Extending SDN control to large-scale networks : Taxonomy, challenges and solutions

Fetia Bannour

UNIVERSITÉ PARIS-EST

École doctorale Mathématiques et STIC

Mathématiques et Sciences et
Technologies de l'Information et de la Communication

Université Paris-Est Créteil

THÈSE

Pour l'obtention du grade de

Docteur de l'Université Paris-Est

Spécialité: "Informatique, Génie Informatique, Réseaux"

Présentée par

Fetia BANNOUR

Contributions pour le contrôle distribué dans les réseaux SDN

Directeur de thèse : **Abdelhamid MELLOUK**,
Professeur des Universités, LISSI, Université Paris-Est Créteil (UPEC), France

Co-encadrant de thèse : **Sami SOUIHI**
Maître de conférences des Universités, LISSI, Université Paris-Est Créteil (UPEC), France

Soutenue publiquement

le 19 Novembre 2019

Devant le jury d'examen composé de:

Gérardo Rubino,	Rapporteur	Directeur de Recherche, IRISA, INRIA, France
Olivier Festor,	Rapporteur	Professeur des Universités, LORIA, Telecom Nancy, France
Djamal Zeghlache,	Examinateur	Professeur, Télécom SudParis, Université Paris-Saclay, France
Samia Bouzefrane,	Examinatrice	Professeure des Universités, CEDRIC, CNAM, France
Thierry Divoux,	Examinateur	Professeur des Universités, CRAN, Université de Lorraine, France
Fabrice Guillemin,	Examinateur	Directeur de recherche, Orange Labs, France

LiSSiX **Laboratoire Images, Signaux et Systèmes Intelligents**

TINC Net **Équipe de contrôle intelligent dans les réseaux**

THESE

UNIVERSITÉ PARIS-EST

MSTIC doctoral school

Mathématiques et Sciences et
Technologies de l'Information et de la Communication

University of Paris-Est Créteil

DISSERTATION

Submitted in partial fulfillment of the requirements for the Degree of

Doctor of Philosophy of Paris-Est University

in "Computer Science, Computer Networking" Specialization

Presented by

Fetia BANNOUR

Software-Defined Networking: Extending SDN control to large-scale networks

PHD Thesis Advisor: **Abdelhamid MELLOUK**,
Full University Professor, LISSI, University of Paris-Est Créteil (UPEC), France

PHD Thesis co-Supervisor: **Sami SOUIHI**
Associate University Professor, LISSI, University of Paris-Est Créteil (UPEC), France

Defended publicly

on November 19, 2019

In front of a jury composed of:

Gérardo Rubino,	Reviewer	Research Director, IRISA, INRIA, France
Olivier Festor,	Reviewer	Full University Professor, LORIA, Telecom Nancy, France
Djamal Zeghlache,	Examiner	Full Professor, Telecom SudParis, University of Paris-Saclay, France
Samia Bouzefrane,	Examiner	Full University Professor, CEDRIC, CNAM, France
Thierry Divoux,	Examiner	Full University Professor, CRAN, University of Lorraine, France
Fabrice Guillemin,	Examiner	Research Director, Orange Labs, France

LiSSiX **Laboratory of Images, Signals and Intelligent Systems**

TINC Net **Team of Intelligent Control for Networks**

THESIS

Résumé

Le « Réseau défini par le logiciel », ou communément appelé le SDN, est un nouveau paradigme d'architecture réseau où le plan de contrôle est découplé du plan de données, et déplacé dans une entité centrale appelée le contrôleur SDN. Les architectures SDN centralisées soulèvent de nombreux défis d'évolutivité et de fiabilité. Pour y répondre, il est nécessaire de faire évoluer l'architecture SDN vers une approche de systèmes physiquement distribués, mais logiquement centralisés. Néanmoins, il faut lever les verrous inhérents à certains cas d'application. Cette thèse traite du problème de la décentralisation du plan de contrôle SDN dans le contexte des réseaux à large échelle. Après une étude de l'état de l'art et une classification des approches existantes, nous proposons trois nouvelles approches pour répondre à des défis majeurs associés à la décentralisation du plan de contrôle SDN dans les réseaux à large échelle. La première contribution aborde le problème de placement de contrôleurs SDN. Les stratégies mises en œuvre prennent en compte plusieurs critères d'évolutivité et de fiabilité pour le placement de contrôleurs SDN. Les deuxième et troisième contributions étudient le problème de cohérence des données dans un cluster SDN distribué en proposant des modèles de cohérence adaptatifs et continus. L'apport principal de ces deux contributions est de mettre au point une stratégie d'adaptation de cohérence qui permet, au moment de l'exécution, de trouver un compromis entre les exigences continues de l'application en termes de performance et de cohérence. Ces compromis devraient permettre de minimiser en temps réel la surcharge engendrée sur le réseau tout en satisfaisant les seuils définis par l'application qui peuvent être spécifiés dans les contrats de niveau de service. Ces modèles s'intéressent, dans un premier temps, au mécanisme de réconciliation Anti-Entropie qui s'adapte aux besoins réels en termes de cohérence, des applications SDN. Dans un second temps, ils s'intéressent aux stratégies de réplication en proposant un modèle intelligent basé sur le vote majoritaire (Quorum). Ces approches ont été validées en utilisant le contrôleur ONOS dans le cadre de deux applications SDN: une application de routage à la source et une application de délivrance de contenus.

Abstract

Software-Defined Networking (SDN) is an emerging network architecture paradigm that separates the network control and data planes, and moves the control logic to a centralized entity called the SDN controller. Centralized SDN designs raise many challenges including the issues of control plane scalability and reliability. To meet these challenges, it is necessary for the SDN control architecture to evolve towards a physically decentralized system. However, such physically-distributed, but logically-centralized, SDN designs bring an additional set of challenges. This thesis deals with the problem of decentralizing the SDN control plane in the context of large-scale networks. After a thorough state-of-the-art study on distributed SDN control followed by original classifications of existing SDN controller platforms, three novel approaches are proposed to tackle some of the most prominent challenges related to the decentralization of the SDN control plane in large-scale networks. The first approach addresses the SDN controller placement problem by proposing scalability and reliability aware strategies for the placement of the SDN controllers at scale using different types of multi-criteria optimization algorithms. The second and third approaches investigate the knowledge sharing problem in a distributed SDN cluster by proposing adaptive and continuous consistency models. The main aim of these two approaches is to achieve a consistency adaptation strategy that provides at runtime balanced trade-offs between the application's continuous performance and consistency requirements. These real-time trade-offs should provide minimal application inter-controller overhead while satisfying the application-defined thresholds specified in the application SLAs. These models focus, in the first place, on the Anti-Entropy reconciliation mechanisms that can adapt to the applications' real-time requirements in terms of performance and consistency. Then, they address the replication mechanisms by putting forward an intelligent Quorum-based replication strategy. These approaches were validated using two SDN applications with eventual consistency needs that are developed on top of the ONOS controllers: a source routing application and a CDN-like application.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface.

AS Autonomous System.

CAP Consistency Availability Performance.

CDN Content Delivery Network.

CLARA CLustering LARge Applications.

CPP Controller Placement Problem.

DDBS Distributed DataBase System.

DHT Distributed Hash Table.

DoS Denial-of-service.

ForCES Forwarding and Control Element Separation.

FSM Finite-State Machine.

IETF Internet Engineering Task Force.

IoT Internet of Things.

IXP Internet eXchange Point.

MD-SAL Model-Driven Service Abstraction Layer.

ML Machine Learning.

MOCO Multi-Objective Combinatorial Optimization.

NIB Network Information Base.

NSGA-II Non-dominated Sorting Genetic Algorithm II.

ODL OpenDayLight.

OF OpenFlow.

ONF Open Networking Foundation.

ONOS Open Network Operating System.

PACELC Consistency Availability Performance Else Latency Consistency.

PAM Partitioning Around Medoids.

POCO Pareto-Optimal COntroller.

QL Q-Learning.

QoE Quality of Experience.

QoS Quality of Service.

RL Reinforcement Learning.

RSM Replicated State Machine.

SDN Software-Defined Networking.

SDX Software-Defined eXchange.

SLA Service-Level Agreement.

SPOF Single Point of Failure.

TE Traffic Engineering.

WAN Wide Area Network.

XFSM eXtended Finite-State Machine.

List of publications

International journals

- **F. Bannour**, S. Souihi, A. Mellouk. "Distributed SDN Control: Survey, Taxonomy, and Challenges". *IEEE Communications Surveys and Tutorials (CST)*, 20(1):333–354, 2018.

International conferences

- **F. Bannour**, S. Souihi, A. Mellouk. "Adaptive Quorum-inspired SLA-Aware Consistency for Distributed SDN Controllers", *15th International Conference on Network and Service Management (CNSM)*, Halifax, Canada, 21-25 October, 2019.
- **F. Bannour**, S. Souihi, A. Mellouk. "Adaptive State Consistency for Distributed ONOS Controllers", *IEEE Global Communications Conference (GLOBECOM)*, Abu Dhabi, 9–13 December, 2018.
- **F. Bannour**, S. Souihi, A. Mellouk. "Scalability and Reliability Aware SDN Controller Placement Strategies", *13th International Conference on Network and Service Management (CNSM)*, Tokyo, Japan, 26-30 November, 2017.

National conferences and symposiums

- **F. Bannour**, S. Souihi, A. Mellouk. "Adaptive state consistency for distributed ONOS controllers", *2018 SDN DAY « IDNs (Intelligence-Defined Networks) »*, invited talk, November 22, 2018, Paris, France.
- **F. Bannour**, S. Souihi, A. Mellouk. "Software-Defined Networking: A self-adaptive consistency model for distributed SDN controllers", *2017 RESCOM summer school « Virtualisation & Dehardwarization »*, CNRS GDR RSD, 19-23 June, 2017, Le Croisic, France.

- **F. Bannour**, S. Souihi, A. Mellouk. "Software-Defined Networking: Distributed SDN control", *2017 ARC of the CNRS GDRMACS « Automation and Communication Networks »*, selected presentation after call for submission, May 16, 2017, Paris, France.
- **F. Bannour**, S. Souihi, A. Mellouk. "The SDN controller placement problem", *2016 RESCOM summer school « 5G and Internet of Things »*, CNRS GDR RSD, selected poster, 13-17 June, 2016, Guidel-plages, France.

Introduction

« We are all now connected by the Internet, like neurons in a giant brain »

Stephen Hawking

1 General context

The unprecedented growth in demands and data traffic, the emergence of network virtualization along with the ever-expanding use of mobile equipment in the modern network environment have highlighted major problems that are basically inherent to the Internet's conventional architecture. That made the task of managing and controlling the information coming from a growing number of connected devices increasingly complex and specialized.

Indeed, the traditional networking infrastructure is considered as highly rigid and static as it was initially conceived for a particular type of traffic, namely monotonous text-based contents, which makes it poorly suited to today's interactive and dynamic multimedia streams generated by increasingly-demanding users. Along with multimedia trends, the recent emergence of the Internet of Things (IoT) has allowed for the creation of new advanced services with more stringent communication requirements in order to support its innovative use cases. In particular, e-health is a typical IoT use case where the healthcare services delivered to remote patients (e.g. diagnosis, surgery, medical records) are highly intolerant of delay, quality and privacy. Such sensitive data and life-critical traffic are hardly supported by traditional networks.

Furthermore, in the traditional architecture where the control logic is purely distributed and localized, solving a specific networking problem or adjusting a particular network policy requires acting separately on the affected devices and manually changing their configuration. In this context, the current growth in devices and data has exacerbated

scalability concerns by making such human interventions and network operations harder and more error-prone.

Altogether, it has become particularly challenging for today's networks to deliver the required level of Quality of Service (QoS), let alone the Quality of Experience (QoE) that introduces additional user-centric requirements. To be more specific, relying solely on the traditional QoS that is based on technical performance parameters (e.g. bandwidth and latency) turns out to be insufficient for today's advanced and expanding networks. Additionally, meeting this growing number of performance metrics is a complex optimization task that can be treated as an NP-complete problem. Alternatively, network operators are increasingly realizing that the end-user's overall experience and subjective perception of the delivered services are as important as QoS-based mechanisms. As a result, current trends in network management are heading towards this new concept commonly referred to as the QoE to represent the overall quality of a network service from an end-user perspective.

That said, this huge gap between, on the one hand, the advances achieved in both computer and software technologies and on the other, the traditional non-evolving and *hard to manage* [1; 2] underlying network infrastructure supporting these changes has stressed the need for an automated networking platform [3] that facilitates network operations and matches today's network requirements such as the IoT needs [4]. In this context, several research strategies have been proposed to integrate automatic and adaptive approaches into the current infrastructure for the purpose of meeting the challenges of scalability, reliability and availability for real-time traffic, and therefore guaranteeing the user's QoE.

While radical alternatives argue that a brand-new network architecture should be built from scratch by breaking with the conventional network architecture and bringing fundamental changes to keep up with current and future requirements, other realistic alternatives are appreciated for introducing slight changes tailored to specific needs and for making a gradual network architecture transition without causing costly disruptions to existing network operations.

In particular, the early Overlay Network alternative introduces an application layer overlay on the top of the conventional routing substrate to facilitate the implementation of new network control approaches. However, the obvious disadvantage of Overlay Networks is that they depend on several aspects (e.g. selected overlay nodes) to achieve the

required performance. Besides, such networks can be criticized for compounding the complexity of existing networks due to the additional virtual layers.

On the other hand, the recent Software-Defined Networking (SDN) paradigm [5] offers the possibility to program the network and thus facilitates the introduction of automatic and adaptive control approaches by separating hardware (data plane) and software (control plane) enabling their independent evolution. SDN aims for the centralization of the network control, offering an improved visibility and a better flexibility to manage the network and optimize its performance. When compared to the Overlay Network alternative, SDN has the ability to control the entire network not only a selected set of nodes and to use a public network for transporting data. Besides, SDN spares network operators the tedious task of temporarily creating the appropriate overlay network for a specific use case. Instead, it provides an inherent programmatic framework for hosting control and security applications that are developed in a centralized way while taking into consideration the IoT requirements [4; 6] to guarantee the user's QoE.

2 Problem statement and motivations

Despite the great interest in SDN, its deployment in the industrial context is still in its relative early stages. There might be indeed a long road ahead before technology matures and standardization efforts pay off so that the full potential of SDN can be achieved.

Indeed, along with the hype and excitement, there have been several concerns and questions regarding the widespread adoption of SDN networks. For instance, research studies on the feasibility of the SDN deployment have revealed that the physical centralization of the control plane in a single programmable software component, called the controller, is constrained by several limitations such as the issues of scalability, availability and reliability. Gradually, it became inevitable to think about the control plane as a distributed system [7], where several SDN controllers are in charge of handling the whole network, while maintaining a logically centralized network view.

In that respect, networking communities argued about the best way to implement distributed SDN architectures while taking into account the new challenges brought by such distributed systems. Consequently, several SDN solutions have been explored and many SDN projects have emerged. Each proposed SDN controller platform adopted a specific architectural design approach based on various factors such as the aspects of interest, the performance goals, the deployed SDN use case, and also the trade-offs involved in the

presence of multiple conflicting and competing challenges.

At this point, we underline the importance of conducting a serious analysis of the proposed SDN solutions in envisioning the potential trends that may drive future research in the area. In particular, we place a special focus on distributed SDN control designs with the aim of solving some of the major challenges encountered in the decentralization of the SDN control planes in the context of large-scale deployments.

The main motivations of this work are the following:

- Ensuring a thorough understanding of existing state-of-the-art distributed SDN controller platforms, and developing a critical awareness of the ongoing and future key research and operational challenges facing the design and deployment of such platforms.

- Proposing novel approaches for decentralizing the SDN control plane in large-scale networks. Such a decentralized SDN control plane should be efficient (i.e. scalable, high-performance and robust) as it should meet the SDN controller application requirements (e.g scalability, reliability and consistency).

- Paving the way for the emergence of a new common standard for the distributed SDN control plane. That standard should also ensure the inter-controller communication between different vendor-specific controller technologies (i.e. the interoperability challenge).

3 Main contributions

In this section, we outline the main contributions of this work. More specifically, we propose novel approaches for decentralizing the Software-Defined Networking (SDN) control plane in large-scale networks while tackling some of the major associated challenges:

(1) *Scalability* and *reliability* aware strategies for the *placement of distributed SDN controllers* at scale using different types of multi-criteria optimization algorithms (see Chapter 3).

(2) An adaptive and continuous *consistency* model for the distributed SDN controllers: A novel Anti-Entropy reconciliation mechanism for applications (with eventual consistency needs) on top of the ONOS controllers (see Chapter 4).

(3) An adaptive and continuous *consistency* model for the distributed SDN controllers: A novel Quorum-based replication strategy for applications (with eventual consistency needs) on top of the ONOS controllers (see Chapter 5).

Additionally, given the lack of available literature on the subject of decentralized SDN control and given its relevance nowadays, our work also provides:

- A survey on distributed control in SDN: An overview and taxonomy of current SDN controller platforms (i.e. a physical classification and a logical classification) (see Chapter 1).
- A thorough analysis of the challenges encountered by the discussed state-of-the-art distributed SDN controller platforms, and the different approaches adopted for solving these challenges (see Chapter 2).

4 Dissertation organization

The remainder of this dissertation is organized as follows:

Chapter 1 This chapter presents a survey on SDN with a special focus on distributed SDN control solutions. In addition to explaining the fundamental elements of the SDN architecture, this chapter proposes a taxonomy of the most prominent state-of-the art SDN controllers platforms by classifying them in two different ways: a physical classification and a logical classification.

Chapter 2 This chapter provides a thorough analysis of the major open challenges faced by the existing distributed SDN controller platforms discussed in the previous chapter. These challenges include the issues of scalability, reliability, consistency and interoperability of the SDN control plane. Besides, this chapter explores the potential approaches to tackle these challenges for an optimal SDN deployment, and it provides some useful insights into the emerging and future trends in the design of efficient distributed SDN control planes.

Chapter 3 This chapter addresses the distributed SDN control problem by tackling the SDN controller placement problem in large-scale IoT-like networks. It puts forwards novel scalability an reliability aware controller placement strategies that tackle several aspects of the controller placement optimization problem with respect to multiple reliability and performance criteria and according to different uses and contexts. These strategies use two different types of heuristic-based algorithms: a clustering algorithm based

on PAM and a modified genetic algorithm called NSGA-II. These multi-criteria algorithms are compared in terms of computation time, as well as the quality of final controller placement configurations.

Chapter 4 This chapter addresses the distributed SDN control problem by tackling the knowledge sharing problem between the distributed SDN controllers. It proposes an adaptive multi-level consistency model following the concept of continuous consistency for the distributed SDN controllers. That approach is implemented for a source routing application on top of the open-source ONOS controllers. It consists in turning ONOS's eventual consistency model into an adaptive consistency model using the *Anti-Entropy reconciliation period* as a control knob for an adaptive fine-grained tuning of consistency levels. Our proposed consistency strategy is aimed at ensuring the application's continuous consistency requirements (i.e. Numerical Error bounds) as specified in the given application SLA. Its purpose is also to minimize the Anti-Entropy reconciliation overhead as compared to ONOS's static consistency scheme at scale.

Chapter 5 This chapter further addresses the knowledge sharing problem in the distributed SDN control by proposing an adaptive and continuous consistency model for the distributed ONOS controllers. The approach is implemented for a CDN-like application on top of ONOS. It consists in changing ONOS's eventual consistency model to an adaptive consistency model by turning ONOS's optimistic replication technique into a more scalable replication strategy following Quorum-replicated consistency. The main focus is placed at improving ONOS's replication mechanism: It uses the *read and write Quorum parameters* as adjustable control knobs for a fine-grained consistency tuning, rather than relying on Anti-Entropy reconciliation mechanisms (see previous Chapter). The main objective is to find at runtime optimal partial Quorum configurations that achieve, under changing network and workload conditions, balanced trade-offs between the application's continuous performance (latency) and consistency (staleness) requirements. These real-time trade-offs should provide minimal application inter-controller overhead while satisfying the application-defined thresholds specified in the given application SLA.

Conclusions and perspectives The last chapter concludes this dissertation and gives an insight into our ongoing and future work and perspectives in the area of distributed SDN control.

Chapter 1

Towards a decentralized SDN control architecture: Overview and taxonomy

«The Future of Networking, and the Past of Protocols»

Scott Shenker,
Open Network Summit, 2011

Contents

1.1 Introduction

As opposed to the decentralized control logic underpinning the devising of the Internet as a complex bundle of box-centric protocols and vertically-integrated solutions, the Software-Defined Networking (SDN) paradigm advocates the separation of the control logic from hardware and its centralization in software-based controllers. These key tenets offer new opportunities to introduce innovative applications and incorporate automatic and adaptive control aspects, thereby easing network management and guaranteeing the user's QoE.

However, despite the excitement, SDN adoption raises many challenges including the scalability and reliability issues of centralized designs that can be addressed with the physical decentralization of the control plane. However, such physically-distributed, but logically centralized systems, bring an additional set of challenges.

This chapter presents a survey on SDN with a special focus on the distributed SDN control. In Section 1.2 , we start by exposing the promises and solutions offered by SDN as compared to conventional networking. We also elaborate on the fundamental elements of the SDN architecture.

Then, we expand our knowledge of the different approaches to SDN by exploring the wide variety of existing SDN controller platforms. In doing so, we intend to place a special emphasis on distributed SDN solutions and classify them in two different ways: In Section 1.3, we propose a physical classification of state-of-the-art SDN control plane architectures into centralized and distributed (Flat or Hierarchical) in order to highlight the SDN performance, scalability and reliability challenges. In Section 1.4, we put forward a logical classification of distributed SDN control plane architectures into logically-centralized and logically-distributed while tackling the associated state consistency and knowledge dissemination issues.

1.2 Software-defined networking: A centralized control architecture

1.2.1 Conventional networking and the SDN paradigm

Over the last few years, the need for a new approach to networking has been expressed to overcome the many issues associated with current networks. In particular, the main vision of the SDN approach is to simplify networking operations, optimize network man-

agement and introduce innovation and flexibility as compared to legacy networking architectures.

In this context and in line with the vision of Kim *et al.* [8], four key reasons for the problems encountered in the management of existing networks can be identified:

(i) *Complex and low-level Network configuration*

Network configuration is a complex distributed task where each device is typically configured in a low-level vendor-specific manner. Additionally, the rapid growth of the network together with the changing networking conditions have resulted in network operators constantly performing manual changes to network configurations, thereby compounding the complexity of the configuration process and introducing additional configuration errors.

(ii) *Dynamic Network State*

Networks are growing dramatically in size, complexity and consequently in dynamicity. Furthermore, with the rise of mobile computing trends as well as the advent of network virtualization [9] and cloud computing [10; 11], the networking environment becomes even more dynamic as hosts are continually moving, arriving and departing due to the flexibility offered by VM migration, and thus making traffic patterns and network conditions change in a more rapid and significant way.

(iii) *Exposed Complexity*

In today's large-scale networks, network management tasks are challenged by the high complexity exposed by distributed low-level network configuration interfaces. That complexity is mainly generated by the tight coupling between the management, control, and data planes, where many control and management features are implemented in hardware.

(iv) *Heterogeneous Network Devices*

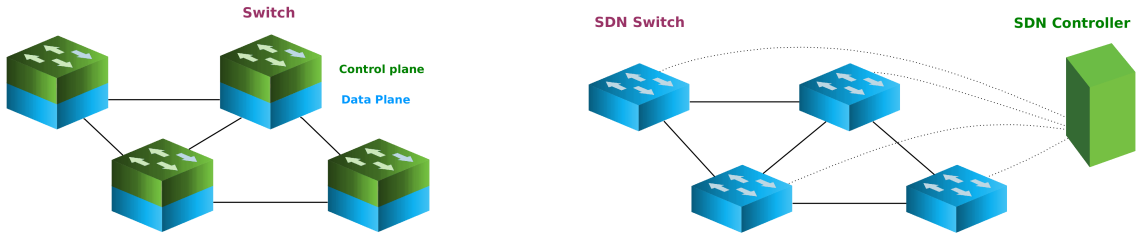
Current networks are comprised of a large number of heterogeneous network devices including routers, switches and a wide variety of specialized middle-boxes. Each of these appliances has its own proprietary configuration tools and operates according to specific protocols, therefore increasing both complexity and inefficiency in network management.

All that said, network management is becoming more difficult and challenging given

that the static and inflexible architecture of legacy networks is ill-suited to cope with today's increasingly dynamic networking trends, and to meet the QoE requirements of modern users. This fact has fueled the need for the enforcement of complex and high-level policies to adapt to current networking environments, and for the automation of network operations to reduce the tedious workload of low-level device configuration tasks.

In this sense, and to deliver the goals of easing network management in real networks, operators have considered running dynamic scripts as a way to automate network configuration settings before realizing the limitations of such approaches which led to misconfiguration issues. It is, however, worth noting, that recent approaches to scripting configurations and network automation are becoming relevant [12].

The SDN initiative led by the Open Networking Foundation (ONF) [13], on the other hand, proposes a new open architecture to address current networking challenges with the potential to facilitate the automation of network configurations , and better yet, fully program the network. Unlike the conventional distributed network architecture (Figure 1.1(a)) where network devices are closed and vertically-integrated bundling software with hardware, the SDN architecture (Figure 1.1(b)) raises the level of abstraction by separating the network data and control planes. That way, network devices become simple forwarding switches whereas all the control logic is centralized in software controllers providing a flexible programming framework for the development of specialized applications and for the deployment of new services.



(a) Traditional architecture (b) SDN architecture

Figure 1.1: Conventional networking Versus software-defined networking

Such aspects of SDN are believed to simplify and improve network management by offering the possibility to innovate, customize behaviors and control the network according to high-level policies expressed as centralized programs, therefore bypassing the complexity of low-level network details and overcoming the fundamental architectural problems raised in (i) and (iii). Added to these features is the ability of SDN to easily cope with

the heterogeneity of the underlying infrastructure (outlined in (iv)) thanks to the SDN Southbound interface abstraction.

More detailed information on the SDN-based architecture which is split vertically into three layers (see Figure 1.2) is provided in the next section.

1.2.2 The SDN architecture

The SDN-based architecture is split vertically into three layers (see Figure 1.2). Detailed information about the SDN architecture is provided in the following subsections:

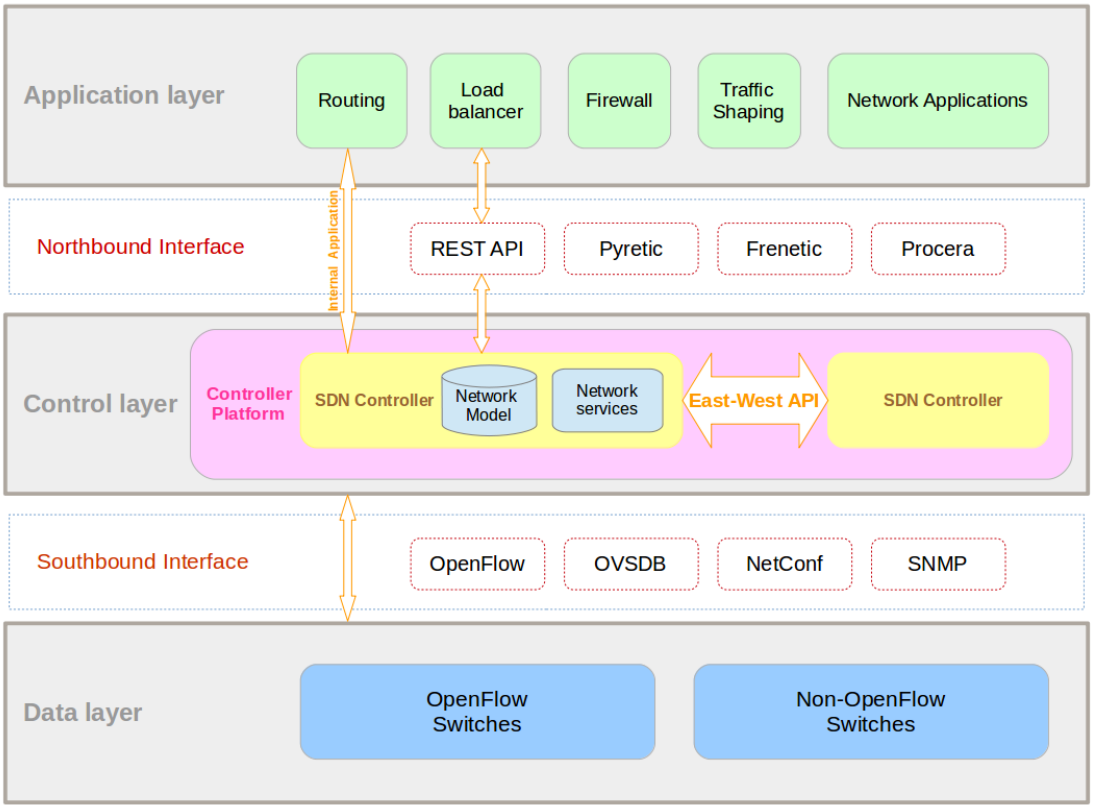


Figure 1.2: A three-layer distributed SDN architecture

1.2.2.1 SDN data plane

The data plane, also known as the forwarding plane, consists of a distributed set of forwarding network elements (mainly *switches*) in charge of forwarding packets. In the context of SDN, the control-to-data plane separation feature requires the data plane to be remotely accessible for software-based control via an open vendor-agnostic Southbound interface.

Both OpenFlow [14] and ForCES [15] are well-known candidate protocols for the Southbound interface. They both follow the basic principle of splitting the control plane and the forwarding plane in network elements and they both standardize the communication between the two planes. However, these solutions are different in many aspects, especially in terms of network architecture design.

Standardized by IETF, ForCES (Forwarding and Control Element Separation) [15] introduced the separation between the control plane and the forwarding plane. In doing so, ForCES defines two logic entities that are logically kept in the same physical device: the Control Element (CE) and the Forwarding Element (FE). However, despite being a mature standard solution, the ForCES alternative did not gain widespread adoption by major router vendors.

On the other hand, OpenFlow [14] received major attention in both the research community and the industry. Standardized by the ONF [13], it is considered as the first widely accepted communication protocol for the SDN Southbound interface. OpenFlow enables the control plane to specify in a centralized way the desired forwarding behavior of the data plane. Such traffic forwarding decisions reflect the specified network control policies and are translated by *controllers* into actual packet forwarding rules populated in the flow tables of OpenFlow *switches*.

In more specific terms, and according to the original version 1.0.0 of the standard defined in [16], an OpenFlow-enabled Switch consists of *a flow table* and an OpenFlow *secure channel* to an external OpenFlow controller. Typically, the forwarding table maintains a list of flow entries; Each flow entry comprises *match fields* containing header values to match packets against, *counters* to update when packets match for flow statistics collection purposes, and a set of *actions* to apply to matching packets.

Accordingly, all incoming packets processed by the switch are compared against the flow table where flow entries match packets based on a priority order specified by the controller. In case a matching entry is found, the flow counter is incremented and the actions associated with the specific flow entry are performed on the incoming packet belonging to that flow. According to the OpenFlow specification [16], these actions may include forwarding a packet out on a specific port, dropping the packet, removing or updating packet headers, etc. If no match is found in the flow table, then the unmatched packet is encapsulated and sent over the secure channel to the controller which decides on the way it should be processed. Among other possible actions, the controller may define a new flow

for that packet by inserting new flow table entries.

1.2.2.2 SDN control plane

Regarded as the most fundamental building entity in SDN architecture, the control plane consists of a centralized software controller that is responsible for handling communications between network applications and devices through open interfaces. More specifically, SDN controllers translate the requirements of the application layer down to the underlying data plane elements and give relevant information up to SDN applications.
The SDN control layer is commonly referred to as the Network Operating System (NOS) as it supports the network control logic and provides the application layer with an abstracted view of the global network, which contains enough information to specify policies while hiding all implementation details.

Typically, the control plane is logically centralized and yet implemented as a physically distributed system for scalability and reliability reasons as discussed in Sections 1.3 and 1.4. In a distributed SDN control configuration, East-Westbound APIs [17] are required to enable multiple SDN controllers to communicate with each other and exchange network information.
Despite the many attempts to standardize SDN protocols, there has been to date no standard for the East-West API which remains proprietary for each controller vendor. Although a number of East-Westbound communications happen only at the data-store level and do not require additional protocol specifics, it is becoming increasingly advisable to standardize that communication interface in order to provide wider interoperability between different controller technologies in different autonomous SDN networks.

On the other hand, an East-Westbound API standard requires advanced data distribution mechanisms and involves other special considerations. This brings about additional SDN challenges, some of which have been raised by the state-of-the art distributed controller platforms discussed in Sections 1.3 and 1.4, but have yet to be fully addressed.

1.2.2.3 SDN application plane

The SDN application plane comprises SDN applications which are control programs designed to implement the network control logic and strategies. This higher-level plane interacts with the control plane via an open Northbound API. In doing so, SDN applications communicate their network requirements to the SDN controller which translates

them into Southbound-specific commands and forwarding rules dictating the behavior of the individual data plane devices. Routing, Traffic Engineering (TE), firewalls and load balancing are typical examples of common SDN applications running on top of existing controller platforms.

In the context of SDN, applications leverage the decoupling of the application logic from the network hardware along with the logical centralization of the network control, to directly express the desired goals and policies in a centralized high-level manner without being tied to the implementation and state-distribution details of the underlying networking infrastructure. Concurrently, SDN applications make use of the abstracted network view exposed through the Northbound interface to consume the network services and functions provided by the control plane according to their specific purposes.

That being said, the Northbound API implemented by SDN controllers can be regarded as a network abstraction interface to applications, easing network programmability, simplifying control and management tasks and allowing for innovation. In contrast to the Southbound API, the Northbound API is not supported by an accepted standard. Despite the broad variety of Northbound APIs adopted by the SDN community (see Figure 1.2), we can classify them into two main categories:

- The first set involves simple and primitive APIs that are directly linked to the internal services of the controller platform. These implementations include:
  - Low-level ad-hoc APIs that are proprietary and tightly dependent on the controller platform. Such APIs are not considered as high-level abstractions as they allow developers to directly implement applications within the controller in a low-level manner. Deployed internally, these applications are tightly coupled with the controller and written in its native general-purpose language. NOX in C++ and POX in Python are typical examples of controllers that use their own basic sets of APIs.
  - APIs based on Web services such as the widely-used REST API. This group of programming interfaces enables independent external applications (*Clients*) to access the functions and services of the SDN controller (*Server*). These applications can be written in any programming language and are not run inside the bundle hosting the controller software. Floodlight is an example of an SDN

controller that adopts an embedded Northbound API based on REST.

- The second category contains higher level APIs that rely on domain-specific programming languages such as Frenetic [18], Procera [19] and Pyretic [20] as an indirect way for applications to interact with the controller. These APIs are designed to raise the level of abstraction in order to allow for the flexible development of applications and for the specification of high-level network policies.

1.3 Physical classification of existing SDN control plane architectures

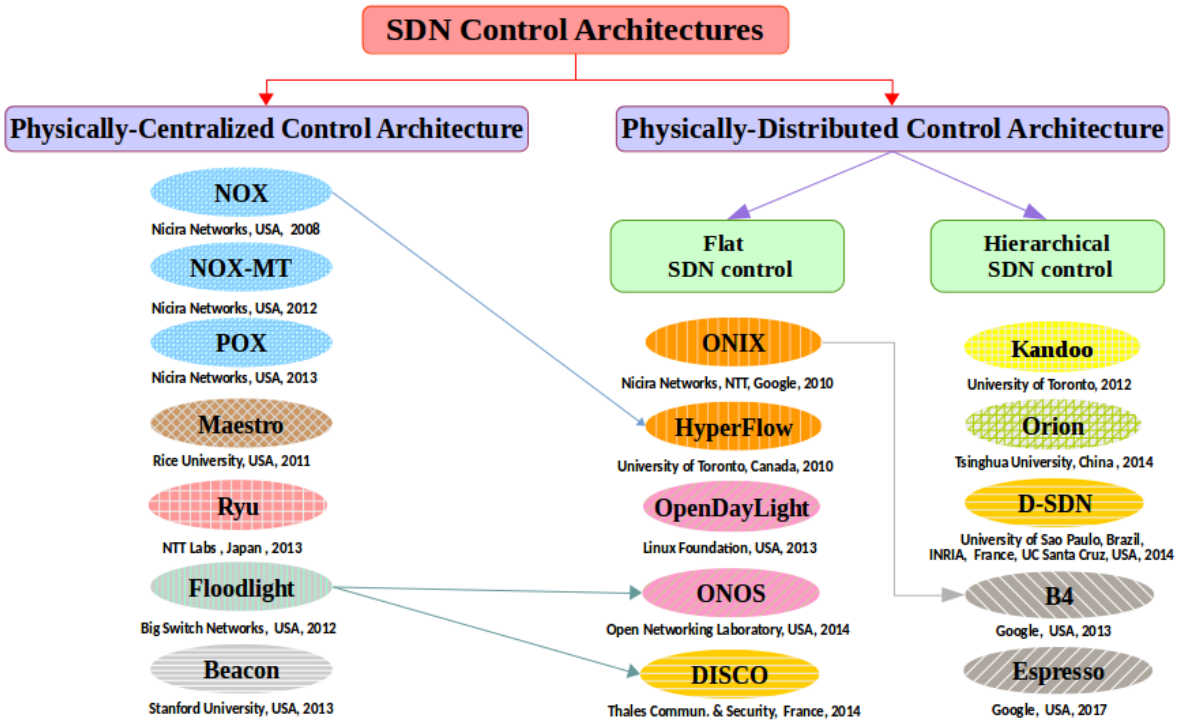


Figure 1.3: Physical classification of SDN control plane architectures

Despite the undeniable strengths of SDN, there have always been serious concerns about the ability to extend SDN to large-scale networks.

Some argue that these scalability limits are basically linked to the protocol standards being used for the implementation of SDN. OpenFlow [14] in particular, although recognized as a leading and widely-deployed SDN Southbound technology, is currently being rethought for potentially causing excessive overheads on switches (*switch bottleneck*). Scalable alternatives to the OpenFlow standard which propose to revisit the delegation of

control between the controller and the switches with the aim of reducing the reliance on SDN the control plane, have been discussed in 1.2.2.1.

Another entirely different approach to addressing the SDN scalability and reliability challenges, which is advocated by the present work, is to physically distribute the SDN control plane. This has led to a first categorization of existing controller platforms into centralized and distributed architectures (see Figure 1.3). Please note that, in Figure 1.3 and Figure 1.4, controllers that present similar characteristics for the discussed comparison criteria are depicted in the same color.

1.3.1 Physically-centralized SDN control

A physically-centralized control plane consisting of a single controller for the entire network is a theoretically perfect design choice in terms of simplicity. However, a single controller system may not keep up with the growth of the network. It is likely to become overwhelmed (*controller bottleneck*) while dealing with an increasing number of requests and concurrently struggling to achieve the same performance guarantees.

Obviously, a centralized SDN controller does not meet the different requirements of large-scale real-world network deployments. Data Centers and Service Provider Networks are typical examples of such large-scale networks presenting different requirements in terms of scalability and reliability.

More specifically, a *Data Center Network* involves tens of thousands of switching elements. Such a great number of forwarding elements which can grow at a fast pace is expected to generate a huge number of control events that are enough to overload a single centralized SDN controller [21; 22]. Studies conducted in [23] show important scalability implications (in terms of throughput) for centralized controller approaches. They demonstrate that multiple controllers should be used to scale the throughput of a centralized controller and meet the traffic characteristics within realistic data centers.

Unlike data centers, *Service Provider Networks* are characterized by a modest number of network nodes. However, these nodes are usually geographically distributed making the diameter of these networks very large [21]. This entails a different type of controller scalability issues for centralized controller approaches, more specifically, high latencies. In addition to latency requirements, service provider networks have large numbers of flows that may generate overhead and bandwidth issues.

