Accepted Manuscript

A lightweight protocol for consistent policy update on software-defined networking with multiple controllers

Diogo Menezes Ferrazani Mattos, Otto Carlos Muniz Bandeira Duarte, Guy Pujolle

PII: S1084-8045(18)30264-9

DOI: 10.1016/j.jnca.2018.08.007

Reference: YJNCA 2191

To appear in: *Journal of Network and Computer Applications*

Received Date: 20 January 2018

Revised Date: 25 July 2018

Accepted Date: 14 August 2018

Please cite this article as: Mattos, D.M.F., Duarte, O.C.M.B., Pujolle, G., A lightweight protocol for consistent policy update on software-defined networking with multiple controllers, *Journal of Network and Computer Applications* (2018), doi: 10.1016/j.jnca.2018.08.007.

A Lightweight Protocol for Consistent Policy Update on Software-Defined Networking with Multiple Controllers

Diogo Menezes Ferrazani Mattos[a,2], Otto Carlos Muniz Bandeira Duarte[b], Guy Pujolle[c]

[a] *Universidade Federal Fluminense - TET/PPGEET/UFF*
Niterói, Brazil
[b] *Universidade Federal do Rio de Janeiro - GTA/POLI-COPPE/UFRJ*
Rio de Janeiro, Brazil
[c] *Laboratoire d'Informatique de Paris 6 - Sorbonne Universities, UPMC Univ Paris 06*
Paris, France

Abstract

Network-policy updates have to be committed in a consistent way on distributed-controller software-defined networking. Otherwise, the network may experience unexpected transitory configuration states, which compromise the performance, the security or, even, the correct operation. In this paper, we propose a scheme that provides consistent policy updates without rule conflicts and transitory states. The main contributions are: (i) a protocol that serializes policy update commitments to provide consistency; (ii) a consensus interface proposal that facilitates controller agreements about the network configuration version; and (iii) an algorithm for checking if a new policy is an update, a refinement, or if it conflicts with already installed policies. We prove that our protocol achieves a global order for all policy updates and that our algorithm correctly composes all policies. Simulation results using real network topologies show that the proposed distributed policy update scheme achieves a per-packet consistent configuration with a low control message overhead.

Keywords:
Software-Defined Network, Distributed Control, Control Model, Consistency, Policy Enforcement

Email addresses: menezes@midiacom.uff.br (Diogo Menezes Ferrazani Mattos), otto@gta.ufrj.br (Otto Carlos Muniz Bandeira Duarte), Guy.Pujolle@lip6.fr (Guy Pujolle)

[1] Laboratório MídiaCom - Universidade Federal Fluminense (UFF) - R. Passo da Pátria, 156 - ZIP Code 24210-240, Niterói, RJ, Brasil phone: +55 21 2629-5696

[2] Grupo de Teleinformática e Automação - GTA - Universidade Federal do Rio de Janeiro (UFRJ) - P.O. Box: 68504 - ZIP Code 21945-972, Ilha do Fundão, Rio de Janeiro, RJ, Brasil phone: +55 21 3938-8635

1. Introduction

Network management engages in a continuous specification of network policies that includes traffic engineering and chaining of middleboxes and network functions [1, 2, 3]. The paradigm of Software-Defined Networking (SDN) simplifies the network management as it decouples the logically centralized control plane from the distributed data plane [4, 5]. Control applications lie at the control plane and access a global network view, which allows defining high-level network policies that encode the expected behavior of the network [6, 7]. Moreover, controller translates the policies into forwarding rules that configure the data plane behavior. In SDN, flow-table configurations of individual switches running on the data plane express the forwarding rules.

The network controller realization as a centralized server implies challenges to security, performance, and scalability of the network [4]. On the other hand, handling policy updates on the SDN control plane is a challenge on distributed computing, in which the proper operation of the network depends on consistently reasoning about concurrent policy updates and the interaction between all applied policies on the network [3, 6, 8, 9].

In this paper, we propose a lightweight protocol for handling policy updates on Software-Defined Networking with a distributed control plane. The main idea is to timely achieve consensus between controllers on the deployment order of the new policies on the network. The network has to react to a policy update request in a short time lapse to be compliant with the next-generation networking requirements [10, 11]. When concomitant policy updates arrive at different controllers, they have to agree about the installation order of all updates, and if a new update does not conflict with others. Hence, our main contributions are threefold:

- A lightweight consistency protocol that serializes commitment of concurrent policy updates launched by different controllers,
- An abstract consensus interface, in which controllers agree with the most current version of the network configuration,
- A simple algorithm for checking if a new policy is an update, a refinement, or if it conflicts with other policies already installed on the network.

In preliminary works[3], we first introduced the idea of a policy update scheme for distributed control plane, in which we assume the consistency model of per-packet consistent configuration update [2, 3, 6]. In this paper, we extend the policy update scheme for resisting to Byzantine faults and, also, we show the protocol convergence overhead under different fault probabilities. A per-packet

[3] D. M. F. Mattos, O. C. M. B. Duarte, G. Pujolle, *Um protocolo simples e eficiente para atualização consistente de políticas em redes definidas por software com controle distribuído*, in: SBRC, 2017. The manuscript is available at: http://portaldeconteudo.sbc.org.br/index.php/sbrc/article/view/2642/2604.

consistent update is the one that a single version of the network configuration processes every in-transit packet on the network, the previous or the updated configuration, but never more than one configuration. Unlike other proposals, which consider an abstract consensus interface among the controllers [6] or a single centralized controller [3, 12, 13], we propose a lightweight consistency protocol to achieve consensus among distributed controllers as part of the policy update scheme. We prove the correctness of our proposed consistency protocol through a formal model. Our analyses show that our proposal achieves a consistent configuration in two round-trip times (RTT), with a minimum number of control messages on the network.

The remainder of the paper is as follows. In Section 2, we present the related work. Section 3 discusses the challenges of updating policies on SDN with a distributed control plane. In Section 4, we propose our consistency protocol for policy updates on software-defined networking with distributed control. Our simulation and analytical results are discussed in Section 5. Section 6 concludes the paper.

2. Related Work

Upon a network-policy update, controllers send messages to switches to install flow processing rules. Although this procedure can be done sequentially, each switch may introduce different delays on processing and installing the rules. Thus, the rules installation in all switches is an asynchronous task, and it can lead to an arbitrary order of switches to finish the commitment of the new network configuration [9, 14]. An in-transit packet detects a mixture of new and old network configurations upon the installation of the policy update. If a mix of network configurations forwards an in-transit packet, it violates network invariants. Thus, policy updating schemes for software-defined networking avoid the occurrence of a mixture of configurations on the network due to partially applied updates. Therefore, the primary challenge is to coordinate the rule installation on switches to achieve an always-consistent network state [3, 6, 9, 15, 16].

Reitblatt *et al.* [3] propose the per-packet consistency model on updating policies in Software-Defined Networking. This model considers that a consistent policy update occurs when just a single global network configuration handles in-transit packets. Moreover, they propose the Two-Phase Update scheme for per-packet consistently updating policies on a software-defined network with centralized control [3]. The main idea is to forward packets according to the network configuration version tag on the packets and, thus, updating the network policies in two phases. The first phase adds new rules on ports of the switches on the network core. The switches apply the new policy to packets tagged with the new configuration version tag. The second phase acts on the ingress ports of edge switches. The scheme updates the rules on the edge switches to tag the in-going packets with the new configuration version number.

Besides, Canini *et al.* [6] extend the Two-Phase Update proposal to software-defined networking with distributed control [6]. Canini *et al.* [6] introduce the consistent policy composition (CPC) problem that stands for accepting policy

updates on a distributed control plane and composing them into a single and consistent network configuration. Thus, accepted policies cannot conflict with already deployed policies or with concomitant policy deployments. They propose a transactional interface for searching for conflicts between policies and just accepting policies that do not conflict with others. They also propose the `ReuseTag` algorithm. `ReuseTag` uses up to $f + 2$ tags to number the network-configuration versions, where f is the number of controller failures that the proposal can correctly handle. Although Canini *et al.* solve the CPC problem and show that they achieve the optimal tag complexity, they assume the previous existence of a consensus abstraction between controllers. Unlike Canini *et al.* [6], we propose a simple consensus protocol for distributed SDN that focuses on solving the consistent policy composition (CPC) problem.

