Efficient Data Management and Policy Composition for Software-defined Networking

Dissertation
for the award of the degree

”Doctor rerum naturalium” (Dr. rer. nat.)
of the Georg-August-Universität Göttingen

within the doctoral program in Computer Science (PCS)
Of the Georg-August University School of Science (GAUSS)

submitted by

Osamah Barakat
from Sana’a, Yemen

Göttingen
in July 2019

Thesis Committee:	Prof. Dr. Xiaoming Fu, Georg-August-Universität Göttingen
	Prof. Dr. Ramin Yahyapour, Georg-August-Universität Göttingen
	PD. Dr. Mayutan Arumaithurai, Georg-August-Universität Göttingen

Examination Board:

Reviewer:	Prof. Dr. Xiaoming Fu, Georg-August-Universität Göttingen
Other Reviewers:	Prof. Dr. Tobias Hoßfeld, Julius-Maximilians-Universität Würzburg
Further Members of the Examination Board:	Prof. Dr. Ramin Yahyapour, Georg-August-Universität Göttingen Prof. Dr. Oliver Hohlfeld, Brandenburgische Technische Universität Prof. Dr. Jens Grabowski, Georg-August-Universität Göttingen Prof. Dr. Marcus Baum, Georg-August-Universität Göttingen PD. Dr. Mayutan Arumaithurai, Georg-August-Universität Göttingen

Date of Oral Examination: 08. July 2019

Abstract

Network softwarization changes the way how should networks be managed. Introducing Software-defined Networking in the last decade helps network administrators focus on network management and write optimized applications that control network behavior. Network administrators communicate with a network controller through an interface named *northbound interface*. This interface and any abstract build on it should be designed to enforce the ease of the network management to align with the primary purpose of Software-defined Networking. The performance of these abstractions is affected by the data organization and software libraries used to deliver northbound interface services to end users.

We start with *Gavel*, an SDN controller that at its heart facilitates a plain data representation based on a graph database. In Software-defined Networking, high-level abstractions typically offer a useful means to avoids writing network applications and policies on lower levels. However, abstractions are typically developed for a specific use case, which in turn results in an abundance of existing abstractions for different networking tasks. As a consequence orchestrating these abstractions to implement a standard network policy becomes an arduous task. To address this challenge, plain data representations of the network and its control infrastructure have been proposed recently, which offer programmable ad-hoc abstractions to administrators. However, these frameworks suffer from quite complex programming requirements and impractical performance in terms of latency, as they are based on relational database engines.

By exploiting the native graph support of the database engine, *Gavel* significantly eases application and policy writing. Additionally, we show by experimental evaluation of several typical applications on multiple different topologies that *Gavel* offers significant performance improvements over state-of-the-art solutions.

In the second part of the thesis, we present *Busoni*, a framework that we build on *Gavel* to provide needed libraries to manage policies on top of Segment Routing. Segment Routing is a promising solution to support services like Traffic Engineering, Service Function Chaining and Virtual Private Networks. It is a source routing based networking architecture that provides an opportunity to include a list of instructions called *segments* in the packet headers. The segments may allow the inclusion of detours for responding to Traffic Engineering needs or Service Function Chains implementations. Even though there is an increasing interest in enhancing and adopting Segment Routing, the administrators are still burdened with the task of manually write and maintain the segment lists. Such type of management

presents several challenges ranging from error-prone configurations to increased response time for network updates.

To address these challenges, we propose *Busoni* that automates and simplifies the process of segments lists management. Additionally, we also provide programming tools to compose and manage Segment Routing policies that operate efficiently even under multi-tenancy environments. Using different use cases we show the programming capabilities offered by our framework. With experimental evaluation, we demonstrate the scalability of our platform and the improvements achieved in response time for dynamic network events.

This thesis investigates the role of efficient data management and policy composition in Software-defined Networking frameworks. It sheds light on the importance of data representation and how it affects the performance of network application. It also presents one of the first frameworks that manage network policies in the new network technology (*i.e.,* Segment Routing). The work presented in this thesis has been implemented, evaluated, and published as an extension to the state-of-the-art knowledge in the related field.

Acknowledgements

In the name of Allah, the Most Gracious and the Most Merciful.

With great pleasure, I would like to acknowledge and wholeheartedly thank all those who have inspired, lead me and been active part of my unforgettable journey of PhD. All praises to Allah for the strengths and His blessing in completing this thesis. I would like to thank my PhD advisers sincerely: Prof. Dr. Xiaoming Fu, Prof. Dr. Ramin Yahyapour, Dr. David Koll, and Dr. Mayutan Arumaithurai, whose support, expertise, continuous guidance, encouragement, and patience has enabled me to author my PhD thesis.

Prof. Dr. Xiaoming Fu: I'm extremely grateful for giving me an opportunity to pursue PhD under your guidance. I thank you for all the support, freedom and opportunities you let me to explore and pursue diverse research topics and to visit top research conferences. Your technical guidance and lessons including the art of communication and networking have had an enormous impact on me. I am immensely grateful for the support and encouragement I have received from you throughout my PhD.

Prof. Dr. Ramin Yahyapour: I would like to express my gratitude to all support and feedback I received during my PhD especially those in the thesis committee meetings.

Dr. David Koll and Dr. Mayutan Arimaithurai: I am lucky to have both of you as advisers. I am thankful for your efforts in teaching 'SDN Block course' during my first semester. The course inspired me the first idea of this thesis. You have been more a friend than just my mentor, not just meticulously planning the course of my work, but consistently motivating and guiding at every step of my PhD.

I am deeply grateful to Prof. Stefano Salsano who kindly suggested to me the idea to enter the world of Segment Routing. Your talk and discussion with me helped me step by step to build the second part of this thesis. Although your time schedule is busy with commitments, you always find a time to set and discuss with me. Your invitation to visit Rome is a remarkable page in my PhD journey.

I am also obliged to my thesis defense committee members: Prof. Dr. Tobias Hoßfeld, Prof. Dr. Oliver Hohlfeld, Prof. Dr. Jans Grabowski, and Prof. Marcus Baum. Their comments and suggestions have greatly improved the thesis.

I thank also Dr. Pier Luigi Ventre, who helped during the Segment Routing related project. I'm extremely grateful to have worked with you, I have learned a lot from you.

I want to thank also the 'German Academic Exchange Service' for their continued support during my PhD thesis. Thank you for your help during my first days in Germany.

I am grateful to my former and current colleagues at the Computer Networks Group at the University of Göttingen, especially Dr. Sameer Kulkarni, Sripriya S. Adhatarao, Dr. Abhinandan S. Prasad, Dr. Hong Huang, Jacopo De Benedetto, Shichang Ding, Bo Zhao, and Dr. Yali Yuan, whose feedback at different stages has contributed to the quality of this thesis.

I am equally thankful and indebted to Federica Poltronieri, Annette Kadziora, Gunnar Krull, Tina Bockler, Carmen Scherbaum, Heike Jachinke and all the staff who have been of great help and support in different matters of need.

I would also like to thank the City and the University of Göttingen for providing such a wonderful and serene atmosphere blend with excellent research opportunities.

Last but definitely not least, I want to thank my parents Dr. Lutf and Belques Barakat, my wife Rehab Aldhabbi, and my children Ala and Aseel, and my siblings for their never-ending support. Without them, this thesis would not have been written in the first place.

Copyright Notice

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of University of Göttingen's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Contents

Table of Contents viii

List of Figures xii

List of Tables xiv

Acronyms xv

1. Introduction 1

1.1. Motivation 1
1.2. High Level Research Problems 3
1.3. Thesis Challenges 5
1.3.1. Performance 5
1.3.2. Portability 6
1.3.3. Expressiveness and Automation 7
1.4. Thesis Contributions 7
1.4.1. Performance 8
1.4.2. Portability 9
1.4.3. Expressiveness and Automation 9
1.5. Thesis Outline 9

2. Background 11

2.1. Network Softwarization 11
2.1.1. SDN 11
2.1.2. NFV 12
2.2. Segment Routing 13
2.2.1. Overview 13
2.2.2. Segment Routing on IPv6: SRv6 14
2.2.3. SRv6 Programming 15
2.3. Graph Database 17

I. A Fast and Easy-to-Use Plain Data Representation for Software defined Networking 19

3. Problem Statement 21
3.1. Introduction 21
3.2. Challenges in SDN Northbound Abstractions 22

4. Related Work 25
4.1. Abstractions 25
4.2. Databases in SDN Controllers 26
4.3. Use of Graph Modeling in Networks 27

5. Software-defined network control with graph databases: Gavel 29
5.1. Introduction 29
5.2. The Case for the Use of Graph Databases 30
5.2.1. Data Representations 30
5.2.2. Drawbacks of Relational Databases 31
5.2.3. Advantages of Graph Databases 31
5.3. Gavel Architecture and Design Choices 34
5.3.1. Network Model 34
5.3.2. Selecting a Graph Database Engine for Gavel 35
5.3.3. Native Graph Functions and Cypher 36
5.3.4. Gavel Architecture 37
5.4. Gavel and Network Application Programming 38
5.4.1. Routing 38
5.4.2. Access Control Firewall 40
5.4.3. Load-Balancer 40
5.4.4. Service Function Chaining 41
5.4.5. Network Slicing 42
5.4.6. Summary 43
5.5. Evaluation 43
5.5.1. Methodology 43
5.5.2. Gavel's Applications 44
5.5.3. Writing Network Applications on Gavel 51

6. Future Prospects 53
6.1. Applicability of Gavel with Other SDN Environments 53
6.1.1. Gavel and SR 53
6.1.2. Gavel and other Northbound Interfaces 53
6.2. Current Limitations and Prospects of Extensions 54

II. Addressing Northbound Interface Challenges in IPv6 Segment Routing 55

7. Problem Statement 57
7.1. Introduction 57
7.2. Challenges in Segment Routing Policy Composition 57

8. Related Work 59
8.1. Segment Routing on IPv6 59
8.2. Northbound Interfaces in SDN 60

9. A Northbound Interface for IPv6 Segment Routing: Busoni 61
9.1. Introduction 61
9.2. Requirements for Segment Routing Policy Framework and Target Scenarios 62
9.3. Busoni Architecture 63
9.3.1. Overall Architecture 63
9.3.2. API Policies Composing 66
9.3.3. Encoding Path Nodes as Segments 67
9.3.4. Busoni in Action 68
9.3.5. Data Store 69
9.3.6. Responding to Network Dynamics 69
9.4. Use Cases 70
9.4.1. Basic policy with SFC 72
9.4.2. Overlay with QoS Policy 72
9.4.3. Responding to a VNF Migration 73
9.5. Evaluation and Discussion 74
9.5.1. Implementation and Lab Setup 74
9.5.2. Scalability 74
9.5.3. Reactivity to Network Dynamics 78

10. Future Prospects 81
10.1. Applicability of Busoni in MPLS-SR Environment 81
10.2. Applicability of Busoni in SRv6 on non-Linux Routers Environment 82
10.3. Current Limitations and Prospects of Extensions 82
10.3.1. Flow Specifications 82
10.3.2. Rules Conflicts 82
10.3.3. Complex Network Dynamics 83

11. Conclusion 85
11.1. Dissertation Summary 85
11.2. Thesis Impact 86

III. Appendix 89

A. Concepts and Definition of Related Terms 93

B. Gavel Internals 95
B.1. Representation of Network Topologies in Graph Database 95
B.2. Comparison of Routing Application Implementation between Gavel and Ravel 96
B.3. ASR Algorithm Implementation 96

C. Busoni Algorithms and Work flow 99
C.1. Busoni's work flow 100
C.2. SRtypes Code Snippets 101
C.3. Path finding in Busoni Code Snippets 105

Bibliography 111

List of Figures

1.1. High-level Research Problems associated with the northbound interface in SDN . . . 4
1.2. The position of *Gavel* and *Busoni*, the thesis contributions, in the SDN architecture . . . 8

2.1. SDN architecture . . . 12
2.2. SRv6 Extension Header . . . 14
2.3. Representing SR network command using IPv6 . . . 16
2.4. An example of declaring segments list and SRv6 behaviors . . . 16
2.5. An example of an SRv6 network topology . . . 17
2.6. An example of *iproute2* command to attach the segments list to packets . . . 17
2.7. The chapters of this thesis organized as a GDB model . . . 18

5.1. Database management systems as SDN controllers . . . 32
5.2. Basic network topology model in Gavel . . . 34
5.3. Cypher snippets to add two switches and their connections to each other to the network topology . . . 36
5.4. Gavel interaction with forwarding plane to learn the topology and install the new forwarding rules . . . 38
5.5. Cypher code snippets . . . 39
5.6. ARS algorithm [1] chooses the shortest path for every next network function until it reaches egress point . . . 41
5.7. Cypher sample to find a path between two hosts within a single slice l . . . 42
5.8. A comparison of the latency induced on different topologies by routing application in Gavel and Ravel, respectively. . . . 45
5.9. A Comparison of the latency induced on different topologies by routing through different function chains in Gavel the lower bound for Ravel . . . 48
5.10. A comparison of routing delay in combination with firewall routines for blocking hosts (BH) and unblocking hosts (UbH) in both Ravel and Gavel in k-ary FatTree networks with k=16,32,64 . . . 49
5.11. A comparison of different delay time induced by routing application with(1-9)/without(0) slices in different topologies . . . 50

9.1. The position of Busoni framework in a SRv6 network 64
9.2. Major software subsystems of Busoni 65
9.3. Using `Match` class to define source and destination addresses 66
9.4. Using `eval` function to add custom packet handling 68
9.5. Illustration of different use cases including initial state of the network topology 71
9.6. Instantiating match object for the use cases 72
9.7. Instantiating an object for the first use case 72
9.8. Instantiating an object for the second use case 73
9.9. Compilation time for different number of policies 77
9.10. Response time for events affect batches of policies 78

B.1. Exemplary specification of two switches (left) and an edge between these switches (right) in Gavel's graph database. Green coloring indicates the respective endpoints of the edge 95
B.2. Peusodo code for processing a routing request in Rave [2] 96
B.3. Code to implement a routing application in Gavel 96

C.1. A flowchart of Busoni's work flow 100

List of Tables

5.1. Graph databases comparison matrix 35
5.2. Topologies used to evaluate Gavel 44

9.1. Summary of the policies used in the evaluation 75
9.2. Average compilation time (ms) and coefficient of variation (%) for every policy with incremental batch size 76
9.3. Response time (ms) (95% percentile) for different number of affected policies in the two dynamic events 79

Acronyms

The following table describes the significance of various abbreviations and acronyms used throughout the thesis. Nonstandard acronyms that are used in some places to abbreviate the names of certain white matter structures are not in this list.

dpid *OpenFlow Data Path ID*

ETSI *European Telecommunications Standards Institute*

FRR *Fast Re-Route*

ICMPv6 *Internet Control Message Protocol version 6*

GDB *Graph Database*

HMAC *Hashed Message Authentication Code*

IGP *Interior Gateway Routing Protocol*

IPv6 *Internet Protocol Version 6*

ISPs *Internet Service Providers*

MPLS *Multiple Protocol Label Switching*

NFV *Network Function Virtualization*

NF *Network Function*

OAM Operations, Administration, and Management

OSPF *Open Shortest Path First*

QoS *Quality of Service*

RDB *Relational Databases*

RDBMS *Relational Database Managmenet System*

RSVP *Resource Reservation Protocol*

SBI *Southbound Interface*

SIDs *Segment Identifiers*

SDN *Software Defined Networking*

SFC *Service Function Chaining*

SPRING *IETF Source Packet Routing in Networking*

SR *Segment Routing*

SRH *SRv6 Extension Header*

SRv6 *Segment Routing on IPv6*

TCP *Transmission Control Protocol*

TLVs *Type Length Values*

UDP *User Datagram Protocol*

VM *Virtual Machine*

VNF *Virtual Network Function*

Chapter 1

Introduction

Not having heard something is not as good as having heard it; having heard it is not as good as having seen it; having seen it is not as good as knowing it; knowing it is not as good as putting it into practice

— Xunzi: The teachings of the Ru. Xun Kuang

1.1. Motivation

Nowadays, computer networking plays a significant role in providing different technologies and services such as Microsoft, Google, and Facebook. Network-layer protocols (*e.g.,* IP and routing) and transport-layer protocols (*e.g.,* TCP) are fundamental elements for computer networking. However, IP networks are complex and challenging to manage [3]. This is evidenced by, for example, the poor utilization (40-60%) of high-cost Wide Area Network (WAN) [4] as a result of lack of coordination between the services that use the network. These services are typically implemented via an end-to-end connection, which would traverse different networking topologies and types (*e.g.,* data-centers, WAN, and carrier-grade networks). Hence, managing, monitoring, and debugging such connection is a tedious job. Another example of network management complexity is poor traffic engineering decisions that lead to locally optimum routes that are nevertheless sub-optimal globally [5] caused by the absence of a global view in the distributed routing mechanism. Moreover, to represent a desired high-level policy to govern the network behavior, network administrators have to configure each single network device using low-level commands which are mostly vendor-dependent. *Automating* these configurations and response procedures to manage vast networks is not feasible in IP networks [3].

To offer flexibility in network control, Software-defined Networking (SDN) introduces the separation of the control plane from the data plane [6]. Here, network administrators can

develop management applications to control the network behavior dynamically through predefined software interfaces (*i.e.,* Northbound Interfaces), which allows the configuration of forwarding devices in the data plane regardless of their hardware specifications. Controlling decisions in SDN as a consequence of the separation is logically centralized which provides a single-point entry for network management. This centralization simplifies the *automation* of configuration procedures. Moreover, SDN enables the notion of *network softwarization*, i.e., making the writing of portable network applications possible. Also, the introduction of Network Function Virtualization (NFV) further advances network softwarization. In an NFV-based network, functions used to process network traffic are programmed to be deployed dynamically in response to the load size and place. In SDN and NFV, decoupling the dependency between the hardware and the software offers the freedom of developing customized network applications and reusing them across different types of networks. There are various realizations of SDN in the current systems which depend on the technology that steers data-plane devices. OpenFlow-SDN is a flavor of SDN that uses OpenFlow [7] open source protocol to communicate with forwarding devices in the data-plane layer. This flavor is now used extensively in academia as it promotes open source and freedom of using software regardless of hardware providers. Another famous flavor is Segment Routing [8], introduced by industry pioneers as a practical SDN realization that takes into account legacy networking. It focuses on providing traffic engineering solutions and network programming with minimum complexity in management comparing to existing networking technologies.

Network management is an essential ongoing task that is needed to ensure a network is an operational round the clock (24/7), and all networked devices are connected and functioning as desired. Given the benefits and power of SDN, network researchers and administrators are considering to migrate existing networks to SDN. Global network providers like Google and Microsoft presented different strategies to adopt SDN in their systems [4, 9]. They showed how their management experience could be improved after applying SDN concepts in their networks. Additionally, they enforced new management policies in the data plane, allowing richer management functions.

Despite these exciting advancements in network management, they pose new challenges in writing network applications in SDN. To make optimized and efficient management decisions, network applications should be designed carefully. In practice, control plane applications in the SDN architecture are typically designed to perform one particular task in the network (e.g., routing), and network administrators usually implement these applications at a low level of abstraction in one big piece of software, which has hindered the adoption of network applications that control SDN behavior (SDN applications) [10]. To tackle this problem, researchers proposed software abstractions that take advantage of the SDN separation nature and provide development libraries to end users. These abstractions hide the details of low-level devices configurations and automate the generation of these commands based on which function call is used. Being located at an intermediate position between user

applications and the network controller, such abstractions are also called *Northbound Interfaces*. SDN developers usually employ an easy-to-write high-level language for northbound interfaces to express application policies, combine these policies into a single network policy, and then translate this policy to a lower level protocol (*i.e.,* OpenFlow).

One important consideration in network programming is the automation of the generation of configurations. As mentioned above, a typical end-to-end communication nowadays would go through heterogeneous networking environments composed of different networking devices. Therefore, managing these large-scale and heterogeneous networks requires the automation of configurations that enable the networking devices to support efficient end-to-end communications. In the case of failures or traffic engineering needs, timely reconfiguration in an automated manner is also required. Minimizing human interaction to fetch updated physical configurations is crucial to avoid unnecessary flaws regarding programming. Additionally, this feature helps the generalization of network programs and hence, re-usability, which saves time, cost, and improves user experiences. To conclude, the automation of the programming cycle is an urgent demand which starts from expressing network policies, goes through collecting network statistics, applying related analysis and optimization and ends by generating suitable network devices configurations.

In addition to automation, the performance of northbound interfaces itself is also essential. For example, the response time to user requests and the compilation time needed to translate users' policies to low-level commands should be minimized. Note that there is already propagation delay caused by physical transmission media and multiple queuing delays caused by forwarding devices (which are geographically distributed). To satisfy a better user experience, northbound interfaces should be written in such a way that keeps in mind the minimization of the delay resulted from generating low-level commands from submitted policies.

This thesis addresses these aspects and develops a couple of approaches to provide efficient data management and a high-performance northbound interface that could be run either with OpenFlow-SDN or Segment Routing. The following section details the overall research questions.

1.2. High Level Research Problems

P1 Performance: Using a northbound interface or an abstract comes with the cost of overhead during the life cycle of a policy composition. One of the contributors to this overhead is the time needed to translate user commands in this abstract language into vendor-depended commands or other open-source southbound protocol (*e.g.,* Open-

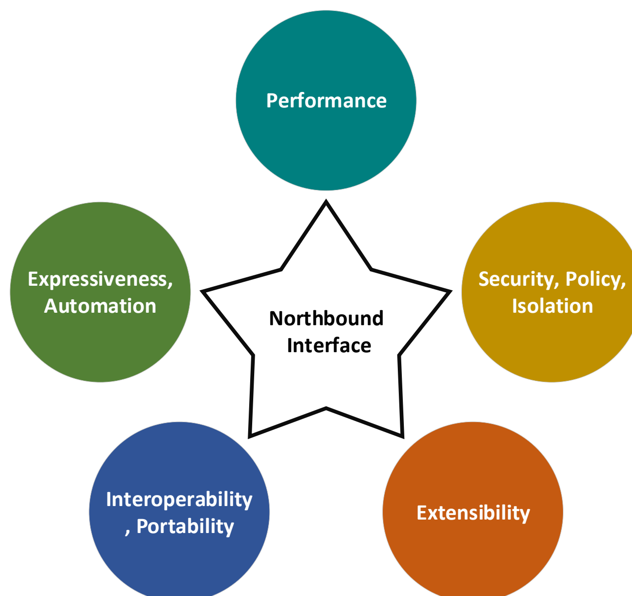


Figure 1.1.: High-level Research Problems associated with the northbound interface in SDN

Flow). We refer to this time in the thesis as *compilation time*. Another source of overhead is the time needed by the policy management or northbound interface to respond to network topology-related updates. This *response time* is critical when policies affected by these updates are sensitive to delays. Both sources show the importance of minimizing these delays while keeping other features available.

P2 Security, Policy, and Isolation: Isolation is when there are tenants who run different policies on their share of the network; each policy should affect only the user's share or part of the network. It is not possible for any user to manipulate any policy of traffic that belongs to other users. The northbound interface should provide tools to help users maintain isolation. It should also provide means for defining policies which control network behavior. Finally, the third component that orchestrates policy and isolation is security. Any attempt to manipulate network behavior should be authenticated and validated. Any attempt from unauthorized users should be rejected, and any submitted policy violates the general controlling policy should also be blocked.

P3 Extensibility: Another problem that faces northbound interfaces is the ability to be extended later. Network environments are evolving with time, and new implementation scenarios are continuing to appear. Such a demanding environment needs a

flexible interface that could be used in new scenarios; the interface or the abstract was not designed for. When a running interface failed to address or implement the new scenario, other abstractions are needed and this complicates the composition of policies.

P4 Interoperability, Portability: SDN architecture presents a clear separation between the control plane and the data plane. This separation means changing the technology that operates the data plane should not imply in its turn update to the control plane and network applications. However, northbound interfaces which were written for OpenFlow-driven SDN cannot operate directly on SR-driven SDN. Seamless interoperability helps network administrators to manage all their network infrastructure from a single view.

P5 Expressiveness and Automation: One of the main goals of the high-level network abstractions is to allow end users to express their intended policies easier than what could be done using low-level commands (*e.g.,* OpenFlow). As a result, writing complex policies should be made easy. This feature is linked to the fact that using software-networking technology is aiming basically to automate policy management and hide low-level details. In this context, It is important that end users should not be bothered with gathering low-level information (*e.g.,* Routers' IP addresses) to get their policies to work correctly.

1.3. Thesis Challenges

Giving the high-level research problems described earlier, this section outlines the main challenges that are addressed in this thesis.

1.3.1. Performance

A wide variety of network abstractions have been developed and each is targeted at a certain type or set of network policies. Besides, network abstractions continue to evolve to address new emerged network policies requirements. As a result, it is often insufficient to implement a complete network policy with one single abstraction especially when network policy requirements keep changing. In many cases, network administrators have to combine two or more abstractions to formulate and implement their policies. The complexity of combining abstraction policies increases with the number of employed abstractions and can be a tedious task [2, 11].

Existing approaches to this orchestration challenge have only provided a partial solution [10, 12–15]. These solutions either require writing a new wrapping library for enabling a new cooperation pair or depend on common structures (e.g. OpenFlow rules or network state variables) and further increase programming complexity as they work on low-level commands.

To address these challenges, Wang et al. [2] proposed plain data presentations of the network to simplify the complexity of combining and integrating policies resulting in a simplified northbound interface. For instance, in Ravel [2], the whole network is modeled as a relational database and application developers can request ad-hoc views based on the database tables, which can then be queried against. However, these advantages come at the price of performance. In essence, the network model and all relevant information are distributed across different, typically normalized database tables, leading to significant delay when aggregated views are used to establish a complete view. Although inserting specific information (*e.g.,* a firewall entry) is fast, retrieving information that needs to be collected from many tables is costly (*e.g.,* retrieving routes). Consequently, even simple applications need to interact with a large number of database tables. As a result, the processing application request is slow, leading to the conclusion that writing network applications can still be overly complicated.

This thesis is investigating the possibility to re-organize data in the network controller such that the performance of running network applications is enhanced.

1.3.2. Portability

On the one hand, after the introduction of SDN in [7], researchers started to develop different solutions on top of SDN. Direct management of the southbound interface (i.e. OpenFlow) was one of the main challenges during the early stages. Researchers responded early to this issue and presented many approaches to ease OpenFlow handling (i.e. northbound interfaces). These approaches could be categorized based on their end objectives. Some were focusing on optimizing resources reservation [16, 17], some supporting multiple composition [10,15,18], and others minimizing the number of forwarding rules in the dataplane [19]. On the other hand, SRv6s is a variation of Segment Routing networking technology that runs on top of IPv6 networks. Segment Routing presents a new way of doing SDN which is easier to integrate with legacy networking more than OpenFlow-based SDN. Migrating the northbound interfaces and abstracts that were written for OpenFlow-based SDN to SRv6 involves an intensive restructuring of the internal software of these abstracts.

