Programming SDN: A Transactional Approach

Marco Canini[1] Petr Kuznetsov[2] Dan Levin[3] Stefan Schmid[4]

[1] Université catholique de Louvain, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium
marco.canini@uclouvain.be

[2] Télécom ParisTech, 46 Rue Barrault, 75013 Paris, France
petr.kuznetsov@telecom-paristech.fr

[3] TU Berlin, Marchstr. 23, 10587 Berlin, Germany
dlevin@inet.tu-berlin.de

[4] TU Berlin & T-Labs, Ernst-Reuter Platz 7, 10587 Berlin, Germany
stefan.schmid@tu-berlin.de

Abstract

Computer networking currently goes through a transition phase: the paradigm of Software-Defined Networking (SDN) is discussed intensively, both in the industry and in the academia. In a nutshell, SDN out-sources the control over the network to a logically centralized software, called the *control plane*. The ability of the control plane to "program" the network (the *data plane*) opens new interesting opportunities to network operators and system designers.

[Regular paper only]

1 Introduction

The emerging paradigm of Software-Defined Networking (SDN) promises to simplify network management and enable building networks that meet specific, end-to-end requirements. In SDN, the *control plane* (a collection of network-attached servers) maintains control over the so-called *data plane* (the packet-forwarding functionality implemented on switching hardware). Control applications operate on a global, logically-centralized network view, which introduces opportunities for network-wide management and optimization. This view enables simplified programming models to define a high-level network policy, *i.e.*, the intended operational behavior of the network encoded as a collection of *forwarding rules* that the data plane must respect.

While the notion of centralized control lies at the heart of SDN, implementing it on a centralized controller does not provide the required levels of availability, responsiveness and scalability. How to realize a robust, distributed control plane is one of the main open problems in SDN and to solve it we must deal with fundamental trade-offs between different consistency models, system availability and performance. Implementing a resilient control plane becomes therefore a distributed-computing problem that requires reasoning about interactions and concurrency between the controllers while preserving correct operation of the data plane.

In this paper, as a case study, we consider the problem of consistent installation of network-policy *updates* (*i.e.*, collections of state modifications spanning one or more switches), one of the main tasks any network control plane must support. We consider a multi-authorship setting [8] where multiple administrators, control applications, or end-host applications may want to modify the network policy independently at the same time, and where a conflict-free installation must be found.

We assume that we are provided with a procedure to assemble sequentially arriving policy updates in one (semantically sound) *composed* policy (*e.g.*, using the formalism of [1]). Therefore, we address here the challenge of composing *concurrent* updates, while preserving a property known as *per-packet consistency* [18]. Informally, we must guarantee that every packet traversing the network must be processed by exactly one global network policy, even throughout the interval during which the policy is updated — in this case, each packet is processed either using the policy in place prior to the update, or the policy in place after the update completes, but never a mixture of the two. At the same time, we need to resolve conflicts among policy updates that cannot be composed in a sequential execution. We do this by allowing some of the requests to be *rejected*, but requiring that no data packet is affected by a rejected update.

Our first contribution is a formal model of SDN under fault-prone, concurrent control. We then focus on the problem of *per-packet consistent updates* [18], and introduce the abstraction of *Consistent Policy Composition (CPC)*, which offers a *transactional* interface to address the issue of conflicting policy updates. We believe that the CPC abstraction, inspired by the popular paradigm of software transactional memory (STM) [19], exactly matches the desired behavior from the network operator's perspective, since it captures the intuition of a correct sequential composition combined with optimistic application of policy updates.

We then discuss different protocols to solve the CPC problem. We present a *wait-free* CPC algorithm, called FixTag, which allows the controllers to directly apply their updates on the data plane and resolve conflicts as they progress installing the updates. While FixTag tolerates any number of faulty controllers and does not require them to be strongly synchronized (thus improving concurrency of updates), it incurs a linear *tag complexity* in the number of to-be-installed policies (and hence in the worst-case exponential in the network size). We then present a more sophisticated

protocol called ReuseTag, which applies the replicated state-machine approach to implement a total order on to-be-installed policy updates. Assuming that at most f controllers can fail, we show that ReuseTag achieves an optimal tag complexity $f + 2$.

To the best of our knowledge, this work initiates an analytical study of a *distributed* and *fault-tolerant* SDN control plane. We keep our model intentionally simple and we focus on a restricted class of forwarding policies, which was sufficient to highlight intriguing connections between our SDN model and conventional distributed-computing models, in particular, STM [19]. One can view the SDN data plane as a shared-memory data structure, and the controllers can be seen as read/write processes, modifying the forwarding rules applied to packets at each switch. The traces of packets constituting the data-plane workload can be seen as "read-only" transactions, reading the forwarding rules at a certain switch in order to "decide" which switch state to read next. Interestingly, since in-flight packets cannot be dropped (if it is not intended to do so) or delayed, these read-only transactions must always commit, in contrast with policy update transactions.

In general, we believe that our work can inform the networking community about what can and cannot be achieved in a distributed control plane. We also derive a minimal requirement on the SDN model without which CPC is impossible to solve. From the distributed-computing perspective, we show that the SDN model exhibits concurrency phenomena not yet observed in classical distributed systems. For example, even if the controllers can synchronize their actions using consensus [10], complex interleavings between the controllers' actions and packet-processing events prevent them from implementing CPC with constant tag complexity (achievable using one reliable controller).

Roadmap. We introduce our SDN model in Section 2. Section 3 formulates the CPC problem and Section 4 describes our CPC solutions and their complexity bounds. We discuss related work in Section 5 and conclude in Section 6. Proof sketches are given in the Appendix.

2 Distributed Control Plane Model

We consider a setting where different users (*i.e.*, policy authors or administrators) can issue policy update requests to the distributed SDN control plane. We now introduce our SDN model as well as the policy concept in more detail.

