Providing Reliable FIB Update Acknowledgments in SDN

Maciej Kuźniar
EPFL
maciej.kuzniar@epfl.ch

Peter Perešíni
EPFL
peter.peresini@epfl.ch

Dejan Kostić
KTH Royal Institute of Technology
dmk@kth.se

ABSTRACT

In this paper, we first show that transient, but grave problems such as violations of security policies can occur with real switches even when using consistent updates to Software Defined Networks. Next, we present techniques that are effective in ameliorating this problem. Our key insight is in creating a transparent layer that relies on control and data plane measurements to confirm rule updates only when the rule is visible in the data plane.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

Keywords

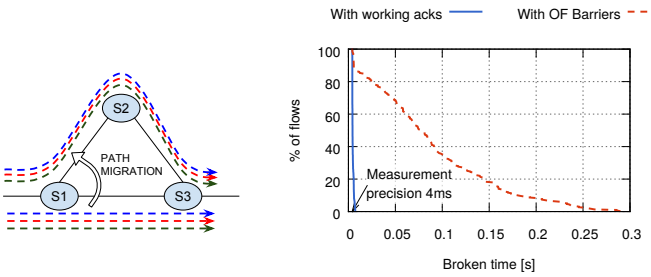
Software-Defined Networking; Reliability; OpenFlow

1. INTRODUCTION

Software Defined Networks (SDNs) enable flexible network configuration by allowing a centralized controller to manipulate forwarding rules in switch flow tables. The controller reacts to events such as topology changes, traffic engineering decisions, and failures by computing and installing the new desired network state. This process is referred to as the *network update*, and involves issuing commands to the switches to install the new set of rules and reconfigure the network. The network update process is complicated and if not conducted carefully, may lead to transient problems such as black holes, forwarding loops, link overload, and packets reaching undesired destinations. There are many approaches that guarantee some correctness properties [3,8,9,11]. These all split an update into many stages, and rely on knowing when a particular rule modification was applied at the switch(es) before issuing further modifications. This necessitates positive acknowledgments confirming rule modifications.

CoNEXT'14, December 2–5, 2014, Sydney, Australia.

http://dx.doi.org/10.1145/2674005.2675006 .



(a) (b)

Figure 1: Consistent network update using a hardware switch. Despite theoretical guarantees, for most flows switch S1 gets updated before S2 and the network drops packets for up to 290 ms. Using our system eliminates this problem.

Unfortunately, in OpenFlow, currently the most popular SDN protocol, there is no mechanism with a sole purpose of acknowledging rule modifications. Instead, there exists a Barrier command with a more general functionality.[1] However, studies [12] show that OpenFlow switches do not satisfy the specification in this crucial for correctness aspect. We further verify that this happens even to the latest-generation switches [7]. This may lead to transient but grave network problems.

To demonstrate the magnitude of the problem, we prepare a small end-to-end test: We set up a network in a triangle topology with the hardware switch S2 and two software switches S1 and S3 (Figure 1a). We preinstall paths for 300 IP flows between hosts H1 and H2 going through switches S1 and S3. Then, we perform an update that modifies the paths to S1-S2-S3 in a consistent manner, such that a given packet can follow the old rules only or the new rules only [11].

Despite using consistent updates, some flows drop packets for an extended period of time (Figure 1b). A detailed analysis shows that the switch sends the barrier reply up to 290 ms before the rule modification becomes visible to data plane traffic. Other switch models not only reply to barriers too early, but also reorder rule updates across barriers [7].

The consequences of this observation have great impact –even though provably correct in theory, *none of the consistent network update techniques work in practice with buggy*

[1]According to the specification, after receiving a barrier request, the switch has to finish processing all previously-received messages before executing any messages after the barrier request. When the processing is complete, the switch must send a barrier reply message [1].