This thesis, in its second part, tries to motivate portability by using what will be presented in its first part. The OpenFlow-based SDN controller in the first part would be used in the second part as an SDN controller for SRv6 networks.

1.3.3. Expressiveness and Automation

The IETF draft [20] introduced the concept of encoding network commands (*i.e.,* SRv6 behavior) as IPv6 addresses in the segments list. Therefore, whenever a network administrator wants to implement a network program (e.g. traffic engineering), she/he needs to inject a segments list that represents his program in the packet's header.

Even with all of these programming capabilities enabled by SRv6, network administrators still face the difficulty of manually constructing segments lists that fulfill their intents and policies. To the best of our knowledge, there is only one proposal that partially automates segments list management [21], where authors proposed to utilize the DNS service in the enterprise network to transfer segments lists between end users and the controller. However, the proposal does not react to networks updates and overrides the forwarding leveraging DNS service instead of using the service IP address, which not make it applicable in several real contexts. In other related SRv6 works [22–26], segments lists were composed manually as topologies used in the evaluation tended to be small. However, in real operated network topologies, manual composition presents various challenges in the context of composing network policies. Challenges such as the errors prone manual segments list composition, responding to dynamic network topology events, finding a correct parameter to pass in the SRv6 command, and possible conflicts between SRv6 behaviors could exist due to behavior misuse.

The thesis in its two parts investigates the automation and expressiveness challenges. The objective is to present northbound interface support that could be easily extended to address future scenarios.

1.4. Thesis Contributions

This thesis presents efficient data management and policy composition for software-defined networking that addresses the challenges mentioned earlier. In Figure 1.2 we show the relation between the two parts of the thesis and how they relate to the standard SDN architecture. We elaborate in this section how the contributions in the thesis relate to the challenges mentioned earlier.

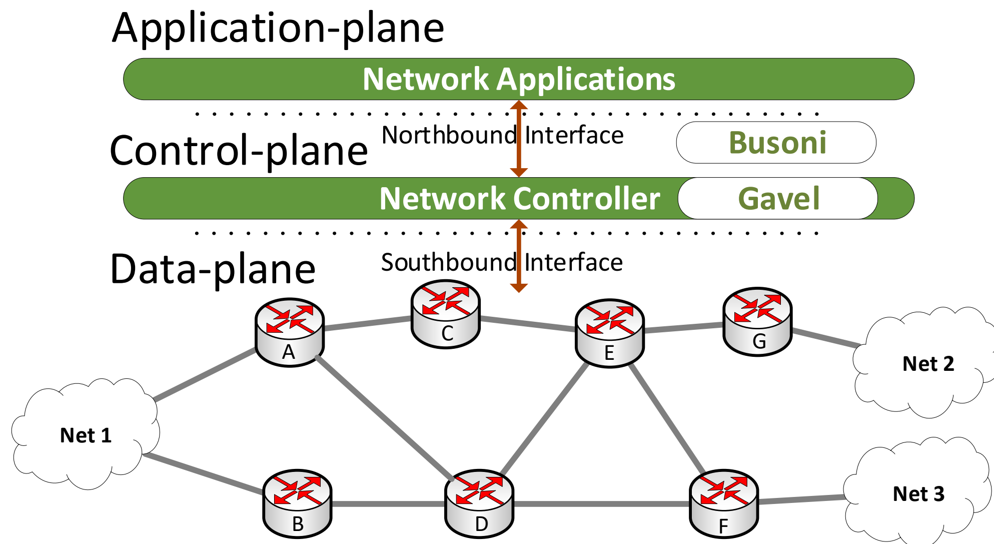


Figure 1.2.: The position of *Gavel* and *Busoni*, the thesis contributions, in the SDN architecture

1.4.1. Performance

We developed *Gavel* [27] that addressed the performance challenge in the data representation. Gavel is an SDN controller that utilizes a *graph database* management system to provide a more natural plain data representation that can be easily queried by network applications. Gavel organizes network topology data in a graph structure which provides at the end a high-performance data representation. Gavel is the first controller to exploit graph databases to produce a plain data representation of a software-defined network, and thereby removes the need for a translation between multiple, different and task-specific network policies. Compared to the RDBMS-oriented state-of-the-art of plain data representations, Gavel significantly reduces programming complexity and is able to scale better in large networks. The key factor for these achievements is facilitating a much more natural native graph support instead of relying on an RDBMS table structure.

We have further implemented a variety of proof-of-concept network applications on top of Gavel. By exploiting the native graph support of the database engine, Gavel significantly eases application and policy writing. Additionally, we show by experimental evaluation of several typical applications on multiple different topologies that Gavel offer *significant* performance improvements over state-of-the-art solutions.

1.4.2. Portability

Taking this challenge into consideration during the design of *Gavel*, we further designed and implemented *Busoni*. Busoni's main objective is to provide automation for policy management on top of SRv6 network; however, utilizing Gavel in the implementation of Busoni shows the portability advantage of Gavel. Although Gavel was designed and introduced in OpenFlow-SDN environment, it operates smoothly with Busoni in SRv6 environment. This feature allows users to write programs without the need to know the exact technology that drives the data plane. The only changes that are needed is the adding of the connecting drivers that sense the topology structure and changes associated with it.

1.4.3. Expressiveness and Automation

Our work *Busoni* addresses automation and expressiveness challenges in the northbound interfaces. *Busoni* provides the proper tools to manage policies on top of SRv6. End-users can use Busoni to automate the generation of their policies. They can define endpoints in flexible terms as we show later, and write their functions or any peculiar behavior that they want to apply to the flow between the defined endpoints. In the case of network dynamics or failure events, the framework will automatically update the affected policies and report any events for which Busoni failed to find enough resources that satisfied the policy's goals.

Using different use cases we show the programming capabilities offered by our framework. We start with a service function chaining scenario where traffic between two areas should be processed through a sequence of network functions. We also show how Busoni would be used to apply a VPN policy. In the evaluation, we demonstrate the scalability of our platform and the improvements achieved in response time for dynamic network events.

1.5. Thesis Outline

This section outlines the main two parts of this thesis and the organization of chapters within these parts. In Chapter §2, we first present the background on state-of-the-art SDN/NFV/SR frameworks which advocate network softwarization. Presenting more on how network programming is done with SR, we elaborate more about SRv6 programming. Finally, we briefly introduce the graph databases.

In Part I, we present *Gavel* an SDN controller that utilizes a graph database management system to provide a more natural plain data representation that can be easily queried by network applications. Chapter §3 outlines the problem statement, Chapter §4 presents

the state-of-the-art solutions and related work, Chapter §5 details our Gavel solution, and Chapter §6 presents some future extensions.

In Part II we present *Busoni*, a framework to compose and manage network policies on top of IPv6 SR networks. Busoni provides the needed programming functions to network administrators as a northbound interface on top of an SR controller. Chapter §7 outlines the problem statement, Chapter §8 presents the state-of-the-art solutions and related work and Chapter §9 details our policy framework to account northbound interface portability, performance, automation, and expressiveness problems.

Finally, in Chapter §11, we revisit the overall contributions and impact of this thesis and outline the key future research prospects of this dissertation. Besides, the supplementary materials in support of this thesis including the relevant pseudo code, proof of theorems, data-flow and sequence diagrams are listed in the appendix Chapters §*A*-*C* of part III.

Chapter 2

Background

We provide in this chapter an elaboration for the fundamentals concepts and technologies, which they serve as a prerequisite to follow and understand the next chapters. We introduce first the primary motivation behind this thesis *Network softwarization*. Later, we present *Segment Routing* and *Graph Databases*, which are used to implement contributions presented in this thesis.

2.1. Network Softwarization

In the last decade, the need to automate network management operations became an essence. One of the factors behind this is the massive size and the varieties of networks comparing to the early days in the '80s and '90s. "***Network Softwarization***" in the form of *Software-Defined Networking (**SDN**)* and *Network Function Virtualization (**NFV**)* is the normal response giving software flexibility is higher than hardware. *Network Softwarization* has influenced and innovates the design, deployment, and management of networks [28].

2.1.1. SDN

To offer flexibility in network control, Software-defined Networking (SDN) introduced the separation of the control plane from the data plane [6]. In legacy networking, these two planes exist together in each device; therefore, each device processes packets according to its view of the network, which is a limited view considering the size of the network. Making the control plane logically centralized provides a single entry to manage the network and to apply different policies. It also helps in generating optimized traffic routes comparing to legacy networking [4]. Figure 2.1 shows the typical SDN architecture, where we can

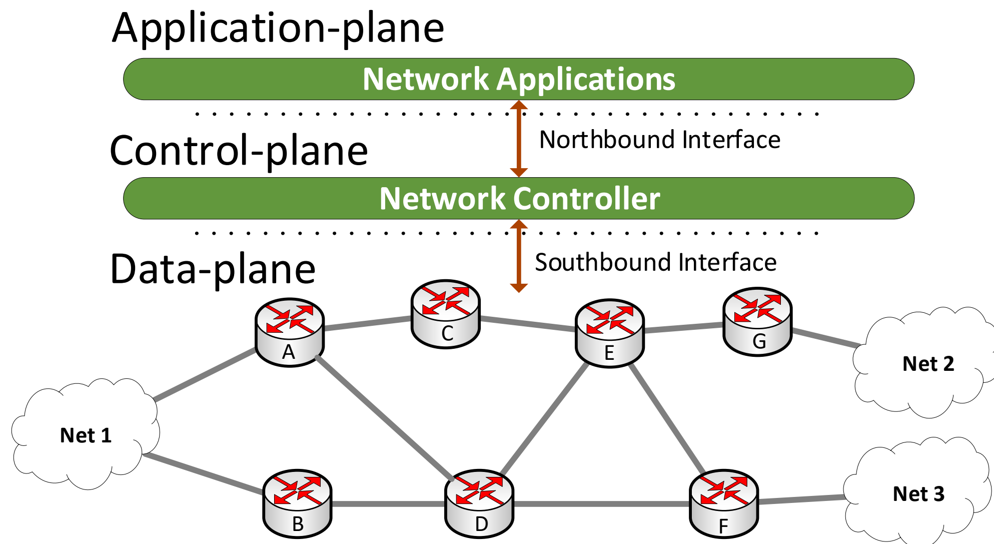


Figure 2.1.: SDN architecture

clearly distinguish three planes; data-plane, control-plane, and application-plane. Devices in data-plane focus only on forwarding packets and delegate thinking and path calculations to control-plane. Additionally, controllers also collect statistics about network state periodically to find better routes. The third plane is the application-plane, which is the main motivation behind this architecture. This plane allows network users to easily compose and define their policies and applications that control network behavior. The SDN architecture also provides two types of communication interfaces to allow smooth interaction between network applications and forwarding devices.

2.1.2. NFV

Middleboxes are one of the fundamental parts of any network infrastructure. Their task is to perform any functions other than the standard router's functions. According to V. Sekar *et al.* [29], the number and the diversity of these devices are observed in nowadays networks. In 2012, the European Telecommunications Standards Institute (ETSI) proposed a paradigm that focuses on managing NFV middleboxes. The new software middleboxes or NFV separates software implementation from propriety hardware of network functions, which delivers three main advantages. The first is the freedom to run these network functions on any platform either as a virtual machine (VM), as a container, or on bare metal. The

second advantage is the separation between software development timeline and hardware/-software maintenance which provides enhanced network functions. The last advantage is the dynamic scaling provided due to the natural process of spawning new instances when there is a demand on a service or function or shrinking down when they are idle and saves, therefore, power consumption.

2.2. Segment Routing

We present here Segment Routing technology and its relation to SDN and network softwarization. Then, we focus on IPv6 variant and how it could be utilized to deliver network as a program service.

2.2.1. Overview

SR [8] was proposed to address issues concerning MPLS control plane manageability. SR is a variation of source routing where instructions, commonly known as *segments*, are attached to packet headers in order to implement detours to the default shortest path. It is also presented as a different SDN implementation to OpenFlow based networking where current networks could utilize SR and facilitate SDN management capabilities by upgrading the legacy routing devices' operating systems.

SR plays a decisive role in network scalability and allows a more effortless network management experience. This experience is possible because SR does not keep the state in the core routers, where classification and embedding segments take place at ingress routers [30]. Moreover, SR reduces the load on network controllers by offloading default routing decisions to data-plane routers provided that not all routing decisions need individual path computations. SR exploits the ability in data plane devices to run distributed shortest path protocols like *Open Shortest Path First* (OSPF) to perform shortest path routing. This ability effectively leaves the non-shortest (constrained) path inquiries to the network controllers as this needs a knowledge of the whole network and its current status (*e.g.,* during traffic engineering).

SR specifications are currently being developed in the *IETF Source Packet Routing in Networking* (SPRING) work group [31]. These specifications target the compliance of SR in different use cases such as SDWAN [32], mobility [33], protocol extensions [34]. Additionally, SR could be implemented on top of either MPLS or IPv6 (*i.e.,* SRv6) networks. Providing this flexibility for network engineers would allow SR to be easily integrated into existing networks. While the detours in MPLS are implemented as MPLS labels, the de-

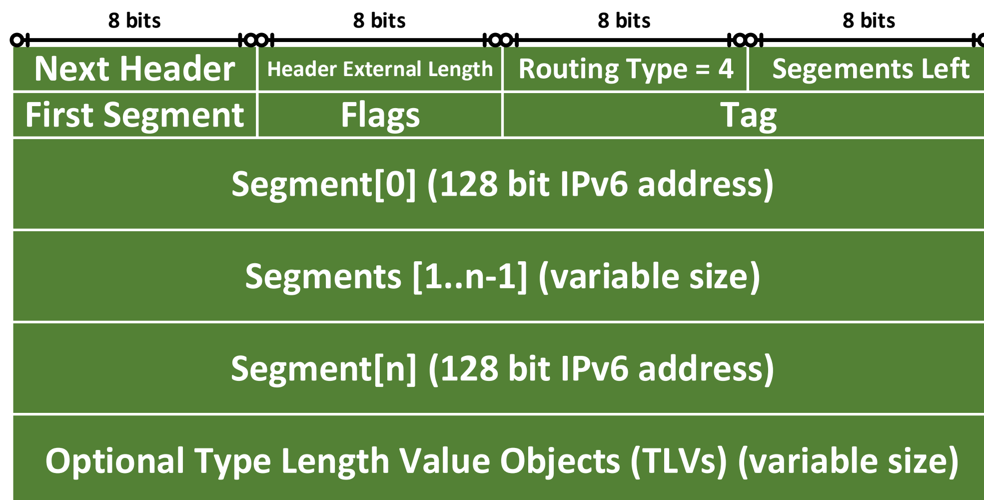


Figure 2.2.: SRv6 Extension Header

tours in SRv6 are represented as IPv6 addresses. Although MPLS variant of SR could be more attractive to *Internet Service Providers* (ISPs), the IPv6 variant is more promising given the massive number of Internet-connected devices (IoT) which yields in need to a vast addressing space.

2.2.2. Segment Routing on IPv6: SRv6

SRv6 as stated before used IPv6 addresses to tag the needed detours in the network path. To do so, it exploits the extension headers support in IPv6 to attach the segments list by defining an *SRv6 Extension Header* (SRH) [34]. This support means that SR-incapable routers and SR-capable routers can co-exist in the same IPv6 domain. When packets with SRH arrive at SR-incapable router, they will appear as standard IPv6 packets and will be routed based on the router table and the destination/source address. Such an environment allows the packets to flow smoothly and any SRH related detour takes place only at SR-capable routers, which results in more comfortable, incremental adoption of SR on the broader network.

In *SRH* as showed in Figure 2.2, *Segment Identifiers* (SIDs) are stacked to indicate the detours that packet should take when it flows through the network. The segments list ordered reversely (*i.e.,* Segment List [0] contains the last segment to visit). To indicate which segment is the next detour, Segments Left is designed for this purpose. *Type Length Values* (TLVs) field contains information regarding Operations, Administration, and Manage-

ment (OAM) functions [35] or authentication information as *Hashed Message Authentication Code* (HMAC) which enhances the security of the source routing. The remaining fields are used as described in the original RFC 8200 [36].

The SRv6 SIDs used in the header could be classified based on reachability either global or local segments. All routers in a single domain can route packets to the global segments, while a single router only reaches local segments. Therefore, in the case of the local segment in the segments list, the global segment of the hosting router should be routed first. Each entry in the segments list consists of 128 bit and coded as IPv6 address. The SIDs also could be classified based on the type they are referring to. There are *Adjacency Segments* which refers to the ports of the routing device. So when a router has four ports, it could have four different adjacency segments. There are also *Node Segments* which represents the routing devices in the data plane. Each router could have only one global node segment.

2.2.3. SRv6 Programming

The introduction of SRH opens the door to new programming features in SR. The IETF draft [20] introduced the concept of encoding network commands (*i.e.,* SRv6 behavior) as IPv6 addresses in the segments list. This means when a network node receives a packet with SRH, and the destination address matches an associated behavior provided by this node, it will execute this defined action. Therefore, if a network administrator wants to implement a network program (*e.g.,* traffic engineering), she/he needs to inject a segments list that represents his program in the packet's header.

Figure 2.3 depicts how a single IPv6 address would look like when we embed an SR network command in it. The first part (named the locator) is used to route the node that hosts the function. The second part holds the function that is needed to be executed which could refer to an app in a container/VM or a stand-alone network device. The last part, which is an optional entry, holds an argument which could be needed to be passed along with the command. The specific length of each part is not fixed to give each network flexibility on how it uses these features.

In a different usage from detour SIDs, SRv6 behaviors or commands and optionally arguments are inserted in the remaining bits after the hosting node's prefix. Thereby, routers in the network will use the node's prefix to deliver this packet and locally, host node (SR capable) will use the behavior bits to forward the packet to the function's holder (VM or container). The advantage in such embedding policy is that there is no need to route functions or behaviors in a flat routing architecture, instead only keeping the host node prefix in the routing table should be sufficient which yields in fewer routing rules in the data plane devices.

Figure 2.3.: Representing SR network command using IPv6

```
self.insert_behavior_first_segment("T_Encaps")
self.insert_behavior_end_segment("End_DT6",self.vpnuser)
```

Figure 2.4.: An example of declaring segments list and SRv6 behaviors

SRv6 behaviors come in different flavors and range from basic instructions related to forwarding actions to more complex instructions such as supporting non-SR capable network functions. For example, the `End` function indicates that the router must advance the packet to the next destination according to the segments list. In another flavor, there is a `End.X` command that specifies the port number to which the packet should be forwarded to. The `End.T` command specifies a look-up table that the router should use when it routes the packet to the next destination and the `End.B6` command to inserts a new SRH on top of the existing one. Besides the predefined behaviors, end users can define a custom set of functions; however, it is necessary to validate if the data plane devices can support the custom set.

Empowering Linux routers with SRv6 programming comes into practice after the last implementation efforts either in Linux kernel [37] or in FD.io project [38]. In Linux kernel (which we only consider in this thesis), SRv6 behaviors could be defined using `seg6` and `seg6local` options in the *iproute2* command. For example, as depicted in Figure 2.4, we instruct the Linux kernel to encapsulate the incoming packets with segments `2001::1` and `2001::2`. In the second command we activate the special `End.DT6` behavior which looks up the next destination using a table named `nh`.

To put the operation of the system as a whole, let us consider a network topology as shown in Figure 2.5 and a scenario where a network admin wants to steer the traffic between `Net 1` and `Net 3` through two network functions. Without any intervention from the controller, routers should use `B, D, F` nodes as the shortest path based on the number of hops. However, to traverse the needed SFC, SR then takes place, and the controller in its turn

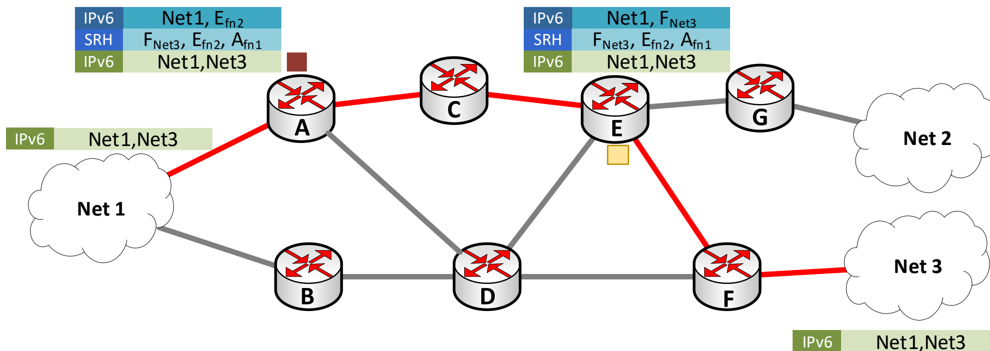


Figure 2.5.: An example of an SRv6 network topology

```
$ip -6 route add Net3 encap seg6 mode encap segs A::F1::, E::F2::, F::BEBE dev eth2
```

Figure 2.6.: An example of *iproute2* command to attach the segments list to packets

tries to find the best path to do so. We assume that virtualized network functions are properly configured and their SIDs are known by the network controller where the function `I` is hosted by node A and the function `II` is hosted by node E. The segments, representing the requested path, in this case, would be $\mathtt{FnI}_{\mathtt{A}}$, $\mathtt{FnII}_{\mathtt{E}}$, `F-Net3` where the packets are free to use the shortest path between these segments. In Linux routers, a command using *iproute2* as discussed earlier should be used. In this practical example, the command would be as shown in Figure 2.6. Where the segments `A::F1::, E::F2::, F::BEBE` represent the first function hosted in router `A`, the second function hosted in router `E`, and the end behavior in router `F` with `BEBE` as `End.X` which will send the links to the adjacency link connecting router F and Net3 respectively. We assume in this example that `Net1` connects to the network using one ingress point. In case of multiple ingress and egress points, the controller needs to repeat the above work for each pair of ingress and egress points.

2.3. Graph Database

Although the term *Graph Database* (GDB) appeared first in the 80s, it comes into real practice only in the last decade [39]. A GDB is a data store where the data structure of its schema and instances modeled as a graph, and any data-related operation is expressed by a graph-based query language [39].

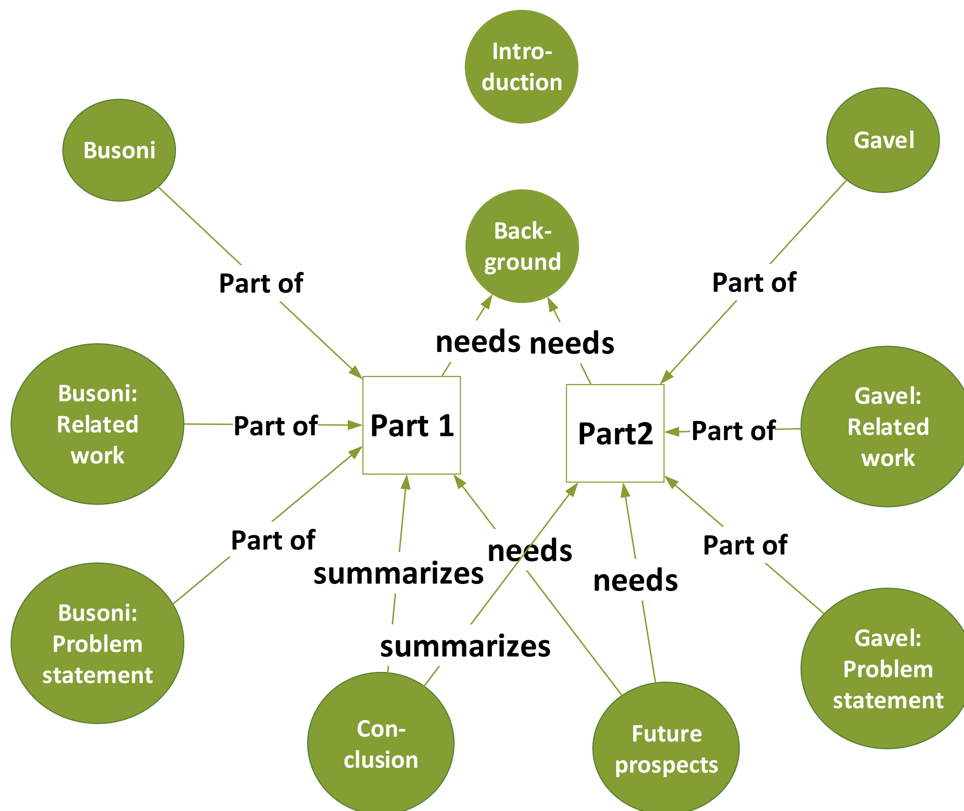


Figure 2.7.: The chapters of this thesis organized as a GDB model

The GDB distinguished itself from a normal relational database (RDB) in the way it organizes and processes its data. The GDB fits better compared to other databases in graph-based environments like road networks, computer networks, mail distribution networks, and so on. Additionally, the GDB also supports graph-based algorithms; therefore, any application depends on such algorithms would integrate easily in such an environment. In GDB, users can use nodes and links to model the target environment. Nodes would be used to describe main environments actors or components and links or edges would be used to model the relationships between the nodes. In Figure 2.7, we show an example of a GDB model which summarizes thesis's chapters and their relationships.

Part I.

A Fast and Easy-to-Use Plain Data Representation for Software defined Networking

Chapter 3

Problem Statement

3.1. Introduction

SDN is a new networking paradigm that promotes easy-management concept through the separation between the control plane and data plane as we described earlier in Chapter 2. Giving this separation, users now can focus on writing network applications regardless of which technology is used to operate the data plane. This separation motivated network administrators to start writing applications at a low abstraction level (e.g., issuing OpenFlow rules), however this hindered modular programming for SDNs [10]. To explain the challenges in using OpenFlow or any *Southbound Interface*s (SBIs) protocol, we can look at modern computers. Although Assembly was presented to ease programming using binary instructions, it is a challenging task even to write and maintain programs in Assembly. The solution was to design a new *high-level* programming language and a compiler to automate the generation of proper assembly instructions.

Going back to networking, almost the same situation applies; high-level northbound interfaces have been proposed recently. These *abstractions* usually promote an easy-to-write high-level language to express network policies, combine these into a single network policy and then translate it to a lower-level protocol (*e.g.,* OpenFlow). For instance, Pyretic [10] and Frenetic [18] proposed functional abstractions as solutions to construct SDN control applications. Both of them use NetCore [40] to automate the generation of OpenFlow entries. Also, FatTire [41] and, Merlin [42] presented extended support on top of Frenetic for fault-tolerance and resource provisioning type policies, respectively. Last but not least, PGA [12], Kinetic [13] and Janus [11] proposed additional abstractions for graph handling, finite state machine support, and real-time realization on top of Pyretic. The main purpose of these abstractions is to ease network programming by providing modular tools and easy-to-use libraries to compose different network policies.

3.2. Challenges in SDN Northbound Abstractions

Although there are this wide variety of network abstractions available, each abstraction usually targets a specific type of network policies. Besides, network abstractions continue to evolve to address new emerged network policies requirements. As a result, one abstraction is often not enough to implement a complete network policy, especially when network policy requirements keep changing. In many cases, network administrators have to combine two or more abstractions to formulate and implement their policies. The complexity of combining abstraction policies increases with the number of employed abstractions and can be a tedious task [2, 11]. Here, one major issue is that different abstractions employ different data representations that would need to be translated in order to combine multiple abstractions.

