A Safe, Efficient Update Protocol for OpenFlow Networks

Rick McGeer
HP Labs, Palo Alto, CA
rick.mcgeer@hp.com

ABSTRACT

We describe a new protocol for update of OpenFlow networks, which has the packet consistency condition of [14] and a weak form of the flow consistency condition of [14]. The protocol conserves switch resources, particularly TCAM space, by ensuring that only a single set of rules is present on a switch at any time. The protocol exploits the identity of switch rules with Boolean functions, and the ability of any switch to send packets to a controller for routing. When a network changes from one ruleset (ruleset 1) to another (ruleset 2), the packets affected by the change are computed, and are sent to the controller. When all switches have been updated to send affected packets to the controller, ruleset 2 is sent to the switches and packets sent to the controller are re-released into the network.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Network Operating System*; C.2.1 [**Computer Communication Networks**]: Network Architecture and Design—*Network communications, Ethernet*

General Terms

Algorithms, Design, Reliability

Keywords

OpenFlow; Reliable Update; Logic Synthesis;

1. INTRODUCTION

The problem of updates in networks has long been recognized. The fundamental problem is simple: even though the behavior of networks under static conditions can be understood, under dynamic conditions, when routes change, it is indeterminate and a variety of pathological conditions can occur. Indeed, in a standard network, where route updates occur at each switch independently and asynchronously in response to a variety of network events, the behavior of the network at any time is only stochastically predictable. OpenFlow[18] improves matters somewhat: since routes are chosen by a controller, and not by the action of an independent algorithm running on an individual switch, updates occur at times chosen by the controller. Indeed, if the controller could ensure that all switch updates happened instantaneously and synchronously (all updates took effect in the network at the same time, or at least at precise times on each switch) then OpenFlow would ensure that network updates resulted in consistent routing for all packets. However, switch updates take time, and are realized at different and unpredictable times for each switch in the network; as a result, the path taken by a packet is dependent on which switches were updated and in which order. Further, packet arrival times at individual switches are unpredictable. Though the resulting sheaf of possible paths and destinations for any packet is computable, the path and destination of the packet is still indeterminate; it could wind up in any destination in the sheaf.

The problem that we face is to ensure that each packet is handled by a consistent ruleset, under conditions of unpredictable update of each individual switch in the network. The destination of each packet under dynamic conditions will still have two possibilities; a packet might be handled by ruleset 1, or by ruleset 2. However, once one packet in a flow has been handled by ruleset 2, all subsequent packets will be handled by ruleset 2. Thus, for each flow, determinacy and causality is preserved. Each flow is handled entirely by ruleset 1; entirely by ruleset 2; or the flow's prefix is handled by ruleset 1 and its suffix by ruleset 2. These are strong guarantees, and are given by our method.

There are a variety of optimization criteria to consider as well. An expensive part of any switch is the Ternary Content Addressable Memory, or TCAM. This is the most general matching engine available on most switches, at least for fast-path traffic. Optimization of TCAM space is a subject of considerable interest. See, for example, [10][12], among many others. In this paper, we conserve switch rulespace. At most one set of rules is present on any switch at any time, and one can ensure, at some cost in control plane bandwidth, that the ruleset is guaranteed to be no larger than the larger of the two specified rulesets: in sum, this procedure introduces at little as zero rulespace overhead.

The remainder of this paper is organized as follows. In Section 2, we briefly review the OpenFlow protocol and its salient properties for this protocol, the identity of switch and Boolean networks, and the formalisms we'll be using in

HotSDN'12, August 13, 2012, Helsinki, Finland.

this paper. In Section 3 we will describe the OpenFlow Safe Update protocol. In Section 4, we demonstrate that the protocol is correct; all packets are processed uniformly by one ruleset or another, and for each flow, ordering determinism is preserved. A flow is uniformly processed by ruleset 1, or by ruleset 2, or its prefix is processed by ruleset 1 and its suffix by ruleset 2. In Section 5 we discuss related work, particularly[14], and contrast the strengths and weaknesses of our work compared with theirs. In Section 6 we discuss future work, including potential hybrid approaches.

