A Distributed SDN Control Plane for Consistent Policy Updates

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

Software-defined networking (SDN) is a novel paradigm that out-sources the control of packet-forwarding switches to a set of software controllers. The most fundamental task of these controllers is the correct implementation of the *network policy*, *i.e.*, the intended network behavior. In essence, such a policy specifies the rules by which packets must be forwarded across the network.

This paper studies a distributed SDN control plane that enables *concurrent* and *robust* policy implementation. We introduce a formal model describing the interaction between the data plane and a distributed control plane (consisting of a collection of fault-prone controllers). Then we formulate the problem of *consistent* composition of concurrent network policy updates (short: the *CPC Problem*). To anticipate scenarios in which some conflicting policy updates must be rejected, we enable the composition via a natural *transactional* interface with all-or-nothing semantics.

We show that the ability of an f-resilient distributed control plane to process concurrent policy updates depends on the tag complexity, *i.e.*, the number of policy labels (a.k.a. *tags*) available to the controllers, and describe a CPC protocol with optimal tag complexity $f + 2$.

[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 [**?**] 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 [**?**]). Therefore, we address here the challenge of composing *concurrent* updates, while preserving a property known as *per-packet consistency* [**?**]. 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* [**?**], 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) [**?**], 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 [?]. 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 [?], 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 **??**. Section **??** formulates the CPC problem and Section **??** describes our CPC solutions and their complexity bounds. We discuss related work in Section **??** and conclude in Section **??**. 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 [?] that allows them to implement, in a fault-tolerant manner, any replicated state machine, provided its sequential specification [?]. The consensus abstraction can be obtained, *e.g.*, assuming the eventually synchronous communication [?] or the *eventual leader* Ω failure detector [?] shared by the controllers, assuming a majority of correct controllers or the *quorum* failure detector Σ [?].

Data plane. Following [?], 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 [?], 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 Drop (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 [?], *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*. Here a rule r is 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 only a part of a packet pk can be modified by a rule: namely, a header field called the *tag* that carries the information that is used to identify which rules apply to a given packet.

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 $dom(\pi)$. Formally, 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 among 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:

- $\mathit{inject}(pk, j)$: 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 $Q_j \cdot pk$.

- $\mathit{forward}(pk, j, pk', k)$, $j \in P$: 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 message 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 sometimes called *wait-free*.

Traces and trace consistency. In a history H, every packet injected to the network generates a *trace*, *i.e.*, a sequence of located packets: each event $ev = \mathit{inject}(pk, j)$ in E results in (pk, j) as the first element of the sequence, $\mathit{forward}(pk, j, pk_1, k_1)$ adds (pk_1, j_1) to the trace, and each next $\mathit{forward}(pk_k, j_k, pk_{k+1}, j_{k+1})$ extends the trace with (pk_{k+1}, j_{k+1}), unless $j_k \in \{textsf{Drop}, \mathsf{World}\}$ 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 $ev = \mathit{inject}(pk, j)$ a history H. Trace $\rho = (pk_1, i_1), (pk_2, i_2), \ldots$ is *consistent with a policy* π if $pk_1 \in \mathit{dom}(\pi)$ 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. Throughout this paper, we will refer to the number of different tags used by a protocol as the *tag complexity*. W.l.o.g., 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. Observe that a protocol of tag complexity x requires $\lfloor \log x \rfloor + 1$ bits in the packet header.

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.

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. Recall that inject and forward events are not under our control, *i.e.*, the packets cannot be delayed, *e.g.*, until a certain policy is installed. Therefore, a packet trace that interleaves with a policy update must be consistent with the policy *before* the update or the policy *after* the update (and not some partial policy), the property is referred to as *per-packet consistency* [**?**]).

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 after the response (resp., before the invocation) 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 $nack_i$) in $H|p_i$ (otherwise it is called *incomplete*). A history 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', $\rho_{ev,H} = \rho_{ev,H'}$.

Sequentially composable histories. A sequential complete history H is *legal* if the following 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 **??** implies that the traffic in H is processed *as if* the requests were applied