ENDEAVOUR: Towards a flexible software-defined network ecosystem

ENDEAVOUR

Project name	ENDEAVOUR
Project ID	H2020-ICT-2014-1 Project No. 644960
Working Package Number	2
Deliverable Number	2.1
Document title	Initial requirements of the SDN architecture
Document version	1.0
Editor in Chief	Canini, UCLO
Author	Bruyere, Canini, Castro, Chiesa, Dietzel, Kathareios, Nguyen
Date	15/12/2015
Reviewer	UCAM
Date of review	14/12/2015
Status	*Public*

Revision History

Date	Version	Description	Author
25/06/15	0.1	First draft	Canini (UCLO), Nguyen (UCLO)
02/07/15	0.2	Additional requirements	Castro (QMUL)
05/07/15	0.3	IXP environments	Bruyere (CNRS)
07/07/15	0.4	DE-CIX environment	Dietzel (DE-CIX)
07/07/15	0.5	Integration of Umbrella	Bruyere (CNRS), Canini (UCLO)
08/07/15	0.6	Review and minor changes	Antichi (UCAM), Canini (UCLO), Nguyen (UCLO)
03/12/15	0.7	Inclusion of analysis of switches	Kathareios (IBM)
10/12/15	0.8	Inclusion of preliminary architecture	Canini (UCLO), Chiesa (UCLO)
14/12/15	0.9	Review	Antichi (UCAM)
14/12/15	1.0	Final version	Canini (UCLO), Chiesa (UCLO)

Executive Summary

ENDEAVOUR addresses limitations of the network interconnection model in the current Internet and enables the next-generation services through SDN-enabled Internet eXchange Points (IXPs). So far, SDN has mostly been considered in intra-domain settings. Bringing SDN to the inter-domain settings would generate much more impact, both for network operators as well as for networked applications, *e.g.*, those deployed in the Cloud. IXPs provide the opportunity to access a very rich network/application ecosystem, by tapping into their function of "internetwork connectivity fabric", interconnecting multiple hundreds of networks of different types. This opportunity comes with many interesting and open questions.

In this context, this deliverable focuses on the requirements of IXP environments and general aspects and preliminary description of the SDN control architecture towards supporting the use cases identified in Deliverable 4.1. In particular, we focus on technical building blocks to the distributed SDN control plane that address certain aspects of the challenges of IXP environments that we identify. Future deliverables will bridge the current gaps and illustrate how the various building blocks can be combined to address specific use cases.

Contents

List of Figures

1 Introduction

An objective of ENDEAVOUR is to research, develop, and evaluate an SDN architecture for the network ecosystem of a large IXP and its members. Most of the state-of-the-art on SDN focuses on single tenant environments, mostly ignoring significant barriers for its adoption in large inter-domain environments: namely, scalability, reliability, and distributed management. Our approach to address these specific scientific and engineering challenges focuses on two main activities of design and implementation:

- **Distributed SDN Control Plane**: Design and implement a distributed SDN control plane that is capable of (1) supporting a multi-user environment, (2) tolerating failures, and (3) scaling to the typical user base of large IXPs.
- **SDN Programming Abstractions**: Design a scalable implementation of programming abstractions able to express (1) forwarding and QoS policies, (2) multi-authored policies including mechanisms such as composition and conflict resolution, and (3) monitoring primitives that, interfacing with the monitoring techniques defined by WP3, collect and expose rich measurement data for SDN applications.

This document presents the progress towards these objectives during the first 6 months of the project. The rest of this document is organized as follows. We first discuss relevant background regarding IXP environments and review the requirements that the SDN control plane should address. Note that since Deliverable 4.1 already focuses on the use cases and their requirements, we here focus on general aspects of the SDN control architecture (*e.g.*, scalability, reliability) that form a basis to support a plethora of specific use cases. Next, we describe the initial design of our SDN control plane. Since the project is still at its early stage, we focus on technical building blocks to the distributed SDN control plane that address certain aspects of the challenges mentioned above. Future deliverables will bridge the current gaps and illustrate how the various building blocks can be combined to address specific use cases.

2 Characteristics of IXP Environments

What is an Internet eXchange Point (IXP)? An IXP is "a physical network infrastructure operated by a single entity with the purpose to facilitate the

exchange of Internet traffic between Autonomous Systems. The number of Autonomous Systems connected should at least be three and there must be a clear and open policy for others to join." This definition is from the Euro-IX organization and it introduces the minimum service offering of an IXP, that is, an Ethernet switching platform supporting bridging domain which allows any-to-any interconnection between members.

To have access to all of "The Internet", Internet Service Providers (ISPs) buy transit connectivity from global service providers also called Tier 1 providers. Tier 1 providers offer reachability to almost every Internet's network and they sell this transit connectivity to smaller ISPs. Small ISPs are paying this transit access based on the amount of traffic exchanged as generally they are using multiple transit upstream providers with at least two Tier 1 providers to achieve redundancy and some degree of choice for outbound traffic forwarding. When two ISPs are directly connected to each other and exchange traffic among themselves, there is not transit cost. This solution is viable if there is considerable traffic between the two networks. Whilst upstream costs are reduced, there is a cost involved in providing physical connection between the two ISP's networks, and this must be considered when calculating the savings by having a direct interconnection. There are, however many thousands of ISPs in Europe alone. It would not be cost effective, scalable or manageable to interconnect with all of them individually.

Internet Exchange Points provide a solution to this problem. An IXP is a single physical network infrastructure, to which many ISPs can connect. Any ISP that is connected to the IXP can exchange traffic with any of the other ISPs connected to the IXP, using a single physical connection to the IXP, thus overcoming the scalability problem of individual interconnections. Also, by enabling traffic to take a more direct route between many ISP networks, an IXP can improve the efficiency of the Internet, resulting in a better service for the end user.

IXPs are not, generally, involved in the peering agreements between connected ISPs. Whom an ISP peers with, and the conditions of that peering, are a matter for the two ISPs involved. IXPs do however have requirements that an ISP must meet to connect to the IXP. Also, since the physical network infrastructure is shared by all the connected ISPs, and activities of one ISP can potentially affect the other connected ISPs, all IXPs have rules that establish the correct usage of the IXP.

Today's IXPs are all using Ethernet bridging with the MAC learning algorithm maintaining at least a single broadcast domain to enable the IXP members' routers to connect with each others. On top of this interconnecting

bridging domain, IXPs provide additional services, and adopt different technologies and/or architectures for scaling and securing their production environment. We review below possible technical solutions.

2.1 Interfaces Characteristics

IXPs have clear public rules for connecting to their infrastructure; this section reviews the fundamentals.

The customer interface. A clear demarcation point between the IXP services and the members is required. This can done either directly on the exchange or via a common demarcation point. This rule of demarcation is essential to determine the responsibility limits.

Ethernet physical interface. IXPs offer IEEE 802.3 Ethernet connectivity on a common switch infrastructure. Service offerings need to be available at least at the following IEEE defined rates (most seen rate first):

- 802.3z 1GE,
- 802.3ae 10GE,
- 802.3ba 40G/100G.

Media type could vary from copper to multi- or mono- mode fiber.

Traffic allowed to be forwarded. Only specific frames are switched by the fabric. The IXP fabric is forwards frames with the following Ethernet types:

- 0x0800 – IPv4,
- 0x86dd – IPv6,
- 0x0806 – ARP.

MAC filtering. For security reason and to limit any other MAC to send unauthorized traffic, IXPs apply MAC address locking mechanism at the member interface port. Only the authorized and well known router address member can be forwarded by the switching fabric.

Public VLANs. The IEEE 802.1q is a standard supporting Virtual LANs (VLAN), using Ethernet frame tagging techniques, permitting to have separated Layer 2 bridging domain on the same physical infrastructure (*e.g.*, IXPs use VLANs to separate between IPv4 and IPv6 traffic).

Private VLAN. Private traffic can be exchanged using a dedicated VLAN for two or more members who want to privately interconnect. The private VLANs use the same IEEE 802.1q standard but public traffic forwarded by the IXP switches need to have precedence over all private traffic. IXP members should dedicate and have separate physical interfaces for their private traffic.

2.2 Interconnecting Links and Topology

IXPs are located in large and economically developed cities, where ISPs and others operators (*e.g.*, data centers, content providers) have infrastructures.
Multiple Point of Presence architecture. IXPs are rarely present at a single location, also called Point of Presence (PoP). The PoPs are interconnected through various redundant path. Different architectures and distributed control plane protocol are used by IXPs to interconnect between PoPs.
Layer 0 – Optical network. The IXP's PoPs are interconnected with optical fibers, which are subjected to stringent operational requirements such as optical path redundancy, optical aggregation with wavelength multiplexing. These requirements have pushed IXPs to use complex optical equipment. Multiplexing and optical path failover technique are the primary feature used here.

2.3 Layer 2 – Resiliency of Connectivity

The IXP switching platform needs a backplane capacity sufficiently large to handle the aggregate traffic of all customers facing ports, without oversubscription. If individual switching elements contain multiple switch fabric modules, the same conditions apply during single component failures.

To maintain connectivity within the IXP fabric, IXPs typically use distributed Layer 2 protocols. We review below common technologies.
Spanning Tree. Spanning Tree is an old technology, but still the only cross-platform dynamic solution available to operators of IXPs for dynamically managing multiple redundant links in their architecture. The IEEE 802.1w Rapid Spanning Tree Protocol (RSTP) provides fast convergence in case of link failure. Member interfaces need to be configured as end-stations who are permitted to send frames without any convergence delay. RSTP has various drawbacks: for example, there is no load sharing between links, backup links are not used to forward traffic and in some conditions the convergence time is rather long.

Operational requirements have driven IXPs to look into new overlay architectures allowing them to resolve these scaling issues. The remainder of this section presents various solutions, several of which are already in use today.

Virtual Private LAN Services (VPLS). The current common state-of-the-art for providing a loop-free topology is VPLS, as defined in RFC4761 (VPLS using BGP signaling) and RFC4762 (VPLS using LDP signaling) [71]. VPLS works by creating an Ethernet broadcast domain on top of a mesh of Label Switched Paths (LSPs) in an MPLS (MultiProtocol Label Switching) network. In addition to providing a loop-free topology, VPLS also brings the possibility to balance traffic over multiple distinct paths in the network, so that redundant links are always used simultaneously.

Transparent Interconnect of Lots of Links (TRILL). TRILL [102] is another approach to optimize traffic flows in switched Layer 2 environments. Much like a VPLS-based topology, TRILL provides an optimal forwarding path through the network for unicast traffic in an "all links active" topology. One of the advantages of TRILL is that it does not require overlaying the Layer 2 service onto an IP substrate.

Virtual Extensible LAN (VXLAN). VXLAN [77] is a technique to tag frames and transport them with UDP. VXLAN discovers and authenticates tunnel endpoints dynamically end to end. The Border Gateway Protocol (BGP) control plane is used to learn and distribute both MAC and IP addresses to avoid the need for flood-and-learn mechanisms. VXLAN uses multicast or unicast to minimize flooding and mitigate ARP flooding.

Ethernet VPN (EVPN). EVPN is an Ethernet Layer 2 VPN (Virtual Private Network) [107] solution that uses BGP as control plane for MAC address signaling and learning over the network as well as for accessing topology and VPN endpoint discovery.

VPLS, VXLAN and EVPN are all running on top of a Layer 3 transport network. The transport network can be constructed using a traditional IGP routing protocol such as OSPF or IS-IS. These solutions come with overheads as they use tagging techniques to extended the network namespace.

2.4 Layer 3 Domain

The typical way to establish connectivity between two IXP members is to establish a direct BGP session between two of their respective border routers. Initially, if two IXP members wanted to exchange traffic via the IXP's switching fabric, they had to establish a bi-lateral BGP peering session

at the IXP. However, as IXPs grew in size, to be able to exchange traffic with most or all of the other members at an IXP and hence reap the benefits of its own membership, a member's border router had to maintain more and more individual BGP sessions. This started to create administrative overhead, operational burden, and the potential of pushing some router hardware to its limit.

Route Server (RS). To simplify routing for its members, IXPs introduced Route Servers [54, 59, 106] and offered them as a free value-added service to their members. In short, an IXP RS is a process that collects routing information from the RS's peers or participants (*i.e.*, IXP members that connect to the RS), executes its own BGP decision process, and re-advertises the resulting information (*i.e.*, best route selection) to all of the RS's peer routers.

If a route server service is offered, it supports both IPv4 and IPv6 and 4-byte AS numbers. The AS number used for the route server implementation is an unique AS number assigned by one of the RIRs. For redundancy, at least two RS are operated and are normally located in different PoP.

The IXP IP space. In order to be independent of any of the connected parties, the IP space used on the "Public Exchange" is a Provider Independent space or other IP space directly assigned by a IANA Regional Internet Register (RIR). This applies to both IPv4 and IPv6. The IXP operator is responsible for obtaining address space from the respective RIR, as well as providing all material for justification, documentation, and applicable fees as required by the RIR.

2.5 Characteristics of the DE-CIX Environment

In contrast to a generic IXP switching platform, the DE-CIX setup is generally more complex. The topology currently consists of seven edge switches (Alcatel Lucent 7950 XRS-20/40[1]) at different data centers across the entire city of Frankfurt and of four core switches. All edges are connected to all four core switches. Smaller points of presence with only a few members are connected to the DE-CIX infrastructure by smaller switches that are in turn connected to the edge switches. All the DE-CIX switches are realized with optical networking equipment (ADVA FSP3000R7[2]) whereby the single line speed is either 10 or 100 Gbps. The heaviest interconnects combine several 100 Gbps interfaces to accommodate up to 800 Gbps links. For

[1] https://www.alcatel-lucent.com/products/7950-extensible-routing-system

[2] http://www.advaoptical.com/en/products/scalable-optical-transport/fsp-3000.aspx

bundling several ports as a single logical link, the link aggregation control protocol (IEEE 802.3ad) is utilized.

On top of this physical infrastructure a transparent Layer 3 network (*e.g.*, MPLS, LSP, VPLS-L2) emulates a Layer 2 broadcast domain for the IXP members while providing the required redundancy and scalability in terms of coping with growing traffic volumes. Redundancy is as crucial for IXPs as for ISPs: it is paramount to be considered as a reliable provider of Internet connectivity. Additionally, load balancing over the core is too complex to be achieved with Layer 2 technologies only.

As the central control plane element, DE-CIX also operates a BGP route server. It announces about 65,000 IP prefixes while around 80% of its total traffic is sent towards those prefixes [106]. Considering these significant numbers, DE-CIX operates two redundant active route servers to whom the members must maintain an active BGP session. For further redundancy a single hot-standby hardware machine as an additional backup for either of the route servers is accommodated. All system operations, adoptions and maintenance decisions are performed with special regards to the importance of the route servers. For instance, the route server's configuration is generated every four hours from an internal repository and deployed to one active route server only. The new configuration is deployed in a soft reload that preserves BGP sessions but recalculates RIB information. A number of checks is performed to verify its faultlessness, *e.g.*, number of prefixes per member, size of RIBs. Eventually, to assure the quality of the announced routes (*e.g.*, only announcing own prefixes or ensuring next hop is the own IP) several filters based on IRR information and RADB are implemented.

2.6 Summary

IXPs are interconnecting ISPs with each other directly without the upstream transit costs of Tier 1 providers. Switching fabrics are the core service used by today's IXP. A variety of distributed Layer 2 protocols are used to create these fabrics, depending on scale requirements. The Layer 2 / Layer 3 dichotomy is crucial for keeping neutrality trustiness at the IXP fabric. IXP members want to keep their BGP configuration habits and be sure to be treated the same as their competitors for all peerings through the same IXP. IXPs offer Layer 3 services like Route Server to help their members with a single multilateral peering. As such, route servers play an important role for inter-domain routing in today's Internet. It already implements one important SDN paradigm: the data plane is separated from the control

plane. This is a promising starting point for ENDEAVOUR.

3 Requirements of the ENDEAVOUR SDN Architecture

In order to deploy SDN at IXPs, the SDN architecture needs to fulfill several requirements. These requirements include functional requirements (*i.e.*, the features required to control and manage the IXP network) as well as scalability and reliability requirements. These consist of high availability, high performance and resiliency to failures of different components.

Functional requirements. In Section 2, we reviewed certain functional requirements of IXP environments that are nowadays well understood. In addition, Euro-IX's wishlist [56] documents today's IXPs operational requirements and several recent measurement studies [2, 106] have also shed light on certain key requirements. For example, the IXP implements and exposes a simple Layer 2, plug and play semantics while it employs several state-of-the-art technologies to provide necessary capabilities. For instance, route servers are typically deployed at and operated by IXPs as a core element to enable IXP members to peer with one another in a scalable fashion.

In the 2015 annual forum of Euro-IX [40], Microsoft, a customer of IXPs, explicitly states the future desired features of an IXP. Many of them are conceptually similar in spirit to the fundamental element of the SDN architecture such as a centralized management approach via a single application programming interface (API) that supports operations and maintenance, and a robust API to retrieve statistics and utilization information.

Although the documented functional requirements might not fully consider new applications and services that could be realized for SDN-enabled IXPs (which ENDEAVOUR will consider), they provide an initial set of guidelines to approach the following problem: how could one operate an SDN-enabled IXP that is at least logically equivalent to an existing IXP?

Moreover, as emerges from Deliverable 4.1, the literature shows that beyond the functionality of current IXPs, members wish to customize inter-domain routing in order to achieve concrete objectives such to optimize traffic forwarding, select best peerings, and block DDoS traffic. On the other hand, IXPs wish to increase efficiency of their switching fabric, lower complexity and costs and offer new services.

In summary, we gather the following functional requirements:

- Expose a switching fabric with an equivalent Layer 2 semantics to IXP members.
- Maintain compatibility with BGP to exchange reachability information.
- Enable IXP members to customize or override default BGP routing behavior.
- Introduce a flexible data plane that enables fine-grained routing decisions, filtering and monitoring.

Scalability and reliability requirements. A main difference of SDN from the traditional network architecture is the decoupling of control plane from the data plane. The control plane configures the data plane via a standardized open interface such as OpenFlow [82, 94]. To provide better opportunities for network-wide optimizations and reduce management complexities, the control plane adopts a global view of network state, and so, the control is usually centralized and manages the data plane of the entire network.

However, this architecture creates a potential bottleneck at the controller and results in the problem of *controller scalability* which limits the number of tasks and the size of the network that can be served by a certain controller. This problem is carefully discussed in [118]. Similar to other large-scale systems, the scalability goes together with the problem on *reliability* in terms of keeping the service *highly available*, *high performance* and *resilient to failures*. Therefore, it is difficult to answer the question about the equivalence between an SDN-enabled IXP and an existing IXP without having a more precise, quantitative knowledge of its scalability and reliability.