Proposals optimize to the Two-Phase update scheme to reduce the overload on the network. Vissicchio and Cittadini [17] argue that Two-Phase Update leads to a high number of installed rules, which may avoid the applicability of the technique if there is not enough available memory on the switches. They propose an algorithm to compact the search space and compute an operational sequence for adding and changing rules during a policy update process on an SDN. Katta *et al.* [13] propose to perform the update in rounds. In each round, they perform a Two-Phase update on a subset of flows. The main idea is to remove the old set of rules after each round, freeing switch memory space. McClurg *et al.* [18] propose an algorithm for automatically seeking for an order to update the network configuration, in which all transitory configuration states keep invariant properties of the network.

In a previous paper, we have proposed the reverse update procedure [15], arguing that the policy update installation on the network is per-packet consistent if it follows the reverse path of the flows on the network. McGeer proposes to buffer in-transit packets on the controller while updating the network. After the update, the packets are re-injected on the network [12]. These proposals, however, still focused on software-defined networks with one centralized controller. Kablan *et al.* [19] propose to decouple the implementation of network functions from their state. They aim to store the function state in a distributed data storage and, thus, control and data plane queries and updates the network function state on the distributed storage. Hence, a policy update resides in updating the stored state. In this paper, in turn, we focus on a network that is controlled by a set of controllers, in which the controllers should arrive in a consensus about accepting concomitant updates and about which will be the tag of the new configuration version.

The most straightforward consensus decision among distributed nodes is whether a transaction should be committed [20]. A simple protocol for achieving consensus is the Two-Phase commit that uses one phase to propose a commitment and, after every node confirming to commit the transaction, a second phase sends the commit command to all nodes. Nonetheless, the two-phase commit may run into a deadlock if the node that starts the protocol fails. As a solution, the three-phase commit protocol proposal adds an extra phase between voting and committing a transaction. The third phase enables the state recov-

ery in case of an initiator node fault. Besides, a standard protocol to achieve a universal consensus, not just about committing or aborting a transaction, is the Paxos protocol. Paxos proposes a complex logic which provides availability and termination even when f server replicas fail [21]. Dang *et al.* argue that implementing Paxos logic into SDN switches enhances the performance of the network as it benefits distributed applications. Thus, they propose a simplified version of Paxos for achieving consensus between switches, called NetPaxos [22].

Besides some variations of Paxos consensus protocol, Raft is a consensus proposal that leads to a straightforward creation of application due to its simplicity to be understood [23]. Nevertheless, Raft also introduces an above operation to elect and to maintain a leader. Panda *et al.* [14] argue that strong consistency model, as provided by Paxos consensus protocol, is a hard assumption for controlling SDN. Hence, they propose an eventual-consistency model, where single image controllers are slightly modified to couple with some properties and, then, the controllers will eventually behave like a distributed controller. Although this approach may reduce the response time of the network to react to an event, it imposes the limitation that all controllers should be running a single image and they should behave similarly. We argue that electing a leader is a needless operation in the context of SDN policy updates due to the volatility of the updating task. Thus, we claim Paxos, and even Raft, add extra complexity to the policy update process. Hence, our proposal extends the two-phase commit protocol, creating a wait-free protocol for resolving the conflict between policies and for distributed reasoning about the most current configuration version tag. Our approach considers that each controller may receive different update requests and the distributed control have to agree on which updates, and in which order, they should be installed. The key aspect of our proposal is to consider each controller as a potential leader for the consensus protocol and, thus, each controller may start the consensus protocol on its initiative.

3. Policy Update Consistency

The logically centralized network control plane, on software-defined networking, consists of an abstraction of a global network view shared by all network controllers [4]. Hence, all controllers ought to have access to a consensus interface to update their global network view. In this scenario, distributed controllers

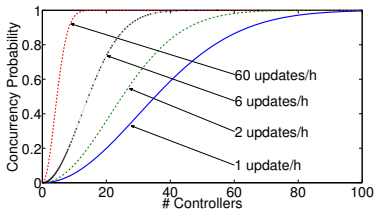


Figure 1: Probability of concurrence between two or more updates issued by different controllers on the network.

may issue simultaneous policy update requests. Figure 1 shows the probability of two or more controllers issuing a policy-update request at the same time. The probability of concurrency between updates is calculated as the birthday paradox, and we consider that each update may take up to 4 s to be completely installed on the network [24, 25]. We consider four different cases, in which each controller may issue 60 updates/h; 6 updates/h; 2 updates/h; or 1 update/h. In the case of the most dynamic network, 60 updates/h, the probability of concurrency between updates is over 0.5 when there are just 3 controllers on the network. If we consider an update rate of 2 updates/h, the concurrency probability reaches 0.5, when there are approximately 23 controllers on the network. To consistently apply these updates on the network, each controller has to be aware of update-request commitment order, and whether a new policy conflicts with others. In this way, Canini *et al.* states the Consistent Policy Composition problem [6], that is composing concurrent policy updates into one single and consistent global network configuration.

Considering the policy update problem in a distributed control environment, we identify two properties that should be satisfied for an algorithm to apply policy updates on a distributed control plane: consistency and composition. Unlike Canini *et al.*, we tackle the consistency problem separately from the composition problem. The consistent update is scheduling update requests and assuring that it is possible to establish a global order among all scheduled requests. After that, we consider the policy composition problem. In turn, the composition problem deals with accepting a new policy, or a policy update, regarding policies already applied to the network. Therefore, based on Consistent Policy Composition [6], we define our two sub-problems: request serialization and policy composition.

The request serialization sub-problem consists of defining a consistent global order to commit all policy updates. Let H be the history, that is, the set of all events that happen in the network. Let $<_H$ be a partial order relation on the events of history H. A request req precedes another request req' in the history H, represented by $req <_H req'$, if the response to req appears before the call to req' in H. If two requests are not related to precedence order, it is said that the requests are concomitants. Similarly, a networking event ev, such as a packet input, precedes a request req in H, represented by $ev <_H req$, if ev appears before the call of req in H. Besides, ev succeeds req in H, $req <_H ev$, if ev occurs after the response of req. An event ev is concurrent with a request req if $ev \not<_H req$ and $req \not<_H ev$. History H is sequential if it does exist neither two concurrent requests nor a concurrent packet input event with a request.

Let H_{c_i} be the local history of controller c_i, a subsequence of H, which consists of all events of c_i. Locally, we verify that all history H_{c_i} is sequential, hence, a controller accepts a new request if, and only if, there is no previous request without response. A history H is consistent if the precedence relation between two requests in the local history of a controller is the same into any local history of other controllers. Hence, a consistent update keeps

$$req <_{H_{c_i}} req' \rightarrow req <_{H_{c_j}} req' \mid \forall c_i, c_j \in C, \tag{1}$$

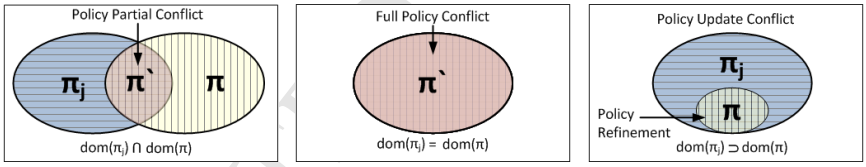
where C is the set of all network controllers. Therefore, we consider that the union of partial local orders of all controllers is consistent with the global order in history H.

From now on, we have defined a global order between all policy update requests on the network. Nonetheless, it is still necessary to correctly define the composition between new and currently applied policies. To address the composition of policy updates, we consider that H has to respect the following properties [6]:

- A policy is successfully accepted to be added to H if, and only if, it does not conflict with any other policy previously applied in H,
- For each packet input event ev in the network, the behavior of the network is consistent with the composition of all successfully applied policies that precede the event $ev \in H$.

It is noteworthy that conflict between policies arises when two policies act on an overlapping set of flows [26, 27]. Conflict-free policies are the ones that present entirely disjoint domains[4]. Let π be a policy update, Π be the set of all policies already installed on the network, and $dom(\pi)$ be the domain of the policy π. Then, a non-conflicting policy respects

$$dom(\pi) \cap dom(\pi_j) = \emptyset, \forall\ \pi_j \in \Pi. \tag{2}$$



(a) Partial conflict between π and π_j.

(b) Full conflict between π and π_j.

(c) Update conflict between π and π_j.

