UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

CRISTIAN CLEDER MACHADO

ARKHAM: an Advanced Refinement Toolkit for Handling Service Level Agreements in Software-Defined Networking

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Alberto Egon
Schaeffer-Filho
Coadvisor: Prof. Dr. Lisandro Zambenedetti
Granville

Porto Alegre
February 2015

CIP – CATALOGING-IN-PUBLICATION

Machado, Cristian Cleder

ARKHAM: an Advanced Refinement Toolkit for Handling Service Level Agreements in Software-Defined Networking / Cristian Cleder Machado. – Porto Alegre: PPGC da UFRGS, 2015.

104 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Advisor: Alberto Egon Schaeffer-Filho; Coadvisor: Lisandro Zambenedetti Granville.

1. Policy-based management. 2. Policy refinement. 3. Software-defined networking. 4. Service level agreement. I. Schaeffer-Filho, Alberto Egon. II. Granville, Lisandro Zambenedetti. III. Título.

ACKNOWLEDGMENTS

I would like to thank God for giving me strength and health to successfully complete this journey of my life.

I would like to thank my mother Elenir, who has always been example of humility, perseverance, courage, honesty, and many others qualities. For you mom, my eternal thanks. You're my Idol. I love you.

My emotive thanks goes to my wife Bárbara for the unconditional support no matter the circumstances, for encouraging me on my professional projects, and for teaching me the great value of true love. Also, thanks for her patience, affection, and companionship. You make me a better person. Thank you for being my shoulder to cry on. This achievement was only possible because of you: my friend, my love, my life. I love you.

A special thanks to my advisor Prof. Alberto and my co-advisor Prof. Lisandro for all intellectual and philosophical lessons, experiences, opportunities, and criticisms, and also for all talks about life decisions. Today, I understand that all this was extremely important to turn me a better researcher. Also, thanks for believing in my potential and giving me the opportunity to show my talents and skills. I also thank the professors Luciano P. Gaspary, Liane M. R. Tarouco, Carlos A. Kamienski (UFABC), Cláudio F. R. Geyer, and Philippe O. Navaux for always requiring the best of me. Certainly my skills in producing research was great ly improved by each of you.

I can say that not only met friends that supported me and made me feel at Home, but I found a new family. I'm very thankful to all my great friends of the research lab 212. Thank you Anderson (Mr. Andersen), Gabriel (Doceiro), GerMano (Mano/Little brother), Juliano (no nickname), Lucas (Bondan), Luis (Bride[No offense!]), Pedro (Catarina/Peter), and Vinicius (Funai). The experience of working and exchanging ideas with all of you was very important to me, and the lessons I learned will follow me for life. I'm grateful to the group that we created for having fun and supporting each other: The Group of hipsters (Grupo dos descolados). The unconditional support, in good and bad moments, offered by this amazing group was essential for me.

Thanks to all the other friends of Computer Networks Group: Carlos (Ranieri), Cristiano, Daniel, Jéferson, Leonardo, Lucas (Müller), Maicon, Marcelo (Marotta), Márcio, Matheus (Cadori), Matheus (Ganso), Miguel, Oscar, Rafael (Esteves), Rafael (Hansen), Ricardo, and Rodolfo. Being member of this group added values to me that goes beyond Master's degree. Forgiveness if I forgot to mention someone. I thank to Marotta for having given me a powerful nickname, calling me Batman, a respect position for the skills that Batman has. It could have been worse.

I would like to thank Thiago for being a great friend and brother by understanding my absences in our company and unconditionally support me for achieving of this step of my life.

My thanks to my colleagues and friends from URI (Integrated Regional University): André, Dioni, Eliakim, Gustavo, Henrique, Lucas, Rafael, Rodrigo (big), Ronaldo (Maninho), and Vagner from Information Technology and Communication Department, while I was still working in this department and specially Éderson (Palmera), Érlon, and Rodrigo (Góis) by moments of friendship, fun, and for supporting me in the Network Management Sector when I was absent. Catiane, Giancarlo, Leandro, and Marcos, specially Maurício and Clicéres from Department of Computer Science for supporting and understanding the importance of this step in my professional career. Elisabete, Clóvis, Nestor, and Silvia for supporting, believing, and giving me the opportunity to develop my skills and reinforce my knowledge, making me a better professional.

Finally, I would like to thank the others friends, colleagues, professors, and staff of the Informatics Institute from UFRGS and from URI (as a whole), by their support during the accomplishment of this work. Thank you all!

AGRADECIMENTOS

Eu gostaria de agradecer a Deus por me dar força e saúde para concluir com êxito esta jornada da minha vida.

Eu gostaria de agradecer a minha mãe Elenir, que sempre foi exemplo de humildade, perseverança, coragem, honestidade, e muitas outras qualidades. Para você mãe, meu eterno agradecimento. Você é meu ídolo. Amo você.

Meu emotivo agradecimento vai para minha esposa Bárbara pelo apoio incondicional, não importando as circunstâncias, por encorajar meus projetos profissionais, e por me ensinar o grande valor do verdadeiro amor. Além disso, obrigado por sua paciência, carinho e companheirismo. Você me tornar uma pessoa melhor. Obrigado por ser meu ombro para chorar. Essa conquista só foi possível por causa de você: minha amiga, meu amor, minha vida. Eu te amo.

Um agradecimento especial ao meu orientador Prof. Alberto e meu co-orientador Prof. Lisandro por todas as lições intelectuais e filosóficas, experiências, oportunidades e críticas, e também por todas as conversas sobre decisões de vida. Hoje, eu entendo que tudo isso foi extremamente importante para me tornar um melhor pesquisador. Além disso, obrigado por acreditarem no meu potencial e me dar a oportunidade de mostrar meus talentos e habilidades. Agradeço também aos professores Luciano P. Gaspary, Liane M. R. Tarouco, Carlos A. Kamienski (UFABC), Cláudio F. R. Geyer e Philippe O. Navaux por sempre exigir o melhor de mim. Certamente minhas habilidades na produção de pesquisas foram melhoradas por cada um de vocês.

Posso dizer que não só encontrei amigos que me apoiaram e me fizeram sentir em casa, mas eu encontrei uma nova família. Eu sou muito grato a todos os meus grandes amigos do laboratório de pesquisa 212. Obrigado Anderson (Mr. Andersen), Gabriel (Doceiro), GerMano (Mano/little Brother), Juliano (no nickname), Lucas (Bondan) , Luis (Bride[Sem ofensas!]), Pedro (Catarina/Peter) e Vinicius (Funai). A experiência de trabalhar e trocar ideias com todos vocês foi muito importante para mim, e as lições que aprendi vão me seguir por toda a vida. Sou grato ao grupo que criamos para descontração e apoiar uns aos outros: O Grupo dos Descolados. O apoio incondicional, nos bons e maus momentos, oferecido por este incrível grupo foi essencial para mim.

Obrigado a todos os outros amigos do grupo de Redes de Computadores: Carlos (Ranieri), Cristiano, Daniel, Jéferson, Leonardo, Lucas (Müller), Maicon, Marcelo (Marotta), Márcio, Matheus (Cadori), Matheus (Ganso), Miguel, Oscar, Rafael (Esteves), Rafael (Hansen), Ricardo e Rodolfo. Ser membro deste grupo acrescentou valores que vai além do Mestrado. Perdão se eu esqueci de mencionar alguém. Agradeço ao Marotta por ter me dado um poderoso apelido, me chamando de Batman, uma posição de respeito pelas habilidades que o Batman possui. Poderia ter sido pior.

Gostaria de agradecer ao Thiago por ser um grande amigo e irmão compreendendo minhas ausências em nossa empresa e me apoiar incondicionalmente para a realização desta etapa da minha vida.

Meus agradecimentos aos meus colegas e amigos da URI (Universidade Regional Integrada): André, Dioni, Eliakim, Gustavo, Henrique, Lucas, Rafael, Rodrigo (Grande), Ronaldo (Maninho) e Vagner do Departamento de Tecnologia da Informação e Comunicação, enquanto eu ainda estava trabalhando neste departamento e especialmente Éderson (Palmera), Érlon e Rodrigo (Góis) por momentos de amizade, diversão, e por me apoiar no Setor de Gerência de Rede, quando eu estava ausente. Catiane, Giancarlo, Leandro, e Marcos, especialmente ao Maurício e Clicéres do Departamento de Ciência da Computação pelo apoio e compreensão da importância desta etapa na minha carreira profissional. Elisabete, Clóvis, Nestor e Silvia por apoiar, acreditar, e me dar a oportunidade de desenvolver minhas habilidades e reforçar o meu conhecimento tornando-me um profissional melhor.

Finalmente, gostaria de agradecer ao restante dos amigos, colegas, professores e funcionários do Instituto de Informática da UFRGS e da URI (como um todo), por seu apoio durante a realização deste trabalho. Obrigado a todos.

"Intelligence is the ability to adapt to change".

— STEPHEN HAWKING.

ABSTRACT

Software-Defined Networking (SDN) aims to provide a more sophisticated and accurate architecture for managing and monitoring network traffic. SDN permits centralizing part of the decision-making logic regarding flow processing and packet routing in controller devices. Despite this, the behavior of network devices and their configurations are often written for specific situations directly in the controller. This becomes an issue when there is an increase in the number of network elements, links, and services, resulting in a large amount of rules and a high overhead related to network configuration. As an alternative, techniques such as Policy-Based Management (PBM) and policy refinement can be used by high-level operators to write Service Level Agreements (SLAs) in a user-friendly interface without the need to change the code implemented in the controllers. However, policy refinement in the new research area of SDN has been a neglected topic, in part, because refinement is a nontrivial process. When using SLAs, their translation to low-level policies, *e.g.*, rules for configuring switching elements, is not straightforward. If this translation is not performed properly, the system elements may not be able to meet the implicit requirements specified in the SLA. In this context, we introduce ARKHAM: an Advanced Refinement Toolkit for Handling Service Level Agreements in Software-Defined Networking. This work presents (*i*) a Policy Authoring Framework that uses logical reasoning for the specification of business-level goals and to automate their refinement; (*ii*) an OpenFlow controller which performs information gathering and configuration deployment; and (*iii*) a formal representation using event calculus that describes our solution. As a result, our approach is capable of identifying the requirements and resources that need to be configured in accordance with SLA refinement, and can successfully configure and execute dynamic actions for supporting infrastructure reconfiguration.

Keywords: Policy-based management. Policy refinement. Software-defined networking. Service level agreement.

ARKHAM: um avançado conjunto de ferramentas de refinamento para manipulação de acordos de nível de serviço em redes definidas por software

RESUMO

Redes definidas por software (*Software-Defined Networking* – SDN) tem como objetivo fornecer uma arquitetura mais sofisticada e precisa para gerenciar e monitorar o tráfego da rede. SDN permite centralizar parte da lógica de tomada de decisão sobre o processamento de fluxo e roteamento de pacotes em dispositivos chamados controladores. Apesar disso, o comportamento dos dispositivos de rede e suas configurações são muitas vezes escritos para situações específicas diretamente no controlador. Isto torna-se um problema quando há um aumento no número de elementos, ligações e serviços de rede, resultando numa grande quantidade de regras e uma elevada sobrecarga relacionada à configuração da rede. Como alternativa , técnicas, tais como gerenciamento baseado em políticas (*Policy-Based Management* – PBM) e refinamento de políticas podem ser utilizadas por operadores de alto nível para escrever Acordos de Nível de Serviço (*Service Level Agreements* – SLAs) em uma interface amigável, sem a necessidade de alterar o código implementado nos controladores. No entanto, o refinamento de políticas na nova área de pesquisa SDN tem sido um tema negligenciado, em parte, porque o refinamento não é um processo trivial. Ao utilizar SLAs, a sua tradução para políticas de baixo nível, por exemplo, regras para a configuração de elementos de comutação, não é simples. Se essa tradução não for realizada corretamente, os elementos do sistema podem não ser capaz de cumprir os requisitos implícitos especificados no SLA. Neste contexto, este trabalho apresenta ARKHAM: um avançado conjunto de ferramentas de refinamento para manipulação de acordos de nível de serviço em redes definidas por software. Este conjunto de ferramentas é composto por (*i*) um framework para criação de políticas que usa raciocínio lógico para a especificação de objetivos de nível de negócio e automatização de seu refinamento; (*ii*) um controlador OpenFlow que realiza a coleta de informações e implantação de configurações na rede; e (*iii*) uma representação formal de políticas de alto nível utilizando *Event Calculus* e aplicando raciocínio lógico para modelar tanto o comportamento do sistema quanto o processo de refinamento de políticas para o gerenciamento de SDN. Como resultado, a abordagem é capaz de identificar as necessidades e os recursos que precisam ser configurados de acordo com o refinamento do SLA, podendo assim configurar e executar com sucesso ações dinâmicas de suporte à reconfiguração de infraestrutura.

Palavras-chave: Gerenciamento de baseado em políticas. Refinamento de políticas. Redes definidas por software. Acordo de nível de serviço.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

BDIM	*Business-Driven IT Management*
BLOs	*Business-Level Objectives*
DHCP	*Dynamic Host Configuration Protocol*
DMTF	*Distributed Management Task Force*
DNS	*Domain Name Service*
ECA	*Event-Condition-Action*
GB	*Gigabyte*
GUI	*Graphical User Interface*
ICMP	*Internet Control Message Protocol*
I/O	*Input/Output*
ICT	*Information And Communication Technology*
IETF	*Internet Engineering Task Force*
IP	*Internet Protocol*
KAOS	*Knowledge Acquisition in autOmated Specification*
LAN	*Local Area Network*
LDAP	*Lightweight Directory Access Protocol*
LLDP	*Link Layer Discovery Protocol*
MAC	*Media Access Control*
MB	*Megabyte*
MIB	*Management Information Base*
OP	*Orchestration Plane*
OSS	*Operations Support Systems*
P2P	*Peer-to-Peer*
PBM	*Policy-Based Management*
PBNM	*Policy-Based Network Management*
PCIM	*Policy Core Information Model*

PDL	*Policy Description Language*
PDP	*Policy Decision Point*
PEP	*Policy Enforcement Point*
PMT	*Policy Management Point*
PR	*Policy Repository*
QoS	*Quality of Service*
RFC	*Request For Comments*
RTT	*Round-Trip Time*
SDN	*Software-Defined Networking*
SLA	*Service Level Agreement*
SLO	*Service Level Objective*
SLS	*Service Level Specification*
SNMP	*Simple Network Management Protocol*

CONTENTS

1 INTRODUCTION

For many years, organizations have developed management strategies for dealing with the scale and computational complexity in Information and Communications Technology (ICT) infrastructures. The emergence of Software-Defined Networking (SDN) (NUNES et al., 2014) aims to provide a more sophisticated and accurate architecture for managing and monitoring network traffic. It provides an extensible platform for delivery of network services, capable of responding quickly to service requirement changes. To accomplish this, SDN simplifies network management by removing part of the decision-making logic from the switching elements. These decisions can be centralized in a network component called controller, while switching elements become simple packet forwarding devices. Thus, controllers can have a global view of the network traffic, and switching elements can be configured through an open interface, such as the OpenFlow protocol (MCKEOWN et al., 2008), independently from the hardware (BAKSHI, 2013). As a result, this allows an effective way for providing dynamically – and at runtime – services that support, for example, Quality of Service (QoS) reconfiguration, access control, and load balancing (SEZER et al., 2013).

1.1 Problem and Motivation

Despite the benefits of SDN, the intended behavior of network devices are usually defined by static rules written to cope with specific situations directly in the controller (MONSANTO et al., 2013; SEZER et al., 2013; HYOJOON; FEAMSTER, 2013). As a result, this programming model is susceptible to traditional problems in the network management domain, such as:

- human work overload due to the need of writing a large set of rules;
- limit or hinder the development and deployment of new services and resources that were not anticipated when hard-coded rules were written; and
- low-level rules that do not faithfully fulfill high-level goals, as network programmers may not be aware of business goals.

A possible solution to alleviate these problems is the use of Policy-Based Management (PBM). The use of PBM aims to reduce the complexity of network management tasks, enabling the system to gain a certain level of autonomy. PBM approaches can be used to govern complex network infrastructures through a set of rules rather than managing device-by-device configurations (HAN; LEI, 2012). In PBM systems an administrator specifies the infrastructure objectives/goals and restrictions in the form of rules to govern the behavior of the managed system elements. In addition, policy refinement techniques can be used to automatically translate high-level policies – *e.g.*, specified as a Service Level Agreement (SLA) – into a set of low-level ones, which can be enforced directly by network devices. Policies can be specified for each element, group or domain, targeting the operation and requirements of the network infrastructure.

Additionally, at runtime, the system may receive information from the infrastructure, and new rules can be dynamically deployed, supporting adaptive system behavior. Thus, the behavior of network components can be modified without the need to recode them, and without human intervention or the need to stop the system (VERMA, 2002).

The use of PBM and, in particular, policy refinement for management of traditional networks has been investigated with relative success (BANDARA et al., 2004; BANDARA et al., 2005; WALLER et al., 2006; CRAVEN et al., 2011). However, we argue that policy refinement in the new research area of SDN has been a neglected topic, in part, because refinement is a nontrivial process. When using SLAs, their translation to low-level policies, *e.g.*, rules for configuring switching elements, is not straightforward. If this translation is not performed properly, the system elements may not be able to meet the implicit requirements specified in the SLA (CRAVEN et al., 2011). For this reason, when applying policy refinement, specific problems must be addressed:

- determining which resources are needed to fulfill the policy requirements;
- translating high-level policies into operational policies that the system can enforce; and
- checking whether or not the low-level policies are accurate and faithfully satisfy the requirements specified by the high-level policy (BANDARA et al., 2004).

1.2 Aims and Main Contributions

In this context, we investigate in this work a solution for the refinement of high-level policies (expressed in a controlled natural language) into SDN rules to be enforced by the network controller. We specifically introduce ARKHAM: an advanced refinement toolkit for handling service level agreements in software-defined networking. This toolkit presents:

- a Policy Authoring Framework that uses logical reasoning for the specification of business-level goals and to automate their refinement. On the one hand, abductive reasoning assists Policy Authoring Framework operators in identifying the QoS classes that can support the specifications of the SLA. On the other hand, inductive reasoning assists operators in specifying system parameters in the configuration of QoS classes that can fulfill requirements of an SLA;
- an OpenFlow controller which performs information gathering, which is later used, for example, to calculate optimal routes. Additionally, it configures the rules in the switches for optimization of network resource usage at runtime; and
- a Formal Representation of high-level SLA policies using Event Calculus (EC) and the used of logical reasoning to model both the system behavior and the policy refinement process in SDN architectures. It is our aim with this formalism to assist network operators to develop refinement tools and configuration approaches to achieve more robust SDN deployments. Thus, the development of the refinement approaches becomes independent

of the network controller implementation or policy language used.

To the best of our knowledge, this is the first time that policy refinement has been applied to SDN. The development of a policy refinement strategy specifically for SDN can benefit from several aspects found in these environments. Firstly, it is possible to more easily collect monitoring information and network traffic data, with the aim of checking whether high-level goals are being fulfilled by low-level rules or not. Secondly, we have the ability to refine policies to lower level ones, which can be (re)implemented in any network device, since the SDN controller uses a standard protocol and interface.

The main contributions of this work are:

- refined policies with minimal human intervention;
- analysis of the infrastructure's ability to fulfill the requirements of high-level policies;
- decreased amount of network rules coded into the controller; and
- management and deployment of new rules with minimal disruption to the network.

In order to obtain better analysis and results, we have limited our scope to scenarios for QoS management. As a proof-of-concept, we executed several experiments considering different SLAs by changing the number of expressions and their increasing complexity, different scenarios by changing the number of network elements, different populations stored in the repository, among others. Further, to validate our formal representation, we present some case studies that show the representation of network infrastructure and their elements, services, QoS classes, and SLAs. Furthermore, to validate the use of logical reasoning and event calculus, we present case studies that illustrate the end-to-end decomposition of rules and demonstrate how our refinement solution performs. In addition, our formalism demonstrated that policies can be designed, analyzed and, if necessary, refined and/or adapted before they are deployed in the network infrastructure.

1.3 Document Outline

This dissertation is organized as follows. Chapter 2 provides an overview of the main concepts employed in our solution. We present in details the refinement solution developed, together with the elements, techniques, and concepts that are part of the same in Chapter 3. In Chapter 4 we define and demonstrate a formal representation of high-level SLA policies to model both the system behavior and the policy refinement process for SDN management. We present the prototype, experiments, and a discussion about the achieved results in Chapter 5. In Chapter 6 we discuss the related work in this research area. Finally, in Chapter 7 we conclude this work with final remarks, along with a proposal for future work.

2 BACKGROUND

This chapter presents an overview of the main elements, techniques, and concepts involved in our policy refinement toolkit. Below, we start with a brief discussion of the main concepts of SDN with OpenFlow in Section 2.1. In Section 2.2 we explain the notions of Policy-Based Management (PBM) and Policy Refinement.

2.1 Software-Defined Networking (SDN)

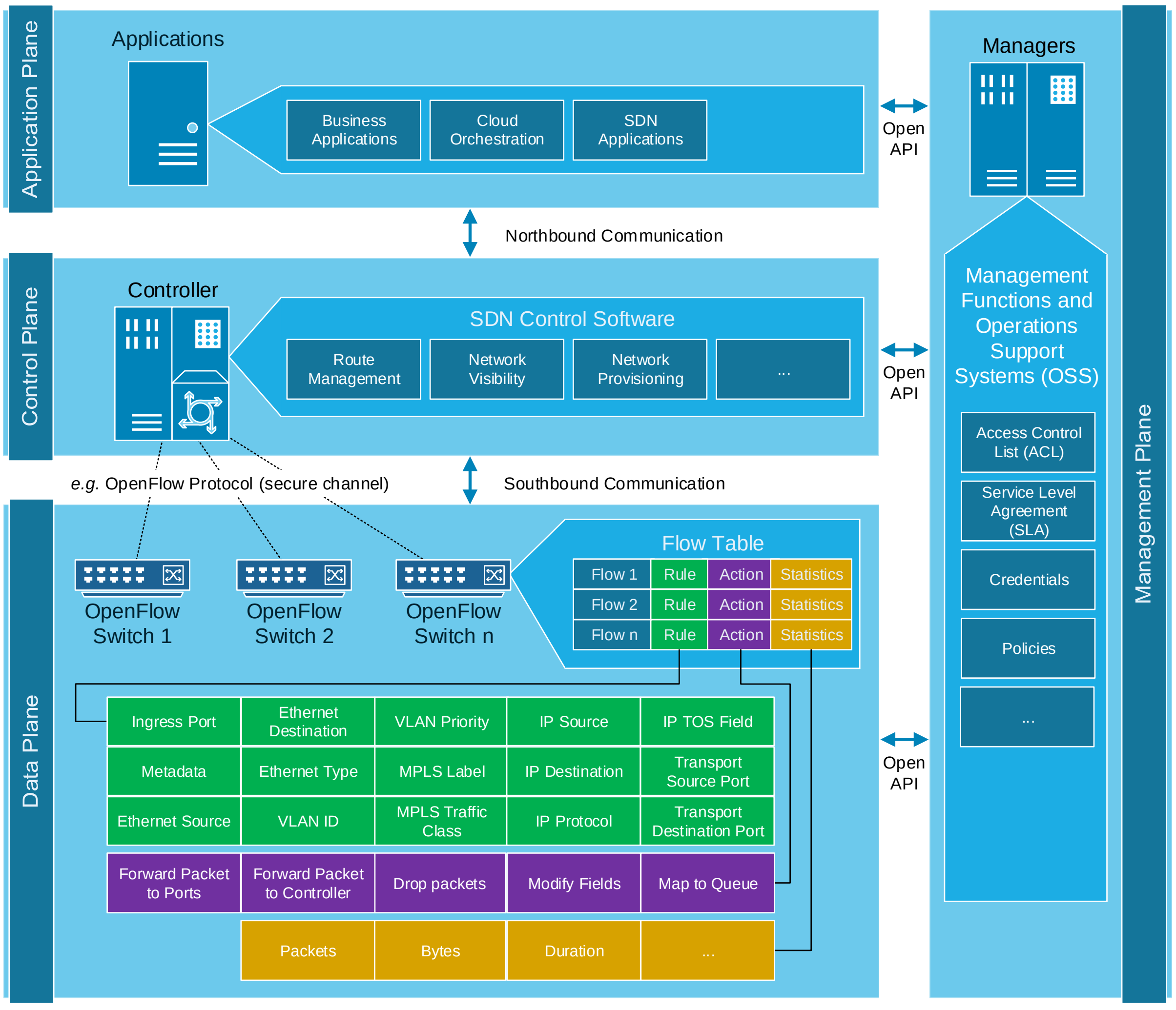
Software-Defined Networking is a dynamic, adaptable, controllable, and flexible network architecture. It provides an extensible platform for delivery of network services, capable of responding quickly to service requirement changes (DAVIS et al., 2012). An SDN architecture comprises four planes: control plane, data plane, application plane, and management plane (BETTS et al., 2014). The control plane is responsible for the protocols and the decision making that result in the updating of forwarding tables. The data plane, known as the forwarding plane, manages the switching and routing of network packets. The application plane includes SDN applications (*e.g.*, firewalls, load balancers), business applications (*e.g.*, e-commerce portals, enterprise management systems) or Cloud Orchestration (*e.g.*, OpenStack, CloudStack). Each application has exclusive control of a set of resources provided by SDN controllers. The management plane includes management systems performing operations and functions to support the infrastructure, *e.g.* SLAs and low-level policies to drive SDN applications and SDN controllers.

In traditional networks, the control plane is executed in each network device. Each device has its proprietary protocols becoming difficult to be programmed. Often is not possible to carry out the management decision-making that has not been anticipated. Differently, SDN is characterized by a logically centralized control plane, which allows moving part of the decision-making logic of network devices to external controllers. This provides controller devices with the ability to have an overall view of the network and its resources, thus becoming aware of all the network elements and their characteristics. Based on this centralization, network devices become simple packet forwarding elements, which can be programmed through an open interface, such as the OpenFlow protocol (WICKBOLDT et al., 2015; NUNES et al., 2014).

OpenFlow is an open protocol that allows the development of programmable mechanisms based on a flow table in different forwarding devices. The OpenFlow protocol establishes a secure communication channel between OpenFlow switches and the controller, using this channel for controlling and establishing flows according to customizable programs (MCKEOWN et al., 2008).

Briefly, the main elements of an OpenFlow-based SDN architecture are (depicted in Figure 2.1): (*i*) a flow table in each switch containing entries for each active flow; (*ii*) a controller that executes customized programs to decide which rules and actions are going to be installed to control packet forwarding in each switch element; and (*iii*) an abstraction layer that communicates securely with a controller reporting on new input flows that are not present in the flow table. Each entry in the flow table consists of: (*a*) a mask of fields found in the packet header, which is used to match the incoming packets, (*b*) counters for collecting statistics for each specific flow, such as number of bytes, number of packets received, and flow duration, and (*c*) a series of actions to be performed when a packet matches the corresponding mask (BAKSHI, 2013; BETTS et al., 2014).

Figure 2.1: OpenFlow-Based SDN architecture.



Source: Betts et al. (2014).

SDN is characterized by the separation between the data plane and the control plane. Thus, with the OpenFlow protocol, on the one hand, the data plane is concerned with the forwarding of packets based on rules, called OpenFlow actions, associated with each table entry in the switch. On the other hand, the control plane enables the controller to manage the entries in the flow table associated with the desired traffic (BAKSHI, 2013; MCKEOWN et al., 2008).

2.2 Policy-Based Management (PBM)

Policies are defined as a collection of rules which express and enforce the required behavior of a resource. RFC 3198 (WESTERINEN et al., 2001) provides the following definitions for a policy:

- A defined goal or action that determines how present and future decisions are taken. Policies are established or executed within a particular context;
- Policies refer to a set of rules to manage and monitor access to features of a particular ICT infrastructure (MOORE et al., 2001).

In Policy-Based Management (PBM) systems an administrator specifies the infrastructure objectives/goals and constraints in the form of rules to guide the behavior of the elements in a system (VERMA, 2002). The use of PBM presents three main benefits (HAN; LEI, 2012). Firstly, policies are predefined by administrators and stored in a repository. When an event occurs, these policies are requested and accessed automatically, without the need of manual intervention. Secondly, the formal description of policies permits automated analysis and verification with the aim of guaranteeing consistency to some extent. Thirdly, because of the abstraction of technical details, policies can be inspected and changed dynamically at runtime without modifying the underlying system implementation.

Policies may be seen in two principal levels of abstraction: *low-level policies*, which are related to a domain or a device, and *high-level policies* that are more user-friendly. A simple example of a low-level policy is the settings on routers so multimedia traffic packets have higher priority over peer-to-peer (P2P) traffic packets. An example of a high-level policy is an Service Level Agreement (SLA) (BLAKE et al., 1998). In PBM, using techniques for refinement, a high-level policy such as an SLA can be translated into low-level policies that are applicable in various elements of a system (BANDARA et al., 2004; MOFFETT; SLOMAN, 1993). SLAs are generally business-oriented, and they leave aside the technical details, which are guided by a Service Level Specification (SLS) and a Service Level Objective (SLO). The SLS is a technical interpretation of the SLA. The SLO is a sub-item of the SLS that contains the parameters to achieve the SLS (AIB; BOUTABA, 2007).

2.2.1 Typical Categories of Policy-Based Management

PBM approaches can be categorized by observing they management focus and environment elements. In the following, we introduce some of the main classifications:

- **Policy-Based Network Management (PBNM) –** In PBNM, the main goal is to specify how the devices and network resources comply with system operation requirements. From the view point of the network operation, the use of PBNM aims to reduce the complexity of the network management tasks allowing the system to gain a certain autonomy-level (HAN; LEI, 2012). Several network devices, whether domestic or corporate, support some type of management. For example, home-routers can support the IP addresses distribution using a Dynamic Host Configuration Protocol (DHCP) server, binding the MAC address of client device interface. Further, management systems that support quality of service guarantees can be programed to configure each network device, removing the overload of manual work.

- **Policy-Based Security Management –** In Policy-Based Security Management there is a focus on system resources protection, either physically (*e.g.*, environments) or logically (*e.g.*, confidential data, copyright, and user privacy) (WALLER et al., 2011). Many security-related problems are linked to application developers. Oftentimes, developers are unaware of the safety practices, creating applications that only meet business requirements, not realizing security and compliance tests. In this management type, problems such as connection ports open, encryption changes, among others, can be monitored, identified and corrected.
- **Business-Driven IT Management (BDIM) –** Organizations have high expectations on their computing infrastructure to achieve their Business-Level Objectives (BLOs). However, there are several critical management problems due to complexity of such systems (FITO et al., 2012). BDIM aims to monitor and measure IT resources for investment recommendations, such as purchase of equipment in order to improve the business.
- **Policy-Based Cloud Management –** Cloud computing offers the outsourcing of resources and services, reducing infrastructure costs for both providers and customers. This does not allow customers to outsource the responsibility of the confidentiality, integrity, and access control to providers (PUESCHEL; PUTZKE; NEUMANN, 2012). In the same direction, also does not allow providers to be able to predict and provide efficiently resource because the service range is not fully estimated yet. Because of this, and also by the fact that cloud computing is transparent to programmers and users, it induces challenges that were not present in previous models of computation (SQUICCIARINI; PETRACCA; BERTINO, 2012). Thus, Policy-Based Cloud Management is still an open question and its concept is very wide and undefined. However, we can apply various techniques built for Distributed Systems Management such as Quality of Service systems, Security systems, among others (VILLEGAS et al., 2012).

2.2.2 Policy Languages Paradigms

Paradigms may contain specific components such as (HAN; LEI, 2012):

- **Event –** It is an occurrence in the system which was specified in a rule and was being monitored. Event triggers the policy to be enforced.
- **Condition –** It is a predicate expression that can be evaluated as True, False or Not Ap-

plicable. There are several conditions, such as run-time or waiting for something, a type of application, among others. If the condition is met (True), then the next step (Action) should be executed.

- **Action –** It is the execution of something that was determined in the policy. An action can generate an action set, *e.g.*, the action "RebootServer" can trigger other actions such as verifying the online users, save and close open programs, among others.

In general, rules in most policy languages are based on the following paradigms (HAN; LEI, 2012): Condition-Action paradigm and Event-Condition-Action paradigm. In Condition-Action paradigms policies are implemented through a set of rules. Each policy rule is built up by a set of conditions having a set of actions. Thus, this type of policy rule follows an "***IF*** *(Condition)* ***THEN*** *(Action)*". When the conditions of a policy rule are **true** the corresponding actions can be performed. The Event-Condition-Action (ECA) paradigm is similar to the Condition-Action paradigm. The only difference is the need of an event to occur in the system to trigger the execution of the policy. This type of policy rule follows an "***ON*** *(Event)* ***IF*** *(Condition)* ***THEN*** *(Action)*".

2.2.3 Basic Entities

IETF/DMTF introduces four basic entities to model the architecture of a policy-based system (WALLER et al., 2011):

- **Policy Management Tool (PMT) –** allows the administrator to manage policies;
- **Policy Repository (PR) –** stores policy-related information;
- **Policy Decision Point (PDP) –** searches, verifies, and validates the necessary conditions for policies;
- **Policy Enforcement Point (PEP) –** executes and monitors policies also providing feedback of relevant information during runtime.

The key of this architecture is the PDP. PDP exerts a important part of the control processing in the system. In this entity, high-level policies are translated into actions understood by system elements, and remain awaiting at sometime be executed in the PEPs. For correct operation, the PDP should differentiate every detail of each PEP in the system to provide increased accuracy and refinement in each policy.

In this architecture type, administrators define management policies that are inserted into the PR –, *e.g.*, Lightweight Directory Access Protocol (LDAP) – through a PMT. After that, a PDP performs event monitoring on the system, following the administrator settings. When a specific events occur, the PDP will be triggered to retrieve from the PR applicable policies to each individual case. For each policy retrieved from the specific event, when specific conditions are met, the corresponding actions are enforced by the PEP associated with the monitored element.

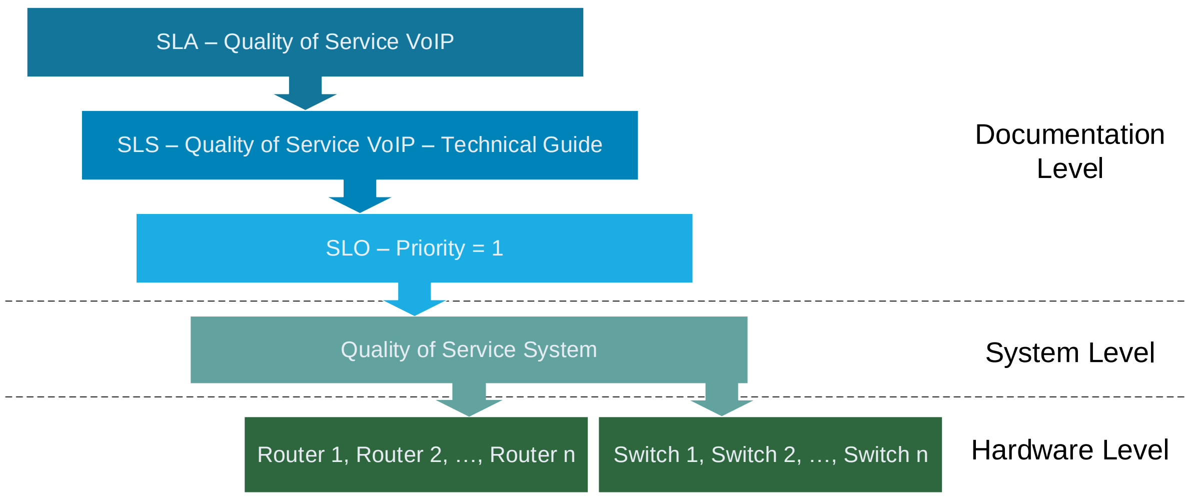
2.2.4 SLA Policy Refinement

Policy refinement aims to translate a high-level policy into a set of corresponding low-level policies. In other words, using techniques for refinement, a high-level policy such as a Service Level Agreement (SLA) can be translated into low-level policies that are applicable in various elements of a system (BANDARA et al., 2004; MOFFETT; SLOMAN, 1993).