2. BACKGROUND AND CONVENTIONS

An OpenFlow network[13][18] consists of a graph of forwarding tables, where transmission of a packet along an edge is given as a simple boolean function of the input packet and the forwarding table at the source of the edge. The tables are stateless logic functions of the bits in the packet headers. State and complexity are contained in a possibly-centralized network software controller.[2][3][6][9][17].

OpenFlow originated in a desire to make enterprise networks simpler and more controllable[3][4]. It has since spawned a revolution in networking research and practice, introducing the concepts of a network operating system [6][9] and a virtualized and separable forwarding plane[5][15][16]. OpenFlow networks blend the flexibility and controllability of software-defined networking and routing with the efficiency of a hardware forwarding plane. A number of campus networks have adopted OpenFlow as their day-to-day network under the auspices of the NSF GENI project. In this paper, we will use the OpenFlow switch model [11][12] used in verification of OpenFlow networks[8][11]: a switch transfer function is a simple Boolean function of the packet header bits.

[14] shows that the simplicity of an OpenFlow switch yields a reliable update protocol. We will discuss this paper in more detail in Section 5.

We assume the OpenFlow model of a switch. A switch can take one or more of four actions with respect to a packet: (1) Forward to a port or set of ports; (2) Drop the packet; (3) Modify one or more fields of the header; or (4) Forward the packet to a controller for further handling.

Actions that a switch takes are based on possibly prioritized *rules*. A rule is a specification of values of a subset of the bits in some tuple of fields of the packet header. The rule matches a packet when the values in the appropriate bits of the packet header match the rule.

Following [8][11][12], we characterize the *transfer function* of a switch s as a Boolean function $f^s : X \to Y$, where X is the *domain* of the rule (input packet header bits and port selector) and Y is the *range* of the rule (action).

We will be considering the transition from a ruleset, which will denote as the family of switch functions f_1^s, to a second ruleset, which will be denoting f_2^s. At each switch, we will be constructing an intermediate function, which we will denote f_{12}^s.

For a packet p, there is a *network transfer function* $F(p)$, which is the *actual* behavior of the packet as it traverses the network. The network transfer function for a packet is obtained in the obvious way from composition of the switch transfer functions: if (s, t, u, v) is the path traversed by packet p under a family of switch transfer functions, $F(p) = f^v(f^u(f^t(f^s(p))))$. We denote the network transfer function corresponding to the switch transfer family f_i^s as F_i. The intended behavior of a packet during the transition $f_1^s \to f_2^s$ we denote as $F_{12}(p)$. We will also be concerned about the network transfer function of a packet after it has partially completed its traversal of the network. We will denote the function $F^s(p)$ (and, similarly, $F_1^s(p), F_2^s(p), F_{12}^s(p)$ as the transfer functions for p *beginning at switch* s. Thus, in our previous example, $F^u(p) = f^v(f^u(p))$; note $F^s(p) = F(p)$ where p enters the network at s. We will also be concerned about the set of packets arriving at switch s under a ruleset: we denote this as D^s (for the *domain* of switch s, and, similarly, D_1^s, D_2^s.

The convention in the above is the foregoing: functions with subscripts denote the *specified* behavior of a packet under a control regime (1, 2, or 12); functions without subscripts denote the *actual* behavior of the packet as it traverses the network.

3. THE PROTOCOL

In this section, we describe the *OpenFlow Safe Update Protocol*. It achieves the two goals of basic protocol correctness. Given rulesets f_1^s, f_2^s:

1. $F(p) = F_1(p)$ or $F_2(p)$
2. $F(p) = F_2(p)$ for packet p in a flow implies $F(q) = F_2(q)$ for all successive packets q in the flow.