In general, Wide Area Network (WAN) deployments typically impose strict resiliency

requirements. In addition, they present higher propagation delays as compared to data center networks. Obviously, a centralized controller design in a SD-WAN cannot achieve the desired failure resiliency and scale-out behaviors [24]. Several studies have emphasized the need for a distributed control plane in a SD-WAN architecture: They indeed focused on placing multiple controllers on real WAN topologies to benefit both control plane latency and fault-tolerance [25; 26].

That said, the potential scalability, reliability and vulnerability concerns associated with centralized controller approaches have been further confirmed through studies [27; 28] on the behavior of state-of-the-art centralized SDN controllers such as NOX [29], Beacon [30] and Floodlight [31] in different networking environments.

In particular, NOX classic [29], the world's first-generation OpenFlow controller with an event-based programming model, is believed to be limited in terms of throughput. Indeed, it cannot handle a large number of flows, namely a rate of 30k flow initiation events per second [28; 32]. Such a flow setup throughput may sound sufficient for an enterprise network, but, it could be arguable for data-center deployments with high flow initiation rates [23]. Improved versions of NOX have been consequently developed by the same community (Nicira Networks) such as NOX-MT [33] for better performance and POX [34] for a more developer-friendly environment.

However, while none of these centralized designs is believed to meet the above scalability and reliability requirements of large-scale networks, they have gained greater prominence as they were widely used for research and educational purposes.

Additionally, Floodlight [31] which is a very popular Java-based OpenFlow controller from Big Switch Networks, suffers from serious security and resiliency issues. For instance, Dhawan *et al.* [35] have reported that the centralized SDN controller is inherently susceptible to Denial-of-Service (DoS) attacks. Another subsequent version of Floodlight, called SE-Floodlight, has therefore been released to overcome these problems by integrating security applications. However, despite the introduced security enhancements aimed at shielding the centralized controller, the latter remains a potential weakness compromising the whole network. In fact, the controller still maintains a single point of failure and bottlenecks even if its latest version is less vulnerable to malicious attacks.

On the other hand, given its obvious performance and functionality advantages, the open-source Floodlight has been extensively used to build other SDN controller platforms supporting distributed architectures such as ONOS [36] and DISCO [37].

1.3.2 Physically-distributed SDN control

Alternatively, physically-distributed control plane architectures have received increased research attention in recent years since they appeared as a potential solution to mitigate the issues brought about by centralized SDN architectures (poor scalability, Single Point of Failure (SPOF), performance bottlenecks, etc). As a result, various SDN control plane designs have been proposed in recent literature. Yet, we discern two main categories of distributed SDN control architectures based on the physical organization of SDN controllers: A flat SDN control architecture and a hierarchical SDN control architecture (see Figure 1.3).

1.3.2.1 Flat SDN control

The flat structure implies the horizontal partitioning of the network into multiple areas, each of which is handled by a single controller in charge of managing a subset of SDN switches. There are several advantages to organizing controllers in such a flat style, including reduced control latency and improved resiliency.

Onix [38], Hyperflow [39] and ONOS [36] are typical examples of flat physically-distributed controller platforms which are initially designed to improve control plane *scalability* through the use of multiple interconnected controllers sharing a global network-wide view and allowing for the development of centralized control applications. However, each of these contributions takes a different approach to distribute controller states and providing control plane scalability.

For example, Onix provides a good scalability through additional partitioning and aggregation mechanisms. To be more specific, Onix partitions the NIB (Network Information Base) giving each controller instance responsibility for a subset of the NIB and it aggregates by making applications reduce the fidelity of information before sharing it between other Onix instances within the cluster. Similar to Onix, each ONOS instance (composing the cluster) that is responsible for a subset of network devices holds a portion of the network view that is also represented in a graph. Different from Onix and ONOS, every controller in HyperFlow has the global network view, thus getting the illusion of control over the whole network. Yet, HyperFlow can be considered as a scalable option for specific policies in which a small number of network events affect the global network state. In that case, scalability is ensured by propagating these (less frequent) se-

lected events through the event propagation system.

Furthermore, different mechanisms are put in place by these distributed controller platforms to meet *fault-tolerance* and *reliability* requirements in the event of failures or attacks.

Onix [38] uses different recovery mechanisms depending on the detected failures. Onix instance failure is most of the time handled by distributed coordination mechanisms among replicas whereas network element/link failures are under the full responsibility of applications developed atop Onix. Besides, Onix is assumed reliable when it comes to connectivity infrastructure failures as it can dedicate the failure recovery task to a separate management backbone that uses a multi-pathing protocol.

Likewise, Hyperflow [39] focuses on ensuring resiliency and fault tolerance as a means for achieving availability. When a controller failure is discovered by the failure detection mechanisms deployed by its publish/subscribe WheelFS [40] system, HyperFlow reconfigures the affected switches and redirects them to another nearby controller instance (from a neighbor's site). Alongside this ability to tackle component failures, HyperFlow is resilient to network partitioning thanks to the partition tolerance property of WheelFS. In fact, in the presence of a network partitioning, WheelFS partitions continue to operate independently, thus favoring availability.

Similarly, ONOS [36] considers fault-tolerance as a prerequisite for adopting SDN in Service Provider networks. ONOS's distributed control plane guards against controller instance failures by connecting, from the onset, each SDN switch to more than one SDN controller; its master controller and other backup controllers (from other domains) that may take over in the wake of master controller failures. Load balancing mechanisms are also provided to balance the mastership of switches among the controllers of the cluster for scalability purposes. Besides, ONOS incorporates additional recovery protocols, such as the Anti-Entropy protocol [41], for healing from lost updates due to such controller crashes.

Recent SDN controller platform solutions [42–47] focused specifically on improving fault-tolerance in the distributed SDN control plane. Some of these works assumed a simplified flat design where the SDN control was centralized. However, since the main focus was placed at the fault-tolerance aspect, we believe that their ideas and their fault-tolerance approaches can be leveraged in the context of medium to large scale SDNs

where the network control is physically distributed among multiple controllers.

In particular, Botelho et. al [48] developed a hybrid SDN controller architecture that combines both passive and active replication approaches for achieving control plane fault-tolerance. SMaRtLight adopts a simple Floodlight [31]-based multi-controller design following OpenFlow 1.3, where one main controller (the primary) manages all network switches, and other controller replicas monitor the primary controller and serve as backups in case it fails.

This variant of a traditional passive replication system relies on an external data store that is implemented using a modern active Replicated State Machine (RSM) built with a Paxos-like protocol (BFT-SMaRt [49]) to ensure fault-tolerance and strong consistency. This shared data store is used for storing the network and application state (the common global NIB) and also for coordinating fault detection and leader election operations between controller replicas that run a lease management algorithm.

In case of a failure of the primary controller, the elected backup controller starts reading the current state from the shared consistent data store in order to mitigate the cold-start (empty state) issue associated with traditional passive replication approaches, and thereby ensure a smoother transition to the new primary controller role.

The limited feasibility of the deployed controller fault-tolerance strategy is warranted by the limited scope of the SMaRtLight solution which is only intended for small to medium-sized SDN networks. On the other hand, in large-scale deployments, adopting a simplified Master-Slave approach, and more importantly, assuming a single main controller scheme where one controller replica must retrieve all the network state from the shared data store in failure scenarios, have major disadvantages in terms of increased latency and failover time.

Similarly, the Ravana controller platform proposal [44] addresses the issue of recovering from complete fail-stop controller crashes. It offers the abstraction of a fault-free centralized SDN controller to unmodified control applications which are relieved of the burden of handling controller failures. Accordingly, network programmers write application programs for a single main controller and the transparent master-slave Ravana protocol takes care of replicating, seamlessly and consistently, the control logic to other backup controllers for fault-tolerance.

The Ravana approach deploys enhanced Replicated State Machine (RSM) techniques that are extended with switch-side mechanisms to ensure that control messages are pro-

cessed transactionally with ordered and exactly-once semantics even in the presence of failures. The three Ravana prototype components, namely the Ryu [50]-based controller runtime, the switch runtime, and the control channel interface, work cooperatively to guarantee the desired correctness and robustness properties of a fault-tolerant logically centralized SDN controller.

More specifically, when the master controller crashes, the Ravana protocol detects the failure within a short failover time and elects the standby slave controller to take over using Zookeeper [51]-like failure detection and leader election mechanisms. The new leader finishes processing any logged events in order to catch up with the failed master controller state. Then, it registers with the affected switches in the role of the new master before proceeding with normal controller operations.

1.3.2.2 Hierarchical SDN control

The hierarchical SDN control architecture assumes that the network control plane is vertically partitioned into multiple levels (layers) depending on the required services. According to [52], a hierarchical organization of the control plane can improve SDN scalability and performance.

To improve *scalability*, Kandoo [53] assumes a hierarchical two-layer control structure that partitions control applications into local and global. Contrary to DevoFlow [54] and DIFANE [55], Kandoo proposes to reduce the overall stress on the control plane without the need to modify OpenFlow switches. Instead, it establishes a two-level hierarchical control plane, where frequent events occurring near the data path are handled by the bottom layer (local controllers with no interconnection running local applications) and non-local events requiring a network-wide view are handled by the top layer (a logically centralized root controller running non-local applications and managing local controllers).

Despite the obvious scalability advantages of such a control plane configuration where local controllers can scale linearly as they do not share information, Kandoo did not envision *fault-tolerance* and resiliency strategies to protect itself from potential failures and attacks in the data and control planes. Besides, from a developer perspective, Kandoo imposes some kandoo-specific conditions on the control applications developed on top of it, in such a way that makes them aware of its existence.

On the other hand, Google's B4 [56; 57], a private intra-domain software-defined WAN

connecting their data centers across the planet, proposes a two-level hierarchical control framework for improving *scalability*. At the lower layer, each data-center site is handled by an Onix-based [38] SDN controller hosting local site-level control applications. These site controllers are managed by a global *SDN Gateway* that collects network information from multiple sites through site-level TE services and sends them to a logically centralized *TE server* which also operates at the upper layer of the control hierarchy. Based on an abstract topology, the latter enforces high-level TE policies that are mainly aimed at optimizing bandwidth allocation between competing applications across the different data-center sites. That being said, the TE server programs these forwarding rules at the different sites through the same gateway API. These TE entries will be installed into higher-priority switch forwarding tables alongside the standard shortest-path forwarding tables. In this context, it is worth mentioning that the *topology abstraction* which consists in abstracting each site into a *super-node* with an aggregated *super-trunk* to a remote site is key to improving the scalability of the B4 network. Indeed, this abstraction hides the details and complexity from the logically centralized TE controller, thereby allowing it to run protocols at a coarse granularity based on a global controller view and, more importantly preventing it from becoming a serious performance bottleneck.

Unlike Kandoo [53], B4 [56] deploys robust *reliability* and *fault-tolerance* mechanisms at both levels of the control hierarchy in order to enhance the B4 system availability. These mechanisms have been especially enhanced after experiencing a large-scale B4 outage. In particular, Paxos [58] is used for detecting and handling the primary controller failure within each data-center site by electing a new leader controller among a set of reachable standby instances. On the other hand, network failures at the upper layer are addressed by the logically centralized TE controller which adapts to failed or unresponsive site controllers in the bandwidth allocation process. Additionally, B4 is resilient against other failure scenarios where the upper-level TE controller encounters major problems in reaching the lower-level site controllers (e.g. TE operation/session failures). Moreover, B4 guards against the failure of the logically centralized TE controller by geographically replicating TE servers across multiple WAN sites (one master TE server and four secondary hot standbys). Finally, another fault recovery mechanism is used in case the TE controller service itself faces serious problems. That mechanism stops the TE service and enables the standard shortest-path routing mechanism as an independent service.

In the same spirit, Espresso [59] is another interesting SDN contribution that represents the latest and more challenging pillar of Google's SDN strategy. Building on the previous three layers of that strategy (the *B4* WAN [56], the *Andromeda* NFV stack and the *Jupiter* data center interconnect), *Espresso* extends the SDN approach to the peering edge of Google's network where it connects to other networks worldwide. Considered as a large-scale SDN deployment for the public Internet, Espresso, which has been in production for more than two years, routes over 22% of Google's total traffic to the Internet. More specifically, the Espresso technology allows Google to dynamically choose from where to serve content for individual users based on real-time measurements of end-to-end network connections.

To deliver unprecedented *scale-out* and efficiency, Espresso assumes a hierarchical control plane design split between *Global controllers* and *Local controllers* that perform different functions. Besides, Espresso's software programmability design principle externalizes features into software thereby exploiting commodity servers for scale.

Moreover, Espresso achieves higher availability (*reliability*) when compared to existing router-centric Internet protocols. Indeed, it supports a fail static system, where the local data plane keeps the last known good state to allow for control plane unavailability without impacting data plane and BGP peering operations. Finally, another important feature of Espresso is that it provides full *interoperability* with the rest of the Internet and the traditional heterogeneous peers.

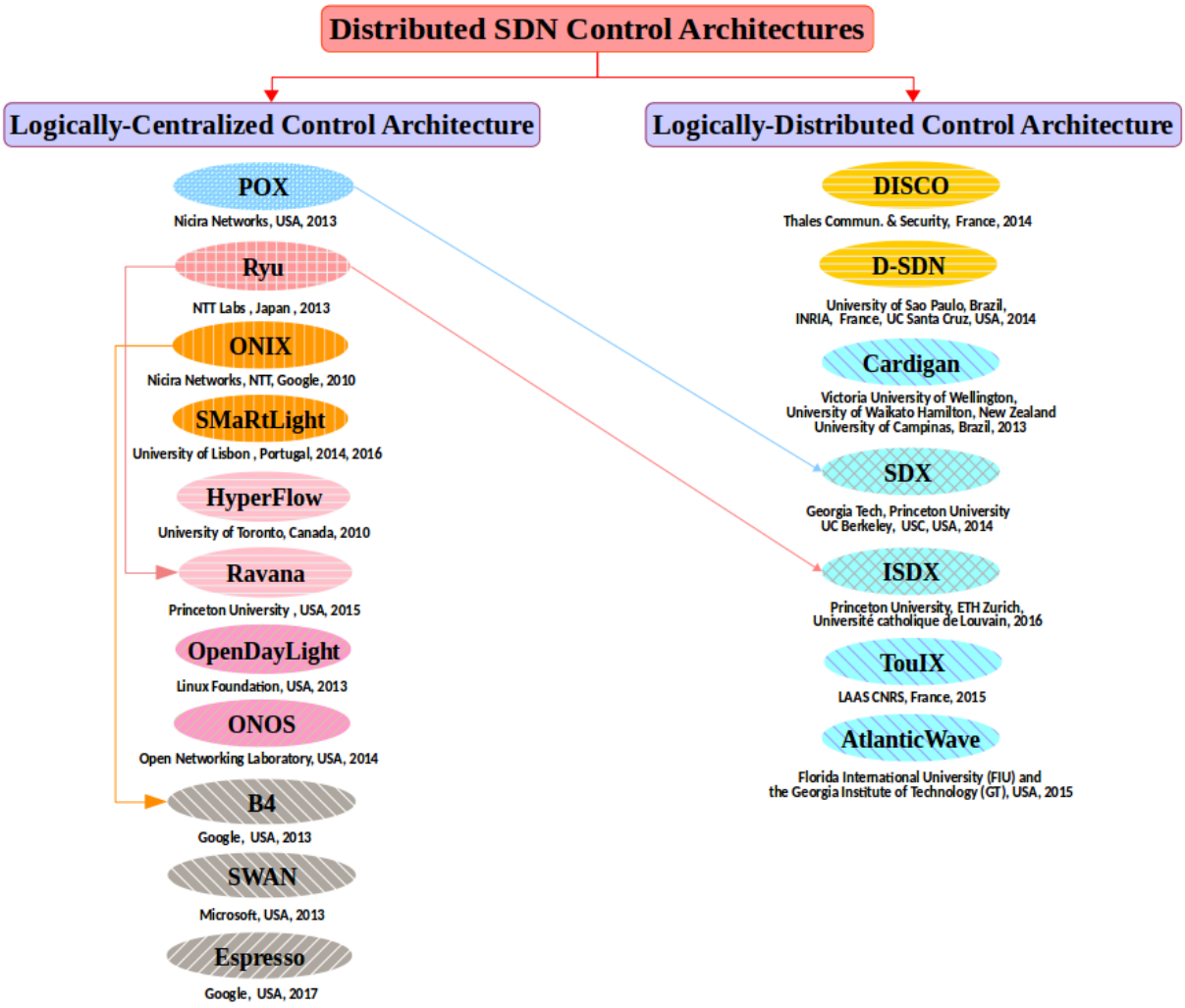


Figure 1.4: Logical classification of distributed SDN control plane architectures

1.4 Logical classification of existing SDN control plane architectures

Apart from the physical classification, we can categorize distributed SDN control architectures according to the way knowledge is disseminated among controller instances (the *consistency* challenge) into logically centralized and logically distributed architectures (see Figure 1.4). This classification has been recently adopted by [60].

1.4.1 Logically-centralized SDN control

1.4.1.1 Onix and SMaRtLight

Both Onix [38] and SMaRtLight [48] are logically centralized controller platforms that achieve controller state redundancy through state replication. But the main difference is that Onix uses a distributed data store while SMartLight uses a centralized data store

for replicating the shared network state. They also deploy different techniques for sharing knowledge and maintaining a consistent network state.

Onix is a distributed control platform for large-scale production networks that stands out from previous proposals by providing a simple general-purpose API, a central NIB abstraction and standard state distribution primitives for easing the implementation of network applications.

In more specific terms, Onix uses the NIB data structure to store the global network state (in the form of a network graph) that is distributed across running Onix instances and synchronized through Onix's built-in state distribution tools according to different levels of consistency as dictated by application requirements. In fact, besides interacting with the NIB at run-time, network applications on top of Onix initially configure their own data storage and dissemination mechanisms by choosing among two data-store options already implemented by Onix in the NIB: A replicated transactional database that guarantees strong consistency at the cost of good performance for persistent but slowly-changing data (state), and a high-performance memory-only distributed hash table (DHT) for volatile data that does not require strict consistency.

While the main advantage of Onix is its programmatic framework created for the flexible development of control applications with desired trade-offs between performance and state consistency (strong/eventual), it carries the limitations of eventually consistent systems which rely on application-specific logic to detect network state inconsistencies for the eventually-consistent data and provide conflict resolution methods for handling them.

As mentioned in Section 1.3.2.1, SMaRtLight is a fault-tolerant logically centralized Master-Slave SDN controller platform, where a single controller is in charge of all network decisions. This main controller is supported by backup controller replicas that should have a synchronized network view in order to take over the network control in case of the primary failure. All controller replicas are coordinated through a shared data store that is kept fault-tolerant and strongly consistent using an implementation of Replicated State Machine (RSM).

Consistency between the master and backup controllers is guaranteed by replicating each change in the network image (NIB) of the master into the shared data store before modifying the state of the network. However, such synchronization updates generate additional time overheads and have a drastic impact on the controller's performance. To ad-

dress this issue, the controllers keep a local cache (maintained by one active primary controller at any time) to avoid accessing the shared data store for read operations. By keeping the local cache and the data store consistent even in the presence of controller failures, the authors claim that their simple Master-Slave structure achieves, in the context of small to medium-sized networks, a balance between consistency and fault-tolerance while keeping performance at an acceptable level.

1.4.1.2 HyperFlow and Ravana

Both HyperFlow [39] and Ravana [44] are logically centralized controller platforms that achieve state redundancy through event replication. Despite their similarities in building the application state, one difference is that the Ravana protocol is completely transparent to control applications while HyperFlow requires minor modifications to applications. Besides, while HyperFlow is eventually consistent favoring availability, Ravana ensures strong consistency guarantees.

More specifically, Hyperflow [39] is an extension of NOX into a distributed event-based control plane where each NOX-based controller manages a subset of OpenFlow network switches. It uses an event-propagation publish/subscribe mechanism based on the distributed WheelFS [40] file system for propagating selected network events and maintaining the global network-wide view across controllers. Accordingly, the Hyperflow controller application instance running on top of an individual NOX controller selectively publishes relevant events that affect the network state and receives events on subscribed channels to other controllers. Then, other controllers locally replay all the published events in order to reconstruct the state and achieve the synchronization of the global view.

By this means, all controller instances make decisions locally and individually (without contacting remote controller instances): They indeed operate based on their synchronized eventually-consistent network-wide view as if they are in control of the entire network. Through this synchronization scheme, Hyperflow achieves the goal of minimizing flow setup times and also congestion, in other words, cross-site traffic required to synchronize the state among controllers. However, the potential downside of Hyperflow is related to the performance of the publish/subscribe system which can only deal with non-frequent events. Besides, HyperFlow does not guarantee a strict ordering of events and does not handle consistency problems. This makes the scope of HyperFlow restricted to applications that does not require a strict event ordering with strict consistency guar-

antees.

To correctly ensure the abstraction of a "logically centralized SDN controller", an elaborate fault-tolerant controller platform called Ravana [44] extended beyond the requirements for controller state consistency to include that for switch state consistency under controller failures.

Maintaining such strong levels of consistency in both controllers and switches in the presence of failures, requires handling the entire event-processing cycle as a transaction in accordance with the following properties: (i) events are processed in the same total order at all controller replicas so that controller application instances would reach the same internal state, (ii) events are processed exactly-once across all the controller replicas, (iii) commands are executed exactly-once on the switches.

To achieve such design goals, Ravana follows a Replicated State Machine (RSM) approach, but extends its scope to deal with switch state consistency under failures. Indeed, while Ravana permits unmodified applications to run in a transparent fault-tolerant environment, it requires modifications to the OpenFlow protocol, and it makes changes to current switches instead of involving them in a complex consensus protocol.

To be more specific, Ravana uses a two-stage replication protocol that separates the reliable logging of the master's event delivery information (stage 1) from the logging of the master's event-processing transaction completion information (stage 2) in the shared in-memory log (using Viewstamped Replication [61]) in order to guarantee consistency under joint switch and controller failures. Besides, it adds explicit acknowledgement messages to the OpenFlow 1.3 protocol and implements buffers on existing switches for event retransmission and command filtering. The main objective of the addition of these extensions and mechanisms is to guarantee the exactly-once execution of any event transaction on the switches during controller failures.

Such strong correctness guarantees for a logically centralized controller under Ravana come at the cost of generating additional throughput and latency overheads that can be reduced to a quite reasonable extent with specific performance optimizations. Since the Ravana runtime is completely transparent and oblivious to control applications, achieving relaxed consistency requirements for the sake of improved availability as required by some specific applications, entails considering new mechanisms that consider relaxing some of the correctness constraints on Ravana's design goals.

A similar approach to Ravana [44] was adopted by Mantas et. al [62] to achieve a con-

sistent and fault-tolerant SDN controller platform. In their ongoing work, the authors claim to retain the same requirements expressed by Ravana, namely the transparency, reliability, consistency and performance guarantees, but without requiring changes to the OpenFlow protocol or to existing switches.

Likewise, Kandoo [53] falls in this category of logically centralized controllers that distribute the control state by propagating network events. Indeed, Kandoo assumes, at the top layer of its hierarchical design, a logically centralized root controller for handling global and rare network events. Since the main aim was to preserve scalability without changing the OpenFlow devices, Kandoo did not focus on knowledge distribution mechanisms for achieving network state consistency.

1.4.1.3 ONOS and OpenDayLight

ONOS and OpenDayLight [63] represent another category of logically centralized SDN solutions that set themselves apart from state-of-the-art distributed SDN controller platforms by offering community-driven open-source frameworks as well as providing the full functionalities of Network Operating Systems. Despite their obvious similarities, these prominent Java-based projects present major differences in terms of structure, target customers, focus areas and inspirations.

Dissimilar to OpenDayLight [64] which is applicable to different domains, ONOS [36] from ON.LAB is specifically targeted towards service providers and is thus architected to meet their carrier-grade requirements in terms of scalability, high-availability and performance. In addition to the high-level Northbound abstraction (a global network view and an application intent framework) and the pluggable Southbound abstraction (supporting multiple protocols), ONOS, in the same way as Onix and Hyperflow, offers state dissemination mechanisms [65] to achieve a consistent network state across the distributed cluster of ONOS controllers, a required or highly desirable condition for network applications to run correctly.

More specifically, ONOS's distributed core eases the state management and cluster coordination tasks for application developers by providing them with an available set of core building blocks for dealing with different types of distributed control plane state, including a *ConsistentMap* primitive for state requiring strong consistency and an *EventuallyConsistentMap* for state tolerating weak consistency.

In particular, applications that favor performance over consistency store their state in

the shared eventually-consistent data structure that uses optimistic replication assisted by the gossip-based Anti-Entropy protocol [41]. For example, the global network topology state which should be accessible to applications with minimal delays is managed by the *Network Topology store* according to this eventual consistency model. Recent releases of ONOS treat the network topology view as an in-memory state machine graph. The latter is built and updated in each SDN controller by applying local topology events and replicating them to other controller instances in the cluster in an order-aware fashion based on the events' logical timestamps. Potential conflicts and loss of updates due to failure scenarios are resolved by the anti-entropy approach [41] where each controller periodically compares its topology view with that of another randomly-selected controller in order to reconcile possible differences and recover from stale information.

On the other hand, state imposing strong consistency guarantees is managed by the second data structure primitive built using RAFT [66], a protocol that achieves consensus via an elected leader controller in charge of replicating the received log updates to follower controllers and then committing these updates upon receipt of confirmation from the majority. The mapping between controllers and switches which is handled by ONOS's *Mastership store* is an example of a network state that is maintained in a strongly consistent manner.

Administered by the Linux Foundation and backed by the industry, OpenDayLight (ODL) [64] is a generic and general-purpose controller framework which, unlike ONOS, was conceived to accommodate a wide variety of applications and use cases concerning different domains (e.g. Data Center, Service Provider and Enterprise). One important architectural feature of ODL is its YANG-based Model-Driven Service Abstraction Layer (MD-SAL) that allows for the easy and flexible incorporation of network services requested by the higher layers via the Northbound Interface (OSGi framework and the bidirectional RESTful Interfaces) irrespective of the multiple Southbound protocols used between the controller and the heterogeneous network devices.

The main focus of ODL was to accelerate the integration of SDN in legacy network environments by automating the configuration of traditional network devices and enabling their communication with OpenFlow devices. As a result, the project was perceived as adopting vendor-driven solutions that mainly aim at preserving the brands of legacy hardware. This represents a broad divergence from ONOS which envisions a carrier-grade SDN platform with enhanced performance capabilities to explore the full potential

of SDN and demonstrate its real value.

The latest releases of ODL provided a distributed SDN controller architecture referred to as ODL clustering. Differently from ONOS, ODL did not offer various consistency models for different types of network data. All the data shared across the distributed cluster of ODL controllers for maintaining the logically centralized network view is handled in a strongly-consistent manner using the RAFT consensus algorithm [66] and the Akka framework [67].

1.4.1.4 B4 and SWAN

Google's B4 [56] network leverages the logical centralization enabled by the SDN paradigm to deploy centralized TE in coexistence with the standard shortest-path routing for the purpose of increasing the utilization of the inter-data-center links (near 100%) as compared to conventional networks and thereby enhancing network efficiency and performance. As previously explained in Section 1.3.2.2, the logically centralized *TE server* uses the network information collected by the centralized *SDN Gateway* to control and coordinate the behavior of site-level SDN controllers based on an abstracted topology view. The main task of the TE server is indeed to optimize the allocation of bandwidth among competing applications (based on their priority) across the geographically-distributed data-center sites.

That being said, we implicitly assume the presence of a specific consistency model used by the centralized SDN Gateway for handling the distributed network state across the data-center site controllers and ensuring that the centralized TE application runs correctly based on a consistent network-wide view. However, there has been very little information provided on the level of consistency adopted by the B4 system. As a matter of fact, one potential downside of the SDN approach followed by Google could be the fact that it is too customized and tailored to their specific network requirements as no general control model has been proposed for future use by other SDN projects.

Similarly, Microsoft has presented SWAN [68] as an intra-domain software-driven WAN deployment that takes advantage of the logically-centralized SDN control using a global TE solution to significantly improve the efficiency, reliability and fairness of their inter-DC WAN. In the same way as Google, Microsoft did not provide much information about the control plane state consistency updates.

1.4.2 Logically-distributed SDN control

The potential of the SDN paradigm has been properly explored within single administrative domains like data centers, enterprise networks, campus networks and even WANs as discussed in Section 1.4.1. Indeed, the main pillars of SDN – the decoupling between the control and data planes together with the consequent ability to program the network in a logically centralized manner – have unleashed productive innovation and novel capabilities in the management of such intra-domain networks. These benefits include the effective deployment of new domain-specific services as well as the improvement of standard control functions following the SDN principles like intra-domain routing and TE. RCP [69] and RouteFlow [70] are practical examples of successful intra-AS platforms that use OpenFlow to provide conventional IP routing services in a centralized manner.

However, that main feature of logically-centralized control which has been leveraged by most SDN solutions to improve network management at the intra-domain level, cannot be fully exploited for controlling heterogeneous networks involving multiple Autonomous Systems (ASes) under different administrative authorities (e.g. the Internet). In this context, recent works have considered extending the SDN scheme to such inter-domain networks while remaining compatible with their distributed architecture. In this section, we shed light on these SDN solutions which adopted a logically distributed architecture in accordance with legacy networks. For that reason, we place them in the category of logically distributed SDN platforms as opposed to the logically centralized ones mainly used for intra-domain scenarios.

1.4.2.1 DISCO and D-SDN

For instance, the DISCO project [37] suggests a logically distributed control plane architecture that operates in such multi-domain heterogeneous environments, more precisely WANs and overlay networks. Built on top of Floodlight [31], each DISCO controller administers its own SDN network domain and interacts with other controllers to provide end-to-end network services. This inter-AS communication is ensured by a unique lightweight control channel to share summary network-wide information.

The most obvious contribution of DISCO lies in the separation between intra-domain and inter-domain features of the control plane, while each type of features is performed by a separate part of the DISCO architecture. The intra-domain modules are responsible for ensuring the main functions of the controller such as monitoring the network and re-

acting to network issues, and the inter-domain modules (Messenger, Agents) are designed to enable a message-oriented communication between neighbor domain controllers. Indeed, the AMQP-based Messenger [71] offers a distributed publish/subscribe communication channel used by agents which operate at the inter-domain level by exchanging aggregated information with intra-domain modules. DISCO was assessed on an emulated environment according to three use cases: inter-domain topology disruption, end-to end service priority request and Virtual Machine Migration.

The main advantage of the DISCO solution is the possibility to adapt it to large-scale networks with different ASes such as the Internet [60]. However, we believe that there are also several drawbacks associated with such a solution including the static non-evolving decomposition of the network into several independent entities, which is in contrast to emerging theories such as David D. Clark's theory [72] about the network being manageable by an additional high-level entity known as the Knowledge Plane. Besides, following the DISCO architecture, network performance optimization becomes a local task dedicated to local entities with different policies, each of which acts in its own best interest at the expense of the general interest. This leads to local optima rather than the global optimum that achieves the global network performance. Additionally, from the DISCO perspective, one SDN controller is responsible for one independent domain. However, an AS is usually too large to be handled by a single controller. Finally, DISCO did not provide appropriate reliability strategies suited to its geographically-distributed architecture. In fact, in the event of a controller failure, one might infer that a remote controller instance will be in charge of the subset of affected switches, thereby resulting in a significant increase in the control plane latency. In our opinion, a better reliability strategy would involve local per-domain redundancy; Local controller replicas should indeed take over and serve as backups in case the local primary controller fails.

In the same spirit, INRIA's D-SDN [73] enables a logical distribution of the SDN control plane based on a hierarchy of *Main Controllers* and *Secondary Controllers*, matching the organizational and administrative structure of current and future Internet. In addition to dealing with levels of control hierarchy, another advantage of D-SDN over DISCO is related to its enhanced security and fault tolerance features.

1.4.2.2 SDX-based controllers

Different from DISCO which proposes per-domain SDN controllers with inter-domain functions for allowing autonomous end-to-end flow management across SDN domains, recent trends have considered deploying SDN at Internet eXchange Points (IXPs) thus, giving rise to the concept of Software-Defined eXchanges (SDXes). These SDXes are used to interconnect participants of different domains via a shared software-based platform. That platform is usually aimed at bringing innovation to traditional peering, easing the implementation of customized peering policies and enhancing the control over inter-domain traffic management.

Prominent projects adopting that vision of software-defined IXPs and implementing it in real production networks include Google's Cardigan in New Zealand [74], SDX at Princeton [75], CNRS's French TouIX [76] (European ENDEAVOUR [77]) and the AtlanticWave-SDX [78]. Here we chose to focus on the SDX project at Princeton since we believe in its potential for demonstrating the capabilities of SDN to innovate IXPs and for bringing answers to deploying SDX in practice.

The SDX project [75] takes advantage of SDN-enabled IXPs to fundamentally improve wide-area traffic delivery and enhance conventional inter-domain routing protocols that lack the required flexibility for achieving various TE tasks. Today's BGP is indeed limited to destination-based routing, it has a local forwarding influence restricted to immediate neighbors, and it deploys indirect mechanisms for controlling path selection. To overcome these limitations, SDX relies on SDN features to ensure fine-grained, flexible and direct expression of inter-domain control policies, thereby enabling a wider range of valuable end-to-end services such as Inbound TE, application-specific peering, server load balancing, and traffic redirection through middle-boxes.

The SDX architecture consists of a smart SDX controller handling both SDX policies (*Policy compiler*) and BGP routes (*Route Server*), conventional Edge routers, and an OpenFlow-enabled switching fabric. The main idea behind this implementation is to allow participant ASes to compose their own policies in a high-level (using Pyretic) and independent manner (through the virtual switch abstraction), and then send them to the SDX controller. The latter is in charge of compiling these policies to SDN forwarding rules while taking into account BGP information.

Besides offering this high-level softwarized framework that is easily integrated into the existing infrastructure while maintaining good interoperability with its routing protocol,

SDX also stands out from similar solutions like Cardigan [74] by the efficient mechanisms used for optimizing control and data plane operations. In particular, the scalability challenges faced by SDX under realistic scenarios have been further investigated by iSDX [79], an enhanced Ryu [50]-based version of SDX intended to operate at the scale of large industrial IXPs.

However, one major drawback of the SDX contribution is that it is limited to the participant ASes being connected via the software-based IXP, implying that non-peering ASes would not benefit from the routing opportunities offered by SDX. Besides, while solutions built on SDX use central TE policies for augmenting BGP and promote a logical centralization of the routing control plane at the IXP level, SDX controllers are still logically decentralized at the inter-domain level since no information is shared between them about their respective interconnected ASes. This brings us back to the same problem we pointed out for DISCO [37] about end-to-end traffic optimization being a local task for each part of the network.

To remedy this issue, some recent works [80] have considered centralizing the whole inter-domain routing control plane to improve BGP convergence by outsourcing the control logic to a multi-AS routing controller that has a "Bird's-eye view" over multiple ASes.

It is also worth mentioning that SDX-based controllers face several limitations in terms of both security and reliability.

Because the SDX controller is the central element in the SDX architecture, security strategies must focus on securing the SDX infrastructure by protecting the SDX controller against cyber attacks and by authenticating any access to it. In particular, Chung *et al.* [81] argue that SDX-based controllers are subjected to the potential vulnerabilities introduced by SDN in addition to the common vulnerabilities associated with classical protocols. In that respect, they distinguish three types of current SDX architectures and discuss the involved security concerns. In their opinion, Layer-3 SDX [74; 75] will inherit all BGP vulnerabilities, Layer-2 SDX [82] will get the vulnerabilities of a shared Ethernet network, and SDN SDX [17] will also bring controller vulnerabilities like DDoS attacks, comprised controllers and malicious controller applications. Moreover, the same authors of [81] point out that SDX-based controllers require security considerations with respect to Policy isolation between different SDX participants.

Finally, since the SDX controller becomes a potential single point of failure, fault-tolerance and resiliency measures should be taken into account when building an SDX ar-

chitecture. While the distributed peer-to-peer SDN SDX architecture [83] is inherently resilient, centralized SDX approaches should incorporate fault-tolerance mechanisms like that discussed in Section 2.3 and should also leverage the existing fault-tolerant distributed SDN controller platforms [36].

1.5 Conclusion

In this chapter, we provide a detailed analysis of state-of-the-art distributed SDN controller platforms: Thereby, we assess their architecture components and design patterns, and we classify them in novel ways (physical and logical classifications) in order to provide useful guidelines for SDN research and deployment initiatives.

Additionally, our thorough analysis of these SDN platform proposals allowed us to achieve an extensive understanding of their advantages and drawbacks and, most importantly, to develop a critical awareness of the challenges facing the distributed control in SDNs. These open challenges are further discussed in the next chapter (Chapter 2).

Chapter 2

Decentralized SDN control: Major open challenges

« The road to SDN »

Nick Feamster

Contents

2.1 Introduction

While offering a promising potential to transform and improve current networks, the SDN initiative is still in the early stages of addressing the wide variety of challenges involving different disciplines. In particular, the distributed control of SDNs faces a series of pressing challenges that require our special consideration.

This chapter provides a thorough discussion of the major challenges of distributed SDN control along with some insights into emerging and future trends in that area. These challenges include the issues of Scalability (Section 2.2), Reliability (Section 2.3), State Consistency (Section 2.4), Interoperability (Section 2.5), Monitoring and Security (Section 2.6).

In the previous chapters, we surveyed the most prominent state-of-the art distributed SDN controller platforms and more importantly we discussed the different approaches adopted in tackling the above challenges and proposing potential solutions. Table 2.1 gives a brief summary of the main features and KPIs of the discussed SDN controllers. Physically-centralized controllers such as NOX [29], POX [34] and FloodLight [31] suffer from scalability and reliability issues. Solutions like DevoFlow [54] and DIFANE [55] attempted to solve these scalability issues by rethinking the OpenFlow protocol whereas most SDN groups geared their focus towards distributing the control plane. While some of the distributed SDN proposals such as Kandoo [53] promoted a hierarchical organization of the control plane to further improve scalability, other alternatives opted for a flat organization for increased reliability and performance (latency). On the other hand, distributed platforms like Onix [38], HyperFlow [38], ONOS [36] and OpenDaylight [64], focused on building consistency models for their logically centralized control plane designs. In particular, Onix [38] chose DHT and transactional databases for network state distribution over the Publish/Subscribe system used by HyperFlow [39]. Another different class of solutions has been recently introduced by DISCO which promoted a logically distributed control plane based on existing ASs within the Internet.

In previous chapters, we classified these existing controllers according to the physical organization of the control plane (*Physical classification*) and, alternatively, according to the way knowledge is shared in distributed control plane designs (*Logical classification*). Furthermore, within each of these classifications, we performed another internal classification based on the similarities between competing SDN controllers (*The color classifica-*

	Control Plane Architecture	Control Plane Design	Programming language	Scalability	Reliability	Consistency
NOX [29]	Physically Centralized	–	C++	Very Limited	Limited	Strong
POX [34]	Physically Centralized	–	Python	Very Limited	Limited	Strong
Floodlight [31]	Physically Centralized	–	Java	Very Limited	Limited	Strong
SMaRtLight [48]	Physically Centralized	–	Java	Very Limited	Very Good	Strong
Ravana [44]	Physically Centralized	–	Python	Limited	Very Good	Strong
ONIX [38]	Physically Distributed Logically Centralized	Flat	Python C	Very Good	Good	Weak Strong
HyperFlow [39]	Physically Distributed Logically Centralized	Flat	C++	Good	Good	Eventual
ONOS [36]	Physically Distributed Logically Centralized	Flat	Java	Very Good	Good	Weak Strong
OpenDayLight [64]	Physically Distributed Logically Centralized	Flat	Java	Very Good	Good	Strong
B4 [56]	Physically Distributed Logically Centralized	Hierarchical	Python C	Good	Good	N/A
Kandoo [53]	Physically Distributed Logically Centralized	Hierarchical	C C++ Python	Very Good	Limited	N/A
DISCO [37]	Physically Distributed Logically Distributed	Flat	Java	Good	Limited	Strong (inter-domain)
SDX [75]	Physically Distributed Logically Distributed	Flat	Python	Limited	N/A	Strong
DevoFlow [54]	Physically Distributed Logically Centralized	N/A	Java	Good	N/A	N/A
DIFANE [55]	Physically Distributed Logically Centralized	N/A	–	Good	N/A	N/A

Table 2.1: Main characteristics of the discussed SDN controllers

tion shown in Figure 1.3 and Figure 1.4).