SLAs are generally business-oriented, and they leave aside the technical details, which are guided by a Service Level Specification (SLS) and a Service Level Objective (SLO). The SLS is a technical interpretation of the SLA. The SLO is a sub-item of the SLS that contains the parameters to achieve the SLS (AIB; BOUTABA, 2007).

Figure 2.2 presents an overview of the process of refining SLAs. The SLA, SLS and SLO represent, respectively, the *documentation* describing the service in a formal way, the technical form (guide) for its functioning requirements, and the parameters aimed at quality and satisfaction. At the *system level*, a *quality of service* system interprets the management requirements, and enforces policies for the configuration of elements at the *hardware level*.

Figure 2.2: Example of SLA policy refinement.


SLA – Quality of Service VoIP
SLS – Quality of Service VoIP – Technical Guide
SLO – Priority = 1
Quality of Service System
Router 1, Router 2, ..., Router n
Switch 1, Switch 2, ..., Switch n
Documentation Level
System Level
Hardware Level

Source: by author (2015).

The refinement process typically involves stages of decomposition, operationalization, implementation, operation and re-refinement of goals and subgoals (CRAVEN et al., 2011; CRAVEN et al., 2010). Policy refinement aims to automate these stages to get the translation of policies relating to objects and implementable actions, and ensure that the low-level policies still satisfy the goals defined by the high-level policy.

The main objectives of policy refinement are identified by Moffett and Sloman (MOFFETT; SLOMAN, 1993) as:

- To determine what resources are needed to fulfill policy needs;

- To translate the high-level policy into a set of operational policies that the system can enforce;
- To examine whether the low-level policies actually meet precisely the requirements specified by the high-level policy.

3 POLICY REFINEMENT TOOLKIT

In this chapter we described ARKHAM: an **A**dvanced **R**efinement tool**K**it for **H**andling service level **A**gree**M**ents in software-defined networking. Our toolkit introduces the use of PBM paradigms to solve problems found in SDN architectures such as static rules and configurations often written for specific situations directly in the controller. Using PBM we reduced the amount of static rules and configurations writing more generic code which deploys specific rules obtained from a policy repository.

Our solution borrows ideas from previous investigations (BANDARA; LUPU; RUSSO, 2003; BANDARA et al., 2005; CRAVEN et al., 2011), which have been limited due to the characteristics of traditional IP networks, such as best-effort packet delivery and distributed control state in forwarding devices (SEZER et al., 2013). Thus, simultaneously, we leverage SDN's main features to enhance the policy refinement process.

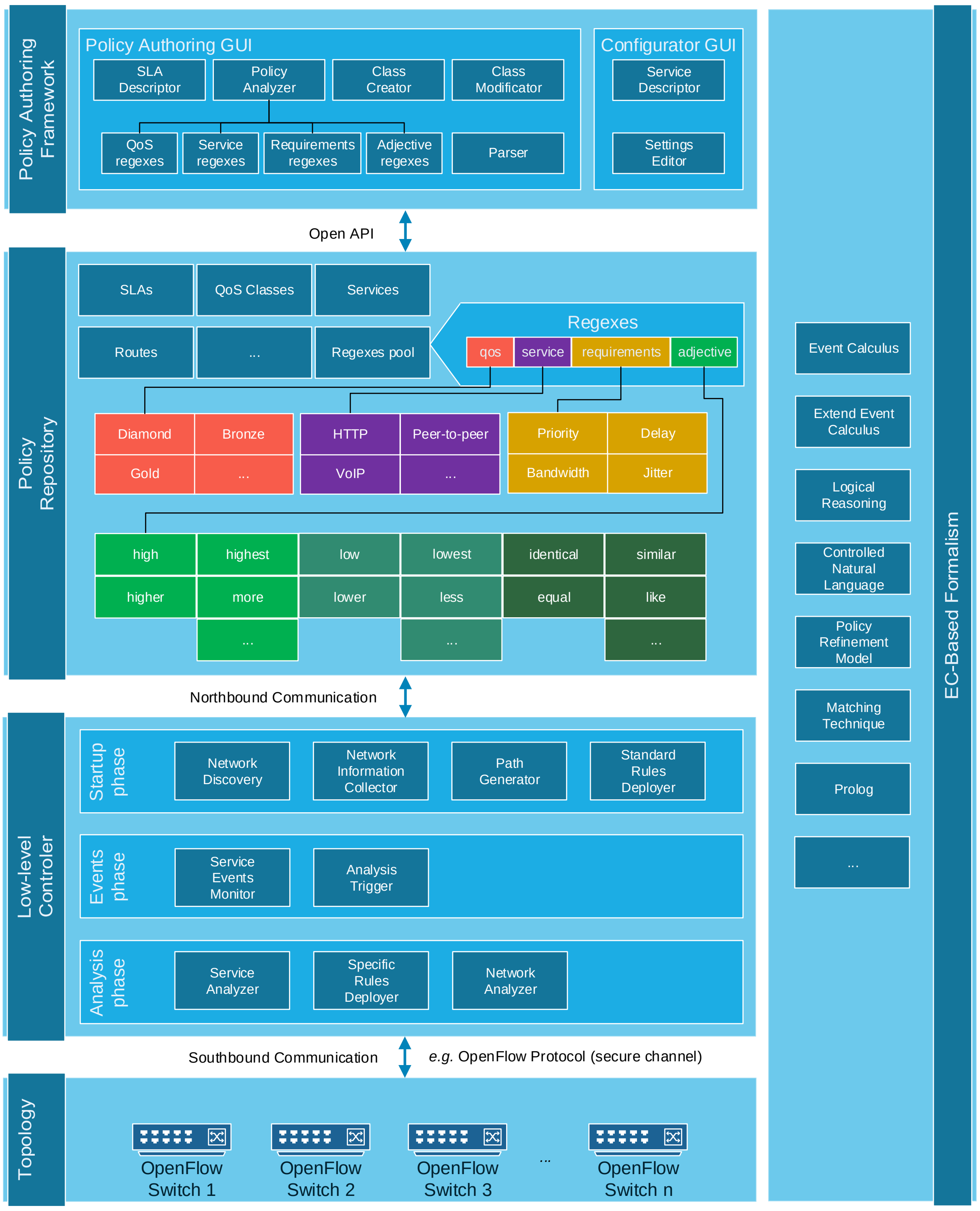
In SDN, these limiting factors of traditional networks can be overcome, since in implementations such as OpenFlow the switching elements have their control plane centralized in an element called controller. The controller receives information from all network elements and so can have an overall view of what happens in the network infrastructure. This architecture where network traffic information is centralized in a controller is valuable to our policy refinement solution. It makes it easier to retrieve information from the network infrastructure, and to validate SLA requirements more accurately.

Below, we present an overview of our policy refinement toolkit in Section 3.1. In Section 3.2 we introduce a low-level controller to collect network information and deploy rules. Finally, we describe a policy authoring framework to write and refine SLAs in Section 3.3.

3.1 Policy Refinement Toolkit: An Overview

In this section we present conceptually our policy refinement toolkit (Figure 3.1). In order to obtain better results, we have limited our scope to QoS management. It is worth noting that our QoS management is based on routing. QoS mechanisms allow network administrators to use existing resources efficiently and ensure the required level of service without the need of expanding or over provisioning their networks. However, to ensure that QoS requirements are satisfied across the network is difficult, as network devices such as switches and routers are heterogeneous and have proprietary interfaces. Moreover, QoS architectures such as DiffServ (BLAKE et al., 1998) and IntServ (BRADEN; CLARK; SHENKER, 1994) are built over current networks. These are based on distributed hop-by-hop routing, without a broader perception of global, network-wide capabilities. Thus, QoS management is a suitable case-study to be shown.

Figure 3.1: Overall Policy Refinement Toolkit.



Source: by author (2015).

3.1.1 Main Components

Our toolkit consists of several components, process, concepts, and techniques that are placed inside the following fundamental elements:

- **OpenFlow Controller –** its operation is divided into three phases: *(i) Startup Phase*: discovers network elements and their possible paths. In addition, it performs information gathering such as number of hops, delay, and available bandwidth for each path. Moreover, it deploys standard rules with the idea of best-effort packet delivery; *(ii) Events Phase*: identifies service events and analyzes the duration of each flow to propose modifications to the network configuration triggering the *Analysis Phase*; and *(iii) Analysis Phase*: determines the best path based on the characteristics of the network and service requirements. Additionally, it implements the rules and monitors the *Events Phase* in order to identify possible enhancements to the active flows and reconfigure the infrastructure. The controller phases will be explained in more detail in Section 3.2.
- **Policy Authoring Framework–** is divided into two modules: (*i*) *Policy Authoring Graphical User Interface*: enables the writing of SLAs in a Controlled Natural Language (CNL) and automates their translation into the low-level controller configuration. The *Policy Analyzer Component* analyses the SLA requirements that match the *regexes* (regular expressions) stored in the *Policy Repository*, and uses abductive reasoning to suggest the more appropriate QoS class/classes to the SLA. In addition, if the network cannot accommodate the SLA requirements, the *Policy Analyzer* performs a process using inductive reasoning to suggest classes or the creation of a new class that can fulfill the SLA requirements; and (*ii*) *Configuration Graphical User Interface*: permits the specification of regular expressions and technical characteristics of services and their parameters such as TCP/UDP port numbers, service name. The policy authoring operations will be described in more details in Section 3.3.
- **Policy Repository –** stores both the information about the behavior of the infrastructure, which is obtained during the controller phases, and policy authoring operations. For example, the repository stores all the possible links between elements, number of elements, and available bandwidth, delay, and jitter. Additionally, the repository maintains a list of all services and their parameters, and some QoS classes. The information in this repository will be used later, in the *Events Phase* and *Analysis Phase* (see Section 3.2) and in the *Policy Authoring Operation* (see Section 3.3). Finally, the repository stores a wide range of regular expressions separated by type. The regular expressions will be presented in Section 3.1.2.
- **EC-based formalism –** permits the representation of high-level policies in the form of SLAs using Event Calculus (EC) and the use of logical reasoning to model both the system behavior and the end-to-end policy refinement process. For example, we describe the SLAs in EC, and use logical reasoning to derive the SLOs and QoS classes based on a system model description. We also model the state of routes and links maintained by the controller, and use logical reasoning to match the best route based on the requirements of the SLA. The formalism will be presented in more detail in Chapter 4.

As mentioned previously in Section 2.2.3, a policy-based system is composed of four basic entities: (*i*) *Policy Management Tool (PMT)* allows the administrator to manage policies; (*ii*) *Policy Repository (PR)* stores policy-related information; (*iii*) *Policy Decision Point (PDP)* searches, verifies, and validates the necessary conditions for policies; and (*iv*) *Policy Enforcement Point (PEP)* executes and monitors policies also providing feedback of relevant information during runtime. In the proposed Policy Refinement Tookit, these entities can be mapped as follows: a Policy Authoring Framework represents the PMT; a MySQL database represents the PR; an OpenFlow controller represents the PDP; and OpenFlow switches represent PEPs.

3.1.2 Controlled Natural Language (CNL)

We created a Controlled Natural Language (CNL) to establish restrictions and requirements for writing business-level goals. Thus, our CNL aims to improve translation between what the humans (business-level operators and infrastructure programmers) mean and what the Policy Refinement Toolkit need to do. In addition, our CNL is a base for creating natural and intuitive representations for formal notations. Furthermore, it can be adapted as needed for new applications, such as monitoring, firewalls, and access control. The grammar of the controlled natural language is defined below:

Listing 3.1: Grammar of the controlled natural language.

```
Language :→ (<QoS>|<Service>)<ModalVerb><Expression>
QoS :→ qos−regexes
Service :→ service−regexes
ModalVerb :→ should | should not
Expression :→ <Term>|<Term><Connective><Expression>
Term :→ <Parameter><Operator><Value>
Parameter :→ requirements−regexes
Operator :→ adjective−regexes
Value :→ v
Connective :→ and | or
```

Source: by author (2015).

Our toolkit uses *regexes* as a concise and flexible way of identifying strings of interest such as particular characters (*e.g.*, $>$, $<$, $=$, $\neq$, $\leq$, $\geq$) or words (*e.g.*, high, low, http, ftp, gold, silver). We defined the following types of *regexes*:

- *qos-regexes* – regular expression to identify QoS classes;
- *service-regexes* – regular expression to identify services;
- *requirements-regexes* – regular expression to identify service requirements;
- *adjective-regexes* – regular expression to identify adjectives in service requirements.

Table 3.1 shows examples of regular expressions that can be contained in an SLA. Figure 3.2 shows an SLA containing these regular expressions.

We have configured these requirements and grouped them into classes, because it is an easier way to deal with the increasing number of policies that have common goals. However, if at any time it is discovered that an SLA cannot be fulfilled or that some SLO cannot be achieved, the system is flexible enough to allow the necessary adjustments.

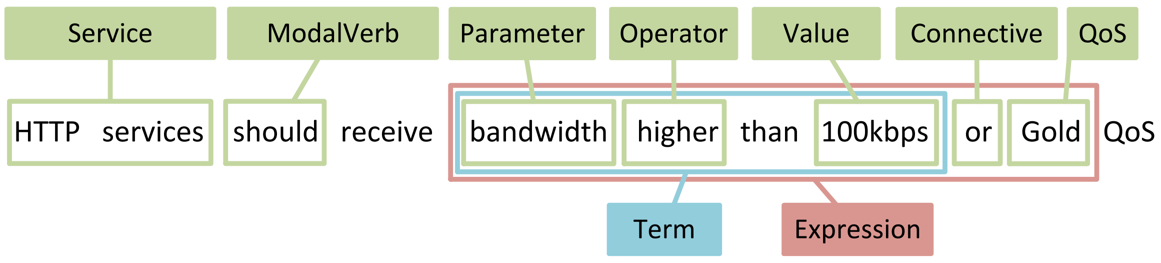
We apply the concept of logical reasoning to support the operator in the refinement of an SLA. We use two modes of logical reasoning: abduction to assist the Policy Authoring operator in identifying the QoS classes that can support the specifications of the SLA; and induction to assist the operator in specifying the system parameters in the configuration of the QoS classes that can meet an SLA.

Table 3.1: Examples of regular expression.

Type	**Expression**	
qos-regexes	Bronze, Silver, Gold, Platinum, Diamond ...	N/A
service-regexes	VoIP, Streaming, HTTP, FTP, SMTP, POP, P2P ...	N/A
requirements-regexes	Priority, Bandwidth, Delay, and Jitter	N/A
adjective-regexes	more, high, higher, highest, up, over ...	$>$
	equal, like, even, uniform, equable, same, similar ...	$=$
	less, low, lower, lowest, down, below ...	$<$

Source: by author (2015).

Figure 3.2: Example of occurrences of elements of our grammar in an SLA.



Source: by author (2015).

3.1.3 Refinement Process: Bottom-up and Top-down Stages

Policy refinement performs a process of derivation/translation of SLOs which must meet the SLA. These SLOs are considered QoS-class requirements, *e.g.*, priority, bandwidth.

As mentioned previously, we identified the business-level objectives and high-level policies as Service Level Agreements (SLAs). From this, our refinement process consists of a technique that extracts *regexes* from an SLA and decomposes them into meta-goals or Service Level Objectives (SLOs), *i.e.*:

```
SLA1→SLO1−1,
SLA1→SLO1−2,
SLA1→SLO1−3,
SLA1→SLO1−n.
```

These SLOs are identified from a query to the repository of *requirements-regexes*, such as delay (D), jitter (J), bandwidth (B), and priority (P), or *qos-regexes*, such as Diamond, or *service-regexes*, such as HTTP, according to Table 3.1. Thus,

```
SLO1−1→D,
SLO1−2→J,
SLO1−3→B,
SLO1−4→P.
```

or

```
SLO1−1→ qos-regexes
```

or

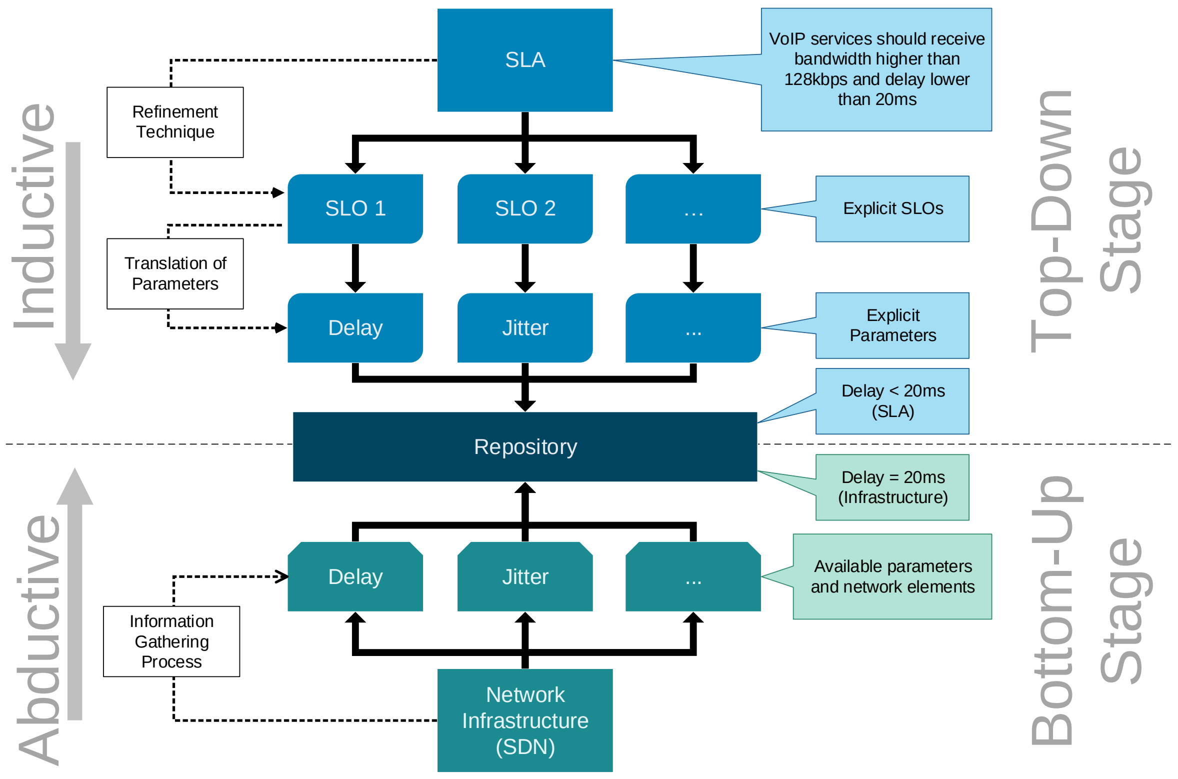
```
SLO1−1→ service-regexes
```

The refinement approach is split into two stage (Figure 3.3): The first stage, called *bottom-up*, consists of the network information (*e.g.*, bandwidth, delay) gathering process. A key element of this stage is the OpenFlow controller (described in Section 3.2) which performs the data collection process. The information gathered is stored in a repository. Using this information, the Policy Authoring framework (described in Section 3.3) uses *abductive reasoning* to indicate to the business-level operator what are the possible configurations. These indications are provided through settings performed previously – other SLAs or policies created manually by the business-level operator or by the infrastructure-level programmer – along with the characteristics that the network can support.

The second stage, called *top-down*, is a technique for the refinement of high-level goals, extracted from SLAs and translated into achievable goals (SLOs). An operator writes the SLAs and create – if necessary – the QoS classes needed to fulfill them. As mentioned previously, the *bottom-up* stage will try to indicate using *abductive reasoning* which are the best configurations for the SLA that is being written. Thus, multiple configuration options of policies will be offered to the operator, who can select or customize an existing configuration, or even create a new configuration.

After any selection or customization performed by the operator, the policies will be stored in the *Policy Repository*. Subsequently, the OpenFlow controller reads from the repository these new policies starting the *Analysis Phase* for setting up the rules in the switches. We use in our approach a formal representation both for the refinement of SLAs, as to confront information about the network infrastructure behavior. The formalization of the policy refinement approach

Figure 3.3: Deriving SLOs/parameters from goal and gathering network information.


SLA
VoIP services should receive bandwidth higher than 128kbps and delay lower than 20ms
Refinement Technique
SLO 1
SLO 2
...
Explicit SLOs
Translation of Parameters
Delay
Jitter
...
Explicit Parameters
Delay < 20ms (SLA)
Repository
Delay = 20ms (Infrastructure)
Delay
Jitter
...
Available parameters and network elements
Information Gathering Process
Network Infrastructure (SDN)
Inductive
Abductive
Top-Down Stage
Bottom-Up Stage

Source: by author (2015).

is based on Event Calculus (BANDARA; LUPU; RUSSO, 2003).

3.2 Low-level Controller Configuration

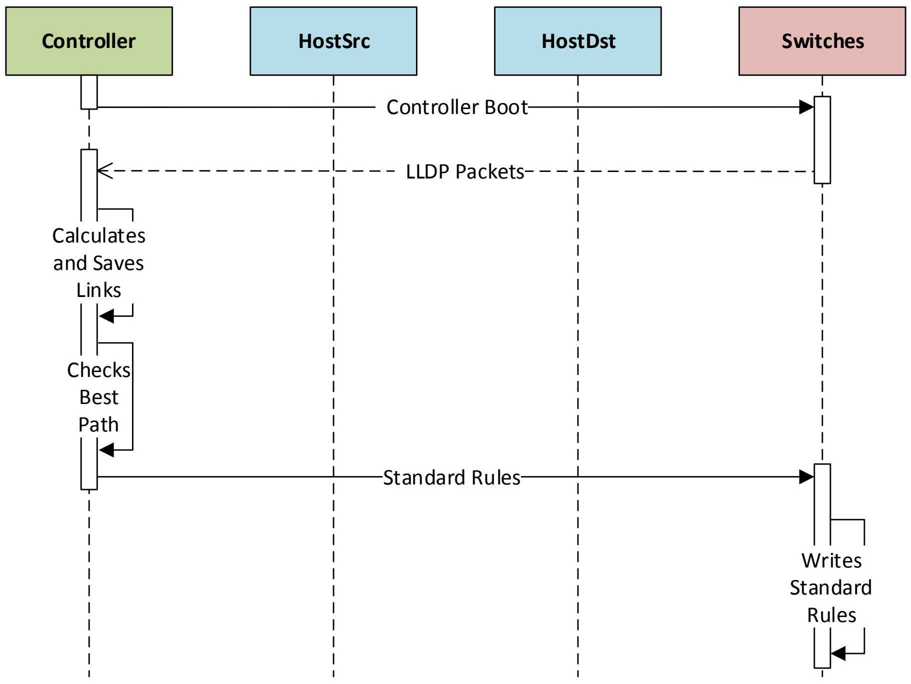
This section presents the operation of the controller, which is divided into three phases: *Startup Phase*, *Events Phase*, and *Analysis Phase*. These three controller phases handle the actions of the *bottom-up* stage. The operation of each phase is detailed in the following.

3.2.1 Startup Phase

In this phase (see Figure 3.4), the controller reads information from a repository that contains descriptions and requirements of all services that are initially scheduled to run in the network. This information is identified and set by the administrator as requirements to perform the low-level policies in order to fulfill the SLA definitions. Thereafter, the controller monitors the network in order to discover the devices and topology. Network devices are instructed to send Link Layer Discovery Protocol (LLDP) packets to report their location on the topology, which is stored by the controller in an internal data structure. While the LLDP packets are received and stored, a routine for calculating paths between all elements is performed using

Dijkstra's Shortest Path Algorithm (DIJKSTRA, 1959). We use a spanning tree library to find all the paths between the network devices that are being found. We assume at first that the best path is the one with the smallest number of hops, because at this stage values such as delay and jitter are unknown, thus we consider that these values are initially zero. Subsequently, all paths are sorted from the shortest to the longest path and stored in the repository.

Figure 3.4: Diagram of the StartUp Phase.



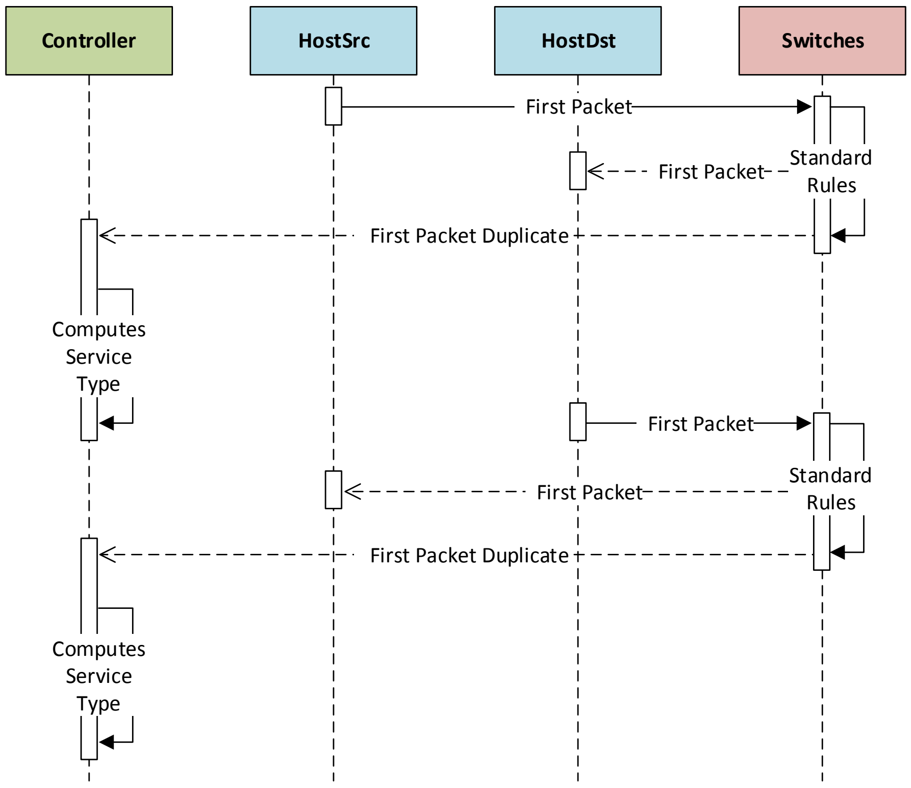
Source: by author (2015).

Once all possible paths between network elements have been calculated, the controller writes rules (which we call *standard rules*) in the flow table of the switches that are in the best path between each of the network elements. The mask that comprises the standard rule is very simple. It indicates that each packet that has the *Ingress Port* field will receive the actions *Forward Packet to Ports* and *Forward Packet to Controller*. Moreover, this rule has the lowest priority in the flow table of each network element. This priority has been established so that the standard rules do not conflict with the specific rules that will emerge. The purpose of these rules is to handle each new service flow in such a way that its first packet is duplicated, *(i)* forwarding one copy of the packet to the controller in order to inform it that there is a new service flow, which will be further evaluated (see Section 3.2.3), but also *(ii)* handling the duplicate packet with the idea of best-effort network and no-act delay Round-Trip Time (RTT) (KARN; PARTRIDGE, 1987). After setting up the rules in the switches, the controller inserts in the repository all the update network information collected at this stage.

3.2.2 Events Phase

This phase aims to identify events in the network infrastructure. At the moment, *(i) Dataflow Event* is the only event handled by our controller, but we may extend it to handle events such as *(ii) New Device Addition to the Topology* and *(iii) Dropped Communication Link. Dataflow Events* are generated by a new type of service in the network (*e.g.*, a video streaming), and are stored in a services dictionary.

Figure 3.5: Diagram of the Events Phase.



Source: by author (2015).

The *Events Phase* stays in a loop during the operation of the infrastructure (see Figure 3.5). When running a specific protocol, such as HTTP communication, the first few packets of the communication are initially treated by the standard rules and switches are instructed to duplicate the first packet of each new service flow (*e.g.*, between HostSrc [*source host*] and HostDst [*destination host*]), sending a copy to the controller. The controller stores in a list the protocols that are running in a link, which are identified by a function that reads the IP packet header and gets the information from the protocol field (*e.g.*, source and destination MAC addresses, IP addresses, and TCP/UDP port numbers). We use standard TCP/UDP port numbers to register a given service in the repository (*e.g.*, HTTP = [80,8080], SSH = [22], SMTP = [25,587]). This list is previously supplied by the network programmer, using RFC

1700 (REYNOLDS; POSTEL, 1994) recommendations. Then, the controller checks what are the necessary requirements for the proper functioning of such protocol through the information loaded from the repository and decides whether the controller should start the *Analysis Phase* (see Section 3.2.3). The *Analysis Phase* is initiated immediately if the protocol of the new flow is not yet included in the list of protocols for the link, or if some other flow is already sharing the same path but using a different protocol.

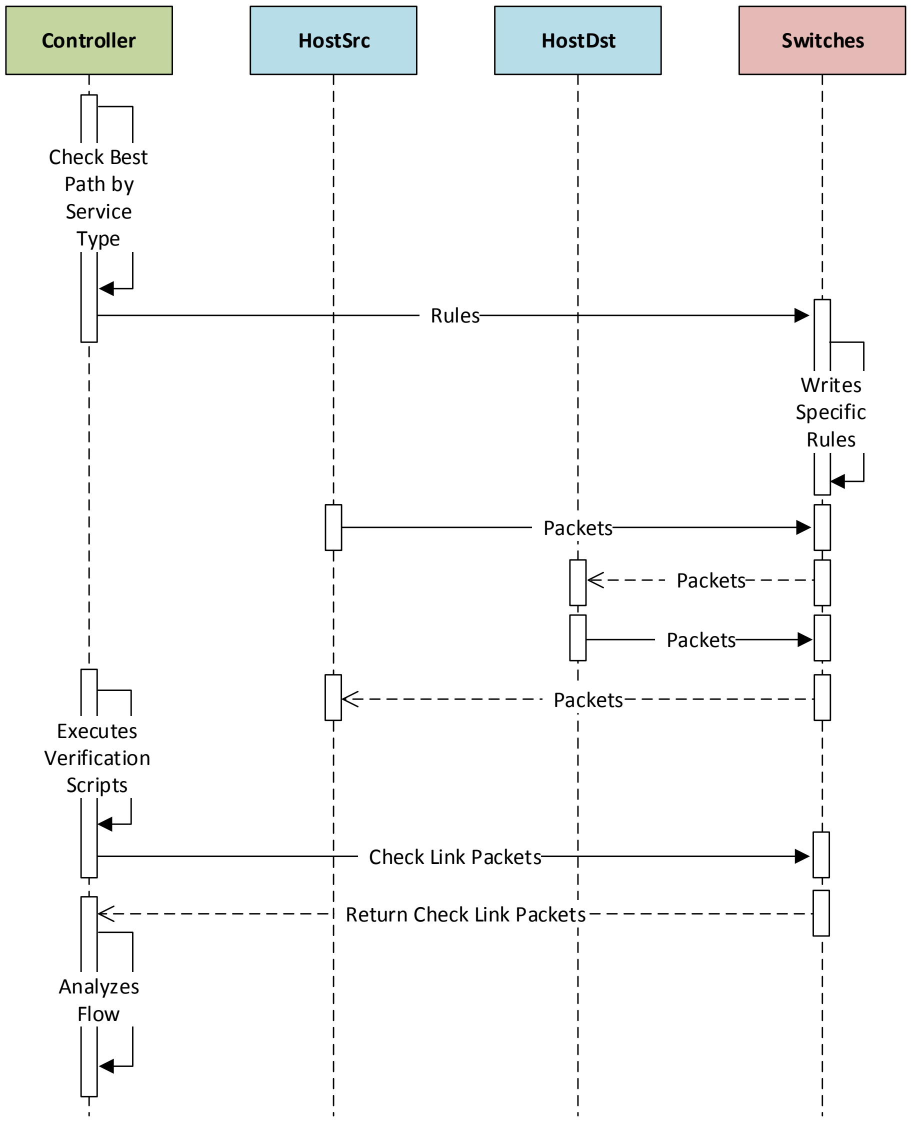
3.2.3 Analysis Phase

After identifying the service requirements, the controller calculates the new rules for best path, taking into account the specific parameters for that service (see Figure 3.6). The calculations are carried for the paths using as weights the bandwidth (BW), throughput (T), delay (D), jitter (J), loss rate (LR), and number of hops (NH) in each link of the physical topology (PARTRIDGE, 1992). *E.g.*, for VoIP we define the requirements for calculation of the best link as an amount of bandwidth that varies according to the encoding used, low delay and low jitter, and priority (P) (KEEPENCE, 1999). We check the value of each of these requirements using an analysis function that sends Internet Control Message Protocol (ICMP) (POSTEL, 1981) packets and stores the return value in a state vector of links. When calculating the jitter, we store a vector with the 30 last values and calculate the average. For the purposes of our experiments, the requirements of VoIP SLA goals were considered as $D \leq 200ms$, $BW \geq 128kbps$, $J \leq 20ms$, and $P = 99$. Each requirement/value pair is presented in order of importance. The link that satisfies the largest number of requirements with priority numbered from left to right is chosen as the best path.

At this moment, the best path should be converted into specific rules. A specific rule comprises a mask indicating that a packet containing the *Transport Destination Port* field will receive the actions *Modify Fields* and *Forward Packet to Ports*. The *Modify Fields* action changes the value of the ToS field of the IP packet header to the priority value defined by the QoS class of the service. Moreover, each rule has a priority in the flow table of each network element. Each priority value is set according to the priority value of each QoS class, *i.e.*, the QoS classes with higher priority will take preference over QoS classes with lower priority. There are two types of specific rules: The specific rules deployed in the switches having links with hosts and specific rules deployed in switches having links with other switches. The only difference is that in cases of links between switches there is no need to change the value of the ToS field, because this change is performed in the switch where the host is connected.

These new rules are reconfigured at runtime and only on switches that have the flow, aiming to reduce processing overhead where there is no need for such rules. Unlike the standard rules, these *specific rules* are configured with a timeout in the flow table of each element. In our experiments, we set the value of the timeout to *15s* (but it can be easily adjusted based on the analysis of the services being executed).

Figure 3.6: Diagram of the Analysis Phase.



Source: by author (2015).

Periodically, the controller checks if the the configured links remain the best choices for the current flow. In our experiments, we set the checking intervals to *10s* (but these can also be easily adjusted). If at any time the controller identifies that there is a better alternative path, new rules are sent to the switches in order to process the flow as efficiently as possible. If the current path remains the best, the controller only increases the value of the timeout for the rules on each switch for the corresponding flow. This phase stays in a loop during the operation of the infrastructure.

3.3 Policy Authoring Operation

The policy authoring framework handles the actions of the *top-down* stage (Figure 3.3). Regarding the Policy Authoring Operation, the operator inserts an SLA that defines explicitly or implicitly business-level goals. When inserting each policy, the *Policy Analyzer* component (Figure 3.1) uses *regexes* (regular expressions) – previously stored in the *Policy Repository* – to match the expressions written in controlled natural language, and suggest the more appropriate QoS class/classes to the SLA.

This Policy Authoring relies on abductive reasoning to suggest QoS classes. As mentioned previously, in abductive reasoning, starting from a *conclusion* and a known *rule*, it is possible to explain a particular *premise*. We use the following SLA to illustrate how logical reasoning works and, subsequently, we use the same SLA to explain how the Policy Authoring operates:

"HTTP services should receive bandwidth higher than 100kbps and delay lower than 300ms".

