POLITECNICO DI TORINO

POLITECNICO DI TORINO
Master Degree course in Communications and Computer Networks Engineering

Master Degree Thesis

Design of State Replication for Stateful Software Defined Networking

Supervisors
Prof. Paolo Giaccone
Prof. Andrea Bianco

Candidate
German Sviridov

Academic Year 2016-2017

Acknowledgements

I would like to thank my supervisors Prof. Paolo Giaccone and prof. Andrea Bianco for their guidance during the months I spent working with them. In particular I would like to thank Prof. Paolo Giaccone for always being willing to listen to my ideas and trying to help me develop them in something concrete. On the other hand I thank Prof. Andrea Bianco for enlightening me with his constructive criticism which I will treasure for years to come.

I would also like to thank my family for the support they provided me through my entire life and for helping me to become the person I am today. Additional thanks go to all of my friends, which are too many to thank individually, who supported me throughout my academic journey and provided me with motivation to always give the best of myself.

Finally I would like to thank Politecnico di Torino and all the academic staff involved in my Master's Degree for the passion they put in their work and for the exceptionally high organizational and teaching quality.

Abstract

The field of software defined networking (SDN) has been rapidly evolving in the recent years. Multiple per-packet processing functions are being defined in order to introduce a tighter control over the network. However, these new functionalities can create considerable overhead and significant delays for control information exchanged with the controller. These factors lead to a reduced reactivity to network events and an overall performance degradation.

In order to overcome these limitations a big effort is being devoted towards the definition of a novel approach based on stateful functionalities. The use of stateful approaches allows to offload some of the controller functions directly to the switches by defining persistent states that can hold state information related to per-packet processing rules. The proposed solutions however make use of a centralized state placement, eventually forcing the majority of traffic to pass through a single device which can create single points of failure and reduced performance.

The purpose of this research is to design and implement a scalable stateful solution that would lead to an increased performance and reliability. The idea behind the proposed approach is to distribute the state information across multiple forwarding devices while guaranteeing high consistency among the states. The proposed solution is able to provide lower overhead and thus better performance in respect to the solutions present in literature and is able to prevent state information loss in case of isolated failures.

The implementation and evaluation has been performed in an emulated network environment with virtual switches programmed in P4, an emerging data-plane programming language.

In a scenario of a DDoS attack towards an autonomous system by neighboring autonomous systems the solution based on distributed states led to a substantial improvement in terms of reactiveness and introduced overhead in respect to solutions based on classical SDN. The obtained results are also compared to the results produced by a collaborating research group whose effort was devoted to a yet non-standardized extension of an existing technology (OPP).

Contents

List of Figures v

1 Introduction 1
- 1.1 Problem description . . . 1
- 1.2 Main contributions . . . 1
  - 1.2.1 Formal problem definition . . . 1
  - 1.2.2 Definition of a scalable heuristic . . . 2
  - 1.2.3 Feasibility analysis . . . 2
  - 1.2.4 Design and Implementation . . . 2
- 1.3 Outline of Thesis report. . . . 3

2 Software Defined Networking 5
- 2.1 SDN as a paradigm . . . 5
  - 2.1.1 Controller . . . 6
  - 2.1.2 OpenFlow . . . 7
- 2.2 Enabling features of SDN . . . 8
  - 2.2.1 Network monitoring . . . 8
  - 2.2.2 Traffic engineering and orchestration . . . 8
  - 2.2.3 Devices' interoperability . . . 9
  - 2.2.4 Network Function Virtualization . . . 9
- 2.3 Issues in SDN . . . 10
  - 2.3.1 Reliability and scalability . . . 10
  - 2.3.2 Reactiveness . . . 10
  - 2.3.3 Absence of stateful operations . . . 10

3 Stateful Dataplanes 13
- 3.1 Stateful Dataplanes . . . 13
- 3.2 Stateful OpenFlow extensions . . . 14
  - 3.2.1 OpenState . . . 14
  - 3.2.2 Open Packet Processor . . . 14
- 3.3 P4 . . . 15
  - 3.3.1 Language specifications . . . 16
  - 3.3.2 Parse-Match-Action pipeline . . . 16
  - 3.3.3 Programmable parser . . . 18

3.3.4 Match-action tables . . . 20
3.3.5 Target-specific objects and functions . . . 21
3.3.6 $P4_{16}$ . . . 22
3.3.7 P4 Future Evolutions . . . 22
3.4 SNAP . . . 23
3.4.1 Problem description . . . 23
3.4.2 State dependencies and interoperability . . . 24
3.4.3 Optimization algorithm . . . 24
3.5 Virtual Network Embedding . . . 24
3.5.1 Problem composition . . . 25
3.5.2 Extensions and limitations . . . 26
3.5.3 Similarity with SNAP . . . 26

4 Data replication 27
4.1 Problem description . . . 27
4.1.1 State Sharding . . . 27
4.1.2 State replication . . . 28
4.2 Consistency Properties . . . 29
4.2.1 CAP Theorem . . . 29
4.2.2 Failure types . . . 30
4.3 CAP-Consistenct replication schemes . . . 31
4.3.1 2-Phase Commit . . . 32
4.3.2 3-Phase Commit . . . 33
4.3.3 Paxos . . . 34
4.3.4 Raft . . . 35
4.4 Eventually consistent replication schemes . . . 37
4.4.1 Motivation behind Eventual Consistency . . . 37
4.4.2 Causal consistency . . . 38
4.4.3 Anti-entropy . . . 38
4.4.4 Conflict-free Replicated Data Types . . . 39
4.5 CAP Theorem criticisms . . . 40
4.6 State-of-art and commercial implementations . . . 41

5 State replication for stateful dataplanes 43
5.1 Problem formalization . . . 43
5.1.1 VNE for state replication . . . 44
5.1.2 ILP formulation . . . 44
5.1.3 Heuristic for state replication . . . 50
5.1.4 Required consistency level for stateful dataplanes . . . 51
5.2 Replication triggering . . . 54
5.2.1 Event-driven triggering . . . 54
5.2.2 Value-delta triggering . . . 55
5.2.3 Time-delta triggering . . . 55
5.2.4 Controller-driven triggering . . . 56

5.2.5 Virtual queue-based triggering . . . 57
5.2.6 Hardware support . . . 58
5.3 Synchronization traffic transport and representation . . . 58
5.3.1 Update transport . . . 58
5.3.2 Update encoding . . . 60
5.3.3 Update dissemination . . . 61

6 Implementation and evaluation 63
6.1 Design choices . . . 63
6.1.1 Problem description . . . 63
6.1.2 Consistency level . . . 64
6.1.3 Update encoding and dissemination . . . 65
6.1.4 Update triggering . . . 65
6.1.5 Update generation . . . 65
6.2 Testbed . . . 66
6.2.1 Target topology . . . 66
6.2.2 Virtual environment . . . 66
6.2.3 Traffic generation . . . 67
6.2.4 Traffic capture . . . 68
6.2.5 Table manipulation . . . 68
6.2.6 Data retrieval . . . 68
6.3 Implementation . . . 69
6.3.1 Double counting . . . 69
6.3.2 Cloning . . . 70
6.3.3 State-update exchange . . . 70
6.3.4 Parser specifications . . . 70
6.3.5 Rate estimation . . . 71
6.3.6 Custom metadata . . . 72
6.3.7 High level description of the prototype . . . 72
6.4 Validation . . . 77
6.4.1 Test conditions . . . 77
6.4.2 Numerical results . . . 78
6.4.3 OPP implementation . . . 80

7 Conclusion and future works 83
7.1 Related Works . . . 83
7.2 Future Works . . . 84
7.2.1 Extensions of State Replication . . . 84
7.2.2 Definition of advanced replication schemes. . . . 84
7.2.3 Embedding application layer functionalities in the dataplane . . . 84
7.2.4 Extensions of the ILP formulation . . . 84
7.3 Conclusion . . . 84

Bibliography 87

List of Figures

2.1 High level description of SDN architecture. 6
2.2 High level description of OpenFlow switch architecture [27]. 8
3.1 High level abstraction of a P4 processing pipeline. 17
3.2 Transition diagram of a simple MPLS and IPv4 parser. 19
3.3 Example of stacked MPLS headers. 19
3.4 Example of single-hop VNR mapping with distinct substrate nodes. 25
3.5 Example of multi-hop VNR mapping with common substrate nodes. 25
4.1 Absence of linearizability in uncoordinated systems. 31
4.2 A summary of the functionality of 2PC algorithm. 33
5.1 State replication in NFV via synchronization VNR mapping. Link mapping not included in order to maintain clarity. 45
5.2 FSM with a criticality condition in state s_3 52
5.3 Example of piggybacking with excessive delays. 59
6.1 Test network topology for the DDoS detection scenario. 67
6.2 High level functionality of switches containing a state-replica. 73
6.3 High level functionality of switches without a state-replica. 74
6.4 Consistency among state-replicas in the scenario of 2 replicas. 79
6.5 Link occupation for data and synchronization traffic in the case of 1,2,4 copies for global state in P4 implementation. 80
6.6 Link occupation for data and synchronization traffic in the case of 1,2,4 copies for global state in OPP implementation. 81

Chapter 1

Introduction

1.1 Problem description

In stateful Software Defined Network (SDN) data planes, switches hold some local state, e.g., flow related states, and can execute simple programs to take local decisions. The recently proposed SNAP framework jointly addresses the placement of the local states and the flow routing problem to minimize the total network data traffic while guaranteeing that all flows traverse the switches storing the flow related states.

SNAP assumes one single copy of each state: this limits SNAP scalability in case of states with a global scope. To overcome this limitation, this work proposes a SNAP extension able to support state replication, i.e. distribution of multiple copies of the same state across the available programmable switches.

1.2 Main contributions

This work has been carried in collaboration with professors and researchers from Politecnico di Torino and Roma Tor Vergata from April 2017 till July 2017. The complete list of the contributors is available in the original paper [7]. Due to the elevated number of contributors involved in the definition of this work only a subset of the actual contributions are presented in deep details. In particular, additional attention is devoted to the contributions which involved an active participation of the candidate while it is still presented a brief overview of the other main contributions.

1.2.1 Formal problem definition

In order to provide support for state replication while still guaranteeing that the number of replicas is minimal and their placement is optimal starting from the SNAP formalization a new formulation is defined. Although the complete formulation of the problem is presented, major emphasis is dedicated to the presentation of the formal definition of the synchronization traffic which involved active participation of the candidate.

1.2.2 Definition of a scalable heuristic

Due to the high complexity of the ILP formulation of the problem an approximation algorithm to solve the problem in large networks is defined. The heuristic is able to solve the ILP problem in polynomial time while still guaranteeing low error margin in respect to the optimal solution.

1.2.3 Feasibility analysis

A large portion of this work is dedicated to the feasibility of providing support for replicated states in stateful dataplanes. The main analysis involved the investigation of state of the art replication schemes with the related advantages and drawbacks. It is discussed whether the currently available replication schemes can be applied in the scenario of stateful dataplanes and in particular how the definition of global rules is affected by the presence of replicated states.

1.2.4 Design and Implementation

The final contribution consists in the design and implementation of a DDoS prevention mechanism exploiting replicated states. The design is heavily influenced by the feasibility analysis, nonetheless by the intrinsic limitations of the involved technologies used to provide a functioning prototype. The implementation involves mainly 2 commercially available stateful dataplane programming languages, namely P4 and Open Packet Processor (OPP).

P4 Implementation

Additional attention is dedicated towards the implementation based on P4 which still composes one of the main contributions by the candidate.

This work was carried during the transition of P4 from version $P4_{14}$ to $P4_{16}$ a progression that brought essential features and alterations in the language definition. Although the implementation of the prototype was performed with the use of $P4_{14}$ and it was concluded before the release of $P4_{16}$, in this work it is felt necessary to present both versions in order to provide the reader with up-to-date understanding of this state of the art technology.

OPP Implementation

The details about the implementation exploiting OPP are left out of this this work since, a part from the proposal of general design choices, it did not receive any significant contribution from the candidate. For this reason the OPP implementation is briefly discussed and the results obtained with the prototype exploiting it are briefly presented and compared with those obtained with P4.

1.3 Outline of Thesis report.

The work is structured as follows. Chapter 2 provides an introduction to SDN and the main motivation that are behind its increasing success. Chapter 3 gives an insight on the current stateful dataplane implementations with major emphasis on P4, provides a description of the SNAP framework and its similarity to the *Virtual Network Embedding (VNE)* problem. Chapter 4 introduces the concept of data replication and analyzes the main types of replication schemes currently employed. Chapter 5 contains the main contributions for what regards the design of an optimal solution for state replication. Additionally, the possibility of implementing a replication scheme for currently available stateful devices is analyzed in deep details by recalling the replication mechanisms presented in Chapter 4. Chapter 6 provide an overview of the developed P4 prototype for state replication alongside with the numerical results and the comparison with the OPP implementation. Finally Chapter 7 contains the conclusions and suggestions for future works.

Chapter 2

Software Defined Networking

Software Defined Networking (SDN) has been an emerging paradigm since its proposal at the beginning of the century. Although SDN has not been a widely adopted technology in past years, nowadays it is receiving big amount of attention both from academia and from industries.

2.1 SDN as a paradigm

In legacy networking, i.e classic paradigm with decentralized coordination mechanism, each network device is responsible of performing coordination routines with other devices comprising the network. This coordination is necessary in order to guarantee the correct functionality of the whole system. An example of such mechanisms in legacy networks is the construction of a spanning tree for Layer 2 forwarding, neighbors discovering, resource allocation etc... However the vast majority of decentralized algorithms require expensive coordination mechanisms that may introduce substantial overhead in terms of exchanged information. Moreover in particular applications the failure of correctly delivering or correctly interpreting the control information may result in significant issues. In a scenario of a network comprising devices that adopt different versions of a spanning tree algorithm the misinterpretation of a control packet may induce a loop leading to partial or even total network blockage.

The main idea behind SDN is the removal of any decisional possibilities from the switches. In SDN switches become simple forwarding devices with a sole role of receiving and emitting packets and they are exempted from performing any decision that may affect the state of the network. In order to achieve this behavior however a decision mechanism capable of providing control information is required. This separation is commonly referred to as a separation of forwarding plane (also referred to as data plane) and control plane. Control plane, i.e. the decisional functionalities, is moved out from the switches and concentrated in a single entity which is able at any given time to have complete knowledge about the network and its condition. With this kind of knowledge the central entity, which referred to as *Controller*, can perform network wide optimization based on the global network state instead of performing decisions based on shortsighted information as it happens in legacy networks. A high level description of the aforementioned architecture

is depicted in Figure 2.1.

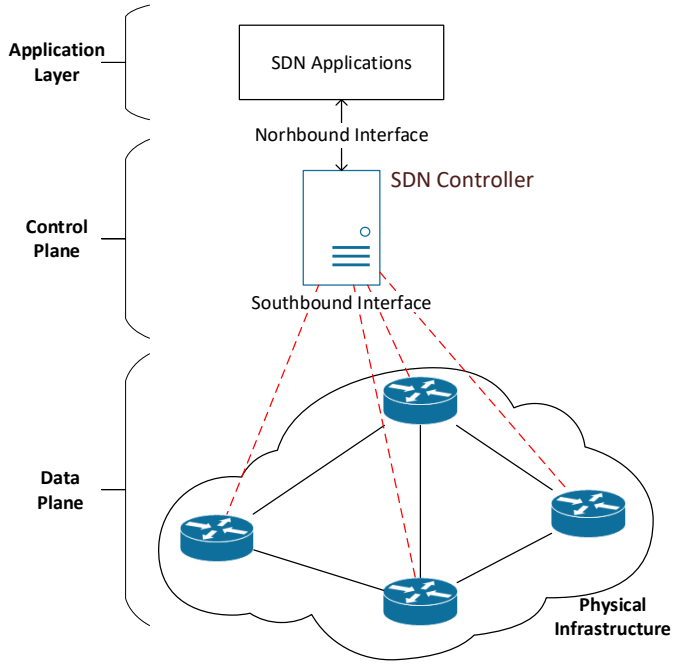

Figure 2.1. High level description of SDN architecture.

Although SDN has a long lasting presence in the academia as a research topic, it is starting to gain considerable attention from the industries. The possibility of having a centralized control over the whole network enables a rich set of features that can be exploited in order to improve network resource utilization and provide better control and manageability of the whole system which consequently leads to considerable reduction in operational costs. Moreover the switches' deprivation of any control mechanism results in much more reduced production costs by the manufacturers as no intellectual property must be included for the integration of control-specific hardware or software. This results in network equipment capable of delivering data-rates comparable with high-end legacy devices for a fraction of the price and with extended flexibility.

2.1.1 Controller

The controller is usually an application running on a dedicated server or in a visualized environment with the main role of providing the management of the physical infrastructure and to satisfy the application demands. In most of the commercially available implementations the controllers present two type of interfaces:

- **Northbound interface**: The role of this interface is to expose the controller to the application running on higher layers. These applications may include the necessity of

particular network conditions (e.g. low delay for VoIP) and are able to obtain these conditions by performing the corresponding request by using the APIs provided by the controller.

- **Southbound interface**: The southbound interface on the other hand provides an interface in order to disseminate the decisions made by the controller to all the forwarding devices by means of a particular southbound protocol.

Once a request is received on the northbound interface it is then the role of the controller to define how to satisfy the requirements of the request. The satisfiability of each request is evaluated by the controller's software and if the satisfiability is achieved the controller provides to reconfigure the network by including the received request. The reconfiguration may potentially involve global modifications in the forwarding plane and. Differently from legacy systems this behavior leads to optimality in the use of network resources, thus allowing to satisfy the maximum possible amount of requests while providing fast reconfiguration with low communication overhead.

2.1.2 OpenFlow

OpenFlow is one of the available southbound interface protocols and among the first standardized protocols that have been proposed in SDN [46]. Since the protocol provides means of communication between the switches and the controller it implies that both entities must be able to interpret it. Although there exist some hardware agnostic controllers which are able to operate independently from the adopted northbound protocol, in the case of OpenFlow switches are required to posses a particular hardware architecture.

OpenFlow switch architecture

The hardware architecture of OpenFlow switches is based on an internal pipeline composed by a series of *Flow Tables* as shown in Figure 2.2. Each flow table contains a fixed number of match fields which depends on the version of the protocol, an instruction (usually referred to as action) and a set of additional fields which description is out of the scope of this work.

Each match field is related to fields of commonly used packet headers such as IPv4, Ethernet, etc... thus allowing to unequivocally identify each flow traversing the switch. Each packet traversing the processing pipeline is subject to a lookup operation which is able to find whether the packet's fields match one or more flow tables. If a match is found, i.e. the packet header contains the same field values as those defined in one of the flow tables, the corresponding instruction is executed. Thanks to efficient hardware implementations this type of architecture allows to achieve wire-speed processing while at the same time keeping the hardware complexity low.

A side note on the intrinsic limitations of the adopted architecture must be done. As mentioned earlier OpenFlow is a standardized protocol with a gap between the release of new standards of 1-2 years. Since the matchable fields tightly depend on the specifications of each OpenFlow release the inclusion of new fields such as for the case of new transport protocols may be subject to considerable delay.

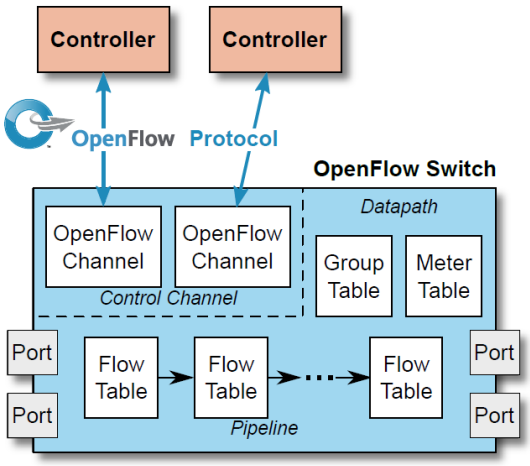

Figure 2.2. High level description of OpenFlow switch architecture [27].

2.2 Enabling features of SDN

2.2.1 Network monitoring

The centralized control and availability of global information allows to perform continuous monitoring which in legacy network was not possible or at least not easily achievable without expensive commercial software. SDN enables a rich set of features and functionalities that can be easily and efficiently implemented on top of any type of SDN-enabled network. The extended control over the state of the network allows to quickly react to sudden changes such as unexpectedly high traffic load for particular services or equipment failures.

2.2.2 Traffic engineering and orchestration

In legacy networks in case of unexpected network events devices react based on local information related the network state, e.g. in case of a failure on upstream links or devices, devices may exploit alternative paths present in the local configuration in order to still provide correct functionality. This method however requires devices which include this type of mechanism which usually comes with an increased manufacturing cost. Moreover since this type of decision does not take into account the state of the portion of the network that is exploited as a backup route, it may happen that if the backup portion of the network is already in a critical state (e.g. high load/congestion) the action of rerouting a flow from a failed component may exacerbate noticeably the performance of a bigger portion of the network. This results in an amplification of an isolated failure on a global scale.

In SDN this type of scenarios can be avoided or at least partially mitigated. In case of isolated failures and presence of criticalities on alternative routes flows that have not been affected by the failure and that are traversing the backup route can be dynamically

rerouted. This action allows to redistribute the load on the whole network in order to accommodate the overhead created by a faulty device. In this scenario instead of observing highly loaded portions of network with noticeable exacerbation in performance of all flows traversing it is possible to achieve only a slight increase in total network load, which depends on the criticality of the failed component. Consequently in this scenario instead of disrupting the service of a subset of serviced flows there is a slight worsening of global quality of the service. At the same time if the network must guarantee particular quality of service for a subset of flows with SDN it is still possible to guarantee it.

SDN can operate in close synergy with application layer. In hybrid networks, which present both an optical domain and the electrical one, the optical domain is capable of delivering high performance in terms of latency and bandwidth at the expense of long reconfiguration time while on the other hand the electrical domain is able to provide fast reconfiguration time at the expense of lower performance in respect to the optical one. In SDN-enabled hybrid networks the optical domain can be exploited in order to accommodate the elephant flows, i.e. flows requiring high throughput for prolonged periods of time, while the electric domain can be kept for generic flows. An example of an elephant flow for which the optical domain can be well suited is the migration of Virtual Machines inside a data centers. In optical domain it can create considerable load and exacerbate the overall performance while achieving low throughput, but since the operation is typically performed between two servers the network can be prearranged in order to accommodate this flow in optical domain thus guaranteeing higher throughput at the expense of an initial delay required for the configuration. In [37] indeed it is showed that with SDN it is possible to provide smaller reconfiguration time in respect to classical solutions when considering hybrid networks.

2.2.3 Devices' interoperability

Although throughout this work for simplicity forwarding devices will referred to as switches in SDN there is no concept of switch or router. In most commercial implementations each forwarding device is capable of performing actions based on any layer of the protocol stack allowing each device to perform simple forwarding based on MAC addresses, routing based on IP addresses, MPLS etc... This leads to more flexibility in the use of network equipment as there comes no need of employing different equipment for different tasks. Instead there is only one device for all operations. The benefits of this property are straightforward: network device manufacturers can concentrate their efforts in developing a common high-end versatile hardware while the consumers are able to obtain devices that share common specifications thus allowing to easily perform planning and business analysis.

2.2.4 Network Function Virtualization

Network Function Virtualization (NFV) is an emerging technology which has been developed concurrently with SDN but that thrives for completely different goals. The main idea behind NFV is the riddance of network middleboxes (such as firewalls, traffic classifiers, mobile network nodes etc...) and their consequent migration in virtualized environments

on commodity servers. The virtualized middlebox are then generally referred to as *Virtual Network Functions (VNF)*. Nonetheless the idea was first proposed in 2012 by representatives of European network operators [17] which must deal on a daily basis with conspicuous amount of middleboxes. The employment of NFV in highly dynamic environments such as *Wide Area Networks (WAN)* noticeably benefits from a joint use with SDN since by employing SDN it is possible to provide greater network orchestration and at the same time dynamic traffic routing towards the virtual middleboxes based on rapidly evolving network policies and resources demands.

2.3 Issues in SDN

Current SDN implementations based on OpenFlow exhibit a set a of intrinsic limitations which are a key factor that is still keeping the consumers away from the technology. As the current SDN technologies leverage on the extensive use of the controller's functionalities in all decisions affecting the control plane a question about scalability and reactiveness emerges.