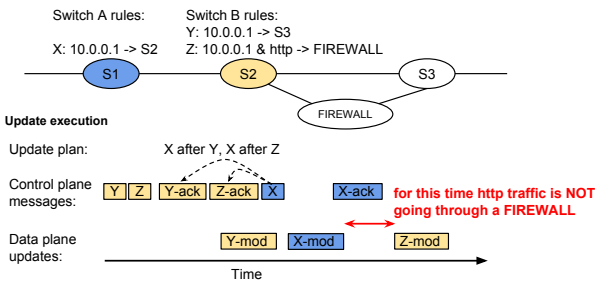


Figure 2: If switch B does not report data plane updates correctly, the theoretically safe update that adds rules for trusted and untrusted traffic from the same host turns into a transient security hole.

switches, and all systems that build upon these techniques are unsafe as well. In particular, buggy behavior may lead to security violations, broken bandwidth guarantees, or black holes – an example of the first is depicted in Figure 2. If the issues that we bring up here are not addressed (*e.g.*, by adopting one of our schemes) the SDN deployments that are increasingly taking place in enterprises are in jeopardy.

While an incorrect barrier implementation may be just a temporary problem and not a fundamental limitation (one of the tested switches does implement barriers correctly), we see three main reasons why it should be immediately addressed. First, there are many solutions that rely on barriers and it cannot be expected that all switches in a network will correctly function. Second, even after five major revisions the OpenFlow specification is unclear – it does not explicitly state that the commands must be applied in the data plane, instead it may be understood that the barrier enforces control plane-ordering only. This in turn means that, unless there is a high pressure from customers, vendors will have no incentive to provide data plane-level confirmations, and therefore the problem might not disappear in future switch generations. Finally, we argue that controllers need acknowledgments of each rule installation, rather than only high level barriers [10]. Therefore, we go one step further than just fixing barriers, and the solutions we propose provide such fine grained rule update acknowledgments. Providing this functionality entails a few challenges: i) handling heterogeneous switches, ii) dealing with the variable delay in installing rules in the data plane, iii) ensuring low overhead.

In this work, we introduce a transparent layer below an SDN controller that provides reliable acknowledgments for rule modifications. In particular, when using our scheme, the controller can never receive an acknowledgment before a corresponding rule is installed in the data plane. Our contributions include proposing various methods, including data plane probing schemes, that achieve the aforementioned goal depending on switch capabilities. The effect of applying one of these techniques is visible in Figure 1b: no packets get dropped. Moreover, we explain how, at the cost of a higher overhead, such a layer provides barrier-like guarantees to the controller working with switches that do not implement barriers correctly.

2. SYSTEM OVERVIEW

We have two main requirements in mind when designing our system called RUM (Rule Update Monitoring). First, it needs to work with existing switches and take into account their capabilities and limitations. Second, the system should provide reliable barrier commands in a backward-compatible way without requiring any modifications to the existing controllers and switches. However, it should allow the RUM-aware controllers to benefit from fine grained acknowledgments.

Acknowledging rule modifications. The first goal of the system is to provide reliable rule modification acknowledgments to the OpenFlow-speaking controllers. We design RUM as a transparent layer between the switches and the controller that intercepts and modifies the communication between them similarly to FlowVisor [13] or VeriFlow [5]. In contrast with these systems, RUM plays a more active role in the interception, as it can buffer, rate-limit, remove or add messages. To allow easy deployment and transparency for controllers that are not designed to work with the fine grained acknowledgments, RUM adapts existing OpenFlow messages to convey successful modifications (such notifications are not available in OpenFlow). Depending on the required precision and available switch properties, the techniques (Section 3) rely only on the control plane communication with the tested switch, or may install additional rules and involve the neighboring switches.

Providing reliable barriers. To provide reliable barriers, RUM intercepts all barrier requests and replies. After capturing a barrier request, RUM holds off sending the corresponding barrier reply and following messages from the switch until it can ensure that the switch completed all pending operations. An ability to correctly acknowledge commands issued to the switch is therefore the key to reliable barriers. Additionally, when working with switches that reorder modifications across barriers, RUM buffers all commands that the controller sends after the last unconfirmed barrier. It releases them to the switch after acknowledging the barrier. The barrier layer uses standard OpenFlow barrier commands and is therefore transparent to any controller.