Existing approaches to this orchestration challenge have only provided a partial solution. These solutions can be broadly categorized in:

1. A high-level perspective [10, 12, 13] in which high-level representations coordinate between each other, which essentially requires writing a new wrapping library for enabling a new cooperation pair

2. Low-level or abstraction agnostic solutions [14, 15] which depend on common structures (e.g., OpenFlow rules or network state variables) and further increase programming complexity as they work on low-level commands.

To address these challenges, Wang et al. [2] have recently proposed plain data representations of the network to simplify the complexity of combining and integrating policies, resulting in a simplified northbound interface. Here, any application-specific structure that might be outgrown by future demands is discarded in a much simpler network model, making orchestration easier. For instance, in Ravel [2], the whole network is modeled as a relational database, and application developers can request ad-hoc views based on the database tables, which can then be queried against.

This approach has three significant advantages: First, SQL as a query language is widely known in the community and among administrators. Second, an RDBMS guarantees consistency and integrity among different views. Third, different applications (*e.g.,* virtual networking) can be realized by exploiting database views.

However, these advantages come at the price of performance. In essence, the network model and all relevant information are distributed across different, typically normalized database tables, leading to significant delay when aggregated views are used to establish a complete view. Although inserting specific information (*e.g.,* a firewall entry) is fast, retrieving information that needs to be collected from many tables is costly (*e.g.,* retrieving

routes). At the same time, implementing a new application requires first a good understanding of the database scheme, and second, manipulating this scheme.

Consequently, even simple applications need to interact with a large number of database tables. For instance, to calculate the shortest path between two nodes in the network, one of several steps requires—for each switch on the path—to query a table storing that switch's port connectivity. As a result, the processing application request is slow, leading to the conclusion that writing network applications can still be overly complicated. In particular, each application needs to create its tables and link them correctly to the existing database. This can be a tedious task for the application developer, especially as the number of applications running in the network increases.

To summarize the challenges as mentioned earlier, using a relational database as a plain data representation in a networking environment presents unnecessary overhead especially when the network topology is huge and complex. There is a need to have higher performance abstraction while maintaining the simple data representation offered by *Ravel*. This part of the thesis answers the question raised in the first chapter regarding performance enhancement using better data organization in 1.3.1.

Chapter 4

Related Work

Related efforts could be categorized into three different classes of directions: north-bound interface abstractions, the use of databases in existing SDN controllers, and the use of graph modeling in networks.

4.1. Abstractions

Many abstractions and policies frameworks have been proposed to ease network programming for different types of applications. Most works on northbound interfaces started with Frenetic [18] and Pyretic [10] which proposed functional abstractions to construct SDN control applications. These two abstractions used NetCore [40] as their core language for forwarding decisions. Later on, Frenetic replaced NetCore with NetKAT semantics [43] which is an extension to NetCore but verifiable for Kleene algebra with Tests [44] (KAT) and complete in the sense of no bugs are missed. Later abstractions kept evolving either to support some scenarios or to extend existing abstractions. For example, FatTire [41] and Merlin [42] allowed for extended support for fault-tolerance, and resource provisioning type policies, respectively. Last but not least, PGA [12], Kinetic [13] and Janus [11] proposed additional abstractions for graph modeled policies, finite state machine support and temporal based policies on top of Pyretic.

Other proposals were developed as a northbound intent framework, for example, [45] which tries to simplify writing network application by enhancing intent framework in ONOS [46], but is still a somewhat limited solution as it was designed for a single SDN controller (i.e., ONOS). Other works that go in this direction include [47, 48] which focus on supporting path finding and writing network intents, but do not study the consequences of composing or simultaneously running multiple intents.

The variety of existed abstractions is a two-sided coin. The positive side is the broad support to different network scenarios. The other side could present a stiff challenge to network flexibility [49], where network administrators need to spend time figuring out which is the best abstract, or the orchestration of different abstracts, to represent a network policy to respond to an event. The need to automate the orchestration of different network programs written with different abstracts would represent another challenge. Referring to the second challenge, there are two ways to combine network applications. The first approach involves a high-level perspective in which a high-level representation [10, 12, 13] coordinates between different applications, and it is then restricted to the use of specific abstractions. The other approach is a low-level conflict resolution which is abstract agnostic [14, 15], hence it depends on common structures like OpenFlow or some network state variables and yields in increasing programming complexity. These two approaches present a partial solution to the orchestration problem.

As a plain data representation, Ravel can support different types of policies and scenarios without worrying about composition operations due to its imperative paradigm [50]. As we will discuss throughout this part, while *R*avel relies on a *r*elational database for representing the network, our approach uses a *g*raph database engine and thereby offers improved application program-ability and controller performance.

4.2. Databases in SDN Controllers

In another direction, although databases are almost in every SDN controller [6], they are only used for state distribution, distributed processing, concurrency, replication control or network state storage which are passive roles. For example, [51] used a database in passive roles to maintain flow statistics and information for every domain registered in the SDN controller. Also [52] uses the graph database in a passive role to keep track of network topology. [53] exploits a database's synchronization capability, and more specifically, a relational database to synchronize the states saved in switches with the controllers and hence, the switches will be able to update routes accordingly. Ravel [2] first leverages the database management system in active roles for controlling an SDN, however it relies on a relational database which affect the performance of running graph-based algorithms and routines. Conversely, our approach is the first controller to employ a powerful graph database engine for network control that supports graph-based algorithms.

4.3. Use of Graph Modeling in Networks

Our approach is not the first system to exploit graph modeling for network management. Researchers have indeed recognized the opportunities of graph libraries in network management before. Especially, in Netgraph [54], the authors presented new primitives to support network management in cloud computing. Later on, other supporting libraries are also developed, such as [55, 56]. As a consequence, Some solutions take advantages of these libraries and presented more efficient solutions such as [57] which enhanced path finding in distributed controllers environment to scale up properly comparing to standard shortest path algorithms. The main advantage using these primitives is easing graph-based operations and routines where programmers will make abstractions of their implementations on the graph level instead of directly perform actual implementations on code-specific data structures (e.g., matrix or trees). However, these libraries still play a passive role in the controller memory.

What we propose, on the other hand, builds the entire network model on a graph database and exploits the graph database engine to facilitate network control so statistics and information will be saved on the desk after the controller is off. Such statistics could be used later on for further analysis after a sudden shutdown or enhancing machine learning module to take better decisions. Assuming that these graph libraries could implement regular back up operations, they still lack other database features such as state synchronization and a descriptive query language [58] which our approach presents as it will be described later.

Chapter 5

Software-defined network control with graph databases: Gavel

In this chapter, we present our graph database defined networking approach. We show in details how we use a graph database to present a plain data representation and provide a high-performance northbound interface to compose different network policies. We first justify our design choices and then elaborate on the detailed architecture of our approach. After that, we evaluate our system and compare its performance to the state-of-the-art.

5.1. Introduction

To address the challenges that presented in Chapter 3, we propose *Gavel*: an SDN controller that utilizes a *graph database* management system to provide a more natural plain data representation that can be easily queried by network applications. More specifically, our contributions are as follows:

- We systematically investigate how graph databases can help to provide plain data representations and present several options for this purpose. Specially, we propose Gavel which exploits the graph nature of the computer networks and consequently employs a *graph database* instead of a relational database. The result is a much simpler network model that yields substantial reductions in programming complexity and query latency.
- We enhance Gavel to be able to connect to underlying data-plane devices via an OpenFlow composer directly.
- We show our system's capabilities by presenting different application types starting from simple routing to complex *Service Function Chaining* (SFC) application and network slicing.

- We conduct extensive evaluations on Gavel's components and compare their performance and impact.
- We provide a public repository [59] that contains the source code of Gavel infrastructure and applications. The repository also includes all testing scripts that were used in the evaluation section to help reproduce the results or investigate further scenarios.

5.2. The Case for the Use of Graph Databases

One major obstacle to integrate different abstractions is the diversity of the data representations within these abstractions. Here, any integration would involve translating one (or several) representation(s) to another. In this work, we aim to provide a *plain* data representation of the network. Any networking application can then query this representation. In this section, we first elaborate different available data representations then explain the choice for a graph database representation.

5.2.1. Data Representations

The integration and data exchange between different abstractions on the same network requires an understanding of how each abstraction internally handles data. Here, exploiting the abstraction-specific APIs for data exchange would lead to a substantial delay in internal processing due to massive routine calls. One approach for a more efficient and flexible data representation relies on exploiting relational databases [2]. Here, the data is stored in different tables and can then be queried by any arbitrary application to retrieve information of interest. Such *plain* data representations would thus be the same for each abstraction.

Employing an RDBMS for representing data also yields additional benefits, such as the ACID properties of these systems. For example, *atomicity* can be utilized to ensure that all database entries corresponding to a freshly installed path are installed or updated successfully; *consistency* can ensure the integrity of the database and, once a network program tries (by accident or intention) to break the consistency, the database management system will stop executing this application and reject the faulty query. *Isolation and durability* would contribute to the availability of the database to multiple-access and the availability of an update for future usage respectively.

5.2.2. Drawbacks of Relational Databases

State-of-the-art solutions of plain data presentation currently employ a relational database, where these operations are provided by creating tables that can later be queried, aggregated, updated, and so on. However, the more complex applications become and the more applications we deploy in our network, more tables are needed, and more complex dependencies among tables become. This increases the programming effort required by the network administrator.

For example, as depicted in Figure 5.1, a representation based on a relational database will need to keep track of different network entities (such as hosts, switches, or links) in different tables. A routing application that tries to find the shortest path between two hosts will have to query and combine needed data from all these different tables, and then insert the result in other tables that maintain the routing information. If another application (*e.g.,* a firewall) is running in parallel, this application may have to modify the database schema (as, *e.g.,* in Ravel [2] firewall rules cannot be represented by the existing topology's tables) or even existing applications (*e.g.,* in Ravel [2] the routing application has to be modified to also respect firewall rules when querying tables). Although in a general case, the database scheme could be designed to keep the need to add new tables to its minimum, this would require a competent prediction of any network application that could be run in the future.

At the same time, the increasing complexity also results in reduced performance. If more applications are deployed in the network, more (and likely more complex) tables will need to be queried to retrieve an answer. Similarly, when updating network information, this information has to be made consistent across all tables. As network control is time-critical, this processing overhead can become prohibitive.

5.2.3. Advantages of Graph Databases

To overcome the drawbacks mentioned above, we propose *Gavel* which employs a graph database as the main component of the control plane architecture (GDBMS in Figure 5.1). Graph databases, such as Neo4j [60], have been mainly considered as an alternative for handling the exponential growth and high complexity of social networks, as they offer significant advantages for modeling networks over traditional relational databases [39].

In this thesis, we advocate to exploit these advantages when modeling SDNs. Since a graph in nature best represents networks, graph databases are a much more natural fit for them. In general, we can model each device in the network as a node in the graph database, and then easily manipulate its attributes. These advantages are best shown by illustrating examples.

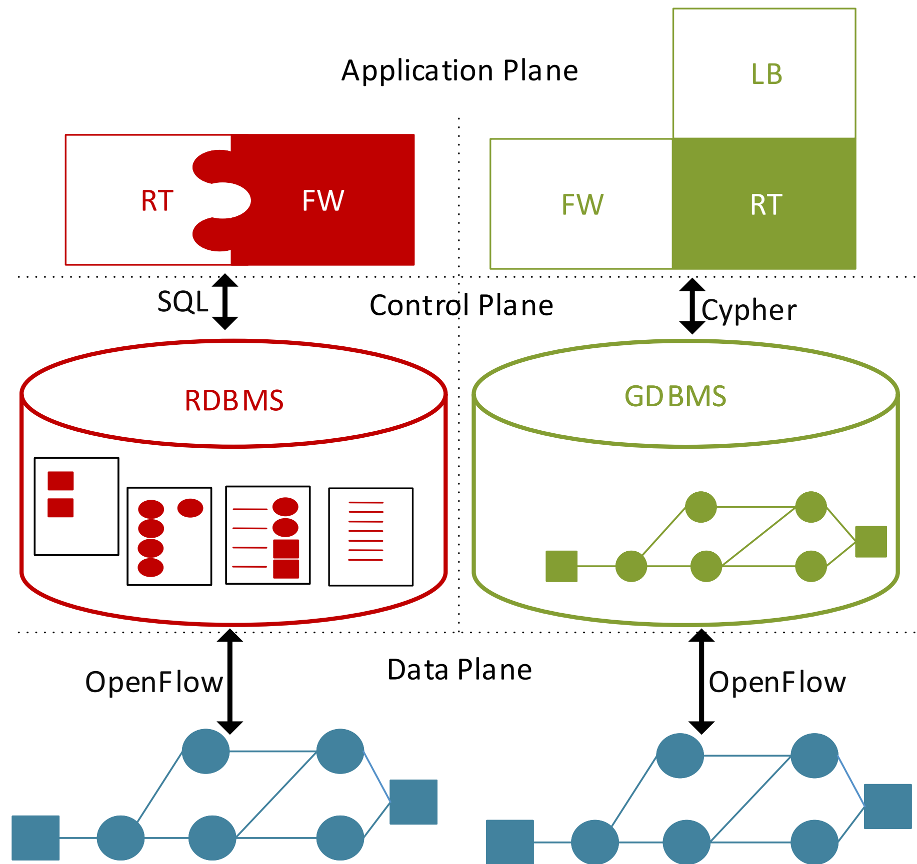


Figure 5.1.: Database management systems as SDN controllers

- Inserting a new switch into a relational database model requires the query first to be disassembled into basic data types, then each of these data types mapped to an appropriate table. In particular, the information of a new switch, all its ports, and all the neighbors that it is connected to, have to be stored in the corresponding tables [2]. In a graph database, on the other hand, inserting a switch only requires adding a node and one edge to each of its neighbors in the graph structure. It is not necessary to query or insert further information about neighbors connected to the newly inserted switch because the graph database management system can update and link neighboring nodes with the inserted edge and node.
- Similarly, requesting data from a relational database (*e.g.,* finding a route between two hosts) will require first to collect the relevant nodes, their attributes and connectivity information from different tables in an RDBMS [61]. In a graph database such as Neo4j, such operations are natively supported. That is, we can query the graph structure for the shortest path between the two nodes (*e.g.,* `shortestpath` function in Neo4j).
- As a result of adopting the RDBMS network model, network administrators will have to consider two different topology schemes every time they write a policy. The First scheme is the physical network topology and the second is the mapping to relational database tables(i.e. the database scheme). In other words, administrators will have to design and implement their applications with respect to RDBMS rules and network topology logic. This is not the case in the graph database, where the network topology will be most likely the same as the database connectivity due to network graph nature (i.e. Gavel). It is good to highlight here that there could be other graph schemes and they may not preserve the advantage of the single view of the network in physical and logical schemes. However, part of this work contribution is to keep the advantage of this scheme design.
- Finally, the orchestration of applications (explicitly talking about Ravel and Gavel), in particular keeping network data consistent, is more relaxed with graph databases. For instance, when implementing a firewall, we need to modify node labels or attributes in the graph to restrict access for certain hosts or interfaces, instead of manipulating a routing application and the database scheme.

While these are simple examples, we believe they generalize well relating to more complex scenarios. Complexity in relational databases increases with increasing network sizes or more complex topologies, as table complexity increases. Besides, running SQL queries to do graph related functions will result in a worse execution performance compared to graph database [62]. The main reason behind that is the functional mismatch between a traversal algebra and the relational algebra [63]. Moreover, SQL query optimizer is unaware of the graph nature of the data; hence, it builds a sub-optimal execution plan, a side effect that would have a significant impact in a network topology with a high number of interconnec-

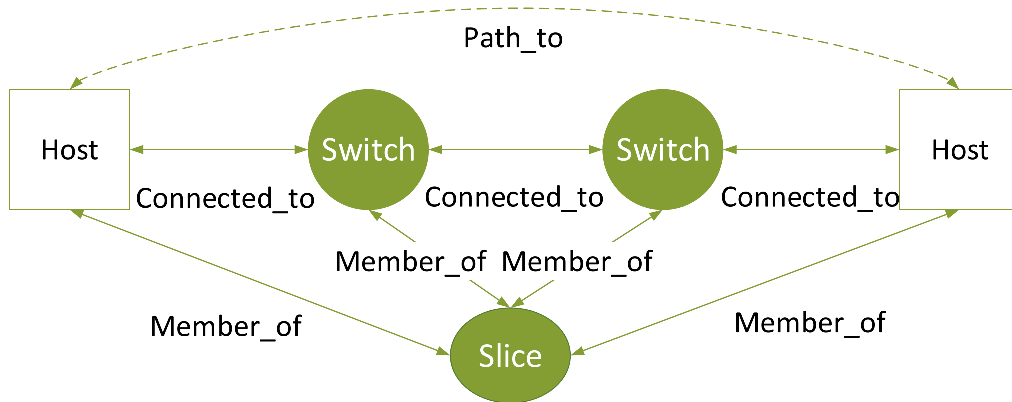


Figure 5.2.: Basic network topology model in Gavel

tions. However, Graph databases have proven to scale well in complex multi-million nodes networks and handle graph-based functions better [64].

5.3. Gavel Architecture and Design Choices

We start in describing Gavel's architecture with the direct mapping between real network entities and graph elements (*i.e.,* nodes and edges). Later, we model the detailed specifications of the network entities by adding them as proprieties to graph elements. Finally, we conclude by employing the graph database engine to function as an SDN controller.

5.3.1. Network Model

Graph databases usually use two entities to model environments; `Nodes` and `Edges`. As shown in Figure 5.2, we employed these entities to model the main network environment components such as switches, links, and policies.

- *Nodes* represent either forwarding devices, hosts, or any added function (*e.g.,* middleboxes). They keep information that suits the object they represent (*e.g.,* IP addresses for hosts and *OpenFlow Data Path ID* (dpid) for switches).
- *Edges* represent the relationship between the nodes, which could be a physical connection (*e.g.,* a physical link between two switches) or a virtual connection (*e.g.,* a path installed between two hosts or membership of a network slice). Edges can hold

Table 5.1.: Graph databases comparison matrix

Database	Graph Algorithms	Query Languages	Open Source
AllegroGraph [69]	Yes	SPARQL, RDFS++, PROLOG	No
ArangoDB [66]	Yes	AQL	Yes
DEX [70]	Yes	Traversal	No
GraphBase [71]	No	-	No
HyperGraphDB [72]	No	HGQuery, Traversal	Yes
InfinteGraph [73]	No	Gremlin	No
InfoGrid [74]	No	-	Yes
Neo4j [60]	Yes	Cypher, Gremlin	Yes
OrientDB [75]	No	Extended SQL, Gremlin	Yes
Titan [67]	No	Gremlin,	Yes

information based on the type of the relationship (*e.g.,* an edge describing a link between two switches can keep track of the connected ports from each device).

Compared to complex table structures where information is dissembled in many tables, this simplicity in modeling the network helps administrators to write network applications that follow graph logic easier as it exists in the network topology.

5.3.2. Selecting a Graph Database Engine for Gavel

There are many graph database engines available to realize our network model, and it is worth noting that Gavel can be implemented on several of them. In Table 5.1 we list several potential candidates. Our implementation uses Neo4j, as it offers three major benefits [55] that are in-line with our requirements: First, it provides native support to graph functions (such as path computation) that are called frequently through network management tasks. Second, in *Cypher* it implements a fast query language which is a good interface to write network applications [65]. Finally, Neo4j is an open-source software that offers more flexibility regarding adding plugins and extends its functions.

The only other graph database with similar characteristics is ArangoDB [66]; however, it has a much smaller developing community. More well-known alternative solutions such as Titan [67] do not implement essential functions, and the Gremlin query language is shown to perform worse than Cypher [68].

```
CREATE (:Switch {id: idvalue1, layer: layervalue2,
         dpid: dpid1});
CREATE (:Switch {id: idvalue2, layer: layervalue2,
         dpid: dpid2});

MATCH (s2:Switch {dpid: dpid2})
MATCH (s1:Switch {dpid: dpid1})

MERGE (s1)-[r:Connected_to]->(s2)
ON CREATE SET r.port1 = s1port, r.port2 = s2port,
      r.node1 = s1.dpid, r.node2=s2.dpid

MERGE (s2)-[r:Connected_to]->(s1)
ON CREATE SET r.port1 = s2port, r.port2 = s1port,
      r.node1 = s2.dpid, r.node2=s1.dpid
```

Figure 5.3.: Cypher snippets to add two switches and their connections to each other to the network topology

5.3.3. Native Graph Functions and Cypher

Native graph support is a critical feature that affects overall performance. When the engine can perform a graph related call using built-in functions, we can usually expect optimized performance compared to user-defined functions.

Neo4j has implemented many functions and algorithms that are used natively in graph structures. For instance, Neo4j can significantly improve the computation times of route calculations in an SDN. Routing is mostly finding the shortest path between two nodes, which is a built-in function in Neo4j. Relational databases cannot perform such functions natively and always need third-party libraries to carry out the same task. For instance, PostgreSQL uses pgRouting [76] for routing, which requires a database redesign to be compatible with the pgRouting library. Further, retrieving the path will require significant processing of tables.

Besides processing native graph functions, Neo4j in Cypher [65] offers its declarative query language. Cypher is simple, easy to read and has a flat learning curve. It could be used not only in manipulating the database but also to create the new nodes and edges as depicted in Figure 5.3. With the help of Cypher, Neo4j can natively traverse paths in the graph via the `connected_to` property as shown in Figure 5.2. Despite its simplicity, Cypher is

strong enough to manipulate a large-scale graph data store in an efficient way [68]. Another advantage of utilizing Cypher is its flexibility towards the combination with other programming languages (*e.g.,* Python or Java; we will later present an example for applications which employ both Cypher and Python).

5.3.4. Gavel Architecture

Following our discussion about graph databases and network model, we elaborate here more about the architecture details. Gavel consists mainly of a graph database engine in the core, interface to end-users, and a composer to translate management decisions into OpenFlow rules.

Starting with the core of Gavel, the graph database, Neo4j is used to store the network topology and also to execute network routines whenever needed. The database could be used later on to keep track of statistics and could be integrated with network applications to carry on some decisions making based on these statistics. For our proof of concept work, we kept the network model information as showed in Figure 5.2.

To allow users and admins to interact with Gavel, there is Cypher as described earlier. Users could run Cypher scripts using different tools. They could either use standard shipped web interface or the command line tool. They could as well embedded these scripts in a high-level language to allow reusing network data and also to ease the automation of such scripts. The network applications covered in this work follows the second way where we embedded them in a Python code file to allow the automation and reuse features.

Finally, to enable real-world Gavel deployments, we implemented an OpenFlow composer component. This component as shown in Figure 5.4 discovers the switches and the connecting links in the forwarding plane and reports them back to Gavel. It is also responsible for flow rules installation in forwarding switches. For the ease of implementation, we used some low-level POX controller [77] libraries to implement the low-level events, such as the discovery of underlying network devices and changes in their state. Gavel will react to these discoveries and updates by triggering appropriate Cypher commands.

The composer further translates Gavel management decisions to OpenFlow entries passed to corresponding switches. Whenever Gavel decides to install a flow rule in the network, this component will traverse all switches related to that flow and install the correct OpenFlow entries that ensure the success of physical connectivity in both directions.

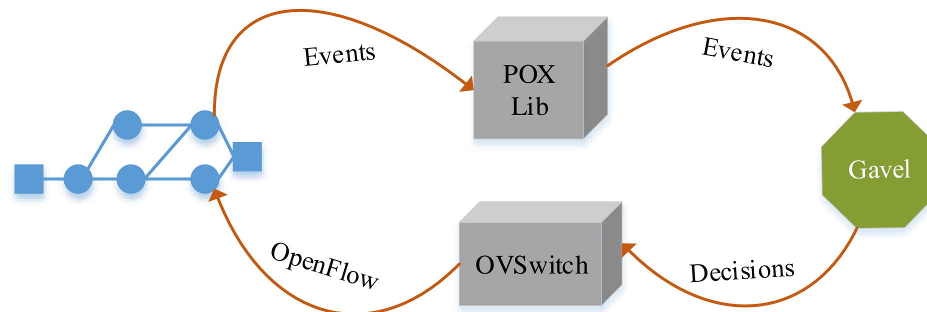


Figure 5.4.: Gavel interaction with forwarding plane to learn the topology and install the new forwarding rules

5.4. Gavel and Network Application Programming

As proof of concept, we next implement a variety of applications on top of Gavel. In this section we show that Gavel can i) indeed profit from native graph functions (by implementing a functional routing application), ii) handle different types of applications (by implementing a stateful access control firewall as a finite state machine application), iii) offer functionality to combine Cypher with different programming languages (by implementing a load balancer in combination with Python), and iv) employ Cypher to implement some popular network applications such as Service Function Chaining and Network Slicing. All source code of these applications including Gavel itself is publicly available online [1].

Additionally, we show that these applications can be run in isolation of each other without the need to be modified (*e.g.*, routing does not need to be modified to work with the firewall), and the changes needed in the database scheme are small.

5.4.1. Routing

As shown in Figure 5.5a, assume two hosts (*h1*, *h2*) are trying to reach each other. First, Gavel will have to locate the two hosts with the help of their IP addresses by querying the graph database using a *Match* statement. Afterwards, using the find-route routine built into Neo4j, it calculates and retrieves the shortest path between the two hosts and as a result, it

[1] https://github.com/engbarakat/Gavel

```
MATCH (h1:Host{ip:srcip}), (h2:Host{ip:dstip})
MATCH p=shortestPath((h1)-[:Connected_to*]->(h2))
WITH h1,h2, p
CREATE (h1)-[pa:Path_to
   {switches:[n in nodes(p)[1..-1]| n.dpid],
   ports:[r in rels(p)[1..]| r.port1]}]->(h2)
RETURN pa.switches, pa.ports;
```

(a) Cypher code snippet for finding a route between two hosts

```
MATCH (h1:Host{ip:srcip})-[r:Connected_to]->(s2:Switch)
REMOVE h1:Host SET h1:blockedHost
RETURN  DISTENCT s2.dpid;
```

(b) Cypher code snippet for blocking a host fix distinct

Figure 5.5.: Cypher code snippets

also collects all relevant path information (switches with ports that connect them ordered from source to destination). To reuse this information in future operations, Gavel stores it in the database as a relationship between the two hosts with a `path_to` label (cf. Figure 5.2).

This yields two advantages. First, installing a reverse path will be easy for Gavel by reversing the original returned switch list. Second, further optimization can be applied so that all forwarding rules for hosts that are connected to the same switch are summarized into a single forwarded message, using route summarizing techniques. This results in a lower number of OpenFlow rules in each switch and consequently saves TCAM space.