2.3.1 Reliability and scalability

If the controller's intervention is required in any minor variation in the network it introduces a considerable network overhead due to constant exchange of messages between the controller and the switches. Moreover the load imposed on the controller by the constant exchange of message on the southbound interface may lead to further performance exacerbation [31]. The issue can be solved by the deployment of multiple coordinated controllers in different sites responsible for the related portions of the network. Even if the total inter-controller synchronization traffic needed to provide full coordination among the controllers grows linearly as shown in [49] the global reconfiguration speed is still penalized.

2.3.2 Reactiveness

It is also true that communication with the controller introduces additional delay in the processing of transiting flows which may eventually lead to a considerably reduced reactiveness if the controller resides in a different site. In case of failures in the infrastructure the time needed to reroute all flows from the involved faulty equipment will largely depend on the *Round Trip Time (RTT)* between the controller and the device that detected the fault leading to prolonged times of absence of service.

2.3.3 Absence of stateful operations

Given the flexibility in the definition of the rules in OpenFlow it is evident how current hardware implementation may benefit significantly from the introduction of stateful functionalities, i.e. functionalities based on persistent memory directly inside the switches. With the currently available OpenFlow hardware is it possible to implement firewalls which are able to perform content filtering based on the protocols' headers. It is instead not possible to efficiently implement stateful firewalls (i.e. firewalls based on a mutable

state related to each flow). A possible implementation of a stateful firewall would require extensive interaction with the controller in order to provide a way of storing the state related to each flow, ultimately leading to expensive packet buffering in the switches and considerable load on the controller.

Chapter 3

Stateful Dataplanes

Section 2.3 presented some of the most relevant limitations of current SDN limitations. It was addressed the need of more versatility directly inside the dataplane and more independence from the controller that still represents a large bottleneck. This Chapter contains some of the proposed solutions that are able to improve classical SDN implementations and their impact in term of enabled functionalities and features. Among the proposed systems additional effort is devoted towards the presentation of P4 language since it is then used later in Chapter 6 for the design of a functional prototype.

3.1 Stateful Dataplanes

As it was discussed in Section 2.3.3 in forwarding devices targeting standard OpenFlow there is no possibility of implementing a stateful firewall directly inside the dataplane, i.e. without any cooperation with the controller. However it is sufficient to introduce the possibility of keeping persistent state information inside the switches in order to enable this feature. Indeed in classical OpenFlow a match-action entry that matches both the intrinsic packet information and the internal state related to the flow the packet belongs to is able to fully emulate the functionalities of a stateful firewall. The firewall example can be easily extended to generic functions like billing based on transfered bits, NAT, traffic policing etc... without introducing any new hardware functionality a part from the persistent memories.

It is evident that the introduction of stateful components inside forwarding devices enables a rich set of functionalities which are able to augment the switches' functionalities and bring them closer to generic multi-purpose hardware. This provides the means of implementing a broad variety of functions directly in the dataplane without requiring any coordination with the controller, similarly to what happens in NFV for VNF that has been briefly presented in Section 2.2.4.

3.2 Stateful OpenFlow extensions

This Section provides a brief insight on current stateful extensions of OpenFlow. The choice of OpenFlow as a reference model is mainly due to its popularity in academia and its wide use in industry. Multiple vendors started to commercialize OpenFlow-enabled hardware without the initial consideration of adding support for stateful functionalities. Below are present two stateful OpenFlow extensions, namely *OpenState* and *Open Packet Processor (OPP)*.

3.2.1 OpenState

OpenState [5] is a stateful OpenFlow extension proposed in 2014 with the aim of introducing support for Finite State Machines (FSM) directly in the forwarding devices. OpenState does not involve significant modifications to the underlying hardware architecture. Instead it exploits the unused fields in the flow tables in order to store the state related to each flow. Consider the set of flows $\mathcal{F} = \{f_1, .., f_N\}$ for which a stateful rule has been defined. Each packet belonging to a flow $f_i \in \mathbf{F}$ is first processed trough a special state flow table T_p which store the information required to unequivocally identify the flow $\mathcal{H}_i = \{field_{1,i}, .., field_{M,i}\}$ alongside with its current state $s_{i,curr}$. $s_{i,curr}$ is appended to the packet as metadata and is then matched against in the subsequent processing stages. The transition to the a new state occurs in a special transitional flow table T_t which contains all possible combinations for state $s_{i,j}$ and flows f_i. In T_t a matching based on both $\mathcal{H}_i$ and $s_{i,curr}$ is performed and depending on the value of the two an action the modification of $s_{i,curr}$ in T_p is performed. This allows for the next processed packet belonging to f_i to be attributed with a different $s_{i,curr}$.

Although OpenState is advantaged by the architectural simplicity it presents considerable limitations in terms of memory requirements. For FSMs containing large number of states (e.g. states counting number of transiting packets belonging to each f_i) this approach will lead to T_t being composed by $L = \sum_i^N f_i S_i$ entries, with S_i being the maximums state-space for flow f_i.

3.2.2 Open Packet Processor

Open Packet Processor (OPP) [6] is considered to be the successor of OpenState for what regards stateful extensions of OpenFlow. OPP extends OpenState and solves the issue related to high memory requirements by introducing registers inside the architecture, thus allowing to avoid the to store all possible states for each flow in case of big state-spaces. The state transition can occur depending on the nature of the FSM: it can either occur in a similar fashion to the OpenState approach for small states or it can be performed via write operations on the registers based on the values returned from the flow tables.

The introduction of stateful components outside of the flow tables however requires modifications to the underlying hardware architecture by introducing processing blocks such as Arithmetic Logic Units (ALU), required to perform state updates in the registers.

3.3 P4

P4 [10] is a novel stateful dataplane programming language introduced in 2014. The main motivation behind the development of P4 was to provide greater flexibility for the dataplane by mitigating issues present in existing technologies. In order to achieve such level of elasticity P4 was built around three main concepts:

- **Protocol-independence**: As it was discussed in Section 2.1.2 OpenFlow is bounded in terms of development time to the ONF that releases new OpenFlow standards. Although the number of matchable fields in OpenFlow greatly increased since the release of version 1.4 it still provides support of only a subset of available protocols. P4 solves the issue by defining a programmable parser and deparsed directly inside the language specifications. This not only allows to define matching actions for all fields present in currently available protocol headers but furthermore allows to define custom protocol headers ex-novo and even extend the matching rules to the packets' payload.

- **Target-independence**: P4 leverages its functionalities on a flexible compiler which, given the specifications of the target hardware, is able to efficiently translate P4 code in hardware instructions and map it to the underlying architecture. Thanks to this strategy it is possible for the commercial vendors to adopt different architectures for their products and provide only the high level abstraction to the compiler in order to allow P4 programs to run on top of them. Furthermore, through additional effort, vendors are also able to provide architectural abstractions for products that were developed prior to the release of P4, thus allowing P4 to target those particular architectures.

- **Reconfigurability**: Exactly as in OpenFlow switches must be able to change their behavior without causing service disruption once they have been deployed in the field. P4 guarantees the same properties as OpenFlow but additionally provides all the reconfiguration possibilities deriving from the programmable parser. Indeed P4 switches are capable of not only changing the packet processing rules but also provide support for new protocols without any delay due to the standardization process as in OpenFlow while guaranteeing absence of service disruption.

Although it is not among the main motivation behind P4, the inclusion of stateful elements inside the language represents a big motivation behind its success. P4 in fact includes, apart from the simple registers, counters, meters and additional target-specific functionalities which can be easily interacted with and programmed on the fly.

Since its release in 2014 P4 has been continuously updated and underwent steady evolution while receiving extended amount of contributions from the developers' community. As mentioned in Section 1.2 in the following sections both versions of P4, namely $P4_{14}$ and $P4_{16}$, are presented. Although the provided examples make use of P_{14} the differences in respect to $P4_{16}$ are later highlighted.

3.3.1 Language specifications

Similarly to OpenFlow and OPP, P4 was build around an architecture exploiting a pipeline composed by flow tables which are referred to as *Match-Action Tables* in the context of P4. Consequently the language specifications and possibilities have been heavily influenced by this decision. The resemblance of the language specifications to the underlying architecture is a key factor that is able to guarantee fast and efficient translation of the user-defined functionalities in hardware operations. Similarly to other low-level programming languages like assembly [15] P4 operates on bit level. At the same time P4 is less flexible than other low-level languages since it allows to use only a predefined set of elementary instructions like reads/writes, sums and bitwise logic operations with the absence of more complex operations like jumps or multiplications. The choice of employing such a low level programming model without exposing complex primitives is understandable if it is considered the fact that line processing speed must be guaranteed at any time in order to achieve correct functionality of the system. Nonetheless commercial implementations of P4-enabled switches [34] are able to provide data-rates of the order of 6Tb/s which directly translates in maximum available time in order to perform packet processing in the order of nanoseconds. However, as it will be later presented in Sections 3.3.5 and 6.3 it is still possible to perform some complex operations without introducing additional critical delay if the underlying architecture exposes hardware specific primitives for those particular operations.

3.3.2 Parse-Match-Action pipeline

P4 follows a modular scheme that defines 1 or more processing stages enclosed at the two extremes by the parser and deparser. Differently from what happens in OpenFlow the flow tables' match entries depend on the parser definition instead of relying on a predefined set of matchable packet's fields. Consequently it is not completely correct to define the P4 pipeline as a simple match-action pipeline since the pipeline depends completely on the operations performed by the parser. For this reason it is preferred to address to the pipeline as a parse-match-action pipeline.

Processing Stages

Since P4 language specifications heavily depend on the target architecture, as processing stages can be found in the hardware implementation they are also found in the P4 language specifications in the form of *Control Blocks (CB)*. Each CB can enclose multiple match-action tables and a causality relationship is present among them (e.g. CB_2 cannot be accessed without first passing from CB_1). Given the causality relationship there exist some limitations on which operations can be performed at each processing stage.

In Figure 3.1 is represented an high level abstraction of a simple processing pipeline composed by two CBs, namely *ingress* and *egress*. After the parsing and initial processing in the ingress stage packets are moved in the egress queues. The presence of queues at this stage of the processing is necessary in order to guarantee scheduling and policing operations. Indeed the ingress stage can be seen as a classifier based on which decisions

different scheduling policies can be applied once the packets enter the queues. The egress stage is responsible of packet processing once they leave the queues and before they are emitted on the wire and can be used in order to perform the same manipulations as the ingress stage with the addition of the possibility of dropping packets. Since both ingress and egress CBs perform processing at line-rate this architecture does not introduce any significant overhead due to the fact that all packets are precessed through ingress CB independently from whether they are to be dropped or not.

Relation between egress ports and queues

Once the packet starts its processing in the egress CB it is not possible anymore to change the egress port and consequently to change the actual wire on which the packet will be emitted. This implies that all routing decisions must be performed in the ingress CB, as the only possibility of changing the egress port once the packet is in the egress CB is that of recirculating the packet through the whole pipeline. Although $P4_{14}$ requires mandatory support of recirculation, in $P4_{16}$ the recirculation is transformed in a target-specific functionality thus not guaranteeing support on all architectures.

Normally each queue has a 1-to-1 mapping with the corresponding egress port, however P4 does not exclude the possibility of having N-to-1 mappings thus providing support, for example, for different scheduling policies such as strict priority scheduler, Time Division Multiplexing (TDM), Round Robin (RR) etc... In addition to N-to-1 mapping it is also possible to perform 1-to-N mappings, thus allowing to easily define broadcast queues or multicast groups. Because of these possibilities P4 makes strict distinction between physical ports and the egress port which instead is referred to as *Egress Spec*. The mapping between each physical port and the corresponding *Egress Spec* is performing by the controller at runtime and it is not precluded the possibility of creation of new mappings or the modification of existing ones on the fly.

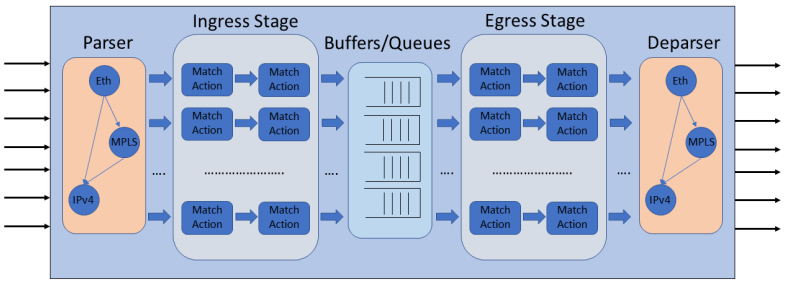

Figure 3.1. High level abstraction of a P4 processing pipeline.

Operations' execution order

It is not generally true that match-action tables are applied in a sequential order. Depending on the architecture the compiler may impose parallel processing inside the pipeline,

thus allowing a packet to be processed simultaneously by two different match-action chains as long as there is no causality relationship between them. In listing 3.1 is represented an example of a P4 code targeting the 2-stage architecture. At each stage a set of match-action tables are applied by invoking the corresponding table name and by using the keyword *apply*. The match-action tables of the ingress stage can be applied concurrently since the table *verify_checksum_table* performs the verification of the IPv4 checksum while *egress_lookup_table* performs only the lookup of the destination port based on the destination IP. Both tables do not perform any modification to the actual packet and do not depend on the actual order in which they are applied. On the other hand at the egress stage table *decrease_ttl_table* decreases the Time to Live (TTL) field in the IPv4 header while table *recompute_checksum_table* recomputes the new checksum. Since the checksum comprises also the TTL field, the concurrent execution of both tables may create race conditions in which packets are emitted with a checksum computed based on the not yet decreased TTL.

```
1 control ingress {
2   apply(verify_checksum_table);
3   apply(egress_lookup_table);
4 }
5
6 control egress{
7   apply(decrease_ttl_table);
8   apply(recompute_checksum_table);
9 }
```

Listing 3.1. Example of 2-stage architecture in $P4_{14}$

3.3.3 Programmable parser

As it was mentioned in the Section's introduction the parser is one of the key features of P4 as it is one of the many traits that clearly distinguishes P4 from other stateful dataplane programming languages. The definition of a parser in P4 follows a simple FSM model in which each stage and the corresponding advancing transition must be defined by the programmer.

In Figure 3.2 is depicted a simple parser logic that is able to correctly parse IPv4 and stacked MPLS headers. When the parser starts it is automatically moved in *parse_ethernet* state which is responsible of extracting all the fields contained in the header instance. The subsequent transitions provide a ramification able to extrapolate the supported Lyaer 3 protocols based on the value of *EtherType* field (expressed as *ethernet.etherType* in the figure) that has been extracted from the Ethernet header. For the sake of simplicity this example provides support only for two transitions from the *ethernet* state: *ethernet*→ *ipv*4 and *ethernet*→ *mpls*. When *ethernet.etherType* assumes the value of $0x800$ the FSM is moved in the state *parse_ipv4* which is responsible of extracting information related to the IPv4 protocol. Once *parse_ipv4* is completed a final transition is performed and the parser ends. On the other hand when *ethernet.etheType* assumes value $0x8847$ the parser is moved to the *parse_mpls* state. In this example it was chosen to provide support for multiple stacked MPLS instances. Figure 3.3 gives an insight on the functionality of the stacked MPLS isntances: multiple headers are inserted before the actual L3 data of which

the final one contains the *Bottom Of Stack (BOS)* field set to 1. Given this information the parser can be programmed to remain in the *parse_mpls* state until it encounters a BOS flag set to 1 after which the termination transition is performed.

In additional to parsing based on a predefined set of fields P4 supports parsing of variable-length fields which is essential in order to provide support for multiple standardized protocols. P4 parser however is limited to the parsing of only one variable-length field and neither in $P4_{14}$ nor in $P4_{16}$ the possibility of parsing more than one variable-length field is supported.

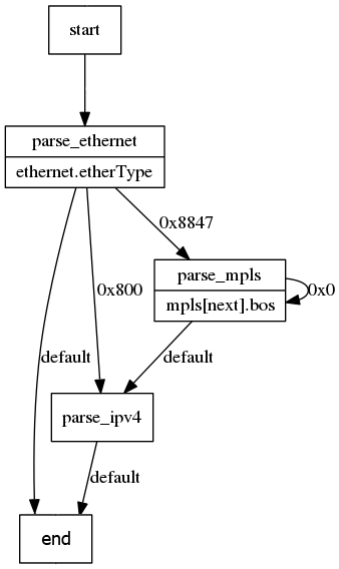

Figure 3.2. Transition diagram of a simple MPLS and IPv4 parser.

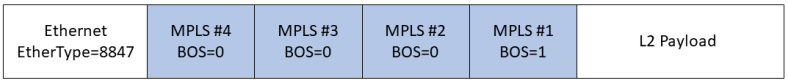

Figure 3.3. Example of stacked MPLS headers.

In listing 3.2 is represented a possible real implementation for the definition of MPLS header type. It is immediate to observe the net similarity between the employed syntax and the definition of struct data-types in C language:

- **Line 2**: Defines a new header-type *mpls_t* similarly to how new data-types are defined in C language.
- **Lines 3-7**: Define the list of fields that belong to the header type in the form of "*name : bit_length*". The order of the fields defines the way the bits will be interpreted by the parser and in order to guarantee correct interpretation it must strictly follow the real order defined by the protocol.

- **Line 10**: Instantiate a new array of headers of type *mpls_t*. The use of an array is of paramount importance if during the processing of the packet it is required to distinguish between different stacked MPLS header instances.

```
1  #define MPLS_MAX_DEPTH 10
2  header_type mpls_t {
3    fields {
4      label : 20;
5      qos : 3;
6      bos : 1;
7      ttl : 8;
8    }
9  }
10 header mpls_t mpls[MPLS_MAX_DEPTH];
```

Listing 3.2. Example of header instance definition in $P4_{14}$

In a similar way in listing 3.3 is presented the portion of the code responsible of the actual parsing of the MPLS header. The keyword *parser* in the first line is responsible of defining a parsing stage inside the FSM while the choice of the next state is represented in the return statement which selects the next stage based on the BOS field of the last parsed MPLS instance which is identified by keyword *latest*. As it was previously stated, a BOS field equals to 0 implies the presence of additional MPLS instances after the last parsed one. In that case the FSM performs a loop transition from state *parse_mpls* while the variable *next* is auto-incremented in order to place the parsed values in the corresponding array position when the defined fields are extracted (line 2).

```
1  parser parse_mpls {
2    extract(mpls[next]);
3    return select(latest.bos){
4      0 : parse_mpls;
5      default : parse_ipv4;
6    }
7  }
```

Listing 3.3. Example of nested MPLS parser in $P4_{14}$

3.3.4 Match-action tables

P4 allows to define very flexible match-action tables: it is possible to match any field provided by the parser and in addition extend the matching to architecture-specific fields such as queue occupancy, internal timestamps, errors returned by the parser etc...

It is the role of the programmer to define adequate matching rules and the corresponding actions during the development cycle. P4 provides simple constructs that allow to easily define custom tables. By recalling the 2-stage architecture example presented in Section 3.3.2 and illustrated in more details in listing 3.1 the table *decrease_ttl_table* can be easily defined as in listing 3.4. The table perform a matching on the validity of the IPv4 header (line 2-4) and supports two actions (line 5-8): *decrease_ttl* and *nop*. While the latter action does not perform any operation and is declared mainly in order to provide support for packets lacking a valid IPv4 header, the former one is a programmer defined action that performs the actual modification of the TTL field. Listing 3.5 provide the

definition of *decrease_ttl* action in which the standard primitive *modify_field* is invoked on the TTL field of the the packet.

```
table decrease_ttl_table {
  reads {
    ipv4 : valid;
  }
  actions {
    decrease_ttl;
    nop;
  }
}
```

Listing 3.4. Example of a match-action table declaration in $P4_{14}$

```
action decrease_ttl() {
  modify_field(ipv4.ttl, ipv4.ttl-1);
}
```

Listing 3.5. Example of an action declaration in $P4_{14}$

3.3.5 Target-specific objects and functions

As mentioned multiple times P4, especially $P4_{16}$ heavily relies on the underlying architecture which provides the general abstraction of the processing pipeline and additional functionalities.

Target-specific metadata

Metadata are additional information that are attached to the parsed representation of the packet while it is processed through the pipeline. Although, as discussed in mode details in Chapter 6, user-defined metadata can be employed in order to transfer information among different match-action tables or among different CBs, each architecture provides a target-specific metadata also referred to as *Intrinsic Metadata*. This kind of metadata is usually read-only and is related to the internal state of the system. This type of information varies among the architectures and it can range from the value of the internal clock to more sophisticated information such as queues' occupancy at the time of packets' arrival.

Schedulers

Although P4 provides the support of multiple queues with an N-to-1 mapping to the corresponding egress physical port, the language specifications do not make any assumption about the employed scheduling policy nor does provide direct support in the language specifications. It is up to the hardware manufacturer to implement the desired scheduling policy and to provide support for different ones. The manufacturers can then expose the possibility of choosing the scheduling policy to the controller directly on the southbound interface.

Extern functions

Externs have been introduced in $P4_{16}$ in order to compensate for the limitations and poor adaptability of P4 to different architectures. Similarly to an external library employed in most of the currently available programming languages, extern function consist in target specific functions that can be invoked inside P4 programs. The use of extern functions allows programmers to exploit target specific possibilities such as crypto-engines or simple multipliers implemented in hardware by invoking the corresponding call. Listing 3.6 provides an example of the inclusion of an extern in $P4_{16}$ able to perform register manipulation via the corresponding primitive. Similarly to the scheme adopted in Object-Oriented-Programming an extern is represented as an *Interface* which exposes the possible methods without providing any definition of the underlying functions.

3.3.6 $P4_{16}$

$P4_{16}$ [18] left early development in June 2017 with the aim of fully substituting $P4_{14}$ and consequently to become a de facto standard. $P4_{16}$ includes a complete revamp of the language syntax and functionalities by bringing the concept of an extern functions and by transforming many of the functions previously standardized in $P4_{14}$ to extern. Among these functions are all the calls to stateful primitives such as reads and writes to registers, counters, etc... By the official specifications $P4_{16}$ Core Library does not provide native support for stateful operations and instead provides functionalities similar to classical OpenFlow switches such as field matching and metadata operations. On the other hand with the release of $P4_{16}$ were introduced complex data types such as strings, integers and complex data structures such as lists and sets which allows to provide more flexibility in the definition of new functionalities. Indeed, the example in listing 3.6 depicts the construct of the register extern for a generic data-type T with the corresponding supported methods while in $P4_{14}$ registers were instantiated by defining only their length in bits.

Additional flexibility towards different architectures has been provided by defining the possibility of choosing the CBs composing the architecture since each single architecture may employ purpose-specific CBs limiting the execution of particular externs to those particular CBs.

```
1 extern Register<T> {
2   Register(bit<32> size);
3   T read(bit<32> index);
4   void write(bit<32> index, T value);
5 }
```

Listing 3.6. Example of an extern function in $P4_{16}$

3.3.7 P4 Future Evolutions

In the initial specifications P4 was proposed as a target independent programming language which leveraged on the fact that the same P4 program can run on different architectures exactly as a C program can run on different CPUs. After 3 years of development the P4 community moved further away from this initial idea and concentrated their effort

on making P4 more flexible and functional under different architectures while still guaranteeing a minimum amount of universally supported features. This basic set of features however only guarantees the capabilities of an elementary forwarding device that is only able to forward packets while employing a programmable parser. The adoption of this strategy resulted in P4 being a target independent programming language in the sense that it can be employed in order to program every P4-enabled network device but by using only the functionalities supported by the particular device.

The decision of allowing the manufacturers to introduce custom functionalities represents a very advantageous point for the industry since it allows the manufacturers to produce a wide variety of P4-enabled devices that are able to meet the specific demands of the customers while allowing to keep competitive prices by including only specific externs in different products. In this way consumers can keep low CAPEX (i.e. fixed cost due to equipment purchase) by employing high-end P4-enabled devices with extendend functionalities only in the critical parts of the network while keeping low profile devices in portions of the networks that requires only basic forwarding.

3.4 SNAP

Network programmability is an emerging concept in the field of SDN. The idea behind this concept is that since in SDN the control plane is centralized in a single entity and there is no need to individually program each device, this centralization can be exploited in order to provide even larger orchestration by embedding complex network services inside the forwarding devices. Instead of considering the network as a set of distinct devices with their related capabilities the network is instead seen as a big black-box with some aggregate capabilities. This kind of abstraction is usually referred to as *One-Big-Switch* (OBS) since from the programmer point of view the network behaves as a single switch with given properties and characteristics.

