AMERICAN UNIVERSITY OF BEIRUT

STATEFUL DISTRIBUTED FIREWALL AS A SERVICE IN SDN

By

ALI HAIDAR ZEINEDDINE

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
January 2018

AMERICAN UNIVERSITY OF BEIRUT

STATEFUL DISTRIBUTED FIREWALL AS A SERVICE IN SDN

by

ALI HAIDAR ZEINEDDINE

Approved by:

[illegible]

Dr. Wassim El Hajj, Associate Professor — Advisor
Computer Science

[illegible]

Dr. Ayman Kayssi, Professor — Member of Committee
Electrical and Computer Engineering

[illegible]

Dr. Ahmad Dhaini, Assistant Professor — Member of Committee
Computer Science

Date of thesis defense: January 30, 2018

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Zeineddine Ali Haidar

[✓ ● Master's Thesis] ○ Master's Project ○ Doctoral Dissertation

[✓] I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after : **One ---- year from the date of submission of my thesis, dissertation, or project.**
Two ---- years from the date of submission of my thesis, dissertation, or project.
Three ---- years from the date of submission of my thesis, dissertation, or project.

[illegible] Feb 12, 2018

Signature Date

ACKNOWLEDGMENTS

This work was carried out during the years 2016-2017 at the Computer Science Department, American University of Beirut.

First and foremost, enormous gratitude is due to my advisor, professor Wassim El Hajj, for his encouragement, support, and guidance throughout the different stages of this work. I appreciate all his contributions in time and effort to make my Master's experience productive and motivational.

I was fortunate to have professor Ayman Kayssi and professor Ahmad Dhaini as thesis committee members. Their comments, advice, and constructive critique have greatly contributed to enhancing the quality of this work.

Finally, I would like to thank my family members who provided me with unfailing support and continuous encouragement, Ibrahim for his prompt assistance on the technical front, and all who- directly or indirectly- helped me make my way.

AN ABSTRACT OF THE THESIS OF

Ali Haidar Zeineddine for Master of Science
Major: Computer Science

Title: Stateful Distributed Firewall as a Service in SDN

Software-defined networking (SDN) is a newly emerging approach in computer networking which abstracts network control functionalities and enables its direct programmability at the management plane. The fundamental difference between software-defined networks and traditional networks is in the network architecture itself. In SDN, the data-plane is separated from the control-plane. The former is composed of SDN dummy switches that are directly programmable with flow-based rules by a logically centralized controller that resides at the control plane. SDN has evolved tremendously throughout the last few years. Although the two main approaches, proactive approach and reactive approach, were being widely addressed as a framework of communication between the control-plane and the data-plane, a new hybrid approach is emerging which combines the advantages of the proactive approach, in pre-installing the flow rules in the data-plane, and the advantages of the reactive approach, in its ability to dynamically react to network events. This hybrid approach utilizes the potential of the SDN switches to recognize and host state machines. While the trending success of SDN is set to continue, this evolving network paradigm requires a new set of tools and strategies to secure the network elements against intrusions and at the same time maintain its efficiency and reliability. In this text, we take advantage of the hybrid approach of network controllability and management to offload the processing of stateful applications from the control-plane to the data-plane and propose our framework, Stateful Distributed Firewall as a Service in SDN (SDFS), that optimizes a distributed stateful application in the data-plane to transform the SDN network into a one big firewall. While maintaining modularity of the framework, SDFS offers an optimized processing burden distribution of the stateful application in the data-plane among the switches in the network with inherent fault-tolerance mechanisms that eliminate the need for immediate controller intervention even in cases of failure or attacks on the network.

CONTENTS

ILLUSTRATIONS

TABLES

Table Pages

CHAPTER 1

INTRODUCTION

Software-defined networking (SDN) is a newly emerging approach in computer networking which abstracts network control functionalities and enables its direct programmability at the management plane. SDN has taken networking to a new level in only few years. In sharp contrast to the traditional networking practices, software-defined networking created a wide spectrum of opportunities to engineer much more efficient networking frameworks. The fundamental difference between software-defined networks and traditional networks is in the network architecture itself.

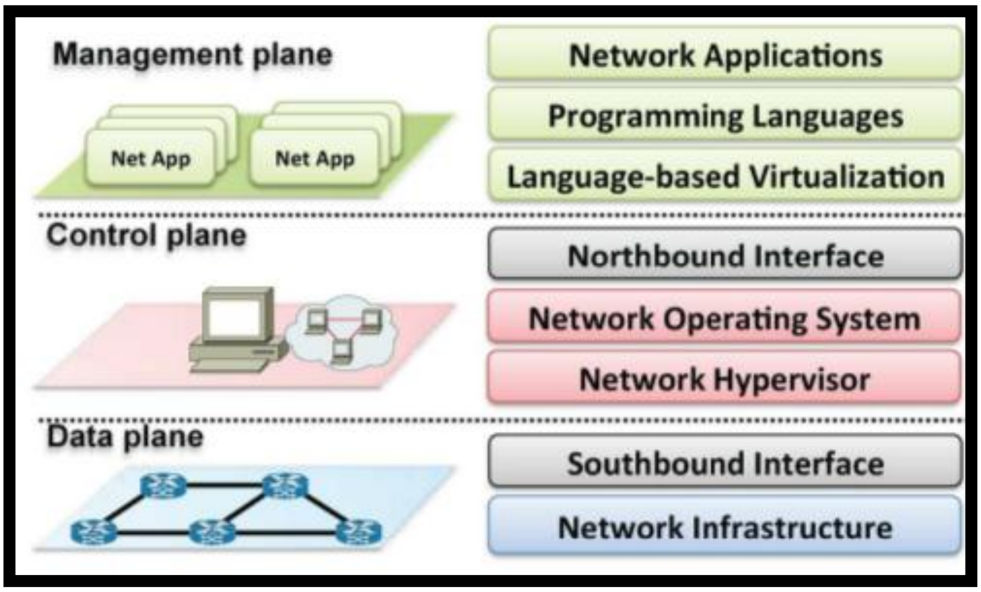


Figure 1: SDN architecture [27]

As depicted in Figure 1, in the SDN architecture, the data-plane is separated from the control-plane. The former is composed of SDN dummy switches that are directly programmable with flow-based rules by a logically centralized controller that resides at the control plane. As such, decisions about how traffic should be forwarded in the network is not taken on the data-plane anymore, like the case in traditional OSPF and BGP, but migrated to the controller. According to the SDN architecture, the controller is the intelligent unit that utilizes a global view of the network. All of the underlying infrastructure is abstracted and optimized at the controller level. This controller can be configured and managed using the Northbound APIs, where it exposes its abilities to programmable network applications. On the other hand, the controller utilizes the Southbound APIs to communicate and program the switches in the data-plane.

While SDN turned from a pure idea to a concrete framework [1, 2, 3], it has only gained recognition by vendors when OpenFlow [4] was first adopted as a standard southbound protocol for communication between the control plane and the data plane as well as remote programmability of the data-plane. Since then, hardware OpenFlow compliant switches were produced such as the Cisco 6500 series and Juniper T-640 along with software OpenFlow compliant switches such as OpenVswitch [35].

These switches employ OpenFlow Tables to forward traffic. A flow table comprises a list of flow table entries (flows). Each flow table entry comprises a **Match** representing a set of packet fields + other OpenFlow specific match fields (inport, metadata,..), an **Action** such as outport, drop, and modify-packet-fields, along with **counters** that hold statistics about the flows and packets. When a packet enters a switch, it is matched against the **Match** fields of the flows in the table. Whenever a match hits, the **Action** of that flow is executed upon the packet, and the **counters** of the flows are updated accordingly [35].

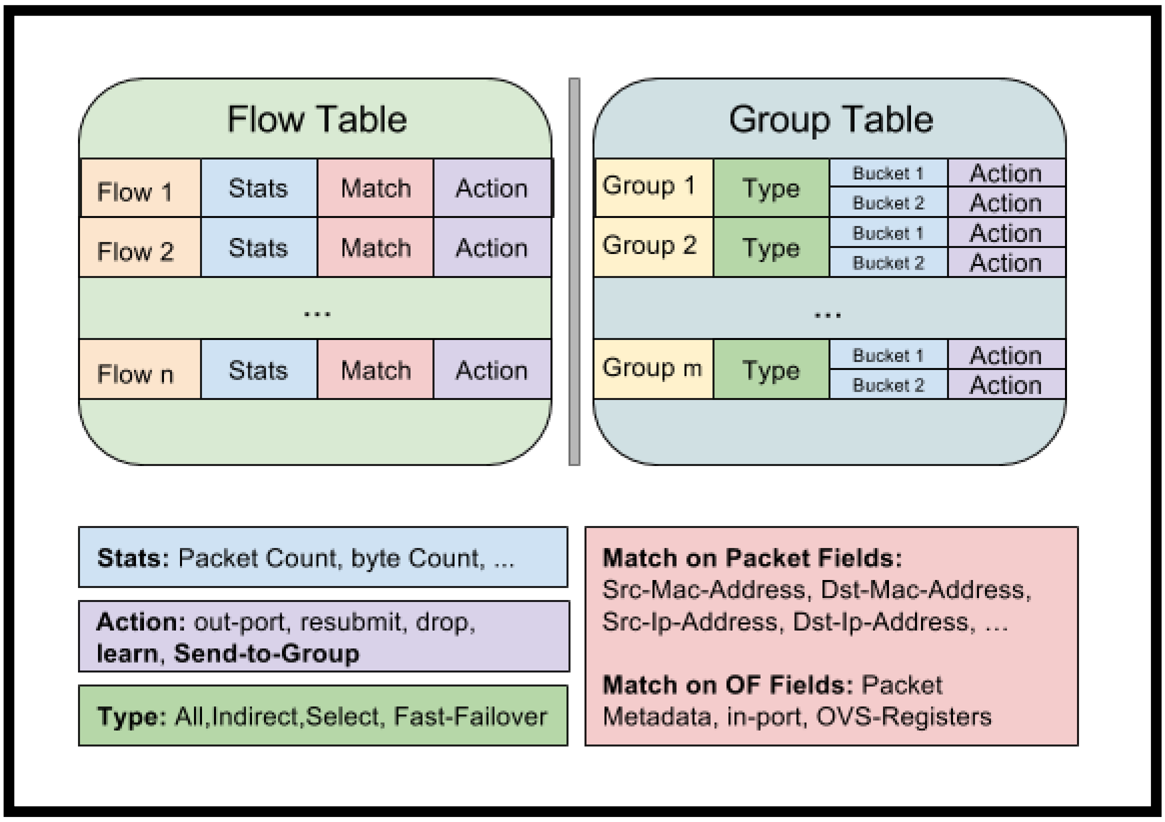

Flow Table
Flow 1
Stats
Match
Action
Flow 2
Stats
Match
Action
...
Flow n
Stats
Match
Action
Group Table
Group 1
Type
Bucket 1
Action
Bucket 2
Action
Group 2
Type
Bucket 1
Action
Bucket 2
Action
...
Group m
Type
Bucket 1
Action
Bucket 2
Action
Stats: Packet Count, byte Count, ...
Action: out-port, resubmit, drop, learn, Send-to-Group
Type: All,Indirect,Select, Fast-Failover
Match on Packet Fields: Src-Mac-Address, Dst-Mac-Address, Src-Ip-Address, Dst-Ip-Address, ...
Match on OF Fields: Packet Metadata, in-port, OVS-Registers

Figure 2: Flow tables and group tables in OpenFlow

With OpenFlow, these switches are presented with a neat abstraction of their flow tables at the control plane. As such, OpenFlow allows remote administration and programmability of the data-plane switches. And since it was first released, it has become a fundamental protocol in the SDN environment.

With the continuous efforts being put into OpenFlow, new features and extensions arise with every release. These features address specific problems and needs. For example, in the older versions of OpenFlow, when a network link fails, the switch had to communicate this failure to the controller which would then intervene to update the flow tables of the switch. However, since many packets will have to be dropped in the process due to the slow switch-controller communication, this approach is disruptive. Consequently, a group table was introduced alongside the flow tables as shown in Figure 2. This group table comprises a set of groups. Each group incorporates a set of buckets where each bucket is a set of OpenFlow **actions**. The actions in the buckets inside a group are executed according to the group type. Currently, four group types exist: Fast-Failover, Select, All, and Indirect. The **fast-failover** group addresses the slow switch-controller communication problem upon link failure as described above. This group incorporates multiple action buckets where each bucket watches the "liveness" of its corresponding port. Whenever a packet is submitted to this group, the first live bucket is executed.

Select group, on the other hand, permits the execution of one bucket after hashing on packet fields. This provides support for load-balancing operations at the switch. For example, when a packet is submitted to this group, the packet fields are used to compute a hash value. According to this hash value, one bucket out of the group is chosen, and the packet is forwarded according to the outport corresponding to this bucket. The remaining two group types are **All** and **Indirect** where **All** executes all the buckets in the group and **Indirect** executes the one bucket in the group.

Additionally, OpenFlow provides support for meters used in QoS and rate control, along with a learning functionality to locally install new rules upon packet events. One example where learning flows are employed is the MAC learning application where the switch installs new flows to automatically respond to ARP requests whenever it discovers a new MAC address in the network. Although OpenFlow offers clean solutions to specific problems, there are no strict specifications or standardizations as to how the rules should be pipelined and used. Accordingly, such specifications are decided upon design and implementation of applications [22].

Different approaches have been adopted to install flows onto the switches. The first one is the proactive mode. In this mode, the controller pre-installs all the flow rules onto the switches. However, this requires a priori knowledge about the network, and in practice, this is often impossible. Consider, for example, the case where the controller installs flows to route packets to different servers. If a server is migrated or newly deployed elsewhere in the network, the controller will not be able to detect it and install the relevant flows.

The second approach is the reactive mode. In this mode, a flow rule with least priority is inserted in every flow table. The **match** field of this rule does not have any constraints on the packet fields in such a way that all packets can match against this rule. Additionally, the **action** field of this rule mandates that the packet is tunneled from the switch to the controller. Thus, when a switch receives a packet that cannot be handled by its flow tables (i.e. the packet does not match on any flow rule in a flow table), it will eventually match against this rule. Accordingly, the switch notifies the controller about the packet it couldn't handle. The controller then installs the appropriate flows onto the switch to handle this scenario in the future. In the example above, the controller is able to detect new ip-addresses and mac-addresses appearing in the network and consequently handle network updates. While this approach allows the controller to dynamically control the network, its performance is disadvantaged by controller-exhaustive interventions.

In this regard, a new approach is emerging which takes the best of the two approaches. This approach entails offloading the stateful processing of the applications from the controller to the data-plane. This is only made possible by the ability of the OpenFlow switches to host state machines in their flow-tables. Thus, instead of driving the traffic to converge at the controller in order to be statefully processed, in this approach, the controller can proactively program the switches to statefully process the traffic on the data-plane at packet speed.

This approach has found its way in the literature in many firewall known applications such as connection filtering and tracking, DNS Tunnel Detection, port-knocking and FTP monitoring. However, as discussed in the literature, the stateful firewall architectures proposed for data-plane processing restrict the stateful processing computation to the direct switch connected to the host/server. Since the stateful application hosted in a switch should fall on the data path of the traffic, handling of connection tracking for example at the direct switch of the protected host can be the safe choice to insure consistency and correctness. However, this design cannot scale if the number of connections through the direct switch exploded. In such scenarios, the direct switch becomes an inevitable network choke-point. Along with network performance degradation, memory issues can arise. Since there is no upper limit on the number of flows in OpenVswitch, these switches are only constrained by memory, where in cases of an attack, the excessive amount of tracking flows can eat up the memory of the system. In contrast, traditional IpTables enforce a limit on the number of connections that can be tracked. This limit is used to prevent DoS attacks that use all of the system's memory. In that case, IpTables will respond by the hideous **ip_conntrack: table full, dropping packet,** after which the server will seem offline.

In this text, we propose our framework, Stateful Distributed Firewall as a service in SDN (SDFS), that transforms the SDN network into a big firewall, in which the SDN switches collectively become one firewall. Processing of stateful applications can be off-loaded from the controller to the data-plane switches where these switches can distributively collaborate in the processing burden. Not only controller applications can be distributed in the data-plane, but also the framework drives a traffic engineering foundation that provisions burden distribution of traditional data-plane applications such as deep packet inspection (DPI) where packets can be mirrored to these special DPI devices in an optimized fashion. This framework inherently incorporates a failover mechanism that provisions a highly available network to provide redundancy of the firewall application at the data-plane and eliminate single points of failure. With this framework, therefore, the network can seamlessly reconform into a new configuration upon the failure of certain devices (or a change in the network topology). Even in cases of an attack or device downtime, the framework maintains its security functions over the network. The ease of manageability in this framework is accomplished by an SDN controller that automatically re-configures and optimizes the network without extensively intervening in the processing procedure. In contrast to traditional distributed firewalls where rip/replace routines are required to update the network, the course of this framework is to augment the network with firewall as a service that is provided with high-availability, connectivity, and manageability and gives the network admin an easy way to configure the network security policies. Instead of investing in a large cluster of expensive dedicated hardware to handle high performance events, this framework leads off the utilization of lightweight low-cost devices to provide high-availability, while maintaining a compatible and efficient usability of dedicated hardware at the core of the network if needed.

In the next section (Section 2) we will present the background on this topic and its related work which includes the security challenges in the SDN paradigm and the security strategies employed to tackle these challenges according to the aforementioned two approaches. As we highlight the disadvantages of these approaches in terms of security, reliability, controllability and performance we present a newly emerging approach in SDN, stateful applications in the data-plane, that takes the best of both approaches by offloading the processing of stateful applications from the controller to the data-plane. Additionally, we highlight in section 2 the challenges that still exist in this new framework with what regards to (i) the processing burden distribution in the data-plane, (ii) the correctness and fault tolerance mechanisms of the application, (iii) and the security of the network. In section 3 we present our proposed framework (SDFS) that builds on top of the stateful data-plane framework to address issues not only on the security level, but also on the correctness, reliability, and high-availability of SDN applications. Specifically, we present in this section SDFS's potential to transform the network into a one big firewall as it offers an optimized processing burden distribution of the stateful application in the data-plane with inherent fault-tolerance mechanisms that eliminate the need for immediate controller intervention even in cases of failure or attacks on the network. Evaluations of this framework are presented in Section 4. Finally, a conclusion and future work are presented in section 5.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Security in SDN

The SDN paradigm has already proven its success in diverse deployment scenarios such as Google's backbone network [5] and Microsoft's public cloud [6]. While this trend is set to continue, the security area of SDN has gained minimal attention compared to other areas.

There are obvious security advantages that are entitled to the SDN architecture. The centralized intelligence allows for a complete view to analyze all of the feedback from the network. Hence, the controller can virtually act as a global anomaly-detection system. Additionally, security policies to prevent an attack can be programmed and optimized on the controller then propagated to the whole network [7].

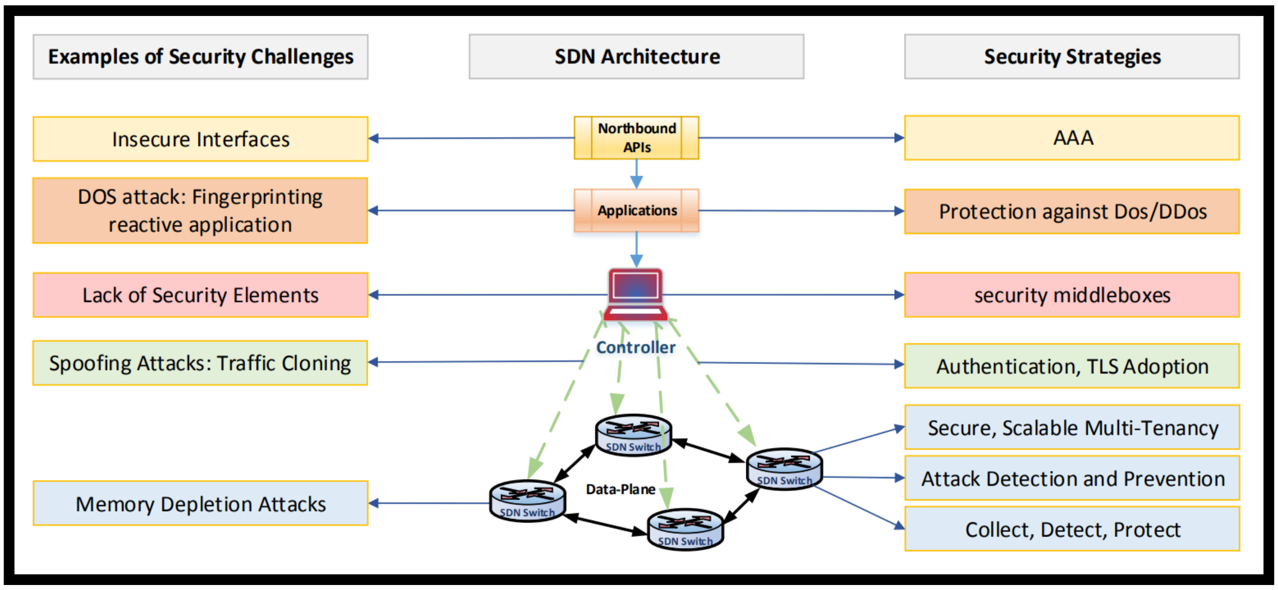

Examples of Security Challenges
SDN Architecture
Security Strategies
Insecure Interfaces
Northbound APIs
AAA
DOS attack: Fingerprinting reactive application
Applications
Protection against Dos/DDos
Lack of Security Elements
security middleboxes
Spoofing Attacks: Traffic Cloning
Controller
Authentication, TLS Adoption
Secure, Scalable Multi-Tenancy
SDN Switch
Data-Plane
Attack Detection and Prevention
Memory Depletion Attacks
Collect, Detect, Protect

Figure 3: Examples of security challenges and strategies in the SDN architecture

On the other hand, as depicted in Figure 3, SDN opens the door to a broad variety of new security challenges. From the perspective of security in SDN, each layer (application, control, data) in the SDN architecture is subjected to different challenges along with challenges that arise in the inter-layer communications. Some of these challenges are specific to SDN, not only due to the particular technologies and protocols employed in the SDN frameworks, but also on a more structural and conceptual level such as the architectural centralization of the controller in the network, and the limitations imposed on the data-plane dummy switches to maintain a simple and programmable behavior. Other challenges are common across different network types. But also due to the SDN architecture, these common challenges can have an augmented impact on the SDN network and accordingly they require special considerations [27].

2.1.1 Challenges in the data-plane

Although the structure and processing mechanism of the SDN switch is fundamentally different than that of a traditional switch, the SDN switches can still be susceptible to traditional attacks. Since the SDN switch does not impose a limit on the number of rules in its flow tables, it can be a subject to **memory depletion attacks**, especially when the switch employs learning flows such as the case of data-plane MAC learning application. Accordingly, packet forging attacks can be performed to overload the switch with flows corresponding to fake MAC addresses [27].

On the other hand, attackers can attempt to guess the installed flows in the switch and forge packets to forcefully increase the counters of the flow rules. **Artificially altering flow statistics** can compromise the optimality of load balancers, for example, or the integrity of billing systems, where a customer will be charged for more traffic [27].

2.1.2 Challenges in data-plane to control-plane communication

Along with the lack of SDN best practices, a network comprised of dummy programmable switches gives rise to an authenticity challenge between its network elements in the data-plane. The controller-switch communication can be jeopardized due to the lack of authentication mechanisms such as TLS adoption. This makes the network vulnerable to **spoofing attacks**.

As such, an attacker could spoof the controller and tamper with the OpenFlow rules on the switches. It would only take the attacker a few seconds to install rules on all of the switches to serve their malicious purposes, an example of which is traffic cloning. These sort of attacks are very hard to detect unless flow rules on all switches are constantly being monitored by the controller [27] [28] [29].

2.1.3 Challenges in the control-plane

The fact that the architecture of SDN networks is centralized, renders the entire network vulnerable if the controller is vulnerable. Controllers that are not logically and physically protected adequately could be exploited by adversaries, much like the lack of security elements such as firewalls and intrusion-detection systems makes other types of network devices exploitable. For example, a vulnerable administrative computer that is directly connected to the control network could grant the attacker the resources and proximity required to launch a wide variety of intrusion attacks and gain access to the controller node and, consequently, the entire network [27] [32] [33].

2.1.4 Challenges in the application layer

Similar to gaining access to the controller, an attacker that can manipulate a controller application can also manipulate the network on a global level. For example, applications can be manipulated to induce the installment of malicious flow rules on the switches to mirror traffic for information theft purposes.

On the other hand, the attacker can exploit the vulnerability of the applications as they respond to the data-plane events; More specifically, the application's vulnerability to Denial of Service attacks [7]. In such a setup, the application is a reactive application, where it executes a certain action when it receives a certain OpenFlow notification from the data-plane switches. Since this setup is predominantly used in the SDN environment, the attacker can perform a **fingerprinting attack** to discover such a vulnerability. For example, the attacker can investigate the latency of the first packet being sent to infer that the application hosted on the controller is reacting to a certain type of packets. Accordingly, the attacker can then launch a Denial of Service attack on a particular application and accordingly crash the whole controller [27] [28] [30] [31].

2.1.5 Challenges in the management plane

Insecure northbound interfaces can provide easy access for adversaries to accomplish an escalation of privileges. Such privileges can range from controlling certain application configurations to quitting the controller process as a whole. On the other hand, upon controller failure, lack of monitoring resources can compromise the debugging process and accordingly impede the recovery process for bringing the network back into a healthy operational mode [27] [28].

2.2 Security Strategies

To tackle the security challenges mentioned above, the authors in [7] classified the potential security enhancements into six categories. Each of these categories tackles a specific type of challenges and in most cases it is tailored to protect a certain area in the SDN architecture as shown in Figure 3.

The first category is **Collect, Detect, Protect**. In this approach, the controller extracts the statistics revealed by OpenFlow about the installed flow rules such as, but not limited to, the number and rate of packets matching each rule. This stage is labeled as the collection stage. The controller can then utilize an intrusion detection intelligence system to globally detect any suspicious interaction in the data-plane, and can protect the network elements by updating the flow rules accordingly [8]. Examples of the successful implementation of this strategy are the Learning-IDS in [9], NetFuse in [10], and OrchSec- an orchestration based architecture [11].

The second category is **Attack Detection and Prevention**. This approach is similar to the first one but is more attack-type specific. In this approach the controller installs specific flows to force the switches to send certain packets to the controller. An example of that would be the first packet of every new tcp connection. The controller, instead of extracting all of the statistics from the switch, could employ its intelligence to detect networking breaches exploiting a certain specific attack. The attack is prevented by adding flow rules to drop the detected malicious packets. This strategy has been employed in OpenWatch [12] and FleXam [13].

Protection against **Dos/DDos** attacks is presented as a separate category. In this approach, the controller regularly retrieves statistics about the average packet per flow for specific flows only. The controller turns to these flows to monitor traffic that is being forwarded to a server hosted in the network. Whenever the rate exceeds a certain threshold, an attack is detected by the controller.

The fourth category is characterized by introducing **security middleboxes** in the network. While this approach is extensively utilized in traditional networking, its integration in SDN networks has been a topic of extensive discussion. In [14], the authors proposed the Slick architecture which utilizes a controller that is responsible for migrating security functions into custom middle boxes. In [15], the authors proposed the FlowTags architecture that modifies middleboxes to interact with an SDN controller. As simple as this solution might seem, it has major downsides. The authors in [16] pointed out three drawbacks of this approach. In the first place, hardware firewalls are very expensive. Secondly, it is very hard to achieve, in contrast with the spirit of SDN, interoperability between the vendor-specific firewalls. Third, failures of firewalls in this scenario will require replacement and reconfiguration of multiple firewall deployments which can inflict a major down-time and produce policy inconsistency in the network.

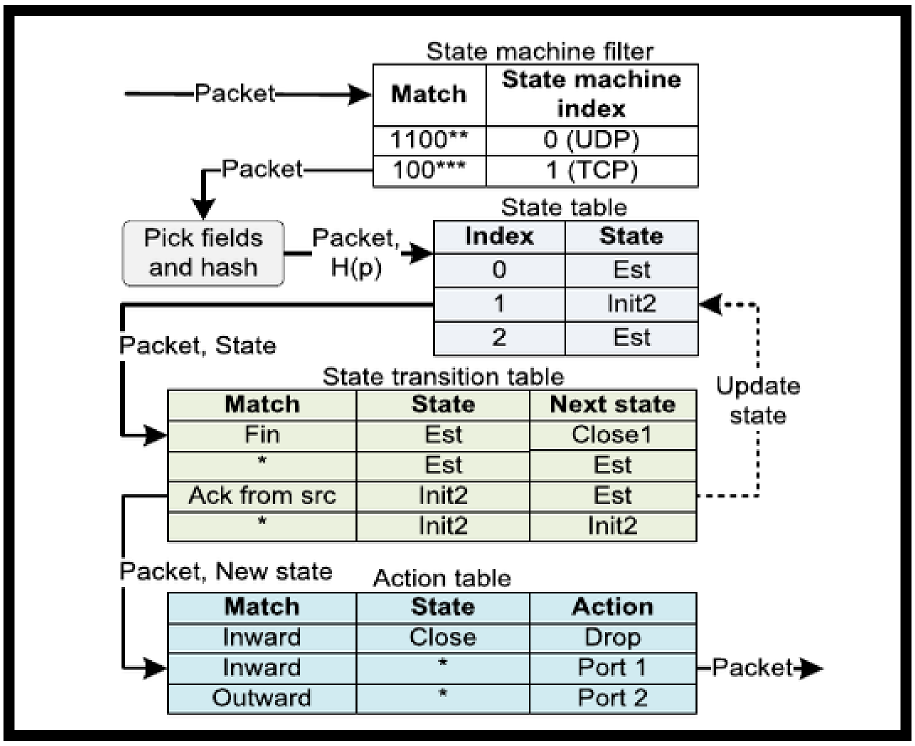
The unauthorized access challenge was tackled by the **Authentication, Authorization, and Accounting** mechanism that comprises the fifth category. SDN-driven AAA was integrated as part of the Application layer on controllers such as OpenDaylight and Floodlight [17].

Utilizing SDN’s potential to dynamically configure virtualized logical networks opened the way for the sixth security category: **Secure, Scalable Multi-Tenancy**. With this approach, the controller can maintain a logical traffic-isolation hierarchy in the data-plane through logical classification and tunneling which allows individual processing of traffic streams to control security vulnerabilities [7]. An implementation of this approach is presented in [18].

In all the above strategies, any attempt to augment the SDN network with security enhancements cannot but extensively rely on the controller as it is the only intelligent unit in the architecture. Many of the above mentioned challenges are intrinsic to the aforementioned SDN architecture itself. With any security function that has to be applied on the network, traffic must converge to the controller and the controller must update the network reactively. In a small SDN network, this can seem fairly practical. It is detrimental, however, for a large SDN network, in which, the controller can be easily overwhelmed with the large influx of traffic to be scrutinized at the application layer. Consequently, a new approach emerged that embraces the idea of discarding the fully centralized architecture of SDN and migrating part of the intelligence to the SDN switches.