Note that in comparison to an RDBMS scheme (i.e.Ravel), Gavel reduces the query overhead significantly. For reverse path installation, a relational database approach will need to issue an additional explicit query for the same source-destination hosts, which is expected to accrue due to TCP three-way handshake to initiate the communication channel. While Ravel controller could implement route summarizing by finding all routes that are installed and share the same source and destination, this would introduce substantial query overhead. Additionally, before actually calculating a routing path, the request must be inserted into several tables to trigger relevant routines. After inserting the calculated route into another table, the information required to install OpenFlow rules along the path needs to be queried for. This involves each switch on the path to query the database again.

5.4.2. Access Control Firewall

Next, to test Gavel's ability to handle a finite state machine application we have implemented a simple stateful firewall application similar to the one implemented in [2].

Our firewall generally allows for access control in the network by managing the visibility of resources to applications. For instance, as described above, the routing application finds the shortest paths between two `host` entities in the graph. Disabling a host can be achieved by simply changing the label of the node from `host` to `blockedHost` as depicted in Figure 5.5b. In general, by changing the type or attributes of a node or edge in the graph, we can alter its visibility to applications.

Importantly, this concept yields isolation of the firewall from other applications. While the firewall needs to alter the database scheme *and* the routing application itself in state of the art (before finding a route, the routing application needs to check for blocked resources in a distinct table), in Gavel, resources are hidden by the firewall and directly not visible to the routing application. For instance, once the firewall blocks an edge in the graph by changing its type, the routing application will not find that edge again and will instead route a request via a different edge, if available, or return a no-route error if unavailable.

Moreover, to enforce the policy on any previously installed path, the application continues to locate all switches connected directly to that host and then installs the respective OpenFlow rules (drop packets targeted at that host) in these switches. As a result, no device will be able to reach the blocked host until a reverse (`unblock`) routine is called. Slightly more challenging is a scenario in which we want to block a path that is not installed yet. Here, the firewall will first compute the relevant path, and then change its type to blocked.

Applying these concepts using a relational DB based controller Ravel would require adding a column to the hosts' table to indicate each host status (i.e., blocked or not), and thus an alteration of the database schema is not avoided.

5.4.3. Load-Balancer

To demonstrate the ability to combine a high-level language (*e.g.,* Python) and Cypher, we also present a load-balancer application. The application cycles through a list of servers by picking one as a destination each time a path to the service is required. Each path is installed with a balance in mind, so the application needs to track the assigned servers in one cycle to assure this feature. Again, implementing the load-balancer does not require any change in the database schema or any other application.

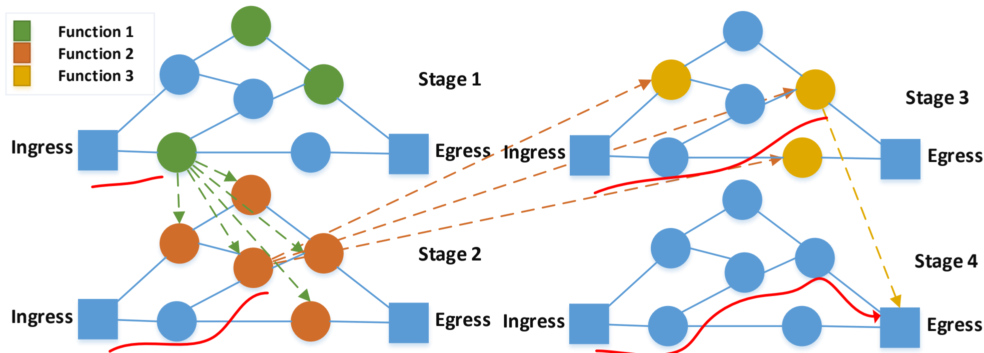


Figure 5.6.: ARS algorithm [1] chooses the shortest path for every next network function until it reaches egress point

5.4.4. Service Function Chaining

Service Function Chaining (SFC) [78] applies a series of network functions (NFs) to a flow. Assuming that the location of NFs is already known, Gavel can also support the ability to route flow through a series of such functions efficiently.

Essentially, what Gavel needs to do is to find the shortest path through all network functions from the source to the destination. Finding the best path from ingress to egress points and through specific nodes is an NP-hard problem [79]. For Gavel, we implement ARS [1], an approximation solution in which the path through the requested NFs is calculated step by step taking into consideration that path cost is a dynamic function.

Implementing this algorithm in Gavel is straightforward. Our application begins with predefined assigned locations for each NF (which could be changed based on the allocation criteria) and assigns NFs to each requested flow based on the policy accompanied each flow. When the request to steer a flow arrives, Gavel starts to find the shortest path to each switch connected to the first NF in the chain and keeps only the node with the shortest path. It iteratively does the same for the remaining NFs in the chain as depicted in Figure 5.6, and finally returns the path. In the graph database, Gavel creates a relationship for this steered path between the two endpoints to keep track of the path information. One last thing to mention here is that Gavel can calculate the shortest path in each step based on a cost function which could be any predefined value or a dynamic value updated with the help of the OpenFlow Composer (cf. 5.3.4) component.

```
MATCH (l:Slice{name:{slicename}})
MATCH (h1:Host{ip:srcip})-[]->(l)
MATCH (h2:Host{ip:dstip})-[]->(l)
MATCH p=allshortestpaths((h1)-[:Connected_to*]->(h2))
WHERE All (x in filter (x in nodes(p) where x:Switch)
WHERE (x)-[:Member_of]-(l))
WITH h1,h2,l,p order by length(p) Limit 1
CREATE (h1)-[pa:PathSlice_to{switches:[n in
nodes(p)[1..-1]| n.dpid], fports:[r in rels(p)[1..]|
r.port1],bports:[r in rels(p)[..-1]| r.port2]}]->(h2)
RETURN pa.switches, pa.fports, pa.bports, h1.mac,
h2.mac;
```

Figure 5.7.: Cypher sample to find a path between two hosts within a single slice l

5.4.5. Network Slicing

Finally, Gavel also supports network virtualization via slicing, which allows multiple applications or tenants to share and utilize the network infrastructure isolated from each other.

Creating additional nodes in the graph model could easily represent slices. Such a node would represent each slice, and the resources belonging to the slice would, in turn, have a `member_of` relationship to that node represented by an appropriate edge in the graph database. Note that no modification of the original database scheme is required in this case.

Afterwards, every Cypher call within this slice would be restricted to operations on nodes which have this relationship with as shown in Figure 5.7. Creating slices in Gavel could also be characterized by dynamic configurations (*e.g.,* link bandwidth). Gavel will keep track of such constraints dynamically by updating related specification in the graph database and push it immediately to the forwarding plane. Whenever it is impossible to reserve or comply with customers' requirements in the slice creation phase, Gavel will turn down the request. It is crucial that Gavel keeps enough resources to fulfill installed slices. To solve conflicts in slice definitions, Gavel can use existing approaches [11, 50].

Implementing such an application on state-of-the-art relational database controller [2] could be done using views technique. At first glance, that would be an easier and convenient way to implement slices. However things could affect performance when multiple views needed to be joined as unneeded tables would be queried, and unfortunately, DBMS optimizers cannot avoid this situation in addition to the problem we mentioned earlier re-

garding SQL optimizers are not aware of the graph nature of the data. Another implementation possibility would involve creating a new table to track all slice' memberships and another one to hold all slices information. Hence, any query would go through these tables to validate membership that adds more complexity to the running system. These possibilities are without considering adding columns as it would change the database scheme and bring us back to the first problem we are trying to avoid.

5.4.6. Summary

In summary, we have provided examples of how different types of applications could be implemented in Gavel based on the Cypher query language. The key advantages of Gavel over current relational database solutions are that i) applications now follow the logic of the network topology rather than data organization, ii) there is no need to alter concurrently running applications, and iii) it is only necessary to update the graph itself and not a multitude of tables. In the next section, we evaluate Gavel and these applications with regards to their performance.

5.5. Evaluation

In the previous sections, we have shown the Gavel network model and how programming a wide range of network applications is simplified with our approach. Throughout this section, we will show and discuss how these network applications perform in a Gavel-controlled network. We find that depending on the network topologies we deploy Gavel on, our approach decreases the latency of networking applications by up to several orders of magnitude.

5.5.1. Methodology

We first describe our evaluation setup before illustrating our results. For all experiments that follow, we have implemented Gavel components, models and applications in Python. We then deploy Gavel on an emulated Mininet [80] testbed and evaluate its performance on different topologies, which are characterized in Table 5.2. In particular, we checked the performance of Gavel in the context of both data-center topologies (FatTree [81] with k=16,32,64) as well as in ISP topologies (Geant and Deutsche Telekom topologies [82]) to cover a wide range of different network types and sizes. Our main evaluation metric is the performance in terms of the latency induced by the controller from the time it receives

Table 5.2.: Topologies used to evaluate Gavel

Topology	Switches	Links	Symmetric
FatTree 16	320	3072	Yes
FatTree 32	1280	24576	Yes
FatTree 64	5120	196608	Yes
Geant2012	40	61	No
Deutsche Telekom	39	101	No

the request to the time it ends processing this request. Finally, we compare our solution to Ravel [2], the current state-of-the-art solution for plain data representations.

5.5.2. Gavel's Applications

5.5.2.1. Single Routing

As discussed earlier, the routing application is one of the most important network functions and should benefit significantly from graph support. In a database based controller, finding the route between two hosts goes through multiple steps. These steps can be summarized as follows:

1. **Path Computation (PC)**: This step involves calculating the shortest path between the two nodes. In doing so, it measures from the time the controllers received the routing request until it finds the path in terms of path nodes and links.

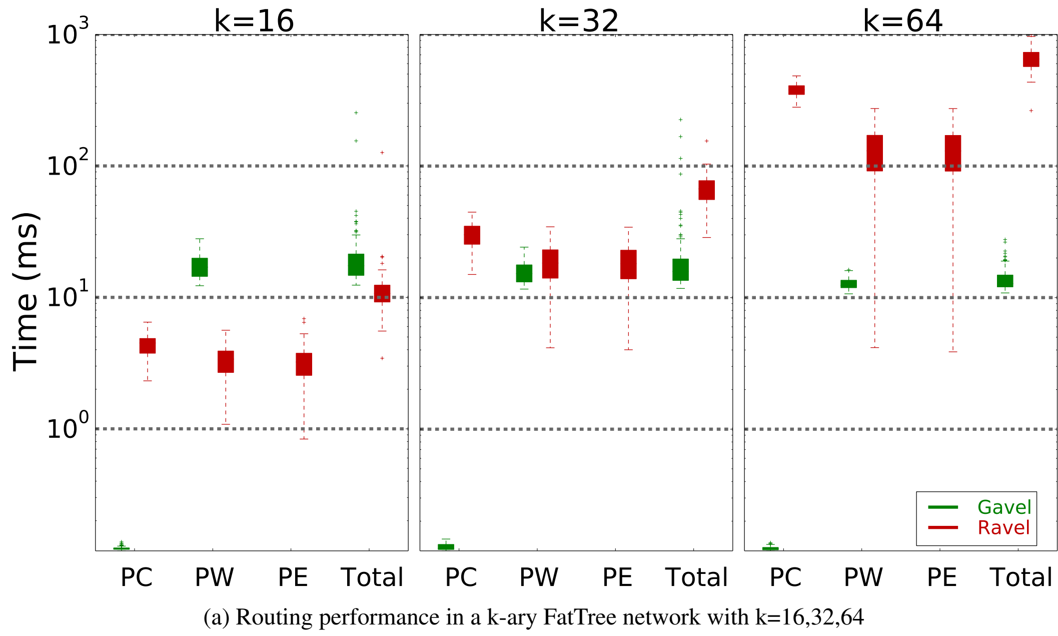
2. **Path Writing (PW)**: After the path has been computed, it needs to be written in the database. This measures the time needed to write query results back to the hard disk.

3. **Port Extraction (PE)**: Finally, to build the respective OpenFlow messages for all switches on the path, all port information for these switches needs to be extracted from the database.

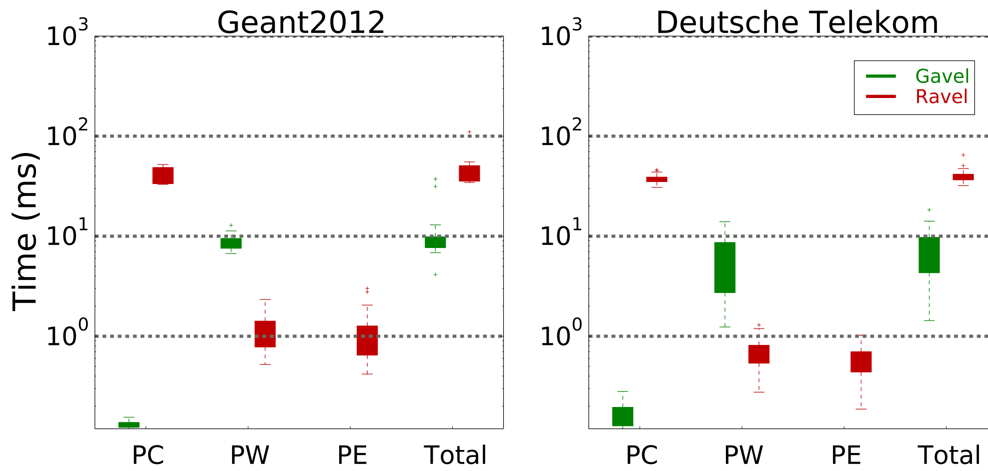
In Figures 5.8a and 5.8b we show our main results. Here, we list the measured delay for each component of the routing process as well as the total delay introduced.

We observe the following:

- Due to its native support for networks, Gavel's graph database engine can compute the path (PC) almost instantly, and almost two orders of magnitude faster than Ravel.
- The cost of all three operations (PC, PW, PE) remains constant within increasing net-



(a) Routing performance in a k-ary FatTree network with k=16,32,64



(b) Routing performance in ISP network topologies

Figure 5.8.: A comparison of the latency induced on different topologies by routing application in Gavel and Ravel, respectively.

work size in Gavel, while Ravel significantly suffers from the increased network complexity. For instance, when comparing different scales of FatTree networks, Ravel performs approximately two orders of magnitude worse in a 64-ary FatTree than in a 16-ary FatTree. For Gavel, we do not notice any difference in performance across different topologies. This indicates the ability of Gavel to scale without any major reported impact on performance.

- We see that path writing (saving to Disk) is *faster* in relational databases for small topologies. We believe that this is caused by highly optimized operations available in these databases (*e.g.,* PostgreSQL), while graph databases have not yet matured to that point. Future releases should resolve these issues.
- Another key benefit of using a graph database is that it does not need to perform any port extraction (PE) at all, while this remains a costly step in relational database controllers. The reason for this advantage is that the quick PC step already yields the relevant information in Gavel.
- In a FatTree topology with k=16, Ravel is slightly faster for the complete routing operation. Note that the size of the network does not cause this (as counter-examples see the performance for the much smaller ISP topologies in Figure 5.8b), but rather by the tree property of the FatTree topology, which results in comparatively short paths between most hosts and thus lesser database queries required in Ravel.
- In all other scenarios, Gavel significantly outperforms Ravel, in some cases by one order of magnitude or more.
- A final note regarding the error bars in Figure 5.8b and in Figure 5.8a are due to the random delay errors coming from the desk writing step (PW).

5.5.2.2. Service Function Chaining

Similarly, SFC is a concatenation of routing calls, with some additional database queries as indicated in Section 5.4.

Since Ravel does not support function chaining currently, we were not able to directly measure its delay. We thus extrapolated our results from the general routing application implemented in Ravel. We compare the empirical results obtained by the Gavel SFC application to a lower bound delay for Ravel. This lower bound includes only the cost of finding and storing the path between the source and the destination while passing through the requested middleboxes. It does not include other costs, such as selecting and finding the appropriate switch data from the database, which can be costly as well. The lower bound can thus be denoted as

$$c(sp) * (|NFs| + 1)$$

Where c(sp) is the average cost of finding the shortest path from the general routing experiment and —NFs— represents the total number of network functions on the path between source and destination. Figure 5.9 shows that, although a lower bound estimate for Ravel, Gavel outperforms the state-of-the-art in all cases except for smaller chains in a 16-ary FatTree. We again observe routing in larger topologies becomes impractical for Ravel (delay of $10^4 ms$ in a 64-ary FatTree), while Gavel is mainly independent of the topology size. Another observation noted in Figure 5.8b where we plotted Deutsch Telekom results, is the superiority of routing module in Gavel in asymmetric topologies (e.g. Geant2012 and Deutsch Telekom). Thus, we kept only Geant Topology as an example for asymmetric topologies.

5.5.2.3. Firewall

Differently, from routing, another aspect of Gavel is the capability to easily extend the graph database model in order to represent more entities needed by other applications. This extension should ideally not influence how Gavel is executing other concurrent applications. In the following, we will look at two different angles of this isolation feature. First, writing a new application should not require modification to any running application. This will be shown in the firewall application. Second, the performance of applications already running should not be affected by modifying the database scheme. We will show this by comparing routing time before and after applying slicing.

As we discussed before in Section 5.4, the firewall application will change the type of a node or edge in the graph structure, while in Ravel, blocking a host means inserting a rule into the firewall table and a table lookup each time the host is referred to in a rule. In Figure 5.10 we show the resulting delay for blocking a host (BH) and the reverse operation of unblocking a host (UbH).

Evaluating the FatTree topologies, we can confirm the trends seen in our previous experiments. Although Gavel is slightly slower for K=16, it performs much better in bigger networks. There are two factors behind this. The first factor is the writing delay in Gavel. It was the leading cause for Gavel's lower performance in K=16 (both systems do the tasks in ¡5ms). The effect of that result starts to diminish in bigger topologies, whereas other sources of delay begin to have higher impact and introduce the second factor, the delays induced by looking tables up for every firewall function call, which are triggered each time the routing application is called.

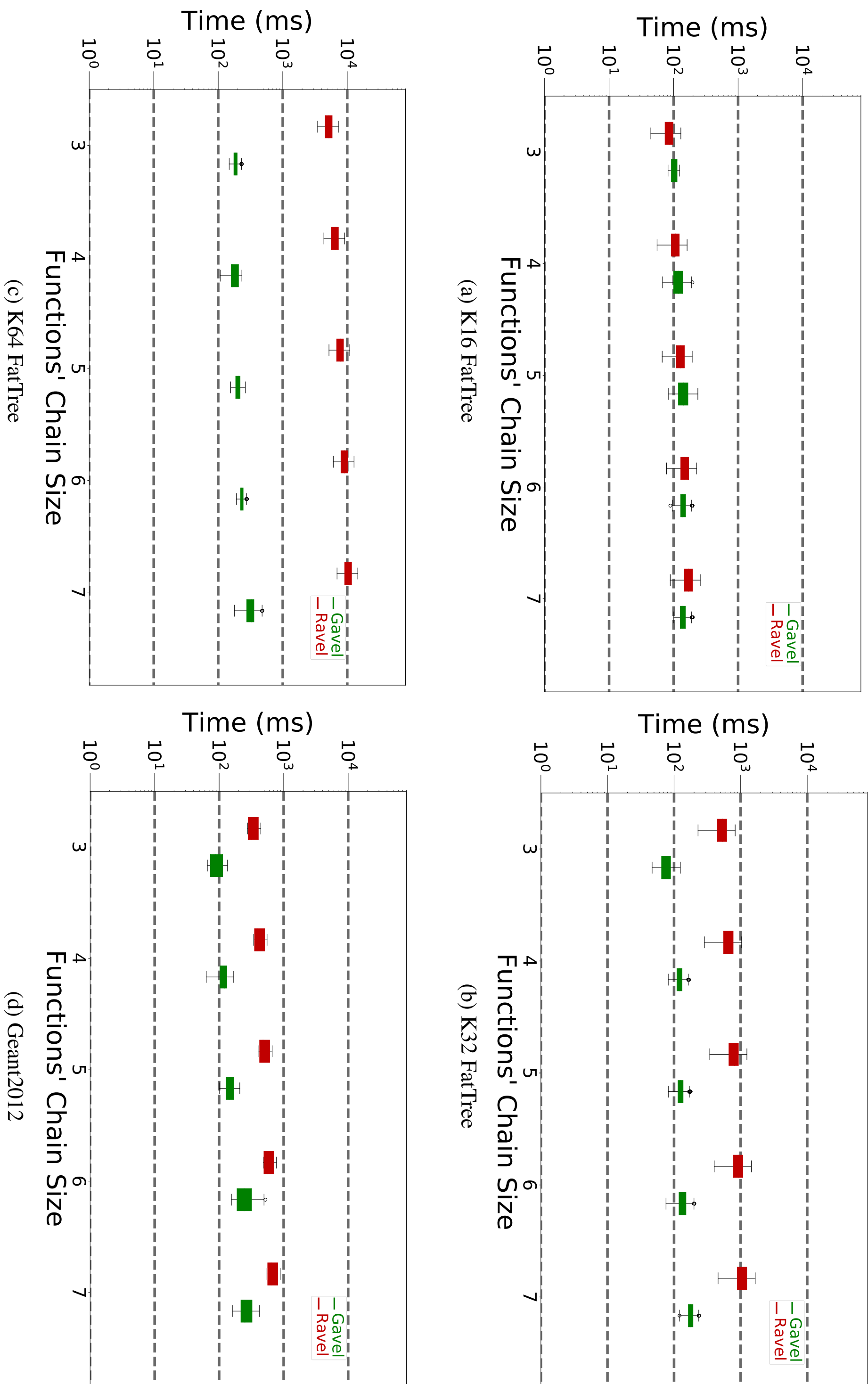


Figure 5.9.: A Comparison of the latency induced on different topologies by routing through different function chains in Gavel the lower bound for Ravel

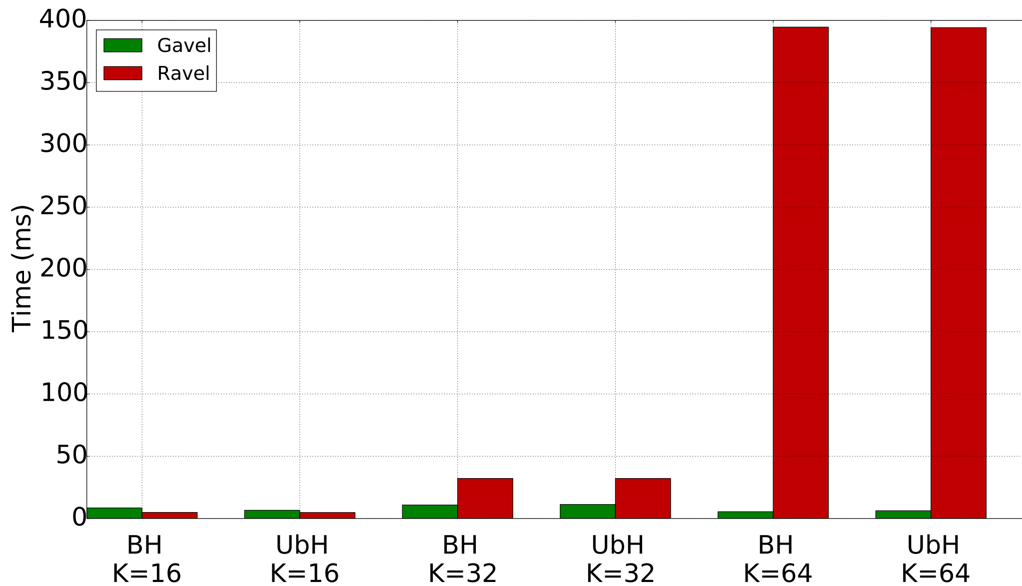


Figure 5.10.: A comparison of routing delay in combination with firewall routines for blocking hosts (BH) and unblocking hosts (UbH) in both Ravel and Gavel in k-ary FatTree networks with k=16,32,64

5.5.2.4. Network Slicing

Ideally, the addition of a slicing layer should have no implication on the performance of other network applications, i.e., the slicing layer should be transparent to these applications. To evaluate Gavel in this regard, we compare the performance of Gavel's routing application running in a network without any slicing present to the performance when different amounts of network slices have been instantiated. Concretely, we measure the delay of finding a routing path between the same (random) pairs of hosts in different topologies with and without the slicing application running and the respective additions to the database being made.

In order to obtain statistically robust results, we execute this experiment repeatedly for a large number of host pairs. Concretely, the number of host pairs ranged from 82 (GEANT topology) to 30000 (FatTree 64).

Figure 5.11 shows the effect of slicing on the computation delay values obtained by the routing application for an increasing number of slices for four different topologies. We observe that there is a slight trend towards a higher latency with more slices being introduced. However, this increase is in the vast majority of cases not statistically significant at the 0.05

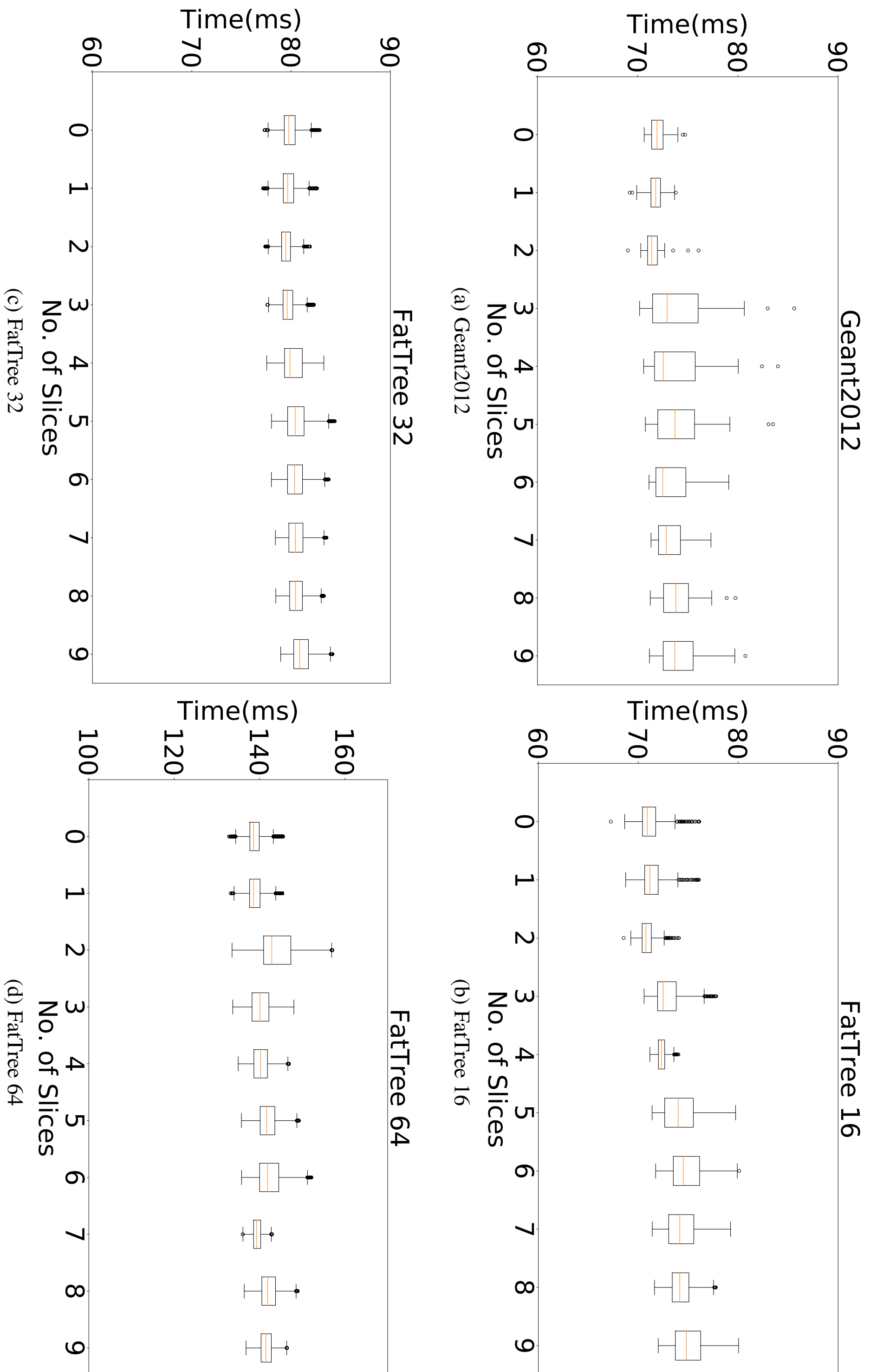


Figure 5.11.: A comparison of different delay time induced by routing application with(1-9)/without(0) slices in different topologies

significance level in a one-sided paired t-test. In most cases, we observe a stable delay with deviations (note that in some cases we also observe a decrease in the delay) ranging within 2-3ms.