Control plane. The distributed *control plane* is modelled as a set of $n \geq 2$ *controllers*, $p_1, \ldots, p_n$. The controllers are subject to *crash* failures: a faulty controller stops taking steps of its algorithm. The controller that never crashes is called *correct* and we assume that there is at least one correct controller. We assume that controllers can communicate among themselves (*e.g.*, through an out-of-band management network) in a reliable but asynchronous (and not necessarily FIFO) fashion, using message-passing. Moreover, the controllers have access to a consensus abstraction [9] that allows them to implement, in a fault-tolerant manner, any replicated state machine, provided its sequential specification [10]. The consensus abstraction can be obtained, *e.g.*, assuming the eventually synchronous communication [7] or the *eventual leader* Ω failure detector [5] shared by the controllers, assuming a majority of correct controllers or the *quorum* failure detector Σ [6].

Data plane. Following [18], we model the *network data plane* as a set P of *ports* and a set $L \subseteq P \times P$ of directed *links*. As in [18], a hardware switch is represented as a set of ports, and a physical bi-directional link between two switches A and B is represented as a set of *directional* links, where each port of A is connected to the port of B facing A and every port of B is connected to the port of A facing B. We additionally assume that P contains two distinct ports, World and Drop, which represent forwarding a packet to the outside of the network (*e.g.*, to an end-host or upstream provider) and dropping the packet, respectively. A port $i \notin \{\mathsf{World}, \mathsf{Drop}\}$ that has no

incoming links, *i.e.*, $\nexists j \in P$: $(j,i) \in L$ is called *ingress*, otherwise the port is called *internal*. Every internal port is connected to Drop (can drop packets). A subset of ports are connected to World (can forward packets to the outside of the network). World and Drop have no outgoing links: $\forall i \in \{\mathsf{World}, \mathsf{Drop}\}$, $\nexists j \in P$: $(i,j) \in L$.

The workload on the data plane consists of a set Π of *packets*. (To distinguish control-plane from data-plane communication, we reserve the term *message* for a communication involving at least one controller.) In general, we will use the term *packet* canonically as a type [18], *e.g.*, describing all packets (the packet *instances* or *copies*) matching a certain header; when clear from the context, we do not explicitly distinguish between packet types and packet instances.

Port queues and switch functions. The *state* of the network is characterized by a *port queue* Q_i and a *switch function* S_i associated with every port i. A port queue Q_i is a sequence of packets that are, intuitively, waiting to be processed at port i. A switch function is a map $S_i : \Pi \to \Pi \times P$, that, intuitively, defines how packets in the port queue Q_i are to be processed. When a packet pk is fetched from port queue Q_i, the corresponding *located packet*, *i.e.*, a pair $(pk', j) = S_i(pk)$ is computed and the packet pk' is placed to the queue Q_j.

We represent the switch function at port i, S_i, as a collection of *rules*. Operationally, a rule consists of a pattern matching on packet header fields and actions such as forwarding, dropping or modifying the packets. We model a rule r as a partial map $r : \Pi \to \Pi \times P$ that, for each packet pk in its domain $dom(r)$, generates a new located packet $r(pk) = (pk', j)$, which results in pk' put in queue Q_j such that $(i,j) \in L$. Disambiguation between rules that have overlapping domains is achieved through priority levels, as discussed below. We assume that every rule matches on a header field called the *tag*, which therefore identifies which rules apply to a given packet. We also assume that the tag is the only part of a packet that can be modified by a rule.

Port operations. We assume that a port supports an *atomic* execution of a *read*, *modify-rule* and *write* operation: the rules of a port can be atomically read and, depending on the read rules, modified and written back to the port. Formally, a port i supports the operation: $update(i, g)$, where g is a function defined on the sets of rules. The operation atomically reads the state of the port, and then, depending on the state, uses g to update it and return a response. For example, g may involve adding a new forwarding rule or a rule that puts a new tag τ into the headers of all incoming packets.

Policies and policy composition. Finally we are ready to define the fundamental notion of network policy. A *policy* π is defined by a *domain* $dom(\pi) \subseteq \Pi$, a *priority level* $pr(\pi) \in \mathbb{N}$, and a unique *forwarding path*, *i.e.*, a loop-free sequence of piecewise connected ports, for each ingress port that should apply to the packets in its domain $dom(\pi)$. More precisely, for each ingress port i and each packet $pk \in dom(\pi)$ arriving at port i, π specifies a sequence of distinct ports $i_1, \ldots, i_s$ that pk should follow, where $i_1 = i$, $\forall j = 1, \ldots, s-1$, $(i_j, i_{j+1}) \in L$ and $i_s \in \{\mathsf{World}, \mathsf{Drop}\}$.

The last condition means that each packet following the path eventually leaves the network or is dropped.

We call two policies π and π' *independent* if $dom(\pi) \cap dom(\pi') = \emptyset$. Two policies π and π' *conflict* if they are not independent and $pr(\pi) = pr(\pi')$. Now a set U of policies is *conflict-free* if no two policies in U conflict. Intuitively, the priority levels are used to establish the order in between non-conflicting policies with overlapping domains: a packet $pk \in dom(\pi) \cap dom(\pi')$, where $pr(\pi) > pr(\pi')$, is processed by policy π.

Conflict-free policies in a set U can therefore be *composed*: a packet arriving at a port is applied the highest priority policy $\pi \in U$ such that $pk \in dom(\pi)$.

Modelling traffic. The traffic workload on our system is modelled using *inject* and *forward* events defined as follows:

- *inject*(pk, j): the environment injects a packet pk to an ingress port j by adding pk to the end of queue Q_j, *i.e.*, replacing Q_j with $Q_j \cdot pk$.
- *forward*(pk, j, pk', k), $j \in P$: the first packet in Q_j is processed according to S_j, *i.e.*, if $Q_j = pk.Q'$, then Q_j is replaced with Q' and Q_k is replaced with $Q_k \cdot pk'$, where $r(pk) = (pk', k)$ and r is the highest-priority rule in S_j that can be applied to pk.