This language makes precise our intuitive notions of correctness in this area: each packet is handled by one, consistent, set of rules, and as far as each flow is concerned, the change in network transfer functions is atomic. (1) is the *per-packet consistency condition* of [14]. [14]'s *per-flow consistency condition* is a stronger version of (2): it requires that $F(q) = F_2(q)$ iff $F(p) = F_2(p)$.

Our strategy is based on two fundamental assumptions, which essentially say that the transition between rulesets is well-formed:

1. Any packet sent to the controller during a transition can be reliably sent to its destination under either ruleset. This does not imply that the controller will send an arbitrary packet to its correct final destination; we leave open the possibility that a packet sent to the controller will be re-injected into the network at an arbitrary point, from which it can complete its journey.
2. The update $f^s \to g^s$ is atomic: switch s computes f^s, g^s, or drops packets which arrive while the transition is occuring, but it does not compute a hybrid function.

In addition, we exploit an explicit property of the underlying OpenFlow protocol: the controller has a direct connection to each switch in the network, through a (logically) distinct control plane, and can reliably deliver any packet to any network switch. The overall strategy behind the protocol is straightforward. During the transition, a switch doesn't know if other switches in the network are forwarding packets according to ruleset 1 or ruleset 2, and so it's unsafe to send any packet which it knows is handled differently under the two network regimes to another switch; it is safe to send it to the controller. Now, a switch doesn't know if an arbitrary packet will be handled differently downstream of itself, but it does know that packets *it* is handling differently are almost certain to be handled differently downstream.

Hence it sends those packets to the controller. Conversely, if it handles a packet the same way under both rulesets, it's safe to send it on. A downstream switch which handles it differently will send it to the controller. Each switch s therefore sends to the controller those packets it will handle differently under ruleset 2 and ruleset 1. This defines the intermediate family of transfer functions f_{12}^s.

The OpenFlow Safe Update protocol divides time into four epochs, each measured at the controller. In the first epoch, all switches are using ruleset 1; the transfer function at each switch is f_1^s, and each packet p is processed exclusively by this family of transfer functions: $F(p) = F_1(p)$. In the second epoch, the family f_{12}^s is being loaded onto each switch. During this epoch, the transfer function at a switch starts at f_1^s and ends as f_{12}^s; $F(p) = F_1(p)$ *or* $F_{12}(p)$ for each packet p, depending upon the relative arrival time of f_{12}^s and p at each switch s.

Each switch sends a completion signal to the controller when a new ruleset is loaded. Once the last switch has sent a completion signal to the controller, the controller waits for the maximum network latency (measured end-host to end-host, NIC-to-NIC), and epoch three begins.

At the start of epoch three, the controller sends f_s^2 to all switches, and releases all packets sent to the controller during epoch two. Since a packet p may arrive at a switch s whose transfer function is still f_{12}^s, it may be sent to the controller. However, the controller continuously releases packets sent to the controller during epoch three, and so these packets will be delivered to their destination; as a result, *all* packets sent during epoch three have $F(p) = F_2(p)$.

Once the last completion signal has been sent to the controller from the switches, the controller waits for the maximum network latency. Epoch four then beings. During Epoch four, $f^s = f_2^s$ for all switches s, and all packets sent into the network in epoch four are processed entirely by switches using the transfer function f_2^s; hence $F(p) = F_2(p)$ for all packets p. After epoch four begins, a new update can begin at any time.

This yields the *OpenFlow Safe Update Protocol*:

1. Send the intermediate transfer function f_{12}^s to each switch s, defined as
$$f_{12}^s(p) = \begin{cases} f_1^s(p) & f_1^s(p) = f_2^s(p) \\ \textit{controller} & \textit{otherwise} \end{cases} \quad (1)$$
2. When a switch updates to f_{12}^s, it sends a message back to the controller indicating it's now forwarding packets according to f_{12}^s.
3. When every switch has sent the f_{12}^s acknowledgement message, wait the maximum network latency, then update each switch to f_2^s and send packets received under the f_{12}^s regime to their f_2^s destinations, preserving original transmission order (if packets p and q are in the same flow, and p precedes q in transmission order, send p to the final destination first).
4. When a switch updates to f_2^s, it sends a message back to the controller indicating it's now forwarding packets according to f_2^s.
5. When every switch has sent the f_2^s acknowledgement message, wait the maximum network latency, note update complete.

