need for a *synchronization* between distributed controllers, assuming that we want them to manipulate the network *consistently* [4]. And this task is highly nontrivial [11].

[*]Supported by the European Research Council (ERC) Starting Grant no. 259085 and by the Israel Science Foundation Grant no. 1386/11.

Today, unfortunately, 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).

Consider, for example, the particularly interesting problem of consistent installation of new forwarding policies, stipulating routes that packets of different header spaces should follow across the network [3, 9, 20, 25]. Installing conflicting forwarding rules, e.g., rules of the same priority defined over non-disjoint flow spaces may lead to pathological network behavior (loops, bleckholes, etc.). Similarly, installing diverging load-balancing policies may, when combined, *increase* the load [18]. To render things more difficult, controllers may also fail, even before their updates have been completed.

In this paper, we look closely at the most popular specification language for SDN control-data plane exchange - OpenFlow [22]. At a high level, the specification represents the exchange in the form of *control messages*, containing commands for installing or modifying *rules* that, in turn, constitute the *flow table* of a data-plane switch used in processing the ongoing data-plane traffic.

We observe that the standard provides the option of *bundling* these messages in one, so that either all of the control messages are processed or none of them is. In particular, if, for some reason, the configuration request contained in one of the bundled messages cannot be processed, all the other requests are rejected and an error message is sent to the controller that issued them. Interestingly, this bundled processing can be, when properly configured, done in a *packet-atomic* way, so that each packet is processed according to the switch configuration either *before* or *after* the request was processed.

Intuitively, if we treat the flow table at the switch as a *shared memory* that can be concurrently read and modified by multiple controllers, the bundling features can be seen as a *multi-write* or, more widely, *multi-modify* operation that modifies multiple memory location in one atomic step. This ability is reminiscent of the hypothetical multi-write shared-memory model which is known to enable solutions for important synchronization problems, such as consensus, impossible to solve otherwise [1]. Indeed, we describe an algorithm that solves consensus among any number of controllers using a single OpenFlow switch that accepts bundle control messages.

Even more, we observe that the standard also assumes *conditional* application of certain types of control messages. A specific OFPFF_CHECK_OVERLAP flag set in the control message containing a new forwarding rule ensures that the rule is applied if and only if it does not *conflict* with any other rule currently in the flow table. This in-band conflict detection mechanism, combined with bundling, allows us to implement specific types of *transactions* that may atomically install multiple rules if certain compatibility conditions are met.

We then discuss how these synchronization mechanisms can be used to achieve consistent network management. We first show that they can be used to replace the read-modify-write primitives voluntarily assumed in the recent implementation of a recent proposal for a consistent composition system for forwarding policies [4]. We then use these transactions directly to implement a lightweight forwarding policy composition system that requires no additional synchronization on the control plane. We then explore how more general classes of policies can be composed *on-the-switch.*

Our results highlight the surprising synchronization power hidden in the OpenFlow standard. The synchronization mechanisms we describe in this paper turn out to be crucial for ensuring consistency of network management on the control plane while preserving consistency of the traffic processed on the data plane.

The rest of the paper is organized as follows. In Section 2, we overview the basics of the OpenFlow protocol and recall the bundle and ofpff_check_overlap features. In Section 3, we specify the implementations of our OpenFlow-based sycnhronization mechanisms. In Section 4, we discuss a few policy composition algorithms that benefit from these mechanisms. In Section 9, we discuss open questions and related work.

2. SDN PRIMER AND OPENFLOW FEATURES

Define the protocol (from Reclaiming the Brain"?) and discuss bundling and check-overlap.

3. TRANSACTIONS IN OPENFLOW

Multi-write and read-modify transactions.

3.1 Dealing with multiple updates

TBD - adding the notion of versions and cleanup/GC. probably no good way to solve infinite versions... maybe check literature on shared memory and multiple writes.

3.2 Improving the scheme with check-overlap

Till now we considered a model that allows multiple updates to match-action entries. Here we extends the model with actions create-entry and delete-entry that may failed and abort the whole transaction depending on the existence on non-existence of specific entries prior to the action.

Next we show how such actions can be used to implement consensus with much less resources. We use entry id as a property used to reference specific entry.