Figure 2: Possible cases of policy update conflicts. A new policy π conflicts with an already applied policy π_j into a flowspace S. a) A partial conflict solution leads to the creation of a policy subset π'. b) A full conflict solution creates the new composed policy π'. c) π is a refinement of policy π_j or an explicitly new version of the old policy. The update conflict is solved applying π on its domain.

Therefore, we identify three types of conflict: partial conflict, full conflict, and update conflict. The partial conflict is when the domain of a policy overlaps the domain of another. In a generic case, the composition of the two policies may lead to the creation of packet handling rules for each subset, as exemplified in Figure 2(a). It is worth noting a Partial Conflict may also arise when

[4]The policy domain is the set of flows that the policy rules. The policy domain is defined based on match fields [3, 26].

$dom(\pi) \supset dom(\pi_j)$, as it leads to a specific subset of rules that should consider both policies. The full conflict occurs when two different policies have the same domain. In this case, shown in Figure 2(b), either the new policy overwrites the previous one, or the new policy is completely rejected. The main difference between partial and full is the occurrence of a subset of composed rules in partial conflict, while in full conflict the final set of rules has the same cardinality than π or π_j. The third conflict possibility is the update conflict, Figure 2(c). Indeed, the update conflict occurs when a controller launches a new version of the previous policy and, thus, the new policy with the same domain should **explicitly overwrite** the previous one. In case of launching a refining policy, shown in Figure 2(c), where the policy domain is part of a previously installed policy, it is handled as an update conflict into the smaller domain. The main difference between the update conflict and the others is that an update conflict explicitly signs how the controllers should resolve the conflict, such as overwriting the previous rules or applying actions from both old and new rules. Thus, the network-policy update request carries a policy that resolves the conflict by itself.

Besides consistently accepting policy updates and composing them, the policy update procedure should apply the updates in a per-packet consistent way [3, 6]. Thus, a single network configuration has to handle an in-transit packet. As a per-packet update assures that no in-transit packet changes during the installation of the update request, it also guarantees that

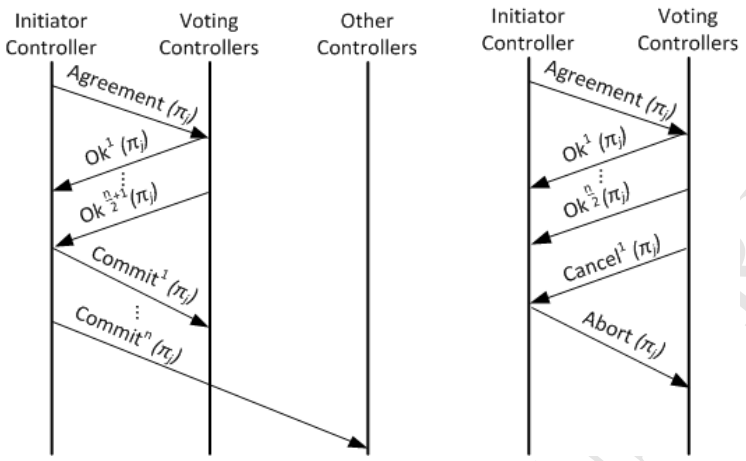
$$\nexists ev \in H | ev \not\prec_H req \; and \; req \not\prec_H ev, \tag{3}$$

, i.e., that no packet handled by the new network configuration while there is no termination of the update request.

4. The Consistency Protocol

We propose a consistency protocol for updating policies on a distributed control plane on Software-Defined Networking. Our proposal assures that policies concomitantly launched by different controllers are applied in the network in a global serialized order and are appropriately composed with previous policies. Moreover, we assume the Two-Phase Update [3], as the scheme of implementing the changes of the rules on the network. This assumption is essential to meet constraints of the packet-consistency model. The proposal for implementing Two-Phase update with distributed control uses `ReuseTag` algorithm. The principal advantage of our proposal, when compared to `ReuseTag` algorithm [6], is that we accomplished a consensus, running our consensus protocol instead of assuming the existence of an abstract consensus interface among all controllers. Furthermore, our proposal achieves a shorter time to converge to a correct order, applying a wait-free algorithm, at the cost of using global order number to tag packets.

The main idea of our proposal is to define a global order between all requests for policy updates. In fact, we consider three steps for the installation of a policy



(a) Correctly committing policy π_j.

(b) Aborting policy π_j installation.

Figure 3: The proposed consistency protocol. The initiator controller starts committing a policy π_j on the network. a) At least $n/2 + 1$ controllers agree to install π_j. The initiator controller sends the commit message to all controllers on the network. b) At least one controller disagree on installing π_j. The initiator controller sends an abort message to all controllers that were voting on the agreement.

on the network. The first one is the policy update request issued by controllers. The second step is to launch the policy on the network, which consists of voting the policy order number and its applicability to the network. In case of acceptance of the policy update by a majority of controllers and the controllers agree on a global order number, it goes to the third step that is the commitment of the policy update. The commitment of a network-policy update is to install rules in the switches on the network, translating the high-level policy into packet-handling rules. Our proposal focuses on the first and the second steps, as we propose an eventual-consistency protocol and a conflict-check algorithm. We apply the Two-Phase Update to commit the new policy on the network because it provides a strong-consistency model for in-transit packet handling. The order number agreed between controllers is the tag that defines the new configuration on the Two-Phase Update. We argue that the combination of two consistency models enables the SDN to achieve a fast answer to the policy update request, as the consistent order requires a small number of nodes. While the correct packet forwarding is accomplished using a strong-consistency model, we assure that any packet in the network will never experience an unexpected state.

The proposed consistency protocol works as follows. A policy update request arrives at a controller[5]. Concomitant requests may arrive in other con-

[5]A request of policy update arrives on a controller by the northbound API or by east/westbound API. The former occurs when a control application dispatches a request for

trollers. The controller which receives the request starts the protocol, `initiator controller`, and sends an `agreement` message to $n/2+1$ controllers, where n is the total number of controllers on the network. It is worth to note that we propose to send a minimal number of messages $(n/2+1)$ to obtain the majority consensus from the whole set of controllers. The fundamental idea is to obtain a fast consensus minimizing the response time from the controllers. Another advantage of our proposal is that it reduces the probability of a collusion attack because each choice of $n/2+1$ converges to a set of well-behaved nodes. The `agreement` contains the policy update request, as well as its application domain, and the most recent policy order number, which is the number of the last policy committed by the initiator controller plus one. When a controller receives an `agreement` message, it has two possible output messages: `ok` and `cancel`. If all $n/2+1$ controllers respond with an `ok`, the initiator controller sends a `commit` message to all controllers on the network, shown in Figure 3(a). Receiving a `commit` message, a controller verifies its order number, and if it is compatible with the next policy that the controller is waiting for, the controller installs the policy on all switches that it controls and updates its order number. If the `commit` message has a number higher than the one that it is waiting for, the controller synchronizes its policy database with the initiator controller. In case the initiator controller receives any `cancel` message with a higher order number, it aborts the policy update launching and synchronizes its policy database with the controller which sent the `cancel` message, shown in Figure 3(b).

During the voting step, a controller may issue a `cancel` message due to three reasons: (i) the `agreement` message signs an order number lower than its own order number; (ii) the received policy within the `agreement` message conflicts against its own set of applied policies; and (iii) it has already received an `agreement` message from other controller. We highlight that, in the first case, controller votes for canceling the consensus because the out-of-date `agreement` message shows that the initiator controller does not have the most updated policy database. In the second case, a controller cancels the consensus because it locally disagrees with the new policy due to a conflict. In the last case, a controller votes for canceling the consensus if there is another consensus running in the network for another policy request, i.e., the controller has already received an `agreement` message for other update request and it has not yet finished the commitment.

Voting against a consensus while the previous consensus hasn't yet finished assures the safety property to our proposal since we guarantee that there is just one consensus attempt which gathers $n/2+1$ votes to conclude. Besides, we introduce a timeout for each message. As the messages are not timely answered, the timeout expires, and controllers roll back the policy update request. This behavior assures liveness to our proposal, even in case of crash failures.

the controller. The latter happens when the controller receives an `agreement` message from another controller on the network.

