PRACTICAL CLOUD COMPUTING INFRASTRUCTURE

by

James Lembke

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Patrick Eugster, Chair
Computer Science

Dr. Kihong Park, Co-Char
Computer Science

Dr. Sonia Fahmy
Computer Science

Dr. Elena Grigorescu
Computer Science

Dr. Pierre-Louis Roman
Computer Science

Approved by:
Dr. Kihong Park

ACKNOWLEDGMENTS

Thanks to Patrick Eugster, Pierre-Louis Roman, and Srivatsan Ravi for their guidance and contributions to this work. Special thanks to Kihong Park for his advice and helpful critiques, as well as his assistance in organization of the presentation of this work. A big thank you to my wife, Pamela Lembke, for her patience and support. Finally a thank you to all of my committee members: Patrick Eugster, Kihong Park, Elena Grigorescu, Sonia Fahmy, and Pierre-Louis Roman.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Cloud and parallel computing are fundamental components in the processing of large data sets. Deployments of distributed computers require network infrastructure that is fast, efficient, and secure. Software Defined Networking (SDN) separates the forwarding of network data by switches (data plane) from the setting and managing of network policies (control plane). While this separation provides flexibility for setting network policies affecting the establishment of network flows in the data plane, it provides little to no fault tolerance for failures, either benign or caused by corrupted/malicious applications . Such failures can cause network flows to be incorrectly routed through the network or stop such flows altogether. Without protection against faults, cloud network providers using SDN run the risk of inefficient allocation of network resources or even data loss. Furthermore, the asynchronous nature existing protocols for SDN does not provide a mechanism for consistency in network policy updates across multiple switches.

In addition, cloud and parallel applications require an efficient means for accessing local system data (input data sets, temporary storage locations, etc.). While in many cases it may be possible for a process to access this data by making calls directly to a file system (FS) kernel driver, this is not always possible (e.g. when using experimental distributed FSs where the needed libraries for accessing the FS only exist in user space).

This dissertation provides a design for fault tolerance of SDN and infrastructure for advancing the performance of user space FSs. It is divided into three main parts. The first part describes a fault tolerant, distributed SDN control plane framework. The second part expands upon the fault tolerant approach to SDN control plane by providing a practical means for dynamic control plane membership as well as providing a simple mechanism for controller authentication through threshold signatures. The third part describes an efficient framework for user space FS access.

This research makes three contributions. First, the design, specification, implementation, and evaluation of a method for fault tolerant SDN control plane that is inter-operable with existing control plane applications involving minimal instrumentation of the data plane runtime. Second, the design, specification, implementation and evaluation of a mechanism

for dynamic SDN control plane membership that all ensure consistency of network policy updates and minimizes switch overhead through the use of distributed key generation and threshold signatures. Third, the design, specification, implementation, and evaluation of a user space FS access framework that is correct to the Portable Operating System Interface (POSIX) specification with significantly better performance over existing user space access methods, while requiring no implementation changes for application programmers.

1. INTRODUCTION

This dissertation advances the infrastructure for cloud computing through improvements to the management of network policies and file system (FS) access frameworks. Specially, it focuses on *consistency* and *fault-tolerance* in network policy management and FS access that is *practical* for deployment in production data-centers and complies with established standards.

1.1 Background

This dissertation focuses on improving consistency and fault-tolerance on two areas of cloud computing: network policy management and FS data storage with specific focus on providing an interface and evaluation showing that improvements are practical to be used in production.

1.1.1 Fault Tolerance and Consistency for Network Policy Management

Routing of network traffic within and between data-centers can vary significantly based on workloads [1]. Traditional network routing protocols [2], [3] are based on shortest paths, such as minimal number of "hops" or delay through the network, and match rules solely on the internet protocol (IP) address of the destination. However for many workloads, such routing heuristics do not provide optimal network utilization. For example, a longer path which sends traffic through a higher number of switches, while requiring additional latency, may offer significantly more bandwidth resulting in faster workload completion. Furthermore, link failures may take significant time to recover, during which, packets might be lost and/or need to be re-transmitted. As such, corporations such as Facebook and Google are moving to a more dynamic approach to network routing within their corporate networks. Software Defined Networking (SDN) provides a means for fine grained control of network routing policies through the use of a centralized controller application. These policies are installed in switches either through requests received from the switches themselves (e.g., in the event of an unroutable packet) or based on some other network event. In addition to being dynamically installed, specific network polices in SDN are far more flexible than those used in traditional network routing protocols allowing routes to be based on any field in the IP header.

However, the nature of a centralized controlling entity results in several concerns:

(1) performance bottleneck for processing requests

(2) single point of failure

A failed SDN controller is not able to perform policy updates to switches preventing new routes and network flows from being established. Several techniques have been proposed for decentralized SDN controllers [4], [5], however these focus on only solving (1) or on robustness in the event of crash-stop failures in which a controller stops responding to all events. A corrupted SDN controller, whether due to a malicious adversary or through software or hardware faults, can cause similar if not more harm to network data [6]–[8]. Therefore a need exists for a robust SDN controller capable of providing protection against more than just 'crash-stop' failures.

Furthermore, switches do not communicate with each other for the purpose of establishing network policies. The role of enforcing network policies is the complete responsibility of the SDN controller. Switches themselves forward packets based on flow table rules set by the controller in response to network events. It is, therefore, crucial to provide a mechanism to ensure that changes to the network routes in response to a network event be done consistently to avoid transient effects to prevent packet loss, data leakage, and or violation of network policies. In addition, data plane switches contain limited hardware resources (e.g. memory and CPU power) and, as such, a solution should require minimal switch instrumentation.

1.1.2 Consistency and Efficiency of File System Access

In a traditional computer system, persistent data is accessed via input/output (I/O) requests to files in a FS via their file descriptor (FD). The operating system (OS) kernel creates an abstraction of files and organizes requests to files translating them to operations performed against persistent media. This translation is traditionally performed by the FS driver which exists within the OS kernel. However, a trend has developed for FS drivers to execute outside of the OS kernel. Such user space FSs can be built using functions not available within the OS kernel, run with reduced privileges when compared to those implemented within the kernel, are often faster to implement, and do not require knowledge of internal kernel interfaces.

User space FS interfaces (e.g. Filesystem in Userspace (FUSE) [9]) exist, however they suffer performance penalties compared to FS drivers implemented within the OS kernel due to overhead imposed by communication between user space applications and kernel space. Requests originate as system calls from an application to the OS kernel. The kernel must then send corresponding messages to user space to service these requests. Each message sent to users space involves a process context switch and requires copying of data between kernel and user space memory buffers. This communication overhead greatly affects workloads where a large amount of small files are created and updated as is the case for many distributed computing workloads [10]. As a result, there exists a need for a new FS infrastructure for user space FSs that can provide:

1. access speeds near that of a kernel driver
2. similar safety guarantees as those provided by POSIX
3. limited or no needed modifications to client applications which access the FS

Existing solutions that attempt to mitigate these problems [11]–[17] do so using contrived mechanisms to emulate the FD interface provided by the OS kernel. As a result, are not consistent and correct in all cases and therefore inappropriate for a practical deployment of cloud applications.

1.2 Problem Statement

No protocol for network policy management exists that provides:

1. Consistency - protection against transient effects of asynchronous network updates
2. Security - protection against failures beyond crash-top

3. Practicality - minimal overhead to the completion of network flows, minimal instrumentation to switches, compatible with existing protocols for network updates, and scales to a large geographically expansive network.

Furthermore, no mechanism for FS access exists that provides:

1. Flexibility - ability to be implemented outside of a kernel driver while allowing application access using established standard system calls
2. Efficiency - access performance similar to kernel driver access
3. Consistency - semantically conforming to all guarantees of established system called standards

1.3 Thesis Statement

The thesis of this dissertation can be divided improvements in two aspects of cloud computing infrastructure: (1) robustness of communication infrastructure and (2) flexibility of data storage infrastructure. Specifically, we can provide a mechanism for network updates that provides *consistency* and *security* while being *practical* for deployment in production. We provide a mechanism for FS access that is *flexible*, *efficient*, and *consistent*.

1.4 Contributions

This dissertation makes the following contributions:

1. Robustness of network policy infrastructure:
   (a) Identification of issues that can arise from faults in existing SDN controllers as well as establishing a practical failure model for the SDN control plane.
   (b) A protocol providing strongly consistent updates to switches for this failure model that ensures progress of network policy updates assuming a correct majority of controllers.
   (c) An optimization of the protocol that exploits commutativity of network policy updates, increasing concurrency, and leading to improved performance.

(d) A mechanism allowing flexibility in the network controller layer allowing controllers to be added and removed without requiring a network outage.

(e) The design and implementation of the protocol extending the Ryu controller framework [18] and the Open vSwitch (OVS) runtime with minimal data plane instrumentation, that is compliant with OpenFlow [19], [20].

(f) An extensive evaluation including throughput and micro-benchmarks on both inter and intra data-center workloads that show that the overhead induced by the prototype is acceptable and in some cases achieves higher throughput than existing crash-only tolerant protocols.

2. Performance and flexibility of data storage infrastructure:

   (a) Identification of the problems with existing user space FS access methods.

   (b) The design of a new user space FS system centering on its novel concepts of bypassed FD lookup, FD stashing, and user space paging.

   (c) An analysis of the user space FS system implementation, with specific focus on its novel concepts and how together they retain Portable Operating System Interface (POSIX) compliance.

   (d) An extensive evaluation of the system's performance with focus on: raw file creation, command line tools (`tar`, `rsync`), and cloud applications using Spark [21] which show that it outperforms existing user space FS access methods by increasing throughput not only for cloud applications, but for general purpose accesses as well.

2. FAULT TOLERANCE FOR NETWORK POLICY UPDATES

This chapter models the interaction between the SDN data plane and a distributed SDN control plane consisting of a set of failure-prone and potentially malicious (compromised) control devices, and implements Robust SDN Controller (RoSCo) [22] a secure and robust controller platform that allows network administrators to integrate new network functionality as with a centralized approach. Concretely, the network administrator may program the data plane from the perspective of a centralized controller without worrying about distribution, asynchrony, failures, attacks, or coordination problems that any of these could cause.

2.1 Introduction

The Software Defined Networking (SDN) *control plane* allows for expressing and composing policies of varying networking applications and translating these to a combined *network policy*, entailing rules installed onto the switches for handling *network flows*. Consider a typical SDN computation model depicted in Figure 2.1: as a packet arrives at a switch port for which there is no matching entry in the switch's *flow table*, the switch generates an *event* sent to the controller platform. The controller subsequently installs a new rule on the switch and possibly on other switches as well.

2.1.1 Crash(-stop) failures

Common centralized controller deployments are trivially prone to crash failures, as a single halting failure of the controller process can disrupt control over the network. Recent research has identified the challenges in building a truly *distributed and fault-tolerant* SDN controller [5], [20], [25], [26]. The core challenge consists in ensuring the consistent installation of network policies under asynchronous and lossy communication. Specifically, a correct implementation of the controller platform must consider subtle conflicts between concurrently raised events and/or their resulting switch updates that entail flow table modifications, and provide progress (or availability) [5] in the face of crash failures.

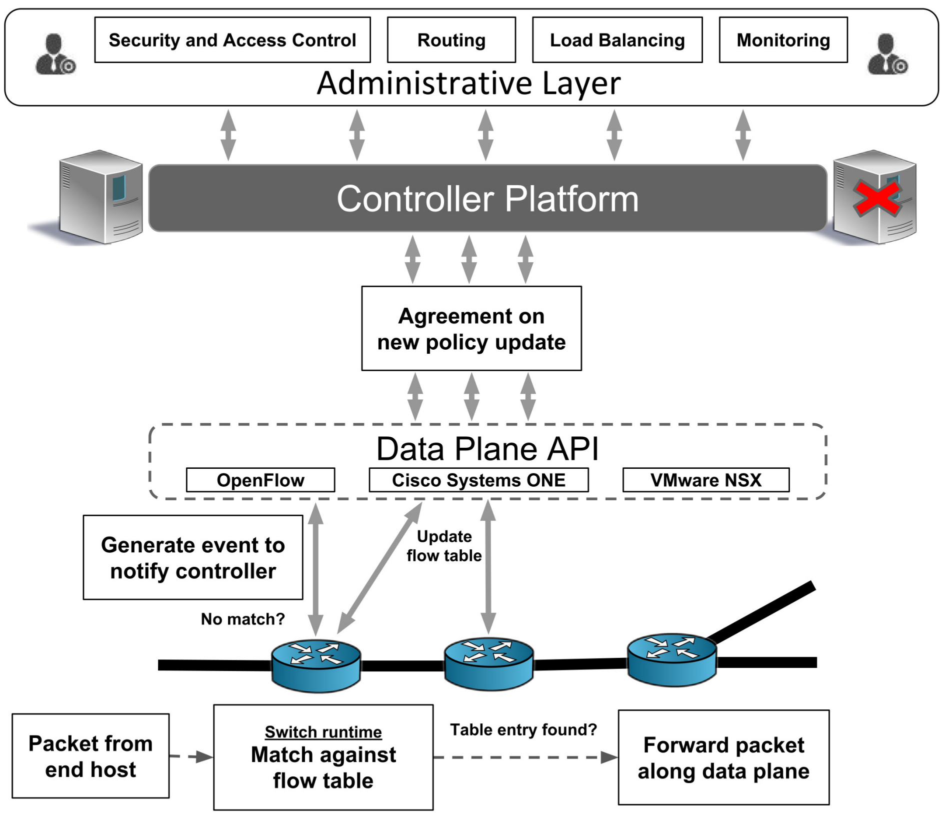


Figure 2.1. SDN system architecture: The logically centralized controller platform allows dynamic installation of network policies on the network switching fabric. Communication between switches and the controller platform is made through an established protocol interface (e.g., OpenFlow [19], Cisco ONE [23], VMware NSX [24]). Some subset of the participating controllers may be faulty. If there is no matching flow table entry for an incoming packet at a switch port, the switch runtime creates an event to be sent to the controller platform.

2.1.2 Controller faults

Moreover, whilst several widely adopted SDN controller platforms like Onix [27] and ONOS [28] provide some pragmatic forms of distribution, these models focus solely on crash failures where distributed controllers cease to respond entirely. Little research effort has been directed towards investigating real-life fault models in which faulty SDN controllers continue to respond, but with corrupted messages. Such situations may happen due to a variety of causes including software flaws or transmission errors. For example, in July 2008 Amazon S3 suffered an outage caused by network communication errors that corrupted messages “such that the message was still intelligible, but the system state information was incorrect" [6]. Again, in 2012, Amazon AWS suffered another outage due to "a latent bug in an operational data collection agent" [7]. Google App Engine also suffered a several hour outage as the "result of a bug in our datastore servers [...] triggered by a particular class of queries" [8]. In all these cases, the failure was not caused by a crash. While these examples focus on general network applications, the principles behind the outages can easily carry over to any SDN controller model. Typical SDN policies are based on matching of a packet prefix. When applying such SDN policies, even the corruption of a single bit can cause a mismatch resulting in incorrect data plane routing.

Furthermore, as Kreutz et al. argue in a position paper [29], several threat vectors motivate the need for a *secure* and *dependable* SDN controller platform, including forged or faked flows, attacks on vulnerabilities in switches and on control plane communication. The effects of such malicious (à la *Byzantine*) adversaries [30], [31] in fact can manifest similarly to many benign faults mentioned earlier such as corrupted communication. In summary, once an SDN switch is thus faulty or compromised (e.g., due to collocation with a compromised application or virtual switch [32]), it may perform any of the following actions: send arbitrary messages within the control plane and to switches in the data plane as well as arbitrarily delay/intercept/modify traffic between controllers and switches [29], eventually compromising the entire data plane. While it is not immediate under what circumstances vulnerabilities cause control devices to exhibit the full spectrum of Byzantine behavior [33], threat vectors inherent to SDN make it vital that the SDN controller platform provides provable correctness against faults beyond crashes.

2.1.3 Existing results and model uniqueness

Though many fundamental results and bounds from traditional fault-tolerant distributed computing still apply to address the consistency of data plane updates in the distributed SDN context, those are based on abstractions/models which are both too generic and specific. More specifically, the problems addressed in the SDN model considered in this research differ from traditional setups in that the latter: (1) consider clients concurrently issuing requests to servers, while the present SDN context has switches issue events to controllers, whose handling yields updates for several switches and not only the one raising the event; (2) assume a single homogeneous network whilst the SDN context further distinguishes between the data plane network and controller-switch network (and possibly client-controller network) which can be physically disjoint; (3) do not typically make assumptions about state and logic managed and shared among servers whereas the SDN logic [34] (e.g., network flows/flow tables) has well-known specific semantics and constraints. As a result, traditional distributed computing solutions deployed as black-boxes are unlikely to be efficient in the SDN computation and threat model.

2.1.4 Technical constraints

A dependable practical SDN controller platform, apart from providing consistency, integrity and availability for applying network policies, must also come with a simple computation and programming model. Specifically, it must allow the network administrator to program the network switching fabric with the ease of programming a centralized SDN controller. For example, unlike with our work, the Hyperflow [35] replicated SDN controller platform requires the application itself to actually manage the replicated state, thus increasing the burden on programmers. A second important requirement for the SDN controller platform is to keep the *instrumentation* on the switch runtime minimal. For example, it may be perfectly plausible to deploy a variant of Paxos [36] (for *crash* failures) as is the case with [26] or deploy Byzantine fault-tolerant (BFT) [37] protocols across every switch and controller, but this can incur complex instrumentation, increase usage of switch runtime resources (which may be limited), and require messaging that is non-compliant with existing data plane APIs like OpenFlow [19], [20].

2.2 Related work

This section presents related work on consistency and fault tolerance in distributed SDN controllers. Please see literature surveys [34], [38] for general overviews of SDN controllers.

2.2.1 Consistency formalizations for network updates

The event linearizability property presented here is based on the original definition for traditional distributed computing [39] and was adapted to the SDN context previously in [40], however our definition expands on this to generalize linearizability in malicious contexts as well as shows the impossibility result which holds even for non-blocking environments. consistent updates (CU) [41] ensure that a packet is processed during an execution either entirely by an old policy or a new one, but never a mix of both. This property is also guaranteed by the strictly stronger property of event linearizability since it reduces the concurrent network update to an equivalent sequential one. network event structures (NES) [42] provide a mechanism for *causally* consistent policy updates where multiple switch updates can cause uncertainty in packet forwarding. However, NES require instrumented data plane switches with the ability to modify packets at ingress and egress switches. One of the goals of RoSCo is to allow consistent switch updates with minimal instrumentation of forwarding logic; additionally causal consistency is weaker than the event linearizability safety property, but incomparable to the update consistency safety property. Unlike RoSCo, NES can not handle malicious controller processes. Several papers study and identify different customizable consistency properties for SDN network updates [43]–[48], but none considers malicious adversaries. Deriving protocols for dealing with malicious adversaries and extending our framework for such consistency definitions is ongoing work.

2.2.2 Distributed SDN controller platforms

The SDN computation model considered in this chapter explicitly separates the data plane and control plane; concretely, it requires that the state of any two switches in the data plane be mutually disjoint and this precludes the use of agreement protocols within the switch runtime. More generally, distributed computing solutions like state machine replication may not be used as a black-box for network updates as is the case in the Net-Paxos protocol [26]. Onix [27] and ONOS [28] are two of the earliest proposals to motivate the need for a distributed control plane. Both suffer from two fundamental drawbacks: the lack of consistency in network updates and the inability to cope with malicious failures in the control plane as well as attacks on network communications which may disrupt network updates. Onix' uses of Zookeeper [49] and ONOS' use of Raft [50] — while providing a mechanism for distributed agreement used for consistent log replication — only provides crash fault tolerance. Akella and Krishnamurthy [25] introduce the study of crash tolerance in distributed SDN controllers and sketch a solution that involves even switches participating in a distributed agreement protocol which is extremely costly. Beehive [51] implements a highly scalable distributed controller platform, but does not provide any linearizability guarantees for network updates nor does it provide resilience against malicious adversaries as RoSCo does. Ravana [5] includes a distributed controller synchronization protocol that provides consistent network updates, but does not deal with malicious adversaries. Rosemary [52] and LegoSDN [53] both provide a protection framework for SDN applications to prevent malicious behaviors and crashes however both focus on resilience of the controller and the application in a single controller environment. In contrast to RoSCo the frameworks do not provide protection against an arbitrary host masquerading as a controller sending malicious unsolicited flow rules to switches. Renaissance [54] is self-stabilizing and hence can tolerate arbitrary and in particular malicious behavior, however, the distributed control plane can be inconsistent for longer periods and only recover after re-convergence occurred.

2.2.3 Malicious adversaries in SDN

Fleet [55] initiated the problem of dealing with malicious adversaries in the SDN context. Just as RoSCo, Fleet sketches a protocol that ensures progress as long as hoNESt majority of controllers exist; but it does not provably satisfy event linearizability or its relaxation. Moreover, unlike RoSCo, Fleet requires extensive data plane instrumentation and their simulation does not provide a characterization of the cost of providing resilience against malicious adversaries nor the subtle protocol differences associated with different consistency properties. The position paper of Kreutz et al. [29] provides a good overview of the motivation for secure and *dependable* SDN controller platforms by discussing threat vectors. Though discussing standard techniques for mitigating different threat combinations, the authors do not actually flesh out a concrete solution for a particular adversary.

2.3 Motivation and Model

We first briefly motivate the need for consistency in applying network policies before outlining our model of failure/threats and computation. Using OpenFlow, an open standard protocol providing a mechanism for communication between the SDN data plane and control plane, switches send events to and receive updates from controllers via a network connection which imposes a nonzero network delay. This delay can cause uncertain and undesirable behaviors even while all processes are functioning as designed, as illustrated in the following.

Table 2.1. Comparison of SDN controller platforms fault tolerance, application interfaces, and complexity.

	Ravana [5]	Ryu [18]	Onix [27]	NetPaxos [26]	RoSCo
Consistency	Observational indistinguishability[a]	Sequential specification	Partial event ordering[b]	Partial event ordering[b]	Event linearizability[c] & update consistency[d]
Distributed	Yes	No	Yes	Yes	Yes
Switch instr. & complexity	Minimal	None	High	Very high	Minimal
Programming model	Centralized	Centralized	Maintain network information base	Centralized	Centralized
Adversary	Crash	Centralized failure	Crash	Crash	Malicious

[a]Observational indistinguishability is the guarantee that any observation of events viewed in distributed controller model is the same as those viewed in the centralized model.
[b]Partial order allows for independent events to be observed in any order.
[c]Event linearizability ensures a total order on event observations.
[d]Update consistency ensures coherence of dependent network updates for flows across multiple switches.

RoSCo is the only distributed controller platform that provides complete linearizability with total event ordering and tolerates faults incurred by malicious actors. Others do provide greater consistency over the centralized model, but either require significant switch instrumentation (Onix, NetPaxos) or only tolerate crash failures (Ravana).

2.3.1 Example

Consider the network topology shown in Figure 2.2a consisting of five OpenFlow compatible switches (s_{1-5}). The arrows represent the current flows routing network traffic to switch s_5. At a particular time, the link between s_4 and s_5 fails. s_4 detects the failure and sends an event to the controller.

The controller requires a network update to alter flows destined for s_5 as shown in Figure 2.2b. This network update involves three individual switch updates; one at s_2, s_3, and s_4. In the SDN model, data plane switches do not communicate with each other for network updates. As such, if the controller sends out all switch updates in parallel, the update at s_3 might be processed first, resulting in an unintended loop in the network as shown in Figure 2.2c. Eventually the switch update at s_2 will be processed and the loop will be removed however, during the time that the network loop exists, significant switch buffer resources may be consumed affecting availability of the network. Performing the switch updates sequentially starting with the update at s_2, followed by s_3, and finally s_4 would prevent the possibility of a network loop. While solutions to this problem have been well-understood for a centralized controller platform, our solution focuses on maintaining strong consistency in a distributed controller environment, the details of which are described in § 2.5.

2.3.2 Failure and threat model

We consider a failure/threat model in which SDN controllers, besides crashing, may become faulty or (partially) compromised as specified by Kreutz et al. [29]. Similarly a network host (not part of the existing control plane) may also be compromised and masquerade as an SDN controller. Such an adversary is allowed to perform any of the following actions on the network: send any arbitrary message to switches in the data plane; send any arbitrary message to other controllers; eavesdrop traffic on the communication network between controllers in the control plane; eavesdrop traffic on the communication network between switches and controllers; intercept and modify traffic between controllers and switches. While it might be the case that the physical medium between switches in the data plane is the same as that between the data plane and the control plane, an adversary may not modify the contents of packets sent between switches in the data plane. Specifically, an adversary may not force the dropping of data packets sent between switches other than through modifications to switches' flow tables. A switch in the data plane may take indefinitely long to respond to controller messages due to network asynchrony. The goal of RoSCo protocols for network updates is to limit the impact an adversary can have on the network as a whole. In the worst case, an adversary should only be able to prevent forward progress. It should not be allowed to alter flows or cause incorrect routing of data packets. As discussed in § 2.5, RoSCo ensures that correct policies are applied to switches through the use of quorum authentication. However, our failure model does not include the case of a dishonest majority through concurrent software or hardware failures or a coordinated attack. Approaches tackling such scenarios e.g., utilizing multiple versions of software and/or intrusion detection [56], [57] have been well established in existing research. As discussed in § 2.6, RoSCo is designed to be flexible to accommodate these approaches.

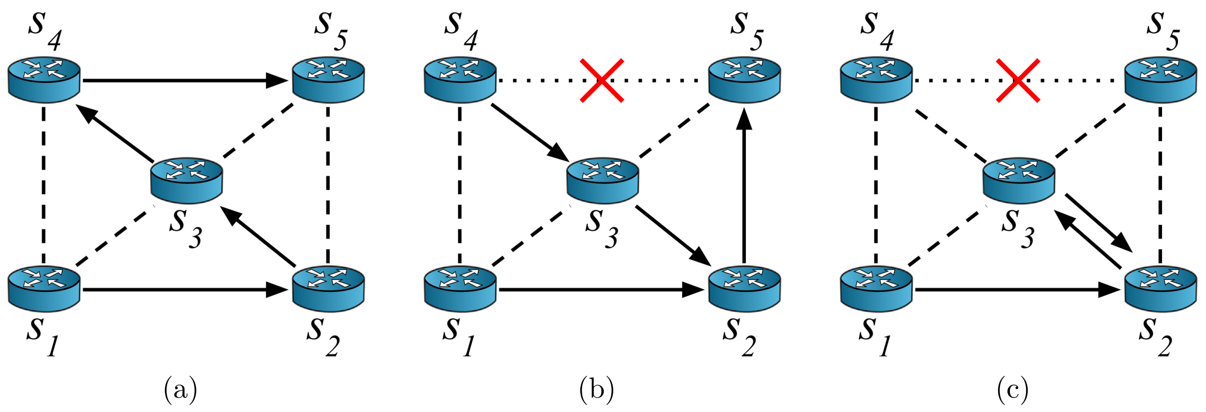


Figure 2.2. Figure 2.2a depicts the state of an SDN controlled network using OpenFlow with five switches and the active flows to switch s_5. Figure 2.2b depicts the intended flows after a network update due to a failure of the link between s_4 and s_5. Figure 2.2c depicts an unintended network loop because switch s_3 applied its switch update before s_2 did.

2.3.3 Computation model

Characterizing the properties (outlined in § 2.1) associated with applying network policies and establishing protocol correctness, besides a failure/threat model, necessitates the need for a computation model. Holistically comparing and evaluating different SDN controller platforms requires us to precisely specify the following crucial properties: what is the *consistency* property that defines the guarantees for within the algorithm for how flow table modifications are made; whether the SDN controller is *distributed* across multiple instances; what are the inherent *switch instrumentation (instr.) and complexity* costs that define the amount of additional instrumentation is required to data plane switches in order to properly use the SDN controller; what is the *programming model*, i.e., the amount of knowledge that a controller application and administrator must have of the underlying protocol.

2.3.4 RoSCo in context

Table 2.1 summarizes pros and cons of popular existing distributed controller platforms and how they compare against RoSCo, described in the following sections. Non-distributed, centralized controller platforms such as Ryu [18], while offering consistent ordering of events in the control plane and centralized administration, do not offer protection from failures. Onix [27] and NetPaxos [26] provide centralized administration and protection from crash failures, however require significant switch instrumentation. Ravana [5] provides *observational indistinguishability* of events, centralized administration, and protection against crash failures however does not protect against other faulty behavior. RoSCo combines the benefits of event linearization and protection from failures (as described in our failure and threat model) while still offering centralized administration and minimal switch instrumentation. Moreover, RoSCo also implements a weakly consistent update procedure and presents an empirical characterization of the induced overhead over the event linearizable protocol.

2.4 Formalizing SDN Computation

Table 2.2. Summary of model notation.

Symbol	Definition
c	Controller process
s	Switch process
π	Network policy
e	Network event
u	Switch update
U	Flow update (seq. of switch updates)
$\mathcal{U}$	Network update (seq. of switch updates)
$\mathcal{E}$	Execution of a network update
$\mathcal{I}$	Controller policy implementation
$\mathcal{H}$	Execution history
$\mathsf{upd}[S(\mathrm{e})]$	Set of switches to be updated as a result of event e
$<_{\mathcal{E}} \quad <_{\mathcal{H}}$	Total order in $\mathcal{E}$ or in $\mathcal{H}$
$\pi_{\mathrm{i}} \prec_{\mathcal{E}} \pi_{\mathrm{j}}$	Precedence of network policies in $\mathcal{E}$

In this section, we provide a formal model of computation for applying SDN policies and prove a fundamental result on the nature of progress: it is impossible to implement *non-*

blocking and strongly consistent application of network policies. Table 2.2 summarizes the notation used thereon. Readers interested in the details of the RoSCo protocol which focus on overcoming this impossibility result assuming an *honest majority* of controllers may wish to proceed directly to § 2.5.

2.4.1 Control plane and data plane

We consider an asynchronous *controller* system in which a set $\mathcal{C} = \{c_1, \dots c_n\}$ controller processes communicate by *sending* and *receiving* messages. The *data plane* is a set $\mathcal{S} = \{s_1, \dots, s_m\}$ of *switches* and a set $\mathcal{L} \subseteq \mathcal{S} \times \mathcal{S}$ of links, either of which may *fail*. We consider a full communication model in which each controller process may send messages to, and receive messages from, any other controller process or any switch. Switches communicate with each other solely for the purpose of sending data plane traffic.

2.4.2 Network policies

Following [40], we define the notion of a *network policy* which intuitively specifies the state of the flow tables in data plane switches for forwarding packets across the network. A network policy π specifies the state (or *flow tables*) of each switch in the data plane. An *event* is initiated by a switch or a controller and results in a *network update* to apply a network policy to the flow tables of some subset of switches. We assume that the application of a network policy π begins with an *event invocation* by a switch followed by a *network update*. A network update consists of a sequence of *switch updates* u.

Note that, as outlined via Figure 2.1, network policies themselves are either set by one or more administrators or generated through a controller application operating in a layer above the controller platform. However without loss of generality and for the simplicity of the model, we assume that a network policy change is initiated by a switch, but the control plane may itself initiate new network policies on the data plane. Furthermore, as noted in Figure 2.1, network policies are set as part of the controller application in a layer above the controller platform.

2.4.3 Executions and configurations

A *switch update* is the modification of the flow table for a switch with the given rule. A *step* of a network update $\mathcal{U}$ is a switch update $u$of $\mathcal{U}$ or a *primitive* (e.g., message send/receive, atomic actions on process memory state, etc.) performed during $\mathcal{U}$ along with its response. A *configuration* (of an SDN implementation) specifies the state of each switch and the state of each controller process. The *initial configuration* is the configuration in which all switches have their initial flow table entries and all controllers are in their initial states. An *execution fragment* is a (finite or infinite) sequence of steps. An *execution* $\mathcal{E}$ of an implementation $\mathcal{I}$ is an execution fragment where, starting from the initial configuration, each step is issued according to the implementation $\mathcal{I}$ and each response of a primitive matches the state resulting from all preceding steps. Two executions $\mathcal{E}_{\mathrm{i}}$ and $\mathcal{E}_{\mathrm{j}}$ are *indistinguishable* to a set of control processes and switches if each of them take identical steps in $\mathcal{E}_{\mathrm{i}}$ and $\mathcal{E}_{\mathrm{j}}$.

2.4.4 Ideal world specification of network policies

Defining the correctness of a concurrent network update protocol resilient against faulty controllers requires specifying the *ideal-world functionality* of the network policy, i.e., how the policy would be implemented on the data plane by a trusted centralized controller. We model this as a standard *mealy machine*: every network policy π_{i} has a *deterministic sequential specification*, i.e., in the absence of concurrency, π_{i} beginning with an event invocation at data plane configuration C_{j} followed by a network update, terminates by returning a response r_{i} and moves the data plane to some configuration $C_{\mathrm{j+1}}$.