5.5.3. Writing Network Applications on Gavel

In Section 5.4 we have shown that Gavel supports different types of applications to be quickly developed in a few lines of code in the Cypher programming language on top of its network model. Thus network administrators can keep only the network logic during writing network application on top of Gavel. We have demonstrated this ability with exemplary results in routing, access control (firewall), load-balancing, service functions chaining, and network slicing applications. We have shown that the programming complexity is significantly reduced over state-of-the-art solutions using relational databases as *fewer* and *less complex* queries are needed (especially with increasing application complexity). We interpret the reason behind the reduction of orchestration complexity as applications do not need to alter the logical flow of other applications, and all applications will automatically have access to the same network state. Recalling routing application from section 5.4, although four lines of code are enough for both Gavel and Ravel to find and store the path, Ravel needs more lines to iterate each switch in the path to retrieve the ports information to install the path.

Chapter 6

Future Prospects

In this chapter, we talk about the prospects of Gavel. We investigate the possibility to operate Gavel either with other northbound interfaces or in a different SDN environment (*i.e.,* SR). After that, we analyze the current limitations of Gavel and suggest some ideas for future extension.

6.1. Applicability of Gavel with Other SDN Environments

6.1.1. Gavel and SR

One of the advantages of Gavel is its flexibility to fit into different SDN technologies. In Part II, we show how we can adapt Gavel to work as a network controller in an SR environment. Although the core components are unchanged, other components that communicate with data plane have been updated to reflect the new SR environment. Gavel worked perfectly in the SR environment as presented in Part II.

6.1.2. Gavel and other Northbound Interfaces

As mentioned in Chapter 5, Cypher is used to query the database and send requests to the network controller in Gavel. Additionally, Java could be used as well to do the same tasks. Gavel uses Neo4j as a database management engine, and it offers a Java API as well. When there is a need to use another northbound interface, Java programming language could be used to integrate the desired abstract with Gavel.

Another form of integration could be realized if the needed routines are coded as *stored*

procedures and then plugged into the database engine itself. Cypher could be used after that to call these procedures. In this way, any new functions that are not available in Cypher's library could be added to Gavel.

6.2. Current Limitations and Prospects of Extensions

Gavel provides simple data representation that delivers higher network controller performance. Nevertheless, there are still challenges that need to be addressed. In this section, we discuss some of these challenges and suggest some extensions to enhance Gavel. Giving the results presented in Figure 5.8b, we can see how writing path information back to the desk takes more time than the relational database approach (*i.e.,* Ravel). As pointed in results' discussion in Section 5.5, the superiority of the relational database writing module could be explained by the maturity of relational database management systems.

To enhance writing to desk module in Neo4j, we could either customize the writing module or restructure the installation of the database engine. In the first suggestion, the customization would convert the database engine from general purpose database engine to a specific one, because any unneeded operations will be omitted and keep those that are related to the networking environment. The second suggestion could be realized by configuring the database engine to save its data on a RAM-based disk and write a tool to dump the data to the physical disk periodically. In this way, the rate to access the hard disk will be fewer which enhance writing speed and keep a version of the data saved for backup when it is needed.

Part II.

Addressing Northbound Interface Challenges in IPv6 Segment Routing

Chapter 7

Problem Statement

7.1. Introduction

As introduced in Chapter 2, SR is a variation of source routing that was presented not only to solve MPLS manageability problems but also to introduce networking programming to current network infrastructures [8, 20]. In Chapter 1 we elaborated on the importance of network programming in the current time and how it will play an important role soon regarding connecting people and services around the globe. Therefore, as SR introduced itself as future network technology, we have to study the current competencies it has in programming aspects.

In the IETF standard draft [20], network programming was introduced as a new feature in the IPv6 variation of SR. Commands that would steer and process traffic in the network are encoded as IPv6 addresses and stacked in the SRH as segments. Any router receives a packet with a destination address that matches any predefined function will be processed according to that. However, this is not enough yet to claim that end users can start writing their network programs on top of SRv6 networks. In this chapter, we explain the challenges and problems that are not solved with current approaches.

7.2. Challenges in Segment Routing Policy Composition

Even with all of these programming capabilities enabled by SRv6, network administrators still face the difficulty of manually constructing segments lists that fulfill their intents and policies. Taking a closer look at latest works, where SRv6 is in its ecosystem, such as [22–26], segments lists were composed manually. Such methodology would work simply in these projects giving the fact that network topologies used in the evaluation tended to be

small, however, in real operated network topologies manual composition presents various challenges in the context of composing network policies.

The first challenge comes from the manual operation itself as it is prone to human errors. Typing manually a list of SIDs (128 bits each) in the command line every time a user wants to send a policy entry to an edge router could increase the resulting probability of manual errors even when a network controller collects the SIDs. The second challenge is observed when certain network events trigger a change in the network topology. The response from network administrators, to update the segments lists which implement the various policies cannot be completed in real time. Even if we assume that IGP protocol would sufficiently handle such failure events due to its high reliability and convergence time, other events such as migrating network functions need immediate response to update affected policies as IGP protocols cannot help in such cases.

Other challenges could also raise due to improper SRv6 behavior usage. Decapsulation before encapsulation or process in order but with different protocols (IPv4 instead of IPv6 and vice versa) are good examples of what could happen when segments lists are composed manually. Also, more challenges with SRv6 behavior usage could be present when parameters are needed to be manually calculated, retrieved, and passed with SRv6 behaviors (e.g. tenant's ID).

Finally, running and managing networks in the presence of tenants can introduce further complexity in segments list management. Assuming that overlay service or VPN is already configured in the corresponding edge routers, administrators would need to attach related behaviors for encapsulation (e.g. `T.Encaps`) and decapsulation (*e.g.,* `End.DT6`) keeping parameters also incorrect value for each customer or tenant.

To summarize the challenges mentioned above, manual composing and management of segments lists are unrealistic given the new advancements in the field of computer networks (e.g. SDN). To make SR networks ready for Self-Driving technology [83], at least composing and managing segments lists and SR policies should be automated. This part of the thesis answers the question raised in the first chapter regarding automation and portability in 1.3.2 and in 1.3.3.

Chapter 8

Related Work

We show in this chapter other work that could share with our framework either the environment or the objectives. We start with SRv6 related works and highlight our work among others who are building systems on top of SRv6. Later, we discuss other works that share some objectives with Busoni, but they operate only with OpenFlow.

8.1. Segment Routing on IPv6

Since the introduction of SRv6 [8], the focus of researchers was mainly in the dataplane [22, 24–26]. They solved many problems regarding integration with existing networks, dataplane performance, or SR procedures optimizations. For example, in [22] researchers showed how to support SR-unaware network functions to reserve segment list integrity. Also, in [26] where the researchers proposed a new SRv6 behavior to ease writing custom functions. Other researchers were focusing on providing added-value services as in [24] or investigating a better way to send segments list to edge routers based on Linux [25]. Only a single work is close to what we propose in this paper: Software Resolved Networks (SRN) [21]. SRN presented an SDN-like architecture to enable network operators to specify policies to control the network. While they allowed end users to express their requirements, the authors used DNS service to convey these requests. If the end users start to use the real IP address instead, there is no guarantee that policy will be enforced. Additionally and in comparison to our framework, SRN's policy tools cannot be easily extended and need to be built from scratch to support other scenarios while in Busoni besides its support to multi-tenancy and VPN related policies, it could be extended given its software architecture is flexible to support different scenarios. Therefore, our framework is more appealing in real-world scenarios.

Another essential difference is in the operations: the process of finding and installing

path in SRN is triggered each time a new host tries to reach service in the network (reactive actions), while Busoni offers the possibility to configure the network proactively. Considering internal operations in the two approaches, we found that SRN almost follows the same steps and algorithms to handle path requests. Both frameworks use Dijkstra for finding the shortest path and `minseg` for encoding the path as segments. Therefore, the type of database used to store network information is the key to differentiate the performance of these two approaches. SRN uses a relational database to store network topology information while Busoni uses a graph database. Looking at the comparison results of the two databases in [27] and considering the networking environment, we can conclude that OVSDB, which is a relational database, would impact the total performance in SRN negatively. The graph database used in Busoni scales well with bigger topologies as it designed to handle massive complex graphs (e.g., social networks).

Continuing our comparison to SRN, we can see that the management of dynamic events is also different; our framework responds to various events, including network functions migration and not only router/link failures. While SRN can react to changes in the bandwidth of the infrastructure links (this feature could be addressed in the future release of Busoni).

8.2. Northbound Interfaces in SDN

After the introduction of SDN in [7], researchers started to develop different solutions on top of SDN. Direct management of the southbound interface (i.e. OpenFlow) was one of the main challenges during the early stages. Researchers responded early to this issue and presented many approaches to ease OpenFlow handling (i.e. northbound interfaces). These approaches could be categorized based on their end objectives. Some were focusing on optimizing resources reservation [16, 17], some supporting multiple composition [10, 15, 18], and others minimizing the number of forwarding rules in the dataplane [19]. However, all of these solutions were designed for OpenFlow devices, and the idea of reusing them for SR would require a redesign from scratch. The operation of OpenFlow and SR differs from each other. While OpenFlow protocol requests the controller to find the path for each new packet, SR involves only constrained shortest path requests and only communicates with edge routers. Other difference, each device in the OpenFlow data plane needs to keep rules for every packet traverses the device. SR keeps matching rules at edge routers which yields in fewer routing rules in the core routers which means better scalability.

Chapter 9

A Northbound Interface for IPv6 Segment Routing: Busoni

In this chapter, we present our SRv6 policy management and a northbound interface framework. We elaborate in details the design choices and the architecture of our framework. After that, we show the operation of the framework using three different use cases. The chapter concludes with profiling experiments to show the scalability and response time of the framework under various loads.

9.1. Introduction

To resolve the challenges outlined in Chapter 7 we present *Busoni*, a framework to compose and manage network policies on top of SRv6 networks. Busoni provides the needed programming functions for network administrators as a northbound interface on top of a SR controller. End-users can use Busoni to automate the generation of their policies. They can define endpoints in flexible terms as we show later, and write their functions or any special behavior that they want to apply to the flow between the defined endpoints. In the case of network dynamics or failure events, the framework will automatically update the affected policies and report any events for which Busoni failed to find enough resources that satisfied the policy's goals. Busoni makes the following contributions:

- **Automate segments list management:** Busoni exploits the benefits of the network controllers and utilizes a data store to keep track of the SIDs announced in the network. This feature allows the administrators to focus on network management goals rather than focusing on physical details of segments and their location. For example, an administrator can create a service network chain using network functions names instead of exact segment identifiers. This provides an opportunity for the network

controller to find a suitable path that implements this chain and thereby eliminate the overhead of administrators' efforts in finding the desired network functions that can produce a shorter path and use their SIDs.

- **SR policy management:** Network administrators can choose to utilize the predefined policies provided by Busoni or build a new policy on top of it. Busoni provides different tools to compose SR policies ranging from packet matching rules to functions that attach SRv6 behaviors to segments lists.
- **Responding to network dynamics:** Busoni updates any affected policy whenever there is a failure or updates in the network topology. This feature complements SR built-in reaction functions and keeps the installed policies resilience to dynamic events.
- **Multi-tenancy support:** Carrier grade networks or cloud service providers deliver overlay services to end-users. Busoni supports multi-tenancy as it embeds tenants IDs in any related SRv6 commands.
- **A prototype implementation and use cases:** Busoni is an open-source framework implemented on top of an SRv6 network running by Linux based routers. We also implemented different use cases, which present how Busoni's libraries could be utilized and further extended to support different policies on top of SRv6.

9.2. Requirements for Segment Routing Policy Framework and Target Scenarios

Current Segment Routing standardization efforts consider various scenarios; however, the current implementation of Busoni supports a subset of these scenarios. A modification to the source code would be required to run Busoni in other scenarios. The assumptions made in Busoni are:

- **Single SR domain:** Busoni assumes a single domain which is responsible for delivering packets from ingress to egress points within the domain's borders. Interdomain routing and passing service information between different domains would be possible with some modifications.
- **Basic Policies:** This refers to any policy when there are no multi-tenancy related constraints. Traffic engineering and SFC are good examples of what we refer here as basic policies.
- **Overlay Policies:** In different network types, tenants expect to manage their networking infrastructure independently. This translates to the possibility of applying different policies on the same router.

In other considerations, designing the policy framework should account for:

- **Simplicity:** The idea behind providing northbound interfaces on top of network data plane is to ease policy writing. Such interface should hide ugly details of forwarding devices and provides enough means for end users to write, manage and validate policies.

- **Automation:** As discussed earlier, manual intervention is a major contributor of time wastage, and it is prone to human errors. Automation should minimize such interventions and allow network administrators to focus on management aspects.

- **Expressiveness:** Libraries and functions provided by the framework should help network administrators express different scenarios and conditions. Although a northbound interface is not a purely human language, it is still possible that high-level computer programming language expresses end-users needs.

- **Extensibility:** The framework should provide means to allow future extensions without any need to do a complete re-writing to the source code. Network policies are evolving [27], thereby administrators should be able to simplify the process of extending the framework to implement new policies.

9.3. Busoni Architecture

In this section, we describe the architecture of Busoni followed by the design details of our framework. We begin by positioning Busoni in a normal SDN operating environment including its internal design. Then, we elaborate on the high-level APIs that end-users and/or network administrators can use to compose their policies. This is followed by a more in-depth explanation of how Busoni compiles the users' policies to generate segments lists followed by how these policies are pushed to the related routers.

9.3.1. Overall Architecture

Busoni is a northbound interface for SRv6 networks. A normal network environment, where Busoni operates, is similar to a network depicted in Figure 9.1. This is similar to the SDN [6] reference architecture, which includes network controllers, protocols to communicate with end-users (i.e. northbound interface) and data plane (i.e. southbound interface). Busoni should be able to communicate with a network controller to query and save needed infor-

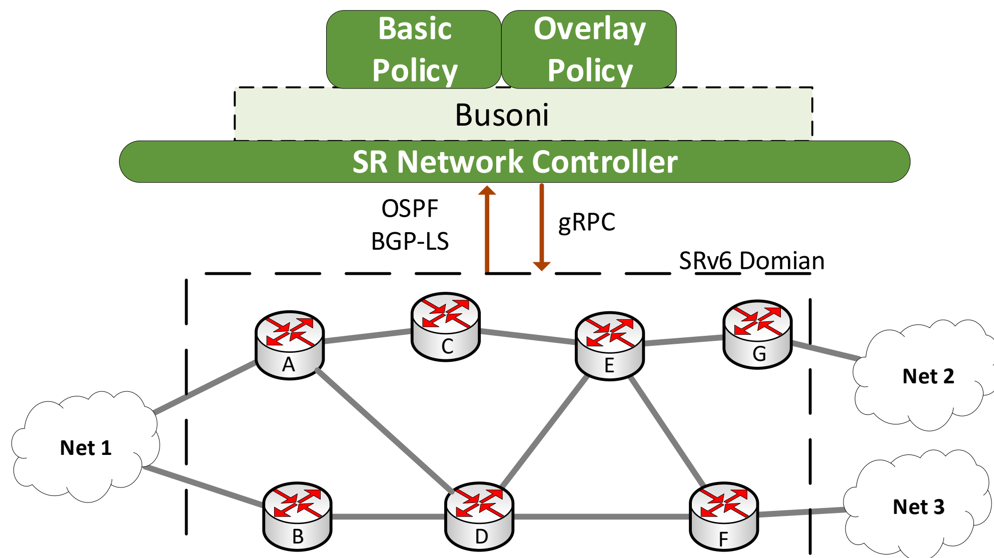


Figure 9.1.: The position of Busoni framework in a SRv6 network

mation for its policy management functions. It is crucial to expose network topology information to Busoni in order to run network management-related tasks (e.g. finding paths). Busoni also needs a data store to track installed policy information and to empower itself with the capability to respond to any raised events. To eliminate the overhead and to save time in moving the data between the two data stores (e.g. Busoni and the controller), we moved Busoni's datastore to the controller.

The networks controller should be able to collect the network's statistics from an SRv6 data plane using available southbound interfaces (e.g. OSPFv3 or BGP-LS). The controller also has to push the commands or segments lists to the edge routers using the available protocol [25] (e.g. gRPC or SSH). Although in our implementation we used a single choice for each of the functions mentioned above (OSPFv3 and gRPC as SBI), other choices can easily be incorporated in Busoni with minor code modifications. Using a specific protocol (e.g. OSPFv3) to run one of the corresponding tasks does not affect how Busoni processes and installs policies. There is a clear separation between layers in the SDN architecture. Busoni interacts with the network controller without knowing how the controller learns the network topology. In order to understand the internal structure of Busoni's operation and its mechanism in handling policies, we elaborate on the internal structure of its individual software components. As shown in the Figure 9.2, there are three main subsystems in Busoni, *SRtypes*, *Path Computation*, and *Database Middleware*. They interact between

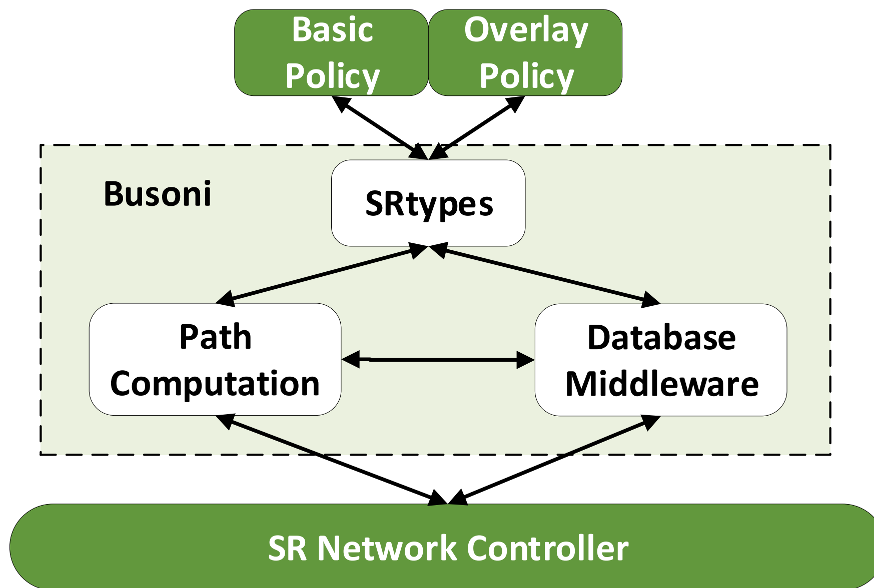


Figure 9.2.: Major software subsystems of Busoni

each other to provide the full fledged functionality of Busoni.

The first component is *SRtypes* which is the entry point to the framework. It contains the main policy class which users would use to build their policies. It also contains the basic types needed by Busoni to hold policy's components (e.g. SRv6 Behaviors) including the class `Match` definition which is used to define matching criteria for incoming packets. This component interacts with the other two subsystems to perform its functions. It uses *Path Computation* to calculate and retrieve a possible shortest path according to policy's conditions. The *SRtypes* also communicates with *Database Middleware* when it needs to save or retrieve any policy information to/from the data store.

Path computation functions are implemented in the second component. It first checks what type of path is requested (e.g., SFC) and calls the appropriate functions accordingly. Whenever it needs to save the computed path to the data store, it communicates with *Database Middleware*. Busoni, for now, allows four different variations of path finding queries: a simple path, a path with QoS only requirements, a path with SFC only requirements, and path with both QoS and SFC requirements. In a simple path case, Busoni would find the shortest path between all ingress and egress points, and it could give the same path as the IGP routing protocol. In the second path finding query, when a QoS requirement should be considered, Busoni would query all possible paths which fulfill the QoS require-

```
def __init__(self, match, qos=None, networkfunctionslist=None, nflistordered=True,matchonsrc
    = False,id=0)
```

Listing 9.1: The construction function of class `PolicySR`

```
Match(srcIP=IPSet(IPRange('C:1::1', 'C:5::1')), dstIP=IPSet(IPRange('C:6::1', 'C:A::1')))
```

Figure 9.3.: Using `Match` class to define source and destination addresses

ments and retrieve the path with a minimum cost. However, in the case where some network functions should be included in the path, finding the shortest path would be an NP-Hard problem [79]. Therefore, Busoni uses a heuristic algorithm proposed in [1], the ASR algorithm, where the final path is composed of shortest paths between waypoints (e.g. routers that host the network functions). In the latter path query type, where QoS and SFC should be considered while finding the path between ingress and egress points, Busoni combines both queries described in the second and third cases.

Finally, the third component (*Database Middleware*) holds all functions and event handling methods that need to interact with the datastore directly. It operates as an interface to ease database interaction with other components. It also listens to any dynamic events and responds by calling the corresponding handler, and whenever it needs to update a policy, it communicates with *SRtypes* to launch finding path function for the update procedure.

9.3.2. API Policies Composing

Busoni provides a standard policy class `SRpolicy` which contains the necessary functions needed to compile submitted policies. End-users inherit this class, define their class, and mainly extend the `eval` function which is automatically called when users instantiate an object from their defined class. Before executing the `eval` function, Busoni needs to do two things. First, it finds a corresponding path that represents the requested policy (e.g. adhere to QoS or go through network functions). Second, Busoni will have to encode the path using available SIDs.

In Listing 9.1, we see that `init` function takes many arguments that help in defining customized policies. The first argument defines the matching criteria that relate to source and destination network layer 3 and layer 4 specifications. For example, both source and destination could be determined using a range of IP addresses. `Match` class is flexible and provides through defined keywords (*e.g.,* dstIP, dstport etc.) a powerful tool to define custom matching rules like exclude a specific IP address. Users would use native python libraries to define their criteria without a need to learn any specific syntax.

The second argument holds *Quality of Service* (QoS) specifications defined by the user's policy. It could relate to any QoS metrics (e.g. bandwidth or latency), where the user has the flexibility to define what is needed in the policy. In the implementation of Busoni, we used a score metric where high score value reflects low latency and vice versa. Any other metrics or a combination of them would be used with some query updates to maintain which condition is preferred, low or high value. It is also an optional argument, and thereby it is not mandatory that policy has some QoS specifications. After that, there are some arguments related to service function chaining and whether they should be visited in order or not. An NFV management framework should provide beforehand which network functions are running in the network and feed the network controller with functions related information such as functions name and SIDs. Data-plane routers on their side, using the IGP protocol, broadcast network functions they host (virtualized or stand-alone). The last arguments determine if routers should match incoming packets against the source IP address or not, tenant ID if any (VPN user) for unique routing table matching when packets exit the network domain, and a policy id which is used internally for dynamic updates.

Users after passing the above inputs, need to declare any special handling needed for their packets. To understand what "*special*" means in this context, we elaborate in Section 9.3.4 how the basic class compiles the policy requirements to generate related segment commands including the *special* handling requested.

9.3.3. Encoding Path Nodes as Segments

One of the fundamental problems in SR is how to represent any calculated path using segments. The idea in SR compared to other source routing protocols is to attach *only* the *needed* detours to reflect policy's requirements rather than attaching all path nodes. As a practical example, in the case, we discussed in Section 2.2 where a controller needs to find a path between sites A and B passing through an SFC. In such a case, the path would be $\texttt{A,FnI}_\texttt{A}\texttt{,C,E,FnII}_\texttt{E}\texttt{,F,Net3}$ and hence the minimum number of segments that would represent the path correctly is $\texttt{FnI}_\texttt{A}\texttt{,FnII}_\texttt{E}\texttt{,F-Net3}$.

Even though most of proposals [84–87] address the problem for MPLS-SR, `minseg` algorithm [87] considers adjacency segment [8] and SRv6, thereby fits with our framework. The main idea of the algorithm is to check if the shortest path between two nodes belonging to the current path, will traverse any middle points in the same path. The `minseg` algorithm was modified to fit in our network model by ignoring network functions' relationships and use only routers' to calculate the correct shortest path between routers.

```
self.insert_behavior_first_segment("T_Encaps")
self.insert_behavior_end_segment("End_DT6",self.vpnuser)
```

Figure 9.4.: Using `eval` function to add custom packet handling

9.3.4. Busoni in Action

To generate a correct list of segments that represents what a user wants from the network, Busoni has to perform some functions. These functions begin with finding a path then encode it as segments (i.e. SIDs) and ends with performing special routines defined by the user (e.g. adding some SRv6 behaviors).

Busoni starts working just when the basic class `PolicySR` is inherited, and its constructor function is called. It first tries to find a path between source and destination addresses that satisfies the policy's conditions, and thereby it sends a request to the *Path Computation* subsystem.

The second step after finding the path is to encode it using segments. In this step, Busoni calculates the minimum number of segments needed to represent the whole path. As described earlier in the algorithm description, middle waypoints that represent network functions are also considered.

After that, when the segment list is available, Busoni calls the `eval` function which contains any custom actions that should be executed before sending the segment list to ingress routers. For example, end users could choose a behavior, from available SRv6 behaviors, which are supported by data plane routers, to be added and concatenated to the segments list. A practical example we can mention here in the multi-tenancy management where users would attach `T.Encaps` and `END.DT6` at the beginning and end of the segments list respectively to get VPN service. Such addition to the segments list would trigger a flag to let *Database Middleware* subsystem to generate proper southbound interface commands and ensure that policy is executed correctly in the data plane.

In the last step, Busoni will store the policy information in the database. This information contains matching criteria composed by the user, the calculated path, the optimized segment list, any path requirements specified by the user (either QoS or SFC related), and the policy ID. Maintaining this information is essential to allow any future updates that could be triggered as a response from network dynamics.