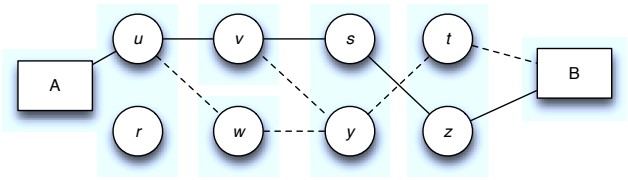

Figure 1: OpenFlow Safe Update Example

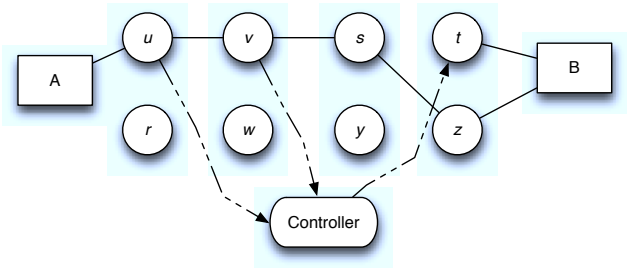

Figure 2: f_{12}^s Intermediate Functions for Example

We illustrate the operation of the OpenFlow Safe Update Protocol using the network illustrated in Figure 1. In this figure, a flow runs from A to B over the switches labelled $\{u, t, s, v, r, w, y, z\}$. The rules in f_1^s are shown in solid lines; the rules in f_2^s are shown in dashed lines.

The f_{12}^s rules are shown in broken arrowed lines in Figure 2. Note in this figure that nodes u, v forward packets to the controller, but w, y, s, z and t do not; instead, each forwards packets according to either f_1 or f_2. The explanation of each is as follows:

1. Packets from A to B whose handling is changing do not arrive at w, y, or t under f_1, and so they cannot arrive under f_{12}. Either they were previously intercepted by f_{12}^u or f_{12}^v, or they were not; in the former case, the packets went to the controller; in the latter, they went along their f_1 path.
2. The handling of packets going through s and z do not change, even though they will not receive packets under f_2. If they continue to receive packets after f_{12}^s, f_{12}^z have been installed, this indicates that f_{12}^u, f_{12}^v have not been installed; therefore it's safe to forward these packets under f_1 rules.
3. w, t don't receive packets under f_1, and therefore won't under f_{12}, so they forward packets as before

A Note on Switch Rulespace. The maximum rulespace taken on switch s during the course of the protocol is the maximum of the sizes of f_1^s, f_2^s, f_{12}^s. The size of a switching ruleset is given by the number of rows in the ruleset, which is in turn given by the number of distinct terms in the domain of the ruleset. In general, the size of f_{12}^s should be bounded above by the maximum size of f_1^s, f_2^s. However, a reviewer pointed out one case where the size of f_{12}^s is given by the sum of the sizes of f_1^s, f_2^s. This occurs when f_1^s and f_2^s partition the domain of s on different header fields, and use incompatible actions. When this happens, the protocol proposed here can take the same rulespace as that of [14] during the transition, and incurs extra control plane traffic on top of that.

However, the function f_{12}^s here denotes the *minimum* set of packets which must be sent to the controller in order to ensure correctness; it is also possible to send the controller some packets whose destinies do not change under the change in rulesets to the controller. Under ruleset two, they will simply complete their journey, after being released from the controller. We therefore have a sheaf of functions, and any g in the sheaf can be chosen to be sent to s in step one; this conserves switch rulespace at the expense of control plane bandwidth. One function g to choose has size exactly that of f_2^s; hence we have a guarantee that one can always choose an intermediate function that ensures zero rulespace overhead on all switches, possibly at the cost of control plane bandwidth.