Note that, as outlined via Figure 2.1, ideal world specification of network policies are determined by one or more administrators or generated through a controller application operating in a layer above the controller platform. The goal of the RoSCo protocols is to enforce this ideal world specification in a highly concurrent failure/threat model.

2.4.5 Control plane and data plane instrumentation

We assume that the control plane and data plane are separate and the data plane switches are themselves mutually independent as is the case in data center SDN deployments [20]. This typically allows for the data plane switches to be deployed with *minimal instrumentation*, i.e., largely forwarding packets according to matching flow table rules [34]. Formally, for every execution $\mathcal{E}$ and any event e invoking a network update $\mathcal{U}$ initiated by switch s, every extension of $\mathcal{E}$ is indistinguishable to s from an execution in which $\mathsf{upd}[\mathcal{S}(\mathrm{e})] = \{s\}$. Here, $\mathsf{upd}[\mathcal{S}(\mathrm{e})] = \{s\}$ denotes the set of switches whose flow tables must be updated on completion of $\mathcal{U}$ as a result of the invocation of e by s.

2.4.6 Network updates & consistency

We now define our consistency properties for network updates.

Strongly consistency or event linearizability. We first define *strongly* consistent network updates, inspired by the definition presented by Canini et al. [40]. To formalize the definition, we introduce the following technical language. A policy π_i *precedes* another policy π_j in an execution $\mathcal{E}$, denoted $\pi_i \prec_{\mathcal{E}} \pi_j$, if the network update for π_i occurs before the network update of π_j in $\mathcal{E}$. If none of two policies π_i and π_j precede the other, we say that π_i and π_j are *concurrent.* An execution without concurrent policies is a *sequential execution.* A network policy is *complete* in an execution $\mathcal{E}$ if the invocation event is followed in $\mathcal{E}$ by a *matching* network update; otherwise, it is *incomplete.* Execution of $\mathcal{E}$ is *complete* if every policy in $\mathcal{E}$ is complete. A *high-level history* H of an execution $\mathcal{E}$ is the subsequence of $\mathcal{E}$ consisting of the network policy event invocations, network updates and network policy responses.

An execution $\mathcal{E}$ is *event linearizable* [39] if there exists a sequential high-level history S equivalent to some *completion* of H such that (1) $\prec_H \subseteq \prec_S$ (policy precedence is respected) and (2) *S respects the sequential specification of policies in H.* A SDN implementation $\mathcal{I}$ implements event linearizable network updates if every execution $\mathcal{E}$ of $\mathcal{I}$ is linearizable.

Update consistency. As depicted in Figure 2.2, the response to an event e may involve updates to multiple switches, each of these updates requiring a specific order of execution to maintain consistency. Furthermore, the handling of an event by the controller application is affected not only by the event itself but also the controller application's global view of the network (state). As such, the ordering of events as input to the controller application is also important.

Event linearizability stipulates that network update $\mathcal{U}_1$ must take effect before $\mathcal{U}_2$ takes effect. However, in some situations this strong guarantee is not necessary. We refer to a *flow update* as an ordered sequence of switch updates $[u_1, \ldots, u_n]$ generated by the controller application in response to some event. We refer to *flow consistency* as ordering of switch updates within a flow update. The need for such consistency is motivated in [44] as well as the example in § 2.3.1. Ordering is also required for updates to the same switch either within or across multiple flow updates. Such updates may occur as the result of multiple events handled by the controller application or if multiple flow updates are generated in the response to a single event. We refer to the ordering of updates to a single switch as *switch consistency*. Our definition for weak consistency, denoted, *update consistency* relaxes event linearizability whenever possible by allowing switch updates be applied in parallel provided both *flow consistency* and *switch consistency* are maintained.

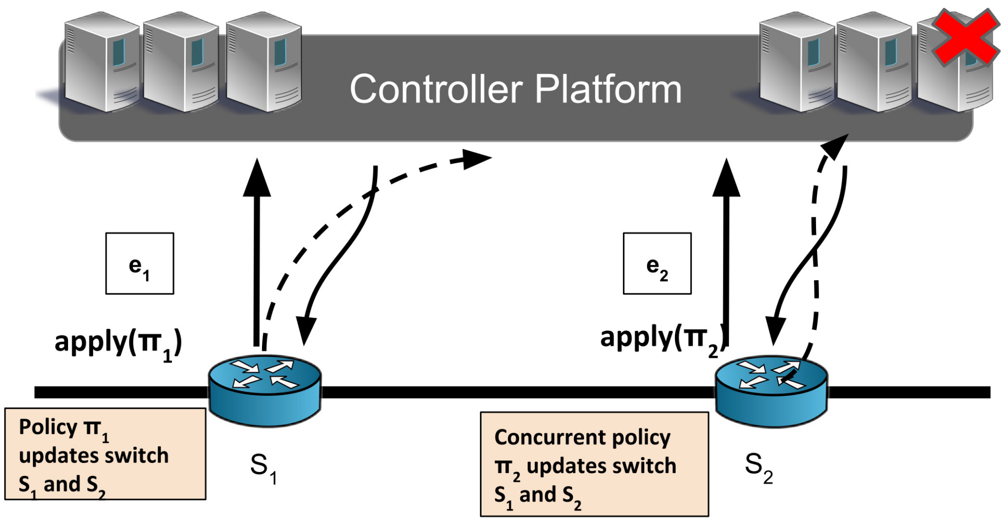

Controller Platform
e$_1$
e$_2$
apply(π$_1$)
apply(π$_2$)
Policy π$_1$ updates switch S$_1$ and S$_2$
S$_1$
Concurrent policy π$_2$ updates switch S$_1$ and S$_2$
S$_2$

Figure 2.3. Illustrating proof of Theorem 2.4.1: Construct an execution involving two non-commutative events e_1 and e_2 which must each update flow tables of both switches s_1 and s_2. By the *non-blocking* property: network updates π_1 and π_2 must complete. The uninstrumented data plane model implies s_1 and s_2 are oblivious of conflicting events. Thus, by the asynchronous nature of the adversary, updates involving e_1 and e_2 might interleave. E.g., switch s_1 may perform the update from e_1, but before s_2 performs its update for e_1, s_1 may perform the update for e_2, violating event linearizability.

2.4.7 Impossibility of non-blocking linearizable SDN control

We now establish a lower bound that places a fundamental limitation on the nature of progress in *asynchronous* SDN implementations: it is impossible to realize *non-blocking* and event linearizable network updates. Intuitively, non-blocking progress in the SDN context ensures that *some* network update always completes successfully independent of the controller failures and network asynchrony. Formally, we say that an implementation $\mathcal{I}$ for network updates in SDN provides *non-blocking progress* if in every execution $\mathcal{E}$ of $\mathcal{I}$, some network update $\mathcal{U}_i$ participating in $\mathcal{E}$ returns a matching response within a finite number of steps. The result and the proof itself is inspired by traditional distributed computing proofs that establish the impossibility of highly parallel non-blocking data structures [58].

Theorem 2.4.1. *No non-blocking* asynchronous *SDN implementation provides event linearizable network updates.*

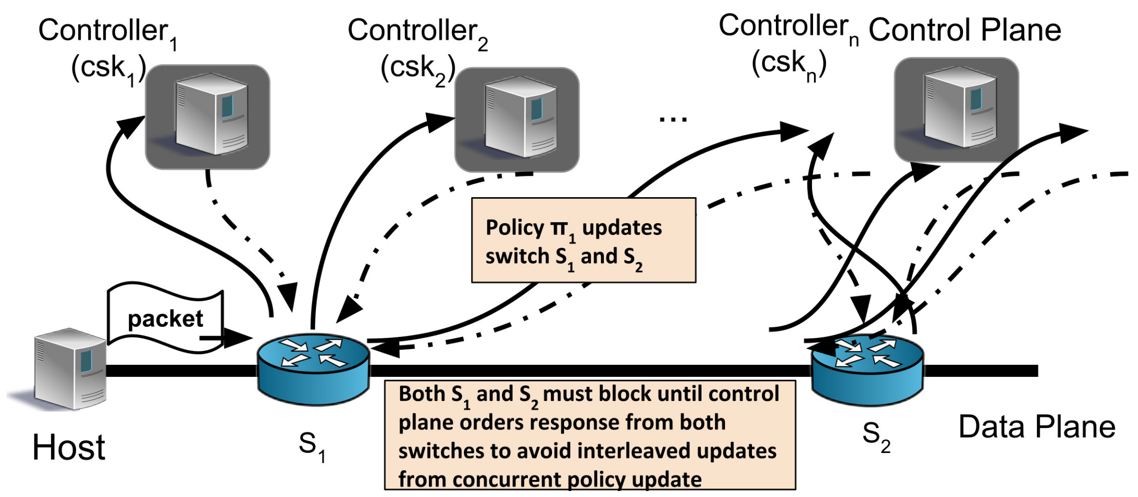

Controller$_1$ (csk$_1$)
Controller$_2$ (csk$_2$)
...
Controller$_n$ (csk$_n$)
Control Plane
Policy π_1 updates switch S_1 and S_2
packet
Host
S_1
Both S_1 and S_2 must block until control plane orders response from both switches to avoid interleaved updates from concurrent policy update
S_2
Data Plane

Figure 2.4. Typical execution of an event linearizable protocol like RoSCo: when two concurrent network policies π_1 and π_2 that are non-commutative over updates to the same set of switches s_1 and s_2, the protocol must *block* to prevent interleaving of the updates to the flow tables of s_1 and s_2. The detailed description of the RoSCo event linearizable protocol is described in Algorithm 1 and Algorithm 2 while the weak update consistency procedure is described in Algorithm 3.

Proof. Suppose by contradiction that there exists a SDN implementation $\mathcal{I}$ in the malicious controller threat model that provides event linearizable network updates and non-blocking progress. As illustrated in Figure 2.3, consider an execution $\mathcal{E}_i$ of $\mathcal{I}$ in which two network updates $\mathcal{U}_1$ and $\mathcal{U}_2$ participate in $\mathcal{E}_i$. We assume that $\mathcal{U}_1$ (and resp. $\mathcal{U}_2$) is invoked by event e_1 (and resp. e_2) by switch s_1 (and resp. s_2). Suppose that the response of e_1 (and resp. e_2) involves the flow table updates of switch s_1 first and then switch s_2 (and resp. s_2 first and then s_1). Since $\mathcal{I}$ is non-blocking, at least one of the events e_1 or e_2 must complete updating the flow tables of s_1 and s_2 and return a matching response within a finite number of steps. By our assumption, $\mathcal{E}_i$ is indistinguishable to s_1 (and resp. s_2) from an execution $\mathcal{E}_j$ in which $\mathsf{upd}[\mathcal{S}(e_1)] = \{s_1\}$ (and resp. $\mathsf{upd}[\mathcal{S}(e_2)] = \{s_2\}$). By the assumption of asynchrony, an adversary may simply delay the communication between the controllers and switches s_1 and s_2 such that the updates to the respective switches as part of events e_1 and e_2 are interleaved. However, this contradicts the assumption that $\mathcal{I}$ is event linearizable: either the updates corresponding to event e_1 first terminates and then the updates corresponding to event e_2 are performed or vice-versa. $\square$

Theorem 2.4.1 implies that providing event linearizable and asynchronous network updates requires relaxing the *non-blocking* progress property. In the next section, we describe a provably event linearizable protocol that is resilient against faulty controllers and ensures *blocking* progress assuming a majority of correct controllers. We additionally present a relaxation of the protocol satisfying update consistency providing demonstrably improved performance.

2.5 RoSCo Protocol

This section presents the core contribution of this chapter: RoSCo, a dependable Software Defined Networking (SDN) controller platform for robust network updates. The detailed description of the RoSCo protocol implementing event linearizability is given in Algorithm 1 and Algorithm 2. The weakly consistent update procedure is described with Algorithm 3.

2.5.1 Overview and notation

RoSCo ensures *flow consistency*, *switch consistency*, and progress assuming that a *majority* of controllers are correct. That is, to tolerate f failures, RoSCo requires that at least $3f + 1$ controllers participate in the execution. Since protocols involved in ordering events/event handling across controllers can be costly, RoSCo makes use of event *batching*.

The protocol makes use of the following notation: A=$[a_1, \ldots, a_n]$ denotes a *sequence* (ordered set) of elements $a_1 \ldots a_n$. A sequence's elements are identified/accessed via index (e.g., A[i] refers to the i-th element of A, $a_{\mathrm{i}} \in A$.). A=$\langle a_1, \ldots, a_n \rangle$ denotes a *tuple* with elements $a_1 \ldots a_n$. A tuple's elements are identified via dereferencing (e.g., $A.a_{\mathrm{i}}$). '$\oplus$' denotes sequence concatenation: $[a_1, \ldots, a_n] \oplus [b_1, \ldots, b_m] = [a_1, \ldots, a_n, b_1, \ldots, b_m]$. Lastly, cardinality is denoted by '$|\ldots|$': $|[a_1, \ldots, a_n]|$=n.

2.5.2 Definitions

In addition, the protocol and algorithms make use of the following definitions:

Event sequence $E = [\mathrm{e}_1, \ldots, \mathrm{e}_n]$: a sequence of one or more events.

Algorithm 1 Algorithm for switch s_{i} with secret key SSK_{i}

```
1:  $CPK_i$                                  ▷ Public key for each controller $c_i$
2:  $P \leftarrow \emptyset$                 ▷ Set of previous seen network updates from each controller
3:  $D \leftarrow \emptyset$                 ▷ Set of received network updates and counts
4:  $quorum \leftarrow \lfloor \frac{|C|-1}{3} \rfloor + 1$
5:  upon receive($packet$) on incoming link do
6:      if no flow table match then          ▷ Other anomalies possible
7:          sendEvent($packet$)
8:      else
9:          forward $packet$ along data plane
10: upon receive($seq \| r \| sig$) from controller $c_i$ do
11:     if verifySign($r, sig, CPK_i$) and $seq \notin P[c_i]$ then
12:         $D[r] \leftarrow D[r] \cup \{seq\}$
13:         if $|D[r]| \geq quorum$ then
14:             $P[c_i] \leftarrow P[c_i] \cup \{seq\}$
15:             apply($r$)
16:             sendAcknowledgement($seq$)
17: procedure sendEvent($packet$)
18:     e $\leftarrow$ generateEventData($packet$)
19:     $sig \leftarrow$ sign(e, $SSK_i$)
20:     send(e$\|sig$) to every controller $c_i$
21: procedure sendAcknowledgement($seq$)
22:     $sig \leftarrow$ sign(ACK$\|seq, SSK_i$)
23:     send(ACK$\|seq\|sig$) to every controller $c_i$
```

Switch update $u = \langle s, r \rangle$: an individual update to switch identified as s with rule r. The rule corresponds to the establishment of a new policy $\boldsymbol{\pi}$ on switch s.

Flow update $U = [u_1, \ldots, u_n]$: a sequence of switch updates such that u_{i} precedes $u_{\mathrm{i}+1}$.

Controller keys CPK_{i} *and* CSK_{i}: public and secret key respectively for controller c_{i}.

Switch keys SPK_{i} *and* SSK_{i}: public and secret key respectively for switch s_{i}.

Note that for simplicity we assume that any event e and rule r is implicitly uniquely identified.

2.5.3 Application interfaces

The protocol uses the following application interfaces:

Algorithm 2 Algorithm for controller c_i with secret key CSK_i

```
1: $SPK_i$                                    ▷ Public key for each switch $s_i$
2: $S \leftarrow \emptyset$                    ▷ Set of all previously seen events
3: $B \leftarrow []$                           ▷ Sequence of collected events for a batch
4: $acks \leftarrow [][]$                      ▷ Verified acknowledgements
5: $log \leftarrow []$                         ▷ Sequence of all network updates
6: $count \leftarrow 0$                        ▷ Count used for sequencing switch updates
7: upon receive(e$\|sig$) from switch $s_i$ do
8:     if verifySign(e, $sig, SPK_i$) and
9:           e $\notin S[s_i]$ then
10:        $B \leftarrow B \oplus$ [e]
11:    if $|B| >$ threshold or timeout then
12:        propose($B$)
13:        $B \leftarrow []$
14: upon decide($E$) do
15:     for each e $\in E$ do
16:         $S[s_i] \leftarrow S[s_i] \cup \{e\}$
17:     $\mathcal{U} \leftarrow$ handleEvents($E$)
18:     $log \leftarrow log \oplus \mathcal{U}$
19:     $count \leftarrow count + |\mathcal{U}|$
20:     update($\mathcal{U}$)
21: upon receive($ack$ = ACK$\|seq\|sig$) from switch $s_i$ do
22:     if verifySign($ack, SPK_i$) then
23:         $acks[s_i][seq] \leftarrow true$
24: procedure sendSwitchUpdate($seq, s, r$)
25:     $sig \leftarrow$ sign($seq\|r, CSK_i$)
26:     send($seq\|r\|sig$) to $s$
27: procedure update($\mathcal{U}$)            ▷ Performs linearized network update
28:     for i = 1..$|\mathcal{U}|$ in increasing order of i do
29:         $seq \leftarrow count - |\mathcal{U}| + i$
30:         for j = 1..$|\mathcal{U}[i]|$ in increasing order of j do
31:             sendSwitchUpdate($seq, \mathcal{U}[i][j].s, \mathcal{U}[i][j].r$)
32:             wait until $R_{ack}[\mathcal{U}[i][j].s][seq]$
```

Switch event generation: generateEventData(p) = e, given packet data p creates the necessary event data to be sent to the controller.

Switch update application: apply(r) applies r to the switch runtime.

Controller application invocation: handleEvents$([e_1, \ldots, e_l])$ returns $[U_1, \ldots, U_m]$: a sequence of flow updates generated in "response" to a sequence of events $[e_1, \ldots, e_l]$ as specified by the ideal world functionality.

Algorithm 3 Weak consistency update procedure. Replaces lines 27–32 of Algorithm 2

```
27: procedure update($\mathcal{U}$)
28:     $seq[i]_{i=1..|\mathcal{U}|} \leftarrow count - |\mathcal{U}| + i$
29:     fork for i = 1..$|\mathcal{U}|$ do                                  ▷ in parallel
30:         for j = 1..$|\mathcal{U}[i]|$ in increasing order of j do
31:             wait until                                              ▷ both, in any order
32:                                                                     ▷ flow consistency
33:                 1. $R_{ack}[\mathcal{U}[i][j-1].s][seq[i]]$ unless j = 1
34:                                                                     ▷ switch consistency
35:                 2. $R_{ack}[\mathcal{U}[i][j].s][last]$ for largest
36:                    $last < seq[i] \mid \exists\langle s, \ldots\rangle \in all[last]$ if any
37:             sendSwitchUpdate($seq[i], \mathcal{U}[i][j].s, \mathcal{U}[i][j].r$)
```

Signature creation: sign(msg, sk), a function to sign a message (msg) with given key (sk).

Signature verification: verifySign(sig, msg, pk), a function to verify a signature (sig) for the given message (msg) using the public key (pk).

Agreement: propose$([e_1, \ldots, e_n])$ is used by controllers to initiate/participate in an agreement protocol on a sequence of events $E = [e_1, \ldots, e_n]$. As the outcome of agreement, controllers receive through a callback decide(E) the sequence of events E to pass to the controller application.

2.5.4 RoSCo protocol procedure

The RoSCo protocol uses an agreement protocol to ensure a total ordering of events processed by controller nodes as well as quorum authentication to ensure agreement among rules to be applied to switches. In short the RoSCo protocol works as follows:

(i) A switch receives an incoming packet for which there is no matching rule in the flow table.

(ii) The switch generates an event, assigns the event a unique sequence number, signs the event with its private key, and broadcasts the signed event to all controllers (Algorithm 1 line 17).

(iii) A controller, upon receiving a signed event from a switch, verifies the signature (ignoring the event if verification fails or if the event has been received previously), and adds the event to the current batch (Algorithm 2 lines 7- 10).

(iv) Once either the batch reaches capacity or a timeout occurs, the controller proposes the batch across controllers (Algorithm 2 line 12).

(v) Once agreement has been reached, the controller passes the event batch to the controller application to determine the network update for it (Algorithm 2 lines 14- 20). Note that agreement decisions are delivered in a sequential manner, i.e., decide executes in mutual exclusion and in order of instances of the agreement protocol.

(vi) The controller executes the network update procedure (described below) for the resulting network update (Algorithm 2 line 20).

(vii) Event batch and network update are cleared after all switch updates are applied to the data plane.

2.5.5 Strongly consistent network update procedure (event linearizability)

To ensure strong consistency, when given a network update, the controller performs each switch update serially. The procedure to perform the network update is as follows:

(i) The controller retrieves the next flow update from the network update (Algorithm 2 line 28).

(ii) The controller retrieves the next switch update from the flow update, signs the update with its private key, and sends the signed update to the switch (Algorithm 2 line 31).

(iii) The controller waits to receive an acknowledgement the switch (Algorithm 2 line 32).

(iv) The controller continues with the next switch update (step (ii)) until all policies from the flow update have been made.

(v) The controller continues with the next flow update (step (i)) until all flow updates have been applied.

2.5.6 Switch update procedure

To ensure protection from a faulty controller, a switch can only apply an update rule after it has received verified update rules from a quorum majority of controllers. The procedure taken by a switch to perform a rule update is as follows:

(i) A switch, upon receiving a switch update from a controller verifies the signature (ignoring the update if verification fails or if the rule has been received previously), and adds the rule to the received rule set (Algorithm 1 lines 10-12).

(ii) If a quorum of verified rules has been received the switch applies the rule, and sends an acknowledgement for the rule to all controllers (Algorithm 1 lines 13-16).

The proof technique for event linearizability is interesting is its own right: it shows how ideas from traditional distributed computing proofs [59] can be adopted for deriving correctness proofs of network update protocols. Recall that the application of a network policy π_i begins with an event invocation by a switch $s_i \in \mathcal{S}$ followed by a *network update*. To prove event linearizability of RoSCo, we need to show that in every execution of the RoSCo protocol, described in Algorithm 1 and Algorithm 2, and for any two network policies π_i and π_j in a given execution, π_i (and resp. π_j) completes entirely before π_j (and resp. π_i) starts. The proof for event linearizability shows that the individual steps of π_i and π_j are not interleaved by constructing a mapping to an equivalent execution in which network policies are never invoked concurrently. This is achieved by assigning appropriate *atomicity points* for the update procedures.

Theorem 2.5.1. *RoSCo protocol described in Algorithm 1 and Algorithm 2 enforces event linearization of network updates.*

Proof. In RoSCo, an event is initiated by a switch and results in a network update to apply a network policy to the flow tables of some subset of switches as illustrated in Figure 2.4. The application of a network policy π_i in an execution $\mathcal{E}$ of RoSCo begins with an *event invocation* by a switch $s_i \in \mathcal{S}$ followed by a *network update*.

Let Line 5 of Algorithm 1 be the event invocation of any network policy π_i in an execution $\mathcal{E}$ and Line 24 in Algorithm 2 denote the response r_i of π_i in $\mathcal{E}$. All steps performed by the state machines described by the pseudocode within these lines denote the *lifetime* of π_i. The proof proceeds by assigning a *serialization point* for a policy which identifies the step in the execution in which the policy *takes effect*. First, we obtain a completion of $\mathcal{E}$ by removing every *incomplete policy* from $\mathcal{E}$. Concretely, if the step in line 15 of Algorithm 1 is not performed, the policy is treated as incomplete.

Let H denote the high-level history of $\mathcal{E}$ constructed as follows: firstly, we derive *linearization points* of operations performed in $\mathcal{E}$ (sendAcknowledgement, sendEvent, verifySign, sendSwitchUpdate, apply and propose). The linearization point of any such operation *op* is associated with a message step performed between the lifetime of *op*. A linearization H of $\mathcal{E}$ is obtained by associating the last event performed within *op* as the linearization point. We then derive H as the subsequence of $\mathcal{E}$ consisting of the network policy event invocations, network updates and network policy responses. Let $<_{\mathcal{E}}$ denote a total order on steps performed in $\mathcal{E}$ and $<_H$ denotes a total order steps in the complete history H. We then define the *serialization point* of a policy π_i; this is associated with an execution step or the linearization point of an operation performed within the execution of π_i. Specifically, a complete sequential history S is obtained by associating serialization points to policies in H as follows: for every complete network update in $\mathcal{E}$, serialization point is assigned to the last event of the loop in line 32 of Algorithm 2.

Claim 1. *For any two policies π_i and π_j in $\mathcal{E}$, if $\pi_i \prec_H \pi_j$, then $\pi_i <_S \pi_j$.*

Proof. This follows immediately from the fact that for a given network update, its serialization point is chosen between the first and last event of the policy implying if $\pi_i \prec_H \pi_j$, then $\delta_{\pi_i} <_{\mathcal{E}} \delta_{\pi_j}$ implies $\pi_i <_S \pi_j$. □

Claim 2. *Let $[\pi_1, \ldots, \pi_n]$ be the ordering for the sequential specification of policies in H. Then, the sequence of network policies as constructed in S is consistent with $[\pi_1, \ldots, \pi_n]$.*

Proof. Let $[U_1, \ldots, U_n]$ be the corresponding sequence of flow updates where for all $i \in \{1, \ldots, n\}$, U_i is the flow update for π_i. Recall that each flow update consists of $[u_1, \ldots, u_n]$: a sequence of switch updates such that u_i precedes u_{i+1}. Firstly, we argue that for all $i < j \leq n$, $\delta_{\pi_i} <_{\mathcal{E}} \delta_{\pi_j}$ implies that the last switch update of U_i (as defined by the linearization point of apply in line 32 of Algorithm 2) precedes the first switch update of U_j. This invariant immediately follows the waiting acknowledgement loop in line 32 of Algorithm 2 which forces all switches in U_i to complete before starting the first switch update in U_{i+1}. Thus, if π_i precedes π_j according to the sequential specification in H, then $\delta_{\pi_i} <_{\mathcal{E}} \delta_{\pi_j}$ implies that π_i precedes π_j in S.

To complete the proof of this claim, we argue that if π_i precedes π_k according to the sequential specification in H, there does not exist $i < j < k$ such that $\pi_i <_S \pi_j <_S \pi_k$. Suppose by contradiction that such a π_j exists. Recall that by the agreement property, every controller node agrees on the output of the sequence of flow updates in line 17 of Algorithm 2. Consequently, the only reason for such a π_j to exist is if the last switch update of U_j precedes the first switch update of U_k. But this is not possible by the invariant proved above—contradiction. □

The conjunction of Claim 1 and Claim 2 together establish that $\mathcal{E}$ is event linearizable.

□

2.5.7 Weakly consistent network update procedure (update consistency)

For weak consistency, when given a network update, the controller can perform the switch updates commutatively provided that flow consistency and switch consistency are maintained. Commutativity is determined based on the network updates received by the controller application. Flow consistency is maintained by ensuring that a switch update within a flow update is not sent to the switch until after receiving an acknowledgment for the previous switch update. Switch consistency is maintained by ensuring that a switch update is not sent to the switch until after an acknowledgment is received for the last switch update to the same switch as determined by the controller application (if any). The procedure to perform the network update is as follows (cf. Algorithm 3): the controller forks a parallel thread for every flow update in the network update (Algorithm 3 line 29).

(i) The controller retrieves the next switch update from the flow update and waits until the following have been satisfied (Algorithm 3 lines 31-36):

    (a) the switch update is the first in the flow update or an acknowledgement for the preceding switch update has been received, and
    (b) an acknowledgement for the last switch update to the same switch from previous network updates has been received.

(ii) The controller signs the update with its private key and sends it to the switch (Algorithm 3 line 37).

(iii) The controller continues with the next switch update (step (i)) until all switch updates have been made.

2.5.8 Discussion on progress with dishonest controller majority

Observe that RoSCo is a *blocking* implementation that provides forward progress as long as a correct majority of controllers participate in a given execution, but safety (i.e., event linearizability and update consistency) is never violated even if a faulty majority exists. Note that even if some *controller manufactures a response to an event* and even if the controller is able to correctly sign the response, if no other controllers send the same response then a quorum of responses will never be received by the switch and the rule associated with the response will never be applied (line 13 of Algorithm 1).

This is the main algorithmic trick in RoSCo: to augment the switch runtime to perform the controller event verification, but with minimal overhead as we demonstrate through our extensive evaluations. We remark that the presented pseudocode does not explicitly depict the RoSCo exception handling (when a dishonest majority affects any step in the protocol execution thus disrupting forward progress) which in practice can be handled via timeouts, optimistic abort-and-retry mechanisms, and/or existing techniques for detecting and defending against coordinated intrusion [56], [57].

2.6 RoSCo Implementation

RoSCo is implemented as an abstraction layer between the controller application and the data plane switches utilizing a totally distributed controller framework where all controllers operate in the same role. In contrast to explicit leader-based network update protocol described in Ravana [5] for dealing with benign faults, events are sent from switches to all controllers avoiding the need for leader election.

2.6.1 Overview

The RoSCo implementation is twofold;

(i) A Java layer (runtime OpenJDK version 1.8.0) which runs on the controller to perform agreement and handle acknowledgements. Using interprocess communication, events are sent from the RoSCo layer to the controller application providing the flexibility of the use of any controller application framework to run unmodified.

(ii) An Open vSwitch (OVS) runtime extension for handling quorum authentication of policy responses.

Figure 2.5 depicts the software organization of the RoSCo implementation. Implicitly, by extending OVS, the RoSCo implementation makes use of the OpenFlow protocol. The sections that follow describe how RoSCo makes use of existing OpenFlow messages and structures. To ensure message integrity and authentication OpenFlow messages are wrapped with a signature for the sender as an extension to the existing OpenFlow message format. In addition, the RoSCo implementation utilizes OpenFlow as follows:

- When a controller or switch sends a message, the transaction identifier (XID) in the OpenFlow message header is set to a unique value. This allows recipients of the messages to drop duplicates.

- A Barrier Request (BarrierReq) is sent by the controller to initiate an acknowledgement, it then waits for a Barrier Response (BarrierRes) from the switch before proceeding.

2.6.2 Controller applications

The controller application contains the logic for network policies and processes events to generate network updates. The RoSCo runtime intercepts all incoming event messages and network updates allowing unmodified controller applications to run in a fault-tolerant manner. RoSCo adds message signatures to the OpenFlow protocol messages, but leaves the message payload for policies unchanged.

2.6.3 Agreement

In order to maintain consistent controller state, events sent from switches must be processed by each controller in the same order, however due to network asynchrony there is no guarantee that events will be received in the same order by controllers. Duplicate and corrupted events must also be discarded. In order to ensure a total ordering of events across controller nodes, the RoSCo controller layer uses the BFT-SMaRt library to agree on an event (set of events with batching). Once a sequence of events is agreed on, each controller processes them from its own event queue and sends the corresponding response(s) directly to the data plane.

2.6.4 Event sequence numbers

Events sent from data plane switches need to be identifiable to ensure ordering and unique event execution. Any event that has been previously processed by a controller node is discarded. To implement sequence numbers, we modified the OVS runtime to use the XID contained in the OpenFlow header. An event received by a controller with the same XID as one received previously is discarded. Similarly for event responses sent by controllers, the XID is also used to assign each response a unique identifier and a response received by a switch with a previously used XID is discarded.

2.6.5 Quorum authentication

Quorum authentication ensures that a policy cannot be set by a minority of malicious controllers, assuming a control plane of four or more controllers. Quorum authentication therefore complements SSL, provided in OVS, that secures communication but does not protect against a trusted controller that becomes compromised by a malicious adversary. Switches must wait for a majority quorum of responses received from verifiable controllers before the rules can be installed. The verification of controllers and the storing of responses is implemented as a modification to the OVS runtime. Switches store verified responses from controllers in a hash map keyed by XID along with a list of the controllers that sent the particular response. The response is passed to the switch runtime for processing only when a majority quorum has been received. At that time the response XID is "retired" and any future responses containing the XID are ignored.