Today it is hard to predict what target numbers will ultimately meet or exceed the bar for IXPs because these organizations have only recently started SDN trials in labs and not enough information regarding real world requirements exists. However, by the same token, IXPs will not deploy SDN in their networks unless they are guaranteed a high performance, resilient "IXP-grade" SDN control plane. On the other hand, the lack of a high performance SDN control plane platform has been a big barrier to SDN deployment and adoption.

This is the same problem currently faced by the ONOS project [91], driven by the ON.Lab. ONOS is a SDN network operating system for service

provider and mission critical networks, architected to provide a resilient, high performance SDN control plane featuring northbound and southbound abstractions and interfaces for a diversity of management, control, service applications and network devices. ONOS was open sourced on December 5th, 2014.

Indeed, paraphrasing from a study of ONOS performance [90], *building an "IXP-grade" SDN control plane that supports these requirements is a challenging design problem that requires thoughtful technical analysis of the trade-offs between high availability, performance and scale as all three are closely related.* Moreover, *high availability is a prerequisite for SDN adoption in IXPs. While it is great that one can scale a SDN control plane by introducing additional servers, it still needs to be complemented by the control plane's ability to automatically handle failures of individual servers. One should also be able to perform software and hardware upgrades without impacting overall system operation.*

Despite the lack of clearly documented performance and availability requirements, their study defines initial targets that according to the authors will meet or exceed the bar for Service Provider networks. Working with service providers has led them to the following characterization as a starting point:

- 1 Million flow setups/sec.
- Less than 100 ms latency for both topology change events and application requests (ideally, ∼10 ms or lower).
- Redundancy measures automatically take effect and the system continues to operate with zero downtime.
- The performance of the system remain nominal and is proportional to the number of resources currently at its disposal.
- When the system heals and the instances rejoin, the system automatically rebalances workload to take advantage of the restored capacity.

However, we only view this as a preliminary set of requirements that still need to be refined as the work in this project advances the understanding of specific use cases from which IXP operators and members stand to benefit. We also avoid in this deliverable to make specific final recommendations in terms of current SDN technology. We note instead that several viable candidates exist including ONOS, Ryu, OpenDaylight, and more.

The next section illustrates our preliminary SDN architecture. In Section 10, we discuss the landscape of commercially available SDN switches and how their characteristics currently address the switching fabric requirements.

In addition, there are a few prior efforts that attempted to clarify, motivate and address the requirements of SDN regarding high performance, availability and scalability aspects. We review a few relevant ones below.

SDX [47] is an earlier attempt at an SDN-enabled IXP. In this work, the authors proposed a virtual switch programming abstraction that allows IXP customers to describe their intended policy in a high-level language and to compile policies into forwarding rules. This work tackles some of the scalability issues by reducing the complexity of data plane state and reducing the compilation time of policies into forwarding rules. Consequently, it highlights the importance of reducing the latency to update a data plane state while maintaining consistency and isolation between different IXP members. Later in Section 7 and 8, we detail our proposals for addressing these problems.

To improve scalability and reliability of the service provider edge, another recent work, Edgeplex [28], demonstrated an approach that is based on sharding customer connections. The service provider edge is responsible for connecting customers using standard protocols such as IP and BGP to the service providers. In this respect, given the similarities with the Route Server service, these techniques inform us about the requirements and design principles for integrating the Route Server service in the SDN control plane as this should be a scalable and easy to manage yet reliable service. Our discussions with route server operators within DE-CIX indicated that with the introduction of SDN at the IXP, there would be interest to improve aspects of the Route Server service. While at this stage a solution is still premature, we are considering the concept of running the Route Server service as a distributed system that improves upon the current level of fault tolerance when a route server fails and enables intelligent load-balancing functionality.

As discussed, the IXP fabrics are Layer 2 broadcast domains, which have by nature some side effects. All hosts belonging to the same broadcast domain receive quite a significant amount of control packets (*i.e.*, ARP requests, DHCP requests, discovery protocols – CDP or LLDP). Broadcasts packets increase the switches CPU utilization and decrease the overall network security (*i.e.*, ARP spoofing). CDP (Cisco Discovery Protocol) packets in particular contain information about the network device, such as the software version, IP address, platform, capabilities, and the native

VLAN, increasing related security risks. Customer routers connected to IXPs typically exchange traffic with many other routers on the fabric. The larger the IXP, the higher the number of peers a router has. For all these peers, the ARP cache entry needs to be regularly refreshed. In addition, routers may have BGP sessions configured for peers that are not active. All together, the amount of broadcast ARP traffic on a large IXP fabric is already significant under normal circumstances. Even more ARP traffic is seen in downtime situations, when many routers attempt to resolve the IP addresses of peers that are not available because of a network outage (*i.e.*, ARP storm effect). Given the growing amount of location discovery traffic under normal conditions, we believe that control traffic reduction techniques are necessary when an exchange starts to scale. ARP-Sponge [117] represents a solution to this problem but it suffers of several limitations, which makes it undesirable for a large IXP. In Section 6, we show a preliminary design of an SDN control mechanism to resolve the broadcast domain flooding issues.

Finally, the recent work of Castro *et al.* [24] motivates us to adopt a future looking vision when thinking about the scale of the SDN control plane at an IXP, because SDN has great potential for enabling remote peering. Remote peering is an emerging type of interconnection where an IP network reaches and peers at a distant IXP via a Layer 2 provider, *e.g.*, using MPLS VPNs. The remote-peering provider delivers traffic between the Layer 2 switching infrastructure of the IXP and remote interface of the customer. On the customer's behalf, the remote-peering provider also maintains networking equipment at the IXP to enable the remote network to peer with other IXP members. Remote peering is present at a majority of IXPs worldwide, and many of the IXP members at the largest IXPs are indeed remote. By connecting distant networks and reselling port capacity at the IXPs, remote peering providers open the doors to a more flexible peering ecosystem. These considerations further highlight the importance for the need to meet scalability and reliability requirements in our design of the ENDEAVOUR SDN controller.

4 Preliminary SDN Architecture

At the core of the ENDEAVOUR architecture (see Figure 1), we envision an IXP fabric that consists of two main components: a set of SDN-enabled switches, which physically interconnects the IXP members with each other, and a "network controller" entity, which manages the SDN switches.

According to the SDN paradigm, the switches are responsible for forwarding packets according to their own forwarding state (i.e., the set of forwarding rules installed in each switch), which is not computed by the switches themselves. Instead, the forwarding state computation is performed by the *network controller*, which is a logically centralized independent entity that acts as the "brain" of the fabric. It computes the forwarding state of the network and installs it into each SDN switch.

The network controller provides to the IXP operators a high-level interface that can be used to deploy customized applications on top of it. Namely, the controller interface exposes to the network operator a logical view of the physical network topology, it presents a coherent and global picture of the network state, and it allows the operator to interact with the switches via a set of high-level primitives. Such primitives can, for instance, allow the network operator to move from one forwarding state to another one without creating routing anomalies (e.g. forwarding loops, blackholes). Roughly speaking, IXP operators leverage the network controller the same way programmers interact with the operating system. As an example, if an IXP operator is interested in deploying a novel service that provides "peering recommendations" to its IXP members, then it can program it by leveraging the interface exposed by the network controller.

In our vision, IXP members can benefit from these novel IXP services that can be built on top of the ENDEAVOUR architecture by using their own controller to communicate with the IXP controller. In any case, in order to support backward compatibility, the SDN-enabled IXP fabric should still handle interactions with IXP members that are running traditional protocols (i.e., BGP) although certain advanced features may not be supported. This can be done by integrating a Route Server within the network controller.

There are several benefits of replacing an old monolithic IXP fabric by an SDN-enabled fabric. First, it introduces a clear separation of concerns between data-plane (i.e., forwarding packets) and control-plane (i.e., installing the forwarding state) functionalities. By decoupling the control-plane functionalities from the physical switches to a logically centralized controller, the network architecture achieves high network modularity, which, in turn, leads to higher flexibility and ease of innovation. Second, it frees network operators from the burden of tweaking their network configurations by means of obscure and indirect mechanisms that are part of traditional routing protocols. Third, both the IXP operators and the IXP member operators can easily control the network behavior by writing their own application software on top of the network controller.

This architecture can therefore meet the requirements set forth earlier.

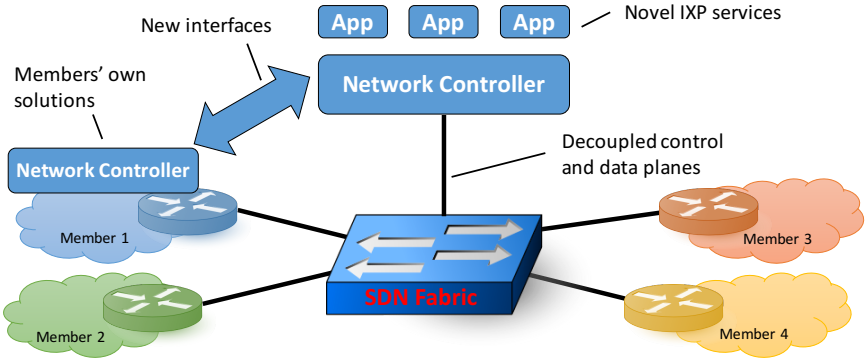


Figure 1: Preliminary architecture of an SDN-enabled IXP.

The SDN data plane brings programmability that enables fine-grained routing decisions, filtering and monitoring so that IXP members can override default BGP behavior. The SDN control plane lowers management complexity and enables new services, which can be realized through programs running at the controller or interfaces exposed to the IXP members.

5 Overview of Technical Building Blocks

We present several building blocks, which we expect are going to play an important role in the creation of the ENDEAVOUR SDN-enabled IXP. The first build block (Section 6) provides the design of a new IXP fabric architecture addressing the scalability issues of a shared broadcast domain. The next two building blocks (Section 7 and 8) focus on the problem of updating network states while providing certain guarantees including the correctness of the process and avoiding introducing congestion in the network. The fourth building block (Section 9) considers the issue of improving application performance by considering a co-design of a software-defined network and a distributed application — in our case, the Paxos consensus protocol.

5.1 Efficient IXP Fabric

IXP fabrics are growing steadily. As a consequence of their size, both the amount of broadcast and neighbor discovery (IPv6 related) traffic increases and exposes the SDN controller to the risk of becoming overloaded during ARP storm events.

In our first building block (Section 6) we introduce a new IXP fabric architecture, where we posit that the divide between the interconnection fabric and the content/service side requires a fundamentally different approach to the management of IXP fabrics. We argue that shifting intelligence from the control plane of current IXP fabrics to their data plane, through SDN programmability, is the key point to improve their scalability, reliability and manageability. We show how this delegation process can be effectively implemented taking as example the ARP management problem.

5.2 Abstractions and Architectures for Network State Updates

SDN is a paradigm that outsources the control of programmable network 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.

As discussed, correctness of forwarding behavior is a critical requirement for IXP environments. However, in today's IXPs, no technical solution prevents the possibility that misconfigurations by the IXP operator or even its members could bring down the IXP. Indeed, one of the project use cases (in Task 4.2) is focusing on resolving the broadcast storm issue by selectively filtering broadcast traffic within the IXP. Even very recently, in an accident on May 2015 [5], a misconfiguration by an engineer at AMS-IX placed a loop in the network that caused a disruption that lasted 10s of minutes. During this period, many parties could not exchange traffic with one another through our platform and therefore a number of websites were not accessible.

We believe that an SDN-enabled IXP provides the opportunity to develop rigorous, automated solutions to avoid several classes of such issues. However, the scale of the environment as well as its multi-user nature, make the problem challenging. Our approach consists of abstractions that isolate the intention of updating network state from its execution, and efficient yet correct by design strategies to implement network updates.

In particular, our second building block (Section 7) 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 (termed 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 $f + 2$.

Our third building block (Section 8) extends these concepts and provides a hierarchical distributed architecture that allows network update to be done in *decentralized* way. In which, the bottleneck at controller to coordinate every single step of update no longer exists, every switch communicates to notify the suitable time to do the update via a peer-to-peer data plane architecture.

5.3 Network-Application Co-Design

SDN offers the tantalizing promise of tailoring networks directly to the needs of distributed applications through increased programmability. ENDEAVOUR considers this question in the context of designing SDN-enabled mechanisms for defining and enforcing SLAs that span across multiple locations/members of the IXP (*e.g.*, customers connected to more than one datacenter). Unfortunately, because the existing standardized interfaces offer only limited functionality, few applications have been able to benefit from the open interface. Thus, the question remains how distributed applications can leverage SDN support, or more generally, what applications need from the network.

Our fourth building block (Section 9) explores the possibility of implementing the widely deployed Paxos consensus protocol in network devices. We present two different approaches: (*i*) a detailed design description for implementing the full Paxos logic in SDN switches, which identifies a sufficient set of required OpenFlow extensions; and (*ii*) an alternative, optimistic protocol which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages. Although neither of these protocols can be fully

implemented without changes to the underlying switch firmware, we argue that such changes are feasible in existing hardware. Moreover, we present an evaluation that suggests that moving Paxos logic into the network would yield significant performance benefits for distributed applications.

6 Umbrella Fabric

IXPs are typically implemented as very simple Layer 2 broadcast domains to which customers connect their BGP routers. In addition, many IXPs operate route servers [54], which facilitate multilateral peerings on the IXP fabric.

The biggest dangers to a Layer-2 broadcast domain are network failures and Ethernet loops who can cause the entire fabric to fail. Spanning tree systems or MPLS architecture do scale and can in some circumstances react quickly and gracefully to such circumstances. However in an IXP, we wish to optimize for a fairly static set of connected devices (changing only when a new customer router is physically installed or decommissioned), while still quickly reacting to network failures. To prevent loops, IXPs typically deploy MAC address based access control filters. These filters ensure that, on a customer port, only traffic from the MAC address of the connected customer router is accepted. This reduces noise from unwanted traffic on the broadcast domain and eliminates Ethernet loops. Because IXPs are relatively static environments, new MAC addresses only appear or disappear from the fabric when a new router is connected or when a router is disconnected. Since the access control filtering requires that all customer MAC addresses are known to the IXP operator, it is possible to use them to program the forwarding tables of the IXP fabric as well, eliminating the need for active MAC address learning on the fabric. This opens the space for an SDN-enabled infrastructure, where the controller can program the devices thanks to its global knowledge of the network.

IXP customer routers come in many forms and sizes. They also differ greatly in their operating system architecture. Some routers with weaker CPUs or older operating systems have problems handling the large amount of ARP traffic on larger IXP fabrics [117]. This would be significantly less problematic if ARP requests for a specific customer router were not sent to all customer ports. Ideally, the IXP fabric would send ARP and IPv6 Neighbour Discovery traffic only to the customer router for which the request is meant.

Previous works have already demonstrated that OpenFlow could solve

this problem [15, 100]. However, these solutions require an always active controller or software daemon that processes the ARP and Neighbor Solicitations, introducing scalability and stability concerns. The Umbrella fabric adds the feasibility of a new Layer 2 SDN-enabled IXP fabric addressing the issues of a shared broadcast domain using MAC manipulation to shift some intelligence from the control plane to the data plane. We justify the need of such an architecture to avoid the use of an active SDN controller (or software daemon) processing broadcast packets which becomes overloaded during ARP storm events. We envision a system where broadcast packets are being directly tackled in the datapath, and the controller is being only used to push the right rules into the switches thanks to its global knowledge of the network. We think that our approach, called Umbrella, can also improve the manageability and reliability of legacy IXPs. We demonstrate how the Umbrella fabric, under certain conditions, can be implemented with OpenFlow switches at the edge and legacy switches in the core minimizing the overall replacement cost.

In the following, we present the Umbrella architecture (§6.1) with its key benefits (§6.2). Finally, we discuss related works (§6.3) before giving a summary of this work (§6.4).

6.1 Umbrella Architecture

In this section, we present Umbrella, a new IXP fabric architecture that shifts some typical control plane operations to the data plane. The primary design goal is to enhance the scalability and stability of legacy IXP fabrics, taking advantage of the SDN paradigm. In particular, we aim for a network architecture able to address the issues of a shared broadcast domain, which scales on existing hardware, and does not require a central point of control to run.

6.1.1 No more broadcast traffic

IXPs apply strict rules [3], [56] to limit the side effects of a Layer 2 shared broadcast domain. They need to know the router MAC address of the member that connects to the peering fabric. Only then the IXP can allocate an ethernet port on the edge switch, an IP address from the peering IXP IP Public Space [92] and configure a MAC filtering ACL with that MAC address. As a consequence, the location of all the member's routers is known and does not change as dynamically as assumed by the Layer 2 protocols. This can be exploited to eliminate location discovery mechanisms

based on broadcast packets (*i.e.*, ARP request, IPv6 neighbor discovery). In particular, the OpenFlow specifications allow to rewrite the destination MAC address of a frame matching a given rule [94], enabling on-the-fly translation of broadcast packets into unicast at the edge of the fabric. To reduce the number of rules at the core switch level, we propose a new encoding scheme for the destination MAC address. Umbrella edge switches explicitly write the per-hop port destination into the destination MAC field of the packet. The first byte of the MAC address represents the output port the core switch has to use. Such kind of encoding scheme comes with a limitation: it is possible to represent a maximum number of 256 output ports per hop. This is not a real limitation though, as more bits in the port encoding (thus mapping more physical ports) can be used. With Umbrella, the number of flow table entries for a core switch will scale with the number of active ports in the IXP fabric. This aspect is important to tackle the address resolution problem directly from within the data plane.

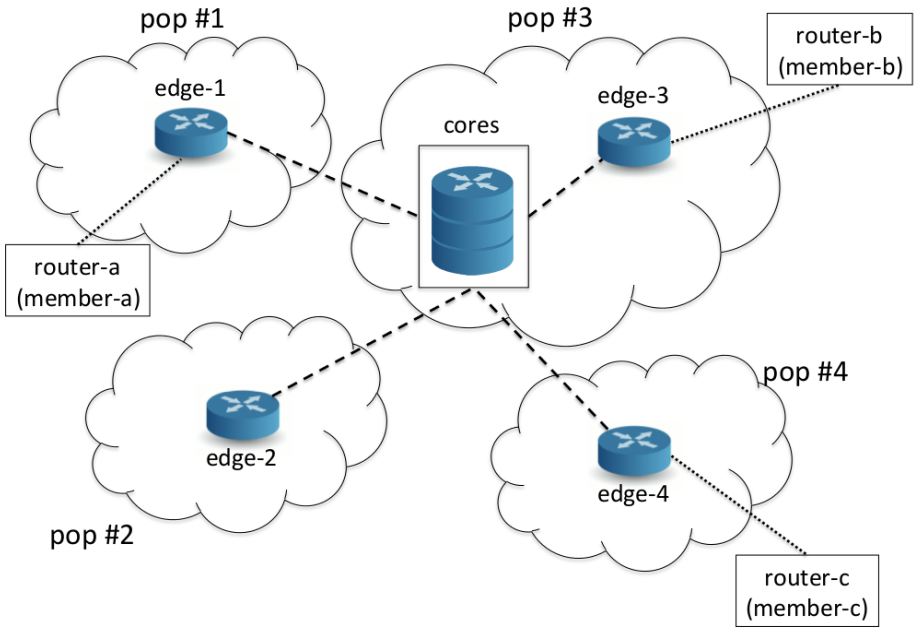


Figure 2: Typical topology of a medium to large IXP.