3. DATA PLANE ACKNOWLEDGMENTS

RUM aims to acknowledge rule modifications as soon as the new rule is active in a switch data plane, but not sooner. Because different switches have different limitations and capabilities, we discuss several possible solutions to the problem at hand.

3.1 Control-plane only techniques

The first class of techniques uses control plane information only and requires modeling the switch behavior.

Using OpenFlow barrier commands. Relying on barrier messages is a natural way to receive acknowledgments in OpenFlow, therefore, we present it as a baseline. A switch must send a barrier reply message only after it finishes processing all previous commands. However, our measurements show that some switches respond to a barrier immediately, before the modifications were applied to the data plane, and as a result the data plane is often between 100 and 300 ms behind what may be assumed based on barrier replies [7]. This confirms previous measurements [12] indicating that barriers cannot be trusted and should not be used as rule update confirmations.

While one can imagine using other OpenFlow commands instead of barriers (*e.g.*, using statistics requests), we believe such an approach does not solve the underlying problem —

the reply from the switch is still based on its control plane view and/or it does not have enough temporal granularity (*e.g.*, flow statistics might be updated only once per second). Therefore, in the rest of this section we introduce techniques that still rely on the barriers, but take into account the data plane delay.

Delaying barrier acknowledgments. The first technique relies on experiments prior to deployment. If the maximum time between the barrier reply and the rule modification being applied is bounded and can be measured, RUM waits for this time after receiving a reply before confirming earlier modifications.

The main drawback of this method is that it requires precise delay measurements or overestimation. We observe that in practice the delay depends on many, often difficult to predict factors and therefore providing strong guarantees is difficult [7]. For example, if the data plane is typically delayed by up to 100 ms, but there are cases of a 300-ms delay, one needs to always wait for 300 ms. Even then, in hard to predict corner cases, the delay may reach several seconds, which is impractical to use as the upper bound. Therefore, waiting for a timeout after each barrier has a negative impact on update performance and rule modification rate.

Adaptive delay. Adaptive timeout improves the performance of the previous technique, but requires even more detailed measurements to develop a precise switch model. Based on such models and knowing the rate at which a controller issues modification commands, RUM estimates when a particular rule modification will take place in the switch. Thus, the timeout is adjusted accordingly. However, this method requires building detailed switch performance models, which is difficult [7].

3.2 Data plane probes

The basic idea of data plane probes is to inject special packets into the network and use these packets, as well as special probing rules, to monitor which rules are active in the data plane. There are two aspects of OpenFlow barrier commands: (*i*) a switch should respond with a barrier reply after it processed all previous commands, and (*ii*) a switch should never reorder commands separated by barriers. In practice some switches violate either the first of these properties (because they process commands in the control plane, but push the rules to the data plane later), or both. The two techniques presented in this section are design to work correctly with such two classes of switches.

3.2.1 Sequential probing

If a switch violates only the first barrier property (responds to barriers too early) two modifications separated by a barrier are never reordered in the data plane. Therefore, a strawman solution follows each real rule modification with a barrier and an additional rule installation for probing. By the time the probing rule is determined to be active (*i.e.*, it forwards a probing packet), the original rule must be in place as well.

Implementation-wise, the probing rule matches only the specially selected probe packets and has a high priority so that no other rule can override it. The probing rule sends the matching packets to the controller. RUM then repetitively injects probe packets (using a PacketOut message) into the switch forwarding pipeline and when the probe arrives back to RUM, it means that the probe rule is installed and therefore the corresponding real rule is active as well. Finally, after probing rule is confirmed, it is no longer needed and can be removed.