2.3 StateFul OpenFlow Approach

As mentioned earlier and to avoid burdening the controller, new approaches were suggested where the regular switches take part in the decision making. In [19], the authors observed that the dummy network switches have, in fact, the potential for locally hosting a state machine as an aggregate of flow rules. This brought into attention OpenVswitch's **Learn** action which is able to install a new flow when a packet matches an old flow. Accordingly, a stateful OpenFlow paradigm was suggested as a new SDN switch primitive [19]. This new approach allows the controller to proactively install flow rules that can express a local state-machine. The state-machine can then dynamically react to local network events and report back to the controller. FAST [19] is an example of the implementation of this approach.



State machine filter

Match	State machine index
1100**	0 (UDP)
100***	1 (TCP)

State table

Index	State
0	Est
1	Init2
2	Est

State transition table

Match	State	Next state
Fin	Est	Close1
*	Est	Est
Ack from src	Init2	Est
*	Init2	Init2

Action table

Match	State	Action
Inward	Close	Drop
Inward	*	Port 1
Outward	*	Port 2

Figure 4: “Implementing TCP state machine in FAST data plane” [19]

To track tcp connection states, FAST utilizes four flow tables as shown in Figure 4. To demonstrate the behavior of FAST’s stateful implementation, Figure 5 below represents a network where switch:green employs the FAST framework to perform connection tracking for host:red. TCP connections from the cloud to host:red are only allowed if and only if host:red initiates the connection.

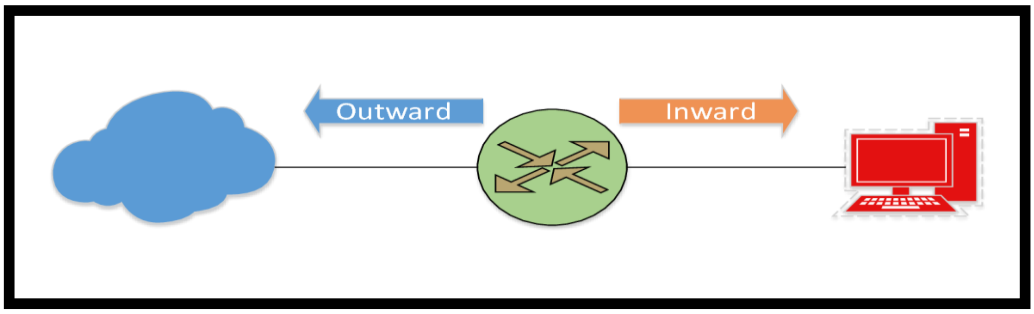


Figure 5: Simple network example for connection tracking using FAST

- Initially, when the packet arrives at the switch, irrespective of its direction, it is matched against the filter table, in which only the tcp packets are re-submitted to the state table.
- The state table holds information about the mapping between every seen connection and its state. In the **state table**, FAST employs the OpenFlow Packet Metadata to tag the packet with the current state of the connection the packet happens to be in. Since the connection is defined by the tcp packet's layer-3 and layer-4 fields, the packet is matched according to these fields to choose the connection, and accordingly the packet's metadata is set to either "established, initialized, or closed." As the tagged packet now holds the state of the connection it is in, the packet is then re-submitted to the **state-transition table**.
- In the **state-transition table** the packet is matched against its tcp flags and accordingly the **state table** is updated using the Learn action. For example, if the packet's metadata states that the packet is in an "established connection" state, and the tcp flag is set to FIN, then the corresponding entry for the connection in the **state table** is updated to become closed, since a FIN packet would have been seen for this connection. On the other hand, the packet continues its way to the **action table**.
- In the **action table**, the packet is matched against its inport to determine whether the packet is going inward (to host:red) or outward (to the cloud). If, for example, the packet is going inward and the connection is closed, this packet is directly dropped since no host is allowed to communicate with host:red. On the other hand, if, for example, the packet is going outward, the packet is forwarded accordingly.

In contrast to the **Attack Detection and Prevention** approach, packets are not required to be tunneled back to the controller in order to keep track of the tcp connection states. These states are tracked down locally at the switch level. It is quite legitimate to argue that maintenance and processing of the local states will pose a performance overhead on the switch. However, the security advantages are granted on several levels. In the first place, this overhead is substantially less than that required to involve the controller in maintaining the state of every tcp connection in the network. Secondly, this approach maintains the correctness and consistency of time-sensitive applications where the time required to communicate with the controller to update the states can be detrimental. Additionally, such a framework radically protects the controller from DoS attacks. Since switches do not have to notify the controller upon events of state updates, the controller is spared this responsibility and is protected against adversaries that attempt to forge packets to artificially exhaust the controller.

In line with the ever-increasing demand for a faster high-performance data-plane, research efforts continued to focus attention on augmenting the data-plane's capacity to handle more intelligent functionalities. More intelligence offloaded from the controller to the data plane means lower cost on performance due to less switch-controller communication.

This approach is at an advantage as long as it does not fundamentally sacrifice the security of the network. While it can be seen that this approach is not compliant with the spirit of SDN in general, i.e. Intelligent control-plane and dummy data-plane, offloading the state-management burden to the switches does not mean that the switches can now take decisions irrespective of the controller. The state-management structure is still programmed by the controller itself. On that account, the real decision maker is still the central controller, with its decisions formalized into data-plane processes [22].

In [20], the authors pushed this approach a step further to include a basic firewall functionality as a policy table. Along with a **state table**, the **policy table** sets some extra rules for communication in the network. For instance, the controller can allow any communication between node A and node B if and only if node A initiates the communication. This is translated in the flow pipeline in the switch as follows: Each packet is matched against its inport, if the packet is coming from node A's side, the state is updated to resemble an open connection. The metadata of each subsequent packet whether from node A or B is set to resemble an **Allow** state. When re-submitted to the policy table, this **Allow** state passes the policy test and is forwarded accordingly. However, if node A never sent a packet to node B, the state would have remained in a **Deny** state and any subsequent packet from node B would have been dropped by the policy table.

In [21], the authors addressed the stateful OpenFlow approach for applications with requirements that extend beyond the local state-machines on the switches. They argued that a pure stateful firewall approach cannot make use of the controller potential which has a global view of the whole network, unless the controller regularly fetches all the OpenFlow statistics revealed by all the switches in the network - as described in the **Collect, Detect, Protect** security enhancement category.

In an attempt to pursue a global connection tracking system, [21] proposed a state tracking framework, STATEMON, which comprises a **global state table** to keep records of all the active connections in the network and a **state management table** that controls the state transitions for every connection. Both tables, however, are hosted on the controller. Although the states are still operational locally on the switches, these states are only managed and updated by the controller. That is, whenever there is a state-changing packet, the packet is forwarded to the controller to keep track of this event and update the local state on the switch. For that purpose, they augmented the OpenFlow protocol with **OpenConnection** protocol that helps the switch send the state-changing packets to the controller accompanied with further signaling information. The OpenConnection table entries are only installed on the switches that are directly connected to the endpoints of the communication. Although this framework allows for a global stateful view of the network, the control-data layer interaction is still excessive and poses a variety of intrinsic risks on the network. Also, most of the processing burden is carried by specific switches in the network which creates a bottleneck in the data-plane. This type of frameworks attempts to avoid OpenFlow's limitations through a two-tiered framework: A central intelligence that manages the states of the network, and a stateless data-plane for packet forwarding. It follows that such a framework is at an advantage for applications that are dependent on global network states and do not have real-time processing time restrictions. However, this will fold down into a dysfunctional framework for applications that require wire-speed forwarding of packets as it is intrinsically dependent on the remote decision maker, the controller. Additionally, the controller is at a high risk of DoS attacks especially that it is easy for an adversary to fingerprint the application in the data-plane to guess information about the flows and accordingly forge packets to exhaust the controller.

Many more proposals proceeded in materializing the stateful data-plane framework on different levels, from on-switch state management to high-level controller APIs and compilers. In [25] for example, the authors proposed an abstract data-plane programing language, the P4 language and its corresponding compiler, that takes into account new header fields and in-switch registers which values persist, are matched against, and manipulated as packets traverse the OF-tables in the switch. As highlighted in [22], utilizing the in-switch registers and arrays for state-table generation is being taken seriously by the Open Networking Foundation (ONF) to incorporate the stateful framework in the OpenFlow standard. [25]’s abstract model, however, is not restricted to OpenFlow switches but is data-plane independent. This means that P4 programs can be mapped onto a various set of device types (OVS, FPGA, ...). Additionally, the P4 abstraction introduces a programmable packet parser [34] handy for analyzing and matching against new header fields. Moreover, as opposed to OpenFlow’s sequential processing of packets, While P4 allows for the programmability of parallel selection and processing of match/action rules, it does not address the issue of distributing this processing burden on switches other than the direct switches. Additionally, the extensive reliance on new headers in the network breaks the modularity of this framework and makes it extremely hard to integrate with other applications and modules.

Another idea that addressed the programmability of stateful data-planes is found in [24]. In 2014, OpenState presented an extended FSM framework, a Mealy Machine, that comprises a four-tuple (S, I, O, T) programmability model. In this model, the S symbolizes the set of states, I is the set of inputs/events, O is the set of outputs/actions, and T stands for the state transition rules. Similar to the frameworks discussed before, these rules provide a mapping between pairs of [state,event] and pairs of [state,action]. The implementation of this framework consists of two tables. The first is the State Table which holds information about the current state of the application, and the second is the extended FSM (XFSM) table which describes the transitions. Whenever a packet is received, a lookup in the State Table is used to determine the state in which the packet is in. This state is appended in the OpenFlow Metadata of the packet. The XFSM table then uses the metadata to determine the specific rule it should match against and applies the transition. The transition output includes updating the State Table with new states, and updating the packet fields before forwarding.

Similar to OpenState and Fast, SDPA [23] is a stateful data-plane management scheme that uses state and state-transition tables. However, the actions are decoupled from the state-transition table and reside in a separate table. The significant difference is that this framework allows for the programmability of multiple stateful applications, with each application hosted in its own triplet of (state, state-management, action) tables. This is made possible by employing a “Forwarding Processor” (FP) that sends the first packet of a flow to the controller and receives table configuration information by which the FP initializes these tables for the specific application. This procedure underlines the controller’s thorough control and management of the applications being processed on the data-plane.

While both OpenState and SDPA exhibit the drawbacks that exist in FAST in terms of being constrained to the direct switch, SDPA’s dependency on the controller to populate the states in the switches upon initial packet events leaves the controller easily permissible to DoS attacks through packet forgery.

In [26], the authors proposed Stateful Network-Wide Abstractions for Packet Processing (SNAP), a programming language and a compiler that abstracts the data-plane into one big switch. This abstraction treats local state variables as global variables in such a way that the programmer does not have to worry about where these variables actually reside in the data-plane while writing the network’s stateful program. SNAP’s compiler uses its optimization algorithms, based on Mixed Integer Linear Programming (MILP), to store these variables in specific switches in the data-plane ensuring that the packets will pass through these switches. This is a very powerful tool to avoid the constraints of only hosting the state variables on the direct switches to the hosts, as appears in StateMon, OpenState, Fast, etc… Such a framework allows the utilization of more switches in the network to host the state variables. Not only this, but SNAP's optimization algorithms can divide the state variables into several parts and distribute them across various switches provided that packets belonging to the same application instance can still traverse these switches in the required order to maintain a consistent update of the states of the application. In order to exchange state variables between switches along the path of the packet, SNAP utilizes special header fields. These header fields contain a dictionary of variables and their corresponding values which are used in the switches to match the states.

One of the examples considered in [26] is the DNS-tunnel-detection. In this program, the compiler decides at first where the state variables will be placed in the network (decide which switch will host the variable) and then decides how packets should be routed to meet these variables. In this process, the compiler must ensure that the packets will pass through these switches in the correct order. This means that the compiler must have explicit information about the paths of packets in the network in order to correctly place the network variables in the switches. For example, the compiler must know that packets being initiated from an internal server will leave the network from a specific gateway following a specific path in the network. The compiler can then choose a switch on the path to host one of the network variables corresponding to this connection. For this reason, SNAP's optimizer forces its own routing on the network. Accordingly, the firewall applications are directly coupled with the routing applications which violates the modularity of the proposed framework. This case becomes more complicated if load balancers are to be be employed in the network. If such path information is not clear to the controller, the compiler can only fold back to storing the global variables on the direct switches, ensuring route convergence onto the network variables at the expense of giving away the ability to utilize the internal network switches for storing and processing the states.

On the other hand, changes in network topology might necessitate a re-compilation of the SNAP program to maintain the consistency of the state-variable placements in the network. Such cases will have detrimental implications on the state of the network whenever the route paths diverge from the switches carrying the state variables, which might happen when ports or switches are down. In such cases, all traffic in the network depending on variables in a particular switch will be dropped or misdiagnosed in the states of the network. This forces a major downtime on the network until the program is recompiled at the controller and rules are reinstalled in the network. Although SNAP does not implement fault-tolerance mechanisms, these problems are not inherent to SNAP but exist in other solutions that do not inherently employ state replication. Table 1 includes a summary of all the features exhibited by the stateful approaches (including ours).

Table 1: Summary of Stateful SDN Data Plane Schemes

Scheme	**Category**	**State Storage**	**State Updates**	**data-plane Utilization**	**Cost aware distribution**	**Modularity for path convergence**	**Fault Tolerance mechanisms**
Fast [19]	platform	Hash table	local	Direct switch	N//A	N//A	Exhibits single point of failure
OpenState [24]	platform	Hash table + TCAM	local	Direct switch	N//A	N//A	Exhibits single point of failure
StateMon [21]	framework	CAM	controller	Direct switch	N//A	N//A	Exhibits single point of failure
SDPA [23]	platform	TCAM + SRAM	controller	Direct switch	N//A	N//A	Exhibits single point of failure
P4 [25]	Compiler and Programming Language	RAM or TCAM	According to implementation of application	Direct switch	N//A	N//A	Exhibits single point of failure
SNAP [26]	Framework	Hash table or CAM	local	Distributed with limitations	Optimization based on link costs	Takes over the routing application Dictionaries in header fields	Exhibits single point of failure Requires controller intervention. Involves recompilation
SDFS (our approach)	Framework	CAM	local	distributed	Optimization based on link costs and switch processing capacity	Separate and pluggable	Inherent maintenance of application correctness in the data-plane

As shown in Table 1, most of the frameworks in the literature rely on the direct switch to host the states in the data-plane (data-plane Utilization column). While some of them still depend on the controller to perform state-updates in the data-plane, others have localized these state-updates at the data-plane to minimize the time and performance-exhaustive communication with the controller and accordingly secure the time-sensitive correctness of the application. Among the frameworks in the literature, only SNAP utilizes the indirect switches in the data-plane for a burden distribution of state updates at the expense of taking over the routing application in the network. However, due to the distribution of states in the data-plane based on the state variables, single points of failure still exist as switches are still strictly dependent on each other to consistently manage the state updates and failure of one switch, that hosts a state variable, to perform variable updates will result in the whole application behaving incorrectly. Upon such failure in the SNAP framework, a controller intervention is required to update the network involving recompilation and reoptimization at the controller which might not be convenient for time-sensitive applications and gives the attackers enough time to perform malicious jobs in the network. To address all the above issues, we propose SDFS, which has the following properties: (1) all state updates are done locally, (2) burden is distributed among switches in an optimal fashion based on mathematical formulations, (3) modularity is maintained, and (4) fault tolerance is achieved. In section 3, the details of SDFS are covered.

CHAPTER 3

PROPOSED FRAMEWORK: A DISTRIBUTED SDN STATEFUL FIREWALL

The literature behind stateful firewalls in SDN seems to have taken a long leap in the last few years. Although major progress in research has taken place, there is still much that must be done in order to perfect an efficient and reliable design. [26]'s SNAP took the SDN stateful firewall to a new level with a framework that abstracts the network into one big switch and offloads the processing burden of the network states from the controller to the switches. On the other hand, this framework still suffers from single points of failure and introduces constraints on the routing behavior of the network to satisfy the requirements of the stateful application in the data-plane. Additionally, it does not provide fault-tolerance mechanisms that are critical to maintain the behavior of the application in the data-plane in cases of failure in the network which gives an opportunity for the attackers to tamper with the network.

In an attempt to create a complete scalable and reliable solution and to tackle issues such as the ones inherent in SNAP, we present SDFS, a new framework with the following properties:

1. Abstracts the network into one big firewall at the management plane
2. Uses an intelligent algorithm to distribute and balance the burden of stateful firewall processing over multiple switches in the network
3. Accounts for weighted paths and network element capacities through a cost-capacity-aware optimization.
4. Relaxes the constraints on requiring path information in the network

5. Maintains modularity: independently plugged into the controller without disrupting other applications
6. Inherently implements fault-tolerance mechanisms that bring forth an application with High Availability
7. Yields a controllable stateful application in the data-plane with minimal controller intervention

Approach:

In order to accomplish the goals set for SDFS, many factors and constraints have to be considered. This section addresses the approach to each one of these considerations. Some of these considerations tackle the low level implementation of the states at the data-plane such as the storage (Section 3.1) and structure (Section 3.2) of the states. Section 3.3 addresses the traffic convergence constraints for the application in the network. These constraints are dependent on both the required application logic and the routing behavior in the network. Section 3.4 addresses the cost-capacity-aware optimization of the burden distribution in the data-plane. Section 3.5 addresses the framework designed to accomplish high availability of the application in the data-plane. Section 3.6 presents the mechanism employed to exchange tracking information between the network switches without the need to communicate with the controller. Section 3.7 presents an example implementation of a connection-tracking application that employs the discussed frameworks. Sections 3.8 and 3.9 address the modularity of the discussed frameworks and accordingly present relaxations on the constraints to attain modularity.

3.1 Storage of states at the data-plane

State variables are data hosted on the data-plane switches that are persistent over multiple packets. These variables are fundamentally values that correspond to packet fields or constants. OpenFlow offers several efficient techniques to achieve such persistence of information. One approach is to represent the state variables by flow entries that are learnt reactively. This is because flow entries can hold persistent information in their match and action sections such as specific values of packet fields and packet metadata. An example of this is the OF-MAC learner where a flow is learnt whenever a new mac-address is found in the network which allows the switches to automatically respond to ARP requests without consulting the controller. In this case, the new flow holds the required information of the state in its match and action sections: **match on** ip-address, and **respond with** mac-address of the found host.

Another mechanism is to employ the arrays of registers that are already supported in emerging software switches such as OpenVSwitch, where eight, 32-bit registers are provided. The values in these registers can be matched against once the packet is processed. Additionally, they are updatable using actions that fall neatly in the OF match-action framework [22]. In this technique, packet fields can still be represented in the registers as hash values. Then, whenever a switch needs to match a packet against a state that holds specific header values, say [SourcePort - DestinationPort] pair, it computes a hash value from the arriving packet's fields and matches it against the corresponding register.

Both of these techniques are fairly simple to implement. Although they can be extremely low level operations, many solutions, such as [25]'s P4 compiler and language, are already provided to abstract this procedure into clean APIs.

3.2 Structure of states at the data-plane

State instances in the data-plane can be comprised of a single state variable- where the variable represents the whole state, or multiple variables where the combination of these variables represents the state. In the case of a multivariable state, the state is really an abstraction of multiple low level single-variable states that assumes atomic procedures with regards to these low level states. An example of multi-variable states is that provided by SNAP as **DNS Tunnel Detection**. The abstract high level state holds two pieces of information: (i) The resolved ip-addresses in the network by client-x, and (ii) a counter that tracks the number of resolved ip-addresses that client-x does not use. Each one of these pieces is represented by an array on the data-plane. These arrays are updated consistently as packets traverse the network and together represent the global state of the network.

As discussed in the literature, [26]'s state distribution framework attempts to distribute the state variables over the network switches. Accordingly, there arises the burden to maintain atomic consistency of the low level states that spans multiple switches. Although it is a great attempt to provide a margin for burden distribution over the network switches, it is rather deviant from the end result. This is because distributing the state variables over multiple switches introduces additional constraints over the network in terms of atomic consistency and packet path convergence, as elaborated upon in section 3.3, as well as additional communication costs where the packets have to carry dictionaries of information in their header fields to exchange state data [26].

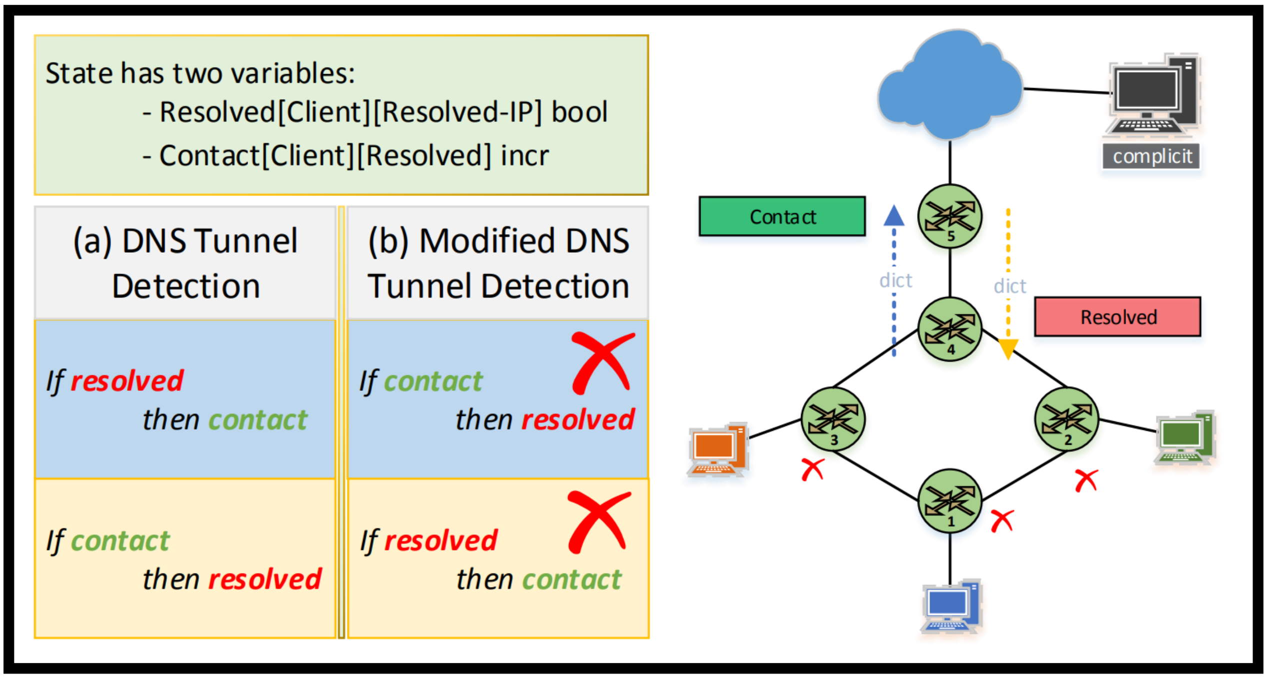


Figure 6: Unidirectional Variable Relationship Constraint

Another constraint that is introduced by distributing the state into state variables instead of state-instances is a commitment to unidirectional conditional variable relationship. As shown in Figure 6 above, we employ the same network in two examples of the DNS Tunnel Detection application where the state is distributed into two variables according to the SNAP framework: **contact** and **resolved**. In these examples, we host the **contact** variable at switch:5 and the **resolved** variable at switch:4.

Figure 6 (a) illustrates the network under the application as described by SNAP. In this example, as the DNS response for (google.com), for instance, arrives from the complicit DNS server (yellow path) to the hosts, say host:green, when the packet arrives at switch:5, **contact**[green][8.8.8.8] is incremented and forwarded to switch:4 where **resolved**[green][8.8.8.8] is assigned as True. Subsequently, as host:green contacts 8.8.8.8 (blue path), when the packet arrives at switch:4, the **resolved** variable is appended in the header as a dictionary and then forwarded to switch:5 which then checks the **resolved** variables in the dictionary and accordingly makes the decision whether to decrement **contact**.

The problem however arises as described in Figure 6 (b) where the application is slightly modified to have a conditional on the **resolved** variable in the yellow path direction to change the value of **contact** that is hosted on switch:5. In this case, when the packet arrives at switch:5, the resolved variable is not yet available in the header of the packet to perform the conditional on. This is because the resolved variable is hosted on switch:4 which falls after switch:5 in the yellow path direction. Thus, for such applications, the engine fails to distribute the burden upon the switches and regresses back to hosting all the variables on the same switch.

SDFS casts aside these unnecessarily introduced constraints by ensuring that atomic state variables be hosted on the same switch, and the burden distribution is a cost-capacity-aware optimization at the level of state instances as a whole rather than state variables. As such, in contrast to dividing the state, in the **DNS Tunnel Detection** example, into two variables hosted on different switches (array for resolved ip-addresses by the clients and another array to keep track of the resolved ip-addresses that are not used by the clients), SDFS divides the state into state-instances hosted at different switches where each instance covers all the variables required for one client.

3.3 Path Convergence Constraints

The problem of path convergence onto the state instances arises as an inherent constraint to any distributed stateful data-plane solution. No such solution can be correct if it does not guarantee the correctness of states in the data-plane. To accomplish correctness of states in the data-plane, all state-updating packets must pass through the switch that hosts the corresponding state instance. Fundamentally, the convergence of packets on their corresponding state-hosting switch is directly coupled with the routing application and procedures employed in the network.

Assume for example we want to host states at the data-plane for a port-knocking application. For the sake of demonstration, assume the network in Figure 7 below where host:orange is required to knock a sequence of ports in the network to access server:blue.

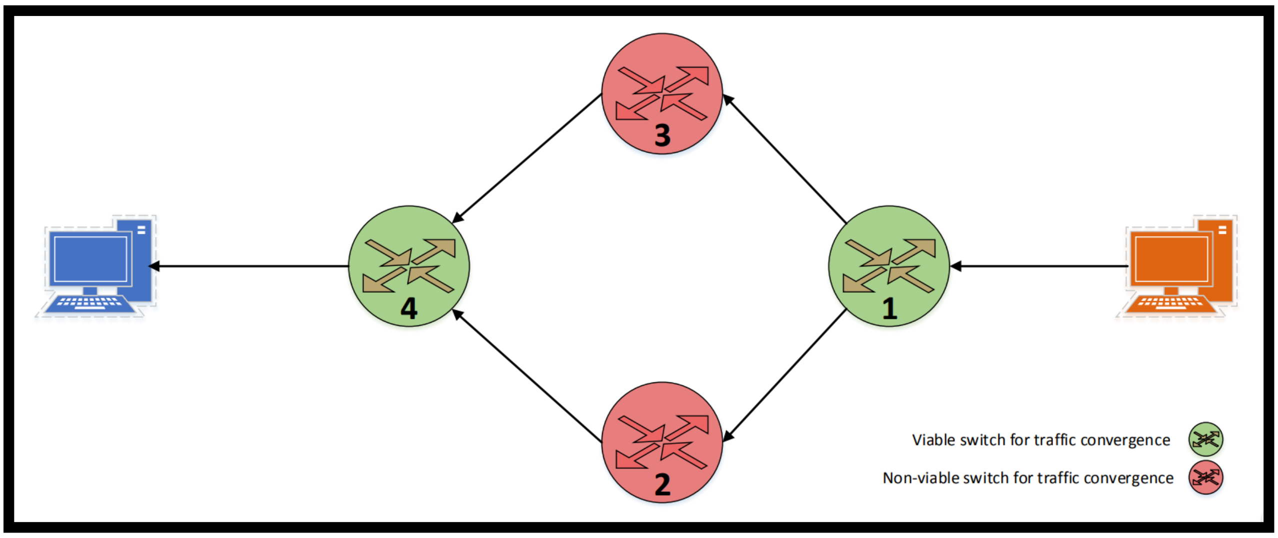

3
4
1
2
Viable switch for traffic convergence
Non-viable switch for traffic convergence

Figure 7: Port-knocking example for convergence constraints

In this example, if the port knocking state instance hosted on one of the switches is to maintain its correctness, packets that are necessary for updating the state must converge on this switch. For this topology, it is easy to see that switch:1 and switch:4 satisfy this requirement, since all packets originating from host:orange destined to server:blue will pass through switches 1&4. Accordingly, if the state instance is hosted on either switch:1 or switch:4, we can ensure that all packets carrying the port knocking sequence will converge on the state-hosting switch.

Additionally, if we were provided with the routing information of the network, we can then realize which path the packet takes from host:orange to server:blue. If packets take the upper path, then switch:3 can be a convergent switch. If packets take the lower path, then switch:2 can be a convergent switch.

Some cases can arise where packets necessary for the port knocking application converge neither on switch:2 nor switch:3. If we introduce a Layer-4 load balancer at switch:1, then a proportion of the packets will take the upper path, while others will take the lower path. Since a layer-4 load balancer diverges the packets using packet fields up to layer-4, then we cannot be sure that all the packets carrying the port sequence will go through the same path, and thus we cannot consider that switch:2 or switch:3 can be convergent switches if such a routing behavior is employed in the network.

Two main approaches can be pointed out to address this issue, each having its own advantages and disadvantages:

Path Information Extraction

In the first approach, the controller extracts network paths from the routing application or from a network monitoring application. The controller then projects these paths onto the data-plane state requirements in order to arrive at a set of switches that are qualified to be convergence points for each state instance. This approach maintains the modularity of the application with regards to routing since it does not override the behavior of the routing application at a fundamental level. However, the firewall application remains limited to what is offered by the routing application. In other words, the convergence points in the network might not be optimally established and distributed through the network.

Network Path Manipulation

In the second approach, the firewall module actually overrides the routing module in the network where an optimizer is employed to draw the routing behavior of the network and arrives at an efficient distribution of convergence points for each state instance. Such an approach is adopted in SNAP using the MILP optimizer as discussed previously. While this approach gives full control over the network routing behavior, it fails to subscribe as a pluggable modular framework.

In this section, we will study the convergence requirements of firewall applications with respect to the routing behavior in the network. Particularly, we will study two properties of the data-plane states. The first property corresponds to the state's requisites on the packet direction, and the second property corresponds to the state's requisites on the packet fields. Figure 8 summarizes the convergence constraints and accordingly presents the decision tree to arrive at the optimization framework that either meets or relaxes these constraints.

