The Case for Reliable Software Transactional Networking

Marco Canini
Petr Kuznetsov
Dan Levin
Stefan Schmid

Technische Universität Berlin /
Deutsche Telekom Laboratories

Bericht-Nummer: 2013-06
ISSN-Nummer: 1436-9915

The Case for Reliable Software Transactional Networking

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

[1] TU Berlin / T-Labs [2] Télécom ParisTech

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 initiates the study of the SDN control plane as a distributed system.

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. The composition is enabled via a *transactional* interface with all-or-nothing semantics. The system behaves as though committed updates are installed *atomically* and every data packet traverses the network *instantaneously*, respecting a sequential composition of previously installed committed updates. Updates that cannot be composed are aborted and do not affect the data plane.

We show that in the asynchronous environment, it is impossible to achieve consistent policy composition that tolerates a single controller crash. We then discuss stronger variants of the model that allow for solving the problem and study algorithmic complexities of such solutions.

Regular and student paper: Dan Levin is a full-time student.

---

[*] Contact author: stefan@net.t-labs.tu-berlin.de, FG INET, TEL 16, Ernst-Reuter Platz 7, 10587 Berlin, Germany

1 Introduction

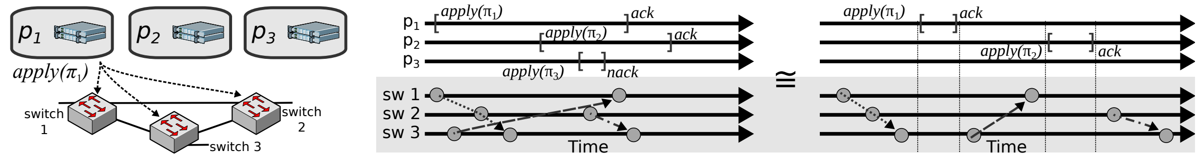
One of the main objectives of network management is secure and correct *operation*: network *policies* must be enforced to isolate the traffic from different user groups (e.g., traffic from the student dorm network cannot reach the faculty LAN). To achieve this isolation, appropriate *forwarding* rules must be installed at the network switches. Even relatively small computer networks such as enterprise networks are already quite complex distributed systems, typically configured by a number of different network protocols and hand-crafted configuration scripts.

The *software-defined networking (SDN)* paradigm simplifies network management by outsourcing the control over the so-called *data plane* (i.e., the packet forwarding done at the switches) to a set of network-attached software controllers. Due to this centralized management of the network resources at the controller, SDN is often considered as an enabler for a *network operating system* [4]. Naturally, to ensure availability of the system, we need to implement the control plane as a distributed system running on multiple redundant controllers. This control plane accepts *policy updates* (access control, routing, etc.) issued concurrently by different users or administrators. The goal of this control plane is to *consistently* compose these updates. One of our important contributions is precisely the notion of consistency of concurrent policy composition.

After giving a short example of a concurrent application of policy updates (Section 2), this paper introduces a formal model for software-defined networking under fault-prone concurrent control (Section 3). The network obeys a *policy*, which in our case consists of a set of rules that determine how data packets are forwarded as they traverse the network. We envision scenarios in which multiple concurrent network-control applications modify the policy by introducing possibly conflicting rules, and we expect that only policy updates that *compose* (induce a *correct* network behavior) are allowed to affect the traffic. *Per-packet consistency* [12] is an important desirable property that simplifies reasoning about correct network operation. Informally, this property ensures that every packet is processed at every switch it encounters in the data plane according to just one and same policy, which is the composition of policies proposed by the time when the packet entered the network.

Then we define the abstraction of *Consistent Policy Composition (CPC)* (Section 4). To address the issue of conflicting policies, the CPC abstraction offers a *transactional* interface. A policy-update request returns *commit* if the update is successfully integrated in the current network policy or *abort* if the update cannot be installed. Our correctness property informally requires that the abstraction, regardless of the actual interleaving of concurrent policy updates and data packets' arrivals, *appears* sequential to every data packet, as though all the committed requests (and possibly a subset of incomplete ones) are applied atomically and no data packet is in flight while an update is being installed. On the progress side, we expect a request to be aborted only if it conflicts with the currently installed policy or a concurrently proposed request. We believe that the CPC abstraction, inspired by a popular paradigm of software transactional memory (STM) [13], 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 show (Section 5) that it is impossible to implement the CPC abstraction in the presence of a single controller's crash failure. The requirement of per-packet consistency allows us to introduce an interesting variant of the bivalency argument [3], where the valency of an algorithm's execution accounts for all possible *paths* a packet may take in all extensions of the execution. Since typically the controllers do not have influence on the network traffic workload, our impossibility proof is able to exploit the intertwined combination of two kinds of concurrency: overlapping policy-update requests may arbitrarily interleave with traffic-related events.