Algorithms, histories, and problems. Each controller p_i is assigned with an *algorithm*, *i.e.*, a state machine that (i) accepts invocations of high-level operations, (ii) accesses ports with *read-modify-write* operations, (iii) communicates with other controllers, and (iv) produces high-level responses. The distributed algorithm generates a sequence of *executions* consisting of port accesses, invocations, responses, and packet forward events. Given an execution of an algorithm, a *history* is the sequence of externally observable events, *i.e.*, *inject* and *forward* events, as well as invocations and responses of controllers' operations.

We assume an asynchronous *fair* scheduler and *reliable* communication channels between the controllers: in every infinite execution, no packet starves in a port queue without being served by a *forward* event, and every message sent to a controller is eventually received.

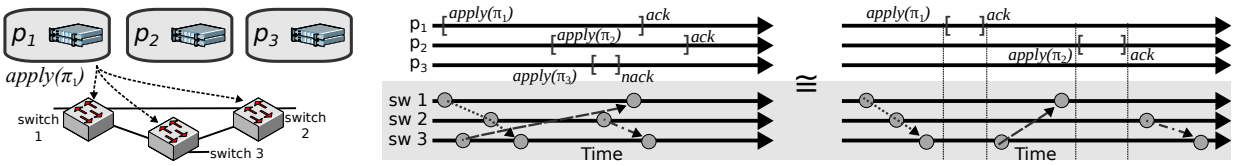
A *problem* is a set $\mathcal{P}$ of histories. An algorithm solves a problem $\mathcal{P}$ if the history of its every execution is in $\mathcal{P}$. An algorithm solves $\mathcal{P}$ *f-resiliently* if the property above holds in every f-resilient execution, *i.e.*, in which at most f controllers take only finitely many steps. An $(n-1)$-resilient solution is called *wait-free*.

Traces and packet consistency. In a history H, every packet injected to the network generates a *trace*, *i.e.*, a sequence of located packets: each event $ev = inject(pk, j)$ in E results in (pk, j) as the first element of the sequence, *forward*(pk, j, pk_1, k_1) adds (pk_1, j_1) to the trace, and each next *forward*$(pk_k, j_k, pk_{k+1}, j_{k+1})$ extends the trace with (pk_{k+1}, j_{k+1}), unless $j_k \in \{\mathsf{Drop}, \mathsf{World}\}$ in which case we say that the trace *terminates*. Note that in a finite network an infinite trace must contain a cycle.

Let $\rho_{ev,H}$ denote the trace corresponding to an inject event $ev = inject(pk, j)$ in a history H. A trace $\rho = (pk_1, i_1), (pk_2, i_2), \ldots$ is *consistent with a policy* π if $pk_1 \in dom(\pi)$ and $(i_1, i_2, \ldots) \in \pi$.

Tag complexity. It turns out that what can and what cannot be achieved by a distributed control plane depends on the number of available tags, used by control protocols to distinguish packets that should be processed by different policies. **[MC: This has nothing to do with control protocols.]** Throughout this paper, we will refer to the number of different tags used by a protocol as the *tag complexity*. Without loss of generality, we will typically assume that tags are integers $\{0, 1, 2, \ldots\}$, and our protocols seek to choose low tags first; thus, the tag complexity is usually the largest used tag number x, throughout the entire (possibly infinite) execution of the protocol and in the worst case.

Monitoring oracle. In order to be able to reuse tags, the control plane needs some feedback from the network about the *active policies*, *i.e.*, for which policies there are still packets in transit. We use an oracle model in this paper: each controller can query the oracle to learn about the tags currently in use by packets in any queue. Our assumptions on the oracle are minimal, and oracle interactions can be asynchronous. In practice, the available tags can simply be estimated by assuming a rough upper bound on the transit time of packets through the network.



(a) (b)

Figure 1: Example of a policy composition with a 3-controller control plane and 3-switch data plane (a). The three controllers try to concurrently install three different policies π_1, π_2, and π_3. We suppose that π_3 is conflicting with both π_1 and π_2, so π_3 is aborted (b). Circles represent data-plane events (an *inject* event followed by a sequence of forward events). Next to H we depict its "sequential equivalent" H_S. In the sequential history, no two requests are applied concurrently and no request is rejected. **[MC: Things to note about the example: (1) apply(π_1) is shown below p_1, might suggest that the request is sent to the switch 1. (2) H and H_S are not identified in the figure. (3) The caption says no request is rejected but π_3 is still aborted. (4) in H_S, why do the second and third sequence of data plane event occur after π_1 is acked?]**

3 The CPC Problem

Now we formulate our problem statement. At a high level, the CPC abstraction of consistent policy composition accepts concurrent *policy-update requests* and makes sure that the requests affect the traffic as a *sequential composition* of their policies. The abstraction offers a transactional interface where requests can be *committed* or *aborted*. Intuitively, once a request commits, the corresponding policy affects every packet in its domain that is subsequently injected. But in case it cannot be composed with the currently installed policy, it is *aborted* and does not affect a single packet. On the progress side, we require that if a set of policies conflict, at least one policy is successfully installed. We require that each packet arriving at a port is forwarded *immediately*; *i.e.*, the packet cannot be delayed, *e.g.*, until a certain policy is installed.

CPC Interface. Formally, every controller p_i accepts requests $apply_i(\pi)$, where π is a policy, and returns ack_i (the request is committed) or $nack_i$ (the request is aborted).

We specify a partial order relation on the events in a history H, denoted $<_H$. We say that a request *req precedes* a request *req*′ in a history H, and we write $req <_H req'$, if the response of *req* appears before the invocation of *req*′ in H. If none of the requests precedes the other, we say that the requests are *concurrent*. Similarly, we say that an inject event *ev precedes* (resp., *succeeds*) a request *req* in H, and we write $ev <_H req$ (resp., $req <_H ev$), if *ev* appears before the invocation (resp., after the response) of *req* in H. Two inject events *ev* and *ev*′ on the same port in H are related by $ev <_H ev'$ if *ev* precedes *ev*′ in H.