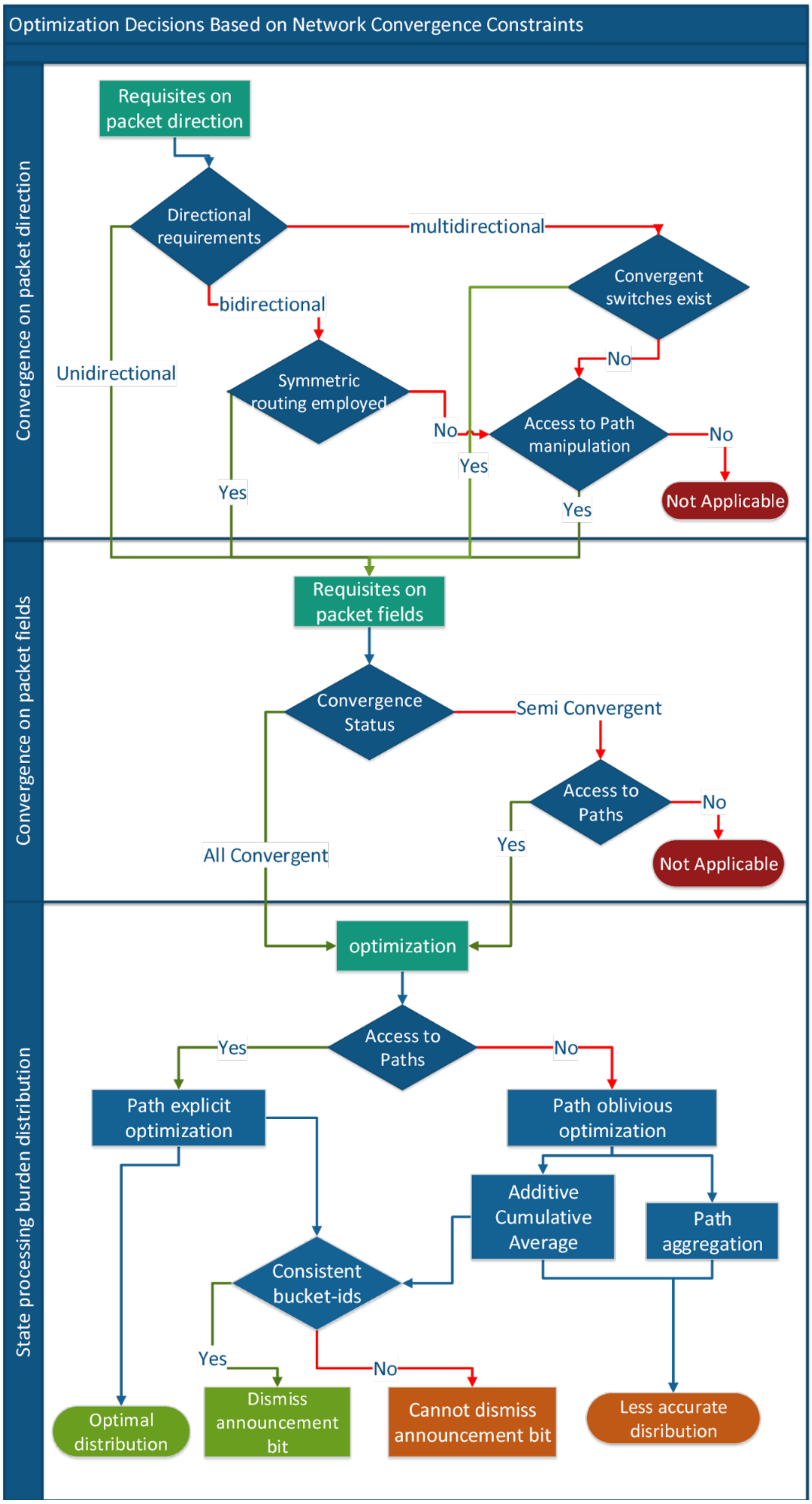

Optimization Decisions Based on Network Convergence Constraints
Convergence on packet direction
Convergence on packet fields
State processing burden distribution
Requisites on packet direction
Directional requirements
multidirectional
Convergent switches exist
bidirectional
Unidirectional
Symmetric routing employed
No
Access to Path manipulation
No
Yes
Yes
No
Not Applicable
Yes
Requisites on packet fields
Convergence Status
Semi Convergent
Access to Paths
No
Not Applicable
Yes
All Convergent
optimization
Access to Paths
Yes
No
Path explicit optimization
Path oblivious optimization
Additive Cumulative Average
Path aggregation
Consistent bucket-ids
Yes
No
Optimal distribution
Dismiss announcement bit
Cannot dismiss announcement bit
Less accurate disribution

Figure 8: Summary on convergence requirements and optimization decisions

3.3.1 Requisites on packet direction:

The literature covers a wide variety of firewall applications. In some of these applications the states are confined to a source-destination pair. Accordingly, these states can either require bidirectional packet information, such as connection-tracking and FTP-monitoring, while others only require unidirectional packet information, such as port-knocking. However, other types of applications might not be restricted to a pair of source-destination, but are multi-directional. An example of these applications is the DNS-Tunnel-Detection addressed in SNAP where three hosts are included in the plot: the original host initiating the DNS request, the DNS server that responds to the request, the host that the request was about and later to be contacted.

States that requisite bidirectional packet information impose more constraints on the convergence points in the network. The reason behind this is that the choice of convergence points (the state-hosting switches) must ensure that the flow of packets in the other direction (response from destination to source) will pass through the convergence point.

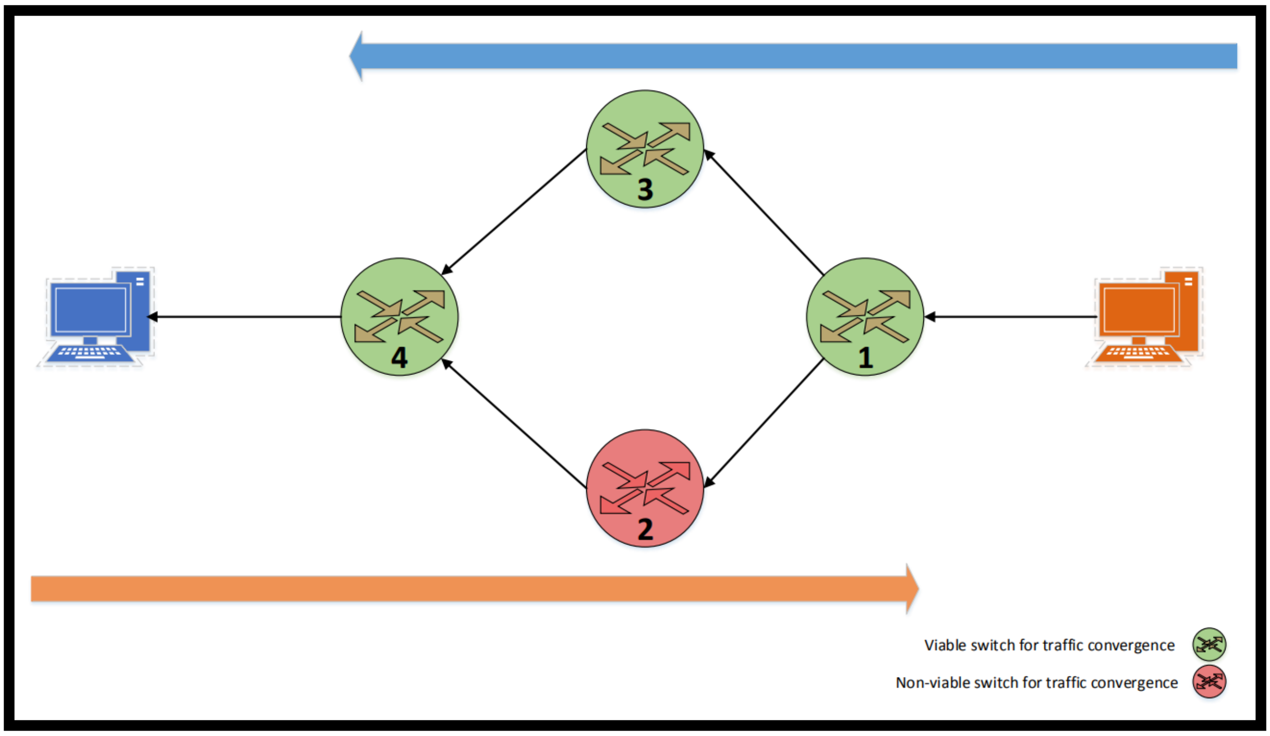

3
4
1
2
Viable switch for traffic convergence
Non-viable switch for traffic convergence

Figure 9: Connection-tracking example for convergence constraints

For demonstration purposes, consider the network in Figure 9 above where we employ a connection-tracking application between host:orange and host:blue. Even if we assume that all traffic from host:orange to host:blue converges on switch:3 by taking the upper path (switch:1 -> switch:3 -> switch:4), it is not granted that switch:3 is a viable convergent switch on which we can host the state of the connection tracking application between the two hosts. This is because the connection tracking state instance hosted at a switch must be able to monitor the traffic in the other direction and accordingly “allow” the traffic back to the protected host. Consequently, if the traffic from host:blue to host:orange takes the lower path (switch:4 -> switch:2 -> switch:1), switch:3 cannot be considered a convergent switch. Thus we will have to look for a different switch that satisfies these conditions, which in this case can be either switch:1 or switch:4.

Symmetric Bidirectional Paths:

This constraint, however, can be relaxed in networks that employ symmetric routing where the response packet takes the inverse path passing through the same switches back to the source. L3 symmetric routes, where routing decisions are based on [Src-Ip, Dst-Ip] pairs, are easy to implement in networks. On the other hand, introducing L4 load-balancing can disrupt route symmetry. Packets can take a different route back to the source. However, in a controlled routing environment, it is possible to maintain symmetric routes in the network while employing L4 Load balancers (or any kind of load balancers), by reactively populating the reverse paths in the network with a simple symmetric routing module that sits on top of the default routing module employing load balancing. This module configures the switches with one extra learning flow that behaves as the MAC-learner: Upon the initial arrival of a packet at the switch, a new flow is learnt to forward packets with the flipped L4 header to the inport. Subsequently, the response packet will match against this flow, given a higher priority, and will follow the inverse path of the request packet.

As summarized in Figure 8, stateful applications that requisite unidirectional flow of packets directly pass the packet direction constraints. These constraints, however, are not directly met for applications that requisite bidirectional and multi-directional flow of packets. If routing controllability is exposed to the stateful application, then this application can manipulate the paths in the network accordingly in order to meet the packet direction constraints. However, if path manipulation is not granted, then only if the network inherently employs symmetric routing the bidirectional information constraints are met. Otherwise, these constraints cannot be met and the application cannot be implemented.

3.3.2 Requisites on packet fields:

These requisites cover the packet fields that are involved in the state variables. A connection-tracking state-instance, for example, stores information about the established connections in the network. As such, the corresponding source and destination addresses need to be extracted from the packet headers and persistently stored and matched against in the switches over multiple packets.

The required packet field information in a state-instance fall into two categories. (i) those that are constant relative to the state-instance, and (ii) those that the state instance sweeps for. If we consider the port-knocking example mentioned above, this can be translated into: for **each** Source-Ip (host-ip) communicating with destination-ip (server-ip), track **all** destination-ports (knocking ports). In this case, each [source-ip, destination-ip] pair corresponds to a single state-instance (category i), while every particular state-instance tracks the different destination ports (category ii). For the state-instance in the example, its category (i) fields comprise the ip-address of host:orange and the ip-address of server:blue, while its category (ii) fields comprise the set of destination ports that host:orange is sequencing. The importance of this classification arises as we discuss possibilities to relax the constraints for convergence of packets upon the state-hosting switches.

All-Convergent Networks

There are cases in which these constraints can be inherently met in the network, in such a way that the network switches are all-convergent with respect to the state-instances. Such conditions can be formalized as follows:

If category (i) required packet fields in a state-instance strictly include all the packet fields by which the routing decision is made, then all of the switches that fall onto the path of a packet are convergent with respect to all the state-instances in the network, provided that the requisites on the path direction are met.

Take for example a network where packets are routed based on their L3 header data, Source-Ip and Destination-Ip addresses. If we want to employ a stateful application at the data-plane that tracks all source-ports used for each combination of Source-Ip and Destination-Ip, then we do not have to worry about the convergence of the packets at the switches. This is because we can be sure that all packets with the same Source-Ip and Destination-Ip will follow the same path in the network. And thus, information about the Source-ports used for each particular combination will traverse the same set of switches on the path. Accordingly, any switch on this path can be a viable convergence point on which we can host the corresponding state-instance. In this example, [Source-Ip, Destination-Ip] packet field requirements in the state are category (i) requirements, and this set strictly includes the packet fields used in the routing decision in the network: [Source-Ip, Destination-Ip]. We should note here that although packets are not routed based on the Source-Ip, the starting point of the path can be directly mapped to the Source-Ip. Thus if we want to follow the path of the packet based on the routing decisions, we need to include the source-ip as the starting point.

A network that employs L3 routing can also be all-convergent for the port-knocking and layer-4 connection-tracking applications where category (i) fields are [Source-Ip, Destination-Ip] and [Source-Ip, Destination-Ip, Source-Port, Destination-Port] respectively. The latter, however, necessitates that the requisites on packet direction are met, because connection-tracking states entail bidirectional flow of packet information.

All convergent networks are not restricted to simple layer-3 routing. A network that employs layer-4 load balancing for example, provided routes are symmetric, can also be all convergent on certain applications including layer-4 connection-tracking where the category (i) packet field requirements of the application strictly include all the packet fields involved in the network routing decision: [Source-Ip, Destination-Ip, Source-Port, Destination-Port].

Table 2 below provides more examples of applications on the topic of convergence with respect to the routing behavior employed in the network:

Table 2: All-convergent application examples according to routing behavior

	Application		
Routing Behavior	Port-Knocking	Conn-Track	FTP Monitoring
Symmetric L3	All-Convergent	All-Convergent	All-Convergent
Non-Symmetric L3	All-Convergent		
Symmetric L4 Load-Balancing		All-Convergent	
Non-Symmetric L4 Load Balancing			

In the next section, we will present a cost-capacity-aware distribution of the processing burden of states on the data-plane. The optimization involved in deciding on which devices to place the state-instances accounts for the path weights in the network along with the capacities of the devices. Additionally, the algorithms employed inherently implement fault tolerance mechanisms to sustain high availability of the distributed application.

Knowing that this framework requires explicit information about packet paths in the network to feed the optimization framework, we will present, in the **packet path oblivious** section (Section 3.9), a relaxation of this constraint where we maintain a fair burden distribution in the network along with the inherent high availability of the application in the data-plane, provided knowledge about the network convergence points.

3.4 Cost & capacity-aware distribution of the state processing burden

The burden distribution of the state processing at the data-plane is scarcely addressed in the literature. Besides SNAP, the frameworks proposed on data-plane state processing all rely on the direct switch framework. Although such an approach offers state consistency in the network elements, it limits the scalability of applications, subjects critical switches in the network to memory-exhaustion attacks, and exposes the network to single points of failure scenarios.

For that reason, state processing in the network should be distributed between SDN devices. Primarily, the distributed processing of states should enhance the overall network performance in comparison to state processing restricted on the direct switch. On the other hand, such a framework should fairly distribute the processing burden. In other words, the framework should attend to the traffic costs in the network along with its device processing capacities. On this account, the distribution should be assessed by clear metrics and criteria. This section will address these issues through a connection-tracking application example.

Direct Switch Framework for Connection Tracking:

The graph below represents a network topology in which the circles represent SDN switches and the PCs represent hosts. Connection tracking is applied for the host:red on its direct switch. That is, if and only if host:red initiated a connection with host:green, host:green can communicate with host:red in the same connection session. Same thing applies for host:orange when host:red initiates a connection with it.

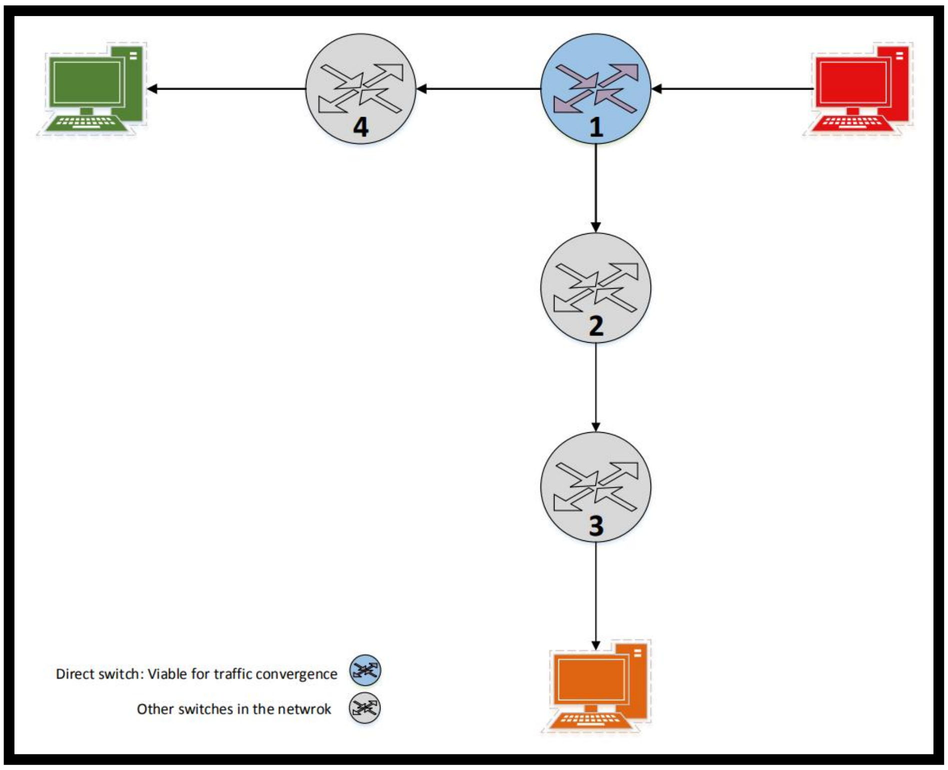

4
1
2
3
Direct switch: Viable for traffic convergence
Other switches in the netwrok

Figure 10: Stateful connection tracking at the direct switch

With the direct switch (switch:1) maintaining the connection tracking, it is the responsibility of switch:1 to allow/deny traffic coming to host:red. This configuration of the network is robust and reliable unless the number of connections initiated by host:red explodes, in which case, switch:1 will be overloaded with a massive amount of tracking computation upon every packet entering its flow tables and will result in a bottleneck in the network.

Distributed Firewall Framework for Connection Tracking:

The bottleneck in the above scenario can be avoided by utilizing a distributed firewall framework to balance the connection tracking burden upon all the network switches.

Here, we can recognize two paths in the topology originating from host:red.

- Path:1 = host:red -> switch:1 -> switch:4 -> host:green
- Path:2 = host:red -> switch:1 -> switch:2 -> switch:3 -> host:orange

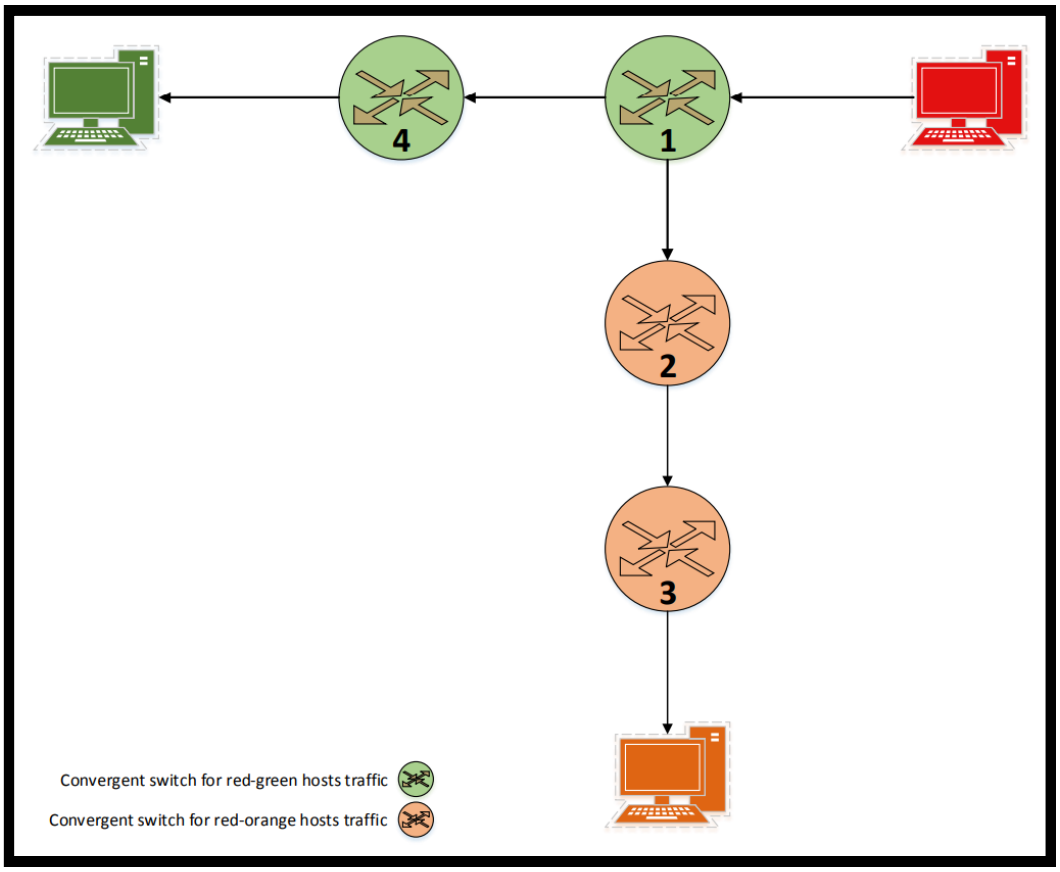

4
1
2
3
Convergent switch for red-green hosts traffic
Convergent switch for red-orange hosts traffic

Figure 11: Stateful distributed connection tracking in the data-plane

If we assume an equal number of connections to both host:green and host:orange, say 50 connections per host (100 in total), we can then deduce from this simplistic scenario that an optimal distribution of connection tracking between switches will result in each switch handling 25 connections.

Accordingly, switch:4 handles half of the connections destined to host:green (25 connections) and switch:1 handles the other half (25 connections). While switch:1 and switch:4 do not handle any of the connections destined to host:orange, switch:2 and switch:3 will then equally share the responsibility of tracking the connections destined to host:orange (25 connections each).

3.4.1 The Tracking Metric

The tracking metric is a measurement of the state processing weight each switch is responsible for upon distributing the state processing burden among the switches. These weights on the switches are computed to provide a balanced burden of processing between the switches on the path of a certain connection. The processing in this example corresponds to the connection tracking state updates.

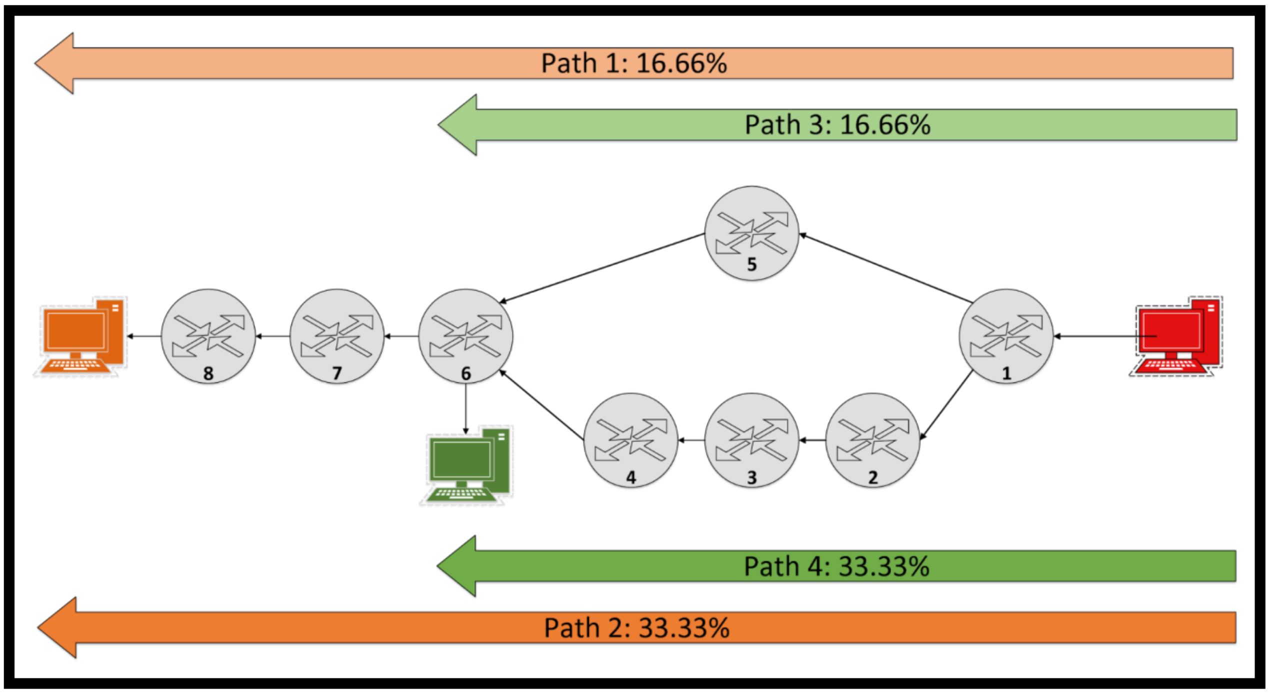


Figure 12: Connection-tracking example with load-balancing for distributed states in the data-plane

In order to illustrate this problem, we will use Figure 12 above that represents a little more of a complicated scenario in which the network employs some symmetric load-balancing procedures. The load-balancer at switch:1 assigns a load-weight 1:2 for the upper and lower outports respectively. If the network assumes one big firewall to track connections initiated from **host:red,** the latter can take multiple paths to reach **host:orange** and **host:green** as described in Figure 12 where each path has its corresponding path weight (Pp) as a percentage out of all the processing weight employed in the network.

Hence, whenever we refer to paths from now on, we mean convergence paths. If the application is not convergent on a specific switch, then this switch is simply not included in the convergence paths and is disregarded in the optimization. For connection-tracking in our case, we assume that the application is convergent on all switches.

We denote by $\mathbf{tp_{n,i}}$ the tracking weight per path assigned to switch:n pertaining to path:i.

Since path:2 does not pass through switch:5, $tp_{5,2}$ is zero.

Note: The tracking weight is not an explicit measurement of the **number** of connections a switch is tracking per path, but a measurement of its ratio with respect to the path weight. For example, if we assign a path weight of 1000 to be equally divided between two switches, then each switch is assigned a weight of 500. Similarly, the path weight itself is not an explicit measurement of the **number** of connections that traverse a particular path, but a measurement of its ratio with respect to the total traffic tracking weight traversing the whole network. This total traffic tracking weight can be an arbitrary number such as 10000. Since the amount of connections that takes place in a network cannot be predicted, we can represent the total traffic tracking weight as an amount of tracking burden in the network per a unit time irrespective of the magnitude this unit time. Since what we want to arrive at is a relational metric for every switch per every path representing the burden percentage taken by this switch out of the total tracking burden (weight) of the path, the actual magnitude of this time unit is as irrelevant as the actual number of total connections that will take place in the network.

The total tracking weight carried by a switch, denoted by $\mathbf{T_n}$ is then the sum of the tracking weights per path for all paths on this particular switch.

For example $T_1 = tp_{1,1} + tp_{1,2} + tp_{1,3} + tp_{1,4}$

Put another way, the tracking burden on switch:1 is the sum of the burden assigned to switch:1 for each of the paths 1,2,3,4.

Hence the definition:

$T_n = \sum_{i=1}^{I} tp_{n,i}$ where **n** is the switch number, **i** is the path number, and **I** is the total number of paths.

3.4.2 Optimization of the Tracking Values

In order to achieve a distributed burden, we employ the least squares method subject to certain path constraints.

3.4.2.1 Objective Function

Our objective function in this optimization is as follows:

$obj.\,function = min \sum_{n=1}^{N} (T_n)^2$ where **n** is the switch number, **N** is the total number of switches, and T_n is the tracking burden for switch n.

Note:

$\sum_{n=1}^{N} (T_n)$ = TTW, where the left expression of the equation is the sum of the tracked connections at every switch and TTW is a constant representing the number of connections to be tracked in the whole network (total traffic tracking weight)

In order to demonstrate how this objective function produces a distributed burden of tracking for every switch, we use the Lagrange Multipliers procedure to show that the solution will lead to exactly equal tracking weight for every switch.

$$obj\ :\ F(T_1, T_2, ..., T_n)\ :\ min\ \sum_{n=1}^{N}(T_n)^2 = min\,(T^2{}_1 + T^2{}_2 + \dots + T^2{}_n) \equiv min\ \tfrac{1}{2}X^T QX$$

, where X is the [T_1 ...T_N] vector and Q=1: 1x1 positive definite matrix.

$$constraints:\ G(T_1, T_2, ..., T_n)\ :\ T_1 + T_2 + ... + T_n = TTW$$

Lagrange:

$$L(T_1, T_2, ..., T_n, \lambda) = F(T_1, T_2, ..., T_n) - \lambda * G(T_1, T_2, ..., T_n)$$

Partial derivatives are zero at minimum: $\nabla L(T_1, T_2, ..., T_n, \lambda) = 0$

- $\nabla_{T_1} L = 2T_1 - \lambda = 0$
- $\nabla_{T_2} L = 2T_2 - \lambda = 0$
- …
- $\nabla_{T_n} L = 2T_n - \lambda = 0$
- $\nabla_{\lambda} L = T_1 + T_2 + ... + T_n - TTW = 0$

Conclusion:

Since the value of λ is a constant, then plugging in the equations gives: $T_1 = T_2 = \dots = T_n = TTW / n$

Thus, prior to introducing the path constraints on the network, the objective function inherently drives the solution into a state of equal burden on all switches.

A Change in Variables:

It does not suffice, however, to find the overall tracking weight per switch as represented by **T_n**. For the program to work properly, it must calculate the required tracking weight resulting from each path on each switch. The decision whether to track a connection is tightly dependent on which path the packet is taking through the network. And thus a single value that represents the sum of the tracking weights for all paths on a switch will yield an inconsistent optimization result in case there was a high variance in the probabilities of paths converging at a switch.

In order to introduce the path constraints, the objective function can further be dissected as follows to accommodate for the path constraints on the variables:

Thus, the objective function can be written as:

- $obj.\,function = min \sum_{n=1}^{N} (\sum_{i=1}^{I} tp_{n,i})^2$ where **n** is the switch number, **N** is the total number of switches, **i** is the path number, and **I** is the total number of paths.

In this representation, the solver will try to find the value of each tracking weight for every switch and for every path. This does not affect the result of the analysis above that showed that the objective function yields a distributed burden. The new objective function still drives the solution into an equal distribution of **the sum of tracking weights for every path** on every switch, which is essentially the goal of our optimization. However, it can now become consistent with the distribution of paths in the network, and optimized accordingly as we take the constraints in the next section into consideration.

3.4.2.2 Constraints

This optimization is subject to the following set of constraints:

(1) $\sum_{n=1}^{N} tp_{n,i} = Pp_i$ The sum of the tracking weights on all switches resulting from path **i** is equivalent to the probability of this path.