(a) (b)

Figure 1: Example of a policy composition: (a) 3-process control plane and 3-switch data plane, (b) a sequentially composable concurrent history H and its sequential equivalent H_S.

The paper then investigates stronger model abstractions which enable fault-tolerant CPC implementations (Section 6). In particular, we find that a slightly more powerful SDN switch interface supporting an atomic read-modify-write allows for a *wait-free* CPC solution, i.e., tolerating $n-1$ failures out of n controllers. However, to distinguish traffic corresponding to different concurrent policies, the packets need to carry certain control information in their headers (in so called *tag* fields). We present a wait-free CPC algorithm, called TAG which incurs exponential (in the size of the network) overhead on the packet size and the switch memory. Assuming an asynchronous (but accurate) feedback from the network on the tags that are no longer used, we describe a *single-controller* algorithm BIT that uses just a single bit for the tag information. However, we conjecture that no fault-tolerant algorithm using small tags exists for scenarios with $n > 1$ fault-prone processes.

2 Policy Composition in SDN: Example

Imagine a network consisting of three switches `sw1`, `sw2` and `sw3` (Figure 1(a)). The switches constitute a complete network and are manipulated by three controllers p_1, p_2, and p_3. The function of the controllers is to accept policy-update requests issued by the control applications and try installing the updates on the switches.

An example of a concurrent history H is presented in Figure 1(b). Here the three controllers try to concurrently install three different policies π_1, π_2, and π_3. Imagine that π_1 and π_2 are applied to disjoint fractions of traffic (e.g., π_1 affects only `http` traffic and π_2 only `ssh` traffic) and, thus, can be installed independently of each other. In contrast, let us assume that π_3 is conflicting with both π_1 and π_2 (e.g., it applies to traffic coming from `src address` = 1.2.3.4). In this history, requests π_1 and π_2 are committed (returned *ack*), while π_3 is aborted (returned *nack*).

While the concurrent policy-update requests are processed, three packets are injected to the network (at switches `sw1`, `sw2`, and `sw3`) leaving three *traces* depicted with dotted and dashed arrows. Each trace is in fact the sequence of ports which the packet goes through while it traverses the network. For example, in one of the traces (depicted with the dotted arrow), a packet arrives at `sw1`, then it is forwarded to `sw2`, and then to `sw1`.

Next to H we present its "sequential equivalent" H_S. In the sequential history, no two requests are applied concurrently and no request is rejected. Also, the traces of H are reshuffled in H_S in such a way that no packet is in flight while an update is being installed. Note that each trace in H is exactly the same as in H_S: no packet can distinguish H from H_S, i.e., the traffic on the data plane is processed *as though* the application of policy updates is *atomic* and packets cross the network *instantaneously*. Here the first packet is processed according to the initial policy π_0, the second–by the composition of π_0 and π_1, and the third by the composition of π_0, π_1, and π_2. Note that the aborted request π_3 does not affect any packet in the network. As we formally define later, the existence of H_S establishes that H is *sequentially composable*.

In the following two sections, we introduce our system model and formally define the notion of sequentially composable histories and the problem of consistent policy composition.

3 Modeling Software Defined Networks

Control plane. We consider a system of $n \geq 2$ *processes* (or *controllers*), $p_1, \ldots, p_n$, that communicate via (asynchronous but reliable) message-passing. These processes constitute the distributed *control plane* of the SDN network. We assume that the processes are subject to *crash* failures: a faulty process stops taking steps of its algorithm. The process that never crashes is called *correct* and we assume that there is at least one correct process.

Data plane. Following [12], we model the physical *network* (or *data plane*) as a set P of *ports* and a set $L \subseteq P \times P$ of *links.* A link $(i,j) \in L$ is called *outgoing* for port i and *incoming* for port j. A port that has no outgoing links is called *ingress*, otherwise the port is called *internal.* Let $L(i)$, $i \in P$, denote the set of *neighbors* of i, i.e., $\{j \in P | (i,j) \in L\}$. We additionally consider two distinct ports, World and Drop used to represent forwarding a packet to the outside of the network and dropping the packet, respectively.