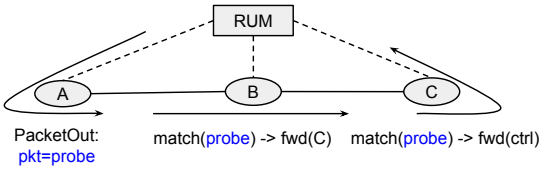


Figure 3: Probing the data plane at switch B. The controller (RUM) sends a probe packet from switch A to switch B. If B installed the probing rule, it forwards the packet to switch C which sends it back to the controller.

There are, however, technical details of the strawman solution that make it impractical and require improvements. First, from the correctness perspective, it assumes that the PacketOut processing and probe rule matching are performed in hardware. Unfortunately, this might not be the case – rules sending packets to the controller are often kept in software and may start forwarding traffic before the previous hardware rules are pushed into the data plane. As such, we modify our solution to use hardware-only probing rules – we use two additional switches[2] as depicted in Figure 3.

Second, inserting one probe rule after each normal rule is prohibitively expensive. Instead, we notice that a single probe rule installed after a batch of several rule modifications acknowledges the whole batch at the same time. This way the probing overhead gets amortized over more rules at the expense of a longer acknowledgment delay. Moreover, the probe rules can be optimized even more – instead of installing a new probe rule for each batch and then deleting it, we use a single probing rule which rewrites a particular field in the packet header (*e.g.*, ToS or VLAN) with a version number of this probing rule. Then, we just update the rule to write the new version number to the probe packet header. RUM recognizes the last version of probe rule based on the probing packet headers it receives back.

Multi-switch deployment. The approach described so far requires setting up different probe rules matching different packets for each probed switch, because otherwise forwarding the probe packet on the next switch will interfere with probe collection on that switch. We overcome this problem by choosing two header fields H_1 and H_2 to be used by probing. These can be any rewritable fields in a packet header. Additionally, we reserve two special values of H_1; we call these values *preprobe* and *postprobe*. In our solution, all switches install a high priority probe-catch rule that sends all packets with $H_1 == postprobe$ to the controller. We also install one probing rule per each switch. It matches packets with $H_1 == preprobe$ and rewriting them to post-probes while also storing the per-switch unique probe rule version in H_2 ($H_1 \leftarrow postprobe$, $H_2 \leftarrow ver$). To do the probing, RUM sends a probe packet with $H_1 = preprobe$ inside a PacketOut message through switch A towards switch B as depicted in Figure 4.

This technique comes with two sources of overhead. First, a switch needs to install the probe rules which reduces its usable rule update rate. Further, the probe rules are probed by data plane packets, which affects the neighboring switches'

[2]In principle, switches A and C can be the same switch. We keep them separated for the presentation purposes.

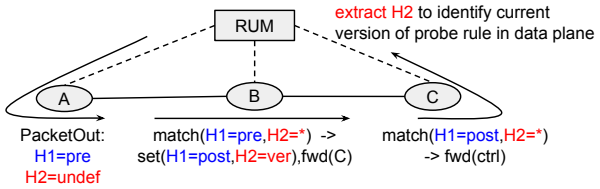


Figure 4: Network-wide probing solution. There are two rules preinstalled at each switch and only the version of the probing rule is updated over time.

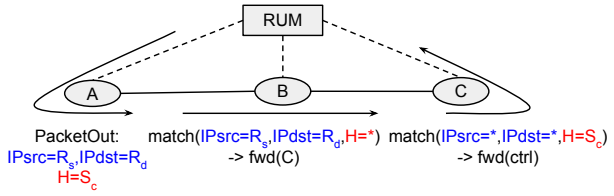


Figure 5: Probing for a rule matching IP packets with source R_s and destination R_d. A probe packet matches the tested rule at B and a send-to-controller rule at C.

control planes though the PacketOut and PacketIn messages. Thus, there is a trade-off between notification delay and the usable update rate.