(2) $tp_{n,i} \geq 0$ for all **n**,**i**

(3) $tp_{n,i} = 0$ for path **i** that does not pass through **switch:n**

(4) $\sum_{i=1}^{I} tp_{n,i} \leq C_n$ The sum of tracking weights at each switch n for all paths is less or equal to the capacity of switch n.

3.4.2.3 Optimization Outcome

Upon the termination of the optimization computation, the system will yield values for each **$tp_{n,i}$**. As they represent the state processing weight for each switch, these values can then be used by the application to distribute the state variables among the network switches.

Normalization

The resulting values of **$tp_{n,i}$** at each iteration are scaled according to the weights or probabilities of the paths that were chosen in the network, P_{pi} . In order to arrive at a common scale, these values are normalized according to the below formula:

$$ntp_{n,i} = \frac{tp_{n,i}}{Pp_i}$$

3.4.3 Non-Convex Quadratic optimization

Although the new representation opened the way to introduce the path constraints on the objective function, it transformed the objective function into a non-convex function. This is because the optimal outcome of the optimization can now be satisfied through many solutions, instead of a single solution: Different combinations of the values of the tracking weights of each path $\mathbf{tp_{n,i}}$ on a single switch can result in the same overall tracking weight on the switch $\mathbf{T_n}$. Put differently, the overall tracking weights on the switches can be driven to an equal value with different combinations of the values of the tracking weight per path as can be seen in the definition:

$$T_n = \sum_{i=1}^{I} tp_{n,i}$$

This means that this problem will have many solutions where the constraints on the objective function stated in section 3.4.2 cannot restrict the range of possibilities into a single solution, which might be the case for example when two paths converge in two switches, in which different combinations of the burden can be taken by the two switches together. In such cases, although all of the existing solutions are optimal and equally accepted, the existence of more than one solution means that the problem itself remains a non-convex problem, and convex optimization solvers that require a positive definite (invertible) matrix cannot be employed in this case.

Then, what is required is a solver that can solve quadratic non-convex optimization problems without requiring an invertible hessian of the objective function. The following algorithm describes a constrained steepest descent that is adapted to this sort of problems.

Input:

```
        - path_weights: Array of size P
        Where path_weights[p] is the weight of path p in the network

        - network_nariables: NxP matrix
        Where network_variables[n,p] = TTW if path p passes through switch n, and 0 otherwise

Constrained Steepest Descent (network_Variables, path_weights):

        - normalize network_variables
        - While loss > accepted_error # as long as we can descend
                - calculate gradient
                - old_obj_value = eval(network_variables)
                - s =1 (initial step_dividend)
                - Backtrack: while step_dividend<max_dividend
                        - step = step - grad/s
                        - update: network_variables = network_variables - step
                        - if network_variables contains negative element, continue to backtrack (stay compliant with
                        constraint-(2))
                        - project network_variables on constraint-(1)
                        - rectify network_variables to comply with constraint-(3) and constraint-(4)
                        - normalize network_variables to comply with constraint-(1)
                        - new_obj_value = eval(network_variables)
                        - loss = old_obj_value - new_obj_value
                        - s = s*2 (update step_dividend)
                        - If loss<0                 # going up!
                                - Continue backtracking
                        - Else
                                - break
```

output:

Network_variables optimized for distributed burden

In each iteration, we descend on the gradient of objective function. The new values after each step are then projected onto the path constraints (constraint-1). But since the projection destroys the compliance with constraint-3 (network_variables[n,p] = 0) , the values are then rectified to maintain compliance with constraint-3 and constraint-4. Moreover, the rectification procedure introduces a small divergence from constraint-1, which is resolved by a normalization procedure that meets constraints 1, 3, and 4. Constraint-2 is constantly maintained through the backtracking procedure.

Complexity Analysis

To address the slow convergence of the steepest descent method, we present an analysis of the rate of convergence of the employed method at its worst case. This analysis comprises two components. The first component addresses the number of iterations required to attain convergence. The second component addresses the computational complexity for each iteration in terms of the number of variables to optimize (number of switches X number of paths).

For a minimization procedure, this computation process halts after iteration k when the **loss ≤ accepted error ($\boldsymbol{\varepsilon}$)**.

Here **loss = f(x_k) - f(x_{k+1})** where f(x_{k+1}) is attained by the following descent formula:

$f(x_{k+1}) = f(x_k - s*g_k)$ where $g_k = \nabla_x f(x_k)$ for $s \geq 0$.

As presented in [36], for general non-convex optimization functions, the number of iterations required to attain an iterate x_k for arbitrarily small $\boldsymbol{\varepsilon}$ is $O(\boldsymbol{\varepsilon}^{-2})$.

On the other hand, the complexity of each of the utilized computations: calculating the value of the gradient, projecting on the constraints, rectifying and normalizing, inside each iteration is O(N*P) where N is the number of switches and P is the number of paths. However, calculating the value of the objective function at each iteration involves the multiplication X^TX, where X is a matrix of size NxP, which adds up to $O(P^2N)$.

Thus it follows that the overall complexity of the optimization algorithm is $O(P^2*N*\boldsymbol{\varepsilon}^{-2})$.

Burden Distribution Results for Connection-Tracking Example:

Referring to Figure 12 where the load balancer at switch:1 outputs ⅓ of the traffic from host:red in the upper path and ⅔ of the traffic from host:red in the lower path, we will assume that the total traffic tracking weight in the network is 600. This number refers to the total amount of connections being initiated from the protected host in the network (host:red) per a unit time. We will also assume that the connections initiated from host:red are divided equally to host:orange and host:green for simplicity. Accordingly, the load balancer will establish the path weights as:

Path:1 = 100, path:2 = 200, path:3 = 100, path:4 = 200. Again, these path weights only represent the ratio of the connections in the network traversing a certain path with respect to the total weight of connections being initiated in the network. For example, connections traversing path:1 are ⅙ of the total connections in the network (initiated from the only protected host, host:red).

Applying the burden distribution optimization framework on the load-balanced network provided previously, without capacity constraints, will produce the following results (rounded):

Table 3: Optimal burden distribution results according to path explicit optimization method

Processing Burden	path:1	path:2	path:3	path:4	Total Processing/ switch
Switch:1	5.2	18	23.2	28.5	75
Switch:2	-	27.4	-	47.6	75
Switch:3	-	27.4	-	47.6	75
Switch:4	-	27.4	-	47.6	75
Switch:5	21.5	-	53.5	-	75
Switch:6	5.2	18	23.2	28.5	75
Switch:7	34	41	-	-	75
Switch:8	34	41	-	-	75
Total path weight	100	200	100	200	600

Looking at the results, it should not be surprising that the total processing burdens for all of the switches are equal since this is the outcome of an optimal distribution of burden among the switches.

To demonstrate a scenario that involves capacity constraints on the switches, we will assume the network with the same path distribution and we introduce capacity constraints as below:

- Max capacity at switch:1 = 40
- Max capacity at switch:3 = 60
- Max capacity at switch:8 = 70
- Other switches are unconstrained in capacity.

Table 4: Optimal burden distribution results with capacity constraints according to path explicit optimization method

Processing Burden	path:1	path:2	path:3	path:4	Total Processing/ switch
Switch:1	7.5	12.808	7.20	12.486	40
Switch:2	-	28.826	-	57.168	86
Switch:3	-	23.325	-	36.674	60
Switch:4	-	28.826	-	57.168	86
Switch:5	22.067	-	63.935	-	86
Switch:6	2.011	18.62	28.862	36.502	86
Switch:7	37.708	48.302	-	-	86
Switch:8	30.709	39.290	-	-	70
Total path weight	100	200	100	200	600

3.5 High availability of the application at the data-plane

The global view and manageability of the SDN controller allows it to detect violations in the network such as device downtime, inconsistent flows, and attack events. Consequently, the controller intervenes to restore the network into a stable state in accordance with the purpose of its applications. This intervention is constrained by the processing and communication time requirements and thus cannot be handled at wire speed. This might work well for applications that do not require immediate rectification. For applications that do, however, such approach is useless and can consequently cripple the availability of the whole service in the network. For example, inconsistent flows in the network can affect connections that are being established in the network. This gives a huge opportunity for adversaries to exploit the network with MiM attacks or gaining unauthorized access [21].

At this point, one might want to consider an intelligent data-plane packet-speed rectification of such scenarios that does not rely on the controller. Such a consideration falls easily within its own restrictions:

- The data-plane devices do not have a global view of the network. This limits their ability to detect such cases of inconsistency in the network.
- Even when it is possible to detect these cases in the data-plane, it is a tremendously tedious task for the network elements to reprogram themselves in a globally consistent way to remedy these inconsistencies.

For this reason, the best option for data-plane distributed firewall applications is to inherently employ high-availability techniques. In other words, in case of an attack or device downtime, the application should be able to automatically maintain the service in the network as a whole. If the network is to be viewed, fundamentally, as one big firewall, this should also manifest in the data-plane's ability to assume this framework. As such, a device downtime must not affect the correctness of the distributed states in the network, or at least, have a minimal effect on the availability of the service.

In contrast to SNAP, which provides an abstraction of the network as one big switch at the management-plane, the framework proposed here not only provides this abstraction, but also offers a high availability of the network at the data-plane as one big switch. In SNAP, if state distribution is to be employed, variables that correspond to state instances are distributed between several switches [26] and communicated on the path through special dictionaries on header fields. Hence, if one of the switches is attacked, the state variables that are hosted on this switch will be lost, which will have a detrimental effect on the correctness of all state instances in the network, even those at the switches that were not attacked, leaving the whole application dysfunctional.

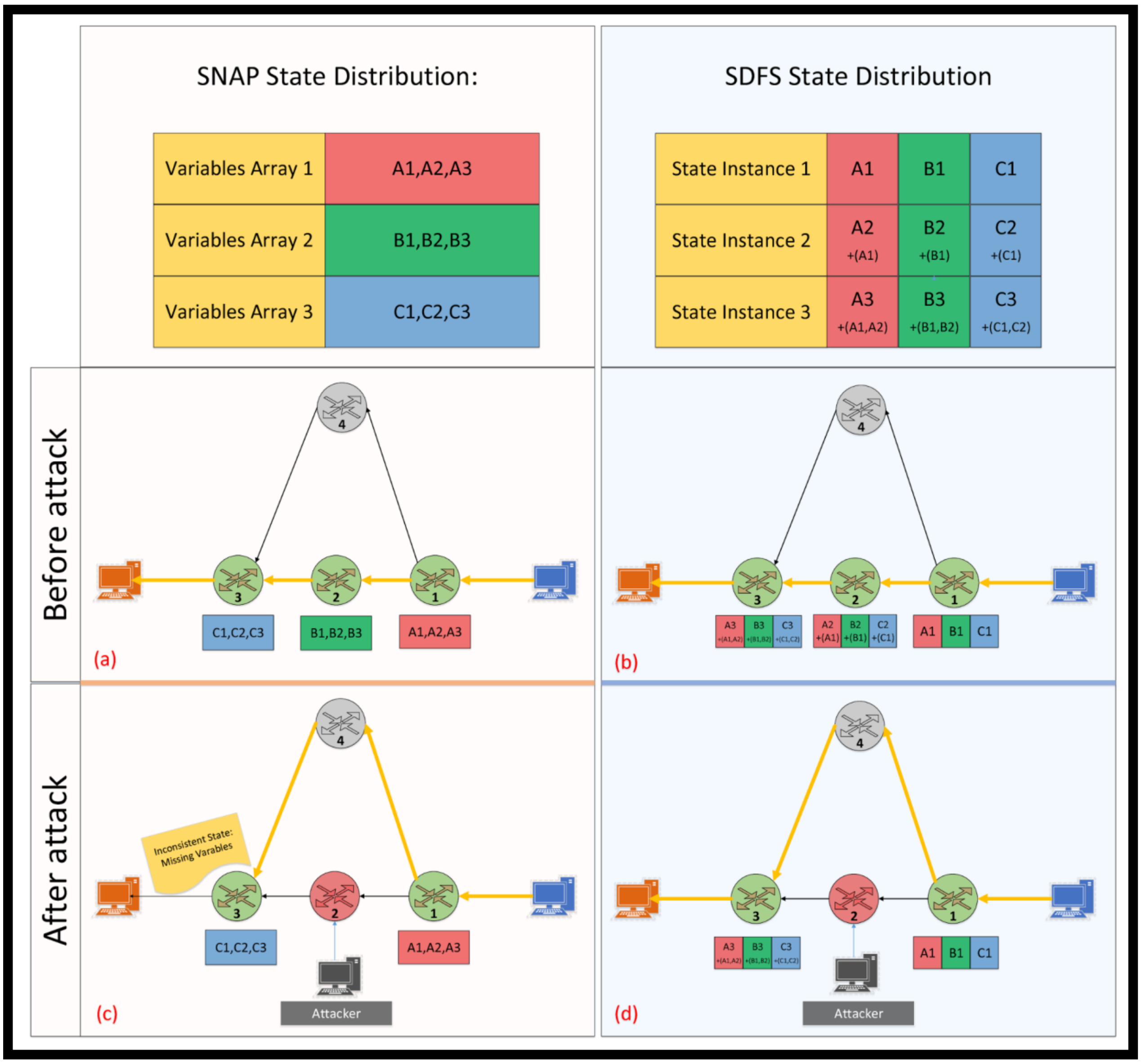


Figure 13: SNAP vs SDFS: Distribution of states and availability of the stateful application under healthy network conditions and after attack conditions

In order to demonstrate this behavior in comparison to our proposal, we will consider the scenarios in Figure 13. All the networks in Figure 13 (a, b, c, d) correspond to the same network. Scenarios (a) and (b) represent the healthy network scenario for SNAP and our proposed framework (SDFS) respectively while (c) and (d) represent the network directly after an attack on switch:2 for SNAP and SDFS respectively.

The first difference to be highlighted, between SNAP and SDFS, is the structure of the distributed state itself. As mentioned in section 5.2 (Structure of States in the

Data-plane), SNAP state distribution relies on distributing the state variables between the switches in contrast to distributing state instances in SDFS. As shown in Figure 13, SNAP hosts the red variables of the state in switch:1, the green variables of the state in switch:2, and the blue variables of the state in switch:3. Whereas SDFS distributes consistent state instances with their required variables on a single switch making each state-instance both atomic and self-sufficient. As such, state-instance:1 is hosted on switch:1, state-instance:2 is hosted on switch:2, and state-instance:3 is hosted on switch:3. It is very important to note here that a state requires all its variables to be accessed and updated consistently in the network for the application to behave correctly. The second difference to be highlighted here is the cumulative structure of the state instances in SDFS, where state-instance:2 subsumes state-instance:1, and similarly state-instance:3 subsumes both state-instance:1 and state-instance:2.

In the healthy network environment, the burden on computation is distributed equally in the network between the three switches. If we assume for SNAP that the computation required for variable updates is equal between the red,green, and blue variables, then each switch is taking ⅓ of the burden to process packets in the network. Also, in SDFS, each switch handles its assigned state instance which is also ⅓ of the burden to process packets in the network.

Upon the failure of a switch to handle state updates, resulting from a successful attack, for example, this story changes drastically between the two approaches. In the SNAP scenario after the attack- scenario (c)- the switch hosting the green variables is down. A fast-failover mechanism on switch:1 will reroute the traffic through the redundant switch- switch:4- destined to host:orange. When the traffic arrives at switch:3 from switch:4, it falls inconsistently with the state updates since the green variables are missing. Accordingly, all traffic from host:blue to host:orange will not be consistent with the application's intended behavior.

On the other hand, upon a successful attack on switch:2 in the SDFS framework- scenario (d)- traffic that was initially being process through state-instance:1 and state-instance:3 is not affected since these state-instances are still intact and consistent at switch:1 and switch:3. Not only the already established (through state-instances 1&3) connections remain intact, but also future traffic that falls in these state-instances remain intact too. Regarding traffic (or connections) that were being tracked at switch:2, these connections will be lost. However, future connections that fall within state-instance:2 will now be taken by switch:3. Since state-instance:3 subsumes state-instance:2, traffic "intended" for state-instance:2 that arrives at switch:3 unprocessed, will be inherently processed by state-instance:3. In this scenario, switch:3 will be processing ⅔ of the traffic in the network. Although the burden is skewed, however the point is to maintain the correctness of the application, for all future connections at least, and give time for the controller to respond and reconfigure the network by distributing the burden through switches 1,4, and 3.

This said, in contrast to SNAP, instead of arriving at a scenario where the whole network is down, we arrive at a scenario where only those connections that were already being processed at the switch:2 are lost, but all remaining connections (already established and future connections) maintain their correct behavior.

What we promise is the network's ability to seamlessly conform to network changes. Upon network disruptions, the degradation effect on the application's performance should be relative to the amount of the processing power lost, for example: the number of dead switches. In that sense, no states in the network will be sacrificed, other than those that were hosted at the dead switch itself. This loss will have no effect on the correctness of the service behavior with what regards subsequent connections in the network. That is, the network inherently re-conforms to distributing the load of burden the dead switch was responsible for, consequently, transforming the application from strict-time dependent into loose-time dependent. This then, resuscitates the techniques that can be used for loose-time dependent applications, i.e. a controller intervention through global detection and prevention techniques.

As discussed previously, the state instances are handled atomically at the switches. In other words, the variables of any state instance are all stored on the same switch and the optimization distributes the state instances among the switches instead of distributing the state variables. The tracking weights, as the outcome of the optimization procedure, discussed in section 3.4, are used at every switch to determine whether to accept the tracking of a connection (processing a state instance) or leave it for the next switch on the path to do so. This decision is taken independently at each switch, but nevertheless, each switch is programmed to take into consideration the tracking weights of the previous switches in order to arrive at a consistent tracking acceptance probability that reflects the switch's position on the path of the packet as discussed below.

3.5.1 Cumulative Chunking for High Availability

If a switch decides to process a state locally, this switch will mark the packet as **tracked**. Otherwise, this decision is left to subsequent switches on the path, where each switch makes the decision according to its assigned acceptance probability which reflects the correct tracking weight as assigned by the optimization algorithm. The power of this rises in that every switch's tracking decision subsumes the previous switches. In the healthy network scenario, each switch carries the responsibility of its assigned burden. However, in case of a faulty behavior, say a defective switch on the path that did not perform its tracking responsibility, the subsequent switches will inherently carry the tracking responsibility.

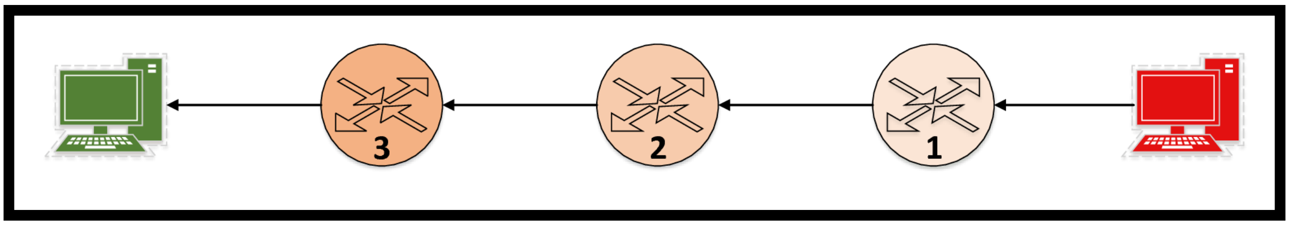


Figure 14: Network example to demonstrate cumulative chunking approach

A simple way to establish this behavior is to make this decision based on chunking the packet fields according to the **accept/deny** probabilities and accordingly matching against these fields. For example, let the first switch on the path have an **accept probability** = ¼ (meaning deny weight = ¾), the second switch have an **accept probability** = ¼, and let the third switch have an **accept probability** = ½. If we apply the chunking procedure based on Source_IP with 'x' as mask bits, then we let:

- If packet at switch:1 matches on Source_IP = [00xx….xxxx] then it decides to track, else it leaves it to switch:2
- If packet at switch:2 matches on Source_IP = [0xxx….xxxx] then it decides to track, else it leaves it to switch:3
- If packet at switch:3 matches on Source_IP = [xxxx….xxxx] then it decides to track

As demonstrated in this example, switch:1 will match on ¼ of the cases (cases where Source_IP starts with two 0 bits). Switch:2 will also match on ¼ of the cases which are really those that start with 01, that is since if Source_IP starts with 00, it will be taken by switch:1. Now since the 1 bit is not explicit in the chunk at switch:2, then if switch:1 fails to track a packet with Source_IP starting with 00 for some reason, it will be taken by switch:2. Same thing goes for switch:3, where in the healthy network scenario, it will match against packets with Source_IP starting with 1, and will take responsibility if switches 1&2 fail to track.

The above example exposes the essence of the chunking procedure. However, such chunking is completely rigid and biased. It does not have the flexibility to assign weights independent of the packet fields. For example, the configuration of the mask bits, above, inherently assumes an equal the amount of traffic emerging from each source. And if we want to integrate the assumption that different sources have different amount of processing participation in the network, we will have to combine the source_ip mask bits with another field, say source_port, where each switch has different combinations of the masking bits for each field which becomes very tedious to configure. Additionally, it is dependent on bit masks to cover the probabilities which cannot be accurate enough to represent combinations of **accept**/**deny** probabilities. To establish this behavior on a more probabilistic level, we employ the cumulative **accept**/**deny** probabilities, as discussed below, in a **group:select** action on the switches. For every path, we assign two buckets for this group where one of them represents **accept** and the other represents **deny**, each with its corresponding weight. Then, when a packet is matched on its path, it is submitted to its corresponding group, where the select action uses a hash function that takes fields from the packet as input and produces a value which falls in one of the buckets respecting their weight distribution. The decision is then made according to the action in each bucket, where the **accept** bucket will initiate the state tracking, and the **deny** bucket will leave it for the next switch. In the next section, we will discuss how we can compute the **Accept/Deny** bucket weights in order to arrive at the proposed high availability framework.

3.5.2 Cumulative Probabilities in Accept/Deny Group:Select Buckets

Each $\mathbf{ntp}_{\mathbf{n,i}}$ value, produced from the optimization procedure, represents switch **n**'s tracking participation in path **i**. In other words, it represents the probability that a certain connection will be tracked by switch **n** and not any other switch on the path of this connection. However, in this crude format, these probabilities do not take into consideration the series of decision events that take place on previous switches on the path. For example, a path comprised of three switches where each switch should participate in a third of the tracking burden for this path, will result in **ntp**=⅓ for each switch. This value, if used as a probability to hit the **Accept** bucket, will be correct only for the first switch on the path. When a packet arrives at the first switch on the path, this switch will give a ⅓ chance of tracking this connection and ⅔ chance of leaving the decision of tracking for subsequent switches on the path, that is, hitting a **Deny** bucket. Now when the packet arrives at the second switch, a ⅓ chance value falls inconsistently with the required behavior since arriving at the second switch, untracked, already subsumes the probability of the event of not being tracked in the first switch.

As described below, arriving at a consistent tracking probability must take into consideration the series of unsuccessful tracking events occurring on the path to the switch arising according to the basis on which the tracking decision is performed. This basis must, above all, include the packet fields in its decision in such a way that the decision remains consistent for connections in the network. That is, packets belonging to the same "connection", for example, must be processed at the same switch. And a switch rejecting to locally process a "connection" must remain consistent in this decision for subsequent packets.

3.5.2.1 Independent Events

For a packet traversing the switches, if at each switch, the decision is made based on an independent input, then the tracking events at the switches are considered stochastically independent. Such independent input is only independent in relation to the decision being made at the previous switches on the path. For example, if the switch generates a hash value out of the packet fields in addition to a switch-unique variable:**hash(packet_fields + unique_id)**, then the decision inputs at the switches can be considered independent provided that the hash function is uniformly distributed.

In that case, we can calculate the probability of tracking at switch:2 using the joint probability formula as below:

$$P(\textit{not processing at switch}:1 \bigcap \textit{processing at switch}:2) = P(\textit{not processing at switch}:1) * P(\textit{processing at switch}:2) = 1/3$$

$$=> \quad P(\textit{processing at switch}:2) = 1/2$$

Also $P(\textit{processing at switch}:3) = \frac{1/3}{P(\textit{not processing at switch}:1) * P(\textit{not processing at switch}:2)} = 1$

In general, for independent decision events, these probabilities can be calculated iteratively according to the below formula:

$Ap_{u,i} = ntp_{u,i} / \prod_{k=1}^{u-1} Dp_{k,i}$,where **u** is the position index of the current switch along path *i. A_p is the probability of processing packets belonging to path:i at switch:u (Accept probability), and D_p is the probability of not processing packets belonging to path:i at switch:u (Deny probability)*

Accordingly, $Dp_{u,i} = 1 - Ap_{u,i}$

3.5.2.2 Dependent Events

On the other hand, if the decision whether to process the packet is made based on a shared value among the switches, then the decision events are not independent as above. If we consider the case where all the switches use the same variable value added to the packet fields to produce a hash value as an input to the decision process:**hash(packet_fields + global_id)**, then the produced hash value is identical at all of the switches. In this case, if we want to assign a ⅓ processing probability for switch:2, it should possess a ⅓ probability increase over that of switch:1. As such, the correct acceptance probability for switch:2 is ⅔. Same goes for the third switch taking a ⅓ probability increase over switch:2, thus 1.

In general, for such a case of dependent events, these probabilities can be accumulated iteratively as below:

$Ap_{u,i} = \sum_{k=1}^{u} ntp_{k,i}$ *, where u is the position index of the current switch along path i. A_p is the probability of processing packets belonging to path:i at switch:u (Accept probability).*

Accordingly, $Dp_{u,i} = 1 - Ar_{u,i}$ *, where D_p is the probability of not processing packets belonging to path:1 at switch:u (Deny probability)*

Note: When cumulative weights are computed as above (i.e. as a summation of all the previous weights on the path up to the current weight), which is made possible by choosing a consistent **global_id** in the network, these weights will build "accumulatively" over each other along a certain path. Since the decision events are dependent, decisions taken at a particular switch will fully subsume the corresponding decision taken at a previous switch. For example, with this configuration, if the decision made at the **switch** was to accept the processing of a packet, then definitely the **next_switch** on the path will make the same decision. On the other hand, this "accumulative" feature is not ingrained in the independent variables case. This is because the input to the decision process at each switch is dependent on a variable that is unique to each switch, and thus, a decision at a **switch** to process a packet gives no information whether the **next_switch** on the path will also make the same decision.

For later references in the text, when cumulative weights possess the aforementioned “accumulative” feature, we will refer to them as cumulative weights with **additive** behavior.

3.5.3 Cumulative Probabilities Results for Connection-Tracking Example

Table 5 and Table 6 below present the results of the cumulative probabilities for each of the approaches. As presented below, both approaches converge to a “full” probability at the last switch to the destination (100/100 for path:1, 200/200 for path:2, etc..). Also, both approaches yield the correctness intended for the optimal burden distribution (75 for each switch). However, each approach computes the processing decision differently and accordingly has the corresponding configuration of the numbers to arrive at the optimal results.

3.5.3.1 Variable Independent Accept Probabilities Results

Applying the variable independent approach on the load-balanced network in Figure 12 and its results in Table 3 will produce the following results (rounded) for the accept cumulative weights calculated based on the variable independent framework:

Table 5 Variable Independent Accept Cumulative Probability Results

Accept cumulative weight per path weights	Path:1 /100	Path:2 /200	Path:3 /100	Path:4 /200	Total Processing/ switch
Switch:1	5.2	18	23.2	28.5	75
Switch:2	NA	30	NA	55.55	75
Switch:3	NA	35.4	NA	77	75
Switch:4	NA	43	NA	125	75
Switch:5	22.6	NA	69.7	NA	75
Switch:6	7	36	100	200	75
Switch:7	50	100	-	-	75
Switch:8	100	200	-	-	75
Total path weight	100	200	100	200	600

3.5.3.2 Variable Dependent Accept Probabilities Results

Applying the variable dependent approach on the load-balanced network in Figure 12 and its results in Table 3 will produce the following results (rounded) for the accept cumulative weights calculated based on the variable dependent framework:

Table 6: Variable Dependent Accept Cumulative Probability Results

Accept cumulative weight per path weights	Path:1 /100	Path:2 /200	Path:3 /100	Path:4 /200	Total Processing/ switch
Switch:1	5.2	18	23.2	28.5	75
Switch:2	-	45.4	-	76.1	75
Switch:3	-	72.76	-	123.8	75
Switch:4	-	100.1	-	171.43	75
Switch:5	26.65	-	76.76	-	75
Switch:6	31.85	118.14	100	200	75
Switch:7	66	159	-	-	75
Switch:8	100	200	-	-	75
Total path weight	100	200	100	200	600

3.5.4 Converting Probabilities into Bucket Weights

As counter-intuitive as it may seem, plugging **accept/deny** ratios as **accept/deny** bucket weights will result in an incorrect behavior of the distribution. This is because bucket weights in the group:select do not directly represent the allocation distribution. For example, if we employ the group:select approach in a load balancing procedure to arrive at 1:2 ratio load distribution on two ports by plugging these weights in the corresponding buckets of the ports, we will not arrive at the expected distribution.

Although the documentation of OpenVswitch [37] states that balancing across buckets is in compliance with the bucket weights, the formula that is used in making the decision suggests otherwise. According to the documentation, to select a bucket out of the group, OpenVswitch hashes the specified packet fields with the bucket_id and multiplies it by the weight of the bucket to arrive at a “score”, then selects the bucket with the highest score:

$\text{Score}_{\text{bucket}}$ = hash(packet_fields + bucket_id) * bucket_weight

For simplicity, we will consider the case where we have only two buckets. This can be applied either to load balancing on two ports or employing the buckets in our application to choose whether to accept the processing of a packet or deny it.

We now consider the scenario where we want bucket:1 to be executed ¼ of the time, and bucket:2 executed ¾ of the time. If the probability distribution is to be assigned directly to bucket weights, bucket weights will be assigned as $\text{weight}_1 = 1$ and $\text{weight}_2 = 3$.

However, what we really aim for is the following: Probability for score of bucket:1 to be larger than that of bucket:2 is ¼: $P(\text{score}_1 > \text{score}_2) = 1/4$

This can be translated into:

$P(\text{hash(packet\_fields + id}_1\text{) * weight}_1 > \text{hash(packet\_fields + id}_2\text{) * weight}_2\text{)} = 1/4$

The important observation to be made here is that the hash functions for different buckets use different bucket_ids (each bucket has a unique ID among the buckets in the group), and thus the resulting hash values can be considered independent variables.

Note: Keep in mind that this falls into the context of multiple buckets in a single group. Since buckets cannot have the same id in the same group, the variables will always be independent. And thus executing a bucket from the group is always based on independent variables. This is different from the previous cumulative tracking weights computation, where in the context of multiple switches, we can assign same bucket_id correspondence between the switches to arrive at dependent events between the switches.

If we replace the independent variables by x and y, we can write the formula as:

$P(x * weight_1 > y * weight_2) = ¼$

Which can be written as:

$P(x * weight_1 /weight_2 > y) = ¼$

The question here is the following: what should be the values of $weight_1$ and $weight_2$ such that the above equation is satisfied.