An inject event *ev* is concurrent with *req* if $ev \not<_H req$ and $req \not<_H ev$. A history H is *sequential* if in H, no two requests are concurrent and no inject event is concurrent with a request.

Let $H|p_i$ denote the *local* history of controller p_i, *i.e.*, the subsequence of H consisting of all events of p_i. We assume that every controller is *well-formed*: every local history $H|p_i$ is sequential, *i.e.*, no controller accepts a new request before producing a response to the previous one. A request issued by p_i is *complete* in H if it is followed by a matching response (ack_i or $nack_i$); otherwise it is called *incomplete*. A history H is *complete* if every request is complete in H. A *completion* of a history H is a complete history H' which is like H except that each incomplete request in H is completed with *ack* (intuitively, this is necessary if the request already affected packets) or *nack* inserted somewhere after its invocation.

Two histories H and H' are *equivalent* if H and H' have the same sets of events, for all p_i, $H|p_i = H'|p_i$, and for all inject events ev in H and H', $\rho_{ev,H} = \rho_{ev,H'}$. **[MC: This definition of equivalent seems too coarse. Consider 2 sequential histories H' and H'':**

$$H' : [req_1, ack_1, inject(pk_1, 1), forward(pk_1, 1, pk'_1, 2), req_2, ack_2]$$

and

$$H'' : [req_1, ack_1, req_2, ack_2, inject(pk_1, 1), forward(pk_1, 1, pk''_1, 3)]$$

These are equivalent histories according to the definition. However, the traces of pk_1 are different.]

Sequentially composable histories. A sequential complete history H is *legal* if these two properties are satisfied: (1) a policy is committed in H if and only if it does not conflict with the set of policies previously committed in H, and (2) for every inject event $ev = inject(pk, j)$ in H, the trace $\rho_{ev,H}$ is consistent with the composition of all committed policies that precede ev in H.

Definition 1 (Sequentially composable history) *We say that a complete history H is* sequentially composable *if there exists a legal sequential history S such that (1) H and S are equivalent, and (2) $<_H \subseteq <_S$.*

Intuitively, Definition 1 implies that the traffic in H is processed *as if* the requests were applied atomically and every injected packet is processed instantaneously. The legality property here requires that only committed requests affect the traffic. Moreover, the equivalent sequential history S must respect the order in which non-concurrent requests take place and packets arrive in H.

Definition 2 (CPC) *We say that an algorithm solves the problem of Consistent Policy Composition (CPC) if for its every history H, there exists a completion H' such that:*

Consistency. *H' is sequentially composable.*

Termination. *Eventually, every correct controller p_i that accepts a requests $apply_i(\pi)$, returns (ack_i or $nack_i$) in H.*

Note that, for an infinite history H, the Consistency and Termination requirements imply that an incomplete request in H can only cause aborts of conflicting requests for a finite period of time: eventually it would abort or commit in a completion of H and if it aborts, then no subsequent conflicting requests will be affected. As a result we provide an all-or-nothing semantics: a policy update, regardless of the behavior of the controller that installs it, either eventually takes effect or does not affect a single packet. Figure 1 gives an example of a sequentially composable history.

4 CPC Solutions and Complexity Bounds

We now discuss how the CPC problem can be solved and analyze the complexity its solutions incur. We begin with a simple wait-free algorithm FixTag which implicitly orders policies at a given ingress port; FixTag incurs a linear tag complexity in the number of to-be-installed policies. Then we present an f-resilient algorithm ReuseTag with tag complexity $f+2$. We also show that ReuseTag is optimal, *i.e.*, no protocol can maintain smaller tags for all networks.

4.1 FixTag: Per-Policy Tags

The basic idea of FixTag is to encode each possible forwarding path in the network by its own tag. Let τ_k be the tag representing the k^{th} possible path. FixTag assumes that, initially, for each internal port i_x which lies on the k^{th} path, a rule $r_{\tau_k}(pk) = (pk, i_{x+1})$ is installed, which forwards *any packet* tagged τ_k and forwards the corresponding packet to the path's successive port i_{x+1}.

Upon receiving a new policy request π and before installing any rules, a controller p_i executing FixTag sends a message to *all* other controllers informing them about the rules it intends to add to the ingress ports; every controller receiving this message rebroadcasts it (making the broadcast reliable), and starts installing the policy on p_i's behalf. This ensures that every policy update that started affecting the traffic eventually completes. **[MC: Why is there a need for rebroadcasting the message? The communication between controllers is reliable, no?] [MC: Who sends the ack?] [MC: A controller that is already processing a different policy will not pause on that.]** Let $i_1, \ldots, i_s$ be the set of ingress port, and π^j be the path specified by policy π for ingress port i_j, $j = 1, \ldots, s$. To install π, FixTag seeks to add a rule to each ingress port i_j; this rule tags all packets matching the policy domain with the tag describing the path π^j. However, since different policies from different controllers may conflict, every controller updates the ingress ports in a predefined order. Thus, conflicts are discovered already at the lowest-order port, and the conflict-free all-or-nothing installation of a policy is ensured.

Observe that FixTag does not require *any* feedback from the network on when packets arrive or leave the system. It just tags all traffic at the network edge; internally, the packets are only forwarded according to these tags.

We have the following theorem.

Theorem 3 FixTag *solves the CPC problem in the wait-free manner, without relying on the oracle and consensus objects.*

However, while providing a correct network update even under high control plane concurrency and failures, FixTag has a large tag complexity, namely linear in the number of to-be-installed policies (which may grow to super-exponential in the network size). If we want to reduce the tag overhead, we should be able to *reuse* tags that are not needed anymore: Ideally, the number of tags should only depend on the number of concurrently installed policies.