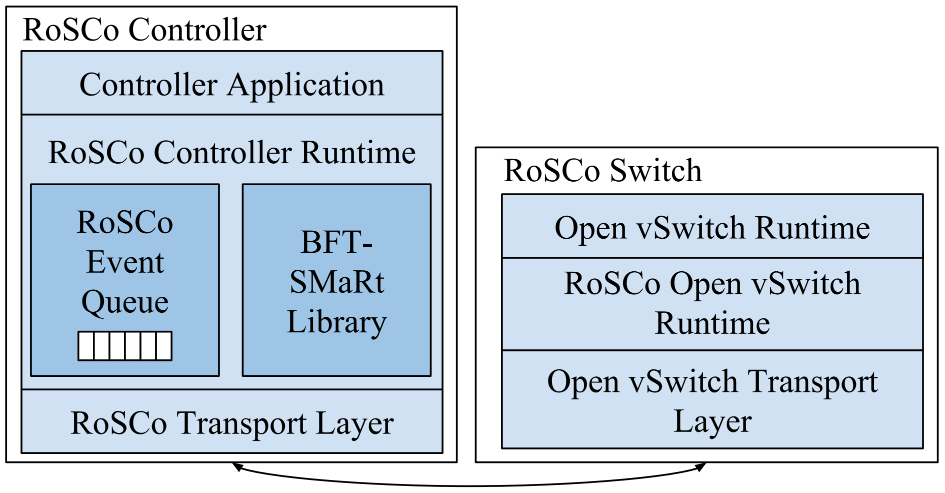


Figure 2.5. Depiction of RoSCo implementation. As OpenFlow events are generated by the switch they are intercepted by the RoSCo OVS runtime, assigned a sequence number and sent to the controller. OpenFlow events are received by the RoSCo runtime and sent to the RoSCo event queue which uses BFT-SMaRt [60] to determine a total order across all controller nodes. Ordered events are sent to the controller application and responses are passed through the RoSCo runtime to the network and sent to the switch. Update rules are held by the RoSCo OVS runtime until a quorum majority have been received from verifiable controllers, which are then sent to the OVS runtime.

2.6.6 Acknowledgments

To ensure consistency, switches must send acknowledgments to the control plane once a rule has been installed. To that end we use OpenFlow Barrier Messages. For each event, after all responses have been sent to relevant switches, the RoSCo layer sends a BarrierReq to each switch and waits for a BarrierRes. Once a quorum of responses is received a switch sends a BarrierRes to each controller.

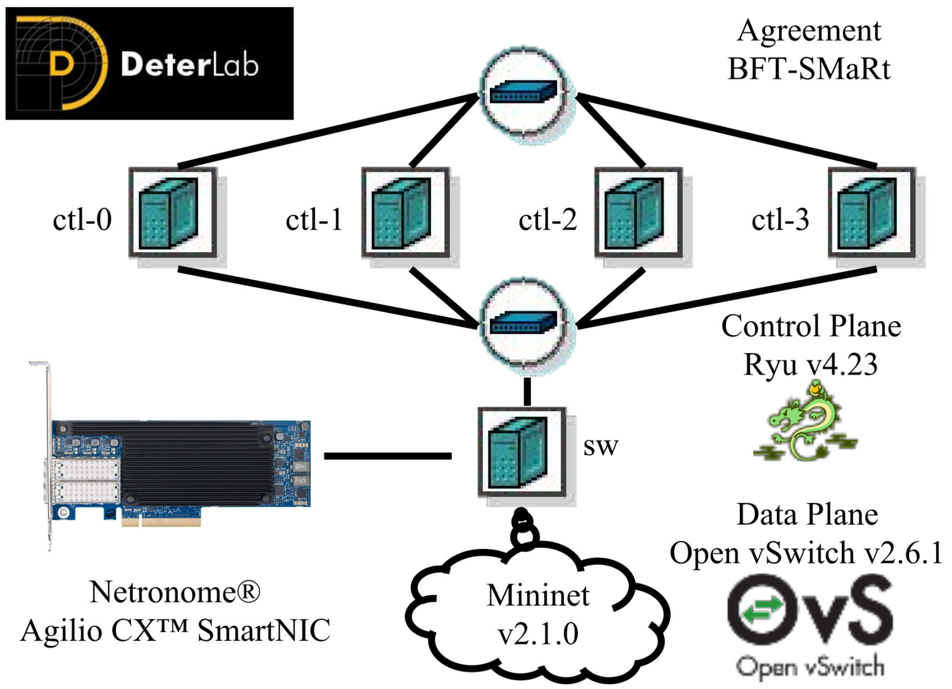


Figure 2.6. Depiction of RoSCo evaluation testbed. Control plane consists of four separate machines from the DETERLab running the RoSCo layer and Ryu. Switch emulation is performed by a separate machine equipped with a Netronome® Agilio CX™ SmartNIC using Mininet. Throughput benchmarking is done using the modified OVS `ovs-appctl` command to generate continuous PacketIn requests recording the ultimate throughput of received FlowMod responses.

2.6.7 Event batching

The time it takes for the agreement protocol to execute can be significant compared to the time for a controller to process the event. In response to this, we have implemented event batching to allow a controller to atomically propose a set of events received from the data plane. BFT-SMaRt ensures that only a single instance of the agreement protocol is performed at a time. Albeit events may be batched in a different order by different controllers, this single agreement behavior, combined with unique event identifiers, ensures that events are processed in the same order by all controllers. Event batching imposes added latency as the controllers must wait for either the batch to fill with events or until a certain amount of time has passed (batch timeout). In many cases, this added latency can be amortized based on the increased throughput. However, for certain events, it may be necessary to ensure that the latency is as short as possible. The RoSCo batching implementation allows the batch size and timeout values to be globally set based on the network time objectives.

2.7 RoSCo Evaluation

In this section, we rigorously evaluate the overhead of our event linearizable and update consistent protocols (cf. § 2.5)
using varying network configurations. The goal is to determine the feasibility of deploying RoSCo in a large-scale SDN by answering the following questions:

(1) What is the effect of event linearizability on throughput?

(2) Does update consistency improve throughput?

(3) How much can a faulty controller affect throughput and how does it compare against protocols that only provide resilience against benign crash failures?

(4) Does RoSCo affect the latency of processing events?

2.7.1 Evaluation hardware

All evaluations were performed on a set of machines provided by the DETERLab Project [61] (cf Figure 2.6). For RoSCo controllers, we used 4 machines (to tolerate 1 failure) each containing 2 Intel® Xeon™ processors running at 3.00 GHz with 4 GB of main memory. For switch emulation, we used a single machine containing two Intel® Xeon™ E5-2450 v2 processors (12 cores total) running at 2.20 GHz with 16 GB of main memory. All machines ran Ubuntu 16.04 LTS with kernel v4.4.0-83 and were connected using a 1 Gbit network. A separate 1 Gbit network was used to connect controller machines dedicated for the agreement protocol. Additionally, the switch machine also contained a Netronome® Agilio CX™ SmartNIC which provides hardware offloading of OpenFlow packet processing through the use of a custom OVS implementation. This implementation is open source which allowed for the modifications necessary for our evaluation. Data plane connectivity was emulated using Mininet v2.1.0.

2.7.2 Evaluation testbed

Throughput was measured using an extension to the OVS ovs-appctl command to create a benchmark that accesses the OVS switch runtime and schedules the sending of PacketIn requests from all switches. The benchmark then waits for an equal number of FlowMod responses from the controller(s), and records the total time to receive the responses and the average response time for each individual PacketIn request. In the control plane, the benchmark uses the cbench application, a Ryu controller application implemented as a benchmark for measuring throughput of OpenFlow messages. Upon receiving a PacketIn message, cbench replies with an empty FlowMod message which does not provide any routing logic to the switch. As such, cbench is not intended for use as deployed controller application and is solely intended for benchmarking to provide minimal latency within the controller application so that measurements actually reflect the throughput of the controller runtime and transport layer. While our benchmark focuses on PacketIn requests and FlowMod replies, any OpenFlow event and response can be used without loss of generality.

Using this benchmark we measured the throughput of RoSCo and compared it to two other implementations: a centralized controller using the Ryu [18] runtime (version 4.23) and Ravana. The authors of Ravana were able to provide their implementation which we used in our evaluations. The data points for all graphs represent the geometric mean of 25 individual trials, with error bars representing one standard deviation from the geometric mean.

2.7.3 Benefits of event batching

The RoSCo controller batches events to increase throughput. Here we evaluate the throughput and latency of RoSCo for varying batch sizes. Figure 2.7a shows the message throughput of events processed through the agreement protocol by RoSCo for various batch sizes and Figure 2.7b shows the corresponding latency imposed. In our experiments, throughput increases as batch size increases, which is the direct result of the reduced number of agreement protocol instances needed for controllers to agree on events. However at a point, the benefits of batching are reduced as performance becomes bounded by other aspects of event processing (e.g., decoding of OpenFlow messages and processing of events).

2.7.4 Throughput and latency

We evaluated the throughput of RoSCo against Ryu and Ravana. Using the OVS benchmark, experiments were run for each implementation with varying numbers of switches in the data plane, which all sent PacketIn requests in parallel to the controller(s). The resulting throughput can be seen in Figure 2.7c. For both RoSCo and Ravana, the maximum batch size was set to 1000 messages with a batch time out of 0.1 s. The RoSCo network update procedures evaluated are as follows:

RoSCo NO-ACK represents RoSCo without acknowledgments. This scenario is impractical as it provides no consistency guarantees for network updates, yet is included as a baseline.

RoSCo LNZ represents RoSCo using the strongly consistent network update procedure. As such, all switch updates are performed sequentially.

RoSCo WEAK-SGL represents RoSCo using the weakly consistent network update procedure with single switch updates. In this scenario the flow updates received in response to an event require updating a single switch.

RoSCo WEAK-ALL represents RoSCo using the weakly consistent network update procedure, however each flow update received in response to an event requires an update to all switches.

RoSCo's throughput is lower than Ryu's due to the overheads of agreement and quorum authentication. The OVS benchmark sends requests without delay representing the worst possible case for RoSCo performance. Our goal is to provide strong guarantees, not maximize update rate/throughput. That said, our scheme provides good performance even in conservative scenarios; it is clearly faster with update consistency than Ravana — whose performance is deemed sufficient for real-life scenarios [5] — in all scenarios with 4 or more switches.

Ryu utilizes multiple threads in order to maintain separate connections to each data plane switch. The Ravana implementation uses additional threading to maintain its distributed event queue. Written as a direct extension to the Ryu runtime and entirely in Python, Ravana suffers significant performance penalties due to the threading behavior of the Python interpreter's global interpreter lock (GIL) [62], which while allowing for a more deterministic execution, prevents threads from executing in parallel. To show how the GIL affects event processing throughput we also implemented RoSCo completely in Python as an extension to the Ryu runtime. The resulting throughput is shown as 'RoSCo Python' in Figure 2.7c and is significantly lower than the Java layer approach. This motivated the use of a framework providing true parallelism for our RoSCo implementation since our goal is to evaluate the overhead of agreement, quorum authentication, and acknowledgements, not the limitations of Python.

2.7.5 Impact of consistency guarantees

Without any acknowledgments, RoSCo is able to achieve throughput near Ryu since all switch updates can be processed in parallel. However, this provides no consistency in network updates. The addition of acknowledgments provides the guarantee of update consistency, however reduces throughput. This result is intuitive as the controllers must wait for an acknowledgments from data plane switches before sending dependent switch updates.

For the strongly consistent network update procedure (RoSCo LNZ) all updates are processed sequentially. While providing the strongest guarantees, it also allows no update commutativity. Relaxing the consistency requirements (RoSCo WEAK-SGL and RoSCo WEAK-ALL) provide significant performance improvements especially as the number of switches increases. In both scenarios, increasing the number of switches increases commutativity of flow updates. Even when a flow update requires a switch update to each switch in the data plane (RoSCo WEAK-ALL), such flow updates can be pipelined as updates are made to individual switches.

In all experiments, more switches requires additional system resources for managing acknowledgements, queued events, and tracking rule quorums. This additional resource

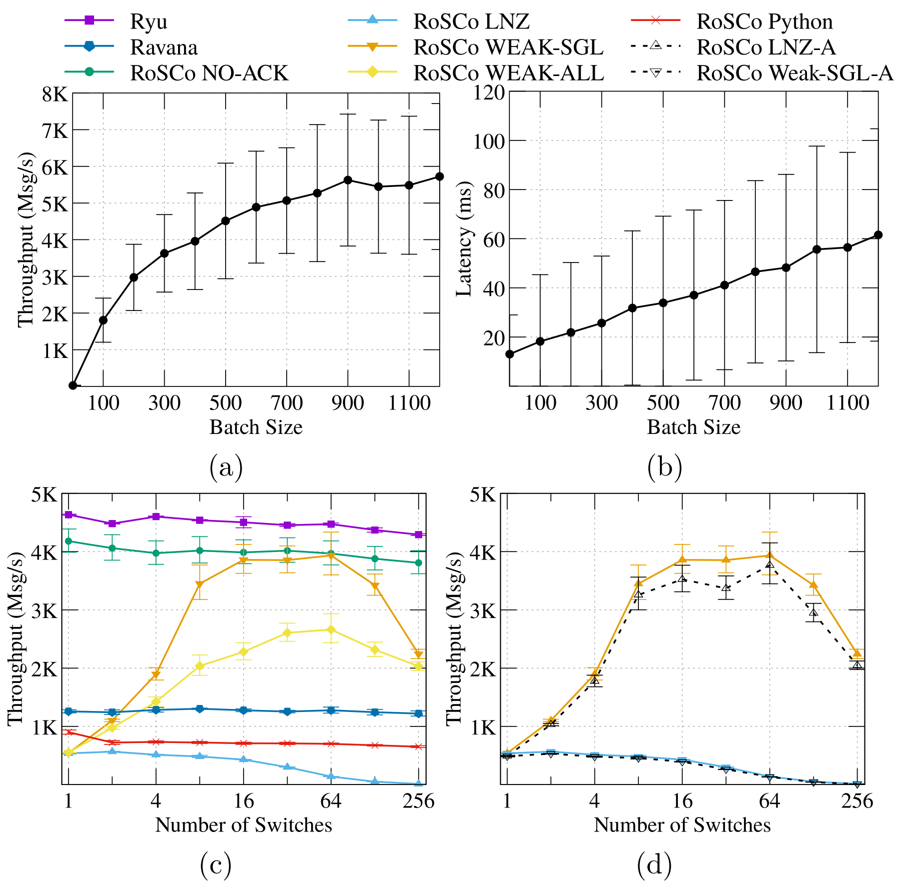


Figure 2.7. Figure 2.7a and Figure 2.7b depict the throughput and latency of RoSCo with varying batch sizes. Figure 2.7c depict the total throughput of RoSCo, Ryu, Ravana, and the Python implementation of RoSCo for varying network sizes and consistency guarantees. Figure 2.7d depicts the throughput of RoSCo when the control plane include a faulty controller. Error bars represent one standard deviation from the geometric mean. Figure 2.7c and Figure 2.7d reflect that at extreme cases for switch counts the overhead for managing acknowledgements, queued events, signature verification, and quorum authentication reduces throughput, but all within acceptable levels compared to the state of the art.

usage reduces the overall throughput in the extreme cases for switch counts. This behavior is also reflected in the centralized Ryu and Ravana experiments.

2.7.6 Impact of faulty controllers

We ran RoSCo using the OVS benchmark under the following two fault scenarios:

(i) A single faulty controller continuously proposes the first event received from the data plane. Since the event is a duplicate, it is continuously dropped, however all controllers must still participate in agreement before the event is discarded.

(ii) A single faulty controller continuously sends a FlowMod to all switches with differing response identifier. The switch determines that the signature as well as the response identifier are correct, however since no other controller in the control plane sends the same update, a quorum is never reached and the update is never applied.

We ran these scenarios through the same network configurations as our throughput evaluation. Figure 2.7d shows the results of RoSCo while under attack scenario (i) (RoSCo WEAK-SGL-A) and RoSCo WEAK-ALL-A in the figure). While the faulty controller is able to affect the overall throughput of events from the increased number of agreement instances, the reduction is only marginal. For scenario (ii) there was no significant difference in throughput. These results show that RoSCo can perform equally well with controller faults as during normal operations.

Table 2.3. RoSCo microbenchmark results

Measurement	Time (ms)
Response Time (RT)	11.42
Acknowledgement Time (AT)	5.50
Quorum Time (QT)	0.28
Agreement Time (AGT)	9.78
Processing Time (PT)	1.36
RoSCo Overhead (RT + AT - PT)	15.56

2.7.7 Microbenchmarks

Table 2.3 shows the average time spent in various places of the RoSCo implementation while processing a single event. *Response Time (RT)* represents the total time to process the event, from the time it is sent by the data plane switch to the time in which the switch processes a quorum response. *Acknowledgement Time (AT)* represents the time for a controller to receive an acknowledgement for a switch update. In the case of dependent switch updates, the controller would not be able to send the next switch update until receiving an acknowledgement. *Quorum Time (QT)* represents the time that a data plane switch waits for a quorum of messages from the control plane (after receiving the first switch update). *Agreement Time (AGT)* represents the time for a single instance of agreement using BFT-SMaRt. *Processing Time (PT)* represents the time spent for the controller application to process the event and send a response. The overall overhead of quorum authentication is minimal and the majority of RoSCo overhead is from agreement and waiting for acknowledgments. The actual overhead for a single request is shown in Table 2.3.

Assuming negligible network delay, the overhead for RoSCo is essentially the total response time plus acknowledgement time minus the processing time. Processing of an event by the controller application to generate a network update is required regardless of protocol. Using the weakly consistent network update procedure, the acknowledgement time can be amortized as independent switch updates can be processed in parallel, however since the values shown in Table 2.3 represent the time to process a single event, this amortization is not reflected. The total overhead can be further reduced by batching events to amortize agreement costs.

2.8 Conclusions

This chapter presents a model of computation for SDN in the presence of faulty control processes as well as a protocol and prototype implementation that shows that only a minimal overhead is induced over existing SDN controller platforms that do not provide resilience against malicious behavior.

While RoSCo can provide strong guarantees on network consistency in an environment with faulty controllers, it is not without limitations. The replicated membership groups of controllers is static and public keys must be pre-distributed to all controllers and switches in the network. A fault tolerant control platform would benefit from the addition of dynamic membership as well as a secure key distribution mechanism. Furthermore, additional evaluation on a set of data-center characteristic workloads would further strengthen the position of the minimal overhead the RoSCo imposes on data plane flows. The following chapter explores these aspects of a fault tolerant control plane presenting a practical solution which includes dynamic membership and pro-active security.

3. PRACTICALITY AND CONSISTENCY FOR NETWORK POLICY UPDATES

This chapter expands on fault tolerance for SDN network updates to the larger distributed SD-WAN network and describes ConsIstent seCurE pRactical cOntroller (Cicero) [63], a construction that focuses on security and consistency of network updates while being practical for deployment in production networks. To be practical, network updates must be both consistent, i.e., free of transient errors caused by updates to multiple switches, and secure, i.e., free of errors caused by faulty or malicious members of the control plane. Besides, these properties must incur minimal overhead to controllers and switches.

3.1 Introduction

The advent of software-defined wide area networking (SD-WAN) has brought the *concurrent network update* problem [64] to the forefront. In short, the challenge is to construct a control plane for SD-WAN capable of covering large geographically separated networks. Building a single consolidated control plane across WANs agnostic of the different underlying *domains* (e.g. constituting autonomous systems or based on some *locality* in the physical topology) can optimize the processing of consistent updates [41], [65]–[67]. Yet, it is likely ineffective and hardly scalable in practice, besides requiring strong trust between the domains. Inversely, managing domains independently, each with a separate control plane, can help perform updates efficiently in parallel (e.g. when updates only affect single domains), and can ensure that failures (e.g. misconfigurations, crashes, malicious tampering) in one domain do not affect others. However, this does not provide support for updates affecting multiple domains in a consistent manner.

3.1.1 Requirements

A viable SD-WAN control plane should reconcile the following three conflicting requirements:

Consistency: Concurrent updates — whether affecting individual domains (intra-domain routes) or multiple domains (inter-domain routes) — should meet the *sequential specification* of the shared network application, i.e., they should not create inconsistencies leading to network loops, link congestion, or packet drops.

Security: The control plane should be able to perform updates in the face of high rates of failures including benign failures (e.g., crashes) as well as malicious failures (e.g., tampering); in particular failures should not spread from one domain to another.

Practicality: Performance should allow for real-life deployments that scale to as many domains and switches as possible while sustaining high update rates, and should impose minimal overhead on switches.

3.1.2 State of the art

Making the control plane tolerate failures has been tackled by several approaches, yet these approaches either solely handle crash failures [5], [27], [28], or handle potentially malicious behaviors [68], [69] but with no control plane authentication for the data plane, thus not fully shielding the data plane against masquerading malicious controllers. In addition, most of these approaches consider only single-domain setups.

Protocols for Byzantine fault tolerance (BFT) [31], a failure model subsuming crash failures, provide safety and liveness guarantees [30], [60] up to a given threshold of faulty participants, most often growing linearly with regards to the number of participants. Most work here similarly considers single domain setups, putting little emphasis on handling failures to quickly yet permanently retain trustworthiness and support cooperation across domains throughout successive failures. Yet while application-specific solutions exist for performance-aware routing [70] or optimal scheduling for network updates [44], we are not aware of any practical system providing a generic protocol to securely enforce arbitrary application network updates across a faulty and asynchronous distributed network environment. Crucially, from the point of view of practical adoption, existing work introducing distributed resiliency techniques to address the network update problem treat both switches and controllers as equal participants in the protocol, thus inducing prohibitive overhead on the switching fabric [26], [69].

Table 3.1. Examples of network changes with their desired behaviors, potential problems, and consistency preconditions.

Example	**Figure 3.1**	**Figure 2.2**	**Figure 3.3**
Network change	Firewall rule changes	Network hardware maintenance	Bandwidth load-balancing
Desired behavior	Policy enforcement	Loop/black-hole freedom	Loop/black-hole freedom Congestion freedom
Potential problems	Compromise or loss of data	Packet loss	Over-provisioning of link resources
Update consistency preconditions	Awareness of existing firewall rules	Awareness of existing flows	Awareness of existing bandwidth usage

3.2 Background

From a high level, network traffic is shaped by policies set by network administrators. Based on an unbounded number of motivating factors (e.g., demand for network resources, application bandwidth requirements, firewall rules, other network tenant requirements), it is impossible to be 100% certain what drives network policies. For a network switch in a data plane, policies are represented by forwarding rules that describe the store and forward behavior of network packets. An individual switch has no understanding of a policy or how it affects the entire network. In an SD-WAN environment, a control plane of one or more controllers enforces policies set by the network administrator by translating policies into flow table entries installed on switches. As network traffic arrives or as network policies change, updates to switch flow tables are needed through network updates. Furthermore, the topology of the network may be dynamic as physical cabling is changed and/or failures happen in switch or fabric hardware. These topology changes may also result in network updates.

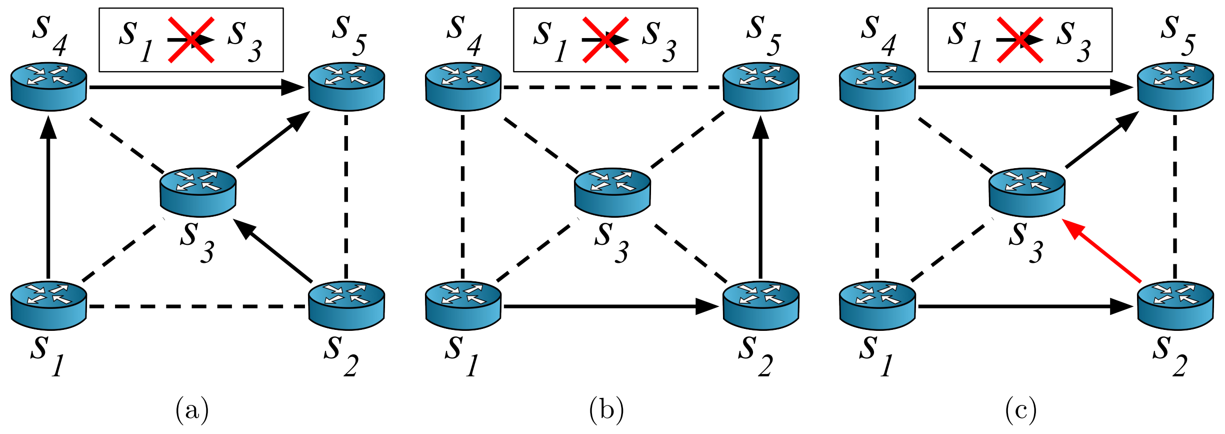


Figure 3.1. **(a)** The state of the flows from s_1 and s_2 to s_5 **(b)** is intended to be modified by an update which respects the firewall rule, **(c)** but s_1 applies the update before s_2 which *breaks the firewall rule*.

3.2.1 Definitions

A *network flow* is an active transfer of packets in the data plane identified by its source, target, and bandwidth requirements. A *route* indicates the specific path that a network flow takes within the network; multiple possible routes may exist for a network flow. Forwarding rules instruct a data plane switch how to forward received packets in a flow. The data plane state consists of all forwarding rules currently in use by all data plane switches. The control plane is thus responsible for maintaining forwarding rules in the data plane state for all routes such that they comply with network policies at all times, even during a change to the data plane state.

3.2.2 Challenges

In this section we outline several motivating examples that show not only the need for consistent network updates performed in a secure manner, but also the need for practicality for policy specification and scalability for deployment in large networks facing a myriad of concurrent network updates.

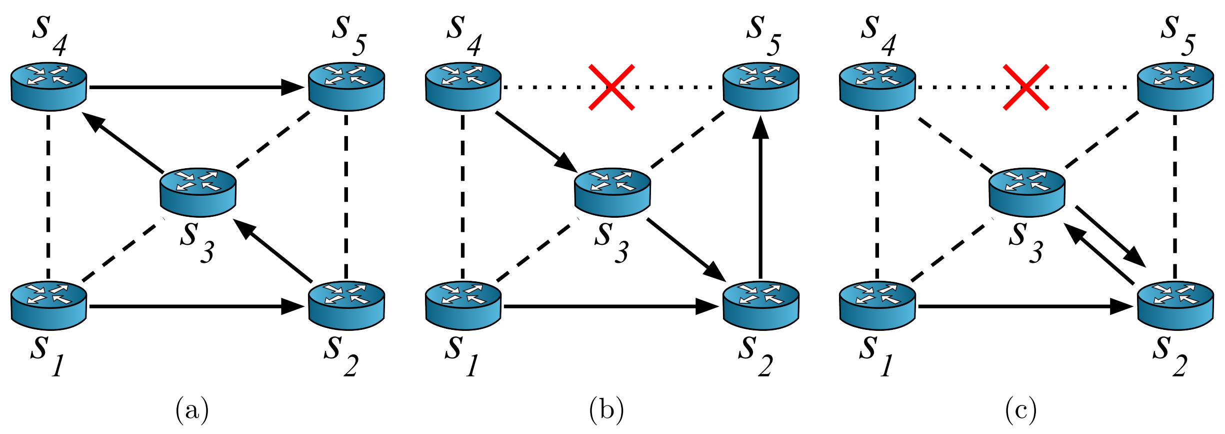


Figure 3.2. **(a)** The state of the flows to s_5 **(b)** is planned to be modified by an update to bypass the failure of the s_4-s_5 link but **(c)** s_3 applies the update before s_2 which creates an *unintended network loop*.

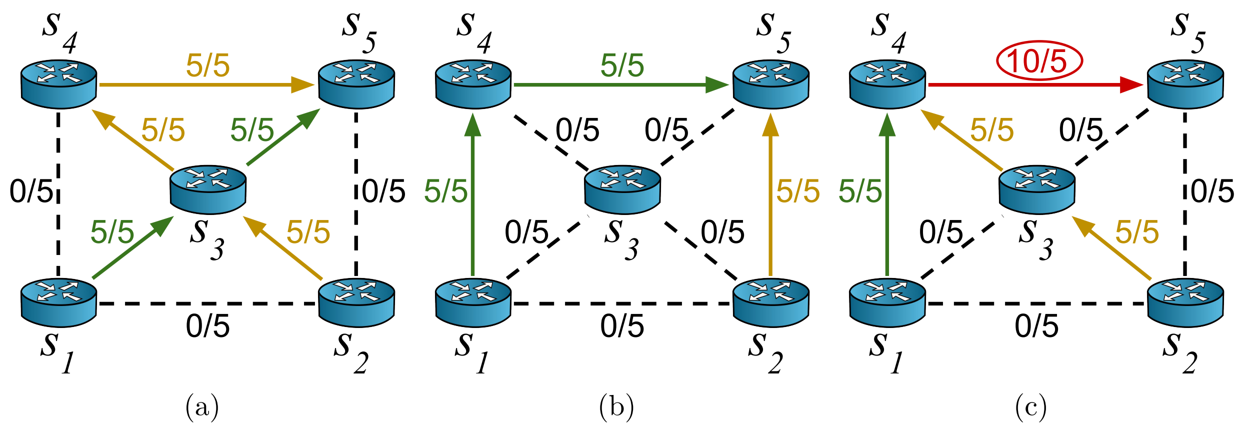


Figure 3.3. **(a)** The state of the flows to s_5 **(b)** is planned to be modified by an update alleviating s_3, **(c)** but the update is applied by s_1 before it is applied by s_2 which causes an *unintended over-provisioning* of the s_4-s_5 link.

3.2.3 Consistency

Asynchrony in network updates can cause transient side effects that can significantly affect switch resources such as overall network availability and/or violation of established network policies. Since data plane switches do not coordinate themselves to ensure update consistency, updates sent to switches in parallel may be applied in any order. While the OpenFlow message layer, arguably the most widely used southbound API for network updates, has proposed bundled updates [71] to provide transaction style updates to switches, it only supports these updates for a single switch. It does not address inconsistencies that can occur due to updates that span multiple switches. Additionally, OpenFlow scheduled bundles require synchronized clocks among switches to enforce the time at which bundles are applied but even the slightest clock skew may provoke transient network behavior.

Table 3.1 summarizes several circumstances as well as potential problems that can arise if update consistency is not provided. For each example, certain preconditions may also be needed by the controller for ensuring update consistency. For instance, even a simple network policy change may have unintended consequences when network updates are not consistent (cf. Figure 3.1). The process of changing data plane state must also be free of transient effects caused by updates to multiple data plane switches: loop and black hole freedom ensures no network loops or unintended drops of network packets (cf. Figure 3.2), and congestion freedom ensures no over-provisioning of bandwidth to network links (cf. Figure 3.3).

3.2.4 Security

When considering a control plane prone to faulty controllers, enforcing a consistent ordering of network updates is not sufficient, those updates must only be applied when received from correct controllers.

A faulty or malicious controller may corrupt or cause loss of network data, violate firewall rules, or even leak network data to a malicious party. While solutions for secure controllers have been proposed, they either focus on resiliency (e.g. intrusion detection, intrusion prevention) for a singleton controller [52], [53] or provide resiliency only in the presence of crash failures [5], [27], [28], [51]. Single controller solutions, proven to be single points of failures [72]–[76], must be avoided.

Many of the existing limitations when considering a faulty control plane arise from shortcomings in the southbound API itself. While OpenFlow enables endpoint authentication through transport layer security (TLS), it assumes correct behavior of the authenticated control plane. However, an authenticated controller that is faulty or compromised is still able to affect network flows in the data plane. For example, Openflow provides a mechanism for the control plane to inject arbitrary packets into the data plane (`PACKET_OUT` [77]). This mechanism alone is enough for a malicious controller to easily launch a denial of service attack against the data plane or to corrupt existing flows [78]. Besides, a malicious controller masquerading as a switch can report incorrect link and switch state to the control plane [79]. A comprehensive solution for security in network updates must be able to tolerate arbitrary controller fault.

3.2.5 Practicality

The usefulness of a system is often evaluated on factors such as ease of use, performance, and efficiency.

Network policy specification must not only be straightforward, but also flexible enough to allow arbitrary network policies. Several solutions for policy specification have been proposed [80]–[82], but are either control plane implementation specific, or provide no mechanism for ensuring update consistency or security. A practical system must allow a network administrator the flexibility to use any solution desired while ensuring consistency and security.

Furthermore, a system for managing changes to the data plane state must scale to a wide network infrastructure consisting of multiple data centers with potentially thousands of switches [83], [84]. Existing work [44] shows that applying updates on commodity switches can require seconds to complete. For data center workloads where flows start and complete in under a second [1], applying updates quickly is vital to guarantee adequate network response time when changing data plane state. However, responsiveness becomes even harder to ensure if updates are to be applied in a consistent manner. In a naïve approach enforcing consistency, updates would be applied sequentially (e.g. by updating s_2, s_1, s_3, s_4 in that order in Figure 3.1), increasing response time. Yet, updates that do not depend on any others, (i.e. causally concurrent updates) may be applied in parallel (e.g. updates to s_3 and s_4 in Figure 3.1). Identifying causally concurrent updates to apply in parallel and improve response times is a challenge.

Finally, the data plane's runtime load for updates must be low to ensure as many resources as possible are used for the network's core purpose; the transmission of network data.