4. APPLICATIONS AND RAMIFICATIONS

Algorithm 1 Advanced Update Algorithm

```
Require: update requests $U$
Ensure: installed policy is consistent with previous one
 1: if first time then
 2:     repeat
 3:         $my\_cur\_id \leftarrow$ unused entry id
 4:         $res \leftarrow$ create entry with id $my\_cur\_id$
 5:     until $res = False$
 6: end if
 7: repeat
 8:     $cur\_id \leftarrow$ read current entry id
 9:     $cur\_policy \leftarrow$ read current policy
10:     $my\_next\_id \leftarrow$ unused entry id
11:     $new\_policy \leftarrow$ apply update requests on $cur\_policy$
12:     start transaction
13:         delete entry with id $1 << (\log n) + cur\_id$
14:         create entry with id $1 << (\log n) + my\_cur\_id$
15:         add $new\_policy$ rules conditioned on $metadata = my\_cur\_id$
16:         update policy selector to $my\_cur\_id$
17:     $res \leftarrow$ commit transaction
18:     $res \leftarrow$ commit transaction
19: until $res = False$
```

5. THE UNDERLYING THEORY

idea: multi-write consensus, a not well-known result!

In order to solve consensus we suggest to use the atomic multiple entries update capability (which is discussed in OpenFlow standard). A shared memory system that supports atomic write to (enough) multiple locations is known to allow the implementation of consensus objects. For example consider the following implementation utilizing n^2 memory locations, $M_{i,j}$, where $i, j \in [n]$, and the suggested values $\{v_i\}_{i\in[n]}$. All locations are initialized to zero.

In order to participate, a server i, writes its suggested value V_i and atomically rewrites row i ($M_{i,*}$) with value "1" and column i ($M_{*,i}$) with value "2". After the atomic write, a server i first reads the diagonal ($M_{j,j}$ for $j \in [n]$), and consider all observed servers as candidate set S. Then server i reads all candidates intersections (locations $M_{j,k}$ where $j, k \in S$. And computes the winner server id w that was overwritten by all other candidates, i.e. such that $M_{w,k} = 2$ for all $k \in Sw$. The consensus value is v_w. Note that a snapshot could replace the two reading phases.

6. BASIC ONE-TOUCH MECHANISM

We address the problem of committing consistent policies to SDN switch concurrently by multiple controllers. We show that assuming minimalistic control protocol, SDN switch configuration space can be used to implement a consensus object thereby allowing controllers to agree on each next policy update.

6.1 SDN switch configuration space as shared memory

Our first observation is that a match-action configuration which is accessed by multiple controllers can be considered as shared memory. Note however that standard memory map and index/offset to a word, while Flow Table entries are not necessarily indexed. However, we assume that each flow entry has some property that allows the operator to specifically reference and update it, for example OpenFlow entry's cookie. In cases where this property value can't be determined by the operator, we can use other variable entry properties to store the index, for example we can represent memory value v in offset i by an match-action entry whose match is $in\_port = i$ and action is "forward port x".

Going back to the policy update problem, the consensus scheme allow all servers to decide on next policy by using consensus value as policy identifier, and writing the policy in advance. However this scheme doesn't actually apply the policy on incoming packets and it remains to configure the switch according to the next policy once it is decided. Making this scheme fail safe (handling failure of winning server) can be ensured by allowing any server to reconfigure the switch.

6.2 One-Touch

In some scenarios it may be desirable to shorten the policy update time by avoiding the consensus reading and policy configuration step. We define this as the one touch policy update, namely we require each server to contact the switch only once in order to (try and) update the policy, in other words participating in reaching consensus and applying it with one atomic write. Solving this problem also has significance in understanding the strength of the SDN switch computation model.

The main idea of our solution is to adjust the previous policy update scheme in a way that performs consensus validation phase during every packet processing thereby requiring the servers only to make the first phase of the consensus - atomically writing the matrix. In order to allow the packets to validate the consensus we make the following observation: while servers can be abstracted as memory writers, the packets processed by the switch are the readers. OpenFlow standard define an atomic write transaction as being atomic in relation to the processed packets thereby making packet processing a multiple read transaction but one read location per table is allowed. This means that if each memory location used in the consensus scheme will be stored in different table then each packet will be able to read all of them.

However, reading each memory location is not enough, as the scheme requires to make a validation that involves multiple values. We utilize the metadata field in order to store all read values, in more details, each memory value $M_{i,j}$ is stored in offset i+j*n in the metadata field. Once all values are in the metadata field, we can detect the consensus winner by trying different matches on the metadata. Although we can have one match entry per possible matrix state, this solution requires exponential number of entries. A better way which we describe next requires only $O(n^2)$ entries utilizing n tables.

Our compact validation use additional temp variable, $temp_winner$, to hold the current best consensus candidate. Each of the n validating tables checks different "column" in the matrix and update $temp_winner$ to keep track of the winner so far (after examining a prefix sub set of the columns). The entry t in table k ($0 <= k < n$) matches the case where current winner=t and location (t, k) in the matrix (packet header) equals 1 and location (k,k) is non-zero. After all validating tables are processed, $temp_winner$ stores the id of the winning server which can be used to match on its policy rules (filtering out other policies) or to use the goto table command to jump to a designated policy table of the server.

