OpenFlow Needs You! A Call for a Discussion About a Cleaner OpenFlow API

Peter Perešíni
EPFL

Maciej Kuźniar
EPFL

Dejan Kostić
Institute IMDEA Networks

***Abstract*—Software defined networks are poised to dramatically simplify deployment and management of networks. OpenFlow, in particular, is becoming popular and starts being deployed. While the definition of the "northbound" API that can be used by the new services to interact with an OpenFlow controller is receiving considerable attention, the traditional, "southbound", API that is used to program OpenFlow switches is far from perfect. In this paper, we analyze the current OpenFlow API and its usage in several controllers and show semantic differences between the intended and actual use. Thus, we argue for making the OpenFlow API clean and simple. In particular, we propose to mimic the process that exists in the Python community for deriving changes that result in a preferably only one, obvious way of performing a task. Toward this end, we propose three OpenFlow Enhancement Proposals: i) providing positive acknowledgment, ii) informing the controller about "silent" modifications, and iii) providing a partial order synchronization primitive.**

I. Introduction

Software defined networks are poised to dramatically simplify deployment and management of networks. OpenFlow, in particular, is quickly becoming popular and starts being deployed. One of the reasons for OpenFlow's success is the separation of control and data planes. The developers write network logic that resides in the controller, which in turn, installs rules in the OpenFlow-enabled switches. Another reasons for the adoption of this protocol is the relative simplicity of the OpenFlow API, which effectively gives programmatic access to the flow tables in the switches.

Programming the network means managing all switches. Recent work shows the difficulty in getting this ensemble of switches to perform correctly [6]. For example, the network forms an asynchronous system in which messages can be lost or delayed, which can lead to violations of correctness properties. Thus, writing correct controller logic is a difficult task.

Once network operators started considering deploying OpenFlow, they started to formulate a higher-level API, the so-called northbound API that enables the operator's service (application logic) to leverage the OpenFlow's ease of deploying new services. The OpenFlow API itself is now referred to as the southbound API. The definition of the northbound API has recently received considerable attention. However, in this paper we argue that the southbound API is far from perfect and that additional work on it is needed to make it easier to develop correct controller programs.

The southbound API itself is part of the OpenFlow specification. With the rapid advancement of the specification (three revisions within just over a year), one would think that seemingly simple task of having a way to install rules in the switches is now complete. Unfortunately, that is not the case. We analyze the current OpenFlow API and its usage in several controllers. Our analysis of the API use reveals semantic differences between the intended and actual use. In particular, we: i) identify issues related to OpenFlow `Barrier` command, ii) discuss the impact of missing responses from switches after receiving control messages, and iii) discuss silent modifications and current changes.

Existing work also shows that there exists room for different interpretation of the specification, which leads to inconsistent behavior of switches in response to the control messages from the controller [9], [10]. Moreover, a large number of features was added to the specification (currently around hundred pages long), which diminishes the chances of getting each of the features implemented correctly.

In this paper, we argue for the importance of the simple and clean OpenFlow API. Toward this end, we propose adjustments to avoid using software "anti-patterns". It is crucial to do make these modifications to the API, because OpenFlow is in early stages of development and many changes and improvements are still possible. If we let developers get used to the anti-patterns and switch vendors to build switches accordingly, we will be stuck with these for the foreseeable future. One only needs to look at the lamentation at the Internet ossification to see what happens when it becomes difficult to change a set of protocols.

We propose to mimic the process that exists elsewhere, for example in the Python community. There, the developers propose new features and cleanups in PEPs - Python Enhancement Proposals. The proposals are accepted after they are carefully evaluated if they are a) useful, and b) consistent with the rest of the language, and c) there is only one obvious way to do it. In particular, we propose three *OpenFlow Enhancement Proposals* (OFEPs): i) providing positive acknowledgment, ii) informing the controller about "silent" modifications, and iii) providing a partial order synchronization primitive.

The remainder of the paper is organized as follows. We analyze the use of the OpenFlow API in Section II, discuss community-involved improvements to it in III. Section IV contains the related work, and we conclude in Section V.