Table 3.2. Comparison of network management solutions for fault tolerance and consistency, considering different features related to security [Sec], consistency [Cons], and practicality [Pract].

System/approach	**Crash FT** [Sec]	**Byzantine FT** [Sec]	**Controller auth.** [Sec]	**Update consistency** [Cons]	**Dyn. membership** [Pract]	**Update domains** [Pract]	**Implementation** [Pract]
Singleton controller							Common [18], [85]–[87]
Singleton controller w/ TLS			✓				Common [18], [85]–[87]
ONOS [28]	✓				✓		Deployed [88], [89]
Ravana [5]	✓						Experimental Ryu extension
Botelho et al. [90]	✓						Experimental
MORPH [69]	✓	✓			✓		Experimental
RoSCo [22]	✓	✓	✓	✓			Experimental Ryu extension
NES [42]				✓			Theoretical specification
Dionysus [44]				✓			Experimental
ez-Segway [91]				✓			Experimental Ryu extension
Optimal Order Updates [92]				✓			Theoretical specification
Cicero	✓	✓	✓	✓	✓	✓	Experimental Ryu extension

3.2.6 Related Work

While the following solutions present methods for solving significant problems that arise in SD-WAN deployments, none however provide the desirable guarantees of consistent network updates in the midst of controller faults while remaining practical. Table 3.2 highlights the shortcomings of these solutions that make them impractical in a realistic deployment.

Consistency

Additionally, there have been several works published in the realm of consistent network updates. McClurg et al. [42] proposed network event structures (NES) to model constraints on network updates. Jin et al. [44] propose Dionysus, a method for consistent updates using dependence graphs with a performance optimization through dynamic scheduling. Nguyen et al. [91] propose ez-Segway, a method providing consistent network updates though decentralization, pushing certain functionalities away from the centralized controller and into the switches themselves. Černỳ et al. [92] show that in some situations it may not be possible to ensure consistent network updates in all cases. As such, it may be desirable to wait until the packets for a particular flow are "drained" from the network prior to applying switch updates. They define this behavior as *packet-waits* and provide an at-worst polynomial runtime called *optimal order updates* which provides a mechanism for detecting such situations.

Fault tolerance

The area of fault-tolerant network updates has been explored in many facets. ONOS [28] and ONIX [27] provide a redundant control plane through a distributed data store, however their primary focus is on tolerance of crash failures. Botelho et al. [90] also make use of a replicated data store, following a crash-recovery model, for maintaining a consistent network state among a replicated control plane built upon Floodlight [93]. Ravana [5], another protocol that only tolerates crashes, differs slightly in its use of a distributed event queue rather than a distributed data store. While Botelho et al. and Ravana ensure event ordering and prevent duplicate processing of events, they do not provide a mechanism for authenticating updates sent to the data plane. RoSCo [22] makes use of a BFT protocol to ensure event-linearizability, but does not support a dynamic control plane and requires extensive key management for controller authentication.

Li et al. [68] proposed a method of a BFT control plane by assigning switches to multiple controllers that participate in BFT agreement. However, this work focuses significantly on the problem of "controller assignment in fault-tolerant SDN (CAFTS)" with little discussion on how BFT is used to ensure protection from faults. MORPH [69] expands the solution of CAFTS with a dynamic reassigner which allows for changes to the switch/controller assignment. Neither method fully protects against malicious updates sent to the data plane;

assuming that controllers participate in some BFT protocol for state machine replication is not enough to ensure the security of such updates. Without control plane authentication, a malicious controller can make arbitrary updates to a data plane switch. While it may seem trivial to add TLS for OpenFlow [94], this requires additional complexities inherent in the protocol. TLS uses certificates to authenticate participants and encryption to ensure confidentiality of data, but does not protect against a faulty controller. Besides, as distributed control plane membership changes, individual controller and switch certificates must be redistributed to all participants.

3.3 Cicero Components

In this section, we detail the various mechanisms Cicero employs to ensure both consistency and security while being efficient enough for practical deployment in a production data center. § 3.3.1 describes how Cicero handles consistency in a modular manner within a local network by using an *update scheduler*; § 3.3.2 describes how update security is enforced in a dynamic control plane through *event authentication*, *controller agreement*, *quorum update authentication*, and *unique key adaptation*; finally, § 3.3.3 describes facilities that Cicero exploits to improve the performance of network updates in both local and wide area networks through *intra-* and *inter-domain update parallelism*, and *signature aggregation*.

3.3.1 System Model and Consistent Network Updates

The *data plane* is a set of *switches* connected by links encompassing multiple *domains* of operation. The *control plane* consists of a *dynamic* set of distributed controllers. A change in data plane state requires a set of network updates $\{u_{\mathrm{i}} = (s_{\mathrm{j}}, r_k)\}$ where (s_{j}, r_k) indicates rule r_k being applied to switch s_{j}. An *update scheduler* determines a schedule by denoting dependencies between updates. Let an update dependence be represented by a tuple (u, D) where u is the update to be applied (consisting of (s, r)), and D is the set of updates that must be applied before u. For all updates u_{i}, the update scheduler must determine the set of all update dependencies.

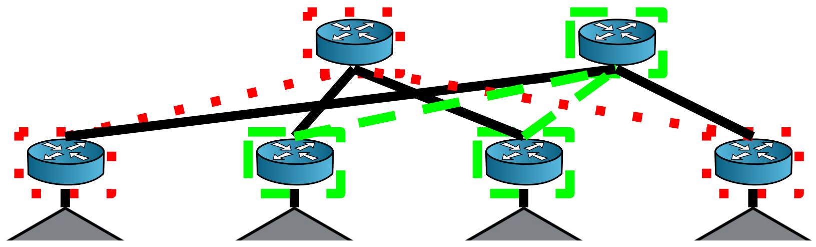


Figure 3.4. The update scheduler determines that there are no dependencies between the updates for the green (dashed) set of switches and the updates for the red (dotted) set.

Figure 3.1 depicts an example which requires a set of updates for switches s_1, s_2, s_3, and s_4. To ensure update consistency, an update scheduler would require the update at s_2 to be completed first and, once that update has been applied, the remaining updates can be performed in any order. This is further depicted in Figure 3.4 where a network update requires modifications to the switches highlighted with green dashes and red dots. While the updates within these two sets of switches may require ordering, modifications across sets involve a disjoint set of switches and can be performed in any order.

The area of update schedulers has been extensively discussed [44], [92], [95], [96]. Our goal is to provide a practical construction and protocol for consistent and secure network updates. As such, we assume the existence of a basic update scheduler implemented using any of these approaches. We discuss in § 3.3.3 how Cicero exploits that update scheduler to perform updates to switches in parallel while still preserving *consistency*, i.e., enforce the specification of the update schedulers even assuming malicious or faulty controllers concurrently invoking network updates.

3.3.2 Security

At its core, secure network updates require switches to apply updates only from a trusted controller. While a message from a controller can be easily validated using message signatures, trust in a single controller is not enough when considering malicious faults (e.g. a compromised controller can easily sign malicious messages with a valid signature). Cicero increases trust in the control plane by requiring the agreement of a quorum majority from multiple controllers on the set of updates. In essence, we employ a dynamic distributed control plane with controllers co-signing network updates.

Event generation – event authentication

A change in data plane state is assumed to be invoked as the direct result of some event, be it the result of a switch detecting an unroutable packet (e.g. mismatch in flow table rules), a change in network policy, a failure of network hardware, or some other factor. Events received by the control plane require validation to ensure that they originated from a reliable source. To this end, Cicero makes use of a public key infrastructure (PKI) system. Each event source is assigned a public/private key pair. When an event is generated, the originator signs the event with their private key. This ensures that events are only generated by known sources avoiding the case where controllers process events from untrusted entities in the network.

Event broadcast – controller agreement

A controller, upon receiving an event, broadcasts the event to all other controllers through an established agreement protocol. Next, a controller, upon receiving an event from another controller, independently responds to the event with network update(s). A switch only applies an update if it has been received from a quorum of trusted controllers. We make use of an atomic broadcast [97] (i.e. consensus) to ensure each controller has a consistent view of the data plane state. Controllers use a PKI system to validate messages sent with the atomic broadcast.

Threshold signatures – quorum update authentication

Each controller signs the updates they emit so switches can verify the origin of the updates they receive. The strawman approach consists of controllers being assigned different pairs of public/private keys for signing updates. Switches only apply updates with valid signatures (i.e., from controllers) that are emitted from a quorum of verified controllers. However, managing all the public keys on all the switches rapidly becomes cumbersome as controllers may be added to and/or removed from the control plane. Moreover, the limited physical resources of switches must be preserved (cf. § 3.3.3)

To this end, we employ a system based on threshold cryptography [98], [99]. In a (t, n)-threshold signature scheme, a *single* public/private key pair is generated for the entire control plane. The public key is distributed to each switch, while each controller obtains a share of the associated private key used for signing updates thanks to Shamir secret sharing [100]. To verify an update, the signature shares received from each controller are combined with an aggregation function to create a signature that is verified against the single public key. The aggregated signature can only be verified if correctly signed by at least t out of n controllers, thus any $t-1$ controllers, with the exception of negligible probability, can never on their own construct a signature that can be verified against the public key for the entire control plane. We set t to the controller quorum size necessary to apply an update, i.e. $t = \lceil \frac{n+1}{2} \rceil$.

We note that to tolerate a single failure, there must be at least 4 members in the control plane (i.e. $\lfloor \frac{n-1}{3} \rfloor \geq 1$). Thus, we assume Cicero never runs on control planes with $n < 4$.

Distributed key generation – unique key adaptation

Using threshold cryptography and secret sharing for update validation establishes a method for secure updates in a dynamic distributed control plane. However, distribution of private key shares when controller group membership changes creates a significant complication: no single controller should ever have knowledge of a private key share other than its own. Verifiable secret sharing (VSS) [101] is a method in which a designated dealer distributes shares of a secret to all participating members. VSS differs from standard secret sharing in that clients can construct a valid share even if the dealer is malicious. These shares can be used in a (t, n)-threshold signature scheme to create message signatures that are only validated if at least t members correctly sign the message with their shared secret. Naïvely, one could employ such a system to distribute private key shares to controllers when the control plane membership changes. However, requiring the setup and maintenance of such a system is impractical as the VSS dealer is a single point of failure for confidentiality.

We instead employ a system based on distributed key generation (DKG) [102] that expands on the concept of VSS to an environment where there is no trusted dealer. In short, each controller acts as a sub-dealer, creating and distributing private key sub-shares to each other controller. The sub-shares are then aggregated to create the private key share for the controller. DKG uses homomorphic commitments to ensure that the corresponding public key for the group is known by all controllers, but except for negligible probability, no one controller can create a signature that is successfully validated by the public key. Once generated, this public key must be shared to all switches. Future instances of DKG ensure that new shares can be generated for the control plane as group membership changes without changing the public key.

3.3.3 Practicality

Amidst consistency and security, for a solution to be feasible in a real data center deployment it must also be practical. Cicero provides an effective solution by exploiting intra- and inter-domain update parallelism, and enabling efficient signature aggregation to alleviate switches runtimes.

Update parallelism – intra-domain parallelism

Using an update scheduler (cf. § 3.3.1) allows Cicero to exploit parallelism in network updates. Given a set of network updates and their corresponding update dependencies determined by the update scheduler, two updates u_i and u_j can be applied in parallel if their dependencies D_i and D_j are disjoint, i.e. $D_\mathrm{i} \cap D_\mathrm{j} = \emptyset$.

Update domains – inter-domain parallelism

Cicero employs an atomic broadcast (cf. § 3.3.2) to ensure a consistent ordering of events processed by the control plane. The responsiveness of such agreement protocols unfortunately greatly deteriorates as the size of the control plane increases, hence creating a trade-off between fault tolerance and performance. Additionally, in large networks such as a collection of data centers, this responsiveness is further impacted by having a geographically

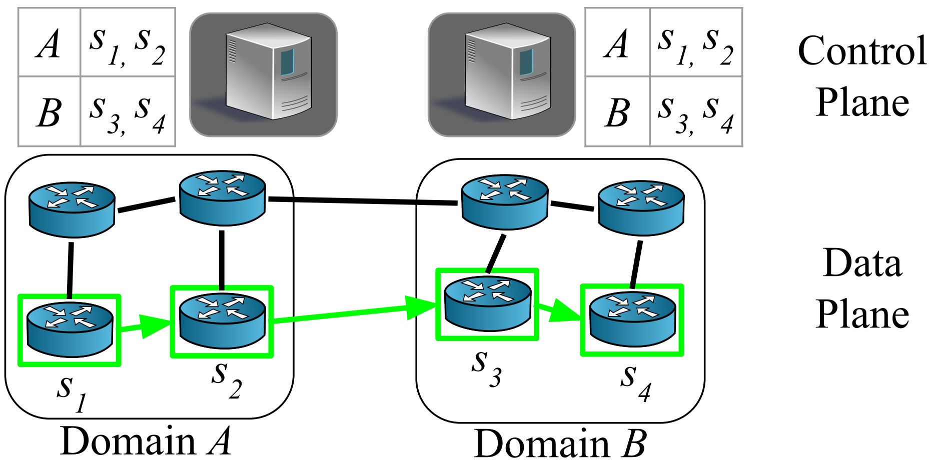


Figure 3.5. Depiction of a two domain network where an event generated by switch s_1 and sent to its local domain control plane. The control plane then uses global domain policies to determine that network updates involve both domains A and B. The control plane of A forwards the event to B and both domains update their local switches to set flow tables rules.

dispersed control plane. This distribution is initially set to minimize latency between local control and data planes, but ultimately increases latency within the global control plane.

As such, Cicero allows the division of network resources into domains, each as its own separate instance of the protocol functioning on disjoint control and data planes, e.g. separate IP subnetworks. Domains may rely on separate update schedulers, agreement communication groups and control plane public keys. The goal of this division is to enable data plane events that involve updates to switches fully contained within the same domain to be processed independently, i.e. in parallel, of other such events in other domains. Events that require updates spanning multiple domains must however be handled in a consistent manner by the control plane as a whole.

Cicero avoids the need for inter-domain agreement through assumptions on setup and *global domain policies*. First, we assume operators of different domains trust each other, e.g. domains are sub-domains of the same institution. This domain isolation thus offers the security that a, potentially faulty, domain's control plane cannot update another domain's data plane, but it may affect flows with a remote origin crossing the data plane it is responsible for. Second, we assume the global domain policies are static. As such, each domain's control plane is able to determine which domains require updates based on a received event. A controller receiving an event that involves updates to multiple domains merely forwards the event to the control plane of each affected domain.

For example, consider the flow outlined in Figure 3.5 where an event generated by switch s_1 in domain A needs a route to s_4 to be established. Using the static global domain policies, the controller in A that receives the event determines that it requires updates to both domain A and B, and forwards the event to the control plane of domain B. Both domains process the event in parallel and update the switches within their domain accordingly, setting the flow table rules of switches to establish a flow from s_1 to s_4.

Update signature aggregation

To verify an update, the signature shares from each controller must be collected and aggregated prior to verification against the threshold public key. Putting this responsibility on switches can put unnecessary load on their hardware. As such, Cicero presents two approaches for signature aggregation:

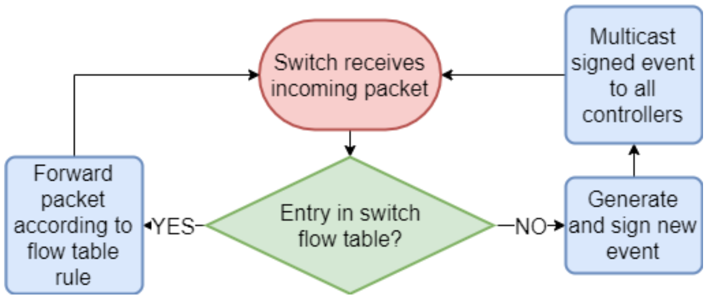
(1) switch aggregation in which each individual switch is responsible for collecting and aggregating update signatures, and

(2) controller aggregation in which a single controller is designated the "aggregator" that collects and aggregates signatures.

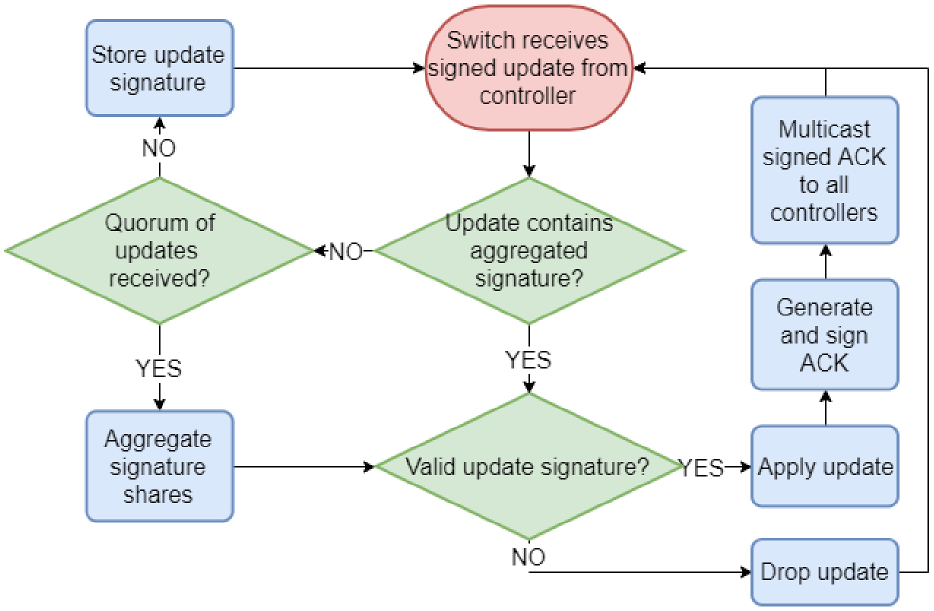
Each approach comes with its own trade-offs. While switch aggregation requires additional resources and instrumentation on switches for storing and aggregating signatures, controller aggregation increases latency since switches must wait for the aggregator to collect and aggregate responses. Furthermore, controller aggregation must be able to handle detection of a failed or malicious aggregator. Our evaluation in § 2.7 further quantifies the trade-offs of each approach.

3.4 Cicero Protocol

In this section, we show how the components depicted in § 3.3 form a protocol with:


Switch receives incoming packet
Multicast signed event to all controllers
Forward packet according to flow table rule
YES
Entry in switch flow table?
NO
Generate and sign new event

(a) Switch forwarding process


Store update signature
Switch receives signed update from controller
Multicast signed ACK to all controllers
NO
Quorum of updates received?
NO
Update contains aggregated signature?
Generate and sign ACK
YES
YES
Aggregate signature shares
Valid update signature?
YES
Apply update
NO
Drop update

(b) Switch update process

Figure 3.6. Flow charts describing the processes of a switch **(a)** handling incoming packets on the data plane and **(b)** handling updates received from the control plane.

(1) secure and consistent network updates,

(2) signature aggregation, and

(3) membership changes.

We further comment on the guarantees of the protocol.

3.4.1 Secure and Consistent Updates

The Cicero secure and consistent network update protocol employing atomic broadcast, threshold cryptography, and acknowledgements is composed of two independent routines: (1) switch runtime and (2) controller runtime. The controller runtime can further be broken down into the handling of events within and across multiple domains.

Switch protocol

Figure 3.6 depicts the update processes for a switch when it receives either a packet from the data plane (Figure 3.6a) or an update from the control plane (Figure 3.6b). Normal operation for a switch is to use the flow table rules established for network policies to store and forward packets in the network. Upon receiving a packet that does not match a flow table rule, a switch generates and signs an event indicating the mismatch and sends it to all members of its domain control plane. Any network update received from the control plane is not immediately applied. The message, containing an update and signature, is stored by the switch until the switch receives a quorum majority of identical updates from control plane members. Once enough messages are received, using the threshold signature aggregation function, the switch aggregates the signatures for the update and verifies the resulting signature against the public key for the control plane. The update is then either applied or ignored, depending on the validity of the signature. Finally, the switch sends a signed acknowledgement to all members of the domain control plane to alert them of the network update application.

Controller protocol

Depictions of the process for a controller when it receives an event, or when agreement is reached on the ordering of events are shown in Figure 3.7 and Figure 3.8. Under normal operations controllers for a domain of switches are idle waiting to receive signed events. Upon receiving an event, the source of the event is verified and the event is either ignored, if

(1) the event was previously processed or

(2) the event source cannot be verified,

or broadcast to all members of the domain's control plane. Using the established network policies and the update scheduler, each member of the control plane independently determines the necessary network updates and dependency sets in response to the event. Each network update is signed with the controller's private key share. Network updates for disjoint dependency sets are processed in parallel with network updates having no dependencies being immediately sent to the corresponding switch(es). As verified acknowledgements for applied updates are received, these updates are removed from dependency sets and additional updates are sent, in parallel, to the switch(es) for empty dependency sets. Since switches are assumed nonfaulty, these received acknowledgements ensure forward progress in event processing despite the presence of loops in the protocol flow.

3.4.2 Inter-domain updates

If, thanks to the global domain policies, a controller determines that an event affects multiple domains, it forwards the event to a controller in each affected domain. The receiving controllers broadcast the event to all other controllers of their respective domain as with any validated event. To select a valid recipient, each controller maintains a set of active controllers in each other domain. This list is updated every time a controller is added or removed to/from any other domain's control plane (cf. § 3.4.4). Furthermore, to prevent never-ending dissemination of the event, a forwarded event is tagged as such to indicate it should not be further forwarded to other domains and only be processed locally.

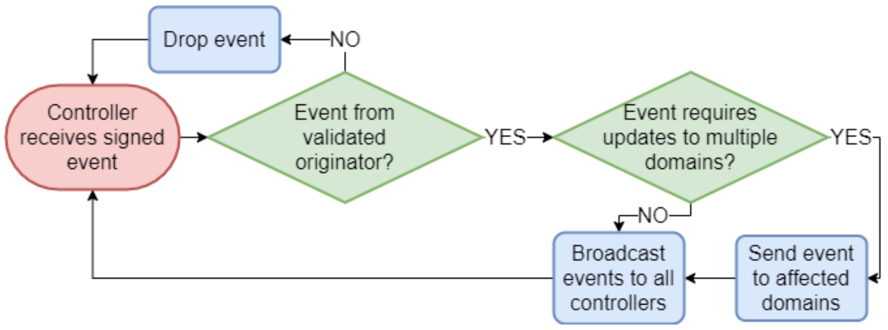

Drop event
NO
Controller receives signed event
Event from validated originator?
YES
Event requires updates to multiple domains?
YES
NO
Broadcast events to all controllers
Send event to affected domains

Figure 3.7. Flow chart for controller receive event process handling incoming events

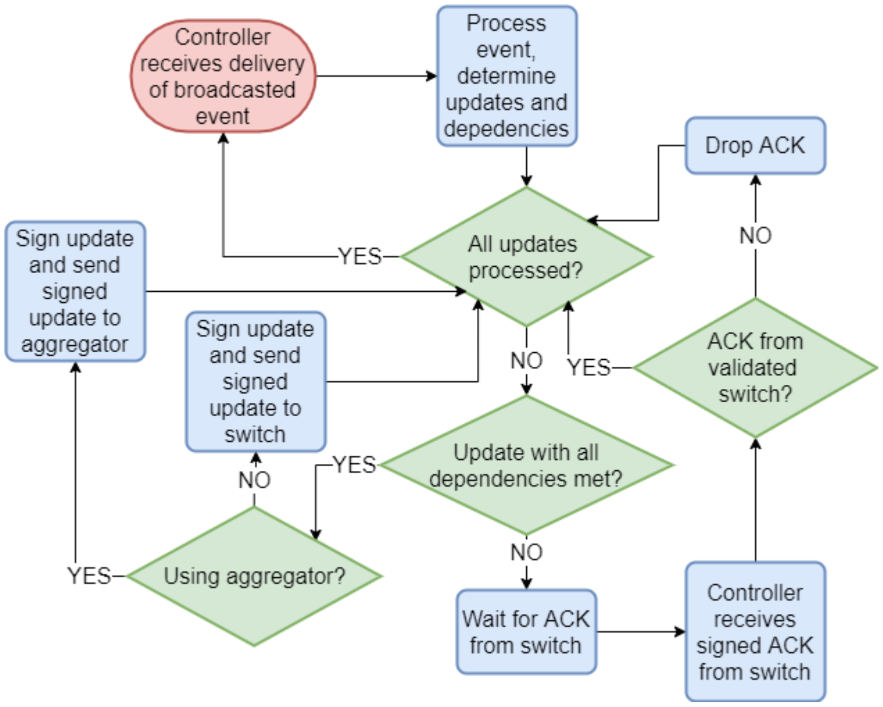

Controller receives delivery of broadcasted event
Process event, determine updates and depedencies
Drop ACK
All updates processed?
YES
NO
Sign update and send signed update to aggregator
Sign update and send signed update to switch
ACK from validated switch?
YES
NO
Update with all dependencies met?
YES
NO
Using aggregator?
YES
NO
Wait for ACK from switch
Controller receives signed ACK from switch

Figure 3.8. Flow chart for controller update process handling updates to be sent to the data plane

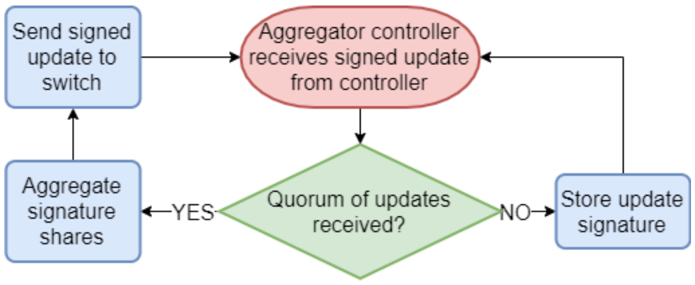

Send signed update to switch
Aggregator controller receives signed update from controller
Aggregate signature shares
YES
Quorum of updates received?
NO
Store update signature

Figure 3.9. Flow chart for aggregator controller process aggregating updates from other controllers

3.4.3 Controller Aggregation

The Cicero protocol outlined in § 3.4.1 specifically focuses on switches aggregating signatures. Optionally, controller aggregation may be used in which a controller is assigned to be the aggregator for both receiving events from switches and collecting (to aggregate) signed updates. The process for controller aggregation is depicted in Figure 3.9. Switches, instead of sending events to all controllers in their domain, only send them to the aggregator. Controllers, instead of sending signed updates to switches, send them to the designated aggregator. The aggregator collects signed switch updates, aggregates the signatures once a quorum has been received, and sends the update along with the aggregated signature to their respective switch. A switch receiving aggregated signatures merely verifies the update's signature against the public key of the control plane and either applies or ignores the update.

Aggregator selection

All controllers for a domain maintain a representation of the control plane communication group containing each controller's identifier, public key, and any information needed for communication (e.g., IP address, port). As new controllers are added (cf. § 3.4.4), they are given the next highest unused identifier. Identifiers are never reused, even when controllers leave the group. At any given time, the aggregator can be determined as the controller with the lowest identifier. Since all controllers in the domain have the same view of the communication group, this provides stability in the selection. Once an aggregator is determined, the control plane members inform switches by sending a signed message.

3.4.4 Control Plane Membership Changes

The process for a domain's control plane membership change is depicted in Figure 3.10. Due to the potential change in quorum size, both add and remove operations require the distribution of new private key shares. The Cicero protocol ensures that no events are processed until after the membership change has completed, which prevents control plane members from having to keep old and new shares concurrently. A phase value records the current iteration of membership change. The phase value is incremented with each controller addition or removal. Controllers must be added and removed one at a time ensuring lockstep increment to the phase. Events broadcast to all domain controllers are tagged with the current phase. Thanks to atomic broadcast, controllers queue events received during a change in control plane membership and only broadcast and treat them after the phase has changed.

Add controller

The procedure to add a controller to the control plane is as follows:

(i) public keys for event originators and existing control plane members are distributed to the new controller alongside its identifier;

(ii) the new controller is added to the control plane communication group though consensus proposed by the bootstrap controller;

(iii) DKG is executed to distribute signature shares to the new controller group reflecting the new quorum size and ensuring that the threshold public key remains the same;

(iv) the data plane state and both local network policies from the control plane and global domain policies are sent to the new controller.

Cicero uses a trusted bootstrap controller to manage additions to the control plane. It is the only control plane member that can initiate consensus rounds to add new controllers.

The final step requires updating all other domains to indicate the new controller as a valid recipient of forwarded events. Here, the bootstrap controller generates and signs an event containing the new controller's communication information and forwards this to a member of each other domain. Each receiving domain, in parallel, processes the event as any other network event (e.g. atomically broadcasts the event to all members of the local domain). However, instead of sending network updates, a controller handles this event by updating its view of the sender's control plane.

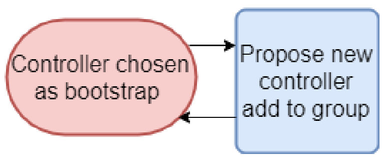
Remove controller

The procedure to remove a controller from the control plane is as follows:

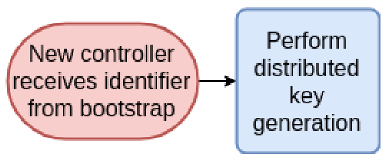
(i) the controller is removed from the control plane communication group;

(ii) DKG is executed to distribute signature shares to the controller group reflecting the new quorum size and ensuring that the threshold public key remains the same;

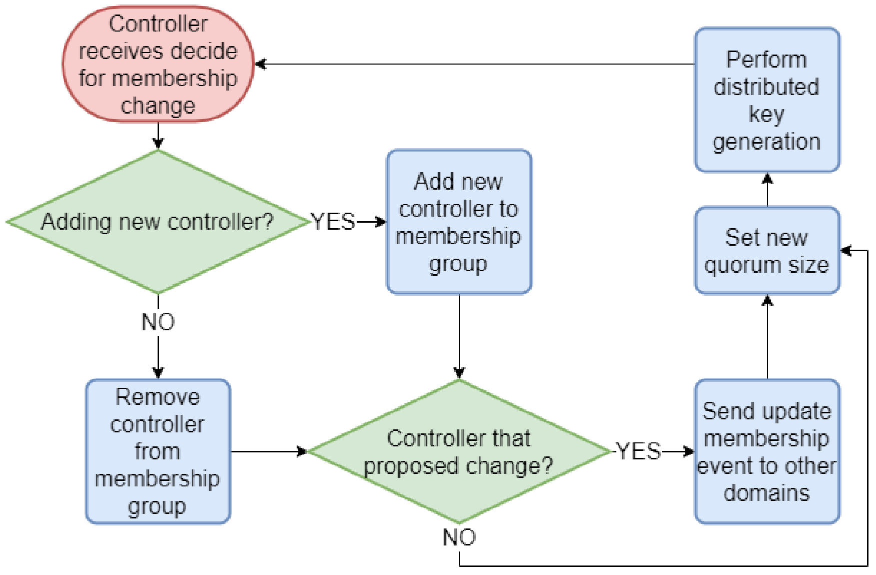
(iii) switches are (potentially) assigned a new aggregator.


Controller chosen as bootstrap
Propose new controller add to group

(a) Controller bootstrap process


New controller receives identifier from bootstrap
Perform distributed key generation

(b) New controller process


Controller receives decide for membership change
Adding new controller?
YES
NO
Add new controller to membership group
Remove controller from membership group
Controller that proposed change?
YES
NO
Send update membership event to other domains
Set new quorum size
Perform distributed key generation

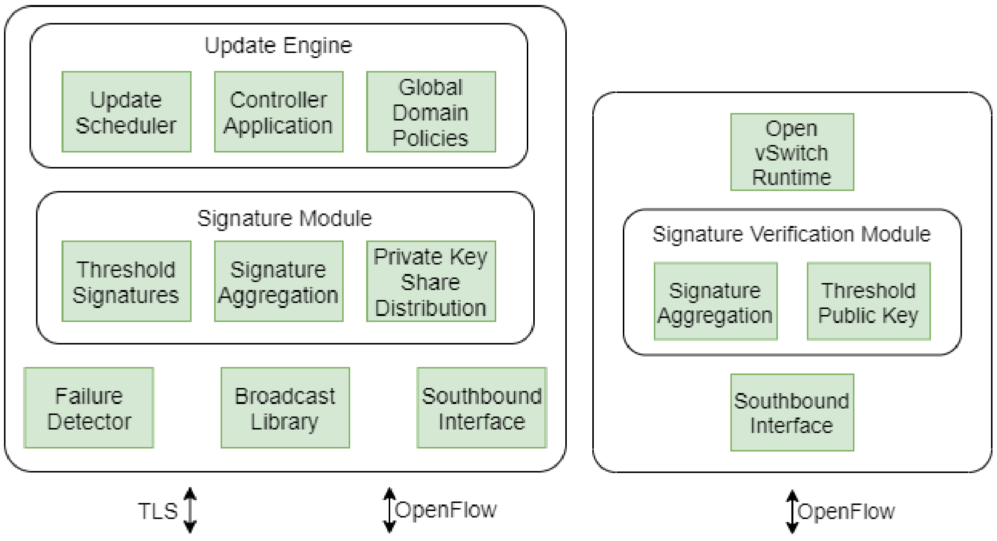
(c) Controller membership consensus

Figure 3.10. Flow charts for controller membership change: **(a)** and **(b)** show the processes for the bootstrap controller and the joining controller respectively, and **(c)** shows the controller process when a membership change consensus is reached.