7. THE PRACTICAL IMPLEMENTATION

8. RELATED WORK

There exists a wide consensus that SDN control planes need to be distributed. [2,6,16] Onix [16] is one of the earliest proposals, and introduced the the notion of Network Information Base (NIB), abstracting network state distribution from control logic, but requiring mechanisms for the detection and resolution of conflicts. Also spatially distributed control planes to improve scalability and latency have been studied intensively in the literature [12, 13, 27] proposes an elastic distributed controller architecture.

Operating and updating SDNs consistently is already nontrivial from the perspective of a single controller. Reitblatt et al. [25] introduced the notion of per-packet consistency and proposed a 2-phase update protocol based on tagging. Mahajan and Wattenhofer [20] considered weaker transient consistency guarantees, and proposed more efficient network update algorithms accordingly. Ludwig et al. [19] studied algorithms for secure network updates where packets are forced to traverse certain waypoints or middleboxes. Ghorbani et al. [10] recently argued for the design of network update algorithms that provide even stronger consistency guarantees.

To the best of our knowledge, the only paper explicitly addressing the consistent network update problem from a distributed controller perspective is STN [3]: STN provides a transactional interface offering all-or-nothing semantics and serializability (the "holy grail" of safety properties), allowing controllers to install policies in a conflict-free manner; if some controllers fail, other controllers can take over. STN [3] is based on atomic read-modify-write primitives, but while this primitive is postulated, no implementation is described. In this paper, we provide this missing link, and also show that other useful and much more powerful synchronization primitives can be implemented.

Our work is also closely related to the recent work on providing more high-level abstractions and programming languages for SDNs, such as Frenetic [8], also enabling policy composition [21].

In terms of distributed computing: TODO PETR: maybe modify a bit the text below and also write more about multi-writer consensus etc.?

Distributed Computing. There is a long tradition of defining correctness of a concurrent system via an equivalence to a sequential one [15, 17, 23]. The notion of sequentially composable histories is reminiscent of linearizability [15], where a history of operations concurrently applied by a collection of processes is equivalent to a history in which the operations are in a sequential order, respecting their real-time precedence. In contrast, our sequentially composable histories impose requirements not only on high-level invocations and responses, but also on the way the traffic is processed. We require that the committed policies constitute a conflict-free sequential history, but, additionally, we expect that each *path* witnesses only a prefix of this history, consisting of all requests that were committed before the path was initiated. The transactional interface exported by the CPC abstraction is inspired by the work on speculative concurrency control using software transactional memory (STM) [28]. Our interface is however intended to model realistic network management operations, which makes it simpler than recent dynamic STM models [14].

Extending the interface to dynamic policies that adapt their behavior based on the current network state sounds like a promising research direction. On the other hand, our criterion of sequential composability is more complex than traditional STM correctness properties in that it imposes restrictions not only on the high-level interface exported to the control plane, but also on the paths taken by the data-plane packets. Also, we assumed that controllers are subject to failures, which is usually not assumed by STM implementations.

9. CONCLUSION

This paper showed that powerful synchronization primitives from distributed computing can be implemented in Openflow. Our work complements existing research on the design of distributed control planes, and also provides some missing links, e.g., for the transactional approach taken by STN. We also hope that our work can nourish the ongoing discussion of what additional in-band features may be useful for future versions of Openflow.

10. REFERENCES

[1] Y. Afek, M. Merritt, and G. Taubenfeld. The power of multi-objects (extended abstract). In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 213–222, 1996.

[2] 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.

[3] 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.

[4] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust sdn control plane for transactional network updates. In *Proc. 34th IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[5] 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.

[6] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an Elastic Distributed SDN Controller. In *HotSDN*, 2013.

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

[8] 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.

[9] S. Ghorbani and B. Godfrey. Towards correct network virtualization. In *Proc. ACM HotSDN*, pages 109–114, 2014.

[10] S. Ghorbani and B. Godfrey. Towards Correct Network Virtualization. In *HotSDN*, 2014.

[11] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

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

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

[14] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software Transactional Memory for Dynamic-sized Data Structures. In *PODC*, 2003.

[15] M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[16] 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.

[17] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9), 1979.

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

[19] 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.

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

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

[22] OpenFlow Website. Specifications. In *http://www.openflow.org/*, 2013.

[23] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26, 1979.

[24] K. Phemius, M. Bouet, and J. Leguay. Disco: Distributed multi-domain sdn controllers. In *Proc. IEEE Network Operations and Management Symposium (NOMS)*, 2014.

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

[26] S. Schmid and J. Suomela. Exploiting locality in distributed sdn control. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, August 2013.

[27] S. Schmid and J. Suomela. Exploiting Locality in Distributed SDN Control. In *HotSDN*, 2013.

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