A Second Example. An anonymous reviewer asked for clarification on the following simple example; the question nicely illustrates some fine points of the protocol, and so we offer the analysis here. Consider a network of two switches `A` and `B`, and the following rulesets:

Ruleset	Switch A	Switch B
One	`*` → `mark VLAN 1`	`if VLAN 1` → port 1
Two	`*` → `mark VLAN 2`	`if VLAN 2` → port 2

The reviewer asked what would happen if the rule update reached switch `B` before it reached switch `A`. The answer is that *switch* `B` *is not updated in this transition: since* $f_1^B = f_2^B$, $f_1^B = f_2^B = f_{12}^B$. Only switch `A` is updated.

Considering this example shows an interesting optimization of the protocol. There is no need to go through the intermediate step of loading f_{12}^A in this case, since only one switch (`A`) is updated. This is a special case of a more general result: a switch s need not be updated with the intermediate function f_{12}^s iff, for each packet p such that $f_2^s(p) \neq f_1^s(p)$, there is no switch t *downstream* of s under *either* ruleset 1 or ruleset 2 such that $f_1^t(p) \neq f_2^t(p)$. This is an interesting analog in the routing world to the well-known idea of *don't-cares* in logic synthesis. Specifically, this is a close analog to the concept of output don't-cares in multi-level logic synthesis systems[1].

4. CORRECTNESS OF THE PROTOCOL

In this section we demonstrate the correctness of the OpenFlow Safe Update Protocol. Specifically, we demonstrate that for each packet and flow, the following two properties hold:

1. For each packet p, p is completely processed by either f_1 or f_2. Specifically, $F(p) = F_1(p)$ or $F_2(p)$.

2. For each flow of the form $(...., p, q, ...)$, where sequence here refers to order of departure from the transmitting node, if p is processed by f_2 then so is q. Formally, if $F(p) = F_2(p)$, then $F(q) = F_2(q)$.

We demonstrate this by considering the behavior of the network during each epoch. During Epoch 2, $f^s = f_1^s$ or f_{12}^s for all s. We will show that each packet p which traverses the network during Epoch 2, or during both Epoch 1 and Epoch 2, must have $F(p) = F_1(p)$ or $F(p) = \textit{controller}$. We will also show that if q succeeds p in a flow, and $F(p) = \textit{controller}$, then $F(q) = \textit{controller}$.

LEMMA 1. *For each packet p that traverses the network during Epoch 2, or during Epoch 1 and Epoch 2, $F(p) = F_1(p)$ or $F(p) =$* controller.

PROOF. Let the sequence of switches traversed by p under ruleset 1 be $(s_1, ..., s_n)$. Either $f^{s_j}(p) = f_1^{s_j}(p)$ for all $1 \leq j \leq n$, or there is some least k such that $F^{s_k}(p) = f_{12}^{s_k}(p) \neq f_1^{s_k}(p)$. In the former case, $F(p) = F_1(p)$ by definition, and in the latter, $f_{12}^{s_k}(p) = \textit{controller}$ by (1), and so $F(p) = \textit{controller}$. □

LEMMA 2. *Let $(..., p, q, ...)$ be a flow, and p and q traverse the network during Epoch 2 or Epoch 1 and Epoch 2, and $F(p) =$* controller. *Then $F(q) =$* controller.