II. Motivation – OpenFlow API use analysis

We motivate our paper by observing that some OpenFlow API uses we came across do not match use cases envisioned by the API designers. In this section we look at some of these (mis)uses and discuss why these are real problems that should be addressed at the API level. Note that some of the problems that we discuss overlap.

A. Case I - `Barrier` as OpenFlow Swiss knife

We start with probably the most pronounced example of an overused command, and that is the `Barrier` command. According to the OpenFlow protocol, a switch can process commands in any order. However, when a switch receives a `Barrier` command, it needs to finish processing all previously received commands before moving to messages after the `Barrier`. Moreover, when finished executing all operations before the `Barrier`, the switch sends a `BarrierReply` message to inform the controller. We observe that developers use the `Barrier` command as a way to achieve different and unrelated goals. In software anti-pattern terminology, such behavior would be called "god object" as it concentrates too many functions into a single part of the design. The right solution is to separate these use cases into different API calls.

`Barrier` to enforce dependencies and order events The most common usage of barrier is to order the execution of messages. The envisioned use case is described directly in the OpenFlow specification [4]:

> "If two messages from the controller depend on each other (e.g. a flow mod add with a following packet-out to OFPP_TABLE), they must be separated by a barrier message."

Another real use can be found in the POX controller [5]: After the connection between the switch and the controller is established, POX resets the switch to a clean state by clearing all switch tables. To avoid accidental mixing of this delete message with further controller actions (such as a proactive controller module installing rules immediately after receiving `SwitchUp` event), the controller issues the `Barrier` command to enforce the message ordering and make sure the switch clears its flow tables first.

`Barrier` as the only way to verify success In the current OpenFlow API, there is no easy way to test for the absence of errors as the controller does not receive success notifications for some commands. As an example, a controller trying to check whether a particular `FlowMod` command (typically used to install new forwarding rules) was successful must wait for some time and listen for errors. If it receives no error message associated with the particular request, the controller may assume that the command succeeded. As a particular use case, consider a controller issuing multiple `FlowMod` commands during a big network reconfiguration task, or after the controller startup. The controller may want to monitor the progress of rule installations and the only way to achieve this goal is to issue barrier commands. Depending on the granularity at which the barriers are sent (*i.e.*, after each `FlowMod` or just after all `FlowMod`s), the controller either risks performance problems due to serialization, or it cannot accurately determine the progress. Further, in the second case the controller verifies if the previous commands are successful less frequently. This means that the switch receives larger batches of commands at once, and may keep installing unnecessary rules despite early errors that may change the installation policy.

Controller synchronization Traditionally, in the concurrent programming, barriers are used to synchronize several threads before they can continue further. We observe that OpenFlow programmers can use barriers in a similar way. The first example of such use is in the *l2_multi* POX controller where the controller waits for the flow installations on all switches on the end-to-end path. Only then it sends the `PacketOut` message (instructing the switch to forward the packet) for a packet that caused the creation of the aforementioned path. Similarly, consistent network updates [15] require the controller to wait until the first phase of two-phase commit ends, and only then the second phase can start. The next usage example is similar to checking if a command is successful, albeit with different goal. In this example, the controller reconfigures a spanning tree by sending `PortMod` messages. However, the controller needs to wait before flooding any packets until all switches confirm their new configuration. Otherwise, the packets may enter transient forwarding loops which will hurt the performance of all other flows.

Are these three uses bad? We start by arguing that the second case is an apparent API misuse as it forces developers to reimplement the "success verification in a clumsy, inefficient and error-prone way. This not only leads to programs which are hard-to-understand, but also wastes switch resources forcing them to process an additional command.

While the first use is very similar to the intended use case, it contains one important distinction: The programmers usually need to enforce only *partial* order – they need to ensure order of several related messages but it is not necessary to ensure ordering between unrelated messages. As an example, consider two concurrent flows, each issuing `FlowMod + Barrier + PacketOut` as in the OpenFlow specification. By inserting barrier between each `FlowMod` and `PacketOut`, we force the switch to serialize all these `FlowMod`s. This disables the switch-specific rule installation optimizations (*e.g.*, batching of TCAM updates, *etc.*). Moreover, the solution incurs an additional penalty for the second `FlowMod`/`PacketOut` pair as it needs to wait till the first command is finished and acknowledged by the `Barrier`. Instead, OpenFlow API should enable programmers to specify real dependencies and enable only

partial ordering of commands.