The *conclusion* of this SLA is *"HTTP services should receive"* certain characteristics. The *rules* for reaching this conclusion are *"bandwidth > 100kbps"* and *"delay < 300ms"*. Thus, we present the *premise* (QoS class in the repository) that has this rule and which can *possibly* arrive at this conclusion.

We define a query that assigns weights to results based on the importance of the *regexes* contained in the SLA. These expressions are compared to the information stored in the repository to sort the results and display them. The ordering thus follows:

1. expressions related to QoS classes;
2. expressions related to services; and
3. expressions related to service requirements.

3.3.1 Matching Process: A Step-by-step to Match Regexes

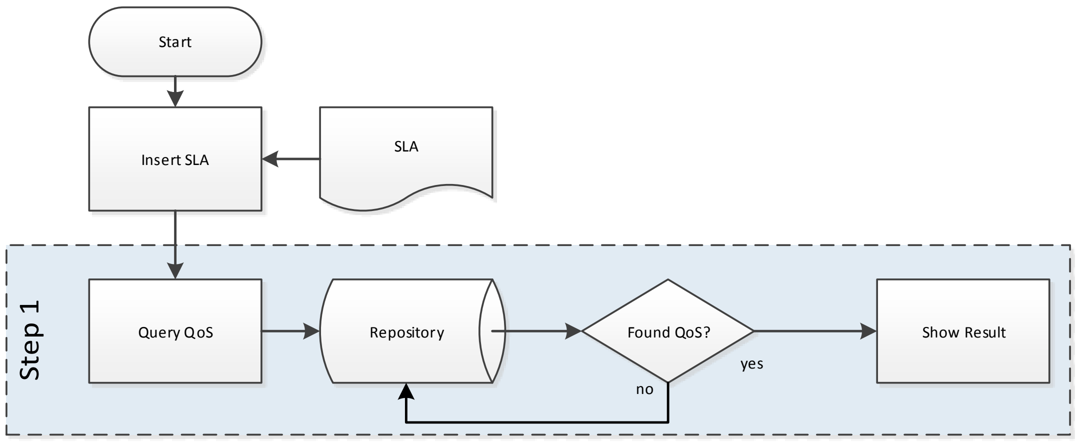
Figures 3.7, 3.8, and 3.9 show the flow diagrams to query regexes in an SLA and display possible classes to match. These steps are the following:

Step 1 – Check if there is any *qos-regexes* expression in the SLA indicating a class, *e.g.*, QoS Gold, Silver. If there are occurrences of these expressions, the Policy Authoring framework returns the QoS class values, based on the identified *qos-regexes*. For the SLA presented in the example, we have no expressions of this type.

Step 2 – Check if there is any *service-regexes* expression in the SLA relating to services, *e.g.*, FTP, VoIP. If there are occurrences of these expressions, the Policy Authoring framework returns the QoS class values to which the services are associated. For the SLA in the example, there is a *service-regexes* (HTTP), which may be associated with a QoS class in the repository.

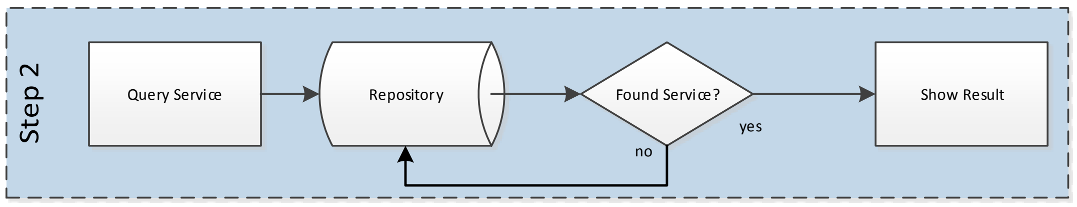
Step 3 – Analyze the expressions indicating service requirements, *e.g.*, priority, bandwidth.

Figure 3.7: Flow diagram to query qos-regexes in an SLA and display classes that match the SLA.



Source: by author (2015).

Figure 3.8: Flow diagram to query service-regexes in an SLA and display classes that match the SLA.



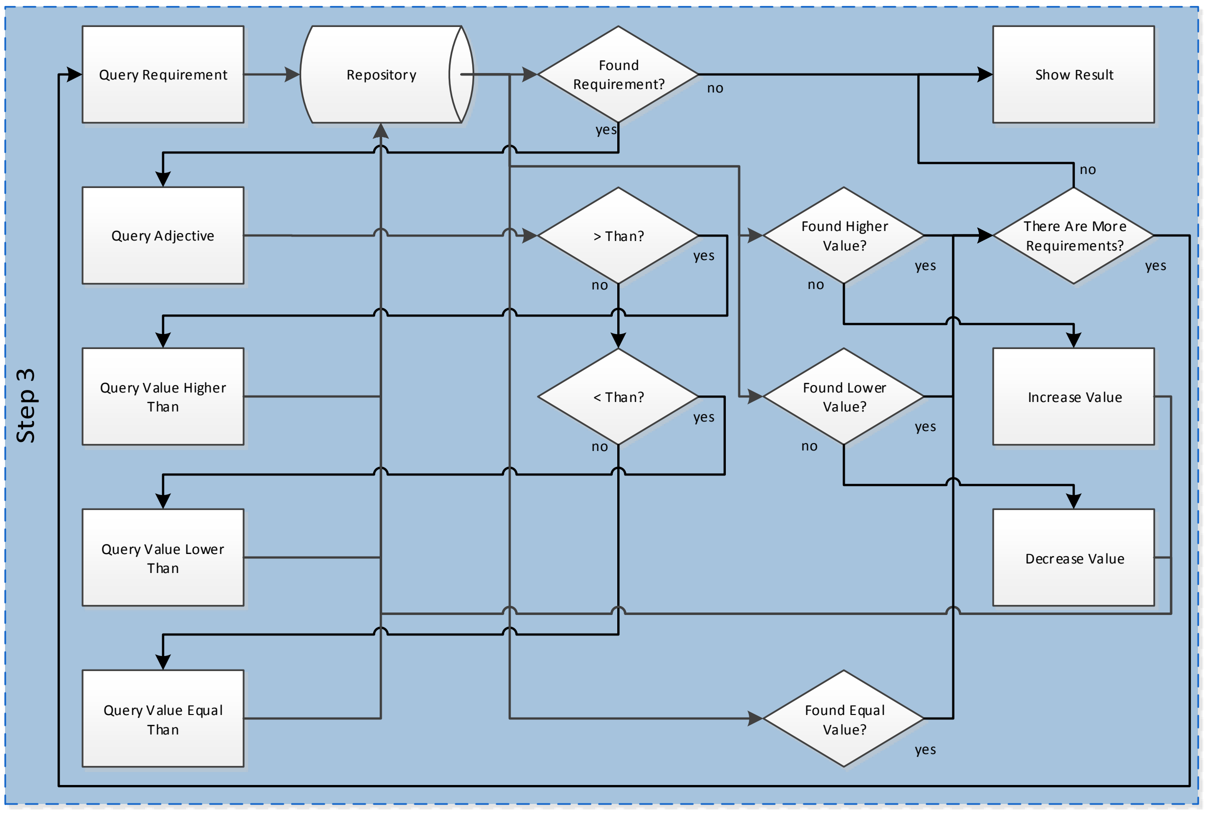
Source: by author (2015).

If there are occurrences of these expressions, the Policy Authoring framework performs the following operations: *(i)* identify and count the *requirements-regexes* found, and *(ii)* identify and count the *adjective-regexes* that come before and after any *requirements-regexes*.

We also developed a technique for identifying and counting the *requirements-regexes*, which allows the operator to optimally match the *adjective-regexes* found with their respective requirements. In the SLA above, we can observe the *adjective-regexes higher* and *lower*, which are related to the *requirements-regexes bandwidth* and *delay*, respectively. The *Policy Analyzer* identifies any *adjective-regexes* and examines the SLA, identifying the proximity of the *adjective-regexes* referring to *requirements-regexes*. This is performed by checking if *adjective-regexes* are located before or after *requirements-regexes*. At the end, the result is presented to the business-level operator.

The Policy Authoring framework uses abductive reasoning to show what are the best configurations for the SLA. Thus, the *Policy Analyzer* can identify, for example, that there is already a QoS class configured with low delay, or that the throughput for the specified network path already exceeds the network configuration, indicating that the policy should be reformulated. Also, the operator can be warned of potential conflicts or even non-compliance with policies.

Figure 3.9: Flow diagram to query requirements-regexes in an SLA and display classes that match the SLA.



Source: by author (2015).

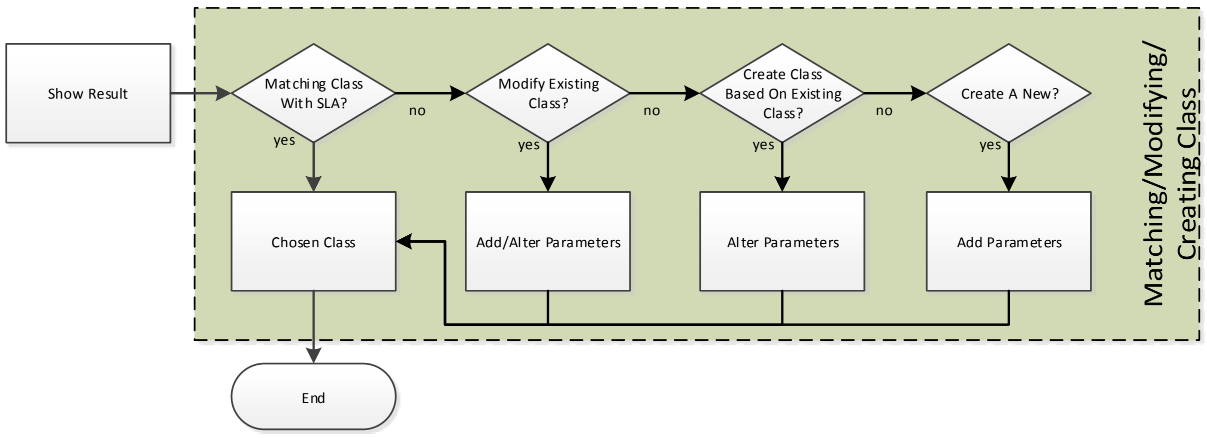
If the operator chooses one of the suggested QoS classes, the Policy Authoring framework will store the information extracted from the SLA, *e.g.*, the service, with the selected class.

3.3.2 Modifying and Creating QoS Class

Suggestions provided through abductive reasoning are not mandatory. If after analyzing them the operator decides they do not meet the high-level goals, the suggestions can be ignored. At this point, the operator can analyze the information presented by the Policy Authoring framework and rely on inductive reasoning to perform the following actions (Figure 3.10:

- **Modify existing policy/class –** This action allows the operator to change a predetermined parameter, *e.g.*, $priority = 100$ to $priority = 101$, or add a parameter that does not yet exist, *e.g.*, $delay = 120ms$. This modification may impact other policies, and the *Analyzer* uses inductive reasoning to identify the classes in the repository that may be impacted. Thus, the operator has the opportunity to analyze policy-by-policy and decide if the change is viable or not.
- **Create policy/class based on existing class –** This action is an alternative to modifying

Figure 3.10: Flow diagram to modification or creation QoS classes.



Source: by author (2015).

an existing class. Through this action, a new class created by the operator inherits the parameters of an existing class, which can be customized as needed. The *Policy Analyzer* uses inductive reasoning to automatically check if the parameter values of this new class are not identical to the ones in an existing class in the repository. If so, the existing class is returned instead.

- **Create a new policy/class –** The creation of new classes can be conducted *(i)* if a class that meets the objectives of the SLA does not exist, or *(ii)* if the parameters of other classes retrieved via abductive or inductive reasoning are not related to the objectives of the new SLA. Thus, the operator can set the new class parameter-by-parameter to meet the SLA objectives.

After any of the actions above is executed, the *Parser* component (Figure 3.1) will be executed and the policies/classes will be stored in the *Policy Repository*. It is based on this information that the Policy Authoring framework estimates the amount of allocated traffic per class and warns if the infrastructure can support or not new policies. Subsequently, the OpenFlow controller reads from the repository these new policies starting the *Analysis Phase* for setting up the appropriate rules in forwarding devices, as explained in Section 3.2.

4 AN EC-BASED FORMALISM FOR POLICY REFINEMENT

In this chapter we define a formal representation of high-level policies in the form of SLAs using Event Calculus (EC) and the use of logical reasoning to model both the system behavior and the policy refinement process in SDN. It is our aim with this formalism to assist infrastructure-level programmers to develop refinement tools and configuration approaches to achieve more robust SDN deployments independent of the network controller implementation or policy language.

Below, we present an overview of a formalism called Event Calculus and Logical Reasoning in Section 4.1. In Section 4.2 we extend Event Calculus for our policy refinement solution. We present our policy refinement model in Section 4.3. In Section 4.4 we demonstrate how our model represents the system elements and refines an SLA and how it decomposes the SLA into a set of low-level configurations. Finally, we present experiments and results obtained with an implementation using prolog in Section 4.4.4.

4.1 Event Calculus and Logical Reasoning

Event Calculus (KOWALSKI; SERGOT, 1986) is a formalism that permits representing and reasoning about dynamic systems (BANDARA; LUPU; RUSSO, 2003). It consists of axioms and predicates that are independent of application or domain. Table 4.1 presents the predicates of *Event Calculus*. We use the form described by Bandara *et al.* (BANDARA; LUPU; RUSSO, 2003; BANDARA et al., 2005), consisting of *(i)* a set of event types, *(ii)* a set of properties (called fluents) that can vary over the system lifetime, and *(iii)* a set of time points.

Table 4.1: Event Calculus Predicates

Predicates	**Description**
initiates(e,f,t)	event e 'initiates' fluent f for all time $> t$
terminates(e,f,t)	event e 'terminates' fluent f for all time $> t$
releases(e,f,t)	event e 'releases' fluent f at time t
$initially_p(f)$	fluent f is initially true
$initially_n(f)$	fluent f is initially false
$happens(e,t_1,t_2)$	event e 'happens' at time t_1 and 'terminates' at time t_2
holdsAt(f,t)	fluent f 'holds' at time t
$clipped(t_1,f,t_2)$	fluent f 'clipped' between times t_1 and t_2
declipped(*t1*,*f*,*t2*)	fluent f 'declipped' between times t_1 and t_2

Source: by author (2015).

Event Calculus can be used for interpreting perceptions and predicting actions and their consequences in order to perform a possible solution plan. It is worth noting that *Event Calculus* uses a representation of time, which is independent of any other events that may occur in the same system. We can then use this formalism for reasoning about events that have a certain duration and not only about instantaneous events. Thus, it is possible to model events occurring within intervals, instead of specifying a specific time for their occurrence.

We apply the concept of *logical reasoning* (SHANAHAN, 2000) to support the operator in the refinement of an SLA. Logical reasoning has three modes: deductive, inductive and abductive. In *deductive reasoning*, a conclusion is reached by using a rule that analyzes a premise. For example, if streaming packets are transmitted the network becomes slower; streaming packets are being transmitted now; therefore, the network is slower. In *inductive reasoning*, the goal is to identify a rule, starting from a historical set of conclusions generated from a premise. For example, every time streaming packets are transmitted the network becomes slower; so, if streaming packets are transmitted tomorrow, the network will be slower. Finally, in *abductive reasoning*, starting from a conclusion and a known rule, we can explain a premise. For example, when streaming packets are transmitted the network becomes slower; the network is slower now; so, possibly streaming packets are being transmitted.

We use two modes of logical reasoning: abduction to assist the Policy Authoring Framework operator in identifying the QoS classes that can support the specifications of the SLA; and induction to assist the operator in specifying the system parameters that classes should have to meet the SLA.

4.2 Extend Event Calculus

A number of variations of the standard Event Calculus (described in Section 4.1) have been presented in the literature. In particular, we build on the one presented by Bandara *et al.* (BANDARA; LUPU; RUSSO, 2003; BANDARA et al., 2005). In order to achieve our goals we customized it with new constants, variables, operations/functions, and predicates as follow:

- **Constants –** these can be defined as SLAs (SLA), services (Serv), classes (Class), parameters/requirements (Par), or objects (Obj_n). Obj may represent a set of objects of the system where *n* represents a source object (Obj_{Src}), a destination object (Obj_{Dst}), an object of link – *i.e.*, the connection between an Obj_{Src} and Obj_{Dst} – (Obj_{Link}), or even a route (Obj_{Route}).
- **Variables –** define V_o to represent the attributes of objects and V_ρ to represent the parameters for the operations supported by objects.
- **Predicates –** specify what the object represent in the system, what is declared about it or relationships between objects. Table 4.2 presents the predicates.
- **Operations –** specify actions used with predicates. For example, a query in a repository

or the triggering of a phase. Table 4.3 presents the operations.

Table 4.2: New Predicates for Event Calculus.

Predicates	**Description**
object(SLA/Serv/Class/Par/Obj_n)	Used to specify that Obj is an object in the system. Objects can be network elements such as routers, switches, controllers, links, or routes.
isElement(Obj_n)	Holds if Obj_n represents a network element, *e.g.*, switch, controller.
method(Obj_n, Action(V$_\rho$))	Defines an action supported by an object.
isLink(Obj_{Link}, Obj_{Src}, Obj_{Dst})	Holds if Obj_{Link} represents a link between an Obj_{Src} and Obj_{Dst}.
isRoute(Obj_{Route})	Holds if Obj_{Route} represents a route.
isMemberRoute(Obj_{Route}, Obj_{Link})	Holds if the object, Obj_{Link}, is a member of the route, Obj_{Route}.
isRouterParameter(Obj_{Route}, Par, V$_o$)	Holds if the object, Par, is a parameter of the route.
isSLA(SLA)	Holds if SLA represents an SLA.
isDescriptionSLA(Serv/Class/Par/Obj_n,SLA)	Defines if Serv/Class/Par/Obj_n is an object contained in the SLA description.
isService(Serv)	Holds if Serv represents a service, *e.g.*, HTTP, VoIP.
isClass(Class)	Holds if Class represents a QoS Class, *e.g.*, gold, silver.
isPar(Par)	Holds if Par represents a parameter/requirement, *e.g.*, delay, priority.
isMemberClass(Class, Serv)	Holds if the object, Serv, is a member of the QoS Class, Class.
isMemberParameter(Class/Serv/SLA, Par, V$_o$)	Holds if the object, Par, is a parameter of a QoS Class, Service or SLA.
newMemberParameter(Class/Serv/SLA, Par, V$_o$)	Holds if the object, Par, is a changing or addition of parameter value of a QoS Class, Service or SLA.
attr(SLA/Serv/Class/Par/Obj_n, V$_o$)	Defines that V$_o$ is an attribute of a Serv, Class, Par, or Obj_n.

Source: by author (2015).

Table 4.3: Operations used with predicates.

Operations	**Description**
state(Obj_n, V$_o$, Value)	Indicates the state of an object in the system.
operation(Obj_n, Action(V$_\rho$))	Indicates the operations and functions specified in a policy or event.
systemEvent(Event)	Indicates any event in the system. It is used to trigger the operations and functions.
doAction(Obj_{Src}, operation(Obj_{Dst}, Action(V$_\rho$)))	Indicates the action performed by Obj_{Src} in Obj_{Dst}.

Source: by author (2015).

4.3 Policy Refinement Model

In our solution we first describe the SLAs in EC, and use logical reasoning to derive the SLOs and QoS classes based on a system model description. We also model the state of routes and links maintained by the controller, and use logical reasoning to match the best route based on the requirements of the SLA. *Requirements-regexes* are used to search for matching QoS classes (*qos-regexes*) that were previously registered in the repository. We use abductive reasoning to identify possible configurations that achieve the goals specified by the SLA. Thus we can maximize the number of inferences between *requirements-regexes* – *i.e.* SLOs – found in the SLA and the parameters of the QoS classes registered in the repository. The technique proposed performs the matching in the following order:

```
$\lambda_1$ : isMemberParameter(SLA$_n$,P,V$_o$) $\wedge$ isMemberParameter(SLA$_n$,B,V$_o$)
    $\wedge$ isMemberParameter(SLA$_n$,D,V$_o$) $\wedge$ isMemberParameter(SLA$_n$,J,V$_o$)

$\lambda_2$ : isMemberParameter(Class$_n$,P,V$_o$) $\wedge$ isMemberParameter(Class$_n$,B,V$_o$)
    $\wedge$ isMemberParameter(Class$_n$,D,V$_o$) $\wedge$ isMemberParameter(Class$_n$,J,V$_o$)

$\lambda_3$ : isMemberParameter(Class$_n$,P,V$_o$) $\wedge$ isMemberParameter(Class$_n$,B,V$_o$)
    $\wedge$ isMemberParameter(Class$_n$,D,V$_o$)

$\lambda_4$ : isMemberParameter(Class$_n$,B,V$_o$) $\wedge$ isMemberParameter(Class$_n$,P,V$_o$)

$\lambda_5$ : isMemberParameter(Class$_n$,P,V$_o$) $\wedge$ isMemberParameter(Class$_n$,D,V$_o$)

$\lambda_6$ : isMemberParameter(Class$_n$,B,V$_o$) $\wedge$ isMemberParameter(Class$_n$,D,V$_o$)

$\lambda_7$ : isMemberParameter(Class$_n$,P,V$_o$) $\vee$ isMemberParameter(Class$_n$,B,V$_o$)
    $\vee$ isMemberParameter(Class$_n$,D,V$_o$) $\vee$ isMemberParameter(Class$_n$,J,V$_o$)

$\phi_1$ : $\lambda_1$ $\leftrightarrow$ $\lambda_2$
$\phi_2$ : ($\lambda_1$ $\leftrightarrow$ $\lambda_3$) $\leftarrow$ $\neg$ $\phi_1$
```

```
 $\phi_3$ : ($\lambda_1$ $\leftrightarrow$ $\lambda_4$) $\leftarrow$ $\neg$ $\phi_2$
 $\phi_4$ : ($\lambda_1$ $\leftrightarrow$ $\lambda_5$) $\leftarrow$ $\neg$ $\phi_3$
 $\phi_5$ : ($\lambda_1$ $\leftrightarrow$ $\lambda_6$) $\leftarrow$ $\neg$ $\phi_4$
 $\phi_6$ : ($\lambda_1$ $\leftrightarrow$ $\lambda_7$) $\leftarrow$ $\neg$ $\phi_5$

$SLA_n$ is an SLA
$Class_n$ is a QoS Class
$V_o$ is a value of a parameter
$\lambda_n$ is a predicate
```

The set of λ rules are used to retrieve the QoS classes that satisfy the largest amount of requirements. Thus, λ_2 will retrieve QoS classes that satisfy all requirements (B, P, D, J), whereas λ_7 will retrieve QoS classes that satisfy at least one of the requirements. The set of ϕ rules is an order of matches that happens until an occurrence of ϕ matches the desired result. Thus, ϕ_1 is an ideal match where all requirements are satisfied while ϕ_6 is a match where at least one requirement is satisfied.

Ultimately, the result presented to the business-level operator is a set of QoS classes ordered by the highest amount of requirements found which can better meet the SLA. In the worst case, the refinement process will propose QoS class(es) which contain at least one of the parameters. From these results, the business-level operator can decide between using the suggested class or to use inductive reasoning to perform:

(i) modification of existing class by changing parameters:

```
 isClass(Class$_n$)
 $\wedge$ newMemberParameter(Class$_n$,Par$_n$,V$_{n'}$)
 $\rightarrow$ isMemberParameter(Class$_n$,Par$_n$,V$_n$)

isClass(Class$_n$) is the QoS class.
newMemberParameter(Class$_n$,Par$_n$,V$_{n'}$) is the changing or addition of parameters.
isMemberParameter(Class$_n$,Par$_n$,V$_n$) is the new parameter value of isClass(Class$_n$).
```

This action indicates that the value V_n of the parameter *Par*$_n$ of the QoS class *Class*$_n$ modifies the QoS class *Class*$_n$ generating a QoS Class *Class*$_{n'}$. This action allows the business-level operator to change a predetermined parameter, *e.g.*, delay $=$ 10ms to delay $=$ 20ms, or to add a parameter that does not yet exist in the SLA, *e.g.*, priority $\leq$ 999. However, this modification may impact other policies. To address that, our policy refinement toolkit uses inductive reasoning to identify the classes in the repository that may be impacted. Thus, the operator has the opportunity to analyze policy-by-policy or parameter-by-parameter and decide whether the change should be applied or not.

(ii) creation of a new QoS class based on an existing one:

```
 isClass(Class$_n$)
 $\wedge$ newMemberParameter(Class$_n$,Par$_{n'}$,V$_{n'}$))
 $\rightarrow$ isClass(Class$_{n'}$)

isClass(Class$_n$) is the existing QoS class.
newMemberParameter(Class$_n$,Par$_{n'}$,V$_{n'}$) is the changing or addition of parameters.
isClass(Class$_{n'}$) is the new QoS class based on Class$_n$.
```

This action is an alternative to modifying an existing class. Through this action, a new class created by the business-level operator inherits the parameters of an existing class, which can be customized as needed. The toolkit uses inductive reasoning to automatically check if the parameter values of this new class are not identical to the ones in an existing class in the repository. If so, the existing class is returned instead preventing QoS classes duplication.

(iii) creation of a new QoS class specifying new requirements and values:

```
  → isClass(Class$_n$)

isClass(Class$_n$) is the new QoS class.
```

Ultimately, the creation of new classes can be conducted *(i)* if a class that fulfills the requirements of the SLA does not exist, or *(ii)* if the parameters of other classes retrieved via abductive or inductive reasoning are not related to the requirements of the new SLA. Thus, the business-level operator can set the new class parameter-by-parameter to fulfill the SLA requirements.

As mentioned previously, the controller, at startup, collects information about the network infrastructure to calculate all possible routes (Route) between two elements. The calculations are carried for the paths using as weights the bandwidth (B), delay (D), jitter (J), and number of hops (NH) in each link of the network infrastructure. Thus, the representation of this operation performed by the controller is:

```
  $\lambda_1$ : isRouterParameter(Route$_n$,NH,V$_o$) ∧ isRouterParameter(Route$_n$,B,V$_o$)
     ∧ isRouterParameter(Route$_n$,D,V$_o$) ∧ isRouterParameter(Route$_n$,J,V$_o$)

  $\lambda_2$ : isRouterParameter(Route$_n$,B,V$_o$) ∧ isRouterParameter(Route$_n$,D,V$_o$)
     ∧ isRouterParameter(Route$_n$,J,V$_o$)

  $\lambda_3$ : isRouterParameter(Route$_n$,NH,V$_o$) ∨ isRouterParameter(Route$_n$,B,V$_o$)
     ∨ isRouterParameter(Route$_n$,D,V$_o$) ∨ isRouterParameter(Route$_n$,J,V$_o$)

  Route$_n$ ← $\lambda_1$
  Route$_{n+1}$ ← $\lambda_2$
  Route$_{n+2}$ ← $\lambda_3$

Route$_n$ is a route
```

As can be observed, λ_1 is a rule that chooses the route that faithfully fulfills all the parameters of the SLA. λ_2 is an alternative route that does not consider the number of hops. We exclude the number of hops since many services do not recognize this as a main requirement for their proper functioning. Ultimately, λ_3 is a rule that selects all routes that satisfy at least one of the parameters. Route$_n$, Route$_{n+1}$, and Route$_{n+2}$ are possible routes sorted by the requirements presented in each λ_n. These routes are selected at runtime whenever competing SLAs are detected. This process is performed in order to establish a load balancing or to create a best route to satisfy the requirements of each SLA.

Thus, after any choice, our refinement model will match the indications performed by the business-level operator with the conditions of the network infrastructure as follows:

```
  $\lambda_1$ : Class$_n$ ↔ Route$_n$
  $\lambda_2$ : Class$_n$ ← Route$_{n+1}$ ← ¬ $\lambda_1$
  $\lambda_3$ : Class$_n$ ← Route$_{n+2}$ ← ¬ $\lambda_2$
```

We have configured the SLA requirements and grouped them into classes, because it is an easier way to deal with the increase in the number of policies that have common goals. However, if at any time it is discovered that an SLA cannot be fulfilled or if some SLO cannot be achieved, the system is flexible enough to allow the necessary adjustments.

4.4 Modeling Case-studies

In this section we demonstrate and discuss the formalism through case studies and an implementation using Prolog. Prolog is a declarative programming language combined with artificial intelligence and first-order logic. Prolog is composed of facts (data) and a set of rules, *i.e.*, relationship between data. The general idea behind Prolog is to describe a situation. Briefly, an interpreter is used to evaluate a logical formula and generate by inference a set of results deduced from the database. Thus, it will indicate whether a Prolog question is true or false and, if variables are present in the formula, it will also indicate what should be the values of the variables. We use Prolog in the sections below to present the formal representation of the network infrastructure and their elements, QoS classes, services, and SLA.

The formal model of the infrastructure, services, QoS classes, and SLAs is presented through simple examples. However, it is used as a basis for more sophisticated inferences that occur during the SLA refinement process such as the creation of rules to match SLA requirements with QoS Classes. This will be shown in Section 4.4.3 and in Section 4.4.4.

Below, in Section 4.4.1 we present the formal representation of the network infrastructure. We present the formal representation of the services, QoS classes, and requirements in Section 4.4.2. In Section 4.4.3 we demonstrate and discuss the refinement of an SLA. We present and discuss the experimental evaluation of our formalism in Section 4.4.4.

4.4.1 Network Infrastructure Formal Representation

In order to formally represent the network infrastructure, we assume the scenario shown in Figure 4.1. There are 11 switches (SWn) and 22 links (Ln).

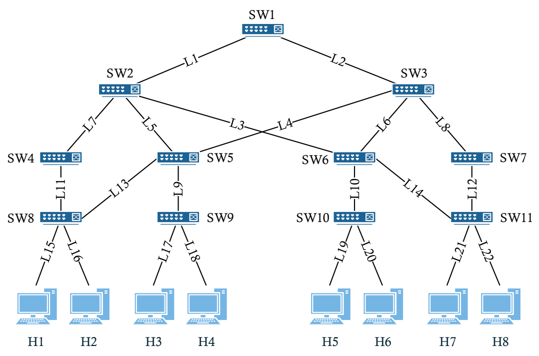
In this scenario, we express the links and routes as elements of the network infrastructure. Figure 4.2 shows alternative routes between *H1* and *H2*. The need to express links becomes relevant when there are alternative routes between the elements. Thus, the system gains dynamism to decide routes between Obj_{Src} and Obj_{Dst}.

As mentioned previously, we modeled our formal representation in Prolog. In Prolog, we represent all switches and all their links in the network infrastructure as follows:

```
isElement(sw1).
isElement(sw2).
      ...
isElement(sw11).
```

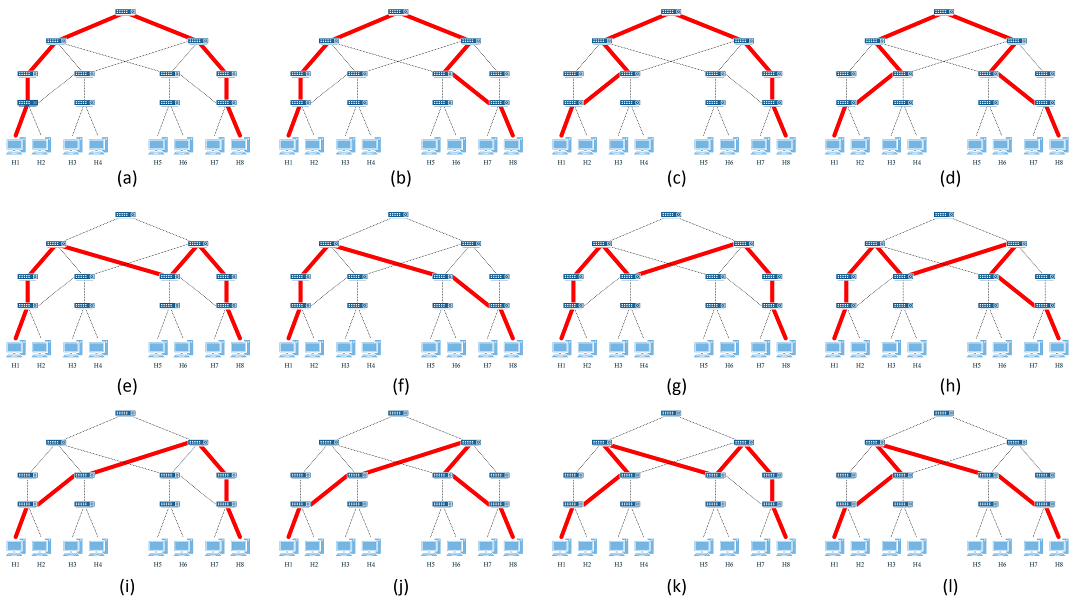
```
isLink(l1,sw2,sw1).
isLink(l2,sw1,sw3).
isLink(l3,sw2,sw6).
isLink(l4,sw3,sw5).
isLink(l5,sw2,sw5).
isLink(l6,sw3,sw6).
isLink(l7,sw4,sw2).
isLink(l8,sw3,sw7).
isLink(l9,sw5,sw9).
isLink(l10,sw6,sw10).
isLink(l11,sw8,sw4).
isLink(l12,sw7,sw11).
isLink(l13,sw5,sw8).
isLink(l14,sw6,sw11).
```

Figure 4.1: Example of network infrastructure.



Source: by author (2015).

Figure 4.2: Example of alternative routes for Figure 4.1.



Source: by author (2015).

Based on formal representation above, if we want to list all switches in the network and all their links we can execute the following command in Prolog and obtaining the following result:

```
?- isElement(Switch); isLink(Link,SwitchA,SwitchB).
```

```
Switch = sw1 ; Switch = sw2 ; Switch = sw3 ; Switch = sw4 ; Switch = sw5 ; Switch = sw6
; Switch = sw7 ; Switch = sw8 ; Switch = sw9 ; Switch = sw10 ; Switch = sw11 ; Link = L1,
SwitchA = sw1, SwitchB = sw2 ; Link = L2, SwitchA = sw1, SwitchB = sw3 ; Link = L3, SwitchA
= sw2, SwitchB = sw6 ; Link = L4, SwitchA = sw3, SwitchB = sw5 ; Link = L5, SwitchA = sw2,
SwitchB = sw5 ; Link = L6, SwitchA = sw3, SwitchB = sw6 ; Link = L7, SwitchA = sw2, SwitchB
= sw4 ; Link = L8, SwitchA = sw3, SwitchB = sw7 ; Link = L9, SwitchA = sw5, SwitchB = sw9 ;
Link = L10, SwitchA = sw6, SwitchB = sw10 ; Link = L11, SwitchA = sw4, SwitchB = sw8 ; Link
= L12, SwitchA = sw7, SwitchB = sw11 ; Link = L13, SwitchA = sw5, SwitchB = sw8 ; Link =
L14, SwitchA = sw6, SwitchB = sw11.
```

If we want to list all links and switch connected to *sw5* we can execute the following command in Prolog and obtaining the following result:

```
?- isLink(Link,Switch,sw5).
```

```
Link = L5, Switch = sw2; Link = L13, Switch = sw8; Link = L4, Switch = sw3; Link = L9,
Switch = sw9;
```

Subsequently, we present the representation of a possible route between *sw8* and *sw11*. The route shown in Figure 4.2(a) can be specified as follows:

```
isLink(l11,sw8,sw4).
isLink(l7,sw4,sw2).
isLink(l1,sw2,sw1).
isLink(l2,sw1,sw3).
isLink(l8,sw3,sw7).
isLink(l12,sw7,sw11).

isRoute(r1).
isMemberRoute(r1,l11).
isMemberRoute(r1,l7).
isMemberRoute(r1,l1).
isMemberRoute(r1,l2).
isMemberRoute(r1,l8).
isMemberRoute(r1,l12).
```

If we want to list all links of *r1* we can execute the following command in Prolog and obtaining the following result:

```
?- isMemberRoute(r2,Link).
```

```
Link = l11 ; Link = l7 ; Link = l1 ; Link = l2 ; Link = l8 ; Link = l12 .
```

Despite being very simple, the isMemberRoute(Route, Link) predicate can be used to discover the number of hops of a specific route. This information will be important for deciding which routes should be chosen to fulfill the requirements of a particular service, as described next.