Let's take as example the topology shown in Figure 2 and consider the case where `edge-3` is connected to a core switch through port number 2 and to `router-b` through port number 3. Finally, let's take the case where

`router-a` sends an ARP request (*i.e.*, broadcast message) to `router-b`. `Edge-1` receives the frame, rewrites the destination MAC address using the following encoding: `02:03:00:00:00:00` and forwards it to the right core switch. Once the frame reaches the core, it is redirected to output port 2 (*i.e.*, the forwarding in the core is based on the most significant byte) to the `edge-3` switch. Finally, the `edge-3` switch forwards the frame through the output port indicated in the second byte of the MAC address and rewrites that field with the real MAC address of `router-b`, which is knowns. In case the source and destination are directly connected to the same edge switch, no encoding is needed, and the broadcast destination address is directly replaced by the target MAC destination address by the edge switch. In an IPv6 scenario, the OpenFlow match pattern indicated in the edge switch needs to be on the IPv6 ND target field of the incoming ICMPv6 Neighbour Solicitation packet [84]. The matching table on the edge switch should maintain an association between IPv6 addresses and their location, as in the IPv4 case.

6.1.2 Towards a segment routing-like approach

The proposed forwarding mechanism allows to reuse legacy switches in the core, thus limiting the burden (and costs) to upgrade an IXP fabric to the Umbrella architecture. In this scenario, a core switch just needs to forward packets based on simple access filtering rules, while the edge switches need to have OpenFlow-like capabilities to rewrite the Layer2 destination field.

While this approach is directly applicable to fabrics that rely on a single hop in the core (as in AMS-IX and DE-CIX), it is not with multiple hops (as in LINX and MSK-IX). With a single hop, the core switch would expect the output port encoded in the most significant byte of the destination MAC address. In the multi-hop case, since a packet can traverse multiple core switches, a new encoding scheme is needed to differentiate the output ports at different core switches.

Figure 3 shows an example with multiple hops in the core. To reach `edge-d`, `edge-a` needs to cross two different core switches through `path b`. This is a fairly common case in hypercube-like topologies, as the ones adopted by LINX or MSK-IX. In this scenario, following the Umbrella approach, it is straightforward to propose an encoding of the L2 destination address where the most significant byte refers to the output port of the first core switch (*i.e.*, `core-a`), the second byte to the second switch (*i.e.*, `core-b`), and so on. Unfortunately, depending on the actual route being used, a core switch might be the first or the second on the path, making the

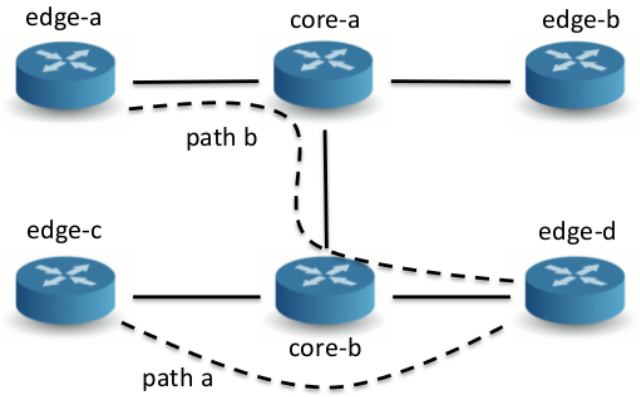


Figure 3: Example of multi-hop in the core.

proposed approach unfeasible. Another solution is to take into account also the `input port` of the frame in the forwarding rules installed in the core switches. Given the input port, it is possible to know where the switch is in the path and therefore looking at the right byte into the L2 destination address. Unfortunately, this approach may not work in arbitrary topologies. Moreover, it experiences a rule explosion in the core, *i.e.*, the number of forwarding entries grows quadratically with the number of possible input ports, making such an idea not very attractive.

These problems can be addressed using a segment routing-like approach. Segment Routing leverages the source routing paradigm, keeping the Umbrella spirit, where the first edge switch is in charge of selecting the path. Segment Routing consists in each node steering a packet through an ordered list of instructions, called segments, in this case the output ports. An ordered list of segments is encoded as a stack of labels. The segment to be processed is on the top of the stack, and popped upon completion of a segment.When a new frame reaches the fabric, it has to pass through a first edge switch in charge of rewriting the MAC destination address with a ordered list of `output ports`. Each port refers to a different core switch on the path towards the destination.When a core switch receives the frame, it looks up the most significant byte of the address to get the destination output port and rewrites the address by shifting the value to make the second byte the new most significant one. Each switch needs only to look at the most significant byte of the address, no matter where it is on the path toward the destination. After the lookup, the address must also be rewritten, making

this solution feasible only when OpenFlow-enabled switches are used in the core. Every core switch must have 2 action tables: forwarding and copy-field. This solution comes with two main practical limitations:

- The maximum number of output ports that can be addressed per-hop is 256, as we embed the output port for each core switch in the most significant byte of the Layer2 destination address.
- The maximum number of hops inside the IXP must be less of 6, as we use the 6 bytes of the MAC address to embed at the edge the overall path of the frame.

Beyond these practical considerations, Umbrella should actually be seen more as a generic approach for IXP fabric operation, rather than a specific solution to Layer2 issues. We believe that the general concept of Umbrella is its main strength, *i.e.*, offloading the control plane with a more intelligent data plane.

6.2 Key benefits

Umbrella has been designed to be flexible. Indeed, it can be made Layer 3 neutral or service (*i.e.*, application) oriented, depending on the settings being used. It addresses the issues of a shared broadcast domain using Layer 2 manipulation at the edge to enhance legacy fabric reliability. This section discusses the key benefits that we believe Umbrella brings at the IXP.

Scalability. Most legacy IXP architectures alleviate ARP storms through the ARP Sponge server approach. At the same time, pure Layer 2 SDN-based architectures leave to the controller the processing of location discovery traffic (*i.e.*, ARP and NDv6). Both solutions suffer from scalability issues given the growth of IXP fabrics in terms of new MAC addresses leading to a constant increase of broadcast traffic.

No central point of failure. Solutions relying on a single point for common operations, *i.e.*, the controller for SDN-based architectures and ARP Sponge server for legacy IXPs, are subject to the single point of failure problem. Umbrella does not need the constant presence of the controller for such operations. The controller works in a pure proactive mode and it is required only to add, remove or change a router MAC entry at the edge. Given the static nature of IXPs in terms of routing [4], the controller does not have a central role in the Umbrella approach.

Service-orientated IXP operators. The segment-routing nature of the forwarding mechanism opens the possibility of making the IXP

fabric service-orientated. A service-orientated IXP operator could create catalogs of network resources and related policies (*e.g.*, QoS parameters and bandwidth) to which applications can be applied as they move into the network. As the path inside the IXP fabric is configured at the edge switch, it is possible to configure different paths for different applications, or redirect some flows to different paths depending on the activated services (*e.g.*, firewalling, quality of service, monitoring). Note that this is just a feature that *can* be enabled. Indeed, Umbrella can also be used in a totally Layer 3 (and above) neutral way, as currently done by IXPs.

Compatibility with legacy switches. As discussed in the previous section, if the topology being used in the IXP has only one hop in the core, legacy switches with MAC policy based routing access lists can be used in the core with Umbrella. Indeed, no additional feature than bitmask Layer 2 destination matching and forwarding actions are applied in the core, thus making the architecture compatible even with non-OpenFlow compliant switches.

Pseudo-wire nature. Pseudo-wire[3] is an emulation of a point-to-point connection over a packet-switching network. As discussed above, with Umbrella, all the broadcast traffic (both ARP IPv4 and ICMPv6 ND) is converted to unicast at the edge, solving problems related to a shared broadcast domain. Umbrella guarantees that each of the IXP members receives only the traffic it is supposed to see, also saving computational power at the edge for the processing and analysis of unwanted traffic (*i.e.*, broadcast packets).

Visibility. In Umbrella, the actual path of packets is encoded in the Layer2 destination address. This implies full visibility of the forwarding paths inside the IXP fabric, which can be exploited to improve data plane troubleshooting, and therefore general IXP operator management.

6.3 Related Works

The idea of introducing OpenFlow in the IXP world is recent. Gupta *et al.* [47] aims to develop an SDN exchange point (SDX) to enable more expressive policies than conventional hop-by-hop, destination-based forwarding. The proposed solution shows that it can implement representative policies for hundreds of participants who advertise full routing tables while achieving sub-second convergence in response to configuration changes and routing updates. However, it is not clear how problems related

[3] `http://tools.ietf.org/html/rfc3985`

to Ethernet loops and large amount of ARP traffic are handled.

Stringer *et al.* [114], with the Cardigan project, implement a hardware based, default deny policy, capable of restricting traffic based on RPKI verification of routes advertised by devices connected to the fabric. While this approach offers the required protections for a stable IXP fabric, it is less suitable for IXPs that wish to remain neutral with regards to IP forwarding.

Enabling MAC-based routing in OpenFlow-based network is a fairly new field of research. Schwabe *et al.* [110] show that the destination MAC address can be used as a universal label in SDN environments and the ARP caches of hosts can exploited as an ingress label table, reducing the size of the forwarding tables of network devices. Agarwal *et al.* [1] demonstrate that, using destination MAC addresses as opaque forwarding labels, an SDN controller can leverage large MAC forwarding tables to manage a plethora of fine-grained paths. While these approaches have been shown to perform well in large-scale networks, they seems less suitable in IXP environments, where problems related to Ethernet loops and large amount of ARP traffic limit the scalability of the entire system (especially the SDN controller). To this end, we address in this paper the issues of a shared broadcast domain using Layer 2 manipulation at the edge.

6.4 Summary

We introduced the design of Umbrella: a new IXP fabric architecture. The primary motivation behind Umbrella was to directly address today's IXPs operational requirements, as expressed in Euro-IX's wishlist [56] focused on reliability and network management. We designed a new fabric for IXPs that fullfills their technical and operational requirements as well as their culture (*e.g.*, neutrality). To design such a fabric, we proposed an OpenFlow-based IXP network architecture. Umbrella takes advantages of the SDN programmability to address the issues of a shared broadcast domain, shifting some intelligence from the control plane of current IXP fabrics to their data plane. We introduced a new MAC base routing in the core, using Layer 2 manipulation at the edge to enhance legacy fabric reliability. It is scalable, enhances the current fabric visibility and can be used in a totally Layer 3 (and above) neutral way (as currently done by IXPs), or could be used in the future for service-oriented IXPs. We see Umbrella as a first step towards SDN architectures less dependent on the control plane, providing reliability by exploiting the data plane capabilities of SDN.

7 Transactional Network Updates

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 *data plane* (realized by programmable, packet-forwarding switches). 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. Designing 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 work, 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 [42] 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 [6]). Therefore, we address here the challenge of composing *concurrent* updates, while preserving a property known as *per-packet consistency* [105]. 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 update requests to be *rejected* entirely,

and requiring that no data packet is affected by a rejected update.

We make the following contributions. 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 [105], 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) [112], 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 term this approach *software transactional networking* [20].

We then discuss different protocols to solve the CPC problem. We present FixTag, a *wait-free* algorithm that 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 possible policies and their induced paths (which may grow to super-exponential in the network size).

We then present a more sophisticated protocol called ReuseTag, which uses the replicated state-machine approach to implement a total order on to-be-installed policy updates. Given an upper bound on the maximal network latency and assuming that at most f controllers can fail, we show that ReuseTag achieves an optimal tag complexity $f + 2$.

Our work also informs the networking community about what can and cannot be achieved in a distributed control plane. In particular, we 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 [51], complex interleavings between the controllers' actions and packet-processing events prevent them from implementing CPC with constant tag complexity (achievable using one reliable controller).

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 focus on a restricted class of forwarding policies, which is sufficient to highlight intriguing connections between our SDN model and conventional distributed-computing models, in particular,

STM [112]. One can view the data plane as a shared-memory data structure, and 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 (unless explicitly intended) nor delayed, these read-only transactions must always commit, in contrast with policy update transactions. This model hence introduces an interesting new kind of atomicity requirement.

Put in perspective, our definition of concurrent and consistent composition of policy updates can be seen as an instance of *transactional* network management. Indeed, in a dynamic system, where both control and data plane are subject to changes (policy modifications, workload variations, failures), it is handy for a control application to have operations with atomic (all-or-nothing) guarantees. This way control applications may "program" the network in a sequential manner, maintain consistent evaluations of network-wide structures, and easily compose network programs [22].

In the following, we first introduce our model (§7.1). We then formulate the CPC problem (§7.2) and describe our CPC solutions and their complexity bounds. We then show that under weaker port models, it is impossible to solve CPC (§7.4). We discuss related work (§7.5) before giving a summary of this work (§7.6).

7.1 Modeling Software-Defined Networks

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 modeled 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. A 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 [43] that allows them to implement, in a fault-tolerant manner, any replicated state

machine, provided its sequential specification [51].[4]

Data plane. Following [105], we model the *data plane* as a set P of *ports* and a set $L \subseteq P \times P$ of directed *links*. 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). 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 [105], *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 a *priority level*, as

[4]The consensus abstraction can be obtained, *e.g.*, assuming eventually synchronous communication [39] or the *eventual leader* (Ω) and *quorum* (Σ) failure detectors [26, 37].

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, for each ingress port, a unique *forwarding path*, *i.e.*, a loop-free sequence of piecewise connected ports that the packets in $dom(\pi)$ arriving at the ingress port should follow. 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 treated according to the highest priority policy $\pi \in U$ such that $pk \in dom(\pi)$.

Modeling traffic. The traffic workload on our system is modeled 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 $Q_j \cdot pk$ (*i.e.*, we add pk to the end of the queue).
- $forward(pk, j, pk', k)$, $j \in P$: the first packet in Q_j is processed according to S_j, *i.e.*, if $Q_j = pk \cdot Q'$ (*i.e.*, pk is the first element

of the queue), 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 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 correct 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 $ev = \mathit{inject}(pk, j)$ in E results in (pk, j) as the first element of the sequence, $\mathit{forward}(pk, j, pk_1, j_1)$ 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 $j_k \in \{\mathsf{Drop}, \mathsf{World}\}$ in which case we extend the trace with (j_k) and 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 $ev = \mathit{inject}(pk, j)$ in a history H. A trace $\rho = (pk_1, i_1), (pk_2, i_2), \ldots$ is *consistent with a policy* π if $pk_1 \in \mathit{dom}(\pi)$ 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 data plane switches to distinguish packets that should be processed by different policies. Throughout this work, 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 work: 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.

7.2 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

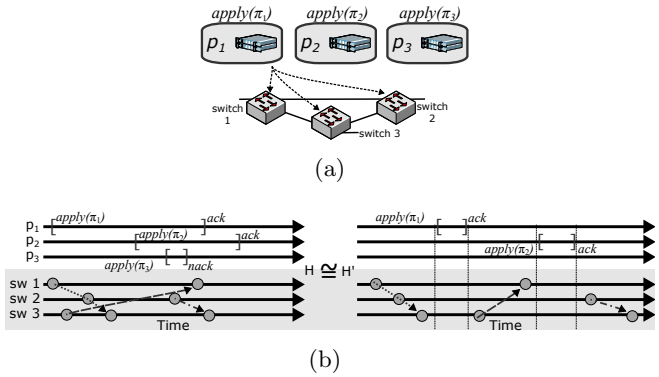


Figure 4: Example of a policy composition with a 3-controller control plane and 3-switch data plane (a). The three controllers try to concurrently install three different policies π_1, π_2, and π_3. We suppose that π_3 is conflicting with both π_1 and π_2, so π_3 is aborted (b). Circles represent data-plane events (an *inject* event followed by a sequence of forward events). Next to the history H (shown on (b) *left*) we depict its "sequential equivalent" H_S (shown on (b) *right*). In the sequential history, no two requests are applied concurrently.

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

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 4 gives an example of a sequentially composable history.

7.3 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, called FixTag, which implicitly orders policies at a given ingress port. FixTag incurs a linear tag complexity in the number of all possible paths that the proposed policies may stipulate; this is the best we can hope for any protocol without feedback from the network. Then we present ReuseTag, an f-resilient algorithm with tag complexity $f + 2$, which is based on an estimate on the maximal packet latency. We also show that ReuseTag is optimal, *i.e.*, no CPC solution admits smaller tag complexity for all networks.

7.3.1 FixTag: Per-Policy Tags

The basic idea of FixTag is to assign a distinct tag to each possible forwarding path that any policy may ever use. Let τ_k be the tag representing the k^{th} possible path. FixTag assumes that, initially, for each internal port i_x that 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 to the path's successive port i_{x+1}.

FixTag works as follows. Upon receiving a new policy request π and before installing any rules, a controller p_i sends a message to *all* other controllers informing them about the policy π it intends to install. 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.

Let $i_1, \ldots, i_s$ be the set of ingress ports, and π^j be the path specified by policy π for ingress port i_j, $j = 1, \ldots, s$. To install π, FixTag adds to each ingress port i_j a rule that tags all packets matching the policy domain $dom(\pi)$ with the tag describing the path π^j. However, since different policies from different controllers may conflict, we require that every controller updates the ingress ports in a pre-defined order. Thus, conflicts are discovered already at the lowest-order ingress port,[5] and the conflict-free all-or-nothing installation of a policy is ensured.

The use of reliable broadcast and the fact that the ingress ports are updated in the same order imply the following:

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

Observe that FixTag does not require *any* feedback from the network on when packets arrive or leave the system. Controllers only coordinate implicitly on the lowest-order ingress port. Ingress ports tag all traffic entering the network; internally, packets are only forwarded according to these tags.

However, while providing a correct network update even under high control plane concurrency and failures, FixTag has a large tag complexity. Namely, this depends in a linear fashion on the number of possible policies and their induced network paths, which may grow to exponential in the network size. Note that this is unavoidable in a scenario without feedback—a tag may never be safely reused for a different path as this could always violate CPC's consistency requirement.

[5] Recall that in our model failures do not affect the data plane; therefore, ports do not fail.

In practice, rules may be added lazily at the internal ports, and hence the number of rules will only depend on the number of different and *actually used* paths. However, we show that it is possible to exploit knowledge of an upper bound on the packet latency, and *reuse* tags more efficiently. Such knowledge is used by the algorithm described in the next section to reduce the tag complexity.