Removing the controller from the communication group is performed via a round of consensus proposed by a member that detects that the member should be removed.

The final step requires updating all other domains to indicate the removed controller is no longer a valid recipient of forwarded events. As when adding a controller, an event is sent to a controller of each other domain. The event is in turn processed in parallel by each domain's control plane where each controller updates its view of the sender's control plane.

We assume the existence of a failure detector [103]–[105] capable of accurately detecting the failure of an existing controller. A controller can also be pro-actively removed merely by either simulating a failure (e.g., loss of power) or proposing its own removal through the consensus protocol. We recognize that it is impossible to ensure 100% accuracy with failure detection. However, premature removal of detected failed controller only affects liveness of



(a) Cicero controller runtime. (b) Cicero switch runtime.

Figure 3.11. Depiction of the Cicero runtime components.

the system. Furthermore the Cicero protocol allows for re-adding of previously removed controllers. Through consensus and quorum authentication, Cicero ensures that it is impossible for network updates to be applied to the data plane in the event of a faulty controller being undetected, provided that the number of undetected faults is at most $\lfloor \frac{n-1}{3} \rfloor$ for n controllers (cf. § 3.3.2).

3.5 Cicero Implementation

As Figure 3.11 shows, Cicero is implemented as a middleware between the controller application, containing network policies, and the data plane switches, storing and forwarding network traffic based on established flow table rules.

3.5.1 Control Plane Components

The controller platform is extended with a Java layer for Cicero, which processes the received events (e.g., signature verification, broadcast) and updates sent to the data plane (e.g., signing with secret share, ordering updates, and handling acknowledgements). Another process in the Java layer handles signature aggregation to be sent to the data plane when controller aggregation is used. A controller is made up of the following nine components:

3.5.2 Controller application

Network policies are set based on the controller application. While Cicero is designed as a separate layer to allow for any controller application, our implementation uses the Ryu [18] runtime and establishes rules for flows based on shortest path routing.

3.5.3 Global domain policies

Cicero requires global domain policies for determining network updates for flows that cross domains. The implementation is specific to the controller application. Our implementation uses global policies based on the shortest path between domains.

3.5.4 Update scheduler

To ensure update consistency, the Cicero runtime depends on the existence of an update scheduler used to determine dependencies between network updates. The update scheduler used for the evaluation assigns dependencies for network updates based on the reverse of a network flow's path. For example, consider a network flow that traverses three switches ($s_1 \rightarrow s_2 \rightarrow s_3$). Establishing this flow requires updating all of these switches. The update scheduler assigns dependencies for these updates such that (1) all updates are applied to s_3 before any updates to s_2 can be applied, and that (2) all updates are applied to s_2 before any updates to s_1 can be applied. This ensures downstream rules for the flow are set before any network data is allowed to traverse the network.

3.5.5 Broadcast library

Cicero utilizes atomic broadcast to distribute events among the members of the control plane communication group. The broadcast library strictly follows atomic broadcast's specifications and guarantees [97] by using the routines of the BFT-SMaRt library [60].

3.5.6 Threshold signatures

Data plane switches authenticate updates with threshold signatures that can only be verified when a quorum of signatures is formed. Our implementation makes use of BLS signatures [106] implemented in the Pairing Based Cryptography library [107].

3.5.7 Private key share distribution

The distribution of private shares for controllers so they can sign switch updates is performed using the DKG library [108].

3.5.8 Southbound interface

We extend the OpenFlow message protocol to add new message types for signed messages, and add a unique identifier to each message to prevent duplicate processing of events and updates.

3.5.9 Signature aggregation

Cicero supports switch and controller aggregation. For the latter, switches are assigned the aggregator with OpenFlow "master/slave role request" messages [109].

3.5.10 Failure detector

We use periodic heartbeat messages to detect failures and use the broadcast library for transport.

3.5.11 Data Plane Components

The Cicero switch platform is an extension to Open vSwitch (OVS) to perform signature aggregation and verification of updates both thanks to threshold public key component. Additionally, changes are made for switches to either send events only to the aggregator controller if there is one, or multicast events to all the members of the control plane. As a further consistency mechanism, acknowledgments are sent to the control plane once updates are applied.

As is clear in Figure 3.11, the switch runtime is considerably simpler than the controller runtime. We specifically designed Cicero to minimize the resource consumption impact on switches because of their low capabilities.

3.6 Cicero Evaluation

We here show how the strong guarantees for consistent and secure updates in Cicero can be achieved with little overhead in practical networked environments. We further show how aggregation and multi-domain parallelism reduce that cost.

3.6.1 Experimental Methodology

We evaluate Cicero against existing update frameworks in typical business-like environments. As such, we compare (1) a centralized controller, (2) a crash-only tolerant update protocol where communication within the control plane is performed using the atomic broadcast provided by the BFT-SMaRt library, but with no quorum authentication of signatures on switches, and (3) the Cicero update protocol on a single-domain setup with and without aggregation on controllers (cf. § 3.6.2) and on a multi-domain setup (cf. § 3.6.3).

Setup

We executed the implementation detailed in § 2.6 on a network simulated atop compute nodes from the DeterLab test framework [61], [110] connected via a 1 Gb test network. Nodes ran Ubuntu 18.04.1 LTS with kernel 4.15.0-43, two Intel® Xeon® E5-2420 processors at 2.2 GHz, 24 GB of RAM and a SATA attached 256 GB SSD. Controllers had their own node, switches and hosts were node-sharing OpenVz [111] instances.

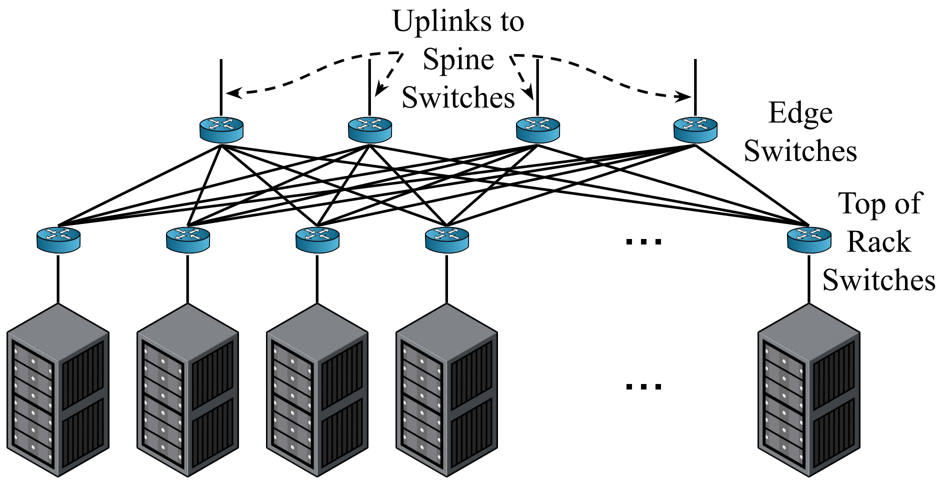


Figure 3.12. Depiction of a server pod, made up of racks and two layers of switches atop, in a Facebook data center [112].

Topology

We simulated the Facebook data center topology [112] where data centers are divided into server pods (as depicted in Figure 3.12) consisting of 40 racks of compute servers. Each rack contains a top-of-rack switch connecting all servers in the rack. Each top-of-rack switch is connected to 4 edge switches that provide high speed bandwidth and redundancy between racks. Edge switches further connect multiple pods to spin switches (unshown in Figure 3.12) linked to the upstream network.

Workloads

We ran Hadoop MapReduce and web server traffic workloads [1] over the given topology and measured their flow completion times according to the shortest path routing policy used by the controller application. We evaluated completion times for 5000 flows using each framework. Flows follow a Poisson distribution using average packet sizes and total flow sizes (in kB) for inter-rack, intra-data center, and inter-data center defined for each workload.

Creating routes

Unless explicitly stated otherwise, rules in flow tables are reused for multiple flows. Flow tables in switches initially contain no forwarding rules. As flows enter the network, events for unroutable packets are generated by switches and sent to the control plane. Controllers respond with network updates sent to switches to establish rules for the flows. As flows complete, these rules remain in switch flow tables and are reused by later flows matching them. As reported in [1] for Hadoop workloads 99.8% of traffic originating from Hadoop nodes is destined for other Hadoop nodes in the cluster. Reusing rules requires fewer overall events. Switches do not need to contact the control plane for each new flow.

3.6.2 Single-domain Evaluation

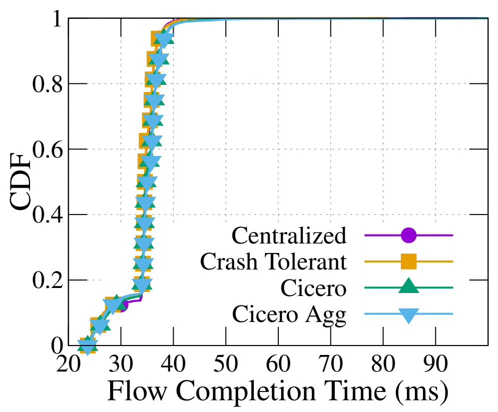
In the following, we used a single server pod topology with a control plane made up of 4 controllers that tolerates 1 failure and results in a quorum size of 3. This evaluated control plane size is similar to evaluations of related work [5], [22], [90].

Flow completion time

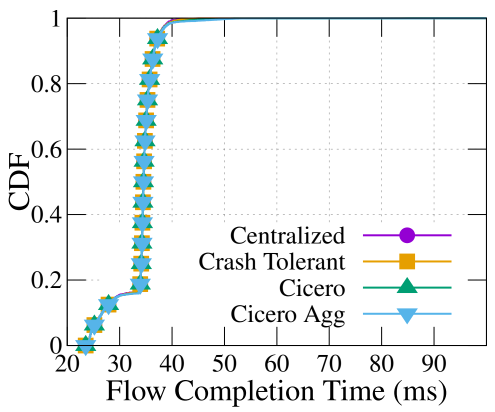
Figure 3.13a and Figure 3.13b show flow completion times for the Hadoop and web server workloads, respectively. Setting up a flow takes an average of ≈2.9 ms for a centralized controller and ≈4.3 ms for a crash fault-tolerant replicated control plane. Cicero is slower due to the extra messaging and therefore takes ≈8.3 ms without and ≈11.6 ms with controller aggregation for flow setup. However, since flow rules are not removed from switches after they are established, they are reused for future arriving flows. Therefore, after initial flow setup, the overhead of Cicero is negligible.

Unamortized flow creation

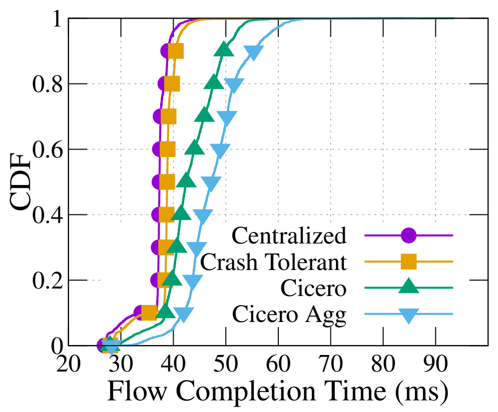
To further investigate the overhead of Cicero, we ran the Hadoop workload using a setup/teardown approach. In this approach, no flow rules for routes are initially set in the data plane. Each flow is managed by a pair of events to inform the control plane to set



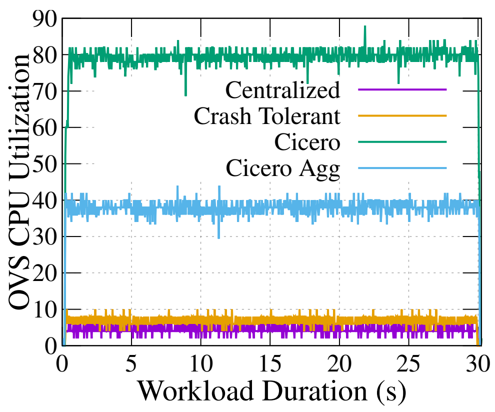
(a) Hadoop flow completion.



(b) Web server flow completion.



(c) Hadoop flow completion unamortized.



(d) Switch CPU utilization.

Figure 3.13. Cicero performance on a single-domain network comparing a centralized solution to a control plane, made of 4 controller replicas, that uses either a crash-tolerant update protocol, Cicero without/with controller aggregation. **(a)** and **(b)** depict the CDF of Hadoop and web server flow completion times, respectively. **(c)** depicts the CDF of Hadoop flow completion times when routes are removed upon flow completion. **(d)** depicts the (switch) CPU utilization of OVS during a Hadoop workload.

the route for the flow before it starts, and clear the flow rules for the route once the flow is completed, hence preventing overhead amortization. Each event results in appropriate network updates. The setup/teardown approach is applicable in hosted networks such as those utilizing subscription-based services.

The average flow completion times are depicted in Figure 3.13c. For Hadoop flows, lasting ≈33.6 ms on average, Cicero has an overhead of 16% with switch aggregation and 29% with controller aggregation over the centralized approach. Setup times are constant regardless of overall flow duration. Since these setup times are the same for all flows, Cicero's overhead with these short-lived flows would be shadowed by the total flow execution time for longer running flows.

Switch resource usage

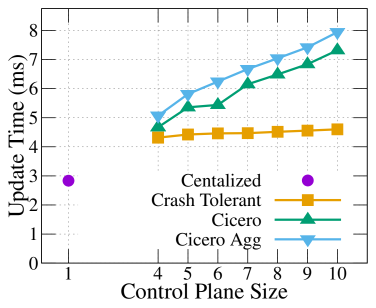
To reduce switches' CPU utilization, update signatures can be aggregated on the control plane at the cost of increased latency (cf. Figure 3.13c). Figure 3.13d depicts OVS CPU utilization on switches for the Hadoop workload. While Cicero signature verification increases CPU utilization on switches, controller aggregation halves switch CPU usage. While using switch aggregations of signatures results in higher CPU utilization, this did not result in an increased latency in the processing of updates.

3.6.3 Multi-domain Evaluation

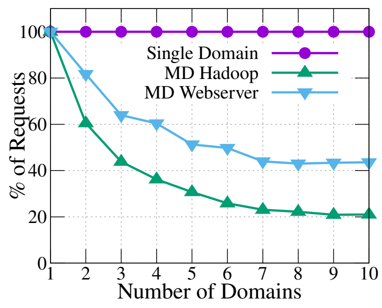
As discussed in section 3.3.3, Cicero provides a means to logically divide the data plane into separate network domains each with its own separate control plane. Events generated within a domain requiring updates solely to the data plane contained in the domain, i.e. local events, can be processed independently of other domains' local events. As we will show shortly, this separation can reduce the load on the control plane(s) and improve scalability. This separation is particularly useful in the face of large networks that share the same large control plane for simplicity. We first evaluate the cost of various control plane size to display the benefit for multiple domains.

Control plane size

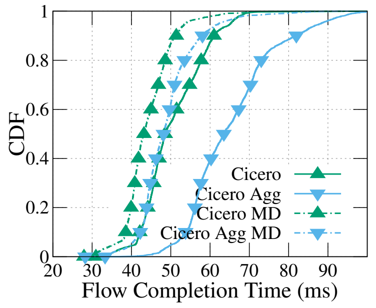
While increasing the control plane membership size allows for more controllers to be faulty, providing additional robustness, it also results in additional messaging for broadcasting events as well as an increased latency due to quorum size, both of which increases the overhead of updates. To examine this overhead we performed a series of updates with control plane sizes varying up to 10 members.



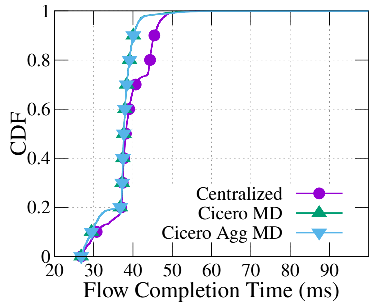
(a) Network update time in one domain depending on the control plan size.



(b) Events handled per control plane with multiple domains (MD) in a pod.



(c) Hadoop flow completion with multiple pods/domains (MD).



(d) Web server flow completion with multiple data centers/domains (MD).

Figure 3.14. Cicero performance for multi-domain networks. **(a)** depicts the average time to apply switch rules in a domain for a varying sized control plane. **(b)** depicts the comparison of events processed by each controller in a pod configured as single vs multi-domain. **(c)** depicts the CDF of Hadoop flow completion times for both single and multiple domains. The single domain is made of 12 controller replicas while the multi-domain consists of 3 domains each with 4 controller replicas (i.e. 12 controllers in total). **(d)** depicts the CDF of web server flow completion times for a larger multi-data centers topology.

The results in Figure 3.14a depict the average time to perform a switch update for an event depending on the size of the control plane. A control plane size of one represents an unprotected centralized control plane. As expected, we see a direct relation between increased control plane size and update time due to the extra messaging needed for broadcast and verification of aggregated signatures. The crash-tolerant update approach is less impacted than Cicero by the growth of the control plane size since switches do not authenticate updates; the additional overhead is merely due to extra messaging.

With Cicero, the overhead for a single switch update can be significant for a large control plane, e.g. 2.5× that of a centralized approach when using 10 controllers to support 3 failures. However, in a data center environment, such a large control plane might be excessive as failures are typically short-lived and failed controllers are quickly replaced with new correct ones. For instance, tolerating 2 concurrent failures is enough to achieve five nines (99.999%) of up-time [113]. Further, splitting the network into disjoint domains may help reduce overhead inherent to a growing control plane.

Event locality

We next investigated how increasing the number of domains within a single pod affects events processing. Due to the locality of flows as reported by Facebook [1], only 5.8% of the Hadoop workload and 31.6% of the web server workload required processing by multiple domains.

Figure 3.14b shows the percentage of total events (for the whole data center) that must be processed by each control plane. For a single network domain, all events must naturally be processed by the single control plane. As the number of domains increases, the number of events processed by each domain's control plane is greatly reduced, however with diminishing returns. While this evaluation shows the gains achievable using multiple domains for one pod, it is more practical to increase the size of the network by adding more pods. To that end, we next evaluated the impact of event locality by increasing the number of pods in the data center with one domain per pod.

Multi-domain flow completion time

We executed the Hadoop workload using 2 server pods, each set into its own domain with a third domain (containing 4 redundant switches) used to interconnect them. Each domain's control plane consisted of 4 controller replicas resulting in 12 replicas for the entire network.

We compared this setup to the same network topology with a single domain and a control plane of 12 replicas.

Figure 3.14c shows flow completion time using Cicero in the single and multi-domain (MD) setup, with and without controller aggregation. Thanks to their locality, most events are processed in parallel when using multiple domains, thus greatly reducing flow completion time compared to a single domain. While flows crossing domains incur extra overhead, an efficient domain architecture can reduce their number.

Multiple data centers

Our final evaluation involved pods located in multiple data centers following Deutsche Telekom's topology as documented by the Internet Topology Zoo [114]. Each data center consisted of 4 pods interconnected via spine and edge switches as described in Facebook data center topology [112]. Each pod was set as its own domain for Cicero, while a single controller was used for the entire network (all data centers) for the centralized approach. We evaluated the completion time of web server flows taking into account their locality as reported by Facebook [1]: 15.7% traverse pods within the same data center and 15.9% traverse data centers.

The results depicted in Figure 3.14d show that the centralized controller suffers from the increased latency for establishment of flows across data centers. However, Cicero does not suffer from this increased latency thanks to domain parallelism and hence performs better than the centralized approach, unlike the single-domain setup, while being much more secure. These results exhibit the benefits of parallelism even under the web server workload (with 15.7%+15.9% crossing flows) that has far less local events than the Hadoop one (3.3%+2.5%).

3.7 Conclusions

This chapter presents Cicero, a practical construction for secure and consistent network updates that exploits parallelism through dependency analysis and ensures scalability to large networks through update domains. Threshold cryptography and distributed key generation allows for flexibility in control plane membership with minimal switch instrumentation.

Through extensive analysis using a functional Facebook data center topology with characteristic workloads we show that Cicero can provide consistency and security with minimal overhead to flow completion time. Additional optimizations using controller aggregation reduce the load on data plane switches.

As future work, we plan to investigate evaluation with other workloads including topology discovery and link state probing. We also plan to integrate a distributed ledger in the control plane state, coupled with the atomic broadcast component, to help detect (potentially transient and malicious) controller failures thanks to the auditability of their decisions.

4. EFFICIENT AND CONSISTENT USER SPACE FILE ACCESS

This chapter models the interaction between applications and user space file systems (FSs). Traditionally, the only option for developers was to implement FSs via drivers within the operating system kernel. However, there exists a growing number of FSs, notably distributed FSs for the cloud, whose interfaces are implemented solely in user space to (i) isolate FS logic, (ii) take advantage of user space libraries, and/or(iii) for rapid FS prototyping. Common interfaces for implementing FSs in user space exist, but they do not guarantee POSIX compliance in all cases, or suffer from considerable performance penalties due to high amounts of wait context switches between kernel and user space processes.

This chapter discusses the design, implementation, and evaluation of DEFUSE: an interface for user space FSs that provides fast accesses while ensuring access correctness and requiring no modifications to applications. DEFUSE achieves significant performance improvements over existing user space FSs interfaces thanks to its novel design that drastically reduces the number of wait context switches for FSs accesses. Additionally, to ensure access correctness, DEFUSE maintains full POSIX compliance thanks to three novel concepts of bypassed file descriptor (FD) lookup, FD stashing, and user space paging.

4.1 Introduction

FSs provide a common *interface* for applications to access data. These interfaces provide an abstract, high-level representation of a *file*, and the FS driver provides the mechanism to translate this abstraction into input/output (I/O) operations sent to physical storage media.

4.1.1 Implementing new FSs

Traditionally FSs are implemented within the operating system (OS) *kernel* [115]–[117] and accessed via system calls defined in the interface. For new FSs, implementing drivers fitting kernel-space interfaces requires effort; porting libraries that exist only in user space [118]–

[120] may even be required. Despite this development effort, kernel-space implementations may be short-lived and removed once deemed obsolete [121].

User space FSs can provide significant benefits over kernel space implementation methods traditionally available to developers. E.g., in the event of a crash or security breach, having the FS driver isolated from the rest of the kernel reduces the risk of data corruption [122], [123]. In particular, microkernel-based OSs extend the rationale behind user space isolation to most services and as such employ user space FSs by default and to great avail [124], [125]. User space FSs can also enable rapid FS prototyping in industrial and academic research [126], [127].

Additionally, cloud FSs (e.g. GlusterFS [128], Databricks File System [129], Alluxio [130], Hadoop Distributed File System (HDFS) [120], Google Cloud Storage [131], Amazon S3 [132]) do not have kernel-space implementations, requiring applications to conform to user space Application Programming Interfaces (APIs). Other state-of-the-art distributed FS drivers [128], [133]–[138] are implemented in user space to benefit from libraries (e.g. Boost C++ library) or programming languages (e.g., Java) not available in kernel space. While command line tools for accessing some of these FSs exist [139]–[141], these tools are also FS-specific. Thus applications must conform to APIs of specific FSs which can be cumbersome when multiple FSs are accessed, or FSs are substituted.

4.1.2 Interface Requirements

A sensible solution for a user space FS interface must meet the needs in:

Flexibility: applications accessing kernel space FS implementations are not required to have knowledge of the underlying FS driver nor require modifications when using different FSs. A user space interface should not require more. This allows applications to access a diverse set of FSs without additional implementation efforts.

Efficiency: the overhead of the FS interface should be minimal to reach access speeds closest to kernel space.

Consistency: kernel-space implementations provide a consistent set of guarantees for files within the FS, e.g. locking semantics, permissions, inheritance of FDs between parent and child processes. In order to establish trust in the FS driver, any user space implementation must also provide the same set of guarantees.

Existing generic solutions fall short on at least one of these requirements. That is, there is no interface for user space FSs that provides flexibility, consistency, and efficiency *all together*.

4.1.3 State of the Art

State-of-the-art solutions rely on one of the following methods to provide user space FS interfaces: an FS-specific user space library, an `LD_PRELOAD`-loaded library, or FUSE.

User space FS libraries provide an interface for FSs through compile-time binding to the drivers' APIs. This method is neither flexible, as each API is unique and requires compile-time binding, nor consistent, as user space FS libraries do not conform to any standard for consistent file behavior.

The `LD_PRELOAD` [11] method relies on diverting system calls typically wrapped by a library (e.g. `libc` [142]) by pre-loading another library with the same interface with the goal of redefining system call wrappers. This method can suffer from incorrect behavior that lead to failures in distributed environments (e.g. data loss, cascading errors, outages [143]), especially with files whose FD are inherited (e.g. Spark [144] passes data via the FS between manager and forked workers). Due to the distributed nature of remote execution, such incorrect behavior can be difficult to detect.

Filesystem in Userspace (FUSE) [9] provides a common interface for an FS driver to be implemented by a user space server while still allowing applications to access FSs using kernel-space system calls. While FUSE can be used to access some of the cloud FSs mentioned above [145]–[150], FS accesses performed with FUSE suffer from major slowdowns inherent to its design [151].

4.1.4 DEFUSE

We present DEFUSE, a novel interface for user space FSs that combines the benefits of a kernel-space FS driver and a user-space library. DEFUSE offers significant performance improvements over FUSE while maintaining consistency of FDs unlike in aforementioned approaches [11]–[17], [152], [153], in particular when FDs are shared between parent and child processes.

DEFUSE achieves these characteristics with full POSIX compliance through three novel concepts:

1. *bypassed FD lookup*, combined with a user space library, reduces the number of wait context switchs and ensures that FDs are managed by the kernel thus improving performance and achieving correct FDs behavior;

2. *FD stashing* ensures continued correct behavior of FDs after the address space of a process is cleared following a common `fork`/`exec`;

3. *user space paging* further ensures correct behavior of memory-mapped files. Our experiments show DEFUSE provides up to 10× the performance of the state of the art (i.e. FUSE) for user space FSs.

4.2 Motivation

In this section, we discuss in more detail the requirements needed for practical user space FS interfaces and pinpoint how the main existing approaches fail to satisfy these requirements.

4.2.1 User Space FS Interface Requirements

Flexibility – FS-agnostic accesses.

FSs implemented as a user space library require applications to bind to the library's API at compile-time in order to access the FS. Through calls to the API, the FS library performs the necessary I/O operation; it typically involves a system call to a kernel space driver, that backs the FS, which result is returned to the application. Execution remains within user space except for system calls performed by the FS library. Command line tools such as `cp` or `find` may require access to multiple FSs through a common interface and cannot be easily re-compiled with each user space file implementation. As a result FS interfaces must be standardized.

Efficiency – low access overhead.

Te ensure that as much system resources as possible are allocated to useful computations, the overhead of the OS interface should remain as low as possible. FS access is no exception to this hence any user space FS interface should require minimal overhead to provide FS accesses with speeds matching those of a kernel-space FS implementation.

Conceptually, the time to perform an I/O system call can be divided into two parts:

1. the time to service the system call (either through the kernel, the user space library, and/or the FUSE server) and
2. the time to perform the I/O operation to the storage media.

In some cases the overhead can be dwarfed by long I/O times to storage media. However, as the speed of storage media continues to improve (e.g. solid state drives, non-volatile memory), the system call service time, and thus the overhead of FS interfaces, becomes more prominent.

Consistency – POSIX compliance.

POSIX provides guarantees to applications that are consistent across OSs. These guarantees allow applications to execute without intricate knowledge of the underlying system implementation.

In addition, thanks to its standard interface, POSIX also guarantees the consistent behavior of FDs for FSs. These include, among others, protection from file corruption when multiple threads access a file concurrently, and valid FDs inheritance from parent processes after a `fork` operation.

Table 4.1. Summary of pros and cons of user space FS interfaces.

FS interface	Flexibility	Efficiency	Consistency
User space FS library	×	✓	×
`LD_PRELOAD` library	✓	✓	×
FUSE	✓	×	✓
DEFUSE	✓	✓	✓

4.2.2 Background

We identify user space FS libraries, `LD_PRELOAD`-loaded (user space) libraries, and FUSE as the three main methods used to provide user space FS interfaces. Table 4.1 summarizes how these methods comply with the aforementioned requirements.

User space FS library.

With a user space FS library, the FS driver is implemented in a code library which the application binds to at compile-time. Examples of this include the MPI-IO library [154] used by MPI, the HDFS C API `libhdfs` [155], and the Amazon Web Services (AWS) software development kit (SDK) for C++ [156].

Flexibility (×): A user space FS library does not provide the flexibility of a kernel space system call. Since there is no standardized API for user space FSs, applications must have prior knowledge of FS function calls to perform I/O operations. Access to multiple FSs implemented as user space FS libraries requires conforming to each FS API. For example, `libhdfs` uses `hdfsOpenFile` to open a file and `hdfsCloseFile` to close it. AWS SDK uses different calls.

Efficiency (✓): Due to the direct binding to the user space FS library, calls to the FS tend to be efficient. The invocation of FS operations bypasses the kernel altogether.

Consistency (×): User space FS libraries do not necessarily provide consistency guarantees. Since the implementation uses its own API for FS access and file metadata is maintained within the library, it is not required to comply to a standardized interface as with FUSE or kernel-space (enforced through the Virtual File System (VFS)). Thus POSIX behavior is not guaranteed [157].

`LD_PRELOAD` library.

The research community has explored several methods to implement efficient interfaces for user space FSs using `LD_PRELOAD` [11]–[17] or linkable libraries [152], [153] that provide access to FSs by intercepting system calls. An `LD_PRELOAD` (-loaded) library, being loaded before `libc`, causes its versions of the functions to be invoked in lieu of those of `libc`. Intercepted I/O calls are sent to the user space FS library via its API, keeping control flow within user space until the operation must be passed to the kernel. An application thus executes as if calls were made to the kernel. Accessing different FSs requires changing the `LD_PRELOAD` library.

Flexibility (✓): Similarly to FUSE and kernel space implementations, applications access FSs via system calls. Applications only need knowledge of existing system calls for FS access, not of the underlying FS driver. This flexibility is provided when multiple FSs are accessed, granted each FS has its own `LD_PRELOAD` library to intercept those calls.

Efficiency (✓): As system calls are intercepted and redirected in user space, the efficiency of the `LD_PRELOAD` approach is comparable to that of a user space FS library.

Consistency (×): The POSIX behavior of FDs and return codes is not guaranteed with the `LD_PRELOAD` scheme, especially for FDs inherited from parent to child processes (cf. § 4.3.2). Furthermore, the `LD_PRELOAD` library must invoke the user space FS library's API which does not necessarily comply with a standard for file metadata representation. Thus, to ensure POSIX behavior, `LD_PRELOAD` must maintain a mapping between the FS library's file metadata and FDs returned to the application. Maintenance of this mapping is trivial in most cases and can consist of a hash translating an FD, manufactured by the `LD_PRELOAD` library (e.g. a number) when it intercepts an `open` library wrapper call, to the metadata of the FS library. This table can be stored in the application's memory; if the table is lost, as with a call to `exec`, FDs are invalid in a child process.

Filesystem in Userspace (FUSE)

With FUSE, when a system call induces an I/O operation sent to the kernel, the FUSE kernel driver sends the I/O request to user space through a FUSE virtual device (`/dev/fuse`). The request is received by the FUSE server in user space and sent to the relevant FS library through an interface similar to that of the VFS [158]. The I/O operation is then processed by the user space server (and may include calls to kernel resources). Results are passed back through the FUSE device to the kernel driver, and finally to the application that issued the I/O operation.

Flexibility (✓): FUSE provides the benefit of FS-agnostic access for applications through system calls without prior knowledge of the underlying FS.

Efficiency (×): The transfer of control between kernel and user space when servicing a system call — a core design decision behind FUSE — requires costly context switching and buffer copying between processes of different spaces. This leads to serious performance penalties (cf. § 4.3.1) which hinders deployments in production environments, centering FUSE's use to FS prototyping [159].