SNAP [2] is a network programming model that leverages on the OBS abstraction in order to define network-wide services and functionalities. Due to the high level of abstraction of the network resources the programmer is exempt from performing manual embedding of the programs inside the network which instead is delegated to SNAP. Moreover, thanks to the optimization techniques employed by SNAP the functionalities are embedded in the network in the optimal way, thus providing optimality in terms of performance, resource utilization and network overhead.

3.4.1 Problem description

Thanks to the introduction of stateful functionalities in the dataplanes is has been provided the possibility of implementing a broad variety of network functions related to per-packet and per-flow processing of transiting traffic. These functions may include billing, detection of network anomalies, traffic monitoring etc... and most of them require a persistent state embedded in the network which is able to keep track of the evolution of the system.

The placement of a state requires that all traffic that must be processed by the corresponding state must traverse the device storing it. This implies the convergence of multiple

flows to a single state-holding device. In small networks the placement of these states and the consequent routing may be performed via an exhaustive approach leading ultimately to an optimal configuration. In case of big networks with multiple states the optimality in the states' placement can still be achieved with an exhaustive approach but since the complexity of this approach is not polynomial the time needed to find the solution grows uncontrollably.

3.4.2 State dependencies and interoperability

Due to high hardware resources' requirements particular network policy may be split across multiple devices. Consider a network function $F_1 = (f_1, f_2)$ which is internally composed by two independent sub-functions, or states as they are referred to in SNAP f_1 and f_2. The two sub-functions can be placed in two devices sw_1 and sw_2 and all the related flows will be constrained to pass through both of them in an arbitrary order. If instead there exist a causality relation among the two sub-functions so that $f_1 \rightarrow f_2$ an additional constraint on the routing is introduced which further increases the complexity of the routing and placement problem. Now consider a second network function $F_2 = (f_2, f_3, f_4)$ which shares with F_1 a common sub-function f_2. An independent placement of F_1 and F_2 would lead to a duplicate of f_2 in the network, thus leading to useless waste of resources since the switch containing f_2 can satisfy the requirements of both of the functions.

SNAP addresses this issue by providing a framework able to decompose the original network policy in their *extended forwarding decision digram (xFDD)*. The xFDD representation provides a diagram involving the causality and interoperability relationships among the original network policies. This decomposition allows to reduce to the minimum the required number of sub-functions to embed in the networks and consequently the required amount of physical resources.

3.4.3 Optimization algorithm

Once the set of required states has been identified SNAP provides an optimal problem formulation able to find the placement of the states and the consequent routing for the related flows. The problem formulation exploits *Mixed-Integer Linear Programming (MILP)* and it is completely integrated in the SNAP framework. The problem receives as input the physical topology of the network with the related demand in terms of traffic alongside with the xFDD representation of the network policies and outputs the optimal state placement and the corresponding routing.

3.5 Virtual Network Embedding

Virtual Network Embedding (VNE) [26] is a problem with big relevance in the field of NFV. As it was briefly described in Section 2.2.4 NFV consist of virtualization of a subset of middle-boxes in the network and their consequent placement in generic commodity severs located throughout the network infrastructure. VNE consist in finding the optimal placement of chains of *Virtual Network Functions (VNF)* usually generalized as *Virtual*

Network Request (VNR), i.e sets of distinct VNF instances with eventually a causality relation among them. The network infrastructure in which the embedding is performed is provided by the *Infrastructure Provider (InP)* and is usually referred to as *Substrate Network (SN)*. VNE does not impose any constraint on the entities composing SN: it is possible to have SNs which are still prevalently composed by legacy devices or SDN-enabled switches as long as there is the presence of devices capable of providing virtualization capabilities.

An example of a VNR with causality requirements would be a chain composed by a virtual crypto-engine and a virtual firewall. An incoming encrypted packet must first traverse the crypto-engine in order to be decrypted and only after that it can traverse the firewall in order to be analyzed.

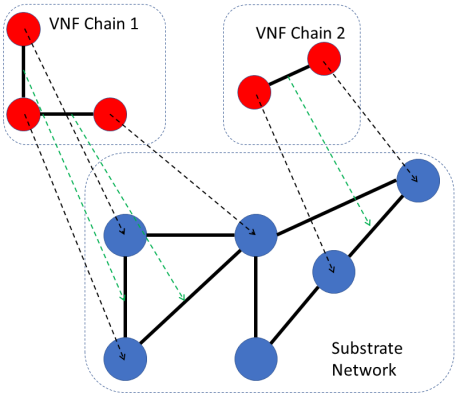

Figure 3.4. Example of single-hop VNR mapping with distinct substrate nodes.

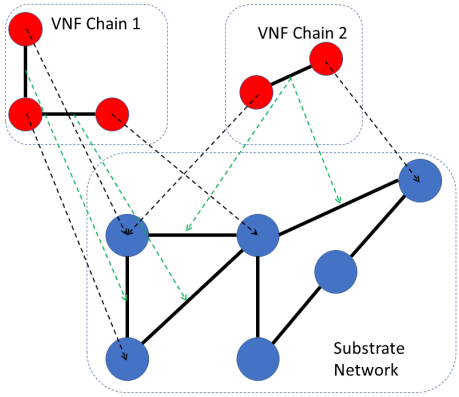

Figure 3.5. Example of multi-hop VNR mapping with common substrate nodes.

Figure 3.4 provides an illustrative example of VNR mapping. Although in this particular example each VNF instance is mapped to a distinct node of the SN, depending on the available resources multiple VNF instances can be mapped to a single substrate node. Moreover it is not mandatory to maintain one hop distance between VNFs belonging to a single chain on the SN. In fact, it is possible to achieve scenario as in Figure 3.5 in which even if VNF chain 2 is composed by 1 virtual link on the SN it traverses 2 distinct links. The two approaches present a trade-off: on one side the approach presented in Figure 3.4 requires a bigger number of dedicated servers (5 in this particular case) while the approach in Figure 3.5 allows to achieve the same behavior with less dedicated servers but at the cost of more resource requirements at each single server and more overhead in case of multi-hop placement.

3.5.1 Problem composition

The goal of VNE is related exactly to the particular problem of finding the optimal placement and routing of VNR on the SN and can be divided in 2 sub-problems:

- Placement: Single instances of VNF must be placed in commodity servers and the placement must satisfy a constraint on the available resources at each node in order to allow a feasible fitting of VNF. This problem is usually referred to as *Virtual Node Mapping (VNoM)*.
- Routing: Assuming that the VNF have been already mapped in the SN each VNF instance belonging to a distinct VNR must be connected via virtual links. This connection requires routing and allocation of physical resources on each link of the SN while still maintaining the resource availability constraint. This problem is usually referred to as *Virtual Link Mapping (VLiM)* in the field of NFV but it can also be seen as the generalization of the *Multicommodity Network Flow* problem [24].

Even if VNE can be decomposed in 2 sub-problems it is not generally true that they can be both solved independently. An optimal mapping does not guarantee optimal routing in terms of introduced overhead and vice-versa. Depending on the desired requirements and on the available node and link resources of SN the optimization problem can be tuned in such a way that allows to achieve a suitable trade-off between the two types of resources.

3.5.2 Extensions and limitations

Multiple formulations have been proposed in order to solve VNE problem and many of them include additional constraints such as delay [35], QoS and resilience guarantees [64], energy awareness [11], etc... The introduction of additional constraints significantly increases the complexity of the problem which translates directly in unacceptably big computational times required for it to be solved which becomes a big bottleneck in case of InP that provides real-time on-demand NFV services to a broad public. In order to overcome the complexity limitation multiple heuristics have been proposed which allow to find a nearly optimal solution in an admissible amount of time (the complete list of proposed heuristics can be found in [26]).

3.5.3 Similarity with SNAP

It is easy to observe how VNE problem presents great similarity with SNAP's placement and routing problem: it is in fact sufficient to represent a state as a VNF and a state dependency as a VNF chain in order to reconcile the two problem into one unique problem. Indeed SNAP can be seed as a simplified VNE formulation which concentrates more on the placement and routing while neglecting parameters such as delay, QoS, etc...

Chapter 4

Data replication

This chapter contains description of main strategies used in order to implement data replications. The advantages and drawbacks of each approach are highlighted and later recalled in Chapter 5 in order to examine the suitability of each proposed strategy for the implementation of a replication model for stateful dataplanes.

4.1 Problem description

In order to overcome the intrinsic limitations of having centralized states such as those present in SNAP it is necessary to adopt approaches based on state distribution among multiple devices in such a way to allow better load balancing and more uniform traffic patterns across the network. The are currently 2 major solutions for state distribution, namely *State Sharding* and *State Replication*.

4.1.1 State Sharding

SNAP mentions *State Sharding* as a possible solution for the large flow convergence to single switches. Sharding consists in the separation of a state variable s in N independent and non-empty states $\mathcal{S} = \{s_0, s_1, ..., s_{N-1}\}$ so that $s = \bigcup_{i=0}^{N-1} s_i$ and their consequent placement in distinct switches. Given a generic network policy $P : X \to Y$ with a discrete codomain $Y = \{0,1\}$ and a generic domain X corresponding to the domain of the state-variable s. Sharding allows to redistribute the total traffic load $\mathcal{F} = \{f_1, .., f_M\}$ across N devices but it relies on the independence of each s_i and s_j for each $i \neq j$ when the policy P is applied. For a generic flow $f_k \in \mathcal{F}$, if the satisfiability of P can be expressed as in (4.1) then sharding can be applied. If (4.1) is not satisfied and $P_{f_k}(s_j) = P_{f_k}(s_i) = 1$ with $s_i \neq s_j \in \mathcal{S}$ then the two shards s_j and s_i can evolve independently thus leading only to a partial knowledge about the state related to flow f_k.

$$P_{f_k}(s) = \sum_{i=0}^{N-1} P_{f_k}(s_i) \leq 1 \quad \forall f_k \in \mathcal{F} \tag{4.1}$$

An example of sharding application can be a distributed hash table. Consider a scenario in which it is necessary to perform traffic policing of each single flow traversing the

network. This means that there must exist a state s that for simplicity is defined as an array a of counters with $|a| = L$ where each counter $a[i]$ is incremented when a generic hash function of the information identifying the flow f is equals to i, i.e. when $h(f) = i$. The network policy P acts with a simple dropping condition when an arriving packet, belonging to a generic flow f_k, increments $a[h(f_k)]$ above a certain threshold Th as in (4.2). In standard SNAP the state s must be placed in a single switch to which all incoming traffic must converge. In enterprise network or even in medium-sized networks it is however generally not possible to sustain such a significant load on a single device. However if s is sharded into multiple states s_i so that $s_i = [a[i\frac{L}{N}], .., a[(i+1)\frac{L}{N} - 1]]$, i.e. each of them contains a subset of the original array a, it is still possible to guarantee the same functionality in respect to a centralized state. In fact it is sufficient to perform $h(f_k)$ at the ingress switches and based on the result of the hash function forward the packet to the switch that contains that chunk of the state. This example however does not hold when instead of policing each flow individually it is required to police the aggregate incoming traffic in the network as in (4.3). In this scenario it is not true anymore that all chunks of s are independent among them since the defined network policy P' operates as a reduce function on the whole s. The relationship with dependent state chunks can be generalized by introducing a generic reduce function r as in (4.4) which does not support state independence and instead requires the aggregation of all shards before the application of the policy.

Furthermore it is not generally true that each state can be sharded even in the presence of a network policy which operates on single chunks s_i independently leading to the impossibility of using sharding as a general technique for load distribution.

$$P_{f_k}(a) = \begin{cases} 1, & \text{if } a[h(f_k)] > Th \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

$$P'_{f_k}(a) = \begin{cases} 1, & \text{if } \sum_{i=0}^{L-1} a[i] > Th \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

$$P^r_{f_k}(s) = P_{f_k}(r(\mathcal{S})) \tag{4.4}$$

4.1.2 State replication

An alternative solution for load distribution among different devices is that of completely replicating each state variable. Given the state variable s, N copies of the state $\mathcal{S} = \{s_1, .., s_N\}$ are placed in N distinct switches and the load is balanced among them. This approach is then equivalent to the case of a single centralized state for what regards policy satisfiability if given two systems:

- A system C exploiting a centralized state variable s_c placed in a single device.
- A system R exploiting N state replicas $\mathcal{S} = \{s_1, .., s_N\}$ placed in N devices.

Under the assumption of same inputs to the two systems, it it true that R can completely emulate the behavior of C in terms of policy satisfaction if at each time instant

t (4.5) and consequently (4.6) are satisfied. The satisfiability of (4.5) and (4.6) provides the possibility of performing reduce operations on the state locally at each switch without the need of contacting any other replica.

$$s_c(t) = s_i(t) \ \forall s_i \in \mathcal{S} \tag{4.5}$$

$$s_i(t) = s_j(t) \ \forall s_i, s_j \in \mathcal{S} \tag{4.6}$$

C and R differ drastically in terms of total introduced network overhead: similarly to the case of sharded states the load is balanced among N devices thus under same input conditions and same state placement the overhead due to packet convergence to the states is equal or even lower (since the satisfiability of (4.1) is not required) in respect to the sharded case. It is however trivial to observe that if (4.5) is satisfied then the possibility of employing (4.4) is also guaranteed.

The presence of replicated states in a system in which only a subset of total incoming flows traverse each replica implies that all replicas evolves independently from each other. Without any information exchange mechanism it is not possible to guarantee the satisfiability of (4.5). The necessity of coordination for each replica leads to an additional overhead due to the presence of synchronization flows and, in case of complex relationships among state values, requires complex replication mechanisms.

4.2 Consistency Properties

The strict satisfiability of (4.5) is generally impossible to achieve in real systems. In order to satisfy the relation there must exist a mechanism able to continuously transmitting any update from each replica s_i to all other replicas s_j with zero delay and without any ambiguity. In real systems however any transmission is not instantaneous and is lower bounded by the propagation delay. Although the propagation delay can be kept low for small networks or for intra data center traffic, it becomes non negligible in geographically distributed networks or in case of cross data center traffic. Moreover, unless complex replication mechanisms are used, it is generally impossible to guarantee the absence of ambiguity in the reception of synchronization packets since packets can be lost or can be arbitrary delayed. In order to overcome these limitations multiple replication mechanisms have been proposed among which the most relevant are presented in this Section.

4.2.1 CAP Theorem

When dealing with replicated systems it is usually necessary to satisfy a set of properties depending on the nature of the system one is willing to design. In [13] a classification of these properties have been proposed to what is now known as *CAP Theorem*:

- ***C**onsistency (linearizability)*: If an operation P_2 is executed after the operation P_1 has been already completed the operation P_2 must see the entire system in the same

state after it was after the completion of P_1 or alternatively must return an error[1].

- ***A**vailability*: Every request received by a *non-failing* node in the system must result in a response [28].
- ***P**artition tolerance*: The system continues to work after an arbitrary number of messages are delayed or dropped by the network.

In the original presentation it was stated that it is impossible to achieve more than 2 of these properties at the same time (later proved in [28]) leading to the design of systems based on 2 of these properties, most notably:

- *CP*: Usually referred to as *Strong Consistency* is the preferred model for current implementations of highly reliable distributed databases. In CP systems in case of major service disruption it is chosen to interrupt the processing of the incoming requests in order to avoid sending responses with possibly outdated information or in order to avoid performing non linearized writes to the system.
- *CA*: Usually employed with single-site systems in order to provide consistent and available service. In case of partitions replicas of such systems can diverge and create inconsistencies which can be mitigated through expensive communication protocols which however, as shown later, leads to the disruption of availability property.
- *AP*: Usually referred to as *Eventual Consistency* is the model targeting systems in which limited periods of inconsistency can be tolerated in order to favor failure resilience and grater availability, thus better service provisioning.

4.2.2 Failure types

For the purpose of the following discussion it is necessary to define the main categories of the failure types which may occur in replicated systems:

- **Fail-Stop**: are the typical failures that occur in the case of failure of equipment for a prolonged amount of time. This fault implies that each request sent to the faulty device will not provide any response and all the request will be lost. It is however assumed that in the presence of this kind of faults all other entities belonging to the system or at least a subset of them, will be notified about the occurrence of the fault.
- **Performance-failure**: this type of failures involve a fault in which an entity is able to process request but with wrong timing, i.e. the replies are received either too late or too soon. This failure type can eventually degenerate into *Omission-failure* if the responses arrive with an infinite delay.

[1]The formal definition of linearizability is not of easy understanding and is provided in [32]. For the purpose of this work it is presented a simplified version which is tailored for the particular analyzed use-case.

- **Byzantine failure**: are the type of failures in which an entity behaves not in conformity to the standard schemes involved by the system. This may lead for example to an entity sending wrong responses to all received requests or sending conflicting information.

4.3 CAP-Consistenct replication schemes

Current implementations of consistency provisioning in terms of CAP theorem (*CAP-Consistency*) rely on the presence of one special nodes inside the replication ensemble which, depending on the adopted scheme, is usually referred to as *Coordinator*, *Master* or *Leader*. The presence of the coordinator is of paramount importance in order to provide linearizability of the operations. Without a coordinator it is in generally not possible to derive the correct order of the operations, although some uncoordinated approaches providing non strict linearizability are later presented in Section 4.6. Consider a scenario of a cluster of servers composed by 3 entities A, B, C and adopting a completely uncoordinated replication scheme. If the entity A at time t_0 commits an operation P_A to the cluster and at time $t_1 > t_0$ entity B commits an operation P_B. Under the presence of comparable propagation delays and unless unexpected behavior from the network the three entities will receive the commits in the same order in which they have been initially done. However if we consider the scenario depicted in Figure 4.1 where $t_{prop_{A\to C}} > t_1 + t_{prop_{B\to C}}$ the two updates are received not in order at entity C which depending on the relationship between P_A and P_B may lead to wrong information being stored at C. Moreover since nodes may fail and messages can be lost some of the commit operations may also be lost. In the presence of a coordinator this scenario can be mitigated by introducing additional phases during the commit of the operations.

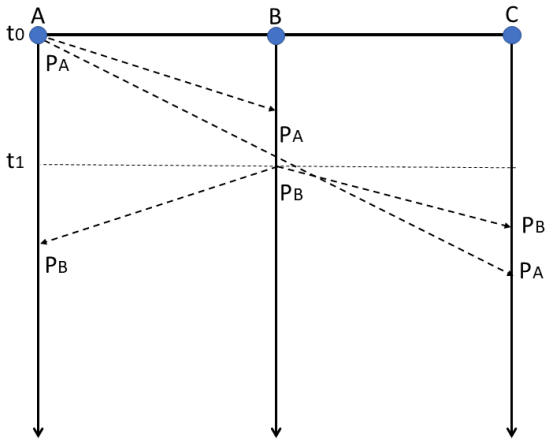

Figure 4.1. Absence of linearizability in uncoordinated systems.

4.3.1 2-Phase Commit

If instead of performing the commit directly to the whole cluster entities A and B adopt a consensus algorithm (i.e. an algorithm relying on the agreement of all entities present in the system) based on 2-phase commit (2PC) scheme [1] it is possible to provide further guarantees to the system. By employing a coordinator 2PC is able to guarantee linearizability properties and moreover provides guarantees that all replicas will receive every update.

Prepare Phase

Consider a server cluster formed by A, B, C, D where D hold the position of the coordinator while A, B, C behaves as *Cohorts*, i.e. commit executioners. As in the previous case at time t_0 A proposes P_A by sending a *Proposal message* to D. Once the proposal P_A is received by D the first phase, namely *Prepare Phase*, begins and D takes care of sending a *Prepare request* to all the cohorts and awaits their reply. Nodes pre-execute the operation P_A contained in the prepare request and may respond with a *Accept* or *Reject* where accept means that the proposal is valid and the cohort is willing to commit it while reject meaning that the proposal is not valid.

Commit Phase

In case of the reception of an unanimous accept response the coordinator moves to the second phase of the commit protocol, namely *Commit Phase* by sending a *Commit Message* to all cohorts, otherwise a *Rollback Message*. After the reception of the commit or rollback message the cohorts respectively complete the commit of the operation or discard any pre-commit made up to that point. Finally in order to avoid situations in which the *Commit/Rollback message* is lost or critically delayed all the cohorts send back an acknowledgment to the *Commit message* and the commit phase is completed. After sending the acknowledgment B can finally propose P_B. A summary of the scheme is depicted in Figure 4.2 where for simplicity all cohorts are aggregated in a unique entity.

2PC Limitations

The proposal message sent by the coordinator at the first stage serves a double purpose: it ensures first of all that all replicas are up and running thus an eventual update will propagate to all the cohorts unless unexpected faults during the second phase. Secondly once a cohort receives the proposal message it automatically locks the resources which are being affected by the operation contained in the proposal in order to avoid conflicting concurrent replications.

The block of resources represents a noticeable limitation of the 2PC which may introduce deadlocks inside the system. In fact, in scenarios in which the coordinator fails after the first phase the cohorts may lock their resources indefinitely. Moreover if one of the cohorts systematically does not reply to the *vote request* the coordinator may stuck in phase 2 awaiting the vote. Although the previous issues may be resolved by setting a timeout and by performing a modification of the ensemble (i.e. remove faulty cohorts or

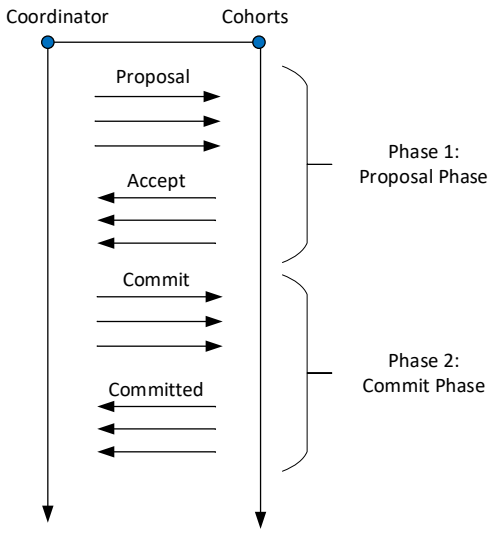

Figure 4.2. A summary of the functionality of 2PC algorithm.

substitute a faulty coordinator) in case of simultaneous failure of both coordinator and a cohort even if the coordinator is substituted the new coordinator will not be able to decide whether all entities accept the proposal or if some of them rejected it thus leaving the new coordinator unable to decide whether to send a commit or a rollback message.

4.3.2 3-Phase Commit

3-Phase Commit (3PC) [56] was designed in order to solve the issues present in 2PC by introducing an additional phase, namely *Prepare Phase*. Instead of pre-committing the operation during the *Propose Phase* the cohorts block the resources and only reply with a Accpet/Reject. After the coordinator has received an unanimous acceptance it moves to the *Prepare Phase* by sending a *Prepare message* which allows all the cohorts to pre-commit the operation. Finally the *Commit Phase* is executed exactly as in 2PC.

With such scheme there is no issues related to deadlocks as in 2PC since in case of a coordinator failure after the completion the *Vote Phase* since the cohorts' resources are freed after a timeout and they continue their normal functionality. If instead the coordinator fails after the completion of the *Prepare Phase* cohorts will perform a timeout and still perform the commit. 3PC still presents some issues in case of multiple failures which however are limited to single failed entities and do not affect the global system as in 2PC (the complete analysis is available in [3]).

4.3.3 Paxos

2PC and 3PC are limited by the fact that they are able to provide the consistency property only while neglecting the availability and partition tolerance ones. Even if 3PC has correct behavior in case of identifiable fail-stop faults, in case of a partition or omission-failures the coordinator will continuously not be able to get an unanimous acceptance since one or more of the cohorts will be replying after the expiration of the timeout. The absence of CAP-Availability is due to the fact that since to the resource are blocked during the replication, eventual new request will be dropped until the replication is complete.

In 1998 [40] Leslie Lamport proposed an replication mechanism, namely Paxos, which is able to provide CAP-Consistency and partition tolerance which is still based on a consensus algorithm but instead of employing the unanimous acceptance of all entities requires only a quorum (i.e. a part) of them.