7.3.2 ReuseTag: Optimal Tag Complexity

The ReuseTag protocol sketched in Figure 5 allows controllers to reuse up to $f + 2$ tags dynamically and in a coordinated fashion, given a minimal feedback on the packets in the network, namely, an upper bound on the maximal network latency. As we 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 [51]), which imposes a global order on the policy updates and ensures a coordinated use and reuse of the protocol tags. For simplicity, we assume that policies 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*(p_i, π), where π is a policy, that always returns `ok`;
- • *pull*(p_i) that returns $\perp$ (a special value interpreted as "no policy tag is available yet") or a tuple (π, tag), where π is a policy and $tag \in \{0, \ldots, f + 1\}$.

Intuitively, p_i invokes *push*(p_i, π) to put policy π in the queue of policies waiting to be installed; and p_i invokes *pull*(p_i) to fetch the next policy to be installed. The invocation of *pull* returns $\perp$ if there is no "available" tag (to be explained below) or all policies pushed so far are already installed; otherwise, it returns a tuple (π, tag), informing p_i that policy π should be equipped with tag *tag*.

The sequential behavior of PS is defined as follows. 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*(p_i) operations in S (performed by p_i).

If S contains exactly k *non-trivial* (returning non-$\perp$ values) *pull*(p_i) operations, then we say that p_i *performed* k *non-trivial pulls in* S. If S

contains *pull*(p_i) that returns $(\pi, t) \neq \perp$, followed by a subsequent *pull*(p_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*(p_i) that returns (π_{k+1}, τ_{k+1}) but does not contain a subsequent *pull*(p_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*(p_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 kth pull of p_i must return some policy if at least k policies were previously pushed and at most f of their tags 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 non-trivial pulls, then $\pi_{i,k} = \pi_{j,k} = \pi_k$ and $\tau_{i,k} = \tau_{j,k} = \tau_k$. Therefore, the kth pull of any controller must return the kth pushed policy π_k equipped with τ_k.

Tag validity: For all $k > 0$, 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*(p_i) operation that returns (π_k, τ_k) is performed. Here τ_0 denotes the tag of the initially installed policy. The intuition here is that the tag for the kth policy is chosen deterministically based on all the tags that are currently not blocked and different from the previously installed tag τ_{k-1}. By the Non-triviality property, at most f tags are blocked when the first controller performs its kth non-trivial pull. Thus, $\{0, \ldots, f + 1\} - \{\tau_{k-1}\}$ contains at least one non-blocked tag.