4.2 ReuseTag: Optimal Tag Complexity

The ReuseTag protocol sketched in Figure 2 allows controllers to reuse up to $f+2$ tags dynamically and in a coordinated fashion. As we will also show in this section, there does not exist any solution with less than $f + 2$ tags. Note that in the fault-free scenario ($f = 0$), only one bit can be used for storing the policy tag.

State machine. The protocol is built atop a replicated state machine (implemented, *e.g.*, using the construction of [10]) that imposes a global order on the policy updates and ensures a coordinated use and reuse of the protocol tags. For simplicity, we assume that the policies in the updates are uniquely identified.

The state machine we are going to use in our algorithm, and which we call PS (for *Policy Serialization*) exports, to each controller p_i, two operations:

- *push*(i, π), where π is a policy, that always returns `ok`;
- *pull*(i) that returns $\perp$ or a tuple (π, tag), where π is a policy and $tag \in \{0, \ldots, f + 1\}$.

Intuitively, p_i invokes $push(i, \pi)$ to put policy π in the queue of policies waiting to be installed; and p_i invokes $pull(i)$ to fetch the next policy to be installed. The invocation of *pull* returns $\perp$ if all policies pushed so far are already installed and there is an "available" tag (to be explained below) **[MC: Why must there be an available tag to return $\perp$?]**, otherwise it returns a tuple (π, tag), informing p_i that policy π should be equipped with *tag*.

Let S be a sequential execution of PS. Let $\pi_1, \pi_2, \ldots$ be the sequence of policies proposed in S as arguments of the *push*() operations (in the order of appearance). Let $(\pi_{i,1}, \tau_{i,1}), (\pi_{i,2}, \tau_{i,2}), \ldots$ be the sequence of non-$\perp$ responses to $pull(i)$ operations in S (performed by p_i). If S contains exactly k *non-trivial* (returning non-$\perp$ values) $pull(i)$ operations, then we say that p_i *performed* k *non-trivial pulls in* S. If S contains $pull(i)$ that returns $(\pi, t) \neq \perp$, followed by a subsequent $pull(i)$, then we say that π is *installed* in S.

We say that τ_k is *blocked* at the end of a finite history S if S contains $pull(i)$ that returns $(\pi_{k+1}, \tau_{k+1}, 0)$ but does not contain a subsequent **[MC: What is 0?]** $pull(i)$. In this case, we also say that p_i *blocks* tag τ_k at the end of S. Note that a controller installing policy π_{k+1} blocks the tag associated with the *previous* policy π_k (or the initially installed policy in case $k = 0$). Now we are ready to define the sequential specification of PS via the following requirements on S:

• **Non-triviality:** If p_i performed k non-trivial pulls, then a subsequent $pull(i)$ returns $\perp$ if and only if the pull operation is preceded by at most k pushes or $f + 1$ or more policies are blocked in S. In other words, the k-th pull of p_i must return some policy if at least k policies were previously pushed and at most f of them are blocked.

• **Agreement:** For all $k > 0$, there exists $\tau_k \in \{0, \ldots, f + 1\}$ such that if controllers p_i and p_j performed k nontrivial pulls, then $\pi_{i,k} = \pi_{j,k} = \pi_k$ and $\tau_{i,k} = \tau_{j,k} = \tau_k$ for some τ_k. Therefore, the controllers compute the same order in which the proposed policies must be installed, with the same sequence of tags.

• **Tag validity:** For all k, τ_k is the minimal value in $\{0, \ldots, f + 1\} - \{\tau_{k-1}\}$ that is not blocked in $\{0, \ldots, n - 1\}$ when the first $pull(i)$ operation that returns (π_k, τ_k, r_k) is performed. The intuition here is that the tags are chosen deterministically based on all the tags that are not currently blocked. Since, by the Non-triviality property, at most f policies are blocked in this case, $\{0, \ldots, f + 1\} - \{\tau_{k-1}\}$ is non-empty.

In the following, we assume that a *linearizable* f-resilient implementation of PS is available [12]: any concurrent history of the implementation is, in a precise sense, equivalent to a sequential history that respects the temporal relations on operations and every operation invoked by a correct controller returns, assuming that at most f controllers fail. Note that the PS machine establishes a total order on policies $(\pi_1, tag_1), (\pi_2, tag_2), \ldots$, which we call the *composition order* (the policy requests that do not compose with a prefix of this order are ignored).

Algorithm operation. The algorithm is depicted in Figure 2 and operates as follows. To

```
Initially:
    seq := ⊥; cur_i := ⊥

upon apply(π̃)
1   cur_i := π̃
2   PS.push(i, π̃)

do forever
3   wait until PS.pull(i) returns (π, t) ≠ ⊥
4   if (seq and π conflict) then
5      res := nack
6   else
7      seq := seq.(π, t)
8      wait until tag(|seq| − 1) is not used
9      install(seq)
10     res := ack
11  if π = cur_i then return res; cur_i := ⊥
```

Figure 2: The ReuseTag algorithm: pseudocode for controller p_i.

install policy $\tilde{\pi}$, controller p_i first pushes $\tilde{\pi}$ to the policy queue by invoking PS.*push*$(i, \tilde{\pi})$.

In parallel, to install its policy and help the others, the controller runs the following task (Lines 3-11). First it keeps invoking PS.*pull*(i) until a (non-$\perp$) value (π_k, τ_k) is returned (Line 3); here k is the number of nontrivial pulls performed by p_i so far. The controller checks if π_k is not conflicting with previously installed policies (Line 4), stored in sequence *seq*. Otherwise, in Line 8, p_i waits until the traffic in the network only carries tag τ_{k-1} (the tag τ_{k-2} used by the penultimate policy in *seq*, denoted $tag(|seq| - 1)$). Here p_i uses the *oracle* (described in Section 2) that produces the set of currently active policies.