PROOF. Let the sequence of switches traversed by p under ruleset 1 be $(s_1, ..., s_n)$. Since $F(p) = \textit{controller}$, there is some least k such that $F^{s_k}(p) = \textit{controller}$. If q arrives at s_k, it arrives after p, and so $f^{s_k}(q) = f_{12}^{s_k}(q) = \textit{controller}$, so $F(q) = \textit{controller}$. So suppose q doesn't reach s_k. Then there is some s_j, $j < k$, such that $f^{s_j}(q) \neq f_1^{s_j}(q)$. But then $f^{s_j}(q) = f_{12}^{s_j}(q) = \textit{controller}$, so $F(q) = \textit{controller}$. □

Lemma 1 and Lemma 2 establish that at the end of Epoch 2, every packet that has exited the network has either reached its destination under ruleset 1 or has been sent to the controller; and if one packet from a flow has been sent to the controller, then every subsequent packet from that flow is also sent to the controller. At the end of Epoch 2, the following condition holds:

- Every packet that has exited the network is either at its ruleset 1 destination, or at the controller;
- If one packet from a flow is at the controller, every subsequent packet that has exited the network is also at the controller; and
- Every packet currently transiting the network entered the network while the switch function f^s at each switch was f_{12}^s. This is because the end of Epoch 2 is one maximum network latency *after* the last switch loaded the rules f_{12}^s. Hence, any packet traversing the network while some f_1^s rules were still loaded has exited the network, either to the controller or to its ruleset 1 destination.

Epoch 3 begins with the dispatch of the ruleset f_2^s to each switch s, and the release of each packet p held by the controller into the network. The protocol does not specify to *which* switch a packet p held by the controller is released; it can be any switch s such that $F_2^s(p) = F_2(p)$. We demonstrate that during Epoch 3, for each packet that traverses the network, $F(p) = F_2(p)$, whether the packet enters the network from an end-host or from the controller.

LEMMA 3. *For each packet p that traverses the network during Epoch 3, or Epochs 2 and 3, $F(p) = F_2(p)$*

PROOF. Let the sequence of switches traversed by p under ruleset 2 be $(t_1, ..., t_m)$. Either $f^{s_j}(p) = f_2^{s_j}(p)$ for all $1 \leq j \leq m$, or there is some least k such that $F^{s_k}(p) = f_{12}^{s_k}(p) \neq f_2^{s_k}(p)$. In the former case, $F(p) = F_2(p)$. In the latter case, $F(p) = \textit{controller}$. But in this case, during Epoch 3, p is returned to the network at some switch s such that $F_2^s(p) = F_2(p)$. p will then either proceed to its destination under ruleset 2 or be returned to the controller But if it is returned to the controller, it will be reinserted into the network at a switch t such that $F_2^t(p) = F_2(p)$. Before the end of Epoch 3, all switches u in the network will have $f^u = f_2^u$, and then $F(p) = F_2(p)$, and done. □

We are now in a position to prove the fundamental result of this section: the correctness of the OpenFlow Safe Update protocol. This involves two proof obligations. First, each packet p is processed by a consistent set of rules: $F(p) = F_1(p)$ or $F_2(p)$. Second, each flow is handled causally: if q follows p in a flow, and $F(p) = F_2(p)$, then $F(q) = F_2(q)$. Both of these follow from Lemmas 1-3.

THEOREM 1. *For packet p, $F(p) = F_1(p)$ or $F_2(p)$*

PROOF. Let the sequence of switches traversed by p under ruleset 1 be $(s_1, ..., s_n)$. Either $f^{s_j}(p) = f_1^{s_j}(p)$ for all $1 \leq j \leq n$, or there is some least k such that $f^{s_k}(p) = \textit{controller}$. In the former case, $F(p) = F_1(p)$. In the latter, if p is inserted into the network during Epoch 2, by Lemma 1, p finishes Epoch 2 in the controller, and by Lemma 3 $F(p) = F_2(p)$. If p is inserted into the network during Epoch 3, by Lemma 3, $F(p) = F_2(p)$. □

Theorem 1 shows that packets are always processed completely by ruleset 1 or 2. We now show that if q succeeds p in a flow, and $F(p) = F_2(p)$, then $F(q) = F_2(q)$.