Consistency (✓): While I/O requests are processed by the FUSE server in user space, as the application accesses the FS through system calls, POSIX compliance is enforced by FUSE's kernel driver. FUSE manages a mapping in user space from `inode` numbers maintained by the kernel to FUSE file information passed to the FS implementation.

4.3 Challenges

This section elaborates on challenges in achieving a user space FS interface that reconciles direct mount's efficiency with FUSE's flexibility and consistency.

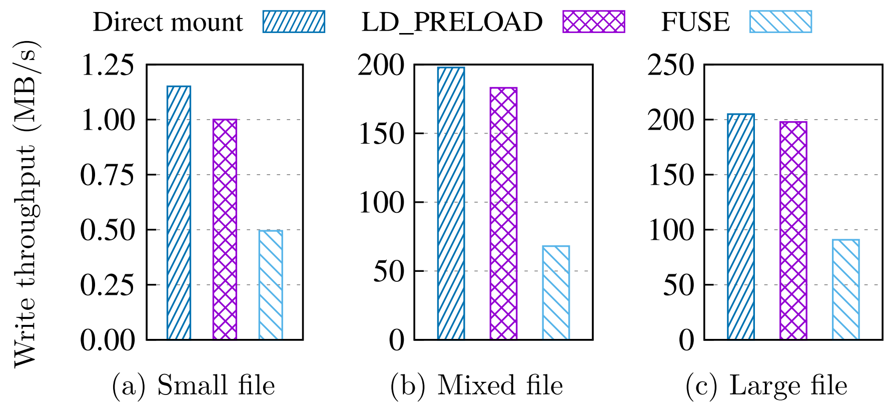


Figure 4.1. Write throughput of a direct mounted `ext4` FS, an `LD_PRELOAD` library and a FUSE wrapper around `ext4` (higher is better). FUSE's throughput ranges between 34% and 44% that of direct mount.

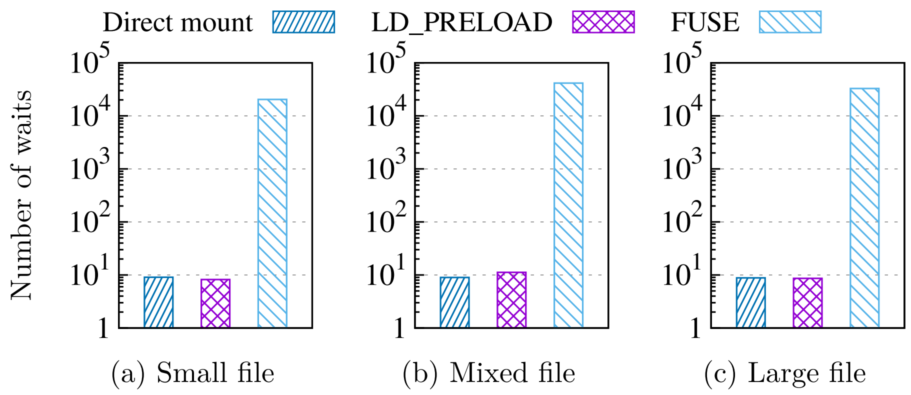


Figure 4.2. Number of waits for the benchmarks used in Figure 4.1 (log scale, smaller is better). FUSE causes tens of thousands of waits while direct mount and `LD_PRELOAD` only ≈9.

4.3.1 Efficiency

While FUSE is both flexible and consistent, it suffers from poor performance due to a high number of wait context switchs (waits), i.e. voluntary context switches used to wait for the completion of an I/O operation [160]. We demonstrate this overhead in Figure 4.1 and Figure 4.2 that depict write throughput and number of waits, respectively. We compared three interfaces: (i) direct access to an `ext4` FS using POSIX system calls to the kernel-space `ext4` driver, (ii) an `LD_PRELOAD` library storing its own FD table, and (iii) a FUSE server that simply wraps over POSIX calls (`fusexmp` [161]). Interfaces were tested with three write-based workloads: (a) small file – 4,096 writes of 128 B to different files, (b) mixed file

– 2,048 writes of size linearly distributed between 64 B and 128 kB to different files, and (c) large file – 1 write of 128 MB to a single file.

FUSE's throughput, shown in cf. Figure 4.1, is visibly worse than the kernel-space baseline in all workloads: 43% and 44% of the direct mount throughput for the small file and large file ones, respectively, and even 34% for the mixed file one. The `LD_PRELOAD` library performs much better, especially for larger operations, with throughputs of 87%, 93%, and 97% of direct mount for small, mixed, and large file workloads, respectively. Figure 4.2 further shows that FUSE's throughput drop is mostly due to the considerable amount of waits, with numbers as high as ≈20,000, ≈41,000, and ≈33,000 for small, mixed, and large file workloads, respectively. The `LD_PRELOAD` library however is on par with the direct mount solution, causing less than 10 context switches on average.

Specifically, an I/O wait entails a context switch between kernel and user space FUSE server which includes (1) saving processor state, (2) changing processor mode, and (3) copying data between kernel and user space buffers [151]. While (1) and (2) induce little overhead, (3) is costly.

4.3.2 Consistency

The `LD_PRELOAD` library interface seems attractive in light of the results presented in § 4.3.1. However, this interface as well as user space FS libraries are subject to two challenges related to FS consistency: the first concerns FD heritage and the second memory-mapped (user space) files.

FD heritage dilemma

As explained in § 4.2.2, when an `LD_PRELOAD` library is used, the mapping table between FDs and user space file metadata may be stored in memory. However, while open FDs and the internal state of the FS maintained by the `LD_PRELOAD` library are copied from parent to child upon a `fork`, the FS state is destroyed when the child invokes `exec` if it is in memory.

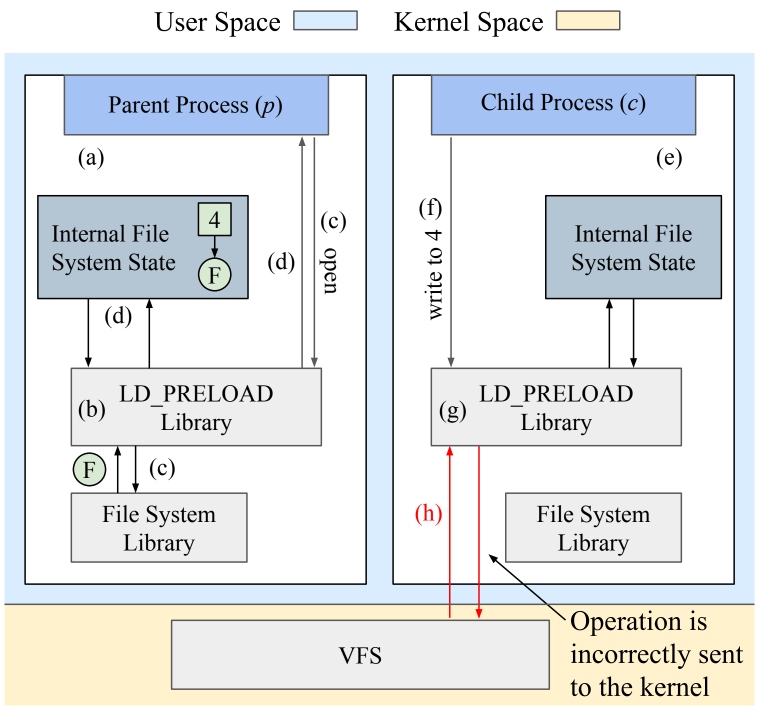


Figure 4.3. Illustration of an error caused by FDs inherited between parent and child with the `LD_PRELOAD` library.

4.3.3 Illustration

Consider the processes in Figure 4.3:

(a) User process p, in the `LD_PRELOAD` environment, executes library function wrapped system calls that are intercepted by the `LD_PRELOAD` library.

(b) Both the `LD_PRELOAD` and FS libraries are stored in shared memory and the internal state of the FS resides within the address space of p.

(c) When p performs an `open` system call, the `LD_PRELOAD` library makes the corresponding call to the FS library. The FS returns an internal representation of the file as F. Process p expects an FD number to be returned from `open`, not the internal representation of F.

(d) To avoid passing the `open` call to the kernel (to not increase latency), the `LD_PRELOAD` library manufactures FD 4, updates its internal file mapping 4 to F, and returns the manufactured FD to p. This manufactured FD is only valid in the context of the `LD_PRELOAD` library and only if the mapping is in memory. Future system calls using FD 4 will be intercepted by the `LD_PRELOAD` library, mapped to F and passed to the FS library.

(e) Process p creates child process c which inherits, among other things, the open FDs and the memory of the `LD_PRELOAD` library. After performing an `exec`, the child's address space, including the internal state for manufactured FDs, is destroyed by the executable of c.

(f) Process c later writes to the inherited FD 4.

(g) As the `LD_PRELOAD` library memory was cleared during `exec`, the library cannot map FD 4 to F, so it passes this I/O to the kernel.

(h) The kernel does not know what FD 4 references since the `LD_PRELOAD` library of parent process p manufactured it. Ultimately, the operation fails in the kernel with an invalid FD and is returned to c via the `LD_PRELOAD` library.

4.3.4 Prominent examples

The FD heritage dilemma occurs commonly in shell file redirection. Consider a shell running under the context of an `LD_PRELOAD` library, where the user executes a cloud application running in Spark [144] by running `spark-submit ... > out.txt` to capture the output to a file located in a user space FS. The shell opens `out.txt` which is intercepted by the `LD_PRELOAD` library prior to creating the `spark-submit` process. Due to the FD heritage dilemma, the FD for the output file `out.txt` inherited by the child is no longer valid after the `exec` call to invoke `spark-submit`. The FD heritage dilemma is also common in widely used tools such as gcc that create child processes for performing subtasks, and in resource manager frameworks such as the PySpark workload daemon [162], which creates child processes for worker tasks at runtime, with each child task sending and receiving data to and from the daemon through FDs.

4.3.5 Impact in other interfaces

The user space FS library interface suffers from a related problem if a parent process needs to pass file metadata to a child. The application must be aware of the FS library API and does not access the FS using FDs, it instead uses the user space file metadata directly. As with open FDs, child processes cannot inherit file metadata.

FUSE is not affected by the FD heritage dilemma since the FUSE server is a single process, separated from the application processes. Hence the FUSE identifier table is unique. When a parent opens a file, the corresponding (inheritable) FD matches a unique inode within kernel space.

Memory-mapped files

Mapping files into memory using `mmap` is a common method to share memory between processes. The `LD_PRELOAD` interface cannot provide consistent memory-mapping of files hosted by user space FSs for the two following reasons.

First, while an `LD_PRELOAD` library can intercept explicitly invoked system calls such as `mmap`, it cannot intercept implicit I/O operations. For instance, accesses to a memory-mapped file such as reads and writes are performed using `load` and `store` paging instructions to the address region of the mapped memory. These instructions, which may result in an I/O operation to the underlying file, are not explicit system calls and therefore cannot be intercepted.

Second, as a direct consequence of the FD heritage dilemma, a child process cannot access a user space file mapped in memory if the mapping has been done by its parent process. For example, consider the following set of actions taken by a process:

(a) A process opens a user space file, the `open` call is intercepted and redirected to the user space FS library.

(b) The process then maps this file into its address space using `mmap` for shared access, however since the FD is manufactured by the intercepting FS library, the call to `mmap` fails because the FD is invalid.

4.3.6 Impact in other interfaces

Directly linked user space FS libraries equally suffer from the same problem for `mmap`-ed files as `LD_PRELOAD` libraries do. FUSE does not, however, since the FUSE kernel driver translates paging operations to `read` or `write` sent to the user space FUSE server.

4.4 DEFUSE Design

We present the design of DEFUSE centering on its novel concepts and the features it enables.

4.4.1 Overview

DEFUSE uses a unique four-fold approach to be the first solution that achieves flexibility, efficiency, and consistency all together (cf. Table 4.1). As shown in Figure 4.4, our approach comprises:

1. a hook into `libc` to intercept and forward FS access calls to a user space FS library — for flexibility and efficiency (§ 4.4.2);

2. an FS kernel driver providing *bypassed FD lookup* semantics to ensure FDs are managed by the kernel and remain correct when inherited by child processes — for consistency and efficiency (§ 4.4.3);

3. a shared memory space for *FD stashing*, allowing FS metadata to be restored after a process's address space is cleared by an `exec` call — for consistency (§ 4.4.4);

4. a memory management framework for user space paging to correctly handle memory mapped pages backed by user space files — for consistency and efficiency (§ 4.4.5).

We demonstrate how DEFUSE's design maintains POSIX compliance (§ 4.4.6), why it provides better fault tolerance than FUSE (§ 4.4.7) and how it delegates caching to the respective user space FS libraries to ensure cache consistency (§ 4.4.8).

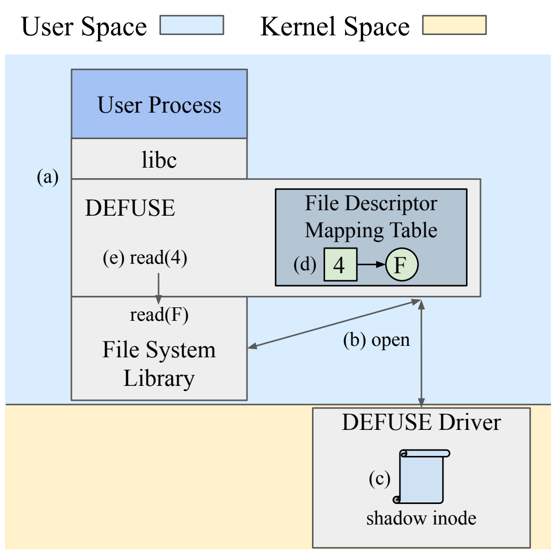


Figure 4.4. Handling of I/O requests in DEFUSE. I/O requests are intercepted by DEFUSE's `libc` extension and directed to the user space FS library. Requests for `open` are sent to the DEFUSE kernel driver to allocate a valid `inode` and FD. FDs are temporarily stashed when an `exec` is intercepted (cf. Figure 4.5).

4.4.2 System Call Redirection

Performance gains of DEFUSE come from *redirecting I/O operations directly to the user space FS library* through a hook in `libc`, thus avoiding waits presented in § 4.3.1. Such hooks are used in other systems for language extension [163] or behavior customization [164]. DEFUSE implements these hooks by directly modifying the system call wrapper functions within the `libc`. This has a two-fold advantage over an `LD_PRELOAD` library (cf. § 4.2.2):

1. using `LD_PRELOAD` requires user environment setup to ensure that the library is correctly loaded before `libc`;
2. it is not possible for `LD_PRELOAD` to intercept system calls for applications that use static binding to `libc`.

4.4.3 Bypassed FD Lookup

Every `open` system call received by FUSE involves multiple `lookup` requests, one per directory in the file path, to ensure the file path is valid and access permissions are sufficient. These requests are sent by the FUSE kernel driver to the user space FS library and thus require waits. Opening a file deep in the directory tree with FUSE can be costly because of this series of `lookup`s.

To avoid the many transfers between kernel and user space, DEFUSE makes use of its own kernel driver to provide *bypassed FD lookup*. Thanks to bypassed FD lookup, the kernel does not need to send the `lookup` requests to a server in user space when it opens a file. Instead, the kernel driver creates and splices a shadow `inode` in the directory cache of the FS, then allocates an FD for the file, and finally allows the `open` to proceed to the user space FS library. In turn, file permission checks are performed by the user space FS library and enforced by the DEFUSE `libc` extension (cf. § 4.4.2). The shadow `inode` is removed by the kernel when all references to its file are removed.

Figure 4.4 depicts the management of I/O requests, and in particular `open` requests, by DEFUSE:

(a) An I/O system call is generated by the application. The system call is invoked as a library call to `libc`, which is intercepted by DEFUSE as explained in § 4.4.2.

(b) For `open` system calls, the operation is sent to the FS library to update the internal state of the FS as well as retrieve a reference to the FS metadata. The `open` call is also sent to the kernel to allocate a valid FD.

(c) The DEFUSE kernel driver allocates a shadow `inode`, initializes the FD entry for the file, and returns the FD to the user application. Once an FD is created by the DEFUSE kernel driver, it serves as index of the FD mapping table managed by DEFUSE to retrieve the matching user space FS metadata.

(d) DEFUSE then maps the returned FD to the FS metadata returned from the FS library.

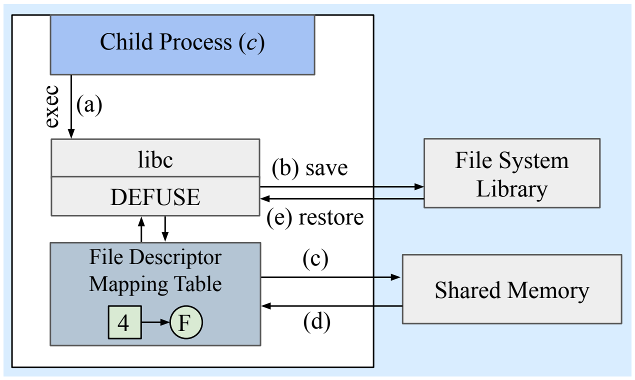


Figure 4.5. Depiction of how DEFUSE's FD stashing uses shared memory to save and restore the internal state of the DEFUSE FD mapping table before and after the invocation of an `exec` system call.

(e) When the application requests access to the file (e.g. `read`), the request is again sent to `libc` and intercepted by DEFUSE. DEFUSE then uses the FD mapping table to retrieve the user space FS metadata and sends the request directly to the user space FS library through its API.

DEFUSE uses a shadow `inode` instead of an existing file (e.g. `/dev/null`) for bypassed FD lookup to allow the file path to be tied to the file so existing FS features function as expected (e.g. file position pointers, `fcntl` locking, file data in `/proc`). DEFUSE complies with POSIX (cf. § 4.4.6).

4.4.4 Managing FDs across `exec` with FD Stashing

As described in § 4.3.2, once a `fork` system call completes, the memory of the child process is a duplicate of its parent. Subsequently, if the child process executes an `exec` system call, its memory is replaced with the memory of the program to be executed. While bypassed FD lookup ensures that FDs are correctly inherited by a child process, it does not prevent FDs from being erased alongside the rest of the process memory upon `exec`'s execution. Without additional management, future accesses by the child process to its FDs will fail.

To ensure correct FD semantics after a call to `exec`, DEFUSE utilizes a novel concept we call *FD stashing* that temporarily saves, then restores, FDs during the execution of an `exec` system call. As depicted in Figure 4.5, FD stashing operates as follows:

(a) The `exec` system call is intercepted by the DEFUSE `libc` extension.

(b) The `save` routine in the user space FS library is invoked to allow the FS library to perform any necessary action to save file meta data.

(c) The internal state of the DEFUSE FD mapping table is saved to shared memory.

(d) After the `exec` system call, and the memory of the process erased, if the process executes an I/O system call, the FD mapping table is restored.

(e) Further, the `restore` routine in the FS library is invoked to allow the user space library to restore the saved state of the file.

Subsequent I/O calls continue to be intercepted and sent directly to the user space FS library.

4.4.5 Managing `mmap` and User Space Paging

As described in § 4.3.5, memory-mapped files are accessed via I/O operations that rely on the `load` and `store` instructions instead of system calls like `read` and `write`. These operations cannot be intercepted by the DEFUSE `libc` extension as presented in § 4.4.2.

However, the `userfaultfd` interface for managing page faults in user space [165] has been added in Linux kernel 4.3. Originally implemented to help Linux-based hypervisors handle virtual machine migration, it allows a user space application to register a virtual address range with the kernel and subsequently be sent an event when a page within that range needs to be copied into memory. Events are received by reading from a special FD created by `userfaultfd` via a system call. Upon receiving an event, page data is copied into memory using the `ioctl` system call. To inform the page fault manager on virtual address space layout changes, `userfaultfd` also supports calls to `fork` and events for memory remapping (`mremap`), unmapping (`munmap`), removal (as result of a `madvise` call using `MADV_REMOVE` and/or `MADV_DONTNEED`).

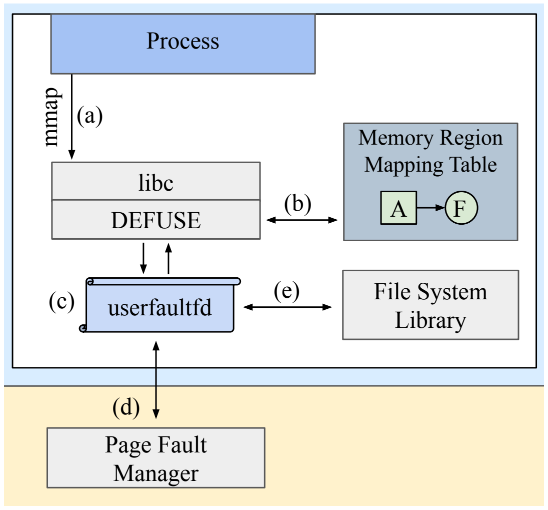


Figure 4.6. Depiction of how DEFUSE utilizes `userfaultfd` and a memory mapping table for user space paging.

DEFUSE takes advantage of `userfaultfd` to intercept and manage memory-mapped file I/O. As depicted in Figure 4.6, DEFUSE's user space paging operates as follows:

(a) The `mmap` system call is intercepted by the DEFUSE `libc` extension.

(b) The mapped address and corresponding FD are stored in a memory region mapping table.

(c) A `userfaultfd` is created to allow the kernel to signal to DEFUSE when a page fault within the address space occurs.

(d) When DEFUSE receives such a signal from the kernel, DEFUSE uses the memory region mapping table to determine which virtual address ranges (for which a fault was emitted) correspond to which FDs along with the offset within the file.

(e) DEFUSE then sends a `read` directly to the user space FS library copying the resulting data into the virtual address space of the process using `ioctl`.

When an address range is unmapped via `munmap`, DEFUSE intercepts the call, ensures the pages are flushed to the user space file, removes the entry from the memory region mapping table, and unregisters the `userfaultfd`. DEFUSE makes use of the `userfaultfd` events for remapped addresses (remap and fork) to ensure that the address space mapping table is kept up to date as the process's virtual address map changes.

4.4.6 Full POSIX Compliance Maintained

Enforcing file permissions

File permissions are used to properly isolate applications. The kernel is responsible for checking that applications have sufficient permission when opening a file so that later read/write operations succeed. DEFUSE enforces access permissions by invoking the `access` function provided by the user space FS library upon intercepting an `open` system call. If access is not allowed DEFUSE prevents the intercepted `open` system call from completing successfully.

File position pointers

Internal file position pointers, which keep track of read/write offsets, must be consistent between parent and child processes. Consider a parent process opening a file, reading some amount of data from the file (advancing the internal file position pointer), then creating a child process passing it this opened FD. The child inherits the opened FD, along with the file position pointer, and then reads from the file expecting the data to be read at the inherited position. Since DEFUSE utilizes a shadow `inode` created by the DEFUSE kernel driver, the internal file position pointer is managed by the kernel to ensure consistency between parent and child processes. As DEFUSE intercepts all read and write operations, redirecting them to the user space FS library, the file position pointer is maintained in the kernel by using the `lseek` system call.

Parallel file access

For parallel file access, read and write operations to files must be performed atomically. For example, if one thread performs a file read, DEFUSE adjusts the file position pointer using `lseek`. However, if another thread performs a write to the same file at the same time, DEFUSE also adjusts the file position pointer using `lseek`. To ensure that there is no contention for the file data and position pointer, DEFUSE uses `fnctl` to lock the file with each I/O operation. This lock is maintained by the DEFUSE kernel driver through the shadow `inode`. DEFUSE similarly ensures atomicity of further operations capable of generating race conditions (e.g., `dup2`, `dup3`).

Paging

POSIX behavior requires that (1) `mmap` succeeds for a valid FD, and that (2) if the file is mapped for shared memory, then the mapping should also be valid within a child process created with `fork`. DEFUSE complies with this requirement thanks to user space paging and FD stashing.

Accessing multiple FSs

In a kernel space implementation, FSs are mounted in the directory tree. This mounting allows the kernel to send I/O operations to the appropriate FS driver. Similarly, the DEFUSE kernel module also presents a FS to the system to perform bypassed FD lookup. Each user space FS must be independently associated with a DEFUSE mount. DEFUSE uses this mount to determine which user space FS library to send system calls to. Details are discussed in § 4.5.3.

4.4.7 Fault Tolerance

Failures in an FS can result in loss of access or even total loss of the data itself if it is not written to permanent storage. Being contained within the running process, DEFUSE requires no centralized server process. DEFUSE duplicates the FS logic to each process accessing the FS and a failure in the process thus does not affect other processes using DEFUSE. In contrast, FUSE relies on a single server process running in user space that may crash (due to, e.g., program error, explicit termination by a user). Upon FUSE server failure, accesses to the FUSE FS fail for all processes in the system.

4.4.8 Cache Delegation

DEFUSE does not manage a cache. Instead, DEFUSE delegates caching to the user space FS libraries it relies on. While FUSE uses a single server to avoid cache inconsistencies (e.g. duplicate entries) across all FSs, DEFUSE's delegation resembles an approach using one server per process.

4.5 DEFUSE Implementation and Semantics

We present DEFUSE's implementation, the user space FSs it already supports, and how to deploy it.

4.5.1 Code Base and Interface

The DEFUSE driver is a Linux kernel module made up of ≈300 source lines of code (SLOC) and the core DEFUSE library of ≈2,000 SLOC. Integrating a user space FS in DEFUSE requires implementation of ≈25 functions that are similar to the interface required by FUSE. Prototypes for these functions are provided in Listing 4.1 and Listing 4.2. Each function is required to return an error value as described by the system error numbers (i.e. `errno`). Function output is specified by output parameters (using references). FS integration only requires a shared library that exports the needed interface functions. DEFUSE loads the library (dynamically) as needed when a user space FS is accessed.

4.5.2 User Space FS Integrations

To test ease of implementation, we created a user space library compatible with DEFUSE for several FSs. These FSs are listed in Table 4.2. The direct wrapper FS wraps the I/O operations directly to their corresponding system call. The corresponding implementation for FUSE is the pass-through function of `fusexmp` [161]. A Virtual File System (AVFS) [166] and CRUISE [167] both provide a user space library that contains the function for all FS operations. Both the FUSE and DEFUSE implementations are able to pass calls directly to those libraries. SSHFS and HDFS consist of ports from an existing FUSE implementa-

Listing 4.1 DEFUSE interface for FS and file operations; similar to that of FUSE.

```
/*************************/
/* File system operations */
/*************************/
// Initialize the FS mounted at specified mount point
//    (called after FS lib is loaded)
void init(const char* mount_point, const char* backend_path);
// Un-initialize FS (called before FS lib is unloaded)
void finalize();

/*************************/
/* File operations       */
/*************************/
// Open file - out_fh is the file handle
//    used to later access the opened file
int open(const char* path, int flags, mode_t mode, uint64_t* out_fh);
// Close a file
int close(uint64_t fh);
// Read data from a file
int read(uint64_t fh, char* buf, size_t size,
    off_t offset, size_t* bytes_read);
// Write data to a file
int write(uint64_t fh, const char* buf, size_t size,
    off_t offset, size_t* bytes_written);
// Synchronize state and store of file
int fsync(uint64_t fh, int data_sync);
// Truncate an opened file to given length
int ftruncate(uint64_t fh, off_t length);
// Retrieve an opened file's attributes
int fgetattr(uint64_t fh, struct stat* stbuf);
// Save file handle data (before FD stashing)
int save(uint64_t fh);
// Restore (FD stash, then) file handle data
int restore(uint64_t fh);
// Retrieve file's attributes by path name
int getattr(const char* path, struct stat* stbuf, int flags);
// Truncate a file by path name
int trunc(const char* path, off_t length);
// Read a symbolic link
int readlink(const char* path, char* buf,
    size_t bufsize, size_t* bytes_written);
// Remove a file by path name
int unlink(const char* path);
```

Listing 4.2 DEFUSE interface for directory and file path operations; similar to that of FUSE.

```
/*************************/
/* Directory operations   */
/*************************/
// Create a directory
int mkdir(const char* path, mode_t mode);
// Remove a directory
int rmdir(const char* path);
// Get directory entry count
int getdirentcount(const char* path, int* count);
// Get directory entries
int getdirents(const char* path, DIR* dirent_buf,
    size_t bufsiz, size_t* ents_written);

/*************************/
/* File path operations   */
/*************************/
// Change ownership of a file/directory
int chown(const char* path, uid_t uid, gid_t, gid);
// Change permissions of a file/directory
int chmod(const char* path, mode_t mode);
// Check permissions of a file/directory
int access(const char* path, int mode);
// Rename a file/directory
int rename(const char* oldpath, const char* newp);
// Update access and modification time of a file/directory
int utime(const char* path, const struct utimbuf* times);
// Create symbolic link to a file/directory
int symlink(const char* target, const char* linkp);
```

tion each of which required about 150 total SLOC to adapt the FUSE implementation to DEFUSE. Using SLOC to evaluate integration effort, we conclude it takes relatively similar effort to integrate a user space FS in FUSE and DEFUSE.

4.5.3 Deploying a DEFUSE-backed User Space FS

The DEFUSE kernel module parses mount parameters to know which user space FS library is associated to each mounted FS. Consider the following command to access an FS

Table 4.2. Integration effort of example user space FSs in DEFUSE and their FUSE equivalent.

User space FS	**Integration methodology**	**DEFUSE SLOC**	**FUSE SLOC**
Direct wrapper	Simple wrapper	280	402
AVFS	Simple library wrapper	304	308
CRUISE	Simple library wrapper	274	503
SSHFS	FUSE port[a] (≈150 SLOC changed)	4,803	4,956
HDFS	FUSE port[b] (≈150 SLOC changed)	26,757	26,713

[a] Ported from `libfuse` SSHFS [119].
[b] Ported from `native-hdfs-fuse` [168]. This FUSE implementation does not use `libhdfs` [155] and is therefore not bound by the use of the Java virtual machine.

with DEFUSE:

```
mount -t defuse -o backend=/src/dst -o library=/usr/usfs.so defuse /mnt/fs
```

In this example the DEFUSE bypassed FD lookup FS is mounted at `/mnt/fs`, the user space library to redirect calls to is located in `/usr/usfs.so` while the back-end location for this FS is `/src/dst`. The library option must be a path in the FS (outside of DEFUSE) that contains the user space FS implementation. While the back-end is required by DEFUSE, it does not need to be a path; it is intended to serve as information for the user space FS implementation to determine how to access files and directories. It may be a path (as in the example), a file (e.g. an FS image), a network host name (e.g. for a network based user space FS), etc.

When an application makes its first access to the FS, DEFUSE retrieves the user space library path from the mount point and invokes `init` (cf. Listing 4.1) to initialize the FS before any other calls are processed. When the application completes, the `finalize` function is invoked to allow the user space FS to clean up any created data structures.

4.6 Evaluation

We evaluated DEFUSE against other FS interfaces with workloads generated following standard methods and real-life functional workloads. Overall DEFUSE's throughput significantly outperforms that of FUSE, in some cases achieving 10× speedups, while even our evaluations with distributed workloads involving communication over the network show that DEFUSE can still achieve 2× speedups over FUSE. Further analysis shows that speedups are a direct result of reduced wait context switch overhead, in some cases from 16,000 or 41,000 with FUSE to 9 with DEFUSE. DEFUSE also displays negligible runtime overhead for FD stashing (e.g. 8.9 µs for 1,024 FDs) and no runtime overhead for user space paging. Lastly, we show DEFUSE also performs up to 2× better than FUSE when accessing the user space AVFS.

4.6.1 File Access Throughput

We first evaluated DEFUSE on a single machine observing I/O throughput and induced number of waits using IOzone [169], and runtime of two throughput-intensive applications.

Setup

First we describe the setup of our file access throughput experiments.

Hardware

Benchmarks were performed on a single machine running Ubuntu 18.04.1 LTS with kernel version 4.15.0-43, two Intel® Xeon® E5-2420 processors with 2.2 GHz, 24 GB of main memory with SATA-attached 256 GB SSD and 1 TB HDD running at 7200 RPM.

Caching policy

FS caches were cleared between each I/O operation of the benchmarks to remove any effect of caching. Files were flushed to disk (`fsync`) after each write operation.

FSs

We used `ext4` [115], JFS [170], FAT [171], and `tmpfs` [172]. Both `ext4` and JFS were chosen due to their large install base in consumer Linux as well as industry servers. While FAT is no longer a commonplace FS for PCs or industry servers, it was chosen for evaluation due to its prevalence in external media. Finally, `tmpfs` was chosen for its low I/O overhead since it resides in memory, allowing us to isolate the FS logic overhead. Note that for FSs built on actual media, the time spent for I/O operations on the media itself may overshadow the time performing FS logic.