9.3.5. Data Store

To keep data integrity, track installed policies, and to respond to network events, Busoni uses a data store (i.e. Database). Busoni uses a graph database [39] to model and save network environment data. This option allows Busoni to recover from an outage and reload the database and synchronize any network updates. In this database model, nodes represent main network components either physical (e.g. routers) or virtual (e.g. policies), and edges represent the connection between the nodes.

Whenever a path is calculated between two points, it is stored to be used later as one of the policy information. Moreover, when Busoni saves a policy in the database, it makes sure that all network functions and routers that are part of the policy path are connected to the policy node. Thereby, if any event regarding these components is raised, Busoni would find it easy to update the policy according to the raised event.

9.3.6. Responding to Network Dynamics

As mentioned before, Busoni responds automatically to network topology changes and updates any affected policies. In SR there are two ways of responding to dynamic events especially router or link state changes. First one is the default IGP related response where routers in the data plane will update their routing tables to reflect the new change in the topology. The second way is using *Fast Re-Route* (FRR) technique in which a pre-computed backup segment list takes action in less than 50 ms [32].

However, depending on IGP features only to adapt with network dynamics is not enough, and in many occasions, the new shortest path would fulfill the connectivity requirement but fail to comply with other possible requirements such as minimum bandwidth or any other QoS requirement. Additionally, there are other dynamic events where IGP nor FRR cannot be helpful such as these events related to moving some network functions around the topology or spawn new network functions to respond to high demands. In such cases, although packets would arrive the old hosting nodes (FRR will not be triggered due to NFV events), these nodes would either not able to locate the network function in case of migration and eventually drop these packets, or route them to overloaded network functions which leads to overloaded states and dropping packets eventually.

As discussed earlier, manually responding to such situations is impractical, thereby automating the response is crucial by listening to network fired events and act accordingly. The network controller will handle the physical implementation of the listening procedures and use available protocols to detect any related topology change. When an event is detected, the controller will update the data store and send a signal reporting the event's information.

```
from SRtypes import PolicySR
class BasicPolicy(PolicySR):
    def __init__(self, match, qos, vnfs, nflistordered=True, matchonsrc=False,id=0):
        super(BasicPolicy, self).__init__(match, qos, vnfs, nflistordered, matchonsrc, id)
    def eval(self):
        pass

class OverlayPolicy(PolicySR):
    def __init__(self,match,qos,vnfs,nflistordered=True,matchonsrc = False,id=0,vpnuser=0):
        self.vpnuser = vpnuser
        super(OverlayPolicy,self).__init__(match,qos,vnfs,nflistordered,matchonsrc, id)
    def eval(self):
        self.insert_behavior_first_segment("T_Encaps")
        self.insert_behavior_end_segment("End_DT6",self.vpnuser)
```

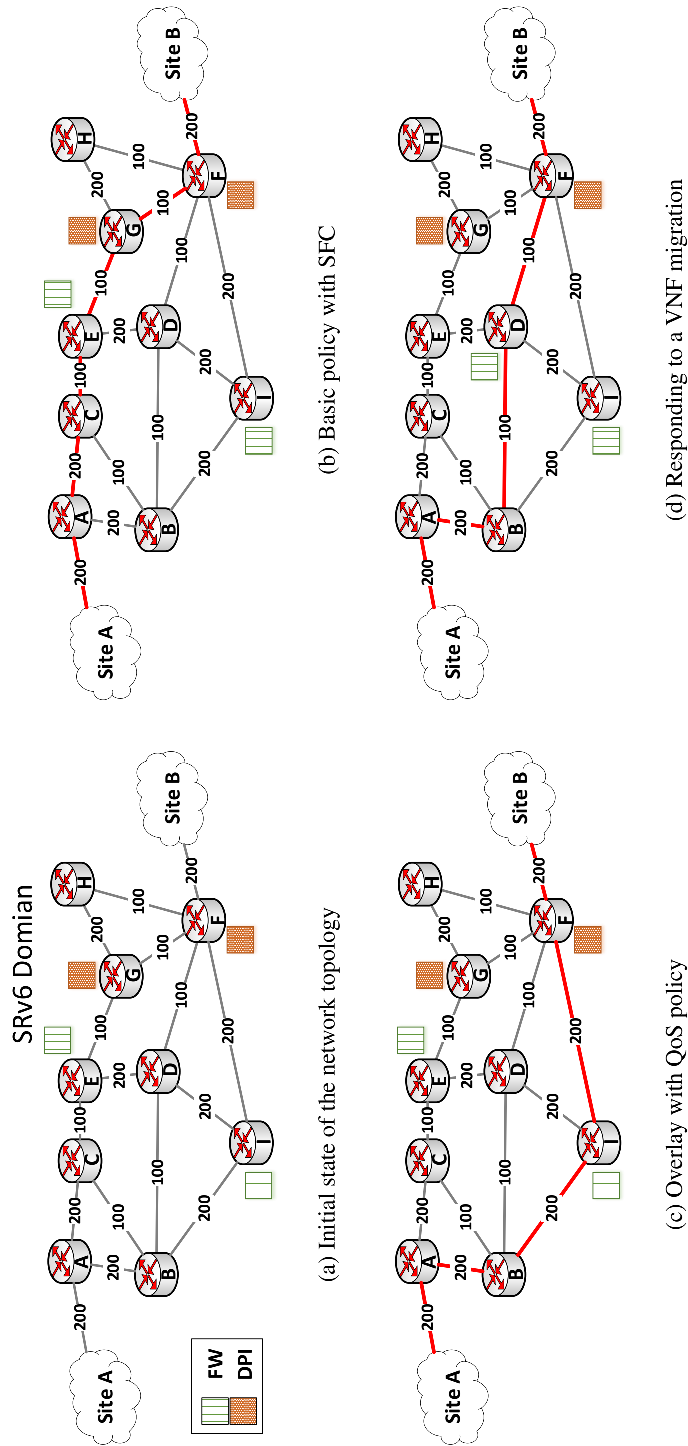
Listing 9.2: The definition of the classes used in the use cases

Listening to such signals, Busoni will query the database to find the affected policies and then updates them according to their conditions. In case it is not possible to find a path that fulfills policy's requirements giving the updated network topology, the framework will remove the policy and report back to the user.

9.4. Use Cases

In this section, we show the functionality of our framework, Busoni, using three realistic use cases. The first use case is similar to the scenario described in Section 2.2, while the other two use cases highlight the use of the framework under multi-tenancy and responding to network changes situations. In all three use cases, we assume an SRv6 SDN-based environment as a single domain and the initial network configurations in addition to network functions settings have been processed before the framework could be used. This initial configuration process also involves installing SRv6 behaviors in each node hosting a network function. We also assume that no requests would arrive from Busoni until the network controller has finished fetching the whole topology. Although any requests that arrive while the topology view is not complete would go through several updates cycle until the topology view at the network controller converges, we prefer to give the network controller enough time to load all topology information and set initial values. For the sake of demonstration, all scenarios are based on a simple network topology depicted in Figure 9.5a.

In the topology, there are two types of network functions Function 1 (firewall) and Function 2 (deep packet inspection) each hosted on a separate router. There are also two network sites; site A connected to the domain using router A and site B connected to the domain through router F. We assign bandwidth values for the links (in megabits) which will be used later to demonstrate a QoS based policy.



(a) Initial state of the network topology

(b) Basic policy with SFC

(c) Overlay with QoS policy

(d) Responding to a VNF migration

Figure 9.5.: Illustration of different use cases including initial state of the network topology

```
m = Match(srcIP=siteAIP, dstIP=siteBIP)
```

Figure 9.6.: Instantiating match object for the use cases

```
p1 = BasicPolicy(m, None,[NF['FW'],NF['DPI']])
```

Figure 9.7.: Instantiating an object for the first use case

9.4.1. Basic policy with SFC

As we defined earlier in Chapter §7, *Basic Policy* is any policy that does not have any multi-tenancy requirements. In this use case, we assume a situation where a network admin would like to steer the traffic between site A and B through two network functions (e.g. firewall and deep packet inspection). The starting point in Busoni is to define a class to hold the policy description. As shown in Listing 9.4, we defined a class `Basic Policy` to represent a container for such type. After the class definition, the network admin should start by defining the match object to specify the source and destination addresses as depicted in Figure 9.6. After that, she/he would instantiate an object from the class and pass the network function names (FW, DPI) as shown in Figure 9.7. Here, it is not necessary to determine which host is hosting the network function. This means, Busoni will choose the hosting router that makes the total route *better* according to what we discussed earlier about path computation in Busoni (Section 9.3). There is no need to call or define further functions as the path gets calculated and installed automatically after instantiating an object from the policy class. The Figure 9.5b shows the computed path between the two sites which traverse the two functions as requested by the users. Although the path would be A, C, E, FW_E, G, DPI_G, F, Site B, the segment list will be FW_E, DPI_G, Site B. This shows the gain in header size achieved with SR. The segment list would be sent to router A only, as it is the only ingress point for site A.

9.4.2. Overlay with QoS Policy

Let us consider the network topology in Figure 9.5a, a network admin would like to implement a quality of service policy where packets between the two sites must use only links with a minimum bandwidth of 200 Mb/s. In this use case also, the traffic belongs to one tenant where it has a special lookup service at the egress router. Therefore, the SRv6 behav-

```
p2 = OverlayPolicy(m, QoS(bw=200), None, vpnuser=102)
```

Figure 9.8.: Instantiating an object for the second use case

iors `T.encaps` and `End.DT6` are needed at ingress and egress nodes respectively. Setting up the decapsulation and the tenant routing table at the egress points should be done during the VPN installation phase.

Following the same steps we did earlier, the network admin would first define the policy class (Overlay Policy) as shown in Listing 9.4. Then, giving that reason, which we need to apply two SRv6 behaviors, the network admin will use the `eval` function to define these actions. As shown in the code, `T.encaps` will be inserted in the top of the segments list and the `End.DT6` with tenant ID as a parameter will be attached at the end of this list. Once the admin creates an object from this class passing the tenant's ID in addition to other inputs as it is shown in Figure 9.8, Busoni will start processing this policy. The path in this case according to the input topology would be `A, B, I, F, Site B` as shown in Figure 9.5c and therefore the segments are `T.encaps, I,` $\texttt{End.DT6(102)}_{\texttt{F}}$. We can notice that adding the node I to the segment list will force the packet to go from A to D following links with the minimum required bandwidth. Also the `End.DT6` behavior takes the name of the table or the tenant ID `102` which will be used to do the special look up that reflects the tenant's settings.

9.4.3. Responding to a VNF Migration

We show in this use case how Busoni will respond automatically to a VNF migration. Once an event took place, the controller will get the event and trigger Busoni to update any affected policies.

Considering the first use case in Figure 9.5b, the network function FW hosted in router E will be migrated to router D. The controller will be notified of this event and hence will trigger Busoni to respond. First, Busoni will have to find affected policies and retrieve its match conditions and their information. Busoni will find that there is an already installed policy that uses the function which was hosted in router E. Then, it deletes the policy from the database and starts the update procedure immediately by calculating a new path that satisfied SFC conditions and the match conditions. The path as shown in Figure 9.5d would be `A, B, D,` $\texttt{FW}_{\texttt{D}}$`, F,` $\texttt{DPI}_{\texttt{F}}$`, Site B` and hence the segments list that represents the updated path is $\texttt{FW}_{\texttt{D}}$`,` $\texttt{DPI}_{\texttt{F}}$`, Site B`.

Summarizing this section, the use cases show how it is simple to define policies using service names instead of worrying about low-level details like the actual SIDs for the network functions and behaviors. Busoni takes care of the necessary hard work to keep policies updated and will respond to network events that affect the installed policies.

9.5. Evaluation and Discussion

In this section, we present the details of the implementation of our framework and evaluate it. We first show our implementation's choices and elaborate on the lab setup and settings which were used for the experiments in these evaluations. After that, we explain the evaluation of our framework concerning its scalability and its response to dynamic events. We follow up by discussing our findings from the results.

9.5.1. Implementation and Lab Setup

Busoni is implemented on top of an SDN controller and an emulated Geant-2012 network topology [82]. We wrote the core components of Busoni using *Python* in around 880 lines of code excluding the supporting files for use cases and evaluation scenarios. Busoni uses *Gavel* [27, 88] which have been introduced earlier in Part I. Busoni uses built-in functions provided by Gavel which are graph-native functions, and thereby it presents highly scalable network functions. These functions support and contribute to Busoni's scalability. To make *Gavel* manage SRv6 networks, we updated the datastore by implementing SRv6 information like SID.

Regarding our evaluation environment, we used a VM hosted in a private Xen cloud. The VM runs Ubuntu 16.04 OS and has 11 GB of RAM with 4vCPUs and 50 GB of storage. We used an emulated real-world topology (i.e., Geant-2012) to populate the database with 40 nodes and 61 links. We defined eight different policies as shown in Table 9.1 to use them in the evaluation experiments. To feed the network topology with network functions, we randomly host three network functions (fn1, fn2, and fn3) at three randomly chosen routers.

9.5.2. Scalability

We start our evaluation study from the first angle, the scalability of the framework. We study this feature by measuring the compilation time of different policies batches using wall clock time. We increase the size of the batch and read how much time is needed by

Table 9.1.: Summary of the policies used in the evaluation

Policy	Type	SFC	QoS
A	Basic	No	No
B	Basic	[fn1, fn2, fn3]	No
C	Basic	No	Yes
D	Basic	[fn1, fn2, fn3]	Yes
E	VPN	No	No
F	VPN	[fn1, fn2, fn3]	No
G	VPN	No	Yes
H	VPN	[fn1, fn2, fn3]	Yes

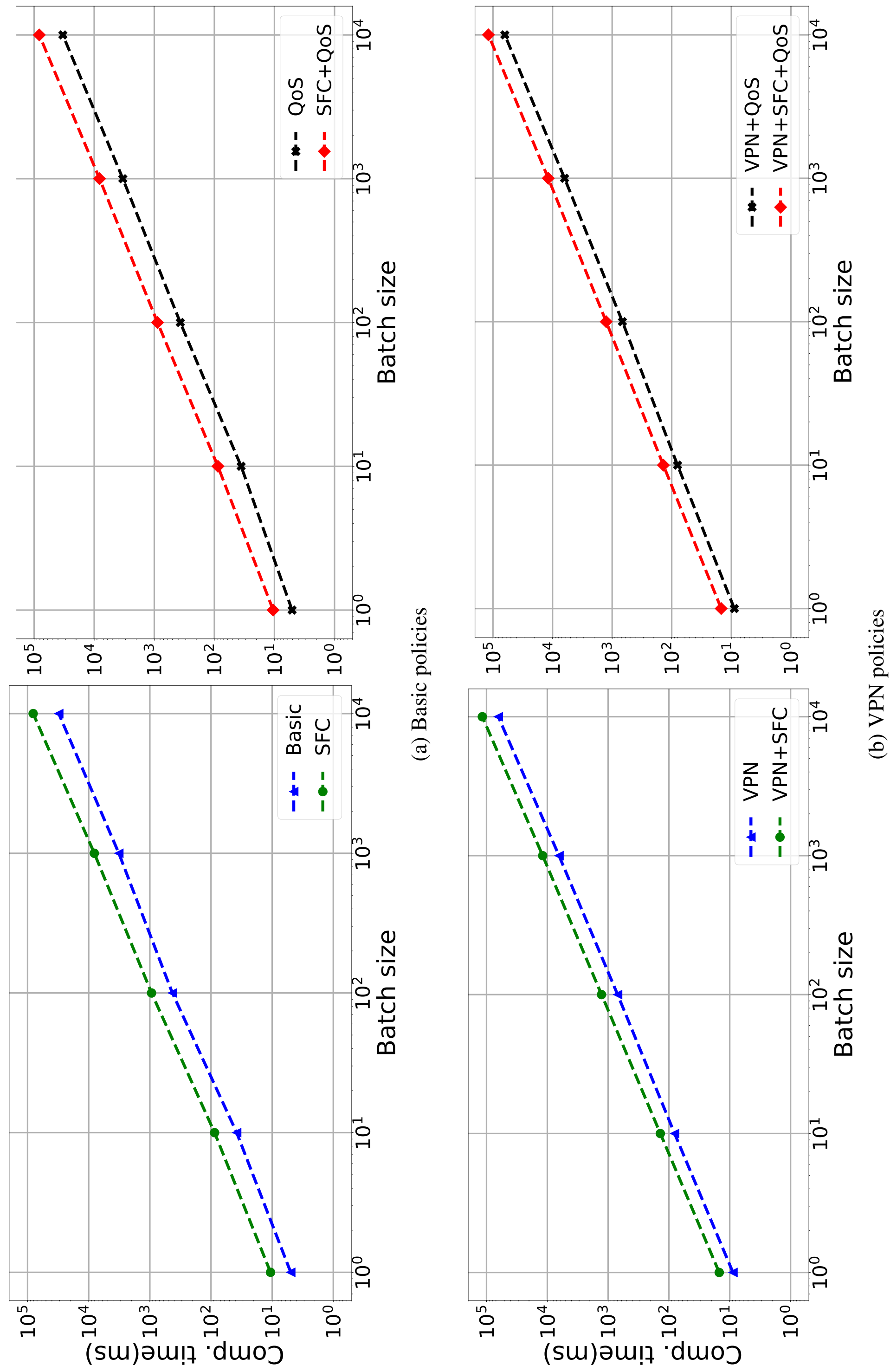
Busoni to process the submitted polices and find the correct segments list that represents each policy.

In Figure 9.9, we show the results obtained after running batches of 1, 10, 100, 1000, and 10000 policies. We plot the mean value (for 1000 runs) which is the time from the batch submission until Busoni finishes processing all requested policies. From the depicted figure and reading the summarized results in Table 9.2, we observe the following.

- Busoni scales linearly with the submitted load. As long as there are available resources, the relationship between the batch size and total time to process this batch is linear as shown in Figure 9.9. This finding shows that our framework could scale up to hardware resources' limit without any negative effect on the compilation time complexity.
- The average time to process a single policy in a batch is lower than the processing time for a single submitted policy. The main reason is that database cache is shared among different queries which save time needed by the data store to locate nodes and process their data.
- Adding SFC request to a policy increases the compilation time. For instance, the time needed to compile policy B is more than A, and the same is also true for policy F and E. This increase is expected as the path computation complexity gets higher when waypoints are needed to be traversed (*i.e.,* network functions).
- Adding extra processing request in function `eval` (i.e., embed tenant information) increases the compilation time. Comparing policies (A, B, C, D) vs (E, F, G, H), shows the $\tilde{4}$ ms difference. This increment is also expected due to the extra commands.
- Looking at the coefficient of variation for the different batch sizes we observe a little variation with batch 10. Other than that, Busoni processes incremental load requests with stable compilation time.

Table 9.2.: Average compilation time (ms) and coefficient of variation (%) for every policy with incremental batch size

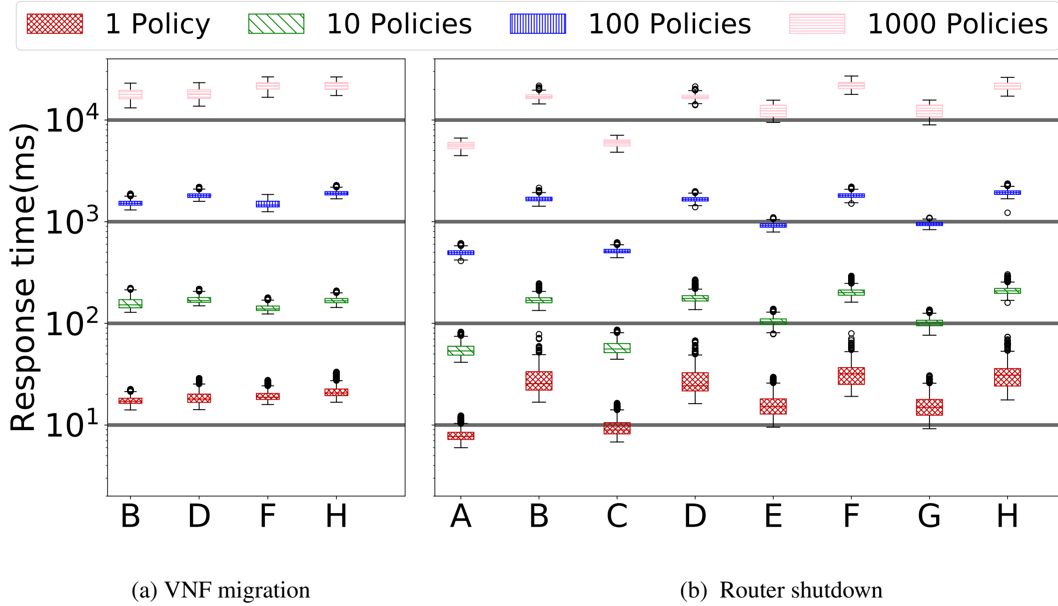
Policy	**A**		**B**		**C**		**D**		**E**		**F**		**G**		**H**	
Batch size	mean	CV	mean	CV	mean	CV	mean	CV	mean	CV	mean	CV	mean	CV	mean	CV
1	4.96	16.53	10.62	14.19	5.08	16.60	10.50	13.16	8.85	14.35	14.89	12.03	8.91	13.47	14.79	11.19
10	37.65	42.74	87.33	33.45	35.82	28.31	87.18	16.86	80.71	31.39	138.40	22.15	79.94	32.57	138.16	21.11
100	311.87	18.87	830.53	12.60	328.60	12.51	828.65	8.77	673.41	5.74	1226.86	6.43	654.55	6.24	1233.30	6.33
1000	3217.12	10.01	8109.55	7.55	3339.43	9.29	8150.13	7.86	6415.46	4.36	11896.34	5.18	6285.12	4.97	11781.75	5.51
10000	30828.53	12.11	80924.88	10.25	33312.29	13.95	82040.82	10.49	64076.63	9.27	117000.68	8.71	63195.48	10.20	119085.60	9.53



(a) Basic policies

(b) VPN policies

Figure 9.9.: Compilation time for different number of policies



(a) VNF migration (b) Router shutdown

Figure 9.10.: Response time for events affect batches of policies

9.5.3. Reactivity to Network Dynamics

One of the features in our framework is the reactivity to network dynamics where affected policies are updated to reflect the changes in the network topology as we elaborated in Section 9.3.6. We measure the response time that Busoni needs to update affected policies. We define the *response time* as the wall clock time needed by Busoni to detect which policies are affected by the triggered event.

$$Response\,Time = T_f - T_e \tag{9.5.1}$$

As the equation shows, the response time is the wall clock time elapsed from the moment an event got triggered by the network controller $\mathtt{T_e}$ until the framework update all affected policies $\mathtt{T_f}$.

We assume in our evaluation that two dynamic events are taking place separately and measure their response time. The first event is a network function migration which is similar to use case 3 in Section 9.4. We designed the migration to affect different batches of policies (1, 10, 100,1000). Referring to the same policies defined in the Table 9.1, only policies (B, D, F, H) are tested as these are the policies with SFC conditions. The second event is a router shutdown that may have occurred due to a failure or scheduled maintenance. We assume in this event that only Busoni is responsible to react to such an event, although in typical

Table 9.3.: Response time (ms) (95% percentile) for different number of affected policies in the two dynamic events

Batch size	**Event**	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
1	Network migration	-	26.44	-	43.72	-	41.49	-	48.99
10		-	226.9	-	245.7	-	206.8	-	231
100		-	1818	-	2164	-	1784	-	2197
1	Router shutdown	19.42	45.25	22.6	46.34	31.3	47.46	32.99	49.09
10		80.66	212.2	84.6	228.8	139.2	264.8	139.6	265.6
100		610.2	1852	620.7	1847	999.5	2023	1024	2162

situations there are multiple levels of responses in SR as discussed earlier in Section 9.3.6. We also controlled the simulation so that the event only affects the designated batch sizes of policies in each run.

Figure 9.10 shows the recorded response time to each event and policy batch. It is observed that response time is growing linearly with respect to the total number of affected policies. Comparing the response time in the two events, we can notice that they are much close to each other. Busoni is able to respond in around 2 seconds (95th percentile of 1000 trials) for complex policies (i.e., policy H) as reported in Table 9.3. Taking this conclusion and considering the reported average compiling time in Table 9.2, we notice that almost half of the response time is the compilation time for the updated policies.

Summarizing our evaluation, Busoni scales linearly with the incremental size of submitted policies as long as there are enough system resources. In case of dynamic events that needed a reaction from the framework, it finishes its response with a reasonable delay time.

Chapter 10

Future Prospects

In this chapter, we talk about the prospects of Busoni. We investigate the possibility to operate Busoni under different environmental settings including different network technology (*i.e.,* MPLS). After that, we analyze the current limitations of Busoni and suggest some ideas for future extension.

10.1. Applicability of Busoni in MPLS-SR Environment

Related to what we discussed earlier in Chapter§2, SR could either operate on top of MPLS or IPv6 driven networks. However, the current implementation of Busoni assumes IPv6 networks; therefore, the question here is *Busoni operable on MPLS networks?*.

To answer this question, we have to differentiate between MPLS and IPv6 SR networks. The main difference is in the detour or SID representation. In MPLS, SID is represented as MPLS tags; therefore, it is easier to integrate with current MPLS networks. The advantage here is there is no need for reservation protocols like the *Resource Reservation Protocol* (RSVP) [16] or a distribution of tags.

Busoni operates as a northbound interface and by definition should not be affected by any changes in the data plane layer. However, MPLS variation of SR lacks programming support; therefore, network applications should take into consideration and try to overcome such problems and relay on legacy alternatives to behaviors provided in SRv6 programming framework. Other than this point, Busoni is fine to operate on other SR networks as long as the network controller keeps the needed information in the data store.

10.2. Applicability of Busoni in SRv6 on non-Linux Routers Environment

Other SRv6 environments could be encountered when non-Linux routers are used instead of Linux as assumed by Busoni. Although SRv6 operations and behaviors should not be affected, the southbound interface is mainly affected. The communication between the network controller should be updated to utilize different protocols which are supported by data plane hardware. The remaining components which are related to policy libraries and management are not affected.

10.3. Current Limitations and Prospects of Extensions

Busoni provides a framework to manage policies on top of SRv6 networks. Nevertheless, there are still challenges that need to be addressed. In this section, we discuss some of these challenges and suggest some extensions to enhance Busoni.

10.3.1. Flow Specifications

Busoni provides class `Match` to determine which packets should be processed by the submitted policy. In the current version of Busoni, there is no possible way to determine any layer 4 protocol information such as TCP or UDP port number. End users usually specify the network flow using this information (*i.e.,* layer 4 protocol) in addition to layer 3 information (*i.e.,* source/destination IPv6 addresses). Extending `Match` class to support these features should be followed by extending *gRPC* functions to generate appropriate `iptable` rules that reflect layer 4 information. This extension could also support another layer 3 protocols such as ICMPv6. To summarize, extending the source code which responsible for network flow specifications will enhance Busoni capabilities and provide end users with a better tool to customize their target flow.