For simplicity, we will assume these x and y independent variables range from 0 to 1, and we draw the line: **$y= x* weight_1 /weight_2$** in Figure 15 below where the red area covers ¼ of the full area. In this case, the red are(r1) is less than the green area (r2).

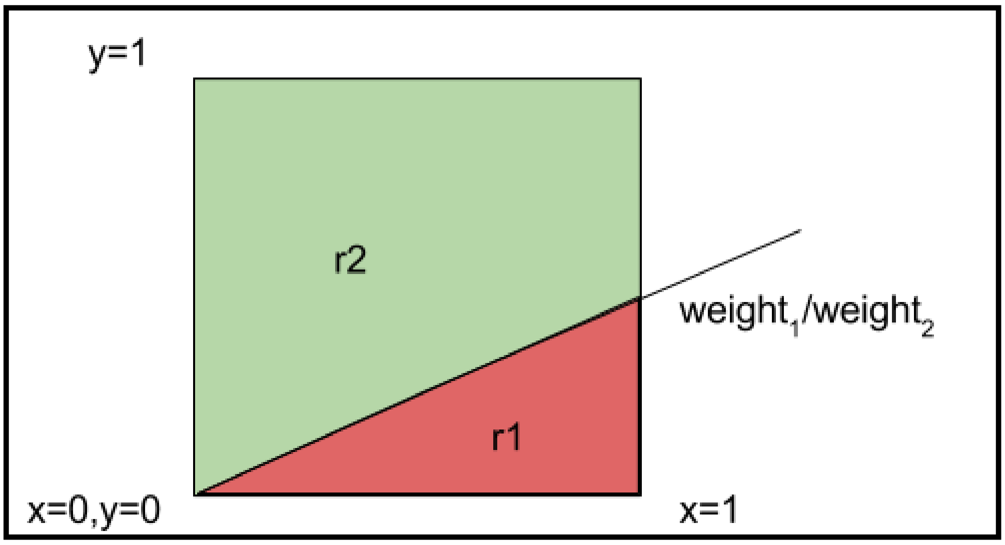


Figure 15: Bucket weight assignment according to probability distribution: case r1<r2

Accordingly, we can formulate the red area in terms of the weights we are looking for as:

Red_area = ½ * $weight_1 /weight_2$ = ¼. Thus $weight_1 /weight_2$ = ½. Accordingly, the correct assignment of weights is then 1:2 to arrive at a 1:3 proportions, in contrast to a direct assignment of 1:3.

If on the other hand, r1>r2, or in other words, the red area is larger than the green area, say 3:2 proportions, the results will appear as follows:

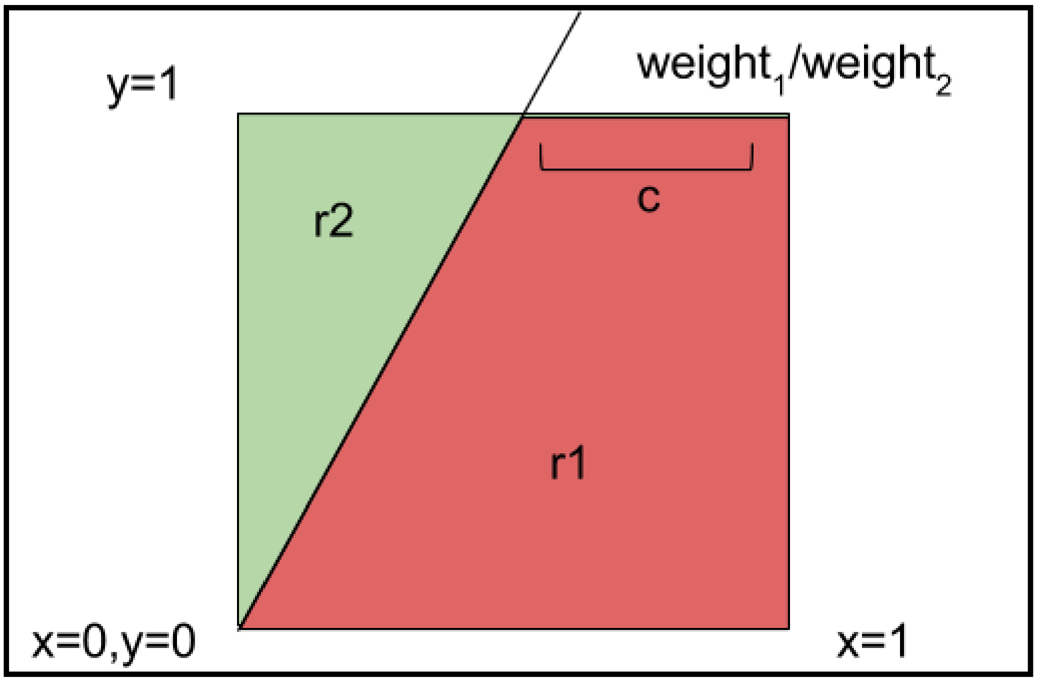


Figure 16: Bucket weight assignment according to probability distribution: case r1>r2

Red_area = ½ * (c+1) = ⅗ , where c = 1-(1/ weight$_1$ /weight$_2$)

Thus weight$_1$ /weight$_2$ = ⅘. Accordingly, the correct assignment of weights is then 4:5 to arrive at a 3:2 proportions, in contrast to a direct assignment of 3:2.

In general if we want to arrive at ratio r1:r2:

If r1<r2:

$$Weight_1 = 2*r1$$

$$Weight_2 = r1+r2$$

Else:

$$Weight_1 = r1+r2$$

$$Weight_2 = 2*r2$$

Applying Bucket Weights to Arrive at Cumulative Probabilities

The calculations in the previous section can be applied for both cases of the decision making process to arrive at the correct cumulative processing weights.

If switches use consistent bucket_ids for both the **accept** bucket and the **deny** bucket, say all accept buckets in the switches have bucket_id = 1, and all deny buckets in the switches have bucket_id = 2, then we can use the dependent events case to compute the cumulative probabilities. Then we can transform these probabilities to bucket weights accordingly.

If bucket_ids corresponding to the **accept/deny** buckets in the network are inconsistent between the switches, we can resort to the independent events case to compute the cumulative probabilities. However, the same transformation is applied in order to convert the probabilities into bucket weights. This is because, as mentioned earlier, in all cases, the bucket selection process is always based on independent variables due to the restriction of having same bucket_ids for both Accept and Deny buckets in the same group.

For demonstration purposes, we will assume consistent bucket_ids for accept and deny buckets among the switches. If we apply this approach on the three-switch example above, we will arrive at the following bucket weights that represent the cumulative processing distribution:

- At Switch:1 (cumulative ratio: 1:3) => (burden contribution ¼)
  => [AcceptBucket = 1, DenyBucket = 2]
- At Switch:2 (cumulative ratio: 2:2) => (burden contribution ¼)
  => [AcceptBucket = 1, DenyBucket = 1]
- Switch:3 (cumulative ratio: 4:0) => (burden contribution ½)
  => [AcceptBucket = 1, DenyBucket = 0]

3.6 The Announcement Bit

In section 3.5, we demonstrated how a switch makes the decision whether to process the state-instance locally or to leave it to be processed by the next switch. However, since the state's decision probabilities are constructed cumulatively, then if a switch handles a state-instance and proceeds to forward the processed packet on the path, then the packet will also hit an **accept** decision on all subsequent switches on the path. Clearly, processing the same state-instance at every subsequent switch is redundant and fundamentally defies the purpose of state processing burden distribution in the network. This is not a substantial problem however, since all that is required is a tag on the packet that marks it as **already processed**. For example, an unused VLAN tag in the network can serve this purpose. Whenever a packet received a tagged packet as **already processed**, it does not subject the packet to the state-processing procedure.

3.7 A Path-Aware OpenFlow Implementation Example for Connection-Tracking

We will present in this section a simple OpenFlow implementation for connection-tracking in a path-aware network in which we employ the above described burden distribution along with cumulative chunking of states.

3.7.1 Provisioning

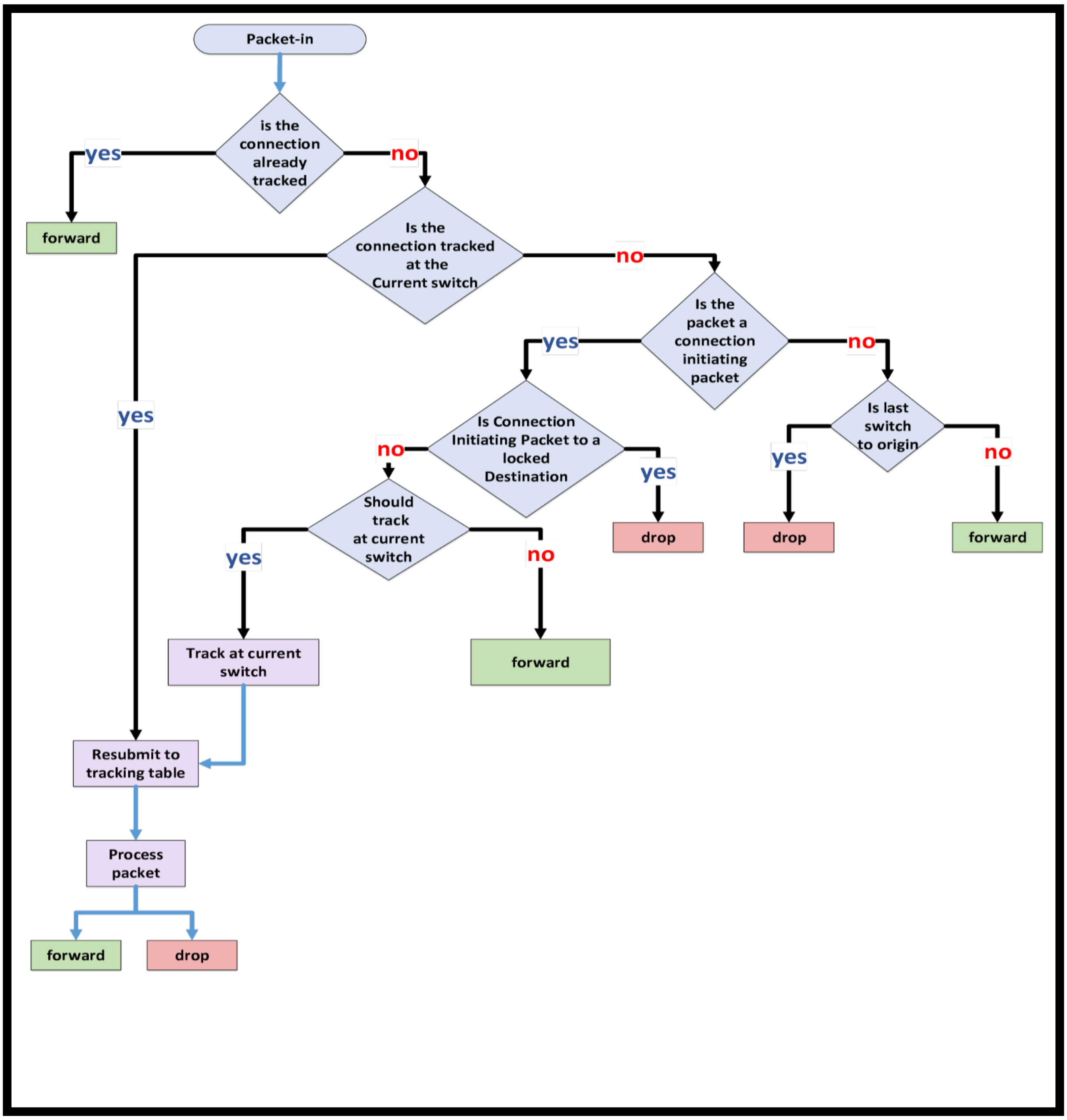
For this implementation, we will assume the following:

- The network employs a symmetric layer-4 load balancing procedure ensuring the bidirectional convergence of packets on the state-instances in the network.

- The network management reserves a VLAN tag value, say 100, for this application to mark the packets as **already processed**.
- The controller has access to packet path information in the network which is used at the switches to identify the path that each packet belongs to.

Figure 17 below describes the tracking decision tree of the connection-tracking application at the data-plane. The decision tree is followed by a walkthrough that describes the whole procedure.

3.7.2 Tracking Decision Tree



Packet-in
is the connection already tracked
yes
no
forward
Is the connection tracked at the Current switch
no
yes
Is the packet a connection initiating packet
yes
no
Is Connection Initiating Packet to a locked Destination
no
yes
Is last switch to origin
yes
no
Should track at current switch
yes
no
drop
drop
forward
Track at current switch
forward
Resubmit to tracking table
Process packet
forward
drop

Figure 17: Switch-level decisions for connection-tracking application

3.7.3 The Tracking Decision Procedure

Upon packet entry: is the connection already tracked?

When a packet reaches a switch, it is subjected to the OpenFlow pipeline where the packet is matched against the flow rules. Openflow pipelines can be designed in different ways to serve the same function. The design can have drastic effects on the performance. In other words, it is very important to establish a pipeline that minimizes the number of rules a packet is matched against before it is forwarded on an outport. For this purpose, many optimizers have been proposed such as P4's abstract language and compiler [25].

Packets are initially classified as to whether they are **already tracked** (table=10). This classification is based on a **tag** that is given to the packet when it is tracked by a previous switch on the path. This procedure helps switches recognize packets that fall in an already tracked connection. Without further re-computations, the packets recognized as **already tracked** can then bypass the connection tracking pipeline on the current switch since they are already handled by a previous switch on the path.

Is the connection tracked at the current switch?

Packets arriving at the switch that are not tagged as **already tracked** are then checked against all connections that are currently being tracked by the switch (table=20). If the switch is already tracking this connection, the packet can be resubmitted directly to the connection tracking table in which connection states are updated/learned. Otherwise, the packet is resubmitted as below, to be checked if it is a connection initiating packet.

Is the packet a connection initiating packet?

Packets that are neither tagged as **already tracked** nor tracked at the current switch are classified at every switch to decide whether these packets should be subjected to the connection tracking procedure (table=30). This stage is ruled by policies inserted by the network manager to determine which hosts require connection tracking to communicate with and which do not. For easier reference, hosts that require the connection tracking procedure to communicate with are labeled as **locked** hosts. Inversely, those which do not require connection tracking are **unlocked** hosts. If the packet appears to be a connection initiating packet originating from a locked host, the packet is then resubmitted to check if the **destination host is also locked**. On the other hand, if the sender was an unlocked host, the packet is resubmitted to be checked if it is on the **last switch** destined to a locked host.

Is Connection Initiating Packet to a locked Destination?

At this stage, the packet has been classified as a connection initiating packet originating from a locked host. However, if it appears that the packet is destined to a locked host also, the packet will be dropped (table=50). This is because no connection can automatically be established between two locked hosts unless the network manager specifically allows it according to a specific rule. Otherwise, the packet is resubmitted to be checked if it **should be tracked here**, at the current switch.

Is last switch to origin?

If the packet is not being tracked on this switch (and of course not on a previous switch on the path because it is not tagged as **already tracked**), a check is made to determine if this switch is the last switch to the locked host (table=70). In the example in Figure 12, host:red is a locked host, that is only host:red can initiate a connection.

The check in this context will determine whether a packet- that according to the policy table should be subjected to connection tracking- should be dropped because it was not tracked on a previous switch on the path **and** the current switch- being the last switch to the locked host- is not tracking this connection.

This is the scenario that will take place if host:orange tries to initiate a connection with host:red. Since host:orange is not allowed to initiate a connection to host:red, according to the connection-tracking protocol, the switches in the network will not allow such communication and consequently drop the packets before they arrive at host:red.

Thus, after the packet from host:orange passes switch:3 and switch:2, switch:1- which is the last switch to host:red- will make the decision to drop the packet because no previous switch on the path had already tracked the packet and the connection is not being tracked on switch:1 either.

Should track at current switch?

This decision is made by assigning an accept-weight and deny-weight for each path that passes through the current switch. As described before, after the packet is matched to the path (table=90), it is submitted to the corresponding **group:select** where if the packet hits an **accept**, the packet is resubmitted to the tracking table. Additionally, a **tracked here** flow is learnt for future packets in the connection to match against (table=91). If the packet hits a **deny**, the packet is not tracked and forwarded to the next switch (table=92).

Note: The mechanism of how a packet matches on the path is application specific. As discussed previously, this requires explicit information about the path of the packet to be available at the switch. In a load-balanced network for example, the switch can be distributing the load based on L3 fields. In that sense, matching on the paths can be as simple as matching on the destination-ip. On the other hand, in our load-balancing network example, the switch can be distributing the load based on L4 fields. In this case, matching on the path will require matching on explicit combinations of L4 fields to determine the path. Another way to match on the path in L4 load-balanced network is by extracting the bucket distribution weights from the switch and mirroring the same exact weights to match on the path. In the latter case, matching on the path can be as simple as matching on the destination-ip plus the corresponding bucket, provided that packet fields are not altered directly before being routed.

It is very important at this point to note that if the current switch handling the decision whether to track is the last switch on the path to the locked host, the decision will always hit an accept. As discussed previously, the cumulative probabilities of the accept weight converge to one as the packet gets nearer to the last switch on the path.

The tracking table

When a switch makes the decision to track a connection, it adds a new flow in its connection tracking table that handles the connection state (table=110). When a switch receives another packet of the same connection, it directly resubmits this packet to the connection tracking table and the state updates are handled accordingly.

If the connection established belonged to a connectionless protocol such as UDP, the state of the connection is updated to **CLOSED** when the hard_timeout of the learnt tracking flow expires.

However, if the connection was TCP, the switch keeps accepting packets of the tracked connection until it receives a tcp:fin packet, in which case the state is updated to **CLOSED** and the connection is torn down.

3.8 Dismissing the Announcement Bit

As previously discussed, one mechanism for announcing that a state-instance has been processed by a previous switch on the path of a packet is by tagging the packet with a special header that signifies this information. However, special tags flowing through the network might not fall consistently with other applications employed in the network. As a result, the behavior of these applications can be disrupted unless adjusted to handle the tagged traffic. In an attempt to maintain the modularity of the proposed framework, we will present another mechanism that replaces the announcement bit. This mechanism utilizes the capacity of the cumulative chunking framework in revealing such information to subsequent switches on the path without actually forwarding the information in header fields.

Two pieces of information must be provided **locally** to the switch in order to determine whether a packet has been processed.

The switch must be able to identify the path that this packet belongs to.

The switch must host, locally, the pre-programmed cumulative chunk weight of the **direct previous** switch corresponding to this path, provided that the cumulative weights possess the additive behavior where each cumulative weight at **any** switch subsumes the corresponding cumulative weight on its previous switch.

When chunks are built additively, a switch can try to project the packet against the chunk of the previous switch, and if it matches, then the switch discovers that the packet has been processed by the previous switch. If it does not match, then the switch will know that the packet has not been processed yet, and accordingly, it subjects the packet to its own cumulative chunk to decide whether to process the packet locally or leave it to the next switch on the path.

As **noted** in the “dependent events” case, for cumulative chunks to build additively, bucket_ids corresponding to **accept/deny** buckets must be consistent in the whole network. If groups have consistent bucket_ids, then the processing decision events are dependent and the probabilities are built additively. This is very important as it preserves information about the past decisions that were taken previously on the path for a particular packet. On the other hand, if bucket_ids are inconsistent in the network, then the decision events are independent and cumulative probabilities do not build additively, and thus, the cumulative chunk of the previous switch if used at the current switch to test whether a packet has been processed does not provide a complete measurement. In the latter case, a complete measurement can only be attained if the current switch is provided with the cumulative probabilities of all the previous switches on the path to test, and not only the direct previous switch, which is both memory and performance exhaustive process.

It is important to note here that this approach will fail to maintain correctness if the packet fields upon which the hash is computed change along the path. And thus utilizing cumulative chunking to dismiss the announcement bit can only be used between consecutive switches that do not manipulate the packet fields corresponding to the chunk.

The trade-off put forward here gives away the trouble of dealing with special headers in the network at the expense of more information being hosted at the switches (in openflow-rule format). In this framework, the switch not only has to possess its own path-to-cumulative-weight information, but also has to possess that of all the adjacent switches for each path.

To illustrate this behavior, we will use the same three-switch example as before in Figure 14.

In the network in Figure 18 below, let switches 1, 2, and 3 be convergent switches on the path from host:red to host:green where the distribution burden is divided equally among the switches.

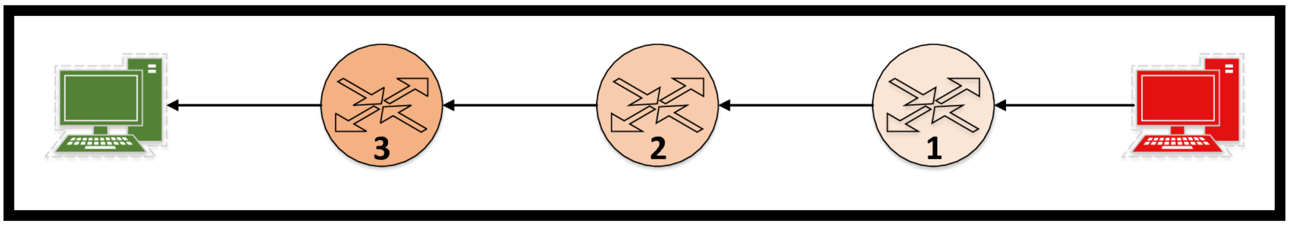


Figure 18: Simple network example to demonstrate the approach for dismissing the announcement bit

Then, the accept probabilities for switches 1, 2, and 3 are ⅓, ⅔, and 1 respectively. These probabilities are then mapped to a select group bucket weights such as the following:

- At Switch:1 (cumulative ratio: 1:2) => (burden contribution ⅓) => [AcceptBucket = 2, DenyBucket = 3]
- At Switch:2 (cumulative ratio: 2:1) => (burden contribution ⅔) => [AcceptBucket = 3, DenyBucket = 2]
- At Switch:3 (cumulative ratio: 3:0) => (burden contribution 1) => [AcceptBucket = 1, DenyBucket = 0]

To host the auxiliary cumulative information at the switches, the switches must include the extra information about the corresponding previous switch as below:

- At switch:1 => no previous switch
- At switch:2 => previous_switch : [AcceptBucket = 2, DenyBucket = 3]
- At switch:3 => previous_switch : [AcceptBucket = 3, DenyBucket = 2]

Now, let's assume a scenario where the state-instance is processed at switch:2.

In this scenario, switch:1 will subject the packet into the buckets that correspond to its previous switch. But since it is the first switch on the path, no such information exists, and thus a match will fail. As a result, switch:1 will subject the packet to its own buckets [AcceptBucket = 2, DenyBucket = 3] to try to process the state-instance locally. The match will fail and switch:1 forwards the packet unprocessed to switch:2.

At switch:2, the packet is subjected to the corresponding buckets of its previous switch (switch:1), in this case [AcceptBucket = 2, DenyBucket = 3], which will also fail as it already failed on switch:1. Switch:2 then subjects the packet to its own buckets [AcceptBucket = 3, DenyBucket = 2] which in this case hits an accept bucket and succeeds in processing the state. The packet is then forwarded on the path to host:green to reach switch:3.

Switch:3 however, when subjecting the packet to the buckets corresponding to its previous switch [AcceptBucket = 3, DenyBucket = 2], hits an accept bucket. With this, switch:3 discovers that the packet is being processed at some previous switch on the path and accordingly the packet is directly forwarded to host:green bypassing the state-processing decision pipeline at switch:3.

3.9 Packet Path Oblivious Approach

In the previous sections, we presented a framework for burden distribution that accounts for the path costs in the network. This framework provides a precise distribution provided that the switches are aware of the path the packet is taking. The path of the packet referred to here not only includes the future switches the packet will pass through, but also the sequence of switches the packet already passed through in order to arrive at the current switch. Since the cumulative weights are tightly coupled with the packet path information, the switch needs this information in order to choose which cumulative weight it should subject the packet to.

This information can be easily accessible in networks that employ simple L2-forwarding/L3-routing. Although it can also be available in networks that employ load-balancing measures, the amount of paths in the network explodes combinatorially. In that sense, exhausting the switches with path information to match against might be very impractical in cases where there is a large number of paths, and impossible in cases of unavailable path information.

Below, we will present two approaches that fundamentally relax the network from the explicit path information constraints, provided that the convergence requirements in the network are still satisfied. In other words, it is enough to ensure the convergence of the traffic on the switches and provide the controller with metadata about the amount of traffic traversing the network, without requiring explicit information about the path of every packet according to its combination of header field values. While both approaches often do not yield optimal burden distribution results as that of the path explicit approach, the first approach, **additive cumulative average approach**, maintains the ability in the network to dismiss the extra header field for announcing the processing decisions of a switch to subsequent switches. This comes at an expense of a less accurate burden distribution. The second approach, **path aggregation approach**, abandons the ability to dismiss the announcement bit (i.e. requires extra header fields) however yields a better accuracy in the burden distribution.

3.9.1 Additive Cumulative Average Approach

What is behind the relaxation on the path constraints is the lack of enough identifiers at the switches to differentiate between the paths and consequently map every packet to its corresponding path. But since mapping the packet to its corresponding path- and accordingly its corresponding cumulative weight to be projected upon- is an essential part in the decision making process (i.e. whether the switch decides to process the packet or leave to the next switch), this approach tries to "approximate" the path of the packet without needing to arrive at the actual path that the packet is taking.

However, if each switch makes an unsupervised approximation to the path of the packet, then the packet might find itself being subjected to a cumulative weight at a certain switch which is **less** than that from the previous switch. In this case, the cumulative variables will lose their additive behavior, which is a necessary condition for dismissing the announcement bit. Granted that the cumulative weights in each path are still calculated based on the "dependent events" case (i.e. with consistent bucket_ids), but since the packet is being subjected to cumulative weights from different paths that do not positively accumulate over each other (cumulative weight at any switch is not larger than that at the previous switch), information about "whether some previous switch on the path of the packet has made the decision to process the packet" cannot be inherently found in the **direct previous switch**.

In an attempt to maintain the positive accumulation of the cumulative weights along the paths without conceding to every path separately, this approach tackles the burden distribution on the basis of averaging out the cumulative weights of multiple paths, at each switch, while asserting the addictive behavior of these average cumulative weights among the switches.

Simply put, the cumulative average at switch:n can be calculated as below for each source-destination:

$CA_n = \frac{1}{\sum_{i=0}^{p} PW_i} \sum_{i=0}^{p} CW_{n,p} * PW_i$, where **CA$_{\mathbf{n}}$** is the cumulative average weight at switch:n, **p** is the set of paths passing through switch:n for the corresponding source-destination pair, **CW$_{\mathbf{n,p}}$** is the cumulative weight of path:p at switch:n, and **PW$_{\mathbf{p}}$** is the path weight of path:p.

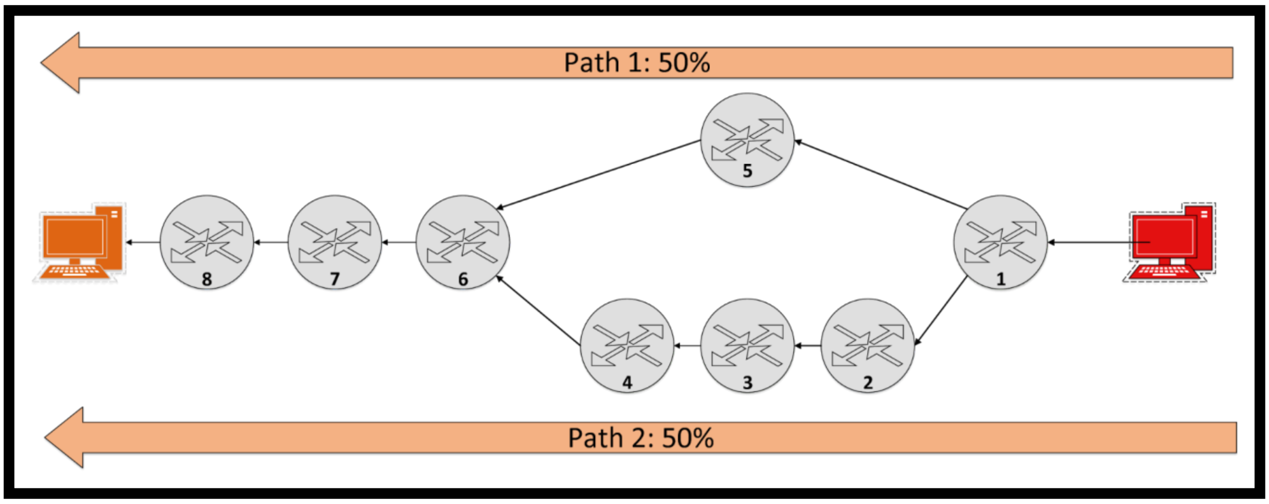


Figure 19: Network example for path oblivious approaches

Taking the network in Figure 19 above where load-balancing is employed at switch:1, to calculate the additive average cumulative weights, we propagate backwards from destination to source starting at the last switch tp destination, which will definitely host a full cumulative weight to cover all relevant traffic processing. Then, whenever we calculate the cumulative average at a previous switch, we make sure that it is less than that at the current switch. For example, the average cumulative calculated at switch:6 must be less than that at switch:7. Similarly switch:4 and switch:5 must, each, host a cumulative average less than that of switch:6. This assertion ensures that the cumulative weights hosted in the switches will always possess the additive behavior regardless of which path the packet takes in the network.

Under adequate conditions, averaging out the cumulative weights into a single cumulative weight should not affect the actual burden responsibility taken by a switch. This is because the average inherently takes into account the path weights. However, a change in the average cumulative weight- due to assertions- at a previous switch, might result in a suboptimal actual distribution of burden in the network.

In the explicit path information case, the optimization function at the controller will yield the following values for state-processing burdens for some chosen path weights, say 100 for path:1 and 100 for path:2.

Table 7: Optimal burden distribution cumulative probability results according to path explicit optimization method

Processing Burden	path:1	Path:1 cumulative	path:2	Path:2 cumulative	Total Processing/ switch
Switch:1	18.75	18.75	6.25	6.25	25
Switch:2	0	NA	25	31.25	25
Switch:3	0	NA	25	56.25	25
Switch:4	0	NA	25	81.25	25
Switch:5	25	43.75	0	NA	25
Switch:6	18.75	62.5	6.25	87.5	25
Switch:7	18.75	81.25	6.25	93.75	25
Switch:8	18.75	100	6.25	100	25
Total path weight	100	FULL	100	FULL	200

Based on the optimal results, we can then calculate the cumulative average weights corresponding to path:1 and path:2 as the following:

$CA_7 = 81.25*50\% + 93.75*50\% = 87.5$

$CA_6 = 62.5*50\% + 87.5*50\% = 75$

$CA_5 = 43.75*100\% = 43.75$

$CA_4 = 81.25*100\% = 81.25$

As can be seen in these results, $CA_4>CA_6$. In this case, CA_4 must be adjusted to be less than CA_6.