We model the workload on the data plane as a set Π of *packets.* In order to distinguish between control plane and data plane communication, we will use the term *message* for a communication involving at least one process, and the term *packet* for data plane communication. In general, we will use the term *packet* canonically as a type [12], 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 in the text. For instance, a policy usually refers to packet *types* whereas per-packet consistency refers to packet *instances.*

Ports are attached to *switches*, and 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 the port. 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 $(pk', j) = Q_i(pk)$ is computed and the packet pk' is placed to the queue Q_j.

We represent S_i, the switch function at port i as a collection of *rules.* Here a rule r is a partial map $r : \Pi \to \Pi \times P$ that for each packet in its domain $dom(r)$, generates a new located packet to be forwarded further. We assume that rules $r \in S_i$ are *applicable to a port* i, *i.e.*, r only puts packets to the queues of the ports directly connected to i: $\forall pk \in dom(r)$ and $(pk', j) = r(pk)$, $(i,j) \in P$. We assume that each port is configured to process every packet: $\bigcup_{r \in S_i} dom(r) = \Pi$.

We assume that only a part of a packet pk can be modified by a rule: namely the *tag* [10]. We denote the tag of a packet pk by $\tau(pk)$ and assume that tags are of length $|\tau|$ bits.

Port operations. A port can be accessed with operations *read*, *add-rule*, and *remove-rule.* A *read*(i) operation returns the set of rules representing the switch function of port i. An *add-rule*(i, r) operation adds a new rule r to port i. A *remove-rule*(i, r) operation removes rule r from port i (if present). We discuss a stronger port model in which a port can be accessed with an atomic read-modify-write (RMW) primitive in Section 6.

Policies and policy composition. A *policy* π is defined by a *domain* $dom(\pi) \subseteq \Pi$ and a unique *forwarding path* for each ingress port that should apply to the packets matching its domain $dom(\pi)$ (we assume that paths are loop-free). Thus, π induces a set of paths $\pi = \{\pi^{(1)}, \ldots, \pi^{(s)}\}$, one for each ingress port.

Each policy π comes with a *priority* level $\ell(\pi) \in \mathbb{N}$. In case a packet is matched by multiple rules, it should be forwarded according to the rule of the highest priority policy. If different policies have the same priority and the rule is not unique, we have a *conflict.*

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 $\ell(\pi) = \ell(\pi')$. Now a set U of policies is *conflict-free* if no two policies in U conflict.

Intuitively, conflict-free set of policies can be *composed*, *i.e.*, their rules can be independently installed so that every packet is processed consistently with the composed policy. If S_i is conflict-free, then a packet pk arriving at port i is processed according to the highest priority rule r that applies to the packet ($pk \in dom(r)$). We assume that if S_i is not conflict-free and the packet is subject to two conflicting rules, then the port is configured to apply some default deterministic rule to select one of this rule to be applied to the packet.

Executions. In order to formally reason about correctness, we need the concept of *executions.* An *execution* is a sequence of atomic *events*, where in each event, either a packet is atomically processed on one of the ports, or a process updates the port's configuration. More precisely, an event is one of the following:

- $p_i.inv(op)$, $p_i.rsp(op)$: invocation and response, respectively, of a high-level operation op at process p_i;
- ($p_i.local(m, p_j, m')$: p_i extracts a message m in its *message buffer* (storing all messages previously sent to it but not yet delivered), performs local computations, and puts a message m' to the message buffer of p_j;
- $p_i.receive(m)$: p_i receives a message m previously sent to p_i;
- $p_i.read(j)$: p_i reads S_j, the current switch function of port j;
- $p_i.add\text{-}rule(j, r)$: p_i adds rule r to S_j.
- $p_i.remove\text{-}rule(j, r)$: p_i removes rule r from S_j;
- ($inject(pk, j)$: the environment injects a packet pk to an ingress port j by replacing Q_j with $Q_j.pk$.
- ($forward(pk, j, pk', k)$, $j \notin \{\mathsf{World}, \mathsf{Drop}\}$: 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 $Q(k).pk'$, where $r(pk) = (pk', k)$ and r is the highest-priority rule in S_j that can be applied to pk. Recall that if there is a conflict, *i.e.*, two different rules applicable to pk have the highest priority, then the conflict is resolved using some deterministic procedure.