Replication agents

While in 2PC and 3PC the entities are classified in 2 groups, namely cohorts and leaders, in Paxos they are categorized in 4 different types of agents. Each agent is not required to run on a separate device and more than one agent type can be assigned to each device up to the point in which a single device can assume all 4 roles:

- *Proposers*: Proposers are the entities reliable of proposing new updates to the quorum. Similarly to what happens in 3PC for entities proposing values to the coordinator.
- *Acceptors (Voters)*: Acceptors are special agents needed in order to guarantee fault-tolerance and are used in order to form quorums.
- *Learners*: Learners are the agents that store and execute the value proposed by the proposers and accepted by the acceptors.
- *Leader (coordinator)*: Similarly to what happens in 2PC and 3PC the leader is responsible of providing linearization of operations.

Operational phases

Due to the fact that the actual implementation of Paxos is of extreme difficulty in this section is presented a simplified version of Paxos which for the sake of simplicity requires the agreement and commit of one number v among all replicas. The algorithm is divided in Paxos instances with each instance reliable of providing one and only one consensus among all the present proposals. Each Paxos instance evolves in two phases:
Phase 1.: A proposer receives a request from outside of the system or generates a new one ex-novo. Assuming the request to be a commit request of a value v_p, in order to propagate the request towards the learners, thus attempting to commit the request, it must first receive an agreement from a quorum of acceptors. At this point the proposer assumes the role of leader and selects a sequence number n_1 and generates a proposal $P(v_p, n_p)$ and propagates it to the acceptors. Upon the reception of the proposal each acceptor replies

with an accept or a reject. The criterion on which the reply is selected strictly depends on the proposal sequence number v_p:

- If $P(v_p, n_p)$ is the overall first proposal that is received by the acceptor the acceptor or if v_p is bigger than the last sequence number stored in the acceptor's memory, i.e. (4.7) is satisfied then the acceptor replies with an accept.
- If instead (4.7) is not satisfied the acceptor replies with a reject alongside with the the proposal $P(v_{acceptor_i}, n_{acceptor_i})$ containing the last accepted sequence number and the corresponding value.

$$v_p > v_{acceptor_i} \tag{4.7}$$

This mechanism guarantees that in a given Paxos instance if $P(v_p, n_p)$ has been agreed upon by the majority quorum every higher numbered proposal, i.e. with $n'_p = n_p + 1$ has value v_p. The proof relies on the fact that if v_p has been agreed upon by a quorum Q_m composed by a majority of acceptors then if any other proposal $P(v'_p, n'_p)$ will fail. In fact if a majority quorum Q'_m replies to $P(v'_p, n'_p)$ it is guaranteed that some of them will reply with a "no" alongside with $P(v_p, n_p)$ leading to the number of acceptances less than the majority. Per contra if no acceptor replies with a reject it means that the quorum Q'_m is not composed by a majority of acceptors and again $P(v'_p, n'_p)$ fails.
Phase 2.: Assuming the success of *Phase 1* the proposer can now proceed with the commit of the value v_p by sending an *Accept request* $A(v_p, n_p)$ to the acceptors. If n_p is still not violating (4.7) then $A(v_p, n_p)$ is propagated towards the learners and the leader in the form of an *Accept*. The *Accept* message allows the learners to learn the value on which a consensus has agreed and on the other side allows the leader to terminate the Paxos instance if a majority quorum has replied with an *Accept*.

Phase 2 follows a similar pattern of *Phase 1* and the proof of its correctness is similar to the aforementioned one. The overall correctness of the algorithm in the presence of faults is subject to multiple particular cases which for brevity is not presented here but is available in [40].

Multi-Paxos

It is worth mentioning the fact that *Phase 1* can be avoided at each Paxos instance in order to reduce the amount of exchanged messages and the overall performance. In fact, in the presence of a stable, non faulty leader a single Paxos instance can be composed by multiple *Phase 2* rounds and a new instance is initialized only in case of a leader failure or in the case of a network partition leading to the leader not being able to reach a majority quorum. This kind of optimization is referred to as *Multi-Paxos* and is currently the one implemented in the majority of commercial products.

4.3.4 Raft

Paxos has been widely used in commercial products but it remained significantly infamous due to its high complexity leading to multiple articles being published in order to provide

an understandable explanation of the algorithm [16, 21, 41, 48].

In order to overcome the complexity of Paxos and provide a simple, not so easily subject to implementation errors consensus algorithm, Raft [50] has been proposed.

Replication agents

Raft algorithm simarly to Paxos is designed around the concept of a *Leader* and it comprises three types of agents:

- *Leader*: The leader handles all the requests coming from the client and are responsible of executing them in the cluster.
- *Follower*: Followers are entities which do not issue any request and instead reply to the proposals of the leader.
- *Candidate*: The candidate state is acquired by one of the followers whenever it wants to become a new leader.

Leader election

Similarly to Multi-Paxos Raft assumes the presence of one leader for arbitrarily long amount of time. At any given time a term number t is associated to the leader and its election is based upon a heartbeat mechanism: the leader periodically sends heartbeat commands to all the followers in order to maintain its authority. Once a given amount of time has passed since the last heartbeat has been received by a follower it goes in timeout and changes its state to candidate. By doing so it increments its term number to $t_{new} = t + 1$ and tries to become a new leader by issuing a *Vote* command to all other entities. Upon the reception of a *Vote* command each entity replies to a leader and once a majority quorum of replies have been received the candidate eventually becomes a leader. In order to avoid multiple leaders during each term t each entity is allowed to vote only once thus guaranteeing that there will be one and only one majority quorum.

Log replications

Raft, similarly to Paxos, assumes the existence of a sequence number n which is needed in order to provide operations' linearization and keep track of lost commits. But differently from Paxos, Raft is based on a replicated log L and n has the purpose of keeping track of the indexes of the operations present in the log. Each element in the log is unequivocally identified by n and contains the operation p alongside with the term number t of the leader that appended the entry to the log.

Assuming the presence of a single leader, upon the reception (or a generation) of a new request p to be replicated the leader exploits the heartbeat message in order to propagate the request to the followers in the format (p, t, n). Assuming that each followers' log contains $n - 1$ elements each of them upon the reception of the request appends (p, t) to their log in position $L[n]$ and send a *Confirm* message to the leader. Once the leader has received the majority quorum of *Confirm* messages it exploits again the heartbeat in order

to actually commit the request, thus allowing the followers to actually execute p. Once the entry is committed it is store in the persistent memory and is not modified under any circumstances. The presence of the term t inside the log is crucial in order to guarantee fault recovery after a partition in which multiple leaders may have taken place. Thanks to the presence of t in L it is possible, once the partition has been healed, to override uncommitted changes inside the logs of followers which were under the authority of a leader belonging to a partition not containing the majority of quorum. As in the previous case for the sake of brevity the proof of correctness in the case of multiple failures is not presented but is available in the original paper [50].

Limitations

It is essential for further discussion to underline the fact that in the case of replication of a set of processes $\mathcal{S} = \{s_1, ..., s_n\}$ in which there is a causality relation between s_i and s_{i-1} Raft and Paxos require to store a log of committed processes which requires theoretically an infinite amount of memory at each learner in order to be able to provide convergence after arbitrary long partitions. Moreover in both algorithms there is absolutely no support of byzantine faults. In fact, in Paxos a faulty proposer can proceed by sending *Accept* messages even if it did not obtain the majority quorum agreement. Similarly in Raft a faulty candidate can assume the role of a leader and completely hijack the consensus.

4.4 Eventually consistent replication schemes

The replication mechanisms discussed in Section 4.3, although present strong properties in terms of consistency and partition tolerance, neglect completely availability. In fact, consider an example of a database in which a customer performs an insert request followed immediately by a read request and another insert request. In 2PC and 3PC upon the reception of the write request resources will be blocked and both the subsequent read and insert will return an error. Similarly in Paxos and RAFT, after the insert request both systems will be waiting for a quorum which may take an arbitrarily large amount of time to be achieved leading eventually to an error during the read and insert. Although the customer is reassured by the fact that the system he is using behaves in a completely predictable way he may not be satisfied at all by the low quality of the service due to large delays.

4.4.1 Motivation behind Eventual Consistency

Eventually Consistent algorithms [59] target these particular scenario in which the availability of the data is of paramount importance in respect to their actual consistency. Given a server cluster exploiting an eventually consistent scheme that at time $t = t_0$ contains a value $v = v_0$. In case of an update operation $u : v_0 \rightarrow v_1$ at time $t = t_1 > t_0$, at time $t = t_2 > t_1$ a read operation $read(v)$ will always return a non-error message containing the value of v. However there is no guarantee on the actual value of v, it is possible in fact that instead of returning $v = v_1$, $v = v_0$ will be returned if the read operation was

processed before the new value was replicated. Eventual consistency in fact guarantees that each operation will be eventually replicated without providing any guarantee on the time needed for each replica to converge to a single value. This behavior leads to each entity belonging to the replication ensemble having its own value v_i at any given time which does not necessarily corresponds to the most "fresh" value.

The choice of eventual consistency over strong consistency is a design choice which is usually characterized by the necessity of having big scalability and by the presence of data which may tolerate inconsistency periods. In systems exploiting strongly consistent algorithms such as the ones discussed in Section 4.3 each operation requires the participation of the whole replication ensemble (or the majority of it) for it to become committed. In large systems this translates directly in unsustainable network load and overall performance degradation due to high resource utilization. Due to the fact that eventually consistent schemes do not require immediate convergence, the replication mechanism can exploit *Gossip Protocols* [22] (also known in the literature as *Epidemic Protocols*), i.e. protocols in which the information propagates in a probabilistic way among the entities without necessarily involving all of them. The use of such algorithms can lead to lower network overhead and overall better performances in terms of network utilization.

One of the issues that has already been discussed in 4.3 is the necessity to satisfy the causality relationships between multiple updates. Eventually consistent replication algorithms do not provide any strict guarantee on the order in which the updates will be received. It follows that given 2 requests s_1 and s_2 with a causal relationship $s_1 \rightarrow s_2$ it may happen that at some point in the replication ensemble some of the replicas will perform a commit of s_2 followed by s_1 as shown in Figure 4.1 potentially leading to ambiguous results. In order to overcome this limitation multiple mechanisms have been proposed with some of them cited hereafter.

4.4.2 Causal consistency

Causal consistency [43,63] extends eventual consistency by introducing causal relationship among the operations. This relationship is computed at the client side following multiple update requests towards the server and is attached to each subsequent request. By recalling the the example with 2 updates s_1 and s_2 with a causality relationship $s_1 \rightarrow s_2$, after sending the update request for s_1 the update request s_2 is modified by including the summary of all the preceding dependencies, thus becoming $s_2' = (s_2, h([s_1]))$. Upon the reception of s_2' the entities will check whether the summary of locally committed operation is equal to $h([s_1])$ and in the positive case will proceed by committing s_2', otherwise they will store s_2' until s_1 is eventually received and committed.

4.4.3 Anti-entropy

There may occur scenarios in which some updates may be lost or delayed for indefinitely long amount of time a scenario such as in the case of the loss of s_1 in the previous example. In that case if the client keeps on submitting request in the form of a chain of causality relationship, i.e. $s_i' = (s_i, h[s1, .., s_{i-1}])$ what may happen is that at some point the server will not be able to store anymore any new uncommitted update request due to the amount

of memory dedicated for pending updates. The exist mechanisms able to mitigate this issue and eventually heal any stale inconsistency by performing periodic comparison among state of the replicas. These mechanisms are usually referred to as *Anti-entropy*. The main idea behind the family of these algorithms is to periodically compare the replication state of 2 or more entities chosen at random and try to resolve eventual conflicts that may have been created. The actual mechanism of resolving the conflicts varies depending on the actual data type: in case of a grow-only counter for example the conflict resolution can be simply performed by picking the maximum value among those that are compared, while for more complex data types different rules may apply. It is worth noting that anti-entropy does not guarantee in any way the overall freshness of the data after the conflicts between two or more replicas have been solved but guarantees that after the conflict resolution the replicas will for sure contain a fresher value or in the case of absence of conflicts will not perform any modification.

4.4.4 Conflict-free Replicated Data Types

The example of conflict resolution in the case of a grow-only counter is part of a families of data type known as *Conflict-free Replicated Data Types (CRDT)* which are proven in [54] to converge to a single value independently from the amount of isolated partitions, as long as all replicas satisfy the connectivity property. The goal of CRDT is not to define an eventually consistent replication scheme allowing to provide causality relationships but instead it is to define suitable data types and merge functions which allow to avoid conflicts from the very beginning. This is achieved by defining data types which have as a sufficient condition the support of merge functions satisfying 3 properties: *Commutativity*, *Associativity* and *Idempotence*.

Definition 1 ***Idempotence****: Given a binary operator f it is said to be idempotent if* $f(f(x, y)) = f(x, y)$

From definition 1 it follows that each replica does not have to keep track of the previously committed requests, in fact if any replica receives the same commit request more than one time the final outcome will be the same as if it was received only once. Moreover the commutative and associative properties guarantee that each commit request can be received out of order leading always to the same result in case of packet reordering by the network.

Merge function

There is a limited number of known CRDTs. In particular the grow-only counter previously discussed is one of them. Consider x_i to be the counter value at node i of a replication ensemble composed by N nodes and $f(x, y) = max(x, y)$. It is true that $f(x, y)$ is idempotent since $max(max(x, y)) = max(x, y)$ and it is also trivial to show that $f(x, y)$ is commutative and associative. Upon the reception at node j of the counter value x_i the new value for the local counter at node j is evaluated as $x'_j = max(x_i, x_j)$ if we consider $x_k \leq x_{max}\ \forall k = 1..N$ it can be shown that after a finite amount of time and in the absence

of any new update all x_k will converge to x_{max}. At this point is possible to extend the definition of x to a generic positive-negative counter by defining two separate grow-only counters x' and x''. Each counter increase operation will increment x' while each decrease will increment x''. In order to obtain the actual counter value during the read operation the node need to perform a local merge by evaluating $x = x' - x''$ leading to the actual counter value. Among other known CRDTs there are two-phase sets, sequence lists and graphs [53] and similar merge functions are adopted for each of them.

CRDT limitations

One of the major limitations of CRDTs is that in order to provide any guarantees on partition tolerance and fast convergence each node requires $N \cdot L$ memory with L being the size in bits of replicated state variable. This requirement results in poor scalability and makes this approach poorly suitable in the case of dedicated hardware with limited amount of memory such as stateful switches. There exist however CRDTs, namely operation based CRDTs (CmCRDTs), which allow to keep the memory occupation close to L but at the cost of sacrificing partition tolerance and resilience to packets' replay. In Section 5.3.3 is presented a possible solution for the decrease of memory overhead by exploiting topological properties of the network.

4.5 CAP Theorem criticisms

CAP Theorem provides very strict definitions of *Consistency*, *Availability* and *Partition Tolerance*. The entire theorem and consequently the formal proof is based on the strict definition of these 3 properties. This aspect has risen multiple critiques [12, 29, 39] due to unsuitability of the CAP properties in real life scenarios.

Following the definition of *Availability* it is easy to notice how its application in realty does not provide what is generally defined availability. The definition in fact states that a request will eventually generate a non-error response without however defining any upper bound on the latency of the response. A web server may be employing a scheme that provides *CAP-Availability* but if the response has a latency in the order of minutes or even seconds any request sent by the clients will eventually timeout leading to the apparent absence of service.

Similarly the definition of *Strong Consistency*, although it precludes theoretically the possibility of having inconsistent reads, in real life applications is not of high suitability. If a network partition occurs a *Strongly consistent* scheme should exclude any operation from the entities belonging to the minority of the partition Q_m, thus return an error message to the clients. In real life however it is inconceivable to think that due to an internal partition the service must stop completely for a certain subset of clients. In fact, what may happen is that requests may be simply redirect to the partition containing the majority Q_M of replicas or alternatively the clients may continue to perform reads from Q_m while excluding any write operations until the partition is healed, similarly to what have been proposed for Apache ZooKeeper [33]. A system that behaves in this way is not CAP-Available neither CAP-Consistent but it presents a trade-off between the two

properties and usually said to employ *Tunable Consistency* [61].

It is easy to observe how it is possible to build real life applications which do not follow in any way the CAP definition of consistency and availability but are still able to provide, up to a certain degree, reliable and highly available service.

4.6 State-of-art and commercial implementations

Due to the rapid increase in the quality of service demand in the field of telecommunication multiple companies are adopting the approach of moving the service closer to the client. This consist in the creation of multiple data centers geographically sparse in order to reduce the latency and the access delay to the desired content. However with the increase in the number of secondary data centers increases also the cross data center network load due to the necessity of keeping continuous consistency among the data centers. Because of this aspect, scalability is starting to become a non negligible factor when a new replication system is designed. In particular multiple solutions have been proposed in order to decrease the impact of having extensive communication due to strongly consistent mechanisms. Among them there are solutions, such as Cassandra [60], trying to reduce to the minimum the use of algorithms based on 2PC by using eventual consistency or tunable consistency for non critical data. Other solutions instead exploit dedicated hardware such as GPS and atomic clocks in order to provide conflict resolution by employing highly accurate timestamps such as Google's Spanner [19]. On the other hand there exist systems based completely on eventual consistency, in particular Akka [9] which is entirely based on CRDTs.

Even if the inclusion of *Partition Tolerance* is seen as a mandatory move when developing a replication scheme, there still exist systems adopting CA schemes. In fact, although partitions may occur, their frequency can be significantly reduced by designing highly redundant systems leading to significant reliability [4]. Systems such as Dynamo [59] can fully exploit CA properties while providing periodic version reconciliation in order to guarantee convergence in case of partitions.

Chapter 5

State replication for stateful dataplanes

As mentioned in 3.4 SNAP allows to exploit the OBS abstraction by constructing xFDDs and by performing optimal placement (also referred to as *Mapping*) of states based on the available resources and traffic patterns in order to minimize the total introduced data overhead. However the fact that for each placed state there exist only one switch responsible for the state-information presents a large bottleneck since in the case of states with a global scope (e.g. total incoming packets in the network) the majority of flows must converge to that single switch thus creating large congestion and performance degradation.

This chapter contains the formalization of the problem related to state replication and provides design guidelines for the case of state replication for stateful dataplanes.

5.1 Problem formalization

In Chapter 4 it was mentioned that thanks to state replication and state sharding it is possible to extend a network policy to a broader set of flows which is not bottlenecked by the maximum throughput capacity of a single device. It is possible to observe that with the use of schemes based on replicated states it is also possible to substantially reduce the introduced overhead due to the convergence of packets to a single centralized device. Similarly to what happens in SNAP in order to provide maximum possible reduction in overhead the placement of the states must be performed in an optimal way by keeping into account not only data flows but also the synchronization flows.

In this section is proposed an *Integer Linear Programming (ILP)* formulation of the problem capable of finding the optimal solution for replicated state placement. Due to the complexity of the problem Section 5.1.3 provides some insights on the possible alternatives which are able to solve he problem by reducing drastically its complexity, but with the cost of obtaining sub-optimal results. For the sake of simplicity some of the equations belonging to the ILP formulation are not linearized and left purposefully in the product form. However all of them can be easily linearized with the available techniques [30].

5.1.1 VNE for state replication

The formulation of SNAP for what regards placement and routing problem can be generalized as a particular case of the VNE problem. In fact, xFDDs defined in SNAP can be seen as VNR with causality constrains and interoperability properties. Consequently the optimal state placement and flow routing can be achieved by exploiting a subset of constrains defined in VNE.

Although some works as [47] introduce multiple copies of the same state their sole purpose is that of allowing to perform load balancing among different branches of the same VNR. Consequently it is possible to classify them under the category of state sharding. The extension of SNAP with the introduction of replicated states can also be categorized as a VNE problem with the presence of synchronization flows between the replicas. This type of behavior can be easily achieved without major modifications to the currently available formulations by considering a set of replicated VNRs $\mathcal{V} = \{v_1, .., v_n\}$ each of them composed by $\mathcal{F} = \{f_1, .., f_m\}$ VNFs and additionally a set of synchronization chains $\mathcal{S} = \{s_1, .., s_m\}$ each composed by $\mathcal{T} = \{t_1, ..t_n\}$ VNFs with a full mesh interconnection. Assuming $x_{i,j}$ to be the VNF i belonging to VNR j for $x \in \{f, t\}$. During the problem formulation it is sufficient to impose co-location of the two chains on the substrate node so that (5.1) is satisfied, as shown in Figure 5.1. This kind of approach forces each VNF belonging to a distinct replicated branch to be overlapped by a synchronization VNR, thus creating a full-mesh virtual link interconnection among each VNF while guaranteeing that the capacity constraint on the SN is not violated. However in order to fully exploit the presence of state replication additional modifications to the formulation must be made. In particular, flows must acquire the freedom of choosing a different VNR after traversing each f_i. This allows the flows to be processed by different replicated VNR at each stage depending on their final destination and leads to further reduction in overhead. Without this type of degree of freedom this solution degenerates to simple load balancing as proposed in [47].

$$position(f_{i,j}) = position(t_{j,i}) \ \forall i, j \ \ i = 1..m, \ \ j = 1..n \tag{5.1}$$

5.1.2 ILP formulation

Both in NFV and SDN fields no work have been done on the analysis and definition of systems with replicated virtual chains that at the same time guarantees the interchangeability of VNRs among each-other. In order to support this type of system the formulation of the problem must include the constraints related to VLiM, VnoM and moreover the constrains related to the definition and routing of the synchronization traffic among the state replicas which can be seen as an extension of VLiM problem.

Input and output variables

Table 5.1 contains the set of input variables needed in order to formalize the problem. The majority of the inputs are predominantly related to the underlying network topology, the nature of the flows traversing the system and consequently tthe policy required for their

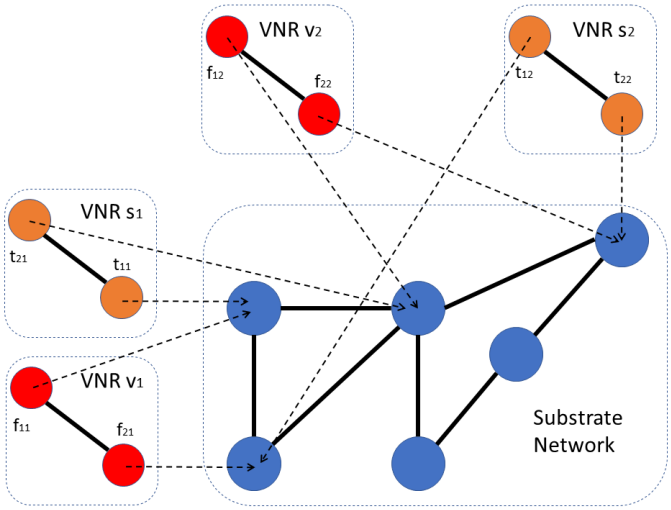

Figure 5.1. State replication in NFV via synchronization VNR mapping. Link mapping not included in order to maintain clarity.

Table 5.1. Input variables for the solver.

Variable	Description	Range
Λ	set of all the flows	
f_s	source node for flow f	$1 \dots N$
f_d	destination node for flow f	$1 \dots N$
n, i, j	generic node in the network	$1 \dots N$
$E_i(n)$	set of edges entering node n	$\subseteq E$
$E_o(n)$	set of edges leaving node n	$\subseteq E$
λ_f	traffic demand for flow f	> 0
c_e	capacity of edge e	> 0
S	set of all state variables	-
S_f	set of state variables needed for flow f	$\subseteq S$
H_{uv}	set of sequence of state variable copies needed for flow uv where $h \in H_{uv}$	-
C_s	number of copies of state variable s where $f \in C_s$	≥ 1
c	c-th copy of state variable s	$1, \dots, C_s$

processing. On the other hand table 5.2 contains the set of output variables which specify the position of the state variables and the information related to the routing of both data and synchronization flows.