10.3.2. Rules Conflicts

Another challenge that may affect Busoni accuracy after some running time is the possible conflict between newly added policy and old routing commands. Each time Busoni accepts a new policy, it adds policy-related routing entries to the routing table of the ingress routers. However, with time there will be many entries that related to different policies and conflict

between one of them and any newly added routing entry. A possible conflict would result in rejecting the addition procedure and policy installing process will fail.

To overcome such shortcoming, Busoni should check if there is any possible conflict with the router's entries. If the check was positive, Busoni would need to confirm either the installed entries relate to an active policy or should be deleted. Consequently, the time needed to install route entries in the edge routers would take longer time.

10.3.3. Complex Network Dynamics

In addition to what we have discussed above, Busoni should add support to handle complex network dynamics. A complex network dynamic event could be triggered when the running *Interior Gateway Routing Protocol* (IGP) changes the shortest path between nodes due to dynamic cost changes. In SR the default shortest path calculated between nodes is used to minimize the needed number of segments that should be added to the segments list. Although the IGP can guarantee this path at calculation time, any changes that affect the IGP shortest path algorithm would result in a different path that violates the policy. Busoni could use any traces techniques to validate that every segments list complies with its policy; therefore, any detected violations could be handled.

To conclude, Busoni implements the essential functions that are needed in any north-bound interface; however, to ensure its correctness in different environments, some updates and enhancements should be implemented.

Chapter 11

Conclusion

This thesis presents efficient data management and policy composition for Software-defined Networking frameworks. The key components of this work have addressed different problems related to the performance and the data representation of the northbound interfaces in SDN and SR.

11.1. Dissertation Summary

Firstly, we have presented *Gavel* as a complete ecosystem, a graph database based SDN controller. Gavel is the first controller to exploit graph databases to produce a plain data representation of a software-defined network, and thereby removes the need for a translation between multiple, different and task-specific network policies. Compared to the RDBMS-oriented state-of-the-art of plain data representations, Gavel significantly reduces programming complexity and able to scale better in large networks. The key factor for these achievements is facilitating a much more natural native graph support instead of relying on an RDBMS table structure. We further implemented a variety of proof-of-concept network applications on top of Gavel. In our evaluation, we show that Gavel further significantly outperforms the state-of-the-art of plain data representations and—in contrast to those—scales well with increasing network sizes.

Secondly, we have presented a policy composer and management framework for SRv6 networks. We have demonstrated the capabilities of the framework and the tools provided to compose and manage different SRv6 policies. Service providers will be able to use this tool to write different policies easily and ensure that these policies will be automatically and dynamically adapted to any network updates. After that, we have presented different use cases to demonstrate how this framework could be used. We have also presented an

evaluation with experiments to show the framework scalability under incremental requests and gave an upper bound to the needed time to respond to some dynamic network events.

All the corresponding source code and platform implementations are open-sourced and made available online.
Part I (Gavel):
`https://github.com/engbarakat/Gavel`

Part II (Busoni):
`https://bitbucket.org/engbarakat/srv6_composing_policies`

11.2. Thesis Impact

The author of this dissertation was the lead investigator and first author of several research papers. In particular, the work on designing, implementing, and evaluating *Gavel* that appeared in Part I has appeared in the following peer-reviewed international publication proceedings:

- **Osamah L. Barakat**, David Koll, and Xiaoming Fu. "Gavel: A Fast and Easy-to-Use Plain Data Representation for Software-defined Networks." In: *IEEE Transaction on Network and Service Management 2019, Journal Article*.
- **Osamah L. Barakat**, David Koll, and Xiaoming Fu. "Gavel: Software-defined network control with graph databases." In: *Proceedings of the 20th Conference on Innovations in Clouds, Internet and Networks (ICIN) IEEE, 2017*.
- **Osamah L. Barakat**, David Koll, and Xiaoming Fu. "Gavel: Towards a graph database Defined Network.". In: *Proceedings of the 24th International Conference on Network Protocols (ICNP) 2016, Poster Session*.

The work presented in Part I (*i.e.,* Addressing northbound interface challenges in IPv6 Segment Routing) has been presented in the following articles:

- **Osamah L. Barakat**, Pier Luigi Ventre, Stefano Salsano, and Xiaoming Fu. "Busoni: Policy Composition and Northbound Interface for IPv6 Segment Routing Networks" Submitted to: *Proceedings of the 27th International Conference on Network Protocols (ICNP 2019)* **under submission**.
- **Osamah L. Barakat**, Pier Luigi Ventre, Stefano Salsano, and Xiaoming Fu. "Busoni: Towards a Northbound Interface for Segment Routing Networks" In: *Proceedings of The 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2018), Poster Session*.

Building on the work listed above, the author has further supervised and identified topics for a Master project which is published in peer-reviewed international conference proceedings:

- **Osamah L. Barakat**, Tayyebe Emadinia, David Koll, and Xiaoming Fu. "Paving the Way towards Enterprise SDN Adoption: New Selection Strategies for Hybrid Networks." In: *Proceedings of the 22nd Conference on Innovations in Clouds, Internet and Networks (ICIN) IEEE, 2019*.

Part III.

Appendix

The appendix complements the thesis with more detailed descriptions of algorithms and examples that we felt were too detailed, complicated, or formal for the main text. Nevertheless, the contents are important results of the thesis and form a considerable part of the overall contribution.

Chapter A

Concepts and Definition of Related Terms

Cypher refers to a declarative graph query language works with Neo4j; a graph database management system. It allows for expressive and efficient querying and updating of a property graph.

gRPC is an open source remote procedure call (RPC) system initially developed at Google. It allow remote execution of functions without any binding to a programming language.

Middlebox refers to a network component placed in the path between source and destination and perform functions other than routing. It could be either a software-based or hardware-based.

Network Function refers to any component within a network infrastructure that processes packets. NF can be either a physical compute node or a virtual node *i.e., Virtual Network Function* (VNF).

OpenFlow is a southbound interface protocol which used in SDN to communicate with forwarding devices in the data plane.

Segment ID is an identification mark which used to label routing devices and links connecting these devices in Segment Routing networks. The ID could be represented either as IPv6 address or as MPLS tag depending on the Segment Routing variation.

Service Function Chaining refers to a chain of network functions that should process network traffic first before delivering to the destination.

Chapter B

Gavel Internals

B.1. Representation of Network Topologies in Graph Database

```
"Node": [{
      "id": "284263",
      "labels":["Switch"],
      "properties": {
      "dpid":
      "0000000000001c01",
      "id": "0_28_1",
      "layer": 0}}]

"Node": [{
      "id": "284263",
      "labels":["Switch"],
      "properties": {
      "dpid":
      "0000000000001901",
      "id": "0_25_1",
      "layer": 0}}]
```

```
"Edge": [[{
     "dpid":
     "0000000000001c01",
     "id": "0_28_1",
     "layer": 0 },
     {
     "node2":
     "0000000000001901",
     "node1":
     "0000000000001c01",
     "port1": 5,
     "port2": 9},
     {
     "dpid":
     "0000000000001901",
     "id": "0_25_1",
     "layer": 0}]]
```

Figure B.1.: Exemplary specification of two switches (left) and an edge between these switches (right) in Gavel's graph database. Green coloring indicates the respective endpoints of the edge

B.2. Comparison of Routing Application Implementation between Gavel and Ravel

```
INSERT route request INTO reachability table
INSERT route request INTO rm_delta table
Calculate shortest path
INSERT returned path INTO configured flow table
FOR each switch on path:
    SELECT ports FROM topology table
ENDFOR
```

Figure B.2.: Peusodo code for processing a routing request in Rave [2]

```
MATCH (h1:Host{ip:srcip}), (h2:Host{ip:dstip})
MATCH p=shortestPath((h1)-[:Connected_to*]->(h2))
WITH h1,h2, p
CREATE (h1)-[pa:Path_to
   {switches:[n in nodes(p)[1..-1]| n.dpid],
   ports:[r in rels(p)[1..]| r.port1]}]->(h2)
RETURN pa.switches, pa.ports;
```

Figure B.3.: Code to implement a routing application in Gavel

B.3. ASR Algorithm Implementation

```
1 def getsubroute(session, src, srctype, dst, dsttype):
2   if src == dst :
3     return Subpath(src ,[0],[0], dst)
4   else:
5     if (srctype == 0):
6       result = session.run(''' MATCH (h1:Host {ip:{firstip}}), (h2:Switch {dpid:{secondip}})
       Match p= allshortestPaths ((h1)−[:Connected_to∗]−>(h2)) return
7       reduce(cost=0, r in relationships (p) | cost+r.cost) AS cost ,[n in nodes(p) [1..]| n.dpid
       ] as switches,
8       [r in relationships (p) [1..]| r.port1] as ports , h2.dpid as node order by cost ASC limit
       1;''' ,{" firstip ": src, "secondip": dst} )
```

```
9           for  pathins  in  result :
10            return  Subpath(pathins["switches"], pathins["ports"], pathins["cost"], pathins["node"])
11        elif  (dsttype  == 0):
12          result  = session.run('''MATCH (h1:Switch {dpid:{firstip}}), (h2:Host {ip:{secondip}})
          Match p= allshortestPaths ((h1)-[:Connected_to*]->(h2)) return
13          reduce(cost=0, r  in  relationships (p) |   cost+r.cost) AS cost ,[n in nodes(p)[1..-1]| n.
          dpid] as switches,
14          [r  in  relationships (p) [0..]|  r.port1] as ports, h2.ip as node  order by cost ASC limit 1;
          ''',{"firstip": src, "secondip": dst} )
15          for  pathins  in  result :
16            return  Subpath(pathins["switches"], pathins["ports"], pathins["cost"], pathins["node"])
17        else:
18          result  = session.run('''MATCH (h1:Switch {dpid:{firstip}}), (h2:Switch {dpid:{secondip}})
          Match p= allshortestPaths ((h1)-[:Connected_to*]->(h2)) return  reduce(cost=0, r  in
          relationships (p) |   cost+r.cost) AS cost ,
19          [n in nodes(p) [1..]|  n.dpid] as switches, [r in relationships (p) [0..]|  r.port1] as ports,
          h2.dpid as node order by cost ASC limit 1; ''',{"firstip": str(src), "secondip": str(dst)}
          )
20          for  pathins  in  result :
21            return  Subpath(pathins["switches"], pathins["ports"], pathins["cost"], pathins["node"])
22
23  def  getallhostingSwitches (session, listofFun):
24    for fn  in  listofFun :
25      result  =  session.run('''Match (s1:Switch)-[hosting_MB]-(mb:MiddleBox{dpid:{fnID}})
      return s1 as hosting ''',{"fnID",fn})
26      for  switches  in  result :
27        fn. setlistofhostednodes (switches["hosting"])
28
29
30  def  asrroutecalculation (session,  srcIP, dstIP, listofFun):
31    fullpath = Subpath ([],[],0, "")
32    lastvisitednode  = None
33    for  index  in  range  (len( listofFun )):
34      listofsubpath  = []
35
36      if (index ==0):#if  it  is  the  first  fn
37        for  mb in  listofFun [index]. listofhostednodes :
38          a = getsubroute(session,  srcIP,  0,  mb,1)
39          listofsubpath .append(a)
40        templist  = min( listofsubpath ,key= attrgetter ("cost"))
41        fullpath. switcheslist .extend( templist. switcheslist )
42        fullpath. portslist .extend( templist. portslist )
43        fullpath.cost += templist.cost
44        fullpath.lastnode  = templist.lastnode
45        lastvisitednode  = fullpath.lastnode # to get last visited node
46      elif (listofFun [index]==listofFun [-1]):#if it is the last fn
47        for  mb in  listofFun [index]. listofhostednodes :
48          a = getsubroute(session,  lastvisitednode , 1, mb,1)
49          listofsubpath .append(a) #fix how to get the last node before asking for subpath
50        templist  = min( listofsubpath ,key= attrgetter ('cost'))
```

```
51          fullpath . switcheslist .extend( templist . switcheslist )
52          fullpath . portslist .extend( templist . portslist )
53          fullpath . cost += templist . cost
54          fullpath . lastnode = templist . lastnode
55          lastvisitednode = fullpath . lastnode
56       else :
57         for mb in listofFun [index ]. listofhostednodes :
58           a = getsubroute ( session , lastvisitednode , 1, mb,1)
59            listofsubpath .append(a) #fix how to get the last node before asking for subpath
60          templist = min( listofsubpath ,key= attrgetter ('cost'))
61          fullpath . switcheslist .extend( templist . switcheslist )
62          fullpath . portslist .extend( templist . portslist )
63          fullpath . cost += templist . cost
64          fullpath . lastnode = templist . lastnode
65          lastvisitednode = fullpath . lastnode #to get last visited node
66    tempsubpath = getsubroute ( session , lastvisitednode , 1, dstIP ,0)
67    fullpath . switcheslist .extend(tempsubpath. switcheslist )
68    fullpath . portslist .extend(tempsubpath. portslist )
69    fullpath . cost += tempsubpath.cost
70    installpathasr ( session , fullpath , srcIP , dstIP )
71    updatelinkcost ( session , srcIP , dstIP , fullpath )
72
73 def installpathasr ( session , path , srcIP , dstIP ):
74    result = session .run( ''' MATCH (h1:Host {ip:{ firstip }}), (h2:Host {ip:{secondip}})
75    create (h1)−[pa:SFC_Path{switches:{ listofswitches }, ports :{ listofports }, SFCID:{sfcid}, cost
        :{ cost }}]−>(h2)
76    return pa. cost ; ''' ,{" firstip ": srcIP , "secondip": dstIP , " listofswitches ":path. switcheslist ,
        " listofports ":path. portslist , "sfcid": srcIP , "cost": path.cost} )
77
78 def updatelinkcost ( session , srcIP , dstIP , path ):
79    result = session .run( ''' MATCH (h1:Host {ip:{ firstip }})−[r]−> (h2:Switch {dpid:{secondip}})
        set r.cost = r.cost+1; ''' ,{" firstip ": srcIP , "secondip": path. switcheslist [0]})
80    for index in xrange (1, len(path. switcheslist ),1) :
81      if (index==len(path. switcheslist )−1):
82        result = session .run( ''' MATCH (h1:Switch {dpid:{ firstip }})−[r]−> (h2:Host {ip:{secondip
        }}) set r.cost = r.cost+1; ''' ,{" firstip ": path. switcheslist [−1], "secondip": dstIP })
83      else :
84        result = session .run( ''' MATCH (h1:Switch {dpid:{ firstip }})−[r]−> (h2:Switch {dpid:{
        secondip}}) set r.cost = r.cost+1; ''' ,{" firstip ": path. switcheslist [index ], "secondip":
        path. switcheslist [index+1]})
```

Listing B.1: Code snippets of ASR algorithm implementation

Chapter C

Busoni Algorithms and Work flow

C.1. Busoni's work flow

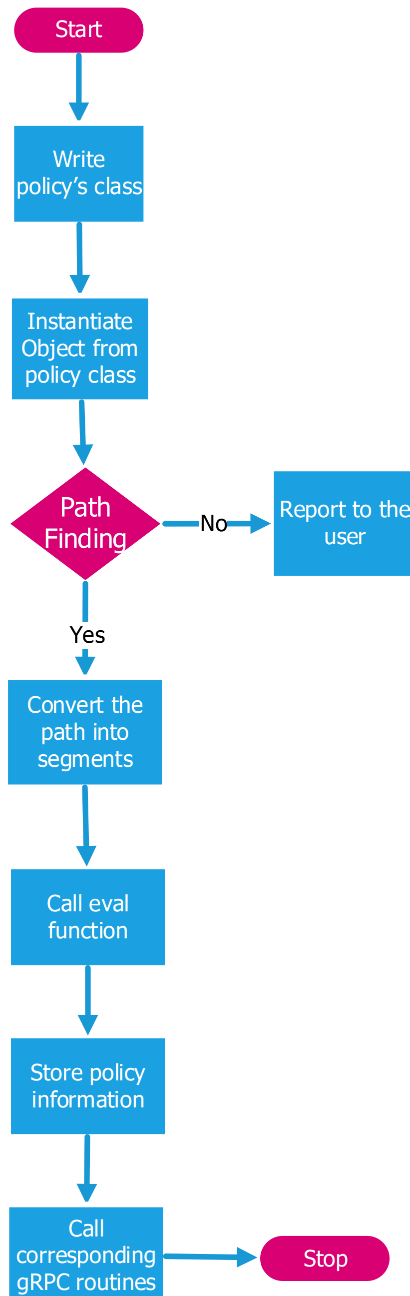


Figure C.1.: A flowchart of Busoni's work flow

C.2. SRtypes Code Snippets

```
1  class PolicySR:
2      __metaclass__ = ABCMeta
3      counter = 0
4      def __init__(self, match, qos=None, networkfunctionslist=None, nflistordered=True,
       matchonsrc = False, id=0):
5          if not any(elem in match.map_dict.keys() for elem in ["srcIP","dstIP"]):
6              raise KeyError
7          #self.id = str(timeit.default_timer())
8          PolicySR.counter = PolicySR.counter +1
9          self.id = id if id else PolicySR.counter
10
11         self.networkfunctionslist = networkfunctionslist
12         self.nflistordered = nflistordered
13         self.qos = qos
14         self.controller = "Gavel"
15         self.segments = []
16         self.path=[]
17         self.matchpkt = match
18         self.matchonsrc = matchonsrc
19         self.behaviorlist = []
20         self.adj_sid_to_VNFs = []
21         if networkfunctionslist:
22             for adj in self.networkfunctionslist:
23                 self.adj_sid_to_VNFs.extend(adj.adjconnections)
24         self.qosscore = qos.getcost() if qos else -1
25         self.findpath()
26         #self.get_path_as_segments()
27         for a in self.path:
28             logger.debug(a)
29         self.get_path_as_segments()
30         self.eval()
31         if self.send_segements_to_edgeRouters():
32             self.insertIntentSRDB()
33         else:
34             logger.error("Error happened during send segments to routers")
35
36     @abstractmethod
37     def eval(self):
38         pass
39
40     def edge_in_dag(self, dag_root, u, v):
41         from Gavel.DatabaseEngine import ADJ, DIST, EDGES
42         return DIST[dag_root][u] + ADJ[u][v] == DIST[dag_root][v]
43
44     def dag_indegree(self, dag_root, v):
45         from Gavel.DatabaseEngine import ADJ, DIST, EDGES
46         indeg = 0
```

```
47         #print EDGES
48         #print EDGES[v]
49         for u, _ in EDGES[v]:
50             if self.edge_in_dag(dag_root, u, v): indeg += 1
51         return indeg
52     def prepare_adj(self):
53         self.pro_segments = []
54 
55         for singlepath in self.path:
56             adjdict = OrderedDict()
57             for adj in singlepath.portslist:
58                 adjdict[adj] = int(adj.split(":")[1], 16) -1
59             self.pro_segments.append(adjdict)
60             #print adjdict
61 
62     def get_path_as_segments(self):
63         self.prepare_adj()
64         for i in range(len(self.pro_segments)): # iterate over paths stored for this policy
65             dag_root = self.pro_segments[i][self.path[i].portslist[0]]
66             seg = []
67             seg_weight = 0
68             for j in range(len(self.path[i].portslist) - 1): # iterate for every node
69                 u = self.pro_segments[i][self.path[i].portslist[j]]
70                 v = self.pro_segments[i][self.path[i].portslist[j+1]]
71                 #if v is mb
72                 if self.path[i].portslist[j+1] in self.adj_sid_to_VNFs:
73                     seg.append(self.path[i].portslist[j+1])
74                     seg_weight += 1
75                     dag_root = v
76                 elif self.path[i].portslist[j] in self.adj_sid_to_VNFs:
77                     pass
78                 elif not self.edge_in_dag(dag_root, u, v) or self.dag_indegree(dag_root, v) >
   1:
79                     if self.dag_indegree(u, v) == 1:
80                         # there is no ECMP from u to v, we add node segment u
81                         seg.append(self.path[i].portslist[j])
82                         seg_weight += 1
83                         dag_root = u
84                     else:
85                         # there is ECMP from u to v, we need adjacency segment (u, v)
86                         seg.append(self.path[i].portslist[j+1])
87                         seg_weight += 2
88                         dag_root = v
89             # add source and destination, if necessary
90             if len(seg) == 0 or self.path[i].portslist[0] != seg[0]:
91                 seg = [self.path[i].portslist[0]] + seg
92                 seg_weight += 1
93             if len(seg) == 0 or self.path[i].portslist[-1] != seg[-1]:
94                 seg.append(self.path[i].portslist[-1])
95                 seg_weight += 1
```

```
96                 self.segments.append(SegmentsList(seg, self.path[i].srcIP, self.path[i].dstIP))
97                 #print seg
98                 logger.debug(seg)
99                 #return seg_weight, seg
100    def send_segements_to_edgeRouters(self):
101        for p in self.segments:
102            return send_segmentslist_edgerouter(p, self.matchpkt, self.matchonsrc)
103    def updateIntentSRDB(self):
104        if not self.segments:
105            logger.error("An error is accured. The segments are empty means either no path
       found or problem in system")
106        else:
107            if self.controller == "Gavel":
108                update_policy_DB(self.__class__.__name__, self.id, self.matchpkt, self.segments
       [0].segs, self.path[0].ID,
109                                 self.qosscore, [f.nfid for f in self.networkfunctionslist] if
       self.networkfunctionslist else [], self.nflistordered, self.matchonsrc, self.vpnuser)
110    def insertIntentSRDB(self):
111        if not self.segments:
112            logger.error("An error is accured. The segments are empty means either no path
       found or problem in system")
113        else:
114            if self.controller == "Gavel":
115                insert_policy_DB(self.__class__.__name__, self.id, self.matchpkt, self.segments
       [0].segs, self.path[0].ID,
116                                 self.qosscore, [f.nfid for f in self.networkfunctionslist] if
       self.networkfunctionslist else [], self.nflistordered, self.matchonsrc, self.vpnuser if
       hasattr(self, 'vpnuser') else 0)
117    def findpath(self):
118        srcIPset = self.matchpkt.map_dict["srcIP"]
119        dstIPset = self.matchpkt.map_dict["dstIP"]
120        # lists_1 = [srcIPset, dstIPset]
121        list_of_src_dst = [(srcIPset, dstIPset)]
122        #for element in itertools.product(*lists_1):
123        #    list_of_src_dst.append(element)
124        #print type(srcIPset)
125        logger.debug("a list of src, dst pairs created from %s and %s is : %s" % (srcIPset,
       dstIPset, list_of_src_dst))
126        for pair in list_of_src_dst:
127            logger.debug("finding the path between src {} and dst {}".format(pair[0], pair[1]))
128            self.path.append(SR.findpathinSRv6(pair[0], pair[1], self.qosscore, self.
       networkfunctionslist))
129            logger.debug(self.path)
130    def __add__(self, other):
131        pass
132    def __mul__(self, other):
133        pass
134    def insert_behavior_first_segment(self, behavior, argument=0):
135        for p in self.segments:
136            segbehavoir = isbehavioravailable(p.segs[0], behavior)
```

```
137                 if  segbehavoir :
138                     if  argument:
139                         segbehavoir  =  self . attchargument_to_behavior ( segbehavoir ,  argument)
140                         p.segs . insert (1,  segbehavoir )
141                         p.  behavior_first   = True
142                     else :
143                         p.segs . insert (1, segbehavoir )
144                         p.  behavior_first   = True
145                 else :
146                     logger . error ( 'Behavior  {}  is  not  offered  by  the  router  {}'.format( behavior ,p
        [0]) )
147 
148         def  attchargument_to_behavior ( self ,b,a) :
149             temp = ""
150             for  s  in  b. split (":") [:3]:
151                 temp = temp + s+":"
152             return  temp+"%X"%a+"::"
153 
154         def  insert_behavior_end_segment ( self , behavior , argument=0):
155             for  p  in  self .segments:
156                 segbehavoir  =  isbehavioravailable  (p.segs[−1],behavior )
157                 if  segbehavoir :
158                     if  argument:
159                         segbehavoir  =  self . attchargument_to_behavior ( segbehavoir , argument)
160                         p.segs[−1]= segbehavoir
161                         p.behavior_end  = True
162                     else :
163                         p.segs[−1] = segbehavoir
164                         p.behavior_end  = True
165                 else :
166                     logger . error ( 'Behavior  {}  is  not  offered  by  the  router  {}'.format( behavior ,p.
        segs[−1]))
167         def  __str__ ( self ) :
168             return  'This  is  a  Segment Routing policy  the  source  address  is :  {0} and  the  destination
        address  is :  {1}. \nThe Qos needed is  {2} and  the  NFs that  must be  in  the  path  are :  {3} '.
        format(
169                 self .matchpkt.map_dict["srcIP"],  self .matchpkt.map_dict["dstIP"],  self .qosscore ,  [
        str (n)  for  n  in  self .  networkfunctionslist  ])
170         @staticmethod
171         def  delete_policy  (policyID) :
172             delete_policy_DB (policyID)
173 
174 class  Match():
175         '''
176         This  class  should  help  users  define  their  match
177         '''
178         def  hascomplexrules( self ) :
179             return  False
180         def  __init__  ( self ,∗args ,  ∗∗kwargs):
181             self .map_dict  =  dict(∗args ,  ∗∗kwargs)
```