A finite execution unambiguously identifies a *final state*, *i.e.*, the configuration of the ports, the port queues and the processes.

Scheduling. We assume a *fair* scheduler and communication channels between the processes are *reliable* in the sense that in every infinite execution, there are infinitely many $forward(j)$ events for each port $j \notin \{\mathsf{World}, \mathsf{Drop}\}$, every message sent to a process is eventually received, and every correct process takes infinitely many *steps* (local events or port accesses).

Paths and per-packet consistency. Every packet injected to the network in an execution E (resp., a history H) 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, $forward(j)$ performed on (pk, j) adds (pk_1, j_1) to the trace, and each next $forward(j_k)$ performed on (pk_k, j_k) extends the trace with (pk_{k+1}, j_{k+1}). Note that in a finite network an infinite trace must contain a cycle.

Let $\rho_{ev,E}$ (resp. $\rho_{ev,H}$) denote the trace corresponding to an inject event ev in an execution E (resp., 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 $(i_1, , i_2, \ldots) \in \pi$.

Histories, problems, and algorithms. The *history* H of an execution E is a subsequence of it containing only invocation and responses of high-level operations, *forward* and *inject* events. A *problem* is a set $\mathcal{H}$ of desirable histories.

Each process p_i is assigned an algorithm, a state machine that accepts invocations of high-level operations, accesses ports with *read*, *add-rule* or *remove-rule* operations, communicates with other processes, and produces high-level responses. An algorithm solves a problem $\mathcal{H}$ if the history of its every execution is in $\mathcal{H}$. An algorithm solves $\mathcal{H}$ *t-resiliently* if the property above holds in every t-resilient execution, *i.e.*, in which at most t processes take only finitely many steps. An $(n-1)$-resilient solution is called *wait-free*.

4 The CPC Problem

At a high level, the 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 transactional interface where requests can be *committed* or *aborted*. Intuitively, once a request commits it affects every packet in its domain that is subsequently injected. But in case it cannot be composed with the currently installed set of histories, it is *aborted* and does not affect a single packet.

Formally, every process 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).

Now 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 succeeds* (resp., *precedes*) in H a request *req*, 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 no two requests are concurrent in H and no inject event is concurrent with a request.

Let $H|p_i$ denote the *local* history of process p_i, *i.e.*, the subsequence of H consisting of all events of p_i. We assume that every process is *well-formed*: every local history $H|p_i$ is sequential, *i.e.*, no process 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* or *nack* inserted somewhere after its invocation.

A sequential complete history H is *legal* if the set of committed policies in H is *conflict-free* and for every inject event $ev = inject(pk, j)$ in H, the trace $\rho_{H,ev}$ is consistent with the *composition* of all committed policies that precede *ev* in H.

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', $\rho_{ev,H} = \rho_{ev,H'}$.

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.*

Progress. *If a request* $req = apply_i(\pi)$ *completes with* $nack_i$ *in* H'*, then there exists* $req' = apply_j(\pi)$ *in* H'*, concurrent or committed before the response of* req *such that* π *and* π' *are conflicting (*$\{\pi, \pi'\}$ *is not conflict-free).*

Termination. *Eventually, every correct process* 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 Progress 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 process that installs it, either eventually takes effect or does not affect a single packet.

5 The Impossibility of Asynchronous Fault-Tolerant CPC

In this section, we show that the CPC problem is impossible to solve in the presence of even one crash failure and even for a network with a single port.

Suppose that the network consists of exactly one port that initially is configured to forward all the traffic to World (we denote this policy by π_0). Processes p_1 and p_2 accept two policy-update requests $req_1 = apply_1(\pi)$ and $req_2 = apply_2(\pi')$, respectively, such that π and π' are conflicting.

By contradiction and assume that there is a 1-resilient CPC algorithm A. A *descendant* of a state σ is the state of A that results after an extension of the execution corresponding to σ (the extension can be empty, *i.e.*, σ is also considered to be an extension of itself). A state is *i-valent* ($i = 0, 1, 2$) if in all its descendants, all packets in $dom(\pi_i)$ are only processed according to π_i. A state is *undecided* if it is not i-valent for some $i \in \{0, 1, 2\}$. By definition, any state that has an undecided descendant is also undecided.