Then the controller tries to install π_k on all internal ports followed by the ingress ports, one by one, in a pre-defined order, employing the "two-phase update" strategy of [18] (Line 9). The update of an internal port p is performed using an atomic operation that adds the rule associated with π_k equipped with τ_k to the set of rules currently installed on p. The update on an ingress port p simply replaces the currently installed rule with a new rule tagging the traffic with τ_k which succeeds *if and only if* the port currently carries the policy tag τ_{k-1} (otherwise, the port is left untouched). Once all ingress ports are updated, old rules are removed, one by one, from the internal ports. If π_k happens to be the policy currently proposed by p_i, the result is returned to the application.

Intuitively, a controller blocking a tag τ_k may still be involved in installing τ_{k+1} and thus we cannot reuse τ_k for a policy other than π_k. Otherwise, the slow controller may wake up and update a port with an outdated rule. But since a slow or faulty controller can block at most one tag, there eventually must be at least one available tag in $\{0, \ldots, f + 1\} - \{\tau_{k-1}\}$ when the first controller performs its k-th nontrivial pull. In summary, we have the following result.

Theorem 4 ReuseTag *solves the CPC Problem f-resiliently with tag complexity $f + 2$ using f-resilient consensus objects.*

A natural optimization of the ReuseTag algorithm is to allow a controller to broadcast the outcome of each complete policy update. This way "left behind" controllers can catch up with the more advanced ones, so that they do not need to re-install already installed policies.

Note that since in the algorithm, the controllers maintain a total order on the set of policy updates that respects the order, we can easily extend it to encompass *removals* of previously installed policies. To implement removals, it seems reasonable to assume that a removal request for a policy π is issued by the controller that has previously installed π.

The tag complexity of ReuseTag is, in a strict sense, optimal. Indeed, we now show that there exists no f-resilient CPC algorithm that uses $f + 1$ or less tags in any network. By contradiction, for any such algorithm we construct a network consisting of two ingress ports connected to f consecutive loops. We then present $f + 2$ composable policies, $\pi_1, \ldots, \pi_{f+2}$, that have overlapping domains but prescribe distinct paths. Given that only $f + 1$ tags are available, we can construct an execution of the assumed algorithm in which a policy update installing f_i invalidates the previously installed f_j by using the same tag, contradicting the Consistency property of CPC. Thus:

Theorem 5 *For each $f \geq 1$, there exists a network such that any f-resilient CPC algorithm using f-resilient consensus objects has tag complexity at least $f + 2$.*

5 Related Work

Distributed Computing. There is a long tradition of defining correctness of a concurrent system via an equivalence to a sequential one [12,15,17]. The notion of sequentially composable histories is reminiscent of linearizability [12], where a history of operations concurrently applied by a collection of processes is equivalent to a history in which the operations are in a sequential order, respecting their real-time precedence. In contrast, our sequentially composable histories impose requirements not only on high-level invocations and responses, but also on the way the traffic is processed. We require that the committed policies constitute a conflict-free sequential history, but, additionally, we expect that each *path* witnesses only a prefix of this history, consisting of all requests that were committed before the path was initiated.

The transactional interface exported by the CPC abstraction is inspired by the work on speculative concurrency control using software transactional memory (STM) [19]. Our interface is however intended to model realistic network management operations, which makes it simpler than more recent models of dynamic STMs [11]. Also, we assumed that processes are subject to failures, which is usually not assumed by STM implementations.

Software Defined Networking. At the heart of Software-defined networking (SDN) lies the decoupling of the system that makes decisions about where traffic is sent (the *control plane*) from the underlying systems that forward traffic to the selected destination (the *data plane*). For an introduction to SDN as well as for a discussion of the differences to concepts such as active networks (where packets carry code and which are hard to formally verify) and protocols such as MPLS (which do not come with a software control plane and which do not allow users to specify even basic consistency properties), we refer the reader to [4].

Onix [14] is among the earliest distributed SDN controller platforms. Onix applies existing distributed systems techniques to build a Network Information Base (NIB), *i.e.*, a data structure that maintains a copy of the network state, and abstracts the task of network state distribution from control logic. However, Onix expects developers to provide the logic that is necessary to detect and resolve conflicts of network state due to concurrent control. In contrast, we study concurrent policy composition mechanisms that can be leveraged by any application in a general fashion.

For the case of a single controller, Reitblatt *et al.* [18] formalized the notion of per-packet consistency and introduced the problem of *consistent network update*. Mahajan and Wattenhofer [16] introduced several new variants of network update problems, and presented more efficient, dependency-based protocols. We complement this line of research by assuming a distributed computing perspective, and by investigating robust and concurrent policy installations. Our work also introduces the notion of tag complexity.

Bibliographic Note. In our SIGCOMM HotSDN workshop paper [3], we introduced the notion of software transactional networking, and sketched a tag-based algorithm to consistently compose concurrent network updates. However, the algorithm proposed there is not robust to any controller failure, and features an exponential tag complexity. (A simple corollary of the present paper is that the non-failure setting can be solved with two tags only.)

6 Concluding Remarks

We believe that our paper opens a rich area for future research, and we understand our work as a first step towards a better understanding of how to design and operate a robust SDN control plane. As a side result, our model allows us to gain insights into minimal requirements on the network that

enable consistent policy updates: *e.g.*, in Appendix A, we prove that consistent network updates are impossible if SDN ports do not support atomic read-modify-write operations.

Our FixTag and ReuseTag algorithms highlight the fundamental trade-offs between the concurrency of installation of policy updates and the overhead on messages and switch memories. Indeed, while being optimal in terms of tag complexity, ReuseTag essentially reduces to installing updates sequentially. Our initial concerns were resilience to failures and overhead, so our definition of the CPC problem did not require any form of "concurrent entry" [13]. But it is important to understand to which extent the concurrency of a CPC algorithm can be improved, and we leave it to future research.

Another direction for future research regards more complex, non-commutative policy compositions: while our protocol can also be used for, *e.g.*, policy removals, it will be interesting to understand how general such approaches are. We have also started to develop a proof-of-concept prototype implementation of our distributed control plane [2].

References

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. In *POPL*, 2014.