4.4.2 Services, QoS Classes, and Parameters Formal Representation

We present in this section the formal representation of a service and its QoS Class. We set five classes with different QoS requirements. This configuration aims to show the variation of requirements for each QoS class. The pre-configured classes are shown in Table 4.4.

Table 4.4: QoS Classes

QoS Class	Priority	Bandwidth	Delay	Jitter
Bronze	100	128	500	50
Silver	300	256	400	40
Gold	500	512	300	30
Platinum	700	1024	200	20
Diamond	900	2048	100	10

Source: by author (2015).

If we want to find every priority value for every registered class, we can perform the following command in Prolog and obtaining the following result:

```
?- isParametersClass(Class,priority,Value).
```

```
Class = diamond, Value = 900 ; Class = platinum, Value = 700 ; Class = gold, Value = 500 ;
Class = silver, Value = 300 ; Class = bronze, Value = 100.
```

As mentioned previously, this formal representation, although simple, is used as a basis for more sophisticated inferences that occur during the refinement process of the SLAs. In the following sections, we will discuss how these representations can be combined into more elaborate queries such as discovering the highest value of a parameter.

4.4.3 SLA Refinement

In this section we demonstrate how our formal model can be used to refine an SLA into a set of low-level configurations. It is important to emphasize that we applied the concept of logical reasoning to support the operator in the refinement of an SLA. We use two modes of logical reasoning: abduction to assist the operator in identifying the QoS classes that can support the specifications of the SLA; and induction to assist the operator in specifying the system parameters in the configuration of the QoS classes that can meet an SLA. We use the following SLA as a case-study:

"HTTP traffic should receive lowest priority and lowest bandwidth".

The formalism to represent the HTTP SLA is presented as follows:

```
object(sla).
object(http).

isSLA(sla).
isService(http).

attr(sla,status).
attr(http,status).

state(sla,status,enabled).
state(http,status,authorized).
isMemberParameter(http,priority,lowest).
isMemberParameter(http,bandwidth,lowest).

isDescriptionSLA(sla,http).
isDescriptionSLA(sla,priority).
isDescriptionSLA(sla,bandwidth).

isPar(priority).
isPar(bandwidth).
```

We represent our refinement model using Event Calculus notation and its standard predicates as declared in Table 4.2. For a better understanding of the formal representation of the refinement process, we use friendly names to indicate to the reader where each process occurs in our solution. For example, Policy Analyzer is used to indicate the module that performs the regex analyzing operation, and PolicyAuthoringFramework is used to indicate where this operation occurs. Also, we use lambda (λ_n) to indicate a set of predicates or operations used with predicates. As mentioned previously, when an operator inserts an SLA, our refinement model uses *regexes* (see Table 3.1) – previously stored in the *Policy Repository* – to match the expressions written in semi-structured natural language, and suggests the more appropriate QoS class/classes to the SLA. As a result, suggestions will be displayed to the operator only after all possible matches.

Regarding the refinement process, on the one hand, a *top-down* stage initializes a policy analyzer process aiming to search *qos-regexes* explicitly written in the SLA. In this case, as there is no *qos-regexes* in the SLA, no compatible class will be returned by the query.

```
λn : initiates(doAction(PolicyAuthoringFramework,
       operation(PolicyAuthoringFramework,
       policyAnalyzer(qos-regexes))),state(http,status,enabled),T).

λn+1 ← (happens(doAction(PolicyAuthoringFramework,
       operation(Repository,request(qos-regexes))),T+1)
      ← λn).

λn is a predicate
```

Following, we search *service-regexes* in the SLA.

```
λn+2 ← (holdsAt(operation(PolicyAuthoringFramework,
       policyAnalyzer(service-regexes)),T+2)
      ← ¬ λn+1).
```

At this time, *service-regexes* “HTTP” is found. From this occurrence, we perform a query in the repository to find if exist a QoS class associated with the HTTP service.

```
$\lambda_{n+3}$ ← (happens(doAction(PolicyAuthoringFramework,
      operation(Repository,request(service-regexes(http)))),T+3)
    ← $\lambda_{n+2}$).

$\lambda_{n+4}$ ← (happens(doAction(Repository,
      operation(PolicyAuthoringFramework,return(Class))),T+4)
    ← $\lambda_{n+3}$).
```

In this case, HTTP service is not associated with any QoS class. Thus, a search for *requirements-regexes* and their *adjectives-regexes* is performed.

```
$\lambda_{n+5}$ ← ((holdsAt(operation(PolicyAuthoringFramework,
      policyAnalyzer(requirements-regexes)),T+5)
    ∧ holdsAt(operation(PolicyAuthoringFramework,
      policyAnalyzer(adjectives-regexes)),T+5))
    ← $\lambda_{n+4}$).
```

In the given SLA, we found the *requirements-regexes* “priority (P)” and “bandwidth (B)” and the occurrences of *adjective-regexes* “lowest” and “lowest”. As our refinement model associates the number of occurrences of *adjectives-regexes* with the number of occurrences of *requirements-regexes*, the toolkit needs to check what are the correct *adjectives-regexes* for the *requirements-regexes* found. We used a factor of proximity to analyze in which position of the SLA each *adjective-regexes* is located relative to the position of the *requirements-regexes*. At this point, we need to associate the *adjectives-regexes* “lowest” with something in the SLA (*e.g.*, another service). As we have no explicit statement in the SLA indicating that priority and bandwidth in HTTP service must be lower than something, we assume that this service should receive the lowest priority and lowest bandwidth registered in the repository.

```
$\lambda_{n+6}$ ← (((happens(doAction(PolicyAuthoringFramework,
      operation(Repository,request(requirements-regexes(priority)))),T+6)
    ∧ happens(doAction(PolicyAuthoringFramework,
      operation(Repository,request(adjectives-regexes(low)))),T+6))
    ∧ (happens(doAction(PolicyAuthoringFramework,
      operation(Repository,request(requirements-regexes(bandwidth)))),T+6)
    ∧ happens(doAction(PolicyAuthoringFramework,
      operation(Repository,request(adjectives-regexes(low)))),T+6)))
    ← $\lambda_{n+5}$).
```

Ultimately, our toolkit applies abductive reasoning to build a query based on regexes found in the SLA. This query will be executed and will return to the operator the QoS class that best matches the SLA requirements, in this case, Bronze QoS Class as a top choice. Abductive reasoning reaches this conclusion, because if the search is for “lowest priority” and “lowest bandwidth”, Bronze QoS Class is the class which has lowest priority and lowest bandwidth among the registered QoS Classes (Table 4.4). After the choice performed by the operator, our toolkit will register in the repository the SLA associating it with Bronze QoS Class.

```
$\lambda_{n+7}$ ← (happens(doAction(PolicyAuthoringFramework,
      operation(Repository,registerNew(sla(http)))),T+7)
    ∧ happens(doAction(PolicyAuthoringFramework,
      operation(Repository,associateNewSLA(isMemberClass(bronze,http))))),T+7)
      ← $\lambda_{n+6}$).

terminates($\lambda_{n+7}$, state(SLA,status,associated),T+7).
```

On the other hand, a *bottom-up* stage performs the controller phases. When starting the controller, it monitors the network in order to discover the network elements and their links (StartUp phase). Forwarding devices are instructed to send Link Layer Discovery Protocol (LLDP) packets to report their location in the topology. The controller collects these packets and performs a routine for calculating all possible links between all elements. For each result of this calculation a standard rule is created. As a result, the controller comes to know the position of all network elements and what is the cost (per link) to reach them. This information about links between forwarding elements and their standard rules is stored in the repository.

```
$\lambda_m$ :  initiates(doAction(Controller,
        operation(Controller,startupPhase(packets)),state(Controller,status,on),T)).

$\lambda_{m+1}$ ← (happens(operation(Controller,discoveryTopology(LLDP)),T+1)
      ∧ happens(operation(Controller,discoveryLinks(LLDP)),T+1))
      ← happens(doAction(Switch,operation(Controller,sendPacket(LLDP)),T+1))
      ← $\lambda_m$

$\lambda_{m+2}$ ← (happens(doAction(Controller,
        operation(Repository,registerSwitchId(idSwitch)),T+2))
      ∧ happens(doAction(Controller,
        operation(Repository,registerSwitchLink(linkSwitch)),T+2)),
      ∧ happens(doAction(Controller,
        operation(Repository,registerRule(standardRule)),T+2))
      ← $\lambda_{m+1}$.

$\lambda_m$ is a predicate
```

Subsequently, the controller reads the QoS Classes - which have been previously registered through the policy refinement process – from a repository that contains descriptions and requirements of all services that are initially scheduled to run in the network and the standard rules to address best efforts. The controller compares each QoS Class with the links previously analyzed (when the controller was initialized) and creates a spanning tree with the links that has the best possibility to fulfill the needs of the QoS Class. Each spanning tree is created by specific rules that will be set up in forwarding devices. These rules are composed by the flow priority (that identifies the order in which the packets should be processed), TCP/UDP destination port (that identifies to which service the packet is addressed), output port (that indicates to which port the switch will send the packet). Also, the controller uses the standard rule to create a spanning tree based on the best links between any two elements. This spanning tree aims to set up best-effort routes to initially address any service that appears on the network without causing transmission delay in the first packets while specific rules for each new service (not provided in an SLA) have not been established yet. Finally, the controller installs each rule in the flow tables of the switches.

$\lambda_{m+3.1}$ ← `happens(doAction(Controller,`
`operation(Repository,request(QoSClass)),T+3))`
← λ_{m+2}.

$\lambda_{m+3.2}$ ← `happens(doAction(Controller,`
`operation(Repository,request(standardRule)),T+3))`
← λ_{m+2}.

$\lambda_{m+4.1}$ ← `happens(doAction(Repository,`
`operation(Controller,return(bronze)),T+4))`
← $\lambda_{m+3.1}$.

$\lambda_{m+4.2}$ ← `happens(doAction(Repository,`
`operation(Controller,return(standardRule)),T+4))`
← $\lambda_{m+3.2}$.

$\lambda_{m+5.1}$ ← `holdsAt(operation(Controller,`
`calculateSpanningTreeClass(bronze)),T+5)`
← $\lambda_{m+4.1}$.

$\lambda_{m+5.2}$ ← `holdsAt(operation(Controller,`
`calculateSpanningTreeStandard(standardRule)),T+5)`
← $\lambda_{m+4.2}$.

$\lambda_{m+6.1}$ ← `happens(doAction(Controller,`
`operation(Switch,registerRule(specificRule)),T+6))`
← $\lambda_{m+5.1}$.

$\lambda_{m+6.2}$ ← `happens(doAction(Controller,`
`operation(Switch,registerRule(standardRule)),T+6))`
← $\lambda_{m+5.2}$.

$\lambda_{m+7.1}$ ← `holdsAt(operation(Switch,writeRule(specificRule)),T+7)`
← $\lambda_{m+6.1}$.

$\lambda_{m+7.2}$ ← `holdsAt(operation(Switch,writeRule(standardRule)),T+7)`
← $\lambda_{m+6.2}$.

Subsequently, the controller enters a phase that stays in a loop awaiting the occurrence of events during the operation of the infrastructure. When running a specific service, such as HTTP, the source host (hostSrc) sends packets to the switch. If there is a specific rule for that type of service, the switch forwards the packet to destination host (hostDst). If there is no specific rule, the switch forwards the packet to hostDst and a copy of the packet to the controller. Subsequently, the controller performs a similar process, as outlined above, checking out from the repository if there is any QoS class establishing specific rules for this new service.

λ_{m+8} ← `initiates(doAction(Controller,`
`operation(Controller,eventsPhase(packet)),state(packet,status,on),T+8)).`

λ_{m+9} ← `happens(doAction(HostSrc,operation(Switch,sentPacket(packet)),T+9))`
← λ_{m+8}.

λ_{m+10} ← `holdsAt(operation(Switch,verifySpecificRule(packet)),T+10)`
← λ_{m+9}.

λ_{m+11} ← `(happens(doAction(Switch,operation(HostDst,sentPacket(packet)),T+11)),`

```
      ∧ happens(doAction(Switch,operation(Controller,sentPacket(packet)),T+11))).
      ← initiallyFalse(λm+10)

λm+3.1,
λm+4.1,
λm+5.1,
λm+6.1,
λm+7.1.
```

The purpose of the EC-based representation described in the previous sections is to formally specify the operation of the policy authoring, low-level controller, and policy refinement process. Although the logical inferences achieved with the aid of the Prolog interpreter are not integrated with the refinement toolkit, it is our aim as part of our future work to incorporate a Prolog engine as part of the refinement toolkit prototype.

4.4.4 EC-Based Formalism Experimental Evaluation

In this section, our goal was to measure the amount of iterations and rules to find QoS classes that fulfill the requirements of different SLAs. The experimental evaluation was modeled and performed in Prolog 6.6.4. Each experiment was run ten times. The experiments were performed on an AMD 2.0 GHz Octa Core with 32 GB RAM memory.

4.4.4.1 Scenarios

We created six SLAs by changing the number of expressions according to Table 4.5. We applied the six SLAs to five repository containing different amounts of classes. We populated each repository according to Table 4.6. Each QoS class considers all QoS requirements, *i.e.*, priority, bandwidth, delay, and jitter. Each QoS requirement has different values[1].

4.4.4.2 Experiments and Discussion

In this section, our goal was to measure the number of iterations for the identification of QoS requirements and for the query of all suggestions of QoS classes. Each SLA (Table 4.5) generated a set of prolog rules to find at least one occurrence of a QoS class. Each rule can be composed by a set of prolog operations. Each prolog operation is connected by connectives. Connectives can be of the type *"and"* (represented by a comma [,]) or of the type *"or"* (represented by a semicolon [;]). The prolog rules generated for each SLA are described in Table 4.7. The prolog operations used in the prolog rules are described in Table 4.8.

As can be seen in Table 4.7 some SLAs have generated more rules such as SLA 1(4), SLA

[1]The values for QoS requirements were generated randomly: between 0 and 999 for priority (where 0 is highest priority and 999 is lowest priority), between 2 kbps and 2^{12} kbps for bandwidth, between 1 ms and 999 ms for delay, and 10% of the delay value for jitter.

Table 4.5: Description of SLAs used in the experiments.

SLA	**Description of SLAs**
SLA_1	Streaming traffic should receive highest priority, lowest delay, lowest jitter, and highest bandwidth.
SLA_2	Peer-to-peer traffic should receive lowest priority, highest delay, lowest jitter, and highest bandwidth.
SLA_3	FTP traffic should receive highest bandwidth.
SLA_4	VoIP traffic should receive delay lower than 20ms and bandwidth higher than 128kbps.
SLA_5	SNMP traffic should receive priority higher than 500 and bandwidth lower than 16kbps.
SLA_6	SSH traffic should receive bandwidth higher than 1024kbps, delay lower than 50ms, and lowest jitter.

Source: by author (2015).

Table 4.6: Number of classes registered in the repository.

Repository	**Number of Classes**
Repository A	5
Repository B	10
Repository C	50
Repository D	100
Repository E	250

Source: by author (2015).

2(3), SLA 4(2), and SLA 6(3). This happens when the rule for an SLA does not match the information stored for QoS classes. Thus, our technique applies a first rule that includes all parameters found in an SLA. If there is not a match, new rules are generated until a match is found. In the worst case, the rule generated will fetch the parameters using the connective or (represented by a semicolon), for example, $parameter_1$ *or* $parameter_2$ *or* $parameter_3$ *or* $parameter_4$.

Table 4.9 shows the relationship between the number of iterations generated by each rule and the number of classes found. As can be seen, in all cases at least one class has been found by a rule. In addition, some rules have found multiple classes such as S5R1. This happens because each rule aims to find an ideal match of parameters ignoring the other parameters registered in the class. Thus, a rule aiming values of priority = 12 and bandwidth = 128kbps can identify classes with the following information: Class1: priority = 12, bandwidth = 128kbps; Class2: priority = 12, bandwidth = 128kbps, and delay = 10ms; Class3: priority = 12, bandwidth = 128kbps, delay = 10ms, and jitter = 1ms; Class4: priority = 12, bandwidth = 128kbps, and delay = 200ms.

Table 4.7: Rules performed for each SLA in each scenario.

SLA	Name*	Rule
1	S1R1	lowestValue(Class,priority,Priority), lowestValue(Class,delay,Delay), lowestValue(Class,jitter,Jitter), highestValue(Class,bandwidth,Bandwidth).
	S1R2	lowestValue(Class,priority,Priority), lowestValue(Class,delay,Delay), highestValue(Class,bandwidth,Bandwidth).
	S1R3	lowestValue(Class,priority,Priority), highestValue(Class,bandwidth,Bandwidth).
	S1R4	lowestValue(Class,priority,Priority), lowestValue(Class,delay,Delay).
2	S2R1	highestValue(Class,priority,Priority), highestValue(Class,delay,Delay), highestValue(Class,jitter,Jitter), lowestValue(Class,bandwidth,Bandwidth).
	S2R2	highestValue(Class,priority,Priority), highestValue(Class,delay,Delay), lowestValue(Class,bandwidth,Bandwidth).
	S2R3	lowestValue(Class,bandwidth,Bandwidth), highestValue(Class,priority,Priority).
3	S3R1	highestValue(Class,bandwidth,Bandwidth).
4	S4R1	findParLower(Class,delay,20), findParHigher(Class,bandwidth,128).
	S4R2	findParLower(Class,delay,20); findParHigher(Class,bandwidth,128).
5	S5R1	findParHigher(Class,priority,500), findParLower(Class,bandwidth,16).
6	S6R1	findParHigher(Class,bandwidth,1024), findParLower(Class,delay,50), lowestValue(Class,jitter,Jitter).
	S6R2	findParHigher(Class,bandwidth,512), findParLower(Class,delay,20).
	S6R3	findParHigher(Class,bandwidth,1024); findParLower(Class,delay,50); lowestValue(Class,jitter,Jitter).

$*$ Name is the short representation of a rule where S = SLA and R = Rule. Thus, S1R1 is the first rule for SLA_1.

Source: by author (2015).

Table 4.8: Description of Operations used in the rules.

Prolog Operation	**Description**
findParLower(Class,Parameter,Value) :- isMemberParameter(Class,Parameter,Value2), Value2 < Value.	Used to find a parameter *lower* than a specific value.
findParHigher(Class,Parameter,Value) :- isMemberParameter(Class,Parameter,Value2), Value2 > Value.	Used to find a parameter *higher* than a specific value.
findParIdentical(Class,Parameter,Value) :- isMemberParameter(Class,Parameter,Value2), Value2 = Value.	Used to find a parameter *identical* the specific value.
lowestValue(Class,Parameter,Value) :- isMemberParameter(Class,Parameter,Value), (isMemberParameter(Class2,Parameter2,Value2), Parameter = Parameter2, Value2 < Value, Class ! = Class2).	Used to find the QoS Class which has the *lowest* parameter among the registered QoS Classes.
highestValue(Class,Parameter,Value) :- isMemberParameter(Class,Parameter,Value), (isMemberParameter(Class2,Parameter2,Value2), Parameter = Parameter2, Value2 > Value, Class ! = Class2).	Used to find the QoS Class which has the *highest* parameter among the registered QoS Classes.

Source: by author (2015).

Figure 4.3 shows the number of iterations for each SLA in each repository. Each color indicates a different prolog rule. For example, in Figure 4.3(a) the repository with 50 classes performed 4 prolog rules to find at least one class.

As can be observed in Figure 4.3, by increasing the number of classes the amount of iterations for each repository grows. This increase is visible in all experiments performed with the SLAs. This behavior is expected, since the number of classes has influence on the number of iterations to obtain the ideal matches between SLAs and QoS classes. In addition, the SLAs requiring extreme values, *e.g.*, lowest delay, highest bandwidth also increase the number of iterations because they needed to find lowest or highest values.

Table 4.9: Results of the iterations and classes found for each scenario.

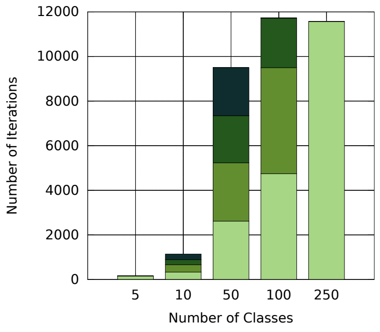
SLA	**Rule***	**Number of Classes**									
		5		10		50		100		250	
		Iterat.	**Found**	**Iterat.**	**Found**	**Iterat.**	**Found**	**Iterat.**	**Found**	**Iterat.**	**Found**
1	S1R1	158	1	332	0	2617	0	4747	0	11573	1
	S1R2	N/A	N/A	332	0	2617	0	4747	0	N/A	N/A
	S1R3	N/A	N/A	226	0	2111	0	2232	1	N/A	N/A
	S1R4	N/A	N/A	252	1	2157	1	N/A	N/A	N/A	N/A
2	S2R1	77	0	137	0	1783	1	1352	0	10763	0
	S2R2	77	0	137	0	N/A	N/A	1352	0	10763	0
	S2R3	118	1	270	1	N/A	N/A	12218	1	40923	1
3	S3R1	68	1	289	2	1743	1	2137	2	13563	6
4	S4R1	19	0	12	0	58	2	114	4	290	13
	S4R2	14	1	24	1	N/A	N/A	N/A	N/A	N/A	N/A
5	S5R1	14	2	33	5	125	20	293	45	718	107
6	S6R1	7	0	12	0	564	3	2126	4	16579	13
	S6R2	7	0	12	0	N/A	N/A	N/A	N/A	N/A	N/A
	S6R3	100	1	314	2	N/A	N/A	N/A	N/A	N/A	N/A

$*$ Rules described in Table 4.7.

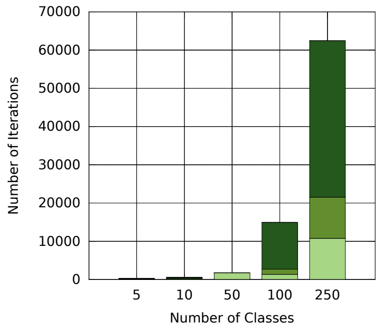
The occurrence of an N/A means that a previous rule found a class.

Source: by author (2015).

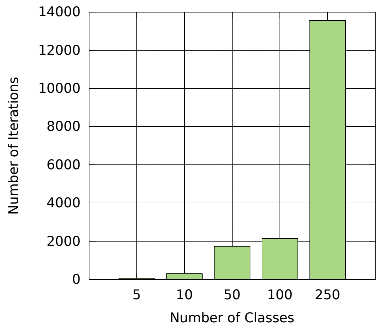
Figure 4.3: Average number of iterations for each SLA performed in each scenario.



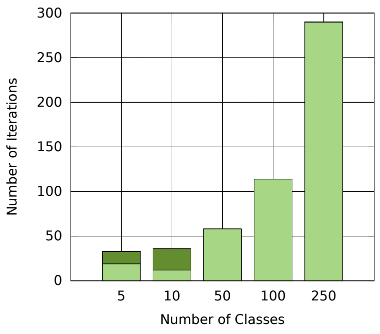
(a) SLA 1



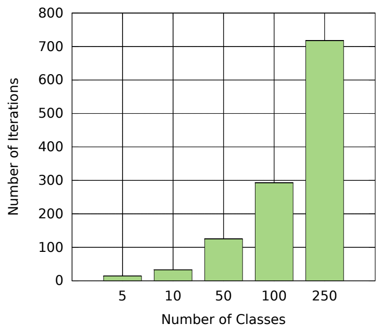
(b) SLA 2



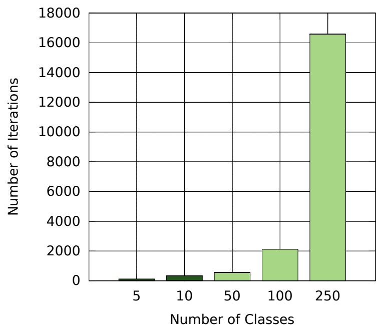
(c) SLA 3



(d) SLA 4



(e) SLA 5



(f) SLA 6

Rule 1 🟩 Rule 2 🟩 Rule 3 🟩 Rule 4 ⬛

Source: by author (2015).

5 PROTOTYPE AND EXPERIMENTAL EVALUATION

This chapter provides an overview of the prototype developed and the experiments in order to validate our policy refinement toolkit. In Section 5.1 we describe the prototype. In Section 5.2 we present and discuss the experimental evaluation.

5.1 Prototype

In this section we present the prototype developed. We briefly describe our low-level controller in Section 5.1.1. In Section 5.1.2 we describe our Policy Authoring Framework.

5.1.1 Low-level Controller Prototype

In order to apply our solution we need to customize and to create some functionality of an OpenFlow controller. This was required for collecting information about the network infrastructure, which is later used, for example, to calculate optimal routes. This customization is based on SDN native features. Thus, it can be applied to any controller implementation. Therefore, our solution is not tied to any specific controller design or language. We emphasize that even though the commands supported by the forwarding elements are standardized, the controllers require different programming languages and/or support different features. This difference among controllers can reflect in the effort for customization that must be spend by an infrastructure-level programmer. For example, topology discovery, which is available in the POX (MCCAULEY et al., 2013) controller from the discovery module, is a feature that can be achieved by others controllers in different implementations by collecting Link Layer Discovery Protocol (LLDP) packets that are sent from the network devices.

The controller was designed using POX (MCCAULEY et al., 2013), written in Python Language (PYTHON, 2014). We take the *l2_multi* module as a kernel basis for development of our customized controller. This module has some important presets for our solution such as support to *discovery library*, used for example for discovery of network elements and *spanning tree library*, used for example to calculate and deploy different paths between source and destination elements.

We used the OpenFlow Protocol 1.0 (PFAFF et al., 2009; MCKEOWN et al., 2008) to coordinate network devices because it is the currently stable version of the protocol. We know the resources provided by the newer versions introduced a more flexible pipeline with multiple tables and the use of groups to organize flow rules. However, we found in 1.0 version all required support to our solution. Obviously, the use of newer versions can optimize issues such as processing and even organization and grouping of rules in the switches. Nevertheless, these issues are handled by our solution which has shown satisfactory results.

As mentioned previously, the behavior of the customized OpenFlow controller is divided into three phases. In the following we present an overview of the modules implemented in each phase:

1. Modules of the Startup phase:
    - **Network Discovery –** monitors the network in order to discover the elements and topology using a customized POX discovery library.
    - **Network Information Collector –** sends ICMP packets to analyze the network characteristics such as delay and available bandwidth. Additionally, based on the delay values, it calculates the jitter for each path.
    - **Path Generator –** calculates the paths between all elements using Dijkstra's Shortest Path Algorithm (DIJKSTRA, 1959) and a customized POX spanning tree library. In order to generate each path, it creates various spanning trees between the source and destination elements. Later, it triggers the *Network Information Collector* to perform the gathering and processing of information from each tree created. Each path is generated considering the available bandwidth, number of hops, average delay and jitter.
    - **Standard Rules Deployer –** deploys the standard rules based on best-effort paths found by the *Path Generator* module using a customized POX spanning tree library.
2. Modules of the Events phase:
    - **Service Events Monitor –** waits for service packets without specific rules.
    - **Analysis Trigger –** analyzes the duration of each flow to propose modifications in the network configuration triggering the *Analysis Phase*.
3. Modules of the Analysis phase:
    - **Service Analyzer –** finds services requirements and its specifics rules in the policy repository.
    - **Specific Rules Deployer –** deploys specific rules based on service requirements found by the *Service Analyzer* module and the *Path Generator* module using a customized POX spanning tree library.
    - **Network Analyzer –** sends ICMP packets to analyze the network changes and triggers the *Service Analyzer* module.

5.1.2 Policy Authoring Framework Prototype

In this section we describe the design/operation of our Policy Authoring framework for SDN management. The main goal is to enable operators to express business goals, *e.g.*, Service Level Agreements (SLAs), without having to specify in detail what elements in the network infrastructure should receive the configurations and how they should be configured.

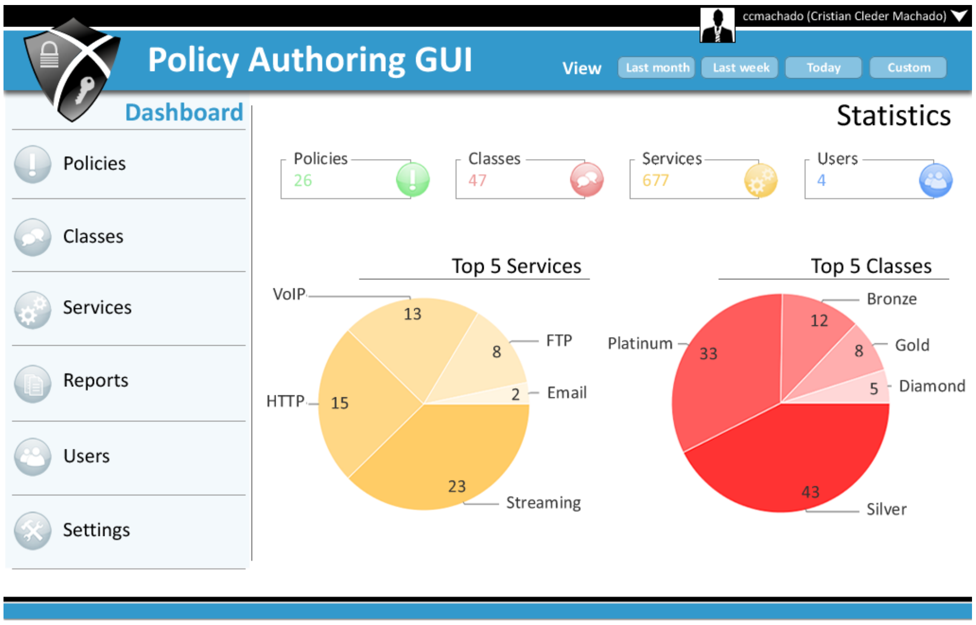
We developed the Policy Authoring GUI module and the Configurator GUI module using the Django web framework[1]. We chose Django due to its support to the Python[2] language and the support it provides to create web applications. For the interface design we used the Bootstrap front-end framework[3].

Section 5.1.2.1 introduces the *Policy Authoring GUI Module*. We show the *Configuration GUI Module* in Section 5.1.2.2.

5.1.2.1 Policy Authoring GUI Module

We developed a user-friendly Graphical User Interface (GUI) module for Policy Authoring in order to allow the configuration of the network through business goals. Thus, a business-level operator uses the Policy Authoring GUI to express high-level goals and receive feedback from his requests.

Figure 5.1: Policy Authoring Graphical User Interface Module.



Source: by author (2015).

Figure 5.1 illustrates the home screen of the Policy Authoring GUI. It presents statistics about the number of policies, classes, services, and users registered. In addition, it shows two pie charts about the top 5 services that most appear in policies and the top 5 QoS classes that most have linked policies. The *dashboard* is composed of the following items:

- **Policies** – Used by business-level operators to create, search, edit, remove, enable or disable policies. Operators can also associate a high-level SLA with the QoS class that

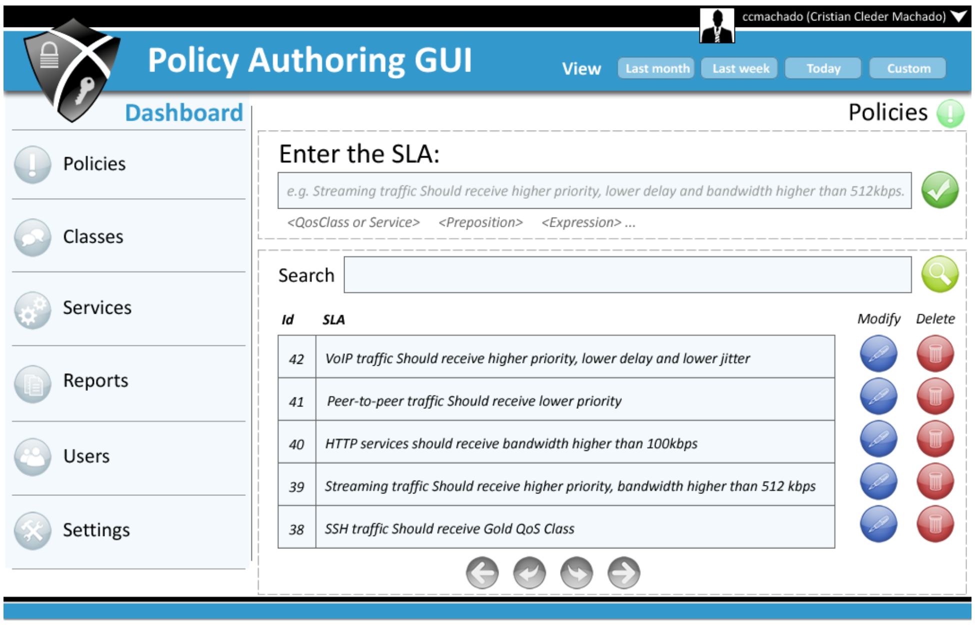
[1]http://www.djangoproject.com/
[2]http://www.python.org/
[3]http://getbootstrap.com/

best meets the SLA requirements.

- **Classes** – Used to specify QoS classes. Infrastructure-level programmers and business-level operators can perform the necessary parameter settings for each class.
- **Services** – Used by infrastructure-level programmers to record, edit, and delete services. Also, it is here that a service can be associated with a QoS class.
- **Reports** – Used by business-level operators and infrastructure-level programmers to view reports, *e.g.*,, number of policies, services, classes; classes containing more policies; services that most appear in policies; services that less appear in policies. Additionally, some reports can be filtered by specific parameters, *e.g.,* priority, delay.
- **Users** – Used to manage system users.
- **Settings** – Used to determine the system settings, such as settings to connect to the database.

The input screen for writing SLAs is depicted in Figure 5.2.

Figure 5.2: Input screen to SLA operations.



e.g. Streaming traffic Should receive higher priority, lower delay and bandwidth higher than 512kbps.

<QosClass or Service> <Preposition> <Expression> ...

Id	SLA	Modify	Delete
42	VoIP traffic Should receive higher priority, lower delay and lower jitter		
41	Peer-to-peer traffic Should receive lower priority		
40	HTTP services should receive bandwidth higher than 100kbps		
39	Streaming traffic Should receive higher priority, bandwidth higher than 512 kbps		
38	SSH traffic Should receive Gold QoS Class		

Source: by author (2015).

5.1.2.2 *Configurator GUI Module*

Our aim is to facilitate not only the description of business objectives but also the configuration of the infrastructure. The *Configurator GUI* is designed to manage the registration of services and parameters. An infrastructure-level programmer inserts service information, such as *ServiceName* and *ServicePort* (as used in TCP/IP). Subsequently, the infrastructure-level programmer may create QoS classes with parameters and their respective values. The fields that may be informed are *ClassName*, *Priority, Bandwidth, Delay,* and *Jitter*.

We decided to group services by class, thus after QoS classes have been defined, each service is associated with a QoS class. This step is important because if services are previously associated with some class, the toolkit will have a better performance since there will be an entry in the repository for a group of services as opposed to one entry for each service. Thus, services with similar requirements can be grouped into a single class while maintaining fairness among competing services in the same link.