Finally, we see that in the third use, programmers use barriers as a notification service. This again hurts the performance of the system because of the unnecessary serialization.

B. Case II - Missing response from the switch

As already pointed out in the previous section, OpenFlow relies only on the negative acknowledgment (*e.g.*, OpenFlow switch does not have a way to inform the controller about success of some events) which we believe is insufficient. One particular example can be found in the POX spanning tree module. When the spanning tree needs to query the switch for the current port configuration, it sends `FeaturesRequest` to the switch because the OpenFlow protocol does not provide any direct command to query the port configuration. While `FeaturesRequest` does the job and returns the port configuration, it does that at a cost of sending all port configurations in the same message. Moreover, the `FeaturesRequest` message is commonly used only during the switch connection setup and therefore it may be a point of further interoperability issues (*e.g.*, vendor not implementing `FeaturesRequest` in the middle of the connection).

Similarly to the spanning tree module in POX, controller platforms which return an answer to the query via northbound API must know if the configuration change was applied correctly before returning the answer. Finally, we conclude this section by noting that absence of negative acknowledgment is not equivalent to receiving a positive acknowledgment. Authors of SOFT [10] show that sometimes switch implementations lack correct error reporting or silently change requests as discussed in the next section.

C. Case III - Silent modifications and concurrent changes

One of the frequently discussed issues in the NOX/POX mailing list archives concerns OpenFlow 1.0 switches. Because of the lack of precise specification, some OpenFlow switch implementations choose to accept and sanitize ambiguous requests. As such, `FlowMod` request installing rule "match(ip_src=10.0.0.1)" without including "match(ethertype=0x800 (IP protocol))" would be sanitized to match all packets regardless of their ethertype or ip addresses. While this issue was clarified in the subsequent versions of the protocol, we believe that silent modifications might be a real threat to any OpenFlow deployment and not all cases can be properly caught by the specification. Here we showcase popular software vs. hardware rule problem.

Vendors currently use their internal knowledge of hardware limitations to place the rules in software or in hardware. This decision is often not very well documented and developers might spend a lot of time decyphering it from the experience [13]. As a point of reference, we look at switch from one of the vendors – the switch does not support `priority` field in the hardware and the specification is not dictating any particular behavior in this case (*e.g.*, should the switch allow only single value of `priority` field?). Thus, instead of failing all `FlowMod` requests with different priorities, the switch silently ignores this field.

Moreover, silent modifications are not the only concern. In particular, while an OpenFlow controller is often presented as a sole component in control of the switches, there still may be concurrent access issues. One particular example can be found in the documentation inside the POX spanning tree module:

> "When the spanning tree adjusts the port flags, the port config bits we keep in the Connection become out of date. We don't want to just set them locally because an in-flight port status message could overwrite them. We also might not want to assume they get set the way we want them." [11]

Clearly, the authors of the module code were reluctant to update the port configuration variables directly in the controller because of fear of concurrent updates. In particular, such updates can stem from i) other controller modules; ii) network operators managing switches via console; iii) other controllers in the multi-controller scenario; iv) simply port state changes. In the first case, changing the configuration variables and sending configuration requests by several concurrent modules may cause race conditions, especially when some of the configuration requests will fail with an error and the modules would not be able to clean up the controller state consistently. In the second and third cases, a blind faith that the controller configuration did not change will cause similar race conditions with controller state diverging from the switch state.

III. Improving the OpenFlow API while involving the community

The point we made so far is that the OpenFlow API does not always provide the right tools to perform given tasks. Inspired by Python's success, we believe that the right way to fix the problems is to engage in a deep and transparent discussion within the community. In this paper, we argue that adapting some of Python's design principles can guide API design toward a cleaner and simpler API. Specifically, we will mention two important basic principles:

One obvious way to do it One of the design principles behind Python programming language is very well captured in the following citation from The Zen of Python [14]:

> "There should be one – and preferably only one – obvious way to do it."