Lemma 3 *In an undecided state, no process can commit its request and at least one process cannot abort its request.*

In the initial state of the algorithm, no process has started its update yet and, thus:

Lemma 4 *The initial state is undecided.*

Lemmas 3 and 4 establish the existence of a *critical* undecided state of A:

Lemma 5 *There exists an undecided state* σ *of* A *and a set* s_i *of a process* p_i *such that for every* σ'*, a descendant of* σ *in which* p_i *takes no steps after* σ*, the extension of* σ' *with* s_i*, denoted* $\sigma'.s_i$ *is* univalent *(*0-*,* 1-*, or* 2-*valent).*

Now we are ready to prove our impossibility result:

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

Proof. Suppose, by contradiction that a 1-resilient CPC algorithm A exists.

By Lemma 5, there exists an undecided state σ and a step s_i of p_i such that $\sigma'.s_i$ is univalent for each descendant σ' of σ in which p_i takes no steps after σ. Suppose that $\sigma.s_i$ is j-valent and let σ' be a descendant of σ such that $\sigma'.s_i$ is j-valent, where $j, k \in \{0, 1, 2\}$, $j \neq k$.

Consider the sequence of descendants of σ, $\sigma_0 = \sigma$, $\sigma_1 = \sigma.e_1$, ..., $\sigma_t = \sigma.e_1.e_2 \dots e_t = \sigma_t$. Since $\sigma_0.s_i$ is j-valent and $\sigma_t.s_i$ is k-valent, there exists $m \in \{0, \dots, t-1\}$ such that $\sigma_m.s_i$ is j-valent and $\sigma_{m+1}.s_i$ is k-valent.

Note that in s_i, p_i must access the port with a nontrivial operation (*add-rule* or *remove-rule*). Otherwise, if p_i crashes right after performing s_i, then no other process will be able to distinguish a descendant of $\sigma_m.e_{m+1}.s_i$ from a descendant of $\sigma_m.s_i$. But the two states have different valencies and thus packets should be processed in them using different policies—a contradiction.

By the construction of σ', e_{m+1} is an event of a process p_j other than p_i. Similarly, in e_{m+1}, p_j must modify the state of the port. Otherwise, there is no way to distinguish $\sigma_m.e_{m+1}.s_i$ and $\sigma_m.s_i.e_{m+1}$ are identical.

Consider $\sigma_m.e_{m+1}.s_i$ and $\sigma_m.s_i.e_{m+1}$. The rules modified by the updates in s_i and e_{m+1} result in the same state of the port, where conflicts between installed rules are resolved in exactly the same way. Thus, again, no future packet will be able to distinguish the two states and thus select a policy to comply with—a contradiction with the Consistency property.

Thus, no CPC algorithm can tolerate a single crash failure. ☐

6 Solving the CPC Problem

We show that a slightly stronger port model allows for a wait-free solution to the CPC problem. In this section, we assume that a port may support 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. We will refer to such a port as a *rmw-port*.

Formally, a port i supports the following operation: *rmw*(i, f), where f is function that is applied to the rules at i depending on their state. For example, f may involve adding a rule that attaches a new tag τ to the header of all incoming packets.

The Tag Protocol. We start with describing a simple protocol TAG that solves CPC *wait-free* (*i.e.*, tolerating up to $n-1$ failures) in the atomic *rmw*-port model. TAG specifies algorithm for *apply*$_i(\pi)$ by which process p_i installs a new policy π. Recall that each policy π defines a unique *forwarding path* for each ingress port, and for all packets matching its domain *dom*(π); it can hence be represented by a set of paths $\pi = \{\pi^{(1)}, \dots, \pi^{(s)}\}$.

The basic idea of TAG is to encode each possible forwarding path in the network that any policy may ever use, by its own tag. Let τ_k be the tag representing the k^{th} possible path. TAG 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 matches *any packet* with tag τ_m, and forwards the corresponding packets to the port i_{x+1} which is given by path. Note that TAG does not not modify the initial rules at the internal ports.