5.2 Experimental Evaluation

Firstly, in Section 5.2.1, we present experiments with the controller prototype. Secondly, we present experiments with the policy authoring framework in Section 5.2.2. Thirdly, in Section 5.2.3, we present experiments with the end-to-end process, *i.e.*, from the policy authoring process to deployment of low-level rules.

5.2.1 Controller Experiments

This section describes our test environment and some initial results with the controller prototype. The experiments were performed on an AMD 2.0 GHz Octa Core with 32 GB RAM memory. Each experiment was run thirty times. The scenarios were created using the Mininet emulator.

5.2.1.1 Scenarios

Five scenarios were created in our experiments. The scenarios that we built had increasing numbers of switches and redundant links, thus increasing path diversity between any two hosts. Table 5.1 shows the number of hosts, switches, and links in each scenario.

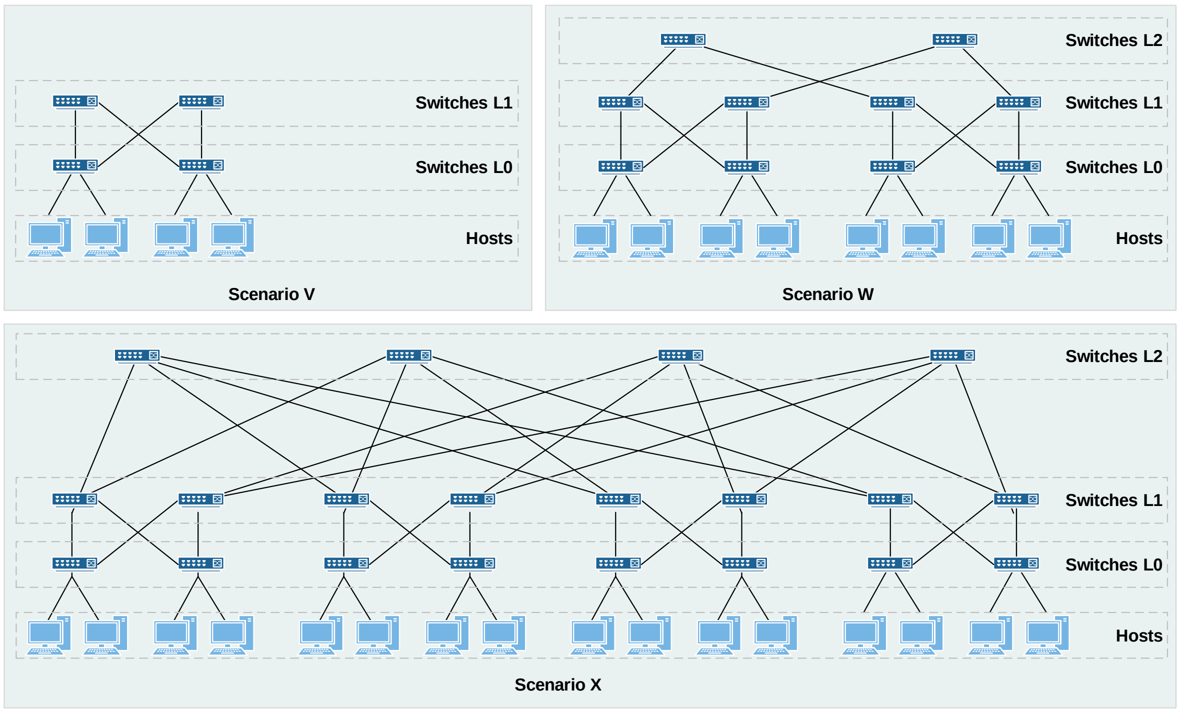
Table 5.1: Number of hosts, switches, and links in each scenario.

Scenario	**Switch L0**	**Switch L1**	**Switch L2**	**Switch L3**	**Host**	**Link**
V	2	2	0	0	4	4
W	4	4	2	0	8	12
X	8	8	4	0	16	32
Y	16	16	8	4	32	80
Z	32	32	16	8	64	96

Source: by author (2015).

The topologies used in the experiments were based on Fat-Tree topologies (LEISERSON, 1985). An overview of scenarios V, W, and X is shown in Figure 5.3. Due to the large number of elements, scenarios Y and Z could not be clearly presented in a figure. By increasing the number of switches and redundant links, these topologies can be used to, for example, maintain system availability in the presence of problematic links and to reduce traffic congestion in the infrastructure.

Figure 5.3: Scenarios for the experiments with increasing number of switches and link redundancy.



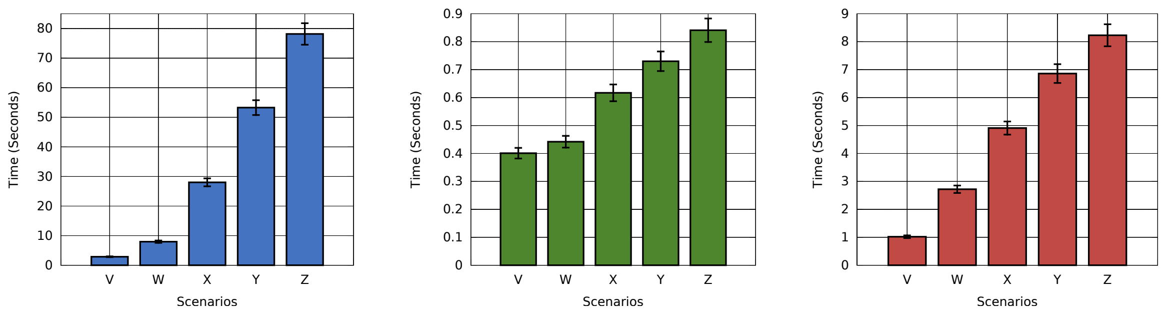
Source: by author (2015).

5.2.1.2 *Experiments and Discussion*

Our experiments with the controller aim to measure the average time to discover the network, *i.e.*, all network elements and all possible paths between these elements and deploying standard rules (performed in the *Startup Phase* of the controller) (Figure 5.4(a)); deploying specific rules from a previously known topology (Figure 5.4(b)); and deploying specific rules without previous knowledge of topology (Figure 5.4(c)).

As can be observed in Figure 5.4(a), with the increase in the number of elements and links in the topology, the time to calculate all paths between any network element increases, reaching 78.17 seconds in scenario Z. Even with this growth, we justify the execution of this calculation and installation of these initial standard rules because we want to avoid any delay when new flows arrive. We also take into consideration that, due to the initial calculation, later, in the *Analysis Phase* (see Section 3.2.3), we have already all possible flow paths, which makes the calculation of specific rules faster, reaching 0.841 seconds in scenario Z (as can be seen in Figure 5.4(b)). The increase in the number of elements in each topology reflects in a small increase in the time for deploy of the specific rules as can be observed in Figure 5.4(b). Thus, there is a minimum time necessary for deploying the specific rules, but that does not delay service flow processing due to the pre-established standard rules.

Figure 5.4: Average time for recognition of links, calculation, and installation of rules in the switches.



(a) Time for calculation and installation of standard rules in the switches in the startup phase and recognition of topology in each experiment.

(b) Time for calculation and installation of specific rules in the switches in each experiment.

(c) Time for recognition of the first link between source and destination and calculation and installation of specific rules in the switches in each experiment.

Source: by author (2015).

For comparison purposes, Figure 5.4(c) shows what the time for recognition of a link and installation of rules would be without a previous knowledge about the topology and its links. As can be observed, the time for scenario Z reaches 8.23 seconds. This time is nearly 10 times lower than the time spent by our solution for previous recognition of the topology and deploy standard rules (78.14 seconds according to Figure 5.4(a)). However, our experiments have shown that an initial calculation of best-effort paths between network elements can increase the performance of the network during runtime. Certainly there is an overhead related to the startup phase of the controller (Figure 5.4(a)). Nevertheless, this is justifiable as it improves the performance of the network during runtime when observed the time of 0.841 seconds for scenario Z in Figure 5.4(b) that is approximately 10 times lower than the time of 8.23 seconds for scenario Z in Figure 5.4(c). As mentioned previously, these results show that due to the calculations performed initially, our proposal is able to perform a quick reconfiguration of specific rules, since we already have a populated list of the best links between network elements. Thus, if we observe that in a network several flows are being forwarded, in a short time our solution obtains a better time compared with approaches without a previous knowledge of the topology.

It is noteworthy that the path chosen in the experiments (Figure 5.4(c)) is the first one found between a source and a destination. Thus, there is no checking if this link is the best link to a particular flow, *i.e.*, if it complies with best-effort requirements of a particular flow. Nevertheless, rules deployed by our solution intend to choose the best rules to fulfill the service requirements among several possible links.

5.2.2 Policy Authoring Experiments

In this section, our goal was to measure the response time of the Policy Authoring Framework, according to the increased complexity of each SLA and number of classes stored in the repository. Each experiment was run thirty times. The experiments were performed on an AMD 2.0 GHz Octa Core with 32 GB RAM memory.

5.2.2.1 Scenarios

We created three SLAs (Table 5.2) by changing the number of expressions. We applied the three SLAs to five repository containing different amounts of classes. We populated each repository according to Table 5.3. Each QoS class considers the possibility of a different amount of QoS requirements, *i.e.*, a QoS class may contain all QoS requirements such as priority, bandwidth, delay, and jitter whilst another QoS class may contain only one QoS requirement such as bandwidth. Each QoS requirement has different values[4].

Table 5.2: Description of SLAs used in the experiments.

SLA	Description of SLAs
SLA_1	Peer-to-peer traffic should receive lowest Quality of Service and lowest priority compared with other services.
SLA_2	Streaming traffic should receive highest priority, lowest delay and bandwidth higher than 512kbps.
SLA_3	VoIP traffic should receive highest priority, delay lower than 200ms, lowest jitter, and bandwidth higher than 128kbps.

Source: by author (2015).

5.2.2.2 Experiments and Discussion

We measured the time spent on the identification of regexes and on the query of all suggestions of classes in the repository. Figure 5.5 shows the average time for each SLA in each repository.

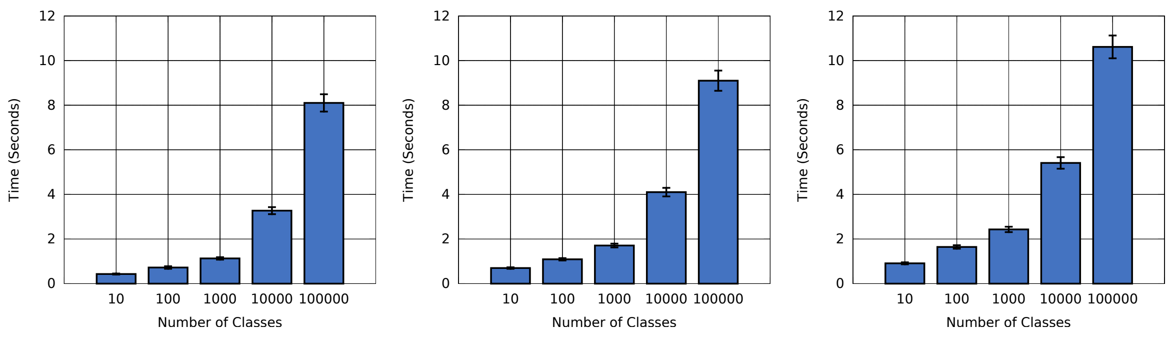
[4]The values for QoS requirements were generated randomly: between 0 and 999 for priority, between 2 kbps and 2^{10} kbps for bandwidth, between 1 ms and 999 ms for delay, and 10% of the delay value for jitter.

Table 5.3: Number of classes registered in the repository.

Repository	**Number of Classes**
Repository A	10
Repository B	100
Repository C	1000
Repository D	10000
Repository E	100000

Source: by author (2015).

Figure 5.5: Average response time for each SLA.



(a) SLA 1 (b) SLA 2 (c) SLA 3

Source: by author (2015).

As can be observed in Figure 5.5, with the increase in the number of classes and the level of complexity of the SLA, the time to calculate all suggestions of classes increases. Even with this growth, the policy authoring process performs well when we observe and compare the values of standard deviation and confidence interval according to Table 5.4.

5.2.3 End-to-end Process Experiments

We present in this section experiments and initial results obtained with the implemented toolkit. Our goal is to measure the response time of the end-to-end process, *i.e.*, from policy authoring to deployment of low-level rules in the controller device. Each experiment was run thirty times. The scenarios were created using the Mininet emulator and experiments were performed on an AMD 2.0 GHz Octa Core with 32 GB RAM memory.

Table 5.4: Experiment results.

	Repository	Standard deviation	Confidence interval
SLA_1	A	0.02	0.000435567
	B	0.02	0.000567653
	C	0.03	0.000965765
	D	0.04	0.000987866
	E	0.06	0.001676778
SLA_2	A	0.13	0.004652696
	B	0.17	0.005444554
	C	0.21	0.007234412
	D	0.24	0.006231234
	E	0.29	0.008654578
SLA_3	A	0.25	0.009656654
	B	0.31	0.009356912
	C	0.33	0.010324475
	D	0.65	0.010344532
	E	0.84	0.012644334

Source: by author (2015).

5.2.3.1 Scenarios

In order to perform the experiments, we created three SLAs (Table 5.5) by changing the number of expressions, where SLA_2 has more expressions than SLA_1 and SLA_3 has more expressions than SLA_2. Our goal is to show the robustness and efficiency of the refinement process when we increase the number of expressions that should be compared. We also created three scenarios (Table 5.6) by varying the number of network devices and adding redundant links between some network devices. The scenarios used in the experiments were based on Fat-Tree topologies (LEISERSON, 1985). Our goal was to demonstrate the ability of the framework to operate in increasingly large topologies.

We applied the three SLAs to five different repositories and populated each repository according to Table 5.7. We performed experiments on all variations of SLAs, repositories, and scenarios.

Table 5.5: Description of SLAs used in the experiments.

SLA	Description of SLAs
SLA_1	HTTP traffic should receive lowest priority.
SLA_2	Streaming traffic should receive highest priority, lowest delay and bandwidth higher than 512kbps.
SLA_3	VoIP traffic should receive highest priority, delay lower than 200ms, lowest jitter, and bandwidth higher than 128kbps.

Source: by author (2015).

Table 5.6: Number of switches and links in each scenario.

Scenario	Switch L0	Switch L1	Switch L2	Switch L3	Switch L4	Host	Link
X	16	8	8	4	0	32	88
Y	32	16	16	8	4	64	176
Z	64	32	32	16	8	128	210

Source: by author (2015).

Table 5.7: Number of classes registered in the repository.

Repository	Number of Classes
Repository A	10
Repository B	100
Repository C	1000
Repository D	10000
Repository E	100000

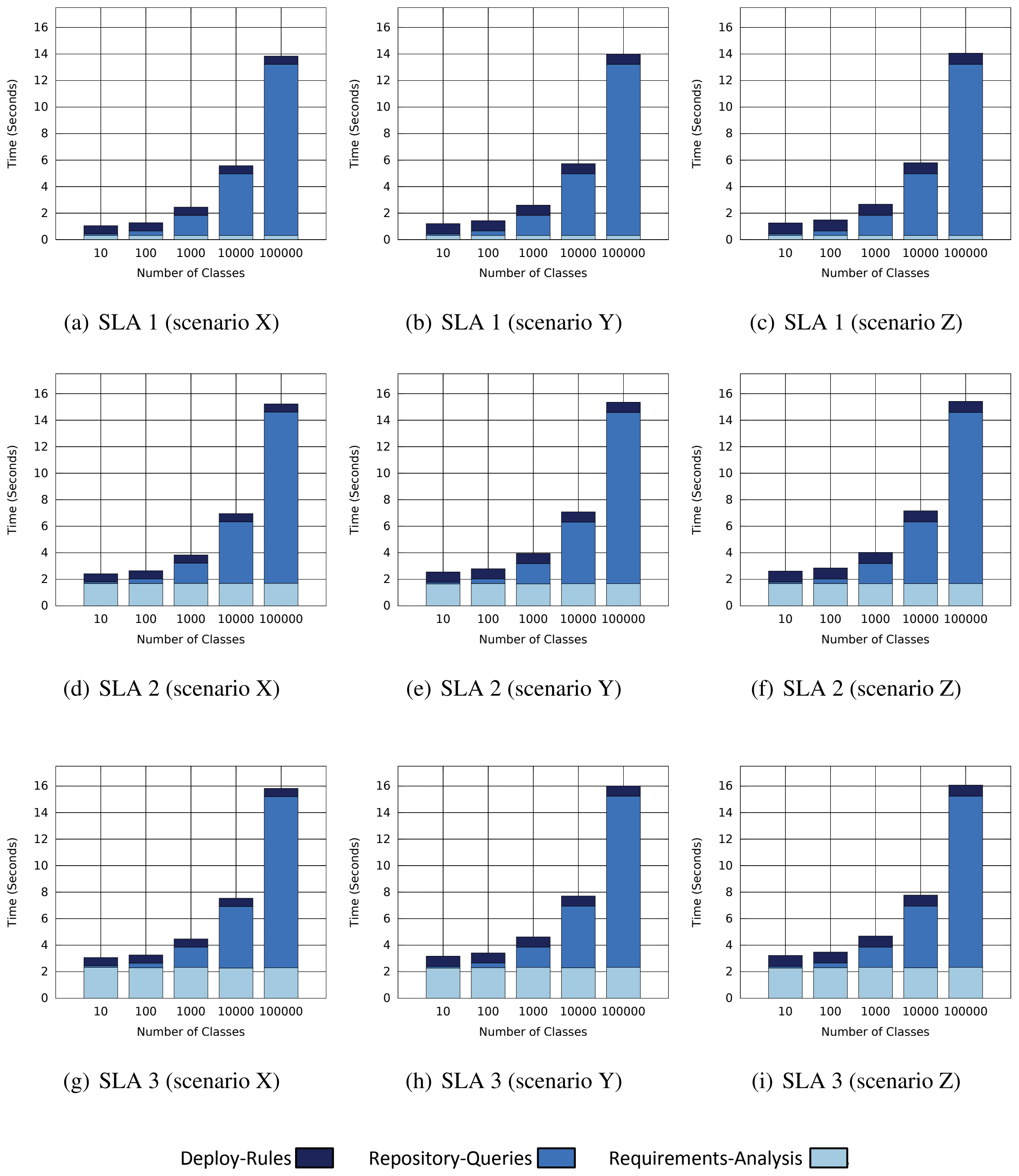
Source: by author (2015).

5.2.3.2 *Experiments and Discussion*

In particular, the experiments described in this sections intend to evaluate our prototype in terms of average execution time and percentage of the total time occupied by each stage of our policy refinement toolkit.

Figure 5.6 shows the average response time for each SLA in each scenario. We break the total execution time down in three categories, namely requirements analysis (*i.e.*, parse the SLAs and their regexes), repository queries (*i.e.*, search for the best matching QoS class), and deploy rules (*i.e.*, install the flow rules in the controller). By increasing the number of classes,

Figure 5.6: Average response time for SLAs 1, 2, and 3 performed in scenarios X, Y, and Z.

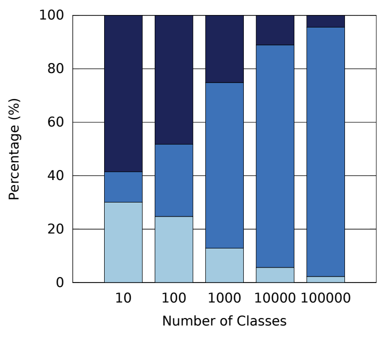


Source: by author (2015).

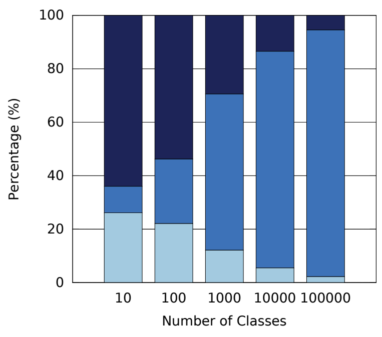
it is possible to observe that the average time spent performing repository queries also grows. This increase is visible in all experiments performed with SLAs 1, 2, and 3. This behavior is expected, since the number of classes has influence on the number of queries to obtain the ideal matches between SLAs and QoS classes.

In Figure 5.7 the y-axis shows the percentage of the total time occupied by each process in the experiments performed with SLAs 1, 2, and 3 in each scenario. From these results it is possible to note that, according to the level of complexity of each SLA, the percentage of time for analyzing requirements also increases. This happens due to the increase in the number of occurrences of regular expressions found in each SLA.

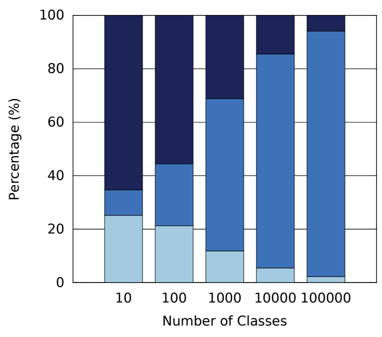
Figure 5.7: Percentage of total time for SLAs 1, 2, and 3 performed in scenarios X, Y, and Z.



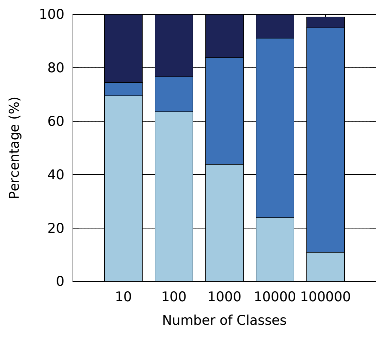
(a) SLA 1 (scenario X)



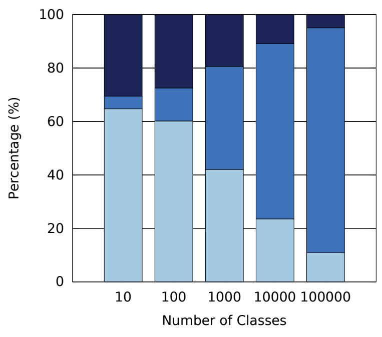
(b) SLA 1 (scenario Y)



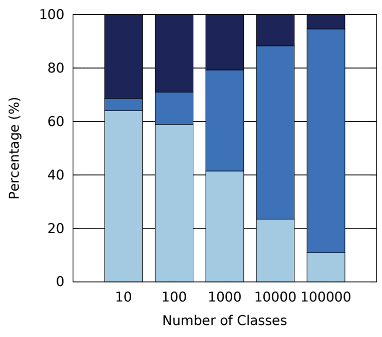
(c) SLA 1 (scenario Z)



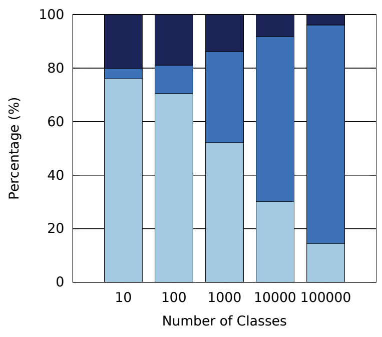
(d) SLA 2 (scenario X)



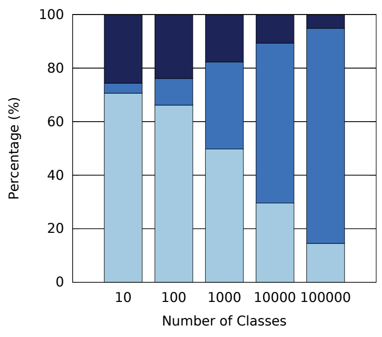
(e) SLA 2 (scenario Y)



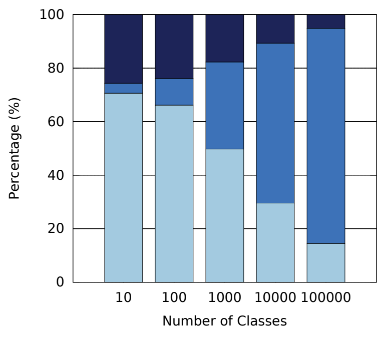
(f) SLA 2 (scenario Z)



(g) SLA 3 (scenario X)



(h) SLA 3 (scenario Y)



(i) SLA 3 (scenario Z)

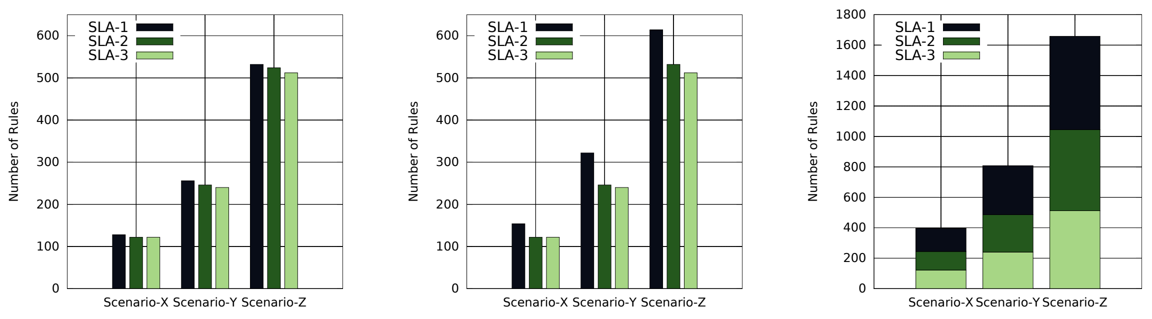
Deploy-Rules ■ Repository-Queries ■ Requirements-Analysis ■

Source: by author (2015).

Figure 5.8(a) shows the total number of rules generated by refining each SLA *separately* in each scenario. As can be observed, each SLA generates practically the same number of rules in each scenario. SLA 1 shows a small difference in the number of rules deployed compared to SLAs 2 and 3. This occurs because SLA 1 has lower QoS requirements (*i.e.,* low priority) compared to other services, which causes the choice of routes with more hops and consequently causes rules to be deployed in more devices. It is worth mentioning that the total number of rules generated by the policy authoring framework is smaller than the total number of rules that would have to be manually created on all network devices. This is because, as our approach is based on routing, it creates a spanning tree to find all routes between sources and destinations. Thus, some routes may be common between different sources and destinations. As a result, a number of switches do not need to be configured, thus reducing the total number of rules required in each scenario.

The growth in the total number of rules in SLA 1 appears more clearly when we performed *simultaneously* the refinement of the three SLAs in each scenario (Figure 5.8(b)). Our framework attempts to fulfill the requirements of each SLA. In order to achieve this, it identifies the possibility of routing (balancing) each SLA by alternative routes without failing to fulfill their requirements. Thus, SLA 1 receives routes with more hops in order not to compete with SLAs 2 and 3 which have higher priority requirement.

Figure 5.8: Number of rules deployed in each scenario.



(a) Total number of rules deployed by each SLA performed separately in each scenario.

(b) Total number of rules deployed by each SLA performed simultaneously in each scenario.

(c) Total amount of rules deployed by all SLAs in each scenario.

Source: by author (2015).

Finally, Figure 5.8(c) shows the total amount of rules generated by all SLAs in each scenario. This illustrates the benefits of our policy authoring and refinement approach, in which the infrastructure-level programmer does not need to be concerned with the number of low-level configuration rules to be deployed in the network. Our results suggest that the prototype is able to support the refinement of SLAs and the installation of flow rules in large-scale deployments. Even if we consider the scenario with the largest number of switches and links (Figure 5.6(c)), and the largest number of QoS classes, the total measured time remains within acceptable bounds. Moreover, as mentioned previously, the framework optimizes the deployment of rules according to the requirements of each SLA and according to each scenario.

6 RELATED WORK

In this work, we advocate that policy refinement techniques may reduce the manual work overload for Software-Defined Networking configurations. The literature presents studies using separately techniques and frameworks to policy refinement, and studies using SDN to achieve improvements in the network resource management. SDN studies which may use and are already using any policy/rule type are described in Section 6.1. Section 6.2 presents works about PBM and Policy Refinement.

6.1 Software-Defined Networking

SDN features have been employed to enhance the monitoring and management of network traffic. Foster *et al.* (FOSTER et al., 2011) introduced a new language for network programming supporting OpenFlow called Frenetic. Frenetic has a set of operators for handling network traffic flows, and a runtime that abstracts the details related to installing and uninstalling low-level rules in switches. Monsanto *et al.* (MONSANTO et al., 2013) introduced the Pyretic language. Pyretic introduced two main programming abstractions that have simplified the creation of modular management programs. Nonetheless, Frenetic and Pyretic are just languages to model low-level rules and do not employ any policy refinement technique to translate high-level policies into a set of low-level policies.

Rubio-Loyola *et al.* (RUBIO-LOYOLA et al., 2011) investigated the sharing of virtualized network resources in software-defined networking. The authors proposed an autonomic management system capable of separating control and data planes, providing greater isolation in the execution of applications. Also, an Orchestration Plane (OP) was presented, which aims to manage the system behavior in response to context changes, and in accordance with business goals and policies. However, the authors do not specify if there is any refinement approach to automate the translation of policies in different levels of abstraction.

Min *et al.* (MIN et al., 2012) developed a FlowVisor to enhance the admission control and bandwidth scheduling to network resources. The visor has provided an improvement in the dynamic control of network resources based on requirements and conditions for using the network. Notwithstanding, FlowVisor does not provide a framework to write high-level policies in a natural language and consequently does not use any policy refinement technique. In addition, FlowVisor does not provide or use information about network guarantees to accommodate service requirements.

Kim and Feamster (HYOJOON; FEAMSTER, 2013) presented a discussion about three problems with the current state-of-the-art in network management: changes to network behavior; a high-level language for network configuration; and vision and control over tasks for performing network analysis and troubleshooting. Further, the authors described the use of the Procera language to assists operators to express network policies that react to various types of events using a high-level functional programming language. Despite producing a broad discussion on network management problems, this work does not investigate aspects concerning policy refinement and its implementation issues.

6.2 Policy-Based Management

The use of PBM and policy refinement in computer networks has been investigated for over a decade. Bandara *et al.* (BANDARA et al., 2005) presented the use of goal design and abductive reasoning to derive strategies that attain a specific high-level goal. Policies can be refined by combining strategies with events and restrictions. The authors provide tool support for the refinement process, and use examples of DiffServ (BLAKE et al., 1998) QoS management. Craven *et al.* (CRAVEN et al., 2011) presented a policy refinement process for authorization and obligation that involves stages of decomposition, operationalization, re-refinement, and deployment. The work described in detail how a formalization of UML information on system objects, a high-level policy, and decomposition rules that relate actions can produce concrete low-level policies. Rubio-Loyola *et al.* (WALLER et al., 2006) presented a goal-oriented approach for goal decomposition using KaOS. The refinement approach makes use of linear temporal logic and reactive systems analysis techniques, thus generating deployable policies in Ponder. These works were limited due to the characteristics of traditional IP networks, such as best-effort packet delivery and distributed control state in forwarding devices. In addition, it does not use information about network guarantees to accommodate the service requirements.

Reeder *et al.* (REEDER et al., 2007) investigate usability challenges in policy authoring interfaces. Some of the challenges that were found are related to the use of consistent terminology and how users express their policies. Johnson *et al.* (JOHNSON et al., 2010) present a template-based framework for policy authoring. The work describes the relationship between general templates and specific policies, and the skills required from users to produce high-quality policies. Although these research efforts investigate important issues regarding policy authoring, none of them specifies a formal language for authoring, the use of logical reasoning to assist the refinement process, or experimental results.

Zhao *et al.* (ZHAO; SAKR; LIU, 2013) describe the design and implementation of an end-to-end framework for the management of cloud-hosted databases from a consumer's perspective. The approach is based on the interpretation of SLAs to assist the dynamic provisioning of databases. The framework checks if SLAs have changed and automatically performs corrective actions to enforce the new specifications. Villegas *et al.* (VILLEGAS et al., 2012) present a framework for the analysis of provisioning and allocation policies for Infrastructure-as-a-Service clouds, *i.e.*, policies to dynamically allocate resources which remain largely underutilized over time. Oriol Fito *et al.* (FITO et al., 2012) introduce a Business-Driven ICT Management (BDIM) model to satisfy the business strategies of cloud providers. The objective is to evaluate the impact of events related to ICT using business-level metrics. A Policy-Based

Management system analyzes these events and is able to determine automatically the ICT management actions that are most appropriate. However, the authors do not specify if there is any refinement approach to automate the translation of policies into lower levels of abstraction.

Carey and Wade (CAREY; WADE, 2008) presented the design, implementation and evaluation of a toolkit for the refinement of high-level policies into low-level policies. The work presented a framework describing the adaptive refinement of high-level policy into system rules and system behavior for policy execution. Brodie *et al.* (BRODIE et al., 2008) present a platform-independent framework to specify, analyze, and deploy security and networking policies. A portal prototype for policy authoring, based on natural language and structured lists, allows the management of policies from their specification to enforcement. The policy authoring portal enables web users to write policies, using a high-level language, which are translated and mapped to specific low-level configurations. Note that the work of Carey and Wade (CAREY; WADE, 2008) and the work of Brodie *et al.* (BRODIE et al., 2008) suffer from the same problems found in (BANDARA et al., 2005), (CRAVEN et al., 2011), (WALLER et al., 2006), (JOHNSON et al., 2010), and (REEDER et al., 2007).

In summary, SDN enables the deployment of network applications that perform sophisticated traffic monitoring and traffic processing. As a result, SDN has become a suitable scenario for the application of techniques and approaches for improved infrastructure management. SDN has used policies for defining some aspects of network management. However, policies are often written for specific situations directly in the controller, and specific techniques that can facilitate policy refinement in SDN have not been explored yet.

Despite efforts related to PBM have achieved satisfactory results, they were also limited by the characteristics imposed by traditional IP networks, such as best-effort packet delivery and distributed control state. We distinguish our policy refinement solution from other existing approaches by exploring the characteristics of SDN architectures, such as centralized control plane and overall view of the network infrastructure to enhance the policy refinement process.

The work presented in this dissertation relied on policy refinement to model the behavior of SDN components without the need to recode them or interrupt the network operation. Although some of the ideas in our toolkit are inspired by the research efforts above, to the best of our knowledge, this is the first time a policy refinement toolkit for SDN management is presented.

7 CONCLUDING REMARKS

Despite the benefits of SDN, the expected behavior of the network and their elements are defined by static rules written to handle with specific circumstances. This approach presents several problems such as human work overload to write, analyze, and manage a large set of hard-coded rules. It also limits or prohibits the development and deployment of new services and resources that were not anticipated when rules were written in the controller. Finally, the low-level rules are difficult to analyze and do not provide guarantees for compliance with the high-level goals.

In this context, we advocate that a possible solution to minimize these problems is the use of the PBM paradigm and policy refinement techniques. PBM in SDN can be used to specify goals and constraints in the form of rules to guide the operation of network elements. Policy abstractions can be used not only to adapt the controlled system, but also to adjust the policies themselves, changing their behavior to better achieve the system goals. For example, a policy may add QoS control rules for network monitoring and traffic analysis. If the rate at which the network is read is too high, communication costs for monitoring will end up interfering, and generating more traffic. However, if it is excessively low, the system may not be sufficiently aware of changes, failing to fulfill specific objectives. To solve this problem, trends in behavior can be analyzed and policies adjusted dynamically to better reflect the needs of specific resources.

7.1 Summary of Contributions

In this work, we presented a high-level policy refinement toolkit for software-defined networking management. We aim to remove much of the manual workload of administrators in the configuration of network elements. In particular, we focused on the refinement of QoS requirements for different applications and services (specified in SLAs) into the configuration of controllers and switches. As a result of our toolkit, we identified the resources that need to be configured in accordance with the SLAs, and successfully executed reactive dynamic actions used in the reconfiguration of the infrastructure.

We presented a controller that performs information gathering and calculates ideal routes that support the requirements of each service flow. Additionally, it configures the rules in the switches for optimization of network resource usage at runtime with minimal disruption to the network. Further, the controller provides an improved flow processing strategy, by reorganizing flows upon the arrival of new service requests. Our experiments have shown that the controller, the initial recognition of topology elements and their possible links improve the deployment time of specific rules for each service.