In essence, we should have *the right tools for the right job*. While easy to write, this principle is hard to realize in practice and certainly cannot be realized in a single version of the specification. Instead, the API should evolve concurrently with the evolution of its use cases as is the case with the `Barrier` command.

The Principle of Least Astonishment In programming, surprises lead to unexpected bugs which are hard to debug. Developers follow their intuition when using an API so they tend to be confident that such use is correct and look for the errors elsewhere. The API therefore should be consistent, easy to understand and with no corner cases or race conditions possible. Current OpenFlow API violates this principle, for example, if the port configuration changes concurrently to the controller sending `PortMod` command.

Inspired by Python community PEPs (Python Enhancement Proposals) [17], we recommend publishing *OpenFlow Enhancement Proposals* (OFEPs) which will describe all important aspects of the issue starting from the use cases and impact analysis, and ending at describing possible implementations and implications for ASIC vendors. After such proposals are made, the community should critically judge and discuss their usability and overall aspect on the clean and simple API design. After the discussion concludes with clear, intuitive and practically feasible solution, the OpenFlow API should be modified accordingly in the next version. To provide some examples, we created the first few OFEPs.

A. OFEP-0001 - Positive acknowledgment (Report not only errors but also successes)

Abstract This OFEP proposes that the switch should confirm all completed commands, even if there was no error involved.

Motivation Programmers tend to misuse the `Barrier` command to check whether the switch successfully finished processing of `FlowMod` commands. To amend this issue, we propose that OpenFlow protocol specifies that the switch should always report success/error information about every command it processes.

Implementation proposal We propose to add new `OFP_ACTION_COMPLETED` OpenFlow message holding the information (*i.e.*, `xid`, completion time, *etc.*.) about the completed request. An alternative to introducing new message is to introduce a new error code `E_NO_ERROR`.

Pros/Cons While making the API symmetric can greatly help controller programs with the main focus on consistency, the change may negatively affect high-performance controllers because of the additional controller-to-switch bandwidth and processing these responses incur. Here, we propose two possible improvements:

1) Make success notifications optional, for example by adding a single-bit flag to the OpenFlow header indicating if the sender is interested in receiving any success response; and
2) Piggyback notifications on top of other messages (*e.g.*, `PacketIn`).

B. OFEP-0002 - Inform about "silent" modifications

Abstract This OFEP proposes to add detailed feedback about the action that the switch performed. In particular, if the switch applied modifications to the request, the controller should be able to learn about them.

Motivation Many current switch implementations sanitize requests and may silently change or ignore parts of the request. Although the specification is getting better at finding and eliminating these cases, the switch vendors may lack the full implementation of the specification, or may simply fail to implement it correctly. Consequently, determining what the switch really does is hard to debug.

Implementation We propose to extend OFEP-0001 to not only provide positive acknowledgment, but to include exact modifications performed to the request. As an example, if the request is `FlowMod`, the reply will include the sanitized match field (*i.e.*change fields ignored by the hardware to wildcards). Additionally, we argue that the `FlowMod` response should contain new `is_in_hardware` field indicating whether the switch installed the flow in a hardware table or the flow is processed in the software.

Pros/Cons Including the response can greatly help with debugging of interoperability issues as the controller will learn about particular switch nuances. The response, however, creates an additional load on the control channel. We advocate to use the same mechanism as in OFEP-0001 to indicate whether the switch should send positive acknowledgment.

C. OFEP-0003 - Provide partial order synchronization primitive

Abstract This OFEP proposes to add the possibility of specifying dependencies between different controller messages.

Motivation `Barrier` synchronization is important primitive in OpenFlow. Unfortunately, applications that do not require total order of commands still need to use `Barrier` if there is any dependency between messages they issue. Doing so hurts the performance.