Table 5.2. Output variables of the solver

Variable	Description	Range
R_{fhe}	flow $f \in \mathcal{F}$ along $h \in H_f$ on edge $e \in E$	Binary
$\hat{R}_{scge}$	state synchronization traffic from copy c to copy g of state variable $s \in S$ on edge $e \in E$	Binary
P_{scn}	1 if copy c of state $s \in S$ is stored in node $n \in V$, 0 otherwise	Binary
P_{fce}	1 if flow $f \in \mathcal{F}$ on edge $e \in E$ has passed $c \in h \in C_s$	$[0, R_{fhe}]$
X_{hf}	1 if flow f is assigned $h \in H_f$	Binary

Objective function

The definition of the objective function is subject to the requirements of the InP. Since among other things, as described at the beginning of the chapter, the goal of state replication is to overcome the intrinsic limitations due to increased congestion on links belonging to state-holding switches the objective function can aim at minimizing the maximum congestion on each link as shown in (5.2) or similarly to what has been proposed in SNAP it can minimize the total data and synchronization traffic in the network as shown in (5.3). In both equation the first term corresponds to the amount of data traffic introduced by the optimization procedure while the second term represents the amount of synchronization traffic.

$$\min \max_{e \in E} \left(\sum_{f \in \mathcal{F}} \sum_{h \in H_f} R_{fhe} \lambda_f + \sum_{s \in S} \sum_{c \leq C_s} \sum_{\substack{g \leq C_s \\ g \neq c}} \hat{R}_{scge} \hat{\lambda_s} \right) \tag{5.2}$$

$$\min \left(\sum_{f \in \mathcal{F}} \sum_{h \in H_f} \sum_{e \in E} R_{fhe} \lambda_f + \sum_{s \in S} \sum_{c \leq C_s} \sum_{\substack{g \leq C_s \\ g \neq c}} \hat{R}_{scge} \hat{\lambda_s} \right) \tag{5.3}$$

Data routing constraints

The data routing constraint follows closely the formulation of the Multi-Commodity Flow problem with the sole modification due to the presence of state sequence requirements for each flow $h \in H_f$. In order to choose the right sequence variable an auxiliary variable X_{fh} has been introduced which assumes values in {0,1} and assumes a value of 1 if the sequence $h \in H_f$ is assigned to flow $f \in F$. The variable is defined in equation (5.4) which allows to set $h \in H_f$ directly at the source node. Additionally constraint (5.5) is required in order to guarantee that there is only one chosen sequence per each flow. A similar constraint, represented in equation (5.6), is defined for the destination node in order to guarantee the flow conservation at the endpoint. Additionally the flow conservation constraint at each

intermediate node is defined in (5.7)

$$X_{fh} = \sum_{e \in E_O(f_s)} R_{fhe} - \sum_{e \in E_I(f_s)} R_{fhe} \tag{5.4}$$

$$\sum_{h \in H_f} X_{fh} = 1 \tag{5.5}$$

$$\sum_{h \in H_f} \left(\sum_{e \in E_I(f_d)} R_{fhe} - \sum_{e \in E_O(f_d)} R_{fhe} \right) = 1 \tag{5.6}$$

$$\sum_{e \in E_I(n)} R_{fhe} = \sum_{e \in E_O(n)} R_{fhe} \; \forall n \in V \setminus \{f_s, f_d\} \tag{5.7}$$

The final constraint is the one related to the link capacity c_e and is needed in order to avoid capacity violation at each link. The constrain depicted in equation (5.8) considers the sum of both data and synchronization traffic on each link.

$$\sum_{f \in \mathcal{F}} \sum_{h \in H_f} R_{fhe} \lambda_f + \sum_{s \in S} \sum_{c \leq C_s} \sum_{g \leq C_s g \neq c} \hat{R}_{scge} \hat{\lambda}_s \leq c_e \tag{5.8}$$

Placement constrains

Equation (5.9) is a modification of the constraint initially used in SNAP originally it was used in order to guarantee the uniqueness of each state inside the network. In the case with state replication the constraint guarantees that no more than 1 copy of each state is placed at each node.

$$\sum_{n \in V} P_{scn} = 1 \quad \forall c \leq C_s, \; \forall s \in S \tag{5.9}$$

Each flow is forced to pass through the states present in the particular sequence h chosen in (5.4) as shown in (5.10). The flow traverses the node n only if the sequence h is adopted by the flow (i.e. if $X_{fh} = 1$) and if that particular node stores a copy h_s of the state s.

$$\begin{gathered} \sum_{e \in E_I(n)} R_{fhe} \geq P_{sh_s n} + X_{fh} - 1 \\ \forall n \in V \setminus \{f_s, f_d\}, \forall f \in \mathcal{F}, \forall h \in H_f, \forall s \in S_f \end{gathered} \tag{5.10}$$

Additional auxiliary variable $T_{fsh_s e}$ is defined in (5.11) in order to keep track of the states already traversed by a flow. For a flow f traversing copy h_s of state s $T_{fsh_s e} = 0$ for all edges along the path before entering the node with copy h_s of s. Similarly $T_{fsh_s e} = 1$ for all edges in the path after h_s.

$$T_{fsh_s e} \leq R_{fhe} \; \forall f \in \mathcal{F}, \; \forall s \in S_f, \; \forall h \in H_f, \; \forall e \in E \tag{5.11}$$

In order to model the transition of $T_{fsh_s e}$ between 0 and 1 whenever a flow leaves a node containing a copy of a state h_s equation (5.12) must hold. The equation guarantees

that when the node n stores a copy of a state h_s and f requires that particular state (i.e. $P_{sh_sn}X_{fh} = 1$) only then the net flow balance at node n is equal to 1.

$$\begin{gathered} P_{sh_sn}X_{fh} = \sum_{e \in E_O(n)} T_{fsh_se} - \sum_{e \in E_I(n)} T_{fsh_se} \\ \forall f \in \mathcal{F}, \forall s \in S_f, \forall h \in H_f, \forall e \in E, \forall n \in V \setminus \{f_s, f_d\} \end{gathered} \tag{5.12}$$

At this point it is immediate to define the condition forcing all flows to traverse all of the required states before reaching the destination node. In fact il is sufficient to impose $T_{fsh_se} = 1$ for $e \in E_I(f_d)$ which is defined in its complete form in (5.13)

$$\begin{gathered} T_{sh_sf_d}X_{fh} + \sum_{e \in E_I(f_d)} T_{fsh_se} = X_{fh} \\ \forall f \in \mathcal{F}, \forall s \in S_f, \forall h \in H_f \end{gathered} \tag{5.13}$$

Equation (5.13) guarantees that each flow traverses each state assigned to it. However the equation does not guarantee any causality property thus not allowing to define any strict order under which the states are traversed. Equation (5.14) allows to guarantee causality between states by forcing flows to cross $h_s \in H_f$ before traversing $h_{s'} \in H_f$. In fact, if flow f has been assigned state sequence h (i.e. $x_{fh} = 1$), or copy $h_{s'}$ is not present at node n then T_{fsh_se} is forced to 1 before entering node n, therefore the flow must have traversed h_s before entering $h_{s'}$ thus leading to correct causality relationships.

$$\begin{gathered} P_{sh_sn} + \sum_{e \in E_I(n)} T_{fsh_se} \geq P_{s'h_{s'}n} + X_{fh} - 1 \\ \forall f \in \mathcal{F}, \forall s, s' \in S_f, \forall h \in H_f, e \in E_I(n) \end{gathered} \tag{5.14}$$

Finally it must be ensured that once the flow f traverses one of the edges of a node n storing a copy h'_s of the state variable s (i.e. $T_{fsh_{s'}e} = 1$) then it must have already crossed also h_s, thus ensuring $T_{fsh_se} = 1$. This behavior is guaranteed by equation (5.15).

$$\begin{gathered} T_{fsh_se} \geq T_{fs'h_{s'}e} \\ \forall f \in \mathcal{F}, \forall s, s' \in S_f, \forall h \in H_f, e \in E_I(n) \end{gathered} \tag{5.15}$$

State synchronization constraints

The definition of synchronization traffic is performed concurrently with the definition of data traffic and exploits the multi-commodity flow formulation in order to define flows among state replicas. The synchronization traffic is expressed by a decisional variable $\hat{R}_{scge}$ $\forall s \in S$, $\forall c \neq g \leq C_s$ which defines the existence of a synchronization flow for state-replicas $c \neq g \leq C_s$ of state $s \in S$ for each edge $e \in E$ belonging to the topology. The formulation makes use of products between decisional variables which are purposefully left not linearized in order to facilitate the understanding of the formulation.

Equation (5.16) defines an auxiliary variable P_s which represents the number of state-replicas placed for each state $s \in S$. It is defined mainly for clearness and can be avoided

during the implementation by substituting P_s with the unfolded notation in each subsequent equation. Equations (5.17),(5.18),(5.19) provide the formulation for the multi-commodity flow problem for the synchronization traffic: constrains (5.17) and (5.18) provide the flow conservation and the source and at the destination while (5.19) provides the flow conservation at intermediate nodes. In particular (5.17), for each node behaving as a state-replica forces the presence of at least one synchronization flow on all the outgoing links. The inequality relationship is necessary on one hand in order to guarantee the possibility for the state-replicas to behave as transient nodes and on the other hand to allow nodes not containing any state-replica to forward the flows. Similar behavior happens in equation (5.18) that accounts for the incoming traffic. Finally equation (5.19) provides flow conservation for transient nodes. For nodes not containing a state-replica the incoming synchronization flows are forced to leave the node without consuming any of them while on the contrary case for nodes storing a state-replica the outgoing flows are forced to be smaller or equal to the incoming flows in order to provide the possibility of exploiting the nodes as transient nodes.

$$P_s = \sum_{g \neq c \leq C_s} \sum_{n \in V} P_{sgn} \quad \forall s \in S \tag{5.16}$$

$$\sum_{e \in E_O(n)} \hat{R}_{sgce} \geq P_{sgn} \quad \forall c, g \; c \neq g \in C_s, \; \forall s \in S, \; \forall n \in V \tag{5.17}$$

$$\sum_{e \in E_I(n)} \hat{R}_{scge} \geq P_{sgn} \quad \forall c, g \; c \neq g \in C_s, \; \forall s \in S, \; \forall n \in V \tag{5.18}$$

$$\sum_{g \neq c \leq C_s} \sum_{e \in E_O(n)} \hat{R}_{scge} - \sum_{g \neq c \leq C_s} \sum_{e \in E_I(n)} \hat{R}_{scge} \leq P_{scn}(P_s - 1) \quad \forall n \in V, \; \forall c \leq C_s, \; \forall s \in S \tag{5.19}$$

In real-life replication schemes based on non-periodic updates (such as 3PC or Paxos) the frequency in the transmission of synchronization packets is directly related to the frequency of incoming requests (update requests). It follows that in order to achieve a more realistic formulation the objective function (5.3) can be modified as in (5.20). The new objective function forces a relation of proportionality among the synchronization traffic and the number of data flows traversing the state-replica. Since the data flows can be exploited in order to derive an estimate for the frequency of incoming update requests this approach provides a more accurate estimate for the size of the synchronization traffic.

$$\min \quad \sum_{f \in \mathcal{F}} \sum_{h \in H_f} \sum_{e \in E} R_{fhe} \lambda_f + \sum_{s \in S} \hat{\lambda}_s \sum_{c \leq C_s} \sum_{f \in \mathcal{F}} \sum_{e' \in E_I(n)} \sum_{h \in H_f} R_{fhe'} P_{scn} \sum_{g \neq c \leq C_s} \sum_{e \in E} \hat{R}_{scge} \tag{5.20}$$

As a final remark it is worth mentioning that in the case of an objective function as in (5.3) there is no need to model the state synchronization traffic as described above. Since

the routing of synchronization flows is not constrained in any way other than with the source and destination node, it is sufficient to define an a priori matrix $P = [p_{ij}]$ for $i, j = 1..|V|$ where p_{ij} represents the shortest path in terms of number of hops from node i to node j. Since the shortest paths guarantee always the minimum amount of introduced network overhead it is possible to avoid the definition of routing constraints for the synchronization traffic. It is true that in replication schemes based on periodic updates (such as RAFT, Anti-Entropy,etc. . .) the contribution due to the synchronization traffic never changes and the size of the synchronization flows remains constant independently from the amount of data flows. In such scenarios the complexity of the problem is highly reduced since the amount of overhead due to synchronization flows depends only on the placement of the state-replicas. Same concept applies in the case of non-periodic replication schemes whose update frequency depends on the amount of flows traversing each state-replica, although in this case the synchronization overhead depends both on the placement of state-replicas and on the routing of data-flows.

5.1.3 Heuristic for state replication

Due to the elevated number of constrains and consequently high complexity of the formulation an heuristic to the original problem have been proposed. Although the heuristic is not presented in deep details in this work, it is available in the original paper [7]. The algorithm exploits a 3-phase mechanism:

- *Phase 1*: The network graph G is partitioned into C_s clusters in such a way to minimize the maximum distance among the elements within a cluster. This type of partition then allows to distribute the replicas across the whole network in a balanced way, exploiting the spatial diversity offered by each cluster.

- *Phase 2*: In each cluster a replica placement is performed by exploiting the nodes with the highest *Betweenness Centrality* metric in order to minimize the mean distance to all other nodes in the cluster and thus minimize the total introduced data traffic.

- *Phase 3*: The position of each copy is perturbed at random using a local search to improve the solution with respect to one obtained in the previous two phases.

Although there exist multiple types of centrality metrics in this particular case Betweenness Centrality lead to best performance in terms of error ratio in respect to the ILP solution while still maintaining low computational complexity. Betweenness Centrality centrality is a local centrality metric defined for each node n as the number of shortest paths passing through n $\sigma(n)_{ij}$ between couple of nodes (i, j) $i, j \neq n$. This value is then normalized as shown in (5.21) by the total number of shortest paths between any pair of nodes in order to keep into account the presence of multiple shortest path of equal length.

$$C(n) = \sum_{i,j \neq n} \frac{\sigma(n)_{ij}}{\sigma_{ij}} \tag{5.21}$$

Since the algorithm does not take into account in any way the overhead introduced by the synchronization traffic *Phase 3* is introduced in order to compensate for this limitation. The position of each replica is perturbed a limited amount of times (the actual number of perturbations can be tuned in order to find a suitable trade-off between computational time and the desired approximation ratio) by moving each replica to one of the neighboring nodes. The obtained solutions are then confronted with the previous ones and the best one is chosen as the final solution.

5.1.4 Required consistency level for stateful dataplanes

In the scenario of stateful dataplanes with state replication it is crucial to provide partition tolerance in the CAP sense since packet losses, corruption and outages may occur. At this point considering CAP theorem a choice between CAP-Consistency and CAP-Availability must be done.

Objective

The introduction of stateful functionalities inside dataplanes has as the main goals a) the reduction in the load on the controller and b) the increase of reactiveness during the occurrences of critical events. Both goals are achieved thanks to the offloading of the related information from the controller directly in the forwarding device with the consequent reduction in communication overhead between the forwarding devices and the controller. When performing the choice of the required consistency level for the case of replicated states it must be taken into consideration that preferably both a) and b) should be satisfied.

Strong Consistency for stateful dataplanes

In case of consistency schemes employed directly in the dataplane (i.e. without any intervention from the controller) it is trivial to notice that a) is satisfied. However if the replication mechanism employs *Strong Consistency* the satisfiability of b) is not guaranteed due to the low provided availability. Consider a network composed by $\mathcal{F} = \{n_1, .., n_N\}$ forwarding devices each belonging to the replication ensemble of state s and for simplicity lets assume $s = \{s_1, s_2, s_3\}$ to be a finite state machine (FSM) with 3 states as shown in Figure 5.1.4. Consider state s_3 to be a critical state after entering which a network wide modification to the forwarding plane (e.g. steering of all incoming traffic towards a particular device) is required. If at time $t = t_0$ the system is in state s_1 and event e_1 occurs at the forwarding device n^* it will start the replication of the change of state $s_1 \rightarrow s_2$ to all other switches belonging to the replication ensemble. While n^* awaits for for the unanimous agreement at time $t_1 > t_0$ the event e_2 occurs which should lead the system to a critical state s_3. At this point n^* cannot apply the transition of state $s_2 \rightarrow s_3$ since the resources are still blocked awaiting for the commit of $s_1 \rightarrow s_2$ leaving the network in state s_1 even if it must respond to the criticality caused by e_2. Depending on the latency between each forwarding device and between the forwarding devices and the controller it

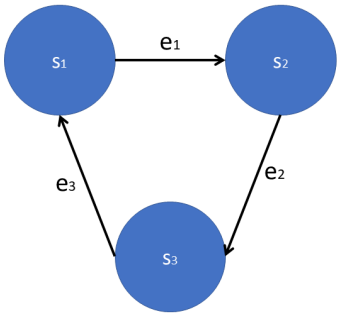

Figure 5.2. FSM with a criticality condition in state s_3

is not guaranteed that b) will be satisfied. Given the average time to commit an operation $\bar{t}_{tc}$ and the average round trip time delay between the controller and the forwarding devices $\bar{t}_{RTT_{sc}}$ if (5.22) is satisfied the introduction of state replication will lead to lower reactiveness in respect to the classical scheme employing the controller's intervention.

$$\bar{t}_{tc} \geq \bar{t}_{RTT_{sc}} \tag{5.22}$$

If instead of waiting for the unanimous agreement n^* waits only for the majority quorum, $\bar{t}_{tc}$ can be reduced since there will be no need to wait for slow or faulty switches. This means that only the majority of switches will perform the transition $s_2 \rightarrow s_3$, leading to an inconsistent state for s among the devices. It must however be taken into account the fact that a stateful dataplane, differently from the common distributed systems, is not a completely uncoordinated system. The presence of the controller in fact, guarantees that any failure in the dataplane must and will be detected in a certain amount of time [14,38] and the switches will eventually be notified about this event. This leads to the relaxation of the definition of *Network partition* to transient events related to single packets during their path, such as losses or corruption, and not to the failure of equipment. Therefore in real systems it is definitely possible to reconcile any type of inconsistency in the devices not belonging to the majority quorum in a finite amount of time (ideally in 1 additional RTT). However even if the inconsistency is transient it can still potentially lead to loops and black-holes (i.e. scenarios in which one device or part of the network loses or drops part of the received traffic).

Although one can argue that the manifestation of event e_2 before the commit of $s_1 \rightarrow s_2$ has been completed is an extreme case it is important to underline the fact that strongly consistent schemes present a delay of at least 2 RTTs in order to commit an operation. In real scenarios the presence of losses and packet corruptions alongside with the enqueuing delays must be added to the minimum delay, even if the latter can be partially mitigated by defining a suitable scheduling policy for signaling packets. When considering the fact that s may be defined as a per-packet processing rule with variation frequency of ΔS *operations*$/s$, under assumption of no network partitions, it is clear that, if (5.23) with $\alpha = 3$ holds, it will be impossible to provide real-time consistency and the system

will always replicate a value s related to a moment in the past.

$$\frac{1}{\Delta S} \leq \alpha \cdot max(RTT_{n^*,n_i}) \ \forall n_i \in \mathcal{N} \setminus n^* \tag{5.23}$$

$$\frac{1}{\Delta S} = \alpha \cdot max(RTT_{n^*,n_i}) \ \forall n_i \in \mathcal{N} \setminus n^* \tag{5.24}$$

As a final remark in Section 4.3 it was shown how strongly consistent schemes such as Raft of Paxos may require an unbounded amount of memory at each device in order to provide correctness properties in case of prolonged network partitions. This requirement can be easily satisfied in the case of commodity servers since the provided amount of memory is in the order of Gigabytes or even Terabytes. In the case of forwarding devices it is common to find memory capacity in the order of Megabytes [55] which should be shared among all the functions defined in the switch, thus leading to considerable implementation limitations.

Eventual Consistency for stateful dataplanes

With an eventually consistent scheme it is possible to achieve fast writes to the system thus satisfying both a) and b) since the system will be able to replicate each state in real-time up to the extreme case of (5.23), i.e. as (5.24) with $\alpha = 1/2$ (assuming propagation delays to be equal in both directions). There is however no way to guarantee CAP-Consistency of states at any given time. Similarly to the previous case it is possible to guarantee that after a given amount of time and under the condition of small ΔS eventually each replica will contain the same value of s. It is evident how both schemes are not able to provide satisfactory properties for state replication. If it is however possible to tolerate a certain degree of error on the state of the replicated data, it is possible to employ eventual consistency and achieve high reactiveness. Consider $\mathcal{N} = \{n_1, .., n_N\}$ forwarding devices each of them storing an estimate $\hat{s}_{n_i}$ of the true value s_{true} of the state s. Under assumption of no network partitions, it is true that under (5.23) with $\alpha \geq 3$ a strongly consistent scheme will lead to (5.25) where t_{DS} is the result of the second term of (5.23). With an eventually consistent scheme employing optimistic updates Equation (5.26) holds with $t_{DE_i} = \frac{1}{2} RTT_{n^*,n_i}$ where $n^* \in \mathcal{N}$ is the originator of the update. It is easy to observe that with an eventually consistent scheme each n_i stores a fresher value of $\hat{s}_i$ in respect to the one employing strongly consistent algorithms, although at any given point there is no guarantee in the fact that each $\hat{s}_i$ is the same.

$$\hat{s}_{n_i}(t) = \hat{s}_{n_j}(t) = s_{true}(t - t_{DS}) \ \forall n_i, n_j \in \mathcal{N} \tag{5.25}$$

$$\hat{s}_{n_i}(t) = s_{true}(t - t_{DE}) \ \forall n_i \in \mathcal{N} \tag{5.26}$$

Trade-off between Strong and Eventual consistency

For states which require strict consistency it is mandatory to employ CP schemes in spite of all the related limitations. However if the state is defined in such way to tolerate slight

inconsistency in respect ot the true value eventually consistent algorithms lead to better results in terms of replication delays and total introduced overhead due to synchronization traffic. Moreover eventually consistent schemes require less resources in terms of computational power and memory which is a very well suited property in case of dedicated hardware such as the one employed for stateful forwarding devices.

5.2 Replication triggering

In order to provide a functional replication scheme for stateful dataplanes it is necessary to provide means to trigger the actual replication process. Differently from systems based on general purpose hardware and employing flexible real-time operating systems, forwarding devices are based on a very specific hardware and typically do not employ any operating system at all. This leads to numerous challenges related to the definition of more complex functionalities. Among multiple limitations due to the absence of a flexible operating system forwarding devices do not have any native capability of scheduling events or performing polling. This factor represents a major limitation since it seemingly prohibits completely the use of consistency schemes based on timeouts such as *Delta Consistency* and the majority of strongly consistent algorithms. Typical forwarding devices can in fact be modeled as *Event-Driven* systems, i.e. systems which evolve only at discrete, not necessarily periodic time instants. In particular, in forwarding devices an event is defined as either the reception of a new packet or its progression through the internal pipeline. Nonetheless only when the processing of a packet starts it is possible to perform a match-action thus allowing the system to evolve.

In this section are briefly presented some of the solutions to this particular issue with major emphasis on their advantages and drawbacks.

5.2.1 Event-driven triggering

The simplest way to trigger the replication of a state is whenever the state is actually updated. In order for it to be updated an external stimulus must be fed into the system and as stated above this stimulus can only be the processing of a packet or its progression through the pipeline. Since the time interval between the reception of a packet and its processing at the last stage of the switch is negligibly small, an event will be assumed to be the actual reception of a packet at one of the ingress ports of the switch.

Event-drive triggering can provide means for replication triggering whenever the state is actually modified, thus allowing to avoid useless overhead due to synchronization traffic whenever the state does not undergo any change. It follows that the replication frequency adapts automatically to the rate of the generation of events affecting the state.

This approach is very well suited for states with small variation frequency or for states that require small or no error on other replicas. On the other hand it presents a considerable drawback due to the fact that in some scenarios it may generate unsustainable synchronization traffic load between the replicas. In a scenario of a system composed by $\mathcal{N} = \{n_1, .., n_N\}$ replicas each of them storing a local counter c_i which is replicated whenever it is incremented by 1. The triggering event in this case corresponds to the the

reception of a packet which matches a match-action table that has as action *Increment(c_i)*. This means that for each incoming packet belonging to flow f_d matching that rule a new synchronization flow f_s is generated in order to carry the update information towards other replicas. Consequently the total introduced traffic in the network can be expressed as in (5.27). If the synchronization flow is defined as $f_s = \alpha f_d$ where α is a constant defined as the average size of the synchronization packet normalized in respect to the average size of a packet belonging to f_d (5.27) can be rewritten as (5.28). From (5.28) follows that each incoming packet belonging to f_d is amplified by a factor of $(1 + \alpha N)$. It is intuitive to notice that the total introduced traffic grows linearly with αN. For large number of replicas or for large α, i.e. for states encoded with large number of bits (e.g. long hash tables) the introduced overhead may start to play a significant role. For scenarios of Distributed Denials of Service (DDoS), when the network expects significant loads that has as a purpose the sole disruption of the service, this kind of triggering scheme may amplify all the incoming flows leading to an even worse exacerbation of the service.

$$f_{tot} = f_d + N f_s \tag{5.27}$$

$$f_{tot} = f_d(1 + \alpha N) \tag{5.28}$$

5.2.2 Value-delta triggering

A possible solution to the case of pure event-driven triggering can be the definition of an interval ΔV which allows to perform the replication of the state only when a variation of ΔV of the state value has been observed in respect ot the previously replicated value. This approach allows to reduce the introduced synchronization overhead from (5.28) to (5.29). This scheme has as cost the reduced estimate accuracy yet it still provides a lower bound on the estimate error on other replicas in the absence of network partitions. The introduction of the update interval however does not solve the issue related to DDoS amplification as it only reduces the total introduced traffic by a fraction which must still be comparable with the updates' frequency.

$$f_{tot} = f_d(1 + \frac{\alpha}{\Delta V} N) \tag{5.29}$$

5.2.3 Time-delta triggering

As it was stated previously current implementations of stateful forwarding devices do not provide support for arbitrary polling without external stimuli. It is however possible to access the internal clock on the vast majority of the devices. Given this possibility it is possible to exploit the transient events in order to provide a time-periodicity for the generation of new ones.