Considerations for Bidirectional Processing

In applications such as connection tracking, although only the traffic initiated from the protected hosts can initiate a processing state-instance, the traffic in the reverse direction also participates in the corresponding state-instance. For example, traffic destined to a protected host (reverse direction) is only allowed when the connection is already initiated from the protected host. Since we are not using the announcement bit, we cannot leave the packet until it arrives at the last switch to the protected host, as mentioned in section 3.7.4, to decide whether to forward the packet to the protected host or drop it depending on the announcement header. Hence, the decision whether to forward or drop the packet must be made independently at each switch.

In the framework where the announcement bit is dismissed, depending on the packet's header field values, the corresponding switch which is responsible to make this decision (forward vs drop) is already predetermined on its path. This switch is the one that holds the cumulative weight value that is enough for the packet to fall into. For example, in the reverse direction, if the packet's hash value produces say 82, then it is the responsibility of switch:7 to process this packet, since switch:7 holds hosts the cumulative weight of 87.5, which is the least cumulative weight higher than that produced by the hash value of the packet header fields (or the whole connection in general). Once it passes switch:7, switch:6 cannot make the decision on this particular packet. This is because this packet falls in a connection that should be processed at switch:7. Thus, only switch:7 can know whether this connection is established or closed, and accordingly, only switch:7 makes the decision to allow this packet in the network or drop it.

A problem arises at points of path divergence in the reverse path, that is at switch:6 in our example above, particularly because switch:6 does not have information about the future direction of the packet. To demonstrate this problem, we will assume at first that CA_4 is chosen to be 60 to assert the additive behavior in the forward direction. Then, if a packet falls in a connection with a corresponding hash value of 50, then if the packet takes reverse path:1, switch:6 must process this packet, however, if the packet takes reverse path:2, switch:6 must leave it to switch:4 to make this decision- since CA_4 (60) is still larger than that produced for the connection (50). But since switch:6 does not know the future reverse path of the packet, then switch:6 cannot make this decision - that is whether to process it or leave it for the next switch. Switch:6 cannot also resort to making a safe decision and process the packet anyway, since if it was the case that the packet takes reverse path:2, then switch:6 might decide to drop the packet (because the connection is not detected as established on switch:6) although it is being processed at switch:4.

To resolve this issue for applications with bidirectional processing requirements, we add a second assertion. The second assertion ensures that switches after a point of divergence in the reverse path must possess the same cumulative value. In this case, $CA_4=CA_5$. As such, the confusion at switch:6 can be resolved without the need to determine the future path of packets at switch:6. This problem does not appear in the forward path simply because the additive behavior in the forward path ensures that the previous switch always possesses a less average cumulative weight.

In general, the average cumulative weight assignment can be computed as below:

- For each source-destination pair in the network, extract the directed acyclic graph (DAG) from destination to source

- In this DAG, group switches that have common predecessor into ***same_level_switches*** *groups. For example, if switch:4 and switch:5 are preceded by switch:6, then switch:4 and switch:5 fall in the same* ***same_level_switches*** *group. If also switch:5 and switch:9 are preceded by switch:10, then all switches 4,5, and 9 fall in the same* ***same_level_switches*** *group. These groups can contain a single switch if they don't have siblings.*

- Traverse the DAG from destination to source while calculating the additive cumulative average weights (ACAs) as below:

```
        - parent_limit_assertion = ALL_TRAFFIC_WEIGHT (sum of all paths)
        - Get same_level_switches of the current node
        - Initialize ACA = 0
```

```
- Initialize sum_of_path_weights = 0
- For each switch:n in same_level_switches:
        - For each path:p for the corresponding source-destination pair:
                - ACA = ACA + $CW_{n,p}$*$PW_p$
                // where $CW_{n,p}$ is the cumulative weight of path:p at switch:n and $PW_p$ is // the path
                weight of path:p
                - Sum_of_path_weights = sum_of_path_weights + $PW_p$
- ACA = ACA/Sum_of_path_weights
- If ACA>parent_limit_assertion:
        - ACA = parent_limit_assertion
- Assign ACA to all switches in same_level_switches
- Update: Parent_limit_assertion = ACA
```

Such a distribution maintains the positive accumulation of the cumulative weights on any path the packet is taking. That is, whatever path the next switch will choose to approximate with, its **accept** cumulative weight is always higher than that of the current switch. For this reason, we can use the approach discussed previously to replace the announcement bit where the two required pieces of information that should be available locally at the switch are satisfied by (1) augmenting the local switch cumulative processing weights with that of the previous switch, and (2) aggregating the paths of the packet into a single path based on the average cumulatives instead of the exact path cumulatives whenever exact path information is not available.

Table 8 below shows the burden distribution for the switches in the network under additive cumulative average approach.

Table 8: Burden distribution cumulative probability results according to additive cumulative average approach

Processing Burden	Additive Cumulative Average Percentage	Total Processing/ switch
Switch:1	12.5	25
Switch:2	31.25	18.76
Switch:3	56.28	25
Switch:4	62.5	6.23
Switch:5	62.5	50
Switch:6	75	25
Switch:7	87.5	25
Switch:8	100	25
Total processing	100%	200

Since paths are now approximated by the additive cumulative average approach, the burden of state processing in the network loses its optimal distribution. The optimal processing weight of switch:5 is 25% of the total weight of path:1. But since the cumulative weight of switch:1 pertaining to path:1 is now 12.5 instead of 18.75, and since the cumulative weight of switch:5 increased from 43.75 to 62.5, the burden at switch:5 now becomes (62.5%-12.5%)*100 = 50. Here, 100 is the path weight of path:1. Similarly, the burden at switch:4 decreases from its optimal value of 25 to 6.23.

3.9.2 Path Aggregation Approach

This approach deals with the burden distribution problem by aggregating the optimal weights that are computed for every path into a single weight. This weight is nothing but the sum of the assigned weights for each path corresponding to each particular switch. By virtue of using an aggregated weight as the processing weight and applying this weight consistently on all packets irrespective of their paths, the paths are then assumed aggregated on the switches. What we aim for here is that the processing contribution of every switch stays the same as that when we had a specific processing weight for each path, or at least, as close to it as possible. Heuristically, the switch will average out its processing burden on all paths to maintain approximately the same contribution. Accordingly, the network can arrive at a burden distribution as accurate as that of the path information explicit cases. Thus, it is an extremely powerful tool for burden distribution that relieves the data-plane from the exhaustive path information requirements and at the same time maintains the modularity of the application in not requiring explicit information about the paths in the network, and above all, any manipulation of other applications.

Computing the aggregated weights is not enough to feed the network with the right bucket weights. As in all the cases above, we need to arrive at the cumulative weights in the network to use the cumulative chunking technique essential for maintaining the high availability of the application at the data-plane. In this case, we need to compute the cumulative aggregate weights based on the aggregate weights.

However, building the cumulative aggregate weights on the basis of the aggregate weights does not necessarily yield the additive behavior of the cumulative weights as before. In other words, cumulative aggregate weights along the path of a packet do not have to be steadily increasing. A packet can visit a switch with a certain **accept** cumulative aggregate weight, and later visit a subsequent switch on the path with an **accept** cumulative aggregate weight of lesser value. Thus, the approach used in the additive cumulative weights for detecting whether a packet has been processed does not hold anymore, and the application at the data-plane has to regress back to using the announcement bit.

We will present below a greedy computation of the cumulative aggregate weights by incrementally feeding the values into the switches as we traverse the paths from source to destination. As stated, this approach might not yield the same results as that provided by the path explicit approach. Thus, whenever the result is not as accurate, we will arrive at a certain discrepancy in the contribution of some of the switches. Although the greedy algorithm ensures correctness of the application, it does not optimally redistribute this discrepancy among the switches. While we see that this discrepancy is very minimal compared to the actual outcome, we will present the optimization approach which can deal with such discrepancies for the path aggregation approach.

3.9.2.1 Greedy Computation of Cumulative Aggregate Weight

The first step in this calculation is to compute the optimal aggregated weight we are targeting for each switch based on the outcome of the optimization.

$$AW_n = \sum_{p=0}^{P} w_{n,p}$$

where n is the switch number, p is the path number, P is the number of all paths, $w_{n,p}$ is the processing weight for switch:n corresponding to path:p, and AW_n is the aggregate processing weight at this switch.

- Denote by CAW_n the cumulative aggregate weight at switch:n
- Denote by TTW the sum of all path weights in the network (total tracking weight in the whole network)
- Denote by $HCW_{n,p}$ the highest previous cumulative weight seen on path:p from switch:n.
- Denote by PW_p the path weight of path:p

```
- For each source-destination pair in the network, extract the directed acyclic graph (DAG) from source to destination
- Starting at the direct switch to the source, add the switches of the DAG in a queue, where a switch is appended to the
queue if and only if all of its previous switches in the DAG are already in the queue.
- For each switch in queue:
        - Solve for $CAW_{switch}$:
            $\sum_{p=0}^{P} \frac{CAW_{switch} - HCW_{switch,p}}{TTW} * \frac{PW_p}{TTW} = \frac{AW_{switch}}{TTW}$
- Update HCW for each path
```

The equation provided above solves for CAW_{switch} in such a way to enforce the required AW_{switch} taking into consideration the CAW of previous switches on the path. However, since we are building the cumulative aggregate weights for switches over each other, the only one that will matter is the one of the highest value corresponding to each path, hence the HCW. This is because, with regards to packets arriving at the switch unprocessed, when the CAW of the switch is less than the HCW seen by it on the path, the CAW of the switch will not be sufficient to accept the state processing of the packet. And thus the possibility of processing only arises when the CAW of the switch is higher than the HCW seen by it.

Note: These values can be directly translated into probabilities to feed the bucket weights through dividing by the TTW, which is the amount of tracking in the whole network.

Considerations:

Whenever we arrive at a CAW_{switch} smaller than the HCW of a considered path **p** in the summation, this path is ignored and we reiterate the equation while disregarding this path. This is because a cumulative aggregate weight that takes into consideration a path in which it is lesser than its HCW, will have no effect on the contribution towards packets arriving on this path, and thus this path will not have been considered. The equation is reiterated until the cumulative aggregate weight is consistent with the paths under consideration.

Whenever we arrive at a CAW_{switch} larger than TTW, the CAW_{switch} is considered exactly equal to TTW. This is because it does not make sense to assign the switch more responsibility to process state-instances than the total responsibility required in the whole network, which translates to processing probability $\mathbf{CAW_{switch}}$/TTW **> 1**. This impossible extra effort required by the switches to assume their optimal responsibilities is the reason behind the distribution inconsistencies mentioned earlier.

Previously, the cumulative processing probability of the last switch to destination used to be always 100%, where it allows no packet to pass through the network unprocessed. If inconsistencies arise in the distribution, it will propagate to the last switch where its cumulative processing probability will not hit 100%. To maintain the correctness of this approach, the last switch must extend its assigned cumulative probability to 100%. As mentioned earlier, the question of how to redistribute the inconsistencies can be answered through an optimization approach to path aggregation.

If we apply this approach to the previous network example in Figure 19, we will arrive at the following results:

Table 9: optimal burden distribution cumulative probability results according to path aggregation approach

Processing Burden	Aggregate weights	Cumulative Aggregate Weights	Actual Processing Weights
Switch:1	25	25	25
Switch:2	25	75	25
Switch:3	25	125	25
Switch:4	25	175	25
Switch:5	25	75	25
Switch:6	25	125	25
Switch:7	25	175	25
Switch:8	25	200	25
Total processing weight in network	200	FULL	200

As per these results, the actual processing of each switch appears to be the same as the optimal aggregate processing weight. Without any explicit path considerations, the network converges to the optimal distribution conditions on all paths from host:red to host:orange. We also point out that the CAW of switch:6 is less than that of switch:4, but higher than that of switch:5. This means that switch:6 will only be processing packets arriving from path:1, since if it arrives from path:2, the CAW at switch:6 is not enough to cover $HCW_{6,2}$= 175.

To illustrate a scenario where inconsistencies appear, we skew the path weights to the side of higher number of switches such that: path:1 = 100, path:2 = 800. The results appear as follows:

Table 10: Non-optimal burden distribution cumulative probability results according to path aggregation approach

Processing Burden	Aggregate weights	Cumulative Aggregate Weights	Actual Processing Weights
Switch:1	114.28	114.28	114.28
Switch:2	114.28	242.85	114.28
Switch:3	114.28	371.42	114.28
Switch:4	114.28	500	114.28
Switch:5	100	900	87.3
Switch:6	114.28	628.57	114.28
Switch:7	114.28	757.14	114.28
Switch:8	114.28	886	114.28
Total processing weight in network	900	Deficit	886

Here, switch:8 did not arrive at a CAW = TTW (900), but 14.3 less. This is because switch:5 reached its maximum applicable responsibility at 87.3 where it should have taken 100. Thus, the 12.7 deficit in switch:5 propagated along the path to appear on the converged path as 14.3 according to:

$$12.7 * \frac{TTW}{TTW - sumOfPathsAtswitch{:}5} * \frac{sumOfPathsAtswitch{:}5}{weightPath{:}1} = 14.3$$

It is important to note here that the inconsistency appeared has nothing to do with the fact that the aggregate paths did not show up to be equal on all the switches, but because originally the explicit path information is lost and switch:1 cannot allocate a single value of processing that can make the network converge to its optimal conditions of distribution. Subsequently, this inconsistency propagates from the allocation taken at switch:1. Any attempt to recalibrate the cumulative processing at switch:1 to trick the network to reach its optimal conditions will, in itself, break the optimal conditions. This is because, the optimal conditions are based on the fact that switch:1's processing weight is exactly 114.28 out of 900.

3.9.2.2 Optimized Computation of Aggregate Cumulative Weight

Similar to the least squares method used before, an optimized approach to the aggregate cumulative weights computation can be done by minimizing the square of the difference between the actual weight on the switches and the optimal aggregate weight.

- $obj.\,function \;=\; min \sum_{n=1}^{N} (\; TW_n - AW_n\,)^2$, where n is the switch number, TW_n and AW_n is the actual weight and computed optimal aggregate weight of switch n respectively

The CAW is included in the actual weights on the switches as following:

- $TW_n = (\sum_{p=1}^{P} (CAW_{\,n} - \;HCW_{n,p}) * \frac{pw_p}{TTW})$, where n is the switch number, p is the path number, CAW_n is the cumulative aggregate weight of switch n, $HCW_{n,p}$ is the highest cumulative weight seen on switch n corresponding to path p, pw_p is the path weight of path p, AW_n is the computed optimal aggregate weight of switch n, and TTW is the sum of all processing weights in the network.

Accordingly, the objective function can be written as:

- $obj.\,function \;=\; min \sum_{n=1}^{N} (\; (\sum_{p=1}^{P} (CAW_{\,n} - \;HCW_{n,p}) * \frac{pw_p}{TTW}) - AW_n\,)^2$

Subject To:

- $CAW_N \;=\; TTW$, where N is the last switch to destination
- $CAW_n \leq \; TTW$

HCW cannot be accessed directly in the objective function since it is not precomputed like PW, TTW, and AW. To overcome this, we will further dissect the term that involves HCW into several terms with additional constraints that will satisfy its meaning. The CAW - HCW of a switch on the path is nothing but choosing the highest CAW of one of the previous switches on the path. Also, a prerequisite for accounting for the path in the objective function is that this difference between the CAW of the switch and the highest previous CAW on the path is positive, otherwise, the path is disregarded (i.e. is fully accounted for by previous switches).

Thus $CAW_n - HCW_{n,p}$ Can be written as:

- $\sum_{i=0}^{n-1} \lambda_{n,i}(CAW_n - CAW_i)$, this summation is used to consider the term for each previous switch i. However, it will eventually fold down to a single term since only one $\lambda_{n,i}$ will be non-zero. That is, a single CAW_i will be chosen as the $HCW_{n,p}$. To satisfy these conditions, we subject our objective function to the following extra constraints:

1. $\lambda_{n,i} * (CAW_n - CAW_i) \geq 0$
2. $\lambda_{n,i} * (CAW_i - CAW_k) \geq 0$, where k ranges from [n-1,i+1]
3. $\lambda_{n,i} * (CAW_i - CAW_l) \geq 0$, where l ranges from [i,0]
4. $\lambda_{n,i} \geq 0$

The first additional constraint ensures that the CAW of the current switch (switch:n) is higher than that of previous switch (switch:i) that we are considering it possesses HCW.

The second additional constraint ensures that the CAW of the previous switch under consideration is higher than the CAW of all the switches that fall after it on the path to the current switch.

The third additional constraint ensures that the CAW of the previous switch under consideration is higher that the CAW of all the switches that fall before it on the path.

If any of these constraints is not met, then the switch is not taken as the one possessing HCW. As it appears in the additional constraints 1,2,3, all the terms are multiplied by $\lambda_{n,i}$. If the terms appear to be negative, then in order to satisfy the positivity of the constraint $\lambda_{n,i}$ must be zero or negative. And the fourth constraint then ensures that $\lambda_{n,i}$ will be zero if these terms are negative. Choosing which previous switch has the highest CAW and discarding all the others is asserted by λ , where if it is equal to zero, the term is discarded, and if it is equal to one, the term is taken.

CHAPTER 4

FRAMEWORK EVALUATION

4.1 Testing Environment and Setup

On an Ubuntu Linux machine with kernel version 4.4.0, the network environment is simulated using mininet [38] (version 2.2.1) which takes as input a topology file and accordingly creates the corresponding network comprised of openVswitch bridges (OpenFlow programmable switches, version 2.8.90), links, and hosts (namespaces). The proposed SDFS engine is implemented in python as a proof of concept uploaded to github [39]. The engine's input is a configuration file, and a topology to provision secured with a distributed connection tracking application. The configuration file comprises parameters including the routing behavior in the network (Layer-3 vs Layer-4 load balancing), the path weights in the network, the distribution strategy (path-explicit vs path-oblivious), the decision-processing technique (using vs dismissing the announcement bit), the hosts to protect, and fast-failover adoption.

Upon execution, the engine monitors the topology and determines the set of paths in the network where the convergence switches are discovered. The optimization procedure discussed in section 5.4.3 is then applied to determine the optimal tracking weights for each switch corresponding to each path. Then, depending on the distribution strategy (path-explicit approach, additive cumulative average approach, path aggregation approach), the cumulative weights are computed for each switch and then transformed into Accept/Deny bucket weights. Subsequently, the engine installs the corresponding flows on the switches upon which the distributed application becomes alive in the data-plane to process the packets.

The primary topology structure used in testing is depicted in Figure 20, which corresponds to the campus networks used in SNAP, which also depicts the same structure of tiered data-centers: access layer, aggregation layer, Core layer, and edge routers.

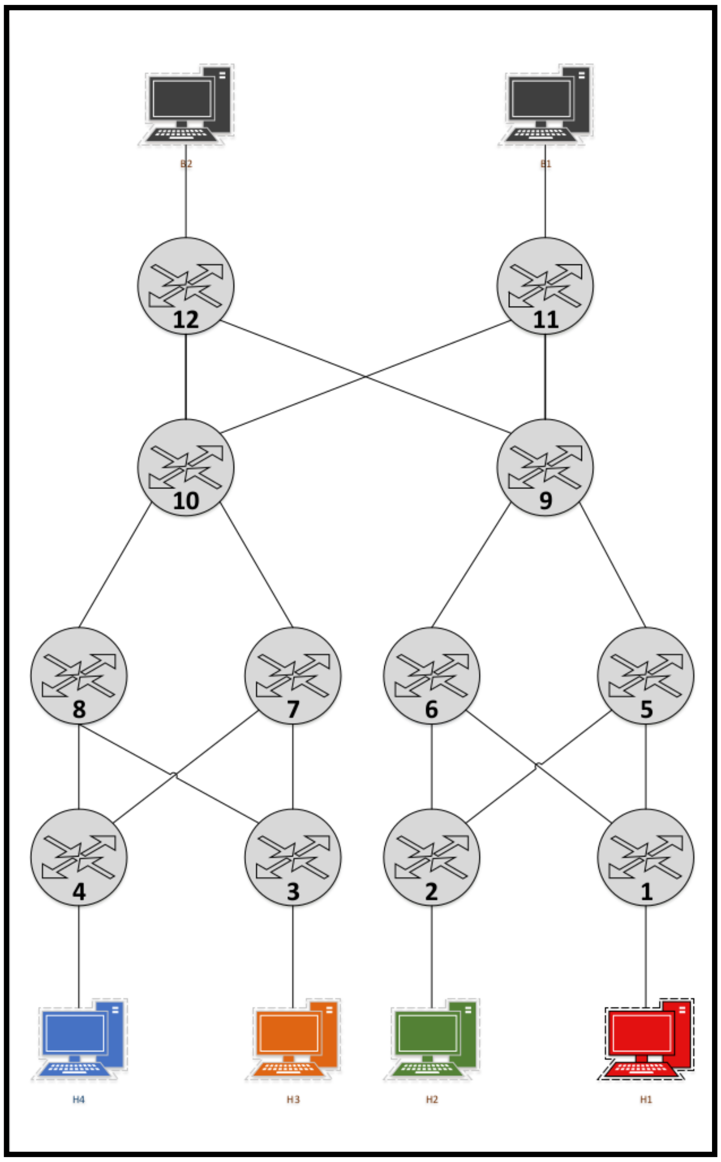


Figure 20: Tiered topology, primary structure

Figure 20 illustrates this topology for a connection tracking application where hosts red, green, orange, and blue are the protected hosts in the network. In other words, the other hosts in the network (black hosts) cannot communicate with the protected hosts unless the protected hosts initiate the connection. In this topology, each protected host has multiple route paths to the edge routers. However, whether L3 or L4-load-balancing was employed, all the switches on the path of any connection are convergent switches on the connection tracking application. Thus, the whole network can be transformed into one big firewall for the connection tracking application. The existence of multiple routes to the destinations qualifies the distributed firewall application for the high available features, where the network can redeem itself to maintain its correctness by handing over the connection processing to other switches upon switch failure.

4.2 Experiments

4.2.1 The direct switch framework experiment

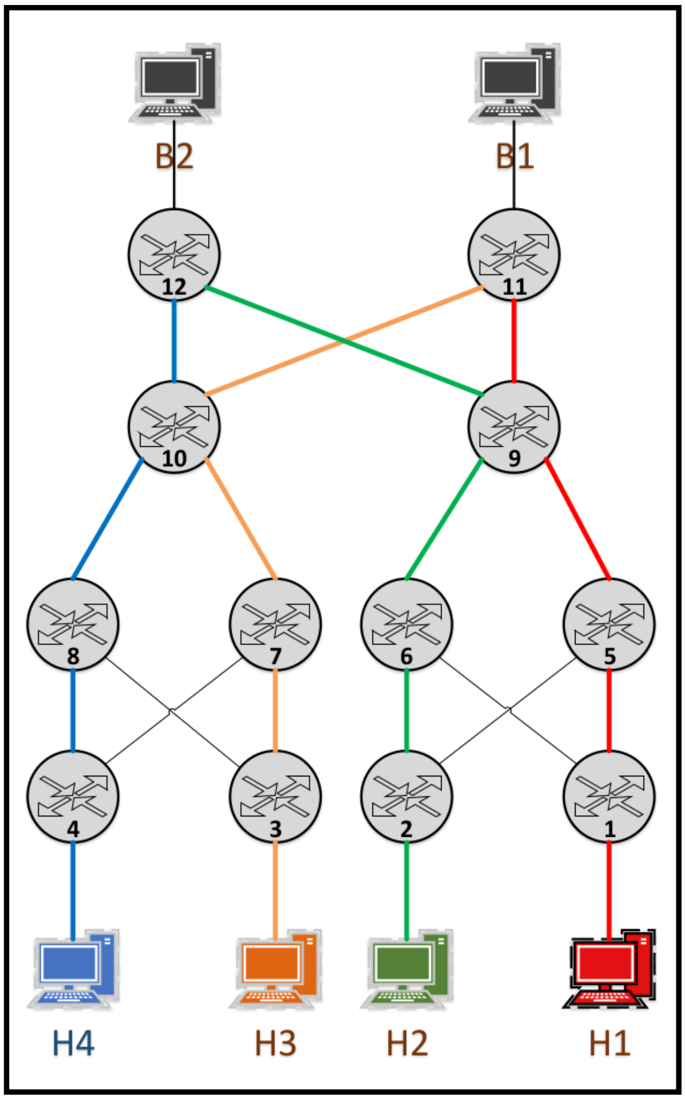


Figure 21: Tiered topology, L3 routing

In this experiment, we study the correctness of the direct switch framework, in particular, the distribution of the processing burden upon the direct switches only (e.g. switch:1 for host:red, switch:2 for host:green). As displayed in the Figure 21, we use the same topology structure with L3 routes as colored in correspondence with the protected hosts. Accordingly, we give each one of the four paths a weight of 1000. Thus in total, we have 4000 connections being established in the network (1000 for each host). We take a snapshot of the amount of connections being processed at each switch after each 100 fired connections from each host (increments of 400 new connections in the network).

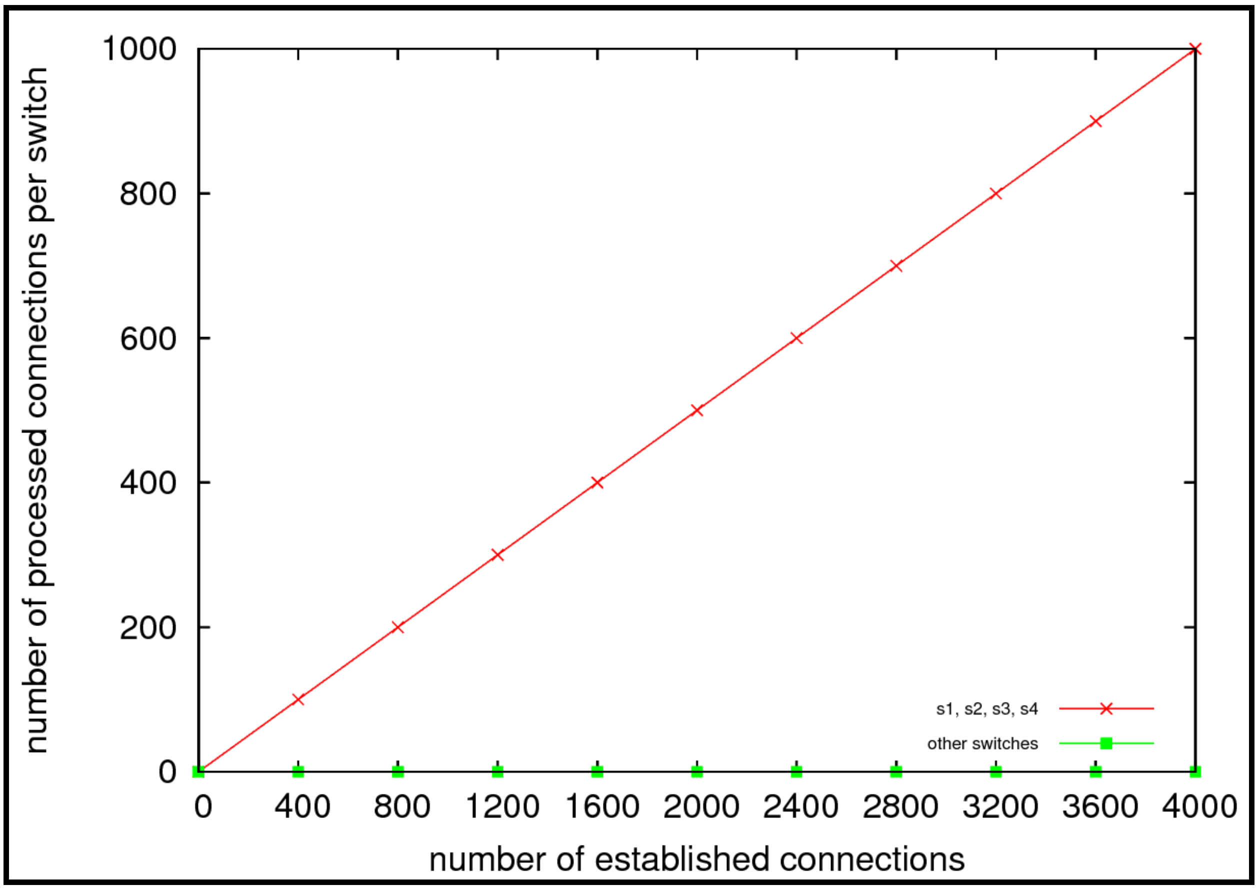


Figure 22: Processing distribution in the direct switch framework

Figure 22 shows the result of the burden distribution. Since we are using the direct switch framework, each direct switch (s1, s2, s3, s4) processes every new 100 connections fired from its corresponding direct host.

4.2.2 The path explicit case experiment

In this experiment, we use the same network and routes as that in Figure 21 to test the path-explicit scenario. However, instead of tracking at the direct switch, we employ all the switches to partake in the processing burden and use an announcement header to disseminated the decision processing information. The cumulative weights are computed as discussed in section 3.5.2.2.