Implementation proposals There are the proposed OpenFlow API changes:

1) An application should track positive responses (as per OFEP 0001) and issue command after receiving confirmation. This proposal does not require changing the wire protocol but incurs additional delay because the messages with the dependency cannot be batched together.
2) Add the `dependency` field to OpenFlow header. The switch will be required to postpone processing of the message until the dependency (identified by an `xid` – message identifier) is processed. The exact behavior needs to be decided. For example, what happens if there is no message with such `xid` and/or for how long the switch needs to retain the `xid` information.

3) Add new `BatchRequest` message which encapsulates a set of OpenFlow messages. The batch is executed sequentially until the first error.

Pros/Cons While the change will help to improve the performance of controllers, its implementation on the switch side is nontrivial. As a workaround, vendors may decide to transparently fall back to sequential processing of batches or treat dependencies as barriers.

D. OFEP-0004 - The controller should be able to learn about configuration changes

Abstract This OFEP proposes to add a new mechanism for the controller to learn about the configuration updates.

Motivation The switch configuration may be changed by itself, other (*e.g.*master) controller or simply via the operator console. Subsequently, if the controller is not informed about these changes, it may diverge from the reality. Thus, according to the Principle of Least Astonishment, the controller should be informed about the configuration changes.

Implementation We propose to add new OpenFlow `OFPT_*_CHANGED` message complementing each of the following messages: `OFPT_SET_CONFIG`, `OFPT_PORT_MOD`, `OFPT_TABLE_MOD`, `OFPT_METER_MOD`. The switch will be required to send the message each time the configuration changes (regardless of who initiated the change). Note that it is possible to extend this proposal by making the reply for the controller-initiated changes optional as in OFEP-0001.

E. OFEP-0005 - Explicit configuration check

Abstract This OFEP proposes to add a mechanism that detects and avoids concurrent modifications of the switch configuration.

Motivation While switch reconfiguration by the controller might be an infrequent process, there is a high-impact risk of inconsistencies if the configuration is changed concurrently i) by other controllers (in multi-controller mode); ii) through the switch operator console; or iii) by two different controller modules.

Implementation proposals We propose to check whether the configuration as known by the controller matches the expected configuration. We provide two possible alternatives:

1) Expand each configuration message with fields holding expected values of the old configuration. The controller then fills the current configuration values (the expected state of the switch) and the switch will issue a new error `E_STALE_CONFIG` if the values do not match.
2) Expand the configuration message with a *version number*. The switch increments the version number each time the configuration changes. Upon receiving a configuration request from the controller, the switch checks the included version number and if the numbers does not match, the switch reports an error. Otherwise, the switch updates its state and increments the version number.

Pros/cons The controller's view of the switch configuration would not go out of sync. This, however, adds additional burden for the controller to maintain the switch state even if the controller is not interested in it. In the extreme case, while configuration changes should be rare events, this proposal will lead to deadlocks if the rate of the configuration change is faster than the controller response time. If this is a concern, additional discussion about the solution will be needed.

F. OFEP-0006 - Check overlap by default

Abstract This OFEP proposes to modify the default behavior of the rule overlap checking.

Motivation Controller bugs may lead to overlapping rules that will go unnoticed, especially because the checking for overlaps is not enabled by default. According to the Principle of the Least Astonishment, this should not be the case.

Implementation Replace the `CHECK_OVERLAP` flag with the `IGNORE_OVERLAP` flag.

IV. Related work

Perhaps the most related work to the effort in this paper is the standardization process maintained by Open Networking Foundation (ONF). What ONF lacks, though, is a real openness:

> "The working and discussion groups are restricted to ONF member company employees." [1]

Moreover, while the specification provides "Release notes" describing all changes between protocol versions, the specification *lacks an explanation* why these changes were proposed and *what purpose* do they serve. In particular, the specification does not (and should not) contain all rejected modification proposals. Instead, we propose a truly open and transparent solution where the proposals are discussed in public, and where the decisions are backed up by a rationale why the proposal was accepted/rejected. A venue that serves a similar purpose as our OFEPs are OpenFlow discussion wiki pages [2], [3]. They form a centralized repository of proposals regarding the future OpenFlow specification versions as well as short explanations what happened to older suggestions. However, they lack frequent updates and do not leave space for open discussion.