Implementation Details. As the proposed protocol shares some mechanisms with the two-phase commit protocol, some implementation details should be highlighted to ensure the deadlock-free feature of the proposal. Firstly, we rely on TCP connections between all controllers. In this sense, the controllers act as a peer-to-peer system, creating an overlay network for control. Controllers keep connection among all of them. Thus, each controller is always aware of which controllers are online, because of the state of the TCP connection. Secondly, we associate a timeout for each message sent by the proposed protocol. If a controller does not respond to an `agreement` message before a timeout expiration event, the initiator controller sends a new `agreement` message to another controller on the network. It is worthy to note that we also add a cryptographic signature in each message sent between controllers. Each controller has its certificate; their pair of public and private key; and Secure Socket Layer (SSL) protocol protects all connections between controllers.

Running the protocol. As depicted in Figure 3(a), if $n/2+1$ controllers agree to commit the policy update, the information is broadcasted to all controllers, and the policy update is committed on the network switches according to the Two-Phase Update scheme [3]. Nevertheless, failures may arise while running our proposed protocol. We identify four cases that have to be handled by the protocol: more than one controller start the protocol at the same time; the initiator controller fails after sending the `agreement` message for any node; a node fails after receiving the `agreement` message; and a node identifies that its policy-order number is outdated.

Case 1: More than one initiator. If two or more controllers concomitantly initiate the protocol, just one of them can adequately achieve $n/2 + 1$ confirmation messages. In case of more than two initiators, none of them may obtain the necessary number of votes and all of them may also receive a `cancel` message, in which shows that there is another transaction occurring. In this case, each controller, which receives a `cancel` message because of concurrent transactions, will wait for a random time until trying to re-launch its update request. It applies an exponential back-off algorithm which exponentially increases the time between re-launching the update requests.

Case 2: Initiator fails. If the initiator controller fails before sending any `agreement` message, there is no damage to the current state of the protocol, because any transaction hasn't started. On the other hand, if it fails after sending any `agreement` message, a controller that has received the former message, resends it, adding its signature into the message, and tags it as a recovery message. The new initiator controller sends the re-signed `agreement` message to a new group of $n/2 + 1$ controllers. In case the policy update has already been committed, the new initiator updates its policy database, commits the policy to its switches, and broadcasts the information. Otherwise, it follows the protocol as if it was the initiator of the request.

Case 3: A controller fails. There are two different ways of handling the case of a controller failure. First, if the controller fails after receiving an `agreement` message, the initiator controller waits for its answer till a timeout or, if the failure breaks the TCP connection, the initiator immediately detects the connection loss. Therefore, the initiator sends a new `agreement` message to other controllers till it achieves $n/2+1$ votes, all of them `ok` messages, or at least one `cancel` message. The second possible case is when the controller is not taking part in any agreement between controllers. In this case, the failure is ignored. When a controller recovers from a failure, it updates its policy database with any other controller on the network.

Case 4: A controller is outdated. A controller knows that it is out-of-date when it receives a message with the policy order number higher than the one it is expecting to. The message may be `agreement`, `cancel`, or `commit`. When it receives an `agreement` or `commit` message, the outdated controller asks the updated controller, who sent the message, the most recent policy database. In the case of a `cancel` message, it aborts the policy launching step and, after, updates its database with the controller who sent the message with the most recent policy order number.

The synchronization procedure takes place as soon as a controller detects that it is out-of-date. The procedure is simple. The controller asks $n/2+1$ nodes about the most recent network version. As the nodes answer it, the controller computes the votes for each version. If there is more than one version among the answers, the controller asks more randomly-chosen nodes until it achieves $n/2$ equal responses for the version order number. Then, the controller chooses one node among all the ones that answered this version number, and synchronize its policy database.

We assume some implementation facilities, as the use of TCP connections that keep the state of each connect controller and assures the reliability and the ordering of the messages. Thus, our proposal also assures a higher availability than the two-phase transaction-commit protocol, although we still achieve the policy commitment in two round-trip times (RTT).

We establish a trade-off between increasing tag complexity and complexity of the consistency. Tag complexity stands for upper bounding on the policy order number, like the one stated at `ReuseTag` [6]. In our work, we perform a lightweight consistency protocol at the cost of considering the policy order number a cyclical counter at $2n$ order, while `ReuseTag` uses only $f+2$ tags, where f is the number of controllers that may fail, at the cost of relying on an abstract consensus interface, which is not explicit nor its complexity is discussed.

The policy composition algorithm. Our composition algorithm always works locally. The main idea of our algorithm is to search for any conflict (full, partial or update conflict) that may arise from a new policy update request π and the set of already defined policies Π. In this way, as a new policy update request arrives on a controller, the algorithm locally checks if the domain of the request, $dom(\pi)$, conflicts with the domain of all other policies,

$\cup\, dom(\pi_i) | \forall \pi_i \in \Pi$. A policy update request may be issued by the controller itself or may arrive as an `agreement` message. In both cases, if the algorithm realizes that it is a partial conflict or a full conflict, it refuses the policy update request. Therefore, if the request has arrived as an `agreement` message, the controller, who rejects it, responds with a `cancel` message, signaling the conflict. If the policy is conflict-free, defined by Equation 2, or an explicit update, the algorithm accepts the new policy and, thus, the policy is installed, as it is, on the controller running the proposed composition algorithm. Therefore, our proposal implements an all-or-nothing interface that completely accepts the new policy or completely refuses it.

Algorithm 1: Policy composition algorithm. The algorithm locally runs on each controller. The algorithm output is the vote of the controller, for or against the installation of the policy update request.

Input : π_i %policy update request
Π %set of installed policies
vote := OK %there is no conflict.
for $\pi_j \in \Pi$ **do**
  if $dom(\pi_i) \cap dom(\pi_j) \neq \emptyset$ ***and*** $isUpdate(\pi_i, \pi_j) = False$ **then**
    vote := Cancel %conflicting policies were found.
  end
end
Output: vote

The Algorithm 1 checks whether the policy update domain overlaps the already installed policy domains. However, the algorithm can be applied, without loss of generality nor correctness, adopting more complex methods of conflict identification and more elaborated composition rules, such as the policy composition defined on `Pyretic` language [28], or the verification of network invariants from `VeriFlow` [16].

Distributed Update Scheme. The network-configuration update is simple, and it performs a generalization of the Two-Phase update [3] for a distributed control plane. Therefore, as soon as a controller receives a policy update request, it starts the consistency protocol and agrees with the other controllers on the order of policy installation, and on the consistent network configuration. Then, the controllers are ready to start the policy installation. The initiator controller performs the coordination between the two phases of the updating scheme. Hence, after committing the update request, all controllers install the new network configuration on the core ports of the network[6]. After this first phase, all controllers report to the initiator controller at the end of the first

[6]Discovering which are core and edge ports is a straightforward task for a controller, as all controllers keep an updated topology of the network on the global network view.

phase. When the initiator receives all confirmations, it sends the permission to all controllers start the second phase, i.e., changing the flow rules on the ingress ports to tag the packets with the new network configuration version. This procedure also follows the failure-recovery routines of the proposed consistency protocol.

Proof of correctness. To prove the correctness and the safety of our proposal, we must prove that the policy commitment is both able to serialize and to launch policies that do not conflict. Thus, first, we prove by contradiction that the global order of policy installation, defined between all controllers, is the same as the local order of any controller. After that, we prove using the induction mechanism that the composed policies are consistent and, thus, there is no conflict between policies.

Theorem 1: The policy-installation local order of any controller is compatible with the global order.

Proof by Contradiction. We assume that the local policy-installation order of any controller is not compatible with the global order. Therefore, we assume that there is a controller c_i, in which the policy π_2 has precedence over policy π_1, $\pi_2 <_{Hc_i} \pi_1$, and, in the global order, π_1 precedes π_2, $\pi_1 <_H \pi_2$. As any controller, other than c_i, is compatible with the global order, we also have a controller c_j, in which the local order is $\pi_1 <_{Hc_j} \pi_2$. As c_i and c_j correctly run the proposed protocol, to instal a policy π_2 before installing π_1, the controller c_i had to have at least $n/2 + 1$ votes of other controllers (`ok` messages), as well as the other controllers had to have $n/2 + 1$ votes to install π_1 before π_2. According to the protocol, a controller cannot vote to contradictory orders. Then, the only way of having a different local order between two controllers is to have different votes for each order. In this way, it would be necessary $(n/2 + 1) + (n/2 + 1) = n + 2$ votes. Hence, as the network has n controllers at most, it is an impossible scenario, and it proves that having a different local order when comparing to the global order, is a logical contradiction. Thus, the proposed protocol achieves a global installation order of policies. ■

Theorem 2: The policy composition is conflict-free.