In light of the above, it is obvious that there are various approaches to building a distributed SDN architecture; Some of these approaches met some performance criteria better than others but failed in some other aspects. Clearly, none of the proposed SDN controller platforms met all the discussed challenges and fulfilled all the KPIs required for an optimal deployment of SDN. At this stage, and building on these previous efforts, we communicate our vision of a distributed SDN control model by going through some of the major open challenges (see Figure 2.1), identifying the best ways of solving them, and envisioning future opportunities.

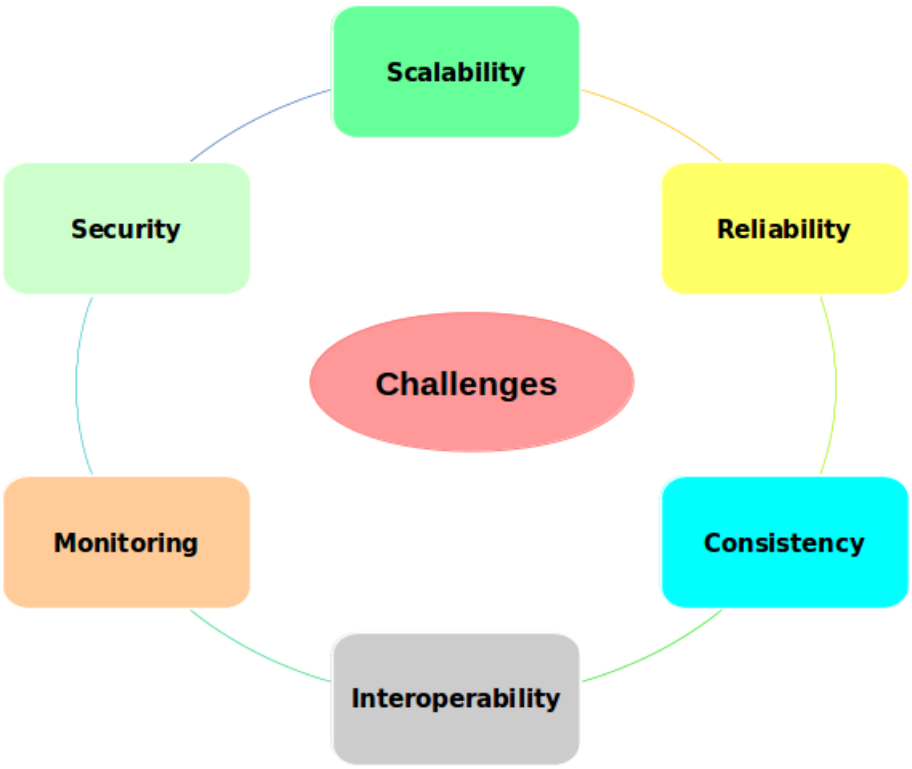


Figure 2.1: The main challenges of distributed SDN control

2.2 Scalability

Scalability concerns in SDN may stem from the decoupling between the control and data planes [84] and the centralization of the control logic in a software-based controller. In fact, as the network grows in size (e.g. switches, hosts, etc.), the centralized SDN controller becomes highly solicited (in terms of events/requests) and thus overloaded (in terms of bandwidth, processing power and memory). Furthermore, when the network scales up in terms of both size and diameter, communication delays between the SDN controller and the network switches may become high, thus affecting flow-setup latencies. This may also cause congestion in both the control and data planes and may generate longer failover times [28].

That said, since control plane scalability in SDN is commonly assessed in terms of both *throughput* (the number of flow requests handled per second) and *flow setup latency* (the delay to respond flow requests) metrics [28], a single physically-centralized SDN controller may not particularly fulfill the performance requirements (with respect to these metrics) of large-scale networks as compared to small or medium scale networks (see Section 1.3.1).

2.2.1 Data plane extensions

One way to alleviate some of these scalability concerns is to extend the responsibilities of the data plane in order to relieve the load on the SDN controller [85]. The main drawback of that method is that it imposes some modifications to the design of OpenFlow switches. Indeed, despite the advantages linked to the flexibility and innovation brought to network management, OpenFlow [14] suffers from scalability and performance issues that stem mainly from pushing all network intelligence and control logic to the centralized OpenFlow controller, thus restricting the task of OpenFlow switches to a dumb execution of forwarding actions.

To circumvent these limitations, several approaches [54; 55; 86; 87] suggest revisiting the delegation of control between the controller and switches and introducing new SDN switch Southbound interfaces.

Notably, DevoFlow [54] claims to minimize switch-to-controller interactions by introducing new control mechanisms inside switches. That way, switches can make local control decisions when handling frequent events, without involving controllers whose primary tasks will be limited to keeping centralized control over far fewer significant events that require network-wide visibility. Despite introducing innovative ideas, the DevoFlow alternative has been mainly criticized for imposing major modifications to switch designs [53].

On the other hand, *stateful* approaches [86; 88–90], as opposed to the original *stateless* OpenFlow abstraction, motivate the need to delegate some stateful control functions back to switches in order to offload the SDN controller. These approaches face the challenging dilemma of programming stateful devices (evolving the data plane) while retaining the simplicity, generality and vendor-agnostic features offered by the OpenFlow abstraction. In particular, the OpenState proposal [86] is a stateful platform-independent data plane extension of the current OpenFlow match/action abstraction supporting a finite-state machine (FSM) programming model called Mealy Machine in addition to the flow programming model adopted by OpenFlow. That model is implemented inside the OpenFlow switches using additional *state tables* in order to reduce the reliance on remote controllers for applications involving local states like MAC learning operations and port-knocking on a firewall.

Despite having the advantage of building on the adaptation activity of the OpenFlow standard and leveraging its evolution using the (stateful) extensions provided by recent

versions (version 1.3 and 1.4), OpenState faces important challenges regarding the implementation of a stateful extension for programming the forwarding behaviour inside switches while following an OpenFlow-like implementation approach. The feasibility of the hardware implementation of OpenState has been addressed in [91]. Finally, the same authors extended their work into a more general and expressive abstraction of OpenState called OPP [92] which supports a full extended finite-state machine (XFSM) model, thereby enabling a broader range of applications and complex stateful flow processing operations.

In the same spirit, the approach presented in [93] explored delegating some parts of the controller functions involving packet generation tasks to OpenFlow switches in order to address both switch and controller scalability issues. The InSP API was introduced as a generic API that extends OpenFlow to allow for the programming of autonomous packet generation operations inside the switches such as ARP and ICMP handling. The proposed OpenFlow-like abstractions include an *InSP Instruction* for specifying the actions that the switch should apply to a packet being generated after a triggering event and a *Packet Template Table (PTE)* for storing the content of any packet generated by the switch.

According to [93], the InSP function, like any particular offloading function, faces the challenging issue of finding the relevant positioning with respect to the broad design space for delegation of control to SDN switches. In their opinion, a good approach to conceiving (eventually standardizing) a particular offloading function should involve a programming abstraction that achieves a fair compromise between viability and flexibility, far from extreme solutions that simply turn on well-known legacy protocol functions (e.g. MAC learning) or push a piece of code inside the switches [94; 95].

The authors of FOCUS [96] express the same challenges but, unlike the above proposals, they reject a performance-based design choice that requires adding new hardware primitives to OpenFlow switches in the development of the delegated control function. Instead, they promote a deployable software-based solution to be implemented in the switch's *software stack* to achieve a balanced trade-off between the flexibility and cost of the control function delegation process.

2.2.2 Control plane distribution

The second alternative, which we believe to be more effective, is to model the control plane in a way that mitigates scalability limitations [85]. In a physically-centralized con-

trol model, a single SDN controller is in charge of handling all requests coming from SDN switches. As the network grows, the latter is likely to become a serious bottleneck in terms of scalability and performance [97]. On the other hand, a physically-distributed control model uses multiple controllers that maintain a logically centralized network view. This solution is appreciated for handling the controller bottleneck, hence ensuring a better scale of the network control plane while decreasing control-plane latencies.

Even though the distributed control model is considered as a scalable option when compared to the centralized control model, achieving network scalability while preserving good performance requires a relevant control distribution scheme that takes into account both the organization of the SDN control plane and the physical placement of the SDN controllers. In this context, we recommend a hierarchical organization of the control plane over a flat organization for increased scalability and improved performance. We also believe that the placement of controllers should be further investigated and treated as an optimization problem that depends on specific performance metrics [25].

Finally, by physically distributing the SDN control plane for scalability (and reliability 2.3) purposes, it is worth mentioning that new kinds of challenges may arise. In particular, to maintain the logically centralized view, a strongly-consistent model can be used to meet certain application requirements. However, as discussed in Section 2.4, a strongly consistent model may introduce new scalability issues. In fact, retaining strong consistency when propagating frequent state updates might block the state progress and cause the network to become unavailable, thus increasing switch-to-controller latencies.

2.3 Reliability

Concerns about reliability have been considered as serious in SDN [98]. The data-to-control plane decoupling has indeed a significant impact on the reliability of the SDN control plane. In a centralized SDN-based network, the failure of the central controller may collapse the overall network. In contrast, the use of multiple controllers in a physically distributed (but logically centralized) controller architecture alleviates the issue of a single point of failure.

Despite not providing information on how a distributed SDN controller architecture should be implemented, the OpenFlow standard gives (since version 1.2) the ability for a switch to simultaneously connect to multiple controllers. That OpenFlow option allows each controller to operate in one of three roles (*master*, *slave*, *equal*) with respect to an

active connection to the switch. Leveraging on these OpenFlow roles which refer to the importance of controller replication in achieving a highly available SDN control plane, various resiliency strategies have been adopted by different fault-tolerant controller architectures. Among the main challenges faced by these architectures are control state redundancy and controller failover.

2.3.1 Control state redundancy

Controller redundancy can be achieved by adopting different approaches for processing network updates. In the Active replication approach [47], also known as State Machine Replication, multiple controllers process the commands issued by the connected clients in a coordinated and deterministic way in order to concurrently update the replicated network state. The main challenge of that method lies in enforcing a strict ordering of events to guarantee strong consistency among controller replicas. That approach for replication has the advantage of offering high resilience with an insignificant downtime, making it a suitable option for delay-intolerant scenarios. On the other hand, in passive replication, referred to as primary/backup replication, one controller (the *primary*) processes the requests, updates the replicated state, and periodically informs the other controller replicas (the *backups*) about state changes. Despite offering simplicity and lower overhead, the passive replication scheme may create (controller and switch) state inconsistencies and generate additional delay in case the primary controller fails.

Additional concerns that should be explored are related to the kind of information to be replicated across controllers. Existing controller platform solutions follow three approaches for achieving controller state redundancy [99]: state replication [38; 48], event replication [39; 44] and traffic replication [100].

Moreover, control distribution is a central challenge when designing a fault tolerant controller platform. The centralized control approach that follows the simple Master/Slave concept [44; 48] relies on a single controller (the *master*) that keeps the entire network state and takes all decisions based on a global network view. Backup controllers (the *slaves*) are used for fault-tolerance purposes. The centralized alternative is usually considered in small to medium-sized networks. On the other hand, in the distributed control approach [36; 38], the network state is partitioned across many controllers that simultaneously take control of the network while exchanging information to maintain the logically centralized network view. In that model, controller coordination strategies should be

applied to reach agreements and solve the issues of concurrent updates and state consistency. Mostly effective in large-scale networks, the distributed alternative provides fault tolerance by redistributing the network load among the remaining active controllers.

Finally, the implementation aspect is another important challenge in designing a replication strategy [47]. While some approaches opted for replicating controllers that store their network state locally and communicate through a specific group coordination framework [101], other approaches went for replicating the network state by delegating state storage, replication and management to external data stores [36; 38; 39] like distributed data structures and distributed file systems.

2.3.2 Controller failover

Apart from controller redundancy, other works focused on failure detection and controller recovery mechanisms. Some of these works considered reliability criteria from the outset in the placement of distributed SDN controllers. Both the number and locations of controllers were determined in a reliability-aware manner while preserving good performance. Reliability was indeed introduced in the form of controller placement metrics (switch-to-controller delay, controller load) to prevent worst-case switch-to-controller reassignment scenarios in the event of failures. Other works elaborated on efficient controller failover strategies that consider the same reliability criteria. Strategies for recovering from controller failures can be split into redundant controller strategies (with backups) and non-redundant controller strategies (without backups) [102].

The redundant controller strategy assumes more than one controller per controller domain; One primary controller actively controls the network domain and the remaining controllers (backups) automatically take over the domain in case it fails. Despite providing a fast failover technique, this strategy depends on the associated standby methods (*cold*, *warm* or *hot*) which have different advantages and drawbacks [103]. For instance, the cold standby method imposes a full initialization process on the standby controller given the complete loss of the state upon the primary controller failure. This makes it an adequate alternative for stateless applications. In contrast, the hot standby method is effective in ensuring a minimum recovery time with no controller state loss, but it imposes a high communication overhead due to the full state synchronization requirements between primary and standby controllers. The warm standby method reduces that communication overhead at the cost of a partial state loss.

On the other hand, the non-redundant controller strategy requires only one controller per controller domain. In case it fails, controllers from other domains extend their domains to adopt orphan switches, thereby reducing the network overhead. Two well-known strategies for non-redundant controllers are the greedy failover and the pre-partitioning failover [104]. While the former strategy relies on neighbor controllers to adopt orphan switches at the edge of their domains and from which they can receive messages, the latter relies on controllers to proactively exchange information about the list of switches to take over in controller failure scenarios.

All things considered, a number of challenges and key design choices based on a set of requirements are involved when adopting a specific controller replication and failover strategy. In addition to reliability and fault-tolerance considerations, scalability, consistency and performance requirements should be properly taken into account when designing a fault-tolerant SDN controller architecture.

2.4 Controller state consistency

Contrary to physically centralized SDN designs, distributed SDN controller platforms face major consistency challenges [105–107]. Clearly, physically distributed SDN controllers must exchange network information and handle the consistency of the network state being distributed across them and stored in their shared data structures in order to maintain a logically centralized network-wide view that eases the development of control applications. However, achieving a convenient level of consistency while keeping good performance in software-defined networks facing network partitions is a complex task. As claimed by the CAP theorem applied to networks [108], it is generally impossible for SDN networks to simultaneously achieve all three of Consistency (C), high Availability (A) and Partition tolerance (P). In the presence of network partitions, a weak level of consistency in exchange for high availability (AP) results in state staleness causing an incorrect behavior of applications whereas a strong level of consistency serving the correct enforcement of network policies (CP) comes at the cost of network availability.

2.4.1 Static consistency

The *Strong Consistency* model used in distributed file systems implies that only one consistent state is observed by ensuring that any read operation on a data item returns the value of the latest write operation that occurred on that data item. However, such con-

sistency guarantees are achieved at the penalty of increased data store access latencies. In SDNs, the strong consistency model guarantees that all controller replicas in the cluster have the most updated network information, albeit at the cost of increased synchronization and communication overhead. In fact, if certain data occurring in different controllers are not updated to all of them, then these data are not allowed to be read, thereby impacting network availability and scalability.
Strong consistency is crucial for implementing a wide range of SDN applications that require the latest network information and that are intolerant of network state inconsistencies. Among the distributed data store designs that provide strong consistency properties are the traditional SQL-based relational databases like Oracle [109] and MySQL [110].

On the other hand, as opposed to the strong consistency model, the *Eventual Consistency* model (sometimes referred to as a *Weak Consistency* model) implies that concurrent reads of a data item may return values that are different from the actual updated value for a transient time period. This model takes a more relaxed approach to consistency by assuming that the system will eventually (after some period) become consistent in order to gain in network availability. Accordingly, in a distributed SDN scenario, reads of some data occurring in different SDN controller replicas may return different values for some time before eventually converging to the same global state. As a result, SDN controllers may temporarily have an inconsistent network view and thus cause an incorrect application behavior.
Eventually-consistent models have also been extensively used by SDN designers for developing inconsistency-tolerant applications that require high scalability and availability. These control models provide simplicity and efficiency of implementation but they push the complexity of resolving state inconsistencies and conflicts to the application logic and the consensus algorithms being put in place by the controller platform. Cassandra [111], Riak [112] and Dynamo [113] are popular examples of NoSQL databases that have adopted the eventual consistency model.

All things considered, maintaining state consistency across logically centralized SDN controllers is a significant SDN design challenge that involves trade-offs between policy enforcement and network performance [114]. The issue is that achieving strong consistency in an SDN environment that is prone to network failures is almost impossible without compromising availability and without adding complexity to network state management. Panda et. al [108] proposed new ways to circumvent these impossibility results but

their approaches can be regarded as specific to particular cases.

2.4.2 Adaptive multi-level consistency

In a more general context, SDN designers need to leverage the flexibility offered by SDN to select the appropriate consistency models for developing applications with various degrees of state consistency requirements and with different policies. In particular, adopting a single consistency model for handling different types of shared states may not be the best approach to coping with such a heterogeneous SDN environment. As a matter of fact, recent works on SDN have stressed the need for achieving consistency at different levels. So far, two levels of consistency models have been applied to SDNs and adopted by most distributed SDN controller platforms: strong consistency and eventual consistency.

In our opinion, a hybrid approach that merges various consistency levels should be considered to find the optimal trade-off between consistency and performance. Unlike the previously-mentioned approaches which are based on static consistency requirements where SDN designers decide which consistency level should be applied for each knowledge upon application development, we argue that an SDN application should be able to assign a priority for each knowledge and, depending on the network context (.i.e. instantaneous constraints, network load, etc), select the appropriate consistency level that should be enforced.

In that sense, recent approaches [107; 115] introduced the concept of adaptive consistency in the context of distributed SDN controllers, where adaptively-consistent controllers can tune their consistency level to reach the desired level of performance based on specific metrics. That alternative has the advantage of sparing application developers the tedious task of selecting the appropriate consistency level and implementing multiple application-specific consistency models. Furthermore, that approach can be efficient in handling the issues associated with eventual consistency models [116].

Finally, in the same way as scalability and reliability, we believe that consistency should be considered when investigating the optimal placement of controllers. In fact, minimizing inter-controller latencies (distances) which are critical for system performance facilitates controller communications and enhances network state consistency.

2.5 Interoperability

2.5.1 Interoperability between the SDN controllers

To foster the development and full adoption of SDN, we must overcome the common challenge of ensuring service interoperability between disparate distributed SDN controllers belonging to different SDN domains and using different controller technologies.

In today's multi-vendor environments, the limited interoperability between SDN controller platforms is mainly due to a lack of open standards for inter-controller communications. Apart from the standardization of the Southbound interface— OpenFlow being the most popular Southbound standard, there is to date no open standard for the Northbound and East-Westbound interfaces to provide compatibility between OpenFlow implementations.

Despite the emerging standardization efforts underway by SDN organizations, we argue that there are many barriers to effective and rapid standardization of the SDN East-Westbound interfaces, including the heterogeneity of the data models being used by SDN controller vendors. Accordingly, we emphasize the need for common data models to achieve interoperability and facilitate the tasks of standardization in SDNs. In this context, YANG [117] has emerged as a solid data modeling language used to model configuration and state data for standard representation. This NETCONF-based contribution from IETF is intended to be extended in the future and it is, more importantly, expected to pave the way for the emergence of standard data models driving interoperability in SDN networks.

Among the recent initiatives taken in that direction, we can mention OpenConfig's [118] effort on building a vendor-neutral data model written in YANG for configuration and management operations. Also worth mentioning is ONF's OF-Config protocol [119] which implements a YANG-based data model referred to as the Core Data Model. That protocol was introduced to enable remote configuration of OpenFlow-capable equipments.

2.5.2 SDN Interoperability with legacy networks

Alongside the concerns about the interoperability between the diverse SDN controller implementations, we highlight another important SDN challenge that is often overlooked, namely the challenge of reaching interoperability with legacy non-SDN technologies. While

the deployment of SDN is fairly straightforward for new networks incorporating new SDN-ready devices, the transition from a legacy networking environment to SDN requires a period of co-existence between SDN and legacy technologies.

In such heterogeneous network architectures operating a mix of SDN and traditional devices, it is extremely important to implement specific protocol mechanisms that support SDN control plane communications while providing efficient compatibility with existing IP control plane technologies. One potential solution is to adopt an incremental deployment strategy [120; 121] according to which a few SDN-enabled devices are deployed in a traditional network among the legacy devices, incrementally replacing them, and forming the so-called hybrid SDN network. In such a network, both SDN and legacy nodes should operate in parallel and may communicate together in order to ensure an effective gradual transition to SDN while reducing the associated operational costs and minimizing the disruption of network services.

2.6 Other challenges

An efficient network monitoring is required for the development of control and management applications in distributed SDN-based networks. However, collecting the appropriate data and statistics without impacting the network performance is a challenging task. In fact, the continuous monitoring of network data and statistics may generate excessive overheads and thus affect the network performance whereas the lack of monitoring may cause an incorrect behavior of management applications. Current network monitoring proposals have developed different techniques to find the appropriate trade-offs between data accuracy and monitoring overhead. In particular, IETF's NETCONF Southbound protocol provides some effective monitoring mechanisms for collecting statistics and configuring network devices. In the near future, we expect the OpenFlow specification to be extended to incorporate new monitoring tools and functions.

Like network monitoring, network security is another crucial challenge that should be studied. The decentralization of the SDN control reduces the risk associated with a single point of failure and attacks (e.g. the risk of a DDoS attack). However, the integrity of data flows between the SDN controllers and switches is still not safe. For instance, we can imagine that an attacker can corrupt the network by acting as an SDN controller. In this context, new solutions and strategies (e.g. based on TLS/SSL) have been introduced with the aim of guaranteeing security in SDN environments.

Another aspect related to SDN security is the isolation of flows and networks through network virtualization. In the case of an underlying physical SDN network, this could be implemented using an SDN network hypervisor that creates multiple logically-isolated virtual network slices (called vSDNs), each is managed by its own vSDN controller [122]. At this point, care should be taken to design and secure the SDN hypervisor as an essential part of the SDN network.

2.7 Conclusion

While the need for a distributed SDN architecture has been ultimately recognized by the SDN community [7; 123], the best approach to designing and implementing an efficient (e.g scalable and reliable) distributed SDN control plane is highly debatable given the many challenges brought by such distributed systems as discussed above.

The scalability, reliability, consistency, and interoperability of the SDN control plane are among the key challenges faced in designing an efficient and robust high-performance distributed SDN controller platform. Although regarded as the main limitations of fully centralized SDN control designs, scalability and reliability are also major concerns when designing a distributed SDN architecture. They are indeed highly impacted by the structure of the distributed SDN control plane (e.g. flat, hierarchical or hybrid organization) as well as the number and placement of the multiple controllers within the SDN network. Achieving such performance and availability requirements usually comes at the cost of guaranteeing a consistent centralized network view that is required for the design and correct behavior of SDN applications. Consistency considerations should therefore be explored among the trade-offs involved in the design process of an SDN controller platform. Last but not least, the interoperability between different SDN controller platforms of multiple vendors is another crucial operational challenge surrounding the development, maturity and commercial adoption of SDN. Overcoming that challenge calls for major standardization efforts at various levels of inter-controller communications (e.g Data models, Northbound and East-Westbound interfaces). Furthermore, such interoperability guarantees with respect to different SDN technology solutions represent an important step towards easing the widespread interoperability of these SDN platforms with legacy networks and, effectively ensuring the gradual transition towards softwarized network environments.

In the next chapters, we propose to tackle the distributed SDN control problem by

focusing on two major manageable challenges which, albeit correlated, could be treated as separate research problems: the controller placement problem (1) and the knowledge dissemination problem (2):

The first problem investigates the required number of SDN controllers along with their appropriate locations with respect to the desired objectives (see chapter 3). The second problem addresses the type and amount of network information to be shared across the SDN controller instances given a desired level of application state consistency and performance (see chapters 4 and 5).

Chapter 3

Scalability and reliability aware SDN controller placement strategies

« The network is in my way »

James Hamilton, Amazon

Contents

3.1 Introduction

In this chapter, we put forward novel strategies that tackle several aspects of the controller placement problem with respect to multiple reliability and performance criteria based on different uses and contexts [124].

Our contribution to solving the controller placement problem is indeed intended for expanding IoT-like networks that face important scalability challenges in addition to reliability issues. The proposed SDN controller placement scheme uses heuristics with low computation time in order to deal with such large-scale and dynamic network environments where fast reevaluations of controller placement configurations are required to adapt in real-time to frequently-changing network conditions. The potential of such heuristics in the context of the SDN controller placement is explored by comparing two different types of heuristic-based algorithms according to various context-based strategies.

The rest of this chapter is organized as follows: In Section 3.2, we give an overview of state-of-the-art contributions that addressed the controller placement problem. Then, in Section 3.3, we review the controller placement optimization problem and investigate the involved reliability and performance metrics. In Section 3.4, we put forward our versatile approach to tackling this problem. In Section 3.5, we display the obtained results. Finally, Section 3.6 critically analyzes and discusses these results before elaborating on the future perspectives.

3.2 Related work

There has been lately a growing interest in designing the distributed SDN control plane.

Heller et al. [125] have first motivated the SDN Controller Placement Problem (CPP) and discussed the challenges of control plane reliability, scalability and performance.

The authors provided useful guidance about *how many* and *where* SDN controllers should be placed in order to achieve high performance in an SDN network. They argued that the optimal number of required SDN controllers must be planned carefully for each network topology based on the concept of diminishing returns. They also claimed that in most topologies one single controller is enough for fulfilling latency requirements but obviously insufficient for achieving control plane resilience.

In their study, the location and placement of a determined number of SDN controllers

was treated as a variant of the *facility location problem*. Their placement strategy was only focused on minimizing the controller-to-switch propagation latency in the context of wide-area SDN deployments (e.g Internet2) and analyzing the trade-offs between optimizing the average latency (*the k-median problem*) and the maximum latency (*the k-center problem*).

Their work has been extended by [126] to incorporate other important performance aspects apart from the controller-to-switch latency in the multi-objective controller placement process such as the resilience metrics with respect to controller failure, network disruption, load imbalance and inter-controller latency. In this context, Hock et al. introduced the resilient Pareto-based Optimal COntroller placement (POCO) optimization framework for providing all possible Pareto-optimal CPP solutions and finding the adequate trade-offs between quality in terms of latency and resilience. Through the assessment of the framework using a range of different real network topologies, the authors argued that the required number of SDN controllers should be around 20 % of all network nodes in order to meet resilience requirements.

While the first version of POCO was intended for small and medium sized networks where an exhaustive exploration of the entire solution space for selecting the optimal controller placement with respect to the considered objectives is computationally feasible, a subsequent version proposed by [127] comprised a *heuristic-based* Multi-Objective Combinatorial Optimization (MOCO) approach, namely Pareto Simulated Annealing (PSA), for dealing with the resilient controller placement problem in large-scale or dynamic network environments. However, when evaluating that approach on a set of real-world network toplogies from the Internet Topology Zoo [128] where the network size ranges between 5 and 50 nodes, the authors only emphasize the geographic extent aspect of large-scale networks and do not assess their scalability in terms of an increased number of network nodes.

Unlike the above strategies which mitigate the impact of specific cases of network failures by minimizing resilience metrics like the worst-case controller-to-switch latency, Hu et al. [129] quantify reliability in terms of connectivity between the forwarding devices and their controllers (and between controllers as well) using a novel metric referred to as *the expected percentage of control path loss*. However, in doing so, the authors omitted important details about the failure probability of a network component which, in our opinion, should be estimated based on a specific network failure model. Finally, different

heuristic algorithms were presented and compared in their study for analyzing the trade-offs between latency and reliability in the reliability-aware controller placement decision.

Yao et al. [130] addressed another variant of the CPP that takes into account the load on controllers in addition to latency considerations in the placement strategy (*the capacitated k-center problem*). They used an effective algorithm for minimizing the maximum propagation delay under a controller capacity constraint. In their experiments, the load on controllers was measured based on the arriving rate of events and the controller capacity was determined according to their access bandwidth. Their approach proved efficient in minimizing the required number of controllers for avoiding controller overload.

The same authors of [127] explored in [131] the potential of *specialized heuristics* to solve the capacitated variant of the multi-objective controller placement problem in large-scale SDN networks by developing the Pareto-Capacitated k-Medoids (PCKM) method based on the k-Medoids clustering algorithm. Such a specialized heuristic that optimizes case-specific criteria, namely the average controller-to-switch latency and the controller load imbalance, was compared to generic heuristics usually destined for arbitrary multi-objective optimization purposes such as the MOCO PSA technique proposed in their previous work [127]. The performance comparison between these optimization heuristics applied to the CPP was also assessed on the Internet Topology Zoo [128] in terms of both the accuracy (with respect to the original Pareto frontier) and the run-time of the obtained solutions.

In the same spirit, Ahmadi et al. [132; 133] formulated the SDN CPP in highly-dynamic or large-scale network environments as a MOCO problem and adapted an efficient multi-objective heuristic algorithm called the Non-dominated Sorting Genetic Algorithm (NSGA-II) to find a good and diverse approximation set of the Pareto Optimal front solutions with respect to multiple competing criteria. Their work provided an extensive analysis of the trade-offs between different combinations of the crucial SDN control plane resilience and performance metrics.

Another interesting work that approaches the resilient SDN controller placement problem from a slightly different perspective is presented in [134]. Muller et al. propose a twofold controller placement scheme for improving the SDN control plane *survivability*. As opposed to previous works which generally take the shortest-path for granted when modeling connections between devices and controllers, the authors leverage the diversity of paths to place controllers at locations where the chance of controller-to-switch connec-

tivity loss in the event of failures is minimized. More specifically, they formulate the placement of controller instances as a MILP problem to maximize the number of node disjoint paths between the controllers and the assigned switches under a controller capacity constraint. The second part of their contribution involved smart recovery mechanisms that use heuristics for selecting the optimal list of backup controllers for each forwarding device based on both proximity and residual capacity considerations. A potential limitation of the resilience-oriented approach adopted by [134] is that performance aspects were overlooked. Enhancing control plane connectivity may indeed come at the cost of generating high controller-to-switch delays.

Finally, in contrast with previous placement strategies that usually optimize the controller *locations* within the network given a fixed number of SDN controllers, works found in [135–137] place an additional focus on minimizing the *number* of SDN controllers using different optimization strategies (e.g. heuristic-based approaches [133], clustering techniques [136; 138] and CPLEX solvers [139]) based on various placement constraints.

3.3 The SDN controller placement optimization problem

3.3.1 Problem statement

Ensuring a scalable and reliable distributed (but logically centralized) SDN control plane depends crucially on the placement of these physically distributed SDN controllers. More specifically, the so-called *Controller Placement Problem*[140] consists in finding the required *number* and the appropriate *locations* of the SDN controllers (among the network nodes) that efficiently partition the network into several SDN controller domains to achieve the best trade-off between performance and reliability metrics (see Figure 3.1).

3.3.2 Problem formulation

The network is viewed as a graph G = (V,E); where the set of nodes V represent the network nodes comprising controllers and switches while the set of edges E represent the links connecting these network nodes. Edge weights represent the shortest-path latencies between each pair of nodes. This information is stored in the available *Global Logical Network Topology Map* (see Section 3.4.1) where $d(s,c)$ denotes the latency from a switch node $s \in V$ to a controller node $c \in V$. We formulate the controller placement problem as a multi-objective optimization problem according to a set of performance and reliability

metrics (see Figure 3.2) that will be discussed below.

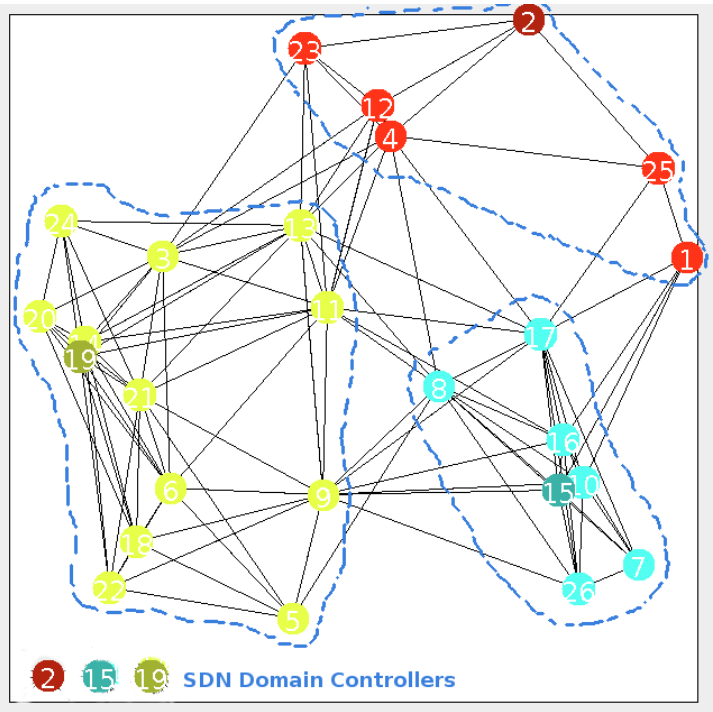


Figure 3.1: The controller placement problem

3.3.3 Placement metrics

3.3.3.1 Performance criteria

Optimizing for control plane performance is of paramount importance in large-scale IoT-like networks with stringent response-time requirements and where high propagation delays may lead to inconsistent and incorrect behaviors of network services.

In particular, the average latency and the maximum latency between the switches and their associated controllers for a given placement C of k controllers among $n = |V|$ network nodes are two different latency-related performance metrics that were first introduced by [125]. Unlike the average latency placement metric (3.1) that evaluates the overall quality of the network performance from a switch-to-controller latency point of view while hiding single cases of unacceptably high latencies, the maximum latency placement metric (3.2) is indeed useful in preventing the occurrence of such high-latency cases in placement scenarios.

- Average switch-to-controller Latency :

$$\pi^{Avg-s2c-Latency}(C) = \frac{1}{n} \sum_{(s \in S)} \min_{(c \in C)} d(s,c) \tag{3.1}$$

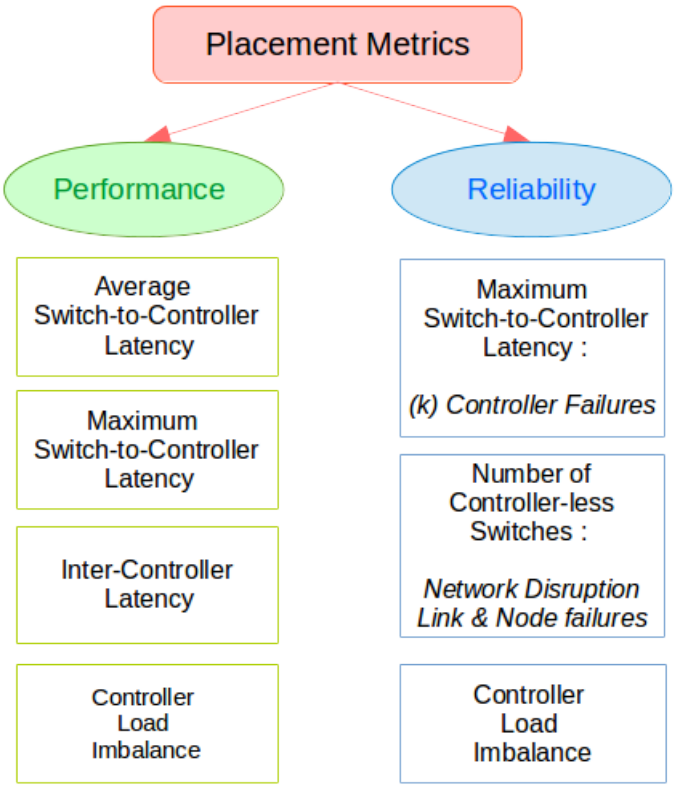


Figure 3.2: Controller placement metrics

- Maximum switch-to-controller Latency:

$$\pi^{Max-s2c-Latency}(\mathrm{C}) = \max_{(s\in \mathrm{S})} \min_{(c\in \mathrm{C})} d(s,c) \tag{3.2}$$

Other important considerations that have a direct impact on the SDN control-plane performance include the inter-controller latencies [127; 141]. Physically-distributed SDN controllers should indeed be placed as closely as possible to each other in order to reduce the cost of maintaining a consistent logically-centralized network view, i.e. the inter-controller communication and the global state synchronization. Accordingly, SDN controller locations can be selected in a way that minimizes the average and the maximum inter-controller latencies defined in (3.3) and (3.4) alongside the previously-mentioned control-to-data plane performance metrics.

- Inter-Controller Latencies:

$$\pi^{Avg-c2c-Latency}(\mathrm{C}) = \frac{1}{|\mathrm{C}|} \sum_{(c_1,c_2\in \mathrm{C})} d(c_1,c_2) \tag{3.3}$$

$$\pi^{Max-c2c-Latency}(\mathrm{C}) = \max_{(c_1,c_2\in \mathrm{C})} d(c_1,c_2) \tag{3.4}$$

Besides the above placement metrics which affect the network performance from a

switch-to-controller or controller-to-controller latency perspective, the controller capacity-awareness is another important performance factor that should be considered in the controller placement process for the purpose of avoiding the chance of controller overload and thereby preventing the related performance issues (additional delays at the controller level, etc).

One way of tackling the controller overload aspect is by assuming that all controllers C have equal capacities Q_c (in terms of the number of controlled nodes in our case) and by guaranteeing an equal distribution of the control plane load (the sum of each load $l(s)$ to control switch s [130]) among these controller instances (3.5). Each controller should be loaded at 80% of its full capacity Q_c, leaving a controller capacity margin of 20% to prevent occasional controller overload. This fair load distribution is achieved by implementing an intelligent well-balanced switch-to-controller assignment method that, given a fixed number of controller instances, assigns each network node to the closest controller provided that the load on that controller did not exceed the imposed load constraint.

Besides, the implemented controller assignment heuristic guarantees that any network node that could not be assigned to its closest controller due to the controller capacity constraint is intended for assignment to the second closest controller that has not yet reached its full load capacity.

- The Controller load constraint:

$$\sum_{(s \in S)} l(s) = 80\% Q_c, \forall c \in C \tag{3.5}$$

An alternative controller load balancing scheme is to keep the usual shortest-path-based switch-to-controller assignment method, relax the fair load balancing constraint and introduce instead an additional load imbalance metric to be minimized through the controller placement optimization (3.6). This metric is defined by [126] as the difference between the maximum and the minimum number of network nodes assigned to a controller for a given controller placement C.

- Load imbalance:

$$\pi^{Load-Imbalance}(C) = \max_{(c \in C)} n_c - \min_{(c \in C)} n_c \tag{3.6}$$

3.3.3.2 Reliability criteria

Although providing several benefits in terms of increased flexibility and better performance, the physical SDN control-to-data plane separation feature introduces additional concerns regarding network reliability as a crucial requirement for operational SDNs. As a matter of fact, one key consideration in the design of distributed SDN networks is to improve the reliability of the SDN control plane. That aspect of SDN reliability can be ensured by placing SDN controllers in a reliability-aware manner that mitigates the impact of controller failures. The most common reliability mechanism used for guarding against the failure of primary controllers is the assignment of the associated network switches to the closest working controllers. In doing so, response-time requirements should be satisfied in order to guarantee controller fault-tolerance. In other words, the propagation latencies of these previously controlled switches with respect to the new backup controllers should remain acceptable.