Our work is certainly not the first attempt to improve the OpenFlow protocol API. There are proposals for extending OpenFlow protocol introducing new functionality like circuit switching [7], [16]. Others [8] suggest to extend the protocol by providing hooks for easier debugging. Mogul *et al.* [12] consider the issues in implementing an OpenFlow agent that stem from limited TCAM space and rule installation speed. Our main goal in this work is to concentrate on making the protocol itself more intuitive and to bootstrap creation of a common platform where other proposals for extensions

(circuit switching, debugging, etc.) can be easily discussed and evaluated by the community.

V. Conclusions

The main goal of this short paper is to emphasize the *importance of getting the OpenFlow (Southbound) API right* – it should be simple and clear. In particular, we believe that the API should adhere to clean design principles as is the *Principle of the Least Astonishment* (*e.g.*, no nasty surprises and special cases) and the principle of *having the right tools for the job* (*e.g.*, allow controllers to be informed about the command success instead of inferring it through some obscure workaround). As such, these principles are not enough.

What OpenFlow needs is a lot of discussion in the community (and not restricted only to ONF members) about the potential uses and problems with the API, as well as a transparent process of integrating this knowledge into the API. Inspired by Python community and its way of working with the language, we propose to create a centralized place for the discussion in a format we tentatively call OFEPs – *OpenFlow Enhancement Proposals*.

To start the discussion, we provide here the first few OFEPs and the rationale behind them. We hope that the community will notice these proposals, create one centralized place to store and discuss them, and continue improving and simplifying the already complex specification – let us jointly create an elegant API for SDN and not turn OpenFlow into a spaghetti mix of RFCs as we used to do in the past.

Acknowledgments

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110.

References

[1] ONF Discussion Groups. https://www.opennetworking.org/working-groups/discussion-groups.

[2] OpenFlow 1.2 Proposal. http://www.openflow.org/wk/index.php/OpenFlow_1_2_proposal.

[3] OpenFlow 1.X Discussion. http://www.openflow.org/wk/index.php/Openflow_1.X_Discussion.

[4] OpenFlow Switch Specification. https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.3.0.pdf.

[5] POX Controller. http://noxrepo.org.

[6] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.

[7] S. Das. Extensions to openflow protocol in support circuit switching, 2010. http://www.openflow.org/wk/images/8/81/OpenFlow_Circuit_Switch_Specification_v0.3.pdf.

[8] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the Debugger for my Software-Defined Network? In *HotSDN*, 2012.

[9] M. Kuźniar, M. Canini, and D. Kostić. OFTEN Testing OpenFlow Networks. In *EWSDN*, 2012.

[10] M. Kuźniar, P. Perešíni, M. Canini, D. Venzano, and D. Kostić. A SOFT Way for OpenFlow Switch Interoperability Testing. In *CoNEXT*, 2012.

[11] J. McCauley. POX Spanning Tree Module. https://github.com/noxrepo/pox/blob/betta/pox/openflow/spanning_tree.py.

[12] J. C. Mogul, P. Yalag, J. Tourrilhes, R. Mcgeer, S. Banerjee, T. Connors, and P. Sharma. Api design challenges for open router platforms on proprietary hardware. In *HotNets*, 2008.

[13] B. Owens. OpenFlow Switching Performance: Not All TCAM Is Created Equal. http://packetpushers.net/openflow-switching-performance-not-all-tcam-is-created-equal/.

[14] T. Peters. The Zen of Python . http://www.python.org/dev/peps/pep-0020/.

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

[16] M. Shirazipour, Y. Zhang, N. Beheshti, G. Lefebvre, and M. Tatipamula. OpenFlow and Multi-layer Extensions: Overview and Next Steps. In *EWSDN*, 2012.

[17] B. Warsaw, J. Hylton, D. Goodger, and N. Coghlan. PEP Purpose and Guidelines . http://www.python.org/dev/peps/pep-0001/.