Proof by induction. We use the induction mechanism over Π, which represents the set of all installed policies on the network.

`Base Case` ($dom(\Pi_0) = \emptyset$)`:` In this case, the set of Π is empty, then it is trivially conflict-free.

`Hypothesis` ($dom(\Pi_i) = \cup_{k=1}^{i} dom(\pi_k)$)`:` Let Π_i be the set of all installed policies till π_i, for $i > 0$. Then, the domain of Π_i is defined as the union of the domain of all policies in it. By hypothesis, the composition of all policies in Π_i is consistent.

`Step` ($dom(\Pi_{i+1}) = dom(\Pi_i) \cup dom(\pi_{i+1})$)`:` We consider that all policies in Π_i are already consistently composed because of our induction hypothesis. Therefore, we have just to prove that the inclusion of the policy π_{i+1} into Π_i is conflict-free. We consider, thus, our policy composition algorithm. The algorithm runs locally, and its return is a requirement for a controller voting in favor of the installation of a policy. Then, adding π_{i+1} to the set Π_i is only possible if, and only if, $n/2 + 1$ controllers vote for it, assuring that there is no conflict between the new policy π_{i+1} and all policies in Π_i. According to the Theorem 1, at least one controller, among the $n/2 + 1$ controllers, has to have all policies already installed and has to be following the global order. If all of the $n/2 + 1$ controllers approve the new policy, it means that the new policy π_{i+1} has no conflict with any other policy that has ever been installed on the network. Otherwise, the policy π_{i+1} is completely rejected. Then, we prove Theorem 2, showing that the composition of Π_i with π_{i+1} is possible and it is conflict-free. ■

Byzantine Fault Tolerant Extension. The designed protocol assumes a scenario in which all controller nodes are trustworthy and reliable. In this scenario, the proposed protocol only tolerates crash-failures. Nevertheless, a natural extension is that a controller may behave arbitrarily, provoking unexpected protocol states [29, 30, 31], i.e., the nodes may present Byzantine failures. For these cases where nodes are Byzantine, we envision a Byzantine Fault Tolerant (BFT) extension for the consistency protocol. The extension implies a failure model in which nodes misbehave, like delaying reordering messages or sending wrong messages, or nodes can maliciously behave, as a result of an attack. If messages are delayed, a timeout mechanism triggers the message retransmission and changes the subset of controllers charged with voting for a policy update request. If a node is under attack and the attacker takes over of one or more network controllers, it can change the vote of these controllers, but it is not able to forge other identities to vote against the proposed protocol.

The key aspect of the BFT extension is that nodes should give evidence of their correct behavior. Thus, we introduce four major changes in the protocol mechanism: (i) requesting votes for all nodes; (ii) signing each `agreement` or `cancel` message; (iii) adding all votes to the `commit` message; and (iv) computing all votes to determine the result of the consensus.

Firstly, to achieve a BFT consistency, all nodes should be involved in the voting phase. Indeed, a BFT resistant protocol must contain $n \geq 3f + 1$ voting nodes, to tolerate f Byzantine faults among n nodes. Thus, the quorum for achieving the consensus is $n - f$ votes for the commitment of the new policy. It is worth noting that the quorum is always greater than $n/2$ votes. This first change avoids that one single Byzantine node always votes for the abortion of the policy requests, avoiding any policy to be committed. The second change in the original protocol is that each voting node has to sign all `agreement` or `cancel` messages with its private key. The voting node signs the message, which contains its vote, and the policy request. Signing the `agreement`, or `cancel`, a

node provides an irrefutable proof of its vote for each given request. Moreover, when sending the `commit`, the initiator controller must append all signed votes in the message. This approach enables the network controllers to locally check each vote to verify if the initiator controller is lying about the result of the consensus protocol. This approach enlarges the `commit` message. An alternative approach to keep minimum the size of the `commit` message is to send just $n - f$ first messages that express the same position of voting controllers, either agreeing or disagreeing, on the policy request. At last, the fourth change in the protocol is to compute all votes and communicate all decisions on the network, even the abortion. Indeed, as we tolerate up to f faults, the threshold to assume the consensus is $C_t = arg_{max}(2f, \frac{n}{2} + f)$ and, thus, all votes should be computed till it achieves more than half of nodes' vote for a single position. After all, the result should be communicated to all nodes through a `commit` with all the computed votes.

We highlight that the proofs of Theorems 1 and 2 are still valid for the BFT extension, as the extension still keep the property of locally checking for conflicts, and it also keeps the idea of having a controller that knows all currently installed policies. The extension of Theorem 1 is trivial if we consider the cardinality of the voting controller group as $\frac{n}{2} + f \geq 2f$, and we extract f fault nodes from it. Hence, we recall the original Theorem 1. Therefore, as the Theorem 1 is valid, the Theorem 2 is also trivially extended for the BFT proposal, as any new committed policy is consistent according to Theorem 1. Nevertheless, as an eventual-consistency protocol, the liveness property of the BFT extension is hard to prove. Thus, we introduce a trade-off between a safe and lightweight consistency protocol, and the liveness constrain.

5. Proposal Evaluation

The proposal evaluation is three-fold. First of all, we analyze our proposed consistency protocol through a formal model. Second, we simulate our protocol to investigate the message overhead and to compare with other proposals. Third, we simulate our proposed scheme for distributed network-policy update and compare it with the Two-Phase Update.

Therefore, our first evaluation aims to assert the robustness of our protocol to malicious or misconfigured nodes in the network. We consider a malicious node as the one which votes against to the correct vote according to the conflict-check algorithm. We consider a p_b probability of a node being well behaved, i.e., the node does not crash, neither presents a Byzantine fault. Hence, we calculate the convergence probability of our consistency protocol that is expressed by

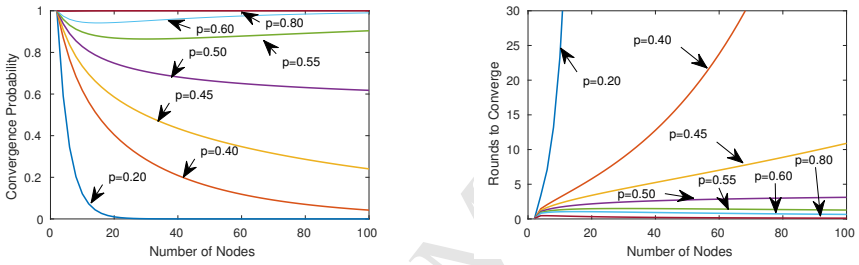
$$p^* = \sum_{i=0}^{i \leq \lfloor \frac{N}{2} \rfloor + 1} C_{N,i}\ p_b^i (1 - p_b)^{N-i}, \tag{4}$$

where p^* is the convergence probability of the consistency protocol, N is the total number of network nodes, and i is the number of malicious nodes in collusion

against the convergence of the protocol. It is worth noting that $C_{N,i}$ is the combination of i among the set of N nodes on the network. From the Equation 4, we can derive the probability of k consecutive failures of achieving the consensus. The probability is given by $p_{fail} = (1 - p^*)^k$. Moreover, we can define an arbitrary confidence interval threshold α that upper limits the probability of k failures. We assume the convergence of the protocol if the probability of k consecutive failures is lower than α and, thus, we assume that we reached the consensus. We estimate the number of rounds to achieve consensus is expressed by

$$k = \frac{log(\alpha)}{log(1 - p^*)}, \tag{5}$$

where α is an acceptable threshold of not reaching consensus after k rounds of our protocol.



(a) Protocol convergence probability.

(b) Rounds for the protocol convergence.

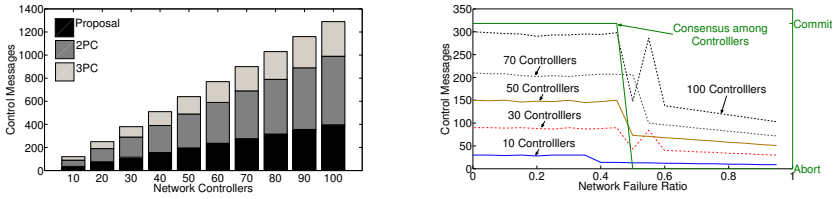
Figure 4: Convergence evaluation of our proposed consistency protocol. a) The probability of a network-policy update being accept (or refused) when it shouldn't be. For a $\boldsymbol{p}$ probability of nodes being well-behaved, b)The number of rounds to reach convergence in an acceptable threshold of $\boldsymbol{\alpha \leq 0.05}$. Let $\boldsymbol{p = 0.80}$, for more than 10 nodes; the protocol converges in just one round. When $\boldsymbol{p = 0.55}$, the proposal reaches convergence in one round for more than 60 nodes.