Listing C.1: Code snippets of SRtypes

C.3. Path finding in Busoni Code Snippets

```
def installpathasr (session, path, srcIP, dstIP):
    try:
        result = session.run(''' Merge (p: Policy_path {from:{srcip}, to:{dstip}, links:{adj} })
 with p
                            Match (r:Router) where r.sequence in {routers}
                            Merge (r)-[:rMemberofP]->(p) return distinct id(p) ''',{"srcip
 ":srcIP,"dstip":dstIP,"routers":path.switcheslist,"adj":path.portslist}).single().value()

        path.setID(result)
        if path.ID >0:
            return path.ID
    except:
        logger.error("Writing path information to the DB failed!")
        return -1
def updatelinkcost(session, srcIP, dstIP, path):
    result = session.run(
        ''' MATCH (h1:Host {ip:{firstip}})-[r]-> (h2:Switch {dpid:{secondip}}) set r.cost = r.
 cost+1;''',
        {" firstip ": srcIP, "secondip": path.switcheslist[0]})
    for index in xrange(1, len(path.switcheslist), 1):
        if (index == len(path.switcheslist) - 1):
            result = session.run(
                ''' MATCH (h1:Switch {dpid:{firstip}})-[r]-> (h2:Host {ip:{secondip}}) set r.
 cost = r.cost+1;''',
                {" firstip ": path.switcheslist[-1], "secondip": dstIP})
        else:
            result = session.run(
                ''' MATCH (h1:Switch {dpid:{firstip}})-[r]-> (h2:Switch {dpid:{secondip}}) set r
 .cost = r.cost+1;''',
                {" firstip ": path.switcheslist[index], "secondip": path.switcheslist[index +
 1]})

def unconditionedroute(session, srcIP, dstIP):
    result = session.run(''' MATCH (h1:Host {ip:{firstip}}), (h2:Host {ip:{secondip}})

                    Match p= allshortestPaths ((h1)-[rc:Connected_to*]->(h2))
                            return reduce(cost=0, r in relationships (p)| cost+r.cost) AS
 cost ,[n in nodes(p)[1..-1]| n.sequence] as switches,[r in relationships (p) [1..]| r.port1]
 as ports order by cost ASC limit 1;''',
                        {" firstip ": srcIP, "secondip": dstIP})
    p = None
```

```
34     for i in result:
35         p = Subpath(i['switches'], i['ports'], None,0,None)
36         p.srcIP = srcIP
37         p.dstIP = dstIP
38     if p.portslist:
39         p.setID( installpathasr (session,p,srcIP,dstIP))
40         if p.ID>0:
41             return p
42     logger.error("Error finding path")
43 
44 def qosroute(session, srcIP, dstIP,score):
45     result = session.run(''' MATCH (h1:Host {ip:{firstip}}), (h2:Host {ip:{secondip}})
46             Match p= allshortestPaths ((h1)-[rc:Connected_to*]->(h2))
47                     where all(wwr in relationships (p) where wwr.cost>{micscore})
48                     return reduce(cost=0, r in relationships (p)| cost+r.cost) AS cost
       ,[n in nodes(p)[1..-1]| n.sequence] as switches,[r in relationships (p) [1..]| r.port1] as
       ports order by cost ASC limit 1;''',
49                 {"firstip": srcIP, "secondip": dstIP, 'micscore':score})
50     p = None
51     for i in result:
52         p = Subpath(i['switches'], i['ports'], None,i['cost'], None)
53         p.srcIP = srcIP
54         p.dstIP = dstIP
55     if p.portslist:
56         p.setID( installpathasr (session,p,srcIP,dstIP))
57         if p.ID>0:
58             return p
59     logger.error("Error finding path")
60 
61 def findpathinSRv6(srcIP, dstIP, score, SFClist):
62     if score>-1 and SFClist:
63         logger.debug("calling routing for intent with SFC and QoS from source {} to destination
       {}".format(srcIP,dstIP))
64         return asrrouteOptimized(getsession(), srcIP, dstIP, SFClist, score)
65     elif score>-1:
66         logger.debug("calling routing for intent with QoS only from source {} to destination {}
       ".format(srcIP,dstIP))
67         return qosroute(getsession(),srcIP,dstIP,score)
68     elif SFClist:
69         logger.debug("calling routing for policy with SFC only from source {} to destination {}
       with SFClist {}".format(srcIP,dstIP,[f.nfid for f in SFClist]))
70         return asrrouteOptimized(getsession(), srcIP, dstIP, SFClist, score)
71     else:
72         #normal path
73         logger.debug("calling routing for basic policy (no QOS, SFC) from source {} to
       destination {}".format(srcIP,dstIP))
74         return unconditionedroute(getsession(),srcIP,dstIP)
75 
76 def asrrouteOptimized(session, srcIP, dstIP, listofFun,cost):
77     fullpath = Subpath([], [],[], 0, "")
```

```
78      fullpath . srcIP=srcIP
79      fullpath . dstIP=dstIP
80      lastvisitednode  = None
81      for index in range(len( listofFun )):
82          if (index == 0):  # if it is the first fn
83              logger.debug(" calling getsubroute with src {} and dst {} and Fn {} and cost {}".
        format( srcIP, listofFun [index ]. nfid , listofFun [index ]. nfid , cost ))
84              a = getsubroute_host_mb ( session ,  srcIP ,  listofFun [index ]. nfid ,  cost )
85              fullpath . switcheslist .extend(a. switcheslist )
86              fullpath . portslist .extend(a. portslist )
87              # fullpath . portslist . insert (−1,"MB")
88              fullpath . cost += a. cost
89              fullpath . lastnode = a. lastnode
90              lastvisitednode = fullpath . lastnode  # to get last visited node
91              logger.debug("The Final path construction is starting now {}".format( fullpath ))
92          # print "the last visited node after fn 100 is " + str ( lastvisitednode )
93          else :
94              if  lastvisitednode  in  listofFun [index ]. listofhostednodes :
95                  fullpath . switcheslist .extend( lastvisitednode )
96                  logger.debug("Trying to find path between router space {} and middelBox sid {}
        hosted in {}".format( lastvisitednode , listofFun [index ]. nfid , listofFun [index ].
         listofhostednodes ))
97                  result = getsession () .run( '''  Match (r:Router{space:{ routerspace }})−[re]−>(mb
        :MiddelBox {sid:{mbsid}}) return re.port1  ''' ,{" routerspace ": lastvisitednode ,"mbsid":
         listofFun [index ]. nfid })
98                  for pathins in result :
99                      fullpath . portslist .append( str ( pathins ["re . port1 "]) )
100                 # fullpath . portslist . insert (−1,"MBB")
101                 fullpath . cost+=listofFun [index ]. cost
102             else :
103                 logger.debug(" calling getsubroute with src {} and dst {} and Fn {} and cost {}"
        .format( lastvisitednode , [ str (jk) for jk in listofFun [index ]. listofhostednodes ], listofFun
        [index ]. nfid , cost ))
104                 a = getsubroute_router_mb ( session ,  lastvisitednode ,  listofFun [index ]. nfid , cost )
105                 if a:
106                     fullpath . switcheslist .extend(a. switcheslist )
107                     fullpath . portslist .extend(a. portslist )
108                     # fullpath . portslist . insert (−1,"MB")
109                     fullpath . cost += a. cost
110                     fullpath . lastnode = a. lastnode
111                     # print a. lastnode
112                     lastvisitednode = fullpath . lastnode  # to get last visited node
113                 logger.debug("The Final path construction status is {}".format( fullpath ))
114     # now get the path from last MB to dstIP
115     logger.debug( " calling getsubroute enhanced with src {} and dst {} from last router hosting
         last Fn to dst and cost{} ".format(  lastvisitednode , dstIP , cost ))
116     tempsubpath = getsubroute_router_host ( session ,  lastvisitednode ,  str (dstIP ),  cost )
117     fullpath . switcheslist .extend(tempsubpath. switcheslist )
118     fullpath . portslist .extend(tempsubpath. portslist )
119     fullpath . cost += tempsubpath.cost
```

```
120     logger.debug("The Final path construction is finished and it is: {}".format(fullpath))
121     installpathasr (session, fullpath, srcIP, dstIP)
122     # updatelinkcost (session, srcIP, dstIP, fullpath)
123     return fullpath
124 def getsubroute_host_mb(session, srcIP, mb, cost):
125     logger.debug("getsubroute from host {} to mb {} called and mincost {}".format(srcIP, mb,
        cost))
126     result = session.run(''' MATCH (h1:Host {ip:{firstip}}), (h2:MiddelBox {sid:{mbsid}})
127             Match p= allshortestPaths ((h1)-[rc:Connected_to*]->(h2))
128                     where all(wwr in relationships (p) where wwr.cost>{mincost})
129                     return reduce(cost=0, r in relationships (p) | cost+r.cost) AS cost ,[n in
        nodes(p)[1..-1]| n.sequence] as switches, [n in nodes(p)[1..-1]| n.space] as lastnode,
130                     [n in nodes(p) [1..]| labels(n)] as typeofnodes,[r in relationships (p)
        [1..]| r.port1] as ports order by cost ASC limit 1;''',
131                         {" firstip ": srcIP, "mbsid": mb, "mincost": cost})
132     for pathins in result:
133         return Subpath(pathins["switches"], pathins["ports"], pathins["typeofnodes"], pathins["
        cost"], pathins["lastnode"][-1])
134 def getsubroute_router_mb(session, routerIP, mb, cost):
135     result = session.run(''' MATCH (h1:Router {space:{firstip}}), (h2:MiddelBox {sid:{mbsid}})
136                 Match p= allshortestPaths ((h1)-[rc:Connected_to*]->(h2))
137                         where all(wwr in relationships (p) where wwr.cost>{mincost})
138                         return reduce(cost=0, r in relationships (p) | cost+r.cost) AS cost ,[n
        in nodes(p)[1..-1]| n.sequence] as switches, [n in nodes(p)[1..-1]| n.space] as lastnode,
139                         [n in nodes(p) [1..]| labels(n)] as typeofnodes,[r in relationships (p)
        [0..]| r.port1] as ports order by cost ASC limit 1;''',
140                             {" firstip ": str(routerIP), "mbsid": str(mb), "mincost": cost})
141     for pathins in result:
142         try:
143             # when the last node is the same node that host the mb then this would be null
144             return Subpath(pathins["switches"], pathins["ports"], pathins["typeofnodes"],
        pathins["cost"], pathins["lastnode"][-1])
145         except:
146             return Subpath(pathins["switches"], pathins["ports"], pathins["typeofnodes"],
        pathins["cost"], pathins["lastnode"])
147 def getsubroute_router_host (session, routerIP, dstIP, cost):
148     result = session.run(''' MATCH (h1:Router {space:{firstip}}), (h2:Host {ip:{secondip}})
149                 Match p= allshortestPaths ((h1)-[rc:Connected_to*]->(h2))
150                         where all(wwr in relationships (p) where wwr.cost>{mincost})
151                         and NONE(n IN nodes(p) WHERE n:MiddelBox)
152                         return reduce(cost=0, r in relationships (p) | cost+r.cost) AS cost ,[n
        in nodes(p)[1..-1]| n.sequence] as switches, [n in nodes(p)[1..-1]| n.space] as lastnode,
153                         [n in nodes(p) [1..]| labels(n)] as typeofnodes,[r in relationships (p)
        [0..]| r.port1] as ports order by cost ASC limit 1;''',
154                             {" firstip ": routerIP, "secondip": dstIP, "mincost": cost})
155     for pathins in result:
156         return Subpath(pathins["switches"], pathins["ports"], pathins["typeofnodes"], pathins["
        cost"], None)
```

Listing C.2: Code snippets of Path finding in Busoni

Bibliography

Bibliography

[1] A. Dwaraki and T. Wolf, "Adaptive Service-Chain Routing for Virtual Network Functions in Software-Defined Networks," in *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, ser. HotMIddlebox '16. New York, NY, USA: ACM, 2016, pp. 32–37. (Cited on xiii, 41, 66.)

[2] A. Wang, X. Mei, J. Croft, M. Caesar, and B. Godfrey, "Ravel: A Database-Defined Network," in *Proceedings of the Symposium on SDN Research - SOSR '16*. New York, New York, USA: ACM Press, 2016, pp. 1–7. (Cited on xiv, 5, 6, 22, 26, 30, 31, 33, 40, 42, 44, 96.)

[3] T. Benson, A. Akella, and D. Maltz, "Unraveling the complexity of network management," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 335–348. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1559000 (Cited on 1.)

[4] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486012 (Cited on 1, 2, 11.)

[5] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. Maltz, "Latency inflation with mpls-based traffic engineering," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11. New York, NY, USA: ACM, 2011, pp. 463–472. [Online]. Available: http://doi.acm.org/10.1145/2068816.2068859 (Cited on 1.)

[6] D. Kreutz, F. M. V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015. (Cited on 1, 11, 26, 63.)

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow," *ACM SIGCOMM Computer Communication*

Review, vol. 38, no. 2, p. 69, Mar. 2008. (Cited on 2, 6, 60.)

[8] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The Segment Routing Architecture," in *2015 IEEE Global Communications Conference (GLOBECOM)*, Dec. 2015, pp. 1–6. (Cited on 2, 13, 57, 59, 67.)

[9] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, K. N. B., C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev, S. Padgett, F. Rabe, S. Ray, M. Tewari, M. Tierney, M. Zahn, J. Zolla, J. Ong, and A. Vahdat, "B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018, pp. 74–87. [Online]. Available: http://doi.acm.org/10.1145/3230543.3230545 (Cited on 2.)

[10] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic," *USENIX ;login:*, vol. 38, pp. 40–47, 2013. (Cited on 2, 6, 21, 22, 25, 26, 60.)

[11] A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu, "Supporting Diverse Dynamic Intent-based Policies Using Janus," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: ACM, 2017, pp. 296–309. (Cited on 5, 21, 22, 25, 42.)

[12] C. Prakash, Y. Zhang, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, and P. Sharma, "PGA: Using Graphs to Express and Automatically Reconcile Network Policies," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 29–42, Aug. 2015. (Cited on 6, 21, 22, 25, 26.)

[13] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic : Verifiable Dynamic Network Control," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 59–72. (Cited on 6, 21, 22, 25, 26.)

[14] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin, "A Network-state Management Service," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 563–574. (Cited on 6, 22, 26.)

[15] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-Defined Networks," *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 87–101, 2015. (Cited on 6, 22, 26, 60.)

[16] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois, “A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’15, 2015, pp. 15–28. (Cited on 6, 60, 81.)

[17] V. Heorhiadi, S. Chandrasekaran, M. K. Reiter, and V. Sekar, “Intent-driven Composition of Resource-management SDN Applications,” in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’18, 2018, pp. 86–97. (Cited on 6, 60.)

[18] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: A network programming language,” in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. [Online]. Available: http://doi.acm.org/10.1145/2034773.2034812 (Cited on 6, 21, 25, 60.)

[19] A. Voellmy, S. Chen, X. Wang, and Y. R. Yang, “Magellan: Generating Multi-Table Datapath from Datapath Oblivious Algorithmic SDN Policies,” in *Proceedings of the ACM SIGCOMM Conference*, 2016, pp. 593–594. (Cited on 6, 60.)

[20] C. Filsfils, P. C. Garvia, J. Leddy, D. Voyer, S. Matsushima, and Z. Li, “SRv6 Network Programming,” IETF Secretariat, Internet-Draft draft-filsfils-spring-srv6-network-programming-06, Oct. 2018. (Cited on 7, 15, 57.)

[21] D. Lebrun, M. Jadin, F. Clad, C. Filsfils, and O. Bonaventure, “Software Resolved Networks: Rethinking Enterprise Networks with IPv6 Segment Routing,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’18. ACM, 2018, pp. 6:1–6:14. (Cited on 7, 59.)

[22] A. Abdelsalam, F. Clad, C. Filsfils, S. Salsano, G. Siracusano, and L. Veltri, “Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure,” in *2017 IEEE Conference on Network Softwarization, NetSoft*, 2017, pp. 1–5. (Cited on 7, 57, 59.)

[23] Y. Desmouceaux, P. Pfister, J. Tollet, M. Townsley, and T. Clausen, “6lb: Scalable and Application-Aware Load Balancing with Segment Routing,” *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 819–834, Apr. 2018. (Cited on 7, 57.)

[24] F. Aubry, S. Vissicchio, O. Bonaventure, and Y. Deville, “Robustly Disjoint Paths with Segment Routing,” in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’18, 2018, pp. 204–216. (Cited on 7, 57, 59.)

[25] P. L. Ventre, M. M. Tajiki, S. Salsano, and C. Filsfils, “SDN Architecture and Southbound APIs for IPv6 Segment Routing Enabled Wide Area Networks,” *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 4, pp. 1378–1392, Dec. 2018. (Cited on 7, 57, 59, 64.)

[26] M. Xhonneux, F. Duchene, and O. Bonaventure, “Leveraging eBPF for Programmable Network Functions with IPv6 Segment Routing,” in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’18, 2018, pp. 67–72. (Cited on 7, 57, 59.)

[27] O. L. Barakat, D. Koll, and X. Fu, “Gavel: Software-defined network control with graph databases,” in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, Mar. 2017, pp. 279–286, ©2017 IEEE. (Cited on 8, 60, 63, 74.)

[28] H. Freeman and R. Boutaba, “Networking industry transformation through softwarization [the president’s page],” *IEEE Communications Magazine*, vol. 54, no. 8, pp. 4–6, 2016. (Cited on 11.)

[29] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, “Design and implementation of a consolidated middlebox architecture,” in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 323–336. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar (Cited on 12.)

[30] C. Filsfils, S. Previdi, G. Dawra, W. Henderickx, and D. Cooper, “Interconnecting Millions Of Endpoints With Segment Routing,” Internet Engineering Task Force, Internet-Draft draft-filsfils-spring-large-scale-interconnect-12, Aug. 2018, work in Progress. (Cited on 13.)

[31] “Source packet routing in networking (spring),” Jan. 2019. [Online]. Available: https://datatracker.ietf.org/wg/spring/about/ (Cited on 13.)

[32] A. Bashandy, C. Filsfils, B. Decraene, S. Litkowski, P. Francois, D. Voyer, F. Clad, and P. C. Garvia, “Topology Independent Fast Reroute using Segment Routing,” Dec. 2018, work in Progress. (Cited on 13, 69.)

[33] P. C. Garvia, C. Filsfils, H. Elmalky, S. Matsushima, D. Voyer, A. Cui, and B. Peirens, “SRv6 Mobility Use-Cases,” Internet Engineering Task Force, Internet-Draft draft-camarilloelmalky-springdmm-srv6-mob-usecases-01, Jan. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-camarilloelmalky-springdmm-srv6-mob-usecases-01 (Cited on 13.)

[34] C. Filsfils, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer, "IPv6 Segment Routing Header (SRH)," Internet Engineering Task Force, Internet-Draft draft-ietf-6man-segment-routing-header-16, Feb. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-6man-segment-routing-header-16 (Cited on 13, 14.)

[35] G. Naik, F. Iqbal, B. Peirens, N. Kumar, Z. Ali, C. Pignataro, S. Matsushima, C. Filsfils, J. Leddy, and R. Raszuk, "Operations, Administration, and Maintenance (OAM) in Segment Routing Networks with IPv6 Dataplane (SRv6)." (Cited on 15.)

[36] S. Deering and B. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," Internet Engineering Task Force, Tech. Rep. 8200, Jul. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8200.txt (Cited on 15.)

[37] D. Lebrun and O. Bonaventure, "Implementing IPv6 Segment Routing in the Linux Kernel," in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW '17, 2017, pp. 35–41. (Cited on 16.)

[38] "The Fast Data Project (FD.io)," Jan. 2017. [Online]. Available: http://www.segment-routing.net/open-software/vpp/ (Cited on 16.)

[39] R. Angles and C. Gutierrez, "Survey of Graph Database Models," *ACM Computer Survey*, vol. 40, no. 1, pp. 1–39, Feb. 2008. (Cited on 17, 31, 69.)

[40] C. Schlesinger, M. Greenberg, and D. Walker, "Concurrent NetCore: From Policies to Pipelines," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14. New York, NY, USA: ACM, 2014, pp. 11–24. (Cited on 21, 25.)

[41] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*. New York, New York, USA: ACM Press, 2013, p. 109. (Cited on 21, 25.)

[42] R. Soule, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A Language for Provisioning Network Resources," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: ACM, 2014, pp. 213–226. (Cited on 21, 25.)

[43] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic Foundations for Networks," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,

ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 113–126. (Cited on 25.)

[44] D. Kozen, "Kleene Algebra with Tests," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 3, pp. 427–443, May 1997. (Cited on 25.)

[45] Davide Sanvito, Daniele Moro, Mattia Gulli, Ilario Filippini, Antonio Capone, and Andrea Campanella, "ONOS Intent Monitor and Reroute service: enabling plug&play routing logic," in *NETSOFT 2018, 4th IEEE Conference on Network Softwarisation, 25-29 June 2018, Montreal, Canada*, 2018. (Cited on 25.)

[46] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, and B. Lantz, "ONOS: towards an open, distributed SDN OS," *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*, pp. 1–6, 2014. (Cited on 25.)

[47] F. Chen, C. Wu, X. Hong, and B. Wang, "Easy Path Programming: Elevate Abstraction Level for Network Functions," *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, pp. 189–202, Feb. 2018. (Cited on 25.)

[48] D. Comer and A. Rastegarnia, "OSDF: A framework for software defined network programming," in *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*, Jan. 2018, pp. 1–4. (Cited on 25.)

[49] W. Kellerer, A. Basta, P. Babarczi, A. Blenk, M. He, M. Klugel, and A. M. Alba, "How to measure network flexibility? a proposal for evaluating softwarized networks," *IEEE Communications Magazine*, vol. 56, no. 10, pp. 186–192, October 2018. (Cited on 26.)

[50] W. Wang, W. He, and J. Su, "Redactor: Reconcile network control with declarative control programs In SDN," in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, Nov. 2016, pp. 1–10. (Cited on 26, 42.)

[51] M. Trevisan, I. Drago, M. Mellia, H. H. Song, and M. Baldi, "AWESoME: Big Data for Automatic Web Service Management in SDN," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 13–26, Mar. 2018. (Cited on 26.)

[52] C. Sieber, A. Blenk, A. Basta, D. Hock, and W. Kellerer, "Towards a programmable management plane for sdn and legacy networks," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, June 2016, pp. 319–327. (Cited on 26.)

[53] B. Davie, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. Gude, A. Padmanabhan, T. Petty, K. Duda, and A. Chanda, "A Database Approach to SDN Control Plane Design," *SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 1, pp. 15–26, Jan. 2017. (Cited

on 26.)

[54] R. Raghavendra, J. Lobo, and K.-W. Lee, "Dynamic graph query primitives for SDN-based cloudnetwork management," in *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12*. New York, New York, USA: ACM Press, 2012, p. 97. (Cited on 27.)

[55] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing (HPEC)*, 2012, pp. 1–5. (Cited on 27, 35.)

[56] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud Storage System," *Journal ACM Transactions on Computer Systems*, vol. 33, no. 3, pp. 1–55, Sep. 2015. (Cited on 27.)

[57] K. Qiu, S. Huang, Q. Xu, J. Zhao, X. Wang, and S. Secci, "ParaCon: A Parallel Control Plane for Scaling Up Path Computation in SDN," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 978–990, Dec. 2017. (Cited on 27.)

[58] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "A Performance Evaluation of Open Source Graph Databases," in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, 2014, pp. 11–18. (Cited on 27.)

[59] O. L. Barakat, "Gavel," https://github.com/engbarakat/Gavel/releases/tag/v1.0, 2018. (Cited on 30.)

[60] "Neo4j Database." [Online]. Available: https://neo4j.com/ (Cited on 31, 35.)

[61] M. Hassan, T. Kuznetsova, H. C. Jeong, W. Aref, and M. Sadoghi, "Extending In-Memory Relational Database Engines with Native Graph Support," in *EDBT: 21st International Conference on Extending Database Technology*. Vienna, Austria: OpenProceedings.org, 2018. (Cited on 33.)

[62] M. Paradies, W. Lehner, and C. Bornhövd, "GRAPHITE: An Extensible Graph Traversal Framework for Relational Database Management Systems," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, ser. SSDBM '15. New York, NY, USA: ACM, 2015, pp. 29:1–29:12. (Cited on 33.)

[63] M. A. Rodriguez and P. Neubauer, "A path algebra for multi-relational graphs," in *2011 IEEE 27th International Conference on Data Engineering Workshops*, Apr. 2011, pp. 128–131. (Cited on 33.)

[64] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: A Database Benchmark Based on the Facebook Social Graph," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1185–1196. (Cited on 34.)

[65] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An Evolving Query Language for Property Graphs," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 1433–1445. (Cited on 35, 36.)

[66] "ArangoDB - highly available multi-model NoSQL database." [Online]. Available: https://www.arangodb.com/ (Cited on 35.)

[67] "Titan: Distributed Graph Database." [Online]. Available: http://titan.thinkaurelius.com/ (Cited on 35.)

[68] F. Holzschuher and R. Peinl, "Performance of graph query languages," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, 2013, p. 195. (Cited on 35, 37.)

[69] "Franz Inc. - Semantic Web Technologies." [Online]. Available: https://franz.com/agraph/ (Cited on 35.)

[70] N. Martinez-Bazan, S. Gomez-Villamor, and F. Escale-Claveras, "DEX: A high-performance graph database management system," in *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*. IEEE, Apr. 2011, pp. 124–127. (Cited on 35.)

[71] "Graphbase AI Technology." [Online]. Available: https://graphbase.ai/ (Cited on 35.)

[72] "HypergraphDB - A Graph Database." [Online]. Available: http://www.hypergraphdb.org/ (Cited on 35.)

[73] "InfiniteGraph." [Online]. Available: http://www.objectivity.com/products/infinitegraph/ (Cited on 35.)

[74] "InfoGrid Web Graph Database." [Online]. Available: http://infogrid.org/trac/ (Cited on 35.)

[75] "Multi-Model Database | Graph Database | OrientDB." [Online]. Available: http://orientdb.com/ (Cited on 35.)

[76] A. Patrushev, "Shortest path search in real road networks with pgRouting," *Free and Open Soruce Software For Geospatial*, 2007. (Cited on 36.)

[77] “pox: The POX Controller,” Nov. 2017. [Online]. Available: https://github.com/noxrepo/pox (Cited on 37.)

[78] J. M. Halpern and C. Pignataro, “Service Function Chaining (SFC) Architecture,” Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7665.txt (Cited on 41.)

[79] N. Deo and C.-Y. Pang, “Shortest-path algorithms: Taxonomy and annotation,” *Networks*, vol. 14, no. 2, pp. 275–323, Jun. 1984. (Cited on 41, 66.)

[80] “Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet.” [Online]. Available: http://mininet.org/ (Cited on 43.)

[81] M. Al-Fares, A. Loukissas, and A. Vahdat, “A Scalable, Commodity Data Center Network Architecture,” in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, 2008, pp. 63–74. (Cited on 43.)

[82] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, “The Internet Topology Zoo,” *Selected Areas in Communications, IEEE Journal on*, vol. 29, no. 9, pp. 1765 –1775, Oct. 2011. (Cited on 43, 74.)

[83] P. Kalmbach, J. Zerwas, P. Babarczi, A. Blenk, W. Kellerer, and S. Schmid, “Empowering Self-Driving Networks,” in *Proceedings of the Afternoon Workshop on Self-Driving Networks*, ser. SelfDN 2018, 2018, pp. 8–14. (Cited on 58.)

[84] F. Lazzeri, G. Bruno, J. Nijhof, A. Giorgetti, and P. Castoldi, “Efficient label encoding in segment-routing enabled optical networks,” in *2015 International Conference on Optical Network Design and Modeling (ONDM)*, May 2015, pp. 34–38. (Cited on 67.)

[85] L. Davoli, L. Veltri, P. L. Ventre, G. Siracusano, and S. Salsano, “Traffic Engineering with Segment Routing: SDN-Based Architectural Design and Open Source Implementation,” in *2015 Fourth European Workshop on Software Defined Networks*, Sep. 2015, pp. 111–112. (Cited on 67.)

[86] A. Cianfrani, M. Listanti, and M. Polverini, “Translating Traffic Engineering outcome into Segment Routing paths: The Encoding problem,” in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Apr. 2016, pp. 245–250. (Cited on 67.)

[87] F. Aubry, D. Lebrun, S. Vissicchio, M. T. Khong, Y. Deville, and O. Bonaventure, “SCMon: Leveraging segment routing to improve network monitoring,” in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, Apr. 2016, pp. 1–9. (Cited on 67.)

[88] O. L. Barakat, D. Koll, and X. Fu, "Gavel: A fast and easy-to-use plain data representation for software-defined networks," *IEEE Transactions on Network and Service Management*, pp. 1–12, ©2019 IEEE, 2019. (Cited on 74.)