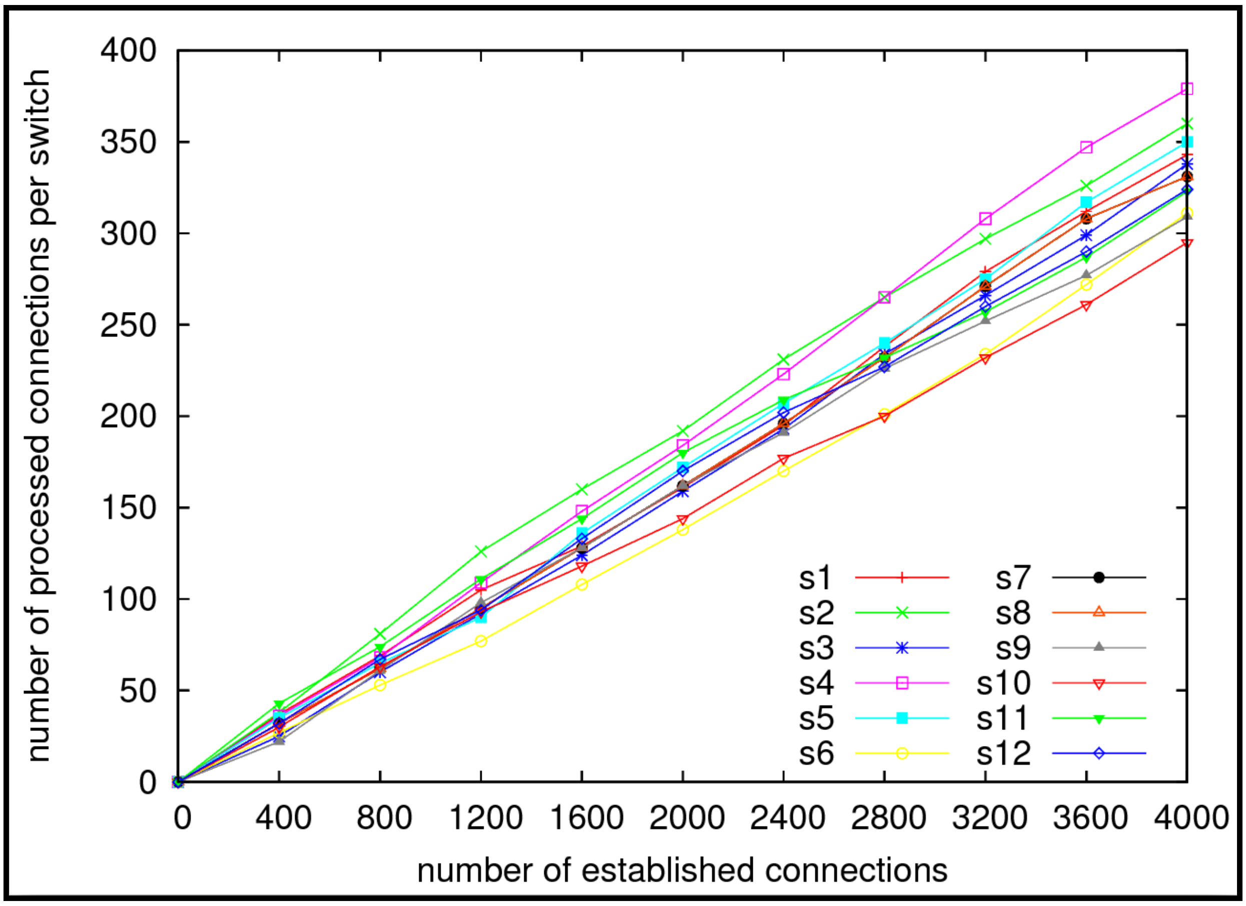


Figure 23: Processing distribution in the Path Explicit framework

Figure 23 shows the amount of processed connections per switch at each snapshot. Since we have 12 switches, optimally, each switch should be sharing 1/12 of the processing burden in the network at any particular snapshot. As can be seen in the results, each switch roughly processes 33 connections when the number of fired connections in the network is 400. Since the packet fields are chosen at random, where each combination resembles a connection, the processing decision taken at each switch is probabilistic. This is why we see a slight divergence in the distribution as we arrive at 4000 established connections. While the optimal processing burden at 4000 established connections is 333 for each switch, switch:10 processes 295 connections while switch:4

processes 379 connections. Hence, fair distribution is achieved even with these slight differences.

4.2.3 The path explicit case with capacity constraints experiment

In this experiment, we use the same properties for provisioning the network as that of Experiment 2 (section 4.2.2), however we add a capacity constraint of 200 on switch 9 as discussed in section 3.4.2.2 as the fourth constraint. This capacity constraint will limit switch 9's relative processing capacity to 200 per the total traffic processing weight (4000 connections).

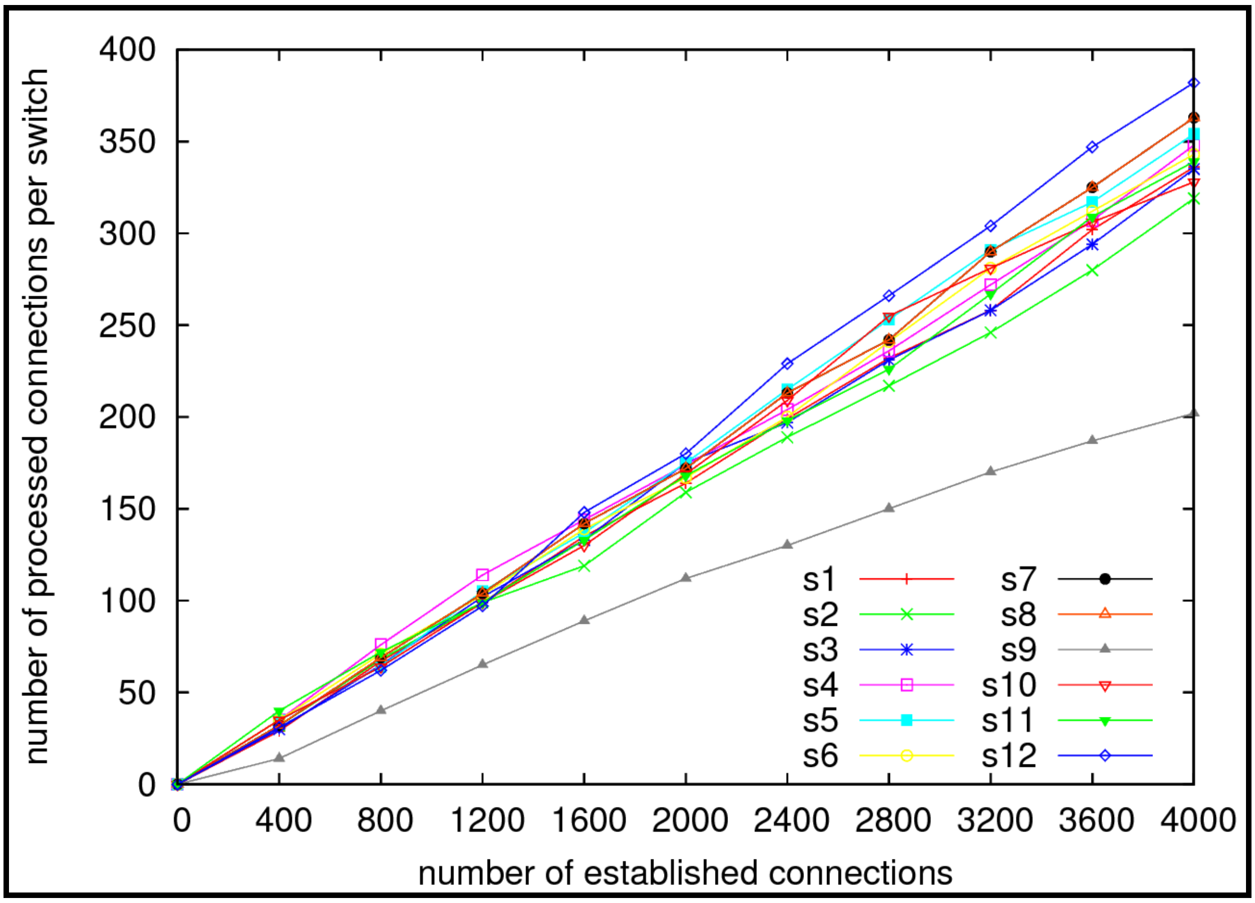


Figure 24: Processing distribution for capacity constrained scenario

As can be seen in Figure 24, at each snapshot taken, switch 9's relative processing burden was 200/4000*number of established connections. For example, when the amount of established connections in the network was 2000, switch 9’s processing burden was constrained to 100. Also, when the amount of established connections was 4000, switch 9’s participation was limited to 200. However, the remaining 3800 connections in the network are distributed equally among the other switches. While the optimal processing burden for the switches was 333 in the unconstrained scenario, this burden increases to 345 in this scenario where the remaining switches collaborate to cover switch 9’s shortage.

4.2.4 The path explicit case with announcement bit dismissed experiment

In this experiment, we investigate the correctness of the application while dismissing the announcement bit. Network is provisioned as that of experiment 2 (section 4.2.2), however checking whether a packet belongs to a connection that is being tracked at the current switch does not entail using a VLAN header but through preprogramming the switches to possess the tracking distribution of the corresponding previous switches as discussed in section 3.8. In this technique, each subsequent packet in a connection is subjected to the select groups to determine whether the packet should be expected to be tracked here and consequently whether to track the packet, forward it, or drop it.

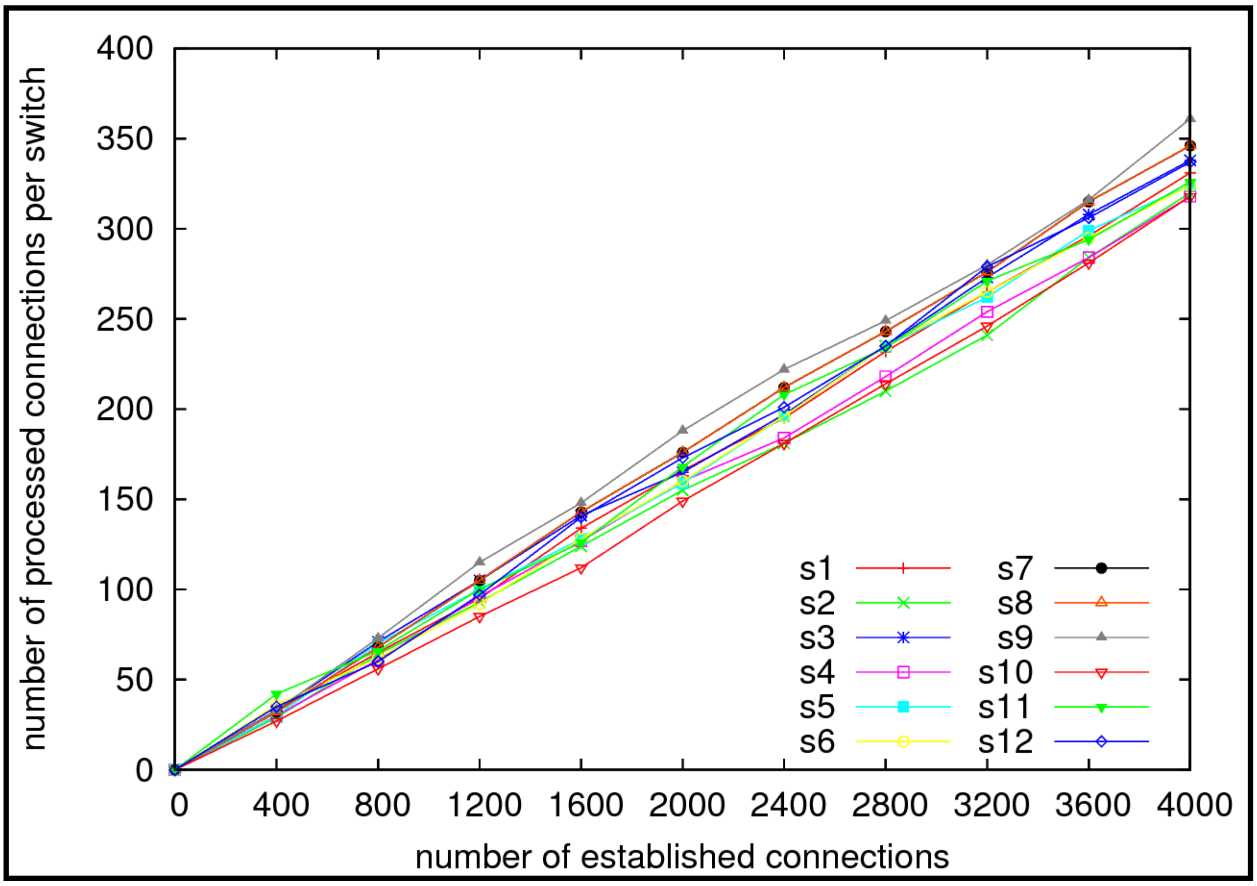


Figure 25: Processing distribution for dismissing announcement bit scenario

The results of this experiment are shown in Figure 25. Similar to experiment 2, the switches share a fairly equal burden through all the snapshots.

4.2.5 The path explicit case with fast failover experiment

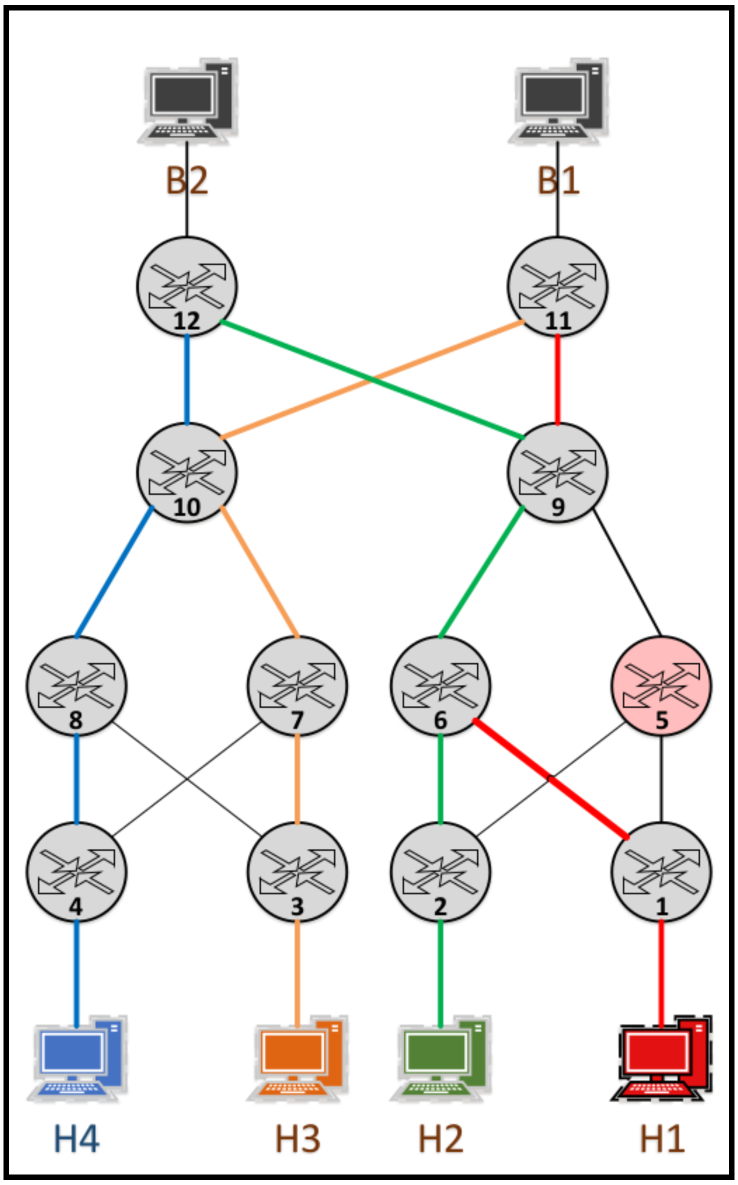


Figure 26: Tiered topology, failover routes

To investigate the high-availability of the application as discussed in section 3.5, we simulate the same provisioned topology as that of experiment 2 (section 4.2.2), however, switch 5 fails midway. Accordingly, a fast-failover mechanism takes place at switch 1 where instead of forwarding traffic to switch 5, switch 1 detects the failure of switch 5 and forwards the traffic to switch 6. As can be seen in Figure 26, switch 6 is a failover switch on the path. And while switch 6 can take part in the burden processing, it does not have to, and accordingly it can effortlessly leave the burden onto switch 9, precisely because the announcement bit keeps track of such decisions. Here we show how the processing burden can be handed-over to switch 9 when switch 5 fails.

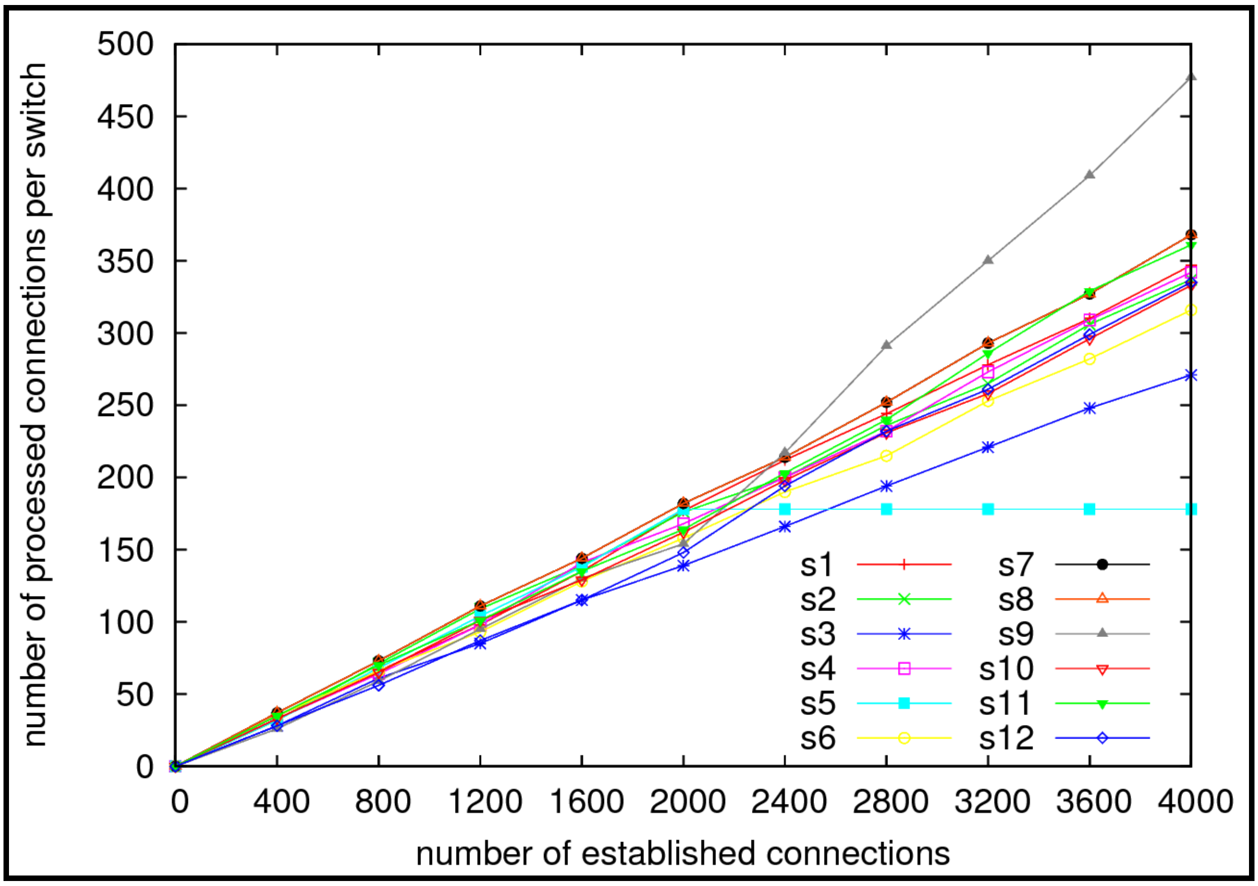


Figure 27: Processing distribution for fast-failover scenario

As can be seen in the results in Figure 27, when switch 5 fails, it stops accepting new connections, while this burden is carried over to switch 9. We note here that switch 5 does not drop back to zero because we are measuring the number of processed connections per switch (which entails the initial packets of the connection), in contrast to the number of the connections that are open simultaneously. In other words, when switch 5 fails, all the connections that were being processed by it are lost. However, subsequent connections that are expected to be processed by switch 5, by default, are now handed over to switch 9.

4.2.6 The path explicit case with fast fail-over and announcement bit dismissed experiment

We provision the same scenario as that of experiment 5 (section 6.2.5), however we dismiss the announcement bit in this experiment. This is to show the correctness of dismissing the announcement bit in the application in a failover scenario. As discussed previously, dismissing the announcement bit necessitates that the switches that diverge in the reverse path have the same cumulative processing weight. In this case, these switches are switch 5 and switch 6. Precisely because switch 9 does not have access to which direction the packet will take in the reverse direction to host:red, it is not able to predict whether it should track or leave for the next switch to track it in the reverse direction as mentioned in section 3.9.1. In this case, we force both switch 5 and switch 6 to have the same cumulative tracking weight. In other words, switch 6 must take part in the burden distribution using the same cumulative weight as that of switch 5.

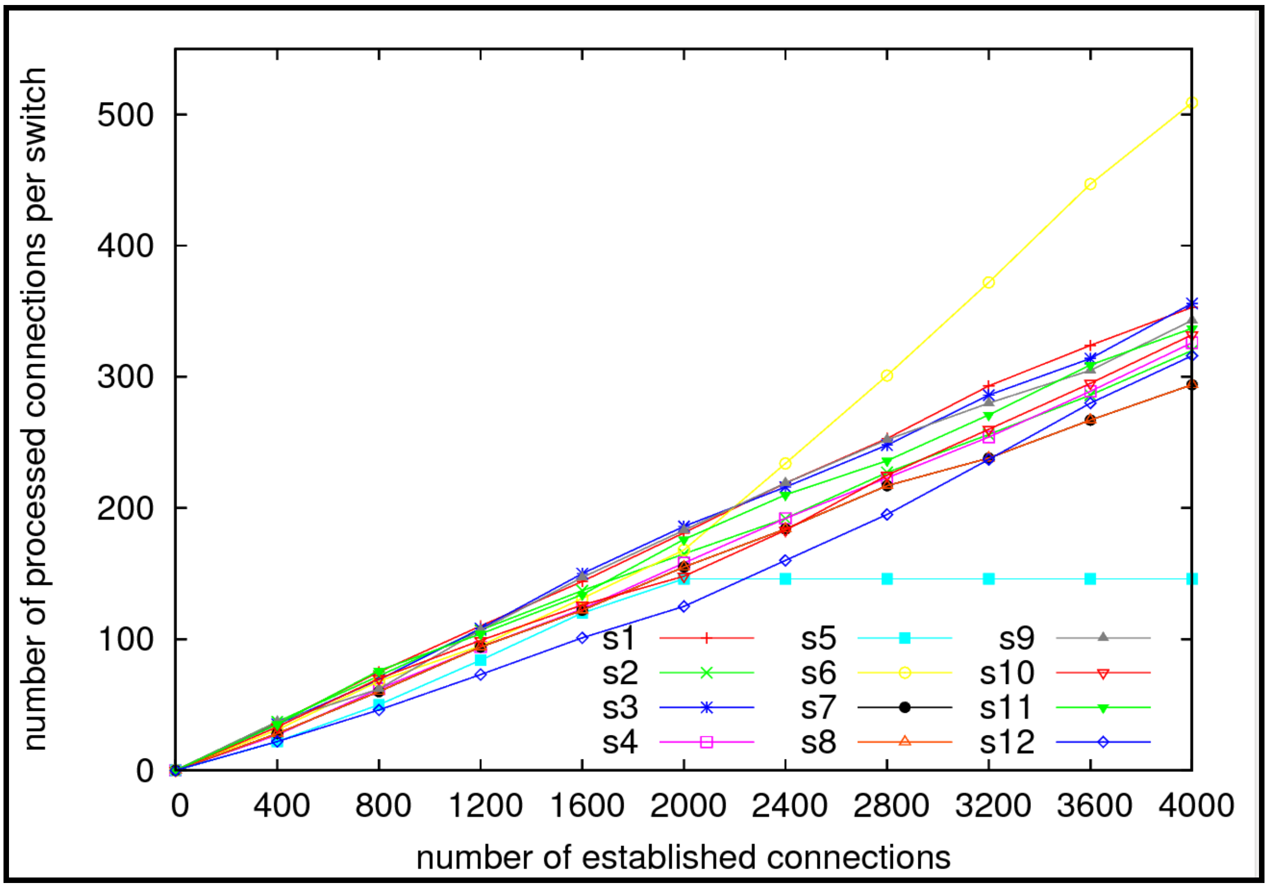


Figure 28: Processing distribution for fast-failover scenario with dismissing announcement bit

The results for this case are shown in Figure 28. Similar to experiment 5 (section 6.2.5), when switch 5 fails, it stops accepting new connections. However, this burden is carried over to switch 6 in this scenario.

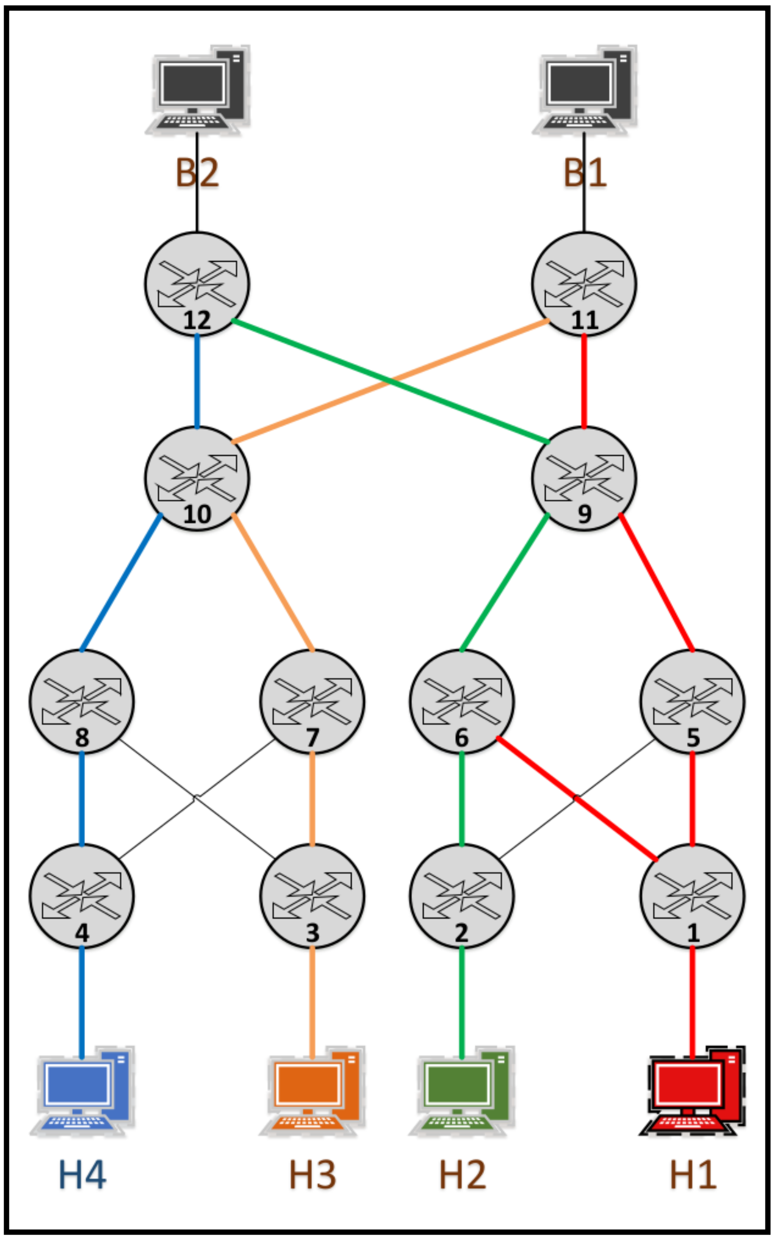


Figure 29: Tiered topology, L4-load-balanced routes

4.2.7 Additive cumulative average approach experiment

To investigate the correctness of the application in a load-balanced scenario, we employ the additive cumulative approach discussed in section 3.9.1 to distribute the burden. As shown in figure:29, switch 1 load-balances the traffic coming from host:red into both switch 5 and switch 6. We assume that the traffic from each protected host still has a weight of 1000. However, the traffic coming from host:red is now divided equally between the two viable paths.

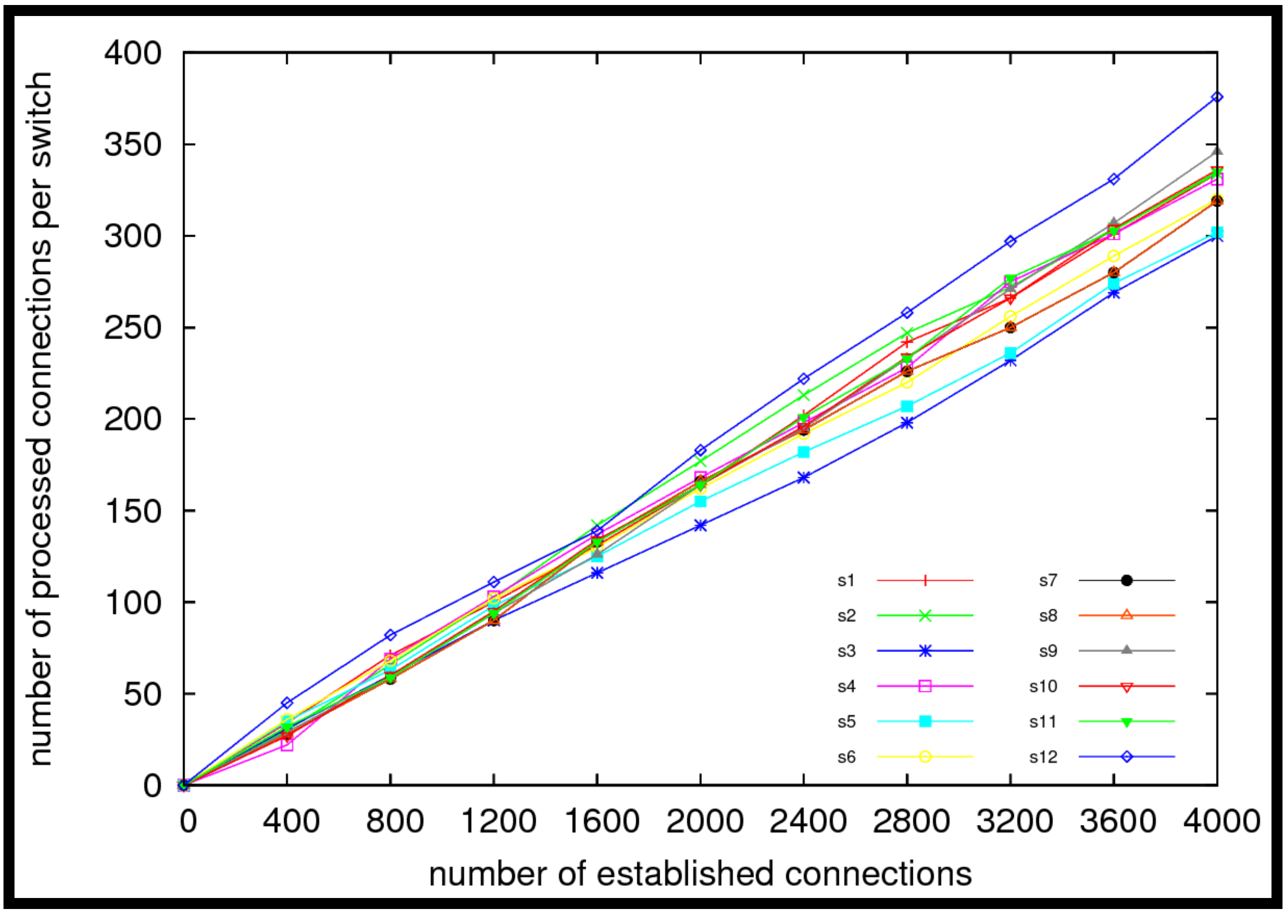


Figure 30: Tiered topology, L4-load-balanced routes

Figure 30 above shows the resulting processing behavior of the switches. Each switch fairly shares its assigned optimal burden. Even when switch 5 is only subjected to 500 connections coming from host:red, and switch 6 is subjected to 1500 (500 from host:red and 1000 from host: green), the approach redistributes the assigned cumulative weights so that the burden distribution arrives at its optimal behavior without necessitating that switch 1 has explicit information about the path of the packet (whether going to switch 5 or switch 6).