As an indicator of reliability against controller instance failures, we use the maximum latency metric (to be minimized) which is computed based on the propagation latencies between the network switches and all the subsets of working controllers C_1 for a placement C according to the considered controller failure scenarios F as defined in the general formula below:

$$\pi_{\mathrm{F}}^{Max-s2c-Latency}(\mathrm{C}) = \max_{(s\in \mathrm{S})} \max_{(\mathrm{C}_1\subseteq \mathrm{C})} \min_{(c\in \mathrm{C}_1)} d(s,c) \tag{3.7}$$

Among these controller failure scenarios, the worst-case scenario for a network switch would be the simultaneous failure of the $(k-1)$ closest SDN controllers. Mitigating that control plane failure scenario implies minimizing the maximum of the latencies between network switches s and their respective furthest functional controllers $\mathrm{C}_{\mathrm{F}u}(s)$ as follows:

$$\pi_{\mathrm{F}(k-1)}^{Max-s2c-Latency}(\mathrm{C}) = \max_{(s\in \mathrm{S}, c\in \mathrm{C}_{\mathrm{F}u}(s))} d(s,c) \tag{3.8}$$

In practice, it is more common for primary controller failures to occur one at a time. Therefore, reducing the impact of that controller failure scenario implies minimizing the maximum of the latencies between network switches s and their respective second closest controllers $\mathrm{C}_{Cl}(s)$ as expressed in the following :

$$\pi_{\mathrm{F(1)}}^{Max-s2c-Latency}(\mathrm{C}) = \max_{(s\in \mathrm{S}, c\in \mathrm{C}_{Cl}(s))} d(s,c) \tag{3.9}$$

3.4 The proposed SDN controller placement scheme

3.4.1 The adopted approach

In this section, we present our two-phase approach to modeling and tackling the controller placement problem using a decentralized simulation framework. At the first stage, we deploy monitoring (data-gathering) mechanisms in order to gather and transmit information about the network topology. This collected information is then used by the controller placement optimization algorithms that we implemented in the second phase of the work.

For a given network, we start by running a distributed leader election scheme. The network nodes communicate with their neighboring nodes by sending leader request messages and then waiting for leader responses. In the meantime, nodes that did not receive a leader reply message may declare themselves as leaders depending on a given leader election probability. This task ensures that each network node is managed by one leader node and that each elected leader node will assume responsibility for some part of the network.

Once the leader election process is completed, the network nodes start sending messages in order to record the desired information about their connected neighbors in a *Neighbor Map* (latency information in our case). At this point, all follower nodes send their neighbor maps to their respective leaders. In this way, leader nodes get the cluster information required to construct their *Local Leader Maps*.

Finally, leaders synchronize their local cluster information and build the *Global Physical Network Topology Map*. Among the set of network leader nodes, only one is nominated as the *Hyper Leader Node* that will be responsible for running the Dijikstra shortest path algorithm and building the *Global Logical Network Topology Map*.

At the Hyper node level, controller placement optimization algorithms are implemented and run based on this available global network view and based on a determined number of network controllers k. Controller placement solutions are then investigated and analyzed in order to find the optimal trade-off between the considered reliability and performance metrics.

3.4.2 Multi-criteria placement algorithms

In order to optimize the placement of k SDN controllers according to the discussed performance and reliability metrics, we use two different algorithms, a clustering algorithm based on PAM (Partitioning Around Medoids) and a modified genetic algorithm called NSGA-II (Non-dominated Sorting Genetic Algorithm II).

PAM [142] is a k-Medoid clustering technique that partitions the data set of N objects (N network nodes) into k clusters represented by k medoids (the SDN controller nodes). The main idea of PAM is to find the optimal set of medoids that improves the overall quality of clustering which is measured based on the average dissimilarity of all data objects to their nearest medoid. In our case, all the considered metrics M are of equal importance, thereby making the dissimilarity function D (to be minimized) for a given placement $C \in CP$ (the considered placement configurations) computed as the normalized sum of all weighted objectives O with the associated weights equal to $\frac{1}{M}$ as follows:

$$D^{PAM-B}(C) = \sum_{i \in M} (\frac{1}{M}) \times N(O_i) \tag{3.10}$$

where:

$$N(O_i) = \frac{O_i(C) - \min_{(C \epsilon CP)} O_i(C)}{\max_{(C \epsilon CP)} O_i(C) - \min_{(C \epsilon CP)} O_i(C)}$$

Algorithm 1, called PAM-B, corresponds to the multi-criteria controller placement algorithm that we developed based on PAM.

Algorithm 1 PAM-B:

1: n nodes, an integer k.
2: Init : Select k nodes at random and define them as medoids.
3: Associate each object to the appropriate medoid according to a well-defined assignment method.
4: **for** each medoid m **do**
5: Compute and store the objective function values of the current configuration.
6: **for** each non-medoid r **do**
7: Swap m and r
8: Associate each object to the appropriate medoid according to the considered assignment method.
9: Compute and store the objective function values of the new configuration.
10: **end for**
11: Compute the maximum and the minimum values of each objective function over the considered configurations.
12: Compute the normalized total dissimilarity of each considered configuration based on the above optima.
13: Select the configuration with the lowest normalized total dissimilarity
14: **end for**

On the other hand, NSGA-II [132] is a popular fast and elitist genetic algorithm for multi-objective optimization. In addition to the classical genetic operators (crossover and mutation), NSGA-II uses other multi-objective ranking mechanisms (non-dominated sorting and the crowding distance) for creating the next generation population of candidate solutions. The main idea of NSGA-II is to make that population evolve towards a set of optimal non-dominated solutions (the Pareto front) representing the best trade-offs between the considered objectives.

In this work, we set a list of NSGA-II parameters as follows:

Parameters	Values
Population	k * 2
Selection Operator	sbx
MaxEvaluations	Depends on the Strategy (see Table 3.2)

Table 3.1: NSGA-II parameters

3.4.3 Gradual strategies

We propose multiple strategies for tackling the SDN controller placement problem according to our performance and reliability criteria. In doing so, we follow a step-by-step

approach based on the gradual incorporation of these placement metrics for assessment by our muti-criteria algorithms (PAM-B and NSGA-II). That way, it becomes possible to investigate the direct impact of these placement metrics on the quality of the controller placement solutions and also to make the controller placement approach adaptable to various use cases. More importantly, such a versatile approach can be leveraged by SDN operators to assist them in finding their optimal controller placement solution tailored to their specific context.

Strategy 1 : *A Latency-based Strategy*

This strategy solves the SDN controller placement problem based on the two latency-related performance metrics shown in (3.1) and (3.2) while keeping the simple usual shortest-path switch-to-controller assignment method. Accordingly, the multi-objective NSGA-II is launched with these two objectives to be minimized while PAM-B minimizes the following dissimilarity function as a normalized sum of the two considered objectives in accordance with (3.10):

$$D_1^{PAM-B}(C) = (\frac{1}{2}) \times N(\pi^{Avg-Latency}(C)) + (\frac{1}{2}) \times N(\pi^{Max-Latency}(C)) \quad (3.11)$$

Strategy 2 : *Strategy 1 under a load capacity constraint*

Strategy 2 incorporates, in addition to the previously-mentioned latency-related performance metrics, a fair load balancing scheme by turning the simple switch-to-controller assignment method of Strategy 1 into an intelligent assignment method that guarantees an equal distribution of the control plane load among controller instances (80% of their equal capacities) and, at the same time, a fair affectation of network switches to their closest lightly loaded controllers (see Section 3.3.3.1). Thus, in Strategy 2, PAM-B minimizes the same dissimilarity function (3.11) used in Strategy 1.

Strategy 3 : *Strategy 1 with a load imbalance metric*

In Strategy 3, along with the performance metrics of Strategy 1, we adopt an alternative load balancing scheme using the load imbalance metric (3.6) proposed by [126] and we investigate the controller overload risk.

The following formula defines the dissimilarity function of PAM-B based on the three con-

sidered objectives:

$$D_3^{\text{PAM-B}}(C) = (\frac{1}{3}) \times N(\pi^{Avg-Latency}(C)) + (\frac{1}{3}) \times N(\pi^{Max-Latency}(C)) + (\frac{1}{3}) \times N(\pi^{Load-Imbalance}(C)) \quad (3.12)$$

Strategy 4 : *Strategy 3 with reliability metrics*

Strategy 4 provides a rich SDN controller placement optimization framework that includes reliability metrics (explained in Section 3.3.3.2) along with performance metrics. As for reliability placement metrics (3.7), users of the framework have the option of including a reliability metric variant that tackles the worst-case controller failure scenario (3.8) or a variant that addresses a more common controller failure scenario (3.9), in addition to the previous performance metrics (3.1), (3.2) and (3.6).

The dissimilarity functions of PAM-B for both variants are calculated in accordance with the following formulas:

$$D_4^{\text{PAM-B}(k-1)}(C) = (\frac{1}{4}) \times N(\pi^{Avg-Latency}(C)) + (\frac{1}{4}) \times N(\pi^{Max-Latency}(C)) + (\frac{1}{4}) \times N(\pi^{Load-Imbalance}(C)) + (\frac{1}{4}) \times N(\pi_{F(k-1)}^{Max-Latency}(C)) \quad (3.13)$$

$$D_4^{\text{PAM-B}(1)}(C) = (\frac{1}{4}) \times N(\pi^{Avg-Latency}(C) + (\frac{1}{4}) \times N(\pi^{Max-Latency}(C)) + (\frac{1}{4}) \times N(\pi^{Load-Imbalance}(C)) + (\frac{1}{4}) \times N(\pi_{F(1)}^{Max-Latency}(C)) \quad (3.14)$$

3.5 Performance evaluation

3.5.1 Simulation settings

In this work, we use the JAVA-based distributed simulation framework Sinalgo (Simulator for Network Algorithms) [143] for implementing our two-phase approach (explained in Section 3.4.1) and evaluating our multi-criteria SDN controller placement algorithms (see Section 3.4.2) according to gradual strategies and various scenarios (see Section 3.4.3).

Table 3.2 summarizes the values corresponding to the *maximum number of objective function evaluations* simulation parameter used as a stopping criterion in NSGA-II algorithm as a function of the number of objectives involved in each strategy and the size of the network in each simulation scenario.

Number of Objectives	Number of Nodes	MaxEvaluations
2 (Strategy 1 and 2)	20, 60, 100, 200, 400 500, 600 700 800 900 1000	10 000 20 000 40 000 60 000 80 000 100 000
3 (Strategy 3)	20, 60, 100, 200, 400 500, 600 700 800 900 1000	20 000 40 000 60 000 80 000 100 000 120 000
4 (Strategy 4)	20, 60, 100, 200, 400 500, 600 700 800 900 1000	40 000 60 000 80 000 100 000 120 000 140 000

Table 3.2: The maximum number of objective function evaluations (MaxEvaluations)

3.5.2 Simulation results

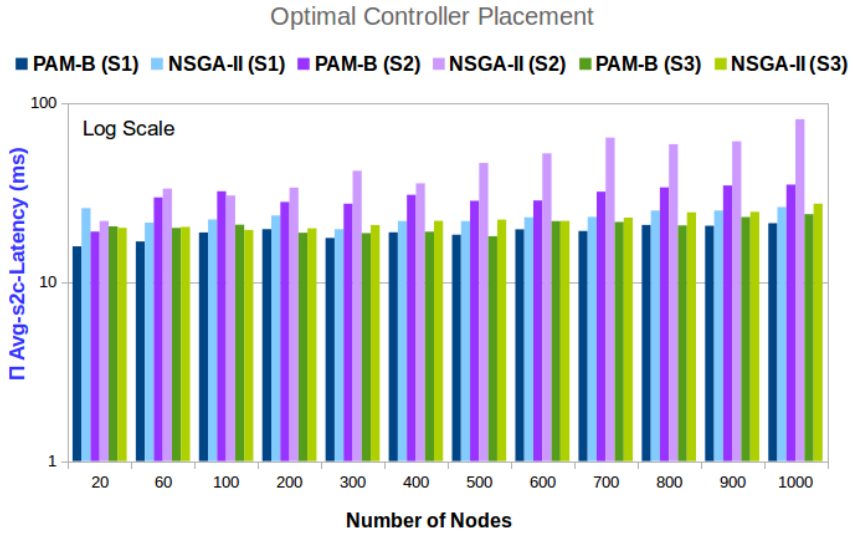
In this Section, we show the simulation results of the proposed approach based on the followed gradual strategies and according to different simulation scenarios. For each strategy, and for a given network topology, we evaluate the controller placement solutions proposed by PAM-B and NSGA-II. PAM-B generates the optimal controller placement clustering solution with respect to the equally-weighted dissimilarity measure defined in (3.10) in which we give equal importance to the considered objectives.

Likewise, for the multi-objective NSGA-II, we consider the fairest controller placement solution (in relation to the desired criteria) among all the generated non-dominated Pareto Optimal solutions representing the possible trade-offs between the considered objectives. This is achieved by selecting the Pareto placement solution S that best reduces the total gap between all the associated objectives M and their respective optimal values across the set of all Pareto optimal solutions P. In our case, this implies considering the

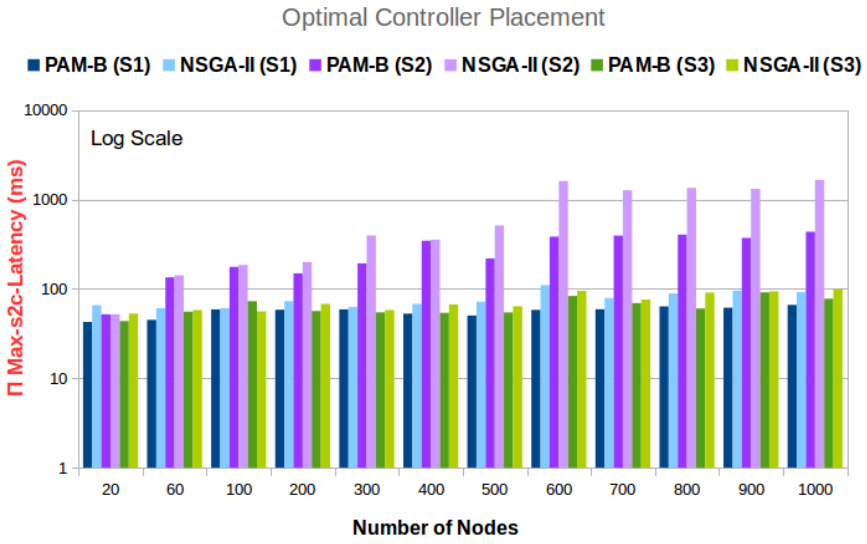
Pareto solution with the minimum value of the following measure (3.15) :

$$a(S) = \sum_{i \in M} (\frac{1}{M}) \times \frac{O_i(S) - \min_{(S \epsilon P)} O_i(S)}{\max_{(S \epsilon P)} O_i(S) - \min_{(S \epsilon P)} O_i(S)} \tag{3.15}$$

Accordingly, several simulation scenarios are performed following the considered strategies and using various types of network topologies of different size; from 20 up to 1000 network nodes. That allowed us to compare our controller placement strategies, analyze the performance of both algorithms for solving the controller placement optimization problem, and also study the scalability of our approach which is mainly intended for large-scale IoT-like deployments.



(a)

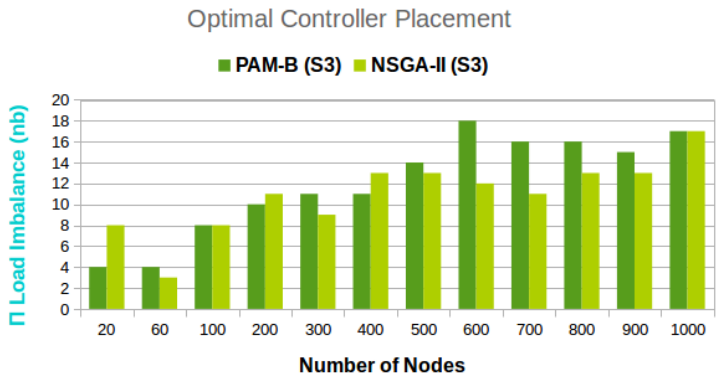


(b)

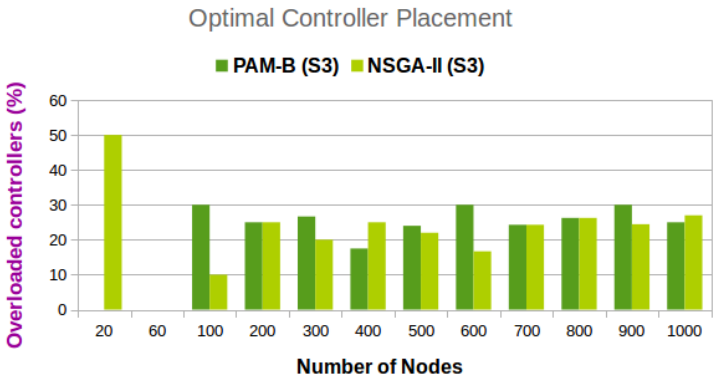
Figure 3.3: Strategy 1, 2 and 3: Latency-based performance metrics

When comparing the optimal controller placement solutions across Strategy 1, 2 and 3 with respect to the considered latency-based performance metrics (see Figure 3.3), we notice that, unlike Strategy 1 and 3 which show similar performance trends, Strategy 2 yields poorer results in minimizing both the Average Latency (3.3(a)) and the Maximum Latency (3.3(b)) performance metrics due to the imposed load balancing constraint. For example, in scenario n°12 (3.3(b)) where the network size is equal to 1000 nodes, both PAM-B and NSGA-II provided, according to Strategy 2, controller placement configurations where the Maximum Latency value is above 400ms compared to less than 100ms for both Strategy 1 and 3. On the other hand, we note that, in all the 12 simulation scenarios considered by these three Strategies, PAM-B is obviously better than NSGA-II at simultaneously minimizing both the Average Latency and the Maximum Latency performance metrics of the obtained controller placement configurations. For instance, when it comes to the Average Latency metric, PAM-B according to Strategy 1 is better (from 6% to 40%) than NSGA-II over all scenarios, (from 10% to 50%) according to Strategy 2 and (up to 20%) according to Strategy 3.

That said, the obvious advantage of Strategy 3 that adds a Load Imbalance metric to strategy 1 (3.4(a)) over Strategy 2 that incorporates a load balancing constraint in strategy 1 is the fact that it did not deteriorate the level of latency-based performance targeted by Strategy 1. However, the potential drawback of Strategy 3 is related to the risk of controller overload as illustrated by Figure 3.4(b) which depicts the percentage of overloaded controllers in the considered scenarios. For instance, in scenario n°10 (3.4(b)) where the network size is equal to 800 nodes and the number of controllers is equal to 80, both PAM-B and NSGA-II produced controller placement configurations where 21 (26,25%) of these controllers are overloaded.



(a)



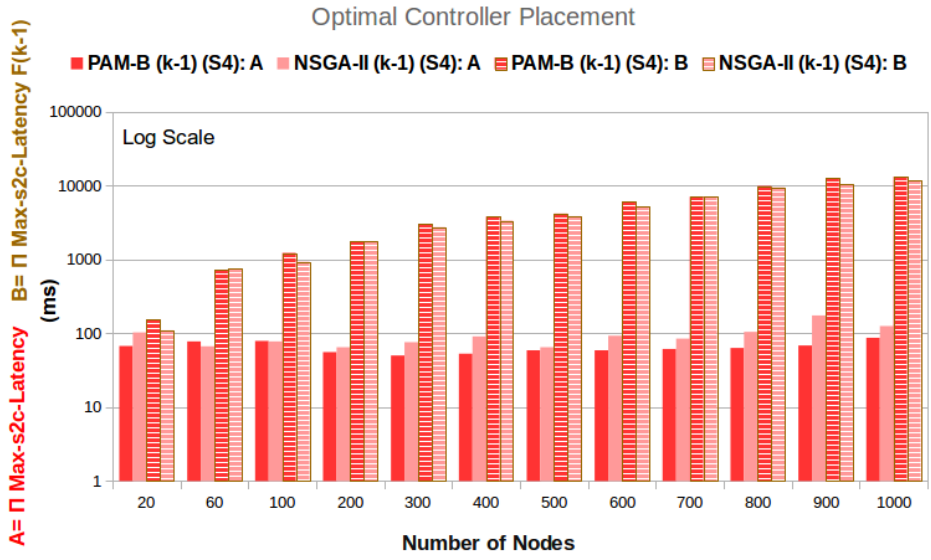
(b)

Figure 3.4: Strategy 3: Load imbalance

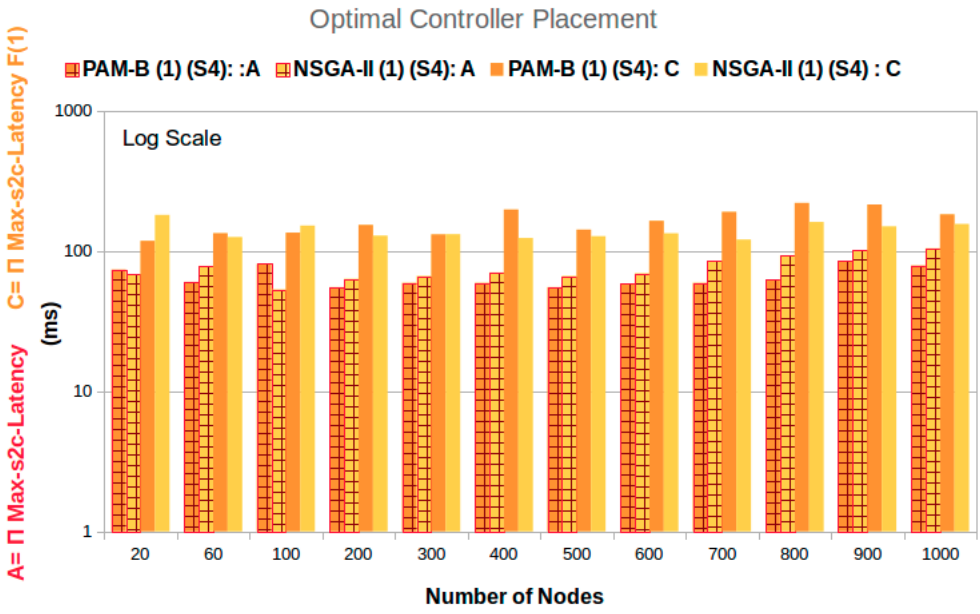
As explained in 3.4.3, Strategy 4 involves reliability metrics in addition to the set of performance metrics considered by Strategy 3. In particular, Figure 3.5 compares, for each variant of Strategy 4 according to all the optimal controller placement solutions, the values of the Maximum Latency metric in the failure free case with that of the Maximum Latency metric in the considered failure case scenario. It also shows that PAM-B and NSGA-II perform in a quite similar fashion when optimizing these metrics.

Figure 3.6 investigates the performance cost of taking into account reliability criteria in the controller placement optimization process. Surprisingly, optimizing for reliability metrics, did not severely impact performance metrics like the Maximum Latency (in the failure free case) (3.6(a)) whose values remained acceptable and comparable to that in Strategy 3 except for a few placement scenarios that were in most cases produced by NSGA-II. Likewise, similar trends are observed across Strategy 3 and 4 for each of the Load Imbalance (3.6(b)) and the Average Latency (3.6(c)) performance metrics of the obtained placement configurations. For example, in scenario n°12 (3.6(c)), PAM-B(k-1) (re-

spectively PAM-B(1)) according to Strategy 4 produced an optimal controller placement configuration where the value of the Average Latency metric is equal to 25,3ms (respectively 23,5ms) compared to 24ms for PAM-B according to Strategy 3. In the same scenario, NSGA-II(k-1) (respectively NSGA-II(1)) according to Strategy 4 generated a controller placement configuration with an Average Latency value equal to 27,6ms (respectively 26,8ms) against 27,3ms for NSGA-II according to Strategy 3.

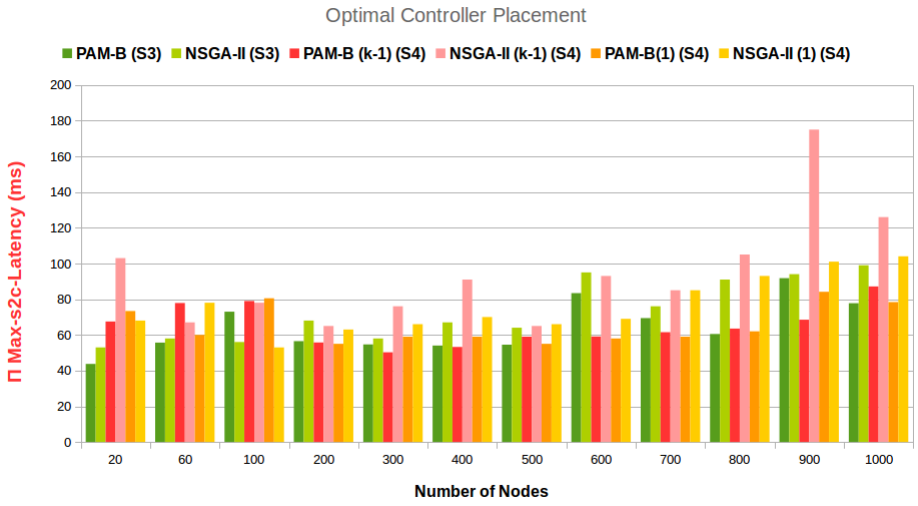


(a)

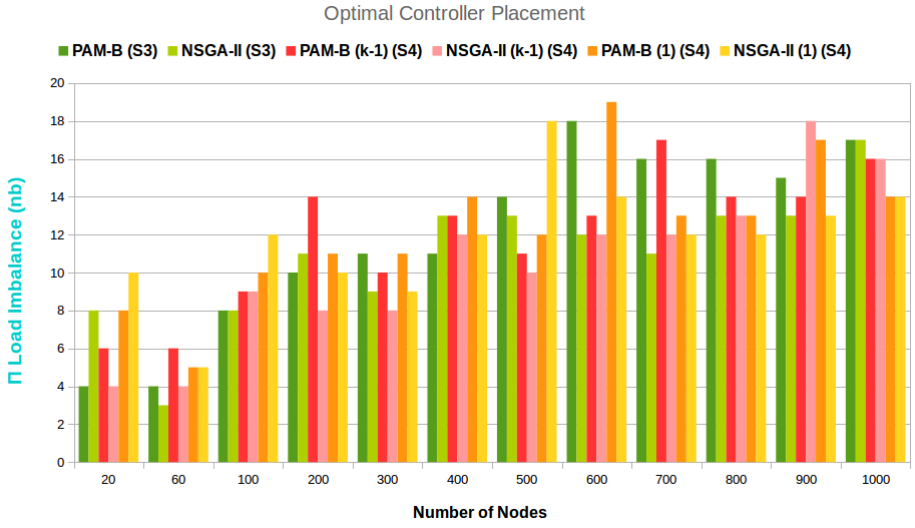


(b)

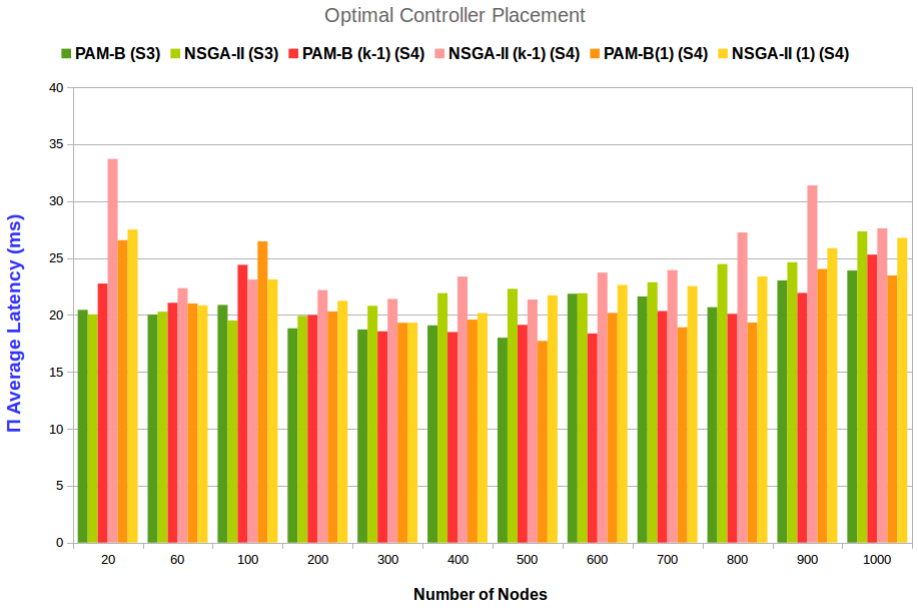
Figure 3.5: Strategy 4: Reliability metrics:
(Maximum latencies in failure free & failure case scenarios)



(a)



(b)



(c)

Figure 3.6: Strategy 4: Performance metrics

3.6 Discussion

The four strategies are put forward to account for some important aspects of the SDN controller placement problem. The first strategy investigates an optimal placement of SDN controllers based on both the average and the maximum latency criteria. Optimizing for such latency-related performance metrics has guaranteed proper shortest-path switch-to-controller assignments. On the other hand, a proper placement of controllers where inter-controller communication costs are optimized requires taking into account additional latency-related metrics such as the latencies between the individual controllers. However, such considerations are beyond the scope of this chapter; Inter-controller communication effects [144] are indeed issues that need further exploration in our following work about the knowledge dissemination part of distributed SDN control.

The second and the third strategies were motivated by the observation that optimizing the locations of controllers based solely on latency-related metrics as in Strategy 1 may generate placement configurations where some controllers are in charge of a big number of network switches, and thus highly exposed to potential overload risks. Imposing a load balancing constraint complemented by an intelligent switch-to-controller assignment method (Strategy 2) guarantees a fair distribution of the control plane load, where SDN controllers are equally loaded at 80% of their total capacity. However, it is intuitively likely that, despite that intelligent assignment technique, some switches will be constrained for assignment to controllers that are relatively far (from a latency point of view) because all their closest controllers have somehow reached the imposed load constraint. On the other hand, relaxing that load constraint and substituting it with a load imbalance metric (Strategy 3), has proved effective in providing switches with better freedom to join their preferred controller cluster, but less immune to the risk of controller overload (see Figure 3.4(b)). A potential solution to addressing this problem could be the implementation of a heuristic method to be launched at the end of Strategy 3 in order to cope with such controller overload cases. One way of doing this, is to turn, in each overloaded controller cluster, a certain number of switches (the closest to the cluster controller) into additional controllers that will handle the extra controller cluster overload. For each overloaded cluster, the number of switches to become controllers might depend on the surplus number of cluster switches above the cluster controller capacity. In all the scenarios considered by Strategy 3, the maximum controller load has never exceeded

200% of its total capacity, thereby confirming the need for no more than one additional controller in each overloaded cluster.
Considering controllers with different capacities as in real network settings is another important factor to take into account. Another controller placement strategy can be proposed and implemented using a load balancing scheme based on different node capacities.

The fourth strategy incorporates reliability metrics in addition to the considered performance metrics. For example, the reliability-aware controller placement that takes into account worst-case failure scenarios, seems to impact the locations of controllers in a way that places them closer to the network center to minimize worst-case latencies with respect to all network switches and thus to ensure an optimized switch-to-controller backup reassignment that preserves performance in case of primary controller outages. However, besides controller failures, other failure scenarios like the failure of network switches and links might occur and can therefore be involved in the controller placement decision-making process in order to enhance the reliability of the SDN control plane.

Evaluating PAM-B and NSGA-II over the proposed strategies revealed that, in most scenarios, PAM-B outperforms NSGA-II in terms of the quality of final solutions with respect to the considered metrics (see Section 4.6.2). More specifically, PAM-B gives a more balanced trade-off between performance and reliability metrics, and more importantly, it produces more stable results over all strategies whereas the performance of NSGA-II with respect to these metrics is sometimes unpredictable and highly dependent on the followed strategy.

Finally, implementing different kinds of heuristic-based algorithms (PAM-B and NSGA-II) that are directed at solving the controller placement problem within a reasonable time frame and testing them over large network instances demonstrated the scalability of the proposed approach and its adequacy with the IoT context. In particular, Figure 3.7 illustrates the run-time comparison between PAM-B and NSGA-II which reflects similar trends over the considered scenarios. In fact, the computational complexities of PAM-B and NSGA-II are close and respectively equal to, $O(k(n-k)^2)$ (k is the number of medoid clusters; the number of SDN controllers in our case, and n is the number of objects; the network size in our case), and $O(MN^2)$ (M is the number of objective functions and N is the population size).

However, it is worth mentioning that the computation time of clustering approaches

like PAM can be significantly improved. For instance, CLARA (CLustering LARge Applications) [145], a sampling-based variant of PAM, is highly recommended for dealing efficiently with large data sets. It has indeed a computational complexity of $O(ks^2 + k(n-k))$, where k is in our case the number of SDN controllers, n is in our case the network size and s is the sample size. As a matter of fact, a CLARA-based approach can be used instead of PAM-B in large-scale network scenarios in order to further reduce the overall computation time.

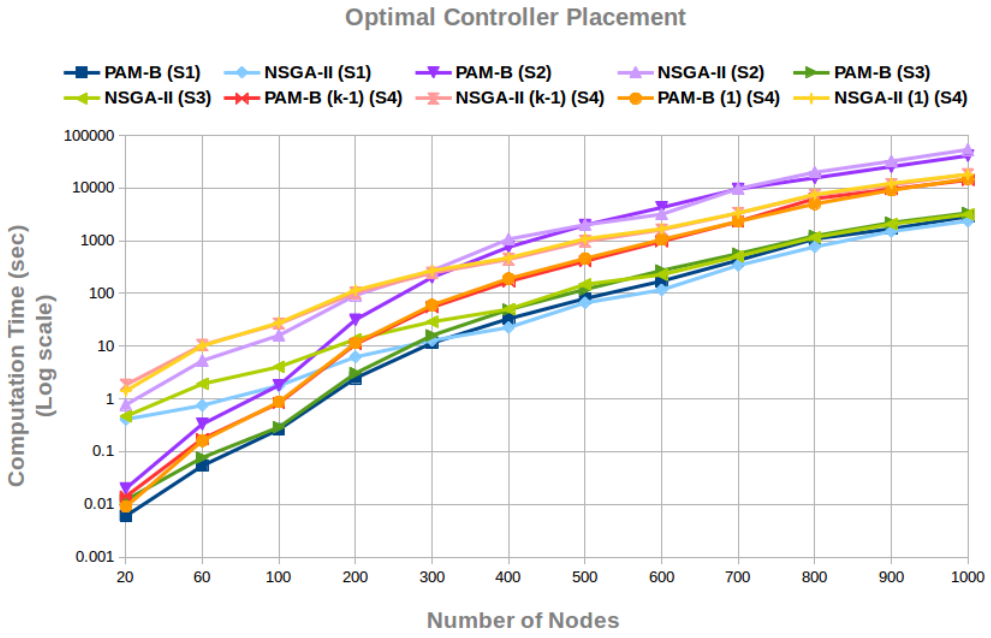


Figure 3.7: Computation time comparison between PAM-B and NSGA-II over the considered strategies

3.7 Conclusion

In this work, we investigated the SDN controller placement issue in the context of large-scale IoT-like networks and underlined the need for an efficient approach to this multi-objective optimization problem. In that respect, several SDN control plane performance and reliability metrics were considered according to different needs and contexts. Throughout these strategies, two heuristic approaches were proposed to find high-quality approximate solutions to the controller placement problem in a reasonable computation time: A clustering approach (PAM-B) based on a dissimilarity score and a modified genetic approach (NSGA-II). Our results demonstrated the potential of clustering techniques in delivering appropriate controller placement configurations that achieve balanced trade-offs

among the competing performance and reliability criteria.

The challenge of determining the required number and locations of SDN controllers represents one particular aspect of the overall process of addressing the distributed SDN control problem. This leads us to the second key aspect that calls for further investigation, namely the knowledge sharing challenge associated with such logically-centralized distributed SDN platforms. However, given the potential correlation between placing multiple SDN controllers and modeling the type of communication among them, it becomes essential to reevaluate certain factors involved in the controller placement problem-solving after studying the variety of data consistency models for inter-controller communication in the context of physically distributed SDN architectures. It is also planned to implement and validate the approaches evaluated in simulation on an experimental SDN testbed based on OpenvSwitch [146] nodes.

Chapter 4

Adaptive and continuous consistency for distributed SDN controllers: Anti-Entropy reconciliation mechanism

« Power is gained by sharing knowledge, not hoarding it »

Maria Khan

Contents

4.1 Introduction

Logically-centralized but physically-distributed SDN controllers are mainly used in large-scale SDN networks for scalability, performance and reliability reasons. These controllers host various applications that have different requirements in terms of performance, availability and consistency. Current SDN controller platform designs employ conventional strong consistency models so that the SDN applications running on top of the distributed controllers can benefit from strong consistency guarantees for network state updates.

However, in large-scale deployments, ensuring strong consistency is usually achieved at the cost of generating performance overheads and limiting system availability. That makes weaker optimistic consistency models such as the eventual consistency model more attractive for SDN controller platform applications with high-availability and scalability requirements. In this chapter, we argue that the use of the standard eventual consistency models, though a necessity for efficient scalability in modern SDN systems, provides no bounds on the state inconsistencies tolerated by the SDN applications.

To remedy that, we propose an adaptive multi-level consistency model for the distributed ONOS controllers following the notion of continuous and compulsory [147] eventual consistency, where network application states adapt their eventual consistency level dynamically at run-time based on the observed state inconsistencies under changing network conditions. That model presents many advantages when compared to the strong consistency and eventual consistency extremes, especially in large-scale deployments.

Our scalable consistency adaptation strategy was implemented for a source-routing application on top of the distributed open-source ONOS controllers: It mainly consists in turning ONOS's eventual consistency model into an adaptive consistency model using the *Anti-Entropy reconciliation period* as a *control knob* for an adaptive fine-grained tuning of consistency levels. As compared to ONOS's static state consistency management scheme at scale, our consistency strategy is aimed at minimizing state synchronization overheads while taking into account the application's continuous consistency Service-Level Agreements (SLAs) (e.g. *Numerical error* bounds) and without compromising the application requirements of high-availability.

The remainder of this chapter is organized as follows: In Section 4.2, we provide an overview of the consistency models used by state-of-the-art SDN controller platforms. In Section 4.3, we review the consistency problem in SDN and investigate the involved

consistency trade-offs. In Section 4.4, we discuss the consistency models implemented in the ONOS controller platform. In Section 4.5, we present our consistency adaptation approach for the distributed ONOS controllers. Finally, Section 4.6 shows and discusses the experimentation results.

4.2 Related work

The challenges related to consistency in distributed SDN control have been recently addressed in the SDN literature. Some works focused on the impact of switch-to-controller state consistency *between switches and controllers* on network application performance. Reitblatt et *al.* [148] studied the consistency of controller-driven flow updates in terms of network policy conservation. They proposed a new type of consistency abstractions to enforce a consistent forwarding state at different levels (per-flow consistency and per-packet consistency). Recent approaches [149] focused on efficiently updating the network data plane state while preventing forwarding anomalies at the switches and maintaining desired consistency properties (e.g. loop and black-hole freedom).

Another category of works, falling within the scope of this chapter, focused on achieving controller-to-controller state consistency *between the distributed controllers* without compromising application performance.

Current implementations of distributed SDN controller platforms offer different state consistency abstractions. They use static mono-level [64] or multi-level [150] consistency models such as the strong, eventual and weak state consistency levels.

Onix [38] offers two separate dissemination mechanisms for synchronizing network state updates between the NIBs stored at the controller instances. These mechanisms are based on two implemented data-store options: A replicated transactional database designed for ensuring strong consistency at the cost of good performance for persistent but slowly-changing states, and a high-performance memory-only distributed hash table (DHT) for volatile states that are tolerant to inconsistency.

Similarly, ONOS [150] provides different state sharing mechanisms to achieve a consistent network state across the cluster of ONOS controllers. More specifically, ONOS's distributed core eases the state management and coordination tasks for application developers by providing them with an available set of core building blocks for dealing with different types of distributed control plane states, including a consistent primitive for state requiring strong consistency and an eventually consistent primitive for state toler-

ating relaxed consistency.