3.2.2 General probing

The final strategy addresses the problem of switches that reorder rule modifications despite the use of barriers. In such a case, confirming that the last update took place is insufficient to acknowledge all previous updates. Specifically, it means that we cannot rely on probe rules described previously. Instead RUM needs to confirm each modification separately, where a modified rule may match an arbitrary set of header fields.

In this strategy, we need to reserve a header field H that is not used in the network, meaning that all normal rules have it wildcarded and no packet has it set to a value used by RUM (*e.g.*, VLAN, MPLS or ToS depending on the deployment). At the beginning, each switch i gets assigned a unique value S_i of field H. Each switch i then installs a high priority probe-catching rule that sends all packets that match on $H == S_i$ to the controller. Figure 5 shows a scenario where RUM confirms the installation of the rule that matches packets with an IP source R_s and IP destination R_d and forwards them to switch C. Assuming the action of the probed rule is to send the traffic to switch C, we use switch C with its probe-catch rule matching on $H == S_c$ to receive the probes. To create the probe packet, RUM computes an intersection of the probed rule on switch B and probe-catch rule on switch C. In our example, the probe packet has $IP_{src} = R_s$, $IP_{dst} = R_d$, $H = S_c$ and arbitrary remaining header fields. This probe gets injected through any neighbor of switch B (*e.g.*, switch A). As soon as the tested rule gets installed, the controller observes the probe packet coming from switch C inside a PacketIn message. The same method can detect rule deletions (probes stop arriving at the controller) or rule modifications (probes reach the controller from a different neighbor of B or have header fields modified to new values in case of header rewriting rules).

Overlapping rules. The previous, simplified description does not take into account the fact that there is more than one rule installed at a given switch. When creating a probe packet for a particular rule, RUM needs to take into account other rules such that the probe does not get forwarded by any other, already installed rule. In particular, probe generation needs to address two issues.

First, the generated probe packet must not match any higher-priority rule which overlaps with the probed rule. While finding a probing packet that hits exactly the tested rule is NP in a general case, others [4,17] show that in practice the problem can be solved quickly for real forwarding tables. Second, the generated probe packet must have a different forwarding action (*i.e.*, either a different output port or a different rewrite action) than the lower-priority rule matching the probe when the probed rule is not installed yet (otherwise we cannot distinguish if the packet was processed by the probed rule or the lower-priority rule). Note that as a special case, we can probe for rules dropping packets if there exists an overlapping lower-priority which does not drop the packets (a common case of ACLs and forwarding rules combination). If no suitable probe exists, RUM falls back to one of the control plane-based techniques. For example, if the probed rule is fully covered by higher priority rules, or if it covers other, already installed lower priority rules that have exactly the same actions, probing cannot reveal when the rule got installed.

Reducing the number of switch-specific values. This technique relies on using a header field and values that are unused by the live traffic in the network. Because there may be few such fields and values, it is essential to reduce the number of required values. However, to prevent the tested switch from sending the probe directly to the controller, each two adjacent switches need to have different identifiers. Thus, instead of using a network-wide unique value of S_i for each switch i, one can solve an instance of the vertex coloring problem for which there are well known heuristics [15].

4. PROTOTYPE

We implement a RUM prototype that works as a TCP proxy between the switches and the controller. The switches connect to the proxy as if it was a controller, and the proxy then connects to a real controller using multiple connections, impersonating the switches. This design allows us to modularly compose RUM as a chain of proxies to add functionality and freely replace components. For example, a barrier layer built on top of the acknowledgment layer is just another proxy. We implement the proxies using the POX platform.

In the current implementation we assume IP-only traffic and rely on the ToS field for probing. Because there are only 64 ToS values, we need to periodically recycle them in longer experiments. Moreover, we assume that the rules do not overlap,[3] and therefore, selecting a probing packet degrades to using the same source and destination addresses as in the rule's match.