We also presented a policy authoring framework that can facilitate the configuration of an SDN based on the interpretation of SLAs. We expect that the framework will assist network administrators and operators to more easily specify overall service requirements, which can then be automatically translated into the configuration of an SDN. An important aspect to be emphasized is that our solution is flexible, and allows the operator to decide whether to accept or not the suggestions given by the framework. Thus, the operator can fully or partially accept the suggestion, or create his own configuration. Also, our experiments have shown that the framework performs well even with the increase in the number of QoS classes and in the complexity of the SLAs.

Finally, we described a formal representation using Event Calculus (EC) and applied logical reasoning to model both the system behavior and the policy refinement process for SDN management. It is our aim with this work to assist network operators to develop refinement tools and configuration approaches to achieve more robust SDN deployments. Thus, the development of the approaches becomes independent of the network controller implementation or policy language used. We demonstrated and discussed the formalism presenting case studies and examples where we described how to model and to represent network elements and their features; SLAs and QoS classes; the phases and their processes and operations executed by the controller; and the operation of the policy authoring modules. Further, we performed experiments to analyze the amount of iterations and suggestion of classes for the rules created by our solution.

7.2 Discussion and Lessons Learned

Several points that were not part of the proposed goals in this work call for our attention. As mentioned previously, we defined a formal representation of high-level policies in the form of SLAs using Event Calculus and used logical reasoning to model both the system behavior and the policy refinement process in SDN. Although this formalism is not integrated with the refinement toolkit, it can be incorporated for validation of the toolkit operations and of the properties of each predicate using logical inferences achieved with the aid of an interpreter, such as a Prolog engine. The integration with an interpreter can be used to increase the reliability of the refinement process, since all SLAs could be modeled as logical predicates and the mapping to low-level rules could be automatically performed via logical inferences. However, by incorporating an interpreter, there is clearly an increase in the response time of the processes, due to the set of inferences to be performed. Thus, it is important to analyze and select the set of processes and predicates requiring logical inferences, so that the time spent with an interpreter has the lowest operating cost.

It is noteworthy that many tools, techniques, and algorithms can be exploited to improve the toolkit processes. An example is related to the replacement of the repository. When observing the toolkit performance experiments, we found that in cases where the amount of information in the repository increases, there is an exponential growth in response time for the queries. By replacing the repository with NoSQL databases or databases with support for semantic metadata can dramatically improve the response time for the queries.

Regarding the controller phases, each phase comprises a set of modules responsible for specific actions. In the event phase, we only found the occurrence of flows in the network. However, several modules can be implemented to assist decisions on new flows and even concerning the occurrence of events in the network elements. An example is the creation of a module which identifies new network elements. Through the identification of a new element, a reformulation of the routes between all network elements can be performed. Another example is the development of a module which monitors the processing overhead on the switches. Through this monitoring, a load balancing process can be activated, for example, to reorganize routes between network elements.

We also noted some questions regarding the toolkit scalability. When we observe the experiment results related to the sum of rules deployed in the switches, it is apparent that a large set of classes leads to an increase in the number of rules produced. However, it is important to remember that the toolkit induces the operator to register each SLA with existing QoS classes, restricting the growth of the number of classes. Despite this, the flexibility provided by the toolkit for creation and modification of (new) QoS classes does not limit the maximum number of classes that can be created. Thus, a study of possible toolkit delimitations can reduce the number of rules produced, optimizing the use of computational resources as a whole.

Another issue that may be better explored is the way network information is gathered. We have used ICMP packets and applied a rule of three to calculate information/parameters/weights of links between network elements. In this context, other approaches, techniques, and tools can be explored.

7.3 Final Remarks and Future Work

We advocate that SDN, compared to traditional networks, can provide more features/information to perform more accurate refinements. However, it is worth noting that SDN cannot solve everything without any manual work. There are still moments where human interventions are necessary, such as, settings for the actions and rules of the controller that will be later implemented and enforced in the routers and switches.

One of the limitation is the behavior of our toolkit in scenarios with multiple controllers. Multiple controllers can generate policy conflicts when sharing the management of the same switches in the topology. As part of our future work, we plan to study this broader applicability of our work. In addition, we intend to investigate techniques for detection and resolution of policy conflicts. Conflicts may arise due to omissions, errors or differing requirements of administrators when specifying policies. One common source of policy conflicts is the refinement process itself, during the translation of high-level goals into implementable low-level policies (LUPU; SLOMAN, 1999). Moreover, we intend to extend the policy authoring framework to support more terms, expressions, prescriptions, and rules, also expressions supporting to source and destination network elements. Further, our approach is limited to rules triggered by the occurrence of an event, *i.e.*, a flow receives a specific action. We intend to extend our grammar to support temporal logic. This will allow the specification of policies defined by an interval of time.

Finally, we intend to analyze the generality of the formalism when managing other resources and types of services, such as access control and load balancing. Thus, despite we limit the scope of our work to QoS management, with some adaptations, the toolkit can be used to perform configuration and manage services and network functions such as monitoring, access control, and firewall.

REFERENCES

AIB, I.; BOUTABA, R. On leveraging policy-based management for maximizing business profit. **IEEE Transactions on Network and Service Management**, New York, v.4, n. 3, p. 25–39, dec. 2007.

BAKSHI, K. Considerations for software defined networking (SDN): Approaches and use cases. In: IEEE AEROSPACE CONFERENCE, 2013, Big Sky, USA. **Proceedings...** Big Sky: IEEE, 2013. p. 1–9.

BANDARA, A. et al. A goal-based approach to policy refinement. In: 5TH IEEE INTERNATIONAL WORKSHOP ON POLICIES FOR DISTRIBUTED SYSTEMS AND NETWORKS, 2004, Washington, USA. **Proceedings...** Washington: IEEE, 2004. v. 5, p. 229 – 239.

BANDARA, A. et al. Policy refinement for diffserv quality of service management. In: 9TH IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT, 2005, Nice-Acropolis, France. **Proceedings...** Nice-Acropolis: IEEE, 2005. v. 9, p. 469–482.

BANDARA, A. K.; LUPU, E. C.; RUSSO, A. Using event calculus to formalise policy specification and analysis. In: 4TH IEEE INTERNATIONAL WORKSHOP ON POLICIES FOR DISTRIBUTED SYSTEMS AND NETWORKS, 2003, Washington, USA. **Proceedings...** Washington: IEEE, 2003. v. 4, p. 26–39.

BETTS, M. et al. SDN architecture. **Open Networking Foundation**, Palo Alto, USA, v.1, n. 2, p. 1–68, jun. 2014. Available at: <https://www.opennetworking.org/sdn-resources/technical-library>. Accessed: jan. 21. 2015.

BLAKE, S. et al. An architecture for differentiated services (RFC 2475). **IETF Request for Comments - Network Working Group**, Morrisville, USA, v.1, n. 1, p. 1–36, dec. 1998. Available at: <https://www.ietf.org/rfc/rfc2475.txt>. Accessed: aug. 1. 2014.

BRADEN, R.; CLARK, D.; SHENKER, S. Integrated services in the internet architecture: an overview (RFC 1633). **IETF Request for Comments - Network Working Group**, Marina del Rey, USA, v.1, n. 1, p. 1–33, jun. 1994. Available at: <https://tools.ietf.org/html/rfc1633>. Accessed: nov. 13. 2014.

BRODIE, C. et al. The coalition policy management portal for policy authoring, verification, and deployment. In: IEEE WORKSHOP ON POLICIES FOR DISTRIBUTED SYSTEMS AND NETWORKS, 2008, Palisades, USA. **Proceedings...** Palisades: IEEE, 2008. p. 247–249.

CAREY, K.; WADE, V. Using automated policy refinement to manage adaptive composite services. In: 5TH IEEE INTERNATIONAL WORKSHOP ON POLICIES FOR DISTRIBUTED SYSTEMS AND NETWORKS, 2008, Salvador, Brazil. **Proceedings...** Salvador: IEEE, 2008. p. 239–247.

CRAVEN, R. et al. Decomposition techniques for policy refinement. In: IEEE INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT, 2010, Niagara Falls, Canada. **Proceedings...** Niagara Falls: IEEE, 2010. p. 72–79.

CRAVEN, R. et al. Policy refinement: Decomposition and operationalization for dynamic domains. In: 7TH IEEE INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT, 2011, Paris, France. **Proceedings...** Paris: IEEE, 2011. v. 7, p. 1–9.

DAVIS, N. et al. Software-defined networking: The new norm for networks. **Open Networking Foundation**, Palo Alto, USA, v.2, n. 4, p. 1–68, nov. 2012. Available at: <https://www.opennetworking.org/sdn-resources/technical-library>. Accessed: jan. 12. 2015.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. **Numerische mathematik**, New York, v.1, n. 1, p. 269–271, 1959.

FITO, J. O. et al. Business-driven it management for cloud computing providers. In: 4TH IEEE INTERNATIONAL CONFERENCE ON CLOUD COMPUTING TECHNOLOGY AND SCIENCE, 2012, Taipei, China. **Proceedings...** Taipei: IEEE, 2012. v. 4, p. 193–200.

FOSTER, N. et al. Frenetic: a network programming language. **ACM SIGPLAN Notice**, New York, v.46, n. 9, p. 279–291, sep. 2011.

HAN, W.; LEI, C. A survey on policy languages in network and security management. **IEEE Computer Networks**, Bridgewater, v.56, n. 1, p. 477–489, jan. 2012.

HYOJOON, K.; FEAMSTER, N. Improving network management with software defined networking. **IEEE Communications Magazine**, Bridgewater, v.51, n. 2, p. 114–119, feb. 2013.

JOHNSON, M. et al. Optimizing a policy authoring framework for security and privacy policies. In: SIXTH SYMPOSIUM ON USABLE PRIVACY AND SECURITY, 2010, Redmond, Washington. **Proceedings...** New York: ACM, 2010. v. 6, p. 1–9.

KARN, P.; PARTRIDGE, C. Improving round-trip time estimates in reliable transport protocols. **ACM SIGCOMM Computer Communication Review**, Washington, v. 17, n. 5, p. 2–7, aug. 1987.

KEEPENCE, B. Quality of service for voice over ip. In: IEEE COLLOQUIUM ON SERVICES OVER THE INTERNET - WHAT DOES QUALITY COST?, 1999, London, Canada. **Proceedings...** London: IET, 1999. p. 1–4.

KOWALSKI, R.; SERGOT, M. A logic-based calculus of events. **New Generation Computing**, Ohmsha, v.4, n. 1, p. 67–95, jan. 1986.

LEISERSON, C. Fat-trees: Universal networks for hardware-efficient supercomputing. **IEEE Transactions on Computers**, Washington, v.34, n. 10, p. 892–901, oct. 1985.

LUPU, E. C.; SLOMAN, M. Conflicts in policy-based distributed systems management. **IEEE Trans. Softw. Eng.**, Piscataway, v.25, n. 6, p. 852–869, nov. 1999.

MCCAULEY, M. et al. POX OpenFlow controller. **NOXRepo**, New York, USA, v.1, n. 1, p. 1–43, dec. 2013. Available at: <http://www.noxrepo.org/pox/about-pox/>. Accessed: dec. 2. 2013.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **SIGCOMM Computer Communincation**, v.38, n. 2, p. 69–74, mar. 2008.

MIN, S. et al. Implementation of an openflow network virtualization for multi-controller environment. In: 14TH IEEE INTERNATIONAL CONFERENCE ON ADVANCED COMMUNICATION TECHNOLOGY, 2012, PyeongChang, South Korea. **Proceedings...** PyeongChang: IEEE, 2012. v. 14, p. 589–592.

MOFFETT, J.; SLOMAN, M. Policy hierarchies for distributed systems management. **IEEE Journal on Selected Areas in Communications**, New York, v.11, n. 9, p. 1404–1414, dec. 1993.

MONSANTO, C. et al. Composing software-defined networks. In: 10TH USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION, 2013, Berkeley, USA. **Proceedings...** Lombard: USENIX Association, 2013. v. 10, p. 1–14.

MOORE, B. et al. Policy core information model (RFC 3060). **IETF Request for Comments - Network Working Group**, New Jersey, USA, v.1, n. 1, p. 1–100, feb. 2001. Available at: <https://tools.ietf.org/html/rfc3060>. Accessed: jul. 5. 2014.

NUNES, B. et al. A survey of software-defined networking: Past, present, and future of programmable networks. **IEEE Communications Surveys Tutorials**, Bridgewater, v.16, n. 3, p. 1617–1634, feb. 2014.

PARTRIDGE, C. A Proposed Flow Specification (RFC 1363). **IETF Request for Comments - Network Working Group**, Palo Alto, USA, v.1, n. 1, p. 1–20, sep. 1992. Available at: <https://tools.ietf.org/html/rfc1363>. Accessed: sep. 7. 2014.

PFAFF, B. et al. OpenFlow Switch Specification - Version 1.0.0 (Wire Protocol 0x01). **Open Networking Foundation**, New York, USA, v.1, n. 1, p. 1–42, dec. 2009. Available at: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf>. Accessed: jan. 21. 2015.

POSTEL, J. Internet control message protocol (RFC 792). **IETF Request for Comments - Network Working Group**, New York, USA, v.1, n. 1, p. 1–21, sep. 1981. Available at: <https://tools.ietf.org/html/rfc792>. Accessed: apr. 12. 2015.

PUESCHEL, T.; PUTZKE, F.; NEUMANN, D. Revenue management for cloud providers–a policy-based approach under stochastic demand. In: 45TH IEEE HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCE, 2012, Maui, Hawaii. **Proceedings...** Maui: IEEE, 2012. v. 45, p. 1583–1592.

PYTHON, M. Python language reference. **Python Software Foundation**, New York, USA, v.2, n. 7, p. 1–50, dec. 2014. Available at: <http://www.python.org>. Accessed: dec. 21. 2014.

REEDER, R. W. et al. Usability challenges in security and privacy policy-authoring interfaces. In: 11TH IFIP TC 13 INTERNATIONAL CONFERENCE ON HUMAN-COMPUTER INTERACTION, 2007, Rio de Janeiro, Brazil. **Proceedings...** Berlin, Heidelberg: Springer-Verlag, 2007. v. 11, p. 141–155.

REYNOLDS, J. K.; POSTEL, J. Assigned numbers (RFC 1700). **IETF Request for Comments - Network Working Group**, Marina del Rey, USA, v.20, n. 1, p. 1–230, oct. 1994. Available at: <https://www.ietf.org/rfc/rfc1700.txt>. Accessed: may. 25. 2014.

RUBIO-LOYOLA, J. et al. Scalable service deployment on software-defined networks. **IEEE Communications Magazine**, New York, v.49, n. 12, p. 84–93, dec. 2011.

SEZER, S. et al. Are we ready for SDN? implementation challenges for software-defined networks. **IEEE Communications Magazine**, Bridgewater, v.51, n. 7, p. 36–43, jul. 2013.

SHANAHAN, M. An abductive event calculus planner. **The Journal of Logic Programming**, v.44, n. 1, p. 207–240, jul. 2000.

SQUICCIARINI, A. C.; PETRACCA, G.; BERTINO, E. Adaptive data management for self-protecting objects in cloud computing systems. In: 8TH IEEE INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT, 2012, Laxenburg, Austria. **Proceedings...** Laxenburg: IEEE, 2012. v. 12, p. 140–144.

VERMA, D. Simplifying network administration using policy-based management. **IEEE Network**, Bridgewater, v.16, n. 2, p. 20–26, mar. 2002.

VILLEGAS, D. et al. An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds. In: 12TH IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD AND GRID COMPUTING, 2012, Ottawa, Canada. **Proceedings...** Ottawa: IEEE/ACM, 2012. v. 12, p. 612–619.

WALLER, A. et al. A functional solution for goal-oriented policy refinement. In: 7TH IEEE INTERNATIONAL WORKSHOP ON POLICIES FOR DISTRIBUTED SYSTEMS AND NETWORKS, 2006, London, Canada. **Proceedings...** London: IEEE, 2006. v. 7, p. 133–144.

WALLER, A. et al. Policy based management for security in cloud computing. In: SECURE AND TRUST COMPUTING, DATA MANAGEMENT, AND APPLICATIONS, 2011, Loutraki, Greece. **Proceedings...** Loutraki: Springer Berlin Heidelberg, 2011. v. 187, p. 130–137.

WESTERINEN, A. et al. Terminology for policy-based management (RFC 3198). **IETF Request for Comments - Network Working Group**, Southborough, USA, v.1, n. 1, p. 1–21, nov. 2001. Available at: <https://www.ietf.org/rfc/rfc3198.txt>. Accessed: may. 14. 2014.

WICKBOLDT, J. et al. Software-defined networking: management requirements and challenges. **IEEE Communications Magazine**, Bridgewater, v.53, n. 1, p. 278–285, jan. 2015.

ZHAO, L.; SAKR, S.; LIU, A. A framework for consumer-centric SLA management of cloud-hosted databases. **IEEE Transactions on Services Computing**, v.43, n. 9, p. 23–28, 2013.

AppendixA PUBLISHED PAPER – AINA 2014

This paper presented the first definitions towards our policy refinement toolkit. This research focused on customizing an OpenFlow controller to obtain policies from a repository. This approach was based on an initial manual process performed by an administrator, followed by an automatic policy refinement process of flow rules executed by an OpenFlow controller. The low-level policies are understood and enforced by network devices for automatic reconfiguration and optimization of network resource usage at runtime. As a result, the proposed approach was able to identify properties and characteristics of applications that require QoS.

- **Title –**
  Towards SLA Policy Refinement for QoS Management in Software-Defined Networking
- **Conference –**
  The 28th IEEE International Conference on Advanced Information Networking and Applications (AINA-2014)
- **Type –**
  Main track (full-paper)
- **Qualis –**
  A2
- **URL –**
  <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6838692>
- **Date –**
  May 13-16, 2014
- **Held at –**
  Victoria, Canada
- **Digital Object Identifier (DOI) –**
  <http://dx.doi.org/10.1109/AINA.2014.148>

Towards SLA Policy Refinement for QoS Management in Software-Defined Networking

Cristian Cleder Machado, Lisandro Zambenedetti Granville, Alberto Schaeffer-Filho, Juliano Araujo Wickboldt

Computer Networks Group
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
Email: {ccmachado, granville, alberto, jwickboldt}@inf.ufrgs.br

***Abstract*— *Software-defined networking* (SDN) is a dynamic, adaptable, controllable and flexible network architecture. It provides an extensible platform for delivery of network services, capable of responding quickly to service requirement changes. As a result, SDN has become a suitable scenario for the application of techniques and approaches for improved infrastructure management, such as *policy-based management* (PBM). In PBM, using techniques such as refinement, a high-level policy – *e.g.*, specified as a *service level agreement* (SLA) – can be translated into a set of corresponding low-level rules, enforceable in various elements of a system. However, when using SLAs, their translation to low-level policies, e.g., for controller configuration, is not straightforward. If this translation is not done properly, the controller may not be able to meet the implicit requirements of the SLA, failing to satisfy the goals described in the high-level policy. This paper proposes a novel approach towards SLA policy refinement for *quality of service* (QoS) management (based on routing) in *software-defined networking*. It consists of an initial manual process performed by an administrator, followed by an automatic policy refinement process executed by an OpenFlow controller. As a result, our approach is capable of identifying the requirements and resources that need to be configured in accordance with SLA refinement, and can successfully configure and execute reactive dynamic actions for supporting dynamic infrastructure reconfiguration.**

Keywords*—*refinement; policy; management; sdn;

I. Introduction

For many years, organizations have been employing management strategies for dealing with the scale and computational complexity of their Information and Communications Technology (ICT) infrastructures [1]. *Software-defined networking* (SDN) is a dynamic, adaptable, controllable and flexible network architecture. It provides an extensible platform for delivery of network services, capable of responding quickly to service requirement changes. SDN facilitates network operations based on the idea of controlling and programming the behavior of network devices, where the rules for packet forwarding can be controlled by software applications developed independently from the hardware [2]. SDN enables the deployment of network applications that perform sophisticated traffic monitoring and traffic processing. This allows an effective way for providing dynamically – and at runtime – services that support, for example, quality of service (QoS) reconfiguration, access control and load balancing.

SDN is characterized by a centralized control plane, which allows moving part of the decision-making logic of network devices to external controllers [3]. This characteristic provides the controller device with the ability to have an overall view of the network infrastructure, thus becoming aware of network elements and network characteristics. As a result, SDN has become a suitable scenario for the application of techniques and approaches for improved infrastructure management, such as *policy-based management* (PBM). Policies can be analyzed and modified as necessary at different levels of abstraction, without changing the functionality implemented in network devices. The behavior of certain features can be modified without changing the code or the implementation of the system, or even, without manual intervention. In addition, at runtime, policy rules can be triggered by conditions monitored in the network, thus providing support for the adaptive reconfiguration of devices enforcing these rules.

Several approaches [4], [5], [6], [7], [8], [9], [10] have been using policies to introduce control rules that govern the operation of SDN. Most of these approaches use rules created directly in a low-level notation, and the rules are introduced straight into the controller. However, when using *service level agreements* (SLAs), their translation to low-level policies, *e.g.*, for controller configuration, is not straightforward. If this translation is not done properly, the controller may not be able to meet the implicit requirements of the SLA, thus failing to satisfy business level goals. To address this issue, our goal is to develop techniques for *policy refinement*, where a high-level policy specification can be translated into a set of corresponding low-level policies, which are deployed in various elements of a network [11]. Thus, controllers can be configured with the specific objectives, written in a low-level language (device rules), that accurately satisfy the high-level policy goals. The use of PBM and especially policy refinement has been historically investigated [12]. Although there have been some promising developments in the area of policy analysis, policy refinement is a nontrivial process and it remains to a large extent a much neglected research area.

This paper proposes a novel approach to SLA policy refinement for QoS management (based on routing) in SDN. Our approach is based on an initial manual process performed by an administrator, followed by an automatic policy refinement process executed by an OpenFlow controller. Therefore, our proposed approach is able to identify properties and characteristics of applications that require QoS. This is accomplished by following the specifications of the high-level SLA policies, and refining them into low-level policies. These low-level policies are understood and enforced by network devices for automatic reconfiguration and optimization of network resource usage at runtime. The controller was designed using POX [13], written in Python Language [14], and uses the OpenFlow protocol [15] to coordinate network devices. Experiments have been performed using the Mininet emulator [16].

To the best of our knowledge, this is the first time that policy refinement has been applied to SDNs. The development of a policy refinement strategy specifically for SDNs can benefit from several aspects found in these environments. Firstly, it is possible to more easily collect monitoring information and network traffic data, with the aim of checking whether high-level goals are being fulfilled by low-level rules or not. Secondly, we have the ability to refine policies to lower level ones, which can be (re)implemented in all network device, since the SDN controller uses a standard protocol and interface to determine rules and actions on any network device.

The paper is organized as follows. Section II provides a brief description of the main concepts used in our approach. Section III presents an overview of our policy refinement approach. Section IV presents the operation of the controller. Section V describes the scenarios, experiments and initial results. In Section VI some studies related to our approach are introduced and discussed. Section VII concludes the paper with final remarks, along with a proposal for future work.

II. Background

This section provides an overview of the main concepts employed in our approach. Section II-A briefly presents software-defined networking (SDN) and OpenFlow. Section II-B explains the notions of *policy-based management* (PBM) and policy refinement. Finally, Section II-C describes properties and requirements for quality of service (QoS).

A. Software-Defined Networking (SDN) and OpenFlow

Software-defined networking is a dynamic, adaptable, controllable and flexible network architecture. It provides an extensible platform for delivery of network services, capable of responding quickly to service requirement changes [2]. SDN is characterized by a logically centralized control plane, which allows moving part of the decision-making logic of network devices to external controllers (Figure 1). This provides controller devices with the ability to have an overall view of the network and its resources, thus becoming aware of all the network elements and their characteristics. Based on this centralization, network devices become simple packet forwarding elements, which can be programmed through an open interface, such as the OpenFlow protocol [2].

Briefly, the main elements of an OpenFlow-based SDN architecture are (depicted in Figure 1): (*i*) a flow table that contains an input and a specific action to be performed for each active flow, and (*ii*) an abstraction layer that communicates securely with a controller reporting on new input flows that are not present in the flow table. Each entry in the flow table consists of: *(a)* a mask of fields found in the packet header, which is used to match the incoming packets, *(b)* counters for collecting statistics for each specific flow, such as number of bytes, number of packets received, and flow duration, and *(c)* a series of actions to be performed when a packet matches the corresponding mask [17].

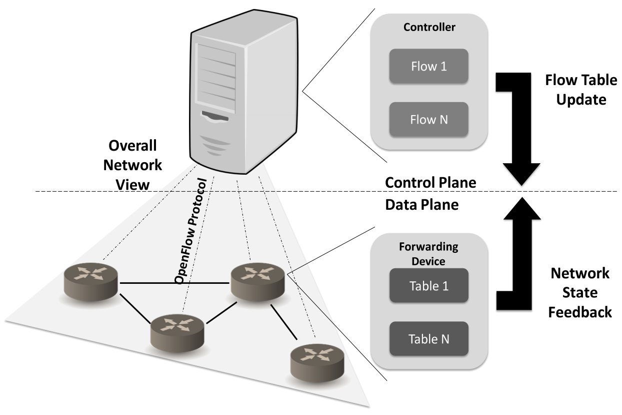


Fig. 1. OpenFlow Architecture.

OpenFlow is an open protocol that allows the development of programmable mechanisms based on a flow table in different forwarding devices. The OpenFlow protocol establishes a secure communication channel between OpenFlow switches and the controller, using this channel for controlling and establishing flows according to customizable programs [15].

The OpenFlow controller is the central element of the approach. The controller runs customized programs to decide which rules and actions are going to be installed to control packet forwarding in each switch element. OpenFlow is also characterized by the separation between the data plane and the control plane. On the one hand, the data plane is concerned with the forwarding of packets based on rules, called OpenFlow actions, associated with each table entry in the switch. On the other hand, the control plane enables the controller to manage the entries in the flow table and the rules associated with the desired traffic [17].

B. Policy-Based Management (PBM) and Policy Refinement

Policies are defined as a collection of rules which express and enforce the required behavior of a resource. RFC 3198 [18] provides the following definitions for a policy:

- A defined goal or action that determines how present and future decisions are taken. Policies are established or executed within a particular context;
- Policies refer to a set of rules to manage and monitor access to features of a particular ICT infrastructure [19].

In PBM an administrator specifies the infrastructure objectives/goals and constraints in the form of rules to guide the behavior of the elements in a system [20]. The use of PBM presents three main benefits [21]. Firstly, policies are predefined by administrators and stored in a repository. When an event occurs, these policies are requested and accessed automatically, without the need of manual intervention. Secondly, the formal description of policies permits automated analysis and verification with the aim of guaranteeing consistency to some extent. Thirdly, because of the abstraction of technical details, policies can be inspected and changed dynamically at runtime without modifying the underlying system implementation.

Policies may be seen in two principal levels of abstraction: *low-level policies*, which are related to a domain or a device, and *high-level policies* that are more user-friendly. A simple example of a low-level policy is the settings on routers so multimedia traffic packets have higher priority over peer-to-peer (P2P) [22] traffic packets. An example of high-level policies are SLAs [23]. Policy refinement aims to translate a high-level policy into a set of corresponding low-level policies. In other words, using techniques for refinement, a high-level policy such as a *service level agreement* (SLA) can be translated into low-level policies that are applicable in various elements of a system [24], [25].

SLAs are generally business-oriented, and they leave aside the technical details, which are guided by a *service level specification* (SLS) and a *service level objective* (SLO). The SLS is a technical interpretation of the SLA. The SLO is a sub-item of the SLS that contains the parameters to achieve the SLS [26].

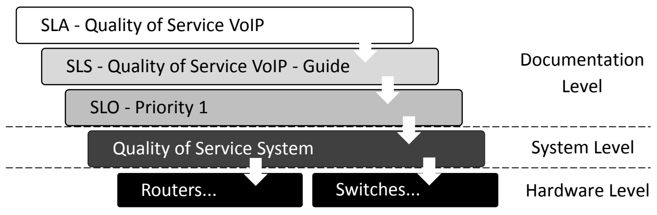

SLA - Quality of Service VoIP
SLS - Quality of Service VoIP - Guide
SLO - Priority 1
Documentation Level
Quality of Service System
System Level
Routers...
Switches...
Hardware Level

Fig. 2. Example policy refinement.

Figure 2 presents an overview of the process of refining SLAs. The SLA, SLS and SLO represent, respectively, the *documentation* describing the service in a formal way, the technical form (guide) for its functioning requirements, and the parameters aimed at quality and satisfaction. At the *system level*, a *quality of service* system interprets the management requirements, and enforces policies for the configuration of elements at the *hardware level*.

The main objectives of policy refinement are identified by Moffett and Sloman [25] as:

- To determine what resources are needed to fulfil policy needs;
- To translate the high-level policy into a set of operational policies that the system can enforce;
- To examine whether the low-level policies actually meet precisely the requirements specified by the high-level policy.

The refinement process typically involves stages of decomposition, operationalization, implementation, operation and re-refinement of goals and subgoals [27], [28]. Policy refinement aims to automate these stages to get the translation of policies relating to objects and implementable actions, and ensure that the low-level policies still satisfy the goals defined by the high-level policy.

C. Quality of Service (QoS)

Quality of service refers to a combination of many properties or features of a service, such as:

- *Availability*: related to the percentage of time that a service is operating;
- *Security*: includes the presence and types of authentication mechanisms, data confidentiality and data integrity, non-repudiation of messages, and resilience to denial-of-service attacks [29];
- *Response time*: time required for a service to respond to individual types of requests;
- *Throughput*: rate at which service requests are processed. QoS measurements include full capacity or a ratio that describes how the throughput is changed according to load [30].

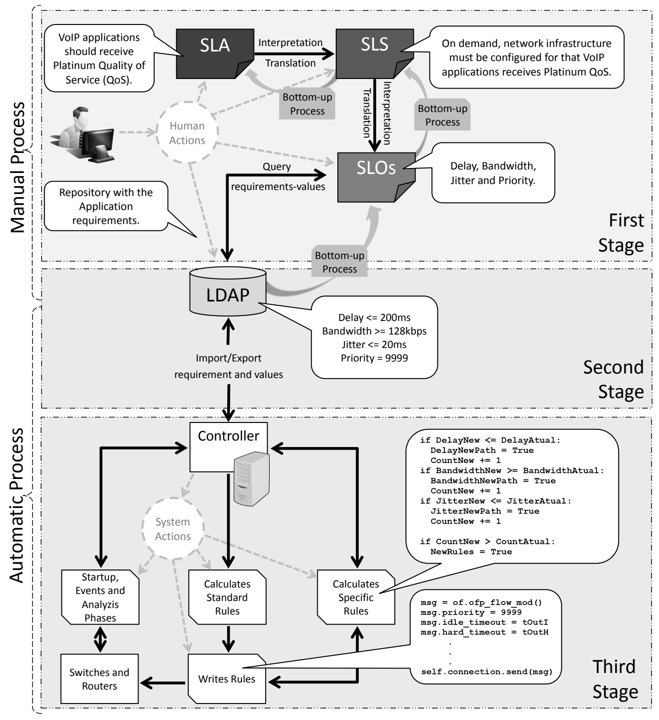
In computer networks, the term *quality of service* [31] is related to a set of standards and mechanisms for ensuring high performance for critical applications. Through the use of QoS mechanisms, network administrators can use existing resources efficiently and thus ensure the required level of service without the need to expand or over-provision their networks.

III. Policy Refinement Approach

In this section we present our SLA policy refinement model, and illustrate it with a case-study. We limit ourselves to a Voice over IP (VoIP) QoS scenario for better presentation of the approach, but our approach is generic and can be used for various applications, such as video streaming, replication, and backup. Similarly, it can be applied to various services, such as monitoring, access control, and load balancing.

Figure 3 shows an overview of our approach for SLA policy refinement. We assume as input an SLA that has a high-level specification and determines which guarantees must be fulfilled for the optimal operation of the services. Currently, the refinement process includes a significant amount of manual labor, in which case it requires an administrator to define the initial rules, interpreting and translating each policy level, and identifying the objectives to be fulfilled at the level below, *i.e.*., SLA $\rightarrow$ SLS and SLS $\rightarrow$ SLOs (*top-down process*). The actions and objectives described in lower-levels should be verified in order to determine whether they are faithfully related to the levels above (*bottom-up process*). In addition, policies can be (re)evaluated based on the feedback provided by the network controller. Finally, the concrete and executable policies translated from the high-level policies are stored in a repository that will serve as input for the implementation of system elements.

The refinement process comprises three stages, which are represented in Figure 3 (the manual and the automatic processes are also indicated in the diagram). In the *first stage*, a high-level policy (SLA) is interpreted and translated manually


VoIP applications should receive Platinum Quality of Service (QoS).
SLA
Interpretation
Translation
SLS
On demand, network infrastructure must be configured for that VoIP applications receives Platinum QoS.
Bottom-up Process
Interpretation
Translation
Bottom-up Process
Human Actions
Query
requirements-values
SLOs
Delay, Bandwidth, Jitter and Priority.
Repository with the Application requirements.
First Stage
Manual Process
Bottom-up Process
LDAP
Delay <= 200ms
Bandwidth >= 128kbps
Jitter <= 20ms
Priority = 9999
Import/Export requirement and values
Second Stage
Controller
System Actions
Startup, Events and Analyzis Phases
Calculates Standard Rules
Calculates Specific Rules
Switches and Routers
Writes Rules
Third Stage
Automatic Process

```
if DelayNew <= DelayAtual:
  DelayNewPath = True
  CountNew += 1
if BandwidthNew >= BandwidthAtual:
  BandwidthNewPath = True
  CountNew += 1
if JitterNew <= JitterAtual:
  JitterNewPath = True
  CountNew += 1

if CountNew > CountAtual:
  NewRules = True
```

```
msg = of.ofp_flow_mod()
msg.priority = 9999
msg.idle_timeout = tOutI
msg.hard_timeout = tOutH
      .
      .
      .
self.connection.send(msg)
```

Fig. 3. Refinement of QoS VoIP SLAs.

by an administrator so that all its technical requirements can be defined. In our case-study, the SLA specifies that *"VoIP applications should receive Platinum quality of service (QoS)"*. The interpretation and translation of the SLA result in the technical SLS: *"On demand, the network infrastructure must be configured so that VoIP applications receives Platinum QoS."*. Next, the interpretation and translation of the SLS are performed in order to analyze the possible objectives (SLOs) that satisfy VoIP Platinum QoS. The value of each objective is obtained by querying a specific QoS class (in our case Platinum QoS) in an LDAP repository [32]. We assume that this repository has been previously populated by an administrator with *(i)* the set of QoS classes (*e.g.*, Platinum, Gold, and Silver), *(ii)* class requirements (*e.g.*, Delay, Jitter, and Bandwidth) *(iii)* and the requirement values (*e.g.*, 200ms, 20ms, and 128kbps). Through a *bottom-up process* we can verify that the *"Delay $\leq$ 200ms"* is the value of the requirement *"SLO - Delay"* for services that require QoS Platinum class. In addition to QoS classes, the LDAP repository also contains a protocol list (*e.g.*, FTP, HTTP, SIP, RTP, and RTSP). This list is also previously supplied by the administrator, using RFC 1700 [33] recommendations and service analysis that will run in the infrastructure, along with the ports and protocols that are not in RFC 1700. The information in this list will be used later, in the Events Phase and Analysis Phase (see Section IV).