On the other hand, ODL [64] supports a strong consistency model in its distributed datastore architecture. In fact, all the data shared across the cluster of controllers for maintaining the logically centralized network view is handled in a strongly-consistent manner using the RAFT consensus algorithm [66].

Recent approaches to handling the issues of controller state consistency [151–153] recommended the use of adaptive consistency for the distributed SDN controller platforms. Aslan et *al.* [151] attempted to mitigate the impact of controller state distribution on SDN application performance by proposing an adaptive tunable consistency model following the delta consistency model. In their model, the automatic control plane adaptation module tunes the consistency level (the synchronization period parameter) based on an application-specific performance indicator that is measured given the current state of the network. To assess their approach, the authors compared the performance of the distributed load-balancing application running on top of adaptive and non-adaptive controllers.

The same authors studied in [154] the feasibility of employing adaptive controllers that are built on top of tunable consistency models similar to that of Apache Cassandra. They presented an adaptation strategy for the SDN controllers that uses clustering techniques to map a given application performance indicator into an appropriate consistency level that can be used to configure the parameters associated with the underlying tunable consistency model. However, the authors did not test the validity of their proposal using a specific SDN application running on top of an SDN controller platform.

In the same spirit, the work described in [152] put forward an adaptive consistency model for distributed SDN controllers following the Eventual Consistency level. The main aim of changing the controller consistency level on-the-fly was to maintain a scalable system that sacrifices application optimality for less synchronization overhead. Accordingly, the authors propose a cost-based approach that bounds the correctness to a tunable threshold, where the consistency level is adapted based on the effort of state convergence after the expiration of a non-synchronization period, and the application inefficiencies due to operations with stale state. The performance of the proposed model was evaluated based on a specific routing application.

4.3 The consistency problem in SDN

4.3.1 Consistency trade-offs in SDN

In distributed SDN architectures, the SDN control plane supports the interaction between multiple controllers through their "east-west" interfaces. Inter-controller communications are indeed needed to synchronize the controllers' shared data structures to maintain a consistent global network view, and therefore ensure the correct behavior of the network applications running on top of the distributed controllers. Such control traffic can be in-band or out-band.

However, distributing the network control state across the SDN control plane affects the performance objectives of the control applications. In fact, many state distribution trade-offs arise as discussed by Levin et *al.* such as the trade-off between application state consistency/staleness (state synchronization overhead) and application performance (objective optimality), and the trade-off between application logic complexity and robustness to inconsistency.

More generally, Brewer's CAP theorem applied to networks [108] investigates the involved trade-offs between Consistency (C), Availability (A), and Partition-tolerance (P). It states that, in SDN networks, it is generally only possible to achieve two out of the three desirable properties: CA, CP or AP.

However, in the context of modern and scalable distributed database systems (DDBS), Abadi's PACELC theorem [155] is believed to be more relevant, as it combines in a single complete formulation, the CAP theorem trade-offs, and in the absence of partitions (E), the Latency (L)/Consistency (C) trade-off. Many popular modern DDBSs do not by default guarantee strong consistency, as stated by CAP. Conversely, they come with trade-offs that are better warranted/represented by the PACELC alternative. For example, Amazon's Dynamo [113], Facebook's Cassandra [111], and Riak are PA/EL systems, MongoDB is PA/EC, yahoo's PNUTS is PC/EL, and finally BigTable and HBase are PC/EC systems.

In this context, we argue that SDN is bringing the network design much closer to the design of distributed database systems. In the same spirit, we argue that PACELC can apply to large-scale SDN controller platforms, in the same way it applies to modern and scalable NoSQL DDBSs.

4.3.2 Consistency models in SDN

Many architectures have been proposed to support distributed SDN controllers, with the goal of improving the scalability, reliability and performance of SDNs. Two main consistency models are used by current controller platforms:

4.3.2.1 The strong consistency model

In SDN, the strong consistency model guarantees that all controller replicas in the cluster have access to the most updated network information at all times. That comes at the cost of increased state synchronization delay and communication overhead, especially in large-scale deployments. In fact, strong consistency relies on a blocking synchronization process that keeps the switches from reading the data, unless the controllers are fully updated, thereby affecting network availability and scalability.

Strong consistency is a requirement for certain applications that favor consistency and correctness properties over availability. In current controller platforms, strong consistency is usually achieved using Paxos, RAFT [66] and similar protocols.

4.3.2.2 The eventual consistency model

In SDN, the eventual consistency model takes a relaxed approach to consistency by assuming that all controller replicas will "eventually" converge and become consistent throughout the network. That means that controllers may temporarily present an inconsistent network view, allowing for some stale data to be read, and potentially causing a transient incorrect application behavior.

Many applications opt for eventual consistency to guarantee high-availability and performance at scale. Modern DDBSs, including Dynamo [113] and Cassandra [111], support eventual consistency settings by default in exchange for extremely high availability (fast data access) and scalability.

4.3.2.3 Adaptive consistency models

Recent research in SDN [151–153] has introduced the concept of adaptive consistency in the context of distributed SDN control. Unlike static consistency approaches, adaptively-consistent controllers adjust their consistency level at run-time to reach the desired application performance and consistency requirements. That alternative offers many benefits: It spares application designers the task of developing complex applications that re-

quire implementing multiple consistency models, it provides applications with robustness against sudden network conditions, and it reduces the overhead of state distribution across controllers without compromising application performance [151].

In the community of modern database systems, the need for adaptable consistency where the consistency level is decided dynamically over-time based on various factors has been recognized. Many adaptive consistency models have been proposed, such as the QUOROM-based consistency [156], RedBlue Consistency [157], Chameleon and Harmony [158], the delta consistency [151], and the continuous consistency [147]. In addition, most of modern database systems such as Cassandra [111] and Dynamo [113] are currently equipped with an adaptive consistency feature, offering multiple consistency options with tunable parameters for application developers.

In our opinion, all the above consistency models could be leveraged by the SDN community to build adaptively-consistent SDN controllers.

4.4 Consistency models in ONOS

In this work, we are particularly interested in the open-source Java-based ONOS controller [150]. In this section, we describe in detail the ONOS approach to state consistency in a distributed controller setting.

To achieve high-availability, scale-out and performance, the ONOS controller platform supports a physically-distributed cluster-based control plane architecture, where each controller is responsible for handling the state of a subsection of the network. To maintain the logically-centralized network view, local controller state information is disseminated across the cluster in the form of events that are shared via ONOS's distributed core. The latter consists of core subsystems tracking different types of network states being stored in distributed data structures and requiring different coordination strategies. Two main state consistency schemes are implemented in ONOS's subsystem stores to provide two different levels of state consistency: strong consistency and eventual consistency.

4.4.1 Strong consistency in ONOS

To ensure strong consistency among replicated network states, ONOS uses (since version 1.4) the Atomix framework that is based on the RAFT consensus protocol [66]. For instance, the store for switch-to-controller mastership (mapping) management is handled

in a strongly consistent manner using that framework. Besides, ONOS exposes a set of core distributed state management primitives that can be leveraged by application developers to implement their application-specific stores. In this respect, applications whose state is maintained in a strongly consistent fashion can leverage the `ConsistentMap` distributed primitive, which guarantees strong consistency for a distributed key-value store.

4.4.2 Eventual consistency in ONOS

For eventually-consistent behaviors, ONOS employs an optimistic replication technique complemented by a background gossip/anti-entropy protocol. For instance, the stores for devices, links, and hosts are managed in an eventually-consistent manner. The distributed topology store is also eventually consistent since it relies on the distributed versions of the device, link and host stores. For the eventual consistency option, ONOS offers the `EventuallyConsistentMap` distributed primitive for control programs and applications. The latter can create different instances of these primitives for managing their eventually-consistent application-specific states.

4.4.2.1 Optimistic replication

Optimistic replication is a key technology that is used in large-scale distributed data sharing systems, meeting the goal of achieving higher availability and scalability as compared to strongly-consistent systems. That strategy for replication propagates changes in the background, and discovers conflicts after they occur. It is based on the "optimistic" assumption that inconsistencies rarely occur and that replicas will converge after some time, thus providing eventual consistency guarantees.

In ONOS, optimistic replication is used in the distributed maps. Whenever an update occurs in the store managed by one controller, the associated `EventuallyConsistentMap` replicates events immediately to the rest of the controllers. That means that maps on each controller will get closely in sync (apart from a small propagation delay) in case the controllers are functioning properly. On each controller, updates are added to an `EventAccumulator` as they are written.

4.4.2.2 Gossip-based Anti-Entropy

Controllers that purely rely on optimistic replication might progressively get out of sync, especially in the event of node failures and partitions or in the case updates get missed or

dropped. The anti-entropy protocol takes care of ensuring that replicas are back in sync by resolving discrepancies, and that the entire cluster converges fairly quickly to the same state.

In ONOS, the gossip-based anti-entropy mechanism is a lightweight peer-to-peer background process that runs periodically: At fixed intervals (3-5 seconds), each controller randomly chooses another controller, they both exchange information in order to compare the actual content (entries) of their distributed stores (based on timestamps). After synchronizing their respective topology views, the controllers become mutually consistent. This reconciliation approach proves useful in fixing controllers when their state drifts slightly, and also in quickly synchronizing a newly-joining controller with the rest of the controllers in the cluster.

4.5 The proposed adaptive consistency for ONOS

In this section, we explain our approach which is mainly aimed at optimizing the consistency management in ONOS.

4.5.1 A continuous consistency model for ONOS

As explained in 4.3.2.3, an adaptive consistency model offers many benefits for the distributed SDN controllers. In particular, the ONOS controllers can benefit from the continuous consistency model proposed in [147]. The latter is based on a middle-ware framework (called TACT) for adaptively tuning the consistency and availability requirements for replicated online services, following the continuous consistency concept. In contrast to the strong consistency model (which imposes performance overheads and limits availability), and to optimistic consistency models (which provide no bounds on system inconsistencies), the continuous consistency model explores the semantic space between these two types of traditional models: It offers a continuum of intermediate consistency models (*multi-level consistency*) with tunable parameters. These quantifiable degrees of consistency can be exploited by applications to explore, at runtime, their own trade-offs between consistency and availability, while taking into account the changing network and service conditions. More specifically, TACT bounds the amount of inconsistency and divergence among the replicas in an application-specific manner. Basically, applications specify their consistency semantics through *conits*; a set of metrics that capture the consistency spectrum: *Numerical Error*, *Order Error*, *Staleness*.

Hence, for each conit, consistency is quantified continuously along a three-dimensional vector:

$$Consistency = (NumericalError, OrderError, Staleness) \tag{4.1}$$

Numerical Order bounds the discrepancy between the value delivered to the application client and the most consistent "final" value. *Order Error* bounds inconsistency by the number of tentative/unseen writes at any replica. *Staleness* places a real-time bound on the delay for propagating the writes among the replicas.

In our opinion, many features from the discussed continuous consistency spectrum can indeed be incorporated when rethinking the ONOS strategy to state consistency, especially in the context of large-scale deployments.

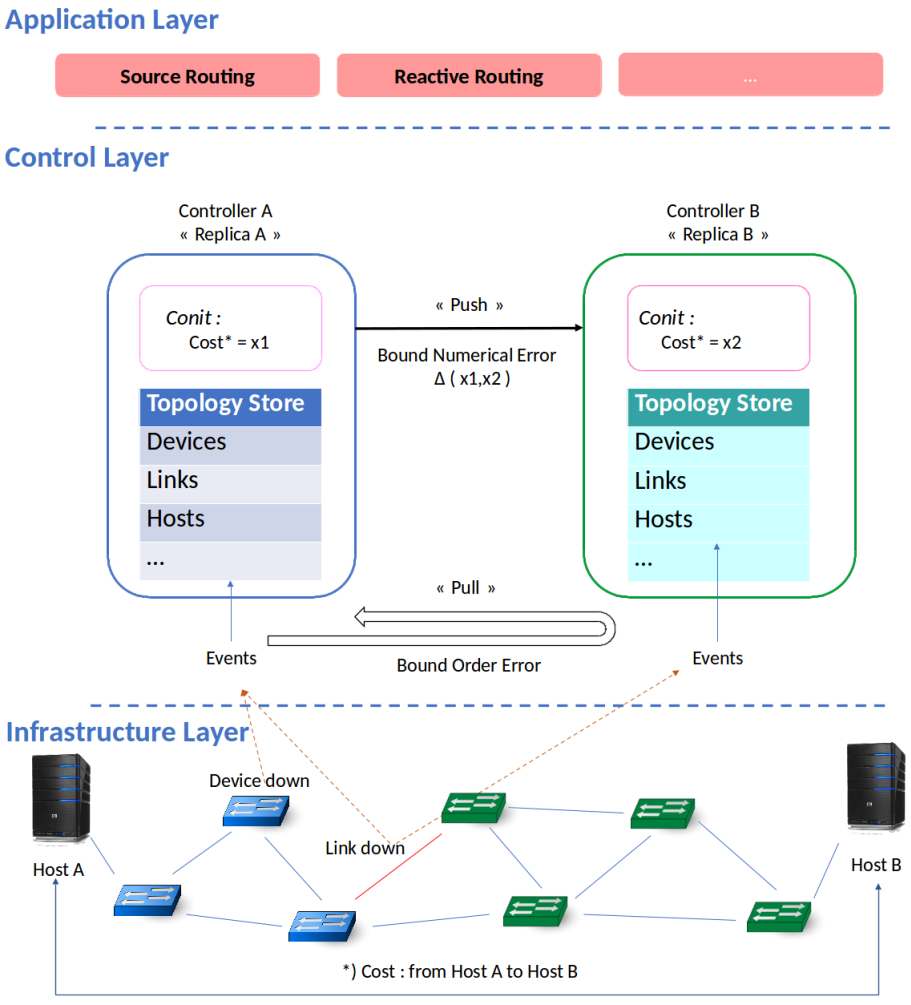


Figure 4.1: The proposed adaptive consistency strategy

4.5.2 Our consistency adaptation strategy for ONOS

We propose to maintain the strong consistency model implemented in ONOS for applications requiring strict consistency guarantees, but we suggest turning the eventual consistency model into an adaptive tunable model following the concept of continuous consistency in order to explore the availability, consistency and scalability benefits of such a model.

With this in mind, we adopt the following strategy when reviewing the eventual consistency model:

- We keep the current implementation of the optimistic replication technique used for replicating events and updates among controllers in the cluster.

- We bring significant changes to the anti-entropy protocol used for eventual consistency in ONOS: Instead of running the anti-entropy process for each controller replica periodically at fixed intervals (each 3-5 seconds) (*voluntary anti-entropy*) causing too much overhead and affecting system scalability and performance, we argue that the anti-entropy process should only be scheduled when the system consistency is at risk (*compulsory anti-entropy*). In other words, the choice of the anti-entropy reconciliation period for each controller replica (*per-replica consistency*) should be based on the correctness of the system with respect to the consistency requirements expressed by the applications. That way, at each controller replica and for each application state, the consistency level is dynamically adapted based on the computed values of the consistency metrics (see Equation (4.1)) capturing the application's consistency semantics with respect to the given thresholds set in advance by the application.

4.5.3 Our implementation approach

To implement our state consistency adaptation approach, we consider a replicated source routing SDN application running on top of a cluster of multiple ONOS controllers. The cost-based source routing application operates on a distributed topology graph for computing the shortest-path costs between source and destination hosts. Since the topology graph state is handled in an eventually-consistent manner, the application's state is also considered as eventually-consistent.

In our routing application f, the path between the source host A and the destination host B (see Figure 5.1) may be defined as a *conit*. Besides, we argue that an important consistency requirement for our control application is the result optimality of the instant path

computation cost (in terms of hop-count in our case) which is captured by the *Numerical Error* metric. The numerical error of our conit C can be defined as relative difference between the value of the "shortest-path" cost x_{local} as perceived by a local replica, and its "final "optimal" value $x_{optimal}$ at a replica that has reached some "final" consistent state. That error is continuously bounded at run-time using an application-defined threshold T(f) (in percentage) as follows:

$$\mathrm{N}umerical\mathrm{E}rror(\mathrm{C}_f) = (\frac{|x_{local} - x_{optimal}|}{x_{optimal}}) < \mathrm{T}(f) \tag{4.2}$$

Finally, it is worth noting that other consistency semantics for the source routing application might be expressed using *Staleness* and *Order Error*.

To implement our approach, we introduce some modifications to the ONOS Java source code. We start by developing our adaptive source routing application (similar to the `Intent Forwarding Application`) for computing the shortest-path cost between host A and host B. Running on each ONOS instance, our application gets information from the in-memory topology cache maintained by each ONOS instance (`DistributedTopologyStore`). Whenever an update occurs in the topology graph (e.g. links/devices failing or joining), our application detects that topology change and updates the "local" shortest-path cost between host A and B accordingly. We also modify the implementation of the `EventuallyConsistentMap` distributed primitive, especially for the eventually-consistent stores that feed the topology store (e.g. link and device stores). We indeed focus on the `BackgroundExecutor` service of the eventually-consistent maps, which runs the background anti-entropy tasks, and we propose a new implementation of the `Runnable` interface used for executing the scheduled anti-entropy thread.

In fact, instead of sending the anti-entropy advertisement messages periodically each 3-5 seconds between the controllers, as it is the case in ONOS, we propose to run, at each replica, a periodic check on the consistency of other replicas with respect to the application's shortest-path cost state, by computing at run-time the relative *Numerical Error* defined in Equation (4.2). In fact, in the event of a controller failure, the rest of the controllers in the cluster that detect that failure, keep a screen-shot of their own topology graph at the moment of the failure. During the periodic consistency check, they use that stored topology graph to estimate the inconsistency of the failed controller; which is equal to the relative difference between the "local" shortest-path cost (computed based on the current topology graph state) and the shortest-path cost as perceived by the controller

after recovery (computed based on the *stale* topology graph state being stored).

When the failed controller recovers, the rest of the controllers make an anti-entropy decision based on the checked numerical error. If the error exceeds an "alarming" consistency threshold set in-advance by the application, then, an anti-entropy process is launched to fix the failed controller state. That is achieved by synchronizing the controllers' eventually-consistent stores that feed the topology view. In the opposite case, the inconsistency is considered as tolerated by the application, and an anti-entropy session might be scheduled afterwards in case the controller state significantly drifts away.

4.6 Performance evaluation

4.6.1 Experimental setup

Our experiments are performed on an Ubuntu 16.04 LTS server using ONOS 1.13. We use the network emulator Mininet 2.2.1. which can create virtual switches,hosts and connect to the ONOS controllers. We also use an ONOS-provided script (*onos.py*) to start an emulated ONOS network on a single machine; including a logically-centralized ONOS cluster, a modeled control network and a data network. Wireshark is used as a sniffer to capture the inter-controller traffic which uses TCP port 9876.

To validate our proposed approach on ONOS, which we will refer to as ONOS-WAC (ONOS-With Adaptive Consistency), we have considered many test scenarios. In each scenario, we run a cluster of N ONOS controller instances, controlling a Mininet network topology of S switches (see Table 4.1).

Test scenarios	N Controllers	S Switches	F Controller Failure scenarios
n°1	3	16	2
n°2	5	36	3
n°3	7	64	4
n°4	9	100	4
n°5	10	121	5

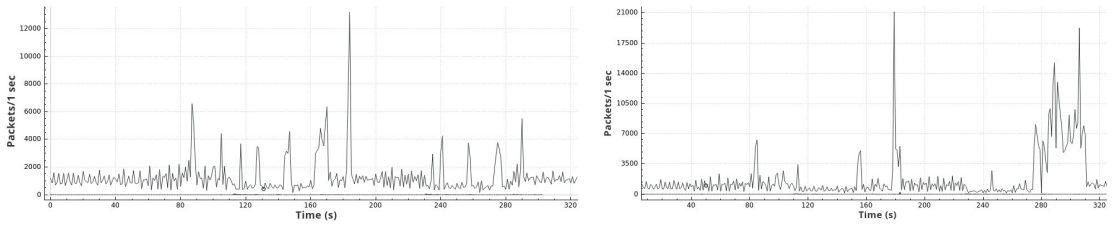
Table 4.1: Test scenarios

In order to create state inconsistencies among the controller instances in the cluster with respect to the shortest-path cost state of our source routing application, we create different controller failure scenarios F. Shortly after a controller failure (in F) in a specific scenario S_i, we consider changing the network topology by taking down network switches and links along the shortest-path (computed by the application) between source host A

and destination host B. That way, after recovery, the controller will have an inconsistent network topology view as compared to the rest of the controllers in the cluster. According to our proposed approach, that inconsistency in the network topology view affects the optimality of the shortest-path cost computation performed by the source routing application instance running on top of the recovered controller. The induced *Numerical Error* is likely to trigger a synchronization process achieved by anti-entropy tasks (see Section 4.5.3).

4.6.2 Results

In Scenario n°1 (Scenario with three controllers as shown in Table 4.1), we adopt the methodology described in 4.6.1. In Figure 4.2, we show the inter-controller traffic captured during the test scenario period in an ONOS cluster (Figure 4.2(a)) and in an ONOS-WAC cluster (Figure 4.2(b)). After running the Mininet topology according to Scenario n°1, the same event sequence is performed for both ONOS and ONOS-WAC clusters. For instance, the first traffic peak in both figures (at $t = 90s$) corresponds to a "pingall" Mininet CLI command executed for topology discovery. At $t = 150s$, we simulate a failure scenario by taking down one controller instance. That action is followed by other topology changes (e.g. links down) corresponding to the subsequent peaks in both figures. At $t = 180s$, we bring back the failed controller, resulting in a traffic peak that appears to be more significant in the case of ONOS-WAC. That increase in traffic is due to the anti-entropy process which has been triggered by an inconsistency (*Numerical Error*) value that exceeded the application threshold. Conversely, in the ONOS network, the anti-entropy traffic is generated periodically over the test period regardless of the observed inconsistencies. Likewise, at $t = 280s$, we repeat the same scenario following the same event sequence, but considering the failure of a different controller.



(a) ONOS (b) ONOS-WAC

Figure 4.2: Scenario n°1: Captured Inter-controller traffic (in packets per second) during the test scenario period (using Wireshark)

In the test scenario described above, the application inconsistency threshold was set to 0%, triggering the start of anti-entropy sessions for any observed inconsistencies in the considered application state. In the following test experiments, we repeat the same scenario (Scenario n°1), but we consider varying the source routing application's inconsistency threshold. As shown in Figure 4.3, ONOS shows an average inter-controller overhead equal to $315kbps$ regardless of the application inconsistency threshold. On the other hand, ONOS-WAC shows relatively less inter-controller overhead (due to low anti-entropy overhead). The latter is impacted by the application's consistency requirements. For example, in the case of strict consistency requirements (application threshold between 0% and 30%), inconsistencies occurring in the application state are more likely to trigger the anti-entropy reconciliation sessions causing much more overhead, when compared to the case of less strict consistency requirements (application threshold between 40% and 50%).

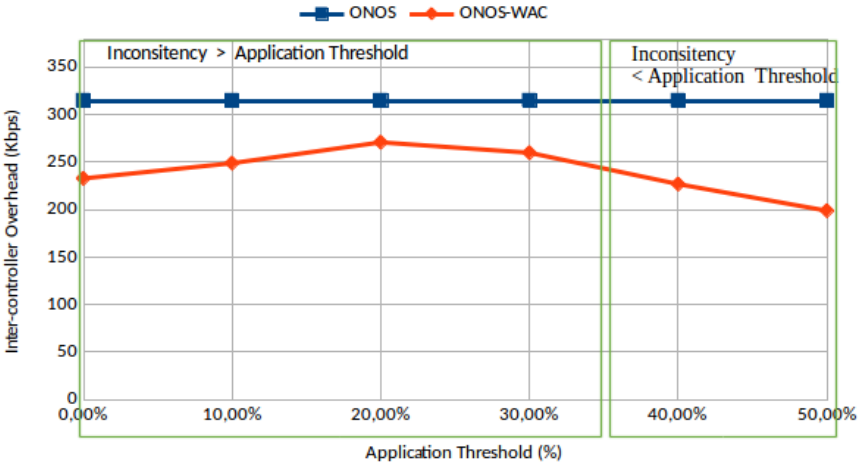


Figure 4.3: Scenario n°1: Inter-controller overhead in ONOS and ONOS-WAC according to the application threshold

In order to assess the gain in Anti-Entropy overhead of our adaptive consistency model implemented on ONOS, as compared to the eventual consistency model of ONOS, we consider estimating the rate ($R(S_i)$) of increase in anti-entropy overhead of ONOS with respect to ONOS-WAC (see Equation (4.3)) as a function of the number of controllers in the cluster (following Scenarios n°1, n°2, n°3, n°4, n°5) (see Figure 4.4).

$$R_i(S_i) = 1 - [\frac{A(S_i) - B(S_i)}{C(S_i) - B(S_i)}] \tag{4.3}$$

-$A(S_i)$: the inter-controller overhead generated by ONOS-WAC *after* the event sequence (4.6.2) following Scenario S_i.

-B(S_i): the inter-controller overhead generated by ONOS-WAC *before* the event sequence (the overall inter-controller traffic without the anti-entropy traffic).
-C(S_i): the inter-controller overhead generated by ONOS *after* the event sequence (4.6.2) following Scenario S_i.

As shown in Figure 4.4, the gain in anti-entropy overhead, when adopting ONOS-WAC, grows almost linearly with the number of controllers in the cluster. For example, in Scenario n°5 (corresponding to 10 controllers in the network cluster), the gain in anti-entropy overhead has reached 25%.

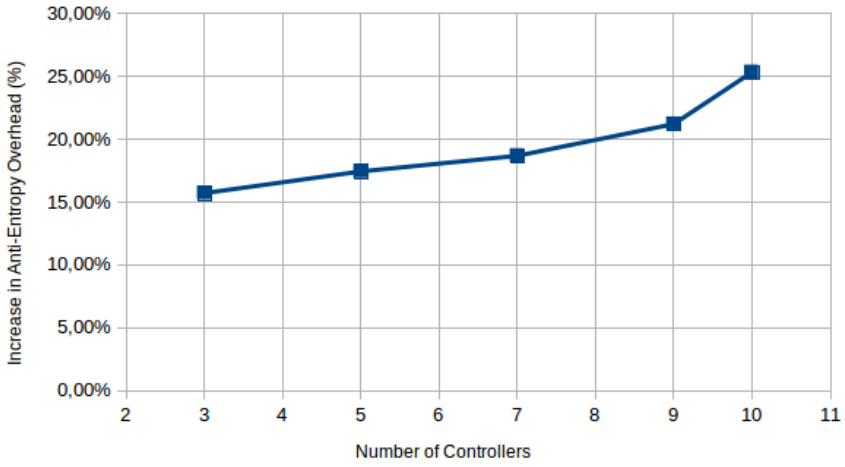


Figure 4.4: Gain in Anti-Entropy overhead of ONOS-WAC with respect to ONOS according to the number of controllers in the cluster

4.7 Conclusion

In this work, we investigated the use of adaptive consistency for the distributed ONOS controllers. Our approach was aimed at turning the eventual consistency model of ONOS into an adaptable multi-level consistency model following the concept of continuous consistency. The latter delivers the performance and availability benefits of an eventual consistency model, but has the additional advantage of controlling the state inconsistencies in an application-specific manner. Our consistency adaptation strategy was implemented for a source routing application on top of ONOS. Besides ensuring the application's state consistency requirements (specified in the given SLA), our results showed a substantial reduction in the Anti-Entropy reconciliation overhead, especially in the context of large-scale networks. As a future work, we consider extending our adaptive consistency approach to the optimistic replication technique used in ONOS's eventual consistency model by leveraging multiple replication degrees as well as the geo-placement of

the controller replicas.

Although the main focus of this work was placed at dynamically adjusting the consistency level of application states (which use controller states), we plan to extend our approach to the controller states (internal controller applications). Indeed, the long-term goal of this work is to design adaptively-consistent controllers that adjust both control and application plane consistency levels under changing network conditions.

Chapter 5

Adaptive and continuous consistency for distributed SDN controllers: Quorum-based replication

« Power is gained by sharing knowledge, not hoarding it »

Maria Khan

Contents

5.1 Introduction

Existing SDN controller platforms have been architected according to different SDN control plane designs with the aim of meeting specific requirements in terms of scalability, high-availability and performance. Consistency has also been regarded as an essential design principle for the distributed SDN controller platforms. The latter use conventional consistency models to manage the distributed state among the controllers in the cluster. As explained in Section 4.3.2 of the previous Chapter, the consistency models used in SDN can be categorized into *strong*, *eventual* and *weak* [38; 150; 159]. These static and standard consistency models [150; 159] have both advantages and drawbacks.

In large-scale SDNs, the *Strong Consistency* control model might be extremely expensive and costly to maintain for certain applications. Indeed, it requires important synchronization efforts among the controller replicas at the cost of causing serious network scalability and performance issues. By contrast, the *Eventual Consistency* control model implies less inter-controller communication overhead as it sacrifices the strict consistency guarantees for higher availability and improved performance. In practice, many scalable control applications running in modern distributed storage systems like Apache's Cassandra [111] and Amazon's Dynamo [113] opt for eventual consistency to provide such requirements on a large scale. However, these applications might suffer from the associated relaxed (weak) consistency guarantees that may temporarily allow for too much inconsistency.

Recent research works in the area of distributed SDN control have explored the concepts of *Adaptive Consistency* control for various applications [115; 116; 160–162]. Such categories of consistency models follow different adaptation strategies that mainly focus on dynamically adjusting the levels of consistency at run-time under various network conditions in order to meet the application-defined consistency and performance needs.

Unlike strong and eventual consistency options, adaptive consistency control models leverage the broad space of intermediate consistency degrees between these two extremes. They, indeed, use time-varying consistency levels to support balanced real-time trade-offs between the desired consistency and performance requirements which can be specified in the application-defined Service-Level Agreements (SLAs) [163].

In this chapter, we propose an adaptive consistency model (based on eventual consistency) for the ONOS controller applications that are deployed in large-scale networks.

Most notably, we target the class of applications that tolerate relaxed forms and degrees of eventual multi-consistency for the sake of scalability and performance, but yet can benefit from improved consistency features.

More specifically, our state consistency adaptation approach was implemented for a CDN-like application we developed on top of the distributed open-source ONOS controllers. It mainly consists in changing ONOS's eventual consistency model to an adaptive consistency model by turning ONOS's optimistic replication technique into a more scalable replication strategy following Quorum-replicated consistency models. Indeed, the adaptive consistency strategy we propose in this chapter focuses on improving ONOS's replication mechanism: It uses the *read and write Quorum parameters* as adjustable *control knobs* for a fine-grained consistency tuning, rather than relying on Anti-Entropy reconciliation mechanisms (see previous Chapter 3) [162]. The main objective is to find at runtime optimal Quorum replication configurations that achieve, under changing network conditions and varying application workloads, balanced trade-offs between the application's continuous performance (*latency*) and consistency (*staleness*) requirements. These real-time trade-offs should provide minimal application inter-controller overhead while satisfying the application-defined thresholds specified in the given application SLA.

The rest of this chapter is organized as follows: In Section 5.2, we conduct a background review of eventual consistency models in modern distributed data-store systems. Inspired by the modern consistency techniques used in these scalable data-stores, we present, in Section 5.3, our adaptive and continuous Quorum-based consistency model for the distributed ONOS controllers in large-scale deployments. In Section 5.4, we describe our methodology for implementing the proposed consistency strategy on a CDN-like application that we designed on top of the ONOS controllers. Finally, Section 5.5 elaborates on the test scenarios we developed to evaluate our proposal and discusses the experimental results.

5.2 Background on eventual consistency in distributed data-stores

5.2.1 Consistency and performance Metrics:

Guaranteeing the consistency of replicated data in distributed database systems has always been a challenging task. Today's fundamental consistency models (e.g. strong con-

sistency, sequential consistency, causal consistency, eventual consistency) ensure different discrete levels and degrees of consistency guarantees. For instance, the strong consistency model offers up-to-date data, but at the cost of high latency and low throughput. As a result, weaker forms of consistency (in the consistency spectrum)-most notably the popular notion of eventual consistency- have been widely adopted in modern distributed data-stores which need to be highly-available, fast and scalable [111; 113]. However, despite being regularly acceptable and desirable in practice for the latency and throughput benefits they offer, eventual consistency models provide no bounds on the inconsistency of data they return. Another major limitation of these models is that the trade-offs they make among consistency and performance (latency) are difficult to evaluate. In fact, measuring the concrete consistency guarantees of eventually-consistent distributed stores remains challenging.

Yu and Vahdat proposed the TACT framework [147] which fills in (captures) the consistency spectrum/space by providing a continuous conit-based and multi-dimensional consistency model. The latter can be leveraged by replicated Internet services to dynamically choose their own tunable and fine-grained consistency-performance and consistency-availability trade-offs based on client, service and network characteristics. In TACT, the authors quantify consistency by bounding the amount of inconsistency/divergence of the replicated data items in an application-specific-manner using three application-independent metrics: *Numerical error*, *Order error* and *Staleness*. Besides, Bailis *et al.* [164; 165] presented an approach based on a set of probabilistic models to predict the expected consistency guarantees as measured by the *staleness* of reads observed by client applications in eventually-consistent Dynamo-style partial quorum systems. The authors introduced the WARS Probabilistically Bounded Staleness (PBS) model which provides bounds on the expected staleness in terms of both versions (using the *k-staleness* metric) and wall clock time (using the *t-visibility* metric). Another interesting work found in [166] proposes an automated self-adaptive consistency approach called Harmony which embraces an intelligent estimation of the *stale read rate* metric in Cloud storage systems, allowing to automatically adjust the consistency level at run-time according to application needs. That was achieved by elastically scaling up or down the number of replicas involved in read operations to preserve a low tolerable fraction of stale reads. When compared to the static eventual consistency approach in Cassandra, Harmony significantly enhances the consistency guarantees by reducing the rate of stale reads while adding only minimal la-

tency. Besides, when compared to the strong consistency model in Cassandra, Harmony improves the performance of the system by increasing the overall throughput while maintaining the desired consistency requirements of the applications.

5.2.2 Adaptive consistency control

Modern distributed database systems supporting standard eventual consistency models suffer from the inevitable trade-offs between consistency, availability and request latency. To overcome this major limitation, these storage systems have introduced the concept of adaptive consistency in order to find appropriate consistency options depending on application requirements and system conditions. In literature, adaptive consistency techniques have been broadly classified into two categories: *user-defined* and *system-defined* [156].

In contrast to user-defined adaptive consistency methods where data and operations need to be mapped in advance to the desired consistency levels (using some parameters), system-defined adaptive consistency methods take into account the fact that user and system behaviors might change dynamically over time making the consistency decision-making process challenging and tricky for application developers. That is why, system-defined techniques usually rely on system intelligence and adaptability to automatically provide fine-grained control over the consistency guarantees at run-time. Accordingly, many factors can be considered to dynamically estimate and predict the appropriate system consistency, including data access patterns, system load, but also the application's consistency SLAs as discussed in Section 5.2.1. One famous form of system-defined adaptive consistency is the continuous consistency model used in TACT [147].

Additionally, it is worth mentioning that designing system-defined adaptive consistency (falling within the scope of this work) requires careful considerations of the appropriate consistency adaptation strategy. In particular, existing adaptive mechanisms use different control knobs to be configured for consistency tuning such as the *consistency level*, the *artificial read delay*, the *replication factor* and the *read repair chance* [167].

5.2.3 Existing modern tunable consistency systems

Popular distributed (Cloud) storage systems, most notably Apache's Cassandra [111], Amazon's Dynamo [113], Riak [112], and Voldemort [168] opt "by default" for eventual consistency guarantees in exchange for extremely high availability. However, these systems

attempt to provide the applications with some control over the consistency and performance trade-offs via built-in settings and features. They indeed extend the concept of eventual consistency by offering tunable consistency levels for application developers and users based on Dynamo-style quorum replication policies.

In Cassandra, the consistency level specifies the size of a quorum for reads and writes, which is the appropriate number of replicas in the cluster that must acknowledge a read or write operation before considering the operation successful. The native and well-known consistency levels/options in Cassandra are three: ONE replica, a QUORUM of replicas, and ALL of the replicas. Accordingly, different choices of read and write consistency levels (quorums) ensure different consistency guarantees. For instance, to achieve the highest strong consistency, different quorum configurations may be selected, but they must satisfy the overlapping quorum property between read and write replica sets (*strict quorums*). On the other hand, to provide acceptable consistency with improved availability (minimum latency), it is desirable to use weaker forms of consistency such as the default eventual consistency option. Such weak consistency levels can be achieved through different quorum configurations that do not satisfy the overlapping quorum intersection property (*partial (non-strict) quorums*).

As a result, modern storage systems like Cassandra can be classified in the category of *user-defined* adaptive consistency as discussed in Section 5.2.2, given that they offer multiple consistency options. However, although these systems offer adaptive consistency on top of tunable consistency models that are aimed at creating balanced trade-offs between consistency and performance, it is usually difficult for application developers to decide in advance about the required consistency options for a particular request [156].

5.3 The proposed adaptive Quorum-inspired consistency for ONOS

In this chapter, we propose a novel quorum-based and *system-defined* adaptive consistency model for the distributed ONOS controllers. Our approach was partly inspired by the quorum-replicated consistency techniques used by the modern data-store systems discussed in Section 5.2.3.

The ONOS approach to state consistency in the latest releases was described in detail in Section 4.4. It mainly relies on two consistency schemes that provide two levels of consistency: strong consistency and eventual consistency. While the strong consistency

model is leveraged by ONOS controller applications that require strong consistency and correctness guarantees, the eventual consistency model is intended for ONOS controller applications that favor scalability and performance over strict consistency.

In this chapter, we target the second class of scalable control applications that have optimistic relaxed consistency needs, but that can benefit from improved performance and automated SLA-aware consistency tuning at scale, as offered by our adaptive continuous consistency strategy.

5.3.1 A continuous consistency model for ONOS

As explained in Section 4.5.1, the applications on top of the ONOS controllers can benefit from the continuous consistency model introduced with TACT [147], by continuously and dynamically specifying their consistency requirements using three application-independent metrics to capture the consistency spectrum and bound consistency: *Numerical Error*, *Order Error*, and *Staleness*.

In this work, we focus on the type of applications whose application-specific consistency semantics can be expressed using the staleness of data as a metric to quantify the level of consistency. With such SLA-style consistency metrics, these applications can avoid the challenges related to potentially *unbounded staleness* as in eventual consistency.

Generally speaking, the staleness metric measures data freshness in distributed data-stores ; it describes how far a given replica lags behind in data operations in comparison to up-to-date replicas, either expressed in terms of time or versions. In the literature, the notion of data staleness falls indeed into two common categories: staleness in time (*time-based staleness*) [147; 164], and staleness in data version (*version-based staleness*) [164].

In TACT [147], the staleness metric places a real-time bound on the amount of time before a replica is guaranteed to see a write accepted by a remote replica. In [165], the authors propose a probabilistic consistency framework that provides expected bounds on data staleness with respect to both versions and wall clock time in eventually-consistent data-stores. In their model, time-based staleness (*t_visibility*) describes the probability that a read operation, starting t seconds after a write commits, will observe the latest value of a data item [164]. On the other hand, version-based staleness (*k_staleness*) describes by how many versions the value returned by a read lags behind the most recent write. It is measured as the probability of returning a value within a bounded number k of versions.

In this work, we adopt the data staleness metric from a strictly time-based perspective.

In our SDN controller application, we characterize staleness by an "Age of Information (AoI)" timeliness metric [169] that describes the difference between the query time of a data item and the last update time on that item. If the last successfully received update was generated at time $u(t)$ then its age at time t is $\Delta(t) = t - u(t)$.