In the following, we assume that a *linearizable* f-resilient implementation of PS is available [53]: 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 implementation 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 5 and operates as follows. To install policy $\tilde{\pi}$, controller p_i first pushes $\tilde{\pi}$ to the policy queue by invoking PS.*push*($p_i, \tilde{\pi}$).

```
Initially:
    seq := ⊥; cur := ⊥

upon apply(π̃)
1   cur := π̃
2   PS.push(p_i, π̃)

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

Figure 5: The ReuseTag algorithm: pseudocode for controller p_i.

In parallel, the controller runs the following task (Lines 3-11) to install its policy and help other controllers. First it keeps invoking PS.$pull(p_i)$ until a (non-$\perp$) value (π_k, τ_k) is returned (Line 3); here k is the number of non-trivial pulls performed by p_i so far. The controller checks if π_k is 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 7.1) that produces the set of currently active policies.

The controller then tries to install π_k on all internal ports first, and after that on all ingress ports, employing the "two-phase update" strategy of [105] (Line 9). The update of an internal port j 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 j. The update on an ingress port j 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, this controller may later update a port with an outdated rule, since it might not be able to distinguish the old policy with tag τ_k from a new one using the same tag. But a slow or faulty controller can block at most one tag; hence, 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 non-trivial 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.*

Proof. 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 $push(p_i, \pi)$ 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 tags 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 $\pi_{\ell-1}$ 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 [105] 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. □

Optimizations and Improvements. 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.

Tag Complexity: Lower Bound. 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 every 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, $\pi_0, \ldots, \pi_{f+1}$, that have overlapping domains but prescribe distinct paths. Assuming that only $f + 1$ tags are available, we construct an execution of the assumed algorithm in which an update installing policy π_i invalidates one of the previously installed policies, which contradicts the Consistency property of CPC.

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 $f + 2$.*

Proof. Assume the network T_f of two ingress ports A and B, and $f + 1$ "loops" depicted in Figure 6 and consider a scenario in which the controllers apply a sequence of policies defined as follows. Let π_i, $i = 1, \ldots, f+1$, denote a policy that, for each of the two ingress ports, specifies a path that in every loop $\ell \neq i$ takes the upper path and in loop i takes the lower path (the

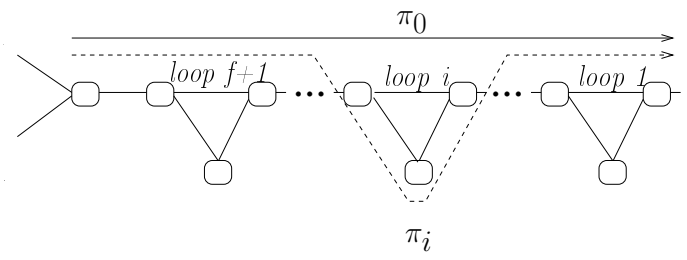


Figure 6: The $(f+1)$-loop network topology T_f.

dashed line in Figure 6). The policy π_0 specifies the path that always goes over the upper parts of all the loops (the solid line in Figure 6).

We assume that for all $i \in \{0, \ldots, f\}$, we have $pr(\pi_i) < pr(\pi_{i+1})$ and $dom(\pi_i) \supset dom(\pi_{i+1})$, *i.e.*, all these policies are composable, and adding policy π_{i+1} to the composition $\pi_0 \cdot \pi_1 \cdots \pi_i$ makes the composed policy more refined. Note that, assuming that only policies π_i, $i = 0, \ldots, f+1$, are in use, for each injected packet, the ingress port maintains one rule that tags and forwards it to the next branching port.

Without loss of generality, let 0 be 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_j, $j = 1, \ldots, i$, is about to access an ingress port with an update operation that, if the currently installed rule uses $j-1$ to tag the injected packets, replaces it with a rule that uses j instead.

For the base case $i = 1$, assume that some controller 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 controller q_1 has updated the tag on one of the ingress port with tag 1 and is just about to update the other port. Now we "freeze" q_1 and let another controller complete the update of the remaining ingress port. Such an execution exists, since the protocol is f-resilient ($f > 0$) and, by the Consistency and Termination properties 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 consider $2 \leq i \leq f+1$ and, inductively, consider the execution E_{i-1}. Suppose that some controller in $\Pi - \{q_1, \ldots, q_{i-1}\}$ completes its ongoing

policy update and now 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 controller in $\{q_1, \ldots, q_{i-1}\}$ takes a step after E_{i-1} and eventually some controller $q_i \notin \{q_1, \ldots, q_{i-1}\}$ 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 controller $q_{\tau+1}$ 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 controller 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. □

7.4 Impossibility for Weaker Port Model

It turns out that it is impossible to update a network consistently in the presence of even one crash failure, which justifies our assumption that SDN ports support atomic read-modify-write operations. To prove this impossibility, we assume here that a port can only 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, 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 controllers p_1 and p_2 accept two policy-update requests $apply_1(\pi_1)$ and $apply_2(\pi_2)$, 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 ingress port 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 $apply_1(\pi_1)$ and $apply_2(\pi_2)$ must appear as committed in the equivalent sequential history.

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. □

7.5 Related Work

Distributed SDN Control Plane. We are not the first to study *distributed* designs of the logically centralized SDN control plane. Indeed, the perception that control in SDN is centralized leads to concerns about SDN scalability and resiliency, which can be addressed with distributed control plane designs [118]. Onix [66] 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. There are also several studies on the design of spatially distributed control planes, where different controllers handle frequent and latency critical events closer to their origin in the dataplane, in order to improve scalability and latency [49, 50, 108]. ElastiCon [38] proposes an elastic distributed controller architecture. We in this work, in contrast, do not consider spatial optimizations but focus on robustness aspects.

Network Updates and Policy Composition. The question of how to consistently update networks has recently attracted much attention. Reitblatt *et al.* [105] formalized the notion of per-packet consistency and introduced the problem of *consistent network update* for the case of a single controller. Mahajan and Wattenhofer [76] considered weaker transient consistency guarantees, and proposed more efficient network update algorithms accordingly. Ludwig *et al.* [75] studied algorithms for secure network updates where packets are forced to traverse certain waypoints or middleboxes. Ghorbani *et al.* [44] recently argued for the design of network update algorithms that provide even stronger consistency guarantees. Finally, our work in [20] introduced the notion of software transactional networking, and sketched a tag-based algorithm to consistently compose concurrent network updates that features an exponential tag complexity not robust to any controller failure.

Distributed Computing. There is a long tradition of defining correctness of a concurrent system via an equivalence to a sequential one [53, 68, 96]. The notion of sequentially composable histories is reminiscent of linearizability [53], 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) [112]. Our interface is however intended to model realistic network management operations, which makes it simpler than recent dynamic STM models [52]. Also, we assumed that controllers are subject to

failures, which is usually not assumed by STM implementations.

7.6 Summary

We believe that our work 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.*, 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" [63]. 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. For instance, it may be interesting to combine FixTag and ReuseTag, in the sense that the fast FixTag algorithm could be used in sparse areas of the network, while the dynamic tag reuse of ReuseTag is employed in dense areas.

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.

As was recently suggested by Casado *et al.* [22], maintaining consistency in network-wide structures and distributed updates, as well as providing the ability of modular composition and formal verification of network programs, are becoming principal SDN challenges. Our suggestion to provide control applications with a *transactional* interface [20] appears to be an adequate way to address these challenges: transactions provide the illusion of atomicity of updates and reads, can be easily composed, and allow for automated verification.

8 Distributed Network Updates

Software-Defined Networking (SDN) is transforming the way networks are controlled and run. In contrast to traditional networks, in which forwarding

devices[6] have proprietary control interfaces over distributed protocols, SDN advocates for standardized interfaces (such as OpenFlow [82]) to control the network in a centralized fashion. In practice, deployments of SDN [55, 58, 65] use a distributed software program as a network controller that manipulates network configuration. This configuration consists of a collection of forwarding rules distributed across network devices. Forwarding rules determine how packets are forwarded between devices.

Several recent projects have demonstrated the value of centrally controlling networks [19, 42, 55, 58, 72, 113]. We observe, like others before us [62, 73, 105], that regardless of their goal, such systems operate by frequently updating the network configuration, either periodically or in reaction to events such as failures or traffic changes. Updating network configuration is challenging because an update involves performing operations at multiple devices in multiple steps, each of which must be planned to minimize disruptions on the applications using the network [21, 62, 64, 101]. For instance, because of the inherent difficulty in synchronizing the changes at different ingress switches, the link load during an update could get significantly higher than that before of after the update, and lead to packet drops due to congestion [73].

The advent of SDN presents a tremendous opportunity for designing general solutions laid on foundational principles rather than point solutions, which has been a recurring pattern in traditional computer networking for several protocol designs and best practices development. In this work, we take such a fundamental perspective to networking and propose a general solution to the problem of consistently updating network configuration while avoiding several classes of forwarding failures.

Prior work for consistent network updates only considered the scenario in which the network controller updates the network configuration whereas switches take just a passive role with respect to solving coordination during update scheduling and do not take advantage of their immediate proximity to exchange information. This centralized approach has two important implications:

First, the controller is involved with the installation of every update and there are inherently higher latencies than in settings where switches can communicate directly. As a result, even with the current dynamic scheduling approaches [62], a network update typically takes in the order of seconds to be completed with recent results showing 99^{th} percentiles as high as 4 seconds.

[6] We also refer to devices as switches throughout this work.

Second, the update scheduling problem is NP-complete in the general case [62]. As a result, centralized approaches resort to greedy heuristics [62] or automatic synthesis through incremental model checking [81], which is computationally expensive at scale.

In contrast with current methods, we investigate the prospect of designing a distributed network update algorithm that entails an active participation role for switches. We argue this approach is practical and supported by several recent works [13, 60, 74] that have demonstrated more programmable switch designs compared to OpenFlow switches (that are limited to a match-action paradigm). Recent work also showed how to introduce greater consistency while updating configuration within a single switch [57, 83]. Moreover, traditional devices already run sophisticated routing protocols such as OSPF, IS-IS, etc. and the networking industry is pursuing newer designs that allow to run open operating systems and custom applications on network switches [12, 41, 93].

A distributed network update is a mechanism in which a set of devices collaborates to schedule an update for the entire network using partial knowledge. It helps reducing the complexity of scheduling computation as well as allowing every switch to update its local forwarding rules. However, it may lead to potential forwarding failures or run into a deadlock if the update order is inappropriate. In this work, we introduce a reliable decentralized scheduling algorithm for switches to update forwarding rules locally without any forwarding failure nor deadlock.

In summary, this work makes the following contributions. We formulate the distributed network update problem and introduce a formal model of the problem. Our model has several distinctions from previous ones (e.g., [21, 62, 105]) and generalizes previous approaches. We describe an algorithm for decentralized network update scheduling and prove it correctly. Our algorithm does not run into deadlock scenarios that can affect prior, centralized approaches.

8.1 Model

8.1.1 Network primitives

We consider a network $\Gamma(\mathbb{S}, \mathbb{L})$, where $\mathbb{S} = \{s_i\}, \forall 1 \leq i \leq N$ denotes the set of all switches; and $\mathbb{L} = \{\ell_{i,j}\}, 1 \leq i < j \leq N$ is the set of all bidirectional links, in which $\ell_{i,j}$ connecting two switches s_i, s_j. Every link $\ell_{i,j}$ has a capacity $v_{\ell_{i,j}}$.

Packet. In a network, switches forward packets via physical links

connecting a pair of switches. Every packet pk has a default size sz_{pk}.

Flow. A flow F_{ij} is an aggregate of packets between an origin switch s_i and a destination switch s_j. When clear from context we denote a flow with F for simplicity. We use the notation $pk_{F_{ij}}$ to denote a packet belonging to flow F_{ij} (or pk_F when omitting the origin and destination switches).

Because each link has a limited capacity, it can only carries a certain number of packets (depending on their size) in a unit time. Consequently, every flow has a predefined *traffic volume* v_F indicating the total volume of traffic the flow will forward in a unit of time. In practice, packet size is measured by b (*bit*) and the unit of traffic volume is *bps* (bit per second). When we need to specify the traffic volume together with a flow, we use notation $F{:}v_F$.

Policy. Packets of a certain flow F are carried over a *set of paths* $\mathbb{P}_F$. The precise behavior how packets are mapped to paths is determined by the forwarding policy r_F. Note that r_F is applied for all paths $p \in \mathbb{P}_F$. Our approach does not specify what the actual network policy is, as this is application-specific and is determined by the SDN controller. Although we leave the definition of policy abstract, recent work [6] has established a precise formalism of forwarding policies. In their formalism, which also applies to our model, a policy is a function that maps from located packets to set of located packets. Our discussion below only assumes an equivalence relation between policies.

Path. All paths of a flow F_{ij} have the same original switch s_i and destination switch s_j. Every path $P \in \mathbb{P}_F$ has a traffic volume v_P, such that $\sum_{P \in \mathbb{P}_F} v_P = v_F$. Notation $P{:}v_P$ is used to denote path p with traffic volume v_P.

A path p is represented by sequence $P(S_P, L_P)$; where $S_P = [s_{1_P}, s_{2_P}, \ldots, s_{k_P}] \subset S$ (i_P is the order in which switches are traversed in path P of length k); and $L_P = \{\ell_{i_P,(i+1)_P} | 1 \leq i < k\} \subset L$ is the set of traversed links. While S_P and L_P can be straightforwardly calculated from each other, we use both notations to simplify our formalism. For simplicity, we also refer to a path from s_{1_P} to s_{k_P} by a natural sequence of its vertices $P = s_{1_P} s_{2_P} \ldots s_{k_P}$. This way, given path $P = s_{1_P} s_{2_P} \ldots s_{k_P}$, $\forall 1 \leq i_P \leq j_P \leq k_P$, we denote a path segment P as:

$$
\begin{aligned}
Ps_{i_P} &= [s_{1_P} \ldots s_{i_P}] && \text{a segment of } P \text{ from the first switch to } s_{i_P} \\
s_{i_P}P &= [s_{i_P} \ldots s_{k_P}] && \text{a segment of } P \text{ from } s_{i_P} \text{ to the last switch } s_{k_P} \\
s_{i_P}Ps_{j_P} &= [s_{i_P} \ldots s_{j_P}] && \text{a segment of } P \text{ between } s_{i_P}, s_{j_P}
\end{aligned}
$$

A flow F is also represented as a set of directional weighted graph $F(S_F, L_F)$, where $S_F = \bigcup_{P \in \mathbb{P}_F} S_P$ and $L_F = \bigcup_{P \in \mathbb{P}_F} L_P$.

In this work, notation $\perp$ is used to represent the *nonexistence* of (1) a flow , (2) a path, or (3) a set of policies.

Comparisons. We introduce the following comparison relations between paths (P_1, P_2) and flows (F_1, F_2):

- $P_1 \sim P_2 \Rightarrow (init_{P_1} = init_{P_2}) \wedge (end_{P_1} = end_{P_2})$.
- $P_1 \equiv P_2$ if P_1 and P_2 have the same sequence $S_{P_1} \equiv S_{P_2}$ (*i.e.* $L_{P_1} \equiv L_{P_2}$).
- $F_1 \sim F_2 \Rightarrow (init_{F_1} = init_{F_2}) \wedge (end_{F_1} = end_{F_2})$.
- $F_1 \equiv F_2 \Rightarrow (\mathbb{P}_{F_1} \equiv \mathbb{P}_{F_2}) \wedge (r_{F_1} \equiv r_{F_2}) \wedge (t_{F_1} = t_{F_2})$

The $\equiv$ relation of path (respectively flow) implies the $\sim$ relation of path (resp. flow).

8.1.2 Packet forwarding.

At the origin switch s_i, a flow of packets with traffic volume limited by $v_{F_{ij}}$, is sent to the destination switch s_j by splitting and forwarding according to the set of paths $\mathbb{P}_{F_{ij}}$. Whenever receiving a packet pk from the predecessor, every intermediate switch forwards the packet to the successor by forwarding function $\omega_{s_i,t}()$. Based on (i) the information carried in the packet, and (ii) the associated information stored in the switch applied to the packet, ω returns one of two possible values – *Step or Drop* – indicating that the packet is forwarded to the successor or dropped, respectively. These two kinds of information (within the packet and switch state) are abstracted with (i) a *tag* representing the information carried in packet and (ii) a *forwarding function* running on the switch.

Tag. Every packet pk has a *tag* (denoted by tag_{pk} for a specific packet or simply *tag* in the general case) contains information that identifies the *flow* of pk. Further, it contains necessary information to allow the forwarding function running on a switch to make decisions on how to forward the packet. Every flow F has a set of possible tags called $\mathbb{T}_F$ that can be assigned to a packet. We later describe how tags are used to map packets into network configurations.

Forwarding function. To forward the packet, a switch must necessarily know what policy applies to the packet and the next hop switch. There

are multiple ways to convey this information; in this work, we model the forwarding function by two primitives as follows:

- $next_{s_i,t}(tag)$ which is called at time t in switch s_i to return the next hop switch of s_i for packet pk according to the tag attached in the packet. If there is no information about the next hop of pk, it returns $\perp$
- $get\_policy_{s_i,t}(tag)$ which is called at time t in switch s_i. It returns the policy associated with the tag of pk_F. If there is no policy associated with the tag, it returns $\perp$.

The *forwarding function* is defined based on the two primitives above:

$$\omega_{s_i,t}(tag_{pk_F}) =$$
$$\begin{cases} Drop \Leftrightarrow (next_{s_i,t}(tag_{pk_F}) \equiv \perp \vee get\_policy_{s_i,t}(tag_{pk_F}) \equiv \perp) \\ Step \Leftrightarrow (next_{s_i,t}(tag_{pk_F}) \not\equiv \perp \wedge get\_policy_{s_i,t}(tag_{pk_F}) \not\equiv \perp) \end{cases}$$

Trace. When a packet pk is forwarded according to the flow, $trace_t(pk)$ is a ordered sequence of all switches that pk traversed until time t. A trace $trace_t(pk) = s_i \dots s_k$ means that pk was forwarded from switch s_i and arrived at s_k by time t. The notation $\tau_{k,pk_{F_{ij}}}$ denotes the time at which packet $pk_{F_{ij}}$ arrived at switch s_k. When the *loop-freedom* property (introduced later) does not hold, there could be multiple appearances of a switch s_k in a trace. The trace information is only used to analyze and define the problem. Neither a packet nor a switch practically store this information.

Successful forwarding. Packet pk is successfully forwarded from switch s_i to s_j if there exists finite time t_j such that $trace_{t_j}(pk) = s_i \dots s_j$.

Forwarding failure. A packet forwarding could fail due to one of following reasons: (i) the packet is dropped because of a disrupted path[7], (ii) the packet follows a path that contains a cycle and the packet loops in the network, (iii) the packet is dropped because it reaches a congested link.

Given a packet pk, these failures are formally defined as follows:

- **Black-hole failure**: Packet pk is dropped in the network.

$$\exists s_k \in \mathbb{S}, \omega_{s_k,\tau_{k,pk}}(tag_{pk}) = Drop$$

[7]Meaning that a packet is dropped at a certain switch when the next hop is $\perp$ although this is not the intended behavior.

- **Loop failure**: There exists a switch s_k such that function $next_{s_k,\tau_{k,pk}}(tag_{pk})$ returns, for packet pk, a next hop switch that has already appeared in $trace_{\tau_{k,pk}}(pk)$

$$\exists s_k, next_{s_k,\tau_{k,pk}}(tag_{pk}) \in trace_{\tau_{k,pk}}(pk)$$

- **Congestion failure**: pk is forwarded from current switch s_i to the next hop switch s_j while its current traffic volume of $\ell_{i,j}$ is greater than its capacity.

$$\exists T, \exists s_1, s_2 \in \mathbb{S},$$
$$\int_{t=T}^{T+\Delta} \sum\nolimits_{pk'|next_{s_1,t}(tag_{pk'})=s_2} sz_{pk'} > v_{\ell_{1,2}}$$

Note that these three types of failures have a different impact on the quality-of-service (QoS) of the network. The *black-hole* and *loop* failures only affect packets that are part of the flows affected by the failure. Instead, the *congestion* failure affects packet forwarding for potentially every flow that shares the same congested links.

8.1.3 Network configuration.

A network configuration $\mathbb{C}$ is the set of all flows in a network. In this work, we assume the UNITY property of flow in a configuration:
Unity. For any two switches s_i and s_j, in every network configuration, there is no more than one flow F forwarding packets from s_i to s_j.

$$\forall F_1, F_2 \in \mathbb{C}, F_1 \nsim F_2$$

Validity. A network configuration is valid if it does not contain the potential factors leading to the failure of forwarding. Consider an arbitrary configuration, where every packet is forwarded with a *tag* that is assigned by the starting switch of a flow. The time to forward a packet is shorter than the time for network to change from one configuration to another [62].

So, given a network configuration $\mathbb{C}$, let $\mathbb{T}_\mathbb{C}$ be the set of all possible tags in a network configuration that can be assigned to a packet ($\mathbb{T}_\mathbb{C} = \bigcup_{F\in\mathbb{C}} \mathbb{T}_F$), $\mathbb{C}$ is *valid* if for any packet forwarded with a $tag \in \mathbb{T}$, there is no failure in forwarding. Formally, the VALIDITY of $\mathbb{C}$ is defined as follows:

- **Black-hole freedom**: no packet is dropped in the network.

$$\forall F_{ij} \in \mathbb{C}, \forall tag \in \mathbb{T}_{F_{ij}},$$
$$\exists P = [s_i \dots s_j] | \forall s_k \in P, \omega_{s_k,t}(tag) = Step$$

- **Loop freedom**: no packet should loop in the network.

$$\forall F_{ij} \in \mathbb{C}, \forall tag \in \mathbb{T}_{F_{ij}},$$
$$\exists P = [s_i \dots s_j] | \forall s_k, s_h \in P, s_k \neq s_h$$

- **Congestion freedom**: no link has to carry a traffic greater than the capacity of the link.

$$\forall t_1, \forall s_i, s_j \in \mathbb{S},$$
$$\int_{t=t_1}^{t_1+\delta t} \sum\nolimits_{pk | next_{s_i,t}(tag_{pk})=s_j} sz_{pk} \leq v_{\ell_{i,j}}$$

8.2 Problem

8.2.1 Network update

Given two network configurations $\mathbb{C}, \mathbb{C}'$, a network update causes the transformation $\Omega_{\mathbb{C}} : \mathbb{C} \mapsto \mathbb{C}'$ from the current network configuration $\mathbb{C}$ to a target configuration $\mathbb{C}'$, as show in Figure 7. In more detail, the old set of flows in $\mathbb{C}$ will be replaced by the new set of flows in $\mathbb{C}'$ such that the replacement does not cause any failures. During this transformation, the network configuration evolves through a sequence of intermediate states until it reaches the target configuration $\mathbb{C}'$. So, a *network update* is an evolution $\mathcal{C}$ of network configuration starting from time t_0, with $\mathcal{C}_{t_0} \equiv \mathbb{C}$, till $\mathcal{C}_t \equiv \mathbb{C}'$, where $\mathcal{C}_t$ denotes the configuration at time t. A network update has three desired properties as follows:

- Termination: After a finite time, network configuration is $\mathbb{C}'$. Formally, $\exists t', \mathcal{C}_{t'} \equiv \mathbb{C}'$.
- Validity: All intermediate configurations are valid. Formally, $\forall t \geq t_0, \mathcal{C}_t$ is valid.
- Per-packet coherence: no packet is forwarded in the mix of the old and new forwarding policy (recall s_i is the origin switch of the

flow). Formally:

$$\forall pk_{F_{ij}}, \forall s_k \in trace_{t_k}(pk_{F_{ij}}),$$
$$get\_policy_{s_k,\tau_{k,pk_{F_{ij}}}}(tag_{pk_{F_{ij}}}) \equiv get\_policy_{s_i,\tau_{i,pk_{F_{ij}}}}(tag_{pk_{F_{ij}}})$$

A naïve solution to network update is to force the origin switch of every flow to stop forwarding new packets until the update finishes. This approach is not practical as it clearly disrupts network performance during the update. Due to capacity limit of physical links, and because of the inherent difficulty in synchronizing the changes at different ingress switches, the link load during an update could get significantly higher than that before of after the update. Therefore, to minimize disruptions to the applications, it is necessary to decompose a network update into a set of small *update operations*. In this work, π denotes an update operation.

Scheduling Given the set of update operations, a *network update schedule* is the execution order of all given operations such that the VALIDITY of network update is not violated by any execution and the total network traffic demand is satisfied.

We next review related work and later discuss in more detail about the decomposition of network update into the set of update operations. We will focus on two principal aspects: (*i*) what is an update operation? And (*ii*) what entity in the network should perform an update operation?

8.2.2 Related work

The network update scheduling problem has recently been widely studied [21,62,64,73,81,83,101,105]. These works use centralized approaches based on the SDN control plane to preserve the logical constraints of network update. The work in [62] shows that this problem is NP-complete in the presence of both link capacity and switch memory constraints; and finding the fastest scheduling is NP-complete with the constraint of link capacity. The definition of network update operation varies in different approaches. However, these approaches consider the case where every flow only has one path and, as we illustrate later, can run into deadlock scenarios. In contrast, we study the network update problem in decentralized settings and generalize it with flows over multiple paths.

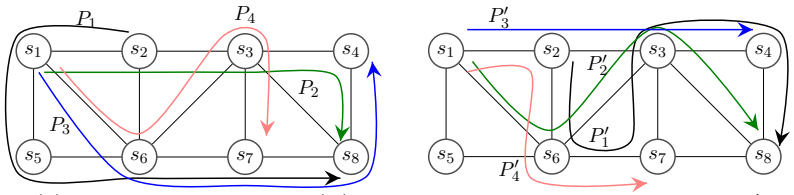
In [62], a scheduling algorithm, called Dionysus, computes a schedule for every path transformation (i.e., update operation). The entity that executes an update operation is a centralized controller that controls all switches in the network. Therefore, the whole path is transformed at the

same time. Dionysus computes a dependency graph that represents the dependencies of update operations on the link capacity resource availability in the whole network. This dependence graph is used by the SDN control plane to perform update operations with a flexible scheduling based on the actual finishing time of update operations across switches.

8.2.3 Network update scheduling

Dependency graph. Given a pair of current and target network configuration $\mathbb{C}, \mathbb{C}'$, any movement π_P from a path P to a the new path P' depends on the availability of related resources in the new path P', while release the corresponding availability to link capacity resource in the old path P. These dependencies are represented with a dependency graph — a bipartite graph $\mathbb{G}(\Pi, L, E_{free}, E_{req})$, where the two of subset of vertices represent the *path transformation set* Π and *link set* L. The value associated to a link vertex $\ell_{i,j} \in L$ represents the current available capacity of $\ell_{i,j}$. The two subset of edges of $\mathbb{G}$, which are E_{req} and E_{free}, indicate the following:

- E_{free} is the set of directed edges from Π to L. A weighted edge e_{free} from transformation $\pi_p \in \Pi$ to a link $\ell_{i,j} \in L$ represents the available capacity that is given to $\ell_{i,j}$ by π_p.
- E_{req} is the set of directed edges from L to Π. A weighted edge e_{req} from link $\ell_{i,j} \in L$ to a transformation $\pi_p \in \Pi$ represents the available capacity of $\ell_{i,j}$ that is necessary to enable π_p.

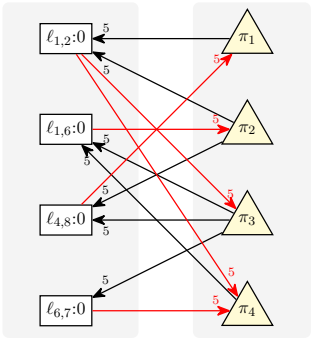


(a) Current configuration ($\mathbb{C}$) (b) Target configuration ($\mathbb{C}'$)

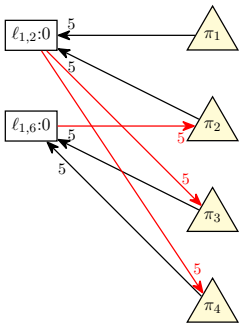
Figure 7: An example of network update

Figure 8(a) shows the dependency graph for the example network update of Figure 7.

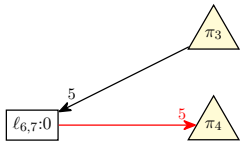
Deadlock. Dionysus [62] creates a dependency graph for the entire network in a central controller. The controller also play a central role in coordinating and deciding what update operation is performed at a particular switch.



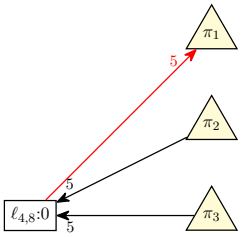
(a) Deadlock for the update of Figure 7



(b) Dependency graph for s_1



(c) Dependency graph for s_6



(d) Dependency graph for s_4, s_7, s_8

Figure 8: Decomposing a network update into three dependency graphs.

However, this approach easily leads to a deadlock situation as illustrated in the example in Figure 7. In this example, the network configuration is updated from $\mathbb{C}$ (Figure 7(a)) to $\mathbb{C}'$ (Figure 7(b)). We assume that every link has 10 unit of capacity and the each path takes 5 unit. So, every link can carry at most 2 paths at the same time. Let π_i be the movement from path p_i in $\mathbb{C}$ to path P_i' in $\mathbb{C}'$. As we do not want to violate the *congestion freedom* property, there is a deadlock that prevents paths to be entirely updated in the whole network. In particular, π_1 cannot be done due to congestion in link $\ell_{4,8}$. The same problem occurs with π_2 and link $\ell_{1,6}$, π_3 and link $\ell_{1,2}$, π_4 and link $\ell_{6,7}$. As shown in Figure 8(a), all movement vertices $\pi_i, (1 \leq i \leq 4)$ depend on the available amount from at least one link capacity node, all of which have current capacity equal to 0. Therefore, there is no schedule that allows to update the network from $\mathbb{C}$ to $\mathbb{C}'$.

8.2.4 Segmentation

Consider the transformation π_1 in our running example. The two segments $s_2P_1s_6$ (resp. $s_7P_1s_8$) can be transformed separately to $s_2P_1's_6$ (resp. $s_7P_1s_8$). The same situation applies for π_2 with two segments $s_1P_2s_3, s_3P_2s_8$; π_4 with two segments $s_1P_4s_6$ and $s_6P_4s_7$. The deadlocked dependency graph in Figure 8(a) can be decomposed into the three dependency graphs in Figure 8(b),8(c),8(d).

This example shows that updating different segments of a path separately avoids the complex scheduling scenario with the complete update, and reduces the potential case of deadlock. In general, instead of updating the whole path, we consider to update disjoint segments having the same starting and ending switches.

The natural questions arises: what is the necessary information for a switch to independently perform such a transformation?

8.2.5 Update operation

An update operation is the smallest unit that should be scheduled by a scheduling algorithm.

Flow transformation. A *flow transformation* is represented as a pair $\phi_F(old, new)$, where $\phi_F.old$, $\phi_F.new$ are the values of flow F before and after the transformation respectively.

A flow transformation could be one of three types: UPDATING (up), REMOVING (rm), and ADDING (add). Let $\Phi_{up}, \Phi_{rm}, \Phi_{add}$ be the set of all UPDATING, REMOVING, ADDING flow transformations respectively. The three types of flow transformations are formally defined as follows.

(1) UPDATING: A flow F is transformed by ϕ_F such that $\phi_F.old \sim \phi_F.new$.

(2) ADDING: A new flow F is added into the new configuration

$$\forall \phi \in \Phi_{add}, \phi_F.new \not\equiv \phi_F.old \equiv \perp$$

(3) REMOVING: A old flow F is removed from the original configuration

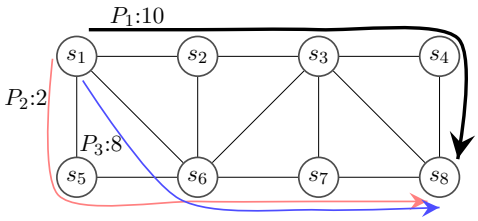
$$\forall \phi \in \Phi_{rm}, \phi_F.old \not\equiv \phi_F.new \equiv \perp$$

Let $\Phi = \Phi_{up} \cup \Phi_{rm} \cup \Phi_{add}$ be the set of all flow transformations. Because of the UNITY property of a flow in a configuration, every flow is transformed by a unique flow transformation.

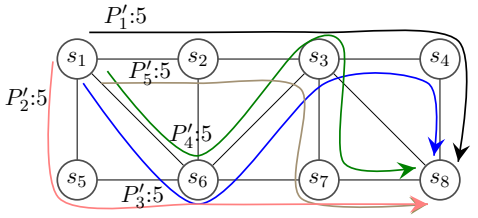
$$\phi_1 = \phi_2 \Leftrightarrow$$
$$\begin{cases} \phi_1.old \sim \phi_2.old \not\equiv \perp \\ \vee((\phi_1.old \equiv \phi_2.old \equiv \perp) \wedge (\phi_1.new \sim \phi_2.new)) \end{cases}$$

In more detail, every flow transformation $\phi_F \in \Phi_{mv}$ consists of: (1) the *policy update* U_F from the set of policies r_F of the original configuration $\mathbb{C}$ to the set of policies $r_{F'}$ in the target configuration $\mathbb{C}'$; (2) the *paths movement* $\Omega_{\mathbb{P}_F} : \mathbb{P}_F \mapsto \mathbb{P}_{F'}$ from set of path $\mathbb{P}_F$ of $\mathbb{C}$ to a new set of path $\mathbb{P}_{F'}$ of $\mathbb{C}'$.

Path movement. For every UPDATING flow transformation ϕ, the traffic volume of the flow after being transformed $\phi.new$ and before being transformed $\phi.old$ could be different. Hence, the number and the traffic volume of paths could also be changed. To avoid impacting network performance, any traffic volume of any removed path in the original configuration need to be replaced by the equivalent traffic volume in the target configuration.



(a) Flow F_{1-8} in current configuration ($\mathbb{C}$)



(b) Flow F_{1-8} in target configuration ($\mathbb{C}'$)

Figure 9: Path movement.

For example, in Figure 9, traffic volumes of paths P_1, P_2, P_3 in configuration $\mathbb{C}$ are $10, 2, 8$, respectively; while in target configuration $\mathbb{C}'$, there are five paths, each of which has the same traffic volume 5. Consequently, paths in the original configuration cannot be directly replace

by another path in the target configuration without degrading network performance.

Split movement. To replace the old traffic volume by an equivalent traffic volume, paths must be split into smaller units, called *split*, such that every unit in the original configuration has a corresponding unit with equal traffic volume in the target configuration. This problem can be solved using a simple allocation algorithm. Consequently, a network update can be considered as a *set of split movements* and *policies update*.

Update operation Applying the idea of updating by segmentation from section 8.2.4 to the split, in this work, we consider an *update operation* is a *movement of a split segment* (*i.e.* the object of scheduling algorithm) with the constraint on the *policies* such that for every packet pk.

- Every switch in the whole split must forward the packet with the same policy;
- But, split can be mixed between split segment from the old configuration and the new configuration.

Every *update operation* is represented as a pair.

$$\pi(old, new) | (\pi.old \sim \pi.new) \vee \neg(\pi.old = \bot \wedge \pi.new = \bot)$$

Where $\pi.old$ (resp. $\pi.new$) is split segment before (resp. after) the movement.

Besides, $\forall \pi(old, new) \in \Pi = \Pi_{mv} \cup \Pi_{rm} \cup \Pi_{add}$, where $\Pi_{mv}, \Pi_{rm}, \Pi_{add}$ are set of all MOVING (mv), REMOVING (rm), and ADDING (add) update operations, respectively.

(1) MOVING: Split segment is replaced by a corresponding split segment with an equivalent traffic volume: $\forall \pi \in \Pi_{mv}, \pi.old \sim \pi.new \wedge v_{\pi.old} = v_{\pi.old}$

(2) REMOVING: $\forall \pi \in \Pi_{rm}, \pi.old \not\equiv \pi.new \equiv \bot$

(3) ADDING: $\forall \pi \in \Pi_{add}, \pi.new \not\equiv \pi.old \equiv \bot$

8.3 Distributed Scheduling

In this section, we firstly introduce an implementation of the two abstractions: Tag(tag) and Forwarding function (ω), introduced in previous section 8.1.

Tag Every tag is a tuple $\langle old_p, new_p, isNew \rangle$, where old_p and new_p are the identities of paths that apply to the packet (in a mutually exclusive fashion), and $isNew$ indicates the packet is forwarded with old or new set of policies.