In the *second stage*, the administrator associates the VoIP service with a specific protocol (the list of protocols can have multiple entries because there might be VoIP services using different protocols). For our tests we set the protocol H.323 [34] in Platinum class registered in the LDAP repository. This association is performed to define which protocols receive Platinum QoS in the controller.

In the *third stage*, the controller loads the rules of the repository, filtered by all classes of QoS with all requirements, values and associated protocols, storing the concrete rules in a policy dictionary. Next, for each new network flow, the controller performs an analysis phase to enforce the QoS requirements for the application. It verifies each requirement in order to identify the best path in the network, which priority should be given and what network elements should receive the rules. To support dynamic reconfigurations, for each valid change in the *second stage*, an identifier (*Serial #*) is incremented to allow version control of the repository. The network controller is configured to periodically read the repository (*e.g.*, every *1 minute*), and if the *Serial #* previously loaded into the system is equal to the one in the repository, nothing needs to change. However, if the *Serial #* has increased, the controller reloads the services and requirements and applies the new rules.

In order to apply our approach there was the need to customize some functionality in the SDN controller. This was required for collecting information about the network infrastructure, which is later used, for example, to calculate optimal routes. This customization was based on SDN native features only, and thus can be applied to any controller implementation. Therefore, our approach is not tied to any specific controller design or language. For example, topology discovery, which is available in a POX controller from the discovery module, is a native feature of SDN achieved by all controllers in different implementations.

The advantage of our approach, compared to previous investigations of policy refinement in non-SDN deployments, is that a controller holds updated information about the domain and the elements that should be considered for deployment of the concrete low-level policies. From this, the administrator has an overview of the current status of the domain, and can recognize the limits that may affect each SLA beforehand. We plan to develop a Graphical User Interface (GUI) that decomposes policies in order to achieve low-level policies with refinement techniques. Thus, infrastructure operators may introduce SLAs for automatic decomposition. Subsequently, developers of controllers can interpret each low-level policy generated with a system description that indicates the rules that must be applied by the controller for execution.

IV. Controller Prototype

This section presents the operation of the controller, which is divided into three phases (Figure 4): *(i) Startup Phase*: discovers services and possible paths between network elements; *(ii) Events Phase*: identifies service events and determines the best path based on the characteristics of the network and service requirements; and *(iii) Analysis Phase*: implements the rules and monitors the network in order to identify possible enhancements for the active flows and reconfigure the structure. The operation of each phase is detailed in the following.

A. Startup Phase

In this phase, the controller reads information from a repository that contains descriptions and requirements of all services that are initially scheduled to run in the network. This information is identified and set by the administrator

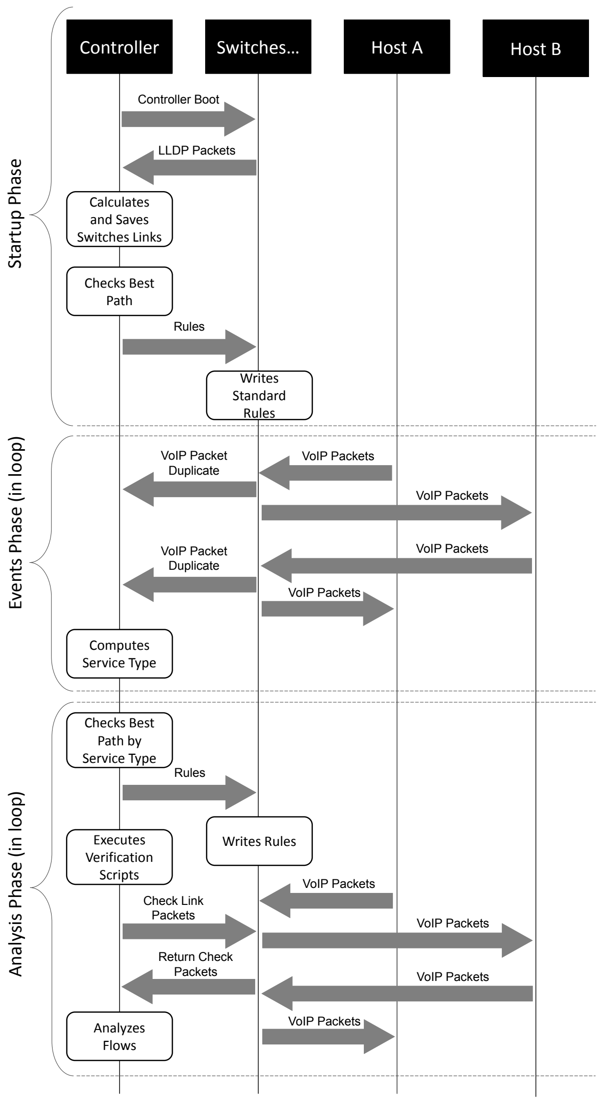


Fig. 4. Flow Diagram VoIP.

as requirements to perform the low-level policies in order to fulfill the SLA definitions. Thereafter, the controller monitors the network in order to discover the devices and topology. Network devices are instructed to send Link Layer Discovery Protocol (LLDP) [35] packets to report their location on the topology, which is stored by the controller in an internal data structure.

While the LLDP packets are received and stored, a routine for calculating paths between all elements is performed using Dijkstra's Shortest Path Algorithm [36]. We assume at first that the best path is the one with the smallest number of hops, because at this stage values such as delay and jitter are unknown, thus we consider that these values are initially zero. Subsequently, all paths are sorted from the shortest to the longest path and stored into a list.

Once all possible paths between network elements have been calculated, the controller writes rules (which we call *standard rules*) in the flow table of the switches that are in the best path between each of the network elements. The purpose of these rules is to handle each new service flow in such a way that its first packet is duplicated, *(i)* forwarding one copy of the packet to the controller in order to inform it that there is a new service flow, which will be further evaluated (see Section IV-C), but also *(ii)* handling the duplicate packet with the idea of best-effort network and no-act delay Round-Trip Time (RTT) [37]. After setting up the rules in the switches, the controller inserts in the repository all the update network information collected at this stage.

B. Events Phase

This phase aims to identify events in the network infrastructure. At the moment, *(i) Dataflow Event* is the only event handled by our controller, but we may extend it to handle events such as *(ii) New Device Addition to the Topology* and *(iii) Dropped Communication Link*. *Dataflow Events* are generated by a new type of service in the network (*e.g.*, a video streaming), and are stored in a services dictionary.

The *Events Phase* stays in a loop during the operation of the infrastructure. When running a specific protocol, such as VoIP communication, the first few packets of the communication are initially treated by the standard rules and switches are instructed to duplicate the first packet of each new service flow (*e.g.*, between host A and host B), sending a copy to the controller. The controller stores in a list the protocols that are running in a link, which are identified by a function that reads the *IP Packet Header* and gets the information from the *Protocol Field*. Then, the controller checks what are the necessary requirements for the proper functioning of such protocol through the information loaded from the repository and decides whether the controller should start the *Analysis Phase* (see Section IV-C). The *Analysis Phase* is initiated immediately if the protocol of the new flow is not yet included in the list of protocols for the link, or if some other flow is already sharing the same path but using a different protocol.

C. Analysis Phase

After identifying the service requirements, the controller calculates the new rules for best path, taking into account the specific weights of the service. The calculations are carried for the paths using as weights the bandwidth (BW), throughput (T), delay (D), jitter (J), loss rate (LR), and number of hops (NH) in each link of the physical topology [38]. *E.g.*, for VoIP we define the requirements [39] for calculation of the best link as an amount of bandwidth that varies according to the encoding used, low delay and low jitter, and priority (P). We check the value of each of these requirements using an analysis function that sends Internet Control Message Protocol (ICMP) [40] packets and store the return value in a state vector of links. When calculating the jitter, we store a vector with the 30 last values and calculate the average. For the purposes of our experiments, the requirements of VoIP SLA goals were considered as $D \leq 200ms$, $BW \geq 128kbps$, $J \leq 20ms$, and $P = 9999$. Each requirement/value pair is presented in order of importance. The link that satisfies the largest number of requirements with priority numbered from left to right is chosen as the best path.

These new rules are reconfigured at runtime and only on switches that have the flow, aiming to reduce processing overhead where there is no need for such rules. Unlike the standard rules, these *specific rules* are configured with a timeout in the flow table of each element. In our experiments, we set the value of the timeout to *15s* (but it can be easily adjusted based on the analysis of the services being executed).

Periodically, the controller checks if the the configured links remain the best choices for the current flow. In our experiments, we set the checking intervals to *10s* (but these can also be easily adjusted). If at any time the controller identifies that there is a better alternative path, new rules are sent to the switches in order to process the flow as efficiently as possible. If the current path remains the best, the controller only increases the value of the timeout for the rules on each switch for the corresponding flow. This phase stays in a loop during the operation of the infrastructure.

V. Scenarios, Experiments and Initial Results

This section describes our test environment and some initial results. The experiments were performed on an Intel 2.4 GHz QuadCore processor with 6 GB RAM memory. The scenarios were created using the Mininet emulator. Five scenarios were created in our experiments. An overview of scenarios A, B, and C is shown in Figure 5. Due to the large number of elements, scenarios D and E could not be clearly presented in a figure, but the elements that comprise each scenario are described in Table I.

A. Scenarios

The scenarios that we built had increasing numbers of switches and redundant links, thus increasing path diversity between any two hosts. Table I shows the number of hosts, switches and links in each scenario.

TABLE I. Number of hosts, switches and links in each scenario

Scenario	A	B	C	D	E
Hosts	4	8	16	32	64
Switches L0	2	4	8	16	32
Switches L1	2	4	8	16	32
Switches L2	0	2	4	8	16
Switches L3	0	0	0	4	8
Links	4	12	32	80	96
Total number of switches	4	10	20	44	88

The topologies used in the experiments were based on Fat-Tree topologies [41] (depicted in Figure 5). By increasing the number of switches and redundant links, these topologies can be used to, for example, maintain system availability in the presence of problematic links and to reduce traffic congestion in the infrastructure.

B. Experiment 1 - Standard Rules

Experiment 1 measures the time spent on the initial calculation of all paths between switches and the establishment of standard rules on all switches in the network. Figure 6 shows

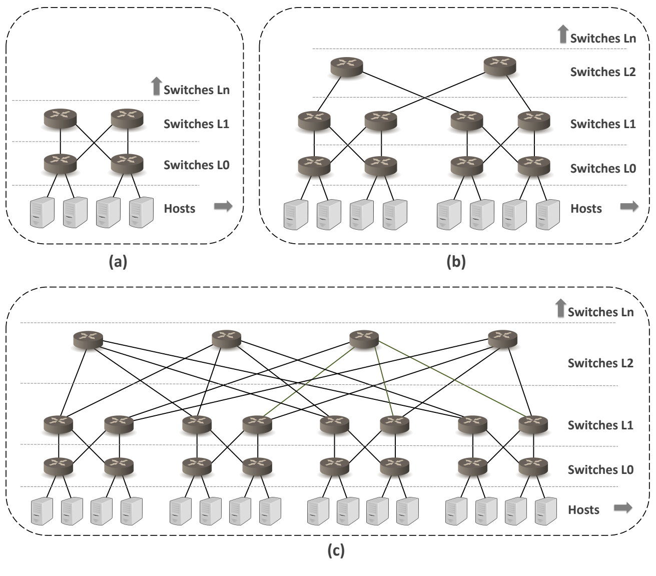


Fig. 5. Scenarios for the experiments with increasing number of switches and link redundancy.

the time for calculation and installation of the standard rules during the *Startup Phase* of the controller.

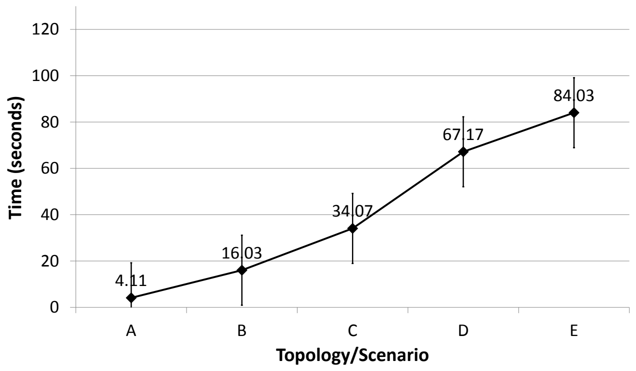


Fig. 6. Time for the calculation and installation of standard rules in the switches in the startup phase and recognition of topology in each experiment.

As can be observed in Figure 6, with the increase in the number of elements and links in the topology, the time to calculate all paths between any network element increases. Even with this growth, we justify the execution of this calculation and installation of these initial standard rules because we want avoid any delay when new flows arrive. We also take into consideration that, due to the initial calculation, later, in the *Analysis Phase* (see IV-C), we have already all possible flow paths, which makes the calculation of specific rules faster (as can be seen in Figure 7).

C. Experiment 2 - Specific Rules

Experiment 2 measures the time spent on calculating and installing specific rules when a service is identified. The scenarios used in Experiment 2 are the same as previously described. Figure 7 shows the calculation and installation time of specific rules for the identified service.

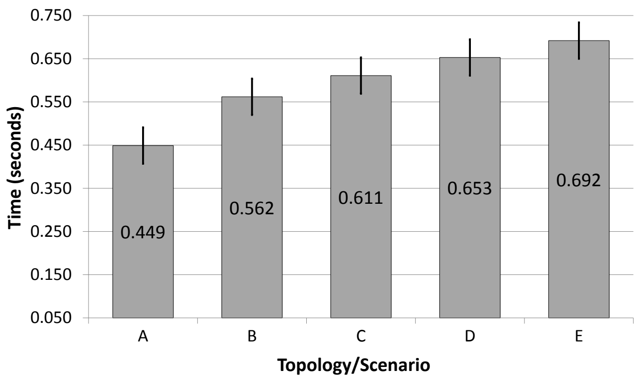


Fig. 7. Time for calculation and installation of specific rules in the switches in each experiment.

The increase in the number of elements in each topology is reflected in an increase in the time for calculation and installation of the rules. There is a minimum time necessary for calculating the specific rules, but that does not delay service flow processing due to the pre-established standard rules.

These results show that due to the calculations performed initially, our proposal is able to perform a quick reconfiguration of specific rules, since we already have a populated list of the best links between switches. As part of our future work, we intend to run more experiments that generate background traffic flows competing with other applications and validate dynamic reconfigurations in our approach.

VI. Related Work

Rubio-Loyola *et al.* [5] have investigated the sharing of virtualized network resources in software-defined networking. It shows that the ability to program network elements helps to dynamically adapt the network to both predictable and unpredictable changes. The authors present an orchestration plane (OP) which aims to manage the system behavior in response to context changes, and in accordance with business goals and policies. However, the authors do not specify if there is any refinement approach to translate policies in the different abstraction levels.

Regarding policy refinement, Bandara *et al.* [12] have presented the use of goal design and abductive reasoning to derive strategies that attain a specific high-level goal. Policies can be refined by combining strategies with events and restrictions. The authors provide tool support for the refinement process, and use examples of DiffServ [23] QoS management. The refinement is built on a systematic approach, making strategies derived for the lower levels satisfy the requirements of a high-level policy. As part of our future work, we intend to use similar techniques for improving our refinement process, and explore the dynamic aspects of software-defined networking.

VII. Conclusions and Future Work

QoS mechanisms allow network administrators to use existing resources efficiently and ensure the required level of service without the need of expanding or over provisioning their networks. However, to ensure that QoS requirements are satisfied across the network is difficult, as network devices such as switches and routers are heterogeneous and have proprietary interfaces. Moreover, QoS architectures such as DiffServ [23] and IntServ [42] are built over current networks. These are based on distributed hop-by-hop routing, without a broader perception of global, network-wide capabilities. Alternatively, in SDN networks, the controller element has an overall view of the infrastructure and the services running on it.

Policy-based management in SDNs can be used to specify goals and constraints in the form of rules to guide the operation of network elements. Policy abstractions can be used not only to adapt the controlled system, but also to adjust the policies themselves, changing their behavior to better achieve the system goals. For example, a policy may add QoS control rules for network monitoring and traffic analysis. If the rate at which the network is read is too high, communication costs for monitoring will end up interfering, and generating more traffic. However, if it is excessively low, the system may not be sufficiently aware of changes, failing to fulfill specific objectives. To solve this problem, trends in behavior can be analyzed and policies adjusted dynamically to better reflect the needs of specific resources.

In this paper, we advocate the use of policy refinement techniques in SDNs. We aim to remove much of the manual workload of administrators in the configuration of network elements. In particular, we focus on the refinement of QoS requirements for different applications and services (specified in SLAs) into the configuration of controllers and switches. As a result of our approach, we identified the resources that need to be configured in accordance with the SLAs, and successfully executed reactive dynamic actions used in the reconfiguration of the infrastructure. Our experiments have shown that an initial calculation of best-effort paths between network elements can increase the performance of the network during runtime. Certainly there is an overhead related to the startup phase of the controller. However, this is justifiable as it improves the performance of the network during runtime. In a traditional SDN network, the first packet of the a new service flow is forwarded to the controller, incurring an RTT delay at the start of each flow, where there is still no rules in the flow tables of switches. Our standard rules created at the startup phase of the controller help to mitigate this problem. Our approach provides an improved flow processing strategy, by reorganizing flows upon the arrival of new service requests.

The policy refinement approach just described is still a work in progress. As part of our future work, we plan to study other case-studies to validate the broader applicability of our work. We are also going to assess how the work presented in this paper can be generalized to other types of SLA and QoS requirements. Furthermore, we are going to investigate techniques that can be used to automate the refinement process. We also intend to investigate techniques for detection and resolution of policy conflicts. Conflicts may arise due to omissions, errors or differing requirements of administrators when specifying policies. One common source of policy conflicts is the refinement process itself, during the translation of high-level goals into implementable low-level policies [43].

References

[1] J. O. Fitó, M. Macias, F. Julia, and J. Guitart, "Business-driven it management for cloud computing providers," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. IEEE, 2012, pp. 193–200.

[2] O. W. Paper, "Software-Defined Networking: The New Norm for Networks," Open Networking Foundation, Tech. Rep., April 2012.

[3] S. Sezer, S. Scott-Hayward, P. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for sdn? implementation challenges for software-defined networks," *Communications Magazine, IEEE*, vol. 51, no. 7, pp. 36–43, 2013.

[4] A. Patel, P. Ji, and T. Wang, "Qos-aware optical burst switching in openflow based software-defined optical networks," in *Optical Network Design and Modeling (ONDM), 2013 17th International Conference on*, 2013, pp. 275–280.

[5] J. Rubio-Loyola, A. Galis, A. Astorga, J. Serrat, L. Lefevre, A. Fischer, A. Paler, and H. Meer, "Scalable service deployment on software-defined networks," *Communications Magazine, IEEE*, vol. 49, no. 12, pp. 84–93, 2011.

[6] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[7] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. USENIX Association, 2010.

[8] P. Calyam, S. Rajagopalan, A. Selvadhurai, S. Mohan, A. Venkataraman, A. Berryman, and R. Ramnath, "Leveraging openflow for resource placement of virtual desktop cloud applications," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, 2013, pp. 311–319.

[9] A. Khan and N. Dave, "Enabling hardware exploration in software-defined networking: A flexible, portable openflow switch," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, 2013, pp. 145–148.

[10] G. Hampel, M. Steiner, and T. Bu, "Applying software-defined networking to the telecom domain," in *INFOCOM, 2013 Proceedings IEEE*, 2013, pp. 3339–3344.

[11] N. Mavrogeorgi, S. Gogouvitis, A. Voulodimos, G. Katsaros, S. Koutsoutos, D. Kiriazis, T. Varvarigou, and E. K. Kolodner, "Content based slas in cloud computing environments," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 977–978.

[12] A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, M. Sloman, P. Flegkas, M. Charalambides, and G. Pavlou, "Policy refinement for diffserv quality of service," *IEEE eTransactions on Network and Service Management*, vol. 3, no. 2, 2005.

[13] POX, "Pox openflow controller," 2013, Accessed: Sept. 2013. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[14] Python Software Foundation, "Python language reference, version 2.7," 2013, Accessed: Sept. 2013. [Online]. Available: http://www.python.org

[15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[16] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010.

[17] K. Bakshi, "Considerations for software defined networking (sdn): Approaches and use cases," in *Aerospace Conference, 2013 IEEE*. IEEE, 2013, pp. 1–9.

[18] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser, "Terminology for policy-based management," RFC Editor, Tech. Rep., 2001.

[19] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy core information model–version 1 specification," RFC 3060, February, Tech. Rep., 2001.

[20] D. C. Verma, "Simplifying network administration using policy-based management," *Network, IEEE*, vol. 16, no. 2, pp. 20–26, 2002.

[21] W. Han and C. Lei, "A survey on policy languages in network and security management," *Computer Networks*, vol. 56, no. 1, pp. 477–489, 2012.

[22] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 335–371, 2004.

[23] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," RFC 2475, December, Tech. Rep., 1998.

[24] A. Bandara, E. Lupu, J. Moffett, and A. Russo, "A goal-based approach to policy refinement," in *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, 2004, pp. 229–239.

[25] J. Moffett and M. Sloman, "Policy hierarchies for distributed systems management," *Selected Areas in Communications, IEEE Journal on*, vol. 11, no. 9, pp. 1404–1414, 1993.

[26] I. Aib and R. Boutaba, "On leveraging policy-based management for maximizing business profit," *Network and Service Management, IEEE Transactions on*, vol. 4, no. 3, pp. 25–39, 2007.

[27] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman, "Policy refinement: Decomposition and operationalization for dynamic domains," in *Network and Service Management (CNSM), 2011 7th International Conference on*, 2011, pp. 1–9.

[28] ——, "Decomposition techniques for policy refinement," in *Network and Service Management (CNSM), 2010 International Conference on*, 2010, pp. 72–79.

[29] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher, *Internet Denial of Service: Attack and Defense Mechanisms (Radia Perlman Computer Networking and Security)*. Prentice Hall PTR, 2004.

[30] D. A. Menascé, "Qos issues in web services," *Internet Computing, IEEE*, vol. 6, no. 6, pp. 72–75, 2002.

[31] J. Korhonen, H. Tschofenig, M. Arumaithurai, A. Lior, and M. Jones, "Traffic classification and quality of service (qos) attributes for diameter," RFC 5777, February, Tech. Rep., 2010.

[32] W. Yeong, T. Howes, and S. Kille, "Lightweight directory access protocol," 1995.

[33] J. Postel and J. K. Reynolds, "Rfc 1700 assigned numbers," *Network Working Group*, 1994.

[34] S. Das, E. Lee, K. Basu, and S. Sen, "Performance optimization of voip calls over wireless links using h.323 protocol," *Computers, IEEE Transactions on*, vol. 52, no. 6, pp. 742–752, 2003.

[35] IEEE, "Ieee draft standard for local and metropolitan area networks–station and media access control connectivity discovery," IEEE, Tech. Rep., 2009.

[36] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[37] P. Karn and C. Partridge, "Improving round-trip time estimates in reliable transport protocols," *ACM SIGCOMM Computer Communication Review*, vol. 17, no. 5, pp. 2–7, 1987.

[38] C. Partridge, "A proposed flow specification," RFC 1363, September, Tech. Rep., 1992.

[39] B. Keepence, "Quality of service for voice over ip," in *Services Over the Internet - What Does Quality Cost? (Ref. No. 1999/099), IEE Colloquium on*, 1999, pp. 4/1–4/4.

[40] J. Postel *et al.*, "Rfc 792: Internet control message protocol," *InterNet Network Working Group*, 1981.

[41] C. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *Computers, IEEE Transactions on*, vol. C-34, no. 10, pp. 892–901, 1985.

[42] R. Braden, D. Clark, and S. Shenker, "Rfc 1633: Integrated services in the internet architecture: an overview, june 1994," *Status: Informational*, 1994.

[43] E. C. Lupu and M. Sloman, "Conflicts in policy-based distributed systems management," *Software Engineering, IEEE Transactions on*, vol. 25, no. 6, pp. 852–869, 1999.

AppendixB ACCEPTED PAPER – IM 2015

In this paper we presented a Policy Authoring framework for SDN management in which operators write high-level policies that are refined into lower-level ones. Policy refinement was accomplished by using logical reasoning for analyzing policy objectives. The Policy Authoring Framework was integrated with a customized OpenFlow Controller. Thus, the policy authoring introduced in this paper was a further step toward a comprehensive policy refinement toolkit for SDN management. As a result, the initial manual process performed by an administrator was removed, enabling an automatic refinement process of SLAs into a set of rules to be deployed by our customized OpenFlow controller. As a result, policies are refined with minimal human intervention, as the framework analyzes regexes in each SLA and applies logical reasoning based on network conditions that can fulfill the requirements of these SLAs. In addition, SLAs are specified and rules are deployed through a user-friendly policy authoring framework with minimal disruption to the network. This paper was accepted to appear in IM 2015.

- **Title –**
  Policy Authoring for Software-Defined Networking Management
- **Conference –**
  The 14th IFIP/IEEE International Symposium on Integrated Network Management (IM-2015)
- **Type –**
  Main track (full-paper)
- **Qualis –**
  B1
- **URL –**
  <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=XXXXX>
- **Date –**
  May 11-15, 2015
- **Held at –**
  Ottawa, Canada
- **Digital Object Identifier (DOI) –**
  <http://dx.doi.org/XX.XXXX/IM.2015.X>

Policy Authoring for Software-Defined Networking Management

Cristian Cleder Machado, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville, Alberto Schaeffer-Filho
Computer Networks Group – Institute of Informatics – Federal University of Rio Grande do Sul
Porto Alegre, Brazil
Email: {ccmachado, jwickboldt, granville, alberto}@inf.ufrgs.br

Abstract— **Software-Defined Networking (SDN) permits centralizing part of the decision-logic in controller devices. Thus, controllers can have an overall view of the network, assisting network programmers to configure network-wide services. Despite this, the behavior of network devices and their configurations are often written for specific situations directly in the controller. As an alternative, techniques such as Policy-Based Network Management (PBNM) can be used by business-level operators to write Service Level Agreements (SLAs) in a user-friendly interface without the need to change the code implemented in the controllers. In this paper, we introduce a framework for Policy Authoring to *(i)* facilitate the specification of business-level goals and *(ii)* automate the translation of these goals into the configuration of system-level components in an SDN. We use information from the network infrastructure obtained through SDN features and logic reasoning for analyzing policy objectives. As a result, experiments demonstrate that the framework performs well even when increasing the number of expressions in an SLA or increasing the size of the repository.**

Keywords—policy authoring; policy refinement; PBNM; SDN;

I. INTRODUCTION

Software-Defined Networking (SDN) [1] has simplified network administration by logically centralizing part of the decision making process in a controller element, such as an OpenFlow controller [2]. Controllers have an overall view of the network, which assists network operators in managing network-wide services [3]. However, we argue that SDN alone does not satisfactory improve the network operator's ability of writing concise yet expressive rules for network management. Despite the benefits of SDN, the intended network behavior is usually defined by static rules written to cope with specific situations [4][3]. This ends up hindering the development and deployment of new network services. Moreover, to deal with diverse situations, the amount of rules can become prohibitive.

An approach to tackle this problem is the use of Policy-Based Network Management (PBNM) [5][6]. In PBNM, an operator specifies infrastructure goals and constraints in the form of high-level policies to guide the behavior of the network. The use of PBNM aims to reduce the complexity of network management tasks [7]. Techniques such as policy refinement can be used to automatically translate high-level policies into a set of low-level policies for configuration of various devices of a system [8]. The use of PBNM in computer networks has been investigated for over a decade [8][9]. However, PBNM and policy refinement in the novel context of SDN has been much less explored [4][10]. We argue that PBNM for SDN management is still in its infancy, and the work in the area often ignores the vast literature on PBNM produced before the advent of SDN. Thus, several aspects of PBNM can be exploited toward more flexible SDN management.

We introduce in this paper a Policy Authoring framework for SDN management in which operators write high-level policies (expressed in a Controlled Natural Language - CNL [11]) that are refined into lower-level ones. In previous work, we have customized features of an OpenFlow controller aiming to collect information about the network infrastructure that can assist in improving the refinement process [12]. The policy authoring process introduced in this paper is a further step toward a comprehensive policy refinement toolkit for SDN which enables refining policies into a set of rules to be deployed by our customized OpenFlow controller. Policy refinement is accomplished by using logical reasoning [13] for analyzing policy objectives. On the one hand, inductive reasoning indicates the goals that should be extracted and fulfilled at lower-levels of abstraction. On the other hand, abductive reasoning confronts these goals with the network characteristics obtained from an SDN controller to indicate whether the network infrastructure can accommodate such goals. We developed a prototype as a proof-of-concept.

The main contributions of this paper are: *(i)* refined policies with minimal human intervention; *(ii)* analysis of the infrastructure's ability to fulfill the requirements of high-level policies; *(iii)* decreased amount of network rules coded into the controller; and *(iv)* management and deployment of new rules with minimal disruption to the network.

In this paper, we have limited the scope of our experiments and evaluation to QoS management. However, the principles of policy authoring presented here can be more generally applicable to other areas. We define three different Service Level Agreements (SLAs) by changing the number of expressions (amount of requirements, values, services, and QoS classes) and present experiments in five different scenarios. Results demonstrate that the policy authoring framework performs well, even when increasing the number of expressions in an SLA or increasing the size of the repository of rules.

This paper is organized as follows: in Section II we provide a briefly description of the main concepts used in our work. In Section III we present details of our policy authoring framework for SDN management. In Section IV we present the experiments and a discussion about the achieved results. In Section V, related work is discussed. Finally, in Section VI we conclude the paper with final remarks and future work.

II. Background: A Toolkit for Policy Refinement

In this section, we present an overview of the main concepts, techniques and elements used in our policy refinement toolkit. We also briefly describe our previous work [12], identifying the benefits of replacing traditional network architectures with SDN. In order to make the policy refinement toolkit independent of the network controller implementation or policy language, we defined a formal representation of high-level SLA policies using Event Calculus (EC) [14] and applied logical reasoning [13] to model both the system behavior and the policy refinement process for SDN management. The formalization aspects of our work are described elsewhere [15].

A. Policy Refinement Toolkit: An Overview

Our toolkit (see Figure 1) consists of three main elements: *(i)* a policy authoring framework (described in Section III), which is used by infrastructure-level programmers to specify the technical characteristics of services and by business-level operators to write SLAs in a *controlled natural language* (CNL) [11]; *(ii)* an OpenFlow controller, which collects information from the network infrastructure (which is the key to improve the refinement process); and *(iii)* a repository to store information from both the controller and the framework.

Our policy refinement toolkit is based on the research efforts of Bandara *et al.* [16] and Craven *et al.* [17]. These studies were limited by the characteristics of traditional networks, such as the notion of best-effort for QoS and the lack of a centralized control plane [3]. In traditional networks, the control plane is executed in each network device. Also, each device has its proprietary protocols thus becoming difficult to be programmed.

Instead, our approach introduces the use of Software-Defined Networking (SDN) [18][19] to enhance the refinement process. In SDN, these limiting factors of traditional networks can be overcome, since SDN is mainly characterized by a clear separation between the forwarding and control planes. Thus, differently from traditional networks, SDN has a logically centralized control plane which allows moving part of the decision-making logic of network devices to external controllers. This provides controller devices with the ability to have an overall view of the network and its resources, thus becoming aware of all the network elements and their characteristics [19][1]. Based on this centralization, network devices become simple packet forwarding elements, which can be programmed through an open interface, such as the OpenFlow protocol [2] SDN architecture in which network traffic information is centralized by a controller is valuable to our policy refinement approach. It makes it easier to retrieve information from the network infrastructure, and to validate SLA requirements more accurately.

We apply the concept of *logical reasoning* [13] to support the business-level operator in the refinement of an SLA. Logical reasoning has three modes:

- In *deductive reasoning*, a conclusion is reached by using a rule that analyzes a premise. For example, if streaming packets are transmitted the network becomes slower; streaming packets are being transmitted now; therefore, the network is slower.
- In *inductive reasoning*, the goal is to identify a rule, starting from a historical set of conclusions generated from a premise. For example, every time streaming packets are transmitted the network becomes slower; so, if streaming packets are transmitted tomorrow, the network will be slower.
- In *abductive reasoning*, starting from a conclusion and a known rule, we can explain a premise. For example, when streaming packets are transmitted the network becomes slower; the network is slower now; so, possibly streaming packets are being transmitted.

Our policy refinement toolkit uses two modes of logical reasoning: on the one hand, inductive reasoning indicates the SLA requirements that should be extracted and fulfilled at lower-levels of abstraction. On the other hand, abductive reasoning compares these requirements with the network characteristics obtained from an SDN controller to determine whether the network infrastructure can accommodate such requirements.

B. Low-level Controller Configuration

In order to support our policy refinement approach it was necessary to customize some functionality in the OpenFlow controller. This was required for collecting information about the network infrastructure, which is later used, for example, to calculate optimal routes. This customization was based on SDN native features only, and thus can be applied to any controller implementation. Therefore, our solution is not tied to any specific controller design or language. For example, topology discovery, which is available in POX [20] is a native feature of SDN offered by all controllers in different implementations. We emphasize that even though the commands supported by the forwarding elements are standardized, the controllers require different programming languages and/or support different features. This difference between controllers can reflect in the effort for customization that must be employed by an infrastructure-level programmer.

Thus, the behavior of the customized OpenFlow controller is divided into three phases: *(i) Startup Phase*: discovers services and possible paths between network elements and writes rules (which we call *standard rules*) in the flow table of the switches that are in the shortest path between each of the network elements; *(ii) Events Phase*: stays in a loop during the operation of the infrastructure to identify service events and determine the shortest path based on the characteristics of the network and service requirements; and *(iii) Analysis Phase*: implements the rules and monitors the network in order to identify possible enhancements for the active flows.