Applications on top of the distributed ONOS controllers could also benefit from SLA-style performance requirements, to continuously specify their own fine-grained trade-offs between performance and consistency. In our work, we consider the read request latency/delay as our performance metric. In addition, we evaluate the inter-controller communication overhead for our ONOS application.

More detailed information about the way we measure our continuous consistency and performance metrics when implementing our state consistency approach for the new controller application we developed on top of ONOS is provided in Section 5.4.

5.3.2 Our Quorum-inspired consistency adaptation strategy for ONOS

5.3.2.1 Quorum consistency

As explained in Section 5.2.3, quorum-replicated systems ensure different consistency guarantees:

- Strong consistency can be guaranteed with *strict quorums* that satisfy the condition that sets of replicas written to and read from need to overlap:

  R + W > N, given N replicas and read and write quorum sizes R and W.

- Eventual consistency occurs with *partial quorums* that fulfill the condition that sets of replicas written to and read from need not overlap:

  R + W ≤ N, given N replicas and read and write quorum sizes R and W.

Traditionally, partial quorum-replicated systems ensure eventually-consistent guarantees, with no limit to the inconsistency of the data returned, which may not be acceptable for certain applications. However, with the PBS model [164], it has been possible for applications to analyze the staleness of the data returned, quantify the consistency level, and therefore measure the latency-consistency trade-offs for partial quorum systems.

Building on these concepts, we propose an adaptive consistency model for the ONOS applications using partial quorums, given the latency and scalability benefits they offer. To measure the consistency semantics (e.g the staleness metric) of these applications and

thus meet their consistency requirements (e.g bounded staleness), we leverage the continuous consistency model discussed in Section 5.3.1.

Furthermore, using eventually-consistent partial quorums, it is possible to configure the size of read and write quorums, denoted respectively as R and W such that $R + W \leq N$, to ensure various consistency levels (e.g degrees of staleness). These multiple quorum configurations allow the applications to achieve different consistency-latency trade-offs.

5.3.2.2 Adaptive architecture

In this work, we propose to turn the eventual consistency model into an adaptive and continuous tunable consistency model using partial quorums. The proposed model uses the quorum replication parameters as the control knob, allowing for an adaptive fine-grained tuning and control over the consistency-performance trade-offs. In the following, we describe the main architecture components of our adaptive consistency model.

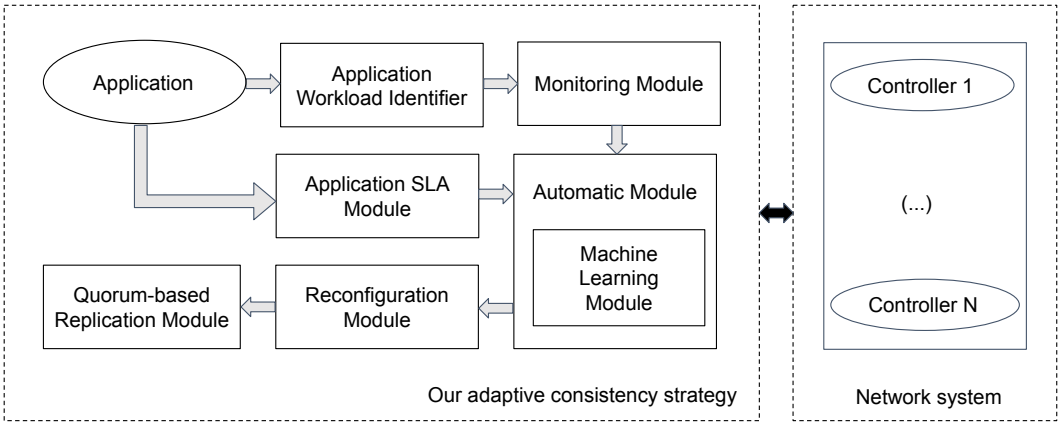


Figure 5.1: Architectural overview of our adaptive Quorum-based consistency strategy

- `Application SLA Module`

  This module offers the possibility for applications on top of ONOS to express their high-level SLA-style consistency and performance requirements such as the staleness and latency guarantees. Accordingly, for a given ONOS application that we develop on top of ONOS, our consistency model continuously measures the real-time metrics involved in quantifying the consistency-latency trade-off. The Automatic Module translates these requirements into appropriate time-varying partial quorum replication configurations (R, W, N) that achieve balanced trade-offs between the specified guarantees.

- `Workload Identifier Module`

  This module identifies the application's workload characteristics. It considers three different workloads that are representative of three different application scenarios [170]. The first workload describes a read-intensive scenario where 70% of operations are read accesses. The second workload has a balanced ratio between read and write operations. Finally, the third workload represents a write-dominated scenario in which 70% of the generated operations are write accesses.

- `Monitoring Module`

  This module is responsible for periodically gathering the application traffic information in a non-intrusive manner. More specifically, the module measures the system KPIs, for different read/write Quorum configurations and according to different application workload scenarios. These KPIs include the performance (e.g. response time) and consistency (e.g staleness) metrics related to client requests for specific application contents, as well as the generated (read and write) application overhead.

- `Automatic Module`

  The choice of the size of read and write Quorums used when executing read and write operations is a fundamental factor that affects the application's consistency guarantees but also the performance provided by the network system. However, selecting the right Quorum configuration is a non-trivial task. Our Automatic Module attempts to find the optimal configuration of the read and write Quorum sizes while taking into account the current application workload conditions. The main objective is to minimize the overhead generated by the application (scalability challenge), and potentially other network and application metrics, while satisfying the consistency and performance SLAs specified by the application.

  This module is fed with a set of application workload characteristics which are gathered by the Workload Identifier Module. In our case, it relies on a Machine Learning Module to predict the expected optimal configuration of the Quorum parameters for the determined workload, and then feed them to the Reconfiguration Module.

- `Machine Learning Module`

This module uses Reinforcement Learning (RL); an area of Machine Learning (ML) inspired by behaviorist psychology, and concerned with how software agents take actions in an environment so as to maximize some notion of cumulative reward.

More specifically, we use a Q-Learning (QL) model-free RL technique [171]. The main idea is to train an *agent* which interacts with its *environment* by performing *actions* that change the environment, going from one *state* to another. These actions result in a *reward* received by the agent as an evaluation of its actions (reinforcement) (see Figure 5.2). In this way, the agent learns some rules and develops a strategy, referred to as a *policy*, for choosing actions that maximize its reward.

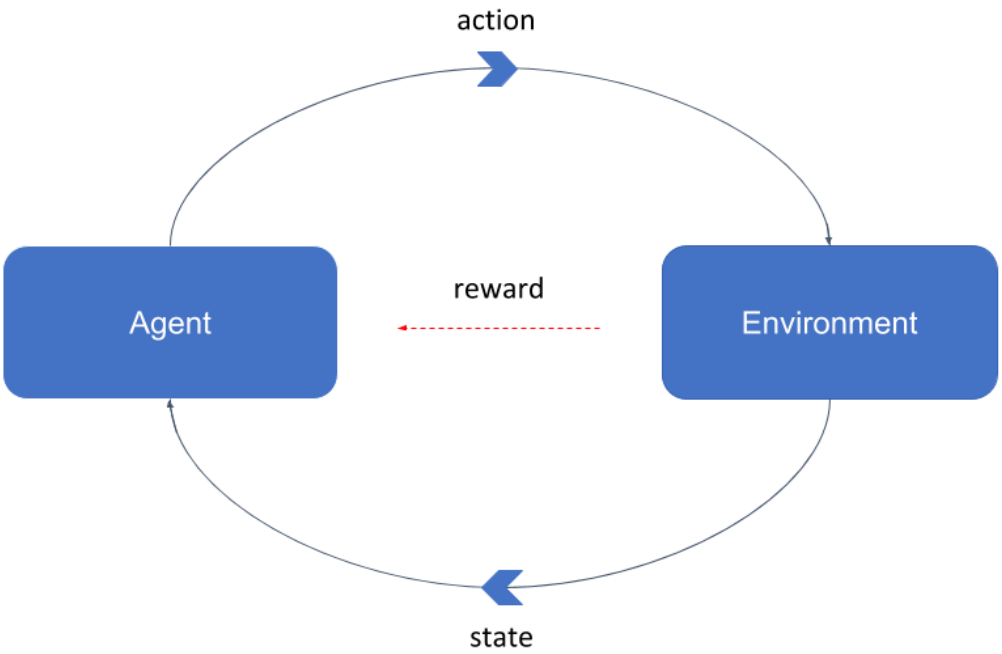


Figure 5.2: Reinforcement Learning (RL) architecture

The Q-Learning update rule makes use of the so-called Action-Value function, commonly known as the *Q-function*, representing the “quality” of a certain action in given state. It takes as inputs the "state" and the "action", and returns the expected future reward of that action at that state. In other words, the Q-function maps state-action pairs to the highest combination of immediate reward with all (discounted) future rewards that might be collected by later actions. The expression of the Q-function is given by the following equation (from Wikipedia):

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}} \tag{5.1}$$

The above function is used for updating the *Q-table* with Q-values at each *episode*. A Q-value is assigned to a possible pair of a state *s* and an action *a*. It is also worth noting that the learning agent should achieve a good strategy for balancing the exploration/exploitation trade-off inherent to reinforcement learning. That dilemma consists in choosing the appropriate action at a given episode: either to *exploit* the environment by selecting the best action at that specific time step given the current knowledge provided by the Q-table, or to *explore* the environment by choosing random actions. After each action, the agent is expected to update the Q-table.

In our case, the Q-learning agent attempts to learn online the best combination of the read and write Quorum size parameters, respectively R and W, in an environment built using our Monitoring Module. An action is defined as an update of R and W to certain possible values, thereby transforming the environment to a state defined by a new estimation of the network (inter-controller overhead) and application (latency and staleness) metrics. In our case, one of four possible actions is allowed at each episode (i.e. incrementing R by one, or decrementing R by one, or incrementing W by one, or decrementing W by one).

The reward received by the agent for updating the Quorum parameter values is a function of the read and write overheads to be minimized. The agent should also learn how to respect some constraints in order to satisfy the application requirements specified in the given SLA.

- `Reconfiguration Module`

  This module is able to dynamically adjust the values of the read and write Quorum sizes, denoted respectively as R and W. It basically relies on the Automatic Module to optimize the configuration of the quorum system. The reconfiguration process launched by this module is a non-blocking process that is able re-configure at runtime the Quorum settings selected by the Automatic Module.

A more detailed description of the way the re-configuration module sets the values of R and W at run-time is provided in Section 5.5.1.1.

- `Quorum-based Replication Module`

  Given the quorum replication settings, we adopt the following consistency strategy when reviewing the two main techniques employed by ONOS's eventual consistency model:

  – Replication Strategy: As explained in Section 4.4.2.1, ONOS's eventually-consistent stores employ an optimistic replication technique that consists in replicating local updates across all the controllers in the cluster, hence causing control plane overhead. Instead, we put forward a partial quorum replication strategy, where an eventually-consistent data store writes a data item on the local replica first and then sends it potentially to another set of replicas, obeying the given write quorum parameter (W). On the other hand, to serve read requests, we propose that the eventually-consistent data store fetches the data from the local replica first and then potentially from another set of replicas, depending on the given read quorum (R). This is in contrast to ONOS's strategy where the read requests are always processed by the local replica.

  – Anti-Entropy reconciliation mechanism: As explained in Section 4.4.2.2, ONOS's optimistic replication strategy is complemented by a background Anti-Entropy mechanism. That periodic reconciliation approach ensures that the system state across all replicas eventually converges to the consistent state. This is particularly useful in repairing out-of-date replicas and fixing state inconsistencies potentially resulting from controller failures. In this work, we assume that the system is reliable as we experiment with well-functioning emulated network topologies in the absence of controller failure scenarios. Therefore, we propose to deactivate the Anti-Entropy protocol, and focus on ONOS's replication strategy. However, it is worth noting that using additional Anti-Entropy (*expanding partial quorums* [164]) might be useful in particular cases where state inconsistencies become high and can no longer tolerated by the concerned applications.

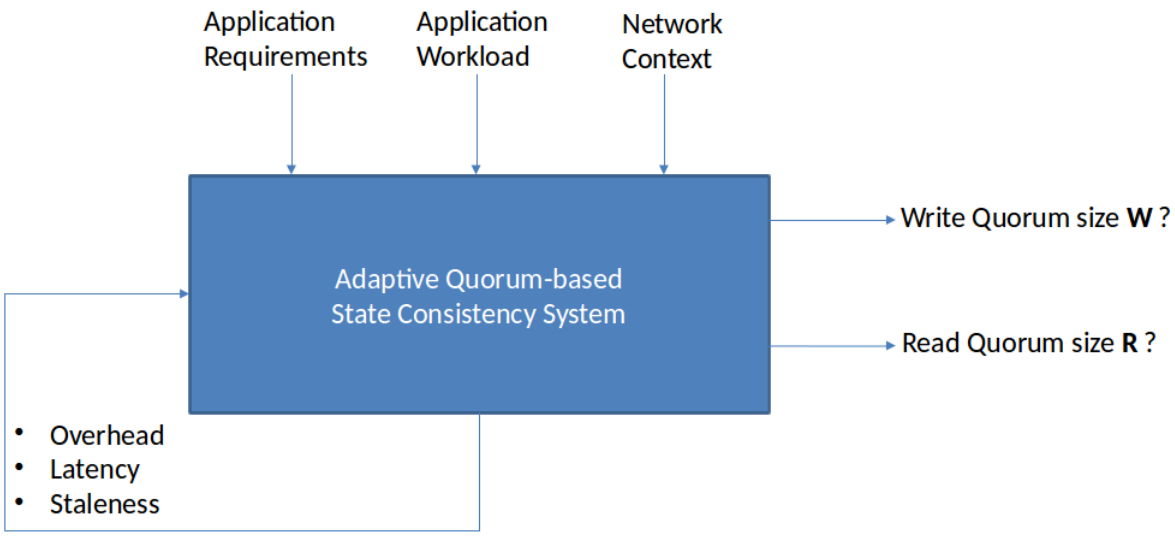


Figure 5.3: The proposed adaptive consistency system

5.4 Implementation approach on ONOS

In this section, we describe the implementation details for realizing the proposed consistency strategy explained in the previous section (see Figure 5.3) on the Java-based open source ONOS controller platform.

5.4.1 Design of a CDN-like application

To validate our adaptive consistency approach, we developed a new distributed Content Delivery Network (CDN) application running on top of a cluster of multiple ONOS controllers in an emulated SDN network. Our application replicates contents from content providers to hosting cache servers that are located in multiple geographical locations (ONOS domains) close to users. These cache servers are Mininet hosts that run simple HTTP web servers. We propose to consider a single origin server located in each ONOS domain. The main idea is to serve client hosts with the most up-to-date copies of the requested content and within a reasonable time (low latency).

More specifically, our application consists of two main components: An `ApplicationManager` and a `DistributedApplicationStore`. The Application Manager component which is an implementation of the Application Service is responsible for creating a virtual network of cache servers and providing mesh connectivity between these server hosts. On the other hand, the Distributed Application Store which is an implementation

of the Application Store performs the task of persisting and synchronizing the information received by the application manager. It is backed by an eventually consistent map with eventual consistency guarantees for storing the service's application state, namely the list of origin servers in the network and their respective set of generated contents:

```
EventuallyConsistentMap < OriginServerID, Set<Content> >
```

Each content that is created on the origin server, and then eventually propagated to cache servers has four properties; a `ContentName`, an identifier `ID`, a real time-based `CreationTime`, a `LogicalTimestamp`, and a `Version`.

Each controller replica that is responsible for a given ONOS domain operates on a local view of the eventually consistent map. That view consists of the local origin server from the same ONOS domain with its generated set of contents, and other potential origin server hosts located in different ONOS domains in the network with their respective set of contents, as seen by the local replica after application state synchronization.

Besides, we design a cached map that is local to each controller application instance and that represents the contents cached in the local CDN server within the same ONOS domain. The local cached map is closely linked to the local view of the eventually consistent map, and it reflects the contents stored in the local CDN server. The latter performs the functions of an origin server and at the same time a cache server. It contains indeed the contents created locally (origin server), and potentially other contents that are replicated from other origin servers (cache server).

More specifically, on a local controller replica, updates to the eventually consistent state map (e.g PUT) might trigger specific actions to feed the local CDN server and consequently update the local cached map. If the update to the content is associated in the map with the local origin server, that means that the updated content has already been generated on that origin server. On the other hand, if the update to the content is associated in the map with another origin server from another ONOS domain, our application checks the relevance of that content. In case the content is important to our application, then the update to the content gets automatically pulled from the origin server to the local CDN server (cache server) and gets cached in the local cached map.

```
CachedMap < ContentName, Set<Content> >
```

5.4.2 State synchronization and content distribution

The custom eventually consistent map we use for the synchronization of our CDN application state is based on our own implementation of the `EventuallyConsistentMap<K,V>` distributed primitive. Indeed, the new implementation we propose for the eventual consistency map abstraction models the quorum-inspired consistency discussed in 5.3.2.1.

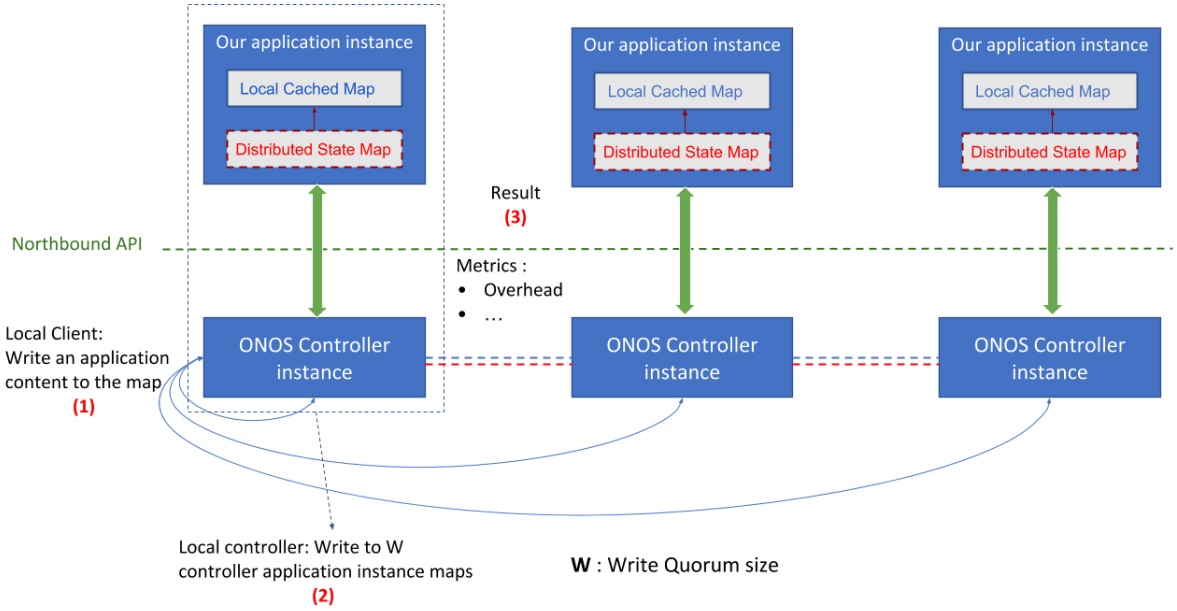


Figure 5.4: Quorum-inspired Write operations in our CDN-like application

In particular, it takes into account the size of the write quorum parameter (W) when replicating the updates related to our application's eventually consistent map among the controllers (see Figure 5.4). On each local replica, updates to the local map are queued in time to different `EventAccumulators` allocated for different controller peers. The latter are selected randomly, and their number depends on the write quorum size W. Whenever an event accumulator is triggered to process the previously accumulated events and propagate them to the associated peer, that peer is removed from the list of quorum peers. New updates will immediately trigger the creation of a new accumulator associated with a new randomly selected peer that is is added to the list of quorum peers. That accumulator will collect the updates together with the other event accumulators associated with the rest of the quorum peers. That way, we guarantee that updates to the eventually consistent map on a local replica are replicated at run-time to exactly W replicas, including the local replica.

As explained in Section 5.4.1, such updates to the eventually consistent map on a lo-

cal controller replica trigger specific actions that might feed the local CDN server with new contents (content distribution) and thus update the local cached map for our CDN application.

5.4.3 Content delivery to customers

During a read operation performed by a client, our controller application instance running on the local controller replica within the same ONOS domain as that client, receives the read request to be fulfilled following Quorum-inspired read consistency protocols (see Figure 5.5).

More specifically, if the read consistency level is higher than ONE (read quorum size R greater than 1), then the local controller node which serves in our case as the coordinator node, sends the read request to the remaining randomly-selected controller replicas forming the read Quorum. The size R of the read Quorum including the local controller replica is set in advance by the read consistency level.

We use ONOS's `ClusterCommunicationService` to assist communications between the local controller node and the rest of the controller cluster nodes in the read Quorum. More specifically, the local controller node sends the read request message with a particular subject to each of the concerned controller nodes using the `sendAndReceive` method of the cluster communication service. It expects a future reply message from each of the involved controllers that have already subscribed to the same message subject.

That said, to serve the client's read request for a specific content (`ContentName`), each controller node that has subscribed to the specified message subject receives the read request and uses the application's handler function for processing the incoming message. Accordingly, the application instance on each controller replica of the read Quorum (including the local replica) consults the local cached map. As explained in Section 5.4.1, the cached map represents the list of contents (created by different origin servers) being observed in the local view of the eventually consistent map, and then pulled to be cached in the local CDN server. Using that map, each application instance compares the cached versions of the requested content (`ContentName`) based on their `LogicalTimestamp` properties in order to determine the freshest version of the content. Then, it produces a reply containing the selected `Content` with its four properties discussed in 5.4.1, and more importantly the IP address of the local cache server that has just delivered the requested content.

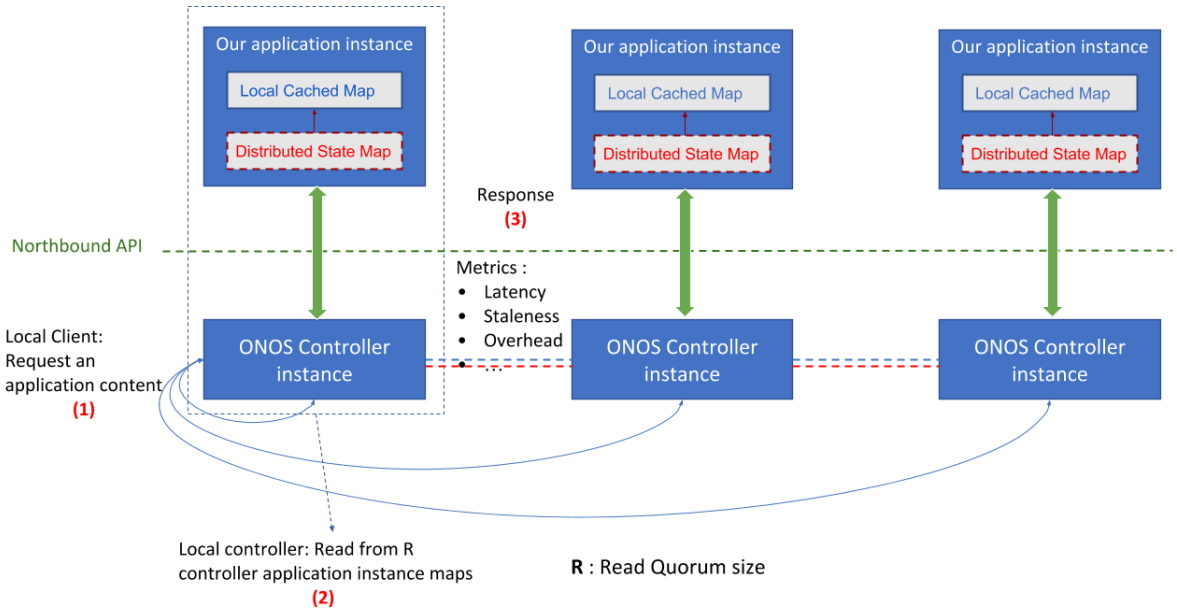


Figure 5.5: Quorum-inspired Read operations in our CDN-like application

The local controller replica playing the role of the coordinator, waits for the read Quorum of replicas to respond. Then, it merges the R responses (including the response produced on the local replica) to figure out the location of the freshest version of the requested content among the concerned CDN servers (equal to R in our scenario). Finally, it sends back the final response to the client and makes sure a host-to-host connectivity intent is added between the client host and the determined cache server host, using the `ONOS Intent Framework`. Based on that response, the client which has issued a HTTP request specifying the URL of the requested content, is redirected, using our CDN-like strategy (described above) and a DNS resolution service, to the selected cache server in order to retrieve the specified version of the content.

After each client request, our application collects the continuous consistency and performance metrics related to that request:

- Performance metrics:
  - Network-related metrics:

    We consider the application inter-controller overhead as a network performance metric. We first capture all inter-controller traffic using TCP port 9876. Then, we filter the captured traffic based on different conditions in order to evaluate the application's inter-controller overhead.

    Our goal is to minimize the application overhead due to write and read oper-

ations, depending on the given application SLA, the application workload and the network context.

$$ApplicationOverhead = WriteOverhead + ReadOverhead \tag{5.2}$$

– Client-centric metrics:

We also consider the response time to a client request as a performance metric. As defined by our application, the response time consists of the delay to fetch the appropriate version of the requested content from the local cached maps of the application instances running on the R controller replicas of the read Quorum (`Latency1`), and the delay to retrieve the specified version of the content from the selected cache server host (`Latency2`).

$$ResponseTime = Latency1 + Latency2 \tag{5.3}$$

- Consistency metrics:

  As explained in Section 5.3.1, we consider the application-specific staleness metric from a strictly time-based perspective: It describes the age of the information in terms of wall-clock time. Accordingly, the staleness of the application content being returned by a read operation at time t is measured as follows:

$$Staleness(Content) = QueryTime - CreationTime(Content) \tag{5.4}$$

  Besides, we set the staleness ranges used in the consistency SLA based on the application content refresh rate.

5.5 Performance evaluation

5.5.1 Experimental setup

Our experiments are performed on an Ubuntu 18.04 LTS server using ONOS 1.13. We also use Mininet 2.2.1 and an ONOS-provided script (*onos.py*) to start an emulated ONOS network on a single development machine; including a logically-centralized ONOS cluster, a modeled control network and a data network. Wireshark is used as a sniffer to capture the inter-controller traffic which uses TCP port 9876.

5.5.1.1 TCL-Expect scripts

In this section, we test our proposed adaptive consistency approach explained in Sections 5.3 and 5.4 which we will subsequently refer to as ONOS-WAQIC (ONOS-With Adaptive Quorum-Inspired Consistency) for brevity.

To that end, we write two `Expect` Tcl-based scripts (`main.exp` and `onos.exp`). In each script, we specify a set of required steps to follow to automate the tasks for our test scenarios on ONOS-WAQIC as summarized below:

1. First, we run our startup `Expect` script (`main.exp`). With Mininet and *onos.py*, we start up an ONOS cluster and a modeled data network for the specified topology. The selected number N of the ONOS controllers that will be forming the ONOS cluster is passed as an argument to the executed script.

2. Then, we run the Mininet CLI built-in `pingall` command to discover the network topology. We also launch a spawned process to install and activate the CDN-like application we developed on ONOS-WAQIC. To force device/switch mastership rebalancing, we connect to one of the running ONOS controller instances, and launch the ONOS CLI `balance-masters` command.

3. First, we parse the output of the `dump` Mininet command using regular expressions in Tcl in order to build a key-value array mapping the IP addresses of hosts to their Mininet names (*array1)*. Then, in the main Expect script, we launch N spawned processes that connect to the N running ONOS controller instances using the same `Expect` script (`onos.exp`) we developed, but run with different arguments (controller IP address, content name, maximum number of content versions). In the `onos.exp` script, we analyze the output of the `masters` ONOS CLI command to construct an array mapping each controller IP with the set of associated switches (MAC IDs) (*array2*). In addition, using the output of the Mininet CLI `hosts` command, we construct two additional arrays: the first array associates each host MAC ID with its IP address (*array3*), and the second array associates each host MAC ID with the switch ID to which it is connected (*array4*).

4. It is worth noting that our `onos.exp` script starts by running two ONOS CLI commands (`set-read` and `set-write`). We created these commands to set the read

and write Quorum sizes R and W to the values specified by the consistency level for a given ONOS controller instance. These values are passed as command arguments.

5. Using *array2*, *array3* and *array4*, each of the N currently spawned processes running the `onos.exp` script for a specific ONOS instance builds another Tcl array (*array5*) that identifies the list of hosts (MAC addresses) associated with each ONOS controller instance (controller IP address) in the network. Based on that array, our script randomly selects, for the specified ONOS controller instance, a list of hosts that will serve as origin cache servers and a list of hosts that will serve as clients in the concerned ONOS controller domain. The number of selected cache and client hosts depends on the application scenario/workload (see Section 5.3.2.2). Each ONOS process communicates the MAC and IP addresses of the origin server to the local application instance using our ONOS CLI `set-cache` command. Our script also runs the ONOS CLI `add-host` command which we created to add the cache server hosts to our application's `EventuallyConsistentMap` (discussed in Section 5.4.1).

   Besides, information about these cache server hosts is sent (using "puts") to the running `main.exp` script process. The latter identifies the Mininet names of these hosts using *array1* and connects to the Mininet CLI command in order to install a `SimpleHTTPServer` on each of the cache server hosts.

6. At this stage, we make sure that our main process (running *main.exp*) and the N spawned processes (running `onos.exp` with different arguments) are synchronized. Afterwards, each of the N spawned processes connecting to an ONOS controller instance starts adding (then updating with a certain *refresh rate*) the contents to the origin server host in the involved ONOS domain. We use the `add-content` ONOS CLI command that we created to add a given content version (second command argument) to the specified origin server host (first command argument) in the application's `EventuallyConsistentMap`. Further details about content distribution and state synchronization using Quorum-inspired write consistency are provided in Section 5.4.2 .

   On the other hand, in parallel with the updating of contents, our main process that is handled by the `main.exp` script starts issuing and serving client requests for specific contents. That was achieved using our `get-IP-content` CLI command which takes one argument, namely the requested `ContentName`, and returns the IP ad-

dress of the cache server containing the freshest/selected version of the requested content, Then, our script retrieves the content from the determined server using "wget". In addition, after each client request, continuous application-specific consistency and performance metrics related to that request are collected with our script using regular expressions in Tcl. More details about the content delivery strategy we follow using Quorum-inspired read consistency are given in Section 5.4.3.

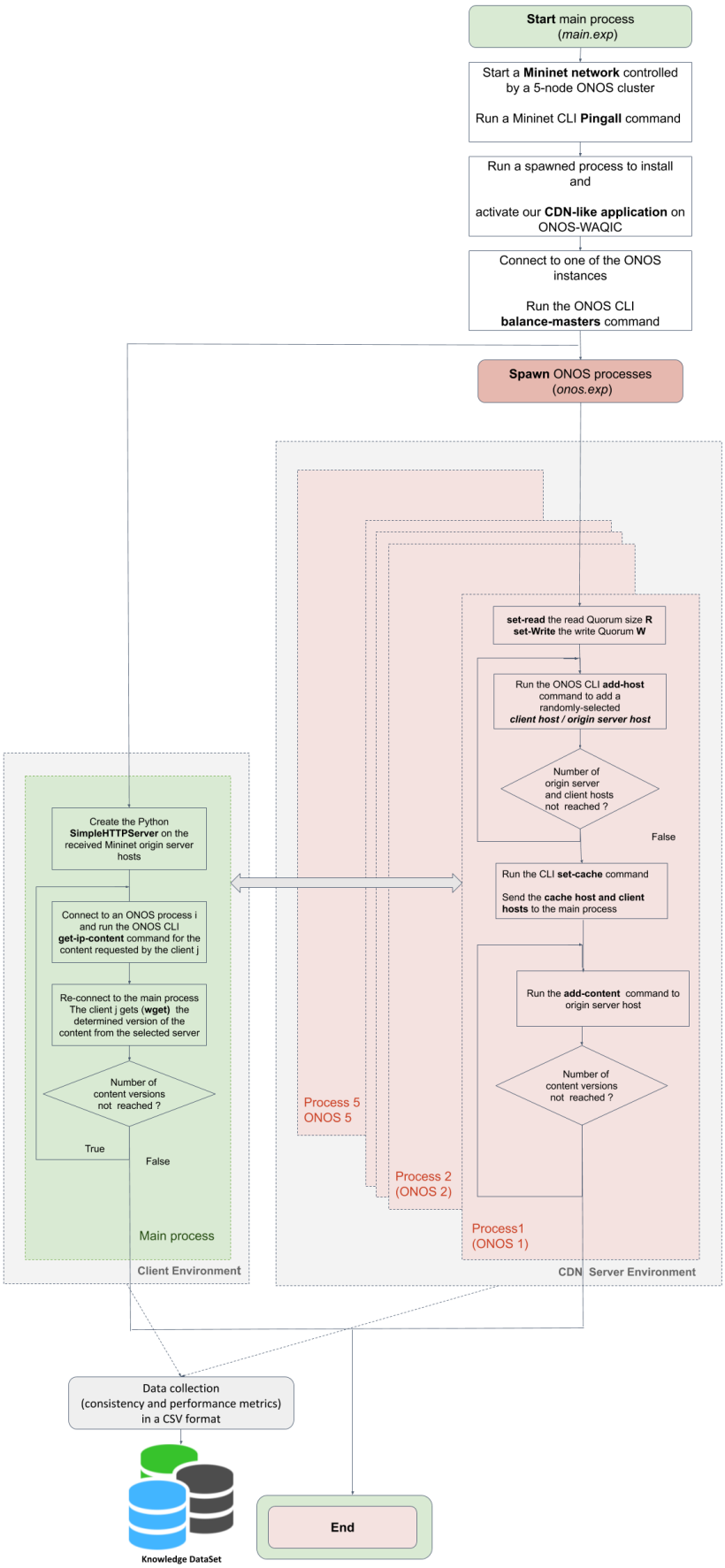


Figure 5.6: Overview of the main tasks executed by our TCL-Expect scripts

5.5.1.2 OpenAI Gym simulator

To implement the Machine Learning Module (see Section 5.3.2.2) for our CDN-like application on ONOS-WAQIC, we build a simulator based on OpenAI Gym [172], an open-source Python toolkit for developing and comparing reinforcement learning algorithms.

More specifically, we build a new environment to simulate knowledge exchange in an ONOS SDN cluster: We start by building an off-line dataset using our TCL-Expect scripts explained in Section 5.5.1.1. Our dataset stores the information collected by the Monitoring Module about client requests for specific CDN contents. As detailed in Section 5.4.3, for a given client request, the returned information contains the current values of the Quorum parameters R and W, the expected returned version of the content (content update step), the actual returned version of the content, *the staleness* of the returned content, *the delay* incurred in searching for the freshest version of the content from R controller replicas (latency1), *the read overhead*, *the write overhead*, and the application scenario determined by the Workload Identifier Module.

The dataset is fed to the Automatic Module which hands it over to the Machine Learning Module to learn online the read and write Quorum size parameters. Implemented with Gym, the latter module uses the dataset to learn the Kernel Density Estimation (KDE with scipy) for each metric using the data of some clients. That client data is selected with respect to the current configuration of R and W parameters. That configuration was set following an action performed by the agent (see the explanation of the Q-learning algorithm in Section 5.3.2.2 for more details). That way, using KDE, our ML Module estimates the expected metrics for each selected Quorum configuration, and then updates the Q-table with the Q-value of that action at that state, at each step (or episode) of the Q-learning algorithm.

5.5.1.3 Various learning agent policies

We implemented three learning agents that adopt different policies. The latter are compared and validated through five scenarios. Each scenario reflects a specific use case (e.g. a latency-sensitive application, a consistency-favoring application). To minimize the application's overall inter-controller overhead, our agents use the estimated overhead as a negative "reward" when performing actions (setting R and W) that change the environment state. The controlled and constrained agents are proposed with the aim to improve the simple greedy agent. Below is a brief description of these agents:

- *A simple* ϵ*-greedy agent* [173]: This agent follows a simple ϵ-greedy policy with a fixed ϵ value, where ϵ is the exploration rate and (1-ϵ) is the exploitation rate. We test three ϵ-agents: ϵ-greedy5 (ϵ=0.5), ϵ-greedy10 (ϵ=0.10) and ϵ-greedy15 (ϵ=0.15).

- *A controlled dynamic* ϵ*-greedy agent*: This agent follows a dynamic ϵ-greedy strategy where the exploration rate ϵ decays as the algorithm's episode count increases. The purpose is to account for the fact that the agent learns more about the environment in time, and becomes progressively more confident and "greedy" for exploitation.

  We use the following decay function for reducing ϵ as a function of episode count. x is the episode number.

$$f(x) = \epsilon * (0.5 + \log_{10}(2 - arctan(\frac{x}{10} - 2))) \tag{5.5}$$

  To attempt to satisfy the application's latency and staleness thresholds, the simple and controlled agents reject, at each exploitation episode, any action violating these constraints and remove its Q-value from the Q-table.

- *A constrained* ϵ*-greedy agent*: To make the agent learn how to satisfy the application's SLA, we create a Q-constraint list that we update over the episodes. Its size corresponds to the number of potential actions: the number of R and W combinations such that R + W ≤ N. The list represents the number of constraint violations by each Quorum configuration. The considered constraints are both the latency and staleness thresholds specified in the SLA. During each exploitation phase, we update the Q-constraint list, and use it to generate a new Q-list containing the Quorum configurations that give less constraint violations. These configurations are then exploited: They are compared using their Q-values in the Q-table (based on the estimated overhead reward) to select the best Quorum configuration (action) at that episode.

5.5.2 Results

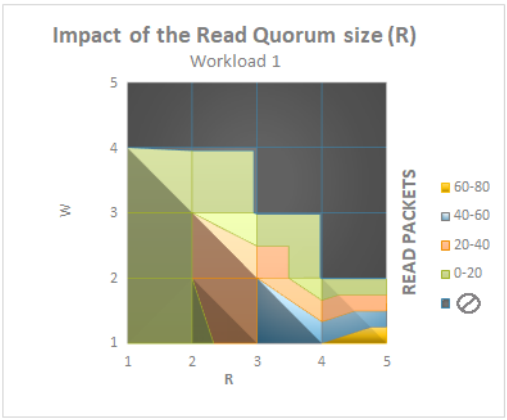
5.5.2.1 Impact of the Read and Write Quorum sizes

In this section, we present an experimental study that is aimed at assessing the impact of using different read and write Quorum sizes (R and W respectively) on the read and write inter-controller overheads of our CDN-like application running on a 5-node ONOS

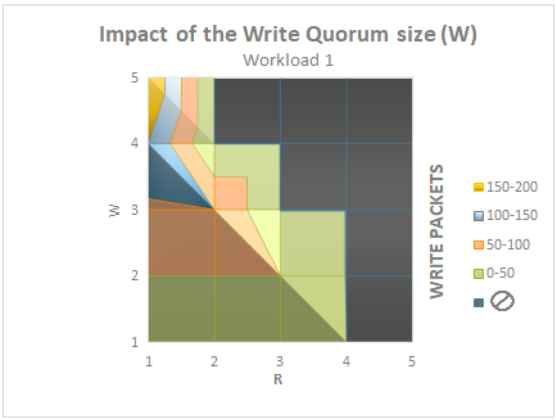
cluster in the network topology.

In the conducted experiments, we consider three application workloads that are representative of three application scenarios (see the `Workload Identifier Module` in Section 5.3.2.2 for more details). For the studied workloads, we show the captured read and write packets within a specified time interval (i.e. 400 ms in our tests) of read and write client operation accesses, for all possible eventually-consistent partial quorum configurations (R,W) (e.g. (R,W) combinations such that $R + W \leq N$ where $N = 5$).