THEOREM 2. *If q succeeds p in a flow, and $F(p) = F_2(p)$, then $F(q) = F_2(q)$.*

PROOF. If q is inserted into the network in Epoch 3, by Lemma 3 $F(q) = F_2(q)$. If q is inserted into the network in Epoch 1 or Epoch 2, then p is inserted into the network in the same Epoch. By Lemma 1, p is sent to the controller (since $F(p) = F_2(p)$) . Hence, by Lemma 2, q is sent to the controller, and then by Lemma 3 $F(q) = F_2(q)$. □

5. RELATED WORK

The network formalism has been explored in a number of papers, mostly in the verification of network transfer functions [8][11] and optimization of rulesets[12].

Update of OpenFlow networks has been explored in [14]. In this paper, the authors establish two criteria for correctness of an update mechanism: *packet consistency*, the requirement that each packet be handled by a single set of rules, and *flow consistency*, the requirement that all packets in a flow be handled by the same set of rules. In our terminology, the packet consistency requirement is that $F(p) = F_1(p)$ or $F_2(p)$ for each packet p, and the flow consistency requirement is that $F(q) = F_1(q)$ iff $F(p) = F_1(p)$. The flow consistency requirement in this paper is weaker; [14] required that packet handling not change in the course of a flow; we require that it changes at most once in response to a change in the network transfer function. There are use cases in which the weaker condition is required[7]. It should be noted that the method of [14] isn't restricted to the strong flow consistency condition; it can implement the weaker condition used here.

[14] represents both sets of rules in each switch, differentiating them with an unused header bit. In our notation, the transfer function at switch s is $f^s = \bar{x}f_1^s + xf_2^s$, where $+$ represents logical disjunction and xf represents x and f. Packets are marked on ingress as to which transfer function should apply; packets are handled consistently by a single set of rules, and marking each packet in a flow consistently ensures the strong flow consistency condition. After all flows marked for ruleset 1 have terminated, the rules $\bar{x}f_1^s$ and be deleted.

This is an elegant method, and has both strengths and weaknesses compared with our protocol. [14] requires that both sets of rules are represented on the network switches; in the worst case, this represents a 100% overhead in the consumption of rulespace resources on the switches, though one can imagine a variety of optimizations to reduce this. Conversely, in the method described here, only one set of rules is present on a switch at any time. The method of [14] has significant strengths compared to our method, however. The greatest is efficient delivery of packets and conservation of control plane bandwidth. Our method sends packets from every flow whose routing changes to the controller; if there are many such flows, or if one flow is particularly heavy, this will consume a great deal of control plane bandwidth. Further, our method holds packets at the controller until switches have completed their transition; this adds latency and may result in timeouts on the end-hosts. This can be overcome by having the controller directly deliver packets to their ruleset-2 destination, rather than holding packets until the transition is complete; the results of Section 4 are unaffected by this change.

Diversion of packets was introduced in DIFANE[19], for the purpose of efficient distribution of rules and to build scalable distributed controllers for OpenFlow networks. In DIFANE, packets were selectively redirected to "authority switches" in the network, which would determine appropriate routing for the packet and update the rule caches on other switches in the network.

DIFANE's purpose was to conserve scarce rulespace on most switches; however, it can be easily adapted to our purpose. The update rules f_{12}^s divert packets to the local authority switch, which initially sends packets according to the family f_1^s; however, when it is updated to f_2^s, it sends the new f_2^s rules to the switches in its domain. This retains all the results given above, but does not involve holding packets in a controller. Again, the results of Section 4 are unaffected by this change.

6. CONCLUSIONS AND FURTHER WORK

In this paper, we have presented a safe update protocol for OpenFlow networks and demonstrated its correctness. This protocol differs from previous approaches[14]; it conserves switch rulespace resources at a cost in control plane bandwidth, and, potentially, end-to-end latency for affected network flows. Which protocol is deemed superior is dependent on which resources are considered most scarce in a given situation. If an operator needs to conserve rulespace, the approach in this paper may be preferable; conversely, if switch rulespace is cheap and plentiful, the previous approach is superior.