```
Function next(tag):
    if (tag.old_p < 0 ∧ tag.new_p < 0) then
        return ⊥;
    else if (tag.new_p < 0) then
        return get_path(tag.old_p);
    else
        cur_p =get_path(tag.old_p);
        if (cur_p = ⊥) then
            cur_p = get_path(tag.new_p);
        return cur_p;
Function get_policies(tag):
    return policies[policies_id];
```

Algorithm 1: Forwarding function running in s_i

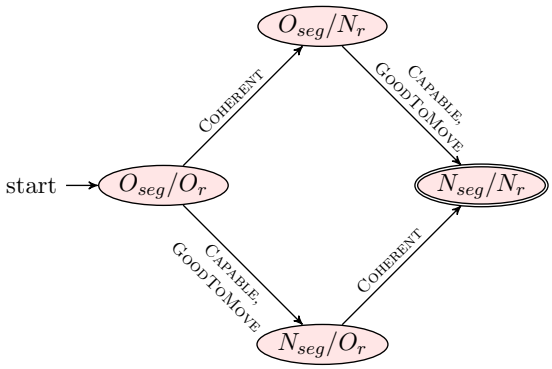
Forwarding function Together with the tag, we also define the way in which paths and rules are stored in the switches; as well as the two primitives, $next_{s_i,t}()$ and $get\_policy_{s_i,t}()$, of forwarding function.

Every switch s_i has a list of paths passing through it and a list of sets of policies corresponding to the paths. The paths and the sets of policies could come from either the old configuration and the new configuration. We assume that every path (resp. every set of policies) has a unique identity.

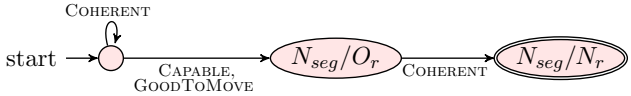
8.3.1 Creating dependency graph

Algorithm 2 computes the dependency graph for update operations related to a switch. This algorithm is executed by any switch that is the final switch of intersection split segment (*i.e.* the starting switch of the disjoint segment) between the old and the new network configuration. Given a switch s_i, the input parameters consist of:

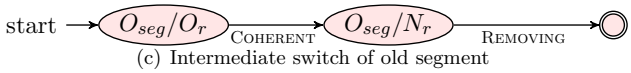
- the set of update operations related to s_i by either *old* split or *new* split (called Π_i).
- the set of links L_i (together with their capacity) that consists of all links appearing in either *old* split segments or *new* split segments of the set of update operations Π.



(a) Origin switch of both new/old split segment



(b) Intermediate switch of new segment



(c) Intermediate switch of old segment

Figure 10: State diagrams

```
Function Scheduling(Π_i, L_i):
    G(Π_i, L_i, E^i_free, E^i_req) ← CreateGraph(Π_i, L_i);
    G(Π_i, L_i, E^i_free, E^i_req) ← SimplifyGraph(G(Π_i, L_i, E^i_free, E^i_req));
    // SimplifyGraph is in the Appendix
    return G(Π, L_i, E^i_free, E^i_req);
Function CreateGraph(Π_i, L_i):
    E^i_free = E^i_req = ∅;
    foreach π ∈ Π do
        if π.old ≢ π.new then
            Π = Π ∪ {π};
            foreach ℓ ∈ π.old do
                if ℓ ∉ π.new then
                    e_free = {π → ℓ : v_π.old};
                    E^i_free = E^i_free ∪ {e_free};
            foreach ℓ ∈ π.new do
                if ℓ ∉ π.old then
                    e_req = {ℓ → π : v_π.new};
                    E^i_req = E^i_req ∪ {e_req};
    return G(Π_i, L_i, E^i_free, E^i_req);
```

Algorithm 2: Creating dependency graph for Π_i

8.3.2 Scheduling an update operation

The scheduler for an update operation π works as a state machine with four states corresponding to the all composition cases of the split segment and the rule in which a switch is able to forward the packet: (1) O_{seg}/O_r: with old configuration. (2) O_{seg}/N_r: with old split segment and new policies. (3) N_{seg}/O_r: with new split segment and old policies. (4) N_{seg}/N_r: with new configuration.. The schedulers are different according to the logical position of switch in a split/path. The accepting state is when a switch can forward packet following to new split segment with the new set of policies.

State transition While CAPABLE is not a message, it is a state of a *link capacity*, in dependency graph, for a corresponding link of split segment, the others are a notifying messages sent by switch.

- GOODTOMOVE message: sent by a switch s_2 in split segment $\pi.new$ with ***π.new*'s identity**. It notifies that $\pi.new$ has no blackhole and the movement does not create the deadlock.
- COHERENT message: sent by a switch s_2 in either old or new path with the **path's identity**. It states that all successors of s_2 in the whole path have the new set of policies installed.

- REMOVING message: sent by the origin switch of both the new and old split segments to the successors in the old segment. This message states that the old split segment can be removed.

An update operation is only performed at switch s_1, when all link capacity nodes of corresponding incoming edges of s_1 in $\mathbb{G}(\Pi_1, L_1, E^1_{free}, E^1_{req})$ and s_1 receive GOODTOMOVE message. Therefore, in every switch s_1 running scheduling algorithm, we assume the simple functions returning if s_1 received a particular type of message given by a update operation identity.

After executed the update operation π, switch s_1 sends REMOVING message to the successor switch in the old split segment $\pi.old$. Upon receiving REMOVING message, from predecessor, switch s_2 adds the additional capacity to corresponding link node of outgoing edges in dependency graph.

Event `Receiving` UPDATING `message` msg `contains` $\Pi_i, policies'_i$**:**
  `StoreNewState(`$\Pi_i, policies'_i$`);`
  $\mathbb{G}(\Pi_i, L_i, E^i_{free}, E^i_{req}) \leftarrow$ `Scheduling(`Π`);`
Event `Receiving` REMOVING `message` msg**:**
  Remove path $\pi_{msg}.old$;
  foreach $\ell \in \pi_{msg}.old$ **do**
    if $\ell \in L_i$ **then**
      $v_\ell = v_\ell + v_{\pi_{msg}.old}$;
  send REMOVING message to the successor switches in $\pi_{msg}.old$;
  `ExecuteScheduling(`$\mathbb{G}(\Pi_i, L_i, E^i_{free}, E^i_{req})$`);`
Event `Receiving` GOODTOMOVE `message` msg**:**
  `// update operation id contained in` msg `exists in` Π_i
  if $(\exists \pi_{msg} \in \Pi_i)$ **then**
    if $s_i = \pi_{msg}.segInit$ **then**
      `ExecuteScheduling(`$\mathbb{G}(\Pi_{msg}, L_i, E^i_{free}, E^i_{req})$`);`
    else if `NoDeadlock(`π_{msg}`)` **then**
      send GOODTOMOVE message to predecessor switches of $\pi_{msg}.new$;
Event `Receiving` COHERENT `message` msg**:**
  if $(\exists \pi_{msg} \in \Pi_i)$ **then**
    send COHERENT message to predecessor switches of π_{msg};

Algorithm 3: Event handler (running in every switch s_i)

Lemma 7 (Congestion freedom) *The scheduling algorithm preserves the congestion freedom.*

(Sketch). Congestion freedom is guaranteed by the dependency graph and `IsExecutable` function. `IsExecutable` ensures that there are only two