[2] M. Canini, D. D. Cicco, P. Kuznetsov, D. Levin, S. Schmid, and S. Vissicchio. STN: A Robust and Distributed SDN Control Plane. In *To appear in Open Networking Summit (ONS)*, March 2014.

[3] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. Software transactional networking: Concurrent and consistent policy composition. In *ACM SIGCOMM HotSDN*, August 2013.

[4] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A retrospective on evolving sdn. In *Proc. Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[5] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[6] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4), 2010.

[7] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.

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

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[10] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, Jan. 1991.

[11] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proc. 22nd Annual Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.

[12] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[13] Y.-J. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

[14] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, Sept. 1979.

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

[17] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.

[18] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.

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

A Impossibility for Weaker Port Model

It turns out that SDN ports must support atomic policy updates (*i.e.*, an atomic read-modify-write); otherwise it is impossible to update a network consistently in the presence of even one crash failure. Concretely, we assume here that a port can be accessed with two atomic operations: *read* that returns the set of rules currently installed at the port and *write* that updates the state of the port with a new set of rules.

Theorem 6 *There is no solution to CPC using consensus objects that tolerates one or more crash failures.*

Proof. By contradiction and assume that there is a 1-resilient CPC algorithm A using consensus objects.

Consider a network including two ingress ports, 1 and 2, initially configured to forward all the traffic to internal ports (we denote this policy by π_0). Let processes p_1 and p_2 accept two policy-update requests $req_1 = apply_1(\pi_1)$ and $req_2 = apply_2(\pi_2)$, respectively, such that π_1 is refined by π_2, *i.e.*, $pr(\pi_2) > pr(\pi_1)$ and $dom(\pi_2) \subset dom(\pi_1)$, and paths stipulated by the two policies to ingress ports 1 and 2 satisfy $\pi_1^{(1)} \neq \pi_2^{(1)}$ and $\pi_1^{(2)} \neq \pi_2^{(2)}$.

Now consider an execution of our 1-resilient algorithm in which p_1 is installing π_1 and p_2 takes no steps. Since the algorithm is 1-resilient, p_1 must eventually complete the update even if p_2 is just slow and not actually faulty. Let us stop p_1 after it has configured one of the ingress ports, say 1, to use policy π_1, and just before it changes the state of 2 to use policy π_1. Note that since p_1 did not witness a single step of p_2 the configuration it is about to write to port 2 only contains the composition of π_0 and π_1.

Now let a given packet in $dom(\pi_1)$ arrive at port 1 and be processed according to π_1. We extend the execution with p_2 installing π_2 until both ports 1 and 2 are configured to use the composition $\pi_0 \cdot \pi_1 \cdot \pi_2$. Such an execution exists, since the algorithm is 1-resilient and π_1 has been already applied to one packet. Therefore, by sequential composability, the sequential equivalent of the execution, both $apply(\pi_1)$ and $apply(\pi_1)$ must appear as committed.

But now we can schedule the enabled step of p_1 to overwrite the state of port 2 with the "outdated" configuration that does not contain π_2. From now on, every packet in $dom(\pi_2)$ injected at port 2 is going to be processed according to π_1—a contradiction to sequential composability. □

B Proofs

B.1 Proof Sketch of Theorem 3

The correctness of the algorithm is based on three pillars.

1. *Global policy order:* The strict port order $\prec$ guarantees that each sequential history respects the total order of policy updates imposed by the ingress port of lowest order.

2. *All-or-nothing semantics:* A policy which started taking effect at some ingress ports will eventually be installed at all ingress ports. This follows from the reliable broadcast implementation: the rebroadcasts ensure that eventually, all processes will learn about (and help finish) the planned policy installation, even if the initiator failed before it notified the other processes.

3. *Consistency:* The proof of per-packet consistency is simple: a packet will be marked with an immutable tag at its ingress port, and the tag defines a unique path in the network.

B.2 Proof of Theorem 4

We study the termination and consistency properties in turn.

Termination: Consider any f-resilient execution E of ReuseTag and let $\pi_1, \pi_2, \ldots$ be the sequence of policy updates as they appear in the linearization of the state-machine operations in E. Suppose, by contradiction, that a given process p_i never completes its policy update π. Since our state-machine PS is f-resilient, p_i eventually completes its $push(i, \pi)$ operation. Assume π has order k in the total order on push operations. Thus, p_i is blocked in processing some policy π_ℓ, $1 \leq \ell \leq k$, waiting in Lines 3 or 8.

Note that, by the Non-Triviality and Agreement properties of PS, when a correct process completes installing π_ℓ, eventually every other correct process completes installing π_ℓ. Thus, all correct processes are blocked while processing π. Since there are at most f faulty processes, at most f policies can be blocked forever. Moreover, since every blocked process has previously pushed a policy update, the number of processes that try to pull proposed policy updates cannot exceed the number of previously pushed policies. Therefore, by the Non-Triviality property of PS, eventually, no correct process can be blocked forever in Line 3.

Finally, every correct process has previously completed installing policy with tag $\tau_{\ell-1}$. By the algorithm, every injected packet is tagged with $\tau_{\ell-1}$ and, eventually, no packet with a tag other than $\tau_{\ell-1}$ stays in the network. Thus, no correct process can be blocked in Line 8—a contradiction, *i.e.*, the algorithm satisfies the Termination property of CPC.

Consistency: To prove the Consistency property of CPC, let S be a sequential history that respects the total order of policy updates determined by the PS. According to our algorithm, the response of each update in S is *ack* if and only if it does not conflict with the set of previously committed updates in S. Now since each policy update in S is installed by the two-phase update procedure using atomic read-modify-write update operations, every packet injected to the network, after a policy update completes, is processed according to the composition of the update with all preceding updates. Moreover, an incomplete policy update that manages to push the policy into PS will eventually be completed by some correct process (due to the reliable broadcast implementation). Finally, the per-packet consistency follows from the fact that packets will always respect the global order, and are marked with an immutable tag at the ingress port; the corresponding forwarding rules are never changed while packets are in transit.