4.2.8 Path aggregation approach experiment

To investigate the correctness of the application using the path aggregation approach discussed in section 5.9.2, we employ this approach using the same load-balanced scenario depicted in experiment 6.2.7.

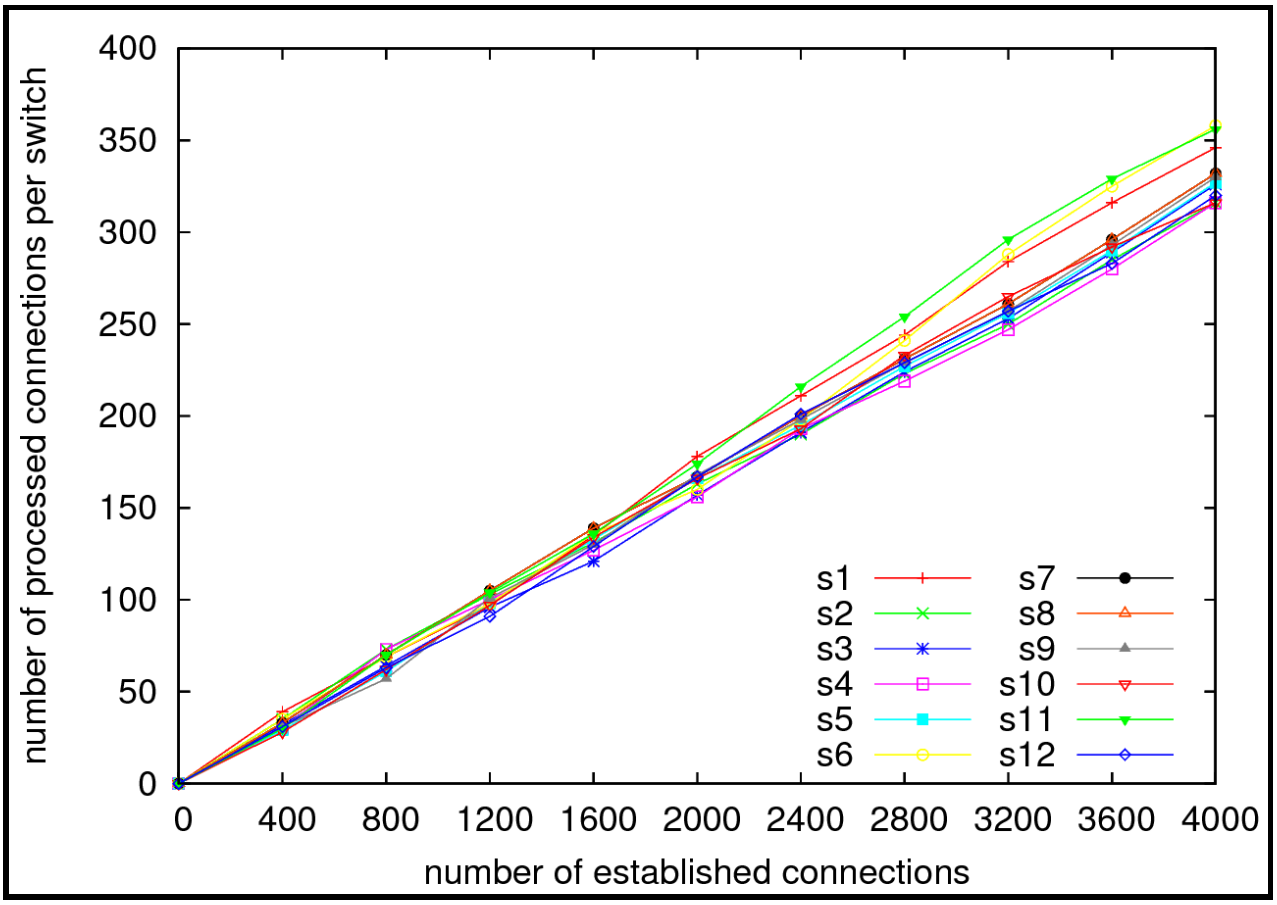


Figure 31: Processing distribution using path-aggregation approach

Figure 31 above shows the resulting processing behavior of the switches. Similar to the results of experiment 6.2.7, each switch fairly shares its assigned optimal burden.

4.2.9 Scaling with topology size experiment

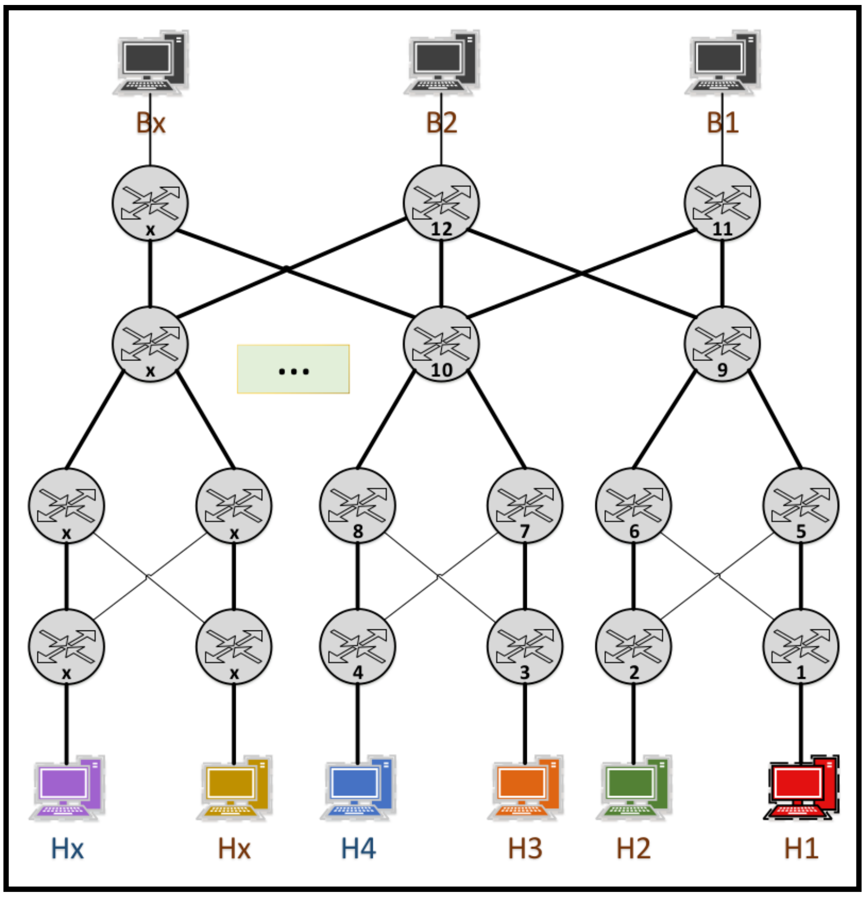


Figure 32: Extended tiered topology

In this experiment, we study the compilation time required to optimize the burden distribution upon the switches given the path weights in the network. For this reason, we incrementally extend the network by blocks of six switches, as displayed in Figure 32, and include the corresponding paths between the protected hosts and the edge routers. At every snapshot of a topology, we run the optimization once using a single route between a protected host and a black host (eg: H1->s1,s5,s9,s11->B1), once using 2 load balanced routes (eg: H1->s1,**s5**,s9,s11->B1 and H1->s1,**s6**,s9,s11->B1), and once using 4 load balanced routes (eg: H1->s1,**s5**,s9,s11->B1 ; H1->s1,**s6**,s9,s11->B1 ; H1->s1,**s6**,s9,s12->B2 ; H1->s1,**s5**,s9,s12->B2).

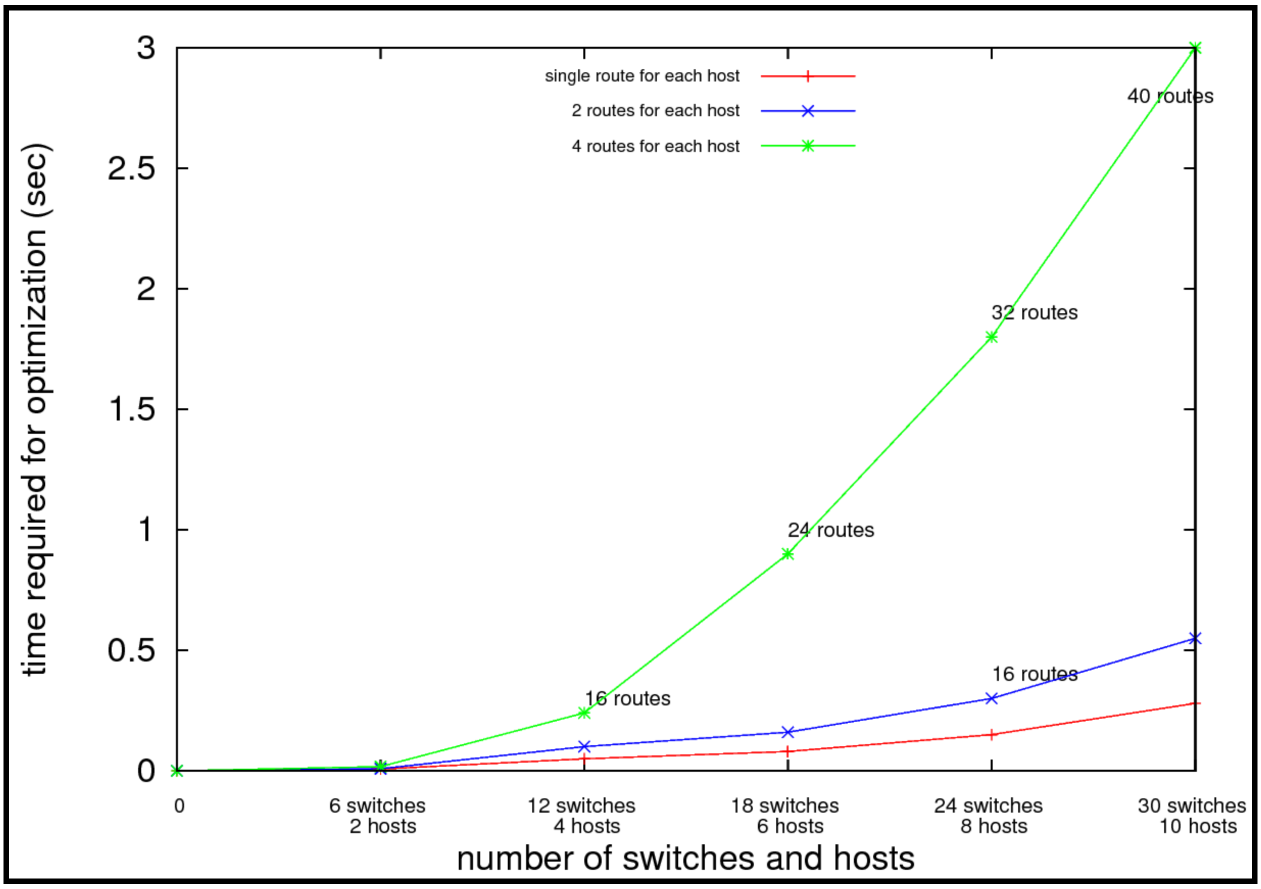


Figure 33: Results for the optimization time required for different topologies and network configurations

As can be seen in Figure 33, the time required to produce the optimization results fairly curves exponentially as the number of paths in the network increases. While essentially this is due to the complexity of the problem itself, it does not posit a restriction for large networks. This is because the network is not timewise critically dependent on obtaining a new distribution of the burden in the network. As mentioned before, the network inherently employs high-availability strategies such that the application in the data-plane can still function correctly even upon topology changes. This leaves the controller a fair amount of time to recompute the distribution.

Additionally, the distribution problem can tolerate semi-accurate results. Accordingly, approximation algorithms can be used to determine the distribution instead of utilizing accurate but slow line-search methods and steepest descent.

The same exponential behavior is also depicted in SNAP at a cold start of the network and upon policy changes. However, SNAP provides an incremental update (partial recompilation) of state-variable redistribution in the network upon topology changes where the time required for this update is critically reduced.

4.2.10 Packet Delay experiment

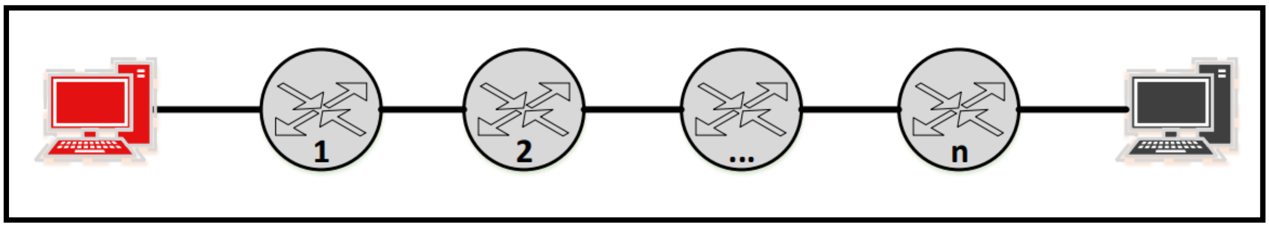


Figure 34: Trailing topology

In this experiment, we measure the effect of the decision announcement techniques on the connection bandwidth.

For the approaches that use the announcement bit technique, only the first packet in the connection initiated from the protected host (TCP:SYN) is subjected to the group select to determine whether to track the packet or leave it for the next switch while the subsequent packets in the connection are tagged with the announcement header (VLAN header) at packet speed.

However, as mentioned previously, for the approaches in which we dismiss the announcement bit, checking whether a packet is expected to be tracked at the current switch necessitates projecting every packet onto the select groups after hashing the L3-L4 fields.

To detect the impact of these processes on the connection bandwidth, we employ the trailing topology structure as shown in Figure 34 where we extend the number of switches as we measure the bandwidth for the three scenarios: direct switch, using the announcement bit, dismissing the announcement bit.

To test the connection bandwidth, we use the iperf tool to establish a connection and burst out packets in the connection from source to destination and back.

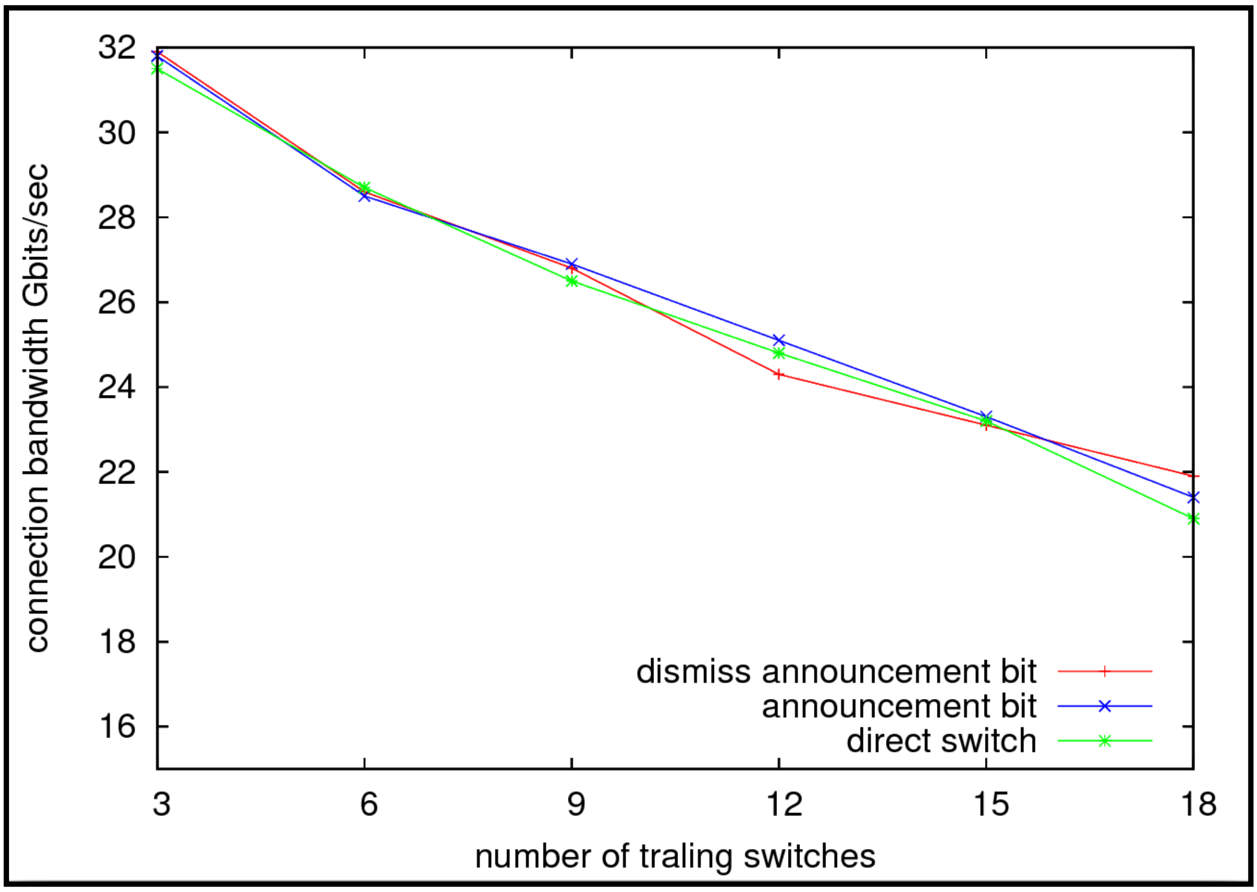


Figure 35: Results for connection bandwidth versus topology length for each decision announcement technique

As shown in Figure 35, no significant impact is recorded between the scenarios with what regards to the packet delay. We can then conclude that the time the packet spends being projected on the select groups or tagged by VLAN headers is overcome by other processes that occur per packet, such as packet-parsing, table resubmissions, or packet input/output.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This section summarizes the SDFS framework as it highlights the advantages introduced by its features along with the gaps that still exist in the framework. Additionally, it reiterates how SDFS relates to SNAP, and puts forward possible courses for extending SDFS in the future.

As discussed in this text, SDFS builds on top of the stateful data-plane framework to address issues pertaining to the security, correctness, reliability, and high-availability of SDN applications. We present SDFS's potential to transform the network into a one big firewall which offers an optimized processing burden distribution of the stateful application in the data-plane with inherent fault-tolerance mechanisms that eliminate the need for immediate controller intervention even in cases of failure or attacks on the network. This, in turn, maintains the correctness of the application in the data-plane, relieves the controller from DoS attacks, and revives the security techniques that can be used for loose-time dependent applications.

5.1 Feature Comparison between SDFS and SNAP

Table 11: Feature Comparison: SDFS VS SNAP

Feature	SDFS	SNAP
State distribution method	State-Instances: atomic, handle bidirectional relationship of variables	State-Variables: global arrays of the state hosted in the data-plane
Optimization	Cost-aware. Distributes burden upon the switches. Capacity-aware: Switches can be relatively constrained in the processing capacity	Cost-aware. Distributes burden upon the switches. Capacity-aware: Optimizes the distribution while considering link capacities
Compilation time	Exponential with respect to the number of policies in the network in a cold start. Does not implement an incremental mechanism for state-placement.	Exponential with respect to the number of policies in the network in a cold start or upon policy change For cases of fixed state placement in the network, provides an incremental mechanism to recompute the distribution
Fault-tolerance	Inherent maintenance of the application correctness upon switch/port failure	Does not implement any particular fault-tolerance mechanism
Header fields for information dissemination	Uses announcement bits to broadcast processing decisions Provides a mechanism to dismiss the announcement bit	Uses dictionaries in packet Header fields to forward state-variable information between the switches
Modularity	Relaxes the constraints on manipulating the routing behavior in the network	Manipulates the routing behavior in the network to ensure traffic convergence
High level programming language	Does not implement a high level abstract language	Provides a high level programming language and its corresponding compiler

In reference to Table 11, SDFS has its advantages over SNAP through several features: State distribution method, fault-tolerance mechanisms, and modularity in terms of absence of header fields and preservation of the routing behavior in the network. On the other hand, SNAP beats SDFS in terms of an intelligent optimizer that reduces the compilation time on subsequent updates in the network by providing an incremental computation of the state-variable placement in the network. Additionally, SNAP provides a high level programming language exposed to the user in order to write SNAP programs. SDFS on the other hand does not implement such a language and its corresponding compiler, however provides the understructure of an optimized distribution as a framework.

5.2 Extending SDFS

We present in this section possible extensions to the SDFS framework to fulfill the gaps that have been highlighted through the text.

Traffic Convergence Discovery: It is not always the case in SDN that the controller has access to path information in the network. As explained previously in this text, load-balancers might be employed in the network in which case the controller is oblivious, at least on a high level, to the particular paths of distinct traffic. For this reason, we studied the path convergence constraints, in section 3.3, in order to point out the scenarios in which the switches in the network can be all-convergent with respect to the requirements of the stateful application and the imposed routing behavior in the network. We also pointed out that in other cases, all-convergence might not be granted, in which case, the controller must be able to construct the convergence in the network. Accordingly, SDFS can be extended to incrementally discover traffic convergence in the network by employing traffic analysis applications that monitor the traffic at the data-plane and accordingly reconstruct the particular paths traversing the network. It is very important for such an application, however, not to burden the controller with a large influx of traffic being tunneled to the controller for analysis purposes. Specialized applications that can be employed here include sFlow, short for “sampled flow”, and Time Series Data Repository (TSDR) for flow data collection and statistical analysis.

Burden Distribution Maintenance in Attack Scenarios: As discussed in section 3.5, the application in the data-plane maintains its correctness even upon switch/port failure which gives enough time to fix the network problem or reoptimize and redistribute the burden upon the remaining switches. However, upon such failure, the burden the dead switch was responsible for migrates to a single switch in the network. This however can have its detrimental effect in the network if the next switch could not handle the newly assigned responsibility and shuts down accordingly, in which case the switches may start choking up in series until the application breaks down in the data-plane. To address such scenarios, SDFS can be upgraded to inherently redistribute the burden in an optimized fashion. Similar to the proactive installation of the cumulative weights in the network for burden distribution, the switches can also be proactively supplied with the corresponding burden distribution to act over, upon a fast-failover mechanism.

Multiple Stateful Applications in the Data-Plane: As noted in section 3.7 under the heading “Should track at current switch”, matching on the path to determine which group:select the packet should be submitted to, necessitates that packet fields used in the group:select- which is employed to compute the hash value that accordingly either hits the Accept or Deny buckets in the group- are not altered before being routed. If multiple stateful applications are to be employed in the network where at least one of these applications alters the packet fields that are correspondingly used by any of the other applications, it is necessary for the SDFS engine to be able to detect such scenarios that might violate the correctness of the applications in the data-plane and consequently prohibit such violations in the distribution process.

High Level Programming Language: Writing low-level stateful programs can be a tedious and error-prone job especially when it involves distributing the global states upon multiple state-instances. While the current implementation of SDFS provides the framework for an optimized distribution of the states where the user does not have to worry about how to distribute the state-instances, the state transitions through OpenFlow learn action must still be implemented. SDFS can be extended to provide a high level programming language that translates the high level stateful application into OpenFlow rules. As such, the language should be agnostic to how the state is distributed into state-instances. Instead of burdening the user in this task, the language compiler should be able to automatically separate the global state into atomic and self-sufficient state-instances.

Incremental Computation of Burden Distribution: The time required to recompile and redistribute the state-instances upon the switches is exponential with respect to the number of policies employed in the network. Although SDFS avoids the urgency of this procedure by employing fault-tolerance mechanisms that maintain the correctness of the application upon changes in the network and provide more time for the controller to recompute the distribution, SDFS can also be augmented with incremental computation of the distribution similar to that employed in SNAP. Here, the compilation and distribution time on a cold start of the network remains exponential. However, this time can be critically reduced upon topology changes.

REFERENCES

[1] Casado, Martin, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. "Ethane: taking control of the enterprise." In *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 1-12. ACM, 2007.

[2] Casado, Martin, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. "SANE: A Protection Architecture for Enterprise Networks." In *Usenix Security*. 2006.

[3] Greenberg, Albert, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. "A clean slate 4D approach to network control and management." *ACM SIGCOMM Computer Communication Review* 35, no. 5 (2005): 41-54.

[4] McKeown, Nick, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks." *ACM SIGCOMM Computer Communication Review* 38, no. 2 (2008): 69-74.

[5] Jain, Sushant, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata et al. "B4: Experience with a globally-deployed software defined WAN." *ACM SIGCOMM Computer Communication Review* 43, no. 4 (2013): 3-14.

[6] Patel, Parveen, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern et al. "Ananta: cloud scale load balancing." In *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 207-218. ACM, 2013.

[7] Scott-Hayward, Sandra, Sriram Natarajan, and Sakir Sezer. "A survey of security in software defined networks." *IEEE Communications Surveys & Tutorials* 18, no. 1 (2015): 623-654.

[8] Giotis, Kostas, Christos Argyropoulos, Georgios Androulidakis, Dimitrios Kalogeras, and Vasilis Maglaris. "Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments." *Computer Networks* 62 (2014): 122-136.

[9] Skowyra, Richard, Sanaz Bahargam, and Azer Bestavros. "Software-defined ids for securing embedded mobile devices." In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pp. 1-7. IEEE, 2013.

[10] Wang, Ye, Yueping Zhang, Vishal Singh, Cristian Lumezanu, and Guofei Jiang. "NetFuse: Short-circuiting traffic surges in the cloud." In *2013 IEEE International Conference on Communications (ICC)*, pp. 3514-3518. IEEE, 2013.

[11] Zaalouk, Adel, Rahamatullah Khondoker, Ronald Marx, and Kpatcha Bayarou. "Orchsec: An orchestrator-based architecture for enhancing network-security

using network monitoring and sdn control functions." In*2014 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1-9. IEEE, 2014.

[12] Zhang, Ying. "An adaptive flow counting method for anomaly detection in SDN." In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pp. 25-30. ACM, 2013.

[13] Shirali-Shahreza, Sajad, and Yashar Ganjali. "Efficient implementation of security applications in OpenFlow controller with FleXam." In *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pp. 49-54. IEEE, 2013.

[14] Hinrichs, Timothy, Natasha Gude, Martın Casado, John Mitchell, and Scott Shenker. "Expressing and enforcing flow-based network security policies."*University of Chicago, Tech. Rep* 9 (2008).

[15] Fayazbakhsh, Seyed Kaveh, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. "FlowTags: enforcing network-wide policies in the presence of dynamic middlebox actions." In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 19-24. ACM, 2013.

[16] Collings, Jake, and Jun Liu. "An OpenFlow-based prototype of SDN-oriented stateful hardware firewalls." In *2014 IEEE 22nd International Conference on Network Protocols*, pp. 525-528. IEEE, 2014.

[17] Dangovas, Vainius, and Feliksas Kuliesius. "SDN-driven authentication and access control system." In *The International Conference on Digital Information, Networking, and Wireless Communications (DINWC),* p. 20. Society of Digital Information and Wireless Communication, 2014.

[18] Ahmed, Mohamed Fekih, Chamssedine Talhi, Makan Pourzandi, and Mohamed Cheriet. "A Software-Defined Scalable and Autonomous Architecture for Multi-tenancy." In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pp. 568-573. IEEE, 2014.

[19] Moshref, Masoud, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. "Flow-level state transition as a new switch primitive for SDN." In*Proceedings of the third workshop on Hot topics in software defined networking*, pp. 61-66. ACM, 2014.

[20] Yuan, Yifei, Rajeev Alur, and Boon Thau Loo. "NetEgg: Programming network policies by examples." In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks,* p. 20. ACM, 2014.

[21] Han, Wonkyu, Hongxin Hu, Ziming Zhao, Adam Doupé, Gail-Joon Ahn, Kuang-Ching Wang, and Juan Deng. "State-aware Network Access Management for Software-Defined Networks." In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies,* pp. 1-11. ACM, 2016.

[22] Dargahi, Tooska, Alberto Caponi, Moreno Ambrosin, Giuseppe Bianchi, and Mauro Conti. "A Survey on the Security of Stateful SDN Data Planes." *IEEE Communications Surveys & Tutorials* (2017).

[23] Zhu, Shuyong, Jun Bi, Chen Sun, Chenhui Wu, and Hongxin Hu. "Sdpa: Enhancing stateful forwarding for software-defined networking." In *Network Protocols (ICNP), 2015 IEEE 23rd International Conference on*, pp. 323-333. IEEE, 2015.

[24] Bianchi, Giuseppe, Marco Bonola, Antonio Capone, and Carmelo Cascone. "OpenState: programming platform-independent stateful OpenFlow applications inside the switch." *ACM SIGCOMM Computer Communication Review* 44, no. 2 (2014): 44-51.

[25] Bosshart, Pat, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger et al. "P4: Programming protocol-independent packet processors." *ACM SIGCOMM Computer Communication Review* 44, no. 3 (2014): 87-95.

[26] Arashloo, Mina Tahmasbi, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. "SNAP: Stateful network-wide abstractions for packet processing." In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pp. 29-43. ACM, 2016.

[27] Kreutz, Diego, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. "Software-defined

networking: A comprehensive survey." *Proceedings of the IEEE* 103, no. 1 (2015): 14-76.

[28] Kloti, Rowan, Vasileios Kotronis, and Paul Smith. "Openflow: A security analysis." In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pp. 1-6. IEEE, 2013.

[29] Wasserman, Margaret, and Sam Hartman. "Security analysis of the open networking foundation (onf) openflow switch specification." (2013).

[30] Shin, Seungwon, and Guofei Gu. "Attacking software-defined networks: A first feasibility study." In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 165-166. ACM, 2013.

[31] Benton, Kevin, L. Jean Camp, and Chris Small. "Openflow vulnerability assessment." In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 151-152. ACM, 2013.

[32] Shin, Seungwon, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. "Rosemary: A robust, secure, and high-performance network operating system." In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pp. 78-89. ACM, 2014.

[33] Porras, Philip, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. "A security enforcement kernel for OpenFlow networks." In *Proceedings of the first workshop on Hot topics in software defined networks,* pp. 121-126. ACM, 2012.

[34] Gibb, Glen, George Varghese, Mark Horowitz, and Nick McKeown. "Design principles for packet parsers." In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pp. 13-24. IEEE Press, 2013.

[35] Lara, Adrian, Anisha Kolasani, and Byrav Ramamurthy. "Network innovation using openflow: A survey." *IEEE communications surveys & tutorials* 16, no. 1 (2014): 493-512.

[36] Cartis, Coralia, Nicholas IM Gould, and Ph L. Toint. "On the complexity of steepest descent, Newton's and regularized Newton's methods for nonconvex unconstrained optimization problems." Siam journal on optimization 20, no. 6 (2010): 2833-2852.

[37] OpenVswitch. “Open vSwitch Manual”. http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt.

[38] “Mininet.” An instant Virtual Network on your Laptop (or other PC). www.mininet.org

[39] “SDFS.” Stateful Distributed Firewall as a Service. https://github.com/ali-hz/SDFS