UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS IN

INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

IN-PRODUCTION CONTINUOUS TESTING FOR FUTURE TELCO CLOUD

UGO GIORDANO

TUTOR: PROF. STEFANO RUSSO

CO-TUTOR:
DR. MARINA THOTTAN, NOKIA BELL LABS
DR. CATELLO DI MARTINO, NOKIA BELL LABS

XXIX CICLO

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

IN-PRODUCTION CONTINUOUS TESTING FOR FUTURE TELCO CLOUD

By
Ugo Giordano

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
"FEDERICO II" UNIVERSITY OF NAPLES
VIA CLAUDIO 21, 80125 – NAPOLI, ITALY
OCTOBER 2017

"To my grandfather Ugo, the best man I have ever meet in my life."

Table of Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interface.

ASIC Application-Specific Integrated Circuit.

BER Bit Error Rate.

CAPEX CAPital EXpenditure.

CM Campaign Manager.

CPU Central Processing Unit.

CRC Cyclic Redundancy Check.

DC Data Collector.

DDoS Distributed Denial of Service.

DOE Design Of Experiments.

EJB Enterprise Java Beans.

EM Experiment Manager.

FEC Forward Error Correction.

FI Failure Injection, Failure Injector.

FIT Failure Injection Testing.

IBN Intent-Based Networking.

IPC Inter-Process Communication.

IPS Intents operations per unit time.

ISP Internet Service Provider.

JMS Java Message Service.

JMX Java Management Extensions.

JSE Java Standard Edition.

JVM Java Virtual Machine.

LG Load Generator.

MTBF Mean Time Between Failures.

MTTF Mean Time To Failure.

MTTR Mean Time To Repair.

NBI Northbound Interface.

NFV Network Function Virtualization.

NFVI Network Function Virtualization Infrastructure.

NOS Network Operating System.

ODL OpenDayLight (registered trademark).

ONF Open Network Foundation.

ONOS Open Network Operating System (registered trademark).

OPEX OPerating EXpenditure.

OS Operating System.

OSGi Open Service Gateway Initiative.

Pub/Sub Publish/Subscribe.

REST Representational State Transfer.

RMI Remote Method Invocation.

SBI Southbound Interface.

SCP-CLUB SDN Control PlaneCLoUd-based Benchmarking.

SDN Software-Defined Networking, Software-Defined Networks.

SFI Software Fault Injection.

SLA Service Level Agreement.

SUT System Under Test.

TC Telco Cloud.

TCSP Telco Cloud Service Provider.

TM Topology Manager.

VM Virtual Machine.

VNF Virtualized Network Function.

WG Workload Generator.

Acknowledgements

I would like to thank my advisor, Prof. Stefano Russo. I'm really thankful for all the tips and advice he gave me during these three years, and for having supported me in pursue the PhD.
Much of the work for this dissertation has been performed during a period at the prestigious NOKIA Bell Labs in Murray Hill, New Jersey, USA, under the supervision of Dr. Marina Thottan and Dr. Catello Di Martino. I am very grateful to NOKIA for the opportunity to work in a top-level research environment. I would like to thank specifically Dr. Thottan and Dr. Di Martino for the inception of the ideas underlying the SCP-CLUB benchmarking framework and the failure injector architecture, the useful discussions, their patient support and their compelling stimuli. Working with them at NOKIA Bell Labs has been as challenging as exciting.
I am also grateful to all friends (because we are friends, not just colleges) of the DEpendable Systems and Software Engineering Research Team (DESSERT) at DIETI, Federico II University, and at the CINI "Carlo Savy" laboratory in Napoli. Thanks to Stefano, Luigi, Flavio, Antonio, Roberto P., Roberto N, Mario, Anna and Alma. A special thank goes to Raffaele, Fabio and Salvatore (vaiiii vaiii!!!). They always supported me during the hard time of this PhD, and sharing a beer with them has always been a pleasure. Really thanks guys, hoping to have been a good friend and supporter for you too.

Thanks to my father, my mother, my bro, and my sisters, and my grandchildren Giulia and Michele (they always made you smile in all situations),

to my family; because without their support I would never be able to accomplish this last step. Thanks for all!
Thanks to my Bernardo and Giovanna, my second family. Thanks for letting me to be part of your family.
Thanks to YOU, Rosaria, my best friend, my greatest support, my biggest comfort, my strongest motivation, my truest smile, my deepest love, my favorite, my forever. Had it not been for her, I might still be in darkness, or I would have invented her. Sincerely, today, I don't know where I would be if she wasn't been on my side, where she has always been. There, close to me, for me. I will love you until death do us part!!! I'll always love you, as if it were the first day, as if there was not a tomorrow (I love you).

Naples, Italy — Ugo Giordano
October 4th, 2017

Disclaimer: The views and opinions expressed in this dissertation are those of the author and do not necessarily reflect the official policy or position of any person, organization or company other than the author.

Chapter 1

Introduction

1.1 Context

Computer networks are nowadays at the basis of most critical infrastructures, and of many services we access in our daily activities - be they business, consumer, social or private. **Software Defined Networking** (SDN) has emerged in the very last few years - from the initial work done at University of California at Berkeley and Stanford University in 2008 - as a paradigm capable of providing new ways to design, build and operate networks. This is due to the key concept underlying it, namely the separation of the network control logic (the so-called **control plane**) from the underlying equipment (such as routers and switches) that forward and transport the traffic (the **data plane**) [1].