Figure 4 shows the convergence of the consistency protocol for p_b varying from 0.20 to 0.80. The value $p_b = 0.80$ represents that a node behaves well 80% of the time[7]. The borderline between the malicious and the well-behaved scenarios is shown in Figure 4(a) for $p_b = 0.50$, where half of the network nodes may have malicious behavior. We highlight that the bigger is the number of nodes in the network, the higher is the probability of reaching the convergence for the consistency protocol. Besides, we observe that the number of rounds to reach the consensus, Figure 4(b), still almost constant for $p_b \geq 0.50$, and $\alpha \leq 0.05$, as we increase the number of nodes in the network. Figure 4(b) also shows that the proposal tends to reach convergence in just one round of the protocol for a higher probability $p_b \geq 0.55$. Hence, the network reaction time for a policy update is upper bounded by the consensus algorithm that

[7]A good behavior is to follow the protocol and be available for the current consensus calculation, not taking part in any other voting.

is two times the round-trip time, as discussed previously. We also highlight that the proposed protocol is resilient to a collusion of up to $n/2 - 1$ nodes. Figure 4 shows, for $p = 0.50$, that the probability of achieving the convergence is approximatively 0.75 for 20 nodes, and reaches 0.95 $(1-\alpha)$ after 3 or 4 rounds of the protocol execution. This behavior is due to the random choice of the new voting nodes, as the previously chosen nodes fail to converge. The search for a new random set of voting nodes is uniform and may iterate over all nodes in the network, eventually finding the well-behaved ones.

The proposed consistency protocol was evaluated by simulating the consensus among the controller nodes of an SDN with distributed control. We developed the distributed-control extension of an SDN simulator [15], based on the SDN model proposed by Reitblatt *et al.* [3] and extended by Canini *et al.* [6].



(a) Number of exchanged messages for ten updates.

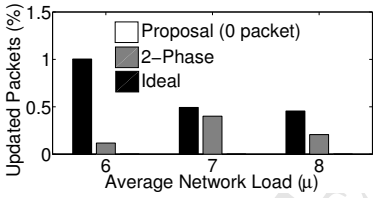
(b) Number of exchanged messages in a fail scenario.

Figure 5: Control-message load comparison between the proposed protocol (`Proposal`), the two-phase commit protocol (`2PC`) and the three-phase commit protocol (`3PC`). a) Control messages according to the number of controllers. The proposal reduces from 25% the number of messages, compared to `2PC`, when number of controllers is greater than 30. b) Number of control messages in function of the network-node failure percentage. The proposal maintains a low number of messages and even converges to a consensus when almost half the network nodes fail. `Commit` indicates the consensus by accepting the update and `Abort`, consensus for aborting the operation.

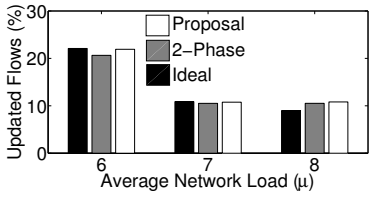
We implemented a prototype of the proposed updating mechanism to assess the load of messages exchanged between nodes. In this experiment, we compare our proposal with the two-phase commit protocol (2PC) and the three-phase commit protocol (3PC). Figure 5(a) compares the number of messages sent by the two-phase commit protocol (`2PC`), the three-phase commit protocol (`3PC`) and the our proposed consensus protocol (`Proposal`) for installing a policy on the network. The considered topologies are a complete mesh of 10 to 100 nodes, where all nodes are controllers. The results show that the number of messages sent by the proposed protocol is lower than that sent by the two-phase commit protocol. When considering 30 controllers, for instance, the proposed protocol reduces the number of control messages into up to 25%, when compared to the two-phase commit protocol, and up to 50%, compared with the three-phase commit protocol. Figure 5(a) shows that, when considering the wait-free protocols, three-phase commit and our proposal, when the number of controllers is greater or equal a 100 controllers, our proposal reduces up 66% the control-message

overhead on the network. Let n be the number of nodes that are running the protocol, the analysis of the behavior of each protocol indicates that the expected number of messages, in a fail-free scenario, for the `2PC` is $4 \times (n-1)$ messages, and for the `3PC`, $6 \times (n-1)$ messages. The proposed protocol, in turn, shows a maximum of $3n$ messages.

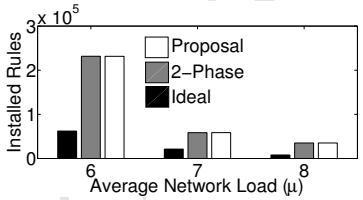
The following experiment assesses the resilience of the proposed protocol regarding the occurrence of node crash failures. Figure 5(b) shows the number of messages sent on the network when nodes fail. Nodes achieve consensus to commit the update request, even when the network failure rate is close to 50%, $n/2-1$ failed nodes. If most of the nodes fail, the requests are not accepted, then, they are aborted by the protocol. Figure 5(b) shows the low load of messages on the network even when transactions are aborted. When the proposal aborts the commitment of policies on the network, the number of messages is reduced and has little impact on the normal operation of the network. When failures occur, the proposed protocol sends new messages to random nodes until it exhausts the search for active nodes, or until it gets the required number of affirmative votes. However, this search can lead to the expiration of the timeout for a response of active controllers, who have already responded to the `agreement` message. For this reason, there are messaging peaks that impact scenarios where network failures are close to 50% of controller nodes, Figure 5(b).



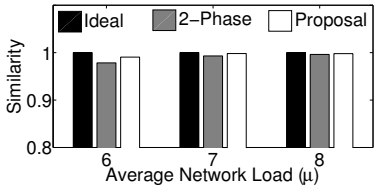
(a) Percentage of in-transit packets that are routed through two distinct settings.



(b) Percentage of flows that are routed through two distinct settings.



(c) Number of rules installed on the network.

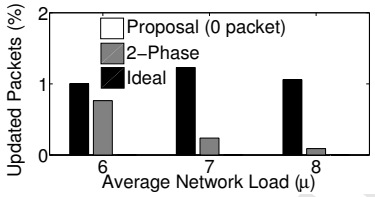


(d) Similarity with the Ideal Update.

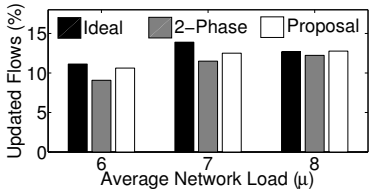
Figure 6: Results for the AT&T network topology. Impact of the policy updates on the network, according to the network load. Ideal and Two-Phase are centralized updating scheme. The proposed scheme coordinates the update procedure among all controllers.

In the second part of the proposal evaluation, we simulate two different real network topologies compliant with SDN. We adopt a simulation-based evaluation because simulating a network is agnostic about the SDN technology, while other evaluation techniques, prototyping or emulating SDN network with

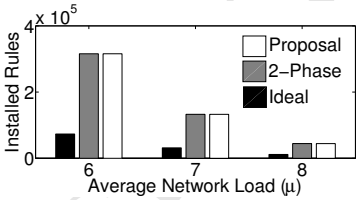
Mininet [32], are dependent on the underlying SDN technology, e.g., Mininet runs on top of OpenFlow. Our simulation runs on top of a discrete event simulator [27] based on the SDN model proposed by Reitblatt et al. [3]. The simulation is achieved by creating objects for switches, queues, links, packets, and controllers. At each simulation step, the packets on the input queue of each switch are checked against the switch's flow table. If no rule shows the next hop of the packet, the switch sends a packet arrival event to the controllers. As the controller responds to the packet arrival event, it installs a new rule on the switch's flow table, and the packet is moved to next switch input queue. The simulation mimics the SDN processing scheme. All simulated topologies are available at The Internet Topology Zoo[8]. The topologies are from the AT&T MPLS network in the U.S., with 25 nodes; and from RNP (Nation Research Network) in Brazil, with 30 nodes. The simulation parameters are defined as follows. The arrival of new flows is uniformly distributed between all nodes of the network and the flow inter-arrival rate follows a *lognormal* distribution, averaging from 6 to 8 ($\mu \in \{6, 7, 8\}$) and having standard deviation equals to 2 ($\sigma = 2$) [15]. The arrival of new flows happens during the early 900 simulation steps. We model each flow to last 50 steps, and the end of the simulation is determined when there are no packets or events to be handled.