```
Function ExecuteScheduling(𝔾(Π_i, L_i, E^i_free, E^i_req)):
    if ∄𝔾(Π_i, L_i, E^i_free, E^i_req) then
        return;
    foreach unexecuted operation π_i ∈ G do
        if IsExecutable(π_i) then
            execute π_i;
            send REMOVING message to successor switch the old path π_i.old;
Function IsExecutable(π_i):
    return (((∄{ℓ → π_i} ∈ E^i_req) ∨ (∀{ℓ → π_i} ∈ E^i_req, v_ℓ > {ℓ → π_i:v}))
      ∧( NoDeadlock(π_i)= True)
      ∧( ReceivedGoodToMoveMsg(π_i)= True);
      ∧( IsNoLoop(π_i)= True);
Function StoreNewState(Π_i, policies'_i):
    Store(Π_i, policies'_i);
    foreach π ∈ Π_i do
        if (s_i = π.new.segEnd)
        ∨(ReceivedGoodToMoveMsg(π)= True) then
            send GOODTOMOVE message to predecessor of π.new;
        if (s_i = π.old.end)
        ∨(ReceivedCoherentMsg(π)= True) then
            send COHERENT message to predecessor of π.old and π.new;
```

Algorithm 4: Distributed network update (running in every switch s_i)

cases in which an update operation is executed: (1) when update operation does not requires any link capacity, (2) when the available capacity of incoming edge greater than the require capacity of the update operation. By this way, no link has to carry a traffic greater than its capacity. □

Lemma 8 (Per-packet coherence) *The scheduling algorithm preserves the per-packet coherence property.*

Proof. A COHERENT message is only sent in two cases:

1. In line 19 of Algorithm 4, when an ending switch of the whole path successfully updates the policies of the new configuration.

2. In line 20 of Algorithm 3, a switch receive the COHERENT from the successor that already updated the policies of the new configuration.

Recursively, every intermediate switch, in a path, receiving COHERENT and sending it to predecessor has successfully updated the new policies.

Consequently, the starting switch of a path only receives COHERENT when all successor switches in the new path segment (logical group of all split segments of the same path) have updated new network configuration. □

Lemma 9 (Blackhole freedom) *The blackhole freedom property holds with the scheduling algorithm.*

Proof. A GOODTOMOVE message is only sent in two cases:

1. In line 16 of Algorithm 4, when an ending switch of a split segment $\pi.new$ successfully updates the corresponding path and policies of the new configuration.

2. In line 16 of Algorithm 3, a switch receive the GOODTOMOVE from the successor that already updated the corresponding path and policies of the new configuration.

Recursively, every intermediate switch, in a split segment, receiving GOODTOMOVE and sending it to predecessor has successfully updated the new network configuration.

Consequently, the starting switch of a split segment only receives GOODTOMOVE when all successor switches in the new split segment have updated new network configuration. □

9 Accelerating Consensus via Co-Design

Software-defined networking (SDN) is transforming the way networks are configured and run. In contrast to traditional networks, in which forwarding devices have proprietary control interfaces, SDNs generalize network devices using a set of protocols defined by open standards, including most prominently the OpenFlow [82] protocol. This move towards standardization has led to increased "network programmability", allowing ordinary programs to manage the network through direct access to network devices.

Several recent projects have used SDN platforms to demonstrate that applications can benefit from improved network support. While these projects are important first steps, they have largely focused on one class of applications (*i.e.*, Hadoop data processing [42,48,78,113]), and on improving

performance via data-plane configuration (*e.g.*, route selection [48, 113], traffic prioritization [42,113], or traffic aggregation [78]). None of this work has fundamentally considered whether application logic could be moved into the network. In other words: *how can distributed applications and protocols utilize network programmability to improve performance?*

This work focuses specifically on the Paxos consensus protocol [69]. Paxos is an attractive use-case for several reasons. First, it is one of the most widely deployed protocols in highly-available, distributed systems, and is a fundamental building block to a number of distributed applications [18, 29, 46]. Second, there exists extensive prior research on optimizing Paxos [70, 80, 98, 99], which suggests that the protocol could benefit from increased network support. Third, moving consensus logic into network devices would require extending the OpenFlow API with functionality that is amenable to an efficient hardware implementation [11, 14].

Implementing Paxos in the network provides a different point in the design space, and identifies a different set of network requirements for protocol implementors. This work presents two different approaches: (*i*) a detailed description of a sufficient set of OpenFlow extensions needed to implement the full Paxos logic in SDN switches; and (*ii*) an alternative, optimistic protocol which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages.

Although neither of these protocols can be fully implemented without changes to the underlying switch firmware, we present evidence to show that such changes are feasible. Moreover, we present an evaluation that suggests that moving consensus logic into the network would reduce application complexity, reduce application message latency, and increase transaction throughput.

In summary, this work makes the following contributions:

- It identifies a *sufficient* set of features that protocol implementors would need to provide to implement consensus logic in network devices.
- It describes an alternative protocol, inspired by Fast Paxos [70], which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages.
- It presents experiments that suggest the potential performance improvements that would be gained by moving consensus logic into the network.

In the following, we first provide a short summary of the Paxos protocol (§9.1), followed by a description of the two approaches to providing network support for Paxos (§9.2). Then, we present the results from our experimental evaluation (§9.3) and discuss related work (§9.4) before giving a summary of this work (§9.5).

9.1 Paxos Background

State-machine replication [67, 109] is a fundamental approach to designing fault-tolerant systems used by many distributed applications and services (*e.g.*, Google's Chubby [18], Scatter [46], Spanner [29]). The key idea is to replicate services, so that a failure at any one replica does not prevent the remaining operational replicas from servicing client requests. State-machine replication is implemented using a *consensus* protocol, which dictates how the participants propagate and execute commands.

Paxos [69] is perhaps the most widely used consensus protocol. Paxos participants, which communicate by exchanging messages, may play any of three roles: *proposers* issue requests to the distributed system (*i.e.*, propose a value); *acceptors* choose a single value; and *learners* provide replication by learning what value has been chosen. Note that a process may play one or more roles simultaneously. For example, a *client* in a distributed system may be both a proposer and a learner.

A Paxos *instance* is one execution of consensus. An instance begins when a proposer issues a request, and ends when learners know what value has been chosen by the acceptor. The protocol proceeds in a sequence of rounds. Each round has two phases. For each round, one process, typically a proposer or acceptor, acts as the *coordinator* of the round.

Phase 1. The coordinator selects a unique round number *c-rnd* and asks the acceptors to promise that in the given instance they will reject any requests (Phase 1 or 2) with round number less than *c-rnd*. Phase 1 is completed when a majority-quorum Q_a of acceptors confirms the promise to the coordinator. Notice that since Phase 1 is independent of the value proposed it can be pre-executed by the coordinator [69]. If any acceptor already accepted a value for the current instance, it will return this value to the coordinator, together with the round number received when the value was accepted (*v-rnd*).

Phase 2. The coordinator selects a value according to the following rule: if no acceptor in Q_a accepted a value, the coordinator can select any value. If however any of the acceptors returned a value in Phase 1, the coordinator is forced to execute Phase 2 with the value that has the

highest round number *v-rnd* associated to it. In Phase 2, the coordinator sends a message containing a round number (the same used in Phase 1). Upon receiving such a request, the acceptors acknowledge it, unless they have already acknowledged another message (Phase 1 or 2) with a higher round number. Acceptors update their *c-rnd* and *v-rnd* variables with the round number in the message. When a quorum of acceptors accepts the same round number (Phase 2 acknowledgment), consensus terminates: the value is permanently bound to the instance, and nothing will change this decision. Thus, learners can deliver the value. Learners learn this decision either by monitoring the acceptors or by receiving a decision message from the coordinator.

As long as a nonfaulty coordinator is eventually selected and there is a majority quorum of nonfaulty acceptors and at least one nonfaulty proposer, every consensus instance will eventually decide on a value. A failed coordinator is detected by the other nodes, which select a new coordinator. If the coordinator does not receive a response to its Phase 1 message it can re-send it, possibly with a bigger round number. The same is true for Phase 2, although if the coordinator wants to execute Phase 2 with a higher round number, it has to complete Phase 1 with that round number.

The above describes one instance of Paxos. Throughout this work, references to Paxos implicitly refer to multiple instances chained together (*i.e.*, Multi-Paxos [25]).

Fast Paxos [70] is a well known optimization of Paxos. It extends the *classic rounds*, as described above, with *fast rounds*. In a fast round proposers contact acceptors directly, bypassing the coordinator. Fast rounds save one communication step but are only effective in the absence of collisions, a situation in which acceptors accept different values in the round, and as a result no value is chosen. Fast Paxos can recover from collisions using classic rounds. In order to ensure that no two values are decided, fast rounds require larger quorums than classic rounds.

9.2 Consensus in the Network

In this section, we identify two approaches to improving the performance of Paxos by using software-defined networking. Section 9.2.1 identifies a sufficient set of features that a switch would need to support to implement Paxos logic (*i.e.*, extensions to OpenFlow). Section 9.2.2 discusses the possibility of implementing consensus using unmodified OpenFlow switches.

9.2.1 Paxos in SDN Switches

We argue that performance benefits could be gained by moving Paxos consensus logic into the network devices themselves. Specifically, network switches could play the role of *coordinators* and *acceptors*. The advantages would be twofold. First, messages would travel fewer hops in the network, therefore reducing the latency for the replicated system to reach consensus. Second, coordinators and acceptors typically act as bottlenecks in Paxos implementations, because they must aggregate or multiplex multiple messages. The consensus protocol we describe in Section 9.2.2 obviates the need for coordinator logic.

A switch-based implementation of Paxos need only implement Phase 2 of the protocol described in Section 9.1. Since Phase 1 does not depend on any particular value, it could be run ahead of time for a large bounded number of values. The pre-computation would need to be re-run under two scenarios: either (*i*) the Paxos instance approaches the bounded number of values, or (*ii*) the device acting as coordinator changes (possibly due to failure).

Unfortunately, even implementing Phase 2 of the Paxos logic in SDN switches goes far beyond what is expressible in the current OpenFlow API, which is limited to basic match-action rules, simple statistics gathering, and modest packet re-writes (*e.g.*, incrementing the time-to-live). Below, we identify a *sufficient* set of operations that the switch could perform to implement Paxos. Note, we are not claiming that this set of operations is *necessary*. As we will see in Section 9.2.2, the protocol can be modified to avoid some of these requirements.

Generate round and sequence number. Each switch coordinator must be able to generate a unique round number (*i.e.*, the *c-rnd* variable), and a monotonically increasing, gap-free sequence number.

Persistent storage. Each switch acceptor must store the latest ballot it has seen (*c-rnd*), the latest accepted ballot (*v-rnd*), and the latest value accepted.

Stateful comparisons. Each switch acceptor must be able to compare a *c-round* value in a packet header with a *c-rnd* value that has been stored. If the new value is higher, then the switch must update the local state with the new *c-round* and value, and then broadcast the message to all learners. Otherwise, the packet could be ignored (*i.e.*, dropped).

Storage cleanup. Stored state must be trimmed periodically.

Recent work on extending OpenFlow suggests that the functionality described above could be efficiently implemented in switch hardware [11, 13, 14]. Moreover, several existing switches already have support of some combinations of these features. For example, the NoviSwitch 1132 has 16 GB of SSD storage [87], while the Arista 7124FX [8] has 50 GB of SSD storage directly usable by embedded applications. Note that current SSDs typically achieve throughputs of several 100s MB/s [95], which is within the requirements of a high-performance, network-based Paxos implementation. The upcoming Netronome network processor NFP-6xxx [86], which is used to realize advanced switches and programmable NICs, has sequence number generators and can flexibly perform stateful comparisons.

Also, rather than modifying network switches, a recent hardware trend towards programmable NICs [10, 85] could allow the proposer and acceptor logic to run at the network edge, on programmable NICs that provide high-speed processing at minimal latencies (tens of μs). Via the PICe bus, the programmable NIC could communicate to the host OS and obtain access to permanent storage.

9.2.2 Fast Network Consensus

Section 9.2.1 describes a sufficient set of functionality that protocol designers would need to provide to completely implement Paxos logic in forwarding devices. In this section, we describe *NetPaxos*, an alternative algorithm inspired by Fast Paxos. The key idea behind NetPaxos is to distinguish between two execution modes, a "fast mode" (analogous to Fast Paxos's fast rounds), which can be implemented in network forwarding devices with no changes to existing OpenFlow APIs, and a "recovery mode", which is executed by commodity servers.

Both Fast Paxos's fast rounds and NetPaxos's fast mode avoid the use of a Paxos coordinator, but for different motivations. Fast Paxos is designed to reduce the total number of message hops by optimistically assuming a spontaneous message ordering. NetPaxos is designed to avoid implementing coordinator logic inside a switch. In contrast to Fast Paxos, the role of acceptors in NetPaxos is simplified. In fact, acceptors do not perform any standard acceptor logic in NetPaxos. Instead, they simply forward all messages they receive, without doing any comparisons. Because they always accept, we refer to them as *minions* in NetPaxos.

Figure 11 illustrates the design of NetPaxos. In the figure, all switches

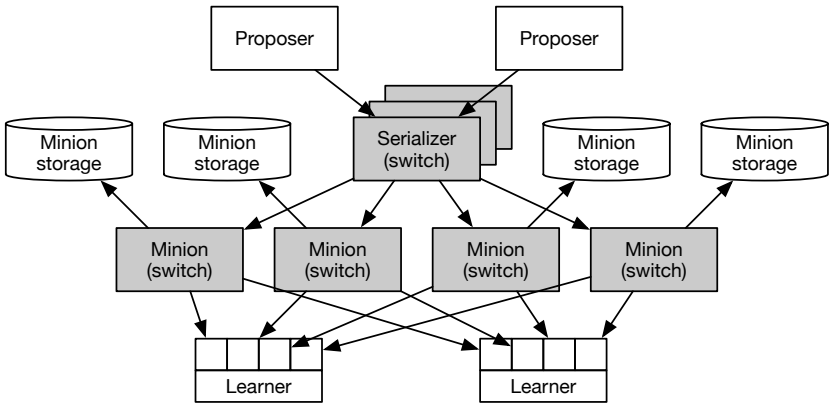


Figure 11: Network Paxos architecture. Switch hardware is shaded grey. Other devices are commodity servers. The learners each have four network interface cards.

are shaded in gray. Proposers send messages to the single switch called a *serializer*. The serializer is used to establish an ordering of messages from the proposers. The serializer then broadcasts the messages to the minions. Each minion forwards the messages to the learners and to a server that acts as the minion's external storage mechanism, used to record the history of "accepted" messages. Note that if switches could maintain persistent state, there would be no need for the minion storage servers. Each learner has multiple network interfaces, one for each minion.

The protocol, as described, does not require any additional functionality beyond what is currently available in the OpenFlow protocol. However, it does make two important assumptions:

1. **Packets broadcast from the serializer to the minions arrive in the same order.** This assumption is important for performance, not correctness. In other words, if packets are received out-of-order, the learners would recognize the problem, fail to reach consensus, and revert to the "recovery mode" (*i.e.*, classic Paxos).

2. **Packets broadcast from a minion arrive all in the same order at its storage and the learners.** This assumption is important for correctness. If this assumption is violated, then learners may decide different values in an instance of consensus and not be able to recover a consistent state from examining the logs at the minion storage.

Recent work on Speculative Paxos [104] shows that packet reordering happens infrequently in data centers, and can be eliminated by using IP multicast, fixed length network topologies, and a single top-of-rack switch acting as a serializer. Our own initial experiments (§ 9.3) also suggest that these assumptions hold with unmodified network switches when traffic is non-bursty, and below about 675 Mbps on a 1 Gbps link.

Fast Paxos optimistically assumes a spontaneous message ordering with no conflicting proposals, allowing proposers to send messages directly to acceptors. Rather than relying on spontaneous ordering, NetPaxos uses the serializer to establish an ordering of messages from the proposers. It is important to note that the serializer does not need to establish a FIFO ordering of messages. It simply maximizes the chances that acceptors see the same ordering.

Learners maintain a queue of messages for each interface. Because there are no sequence or round numbers, learners can only reason about messages by using their ordering in the queue, or by message value. At each iteration of the protocol (*i.e.*, consensus instance), learners compare the values of the messages at the top of their queues. If the head of a quorum with three queues contain the same message, then consensus has been established through the fast mode, and the protocol moves to the next iteration. The absence of a quorum with the same message (*e.g.*, because one of the minions dropped a packet), leads to a conflict.

Like Fast Paxos [70], NetPaxos requires a two-thirds majority to establish consensus, instead of a simple majority. A two-thirds majority allows the protocol to recover from cases in which messages cannot be decided in the fast mode. If a learner detects conflicting proposals in a consensus instance, then the learner reverts to recovery mode and runs a classic round of Paxos to reach consensus on the value to be learned. In this case, the learner must access the storage of the minions to determine the message to be decided. The protocol ensures progress as long as at most one minion fails. Since the non-conflicting scenario is the usual case, NetPaxos typically is able to reduce both latency and the overall number of messages sent to the network.

Switches and servers may fail individually, and their failures are not correlated. Thus, there are several possible failure cases that we need to consider to ensure availability:

- *Serializer failure.* Since the order imposed by the serializer is not needed for correctness, the serializer could easily be made redundant, in which case the protocol would continue to operate despite the failure of one serializer. Figure 11 shows two backup switches for

the serializer.

- *Minion failure.* If any minion fails, the system could continue to process messages and remain consistent. The configuration in Figure 11, with four minions, could tolerate the failure of one minion, and still guarantee progress.

- *Learner failure.* If the learner fails, it can consult the minion state to see what values have been accepted, and therefore return to a consistent state.

A natural question would be to ask: if minions always accept messages, why do we need them at all? For example, the serializer could simply forward messages to the learners directly. The algorithm needs minions to provide fault tolerance. Because each minion forwards messages to their external storage mechanism, the system has a log of all accepted messages, which it can use for recovery in the event of device failure, message re-ordering, or message loss. If, alternatively, the serializer were responsible for maintaining the log, then it would become a single point of failure.

A final consideration is whether network hardware could be modified to ensure the NetPaxos ordering assumptions. We discussed this matter with several industrial contacts at different SDN vendors, and found that there are various platforms that could enforce the desired packet ordering. For example, the Netronome NFP-6xxx [86] has a packet reorder block on the egress path that allows packets to be reordered based on program-controlled packet sequence numbers. A NetPaxos implementation would assign the sequence numbers based on when the packets arrive at ingress. The NetFPGA platform [45] implements a single pipeline where all packet processing happens sequentially. As such, the NetPaxos ordering assumption is trivially satisfied. Furthermore, discussions with Corsa Technology [35] and recent work on Blueswitch [57] indicate that FPGA-based hardware would also be capable of preserving the ordering assumption.

In the next section, we present experiments that show the expected performance benefits of NetPaxos when these assumptions hold.

9.3 Evaluation

Our evaluation focuses on two questions: (*i*) how frequently are our assumptions violated in practice, and (*ii*) what are the expected performance benefits that would result from moving Paxos consensus logic into forwarding devices.

Experimental setup. All experiments were run on a cluster with two types of servers. Proposers were Dell PowerEdge SC1435 2-CPU servers with 4 x 2 GHz AMD cores, 4 GB RAM, and a 1 Gbps NIC. Learners were Dell PowerEdge R815 8-CPU servers with 64 x 2 GHz AMD hyperthreaded cores, 128 GB RAM, and 4 x 1 Gbps NICs. The machines were connected in the topology shown in Figure 11. We used three Pica8 Pronto 3290 switches. One switch played the role of the serializer. The other two were divided into two virtual switches, for a total of four virtual switches acting as minions.

Ordering assumptions. The design of NetPaxos depends on the assumption that switches will forward packets in a deterministic order. Section 9.2.2 argues that such an ordering could be enforced by changes to the switch firmware. However, in order to quantify the expected performance benefits of moving consensus logic into forwarding devices, we measured how often the assumptions are violated in practice with unmodified devices.

There are two possible cases to consider if the ordering assumptions do not hold. First, learners could deliver different values. Second, one learner might deliver, when the other does not. It is important to distinguish these two cases because delivering two different values for the same instance violates correctness, while the other case impacts performance (*i.e.*, the protocol would be forced to execute in recovery mode, rather than fast mode).

The experiment measures the percentage of values that result in a *learner disagreement* or a *learner indecision* for increasing message throughput sent by the proposers. For each iteration of the experiment, the proposers repeatedly sleep for 1 ms, and then send n messages, until 500,000 messages have been sent. To increase the target rate, the value of n is increased. The small sleep time interval ensures that traffic is non-bursty. Each message is 1,470 bytes long, and contains a sequence number, a proposer id, a timestamp, and some payload data.

Two learners receive messages on four NICs, which they processes in FIFO order. The learners dump the contents of each packet to a separate log file for each NIC. We then compare the contents of the log files, by examining the messages in the order that they were received. If the learner sees the same sequence number on at least 3 of its NICs, then the learner can deliver the value. Otherwise, the learner cannot deliver. We also compare the values delivered on both learners, to see if they disagree.

Figure 12 shows the results, which are encouraging. We saw no disagreement or indecision for throughputs below 57,457 messages/second. When we increased the throughput to 65,328 messages/second, we measured no learner disagreement, and only 0.3% of messages resulted in learner

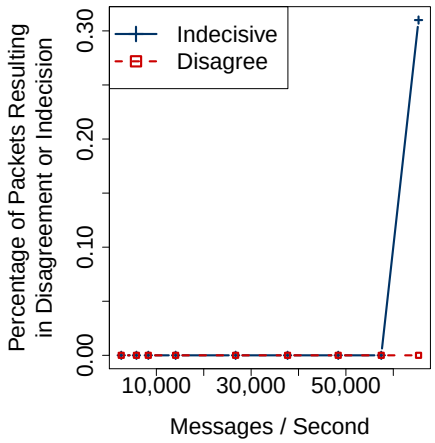


Figure 12: Evaluation of ordering assumptions showing the percentage of messages in which learners either disagree, or cannot make a decision.

indecision. Note that given a message size of 1,470 bytes, 65,328 messages/second corresponds to about 768 Mbps, or 75% of the link capacity on our test configuration.

Although the results are not shown, we also experimented with sending bursty traffic. We modified the experiment by increasing the sleep time to 1 second. Consequently, most packets were sent at the beginning of the 1 second time window, while the average throughput over the 1 second reached the target rate. Under these conditions, we measured larger amounts of indecision, 2.01%, and larger disagreement, 1.12%.

Overall, these results suggest that the NetPaxos ordering assumptions are likely to hold for non-bursty traffic for throughput less than 57,457 messages/second. As we will show, this throughput is orders of magnitude greater than a basic Paxos implementation.

NetPaxos expected performance. Without enforcing the assumptions about packet ordering, it is impossible to implement a complete, working version of the NetPaxos protocol. However, given that the prior experiment shows that the ordering assumption is rarely violated, it is still possible to compare the expected performance with a basic Paxos implementation. This experiment quantifies the performance improvements we could expect to get from a network-based Paxos implementation for a *best case scenario*.