Upon receiving a new policy request π and before installing any rules, a process p_i executing TAG sends a message to *all* other processes p_j ($j \neq i$) informing them about the rules it intends to add to the ingress ports. This *policy message* is to ensure fault-tolerance: whenever a process p_j receives such a message, it starts installing the policy π of p_i as well (in addition to its own policies). Intuitively, as a result, every policy update that affects the traffic is eventually completed and serialized in an equivalent legal sequential history.

Let $\pi = \{\pi^{(1)}, \dots, \pi^{(s)}\}$ be the policy to be installed, and let $i_1, \dots, i_s$ be the respective ingress ports of the paths $\pi^{(1)}, \ldots, \pi^{(s)}$. To install π, TAG 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 $\pi^{(j)}$. However, since different policies from different processes may conflict, TAG imposes a strict order $\prec$ on all ingress ports: all processes will update the ingress ports (using atomic *rmw*) one-by-one according to $\prec$. Thus, conflicts are discovered already at the lowest-order port, and the conflict-free all-or-nothing installation of a policy is ensured. (The order $\prec$ may be based on switch or port identifiers.)

Theorem 7 TAG *solves the CPC problem wait-free.*

Observe that TAG 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.

However, while providing a correct network update even under high control plane concurrency and failures, TAG is very greedy in terms of message overhead and memory usage at the switches: given a good encoding of the k possible paths, $\Omega(\log k)$ bits are needed to uniquely describe a path and $\Omega(k)$ rules may have to be installed at each port.

Constant Tags and One Controller: The Bit Protocol. Can we implement concurrent policy updates with less tags? Intuitively, it seems that TAG is the best we can hope for: when a controller has no information on which packets are still in the network, it can never reuse a tag without risking to violate the per-packet forwarding consistency.

However, with a single controller ($n = 1$) and with some (even asynchronous) feedback from the network about the packets that left the system, network updates can also be performed with very short tags. Note that in a single-controller setting, we do not need atomic *rmw* support by the switches: it is sufficient that rules can be added and removed.

The algorithm BIT solves the CPC problem using a *single bit tag* only. For BIT to work, we will assume an oracle (e.g., a timer or reactive control in practice) which, when queried, informs the controller about the set of tags currently processed by the data plane, *i.e.*, the set of tags that are currently carried by the packets in the port buffers.

BIT proceeds as follows. When the controller receives a new policy p, the next available tag (modulo 2) is chosen from $\{0, 1\}$: $\tau_{\text{old}} = \tau_{\text{new}}$ and $\tau_{\text{new}} = \tau_{\text{old}} + 1 \bmod 2$. However, at this point, the tag τ_{new} may still be in use by old packets in the network, so by the per-packet consistency the forwarding rules cannot be changed yet. Only when the controller is notified that the last packet tagged with τ_{new} has left the system, it starts to modify the rules, according to the following *2-phase rule installation protocol* [12]: First, the new forwarding rules are added at all *internal* ports (using the tag τ_{new}). Subsequently, the following rule is added to all *ingress* ports: all arriving packets are tagged with τ_{new}. These two phases ensure an important invariant of BIT: when packets arrive in the network, the rules are already prepared for the new tag.

Theorem 8 *In a single controller scenario ($n = 1$) with feedback, the CPC problem can be solved with a single-bit tag.*

Tag Complexity: The Price Of Concurrency and Fault-Tolerance? While for a single controller, tags consisting of a single bit only are sufficient to solve the CPC problem (see the BIT protocol), the concurrent TAG protocol requires a tag space that can be exponential in the network size. This raises the question of whether there exist wait-free (or event t-resilient for $0 < t < n - 1$) protocols for $n > 1$ processes which only need constant tags, or whether a large (proportional to the number of controllers or the network complexity) tag space is an inherent price of concurrency and fault-tolerance in SDN control.

We argue that the required tag complexity critically depends on the feedback the controllers obtain about the network state. If controllers do not known anything at all about which tags are still used by packet inside the network, we conjecture that Tag is optimal in an asynchronous environment: a tag can never be *safely* reused without the risk of violating per-packet consistency.

We suggest to use an *oracle* model to formally reason about the information a process can obtain about the state of the network, in order to correctly install its policies. For example, as discussed for the Bit protocol, an oracle might provide an (asynchronous but FIFO) interface where processes can query the set of tags currently used by the packets in the network. Alternatively, an oracle may actively inform all processes about each packet (e.g., header and tag) that enters and leaves the network, so the processes may do a limited form of accounting themselves. In an even stronger model, one may assume that the oracle will even change the network state itself: for example, it may ensure that when no packets subject to a certain tag are left in the network, even the corresponding rules are deleted (e.g., are timed out).