Given the event arrival pattern $\Lambda = \{(e_1, t_1), .., (e_n, t_N)\}$ where e_i represents the arrival event and t_i represents the corresponding arrival time so that $t_i < t_{i+1}\ \forall i = 1..N-1$, with the inter-arrival times forming an arbitrary distribution $f_t(x)$, it is possible to perform the sub-sampling of Λ forming a new list of events $\Lambda' = \{(e'_1, t'_1), .., (e'_M, t'_M)\}$ so that

(5.30) holds, where Δ is the desired periodicity in events and ϵ_i is $i-th$ realization of a random variable describing the error due to the randomness of the event-generating pattern. Assuming $f_t(x)$ to be memoryless it is true that $E[\epsilon] = E[f_t(x)]$ and it follows that for small $E[f_t(x)]$ or large Δ it is possible to obtain small periodicity error ratio since the fraction $E[\epsilon]/\Delta$ tends to 0.

$$t'_{i+1} - t'_i = \Delta + \epsilon_i \quad \forall i = [1..M-1] \tag{5.30}$$

The possibility of generating periodic events allows to exploit consistency schemes such as *Delta Consistency* allowing to obtain a temporal error on the consistency level among the replicas while guaranteeing that in a given time interval all replicas will eventually converge to a single value. This type of replication completely eliminates the issue in the case of DDoS and further allows to perform trade-offs between the desired level of consistency and the introduced synchronization overhead by setting the desired temporal upper bound for the reactiveness to critical events, ultimately leading to highly controllable replication schemes.

Time-delta triggering however presents a big flaw since it assumes Λ to be a stationary process with constant mean and variance. In the reality Λ is modeled by the traffic arrival pattern and it is not generally true that the arrival patterns are stationary. It may in fact happen that a critical event occurs at the reception of a packet at time t_0 and the subsequent packet is received only at time $t_1 > t_0$ with $t_1 - t_0 >> \Delta$ leading to the exposure of the network to a criticality for an unacceptably prolonged amount of time.

5.2.4 Controller-driven triggering

Although it is not possible to guarantee the stationarity of the traffic arrival pattern it is still possible to generate one. In fact since the controller responsible of managing the state of each device it must periodically poll for their information with a *RequireState(ID)* message to which the switches must respond with a *Alive(ID)* message. The Alive message can also incorporate the the set $\mathcal{S}$ of still uncommitted states leading to *Alive(ID,*$\mathcal{S}$*)*. This periodic message exchange can be exploited in order to create a a periodicity of $T = \Delta + \epsilon$ with Δ being a tunable parameter at the controller and ϵ being the error due to the network fluctuations. Under uniform network load it is possible to keep ϵ constant and small in respect to Δ and eventually provide a small periodicity error.

It must be taken into account however that network partitions may occur which may eventually lead to the loss of communication between the forwarding devices and the controller for arbitrary long periods of time. If the partition lasts long enough the controller may exclude the affected devices from the replication ensemble by notifying all other switches with a *ReplicationExcluse(ID,*$\mathcal{S}$*)* message which will also carry the last uncommitted values of the faulty devices allowing to partially prevent the loss of updates. If the network partition is instead healed before the controller decides that the device is faulty, depending on the nature of the partition, the updates may be delayed by a factor of $M\Delta$ with M being the maximum number of unanswered *RequireState(ID)* messages the controller is willing to tolerate before deciding that the switch has failed.

In the case of intermittent partitions or unusually high network load the periodicity error will be further influenced by the loss of *RequireState(ID)* packets and can become significantly big leading to incorrect behaviors of the replication scheme.

5.2.5 Virtual queue-based triggering

The previously presented schemes are based on an external stimuli, i.e. the arrival of a packet to the forwarding device while the progression of the packet through the internal pipeline has been completely neglected. As mentioned previously it is possible to trigger event thanks to the progression of a packet through the internal processing pipeline such as for example its extraction from one of the queues between the processing stages.

If there exists a non-empty queue q^* with infinite capacity it is possible to generate periodic events with a small periodicity error by continuously extracting packets from it. If q^* contains the transiting data packets this approach quickly degenerates in 5.2.1 since the queue can become empty and disrupt the desired periodicity. In order to solve this issue and guarantee that the queue is never empty it is possible to fill it with dummy packets which only scope is that of providing periodic triggering of the system advance. The presence of dummy packets in the common queue however risks to obstruct the processing of data packets enqueued behind the dummy packets by introducing additional processing delay and reducing the overall performance. It is however possible to define a virtual queue q_v destined entirely to the storage of the dummy packets. In this way all data packets will be inserted in their related queues q_i and will not be obstructed by the dummy ones. If non-TDM scheduling schemes are adopted this approach still will not provide any guarantee for the periodicity of events. If at a given time t it is true that $occupancy(q_i) = 0 \ \forall i$ packets from q_v will be extracted and processed at line rate even if there is no update to perform, leading to a mechanisms which in Software Engineering goes under the name of *Busy Waiting*. It is easy to observe how this technique although providing fine grained time accuracy requires a conspicuous amount of resources in terms of energy since the switch behaves as it was under 100% load for the entirety of the time.

The excessive processing load introduced by this technique can be easily mitigated while still providing a fine grained accuracy by employing a policer in combination with a priority-aware scheduler. If the desired replication schemes requires delta consistency with a time interval of ΔT seconds it is possible to impose a policing policy of $1/\Delta T \ pkts/s$ on q^* thus allowing to extract packets with a periodicity of $\Delta \ s$. Furthermore, in order to avoid situations in which the processing of packets from q_i may delay the extraction of a dummy packets a scheduler based on priority queues can be employed in order to guarantee that $q*$ is served as first at any given time. It is expected that policers are implemented in a efficient way directly in hardware and do not introduce any computational overhead and performance degradation as in the case of a busy waiting. In case of an architecture exploiting poorly implemented hardware solution this scheme can still lead to excessive performance degradation and increased energy consumption.

5.2.6 Hardware support

The most versatile and manageable solution in order to achieve polling at arbitrary time is the use of built-in hardware CPU scheduler as it happens in classic real-time systems. CPU scheduling allows to emulate completely the behavior of general purpose hardware, thus eliminating completely the previously mentioned issues due to replication triggering. Most of the commercial hardware however do not provide any support for this kind of feature: even if both P4 and OpenFlow enabled devices support hardware timeouts for flow tables there is no possibility of exploiting such mechanism out of their original scope. Consequently the introduction of arbitrary timeouts for dataplanes results challenging without introducing modification to the underlying hardware.

5.3 Synchronization traffic transport and representation

Assuming that the system exploits one of the triggering mechanisms described in 5.2 the in order to define a new replication systems it is necessary to define how to efficiently represent the update and transport it towards other replicas. Classical approaches do not apply in the case of stateful dataplanes since they do not exploit a centralized knowledge of the network, thus resulting in poorly efficient and rather costly approaches based on extensive exchange of signaling information and avoidable internal memory overhead. Although multiple triggering schemes have been discussed in section 5.2, the actual procedure for the generation of packets ex-novo entirely in the dataplane must still be defined.

5.3.1 Update transport

The generation of new packet at arbitrary time has never been among the essential features required in stateful dataplanes and there has never been any related work targeting this particular issue. Although OpenFlow protocol requires the presence of a mechanism able to generate control packets in order to be able to exchange information with the controller, as mentioned before this particular mechanisms is constrained to its scope and cannot be further extended.

Assuming that due to one of the previously described mechanisms a replication procedure requiring the transport of a state-update $u = (n_1, n_2, s)$ of state s from switch n_1 to switch n_2 has been triggered at n_1. The switch must be able to generate the actual update packet and convey it to n_2. Depending on the adopted triggering schemes 3 procedures for the generation of u have been defined.

Piggybacking

Upon the triggering of a replication procedure due to an external event represented as the reception of packet p_t, the same packet can be exploited in order ot carry the update information u. This information can be directly attached to the original packet, i.e. piggybacked by it, forming $p_u = \{u, p_t\}$ and thanks to the possibility of defining custom network-wide rules it is possible to efficiently forward this packet from n_1 to n_2 through the intermediate network devices. Once p_u has been received by n_2 it undergoes the reverse

transformation by removing the part of the packet carrying the state-update information and is forwarded to its original destination. This scheme is of easy implementation and is currently supported by the vast majority of SDN-enabled architectures.

The intrinsic limitation of this approach is that, in case of large networks, data packets can be significantly delayed. In a scenario depicted in Figure 5.3 switch SW_1 is traversed by only one flow f destined to switch SW_2. At the same time SW_1 and SW_4 belong to the replication ensemble for state s and are separated by a geographical network AS_1 with a non negligible propagation delay. In a scheme exploiting piggybacking packets belonging to flow f will be selectively chosen in order to carry state-update information towards SW_4. Due to the delay introduced by AS_1 this approach may lead to out of sequences for packets belonging to f and depending on the actual value of the delay in extreme cases may lead to timeouts and/or losses. Moreover since p_u is not forwarded based on its conventional information such as for example the original destination IP address but instead is forwarded based on the information contained in the state-update u each intermediate switch in the network will not only require to know how to route the packet based on u, which is mandatory, but also based on the original p_t. This means that each switch belonging to AS_1 will need to contain the routing rules related to flow f even if normally f does not traverse AS_1. In extreme scenarios this may potentially lead to a behavior when the insertion of a new forwarding rule needs to be replicated on all devices belonging to the network in order to coup up with loops and black holes.

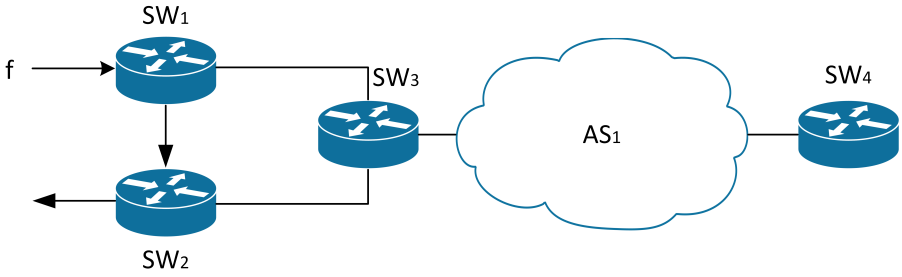

Figure 5.3. Example of piggybacking with excessive delays.

Cloning

A viable alternative to piggybacking is based on packet cloning. Similarly to the previous case the update is still piggybacked by the data packet that triggered the update but instead of forwarding the original packet it is possible to trigger a cloning routine and transmit only the clone while transmitting the original data packet on its regular path. This approach allows to keep the number of total rules in the network unaltered since once p_u reaches its destination it can be simply dropped instead of being sent back to the originator of the update. Moreover there is no manifestation of any issue related to the excessive delay of the data flows as this approach appears to be completely transparent for all involved data flows.

In order to be exploited, this techniques requires the support of cloning or mirroring procedures directly inside the switch. While the vast majority of devices are able to add additional information to the transiting packets, more complex operations such as packet cloning may not be supported by low-end devices such as the ones discussed in Section 3.3.7.

Queue extraction

While some architecture may not be able to perform packet cloning it may not arise any issue if the state-update generation is performed in combination with the triggering mechanism based on Virtual Queues discussed in 5.2.5. Since the state-update is triggered by the extraction of a dummy packet from the corresponding queue the very same packet can be exploited in order to carry update information without interfering with the already present data flows. Since the dummy packets are eventually removed from the queue and are not recirculated as it was happening for the triggering alone this approach necessities a mechanism able to refill the queue with dummy packets. If it is possible to perform cloning it is sufficient to perform the recirculation of the dummy packet inside the queue while exploiting its clone in order to generate the state-update packet. In the complete absence of cloning primitives the refill can occur with the intervention of the controller which may periodically send bursts of dummy packets to the related switches. Alternatively the switch may exploit transiting data packets which are marked to be dropped or are corrupted/invalidated.

5.3.2 Update encoding

Assuming that a suitable triggering mechanism and the corresponding packet generation scheme has been chosen the issue of actually representing the state-update arises. It is necessary to devise an update format able to unequivocal and efficiently represent each state-update packet. For the case of classical packets in non-SDN networks each state-update packet requires a necessary set of information in order to perform the routing through the network, a set that depending on the field of application requires at least a L3 protocol header. Considering the case of IP this means that each packet will eventually require an Ethernet header an IP header and the eventual L3 payload with the state-update information. In case of SDN networks composed by $N << 2^{23}$ switches the use of IP in order to identify the switches leads to the transmission of unnecessary information and results in avoidable overhead as for many uncoordinated network protocols in coordinated systems.

Due to the presence of a controller it is possible to assign a unique progressive identification number $swID$ to each device to be used in order to unequivocally identify each entity in the dataplane. The choice of a unique switch ID alongside with a unique state ID allows to define the minimum amount of routing information that needs to be included in the state-update packet. Similarly to the case of $swID$, since the system is completely coordinated, each state can be identified by a unique progressive number sID that is chosen by the controller when the rule is initially embedded in the dataplane. The encoding of both sID and $swID$ in a progressive integer furthermore allows to provide support for

different architectures and different control plane protocols since sID can be mapped to the desired representation or action directly inside each single forwarding device.

Given the destination switch identifier $swID_{dst}$ is it theoretically possible to unequivocally route the packet among each switch belonging to the replication ensemble. This information however can be further reduced by taking into account the fact that for each state with ID sID there exists a unique set of switches belonging to the replication ensemble. Given two replicated states with IDs sID' and sID'' and the corresponding sets of switches belonging to the respective replication ensemble $\mathcal{S}' = \{swID'_1, .., swID'_n\}$ and $\mathcal{S}'' = \{swID''_1, .., swID''_m\}$ it is possible to tolerate the satisfiability of (5.31). If the forwarding is performed based on a tuple $(sID, swID_{dst})$ there always exist an unequivocal way of deriving the destination of the update even if (5.31) is satisfied since there do not exist any two or more switches with the same sID and $swID$.

Given the set of unique state IDs $\mathcal{S} = \{sID_1, .., sID_N\}$ and the set containing the sets of switches involved in the corresponding replication ensemble $\mathcal{N} = \{\mathcal{N}_1, .., \mathcal{N}_n\}$. The representation based on tuple $(sID, swID_{dst})$ allows to further reduce the average amount of transfered bits related to the forwarding information from (5.32) for the case of forwarding performed based on $swID$ alone to (5.33) for the case based on tuple with overlapping $swIDs$.

In order to accomplish the state-update each update must also carry the actual update-value v_i related to each state sID_i which for simplicity is assumed to be of fixed length $|v_i| = l_i$ bits. The inclusion of l_i leads to (5.34) for the average amount of bits needed in order to provide state replication-update transport in the case of forwarding based only on $swID$. Similarly (5.35) provides the result in case of tuple-based forwarding. It is worth mentioning that the inclusion of sID in (5.33) does not introduce any additional overhead in respect to (5.32) since it appears in both final equations.

$$\mathcal{S}' \cap \mathcal{S}'' \neq \emptyset \tag{5.31}$$

$$\bar{B_f} = 2\lceil log_2(|\bigcup_{i=1}^{n} \mathcal{N}_i|)\rceil \tag{5.32}$$

$$\bar{B_f}' = 2\lceil log_2(\frac{1}{n}\sum_{i=1}^{n} |\mathcal{N}_i|)\rceil + \lceil log_2(n)\rceil \tag{5.33}$$

$$\bar{B} = 2\lceil log_2(|\bigcup_{i=1}^{n} \mathcal{N}_i|)\rceil + \lceil log_2(n)\rceil + \lceil \frac{1}{n}\sum_{i=1}^{n} l_i\rceil \tag{5.34}$$

$$\bar{B}' = 2\lceil log_2(\frac{1}{n}\sum_{i=1}^{n} |\mathcal{N}_i|)\rceil + \lceil log_2(n)\rceil + \lceil \frac{1}{n}\sum_{i=1}^{n} l_i\rceil \tag{5.35}$$

5.3.3 Update dissemination

Final remarks must be addressed on how the the updates are actually propagated through the network. Depending on the nature of the chosen consistency scheme it is possible to further reduce the total introduced overhead defined in (5.35).

Shared Spanning Tree

In schemes employing direct communication among each replicas, thanks to the centralized knowledge, it is possible to avoid performing the update dissemination based on $(sID, swID_{dst})$ and instead switch to a $(sID, swID_{src})$ while completely neglecting $swID_{dst}$ in the update packet representation. This is made possible thanks to the knowledge of the elements composing each replication ensemble for each replicated state. Nonetheless the controller is able to build a unique *Shared Minimum Spanning Tree (SMST)* among the replicas belonging to each replication ensemble while guaranteeing that the tree is loop free. Upon the reception of a packet containing $(sID, swID_{src})$ the switch, based on sID, performs a multicast of the packet on all the ports belonging to the replication tree while avoiding to send it back on the port on which the packet was originally received. If excluding the original reception port no other ports belong to the replication tree the switch stores the update value since by construction it belongs to the replication ensemble of sID and drops the packet. The inclusion of $swID_{src}$ serves as a double-purpose since on one hand it allows to perform forwarding, on the other hand it provides means to identify the originator of the update and to subsequently perform any eventual conflict resolution among conflicting updates.

This approach not only reduces by a factor of 2 the first term in (5.35) but allows also to reduce the total number of rules which are instantiated inside the switches since in the worst case each switch would require an amount of rules equal to the total number of unique states, i.e. n rules.

The proposed state distribution mechanism shares the same ideas with *Protocol independent multicast-sparse mode (PIM-SM)* [25], albeit PIM-SM defines also a mechanism in order to join and leave multicast groups and requires dedicated hardware able to handle this type of requests.

Extensions of SMST

Similar approaches can be employed for the case of selective communications such as gossiping. Given a replication ensemble $\mathcal{R} = \{swID_1, .., swID_n\}$ for a generic state variable s^* of size l^* bits it is possible to avoid probabilistic gossiping by constructing k sets $\mathcal{K}_i \; \forall k = i..M$ so that $\bigcup_{i=1}^{M} \mathcal{K}_i = \mathcal{R}$ and $\mathcal{K}_i \cap \mathcal{K}_j \neq \emptyset$. This allows to employ asynchronous communication and replication schemes based on CRDTs while still preserving good properties in terms of memory occupancy. Each switch can in fact perform replication inside its own replication subset $\mathcal{K}_i$ thus allowing to keep the memory requirements in the order of $|\mathcal{K}_i| \cdot l^*$ while thanks to the existence of intersections among each $\mathcal{K}_i$ guarantee that the updates will be eventually propagated to all other replicas.

Chapter 6

Implementation and evaluation

Given all the challenges that must be affronted in order to provide a functional designing for a replication scheme directly inside the dataplane, exhaustively discussed in Chapter 5, in this Chapter the implementation of a real replication mechanism entirely in the dataplane is presented. The implementation is performed with the use of $P4_{14}$ and each aspect of the design is deeply motivated with the introduced benefits alongside with the limitations.

6.1 Design choices

6.1.1 Problem description

As mentioned in 3.4 whenever there is a rule with a global scope the state placement in SNAP forces unbearable load on the host of the state and consequently on all the related links. An example of this behavior is a state responsible of detecting and preventing DDoS attacks. It is generally true that given an autonomous system AS connected to other autonomous systems via a set of border routers, whenever it undergoes a DDoS attack it is possible to observe a correlated increase in the incoming traffic on all or on a biggest fraction of the border routers. The solution for the state placement obtained with classical SNAP would impose the placement of the state variable in the middle of the network, more generally on a switch n^*, such that the average distance in terms of number of hops from all the border routers towards it is minimum. In a scenario of a real DDoS attack that would imply that all the incoming traffic would converge to n^* leading to the saturation of all the downstream switches' queues, with an apparent (although transient) success of the attack. This situation would require n^* to perform a reaction by first starting to drop the packets and by notifying the controller of the presence of a DDoS attack. However in a scenario like this it may happen that the path to the controller lies on the already attacked network thus any attempt to contact the controller will eventually lead to a failure due to high packet loss probability. In addition to that during the attack n^* may also fail thus losing the state variable and consequently delaying the detection of the DDoS attack.

If instead of employing classical SNAP with a single state variable multiple replicated

states are placed on the border routers of AS the issues related to the signaling of the presence of an attack can be partially mitigated. Since the state variable is distributed among multiple switches once the attack is detected all of them simultaneously start ot drop the incoming packets, leading to the isolation of AS from any potential external attack. In a similar way if one of the border routers fails only a fraction of the actual attack will be able to infiltrate the network while the majority of the attack will still be blocked.

Due to its relevance and simplicity the example of a DDoS attack has been chosen in order to perform the implementation and the consequent evaluation of a prototype. The problem of detecting DDoS attacks is a very well known one and multiple works are available in the literature (an exhaustive survey is available in [62]). Since the definition of the actual detection mechanism is out of the scope of this work it was opted to chose a simple meter implemented as a moving average estimator able to capture the number of packets per second traversing each border router. The actual attack is then detected once the aggregate of the incoming traffic on all border routers exceeds a certain threshold.

It is easy to observe the necessity of performing state replication and the impossibility of employing simple sharding. The DDoS attack example is indeed a variation of the distributed traffic policed presented in Section 4.1.1 which requires a reduction on the aggregate of distributed states. Since the function that defines the satisfiability of the threshold must keep into account the state of all the border routers, by extending the notion introduced in 4.1.2, the detection of the attack occurs when given a set of replicated states $\mathcal{S} = \{s_1, .., s_N\}$ with a reduction function $r(\cdot)$ and a satisfiability function $P(\cdot)$ (6.1) is satisfied. The reduction function performs the sum of all counters belonging to the replicas while the satisfiability function confronts the result against a threshold.

$$P(r(\mathcal{S})) = \begin{cases} 1, & \text{if } \sum_{i=1}^{N} s_i > Th \\ 0, & \text{otherwise} \end{cases} \tag{6.1}$$

6.1.2 Consistency level

It is evident for this particular case how strongly consistent schemes would lead to a large delay in the update of the states. Indeed in the case in which a 3PC replication scheme is employed the reaction time of the whole system can decreases dramatically due to the high communication delay and packet losses caused by the leakage of the attack in the AS. On the other hand an eventually consistent scheme can provide fast synchronization of the distributed meters even if strict consistency will not be guaranteed. The error due to the inconsistency is however not critical in this particular scenario since in the case of a DDoS attack and with no network partition it introduces at most 1 extra update among all replicas. Since the consistency is not guaranteed at any given time the satisfiability of (6.1) at each state-replica degenerates into (6.2).

$$P_j(r(\mathcal{S})) = \begin{cases} 1, & \text{if } s_j + \sum_{i=1,\ i\neq j}^{N} \hat{s_i} > Th \\ 0, & \text{otherwise} \end{cases} \quad \forall j = 1..N \tag{6.2}$$

6.1.3 Update encoding and dissemination

Instead of representing each state-update with a delta in value since the last update it was chosen to define the state-updates as the instantaneous value of the meter at time instant t. The transmission of the complete state value allows to avoid loss of information in the case of packet losses and instead just delay its propagation to other replicas. This choice allows to avoid expensive synchronization mechanisms based on acknowledgments and timeouts by instead employing optimistic updates at the cost of an increased update packet size. An alternative approach exploiting differential updates has been developed and tested under optimal network conditions leading to same results in respect to full state update. Due to the extreme difficulty in implementing a reliable L2 channel in P4 and the lack of computational power in order to provide empirical results it was chosen to opt for the former approach.