We measured message throughput and latency for NetPaxos and an open

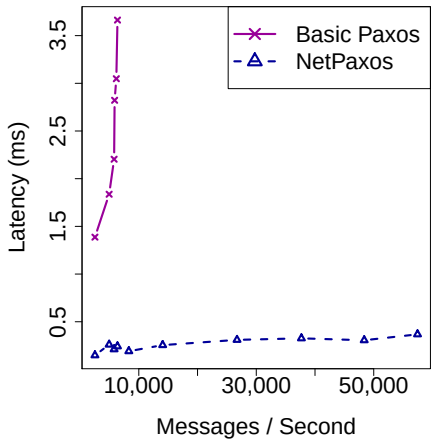


Figure 13: Evaluation of performance showing the throughput vs. latency for basic Paxos and NetPaxos.

source implementation of basic Paxos[8] that has been used previously in replication literature [79, 111]. As with the prior experiment, two proposers send messages at increasing throughput rates by varying the number of messages sent for 1 ms time windows. Message latency is measured one way, using the time stamp value in the packet, so the accuracy depends on how well the server clocks are synchronized. To synchronize the clocks, we re-ran NTP before each iteration of the experiment.

The results, shown in Figure 13, suggest that moving consensus logic into network devices can have a dramatic impact on application performance. NetPaxos is able to achieve a maximum throughput of 57,457 messages/second. In contrast, with basic Paxos the coordinator becomes CPU bound, and is only able to send 6,369 messages/second.

Latency is also improved for NetPaxos. The lowest latency that basic Paxos is able to provide is 1.39 ms, when sending at a throughput of only 1,531 messages/second. As throughput increases, latency also increases sharply. At 6,369 messages/second, the latency is 3.67 ms. In contrast, the latency of NetPaxos is both lower, and relatively unaffected by increasing throughput. For low throughputs, the latency is 0.15 ms, and at 57,457 messages/second, the latency is 0.37 ms. In other words, NetPaxos reduces latency by 90%.

We should stress that these numbers indicate a *best case* scenario for

[8] https://bitbucket.org/sciascid/libpaxos

NetPaxos. One would expect that modifying the switch behavior to enforce the desired ordering constraints might add overhead. However, the initial experiments are extremely promising, and suggest that moving consensus logic into network devices could dramatically improve the performance of replicated systems.

9.4 Related Work

Network support for applications. Several recent projects have demonstrated that large-scale, data processing applications, such as Hadoop, can benefit from improved network support. For example, PANE [42], EyeQ [61], and Merlin [113] all use resource scheduling to improve the job performance, while NetAgg [78] leverages user-defined *combiner* functions to reduce network congestion. These projects have largely focused on improving application performance through traffic management. In contrast, this work argues for moving application logic into network devices.

Speculative Paxos [104] uses a combination of techniques to eliminate packet reordering in a data center, including IP multicast, fixed length network topologies, and a single top-of-rack switch acting as a serializer. NetPaxos uses similar techniques to ensure message ordering. However, NetPaxos moves Paxos logic into the switches, while Speculative Paxos uses servers to provide the role of acceptors.

OpenFlow extensions. To better support the needs of networked applications, there has been an increasing interest in extending OpenFlow with a more generalized API. From academia, there have been several recent proposals [11, 14, 61]. In industry, there has been a longstanding discussion about how to support stateful operations in the new versions of the OpenFlow protocol. The presiding standards body, the Open Networking Foundation (ONF), includes two working groups on the topic: one to standardize extensions to the protocol (EXT-WG), and one focused on forwarding abstractions (FAWG).

Replication protocols. Research on replication protocols for high availability is quite mature. Existing approaches for replication-transparent protocols, notably protocols that implement some form of strong consistency (*e.g.*, linearizability, serializability) can be roughly divided into three classes [27]: (a) state-machine replication [67, 109], (b) primary-backup replication [89], and (c) deferred update replication [27].

At the core of all classes of replication protocol discussed above, there lies a message ordering mechanism. This is obvious in state-machine replication, where commands must be delivered in the same order by all replicas, and

in deferred update replication, where state updates must be delivered in order by the replicas. In primary-backup replication, commands forwarded by the primary must be received in order by the backups; besides, upon electing a new primary to replace a failed one, backups must ensure that updates "in-transit" submitted by the failed primary are not intertwined with updates submitted by the new primary (*e.g.*, [97]).

Although many mechanisms have been proposed in the literature to order messages consistently in a distributed system [36], very few protocols have taken advantage of network specifics. Protocols that exploit *spontaneous message ordering* to improve performance are in this category (*e.g.*, [70, 98, 99]). The idea is to check whether messages reach their destination in order, instead of assuming that order must be always constructed by the protocol and incurring additional message steps to achieve it. As we claim in the proposal, ordering protocols have much to gain (*e.g.*, in performance, in simplicity) by tightly integrating with the underlying network layer.

9.5 Summary

Software-defined networking offers improved network programmability, which can not only simplify network management, but can also enable a tighter integration with distributed applications. This integration means that networks can be tailored specifically to the needs of the deployed applications, and improve application performance.

This work proposes two protocol designs which would move Paxos consensus logic into network forwarding devices. Although neither of these protocols can be fully implemented without changes to the underlying switch firmware, all of these changes are feasible in existing hardware. Moreover, our initial experiments show that moving Paxos into switches would significantly increase throughput and reduce latency.

Paxos is a fundamental protocol used by fault-tolerant systems, and is widely used by data center applications. Consequently, performance improvements in the protocol implementation would have a great impact not only on the services built with Paxos, but also on the applications that use those services.

10 Analysis of Commercially-Available Switches

In this section, we present in brief the capabilities of some OpenFlow-enabled switches, based on publicly available data presented by the manufacturers. Among the switches that were considered were the data planes NoviSwitch

2128 [88] and Corsa's DP64xx family [34], Original Design Manufacturer (ODM) switches and chipsets such as Brocade's VDX series [16], the Mellanox SX1036 series [115] and Broadcom's StrataXGS Trident II switching family [30] and finally commercial off-the-self (COTS) switches such as the Arista 7050SX series [7].

These switches represent a wide variety of performance characteristics and application areas, with throughputs ranging from 240Gbps (NoviSwitch) to 4.03 Tbps (Mellanox and Brocade switches). All of them implement 10 and 40 Gbps Ethernet at their ports, with Mellanox also supporting 56Gbps and Corsa and Brocade supporting 100 Gbps Ethernet. At the same time, the documented latency varies from 220ns (Mellanox SX1036) to 4us (Brocade VDX).

Of these switches, the Mellanox SX1036 supports the 1.0 version of OpenFlow, as does the current version of the Arista 7050SXs Extensible Operating System (EOS). The Trident II switch family implements Broadcom's OpenFlow Data Plane Abstraction (OF-DPA) [32, 33] which supports OpenFlow 1.3+. Brocades VCS Fabric technology [17] implements OpenFlow 1.3, with support for up to 128K flows, also providing the capability to overlap traditional routing and software-defined routing on the same port. Contrary to the previous ones, the NoviFlow and Corsa switches are specifically designed for use on SDN-enabled fabrics. NoviSwitch supports fully OpenFlow 1.3, along with selected features of versions 1.3.5 and 1.4. It supports 28 flow tables, with a programmable TCAM memory that can contain up to 1M flow entries and allows 12K flow modifications per second. Similarly, the Corsa data planes also support OVS-based OpenFlow 1.3+, with 10 programmable flow tables, 1M flow entries and 10K flow modifications per second.

As far as monitoring is concerned, all of these switches, being SDN-enabled, provide the OpenFlow object database with port counters, timers etc. In addition, they can be monitored via SNMP v1, v2, v3 [23] (with the sole exception of the Corsa data plane) and tools built on top of it, such as RMON [116]. Furthermore, the Brocade, Mellanox and Arista switches support the sFlow monitoring protocol [103] for sampled packet export at Layer 2 and port mirroring (a.k.a. Switched Port Analyzer, SPAN), which duplicates all traffic of a designated subset of ports to monitoring ports for analysis. Moreover, Arista provides the CloudVision service [9] and Broadcom offers BroadView [31] to their switches, both enabling real-time streaming of telemetry data, an alternative to legacy polling with SNMP.

11 Acronyms

ODM Original Design Manufacturer

Euro-IX Euro-Internet eXchange

MAC Media Access Control

VLAN Virtual Local Area Network

IGP Interior Gateway Protocol

OSPF Open Shortest Path First

IS-IS Intermediate System to Intermediate System

RS Route Server

RIR Regional Internet Registry

IANA Internet Assigned Numbers Authority

IRR Internet Routing Registries

RADB Routing Assets DataBase

API Application Program Interface

ONOS Open Network Operating System

SDX Software Defined eXchange

DHCP Dynamic Host Configuration Protocol

CDP Cisco Discovery Protocol

LLDP Link Layer Discovery Protocol

CPU Central Processing Unit

CPC Consistent Policy Composition

ND Neighbor Discovery

NDv6 Neighbor Discovery version 6

ICMPv6 Internet Control Message Protocol version 6

L2 Layer 2

RPKI Resource Public Key Infrastructure

FIFO First-In First-Out

NIB Network Information Base

NP Non Polynomial

SSD Solid-State Drive

NIC Network Interface Controller

FPGA Field-Programmable Gate Array

RAM Random-access memory

NTP Network Time Protocol

ONF Open Networking Foundation

SNMP Simple Network Management Protocol

STM Software Transactional Memory

LAN Local Area Network

LDP Label Distribution Protocol

MPLS MultiProtocol Label Switching

LSP Label Switched Path

TRILL Transparent Interconnect of Lots of Links

RSTP Rapid Spanning Tree Protocol

IEEE Institute of Electrical and Electronics Engineers

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

EOS Extensible Operating System

SDN Software Defined Networking

RIB Routing Information Base

BGP Border Gateway Protocol

ISP Internet Service Provider

IXP Internet eXchange Point

QoS Quality of Service

SLA Service-Level Agreement

ISP Internet Service Provider

AS Autonomous System

IP Internet Protocol

DE-CIX German Commercial Internet Exchange

AMS-IX Amsterdam Internet Exchange

MSK-IX Moscow Internet Exchange

LINX London Internet Exchange

PoP Point of Presence

VPN Virtual Private Network

DDoS Distributed Denial of Service

VPLS Virtual Private LAN Services

UDP User Datagram Protocol

EVPN Ethernet Virtual Private Network

VXLAN Virtual Extensible Local Area Network

ARP Address Resolution Protocol

ND Neighbor Discovery

ACL Access Control List

References

[1] K. Agarwal, C. Dixon, E. Rozner, and J. Carter. Shadow macs: Scalable label-switching for commodity ethernet. In *Hot Topics in Software Defined Networking (HotSDN)*. ACM, 2014.

[2] B. Ager, N. Chatzis, A. Feldmann, N. Sarrar, S. Uhlig, and W. Willinger. Anatomy of a Large European IXP. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 163–174. ACM, 2012.

[3] AMS-IX. Allowed traffic types on unicast peering lans. `http://ams-ix.net/technical/specifications-descriptions/allowed-traffic`.

[4] Ams-ix annual reports. `https://ams-ix.net/about/annual-reports--2`.

[5] Follow-up on previous incident at AMS-IX platform. `https://ams-ix.net/newsitems/195`.

[6] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. In *POPL*, 2014.

[7] Arista. Arista 7050SX Series 10/40G Data Center Switches datasheet. `https://www.arista.com/assets/data/pdf/Datasheets/7050SX-128_Datasheet.pdf`.

[8] Arista. Arista 7124FX Application Switch datasheet. `http://www.arista.com/assets/data/pdf/7124FX/7124FX_Data_Sheet.pdf`.

[9] Arista. Arista EOS CloudVision: Cloud Automation for Everyone. `https://www.arista.com/assets/data/pdf/Whitepapers/CloudVision_WP_0815.pdf`.

[10] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. Enabling End Host Network Functions. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, August 2015.

[11] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. OpenState: Programming Platform-Independent Stateful Openflow Applications Inside the Switch. In *SIGCOMM Computer Communication Review (CCR)*, volume 44, pages 44–51, April 2014.

[12] BigSwitch. Switch Light. `http://www.bigswitch.com/products/switch-light`.

[13] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review (CCR)*, 44(3):87–95, July 2014.

[14] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 99–110, August 2013.

[15] V. Boteanu and H. Bagheri. Minimizing arp traffic in the ams-ix switching platform using openflow. Master's thesis, Universiteit van Amsterdam, the Netherlands, 2013.

[16] Brocade. Brocade VDX 8770 Switch datasheet. `http://www.brocade.com/content/dam/common/documents/content-types/datasheet/brocade-vdx-8770-ds.pdf`.

[17] Brocade. Exploring Software-Defined Networking with Brocade. `http://www.brocade.com/content/dam/common/documents/content-types/whitepaper/exploring-sdn-wp.pdf`.

[18] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, November 2006.

[19] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *NSDI*, 2005.

[20] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. Software Transactional Networking: Concurrent and Consistent Policy Composition. In *HotSDN*, 2013.

[21] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *Proceedings of INFOCOM'15)*, Apr 2015.

[22] Martin Casado, Nate Foster, and Arjun Guha. Abstractions for Software-Defined Networks. *Commun. ACM*, 57(10), 2014.

[23] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. Simple network management protocol (snmp), 1990.

[24] Ignacio Castro, Juan Camilo Cardona, Sergey Gorinsky, and Pierre Francois. Remote Peering: More Peering without Internet Flattening. In *Proceedings of CoNEXT*. ACM, 2014.

[25] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, August 2007.

[26] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4), July 1996.

[27] B. Charron-Bost, F. Pedone, and A. Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

[28] Angela Chiu, Vijay Gopalakrishnan, Bo Han, Murad Kablan, Oliver Spatscheck, Chengwei Wang, and Yang Xu. Edgeplex: Decomposing the provider edge for flexibilty and reliability. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, SOSR' 15, New York, NY, USA, 2015. ACM.

[29] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 251–264, October 2012.

[30] Broadcom Corp. Broadcom BCM56850 StrataXGS Trident II Switching Technology. https://www.broadcom.com/collateral/pb/56850-PB03-R.pdf.

[31] Broadcom Corp. Building an Open Source Data Center Monitoring Tool Using Broadcom BroadView Instrumentation Software. https://www.broadcom.com/collateral/tb/BroadView-TB200-R.pdf.

[32] Broadcom Corp. Engineered Elephant Flows for Boosting Application Performance in Large-Scale CLOS Networks. https://www.broadcom.com/collateral/wp/OF-DPA-WP102-R.pdf.

[33] Broadcom Corp. OpenFlow Data Plane Abstraction (OF-DPA): Abstract Switch Specification. https://www.broadcom.com/docs/support/OF-DPA-Specs_v2.pdf.

[34] Corsa Technology. Corsa Product Overview – DP64xx Data Plane Family. http://www.corsa.com/wp-content/uploads/2014/11/Corsa-Product-Overview.pdf.

[35] Corsa Technology. http://www.corsa.com/.

[36] X. Defago, A. Schiper, and P. Urban. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys (CSUR)*, 36:372–421, December 2004.

[37] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight Failure Detection Bounds on Atomic Object Implementations. *J. ACM*, 57(4), 2010.

[38] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. Towards an Elastic Distributed SDN Controller. In *HotSDN*, 2013.

[39] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1), January 1987.

[40] 26th Euro-IX Forum. https://www.euro-ix.net/events/52#event.

[41] Facebook. Introducing "Wedge" and "FBOSS," the next steps toward a disaggregated network. https://code.facebook.com/posts/681382905244727/introducing-wedge-and-fboss-the-next-steps-toward-a-disaggregated-network/.

[42] Andrew Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 327–338, August 2013.

[43] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2), 1985.

[44] Soudeh Ghorbani and Brighten Godfrey. Towards Correct Network Virtualization. In *HotSDN*, 2014.

[45] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown. NetFPGA – An Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers. *IEEE Transactions on Education*, 51(3):160–161, August 2008.

[46] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable Consistency in Scatter. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–28, October 2011.

[47] A. Gupta, L. Vanbever, M. Hahbaz, S.P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. Sdx: A software defined internet exchange. In *SIGCOMM*. ACM, 2014.

[48] Trinabh Gupta, Joshua B. Leners, Marcos K. Aguilera, and Michael Walfish. Improving Availability in Distributed Systems with Failure Informers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 427–441, April 2013.

[49] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *HotSDN*, 2012.

[50] Brandon Heller, Rob Sherwood, and Nick McKeown. The Controller Placement Problem. In *HotSDN*, 2012.

[51] Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.

[52] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-sized Data Structures. In *PODC*, 2003.

[53] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[54] N. Hilliard, E. Jasinska, R. Raszuk, and N. Bakker. Internet exchange route server operations. Technical report, 2014.

[55] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 15–26, August 2013.

[56] M. Hughes, M. Pels, and H. Michl. Internet exchange point wishlist. `https://www.euro-ix.net/documents/1288-ixp-wishlist-pdf`, 2013. [Online; accessed 01-Dec-2014].

[57] Jong Hun Han, Prashanth Mundkur, Charalampos Rotsos, Gianni Antichi, Nirav Dave, Andrew W. Moore, and Peter G. Neumann. Blueswitch: Enabling Provably Consistent Configuration of Network Switches. In *11th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, April 2015.

[58] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[59] Elisa Jasinska, Nick Hilliard, Robert Raszuk, and Niels Bakker. Internet exchange route server. Internet-Draft draft-jasinska-ix-bgp-route-server-03, IETF Secretariat, October 2011. `http://www.ietf.org/internet-drafts/draft-jasinska-ix-bgp-route-server-03.txt`.

[60] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *SIGCOMM*, 2014.

[61] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 297–312, April 2013.

[62] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic Scheduling of Network Updates. In *SIGCOMM*, 2014.

[63] Yuh-Jzer Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4), 2000.

[64] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental Consistent Updates. In *HotSDN*, 2013.

[65] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[66] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[67] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[68] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9), 1979.

[69] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[70] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[71] M. Lasserre and V. Kompella. Virtual private lan service (vpls) using label distribution protocol (ldp) signaling. RFC 4762, RFC Editor, January 2007. `http://www.rfc-editor.org/rfc/rfc4762.txt`.

[72] Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, and Anja Feldmann. Panopticon: Reaping the Benefits of Incremental SDN Deployment in Enterprise Networks. In *USENIX ATC*, 2014.

[73] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.

[74] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring connectivity via data plane mechanisms. In *NSDI*, 2013.

[75] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies. In *HotNets*, 2014.

[76] Ratul Mahajan and Roger Wattenhofer. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.

[77] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. RFC 7348, RFC Editor, August 2014. `http://www.rfc-editor.org/rfc/rfc7348.txt`.

[78] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NetAgg: Using Middleboxes for Application-Specific On-Path Aggregation in Data Centres. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 249–262, December 2014.

[79] Parisa Jalili Marandi, Samuel Benz, Fernando Pedone, and Kenneth P. Birman. The Performance of Paxos in the Cloud. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 41–50, October 2014.

[80] P.J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A High-Throughput Atomic Broadcast Protocol. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 527 –536, June 2010.

[81] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. Efficient Synthesis of Network Updates. In *PLDI*, 2015.

[82] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review (CCR)*, 38(2):69–74, March 2008.

[83] T. Mizrahi, O. Rottenstreich, and Y. Moses. TimeFlip: Scheduling Network Updates with Timestamp-based TCAM Ranges. In *INFOCOM*, 2015.

[84] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor discovery for ip version 6 (ipv6). Technical report, 2007.

[85] Netronome. FlowNICs – Accelerated, Programmable Interface Cards. `http://netronome.com/product/flownics`.

[86] Netronome. NFP-6xxx - A 22nm High-Performance Network Flow Processor for 200Gb/s Software Defined Networking, 2013. Talk at HotChips by Gavin Stark. `http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.60-Networking-epub/HC25.27.620-22nm-Flow-Proc-Stark-Netronome.pdf`.

[87] NoviFlow. NoviSwitch 1132 High Performance OpenFlow Switch datasheet. `http://noviflow.com/wp-content/uploads/2014/12/NoviSwitch-1132-Datasheet.pdf`.

[88] NoviFlow. NoviSwitch 2128 High Performance OpenFlow Switch datasheet. `http://noviflow.com/wp-content/uploads/NoviSwitch2128Datasheet.pdf`.

[89] B.M. Oki and B.H. Liskov. Viewstamped Replication: A General Primary-Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, August 1988.

[90] SDN Performance: Raising the bar on SDN control plane performance, scalability, and high availability. http://onosproject.org/wp-content/uploads/2014/11/PerformanceWhitepaperBlackbirdrelease-technical.pdf.

[91] ONOS Wiki Home. https://wiki.onosproject.org/display/ONOS/ONOS+Wiki+Home.

[92] Open-IX. Ixp technical requirements oix-1. http://www.open-ix.org/standards/ixp-technical-requirements.

[93] Open Network Linux. http://opennetlinux.org/.

[94] Openflow switch specification. http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[95] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 471–484, February 2014.

[96] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26, 1979.

[97] F. Pedone and S. Frolund. Pronto: A Fast Failover Protocol for Off-the-Shelf Commercial Databases. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 176–185, October 2000.

[98] F. Pedone and A. Schiper. Optimistic Atomic Broadcast: A Pragmatic Viewpoint. *Theoretical Computer Science*, 291:79–101, January 2003.

[99] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving Agreement Problems with Weak Ordering Oracles. In *European Dependable Computing Conference (EDCC)*, October 2002.

[100] I. Pepelnjak. Could ixps use openflow to scale? The Middle East Network Operators Group (MENOG), 2012.

[101] Peter Perešíni, Maciej Kuźniar, Marco Canini, and Dejan Kostić. ESPRES: Transparent SDN Update Scheduling. In *HotSDN'14*, Aug 2014.

[102] R. Perlman, D. Eastlake, D. Dutt, S. Gai, and A. Ghanwani. Routing bridges (rbridges): Base protocol specification. RFC 6325, RFC Editor, July 2011. http://www.rfc-editor.org/rfc/rfc6325.txt.

[103] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. Technical report, RFC 3176, 2001.

[104] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2015.

[105] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[106] Philipp Richter, Georgios Smaragdakis, Anja Feldmann, Nikolaos Chatzis, Jan Boettger, and Walter Willinger. Peering at Peerings: On the Role of IXP Route Servers. In *Proceedings of ACM IMC 2014*, Vancouver, Canada, November 2014.

[107] A. Sajassi, R. Aggarwal, N. Bitar, A. Isaac, J. Uttaro, J. Drake, and W. Henderickx. Bgp mpls-based ethernet vpn. RFC 7432, RFC Editor, February 2015. http://www.rfc-editor.org/rfc/rfc7432.txt.

[108] Stefan Schmid and Jukka Suomela. Exploiting Locality in Distributed SDN Control. In *HotSDN*, 2013.

[109] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, December 1990.

[110] A. Schwabe and K. Holger. Using mac addresses as efficient routing labels in data centers. In *Hot Topics in Software Defined Networking (HotSDN)*. ACM, 2014.

[111] D. Sciascia and F. Pedone. Geo-Replicated Storage with Scalable Deferred Update Replication. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, June 2013.

[112] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 1997.

[113] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Merlin: A Language for Provisioning Network Resources. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 213–226, December 2014.

[114] J. Stringer, D. Pemberton, Qiang Fu, C. Lorier, R. Nelson, J. Bailey, C.N.A. Correa, and C. Esteve Rothemberg. Cardigan: Sdn distributed routing fabric going live at an internet exchange. In *Symposium on Computers and Communications (ISCC)*. IEEE, 2014.

[115] Mellanox Technologies. Mellanox SX1036 datasheet. `https://www.mellanox.com/related-docs/prod_eth_switches/PB_SX1036.pdf`.

[116] S Waldbusser, R Cole, C Kalbfleisch, and D Romascanu. Introduction to the remote monitoring (rmon) family of mib modules. RFC 3577, RFC Editor, August 2003. `http://www.rfc-editor.org/rfc/rfc3577.txt`.

[117] M. Wessel and N. Sijm. Effects of ipv4 and ipv6 address resolution on ams-ix and the arp sponge. Master's thesis, Universiteit van Amsterdam, the Netherlands, 2009.

[118] S.H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On Scalability of Software-Defined Networking. *Communications Magazine, IEEE*, 51(2), 2013.