We conjecture that for many relatively simple oracles, concurrent SDN control is impossible with tags τ of size $|\tau| = o(\log n)$. To give some intuition, assume an adapted Bit protocol where in addition to the single bit, processes even encode their unique identifier in the tag; moreover, before making any changes to the network state, processes always inform all other processes (using a *policy message*) about their planned policy updates with the corresponding tag. It may seem that even if a process fails during its policy update, other processes may take over and finish (or remove) this policy, given the policy message. However, other processes may not be able to distinguish an old policy from a new policy of this process; moreover, proactively finishing or removing the policy may be harmful. Intuitively, in case n processes concurrently install pairwise conflicting policies, a process must be able to gather $O(\log(n))$ of information to figure out which of the policies might have taken effect and, thus, must be eventually completed (to ensure Consistency, Progress, and Termination of the CPC abstractions).

7 Related Work

Distributed Computing. There is a long tradition of defining correctness of a concurrent system via an equivalence to a sequential one [6, 8, 11]. The notion of sequentially composable histories is reminiscent of linearizability [6], where a history of operations concurrently applied by a collection of processes is equivalent (indistinguishable for every process) from 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 policy-update request constitute a conflict-free sequential history, but, additionally, we expect that each *path* witnesses only a prefix of this history, consisting 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) [13], thus the term *software transactional networking*. Our interface is however intended to model realistic network management operations, which makes it simpler than more recent models of dynamic STMs [5]. Extending the interface to dynamic policies that adapt their behavior based on the current network state sounds like a promising research direction. On the other hand, our criterion of sequential composability is more complex than traditional STM correctness properties in that it imposes restrictions not only on the high-level interface exported to the control plane, but also on the paths taken by the data-plane packets. Also, we assumed that processes are subject to failures, which is usually not assumed by STM implementations.

Our impossibility result in Section 5 is inspired by the 1-resilient consensus impossibility proof [3]. But our notion of valency is defined having our two-layer nature of concurrency in mind, which makes the reasoning quite different.

Software Defined Networking. A study of SDN from a distributed systems perspective is given in Onix [7], a control plane platform designed to enable scalable control applications. Its contribution is to abstract away the task of network state distribution from control logic, allowing application developers to make their own trade-offs among consistency, durability, and scalability. 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. The costs of implementing logically centralized network control, providing various levels of consistency, over a distributed system are subjected to a sensitivity study in [9].

For the case of a single controller, Reitblatt et al. [12] formalized the notion of per-packet consistency, introduced the problem of *consistent network update*, and described the *two-phase update* technique, used in our BIT algorithm. We extend this work to the distributed control case. Note that while the solutions in [12] requires, in the asynchronous network, an unbounded number of policy tags, our TAG algorithm incurs bounded tags, regardless of the number of installed policy updates.

Failures and especially software errors are considered one of the main disadvantages of SDNs, see [2] for an overview. This paper appears to be the first to formally tackle the fault-tolerance aspect of SDN-enable networks. Finally, our work builds upon our workshop paper [1], where we introduced the notion of software transactional networking, and sketched a tag-based algorithm to consistently compose concurrent network updates. Assuming a fault-free environment, [1] provides a “proof-of-concept” algorithm implementing concurrent policy composition given the user provides semantics for the sequential one. This paper formalizes the notion of *fault-tolerant* policy composition and first derives impossibility results and complexity lower bounds.

8 Concluding Remarks

This paper has initiated the study of the SDN control plane as a distributed system. We argue that only a distributed implementation allows us to reap the SDN benefits without sacrificing availability and robustness properties.

We introduced a formal model that captures the fundamental aspects of concurrent and asynchronous network control and policy management, and facilitates a rigorous analytical study of the control plane properties. In particular, we have shown that our model also allows us to reason about the interfaces that SDN components must provide, in order to implement certain properties. It can be used by the networking community to find the right specification abstractions and APIs.

In general, this work is a first step towards a better understanding of the implications of implementing a distributed fault-tolerant network control plane. In Section 6, we discussed a set of challenging open questions, related to both formalizing the notion of *feedback* from the data plane provided to the controllers and deriving complexity and computability bounds of CPC solutions. Assuming that the feedback is precise but asynchronous, we conjecture that a wait-free CPC solution may require $\Omega(\log(n))$ bits in the tag field. As we conjecture further, the amount and quality of such a feedback play a crucial role in the message overhead and memory use.