Thus, the algorithm satisfies the Consistency property of CPC.

B.3 Proof of Theorem 5

Assume the network T_f of two ingress ports A and B, and $f+1$ "loops" depicted in Figure 3 and consider a scenario in which the controllers apply a sequence of policies defined as follows. Let π_i, $i = 1, \ldots, f$, denote a policy which, for each of the two ingress port, specifies a path that in every loop $\ell \neq i$ takes the upper path and in loop i takes the lower path (the dashed line in Figure 3). The policy π_0 specifies the paths that always go over the upper parts of all the loops (the dashed line in Figure 3).

We assume that for any two policies π_i and π_j, such that $0 \geq j < j \geq f+1$, we have $pr(\pi_i) > pr(\pi_j)$ and $dom(\pi_i) \subset dom(\pi_j)$, *i.e.*, all these the policies are composable, and adding a new policy to the composition makes the composed policy more refined. Note that, assuming that only policies π_i, $i = 0, \ldots, f+1$, are in use, each of the ingress ports may only store one rule per tag that forwards all the packets to the next branching port. Intuitively, the only way to make sure that an injected packet is processed according to a new policy in the set is to equip injected packets with specific tags and forward them further.

Suppose that 0 is the tag used for the initially installed π_0. By induction on $i = 1, \ldots, f+1$, we are going to show that any f-resilient CPC algorithm on T_f has a finite execution E_i at the end of which (1) a composed policy $\pi_0 \cdot \pi_1 \cdots \pi_i$ is installed and (2) there is a set of i processes, $q_1, \ldots, q_i$, such that each q_ℓ, $\ell = 1, \ldots, i$, is about to access an ingress port with an update operation that, if the currently installed rule uses $\ell - 1$ to tag the injected packets, replaces it with a rule that uses ℓ instead.

For the base case $i = 1$, assume that p_1 proposes to install π_1. Since the network initially carries traffic tagged 0, the tag used for the composed policy $\pi_0 \cdot \pi_1$ must use a tag different from 0, without loss of generality we call it 1. There exists an execution in which some process q_1 has updated the tag on one of the ingress port with tag 1 and is just about update the other port. Now we "freeze" q_1 and let another process to complete the update of the remaining ingress port. Such an execution exists, since the protocol is f-resilient, assuming that $f > 0$ and, by the Consistency property of CPC, any update that affected the traffic must be eventually completed. In the resulting execution E_1, q_1 is about to update an ingress port to use tag 1 instead of 0 and the network operates according to policy $\pi_0 \cdot \pi_1$.

Now take $1 < i \leq f+1$ and, inductively, consider the execution E_{i-1}. Now suppose that some process in $\Pi - \{q_1, \ldots, q_{i-1}\}$ proposes to install π_i. Similarly, since the algorithm is f-resilient (and, thus, $(i-1)$-resilient), there is an extension of E_{i-1} in which no process in $\{q_1, \ldots, q_{i-1}\}$

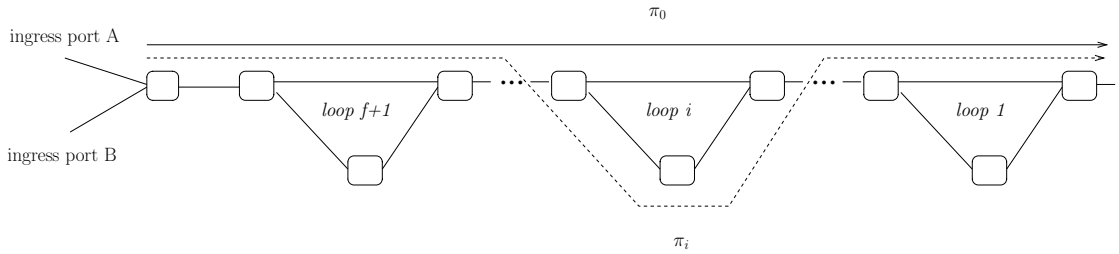


Figure 3: The $(f+1)$-loop network topology T_f.

takes a step after E_{i-1} and eventually some process $q_i \notin \{q_1, \ldots, q_{i-1}\}$ updates one of the ingress ports to apply $\pi_0 \cdots \pi_i$ so that instead of the currently used tag $i-1$ a new tag τ is used. (By the Consistency property of CPC, π_i should be composed with all policies $\pi_0, \ldots, \pi_{i-1}$.)

Naturally, the new tag τ cannot be $i-1$. Otherwise, while installing $\pi_0 \cdots \pi_i$, either q_i updates port i before port $i-1$ and some packet tagged i would have to take lower paths in both loops i and $i-1$ (which does not correspond to any composition of installed policies), or q_i updates port $i-1$ before i and some packet would have to take no lower paths at all (which corresponds to the policy π_0 later overwritten by $\pi_0 \cdots \pi_{i-1}$).

Similarly, $\tau \notin \{0, \ldots, i-2\}$. Otherwise, once the installation of $\pi_0 \cdots \pi_i$ by q_i is completed, we can wake up process p_{t+1} that would replace the rule of tag τ with a rule using tag $\tau+1$, on one of the ingress ports. Thus, every packet injected at the port would be tagged $\tau+1$. But this would violate the Consistency property of CPC, because $\pi_0 \cdots \pi_i$ using tag τ is the most recently installed policy.

Thus, q_i, when installing $\pi_0 \cdots \pi_i$, must use a tag not in $\{0, \ldots, i-1\}$, say i. Now we let q_i freeze just before it is about to install tag i on the second ingress port it updates. Similarly, since $\pi_0 \cdots \pi_i$ affected the traffic already on the second port, there is an extended execution in which another process in $\Pi - \{q_1, \ldots, q_i\}$ completes the update and we get the desired execution E_i.

In E_{f+1} exactly $f+2$ tags are concurrently in use, which completes the proof.