Baselines

We compared DEFUSE to (1) an `LD_PRELOAD` library, (2) Direct-FUSE [153], (3) FUSE (v2.9.7) with and without direct I/O enabled, and (4) an FS mounted using a kernel driver.

The `LD_PRELOAD` library maintains its own FD table (cf. § 4.2.2) that is thus invalid when inherited. Direct-FUSE is a user space FS interface built as an extension to the `libsysio` library [173]. Like DEFUSE, Direct-FUSE intercepts I/O system calls using hooks in `libsysio`. However, Direct-FUSE has similar drawbacks as the `LD_PRELOAD` approach, i.e. it does not address the FD heritage dilemma nor user space paging. `LD_PRELOAD` and Direct-FUSE are inconsistent solutions, they are included for comparison only. The user space FS library for DEFUSE and FUSE is a simple wrapper over POSIX system calls, as expressed in § 4.5.2. Using direct I/O with FUSE makes I/O operations skip kernel FS caches.

4.6.2 Benchmarking tool

We used the widely adopted IOzone [169] FS benchmarking tool — a user space application that creates, writes, and reads files of varying sizes using POSIX system calls. IOzone generated three workloads described further alongside the corresponding results in Figure 4.6.2.

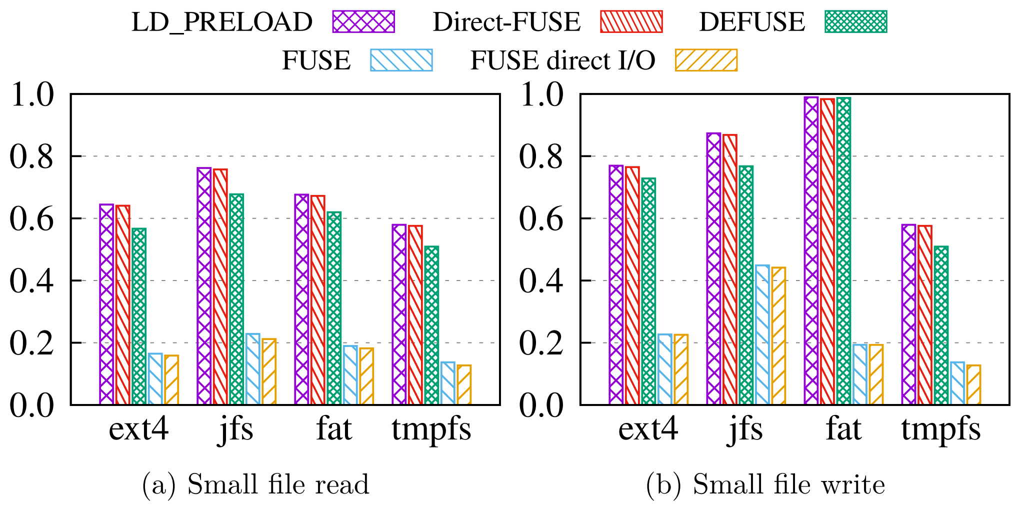


Figure 4.7. I/O throughputs of DEFUSE vs an `LD_PRELOAD` library, Direct-FUSE, and FUSE with and without direct I/O, normalized by the throughput of direct mounted kernel FS for small files (higher is better).

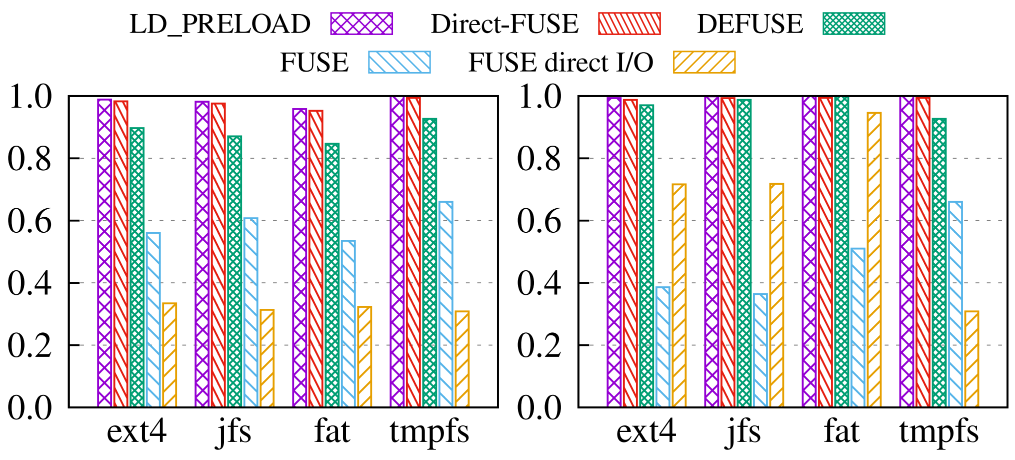
I/O throughputs were measured in terms of data read/written per second. The depicted results were normalized as a ratio of the direct mount to simplify comparisons (otherwise the greatly varying access performance for different FSs would render some figures' y-axis hardly readable).

Throughput results

We evaluated file access throughput separately for small and large file as well as for a mixture of files with different sizes.

4.6.3 Small files

We evaluated the FSs for small file accesses where a large quantity of system calls must be serviced. IOzone created 4,096 files of 128 B in size. This workload is prevalent in many parallel computing applications [10], [174] where large quantities (sometimes in the millions) of files smaller than 64 kB are used to store checkpoint data. Due to the large quantity of system calls needed to create, write, and read that many files, we expected FUSE's performance to be greatly affected as a large quantity of system calls requires an equally large quantity of waits.



(a) Large file read

(b) Large file write

Figure 4.8. I/O throughputs of DEFUSE vs an `LD_PRELOAD` library, Direct-FUSE, and FUSE with and without direct I/O, normalized by the throughput of direct mounted kernel FS for large files (higher is better).

Overall, as Figure 4.7a and Figure 4.7b show, DEFUSE always achieved within 10% of the best-performer `LD_PRELOAD` while the inconsistent Direct-FUSE was within 1%. DEFUSE reached at best equivalent throughput to direct mount (write with FAT) yet at worse 50% (read and write with `tmpfs`). The lower performance is primarily caused by the additional time spent resolving FS metadata from the FD mapping table, however it is only minor when compared to the overhead of FUSE. Direct I/O for FUSE has negligible effect on performance as the benefits are lost due to the small buffer sizes and the large number of system calls to be serviced (cf. discussion on Figure 4.10).

The direct mount actual throughput varied across FSs: (a) `ext4` had 2.6 MB/s write and 6.8 MB/s read throughputs, (b) JFS was slower with 1.1 MB/s write and 4.8 MB/s read throughputs, (c) FAT was also slower than `ext4` with 130 kB/s write and 6 MB/s read throughputs, (d) `tmpfs` I/O operations being completely in memory, achieved the best throughput with 20.4 MB/s for both write and read. However, given the large number of system calls needed to process all the files, achievable throughput is far below the capability of the storage media.

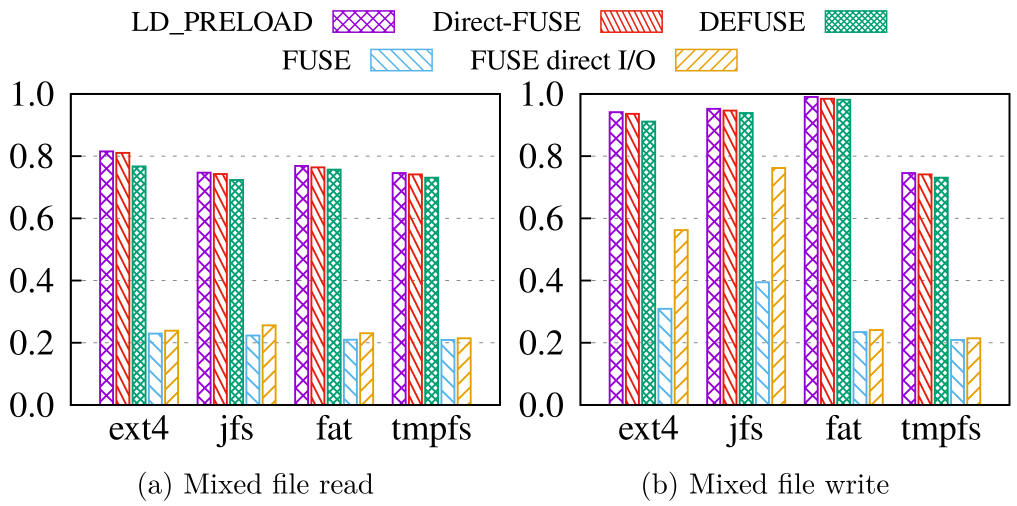


Figure 4.9. I/O throughputs of DEFUSE vs an `LD_PRELOAD` library, Direct-FUSE, and FUSE with and without direct I/O, normalized by the throughput of direct mounted kernel FS for mixed files (higher is better).

4.6.4 Large files

We evaluated the FSs for large file accesses where the fewest quantity of system calls were needed. A single 128 MB file was created. This benchmark highlights the amortization of wait context switch cost for FUSE's system calls.

The results in Figure 4.8a and Figure 4.8b show that DEFUSE reached close to optimal throughputs: between 85% and 93% that of direct mount for reads, and between 92% and 100% for writes. `LD_PRELOAD`'s throughput was almost always above 98% (except for reads on FAT) and, as with small files, Direct-FUSE closely followed. While FUSE performance improved with large files, it remained largely below that of the other interfaces and at best only reached 60% for `tmpfs`. Write performance, on the other hand, almost reached that of direct mount when used with direct I/O for FAT. Since only one file is accessed, the number of system calls serviced by the kernel and FUSE server is small and the total operation time is more dependent on the I/O time to the storage media.

The actual I/O throughput for direct mount is very similar for most FSs, achieving ≈200 MB/s write and ≈900 MB/s read throughput. The exception is `tmpfs` which achieved 2.3 GB/s write and read throughput due to I/O operations being completely in memory.

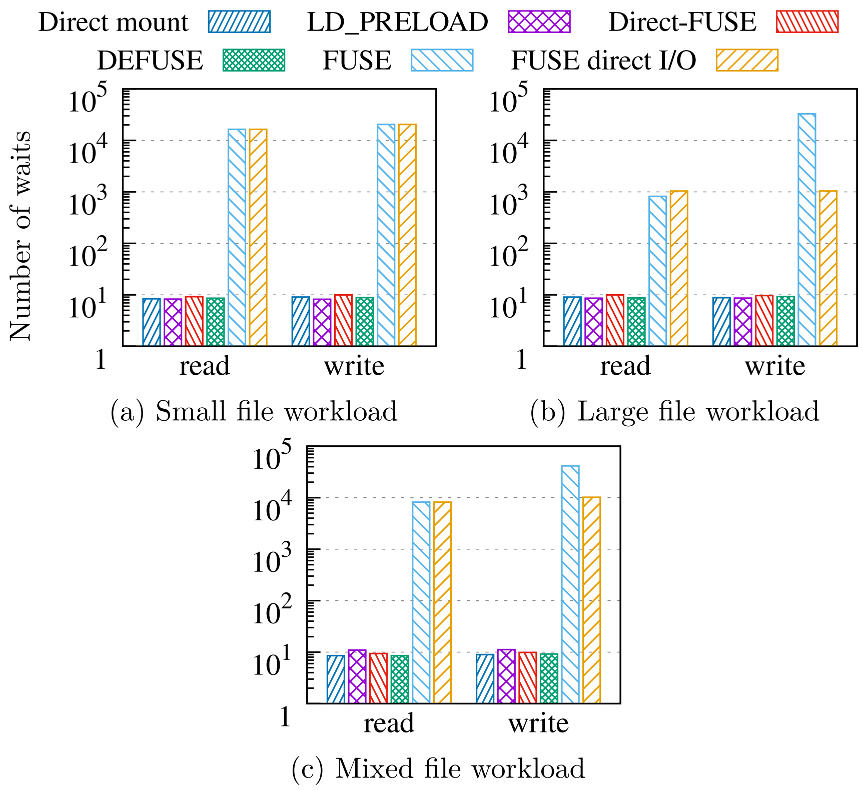


Figure 4.10. Number of waits observed (using the command `time %w`) for the `tmpfs` workloads of Figure 4.7, Figure 4.8, and Figure 4.9. (log scale, smaller is better). waits are context switches initiated to wait for the completion of an I/O operation [160]. Using `tmpfs` as backing store removes the I/O overhead inherent to physical media. For (a), thousands of system calls are made but these process-blocking system calls do not always induce a wait.

4.6.5 Mixed sized files

We evaluated the FSs using mixed sized files to simulate a more diverse workload. The benchmark created 2,048 files linearly distributed in sizes from 64 B to 128 kB. Figure 4.9a and Figure 4.9b show close results in all cases for all interfaces except FUSE that performs significantly worse. As with previous benchmarks, direct I/O slightly improves FUSE, but is still 15% slower than DEFUSE at best (writes to JFS).

Wait context switch results

Our evaluation with small, large, and mixed file workloads show that the throughput of DEFUSE was significantly higher than FUSE and was often close to that of `LD_PRELOAD`. We then evaluated the total number of waits needed to perform each of these benchmarks to confirm that the reduced throughput is directly related to context switching and not to any other bottleneck in FUSE. We thus used the Linux `time` command which has the ability to retrieve statistics including the number of context switches performed for both time slice expiration and the purpose of waiting for I/O operations to complete (using `time %w` [160]). Here, we are only concerned about the subset of context switches made by the process for the latter case, i.e. waits. We ran the benchmarks on `tmpfs` to focus on the context switches caused by the FS interface since I/O requests to access a physical device induce extra context switches.

The results shown in Figure 4.10 (y-axes use a log scale) confirm that the overhead of FUSE is caused in a large part by waits. Direct mount, the `LD_PRELOAD` library, Direct-FUSE and DEFUSE require ≈9 waits for all workloads while FUSE requires orders of magnitude more. For the small file workload (cf. Figure 4.10a), FUSE requires ≈16,000 waits (i.e. permission `lookup` + `open` + `read`/`write` + `close` for each of the 4,096 files) and using direct I/O does not reduce this number. For the large file workload (cf. Figure 4.10b), we note that the FUSE kernel driver splits large writes into multiple requests sent to the FUSE server in user space, each causing a wait. Even with a single (large) file, FUSE requires ≈1,000 waits for a read and ≈33,000 for a write. Direct I/O reduces these considerably, from ≈33,000 down to ≈1,000 for writes, which is still 100× higher than DEFUSE. The mixed file workload (cf. Figure 4.10c) shows an even larger difference in the required waits: ≈9 for DEFUSE but ≈41,000 for FUSE. While there are half as many files created in this workload compared to the small file one (2,048 vs 4,096), files are larger on average thus forcing the FUSE server to split write requests into smaller ones. As with the large file workload, the number of waits is reduced using direct I/O, down to ≈10,000, which is still 1,000× more than DEFUSE.

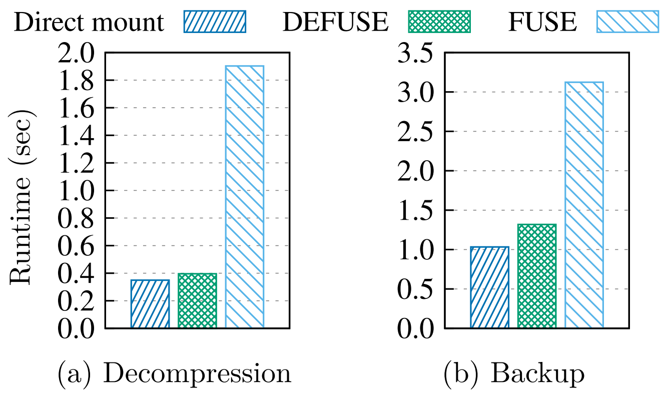


Figure 4.11. Runtime of two throughput-intensive applications (lower is better). (a) depicts time to decompress a 160 MB archive of 70,000 source files. (b) depicts time to back up the decompressed data set using `rsync`.

Application runtime

In addition to synthetic workloads generated by IOzone, we also evaluated DEFUSE's impact on two practical (throughput-intensive) applications. First, we decompressed a 160 MB data set of 70,000 small files (i.e. sources of the Linux kernel v4.13) into an `ext4` backed FS using the `tar` command. We then backed up the decompressed small files from the `ext4` backed FS to a JFS backed FS using `rsync`.

Figure 4.11a shows the decompression time and Figure 4.11b the backup time. For decompression, DEFUSE achieved almost identical runtime to the direct mount while FUSE required as much as 4× the amount of time to finish. Using direct I/O improved FUSE performance but only marginally. Similarly, DEFUSE ran at near direct kernel mount speeds when backing up, while FUSE and FUSE with direct I/O were significantly slower, taking ≈3× longer. This evaluation showed the practical advantages of DEFUSE over FUSE as a user space FS interface for everyday tasks.

4.6.6 Benefits for Distributed Systems

We also evaluated a diverse set of *distributed* cloud workloads to compare DEFUSE, FUSE, and direct kernel mount. We used Apache Spark [21] v2.4.0 on a cluster of 5 machines with identical hardware setups as described in § 4.6.1. These distributed workloads show the

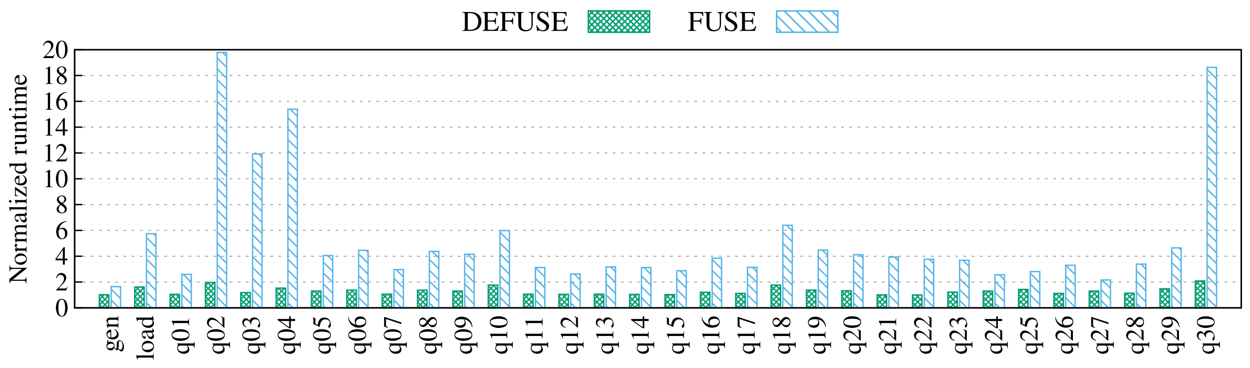


Figure 4.12. Runtime of TPCx-BB queries normalized by the runtime of direct mount (lower is better). `gen` is the time to generate the benchmark data, `load` the time to load that data into the meta store, `q`x the time to run query x. DEFUSE is between 1.6× (`gen`) and 10× (`q02`) faster than FUSE, and 3.7× on average.

local benefits of DEFUSE have important *global impact* on distributed applications despite network latency.

4.6.7 TPCx-BB express benchmark

In our first distributed evaluation, we ran the TPCx-BB benchmark suite [175]. The benchmark consists of 30 different SQL queries in the context of retail stores. Using the Spark SQL [176] implementation provided by the Transaction Processing Performance Council, we used a data scaling factor of 100 and used direct mount, DEFUSE, and FUSE as the interfaces of the HDFS backing store. The runtimes shown in Figure 4.12 are a normalized ratio of HDFS backed by a direct mount `ext4` FS. The entries for `gen` and `load` show the time to generate benchmark data and to load it into the metastore, respectively. Each `q`x shows the time to run query x. Queries `q`22 and `q`30 show the best and worst performance respectively for DEFUSE, while queries `q`27 and `q`02 show the same for FUSE. While the results greatly vary, DEFUSE always outperforms FUSE; DEFUSE is between 1.6× (`gen`) and 10× (`q02`) faster with a 3.7× gain on average for a query.

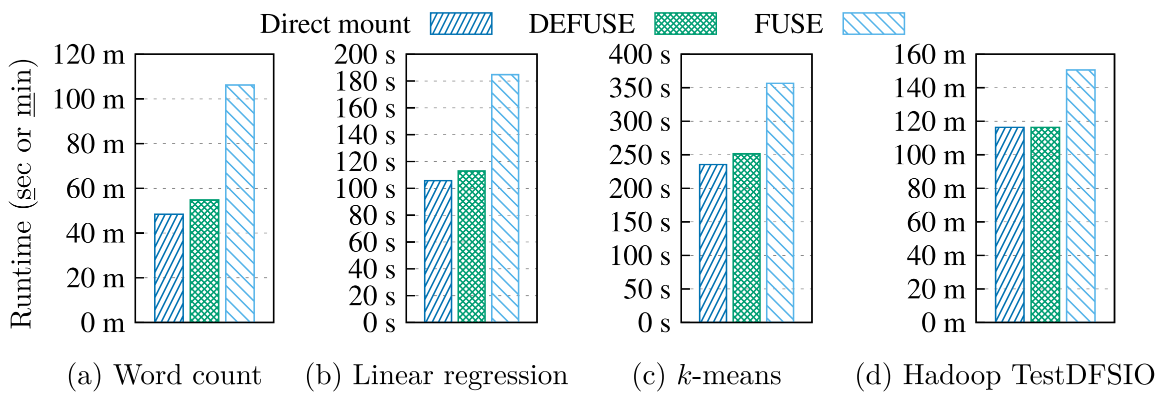


Figure 4.13. Runtime of typical cloud workloads (lower is better). (a) depicts the time to run a Spark word count job using 300 GB of Wikipedia data. (b) depicts the time to run a linear regression of 720 million observation data set. (c) depicts the time to run k-means clustering of 200 clusters of a 1.4 billion observation data set. (d) depicts the time to run the TestDFSIO benchmark writing 300 GB of data. DEFUSE is between 1.27× (d) and 1.94× (a) faster than FUSE for cloud workloads, and on par with direct mount for TestDFSIO (d).

4.6.8 Word count

To further evaluate DEFUSE against FUSE and a direct mounted FS, we ran a Spark word count workload using 300 GB of Wikipedia data. While Spark data inputs and results may be stored locally or in a remote file system, intermediate results from the map and reduce tasks are stored locally on compute nodes. Using the Purdue University MapReduce Benchmarks Suite (PUMA) [177] we show that the overall runtime of a Spark application can be affected by the FS interface of this local storage. Figure 4.13a shows that FUSE significantly increases the completion time of the word count job, even when used only for Spark local storage, with a 2× difference compared to DEFUSE or direct mount. Spark uses memory mapped I/O for reading and writing to local storage so it is not possible to evaluate FUSE with direct I/O enabled.

4.6.9 Machine learning

To evaluate machine learning related workloads, we ran both a linear regression in Spark with a 720 million observation data set (Figure 4.13b) and k-means with a 1.4 billion observation data set (Figure 4.13c) using the same Spark configuration used for word count. As with Spark word count, using FUSE seriously impacts the total workload time, requiring 1.75× longer for the linear regression workload to complete compared to DEFUSE or direct mount. FUSE's overhead for k-means is lower but is still significant as it takes 1.5× longer than DEFUSE to complete.

4.6.10 HDFS

To conclude our distributed evaluations, we ran the TestDFSIO benchmark to evaluate how HDFS is affected by FS interfaces. The benchmark recorded the runtime of writing a single 300 GB file, plotted in Figure 4.13d. HDFS uses in-memory data block caching, and performance is affected more by the speed of main memory than by the speed of the FS. However, even in this case, FUSE takes 1.23× longer to complete compared to direct mount or DEFUSE.

4.6.11 Implementation Microbenchmarks

Last but not least we perform several microbenchmarks to tease apart the savings of DEFUSE.

4.6.12 Overhead of FD stashing.

Recall that FD stashing saves and restores the internal FD map used by DEFUSE which adds additional overhead to the `exec` system call. In order to determine the overhead, we created a benchmark where a parent process opens files within DEFUSE and calls `fork` and `exec` to create a child process, then measured the time for FD stashing to save and restore FDs in the child process. Figure 4.14 shows the results of 50 executions with error bars being one standard deviation from the geometric mean. Intuitively, the FD stashing overhead is

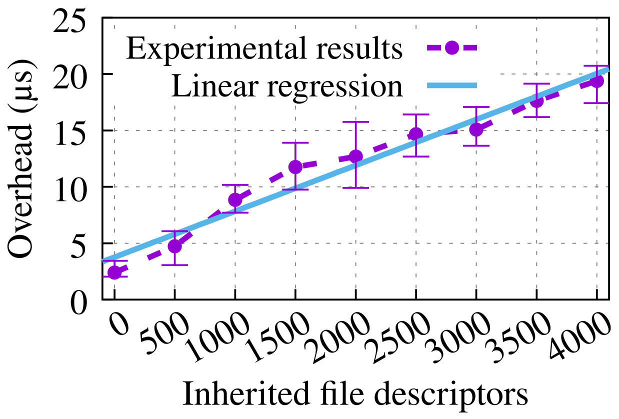


Figure 4.14. FD stashing overhead with DEFUSE for up to 4,000 inherited FDs along with the trend line. Overhead is negligible even for 4,000 FDs.

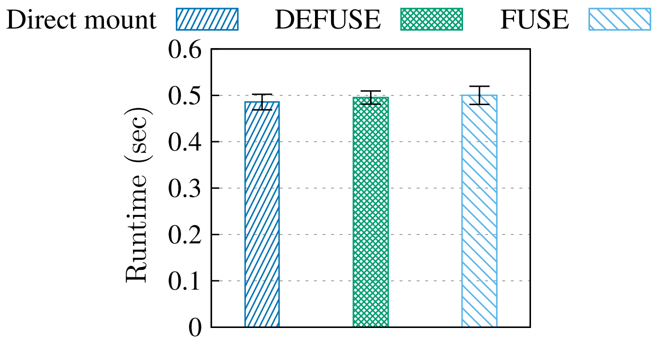
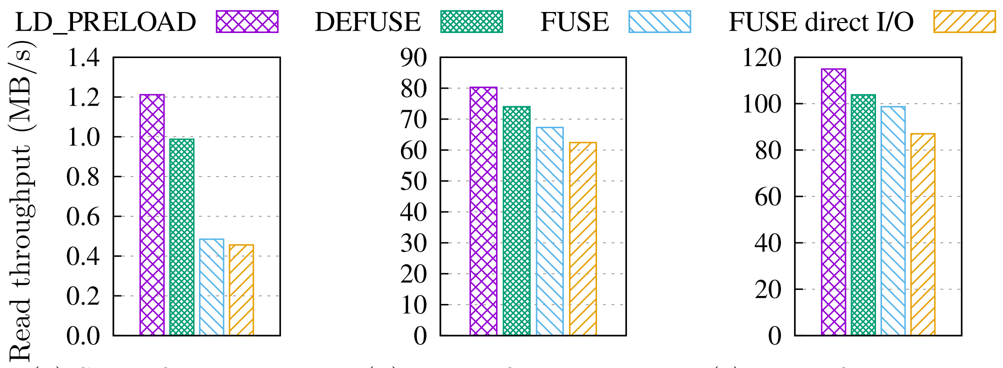


Figure 4.15. Runtime of 1.25 M reads on a file mapped with the user space paging of DEFUSE, direct mount FS and FUSE. DEFUSE and FUSE perform equally.

directly related to the number of inherited FDs as the FD stashing process sequentially saves and restores all FDs. Running a linear regression on the data results in an FD stashing overhead of 4 ns per inherited FD with a base overhead of 3.7 µs to intercept the system calls. Overall, the overhead is virtually nonexistent considering FD stashing is only required once per child process and the typical soft limit for open FDs on Ubuntu Linux is 1,024 (with a total overhead of 8.9 µs).

4.6.13 Overhead of user space paging

To evaluate the performance of DEFUSE user space paging, we evaluated the performance of a file mapped with DEFUSE against a direct mounted FS and FUSE. The comparison



(a) Small file workload (b) Mixed file workload (c) Large file workload

Figure 4.16. Read throughput of an `LD_PRELOAD` library, DEFUSE and FUSE when used with AVFS for small, large and mixed sized file workloads. DEFUSE always performs better than FUSE, with or without direct I/O, and particularly when handling a lot of small files as it reaches much closer to `LD_PRELOAD`'s performance.

does not include FUSE with direct I/O since it is not possible to use it with a memory mapped file — direct I/O bypasses the kernel page cache. We created a benchmark where a 128 MB file is mapped using the `mmap` system call and we then measured the runtime of 1.25 million random reads over the file's address space. Figure 4.15 shows averaged results over 50 executions with error bars of one standard deviation from the geometric mean. Since the performance of each FS is nearly identical, this experiment shows that user space paging used by DEFUSE is equivalent in performance to that of FUSE.

4.6.14 AVFS

To further compare `LD_PRELOAD`, DEFUSE, and FUSE with and without direct I/O enabled, we ran a series of tests on a user space FS implementation that did not have a corresponding kernel driver. Benchmark processes were run for the same small, large, and mixed sized file workloads (cf. § 4.6.1) using both DEFUSE and the FUSE implementation for AVFS [166] (cf. § 4.5.2), a user space only FS that allows direct access to compressed files bypassing the need for a decompression tool. A `gzip` compressed `tar` file was created containing the files of each workload. Figure 4.16 shows DEFUSE achieved higher read throughput for small files, almost double that of FUSE. Both large and mixed file workloads show less gains, but still clear improvements over FUSE.

4.7 Related Work

We discuss work related to user space FS interfaces and performance improvements to FUSE.

4.7.1 Related to User Space Libraries

MPI-IO [178] is developed as an extension to the Message Passing Interface (MPI) with the purpose of improving the performance of collective I/O within a parallel computing system. ROMIO [152] is a high performance user space implementation of MPI-IO. Its use of an abstract interface [179] allows multiple backend FSs to be connected to the library allowing parallel applications written to the MPI standard to access the underlying FSs without specific knowledge of FS API. Operating completely in user space, this solution provides the benefits of FS access speed, while the abstract interface allows for FS-agnostic access. However, it is intended for use with parallel computing systems and applications must be coded specifically to the MPI-IO standard. Applications using POSIX system calls would not be able to access the library.

Like MPI-IO, `libsysio` [173] is another library initially developed for managing access to FSs for high performance computing. However, as with other directly linked user space libraries, it is affected by the FD heritage dilemma nor does it provide user space paging.

4.7.2 Related to FUSE

Numerous solutions have improved FUSE but only few tackle the basic interface overhead largely due to wait context switch latency, without providing POSIX compliance.

Ishiguro et al. [180] propose a method to reduce memory copies and wait context switchs when FUSE is used with a remote FS. Their *direct device access* prevents unneeded calls from the FUSE kernel driver to user space when the backing FS contains a kernel driver.

However, this solution is not intended for FSs whose I/O request handling logic resides in user space.

ExtFUSE [181] proposes to reduce the number of FUSE requests between kernel space and user space using extended Berkeley packet filters (eBPF) [182]. The eBPF interface allows for user space programs to run in kernel space with significant restrictions. While ExtFUSE can reduce the number of wait context switchs, it is isolated to a subset of I/O operations.

Direct-FUSE [153] provides an abstract interface for user space FS libraries within `libsysio` [173]. As mentioned in § 4.6.1, it does not address the FD heritage dilemma, nor does provide an interface for user space paging, and therefore does not provide all the guarantees required by POSIX.

WrapFS [183] provides an efficient *wrapper* of a file system mount onto another mounted location. However it does not provide the bypassed FD lookup behavior needed to reduce the time to service an `open` system call.

Stacked FSs [184] promise ease in deployment of incremental changes to existing FSs. File System Translator (FiST) [185] is a tool aiming to ease creation of stacked FSs by generating the necessary kernel drivers for FS drivers written in a high-level language. However, both stacked FSs and FiST are limited to kernel space FS drivers. Narayan et. al. [186] extend FiST for user space implementations but in a way dependent on FUSE, thus suffering from wait context switch overhead.

Ganesha Network File System (NFS) [187] is a user space NFS client supporting the same protocols as kernel space implementation as in other Unix-like OSs. Yet running in user space allows Ganesha NFS to redirect I/O requests directly to user space FS libraries using an FS abstraction layer [188] similar to FUSE's. While increasing flexibility for NFS exports, it does not improve performance of interaction between kernel and user spaces; the socket interface used by the RPC protocol of NFS has similar performance penalties as the FUSE interface used to communicate between kernel and user space.

Steere et al. [189] propose a caching mechanism running in user space as an optimization to the Coda distributed FS [190]. While the cache improves performance for I/O operations, the system still requires a transfer of control from kernel to user space for *each* directory tree level during lookup. Even with the existence of a user space cache, workloads using input data sets requiring access to potentially millions of small files [10], [174] would take a large number of control transfers to fill the cache, yielding performance comparable to FUSE.