Other research directions include extending this work to failure models beyond crashes, e.g., omission or even Byzantine failures, determining tight bounds on message size and memory usage, and exploring synchronization requirements on controllers and switches.

References

[1] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. Software transactional networking: Concurrent and consistent policy composition. In *Proceedings ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, August 2013.

[2] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test openflow applications. In *Proc. 9th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[3] 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.

[4] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *Computer Communication Review*, 38(3):105–110, 2008.

[5] 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.

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

[7] 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.

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

[9] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically Centralized? State Distribution Trade-offs in Software Defined Networks. In *HotSDN*, 2012.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

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

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

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

A Missing proofs

Lemma 1 *In an undecided state, no process can commit its request and at least one process cannot abort its request.*

Proof. Let σ be an undecided state. By the Consistency property, every state in which p_i ($i = 1, 2$) completes its update with an *ack* is i-valent. Thus, no process can commit its request in σ.

On the other hand, there is a descendant of σ in which some policy π_i ($i = 1, 2$) takes effect, and, by the Consistency property, p_i is not allowed to abort its request. ☐

Lemma 2 *The initial state is undecided.*

Proof. By the Termination, Consistency, and Progress properties, the initial state has a 1-valent descendant in which p_1 commits its request and p_2 is initially faulty and a 2-valent descendant in which p_2 commits its request and p_1 is initially faulty. ☐

Lemma 3 *There exists an undecided state σ of A and a a set s_i of a process p_i such that for every σ', a descendant of σ in which p_i takes no steps after σ, the extension of σ' with s_i, denoted $\sigma'.s_i$ is* univalent *(*0*-,* 1*-, or* 2*-valent).*

Proof. Suppose not. By Lemma 4, the initial state of A is undecided. Starting from the initial undecided state, we inductively construct an infinite fair execution that only goes through undecided states as follows.

Let σ be the finite execution we constructed so far (initially empty) and let p_i be the next process in $p_1, \ldots, p_n$, in the round-robin order (initially, p_1). If in its next step in σ, p_i performs a local event, then we let s_i be the step of p_i in which it extracts the "oldest" message in its message buffer and sends a message to another process according to its algorithm. If in its next step in σ, p_i accesses a port, we let s_i be the corresponding port event. By our assumption, there exists σ', a descendant of σ such that $\sigma'.s_i$ is undecided. Now we assign $\sigma = \sigma'$ and proceed to the next process in the round-robin fashion.

In the limit, we obtain an infinite fair execution E of our algorithm which only goes through undecided states and in which every process (including p_1 and p_2) takes infinitely many steps. Since, by Lemma 3, at least one of the processes p_i ($i = 1, 2$) cannot complete its request in an undecided state, E violates the Termination property of CPC. ☐

Theorem 4 *In the rmw-port model,* TAG *solves the CPC problem $(n-1)$-resiliently.*

Proof. Per-packet consistency follows from the facts that (1) the tags and rules at internal ports are static and do not change, and that (2) each packet is tagged at the ingress port only. Since there are rules matching this tag at the internal ports, the packet will be forwarded along exactly this path.

The eventually complete installation of a policy follows from the facts that (1) at a single ingress port, a rule is installed atomically using *rmw*, (2) due to the strict order $\prec$, each other process is aware of potentially previously installed policies and policy conflicts can be resolved, and (3) using the policy message, at least one surviving process can finish partially installed policies. ☐

Theorem 8 *In a single controller scenario ($n = 1$) with feedback, the CPC problem can be solved with a single-bit tag.*

Proof. The algorithm Bit is an example. By design, Bit only uses a single-bit tag. The algorithm is also correct: (1) The per-packet consistency property follows from the 2-phase rule installation protocol, the fact that only ingress ports tag packets, and the fact that rules at internal ports are only updated once the last old packet with the corresponding tag has left the system. (2) Any incoming packet is tagged at the ingress port without delay, and hence subject to at least one policy. (3) Since packets will eventually leave the system, and hence the controller is notified about the available tag, Bit also ensures that policies are eventually installed. Finally, (4) policy composition is trivial under a single controller. □