Our results clearly show that, when increasing the read Quorum size R, read packets increase, mainly in a read-dominated workload (see Figure 5.7(a)). In addition, increasing the write Quorum size W results in a drastic increase of the write packets, especially in a write-intensive workload. Their number reaches indeed 400 during the specified time interval for a partial Quorum configuration where W is equal to 5 (see Figure 5.8(b).

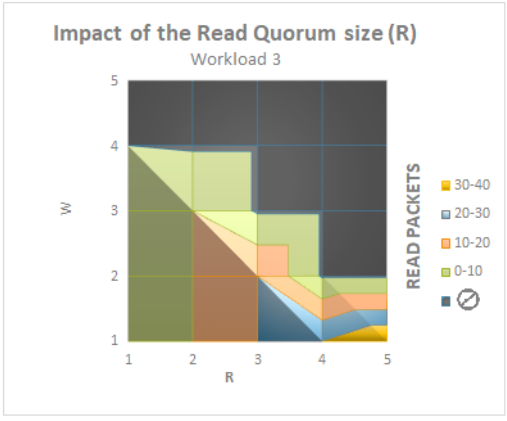


(a) Read packets when varying (R,W)

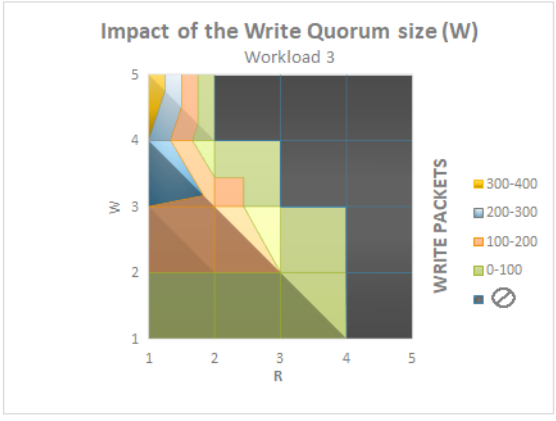


(b) Write packets when varying (R,W)

Figure 5.7: Workload 1: A Read-intensive application scenario



(a) Read packets when varying (R,W)



(b) Write packets when varying (R,W)

Figure 5.8: Workload 3: A Write-intensive application scenario

Given the high inter-controller overheads observed in our experimental data for certain Quorum configurations, we propose to tune the R and W parameters and therefore optimize the configuration of the Quorum system to better match the varying application SLA requirements and the dynamic application workloads, as we further discuss in the following sections.

5.5.2.2 Quorum configuration optimization

5.5.2.2.1 Dynamic application SLA requirements

To evaluate our ONOS-WAQIC proposal for the CDN-like application we developed, we run our TCL-Expect scripts (see Section 5.5.1.1) with a 5-node ONOS cluster according to different scenarios. In these scenarios, we use different partial Quorum configurations (R, W), and we follow various application workloads with respect to different ratios between read and write operations. Then, we use the data collected as an input to our Q-Learning simulator (see Section 5.5.1.2). In the simulator environment, we set $\alpha = \gamma = 0.5$ and the number of episodes to 1000. We also consider different test scenarios that reflect different application requirements in terms of performance and consistency as summarized in table 5.1.

In particular, using our dataset and knowing the refresh rate of our CDN-like application, we learn the t_staleness ranges. In other words, we learn the relationship between the t_staleness value of a certain content being returned and by how many versions that returned content is old. As a result, estimating the t_staleness ranges allowed us to set the time-based staleness thresholds in the SLA while having an idea about the associated version-based staleness thresholds.

Test scenarios	Latency threshold (ms)	*t*_Staleness threshold (ms)	*k*_Staleness Version old
n°1	5	300000	3
n°2	25	220000	2
n°3	50	120000	1

Table 5.1: Application SLA scenarios

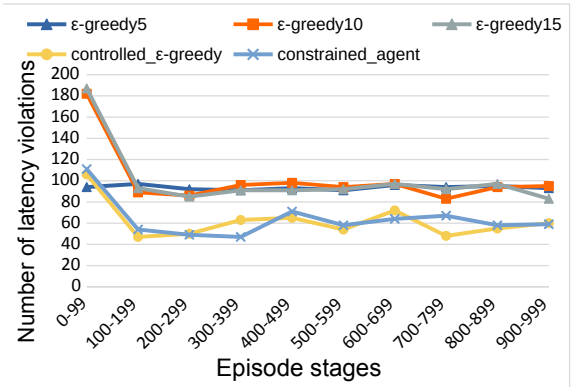
In each test scenario that we run on the simulator, our application expresses the performance and consistency SLAs using the latency threshold (in ms) and the staleness threshold (in ms). For example, in scenario n°3, our application which is consistency-favoring enforces the following SLA: It expects that a read operation gets a reply in under 100ms,

and returns a content value no older than 120seconds (i.e. no older than 1 version stale). Accordingly, our consistency approach attempts to find the best Quorum combination of R and W that minimizes the application's read and/or the write inter-controller overheads while ensuring the desired performance-consistency trade-offs.

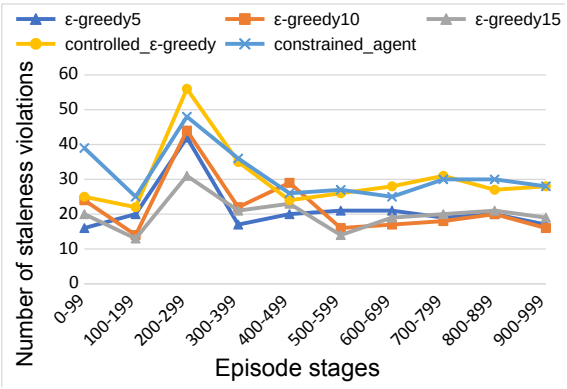
For a given Quorum configuration, we compute the read overhead ratio by normalizing the generated read overhead (bytes/s) with respect to the Quorum configuration generating the maximum read overhead and zero write overhead (the configuration (R = 5,W = 1)) in our case) for each application scenario. We follow the same steps for computing the write overhead ratio based on the the generated write overhead with respect to the Quorum configuration (R = 1,W = 5) which corresponds to the standard implementation of ONOS's eventual consistency model. On the other hand, whenever we aim to minimize both the read and write overheads (e.g. in a balanced workload scenario), we consider the mean of the read and write overhead ratios which we will subsequently refer to as the global overhead ratio.

In Figures 5.9, 5.10 and 5.11, we show the results of our experimental tests for the three considered application scenarios. To study the impact of changing the application SLA requirements, we set the application workload to Workload 2 (a balanced workload scenario that has a balanced ratio between the read and write operation accesses) in which our consistency approach attempts to minimize the global overhead ratio, and satisfy the staleness and latency SLA thresholds set by the application. Moving from one application workload scenario to another (e.g. a read-intensive scenario) will be dealt with in the following section.

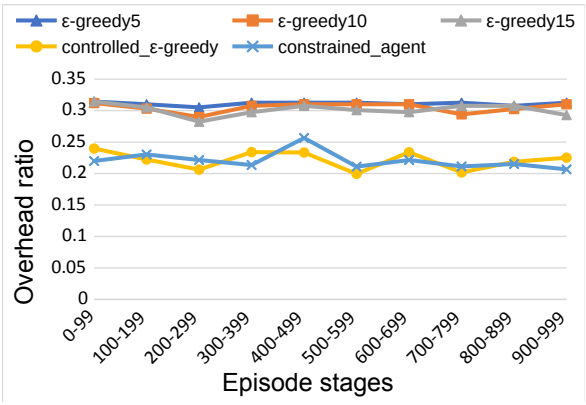
Figure 5.9 shows that, in a latency-sensitive application scenario, the constrained and the controlled agent policies are the most appropriate. The number of constraint violations decreases with episode stages (see Figures 5.9(a) and 5.9(b)), and the generated global (read and write) inter-controller overhead (see Figure 5.9(c)) is minimal as compared to the simple greedy agent policy, and to the standard ONOS implementation. We also notice that the three agents converge towards Quorum configurations where R = 1 (i.e. (R = 1,W = 2), (R = 1,W = 3) and (R = 1,W = 4)). This is due to the given strong constraint on latency.



(a) Latency violations



(b) Staleness violations



(c) Overhead ratio

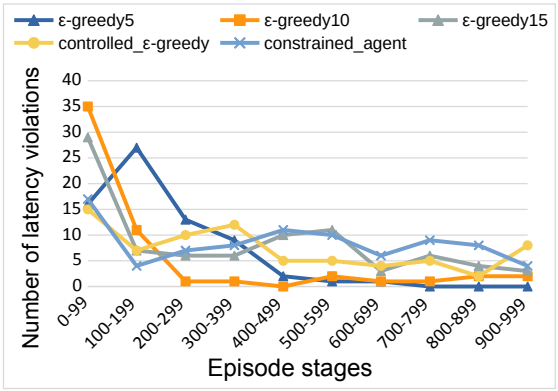
Figure 5.9: Scenario 1: A Latency-sensitive application

Figure 5.10 shows that, in a balanced application scenario, the constrained and the controlled agent policies offer the best real-time trade-offs between the application's latency and staleness needs (see Figures 5.10(a) and 5.10(b)) while ensuring minimal global overhead ratio (approximately 25%) (see Figure 5.10(c)). In particular, the constrained agent converges towards balanced Quorum configurations (i.e. (R = 2,W = 2) and (R = 2,W = 3)). On the other hand, the simple ϵ-greedy agents provide a small number of latency violations, but at the cost of generating more overhead.

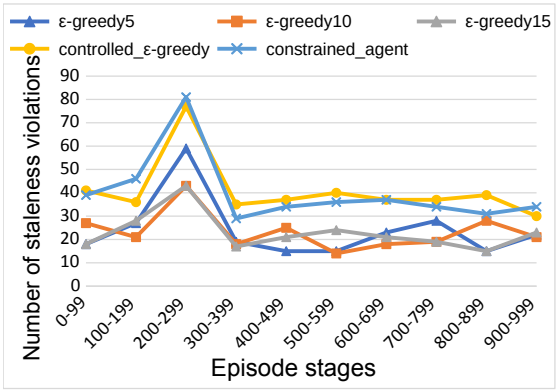
As can be seen from Figure 5.11, in a consistency-favoring application scenario, all agents perform well at reducing the staleness violations (see Figure 5.11(b)), especially the simple greedy agents. Besides, all agents respect the relaxed latency constraint (see Figure 5.11(a). They all converge towards a common Quorum configuration (R = 3,W = 2). We also note that the constrained and controlled agents ensure a significant gain in overhead, almost 80%.

Other scenarios were tested like an application scenario where latency is favored and consistency is completely relaxed ("any"). Our results showed that, in such scenarios, the

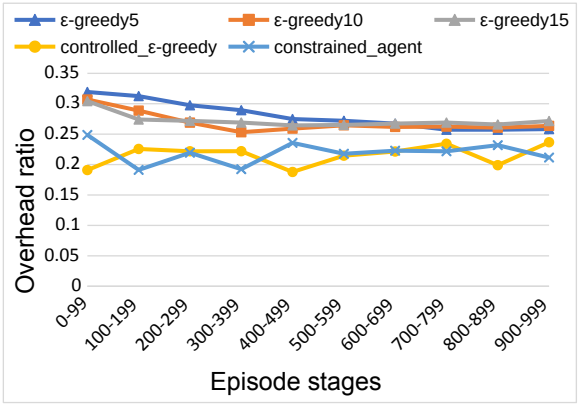
agents converge to a common Quorum configuration (R = 1, W = 1).



(a) Latency violations

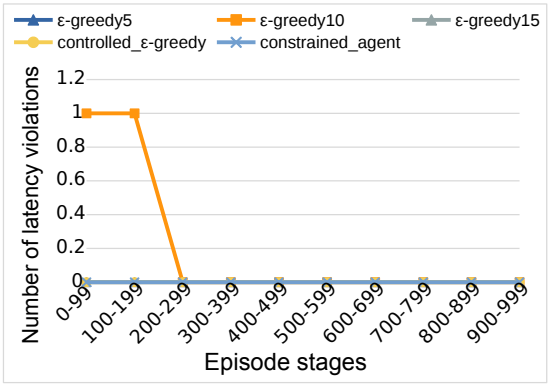


(b) Staleness violations

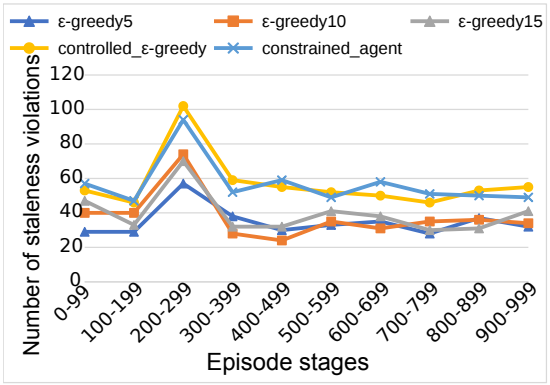


(c) Overhead ratio

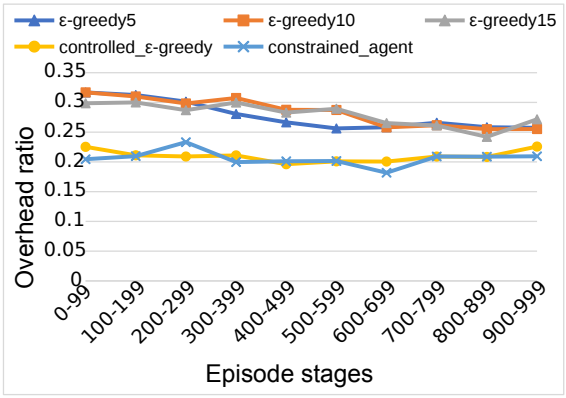
Figure 5.10: Scenario 2: A Consistency/Latency-balancing application



(a) Latency violations



(b) Staleness violations



(c) Overhead ratio

Figure 5.11: Scenario 3: A Consistency-favoring application

5.5.2.2.2 Dynamic application workloads

In this section, we aim to assess the ability of our adaptive Quorum-inspired consistency strategy (ONOS-WAQIC) for the CDN-like application we developed, to adapt to time-varying application workloads. The dynamic changes in such application workload patterns may indeed affect the observed network and application metrics (e.g. inter-controller overhead, staleness and access latency).

Taking that into consideration, our adaptive consistency model attempts to adjust the consistency level at runtime by continuously tuning the Quorum configuration parameters in order to better match the varying workloads.

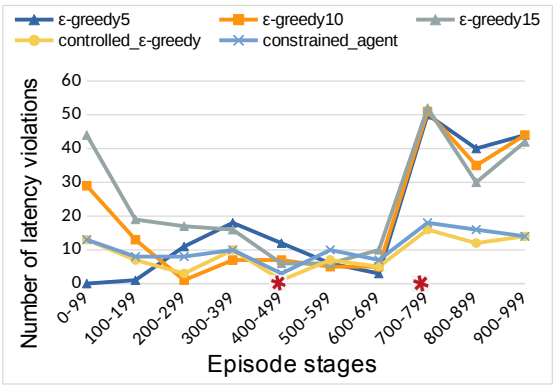
In this context, we consider three workloads as discussed in Section 5.3.2.2 (see the `Workload Identifier Module`). In the three studied workloads, our model aims to satisfy the latency and staleness SLA requirements. Additionally, in the read-dominated workload (Workload 1), our model attempts to minimize the read overhead. Conversely, in a write-intensive workload (Workload 3), it focuses on reducing the write overhead. Fi-

nally, in a balanced workload, our approach aims to minimize both the read and write overheads (the global overhead).

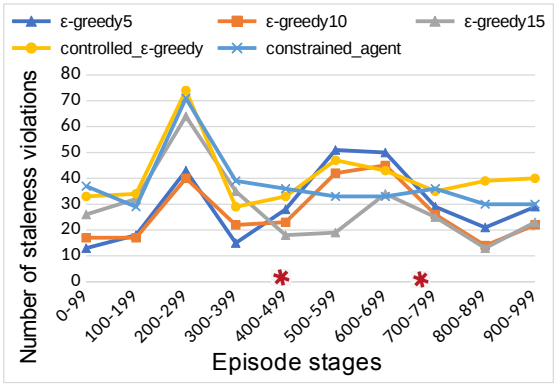
To experiment with these workloads, we fix the application scenario to 2 (see Section 5.5.2.2.1) which represents an application scenario with balanced consistency (staleness)/latency SLA requirements. Then, we conduct some tests on our Q-learning simulator. During these tests, we apply different variations in the application workload. More specifically, the first time period (the first 400 episodes) is characterized by a balanced workload (Workload 2). At episode 400, we run a read-dominated workload (Workload 1). Finally, starting from episode 700, we consider a write-intensive workload (Workload 3).

As we can see from Figure 5.12, our results clearly show that, unlike the simple ϵ-greedy agents, the constrained and controlled agents react quickly to the dynamic workload variations. The latter not only offer balanced real-time trade-offs between the performance (latency) and consistency (staleness) application SLA requirements, but also provide minimal overhead at runtime.

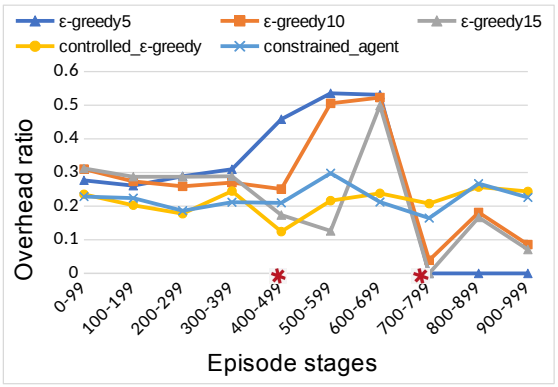
Besides, when analyzing the generated Quorum configurations during the conducted tests, we observe that, in Workload 1, the constrained agent converges to Quorum configurations where R is minimal in order to reduce the application's read inter-controller overhead. On the other hand, in Workload 3, the Quorum configurations where W is small are eventually selected. Finally, in Workload 2, the constrained agent converges to balanced Quorum configurations where R=W=2 to reduce the application's read and write inter-controller overheads.



(a) Latency violations



(b) Staleness violations



(c) Overhead ratio

Figure 5.12: Dynamic changes in the Workload (Workload 2-Workload 1-Workload 3) in a Consistency/Latency-balancing application scenario (Scenario2)

5.6 Conclusion

In this chapter, we further studied the use of an adaptive and continuous consistency model for the distributed ONOS controllers following the notion of partial Quorum consistency. Our consistency adaptation strategy was implemented for a CDN-like application we developed on top of ONOS. It mainly consists in turning ONOS's optimistic replication technique into a more scalable and intelligent Quorum-inspired replication strategy using various Q-learning RL approaches. Our experiments showed that the constrained ϵ-greedy approach we tested in a 5-node ONOS cluster proved efficient in helping our CDN-like application find the appropriate (or optimal) read an write Quorum replication parameters at-runtime. In fact, the adjustable and time-varying partial Quorum configurations determined by our strategy at runtime have achieved, under changing network and application workload conditions, balanced trade-offs between the application's continuous performance (latency) and consistency (staleness) requirements.

In addition, these real-time trade-offs ensured a substantial reduction in the application's inter-controller read and write overhead (especially in a large-scale ONOS network) while satisfying the application-defined thresholds specified in the given application SLA.

Finally, it is worth noting that our proposed adaptive and Quorum-inspired consistency model could be further enhanced by leveraging the compulsory Anti-Entropy reconciliation mechanisms proposed in the previous chapter (*expanding partial Quorums*). Such mechanisms are indeed useful in particular cases (e.g. failure scenarios, controller crashes) where the system consistency observed by the applications is at high risk, and cannot be fixed only by adjusting the Quorum parameters.

Conclusions and perspectives

«Software Defined Networks
and the maturing of the Internet»

Nick Mckeown,
IET Appleton, 2014

1 Summary of contributions

Software-Defined Networking (SDN) has increasingly gained traction over the last few years in both academia and research. The SDN paradigm builds its promises on the separation of concerns between the network control logic and the forwarding devices, as well as the logical centralization of the network intelligence in software components. Thanks to these key attributes, SDN is believed to work with network virtualization to fundamentally change the networking landscape towards more flexible, agile, adaptable and highly automated Next Generation Networks.

Despite all the hype, SDN entails many concerns and questions regarding its implementation and deployment. For instance, current SDN deployments based on physically-centralized control architectures raised several issues of scalability and reliability. As a result, distributed SDN control architectures were proposed as a suitable solution for overcoming these problems. However, there are still ongoing community debates about the best and most optimal approach to decentralizing the network control plane in order to harness the full potential of SDN.

In the early stages of this work, we conducted a comprehensive literature survey of the wide variety of existing SDN controller platforms. Besides reviewing the SDN concept and studying the SDN architecture as compared to the traditional network architecture, we proposed a taxonomy of state-of-the-art SDN controller platforms by categorizing them in two ways: based on a physical classification or a logical classification. Our thorough study of these SDN platform proposals allowed us to shed more light on their advantages

and drawbacks and to develop a critical awareness of the challenges facing the distributed control in SDNs.

In particular, the scalability, reliability, consistency, and interoperability of the SDN control plane are among the key competing challenges encountered in designing an efficient and robust high-performance distributed SDN controller platform. Although considered as the main limitations of fully centralized SDN control designs, scalability and reliability are also major concerns that are expressed in the context of distributed SDN architectures. They are indeed highly impacted by the structure of the distributed control plane (e.g. flat, hierarchical or hybrid organization) as well as the number and placement of the multiple controllers within the SDN network. Achieving such performance and availability requirements usually comes at the cost of guaranteeing a consistent centralized network view that is required for the correct behavior of SDN applications. Consistency considerations should therefore be explored among the trade-offs involved in the design process of a decentralized SDN controller platform.

Giving that rich variety of promising SDN controller platforms with their broad range of major challenges, we argue that developing a brand-new one may not be the best solution. However, it is essential to leverage the existing platforms by aggregating, merging and improving their proposed ideas in order to get as close as possible to a common standard that could emerge in the upcoming years. That distributed SDN controller platform should meet the emerging challenges associated with large-scale deployments and, most importantly, with next generation networks (e.g. IoT [174] and Fog Computing [175]).

With these considerations in mind, we intended to tackle, in the further stages of this work, some of the previously discussed challenges that are associated with the complex problem of designing a distributed SDN control plane. To that end, we propose to split that problem into two manageable challenges which are correlated: The controller placement problem (1) and the knowledge sharing problem (2). The first problem investigates the required number of controllers along with their appropriate locations with respect to the desired performance and reliability objectives and depending on the existing constraints. The second one is related to the type and amount of information to be shared among the controller instances given a desired level of consistency.

Firstly, we address the SDN controller placement optimization problem in the context of large-scale IoT-like networks. To that end, we put forward four scalable strategies that cover different aspects of the multi-objective controller placement optimization

problem with respect to multiple reliability and performance metrics that are considered according to different uses and contexts. To assess these strategies, two heuristic-based approaches were proposed with the objective of finding high-quality approximate solutions to the controller placement problem in a reasonable computation time: A clustering approach (PAM-B) based on a dissimilarity score and a modified genetic approach (NSGA-II). Our results demonstrated the potential of clustering techniques in delivering appropriate controller placement configurations that achieve balanced trade-offs among the competing performance and reliability criteria at scale.

Then, we investigate the knowledge dissemination problem between the distributed SDN controllers by proposing an adaptive multi-level consistency model following the notion of continuous consistency for the distributed SDN controllers. That model presents many advantages for the SDN applications when compared to the strong consistency and eventual consistency extremes, especially in large-scale deployments: It delivers the scalability, performance and availability benefits of an eventual consistency model, but has the additional advantage of controlling the observed state inconsistencies in an application-specific manner. More specifically, we propose two different scalable consistency approaches for the open-source ONOS controllers and compare them with ONOS's static strategies to eventual state consistency.

The first consistency approach was implemented for a source routing application on top of ONOS. It consists in turning ONOS's eventual consistency model into an adaptive consistency model using the *Anti-Entropy reconciliation period* as a *control knob* for an adaptive fine-grained tuning of consistency levels. Besides ensuring the application's continuous consistency requirements (i.e. *Numerical Error* bounds) as specified in the given application SLA, our results showed a substantial reduction in the Anti-Entropy reconciliation overhead as compared to ONOS's static consistency scheme at scale.

The second approach extends the adaptive consistency strategy to the optimistic replication technique used in ONOS's eventual consistency model. It was implemented for a CDN-like application we developed on top of the ONOS controllers. It mainly consists in turning ONOS's optimistic replication technique into a more scalable and intelligent Quorum-inspired replication strategy using various Q-learning RL approaches: In particular, it uses the *read and write partial Quorum parameters* as adjustable *control knobs* for a fine-grained consistency tuning, rather than relying on Anti-Entropy reconciliation mechanisms. Our experiments showed that the proposed constrained ϵ-greedy approach

proved efficient in finding at runtime the appropriate read an write Quorum replication parameters that achieve, under changing network and workload conditions, balanced trade-offs between the application's continuous performance (*latency*) and consistency (*staleness*) requirements. These real-time trade-offs ensured a great reduction in the application overhead while satisfying the application requirements specified in the SLA.

2 Perspectives and future work

Based on the promising results of this work, the study can be further extended with a variety of research perspectives.

- First, the controller placement strategies proposed in the third chapter could be further enhanced by including *a dynamic controller placement policy*. The latter should take into account the dynamic nature of the network such as the network load or a dynamic network topology.

- Our adaptive consistency model proposed for the distributed ONOS controllers reduces the application inter-controller overhead and tunes the consistency level at runtime, in order to achieve, under changing application workload conditions, balanced real-time trade-offs between the application's continuous performance and consistency requirements (as specified in the given SLA). Other important factors to be considered as part of our future work include the *changing network conditions* in the case of in-bound SDN control.

- The adaptive consistency strategies proposed in this work dynamically adjust at runtime the same consistency level for all the SDN controller instances in the cluster in order to meet certain network and application requirements. Another potential approach is to assign different consistency levels to the different controllers (granular *per-controller consistency*) depending on application requirements. Accordingly, in the case of our Quorum-inspired consistency approach, adjusting the consistency level at run-time would imply assigning different Quorum parameter configurations (R, W) to the considered SDN controller replicas in the cluster.

- The Quorum-inspired consistency strategy presented in this dissertation uses a constant *replication factor* that is equal to the total number of controller nodes in the

cluster. It would also be interesting to use the replication degree as a tunable configuration parameter (or a control knob) to disseminate the knowledge in specific geographical areas according to various application scopes and needs.

- Although the main focus of this work was placed at dynamically adjusting the consistency level of SDN application states (which use controller states), the work can be further extended to the controller states (internal controller applications). Indeed, the long-term goal of this work is to design *adaptively-consistent controllers* that adjust the consistency levels for both control and application planes under changing network conditions.

- The adaptive and continuous Quorum-inspired consistency approach proposed in this work was implemented for a certain type of applications (e.g. a CDN-like application), and using a 5-node ONOS controller cluster in a emulated network environment. That helped us to assess the feasibility of our solution in distributed SDN control, and to develop a critical awareness about the faced challenges. The next step of this work is to develop a more effective Proof-of-Concept (PoC) for distributed SDN controllers in a production environment. This can be achieved by setting up an SDN test bed using more than five controller instances in the SDN cluster, and by experimenting with a variety of industrial use cases, in order to test the performance and the degree of functionality of our approach in large-scale real-world SDN deployments.

- In this work, the controller placement problem and the knowledge sharing problem between the distributed SDN controllers are considered as correlated problems, but have been addressed separately. It would be interesting to consider the knowledge dissemination challenge (state consistency metrics) when investigating the optimal placement of controllers. For example, minimizing the inter-controller latencies in the controller placement process reduces the cost of inter-controller communications and enhances network consistency and performance.

- Finally, in this work, we placed a special focus on tackling the control consistency issues in SDN, and we proposed practical solutions that we applied to current SDN controller platforms. In fact, we believe that it is highly important for a distributed SDN architecture to support fault tolerance and consistency checks in order to en-

sure an efficient and secure SDN control plane [176]. As part of our future work, we propose to further address the data/state consistency challenges from a security perspective. In particular, we highlight the necessity to secure the communications between the SDN controllers against the potential threats facing the SDN control plane. A straightforward example of these threats is a malicious SDN controller replica that sends erroneous data to compromise the system by harming a particular service or network, or by favoring its actions to the detriment of the rest of the controller replicas. A potential solution is to use a blockchain to provide a guarantee of non-repudiation and non-alteration (integrity) of data. This blockchain can also be used to store smart contracts. These contracts are programs that control the permission of data exchanges between the parties under certain conditions. In particular, smart contracts can enable a fine-grained access control of the knowledge to be shared between the SDN controller replicas by establishing elaborate rules (e.g. by allowing access to a particular knowledge only for specific controller replicas).

Version abrégée en Français

1 Contexte général

La croissance continue du trafic de données, l'émergence de la virtualisation des réseaux ainsi que l'utilisation sans cesse croissante d'équipements mobiles dans l'environnement réseau moderne ont mis en lumière les nombreux problèmes inhérents à l'architecture conventionnelle de l'Internet. Ainsi, la tâche de gestion et de contrôle des informations provenant d'un nombre croissant d'appareils connectés devenait de plus en plus complexe et spécialisée.

En effet, l'infrastructure réseau traditionnelle est considérée comme très rigide et statique, étant conçue à l'origine pour un type de trafic particulier, à savoir des contenus monotones en texte, ce qui la rend peu adaptée aux flux multimédia interactifs et dynamiques d'aujourd'hui générés par des utilisateurs de plus en plus exigeants et mobiles. Parallèlement aux besoins liés au multimédia, l'émergence récente de l'Internet des Objets (IoT) a permis la création de nouveaux services avancés avec des exigences de communication plus strictes afin de prendre en charge des cas d'utilisation innovants. En particulier, la santé connectée est un cas typique d'utilisation de l'IoT où les services de soins de santé fournis à des patients distants (e.g. diagnostic, chirurgie, dossiers médicaux) sont extrêmement intolérants au regard du délai, de la qualité et de la confidentialité. Ces données sensibles et ce trafic critique ne sont guère pris en charge par les réseaux traditionnels.

De plus, dans l'architecture traditionnelle où la logique de contrôle est purement distribuée et localisée, la résolution d'un problème de réseau spécifique ou le réglage d'une stratégie de réseau particulière nécessite d'agir séparément sur les périphériques concernés et de modifier manuellement leur configuration. Dans ce contexte, la croissance actuelle du nombre d'appareils et de données a accru les problèmes d'évolutivité en rendant ces interventions humaines et ces opérations de réseau plus dures et plus sujettes aux erreurs.

Globalement, il est devenu particulièrement difficile pour les réseaux actuels de fournir le niveau requis de qualité de service (QoS), et encore moins la qualité d'expérience (QoE) qui introduit de nouvelles exigences centrées sur l'utilisateur. Pour être plus précis, se fier uniquement à la QoS traditionnelle basée sur des paramètres de performances techniques (e.g. bande passante et latence) s'avère insuffisant pour les réseaux avancés et en expansion d'aujourd'hui. De plus, répondre à ce nombre croissant d'indicateurs de performance est une tâche d'optimisation complexe qui peut être traitée comme un problème NP-complet. Par ailleurs, les opérateurs de réseaux réalisent de plus en plus l'importance de l'expérience globale de l'utilisateur final et de sa perception subjective des services fournis qui permettent de résoudre des problèmes que les mécanismes fondés sur la qualité de service ont du mal à résoudre. En conséquence, les tendances actuelles en matière de gestion de réseau se dirigent vers ce nouveau concept, couramment appelé QoE, qui représente la qualité globale d'un service de réseau du point de vue de l'utilisateur final.

Cela dit, cet énorme fossé entre, d'une part, les progrès réalisés dans les technologies informatiques et logicielles et, d'autre part, l'infrastructure de réseau sous-jacente traditionnelle, non évolutive et difficile à gérer [1; 2], a souligné le besoin d'une plate-forme de réseau automatisée et adaptable en continu [3] qui facilite les opérations du réseau et réponde aux besoins de l'IoT. Dans ce contexte, plusieurs stratégies de recherche ont été proposées pour intégrer des approches automatiques et adaptatives dans l'infrastructure actuelle afin de relever les défis de l'évolutivité, de la fiabilité et de la disponibilité du trafic en temps réel, garantissant ainsi la QoE de l'utilisateur.

Alors que des alternatives radicales soutiennent qu'une nouvelle architecture de réseau doit être construite à partir de zéro en rompant avec l'architecture de réseau conventionnelle et en apportant des modifications fondamentales pour répondre aux exigences actuelles et futures, d'autres alternatives plus réalistes sont appréciées pour introduire de légères modifications adaptées aux besoins permettant d'effectuer une transition progressive de l'architecture du réseau sans pour autant causer de perturbations coûteuses aux opérations réseau existantes.

En particulier, la première alternative de réseau superposé (Overlay Network) introduit une superposition de couche d'application au-dessus du substrat de routage conventionnel afin de faciliter la mise en œuvre de nouvelles approches de contrôle de réseau. Cependant, l'inconvénient évident des réseaux superposés est qu'ils dépendent de plusieurs aspects (par exemple, des nœuds de superposition sélectionnés) pour obtenir les perfor-

mances requises. En outre, on peut reprocher à ces réseaux d'aggraver la complexité des réseaux existants en raison des couches virtuelles supplémentaires.

Par ailleurs, le paradigme récent du réseau piloté par logiciel (SDN) [5] offre la possibilité de programmer le réseau et facilite ainsi l'introduction d'approches de contrôle automatique et adaptatif en séparant le matériel (plan de données) et le logiciel (plan de contrôle), permettant leur évolution indépendante. SDN vise la centralisation du contrôle du réseau, offrant une visibilité améliorée et une plus grande flexibilité pour gérer le réseau et optimiser ses performances. Par rapport à l'alternative Overlay Network, le réseau SDN a la capacité de contrôler l'ensemble du réseau, non seulement un ensemble sélectionné de nœuds, et d'utiliser un réseau public pour le transport de données. En outre, le SDN évite aux opérateurs de réseaux la tâche fastidieuse de créer temporairement le réseau de recouvrement approprié pour un cas d'utilisation spécifique. Au lieu de cela, il fournit un cadre de programmation inhérent aux applications de contrôle et de sécurité d'hébergement développées de manière centralisée, tout en tenant compte les exigences de l'IoT [6] afin de garantir la QoE de l'utilisateur.

2 Motivations

Malgré le vif intérêt que suscite le SDN, son déploiement dans le contexte industriel en est encore à ses débuts. Il faudra peut-être encore beaucoup de temps avant que la technologie ne mûrisse et que les efforts de normalisation portent leurs fruits pour que le potentiel du SDN soit pleinement exploité.

En effet, parallèlement au battage publicitaire et à l'excitation, plusieurs préoccupations et questions ont été exprimées concernant l'adoption généralisée des réseaux SDN. Par exemple, des études sur la faisabilité du déploiement du réseau SDN ont révélé que la centralisation physique du plan de contrôle dans un seul composant logiciel programmable, appelé contrôleur SDN, est limitée par plusieurs facteurs tels que les problèmes d'évolutivité, de disponibilité et de fiabilité. Peu à peu, il est devenu inévitable de considérer le plan de contrôle comme un système distribué [7], dans lequel plusieurs contrôleurs SDN sont chargés de gérer l'ensemble du réseau, tout en maintenant une vue réseau logiquement centralisée.

À cet égard, les communautés réseau ont débattu du meilleur moyen de mettre en œuvre des architectures SDN distribuées tout en tenant compte des nouveaux défis posés par ces systèmes distribués. En conséquence, plusieurs solutions SDN ont été explorées

et de nombreux projets SDN ont vu le jour. Chaque plate-forme de contrôleurs SDN proposée a adopté une approche de conception architecturale spécifique basée sur divers facteurs tels que les aspects d'intérêt, les objectifs de performance, le cas d'utilisation SDN déployé, ainsi que les compromis liés à la présence de multiples défis conflictuels et concurrents.

À ce stade, nous soulignons l'importance de procéder à une analyse sérieuse des solutions SDN proposées afin d'envisager les tendances potentielles susceptibles d'orienter les recherches futures dans ce domaine. Nous mettons tout particulièrement l'accent sur les conceptions de contrôle SDN distribuées dans le but de résoudre certains des problèmes majeurs rencontrés dans la décentralisation des plans de contrôle SDN dans le contexte de déploiement à grande échelle.

Les principales motivations de ce travail sont les suivantes:

- Garantir une compréhension approfondie des plates-formes de contrôleurs SDN distribuées, les plus récentes et à la pointe de la technologie, et développer une prise de conscience critique des recherches et des défis opérationnels clés en cours et à venir pour la conception et le déploiement de telles plates-formes.

- Proposer de nouvelles approches pour décentraliser le plan de contrôle SDN dans les réseaux à grande échelle. Un tel plan de contrôle SDN décentralisé doit être efficace (c'est-à-dire évolutif, performant et robuste) car il doit répondre aux exigences des application de contrôleurs SDN (par exemple, l'évolutivité, la fiabilité et la cohérence).

- Ouvrir la voie à l'émergence d'un nouveau standard commun pour le plan de contrôle SDN distribué. Cette norme devrait également assurer la communication inter-contrôleurs entre différentes technologies de contrôleurs spécifiques au fournisseur pour une meilleure interopérabilité.

3 Contributions

Dans cette section, nous décrivons les principales contributions de ce travail. Plus précisément, nous proposons de nouvelles approches pour décentraliser le plan de contrôle SDN dans les réseaux à large échelle tout en abordant certains des principaux problèmes associés:

(1) Des stratégies prenant en compte plusieurs critères d'*évolutivité* et de *fiabilité* pour le *placement de contrôleurs* SDN distribués à large échelle à l'aide de différents types d'algorithmes d'optimisation multi-critères (voir Chapitre 3):

Nous avons abordé le problème de contrôle SDN distribué en étudiant le problème de placement de contrôleurs SDN dans les réseaux à large échelle de type IoT. Nous avons proposé des stratégies de placement de contrôleurs sensibles à la fiabilité et l'évolutivité, qui traitent plusieurs aspects du problème d'optimisation de placement de contrôleurs au regard de multiples critères de fiabilité et de performance et selon différents usages et contextes. Ces stratégies utilisent deux types différents d'heuristiques: un algorithme de classification basé sur PAM (Partitioning Around Medoids) et un algorithme génétique modifié appelé NSGA-II (Non-dominated Sorting Genetic Algorithm II). Ces algorithmes multi-critères ont été comparés en termes de temps de calcul, et de la qualité des configurations finales de placement de contrôleurs SDN dans un environnement applicatif.

(2) Un modèle *cohérence* continue et adaptative pour les contrôleurs SDN distribués: un nouveau mécanisme de réconciliation Anti-Entropie pour les applications (avec des besoins de cohérence éventuelle) au-dessus des contrôleurs ONOS (voir Chapitre 4):

Nous avons abordé le problème du contrôle SDN distribué en étudiant le problème du partage des connaissances entre les contrôleurs SDN distribués. Nous avons proposé un modèle de cohérence adaptative à plusieurs niveaux, basé sur le concept de cohérence continue pour les contrôleurs SDN distribués. Cette approche a été implémentée pour une application de routage à la source au-dessus des contrôleurs ONOS d'accès libre (open-source). Elle consiste à transformer le modèle de cohérence éventuelle d'ONOS en un modèle de cohérence adaptative utilisant *la période de réconciliation Anti-Entropy* comme bouton de contrôle pour un réglage fin et adaptatif des niveaux de cohérence. Notre stratégie de cohérence proposée vise à garantir les exigences de cohérence continue de l'application (c'est-à-dire les seuils d'erreur numérique) telles que spécifiées dans le contrat SLA de l'application. L'objectif est également de minimiser la surcharge réseau de réconciliation Anti-Entropie par rapport à la stratégie de cohérence statique à grande échelle d'ONOS.