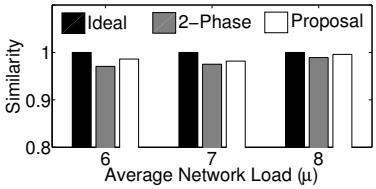
(a) Percentage of in-transit packets that are routed through two distinct settings.



(b) Percentage of flows that are routed through two distinct settings.



(c) Number of rules installed on the network.



(d) Similarity with the Ideal Update.

Figure 7: Results for the RNP network topology. Impact of the policy updates on the network, according to the network load.

The simulation of the distributed scheme was carried out by setting up a controller for each node, and all controllers run the consensus protocol. The consensus protocol agrees on the version of the network configuration and dis-

[8]Available at http://www.topology-zoo.org/.

tributes it among the other controllers. The update scheme with distributed control based on consensus protocol (`Proposal`), was compared with the centralized update scheme, Two-Phases Update (`2-Phase`), and with an optimal update scheme (`Ideal`). The Ideal Update is only feasible in a simulated scenario, in which we consider that the packet forwarding is interrupted during the update process of the forwarding rules in all switches. The ideal update is considered a proportional update scheme, i.e., the one in which the cost of updating the network is proportional to the implemented changes [3]. Thus, when comparing an update scheme with the Ideal Update, indeed, it compares how close the proposed scheme is to a proportional update. We simulate the two network topologies, the ATT&T network results are shown in Figure 6, and the results for the RNP network are shown in Figure 7. We simulate different traffic loads selecting different average inter-arrival intervals between flows. We highlight that our SDN simulation considers as rules the installed flows in the flow tables of the switches. In our simulations, policy updates are path changes and action changes (e.g., changing a VLAN tag on a flow).

Figure 6(a) and 7(a) compare the behavior of policy update schemes taking into consideration the in-transit packets, while the updates take place. The figures show the percentage of in-transit packets that are handled by more than one network configuration. The Ideal scheme updates all in-transit packets as soon as the update request arrives on the network. Therefore, the number of packets updated by the ideal scheme also shows the number of packets that are in-transit on the network when the updates happen. The proposed distributed scheme does not forward any packet over more than one network configuration version, which is the expected behavior of a per-packet consistent scheme. However, when applying a naïve implementation of the Two-Phase Update scheme, a small percentage of packets is forwarded over more than one network configuration. In these cases, the in-transit packets are handled by two different global network configurations. It happens because the considered controller model is the most naïve as possible, and after the update procedure it always forwards the packets with the newest network configuration without saving any state of flows. Thus, flows that have no installed rules in the core switches can reach an already updated controller and, from then on, they are forwarded by a newer network configuration [15]. The chosen controller model keeps the fairness of the evaluation. The fairness of this characterization recalls the controller model was the same for all the evaluated update schemes. The only difference between one scheme set up to others was the update scheme itself. None of the simulated controller stores states for flows.

The distributed controller update scheme acts more readily on the network than the Two-Phase Update with centralized control, which results closer to the Ideal result. As a consequence, the number of flows affected by updates in the network, shown in Figures 6(b) and 7(b), evidences the proposal is closer to the Ideal Update. The distributed scheme updates slightly more flows than the centralized Two-Phase update because of the parallel execution of the update of the tables on the core switches. The behavior of both schemes, however, is quite the same, as the proposed distributed scheme is an extension of the Two-Phase

update to multiple controllers.

Evaluating the percentage of in-transit packets and the number of updated flows on the network supports to infer the importance of a consistent scheme for updating the network. Although the in-transit packets represent 1% of the total number of all forwarded packets, as can be seen in Figures 6(a) and 7(a), representing from 10% to 20% of all flows on the network, shown in Figures 6(b) and 7(b). This observation indicates that up to 20% of the network flows may suffer from an unexpected behavior if an inconsistent reconfiguration is performed [33]. In case of TCP connections, an inconsistent policy update may take the peers to reset their connection, or even experience a short outage.

The total number of installed rules on the network switches is shown in Figures 6(c) and 7(c). This metric indicates how much of the memory of the switches are used for each update scheme. As the proposed distributed update scheme and the centralized Two-Phase update install rules in the network core to ensure per-packet consistency, the number of installed rules by these schemes is up to 4x higher than the Ideal update[9], in all simulated scenarios. It is worth mentioning that the number of the installed rules is a function of the number of flows, the duration of the flows and the frequency of policy updates. Therefore, long-lived flows in a frequent-updated scenario may lead to a high number of installed rules, because a new rule is installed for each flow at each new version of the network configuration.

To conclude, Figures 6(d) and 7(d) compare the result of the packet handling and forwarding at the destination of the packets. Indeed, we introduce a similarity index that measures the portion of forwarded packets by one update scheme, that is received at the destination, compared with the packets forwarded by the ideal update scheme. The similarity index measures how close the packet forwarding in each update scheme is to the ideal. This measure provides an estimator of the quality of each update scheme. As all considered update schemes are consistent, the similarity index shows that the Two-Phase scheme is already very close to the ideal performance. However, the proposed distributed scheme achieves an even better result than the Two-Phase update due to the efficient coordination of actions between the controllers using the proposed consistency protocol. This result is also an effect of the parallelization of the rule installation procedure on the core of the network and the lightweight consensus protocol. The proposed distributed update scheme quickly converges to a composed network configuration, and it benefits from the parallel installation of rules on the core switches by the multiple controllers. The proposal achieves an overall updating duration that is smaller than the Two-Phase update and, thus, our proposal reaches a packet handling that is closer to the ideal.

It is worth to note that the proposed policy update mechanism still depends

[9]There are flows that do not expire on switch flow tables and, thus, are affected by more than one update, generating a higher increase in the number of installed rules than the installation of just one more rule for each flow at each switch.

on packet tagging and it introduces a processing time and a small overhead per packet. Our proposal achieves the lightweight consistency protocol at the cost of using more tags on the network, referenced as tag complexity. Thus, in a hypothetical use case in which is used VLAN tagging, which implies a maximum of 2048 controllers on the network, it is added overhead of 16 bits per packet. As the tag complexity of the proposal is $2n$, in a network with 100 controllers, it is necessary to handle up to 200 tags, which represents a minimum of an 8 bit tag. The tagging overhead, however, in both cases, is less than 1% of the network throughput when considering the maximum transmission unit (MTU) of 1500 B. Another limitation of any proposal, which requires coordination among controllers to reach a consensus before applying the update, is that in a frequently updated network, the inter-arrival time of new policy update requests may become smaller than the time spent for running the consensus protocol and installing the policy on the network. Thus, in this case, wait queue of policy update requests increases and the time for installing a new policy might become unfeasible.

6. Conclusion

Achieving consistent network-policy updates in software-defined networking with distributed control plane is challenging. Controllers should agree in a global order to install policies on the network, and the composition of all policies should avoid conflict between new and already installed policies. In this paper, we proposed a consistency protocol that locally orders the policy installation according to an agreed global order. We also proposed a policy composition algorithm that takes advantage of an abstract consensus interface provided by our protocol. Our algorithm runs locally, and its interaction with the proposed consistency protocol assures that all accepted policies are conflict-free. We proved the correctness of our protocol and our algorithm through a formal model. The analysis of the number of protocol messages shows that our proposal achieves consistent updates into two round-trip times, without going through deadlock states even when up to $n/2 - 1$ controllers fail. We evaluated the probability of convergence of our distributed algorithm, and our results show that we achieve a consistent update after up to 4 rounds of the consensus protocol, even when the nodes have 0.50 probability of behaving well. Our simulations show that the proposed distributed update scheme achieves a per-packet consistent update. The proposal quickly converges and implements efficient coordination between the controller nodes. Our lightweight protocol reduces up to 66% of the number of messages to accept a policy update, while the proposed distributed update scheme behaves closer to the ideal update scheme than the Two-Phase update.

As future work, we intend to implement the proposed protocol as part of a policy update framework on top of SDN controllers to abstract for the running applications the above consistency mechanisms. Moreover, another future work is to extend the proposed protocol to coordinate controllers to apply policy updates in a consistent order to avoid the need of tagging every packet in the network.