Ultimately, the policy authoring framework (described in detail in Section III) can be used to derive QoS class requirements from business-level SLAs. Policy authoring performs a process of computing low-level objectives/rules (SLOs Service Level Objectives) that must meet the high-level goals/policies. Then, the customized controller is capable of assigning the specific type of network traffic described by the SLA to its optimal route, given the set of requirements derived from the SLA. This QoS management strategy is based on routing (using the best path between network devices). The calculations for the best path are carried out using as weights/requirements the bandwidth, delay, jitter, and number of hops in each path of the physical topology. Each weight/requirement is presented in order of importance. If only one occurrence of a

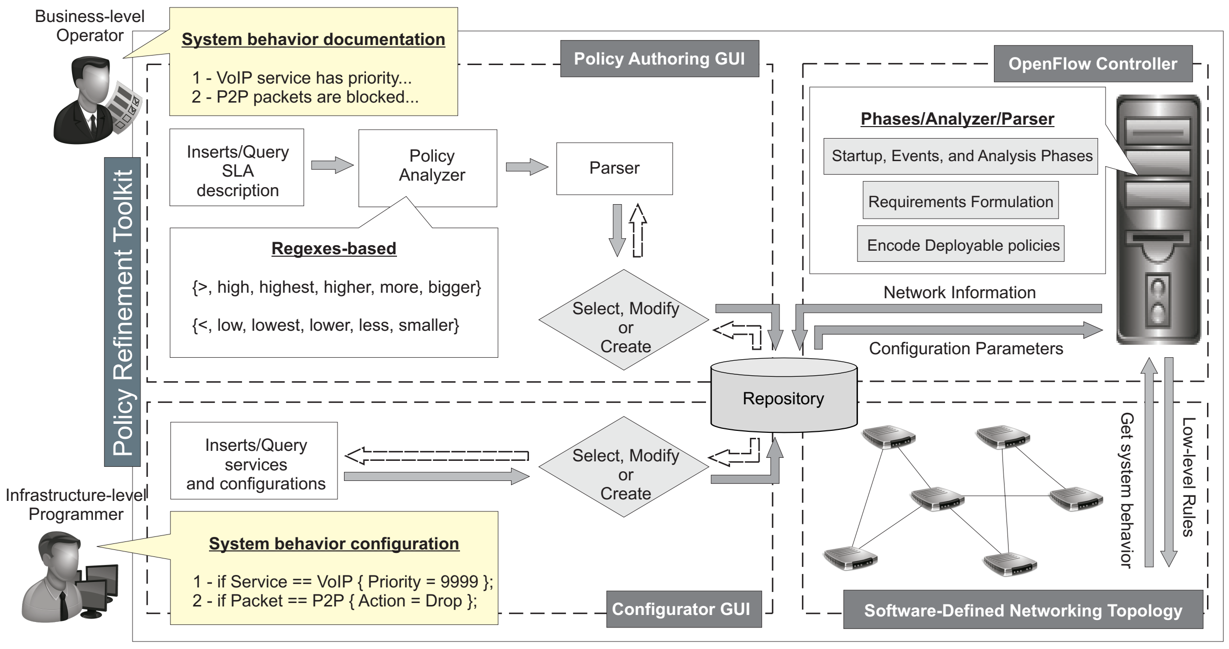

Business-level Operator
System behavior documentation
1 - VoIP service has priority...
2 - P2P packets are blocked...
Policy Authoring GUI
OpenFlow Controller
Policy Refinement Toolkit
Inserts/Query SLA description
Policy Analyzer
Parser
Regexes-based
{>, high, highest, higher, more, bigger}
{<, low, lowest, lower, less, smaller}
Select, Modify or Create
Phases/Analyzer/Parser
Startup, Events, and Analysis Phases
Requirements Formulation
Encode Deployable policies
Network Information
Configuration Parameters
Repository
Inserts/Query services and configurations
Select, Modify or Create
Infrastructure-level Programmer
System behavior configuration
1 - if Service == VoIP { Priority = 9999 };
2 - if Packet == P2P { Action = Drop };
Configurator GUI
Get system behavior
Low-level Rules
Software-Defined Networking Topology

Fig. 1: Overall Policy Refinement Toolkit.

weight/requirement is found, it will be chosen. Otherwise, the path that satisfies the largest number of weights/requirements – compared with the QoS class requirements identified in the policy authoring process – is chosen as the best path. The best path is part of the rule that will be configured by the controller into the flow table of each switch later. Besides informing which one is the best path, each rule receives the established priority in each QoS class. This is what establishes fairness and the distinction between traffic deserving high and low priority in the network. In summary, our strategy uses a requirements-based path and a priority for deciding routes.

Each rule is deployed in the flow table of the network device at runtime, aiming to minimize disruption of the network. Periodically, the controller checks if the configured paths remain the best choices, aiming to reduce processing overhead in network devices. In our experiments, each rule is configured with a timeout. We also set the checking intervals to, for example, *30s*. If at any time the controller identifies that there is a better alternative path, new rules are sent to the switches. If the current path remains the best, the controller only increases the value of the timeout for the rules on each network device. For further details we refer the reader to Machado *et al.* [12].

C. Policy Repository

The *Policy Repository* stores information about the behavior of the infrastructure, which is obtained during the network configuration process. For example, the repository stores all the possible links between elements, number of elements, bandwidth, delay and jitter.

Additionally, the repository maintains a list of all services and their parameters (*e.g.*, the packet identifier of the HTTP protocol), all QoS classes and services associated with them. We use standard TCP/IP information from packet headers to register a given service in the repository.

III. Policy Authoring for SDN

This paper extends our previous work on a refinement toolkit for high-level policies in SDN. Details on the low-level controller operation have been described in Machado *et al.* [12], and in this paper we focus on the policy authoring aspects only. In this section we describe in detail our Policy Authoring framework for SDN management. The main goal is to enable operators to express business goals, *e.g.*, Service Level Agreements (SLAs), without having to specify in detail what elements of the network infrastructure should receive the configurations and how they should be configured.

To provide a more targeted case-study, we concentrated our efforts in the support of policy configurations for QoS classes. The result obtained from the refinement of high-level policies are QoS-class requirements. Thus, the interpretation of an SLA is used for extracting the Service Level Objectives (SLOs). These SLOs are considered QoS-class requirements (*e.g.*, priority, bandwidth) by the Policy Authoring framework.

A. Controlled Natural Language

In this paper, we identify the business-level goals and high-level policies as SLAs. We introduce a *controlled natural language* (CNL) [11] to establish restrictions and requirements for writing business-level goals. The grammar of this language is defined below:

Listing 1: Grammar of the controlled natural language.

```
Language : →(<QoS>|<Service >)<Preposition ><Expression >
QoS : →qos−regexes
Service : →service −regexes
Preposition : →should receive | should not receive
Expression : →<Term>|<Term><Connective ><Expression >
Term : →<Parameter ><Operator ><Value>
Parameter : →requirements −regexes
Connective : →And | Or
Operator : →adjective −regexes
```

10 Value : →v

Our Policy Authoring framework uses *regexes* as a concise and flexible way of identifying strings of interest such as particular characters (*e.g.*, $>$, $<$, $=$, $\neq$, $\leq$, $\geq$) or words (*e.g.*, high, low, http, ftp, gold, silver). We defined the following types of *regexes*: `qos-regexes`: regular expression to identify QoS classes; `service-regexes`: regular expression to identify services; `requirements-regexes`: regular expression to identify service requirements; `adjective-regexes`: regular expression to identify adjectives in service requirements. Table I shows examples of regular expressions that can be contained in an SLA.

TABLE I: Examples of regular expression.

Type	*Expression*	*Operator*
qos-regexes	Bronze, Silver, Gold, Platinum...	N/A
service-regexes	VoIP, Streaming, HTTP, FTP, SMTP, POP, P2P...	N/A
requirements-regexes	Priority, Bandwidth, Delay, and Jitter	N/A
adjective-regexes	more, high, higher, up, over...	$>$
	equal, like, even, same, similar...	$=$
	less, low, lower, down, below...	$<$

B. Bottom-up and Top-down Phases

We specifically introduce in this paper a policy authoring framework where infrastructure-level programmers specify technical characteristics of services, and business-level operators write SLAs in a controlled natural language. This framework and a customized controller [12] compose a refinement toolkit of high-level policies for SDN management. All aspects of the refinement process both in the framework as in the controller are automatically performed. The results generated by the refinement process are a set of rules to be deployed by the controller for the network infrastructure configurations. This toolkit is integrated with a formal representation based on Event Calculus (EC) and applies logical reasoning to model both the system behavior and the policy refinement process in SDN. This formalism assists infrastructure-level programmers to develop refinement tools and configuration approaches to achieve more robust SDN deployments. The EC-based formalism is described in [15].

The refinement process is split into two phases (Figure 2): The first phase, called bottom-up, consists of the network information (*e.g.*, bandwidth, delay) gathering process. A key element of this phase is the OpenFlow controller, which performs the data collection process. Using this information, the Policy Authoring framework uses abductive reasoning to indicate to the business-level operator what are the possible configurations. These indications are provided through settings performed previously – other SLAs or policies created manually by the operator – along with the characteristics that the network can support. More details about Policy Authoring are described in Section III-C.

The second phase, called top-down, refines high-level goals extracted from SLAs and translate them into achievable goals (SLOs). An operator writes the SLAs and creates – if necessary – the QoS classes needed to fulfill them. As mentioned

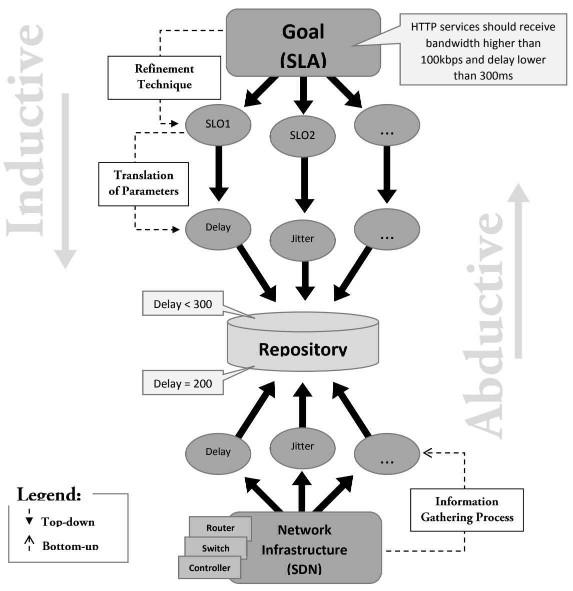

Inductive
Abductive
Goal (SLA)
HTTP services should receive bandwidth higher than 100kbps and delay lower than 300ms
Refinement Technique
SLO1
SLO2
...
Translation of Parameters
Delay
Jitter
...
Delay < 300
Repository
Delay = 200
Delay
Jitter
...
Information Gathering Process
Legend:
Top-down
Bottom-up
Router
Switch
Controller
Network Infrastructure (SDN)

Fig. 2: Deriving SLOs/parameters from goal and gathering network information.

previously, the bottom-up phase will try to indicate using abductive reasoning which are the best configurations for the SLA that is being written. Thus, multiple configuration options will be offered to the operator, who can select or customize an existing configuration, or even create a new configuration.

C. Policy Authoring Framework

Regarding the Policy Authoring framework, the operator inserts an SLA that defines explicitly or implicitly business-level goals. When inserting each policy, the *Policy Analyzer* component (Figure 1) uses *regexes* (regular expressions) – previously stored in the *Policy Repository* – to match the expressions written in natural language, and suggests the more appropriate QoS class/classes to the SLA. The operator can create a set of QoS classes beforehand. Each class may have a number of QoS requirements. For example, Gold QoS class may contain priority = 20, bandwidth = 512kbps, delay = 2ms, and jitter = 1ms, while Bronze QoS class may contain bandwidth = 2kbps.

This Policy Authoring framework relies on abductive reasoning to suggest QoS classes. As mentioned previously, in abductive reasoning, starting from a *conclusion* and a known *rule*, it is possible to explain a particular *premise*. We use the following SLA to illustrate how logical reasoning works and, subsequently, we use the same SLA to explain how the Policy Authoring operates:

"HTTP services should receive bandwidth higher than 100kbps and delay lower than 300ms".

The *conclusion* of this SLA is *"HTTP services should receive"* certain characteristics. The *rules* for reaching this conclusion are *"bandwidth $>$ 100kbps"* and *"delay $<$ 300ms"*. Thus, we present the *premise* (QoS class in the repository) that has this rule and which can *possibly* arrive at this conclusion.

We define a query that assigns weights to results based on the importance of the *regexes* contained in the SLA.

These expressions are compared to the information stored in the repository to sort the results and display them. The ordering thus follows: *(i)* expressions related to QoS classes; *(ii)* expressions related to services, and *(iii)* expressions related to service requirements. The steps to query and display the information to the business-level operator are the following:

Step1 – Check if there is any `qos-regexes` expression in the SLA indicating a class, *e.g.*, QoS Gold, Silver. If there are occurrences of these expressions, the Policy Authoring framework returns the QoS class values, based on the identified `qos-regexes`. For the SLA presented in the example, we have no expressions of this type.

Step2 – Check if there is any `service-regexes` expression in the SLA relating to services, *e.g.*, FTP, VoIP. If there are occurrences of these expressions, the Policy Authoring framework returns the QoS class values to which the services are associated. For the SLA in the example, there is a `service-regexes` (HTTP), which may be associated with a QoS class in the repository.

Step3 – Analyze the expressions indicating service requirements, *e.g.*, priority, bandwidth. If there are occurrences of these expressions, the Policy Authoring framework performs the following operations: *(i)* identify and count the `requirements-regexes` found, and *(ii)* identify and count the `adjective-regexes` that come before and after any `requirements-regexes`.

We also developed a technique for identifying and counting the `requirements-regexes`, which allows the operator to optimally match the `adjective-regexes` found with their respective requirements. In the SLA above, we can observe the `adjective-regexes` *higher* and *lower*, which are related to the `requirements-regexes` *bandwidth* and *delay*, respectively. The *Policy Analyzer* identifies any `adjective-regexes` and examines the SLA, identifying the proximity of the `adjective-regexes` referring to `requirements-regexes`. This is performed by checking if `adjective-regexes` are located before or after `requirements-regexes`. At the end, the result is presented to the business-level operator.

The Policy Authoring framework uses abductive reasoning to show what are the best configurations for the SLA. Thus, the *Policy Analyzer* can identify, for example, that there is already a QoS class configured with low delay, or that the throughput for the specified network path already exceeds the network configuration, indicating that the policy should be reformulated. Also, the operator can be warned of potential conflicts or even non-compliance with policies. If the operator chooses one of the suggested QoS classes, the Policy Authoring framework will store the information extracted from the SLA, *e.g.*, the service, with the selected class.

Suggestions provided through abductive reasoning are not mandatory. If after analyzing them the operator decides they do not meet the high-level goals, the suggestions can be ignored. At this point, the operator can analyze the information presented by the Policy Authoring framework and rely on inductive reasoning to perform the following actions:

- *Modify existing policy/class* – This action allows the operator to change a predetermined parameter, *e.g., priority* $= 100$ to *priority* $= 101$, or add a parameter that does not yet exist, *e.g., delay* $\leq 120ms$. This modification may impact other policies, and the *Analyzer* uses inductive reasoning to identify the classes in the repository that may be impacted. Thus, the operator has the opportunity to analyze policy-by-policy and decide if the change is viable or not.
- *Create policy/class based on existing class* – This action is an alternative to modifying an existing class. Through this action, a new class created by the operator inherits the parameters of an existing class, which can be customized as needed. The *Policy Analyzer* uses inductive reasoning to automatically check if the parameter values of this new class are not identical to the ones in an existing class in the repository. If so, the existing class is returned instead.
- *Create a new policy/class* – The creation of new classes can be conducted *(i)* if a class that meets the objectives of the SLA does not exist, or *(ii)* if the parameters of other classes retrieved via abductive or inductive reasoning are not related to the objectives of the new SLA. Thus, the operator can set the new class parameter-by-parameter to meet the SLA objectives.

After any of the actions above is executed, the *Parser* component (Figure 1) will be executed and the policies/classes will be stored in the *Policy Repository*. It is based on this information that the Policy Authoring framework estimates the amount of allocated traffic per class and warns if the infrastructure can support or not new policies. Further, the policy repository contains a list of services associated with their TCP/UDP port (*e.g.*, HTTP = [80,8080], SSH = [22], SMTP = [25,587]). This list was created using RFC 1700 [21]. Subsequently, the OpenFlow controller reads from the repository these new policies starting the *Analysis Phase* for setting up the appropriate rules in forwarding devices, as explained in Section II-B.

IV. Prototype and Evaluation

In this section we describe a prototype implementation and evaluation of our Policy Authoring framework. For details about the low-level controller implementation we refer the reader to our previous work [12].

A. Prototype Implementation

We developed the prototype using the Django web framework[1]. We chose Django due to its support to the Python language and the support it provides to create web applications. For the interface design we used the Bootstrap front-end framework[2]. The prototype is split into two modules, Policy Authoring GUI and Configuration GUI, described as follows.

1) Policy Authoring GUI: We developed a user-friendly interface for Policy Authoring in order to allow the configuration of the network through business goals. Thus, a business-level operator can use the Policy Authoring GUI to express high-level goals and receive feedback from his/her requests.

Figure 3 illustrates the home screen of the Policy Authoring GUI. It presents statistics about the number of policies, classes,

[1]http://www.djangoproject.com/
[2]http://getbootstrap.com/

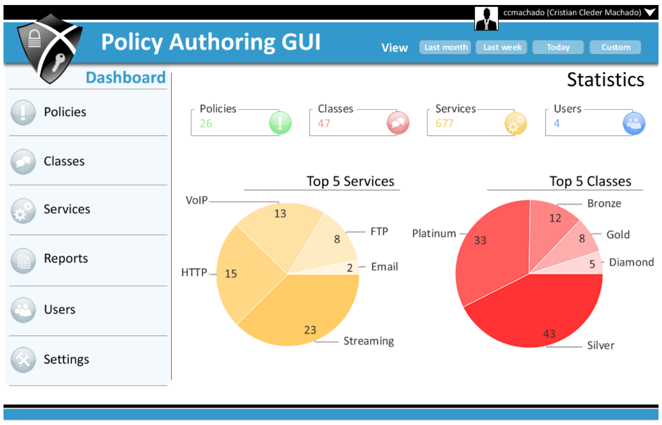


Fig. 3: Policy Authoring GUI Dashboard.

services, and users registered. In addition, it shows two charts about the top 5 services that most appear in policies and the top 5 QoS classes that most have linked policies. The *dashboard* is composed of the following items:

- *Policies* – Used by business-level operators to create, search, edit, remove, enable or disable policies. Operators can also associate a high-level SLA with the QoS class that best meets the SLA requirements.
- *Classes* – Used to specify QoS classes. Infrastructure-level programmers and business-level operators can perform the necessary parameter settings for each class.
- *Services* – Used by infrastructure-level programmers to record, edit, and delete services. Also, through this interface a service can be associated with a QoS class.
- *Reports* – Used by business-level operators and infrastructure-level programmers to view reports of policies, services, and classes. For example, classes that contain most policies or services that appear less frequently in policies. Additionally, some reports can be filtered by specific parameters, *e.g.,* priority, delay.
- *Users* – Used to create, search, edit, remove, enable or disable system users.
- *Settings* – Used to configure system settings, such as database connection information.

2) Configurator GUI: Our aim is to facilitate not only the description of business objectives but also the configuration of the infrastructure. The *Configurator GUI* is designed to manage the registration of services and parameters. An infrastructure-level programmer inserts service information, such as *ServiceName* and *Port* (as used in TCP/IP). Subsequently, the infrastructure-level programmer may create QoS classes with parameters and their respective values. The fields that may be informed are *ClassName*, Priority, Bandwidth, Delay, and Jitter.

We decided to group services by class, thus after QoS classes have been defined, each service is associated with a QoS class. This step is important because if services are previously associated with some class, the toolkit will have a better performance since there will be an entry in the repository for a group of services as opposed to one entry for each service. Thus, services with similar requirements can be grouped into a single class while maintaining fairness among competing in the same link.

B. Evaluation

We present in this section experiments and initial results obtained with the implemented toolkit. Our goal is to measure the response time of the end-to-end process, *i.e.*, from policy authoring to deployment of low-level rules in the controller device. In order to perform the experiments, we created three SLAs (Table II) by changing the number of expressions, where SLA 2 has more expressions than SLA 1 and SLA 3 has more expressions than SLA 2. Our goal is to show the robustness and efficiency of the refinement process when we increase the number of expressions that should be compared. We also created three scenarios (Table III) by varying the number of network devices and adding redundant links between some network devices. The scenarios used in the experiments were based on mesh topologies. Our goal was to demonstrate the ability of the framework to operate in increasingly large topologies. These scenarios were created using the Mininet emulator and experiments were performed on an AMD 2.0 GHz Octa Core with 32 GB RAM memory.

TABLE II: Description of SLAs used in the experiments.

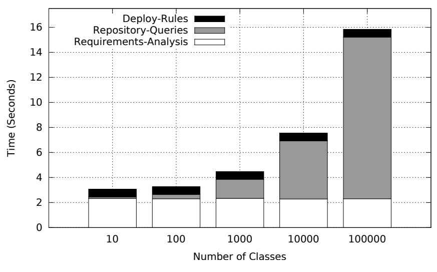
SLA	Description of SLAs
SLA$_1$	HTTP traffic should receive lower Quality of Service and low priority compared with other services.
SLA$_2$	Streaming traffic should receive higher priority, low delay and bandwidth higher than 512kbps.
SLA$_3$	VoIP traffic should receive higher priority, delay less than 200ms, low jitter, and bandwidth higher than 128kbps.

TABLE III: Number of switches and links in each scenario.

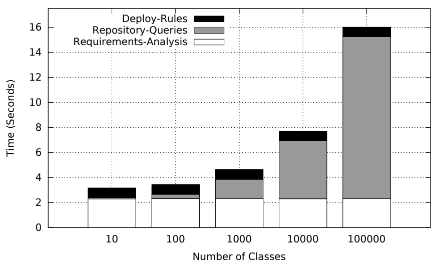
Scen.	SwL0	SwL1	SwL2	SwL3	SwL4	Hosts	Links
X	16	8	8	4	0	32	88
Y	32	16	16	8	4	64	176
Z	64	32	32	16	8	128	210

We applied the three SLAs to five different repositories A-E and populated each repository according to the number of classes, where $A = 10$, $B = 100$, $C = 1,000$, $D = 10,000$, and $E = 100,000$ classes. In addition, each experiment was executed thirty times. We performed experiments on all variations of SLAs, repositories, and scenarios. Due to space limitations, we present the most relevant results only. In particular, the experiments described in this sections intend to evaluate our prototype in terms of average execution time and percentage of the total time occupied by each stage of the policy authoring process.

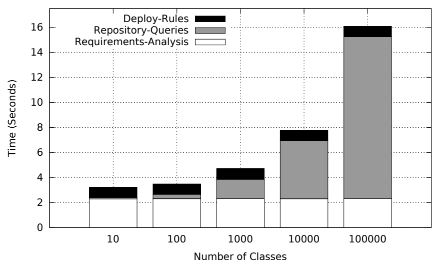
Figure 4 shows the average response time for SLA 3 in each scenario. We break the total execution time down in three categories, namely requirements analysis (*i.e.*, parse the SLAs and their regexes), repository queries (*i.e.*, search for the best matching QoS class), and deploy rules (*i.e.*, install the flow rules in the controller). By increasing the number of classes, it is possible to observe that the average time spent performing repository queries also grows. This increase is visible in all



(a) SLA 3 (scenario X)

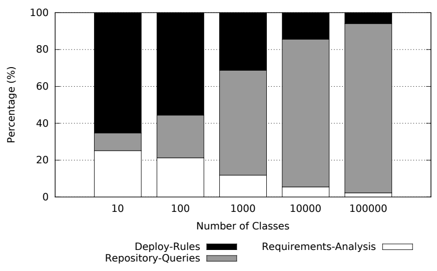


(b) SLA 3 (scenario Y)

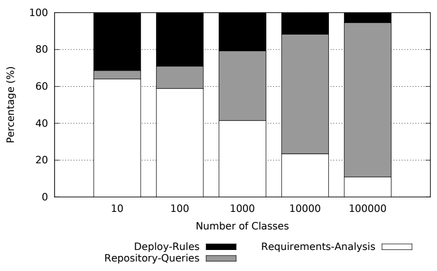


(c) SLA 3 (scenario Z)

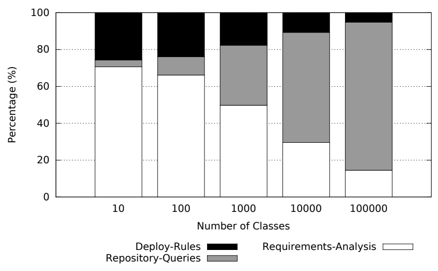
Fig. 4: Average response time for SLA 3 performed in scenarios X, Y, and Z.



(a) SLA 1 (scenario Z)

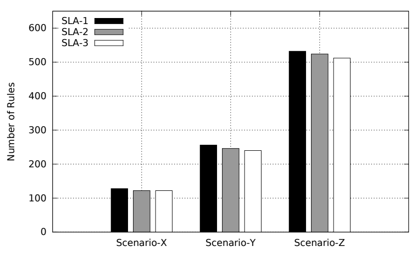


(b) SLA 2 (scenario Z)

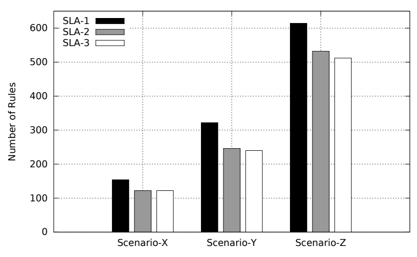


(c) SLA 3 (scenario Z)

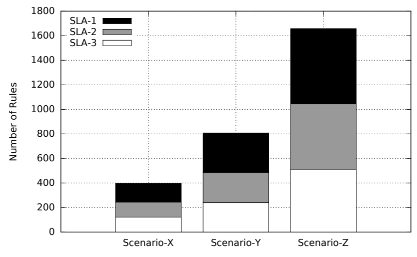
Fig. 5: Percentage of total time for each experiment performed in scenario Z.



(a) Total number of rules deployed by each SLA performed separately in each scenario.



(b) Total number of rules deployed by each SLA performed simultaneously in each scenario.



(c) Total amount of rules deployed by all SLAs in each scenario.

Fig. 6: Number of rules deployed in each scenario.

experiments performed with SLAs 1, 2, and 3. This behavior is expected, since the number of classes has influence on the number of queries to obtain the ideal matches between SLAs and QoS classes.

In Figure 5 the y-axis shows the percentage of the total time occupied by each process in the experiments performed with SLAs 1, 2, and 3 in scenario Z. From these results it is possible to note that, according to the level of complexity of each SLA, the percentage of time for analyzing requirements also increases. This happens due to the increase in the number of occurrences of regular expressions found in each SLA.

Figure 6(a) shows the total number of rules generated by refining each SLA *separately* in each scenario. As can be observed, each SLA generates practically the same number of rules in each scenario. SLA 1 shows a small difference in the number of rules deployed compared to SLAs 2 and 3. This occurs because SLA 1 has lower QoS requirements (*i.e.,* low priority) compared to other services, which causes the choice of routes with more hops and consequently causes rules to be deployed in more devices. It is worth mentioning that the total number of rules generated by the policy authoring framework is smaller than the total number of rules that would have to be manually created on all network devices. This is because, as our approach is based on routing, it creates a spanning tree to find all routes between sources and destinations. Thus, some routes may be common between different sources and destinations. As a result, a number of switches do not need to be configured, thus reducing the total number of rules required in each scenario.

The growth in the total number of rules in SLA 1 appears more clearly when we performed *simultaneously* the refinement of the three SLAs in each scenario (Figure 6(b)). Our framework attempts to fulfill the requirements of each SLA. In order to achieve this, it identifies the possibility of routing (balancing) each SLA by alternative routes without failing to fulfill their requirements. Thus, SLA 1 receives routes with more hops in order not to compete with SLAs 2 and 3 which have higher priority requirement.

Finally, Figure 6(c) shows the total amount of rules generated by all SLAs in each scenario. This illustrates the benefits of our policy authoring and refinement approach, in which the infrastructure-level programmer does not need to be concerned with the number of low-level configuration rules to be deployed in the network. Our results suggest that the prototype is able to support the refinement of SLAs and the installation of flow rules in large-scale deployments. Even if we consider the scenario with the largest number of switches and links (Figure 4(c)), and the largest number of QoS classes, the total measured time remains within acceptable bounds. Moreover, as mentioned previously, the framework optimizes the deployment of rules according to the requirements of each SLA and according to each scenario.

V. Related Work

Policy Authoring approaches to facilitate the writing, analysis, and implementation of high-level policies have been proposed in the past. Brodie *et al.* [22] present a platform-independent framework to specify, analyze, and deploy security and networking policies. A portal prototype for policy authoring, based on natural language and structured lists, allows the management of policies from their specification to enforcement. The policy authoring portal enables web users to write policies, using a high-level language, which are translated and mapped to specific low-level configurations. Johnson *et al.* [23] present a template-based framework for policy authoring. The work describes the relationship between general templates and specific policies, and the skills required from users to produce high-quality policies. Although these research efforts investigate important issues regarding policy authoring, none of them presents a formal language for authoring, the use of logical reasoning to assist the refinement process, or experimental results.

Zhao *et al.* [24] describe the design and implementation of an end-to-end framework for the management of cloud-hosted databases from a consumer's perspective. The approach is based on the interpretation of SLAs to assist the dynamic provisioning of databases. The framework checks if SLAs have changed and automatically performs corrective actions to enforce the new specifications. Villegas *et al.* [25] present a framework for the analysis of provisioning and allocation policies for Infrastructure-as-a-Service clouds, *i.e.*, policies to dynamically allocate resources which remain largely under-utilized over time. Oriol Fito *et al.* [26] introduce a Business-Driven ICT Management (BDIM) model to satisfy the business strategies of cloud providers. The objective is to evaluate the impact of events related to ICT using business-level metrics. A Policy-Based Management system analyzes these events and is able to determine automatically the ICT management actions that are most appropriate. Craven *et al.* [17] introduced a refinement process for obligation and authorization policies that addresses policy translation, operationalization, re-refinement, and deployment. The work describes in details how a UML information-based formalism of system elements, a high-level policy, and translation rules that relate actions can produce concrete low-level policies. Bandara *et al.* [16] presented a tool support for the refinement process, and used case-studies based on DiffServ QoS management. The refinement process introduced the use of goal design and applied abductive reasoning as a strategy to generate low-level policies that aim to achieve a specific high-level goal.

Despite the above research efforts have achieved satisfactory results, they were also limited by the characteristics imposed by traditional IP networks, such as best-effort packet delivery and distributed control state. We distinguish our policy authoring framework from other existing approaches by exploring the characteristics of SDN architectures, such as centralized control plane and overall view of the network infrastructure to enhance the policy refinement process. To the best of our knowledge, this is the first time that policy authoring and refinement techniques have been applied to SDN management.

VI. Concluding Remarks

In this paper we presented a policy authoring framework to facilitate the configuration of SDN architectures based on the interpretation of high-level policies. The proposed policy authoring framework assists business-level operators to more easily specify overall service requirements, which can then be automatically translated into the configuration of an SDN infrastructure. An important aspect to be emphasized is that our approach is flexible, and allows the business-level operator to decide whether to accept or not the suggestions given. Thus, the operator can fully or partially accept the suggestion, or create his/her own configuration. Also, our experiments have showed that the toolkit performs well even with the increase in the number of QoS classes and in the complexity of the SLAs.

Different from past research efforts ([22], [23], [16], [17]), our policy authoring process is based on a policy refinement technique that analyzes the infrastructure ability to fulfill the requirements of high-level policies using the information obtained from an SDN controller. As a result, policies are refined with minimal human intervention, as the framework analyzes regexes in each SLA and applies logical reasoning based on network conditions that can fulfill the requirements of these SLAs. Thus, manual workload related to SDN management can be reduced because the flow rules are automatically generated and installed, instead of requiring the operator to directly write and deploy rules. Further, SLAs are specified and rules are deployed through a user-friendly policy authoring framework with minimal disruption to the network.

While we relied on the use of SDN architectures to improve the refinement process, by using the PBNM paradigm we also indirectly addressed problems typically found in SDN, *e.g.*, the issue of having static rules and configurations that are often written for specific situations directly in the controller. From the viewpoint of the network operation, the use of PBNM aims to reduce the complexity of the network management tasks allowing the system to gain a certain level of autonomy [7]. Thus, by using PBNM we reduced the amount of static rules and configurations. This was achieved by writing reusable code that deploys specific rules obtained from a repository.

As a part of our future work, we intend to extend the policy authoring framework to support more terms, expressions, prescriptions, and rules. In addition, our approach is limited to rules triggered by the occurrence of an event, *i.e.*, a flow receives a specific action. We intend to extend our grammar to support temporal logic. This will allow the specification of policies defined by an interval of time. Moreover, we also intend to investigate techniques for detection and resolution of policy conflicts in different levels of abstraction. Further, we intend to identify ongoing standardization efforts related to policy-based management in order to improve the prototype. Finally, we intend to analyze the toolkit behavior when managing other resources and types of services.

References

[1] Open Networking Foundation, "Software-defined networking: The new norm for networks," Open Networking Foundation ONF, Tech. Rep., April 2012.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, mar 2008.

[3] S. Sezer, S. Scott-Hayward, P. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? implementation challenges for software-defined networks," *Communications Magazine, IEEE*, vol. 51, no. 7, pp. 36–43, July 2013.

[4] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," *NSDI, Apr*, 2013.

[5] D. Verma, "Simplifying network administration using policy-based management," *Network, IEEE*, vol. 16, no. 2, pp. 20–26, Mar 2002.

[6] R. Neisse, E. Pereira, L. Granville, M. Almeida, and L. Rockenbach Tarouco, "An hierarchical policy-based architecture for integrated management of grids and networks," in *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, June 2004, pp. 103–106.

[7] W. Han and C. Lei, "A survey on policy languages in network and security management," *Computer Networks*, vol. 56, no. 1, pp. 477 – 489, 2012.

[8] A. Bandara, E. Lupu, J. Moffett, and A. Russo, "A goal-based approach to policy refinement," in *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, 2004, pp. 229–239.

[9] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou, "A functional solution for goal-oriented policy refinement," in *Policies for Distributed Systems and Networks, 2006. Policy 2006. Seventh IEEE International Workshop on*, June 2006, pp. 133–144.

[10] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "SDX: A software defined internet exchange," *SIGCOMM Comput. Commun. Rev.*, 2014, (to appear).

[11] T. Kuhn, "A survey and classification of controlled natural languages," *Computational Linguistics*, vol. 40, no. 1, pp. 121–170, 2013.

[12] C. C. Machado, L. Z. Granville, A. Schaeffer-Filho, and J. A. Wickboldt, "Towards SLA policy refinement for QoS management in software-defined networking," in *Advanced Information Networking and Applications (AINA-2014), 2014 28th IEEE International Conference on*, 2014, pp. 397–404.

[13] M. Shanahan, "An abductive event calculus planner," *The Journal of Logic Programming*, vol. 44, no. 1, pp. 207–240, 2000.

[14] R. Kowalski and M. Sergot, "A logic-based calculus of events," *New Gen. Comput.*, vol. 4, no. 1, pp. 67–95, jan 1986. [Online]. Available: http://dx.doi.org/10.1007/BF03037383

[15] C. C. Machado, J. A. Wickboldt, L. Z. Granville, and A. Schaeffer-Filho, "An EC-based formalism for policy refinement in software-defined networking." 2015, submitted to ISCC 2015.

[16] A. Bandara, E. Lupu, A. Russo, N. Dulay, M. Sloman, P. Flegkas, M. Charalambides, and G. Pavlou, "Policy refinement for diffserv quality of service management," in *Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on*, May 2005, pp. 469–482.

[17] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman, "Policy refinement: Decomposition and operationalization for dynamic domains," in *Network and Service Management (CNSM), 2011 7th International Conference on*, Oct 2011, pp. 1–9.

[18] K. Bakshi, "Considerations for software defined networking (SDN): Approaches and use cases," in *Aerospace Conference, 2013 IEEE*, March 2013, pp. 1–9.

[19] J. Wickboldt, W. Jesus, P. Isolani, C. Both, J. Rochol, and L. Granville, "Software-Defined Networking: Management Requirements and Challenges," *IEEE Communications Magazine - Network & Service Management Series*, January 2015.

[20] POX, "Pox openflow controller," 2013, Accessed: Sept. 2013. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[21] J. Postel and J. K. Reynolds, "Rfc 1700 assigned numbers," *Network Working Group*, 1994.

[22] C. Brodie, D. George, C.-M. Karat, J. Karat, J. Lobo, M. Beigi, X. Wang, S. Calo, D. Verma, A. Schaeffer-Filho, E. Lupu, and M. Sloman, "The coalition policy management portal for policy authoring, verification, and deployment," in *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on*, June 2008, pp. 247–249.

[23] M. Johnson, J. Karat, C.-M. Karat, and K. Grueneberg, "Optimizing a policy authoring framework for security and privacy policies," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*, ser. SOUPS '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:9.

[24] L. Zhao, S. Sakr, and A. Liu, "A framework for consumer-centric SLA management of cloud-hosted databases," *Services Computing, IEEE Transactions on*, vol. PP, no. 99, 2013.

[25] D. Villegas, A. Antoniou, S. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, May 2012, pp. 612–619.

[26] J. Oriol Fito, M. Macias, F. Julia, and J. Guitart, "Business-driven it management for cloud computing providers," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, Dec 2012, pp. 193–200.