Since the adopted topology consists entirely of P4-enabled switches the were no issues in defining a well suited state encoding mechanism. In order to provide means to unequivocally identify different states it was chosen to employ a progressive positive integer for each of the states. Similarly a progressive positive integer has been chosen for each switch in order to provide unequivocal distinction among the update originators directly in the dataplane. Both informations are instantiated by the controller at start-up.

6.1.4 Update triggering

In Section 5.2 among the presented triggering mechanisms it was shown how the one based on time-delta triggering was the most suited in the case of states targeting the detection of DDoS attacks. The main motivation behind time-delta triggering in massive attacks was due to the fact that this triggering mechanisms does not provide attack amplification. For this particular design however, it was chosen to use the value-delta triggering. The main motivation behind this choice lies in the fact that this particular type of triggering is more flexible towards implementations targeting states of different nature. In fact with value-delta triggering it is possible to implement a large variety of replicated states that require updates based on single events without introducing any significant change to the prototype.

Due to the nature of the generated traffic, as later discussed in Sections 6.2.3 and 6.4.3, the adopted triggering mechanism does not present substantial differences in terms of performance and reactiveness in respect to the time-delta one.

6.1.5 Update generation

As for the actual generation of the state-update packets it was chosen to exploit the built-in cloning primitives since it resulted to be the most efficient for the chosen topology and due to the fact that it does not present any issue in combination with the chosen triggering scheme.

6.2 Testbed

6.2.1 Target topology

In order to provide a real case scenario the topology represented in Figure 6.1 was chosen. The elements composing the topology are the following:

- **ASNX**: Autonomous System Number X are the ASs external to the topology that generate the incoming traffic. For further discussion the internal autonomous system, i.e. the one under control is assumed to be $ASN0$.
- **SWX**: P4-enabled switches which run the DDoS detection mechanisms.
- **RX**: Border routers that connect $AS0$ to other ASs.
- **Server Cluster** x **(SC**x**)**: Networks internal to $AS0$ towards which the DDoS is directed.

In a classical SNAP scenario given this particular topology, due o its symmetricity, the state monitoring the DDoS would have been placed in one of SWX. Assuming that the state is placed in SW1 this means that all the incoming traffic should converge to SW1 before eventually being forwarded to the destination SC. This means that in case of a real DDoS attack, assuming shortest path routing for the convergence to SW1, only links R4-SW3 and R3-SW3 would remain unaffected by the attack until it is eventually detected by SW1 and a countermeasure is applied. This traffic convergence not only introduces unsustainable overhead but, as mentioned earlier, can also disrupt the communication between the SCs leading to an apparent success of the DDoS.

In the case of replicated states, more precisely, which 4 replicas placed at the border routers it is possible to guarantee that in the case of a DDoS attack the connections among SCs will not undergo long-lasting service disruption since it will be possible to mitigate the attack before it is able to completely infiltrate AS0. This guarantee is not provided in case of 2 or 3 replicas however the total introduced overhead due to traffic convergence, as later shown in Section 6.4, can be considerably reduced.

6.2.2 Virtual environment

The design and evaluation of the prototype has been performed entirely in a virtual environment, namely *Mininet* [57]. Mininet is written in Python and through a set of primitives provides the possibility of emulating a broad variety of network devices even if it was initially developed targeting specifically SDN. Differently from simple emulators, Mininet allows to build real-time networks of virtual switches and virtual hosts with the support of on-the-fly modifications to the state of virtual links thanks to the provided built-in primitives. In addition to the proprietary primitives, virtual links are exposed to the host system as virtual interfaces, thus allowing to perform advanced link manipulation directly in the host.

The P4 Language Consortium and the developers' community provided a modified version of virtual switch targeting the *Simple Switch* architecture described in Section

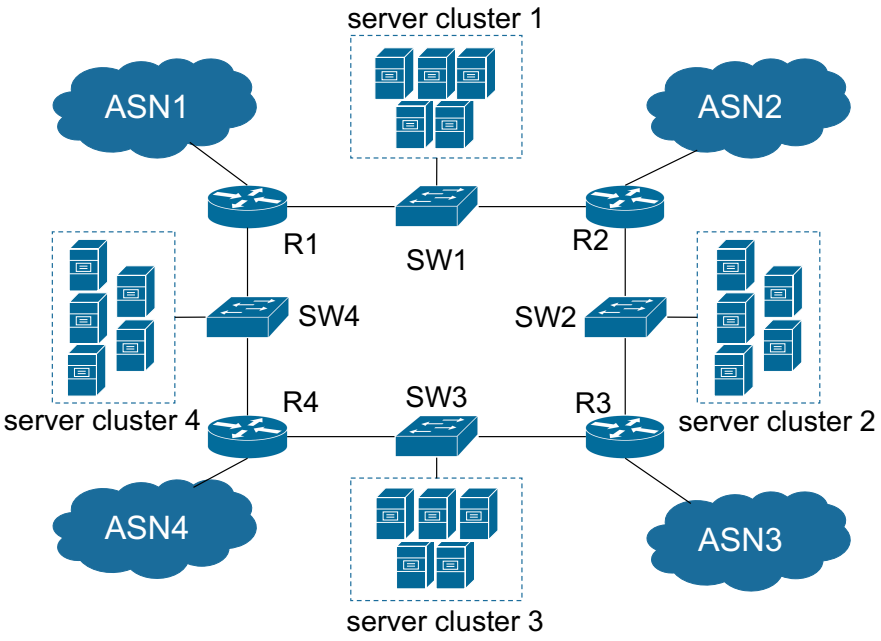

Figure 6.1. Test network topology for the DDoS detection scenario.

3.3 and able to interpret and execute P4 programs. Although different virtual switches targeting different architecture are also provided it was decided to employ the one targeting the former due to increased support of advanced primitives. For simplicity and for the sake of increased flexibility both Rx and SWx were implemented using the P4 virtual switches and both of them will be referred to as *vSWs* throughout the subsequent sections.

6.2.3 Traffic generation

It was chosen to employ constant bit rate (CBR) traffic which, thanks to the simplicity of the traffic pattern, allows to easily unveil any anomalous behavior. In order to provide highly controllable traffic generation two approaches have been used:

- **Scapy**: Scapy [8] is a traffic generator written in Python which provides the possibility of generating packets of arbitrary format. Differently from common traffic generators Scapy allows to easily define and generate packets with a custom format that does not follow any particular standard. Due to the fact that Scapy operates at high machine level it is not possible to generate CBR traffic with a high data rate. Because of this limitation the use of Scapy was employed mainly in early stages of the development and for debug purposes. Additionally, since Scapy supports arbitrarily small data rates, with a sufficiently small packet generation rate the system behaves as if the state-updates are propagated instantly, i.e. with zero delay, since the events that advance the system are of an arbitrarily small frequency. This aspect allowed to further analyze the behavior of the prototype under the most optimal assumptions.

- **Iperf**: In order to evaluate the behavior of the prototype in real case scenarios with a sufficiently big load the packet generation was performed by employing Iperf [58]. Although Iperf does not have the flexibility of Scapy in the definition of the packet format it runs at Kernel level thus allowing to provide high performance in terms of packet and bit rate.

6.2.4 Traffic capture

vSW implementation provides the possibility of directly capturing all traffic traversing the virtual interfaces to a trace file by setting the related parameter at the vSWs' startup. Due to a yet unsolved compatibility issue in the implementation of this particular feature, under particular host systems, the capture files may undergo corruption in the presence of simultaneous bidirectional communications on the virtual interface. However thanks to the exposure of the virtual interfaces to the host system it was possible to mitigate the issue by perfoming the capture directly from the host. The creation of a correct trace has been achieved by instantiating a *tcpdump* [36] instance for each virtual interface of each vSW.

6.2.5 Table manipulation

The employed vSWs, similarly to OpenFlow switches expose a set of application program interfaces (APIs) in order to allow the population of flow tables. Although OpenFlow relies on the messages defined by the protocol, P4 vSWs' table manipulation is performed through the use of Thrift interfaces [52] which although are instantiated by vSWs, appear to run on the host system thus permitting only out-of-band table manipulation. Up to the present day [1] no controller has been implemented specifically for the combined use with vSWs. Consequently, the definition of each match-action rule and the their deployment inside vSWs has been made by hand via a set of different configuration files for each vSW. It is however expected that in the near future a broad variety of currently available controllers will provide support for P4-enabled switches and a controller targeting specifically vSW architecture will be developed.

6.2.6 Data retrieval

The retrieval of data necessary to perform prototype evaluation was performed by extracting the related information from the captured traces and by exploiting the provided APIs in order to retrieve the internal states of each vSW. Although the APIs operate out-of-band, as it will be further discussed in Section 6.4, due to the intrinsic limitations of the host system and the computational load on vSWs each API call undergoes a delay which is related to the amount of the available resources at the host. The distribution of this delay appears to be random but is still correlated to the amount of packets that are processed at each vSW at the time of the API call.

[1] August 2017

6.3 Implementation

Since the release of a stable $P4_{16}$ vSW occurred after the design was already completed the implementation presented in this work is based on $P4_{14}$. It was chosen to use $P4_{14}$ instead of the beta version of $P4_{16}$ mainly due to the lack of support of some essential features such as packet cloning and due to the scarce documentation of $P4_{16}$ vSWs.

As mentioned earlier the employed $P4_{14}$ vSWs are based on the *Simple Switch* architecture comprising two processing stages with an intermediate buffering stage. Due to the prototype's design specifications and in particular due to the necessity of performing dynamic routing based on multiple fields, the vast majority of the implementation is concentrated in the input CB while the output CB is exploited mainly for labeling and dropping policies.

This Section contains a description of the implemented prototype alongside with some parts of the actual implementation. For the sake of clarity and brevity the code is simplified and only the most significant parts of it are presented.

6.3.1 Double counting

Given the used topology it may occur that with a particular placement of state replicas some packets may undergo a double counting, i.e. counted more than one time while they traverse multiple replicas. This is evident in the case in which replicas are placed at each border router: a packet originating from ASN1 and directed towards SC2 will traverse both R1 and R2 and, unless a double counting prevention mechanism, will be counted 2 times. Double counting can be mitigated by defining different policies at each border router for the flows that are categorized as potentially malicious in such a way that only the first traversed state-replica will account for the contribution of the flows. As an example, assuming 4 state-replicas placed at the border routers, R1 may only count packets originating from ASN1, similarly R2 will only count traffic originating from ASN2 etc... More generally, given a set of replicated state-variables $\mathcal{S} = \{s_1, .., s_n\}$, the corresponding set of switches $\mathcal{W} = \{w_1, .., w_n\}$ so that state s_i is placed in switch w_i $\forall i = 1..n$ and the sets of transiting flows $\mathcal{F}$. Given a flow $f \in \mathcal{F}$ routed through path $P : w_s \rightarrow w_d$ a switch w_i will count the contribution of f if and only if, given $P' : w_s \rightarrow w_i$ it is true that $w_j \notin P'$ $\forall j = 1..n,\ j \neq i$.

The approach based the path of each flow guarantees that under static routing the contribution of each packet will be counted only once, although it does not provide means to distinguish between counted and uncounted packets by intermediate switches. Indeed in scenarios in which packets must exploit a path P' in order to converge to a state replica and than exploit P'' in order to reach its destination with $P'' \subseteq P'$, unless complex and inflexible mechanisms, the intermediate switches will not be able to distinguish whether a packet has been counted and must be routed to its original destination or if it must still be converged to a state-replica. This issue is not present in scenarios in which $P' \cap P'' = \emptyset$ since no confusion about the state of a packet may arise. Since the employed topology in the case of 1 and 2 state-replicas presents routing that does not satisfies $P' \cap P'' = \emptyset$, in order to prevent loops it was chosen to count each packet's contribution at the first traversed replica and subsequently tag the related packets by setting the IPv4 DSCP

field to 1 so that any intermediate stage would be able to perform correct routing and any subsequently traversed replica would be able to discriminate between processed and unprocessed packets thus avoiding the double counting issue.

6.3.2 Cloning

P4 offers a set of cloning primitives targeting the *Simple Switch* architecture. In particular there is the possibility of cloning packets at any CB to any other CB, however only the ingress-to-egress cloning has been implemented and working under vSW. This primitive. Although not very suited in order to define clean and understandable code ingress-to-egress cloning can still be exploited in order to provide the generation of update packets: once the packet is cloned from the ingress CB to the egress one the clone is enqueued in the corresponding egress queue while the original packet continues its processing inside the ingress CB. It leads that for the generation of an update packet the original packet must fully complete its normal processing in the ingress CB and instead of completing the processing in a conventional way it must cloned to the egress. The clone then appears to the egress CB as it has completed the processing in the conventional way while the original packet undergoes an additional processing stage allowing to transform it into an update packet.

6.3.3 State-update exchange

Since each state is composed by a value returned by the moving average which for simplicity is assumed to be a 32bit integer. The state update packet exploits the first 32 bits in order to carry the ID of the state and the ID of the originator of the update, thus allowing to unequivocally identify the originator of the update. For debug purposes and in order to provide further support for new features additional 32 bits have been defined in the packet format and left unused, leading to a total size for the state-update packet of 8 bytes not including the size of state-update value.

The state update value is defined as a variable-length field which allows to represent the state-update packet size in the minimum possible amount of bits. Although P4 supports the presence of only 1 variable-length field, it is highly expected that the support of multiple variable-length fields will be added in the near future. The introduction of this feature will eventually allow to carry multiple state-update values of a variable length by prefixing each of them with their explicit length.

The dissemination of the updates on the other hand follows closely the scheme proposed in 5.3.3 based on common spanning trees for each state-variable.

6.3.4 Parser specifications

The system is assumed to work in combination with Ethernet as layer 2 protocol. In order to provide unequivocal distinction among normal data packets and the update packets the parser was programmed in such a way to perform the distinction based on the *EtherType* field of the Ethernet header with the relevant portion of the code depicted in listing 6.1.

The chosen *EtherType* for the update packets is among those currently not used for any Layer 3 protocols.

```
#define ETHERTYPE_IPV4 0x0800
#define ETHERTYPE_UPDATE 0x9999
parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
    ETHERTYPE_IPV4 : parse_ipv4;
    ETHERTYPE_UPDATE : parse_update;
    default : ingress;
  }
}
```

Listing 6.1. Implementation of the programmable parser for the DDoS use-case.

6.3.5 Rate estimation

As mentioned in the previous sections in order to obtain a stable estimate on the number of transiting packets S at each switch a moving average has been used. The actual implementation of a moving average cannot be easily performed in P4 since the language specifications do not provide any support for complex operations such as multiplication and consequently for all its derivatives.

Exponential moving average

The issue can be partially solved by employing an exponential average as defined in (6.3) where $S(t)$ is the estimate at time t, Y is the number of transiting packets in time interval Δ and $\alpha = \frac{1}{2^k}$ for a generic integer k. In this way the multiplication by α can be performed by exploiting bit shifting while the multiplication by $1-\alpha$ will still employ bit shifting for the denominator and an unfolded summation for the numerator leading to the expression in (6.4). This approach however provides correct results only for Δ constant, but as it was explained multiple times it is generally impossible to achieve this level of accuracy in systems that evolve based on random external stimuli. It is still possible to define an exponential average with variable Δ as shown in [23], but the implementation complexity grows significantly and the constraint on the value of α is still present.

$$S(t) = \alpha Y + (1-\alpha)S(t-\Delta) \tag{6.3}$$

$$S(t) = (Y >> k) + \sum_{i=1}^{2^k-1} (S(t-\Delta) >> k) \tag{6.4}$$

Jumping Window

In order to provide a fexible, yet reasonably accurate estimator it was chosen to employ a jumping window estimator with a circular buffer. The algorithm employs a circular buffer B with N slots each storing a value of the counter. The description of the functionality of the algorithm is presented in Listing 6.2. B is shifted by one slot whenever the time at

which a packet is received exceeds the predefined Δ in respect to the time of the last shift (lines 2-3). Since the buffer is of finite size after each shift the value related to time instant $t - N \cdot \Delta$ is lost and replaced by the value related to time instant $t - (N-1) \cdot \Delta$. The fact that the buffer has a finite size in combination to the fact that after the shift operation $B[0] = 0$ introduces an estimation delay that translates in an overall underestimation of S. The underestimation can however be controlled and the total error introduced to the estimate will have an upper bound inversely proportional to N. The returned value is just a sum of all slots of B if Δ is defined in such a way so that (6.5) is satisfied the returned value will be expressed in $pkts/s$, if instead it is not true the returned value will require a normalization. However the normalization can ba avoided by defining a priori the threshold not in terms of $pkts/s$ but in terms of $pkts/N \cdot \Delta s$ thus allowing to avoid the normalization operation each time the value of S is required.

$$N \cdot \Delta = 1 \tag{6.5}$$

This particular estimator has been chosen for the prototype due to its implementation simplicity and its flexibility which is provided by the possibility of defining different levels of accuracy. It was decided to chose $N = 8$ and to provide satisfiability of (6.5) by defining $\Delta = 0.125s$

```
1 int jumping_window(B, time_now, time_old){
2   if(time_now-time_old > DELTA_T)
3     B >> sizeof(B[0]);
4   B[0]++;
5   return(sum(B));
6 }
```

Listing 6.2. Jumping window estimator functionality.

6.3.6 Custom metadata

In Section 3.3 it was presented the concept of metadata and how different architectures may expose different intrinsic metadata. In additional to intrinsic metadata P4 provides the possibility of defining custom metadata which are attached to the packet's processing pipeline as if they were part of the parsed representation of the packet which allows to transport additional information throughout the processing stages of the packet. The prototype relies on the extensive use of custom metadata in order to carry information such as the state ID, switch ID, value of S etc... The carrying of metadata trough multiple CBs allows to perform the minimum possible amount of interactions with stateful elements thus allowing to achieve higher performance and lower power consumption in real life scenarios.

6.3.7 High level description of the prototype

In order to provide the correct behavior the ingress CB must perform the routing based on the packets' original destination and the forwarding towards the concentration point (i.e. one of the switches storing the state variable) if the packet belongs to a flow that matches a DDoS detection rule. Furthermore in the case of switches containing state-replica they

must be able to identify, correctly process and generate update packets. Figures 6.2 and 6.3 depict the high level operation mode of respectively switches containing a state replica and that of switches responsible of forwarding only. For simplicity the former will be referred to as *Counting virtual switch (CvSW)* while the latter one as *Forwarding virtual switch (FvSW)*.

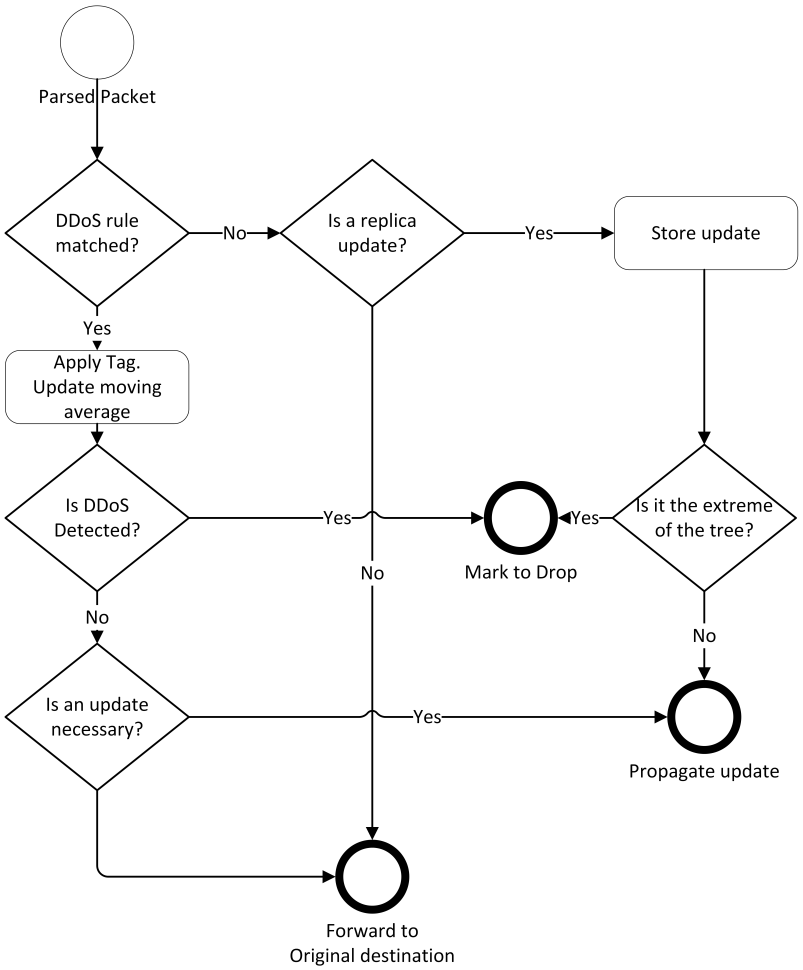

Figure 6.2. High level functionality of switches containing a state-replica.

Forwarding virtual switch behavior

The behavior of *FvSW* is that of a simple forwarding device and can be described with only 2 match-action tables. Upon the reception of a packet the switch must read the destination of the packet in order to understand whether the packet is destined to one of SCs or not. If the destination IP of the packet matches the IP belonging to one of the SC subnets the switch must send it ot the nearest CvSW unless it has already traversed one of CvSWs. In order to achieve this behavior it is sufficient to have a two match-action tables as depicted

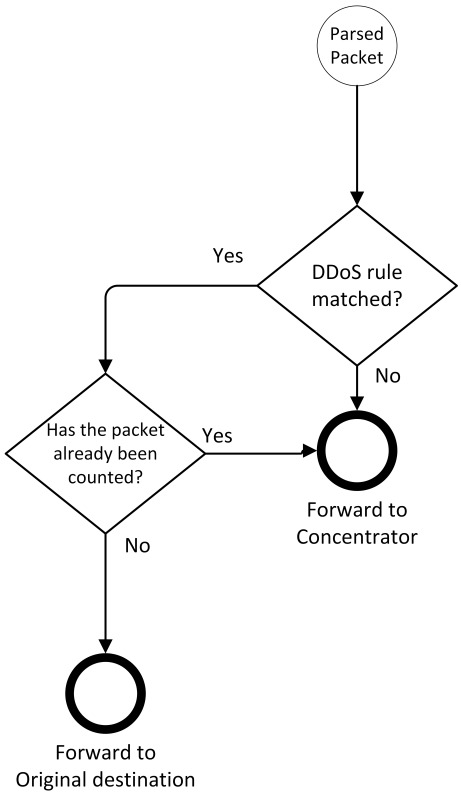

Figure 6.3. High level functionality of switches without a state-replica.

in listing 6.3 with *set_egress_table* setting the egress port based on the destination IP of the packet and only if the packet is uncounted allowing *detect_potential_ddos_table* to overwrite this decision. For the sake of brevity only a simplified version of the code is presented.

```
table detect_potential_ddos_table {
  reads
    ipv4.dstAddr : lpm;
  actions {
    forward_to_concentrator;
    nop;
  }
}

control ingress {
  apply(set_egress_table);
  if(ipv4.diffserv == UNCOUNTED_PKT)
    apply(detect_potential_ddos_table);
}
```

Listing 6.3. Simplified FvSW description in P4.

Counting virtual switch behavior

The behavior of CvSWs follows a more intricate pattern and provides the same functionalities as FvSW in addition to the ones required to perform estimations and replication functions. The simplified version of the processing pipeline of the ingress stage is represented in listing 6.4. For the sake of brevity the rest of the code such as tables declaration, definition of actions, registers and metadata, etc... is not presented.

CvSWs operate in two operation states, namely *Normal* and *Under Attack*. The two states are defined as a simple FSM with one possible transition, *Normal*→*Under Attack*, which is performed when the DDoS is detected. The inverse transition is not implemented as it is assumed that the controller will reset the state of all switches back to *Normal* whenever the DDoS attack is not present anymore. The two states are implemented via a simple 1-bit register, namely *state_register* which is set to 1 when the transition occurs.

Most of the logical conditions present in Figure 6.2 are moved directly inside the match-action tables in the form of matching fields. Only the conditions which involve inequalities are left in the code under the form of classical *if* conditions since there does not exist an easy and efficient way to translate them into match-action rules.

In order to provide greater efficiency and processing speed most of custom metadata (*cm*) are defined in the *detect_potential_ddos_table* and only if the processed packet belongs to the flows for which a DDoS detection rule is defined. This approach allows to concentrate all memory extensive operations in one table while avoiding access to stateful elements in other processing stages and consequently reduces the overall complexity of the prototype.