Acknowledgment

This research is supported by INCT of the Future Internet, CNPq, CAPES, FAPERJ, and FAPESP.

[1] A. M. Medhat, T. Taleb, A. Elmangoush, G. A. Carella, S. Covaci, T. Magedanz, Service function chaining in next generation networks: State of the art and research challenges, IEEE Communications Magazine 55 (2) (2017) 216–223.

[2] J. H. Han, P. Mundkur, C. Rotsos, G. Antichi, N. Dave, A. Moore, P. Neumann, Blueswitch: enabling provably consistent configuration of network switches, in: Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on, 2015, pp. 17–27.

[3] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, D. Walker, Abstractions for network update, in: Proceedings of the ACM SIGCOMM 2012, ACM, New York, USA, 2012, pp. 323–334.

[4] D. Levin, A. Wundsam, B. Heller, N. Handigol, A. Feldmann, Logically centralized?: state distribution trade-offs in software defined networks, in: Proceedings of the First workshop on Hot topics in software defined networks, HotSDN'12, ACM, Helsinki, Finland, 2012.

[5] Y. Cui, S. Xiao, C. Liao, I. Stojmenovic, M. Li, Data centers as software defined networks: Traffic redundancy elimination with wireless cards at routers, IEEE Journal on Selected Areas in Communications 31 (12) (2013) 2658–2672.

[6] M. Canini, P. Kuznetsov, D. Levin, S. Schmid, et al., A distributed and robust SDN control plane for transactional network updates, in: The IEEE INFOCOM 2015, 2015.

[7] A. Akhunzada, A. Gani, N. B. Anuar, A. Abdelaziz, M. K. Khan, A. Hayat, S. U. Khan, Secure and dependable software defined networks, Journal of Network and Computer Applications 61 (2016) 199 – 221. doi:https://doi.org/10.1016/j.jnca.2015.11.012.

[8] S. Brandt, K.-T. Förster, R. Wattenhofer, On consistent migration of flows in SDNs, in: INFOCOM, 2016.

[9] Y. Zhang, L. Cui, W. Wang, Y. Zhang, A survey on software defined networking with multiple controllers, Journal of Network and Computer Applications 103 (2018) 101 – 118. doi:https://doi.org/10.1016/j.jnca.2017.11.015.

[10] C. L. I, S. Han, Z. Xu, S. Wang, Q. Sun, Y. Chen, New paradigm of 5G wireless internet, IEEE Journal on Selected Areas in Communications 34 (3) (2016) 474–482. doi:10.1109/JSAC.2016.2525739.

[11] T. Taleb, A. Ksentini, B. Sericola, On service resilience in cloud-native 5G mobile systems, IEEE Journal on Selected Areas in Communications 34 (3) (2016) 483–496.

[12] R. McGeer, A safe, efficient update protocol for openflow networks, in: ACM SIGCOMM - HotSDN'12, ACM, Helsinki, Finland, 2012.

[13] N. P. Katta, J. Rexford, D. Walker, Incremental consistent updates, in: ACM SIGCOMM - HotSDN'13, ACM, Hong Kong, China, 2013.

[14] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, S. Shenker, SCL: Simplifying distributed SDN control planes, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), USENIX Association, Boston, MA, 2017, pp. 329–345.

[15] D. M. F. Mattos, O. C. M. B. Duarte, G. Pujolle, Reverse update: A consistent policy update scheme for software-defined networking, IEEE Communications Letters 20 (5) (2016) 886–889.

[16] A. Khurshid, X. Zou, W. Zhou, M. Caesar, P. B. Godfrey, VeriFlow: Verifying network-wide invariants in real time, in: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), USENIX, Lombard, IL, 2013, pp. 15–27.

[17] S. Vissicchio, L. Cittadini, FLIP the (flow) table: Fast lightweight policy-preserving SDN updates, in: INFOCOM, 2016, selected as the Best Paper Award Runner-up.

[18] J. McClurg, H. Hojjat, P. Cerny, N. Foster, Efficient synthesis of network updates, in: ACM SIGPLAN - PLDI, ACM, Portland, USA, 2015.

[19] M. Kablan, A. Alsudais, E. Keller, F. Le, Stateless network functions: Breaking the tight coupling of state and processing, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), USENIX Association, Boston, MA, 2017, pp. 97–112.

[20] J. Gray, L. Lamport, Consensus on transaction commit, ACM Trans. Database Syst. 31 (1) (2006) 133–160.

[21] R. D. Prisco, B. Lampson, N. Lynch, Revisiting the Paxos algorithm, Theoretical Computer Science 243 (1-2) (2000) 35 – 91.

[22] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, R. Soulé, Netpaxos: Consensus at network speed, in: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, ACM, Santa Clara, California, 2015, pp. 5:1–5:7.

[23] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm, in: 2014 USENIX Annual Technical Conference (USENIX ATC 14), USENIX Association, Philadelphia, PA, 2014, pp. 305–319.

[24] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, D. Mazières, Millions of little minions: Using packets for low latency network programming and visibility, in: Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, ACM, New York, NY, USA, 2014, pp. 3–14.

[25] T. D. Nguyen, M. Chiesa, M. Canini, Towards decentralized fast consistent updates, in: Proceedings of the 2016 Applied Networking Research Workshop, ACM, 2016, pp. 19–25.

[26] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, S. Krishnamurthi, Hierarchical policies for software defined networks, in: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN'12, ACM, New York, NY, USA, 2012, pp. 37–42.

[27] S. Luo, H. Yu, L. Li, Consistency is not easy: How to use two-phase update for wildcard rules?, Communications Letters, IEEE 19 (3) (2015) 347–350.

[28] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, et al., Composing software defined networks., in: 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13), USENIX Association, Berkeley, CA, USA, 2013, pp. 1–13.

[29] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, P. Verissimo, Efficient byzantine fault-tolerance, IEEE Transactions on Computers 62 (1) (2013) 16–30.

[30] M. Castro, B. Liskov, Practical byzantine fault tolerance, in: Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI '99, 1999, pp. 173–186.

[31] A. Bessani, J. Sousa, E. E. P. Alchieri, State machine replication for the masses with BFT-SMART, in: Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '14, 2014, pp. 355–362.

[32] B. Lantz, B. Heller, N. McKeown, A network in a laptop: rapid prototyping for software-defined networks, in: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets-IX, ACM, New York, NY, USA, 2010, pp. 19:1–19:6.

[33] S. Paris, A. Destounis, L. Maggi, G. Paschos, J. Leguay, Controlling flow reconfigurations in SDN, in: INFOCOM, 2016.

Diogo Menezes Ferrazani Mattos is currently a Professor at the Universidade Federal Fluminense (Niterói, Brazil). He received his degree of D. Sc. in Electrical Engineering from Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brazil, in 2017. Between 2015 and 2016, he had a sandwich scholarship to work on his PhD Thesis on the LIP6 (Laboratoire d'Informatique de Paris 6) at Université Pierre et Marie Curie, Paris, France. He obtained a Master's degree in Electrical Engineering from Universidade Federal do Rio de Janeiro, in 2012. He received a Computer and Information Engineer degree from the same university, in 2010 with a GPA of 9.3 over 10. His interests are in network security, new generation networking, network virtualization, software-defined networking, and Internet of the Future.

Otto Carlos M. B. Duarte received the Eletronic Engineer degree and the M. Sc. degree in Electrical Engineering from Universidade Federal do Rio de Janeiro, Brazil, in 1976 and 1981, respectively, and the Dr. Ing. degree from ENST/Paris, France, in 1985. Since 1978 he is Professor at Universidade Federal do Rio de Janeiro. In 1992/1993 he has worked at Paris 6 University, in 1995, at International Com- puter Science Institute (ICSI) associated to the University of California at Berkeley, and in 2014 at University of California at Berkeley, He worked several times as invited professor at Paris 6 University.

Guy Pujolle is currently a Professor at the Pierre et Marie Curie University (Paris 6), a member of the Institut Universitaire de France. He is an editor for International Journal of Network Management, WINET, Telecommunication Systems and Editor-In-Chief of the indexed Journal Annals of Telecommunications. He was an editor for Computer Networks, Operations Research, Editor-In-Chief of Networking and Information Systems Journal, Ad Hoc Journal and several other journals. Guy Pujolle is a pioneer in high-speed networking having led the development of the first Gb/s network to be tested in 1980.