While the OpenFlow specification lacks messages to confirm that a rule modification was successfully applied, it defines error messages used when something goes wrong. We reuse an error message with a newly defined (unused) error code for positive acknowledgments. Alternatively, one could potentially add vendor-specific messages to the protocol.

Finally, the hardware switch we use does not support priorities but takes the rule installation order to define the rule

[3] Except a low priority drop-all and high priority probe rules

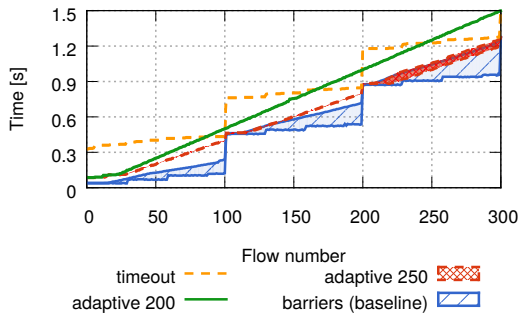


Figure 6: Flow update times when using control-plane only techniques. The reliability depends on correct estimation of the switch performance.

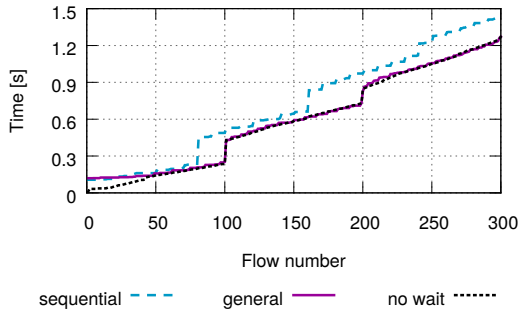


Figure 7: Flow update times with probing. There are no packet drops and the overhead of the general technique is negligible compared to the best achievable update time.

importance. Therefore, we carefully place the low priority rules early, and make sure that other rules do not hide the high priority ones.

5. EVALUATION

We evaluate RUM in the same end to end experiment as presented before, and using a hardware OpenFlow switch (HP 5406zl) that incorrectly implements barriers.[4] Further, we use low level benchmarks to analyze the properties and trade-offs in our techniques. Admittedly, these are just small scale experiments. However, a large scale test would require a testbed built of hardware switches because emulators use software switches that perform differently than the real ones. We do not have access to such a testbed.

5.1 End to end experiment

We first show that the presented techniques solve the dropped packets problem described in Section 1. The setup is as in Section 1 and we send data plane traffic at a rate of 250 packets/s per flow (75000 packets/s in total). We use the previously described control plane-only techniques and in Figure 6 plot the times when the last data plane packet following the old path and the first packet going along the updated path arrives at the destination. The area between the two lines visually represents the periods when packets get dropped.[5]

[4] The precise characteristics can be found in [7].

[5] If the delay between the two packets is lower than our measurement precision, we plot a single line.

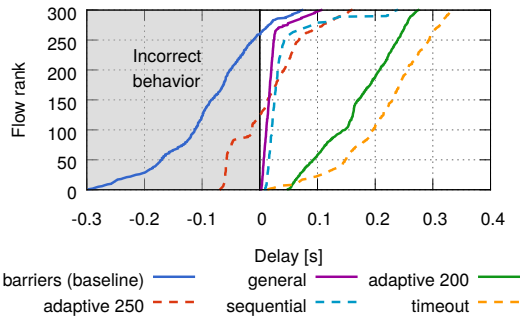


Figure 8: Delay between data plane and control plane activation. Using barriers leads to incorrect behaviors, control plane techniques increase the update time and the data plane techniques strike a balance.

An update with barrier messages is the fastest, but because the barrier replies are arriving too soon, rules at switch S1 get updated before rules at switch S2 are in place, which leads to extensive periods of packet drops (a total of 6000-7500 packets got lost in each of multiple runs of this experiment). The three visible steps in flow installation times are an artifact of the way how the switch synchronizes the data and control plane [7].