There are quite likely to be other interesting points in the trade space for future work. Hybrid approaches, combining the technique explored here and that of [14] are also possible. For example, one approach would use the technique of putting both rulesets on a switch where both could be accomodated, and using the technique described here only for those switches that don't have space for the combined rulesets. The results of Section 4 still hold in this case.

This work essentially combines two central ideas: the use of logic synthesis and verification techniques to determine precisely which packets at each switch are affected by an update, and the use of the controller as a highly-connected oracle to safeguard delivery of packets whose transit would

be interrupted by a flow. These techniques can be separated. For example, one could modify [14] using synthesis techniques described here, by loading the function describing only the packets whose handling is modified in the transition to the previous ruleset (taking care to mark the rules and the packets appropriately). In fact, one can do still better; the set of packets whose handling has changed and the set of all packets form a pair of Boolean functions, the lower and upper bounds of a function space; one can choose any function in that space whose representation is minimal.

Exploitation of don't-cares is another area ripe for future optimization. There are two sets, one of which was mentioned at the end of Section 3: using output don't-care sets to apply only a single update for switches whose packets are handled consistently downstream under either ruleset. We can also use these don't-care sets to minimize the logic functions f_{12}^s, as described in [12].

The work presented here is still very early, and many optimizations are possible. For example, though we have sent packets directly to the controller, we could as easily have sent them to a distributed set of proxies; no correctness result depends on a central repository for diverted packets.

Acknowledgments

The author thanks Jeff Mogul for stimulating conversations. The author also particularly thanks the anonymous reviewers of this paper for deep and thoughtful insights and comments. A number of improvements to the paper, particularly at the end of Section 3, are in direct response to the review comments.

7. REFERENCES

[1] R. K. Brayton, R. L. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang. Mis: A multiple-level logic optimization system. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(6):1062–1081, 1987.

[2] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking enterprise network control. *Transactions on Networking (ToN)*, 17(4):1270–1283, August 2009.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proceedings of ACM SIGCOMM*, August 2007.

[4] M. Casado, T. Garfinkel, A. Akella, M. Freedman, D. Boneh, N. McKeown, and S. Shenker. Sane: A protection architecture for enterprise networks. In *Usenix Security*, 2006.

[5] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. In *PRESTO*, 2010.

[6] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: Towards an operating system for networks. *ACM SIGCOMM CCR*, 38(3):105 – 110, 2008.

[7] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of the Usenix Conference on Network Design and Implementation*, April 2010.

[8] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proceedings of the Usenix Conference on Network Design and Implementation*, April 2012.

[9] 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 *Operating Systems Design and Implementation (OSDI)*, 2010.

[10] A. X. Liu, C. R. Meiners, and E. Torng. Tcam razor: a systematic approach towards minimizing packet classifiers in tcams. *IEEE/ACM Trans. Netw.*, 18(2):490–500, Apr. 2010.

[11] R. McGeer. Verification of switching network properties using satisfiability. In *Proceedings of ICC WSDN*, June 2012.

[12] R. McGeer and P. Yalagandula. Minimizing rulesets for tcam implementation. In *Proceedings IEEE Infocom*, 2009.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69âĂŞ74, 2008.

[14] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Software updates in openflow networks: Change you can believe in. In *Proceedings of HotNets*, 2011.

[15] R. Sherwood, G. Gibb, and M. Kobayashi. Carving research slices out of your production networks with openflow. *ACM SIGCOMM CCR*, 40(1):129–130, January 2010.

[16] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *Operating Systems Design and Implementation (OSDI)*, 2010.

[17] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying nox to the data center. In *Proceedings of ACM HotNets*, 2009.

[18] The openflow switch specification. `http://OpenFlowSwitch.org`.

[19] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *SIGCOMM*, 2010.