(3) Un modèle *cohérence* continue et adaptative pour les contrôleurs SDN distribués: une nouvelle stratégie de réplication basée sur le vote majoritaire (Quorum) pour les

applications (avec des besoins de cohérence éventuelle) au-dessus des contrôleurs ONOS (voir Chapitre 5):

Nous avons étudié plus en détail le problème de partage des connaissances dans le contrôle SDN distribué en proposant un modèle de cohérence adaptative et continue pour les contrôleurs ONOS distribués. L'approche a été mise en œuvre pour une application de type CDN au-dessus d'ONOS. Elle consiste à transformer le modèle de cohérence éventuelle d'ONOS en un modèle de cohérence adaptative en convertissant la technique de réplication optimiste d'ONOS en une stratégie de réplication plus évolutive suivant la cohérence basée sur le Quorum. Son but est d'améliorer le mécanisme de réplication d'ONOS: en utilisant *les paramètres de Quorum en lecture et en écriture* comme boutons de contrôle ajustables pour un réglage fin des niveaux de cohérence en lecture et en écriture, au lieu de s'appuyer sur des mécanismes de réconciliation Anti-Entropie (comme dans la contribution 2). L'objectif principal est de trouver, au moment de l'exécution, des configurations optimales de Quorum partiel qui permettent, en fonction de l'évolution des conditions du réseau et de la charge de travail de l'application, de trouver un compromis entre les exigences continues de l'application en termes de performance (*latence*) et de cohérence (*staleness*). Ces compromis en temps réel devraient permettre de minimiser la surcharge totale entre les contrôleurs tout en respectant les seuils définis par l'application spécifiés dans le contrat de niveau de service donné.

De plus, au regard du manque dans la littérature d'état de l'art autour de la problématique du contrôle distribué dans les réseaux SDN et vu son actualité, nous avons aussi contribué par:

- Une étude sur le SDN, en particulier sur les solutions de contrôle SDN distribuées (voir Chapitre 1):

  En plus d'expliquer les éléments fondamentaux de l'architecture SDN, nous avons proposé une taxonomie des plates-formes de contrôleurs SDN, les plus actuelles, en les classant de deux manières différentes: une classification physique et une classification logique.

- Une analyse approfondie des problèmes rencontrés par les plates-formes récentes de contrôleurs SDN distribués, et des différentes approches adoptées pour les résoudre (voir Chapitre 2):

Nous avons mené une analyse des principaux défis encore ouverts auxquels sont confrontées les plates-formes de contrôleurs SDN distribuées déjà abordées. Ces défis incluent les problèmes d'évolutivité, de fiabilité, de cohérence et d'interopérabilité du plan de contrôle SDN. En outre, nous avons exploré les approches potentielles pour relever ces défis en vue d'un déploiement optimal du SDN, et nous avons fourni des informations utiles sur les tendances émergentes et futures dans la conception de plans de contrôle SDN distribués plus efficaces.

4 Conclusion et travail réalisé

Le réseau défini par logiciel (SDN) a de plus en plus de succès dans le milieu académique ainsi que dans la recherche. Le paradigme SDN fonde ses promesses sur la séparation des préoccupations entre la logique de contrôle du réseau et les équipements de transmission réseau, ainsi que sur la centralisation logique de l'intelligence réseau dans des composants logiciels. Grâce à ces attributs clés, on estime que l'approche SDN, avec notamment les nouvelles technologies de virtualisation réseau (NFV), permettra de changer fondamentalement le paysage réseau et facilitera ainsi le passage aux réseaux de nouvelle génération plus flexibles, agiles, programmables, adaptables et hautement automatisés.

L'approche fondée sur le SDN soulève, néanmoins, de nombreuses préoccupations et questions concernant sa mise en œuvre et son déploiement. Par exemple, les déploiements SDN actuels basés sur des architectures de contrôle physiquement centralisées soulèvent encore plusieurs problèmes d'évolutivité et de fiabilité. Par conséquent, les architectures de contrôle SDN distribuées ont été proposées comme solution appropriée pour tenter de résoudre ces problèmes. Il n'empêche qu'il existe encore des débats communautaires sur la meilleure approche à adopter pour décentraliser le plan de contrôle du réseau afin d'exploiter pleinement le potentiel du réseau SDN. Ces discussions portent principalement sur les différents compromis et challenges impliqués dans la décentralisation du plan de contrôle SDN.

Au démarrage de cette thèse, nous avons mené une étude bibliographique sur la vaste gamme de plates-formes de contrôleurs SDN existantes. Outre la révision du concept SDN et l'étude de l'architecture SDN par rapport à l'architecture réseau traditionnelle, nous avons proposé une taxonomie des plates-formes de contrôleurs SDN les plus récentes, en les catégorisant de deux manières: sur la base d'une classification physique ou d'une classification logique. Notre étude approfondie de ces propositions de plate-formes

SDN nous a permis de mieux comprendre leurs avantages et inconvénients et de développer une prise de conscience critique des défis du contrôle distribué dans les SDNs.

En particulier, l'évolutivité, la fiabilité, la cohérence et l'interopérabilité du plan de contrôle SDN figurent parmi les principaux défis à relever pour la conception d'une plate-forme de contrôleurs SDN distribués robuste et performante. Bien que considérées comme les principales limites des conceptions de contrôle SDN totalement centralisées, l'évolutivité et la fiabilité sont également des préoccupations majeures qui sont exprimées dans le contexte des architectures SDN distribuées. Ils sont, en effet, fortement impactés par la structure du plan de contrôle distribué (par exemple, une organisation plate, hiérarchique ou hybride) ainsi que par le nombre et le placement des multiples contrôleurs au sein du réseau SDN. La réalisation de telles exigences en matière de performance et de disponibilité se fait généralement au détriment de la garantie d'une vue de réseau centralisée cohérente, nécessaire au bon comportement des applications SDN. Il convient donc d'examiner les considérations liées à la cohérence parmi les compromis impliqués dans le processus de conception d'une plate-forme décentralisée de contrôleurs SDN .

Compte tenu de la diversité des plate-formes prometteuses de contrôleurs SDN et de leur vaste éventail de défis majeurs, nous affirmons que le développement d'une plate-forme toute nouvelle n'est peut-être pas la meilleure solution. Cependant, il est essentiel de tirer parti des plates-formes existantes en agrégeant, fusionnant et améliorant les idées proposées afin de se rapprocher le plus possible d'une norme commune qui pourrait émerger dans les années à venir. Cette plate-forme de contrôleurs SDN distribués devrait répondre aux défis émergents associés aux déploiements à grande échelle et, plus important encore, aux réseaux de prochaine génération (par exemple, IoT [174] et Fog Computing [175]).

C'est dans cette optique que nous avons abordé, au cours des étapes ultérieures de ce travail, certains des problèmes évoqués précédemment et associés au problème complexe de la conception d'un plan de contrôle SDN distribué. Pour ce faire, nous avons proposé de scinder ce problème en deux parties gérables et corrélées: le placement de contrôleurs (1) et le partage des connaissances (2). La première partie s'occupe d'étudier le nombre requis de contrôleurs ainsi que leurs emplacements appropriés par rapport aux objectifs de performance et de fiabilité souhaités et en fonction des contraintes existantes. La seconde partie est liée au type et à la quantité d'informations à partager entre les instances de contrôleurs en fonction du niveau de cohérence souhaité.

Tout d'abord, nous abordons le problème d'optimisation de placement de contrôleurs SDN dans le contexte de réseaux à large échelle de type IoT. Pour ce faire, nous proposons quatre stratégies évolutives qui couvrent différents aspects du problème d'optimisation multi-objectifs de l'emplacement des contrôleurs SDN au regard de multiples métriques de fiabilité et de performance prises en compte en fonction de différents usages et contextes. Pour évaluer ces stratégies, deux approches heuristiques ont été proposées dans le but de trouver des solutions approximatives de haute qualité au problème de placement des contrôleurs dans un temps de calcul raisonnable: une approche de partitionnement (PAM-B) basée sur un score de dissimilarité et une approche génétique modifiée (NSGA-II). Nos résultats ont démontré le potentiel des techniques de partitionnement dans la perspective de fournir des configurations de placement de contrôleurs appropriées qui permettent un compromis équilibré entre les critères concurrents de performance et de fiabilité à grande échelle.

Ensuite, nous étudions le problème de partage des connaissances entre les contrôleurs SDN distribués en proposant un modèle de cohérence adaptatif à plusieurs niveaux, basé sur la notion de cohérence continue pour les contrôleurs SDN distribués. Ce modèle présente de nombreux avantages pour les applications SDN par rapport aux extrêmes de cohérence forte et de cohérence éventuelle (c'est-à-dire à terme), en particulier dans les déploiements à large échelle: Il offre les avantages d'évolutivité, de performance et de disponibilité d'un modèle de cohérence éventuel, mais présente l'avantage supplémentaire de contrôler les incohérences d'état observées d'une manière spécifique à l'application. Plus spécifiquement, nous proposons deux différentes approches de cohérence évolutives pour les contrôleurs open-source ONOS et nous les comparons avec les stratégies statiques d'ONOS pour la cohérence d'état éventuelle.

La première approche de cohérence a été mise en œuvre pour une application de routage à la source au-dessus d'ONOS. Elle consiste à transformer le modèle de cohérence éventuelle d'ONOS en un modèle de cohérence adaptative en utilisant la *période de réconciliation Anti-Entropie* comme *un bouton de contrôle* pour un réglage fin et adaptatif des niveaux de cohérence. En plus de garantir les exigences de cohérence continues de l'application (c'est-à-dire les limites d'*erreur numérique*) spécifiées dans le contrat de niveaux de service d'application donné (SLA), nos résultats ont montré une réduction substantielle de la surcharge due à la réconciliation Anti-Entropie par rapport à l'approche de cohérence statique à l'échelle d'ONOS.

La deuxième approche étend la stratégie de cohérence adaptative à la technique de réplication optimiste utilisée dans le modèle de cohérence éventuelle d'ONOS. Elle a été implémentée pour une application de type CDN que nous avons développée sur les contrôleurs ONOS. Elle consiste principalement à transformer la technique de réplication optimiste d'ONOS en une stratégie de réplication inspirée du Quorum, plus évolutive et plus intelligente, portant sur diverses approches d'apprentissage par renforcement, plus précisément de Q-learning: elle utilise notamment *les paramètres de Quorum partiel en lecture/écriture* comme *paramètres de contrôle ajustables* pour un réglage fin de la cohérence, plutôt que de reposer sur des mécanismes de réconciliation Anti-Entropie. Nos expériences ont montré que l'approche ϵ-greedy sous contraintes proposée s'avérait efficace pour trouver, au moment de l'exécution, les paramètres de réplication Quorum en lecture/écriture appropriés permettant d'obtenir, dans des conditions changeantes de réseau et de charge de travail de l'application, des compromis équilibrés entre les exigences continues de l'application en matière de performance (*latence*) et de cohérence (*staleness*). Ces compromis en temps réel ont permis de réduire considérablement la surcharge totale (entre contrôleurs) liée à l'application, tout en satisfaisant les exigences spécifiées dans les SLAs.

5 Liste des publications

Revues internationales avec comité de lecture

- **F. Bannour**, S. Souihi, A. Mellouk. "Distributed SDN Control: Survey, Taxonomy, and Challenges". *IEEE Communications Surveys and Tutorials (CST)*, 20(1):333–354, 2018.

Conférences internationales avec comité de lecture et actes

- **F. Bannour**, S. Souihi, A. Mellouk. "Adaptive Quorum-inspired SLA-Aware Consistency for Distributed SDN Controllers", *15th International Conference on Network and Service Management (CNSM)*, Halifax, Canada, 21-25 Octobre, 2019.

- **F. Bannour**, S. Souihi, A. Mellouk. "Adaptive State Consistency for Distributed ONOS Controllers", *IEEE Global Communications Conference (GLOBECOM)*, Abu Dhabi, 9–13 Décembre, 2018.

- **F. Bannour**, S. Souihi, A. Mellouk. "Scalability and Reliability Aware SDN Controller Placement Strategies", *13th International Conference on Network and Service Management (CNSM)*, Tokyo, Japan, 26-30 Novembre, 2017.

Conférences nationales avec comité de lecture et actes

- **F. Bannour**, S. Souihi, A. Mellouk. "Adaptive state consistency for distributed ONOS controllers", *Journée SDN 2018 « IDNs (Intelligence-Defined Networks) »*, présentation, 22 Novembre 2018, Paris, France.

- **F. Bannour**, S. Souihi, A. Mellouk. "Software-Defined Networking: A self-adaptive consistency model for distributed SDN controllers", *École d'été RESCOM 2017 du CNRS GDR RSD « Virtualisation dans les réseaux informatiques et dans le Cloud »*, 19-23 Juin 2017, Le Croisic, France.

- **F. Bannour**, S. Souihi, A. Mellouk. "Software-Defined Networking: Distributed SDN Control". *Colloque ARC 2017 du CNRS GDR MACS « Automatique et Réseaux de Communication »*, présentation, 16 Mai 2017, Paris, France.

- **F. Bannour**, S. Souihi, A. Mellouk. "The SDN controller placement problem, *École d'été RESCOM 2016 du CNRS GDR RSD « La 5G et l'Internet des Objets »*, poster, 13-17 Juin 2016, Guidel-plages, France.

Bibliography

[1] Diego Kreutz, Fernando M. V. Ramos, Paulo Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):63, 2015. 2, 136

[2] F. Bannour, S. Souihi, and A. Mellouk. Distributed SDN control: Survey, taxonomy, and challenges. *IEEE Communications Surveys Tutorials*, 20(1):333–354, Firstquarter 2018. 2, 136

[3] Nancy Samaan and Ahmed Karmouch. Towards autonomic network management: an analysis of current and future research directions. *IEEE Communications Surveys and Tutorials*, 11(3):22–36, 2009. 2, 136

[4] W. Ren, Y. Sun, H. Luo, and M. Guizani. A Novel Control Plane Optimization Strategy for Important Nodes in SDN-IoT Networks. *IEEE Internet of Things Journal*, 6(2):3558–3571, April 2019. 2, 3

[5] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to SDN: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, April 2014. 3, 137

[6] Y. Li, X. Su, J. Riekki, T. Kanter, and R. Rahmani. A SDN-based architecture for horizontal internet of things services. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–7, May 2016. 3, 137

[7] Marco Canini, Daniele De Cicco, Petr Kuznetsov, Dan Levin, Stefan Schmid, and Stefano Vissicchio. STN: A robust and distributed SDN control plane. Open Networking Summit (ONS) Research track, March 2014. 3, 51, 137

[8] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, 2013. 10

[9] Md. Faizul Bari, Raouf Boutaba, Rafael Pereira Esteves, Lisandro Zambenedetti Granville, Maxim Podlesny, Md. Golam Rabbani, Qi Zhang, and Mohamed Faten Zhani. Data center network virtualization: A survey. *IEEE Communications Surveys and Tutorials*, 15(2):909–928, 2013. 10

[10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *J. Internet Services and Applications*, 1(1):7–18, 2010. 10

[11] M. Abu Sharkh, M. Jammal, A. Shami, and A. Ouda. Resource allocation in a network-based cloud computing environment: design challenges. *IEEE Communications Magazine*, 51(11):46–52, November 2013. 10

[12] Ansible. https://www.ansible.com/. Accessed: 2017-04-11. 11

[13] ONF. Open Networking Foundation. https://www.opennetworking.org/. Accessed: 2016-05-19. 11, 13

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. 13, 16, 41

[15] A. Doria, J. Hadi Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern. Forwarding and Control Element Separation (ForCES) Protocol Specification, March 2010. 13

[16] ONF. OpenFlow switch specification. Technical report, Open Networking Foundation, December 2009. Accessed: 2016-01-27. 13

[17] Pingping Lin, Jun Bi, Stephen Wolff, Yangyang Wang, Anmin Xu, Ze Chen, Hongyu Hu, and Yikai Lin. A west-east bridge based SDN inter-domain testbed. *IEEE Communications Magazine*, 53(2):190–197, 2015. 14, 35

[18] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM. 16

[19] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM. 16

[20] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association. 16

[21] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2):136–141, February 2013. 17

[22] S. Azodolmolky, P. Wieder, and R. Yahyapour. Performance evaluation of a scalable software-defined networking deployment. In *2013 Second European Workshop on Software Defined Networks*, pages 68–74, Oct 2013. 17

[23] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM. 17, 18

[24] O. Michel and E. Keller. Sdn in wide-area networks: A survey. In *2017 Fourth International Conference on Software Defined Systems (SDS)*, pages 37–42, May 2017. 18

[25] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 7–12, New York, NY, USA, 2012. ACM. 18, 43

[26] M. T. I. ul Huque, W. Si, G. Jourjon, and V. Gramoli. Large-scale dynamic controller placement. *IEEE Transactions on Network and Service Management*, 14(1):63–76, March 2017. 18

[27] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. Advanced study of sdn/openflow controllers. In *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, CEE-SECR '13, pages 1:1–1:6, New York, NY, USA, 2013. ACM. 18

[28] Murat Karakus and Arjan Durresi. A survey: Control plane scalability issues and approaches in software-defined networking (sdn). *Computer Networks*, 112:279 – 293, 2017. 18, 40

[29] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008. 18, 38, 39

[30] David Erickson. The beacon openflow controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM. 18

[31] Floodlight Project. Accessed: 2015-12-07. 18, 21, 32, 38, 39

[32] Arsalan Tavakoli, Martin Casado, Teemu Koponen, and Scott Shenker. Applying nox to the datacenter. In *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009. 18

[33] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA, 2012. 18

[34] Pox. http://www.noxrepo.org/pox/about-pox/. Accessed: 2015-111-22. 18, 38, 39

[35] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. Sphinx: Detecting security attacks in software-defined networks. In *NDSS*. The Internet Society, 2015. 18

[36] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. Onos: Towards an open, distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM. 18, 19, 20, 29, 36, 38, 39, 44, 45

[37] Kevin Phemius, Mathieu Bouet, and Jeremie Leguay. DISCO: distributed multi-domain SDN controllers. *CoRR*, abs/1308.6138, 2013. 18, 32, 35, 39

[38] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu Google, Rajiv Ramanathan, Yuichiro Iwata NEC, Hiroaki Inoue NEC, Takayuki Hama NEC, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *9th Conference on Operating Systems Design and Implementation*, pages 351–364, 2010. 19, 20, 23, 25, 38, 39, 44, 45, 79, 96

[39] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. 19, 20, 27, 38, 39, 44, 45

[40] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Morris. Flexible, wide-area storage for distributed systems with wheelfs. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 43–58, 2009. 20, 27

[41] Andrea Bianco, Paolo Giaccone, Samuele De Domenico, and Tianzhu Zhang. The role of inter-controller traffic for placement of distributed SDN controllers. *CoRR*, abs/1605.09268, 2016. 20, 30

[42] Balakrishnan Chandrasekaran and Theophilus Benson. Tolerating sdn application failures with legosdn. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 22:1–22:7, 2014. 20

[43] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 78–89, 2014.

[44] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 4:1–4:12, 2015. 21, 27, 28, 39, 44

[45] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Simple distributed programming in software-defined networks. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 4:1–4:12, 2016.

[46] Balakrishnan Chandrasekaran, Brendan Tschaen, and Theophilus Benson. Isolating and tolerating sdn application failures with legosdn. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 7:1–7:12, 2016.

[47] E. S. Spalla, D. R. Mafioletti, A. B. Liberato, G. Ewald, C. E. Rothenberg, L. Camargos, R. S. Villaca, and M. Martinello. Ar2c2: Actively replicated controllers for sdn resilient control plane. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 189–196, April 2016. 20, 44, 45

[48] F. Botelho, A. Bessani, F. M. V. Ramos, and P. Ferreira. On the design of practical fault-tolerant sdn controllers. In *2014 Third European Workshop on Software Defined Networks*, pages 73–78, Sept 2014. 21, 25, 39, 44

[49] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, 2014. 21

[50] Ryu SDN framework. https://osrg.github.io/ryu/. Accessed: 2016-07-10. 22, 35

[51] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, 2010. 22

[52] Y. Liu, A. Hecker, R. Guerzoni, Z. Despotovic, and S. Beker. On optimal hierarchical sdn. In *2015 IEEE International Conference on Communications (ICC)*, pages 5374–5379, June 2015. 22

[53] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM. 22, 23, 29, 38, 39, 41

[54] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM. 22, 38, 39, 41

[55] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with difane. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 351–362, New York, NY, USA, 2010. ACM. 22, 38, 39, 41

[56] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM. 22, 23, 24, 31, 39

[57] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 1–14, 2015. 22

[58] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. 23

[59] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the edge off with

espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 432–445, New York, NY, USA, 2017. ACM. 24

[60] Redouane Benaini Fouad Benamrane, Mouad Ben mamoun. Performances of openflow-based softwaredefined networks: An overview. *Journal of Networks*, 10(6):329–337, 2015. 25, 33

[61] Brian Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, August 1988. 28

[62] André Mantas and Fernando M. V. Ramos. Consistent and fault-tolerant SDN with unmodified switches. *CoRR*, abs/1602.04211, 2016. 28

[63] A. Bondkovskii, J. Keeney, S. van der Meer, and S. Weber. Qualitative comparison of open-source SDN controllers. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 889–894, April 2016. 29

[64] Opendaylight project. Accessed: 2016.-01-05. 29, 30, 38, 39, 79, 80

[65] A. S. Muqaddas, A. Bianco, P. Giaccone, and G. Maier. Inter-controller traffic in ONOS clusters for SDN networks. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2016. 29

[66] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association. 30, 31, 80, 82, 83

[67] Akka framework. http://akka.io/. Accessed: 2017-02-15. 31

[68] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 15–26, New York, NY, USA, 2013. ACM. 31

[69] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association. 32

[70] Christian Esteve Rothenberg, Marcelo Ribeiro Nascimento, Marcos Rogerio Salvador, Carlos Nilton Araujo Corrêa, Sidney Cunha de Lucena, and Robert Raszuk. Revisiting routing control platforms with the eyes and muscles of software-defined networking. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 13–18, New York, NY, USA, 2012. ACM. 32

[71] AMQP. http://www.amqp.org/. Accessed: 2016-01-05. 33

[72] David D. Clark, Craig Partridge, J. Christopher Ramming, and John T. Wroclawski. A knowledge plane for the internet. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 3–10, New York, NY, USA, 2003. ACM. 33

[73] Mateus A. S. Santos, Bruno Astuto A. Nunes, Katia Obraczka, Thierry Turletti, Bruno Trevizan de Oliveira, and Cintia B. Margi. Decentralizing sdn's control plane. In *IEEE 39th Conference on Local Computer Networks, LCN 2014, Edmonton, AB, Canada, 8-11 September, 2014*, pages 402–405, 2014. 33

[74] Jonathan Philip Stringer, Dean Pemberton, Qiang Fu, Christopher Lorier, Richard Nelson, Josh Bailey, Carlos N. A. Corrêa, and Christian Esteve Rothenberg. Cardigan: SDN distributed routing fabric going live at an internet exchange. In *IEEE Symposium on Computers and Communications, ISCC 2014, Funchal, Madeira, Portugal, June 23-26, 2014*, pages 1–7, 2014. 34, 35

[75] Arpit Gupta, Muhammad Shahbaz, Laurent Vanbever, Hyojoon Kim, Russ Clark, Nick Feamster, Jennifer Rexford, and Scott Shenker. Sdx: A software defined internet exchange. *ACM SIGCOMM*, 2014. 34, 35, 39

[76] Remy Lapeyrade, Marc Bruyere, and Philippe Owezarski. Openflow-based migration and management of the TouIX IXP. In *2016 IEEE/IFIP Network Operations and*

Management Symposium, NOMS 2016, Istanbul, Turkey, April 25-29, 2016, pages 1131–1136, 2016. 34

[77] Endeavour project. https://www.h2020-endeavour.eu/. Accessed: 2017-01-02. 34

[78] Heidi Morgan. AtlanticWave-SDX: A Distributed Intercontinental Experimental Software Defined Exchange for Research and Education Networking. Press Release, April 2015. 34

[79] Arpit Gupta, Robert MacDavid, Rudiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. An industrial-scale software defined internet exchange point. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 1–14, Santa Clara, CA, March 2016. USENIX Association. 35

[80] Vasileios Kotronis, Adrian Gämperli, and Xenofontas Dimitropoulos. Routing Centralization Across Domains via SDN. *Comput. Netw.*, 92(P2):227–239, December 2015. 35

[81] J. Chung, J. Cox, J. Ibarra, J. Bezerra, H. Morgan, R. Clark, and H. Owen. AtlanticWave-SDX: An International SDX to Support Science Data Applications. In *Software Defined Networking (SDN) for Scientific Networking Workshop*, Austin, Texas, 11 2015. 35

[82] Internet2. Advanced layer 2 system. https://www.internet2.edu/products-services/advanced-networking/layer-2-services/. Accessed: 2017-10-09. 35

[83] J. Chung, H. Owen, and R. Clark. Sdx architectures: A qualitative analysis. In *SoutheastCon 2016*, pages 1–8, March 2016. 36

[84] Manar Jammal, Taranpreet Singh, Abdallah Shami, Rasool Asal, and Yiming Li. Software defined networking: State of the art and research challenges. *Computer Networks*, 72:74 – 98, 2014. 40

[85] O. Hohlfeld, J. Kempf, M. Reisslein, S. Schmid, and N. Shah. Guest editorial scalability issues and solutions for software defined networks. *IEEE Journal on Selected Areas in Communications*, 36(12):2595–2602, Dec 2018. 41, 42

[86] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, April 2014. 41

[87] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. 41

[88] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 377–378, 2014. 41

[89] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, 2016.

[90] Roberto Bifulco and Gábor Rétvári. A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems. *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–7, 2018. 41

[91] S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, and C. Cascone. Stateful OpenFlow: Hardware proof of concept. In *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8, July 2015. 42

[92] Giuseppe Bianchi, Marco Bonola, Salvatore Pontarelli, Davide Sanvito, Antonio Capone, and Carmelo Cascone. Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing. *CoRR*, abs/1605.01977, 2016. 42

[93] Roberto Bifulco, Julien Boite, Mathieu Bouet, and Fabian Schneider. Improving SDN with InSPired Switches. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 11:1–11:12, 2016. 42

[94] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(5):81–94, October 2007. 42

[95] Vimalkumar Jeyakumar, Mohammad Alizadeh, Changhoon Kim, and David Mazières. Tiny packet programs for low-latency network control and monitoring. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 8:1–8:7, 2013. 42

[96] J. Yang, X. Yang, Z. Zhou, X. Wu, T. Benson, and C. Hu. Focus: Function offloading from a controller to utilize switch power. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 199–205, Nov 2016. 42

[97] K. Qiu, S. Huang, Q. Xu, J. Zhao, X. Wang, and S. Secci. Paracon: A parallel control plane for scaling up path computation in sdn. *IEEE Transactions on Network and Service Management*, PP(99):1–1, Oct 2017. 43

[98] Shadi Moazzeni, Mohammad Reza Khayyambashi, Naser Movahhedinia, and Franco Callegati. On reliability improvement of software-defined networks. *Computer Networks*, 133:195 – 211, 2018. 43

[99] P. Fonseca and E. Mota. A survey on fault management in software-defined networks. *IEEE Communications Surveys Tutorials*, PP(99):1–1, 2017. 44

[100] P. Fonseca, R. Bennesby, E. Mota, and A. Passito. Resilience of sdns based on active and passive replication mechanisms. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 2188–2193, Dec 2013. 44

[101] Volkan Yazici, M. Oguz Sunay, and Ali Ozer Ercan. Controlling a software-defined network via distributed controllers. *CoRR*, abs/1401.7651, 2014. 45

[102] Nathan Kong. Design concept for a failover mechanism in distributed sdn controllers. In *Master's Project*, 2017. 45

[103] V. Pashkov, A. Shalimov, and R. Smeliansky. Controller failover for sdn enterprise networks. In *2014 International Science and Technology Conference (Modern Networking Technologies) (MoNeTeC)*, pages 1–6, Oct 2014. 45

[104] M. Obadia, M. Bouet, J. Leguay, K. Phemius, and L. Iannone. Failover mechanisms for distributed sdn controllers. In *2014 International Conference and Workshop on the Network of the Future (NOF)*, volume Workshop, pages 1–6, Dec 2014. 46

[105] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. In-band synchronization for distributed sdn control planes. *SIGCOMM Comput. Commun. Rev.*, 46(1):37–43, January 2016. 46

[106] Fabio Botelho, Tulio A. Ribeiro, Paulo Ferreira, Fernando M. V. Ramos, Alysson Bessani, undefined, undefined, undefined, and undefined. Design and implementation of a consistent data store for a distributed sdn control plane. *2016 12th European Dependable Computing Conference (EDCC)*, 00:169–180, 2016.

[107] B. Zhang, X. Wang, and M. Huang. Adaptive consistency strategy of multiple controllers in SDN. *IEEE Access*, 6:78640–78649, 2018. 46, 48

[108] Aurojit Panda, Colin Scott, Ali Ghodsi, Teemu Koponen, and Scott Shenker. CAP for networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 91–96, New York, NY, USA, 2013. ACM. 46, 47, 81

[109] Oracle. https://www.oracle.com. Accessed: 2016-10-24. 47

[110] MySQL. http://www.mysql.fr/. Accessed: 2016-03-02. 47

[111] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. 47, 81, 82, 83, 96, 98, 99

[112] Rusty Klophaus. Riak Core: Building Distributed Applications Without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM. 47, 99

[113] Swaminathan Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM. 47, 81, 82, 83, 96, 98, 99

[114] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 1–6, 2012. 47

[115] M. Aslan and A. Matrawy. Adaptive consistency for distributed sdn controllers. In *2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks)*, pages 150–157, Sept 2016. 48, 96

[116] E. Sakic, F. Sardis, J. W. Guck, and W. Kellerer. Towards adaptive state consistency in distributed SDN control plane. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7, May 2017. 48, 96

[117] Stefan Wallin and Claes Wikström. Automating network and service configuration using netconf and yang. In *Proceedings of the 25th International Conference on Large Installation System Administration*, LISA'11, pages 22–22, Berkeley, CA, USA, 2011. 49

[118] OpenConfig. http://www.openconfig.net/. Accessed: 2016-11-27. 49

[119] OF-CONFIG 1.2: Openflow Management and Configuration Protocol. Technical report, Open Networking Foundation, 2014. Accessed: 2017-01-05. 49

[120] R. Amin, M. Reisslein, and N. Shah. Hybrid SDN Networks: A Survey of Existing Approaches. *IEEE Communications Surveys Tutorials*, 20(4):3259–3306, Fourthquarter 2018. 50

[121] Sandhya, Yash Sinha, and K. Haribabu. A survey: Hybrid SDN. *Journal of Network and Computer Applications*, 100:35 – 55, 2017. 50

[122] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer. Survey on network virtualization hypervisors for software defined networking. *IEEE Communications Surveys Tutorials*, 18(1):655–685, Firstquarter 2016. 51

[123] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust SDN control plane for transactional network updates. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 190–198, April 2015. 51

[124] F. Bannour, S. Souihi, and A. Mellouk. Scalability and reliability aware SDN controller placement strategies. In *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–4, Nov 2017. 54

[125] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 7–12, New York, NY, USA, 2012. ACM. 54, 58

[126] David Hock, Matthias Hartmann, Steffen Gebert, Michael Jarschel, Thomas Zinner, and Phuoc Tran-Gia. Pareto-optimal resilient controller placement in sdn-based core networks. In *25th International Teletraffic Congress (ITC)*, Shanghai, China, 9 2013. 55, 60, 65

[127] Stanislav Lange, Steffen Gebert, Thomas Zinner, Phuoc Tran-Gia, David Hock, Michael Jarschel, and Marco Hoffmann. Heuristic approaches to the controller placement problem in large scale sdn networks. *IEEE Transactions on Network and Service Management*, 12(1):4–17, 2015. 55, 56, 59

[128] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011. 55, 56

[129] Yannan Hu, Wendong Wang, Xiangyang Gong, Xirong Que, and Shiduan Cheng. Reliability-aware controller placement for software-defined networks. In Filip De Turck, Yixin Diao, Choong Seon Hong, Deep Medhi, and Ramin Sadre, editors, *IM*, pages 672–675. IEEE, 2013. 55

[130] Member IEEE Yuliang Li Guang Yao, Jun Bi and Luyi Guo. On the capacitated controller placement problem in software defined networks. In *IEEE Communications Letters*, 2014. 56, 60

[131] S. Lange, S. Gebert, J. Spoerhase, P. Rygielski, T. Zinner, S. Kounev, and P. Tran-Gia. Specialized heuristics for the controller placement problem in large scale sdn networks. In *2015 27th International Teletraffic Congress*, pages 210–218, Sept 2015. 56

[132] V. Ahmadi, A. Jalili, S. M. Khorramizadeh, and M. Keshtgari. A hybrid nsga-ii for solving multiobjective controller placement in sdn. In *The 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pages 663–669, Nov 2015. 56, 64

[133] Ahmad Jalili, Manijeh Keshtgari, and Reza Akbari. Optimal Controller Placement in Large Scale Software Defined Networks Based on Modified NSGA-II. *Applied Intelligence*, 48(9), September 2018. 56, 57

[134] L. F. Müller, R. R. Oliveira, M. C. Luizelli, L. P. Gaspary, and M. P. Barcellos. Survivor: An enhanced controller placement strategy for improving SDN survivability. In *2014 IEEE Global Communications Conference*, pages 1909–1915, Dec 2014. 56, 57

[135] Francisco J. Ros and Pedro M. Ruiz. On reliable controller placements in software-defined networks. *Comput. Commun.*, 77(C):41–51, March 2016. 57

[136] J. M. Sanner, Y. Hadjadj-Aoufi, M. Ouzzif, and G. Rubino. Hierarchical clustering for an efficient controllers' placement in software defined networks. In *2016 Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–7, Oct 2016. 57

[137] N. Perrot and T. Reynaud. Optimal placement of controllers in a resilient sdn architecture. In *The 12th International Conference on the Design of Reliable Communication Networks (DRCN)*, pages 145–151, March 2016. 57

[138] G. Wang, Y. Zhao, J. Huang, and Y. Wu. An Effective Approach to Controller Placement in Software Defined Wide Area Networks. *IEEE Transactions on Network and Service Management*, 15(1):344–355, March 2018. 57

[139] IBM ILOG CPLEX Optimizer. http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud. Accessed: 2017-06-05. 57

[140] T. Das, V. Sridharan, and M. Gurusamy. A survey on controller placement in SDN. *IEEE Communications Surveys Tutorials*, pages 1–1, 2019. 57

[141] Bala Prakasa Rao Killi and Seela Veerabhadreswara Rao. Controller placement in software defined networks: A comprehensive survey. *Computer Networks*, 163:106883, 2019. 59

[142] Neha Soni; Amit Ganatra. Comparative study of several clustering algorithms. *International Journal of Advanced Computer Research*, pages 37–42, 2012. 63

[143] Sinalgo - simulator for network algorithms. http://www.disco.ethz.ch/projects/sinalgo/. Accessed: 2017-02-01. 66

[144] Tianzhu Zhang, Andrea Bianco, and Paolo Giaccone. The role of inter-controller traffic in SDN controllers placement. In *IEEE Conference on Network Function Virtualization and Software Defined Networks (IEEE NFV-SDN)*, 2016. 73

[145] K.Popat Shraddha and M. Emmanuel. Review and comparative study of clustering techniques. *International Journal of Computer Science and Information Technology (IJCSIT)*, 5:805–812, 2014. 75

[146] OVS - Open vSwitch. http://www.openvswitch.org/. Accessed: 2017-01-05. 76

[147] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. 78, 83, 85, 98, 99, 101

[148] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, pages 7:1–7:6, 2011. 79

[149] Thanh Dang Nguyen, Marco Chiesa, and Marco Canini. Decentralized consistent updates in SDN. In *Proceedings of the Symposium on SDN Research*, pages 21–33, 2017. 79

[150] ONOS. https://onosproject.org/. Accessed: 2016-01-02. 79, 83, 96

[151] M. Aslan and A. Matrawy. Adaptive consistency for distributed SDN controllers. In *2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks)*, pages 150–157, Sept 2016. 80, 82, 83

[152] E. Sakic, F. Sardis, J. W. Guck, and W. Kellerer. Towards adaptive state consistency in distributed SDN control plane. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7, May 2017. 80

[153] Ermin Sakic, Nemanja Đerić, and Wolfgang Kellerer. Morph: An adaptive framework for efficient and byzantine fault-tolerant sdn control plane. *IEEE Journal on Selected Areas in Communications*, 36:2158–2174, 2018. 80, 82

[154] M. Aslan and A. Matrawy. A clustering-based consistency adaptation strategy for distributed SDN controllers. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 441–448, June 2018. 80

[155] D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45(2):37–42, Feb 2012. 81

[156] Sathiya Prabhu Kumar. *Adaptive Consistency Protocols for Replicated Data in Modern Storage Systems with a High Degree of Elasticity*. Theses, CNAM, March 2016. 83, 99, 100

[157] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Conference on Operating Systems Design and Implementation*, 2012. 83

[158] Houssem-Eddine Chihoub, María Pérez, Gabriel Antoniu, and Luc Bougé. Chameleon:customized application-specific consistency by means of behavior modeling. Research report, 2013. 83

[159] ODL. http://opendaylight.org/. Accessed: 2010-09-30. 96

[160] Mohamed Aslan and Ashraf Matrawy. A clustering-based consistency adaptation strategy for distributed SDN controllers. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018. 96

[161] Ermin Sakic and Wolfgang Kellerer. Impact of adaptive consistency on distributed SDN applications: An empirical study. *IEEE Journal on Selected Areas in Communications*, page 13, 2018.

[162] F. Bannour, S. Souihi, and A. Mellouk. Adaptive state consistency for distributed ONOS controllers. In *2018 IEEE Global Communications Conference(Globecom)*, pages 1–7, 2018. 96, 97

[163] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, 2013. 96

[164] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012. 98, 101, 102, 107

[165] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying Eventual Consistency with PBS. *Commun. ACM*, 57(8):93–102, August 2014. 98, 101

[166] H. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Pérez. Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage. In *2012 IEEE International Conference on Cluster Computing*, pages 293–301, Sept 2012. 98

[167] Canh Son Nguyen Ba. *Adaptive control for availability and consistency in distributed key-values stores*. PhD thesis, University of Illinois, 2015. 99

[168] Voldemort project. http://www.project-voldemort.com/voldemort/design.html. Accessed: 2016-04-11. 99

[169] Jing Zhong, Roy D. Yates, and Emina Soljanin. Minimizing content staleness in dynamo-style replicated storage systems. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 361–366, 2018. 102

[170] Maria Couceiro, Gayana Chandrasekara, Manuel Bravo, Matti Hiltunen, Paolo Romano, and Luís Rodrigues. Q-OPT: Self-tuning quorum system for strongly consistent software defined storage. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 88–99, 2015. 104

[171] Abdelhamid Mellouk, Said Hoceini, and Hai Anh Tran. *Quality of Experience*, chapter 2, pages 11–31. John Wiley Sons, Ltd, 2013. 105

[172] OpenAI Gym Project. https://gym.openai.com/. Accessed: 2019-04-05. 118

[173] Hai-Anh Tran, Sami Souihi, Duc A. Tran, and Abdelhamid Mellouk. Mabrese: A new server selection method for smart SDN-based CDN architecture. *IEEE Communications Letters*, 23:1012–1015, 2019. 119

[174] M. Ojo, D. Adami, and S. Giordano. A SDN-IoT architecture with NFV implementation. In *2016 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6, Dec 2016. 130, 142

[175] K. Liang, L. Zhao, X. Chu, and H. H. Chen. An integrated architecture for software defined and virtualized radio access networks with fog computing. *IEEE Network*, 31(1):80–87, January 2017. 130, 142

[176] A. Abdou, P. C. van Oorschot, and T. Wan. Comparative analysis of control plane security of sdn and conventional networks. *IEEE Communications Surveys Tutorials*, 20(4):3542–3559, Fourthquarter 2018. 134