Patlasov [191] sketches a set of performance optimizations for FUSE aiming at parellelization thanks to, e.g. multi-threading, direct I/O, and caching. However, wait context switchs are not reduced, the optimizations are only validated using specialized parallel workloads on specialized iSCSI SAN storage rather than on common FUSE, and no codebase is public.

Re-FUSE [192] (which is not ReFUSE [193]) is an extension to FUSE to make the user space FUSE server fault-tolerant. A Re-FUSE server crash is automatically detected and a restart initiated while applications continue to run without knowledge of the failure. FUSE performance is not improved.

4.7.3 Related to `LD_PRELOAD`

OrangeFS Direct Interface [12] uses an `LD_PRELOAD` implementation, but it experiences the FD heritage dilemma. Furthermore, its implementation is tightly coupled with PVFS [16] thus it is not practical as a general purpose FS.

Goanna [194] is a framework for rapid FS implementation in user space based on the `ptrace` kernel interface. `ptrace` allows for a process to monitor for, and intercept system calls of, other processes. Debuggers such as `gdb` [195] use this interface to set breakpoints in executing processes. When a system call from an FS is intercepted, Goanna redirects the operation to a user space FS library as appropriate. This has a similar effect to an `LD_PRELOAD` library without losing internal mappings when a parent creates a child process with open FDs. However, using `ptrace` requires similar wait context switchs between kernel and user space as processes FUSE. Goanna mitigates these by using a modified `ptrace` implementation, however still leaving a large number of wait context switchs. Also, for Goanna to monitor all processes, it must run with superuser privileges, which is a security risk even if it runs in user space.

SplitFS [196] is an FS for persistent memory that is uses an `LD_PRELOAD` implementation to intercept FS related system calls. It contains an implementation similar to FD stashing however does not improve the performance of `open` as DEFUSE does though bypassed FD lookup. For applications that manipulate a large number of small files, the overhead for `open` can dwarf the overhead for doing the I/O operations themselves. Such is the case for high performance computing workloads, some of which create millions of files below 64 kb. In addition, SplitFS is specialized for use with persistent memory and is not a general purpose user space FS interface.

4.7.4 Related to User Space Paging

FluidMem [197] is a library designed to provide memory resource disaggregation by creating a system for memory as a service. It uses the `userfaultfd` call to manage page faults, however its infrastructure acts as a service within a hypervisor providing access to remote memory for a client virtual machine by accessing memory pages on a remote system.

UMap [198] is a library to provide application-driven optimizations for page management. It uses `userfaultfd` to control access to a page map providing applications fine-grained control of page management (e.g. pre-fetching and controlled flushing of pages), however it is not an application-agnostic interface and requires client applications to make explicit calls to set the policies for page management.

4.8 Conclusions

FSs implemented in user space have the advantage of permission isolation, access to user space libraries, and ease of prototyping using a diverse set of programming languages. While deployable interfaces for user space FSs exist, they suffer from significant performance penalties (e.g. FUSE) and/or inconsistent behaviors FDs (e.g. `LD_PRELOAD`-loaded library). Research attempting to improve access performance for user space FSs is often workload-specific (e.g. MPI-IO) or does not address inconsistency issues (e.g. Direct-FUSE).

DEFUSE is a generic interface for user space FSs that enables FS access using existing POSIX system calls with higher access speed than FUSE, up to 2× faster for persistent storage media and as high as 10× faster for memory-based operations in our evaluation. At the same time, DEFUSE improves on existing user space FS interfaces by maintaining consistency of FDs passed between parent and child processes. In future work, we plan to further improve performance of DEFUSE and provide a wrapper interface to streamline porting existing FS implementations from FUSE into DEFUSE such that existing FUSE implementations can be used with DEFUSE with no modification.

5. SUMMARY

This dissertation has advanced the infrastructure for cloud computing through the design, implementation, and evaluation of improvements to the management of network policies and FS access frameworks. Specially, it presents RoSCo, a protocol for network updates that maintains *fault-tolerance*, Cicero, a protocol for network updates that provides *consistency* and *security* while being practical for deployment in a wide area netowrk, and DEFUSE a novel interface for user space FSs that is efficient, flexible and consistent with POSIX requirements.

RoSCo proves that a fault-tolerant control plane is possible beyond the realm of crash failures. An agreement protocol ensures consistency of events handled by the distributed control plane. The use of signatures and quorum authentication to verify network updates ensures correctness while keeping switch instrumentation to a minimum. Evaluation shows that optimizations such as event batching allow for event throughput with acceptable overhead.

Cicero demonstrates that a consistent and secure replicated control plane can be deployed in a wide-area network. Threshold cryptography ensures security of network updates while being flexible to allow for a dynamic control plane minimizing the overhead on networks switches. Exploiting updates parallelism and domain parallelism reduces the latency of network updates providing minimal impact to flows.

DEFUSE provides an interface for user space FSs with a means of efficiency, flexibility, and consistency where other interfaces fail. Bypassed FD lookup ensures correct behavior of FDs inherited by child process, while system call interception through hooks in `libc` ensure efficiency, consistency, and POSIX compliance. FD Stashing ensures valid FD behavior after the invocation of `exec` and user space paging ensures correct behavior of FDs mapped into a process's address space with `mmap`. All together, these features provide the flexibility of an unmodified client application to access user space FSs with minimal overhead.

REFERENCES

[1] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 45, 2015, pp. 123–137.

[2] J. Moy, "OSPF Version 2," RFC Editor, RFC 2178, Jul. 1997, http://www.rfc-editor.org/rfc/rfc2178.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2178.txt.

[3] Y. Rekhter and T. Li, "A Border Gateway Protocol 4 (BGP-4)," RFC Editor, RFC 1771, Mar. 1995, http://www.rfc-editor.org/rfc/rfc1771.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1771.txt.

[4] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN control: Survey, taxonomy, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 333–354, 2018.

[5] N. P. Katta, H. Zhang, M. J. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *SIGCOMM SOSR '15*, 2015, 4:1–4:12.

[6] *Amazon S3 Availability Event: July 20, 2008*. [Online]. Available: https://status.aws.amazon.com/s3-20080720.html.

[7] *AWS Service Event in the US-East Region*. [Online]. Available: https://aws.amazon.com/message/680342.

[8] *Google App Engine outage today*. [Online]. Available: https://groups.google.com/forum/#!topic/google-appengine/985VmzuLMDs.

[9] *Lib FUSE - Filesystem in Userspace*. [Online]. Available: https://github.com/libfuse/libfuse.

[10] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-File Access in Parallel File Systems," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009)*, 2009, pp. 1–11.

[11] S. A. Wright, S. D. Hammond, S. J. Pennycook, I. Miller, J. A. Herdman, and S. A. Jarvis, "LDPLFS: Improving I/O Performance without Application Modification," in *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2012, pp. 1352–1359.

[12] *OrangeFS Direct Interface*. [Online]. Available: http://docs.orangefs.com/v_2_9/Direct_Interface.htm.

[13] *The Plastic File System*, http://plasticfs.sourceforge.net/.

[14] *Solucorp VirtualFS*, http://www.solucorp.qc.ca/virtualfs/.

[15] *AccessFS: Permission Filesystem for Linux*. [Online]. Available: http://www.olafdietsche.de/2002/11/07/accessfs-permission-filesystem-linux/.

[16] R. B. Ross, R. Thakur, *et al.*, "PVFS: A parallel file system for Linux clusters," in *The 4th Annual Linux Showcase and Conference*, 2000, pp. 391–430.

[17] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok, "A Versatile and User-Oriented Versioning File System.," in *3rd USENIX Conference on File and Storage Technologies (FAST '04)*, vol. 4, 2004, pp. 115–128.

[18] *Ryu SDN Framework*, http://osrg.github.io/ryu, Accessed: 2018-04-29.

[19] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, Mar. 2008, ISSN: 0146-4833.

[20] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined Wan," *SIGCOMM CCR*, vol. 43, no. 4, pp. 3–14, Aug. 2013, ISSN: 0146-4833.

[21] *Apache Spark*. [Online]. Available: http://spark.apache.org/.

[22] J. Lembke, S. Ravi, P. Eugster, and S. Schmid, "Rosco: Robust updates for software-defined networks," *IEEE Journal on Selected Areas in Communications*, 2020.

[23] S. Kiran and G. Kinghorn, *Cisco Open Network Environment: Bring the Network Closer to Applications*, Accessed: 2017-07-28. [Online]. Available: http://www.cisco.com/go/one.

[24] *VMware NSX Network Virtualization and Security Platform*, Accessed: 2017-07-28. [Online]. Available: https://www.vmware.com/products/nsx.html.

[25] A. Akella and A. Krishnamurthy, "A Highly Available Software Defined Fabric," in *ACM HotNets-2017*, Los Angeles, CA, USA, 2014, 21:1–21:7, ISBN: 978-1-4503-3256-9.

[26] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos Made Switch-y," *SIGCOMM CCR*, vol. 46, no. 2, pp. 18–24, 2016.

[27] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, and T. Hama, "Onix: A Distributed Control Platform for Large-scale Production Networks," in *OSDI*, vol. 10, 2010, pp. 1–6.

[28] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, and W. Snow, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the third workshop on Hot topics in software defined networking*, ACM, 2014, pp. 1–6.

[29] D. Kreutz, F. Ramos, and P. Verissimo, "Towards Secure and Dependable Software-Defined Networks," in *SIGCOMM HotSDN*, 2013, pp. 55–60.

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

[31] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *TOPLAS*, vol. 4, no. 3, pp. 382–401, 1982.

[32] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid, "Taking control of sdn-based cloud systems via the data plane," in *Proc. ACM Symposium on SDN Research (SOSR)*, 2018.

[33] K. Birman, *Byzantine Clients*, https://thinkingaboutdistributedsystems.blogspot.com/2017/05/byzantine-clients.html, 2017.

[34] W. Braun and M. Menth, "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices," *Future Internet*, vol. 6, no. 2, pp. 302–336, 2014.

[35] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in *INM/WREN*, 2010, pp. 3–3.

[36] L. Lamport, "Paxos Made Simple," *SIGACT News*, vol. 32, no. 4, D. C. column, Ed., pp. 18–25, Dec. 2001.

[37] L. Lamport and M. Fischer, "Byzantine Generals and Transaction Commit Protocols," SRI International, Tech. Rep. 62, Apr. 1982.

[38] N. Feamster, J. Rexford, and E. W. Zegura, "The Road to SDN: an Intellectual History of Programmable Networks," *SIGCOMM CCR*, vol. 44, no. 2, pp. 87–98, 2014.

[39] M. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *TOPLAS*, vol. 12, no. 3, pp. 463–492, 1990.

[40] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A Distributed and Robust SDN Control Plane for Transactional Network Updates," in *INFOCOM*, 2015, pp. 190–198. DOI: 10.1109/INFOCOM.2015.7218382.

[41] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," in *SIGCOMM*, 2012, pp. 323–334.

[42] J. McClurg, H. Hojjat, N. Foster, and P. Černỳ, "Event-driven Network Programming," in *SIGPLAN Notices*, vol. 51, 2016, pp. 369–385.

[43] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing Customizable Consistency Properties in Software-defined Networks," in *NSDI*, 2015, pp. 73–85.

[44] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic Scheduling of Network Updates," in *SIGCOMM*, 2014, pp. 539–550.

[45] K. Förster, R. Mahajan, and R. Wattenhofer, "Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes," in *IFIP NETWORKING*, 2016, pp. 1–9.

[46] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, 2015.

[47] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of Consistent Network Updates," in *ArXiv Technical Report*, 2016.

[48] N. P. Katta, J. Rexford, and D. Walker, "Incremental Consistent Updates," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013*, 2013, pp. 49–54.

[49] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX ATC*, vol. 8, 2010, p. 9.

[50] D. Ongaro and J. K. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX ATC*, 2014, pp. 305–319.

[51] S. H. Yeganeh and Y. Ganjali, "Beehive: Simple Distributed Programming in Software-Defined Networks," in *SOSR*, 2016, p. 4.

[52] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A Robust, Secure, and High-Performance Network Operating System," in *SIGSAC*, 2014, pp. 78–89.

[53] B. Chandrasekaran and T. Benson, "Tolerating SDN Application Failures with LegoSDN," in *HotNets*, 2014, p. 22.

[54] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid, "Renaissance: A Self-Stabilizing Distributed SDN Control Plane," in *Proc. IEEE ICDCS*, 2018.

[55] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: Defending SDNs from Malicious Administrators," in *SIGCOMM HotSDN*, 2014, pp. 103–108.

[56] C. V. Zhou, C. Leckie, and S. Karunasekera, "A Survey of Coordinated Attacks and Collaborative Intrusion Detection," *Computers & Security*, vol. 29, no. 1, pp. 124–140, 2010.

[57] H. T. Elshoush and I. M. Osman, "Alert Correlation in Collaborative Intelligent Intrusion Detection Systems —- A Survey," *Applied Soft Computing*, vol. 11, no. 7, pp. 4349–4365, 2011.

[58] R. Guerraoui and M. Kapalka, *Principles of Transactional Memory,Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[59] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[60] A. Bessani, J. Sousa, and E. E. Alchieri, "State Machine Replication for the Masses with BFT-SMaRt," in *DSN*, IEEE, 2014.

[61] *About DETERLab*. [Online]. Available: https://deter-project.org/about_deterlab.

[62] *CPython Global Interpreter Lock*, https://wiki.python.org/moin/GlobalInterpreterLock, Accessed: 2018-04-29.

[63] J. Lembke, S. Ravi, P.-L. Roman, and P. Eugster, "Consistent and secure network updates made practical," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 149–162.

[64] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, ser. HotNets-XII, 2013, 20:1–20:7.

[65] S. Brandt, K.-T. Foerster, and R. Wattenhofer, “Augmenting flows for the consistent migration of multi-commodity single-destination flows in SDNs,” *Pervasive and Mobile Computing*, vol. 36, pp. 134–150, 2017.

[66] L. Luo, H. Yu, S. Luo, and M. Zhang, “Fast lossless traffic migration for SDN updates,” in *2015 IEEE International Conference on Communications (ICC)*, IEEE, 2015, pp. 5803–5808.

[67] K.-T. Foerster and R. Wattenhofer, “The Power of Two in Consistent Network Updates: Hard Loop Freedom, Easy Flow Migration,” in *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, 2016, pp. 1–9. DOI: 10.1109/ICCCN.2016.7568583.

[68] H. Li, P. Li, S. Guo, and A. Nayak, “Byzantine-Resilient Secure Software-Defined Networks with Multiple Controllers in Cloud,” *IEEE Transactions on Cloud Computing*, vol. 2, no. 4, pp. 436–447, 2014.

[69] E. Sakic, N. Deric, and W. Kellerer, “MORPH: An Adaptive Framework for Efficient and Byzantine Fault-Tolerant SDN Control Plane,” *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2158–2174, 2018.

[70] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, and D. Walker, “Contra: A programmable system for performance-aware routing,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 701–721.

[71] *Openflow switch specification*, ONF TS-025, v1.5.1, Open Networking Foundation, Mar. 2015.

[72] M. Dargin, *Secure your SDN controller*.

[73] S. Hogg, *SDN Security Attack Vectors and SDN Hardening*. [Online]. Available: https://www.networkworld.com/article/2840273/sdn-security-attack-vectors-and-sdn-hardening.html.

[74] D. Asturias, *9 Types of Software Defined Network attacks and how to protect from them*. [Online]. Available: https://www.routerfreak.com/9-types-software-defined-network-attacks-protect/.

[75] M. Brooks and B. Yang, “A Man-in-the-Middle attack against OpenDayLight SDN controller,” in *Proceedings of the 4th Annual ACM Conference on Research in Information Technology*, ser. RIIT ’15, 2015, pp. 45–49.

[76] J. M. Dover, “A denial of service attack against the Open Floodlight SDN controller,” *Dover Networks LCC, Edgewater, MD, USA*, 2013.

[77] *OpenFlow PacketOut*, http://flowgrammable.org/sdn/openflow/message-layer/packetout/.

[78] S. Lee, C. Yoon, and S. Shin, "The smaller, the shrewder: A simple malicious application can kill an entire sdn environment," in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, ACM, 2016, pp. 23–28.

[79] A. Azzouni, R. Boutaba, N. T. M. Trang, and G. Pujolle, "Softdp: Secure and efficient openflow topology discovery protocol," in *2018 IEEE/IFIP Network Operations and Management Symposium*, ser. NOMS'18, IEEE, 2018, pp. 1–7.

[80] *Policy Framework for ONOS*, https://wiki.onosproject.org/display/ONOS/POLICY+FRAMEWORK+FOR+ONOS.

[81] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[82] *OpenDaylight Group Based Policy*, https://docs.opendaylight.org/en/stable-fluorine/user-guide/group-based-policy-user-guide.html.

[83] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in software-defined networking (SDN)," *Computer Networks*, vol. 112, pp. 279–293, 2017.

[84] P. Thai and J. C. de Oliveira, "Decoupling policy from routing with software defined interdomain management: Interdomain routing for SDN-based networks," in *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2013, pp. 1–6.

[85] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: Towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[86] *Cisco Open SDN Controller*, http://www.cisco.com/c/en/us/products/cloud-systems-management/open-sdn-controller/index.html.

[87] *OpenDaylight*, https://www.opendaylight.org.

[88] *Central Office Re-architected as a Datacenter (CORD)*, https://opencord.org/.

[89] *Packet-Optical*, https://wiki.onosproject.org/display/ONOS/Packet+Optical+Convergence.

[90] F. Botelho, T. A. Ribeiro, P. Ferreira, F. M. V. Ramos, and A. Bessani, "Design and Implementation of a Consistent Data Store for a Distributed SDN Control Plane," in *2016 12th European Dependable Computing Conference (EDCC)*, 2016, pp. 169–180.

[91] T. D. Nguyen, M. Chiesa, and M. Canini, "Decentralized consistent updates in SDN," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17, ACM, 2017, pp. 21–33.

[92] P. Černỳ, N. Foster, N. Jagnik, and J. McClurg, "Optimal consistent network updates in polynomial time," in *International Symposium on Distributed Computing (DISC)*, Springer, 2016, pp. 114–128.

[93] R. Wallner and R. Cannistra, "An sdn approach: Quality of service using big switch's floodlight open-source controller," *Proceedings of the Asia-Pacific Advanced Network*, vol. 35, pp. 14–19, 2013.

[94] B. Agborubere and E. Sanchez-Velazquez, "OpenFlow Communications and TLS Security in Software-Defined Networks," in *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, IEEE, 2017, pp. 560–566.

[95] P. Perešíni, M. Kuzniar, M. Canini, and D. Kostić, "ESPRES: transparent SDN update scheduling," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, ACM, 2014, pp. 73–78.

[96] J. McClurg, H. Hojjat, P. Černỳ, and N. Foster, "Efficient synthesis of network updates," in *ACM SIGPLAN Notices*, ACM, vol. 50, 2015, pp. 196–207.

[97] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell University, Tech. Rep., 1994.

[98] Y. Desmedt, "Threshold cryptography," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 449–458, 1994.

[99] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust Threshold DSS Signatures," in *EUROCRYPT*, 1996, pp. 354–371.

[100] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[101] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, "Verifiable secret sharing and achieving simultaneity in the presence of faults," in *26th Annual Symposium on Foundations of Computer Science (SFCS 1985)*, IEEE, 1985, pp. 383–395.

[102] A. Kate, Y. Huang, and I. Goldberg, "Distributed Key Generation in the Wild," *IACR Cryptology ePrint Archive*, vol. 2012, p. 377, 2012.

[103] A. Doudou, B. Garbinato, and R. Guerraoui, "Encapsulating failure detection: From crash to byzantine failures," in *Reliable Software Technologies — Ada-Europe 2002*, Springer, 2002, pp. 24–50.

[104] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[105] N. Hayashibara, A. Cherif, and T. Katayama, "Failure detectors for large-scale distributed systems," in *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS)*, IEEE, 2002, pp. 404–409.

[106] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *Journal of Cryptology*, vol. 17, no. 4, pp. 297–319, Sep. 2004, ISSN: 1432-1378. DOI: 10.1007/s00145-004-0314-9.

[107] B. Lynn, https://crypto.stanford.edu/pbc/.

[108] A. Kate, *Distributed Key Generator*, https://crysp.uwaterloo.ca/software/DKG/.

[109] *OpenFlow Role Request Messages*. [Online]. Available: https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#role-request-message.

[110] *DETERLab PC3000 Node Information*, https://www.isi.deterlab.net/shownodetype.php?node_type=pc3000.

[111] *OpenVz*, https://openvz.org/.

[112] *Introducing data center fabric, the next-generation Facebook data center network*, https://code.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[113] F. J. Ros and P. M. Ruiz, "Five nines of southbound reliability in software-defined networks," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, ACM, 2014, pp. 31–36.

[114] *The Internet Topology Zoo*, http://www.topology-zoo.org/.

[115] *Ext4 (and Ext2/Ext3) Wiki.* [Online]. Available: https://ext4.wiki.kernel.org/.

[116] *Lustre Parallel File System*, http://lustre.org/.

[117] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, USENIX Association, 2006, pp. 307–320.

[118] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, *et al.*, "BetrFS: A Right-Optimized Write-Optimized File System.," in *13rd USENIX Conference on File and Storage Technologies (FAST '15)*, 2015, pp. 301–315.

[119] *SSHFS*, https://github.com/libfuse/sshfs.

[120] D. Borthakur *et al.*, "HDFS architecture guide," *Hadoop Apache Project*, vol. 53, pp. 1–13, 2008.

[121] S. Sharwood, *Linux literally loses its Lustre – HPC filesystem ditched in new kernel*, 2018. [Online]. Available: https://www.theregister.co.uk/2018/06/18/linux_4_18_rc_1_removes_lustre_filesystem/.

[122] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser, "User-Level Device Drivers: Achieved Performance," *Journal of Computer Science and Technology*, vol. 20, no. 5, pp. 654–664, Sep. 2005.

[123] D. Burihabwa, P. Felber, H. Mercier, and V. Schiavoni, "SGX-FS: Hardening a File System in User-Space with Intel SGX," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom'18)*, 2018, pp. 67–72.

[124] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok, "Terra incognita: On the practicality of user-space file systems," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, USENIX Association, 2015.

[125] G. Essertel, R. Tahboub, J. Decker, K. Brown, K. Olukotun, and T. Rompf, "Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, 2018, pp. 799–815, ISBN: 978-1-931971-47-8.

[126] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *High Performance Computing, Networking, Storage and Analysis (SC09)*, 2009, pp. 1–12.

[127] J. T. Inman, W. F. Vining, G. W. Ransom, and G. A. Grider, "MarFS, a Near-POSIX Interface to Cloud Objects," *The USENIX Magazine*, Spring 2017.

[128] *GlusterFS - A Scale-Out Network-Attached Storage File System*, https://www.gluster.org/.

[129] *Databricks File System*. [Online]. Available: https://docs.databricks.com/user-guide/dbfs-databricks-file-system.html/.

[130] H. Li, "Alluxio: A virtual distributed file system," PhD thesis, UC Berkeley, 2018.

[131] *Google Cloud Storage*, https://cloud.google.com/storage/.

[132] *Amazon S3*. [Online]. Available: https://aws.amazon.com/s3/.

[133] *Tahoe-LAFS - Tahoe Least-Authority File Store*. [Online]. Available: https://tahoe-lafs.org/trac/tahoe-lafs/.

[134] *ObjectiveFS*. [Online]. Available: https://objectivefs.com/.

[135] *Moose File System (MooseFS)*. [Online]. Available: https://moosefs.com/index.html.

[136] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "XtreemFS: A Case for Object-Based Storage in Grid Data Management," in *3rd VLDB Workshop on Data Management in Grids, co-located with VLDB*, 2007.

[137] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *26th IEEE Symposium on Massive Storage Systems and Technologies (MSST2010)*, 2010, pp. 1–10.

[138] *IBM Spectrum Scale - Formerly General Parallel File System (GPFS)*, https://www.ibm.com/us-en/marketplace/scale-out-file-and-object-storage.

[139] *Apache Hadoop 2.4.1 - File System Shell Guide*. [Online]. Available: https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-common/FileSystemShell.html#Overview.

[140] *Access DBFS with the Databricks CLI*. [Online]. Available: https://docs.databricks.com/user-guide/dbfs-databricks-file-system.html#access-dbfs-with-the-databricks-cli.

[141] *gsutil tool*. [Online]. Available: https://cloud.google.com/storage/docs/gsutil.

[142] *Linux Manual - Overview, Conventions, and Miscellaneous: libc*. [Online]. Available: http://man7.org/linux/man-pages/man7/libc.7.html.

[143] D. Behrens, M. Serafini, F. P. Junqueira, S. Arnautov, and C. Fetzer, "Scalable Error Isolation for Distributed Systems," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, 2015, pp. 605–620.

[144] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, 2012, pp. 15–28.

[145] *Alluxio-FUSE*. [Online]. Available: https://github.com/Alluxio/alluxio/tree/master/integration/fuse.

[146] *Mountable HDFS*, https://wiki.apache.org/hadoop/MountableHDFS.

[147] *FUSE for Google Cloud Storage*. [Online]. Available: https://github.com/GoogleCloudPlatform/gcsfuse/.

[148] *Amazon S3 FUSE*. [Online]. Available: https://github.com/s3fs-fuse/s3fs-fuse.

[149] *Access DBFS using local file APIs*. [Online]. Available: https://docs.databricks.com/user-guide/dbfs-databricks-file-system.html#access-dbfs-using-local-file-apis.

[150] M. Pillai, R. Gowdappa, and C. Henk, "Experiences with Fuse in the Real World," in *2019 Linux Storage and Filesystems Conference (VAULT'19)*, Feb. 2019.

[151] B. K. R. Vangoor, P. Agarwal, M. Mathew, A. Ramachandran, S. Sivaraman, V. Tarasov, and E. Zadok, "Performance and Resource Utilization of FUSE User-Space File Systems," *ACM Transactions on Storage*, vol. 15, no. 2, May 2019.

[152] R. Thakur, E. Lusk, and W. Gropp, "Users guide for romio: A high-performance, portable mpi-io implementation," Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Tech. Rep., 1997.

[153] Y. Zhu, T. Wang, K. Mohror, A. Moody, K. Sato, M. Khan, and W. Yu, "Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support," in *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*, ACM, 2018, p. 6.

[154] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong, "Overview of the MPI-IO Parallel I/O Interface," in *IPPS'95 Workshop on Input/Output in Parallel and Distributed Systems*, 1995, pp. 1–15.

[155] *C API libhdfs*. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/LibHdfs.html.

[156] *Amazon Web Services SDK for C++*. [Online]. Available: https://aws.amazon.com/sdk-for-cpp/.

[157] *FAT Filesystem Library in R6RS Scheme*, https://gitlab.com/weinholt/fs-fatfs.

[158] *Linux Virtual File System*, http://www.tldp.org/LDP/tlk/fs/filesystem.html.

[159] *Personal conversation with David Bonnie, storage tech lead at Los Alamos National Laboratory and co-designer of OrangeFS/PVFS2, in reference to work on MarFS 11/15/2016*.

[160] *Linux User Manual - Time Command, Option %w for Waits*, https://linux.die.net/man/1/time.

[161] *Fuse example fusexmp*. [Online]. Available: https://github.com/fuse4x/fuse/blob/master/example/fusexmp.c.

[162] *Spark PySpark Daemon*, https://github.com/apache/spark/blob/5264164a67df98b73facae207ec
python/pyspark/daemon.py.

[163] *React Hooks*, https://reactjs.org/docs/hooks-intro.html.

[164] *Emacs Hooks*, https://www.gnu.org/software/emacs/manual/html_node/emacs/Hooks.html.

[165] *The Linux Kernel - userfaultfd*, https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html.

[166] *AVFS - A Virtual File System*. [Online]. Available: http://avf.sourceforge.net/.

[167] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. Panda, "A 1 PB/s File System to Checkpoint Three Million MPI Tasks," in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, 2013, pp. 143–154.

[168] *Native HDFS FUSE*, https://github.com/remis-thoughts/native-hdfs-fuse.

[169] *IOzone Filesystem Benchmark*, http://iozone.org/.

[170] *Journaled File System Technology for Linux*. [Online]. Available: http://jfs.sourceforge.net/.

[171] Microsoft Corporation, "Microsoft Extensible Firmware Initiative FAT32 File System Specification," Microsoft Corporation, Tech. Rep., 2000.

[172] *tmpfs Documentation*. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt.

[173] *SYSIO Library*, https://libsysio.sourceforge.io/.

[174] D. J. Bonnie and A. G. Torres, "Small File Aggregation with PLFS," Los Alamos National Laboratory (LANL), Tech. Rep., 2013.

[175] *TPCx-BB Specification*, https://www.tpc.org/.

[176] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, 2015, pp. 1383–1394.

[177] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, *PUMA: Purdue University Benchmark Suite*, 2012.

[178] MPI Message Passing Forum, *A Message-Passing Interface Standard Version 3.1*, 2015.

[179] R. Thakur, W. Gropp, and E. Lusk, "An abstract-device interface for implementing portable parallel-I/O interfaces," in *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers' 96., Sixth Symposium on the*, IEEE, 1996, pp. 180–187.

[180] S. Ishiguro, J. Murakami, Y. Oyama, and O. Tatebe, "Optimizing Local File Accesses for FUSE-based Distributed Storage," in *High Performance Computing, Networking, Storage and Analysis (SC '12)*, 2012, pp. 760–765.

[181] A. Bijlani and U. Ramachandran, “Extension Framework for File Systems in User Space,” in *2019 USENIX Annual Technical Conference (ATC’19)*, 2019, pp. 121–134.

[182] *Linux Manual - bpf - Perform a Command on an Extended BPF Map or Program.* [Online]. Available: http://man7.org/linux/man-pages/man2/bpf.2.html.

[183] E. Zadok and I. Bădulescu, “A stackable file system interface for Linux,” in *LinuxExpo Conference Proceedings*, Raleigh, NC, May 1999, pp. 141–151.

[184] D. S. Rosenthal, “Evolving the Vnode interface,” in *USENIX Summer*, vol. 99, 1990, pp. 107–118.

[185] E. Zadok and J. Nieh, “FiST: A Language for Stackable Filesystems,” *Proceedings of the Annual USENIX Technical Conference. June 2000*, 2000.

[186] S. Narayan, R. K. Mehta, and J. A. Chandy, “User Space Storage System Stack Modules with File Level Control,” in *Proceedings of the 12th Annual Linux Symposium in Ottawa*, 2010, pp. 189–196.

[187] P. Deniel, T. Leibovici, and J.-C. Lafoucrière, “GANESHA, A Multi-Usage with Large Cache NFSv4 Server,” in *Linux Symposium*, 2007, p. 113.

[188] *NFS Ganesha - File System Abstraction Layer (FSAL)*, https://github.com/nfs-ganesha/nfs-ganesha/wiki/Fsalsupport.

[189] D. C. Steere, J. J. Kistler, and M. Satyanarayanan, *Efficient User-Level File Cache Management on the Sun vnode Interface.* figshare, 1996.

[190] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, “Coda: A Highly Available File System for a Distributed Workstation Environment,” *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[191] M. Patlasov, “Optimizing FUSE for Cloud Storage,” in *Linux Vault*, 2015.

[192] S. Sundararaman, L. Visampalli, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Refuse to Crash with Re-FUSE,” in *6th Conference on Computer Systems (EuroSys’11)*, 2011, pp. 77–90.

[193] A. Kantee and A. Crooks, “ReFUSE: Userspace FUSE Reimplementation using PUFFS,” in *6th European BSD Conference (EuroBSDCon’07)*, 2007.

[194] R. P. Spillane, C. P. Wright, G. Sivathanu, and E. Zadok, “Rapid File System Development using ptrace,” in *Workshop on Experimental Computer Science, Part of ACM FCRC*, 2007, p. 22.

[195] *GDB: The GNU Project Debugger*, https://www.gnu.org/software/gdb/.

[196] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, “SplitFS: Reducing Software Overhead in File Systems for Persistent Memory,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019, pp. 494–508.

[197] B. Caldwell, Y. Im, S. Ha, R. Han, and E. Keller, “Fluidmem: Memory as a service for the datacenter,” *arXiv preprint arXiv:1707.07780*, 2017.

[198] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale, “Umap: Enabling application-driven optimizations for page management,” in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, IEEE, 2019, pp. 71–78.

VITA

James Lembke received his BS and MS in computer science from Michigan Technological University in 2003 and 2005 respectively. He worked as a software engineer for IBM from 2005 until 2013 before leaving to pursue his PhD at Purdue University. His research interests include software defined networking, file systems, and memory management.