Using a 300-ms timeout solves a packet drop problem, but increases the average time it takes before a flow starts following a new path from 592 ms to 815 ms. Finally, while the 300-ms timeout is sufficient when there are up to 300 rules in the switch flow table, it becomes too short when the table occupation grows [7].

Based on the measurements, we set the adaptive timeout to assume that a switch performs 200 and 250 rule modifications per second. We see that the technique offers a stable performance over time, however when flow table occupancy increases and the assumed update rate is overestimated (250), the acknowledgments arrive too early and the network starts dropping packets.

Figure 7 shows the results of the same experiment but when using the data plane probing techniques, which guarantee no packet drops. For comparison, we plot the result when all flow modifications are issued at once to all the switches (*no wait*). It shows the shortest update duration one can get, limited only by the slowest switch update rate, but also offers no theoretical consistency guarantees. The *sequential probing technique* requires additional rule modifications (we modify a probing rule after every 10 real modifications). This fact is noticeable, because the data plane synchronization steps [7] are more frequent which hurts the update performance. On the other hand, the *general probing technique* does not require additional rule updates, but only sending and receiving data plane probes. If probing up to 30 oldest flow modifications at once, every 10 ms, the flows get updated almost as quickly as the lower bound.

We originally send packets belonging to each of the updated flows every 4 ms and observe no drops. To verify that there are no transient periods shorter than 4 ms when packets are dropped, we randomly select a single flow ten times and send traffic for the flow at 10000 packets a second. Once again we observe no drops.

Barrier Layer Performance. To validate the overhead of a full barrier layer, we rerun the same experiment with our reliable barrier layer introduced before and sending a bar-

rier after every 10 flow modifications. When a switch does not reorder modifications across barriers, the total update time and the particular curve of flow update times is the same as for the normal sequential probing technique. On the other hand, if the switch can reorder modifications and RUM needs to buffer them to ensure correct ordering, the overhead is big and the total update time is twice that of the general probing technique. Understandably, this time increases even more (up to 5 times) if the barriers are more frequent (up to a barrier after each command).

5.2 Low level benchmarks

After observing that RUM achieves its main high level goal - allowing for reliable network updates with consistency guarantees even on unreliable switches, we analyze how changing variables in each technique affects various aspects of the update.

The setup in the next two experiments is the same. Initially, there is a single, low priority drop-all-packets rule at the switch. Then, a controller modifies R rules in the switch in a way that at most K modifications are unconfirmed at any time. When a modification confirmation comes, the controller issues a new update. Meanwhile, we send data plane traffic matching the modified rules again at a rate of 250 packets/s for each rule.

Data plane delay. First, we measure when packets matching a particular rule start arriving at the destination (data plane activation) and when the controller receives a confirmation that the rule was installed (control plane activation). In Figure 8 we plot the delay between the data plane and control plane activations for various techniques for $R = 300$ and $K = 300$ (send all rules at once). All values below zero mean incorrect behavior and positive values cause a delay during an update. Thus, the ideal behavior would be a vertical line at $x = 0$. We see that, as mentioned in the introduction, barrier replies arrive even 300 ms before the rule gets applied. Using a 300-ms timeout fixes the correctness problem in this case, but is very inefficient – for the median the update wastes 230 ms on each barrier. The adaptive timeout technique achieves very good results, however, it requires precise models, otherwise the delay can fall below zero (possible inconsistencies). Finally, both probing techniques never incur a negative delay and, accordingly, are within 70 ms and 30 ms after the data plane modification for 90% of modifications.

Impact of probing rules. A technique that relies on installing probing rules to confirm that previous modifications took place requires finding a balance between the frequency of such confirmations and measurement precision. In this experiment we issue $R = 4000$ modifications and vary the number of modifications after which RUM sends a probing rule, as well as the number of allowed, unconfirmed modifications (K). Table 1 shows that the usable modification rate (rate of real modifications, not counting probes) is proportional to the number of rules probed at once and is usually close to the expected rate. When the number of allowed unconfirmed messages is low compared to the number of rules confirmed at once, the controller does not receive the confirmations quickly enough to saturate the switch.