The CvSW's implementation is divided in 4 fundamental blocks:

1. **Destination-based forwarding (line 2)**: Similarly to the case of FvSWs a CvSW sets the egress port based on the destination IP of the packet in order to allow classic forwarding. The table is missed if the packet does not have a valid IP header and no operation is performed.

2. **DDoS prevention (lines 3-10)**: This block comprises all the already discussed functions needed to manage the DDoS prevention mechanism. Lines 5-11 are executed only if the packet belongs to the set of the flows directed towards one of SCs, i.e. if there exists an entry in *detect_potential_ddos_table* containing as a match field the destination IP of the packet that is being processed. In order to avoid double counting the table matches also the value of DSCP field which allows to discriminate between already counted and still uncounted packets leading to a "hit" only if both the conditions on the destination IP, the DSCP value alongside with the operation state equals to *Normal* are satisfied.

   Lines 5-7 are responsible of managing the evolution of the jumping window and they behave in the same manner as depicted in listing 6.2, with *shift_estimate_table* being accessed only if a shift in the buffer is required, i.e if the time at which the packet have been received (*ingress_global_timestamp*) is greater that the time at which a buffer shift must occur. Similarly *update_estimate_table* is responsible of performing the counting, aggregating the values of all slots of the buffer into a

local estimate alongside with the estimates replicated from other CvSWs and finally returning the value under the form of a custom metadata (*aggregate_estimate*).

Lines 8-9 are responsible of providing reaction based on the value of the estimate. Line 8 summarizes the expression (6.1) while line 9 provides an action that performs the transition of the state of the switch from *Normal* to *Under Attack*.

3. **State-Update generation (lines 12-13)**: this block provides the means for state-update generation. The update packet is generated only if the value-delta since the last update *delta_since_update* is greater than the predefined constant value *UPDATE_INTERVAL*. For performance purposes the value-delta is evaluated in *update_estimate_table* which is executed only if *detect_potential_ddos_table* hits a match. Because of this structure any packet that does not hit *detect_potential_ddos_table* will lead to *delta_since_update*= 0 and since by definition *UPDATE_INTERVAL*> 0 no update packet will be generated. The *update_build_table* table follows the exact scheme described in 6.3.2.

4. **State-Update processing and propagation (lines 15-18)**: The last processing block involves the detection and processing of update packets. The block is accessed only if the packet contains a valid update header (line 15) and depending on the information included in the header performs forwarding and/or local state-update. *rx_update_table* checks the update ID and the ID of the update originator and requires update ID to be among those included in the current switch and similarly requires switch ID to be different from the ID of the current switch. This mechanism allows to store only updates for states that are present in the current switch and that are not originated from the local switch.

The last match-action table, namely *forward_update_table* is executed independently from the ID of the update originator and is responsible of setting the egress spec based on the state ID. The egress spec is defined as a multicast queue which contains the physical ports of all switches belonging to the replication tree. For all the packets leaving the multicast tree the egress CB performs an additional check on the destination physical port. If the egress physical port is equal to the original ingress port the packet is not emitted and dropped. This mechanism allows to avoid loops while achieving low memory utilization by maintaining only 1 multicast queue and 1 match-action rule per state.

```
control ingress {
  apply(egress_table);
  apply(detect_potential_ddos_table)
    hit{
      if(intrinsic_metadata.ingress_global_timestamp > cm.t_next_update)
        apply(shift_estimate_table);
      apply(update_estimate_table);
      if(cm.aggregate_estimate > DDOS_THRESHOLD)
        apply(ddos_detected_table);
  }

  if(cm.delta_since_update > UPDATE_INTERVAL)
    apply(update_build_table);

  if(valid(update_header)){
    apply(rx_upd_table);
    apply(forward_update_table);
  }
}
```

Listing 6.4. Simplified CvSW ingress stage description in P4.

6.4 Validation

The validation of the prototype involved a high level behavioral analysis, evaluation the amount of introduced overhead due to the synchronization traffic, estimation of the maximum drift in state estimates and finally analysis of reduction of overhead due to data-traffic convergence to the state-replicas in respect to a single state.

6.4.1 Test conditions

Topology and involved entities

The topology depicted in Figure 6.1 was implemented in Mininet with each link supporting data rates up to 100Mbps. ASs and SCs are modeled by using single virtual hosts with respectively assigned IP addresses of 10.0.1.x and 10.0.0.x, where x is the number of AS/SC. Both the border routers and the switches are based on *Simple Switch* architecture with different configurations depending on the employed scenario.

Traffic specifications

In order to simulate the background traffic each AS generates CBR traffic towards each SC at a rate of 25 pkts/s with a total aggregate packet generation rate of 100 pkts/s. This traffic pattern is generated for 20 seconds after which the rate is doubled, thus reaching an aggregate of 200 pkts/s generated at each AS. The experiment stops after 20 seconds have passed since the begging of the generation of high traffic.

The choice of such small data rates resides in the fact that in an emulated environment which requires the emulation of 8 vSWs and 8 virtual hosts the intrinsic limitations related to the resource management and in particular to the CPU scheduling of the host start play a significant role. Under high load even if the CPU is able to schedule the generation of CBR traffic it starts to introduce random delays at each processing stage of the transiting

packets, thus introduction of random delays at each vSW which ultimately leads to the complete disruption of the original traffic pattern. The use of small data rates allows to reduce the total host's resource requirements and consequently scale down this effect.

Replica placement routing

4 scenarios are considered for the placement of state-replicas:

- **No state-replicas**: A scenario of a network where no DDoS prevention scheme is employed and instead packets are simply forwarded based on their destination and using the shortest path.
- **1 state-replica**: Only 1 replica is placed in SW1. This scenario is used as a pure SNAP scenario. The use of 1 replica only allows to define a margin of improvement due to state-replication when it is later compared to the other cases. All data traffic converges to SW1 and is then routed to the destination following the shortest path. In the case of a destination located at the antipode in respect to SW1 the traffic is routed following the clockwise path.
- **2 state-replicas**: 1 replica is placed in SW1 and another one in SW3. The data packets coming from ASs are routed towards the nearest replica and then exploit the same routing as in the case of only 1 state-replica. In the case of a destination located at the antipode in respect to SW1 or SW3 the traffic is routed following the clockwise path.
- **4 state-replicas**: A replica is placed at each border router. Since state-replicas are located at each entrance of the network no convergence to state-replicas is required. The routing follows the same scheme as in the case of no state-replicas. The synchronization traffic is routed clockwise.

6.4.2 Numerical results

State consistency

Simple Switch architecture provides the APIs necessary to perform read and write operations. These APIs were exploited in order to obtain the information related to the aggregate state information $S(t)$ which is depicted in Figure 6.4 where:

- s_L: Locally measured traffic.
- s_R: Estimate of the traffic measured by other replicas.
- s_G: Estimate of the global variable based on both s_L and s_R

As mentioned in Section 6.2.6 a random delay proportional to the load on the system is introduced whenever a call to the APIs is performed. The introduction of this delay leads to asynchronous reads from the two replicas which has a beneficial effect of removing any temporal correlation among the reads but at the same time does not allow to provide the

precise instant of the call execution. This leads to a result which is related to a time instant comprised between the initial API call and the instant at which a response with the value is returned but no guarantee on the exact time instant is given. This effect introduces an apparent impression of big inconsistencies among the two replicas which, when combined with the delays due to host's resources, becomes even more accentuated in the presence of high traffic (i.e. seconds 25-45).

s_G at both SW1 and SW3 present sufficient consistency in the estimates allowing to easily define the instant of the DDoS. The obtained results overall provide strong validation for the implementation of the replication scheme.

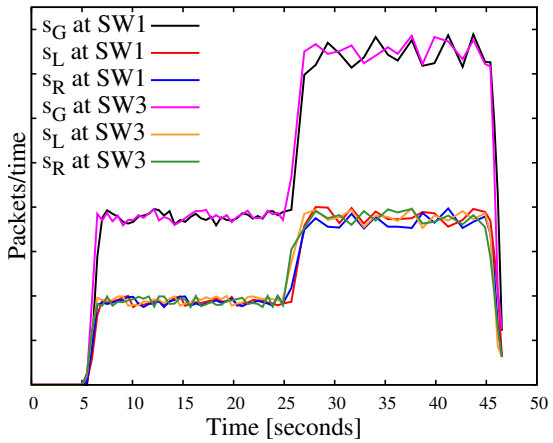

Figure 6.4. Consistency among state-replicas in the scenario of 2 replicas.

Data and synchronization overhead

Figure 6.5 provides the behavior of each link's utilization while Table 6.4.2 represents the detailed occupation for each link. For the case of 1 state-copy the load is not uniformly distributed and highly unbalanced due to the requirement of convergence of all traffic to SW1. This effect is greatly reduced in the presence of 2 state-replicas which provides a reduction by a factor of 1.6 of the total traffic in the network. Similarly in the case of 4 state-replicas the total traffic is further reduced to its minimum providing an additional reduction of 20% in respect to the case of 2 state-replicas. On the other hand due to the increase in the number of replicas the fraction of synchronization packets increases from 14% in the case of 2 state-replicas to 24% in the case of 4 state-replicas. The presented results however depict the overhead in terms of packets and not bits. If the data packets are assumed of 1500 bytes while the synchronization packets of 64 bytes (i.e. the minimum Ethernet frame size) the overhead of the synchronization traffic becomes negligible. This behavior leads in the case of 4 replicas to the possibility of implementing a SNAP-like DDoS detection system with no traffic overhead in respect to the case not employing any detection system.

	Number of transiting packets per scenario.							
Number of Replicas	0		1		2		4	
Traffic type	Data	Sync	Data	Sync	Data	Sync	Data	Sync
R1-SW1	4036	0	14122	0	6054	598	4036	1196
SW1-R2	4036	0	14188	0	6054	600	4036	1200
R2-SW2	4036	0	10082	0	4036	600	4036	1200
SW2-R3	4036	0	6050	0	4036	600	4036	1200
R3-SW3	4036	0	2014	0	6054	600	4036	1200
SW3-R4	4036	0	2018	0	6054	598	4036	1196
R4-SW4	4036	0	6054	0	4036	598	4036	1196
SW4-R1	4036	0	10082	0	4036	598	4036	1196

Table 6.1. Link occupation for data and synchronization traffic in the case of 1,2,4 copies for global state in P4 implementation.

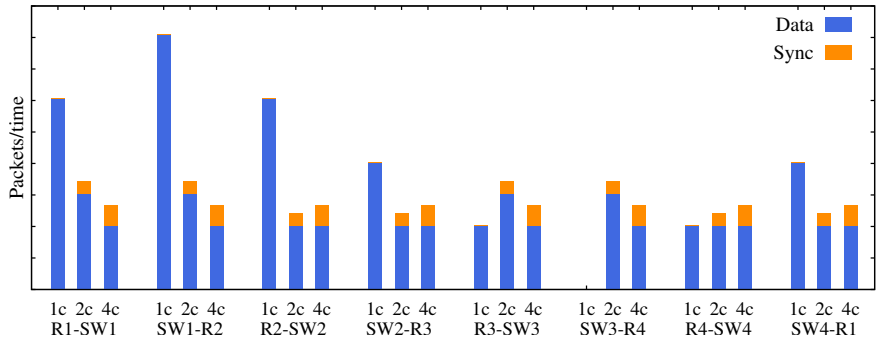

Figure 6.5. Link occupation for data and synchronization traffic in the case of 1,2,4 copies for global state in P4 implementation.

6.4.3 OPP implementation

Similar results have been obtained employing OPP for what regards the total data and synchronization traffic overhead even if the implementation exploited a time-delta triggering scheme instead of a value-delta one. The results depicted in Figure 6.6 present different balancing of the load among the link in respect ot the results obtained with P4 due to different routing policies. In the OPP implementation, instead of exploiting clockwise or counterclockwise convergence in case of antipodal destination the routing is performed based on both source and destination IPs, thus allowing to achieve greater balance. This choice was purposefully avoided in the P4 implementation in order ot provide a scenario in which the flow tables are optimized in order to achieve the minimum amount of required entries.

Nonetheless the overall evaluation of the overhead provides the same results as in the case of P4 with a slight reduction in the fraction of synchronization packets (11% for the case of 2 state-replicas and 23% for the case of 4 state-replicas).

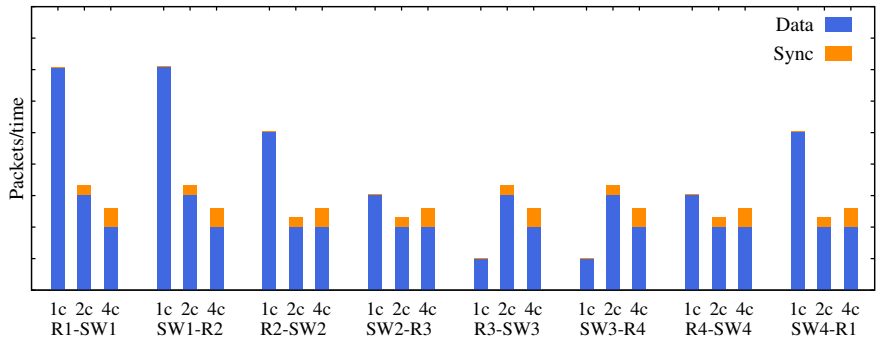

Figure 6.6. Link occupation for data and synchronization traffic in the case of 1,2,4 copies for global state in OPP implementation.

Chapter 7

Conclusion and future works

7.1 Related Works

Stateful NetKAT [45] is a framework for network programming which similarly to SNAP has the objective of providing a programming language for the development of network programs. Although NetKAT provides a native support for state replication with consistent updates the actual replication can be performed only on the nodes positioned at the edges of the network.

In Swing State [44] the authors propose a technique for the consistent migration of a state entirely in the dataplane. Swing State guarantees that the migration will not disrupt the normal functionality of the network and will provide a transparent migration without introducing any error on the state. Although the proposed technique provides the means of moving the state in a fluid way inside the network it requires a permanent rerouting of all involved flows towards the new state location.

E-State [51] provides a complete framework for the management of replicated VNFs however it relies on on-demand pulls of replicas' states. The proposed scheme does not guarantee strict consistency unless the pull is performed with a sufficiently small periodicity. Moreover the adopted mechanism is not suitable for large networks since in the case of significant number of replicas the pull action may cause large incast leading to congestion and consequently loss of state-updates.

In [20] the authors leverage on the functionalities of programmable dataplanes in order to provide better nework support for Paxos replication mechanism. NetPaxos is proposed as an enhancement of the classical Paxos algorithm by offloading some of the functionalities of Paxos inside the dataplane. In particular the functionalities of Acceptors and Serializers are implemented in P4 directly inside the forwarding devices, leading to one order of magnitude smaller latencies. The entire scheme however relies on the absence of packet reordering inside the network in order to guarantee correct functionality.

7.2 Future Works

7.2.1 Extensions of State Replication

Although for the purpose of this work it was considered a simple application exploiting replicated states, it is possible to extend the proposed mechanisms to a broader set of applications outside of the SNAP framework. Indeed state replication can be exploited for a variety of control mechanisms inside the network such as replication of buffer occupancies in order to provide controller-independent, congestion-aware dynamic routing, reactive load balancing and overall better reactiveness to global events. Analogously with replicated states it is possible to provide more flexibility over the control of split flows such as those exploiting MPTCP, ECMP, etc... without imposing strict routing conditions and thus preserving the beneficial effect of multi-path routing.

7.2.2 Definition of advanced replication schemes.

The limitation of strongly consistent replication schemes have been exhaustively discussed and motivated throughout this work. It is still true that there exist replication schemes such as Google Spanner which was briefly discussed in 4.6 which rely on accurate time-stamping instead of expensive communication based on 3PC.

Recent works such as [42] provide means of achieving sub μs accuracy in clock synchronization with virtually no additional cost in terms of communication overhead. These advancements lead to the possibility of mitigating issues related to packet reordering in NetPaxos while additionally opening opportunities for providing support for more robust replication schemes for dataplanes.

7.2.3 Embedding application layer functionalities in the dataplane

NetPaxos was among the first proposed works which put an effort on exploiting stateful dataplanes in order to provide acceleration for application layer functionalities. Due to the novelty of the field the which type of application may benefit by network acceleration is still an open question.

7.2.4 Extensions of the ILP formulation

As anticipated previously the proposed ILP formulation shares big similarities with VNE problem. Due to the fact that none of the proposed VNE formulation provide means for state replication it is possible to extend the proposed formulation in order to support VNF embedding alongside with state placement. By performing such an extension it is possible to provide a broader framework capable of creating heterogeneous networks composed by both VNFs and dataplane acceleration.

7.3 Conclusion

In this work is presented an analytical model targeting the optimal distribution of replicated states for programmable stateful dataplanes. The model provides richer possibilities

in terms of network programming functionalities which allow to achieve better network resources' utilization when embedding network programs in the dataplane.

Although the analytical model itself provides a leading contribution to the field, the major contribution of this work resides in the feasibility analysis of implementing a replication scheme entirely in the dataplane and the consequent design and implementation of a functional prototype.

The feasibility analysis requires a robust understanding of the state of the art replication mechanisms and their suitability in the case of stateful dataplanes. For this reason this work contains a critique towards replication schemes currently implemented in production environments alongside with the proposed state of art alternatives. A major portion of the work is dedicated towards the design of a suitable replication mechanism by considering limited capabilities of the commercially available forwarding devices.

Finally the evaluation of a functional prototype is performed. The results of the evaluation show the net superiority of the proposed model in terms of resource utilization while still providing bounded approximation errors of the global state.

Bibliography

[1] Peter A Alsberg and John D Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering*, pages 562–570. IEEE Computer Society Press, 1976.

[2] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *ACM SIGCOMM*, pages 29–43, New York, NY, USA, 2016.

[3] Muhammad Atif. Analysis and verification of two-phase commit & three-phase commit protocols. In *Emerging Technologies, 2009. ICET 2009. International Conference on*, pages 326–331. IEEE, 2009.

[4] Peter Bailis and Kyle Kingsbury. The network is reliable. *Queue*, 12(7):20, 2014.

[5] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.

[6] Giuseppe Bianchi, Marco Bonola, Salvatore Pontarelli, Davide Sanvito, Antonio Capone, and Carmelo Cascone. Open packet processor: a programmable architecture for wire speed platform-independent stateful in-network processing. *arXiv preprint arXiv:1605.01977*, 2016.

[7] Andrea Bianco, Paolo Giaccone, Abubarak Siddique Muqaddas, German Sviridov, Janvi Palan, Marco Bonola, Angelo Tulumello, and Giuseppe Bianchi. State replication for programmable stateful data planes in SDN. 2017.

[8] P Biondi. Scapy, a powerful interactive packet manipulation program, 2010.

[9] Jonas BonÃ©r. The road to akka cluster and beyond. 2013.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[11] Juan Felipe Botero, Xavier Hesselbach, Michael Duelli, Daniel Schlosser, Andreas Fischer, and Hermann De Meer. Energy efficient virtual network embedding. *IEEE Communications Letters*, 16(5):756–759, 2012.

[12] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[13] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[14] Antonio Capone, Carmelo Cascone, Alessandro QT Nguyen, and Brunilde Sanso. Detour planning for fast and reliable failure recovery in sdn with openstate. In *Design*

of Reliable Communication Networks (DRCN), 2015 11th International Conference on the, pages 25–32. IEEE, 2015.

[15] IBM Knowledge center. Assembler language. "https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.asma400/asmr102112.htm", 1992.

[16] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[17] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Chunfeng Cui, Hui Deng, et al. Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow World Congress*, pages 22–24, 2012.

[18] The P4 Language Consortium. P4_16 language specification. https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf, May 2017.

[19] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[20] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 5. ACM, 2015.

[21] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the paxos algorithm. *Theoretical Computer Science*, 243(1-2):35–91, 2000.

[22] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[23] Andreas Eckner. Algorithms for unevenly-spaced time series: Moving averages and other rolling operators. In *Working Paper*, 2012.

[24] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of time table and multi-commodity flow problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 184–193. IEEE, 1975.

[25] Dino Farinacci, C Liu, S Deering, D Estrin, M Handley, Van Jacobson, L Wei, Puneet Sharma, David Thaler, and A Helmy. Protocol independent multicast-sparse mode (pim-sm): Protocol specification. 1998.

[26] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann De Meer, and Xavier Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys & Tutorials*, 15(4):1888–1906, 2013.

[27] Open Networking Foundation. Openflow switch specification, version 1.5.1, 2015.

[28] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[29] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[30] Fred Glover. Improved linear integer programming formulations of nonlinear integer problems. *Management Science*, 22(4):455–460, 1975.

[31] Xinjie Guan, Baek-Young Choi, and Sejun Song. Reliability and scalability issues in software defined network frameworks. In *Research and Educational Experiment Workshop (GREE), 2013 Second GENI*, pages 102–103. IEEE, 2013.

[32] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[33] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9. Boston, MA, USA, 2010.

[34] Barefoot Networks Inc. The world's fastest and most programmable networks. https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf, 2016.

[35] Johannes Inführ and Günther Raidl. Introducing the virtual network mapping problem with delay, routing and location constraints. *Network optimization*, pages 105–117, 2011.

[36] Van Jacobson, Craig Leres, and Steven McCanne. Tcpdump/libpcap. *Retrieved from http://www. tcpdump. org*, 2005.

[37] Kostas Katrinis, Guohui Wang, and Laurent Schares. Sdn control for hybrid ocs/electrical datacenter networks: An enabler or just a convenience? In *Photonics Society Summer Topical Meeting Series, 2013 IEEE*, pages 242–243. IEEE, 2013.

[38] Hyojoon Kim, Mike Schlansker, Jose Renato Santos, Jean Tourrilhes, Yoshio Turner, and Nick Feamster. Coronet: Fault tolerance for software defined networks. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–2. IEEE, 2012.

[39] Martin Kleppmann. A critique of the cap theorem. *arXiv preprint arXiv:1509.05393*, 2015.

[40] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[41] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[42] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. *Globally Synchronized Time via Datacenter Networks*, pages 454–â467. ACM Press, 2016.

[43] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual consistency. *Communications of the ACM*, 57(5):61–68, 2014.

[44] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. Swing state: Consistent updates for stateful and programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 115–121. ACM, 2017.

[45] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černỳ. Event-driven network programming. In *ACM SIGPLAN Notices*, volume 51, pages 369–385. ACM, 2016.

[46] Nick McKeown, 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(2):69–74, 2008.

[47] Sevil Mehraghdam, Matthias Keller, and Holger Karl. Specifying and placing chains of virtual network functions. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 7–13. IEEE, 2014.

[48] Hein Meling and Leander Jehl. Tutorial summary: Paxos explained from scratch. In *International Conference On Principles Of Distributed Systems*, pages 1–10. Springer, 2013.

[49] Abubakar Siddique Muqaddas, Andrea Bianco, Paolo Giaccone, and Guido Maier. Inter-controller traffic in onos clusters for sdn networks. In *Communications (ICC), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.

[50] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[51] Manuel Peuster and Holger Karl. E-state: Distributed state management in elastic network function deployments. In *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*, pages 6–10. IEEE, 2016.

[52] Andrew Prunicki. Apache thrift, 2009.

[53] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of convergent and commutative replicated data types*. PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[54] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[55] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176. ACM, 2017.

[56] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142. ACM, 1981.

[57] Mininet Team. Mininet: An instant virtual network on your laptop (or other pc), 2012.

[58] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The tcp/udp bandwidth measurement tool. *htt p://dast. nlanr. net/Projects*, 2005.

[59] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[60] Guoxi Wang and Jianfeng Tang. The nosql principles and basic application of cassandra model. In *Computer Science & Service System (CSSS), 2012 International Conference on*, pages 1332–1335. IEEE, 2012.

[61] Haifeng Yu and Amin Vahdat. Building replicated internet services using tact: A toolkit for tunable availability and consistency tradeoffs. In *Advanced Issues of E-Commerce and Web-Based Information Systems, 2000. WECWIS 2000. Second International Workshop on*, pages 75–84. IEEE, 2000.

[62] Saman Taghavi Zargar, James Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *IEEE communications surveys & tutorials*, 15(4):2046–2069, 2013.

[63] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, pages 75–87. ACM, 2015.

[64] Xian Zhang, Chris Phillips, and Xiuzhong Chen. An overlay mapping model for achieving enhanced qos and resilience performance. In *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2011 3rd International Congress on*, pages 1–7. IEEE, 2011.