Number of probes a switch can process. Sending data plane probes requires a switch to process two types of messages. First, an injecting switch receives a PacketOut and forwards a probe packet to the required port. Then, the receiving switch gets the packet, encapsulates it in a PacketIn message, and sends it to the controller. In the previous experiments, we used software switches as sending and receiving switches. Here, we instead benchmark the performance of a real hardware switch. We measure the PacketOut rate by issuing 20000 PacketOut messages and observe when the corresponding packets arrive at the destination. Similarly, we install a rule forwarding all traffic to the controller and inject traffic to the switch to measure the PacketIn rate. The rates are 7006 PacketOut/s and 5531 PacketIn/s, averaged over 5 runs. Both of these values are sufficient to allow RUM to probe the rules frequently.

Probing frequency	$K = 20$	$K = 50$	$K = 100$
after 1 update	51%	51%	51%
after 2 updates	64%	68%	68%
after 5 updates	74%	86%	86%
after 10 updates	76%	93%	94%
after 20 updates	74%	95%	98%

Table 1: Usable rule update rate with the sequential probing technique (normalized to a rate with barriers).

Finally, our additional experiments show that processing PacketIn requests in parallel with rule modifications has minimal impact on the rule modification throughput—new rate is over 96% of the original rate without any other messages. Similarly, processing PacketOut messages in parallel with rule modifications decreases the rule update rate by at most 13% for the ratio of PacketOut messages to rule modifications up to 5:1.

6. RELATED WORK AND CONCLUSIONS

SOFT [6] identifies inconsistencies in the way OpenFlow switches respond to commands, but cannot ensure that the rules are installed in the data plane.

To the best of our knowledge, our work is the first to look at the network update consistency from the practical point of view, using the real switches. There is a large body of work that guarantees particular properties during an update [3,8,9,11,14], but they all assume correctly-functioning switches. We show that this assumption does not hold and propose a workaround that allows the aforementioned solutions to work correctly.

ATPG [17] is a system that determines and injects packets that exercise all rules in a network. It is however an end to end solution designed to work on a coarser granularity than RUM. Its main goal is testing network correctness in a stable state, not during an update.

Oflops [12] is a study that benchmarks switches in a controlled environment and reports some of the problems we describe in this work. In contrast, we propose a solution to these problems.

Finally, there are efforts to build switch models [2, 16]. This work can be help RUM to better estimate timeouts and optimize the probing.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers who provided excellent feedback. 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.

8. REFERENCES

[1] OpenFlow Switch Specification. `http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf`.

[2] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *HotSDN*, 2013.

[3] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. In *HotSDN*, 2013.

[4] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.

[5] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.

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

[7] M. Kuźniar, P. Perešíni, and D. Kostić. What you need to know about SDN control and data planes. Technical Report EPFL-REPORT-199497, EPFL, 2014.

[8] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zUpdate : Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.

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

[10] P. Perešíni, M. Kuźniar, M. Canini, and D. Kostić. OpenFlow Needs You! A Call for a Discussion about a Cleaner OpenFlow API. In *EWSDN*. IEEE, 2013.

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

[12] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. Oflops: An open framework for openflow switch evaluation. In *PAM*, 2012.

[13] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *OSDI*, 2010.

[14] S. Vissicchio, L. Vanbever, L. Cittadini, G. Xie, and O. Bonaventure. Safe Updates of Hybrid SDN Networks. Technical report, UCL, 2013.

[15] D. J. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1), 1967.

[16] M. Yu, A. Wundsam, and M. Raju. NOSIX: A Lightweight Portability Layer for the SDN OS. *ACM SIGCOMM Computer Communication Review*, 44(2), 2014.

[17] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *CoNEXT*, 2012.