Thanks to the clear separation of the two *abstraction levels* - the logic level, corresponding to the control plane, and the physical one, i.e. the data plane - SDN is claimed, and by many experts strongly believed, to be about to introduce a big revolution in computer networking [2]. Along with Network Function Virtualization (NFV), SDN is expected to have a positive

impact on network management costs [3]. Indeed, the logical level may host the network control logic in a *programmable* and *highly flexible* way: advanced network services become software defined, supporting much easier enforcement of networking policies, security mechanisms, reconfigurability and evolution than in current computer networks.

The SDN flexibility is due to the *separation of concerns* between network configuration and policies definition and lower-level equipments for traffic switching and routing, a direct consequence of the separation of abstraction layers. The many advantages promised by SDN in engineering and managing computer networks and in operating their services are very attractive for network operators and Internet Services Providers (ISP). Network operation and management are challenging, and providers face big issues in configuring large networks, enforcing desired policies, and evolving to new technologies - all in a very dynamic environment [4]. This is easily comprehensible thinking, for instance, at the huge difficulties that major technological changes encounter to be applied in large networks. The transition from the Internet network protocols IPV4 to IPV6 is just an example: started about a decade ago, it is probably still far to be completed. And it has to be considered that protocols, which are at the heart of computer networks, are basic blocks from the point of view of the highly demanding modern and future fixed and mobile applications and services.

According to Allied Market Research, the SDN market is expected to reach $132 billion by 2022 [5]. Players in this market include telecommunication operators, ISPs, cloud and data center providers, and equipment manufacturers. Beside the decoupling of service, software and hardware

technology innovations in networking, there is probably a fundamental reason for such big expectation raised in the networking industry. The history of major advances in computer science and engineering is a history of raising the level of abstraction. This is true for instance for programming languages, for operating systems and middleware technologies, for software design (up to modern model-driven techniques) [6]. Abstraction and separation of concerns are fundamental engineering principles, which in the case of SDN may well support its wide spread.

The logical entity hosting software-defined core network services in the control plane (e.g. routing, authentication, discovery) is typically known in the literature as **SDN controller** (or simply controller). Very recently, the concept of controller has evolved to that of **network operating system** (NOS), an operating system - which can possibly run on commodity hardware - specifically providing an execution environment for network management applications, through programmable network functions. In the logical SDN architecture, the controller is below the application layer[1], and atop the data plane, that it controls enacting the policies and the services required by applications. The separation of the planes is realized by means of well-defined application programming interfaces (API) between them. Relevant examples of SDN controllers are NOX [7], Beacon [8], OpenDaylight [9] and ONOS® [10], while probably the most widely known API is OpenFlow [11].

[1]The applications atop the control plane are actually management programs accessing the controller programming interface to request network services or to enforce policies.

1.2 Motivations and contributions

Today's networks will need to adopt a new approach to support the predicted growth in scale, diversity and complexity of use cases [12]. With new services and applications emerging continuously, devices will be connected much more often, and consequently, a distinct competitive market advantage could be created by those network operators capable of implementing new services rapidly.

In order to meet evolving market demands, and increase the flexibility and agility of networks promoting innovative solutions for future network services, telco companies look with interest at migrating towards the emerging ***Telco Cloud*** (TC) paradigm [13] [12] [14], becoming so-called ***Telco Cloud Service Providers*** (TCSPs). Telco Cloud is meant to provide a dedicated cloud computing solution for a network operator, to shift network functions away from dedicated legacy hardware platforms into virtualized software components deployable on general-purpose hardware. This logically allocates cloud and networking capabilities into a multi-service programmable fabric built to precisely meet each of the different service requirements, changing network conditions, unpredictable traffic patterns, continuous streams of apps and services and short innovation cycles.

This ability to focus on what is needed is achieved through the combined use of **Software Defined Networking** (SDN) [1], **Network Function Virtualization** (NFV), and cloud technologies [15], [16] in telco cloud infrastructures.

While the *softwarization* and *cloudification* of the network provides unparalleled level of automation and flexibility, and a drastic reduction of the

network operating margins, it also presents significant challenges, mainly due to the variety of knobs (e.g., SDN-, cloud-, and software- related parameters) to properly fine tune in [16] order to obtain specific levels of service.

In the SDN world, performance it is not only related to the behaviour of the data plane. As the separation of control plane and data plane makes the latter significantly more agile, it lays off all the complex processing workload to the control plane. This is further exacerbated in distributed network controller (e.g., ODL [9], and ONOS [10]), where the control plane is additionally loaded with the state synchronization overhead. Misconfiguration of the control plane can negatively impact the overall network performance, cause customer insatisfaction and, in more extreme cases, network unavailability. Understanding the performance of the SDN control plane in a telco cloud and the factors that influence it is fundamental for planning, sizing and tuning SDN deployments.

Furthermore, the introduction of SDNs technologies has raised advanced challenges in achieving **failure resilience**, meant as *the persistence of service delivery that can justifiably be trusted, when facing changes* [17], and **fault tolerance**, meant as *the ability to avoid service failures in the presence of faults* [18] (these definitions will be used hereafter to refer to the resilience and fault-tolerance of SDN technologies). The decoupling of the control plane from the data plane leads the dependency of the overall network resilience on the fault-tolerance in the data plane, as in the traditional networks, but also on the capability of the (logically) centralized control

functions to be resilient to faults. Moreover, the SDNs are by nature suitable to be implemented as distributed system, introducing further threats to the network resilience, such as inconsistent global network state shared between the SDN controllers, as well as a network partitioning. In addition, compared to the legacy network appliances, which rely on dedicated high-performance hardware, the adoption of technologies for virtualizing network services, introduces performance and reliability concerns, e.g., high overhead/latency and failures, due to new failure scenarios which periodically occur in data center [19] [20].

Consequently, as the controllers technology develops and progressively becomes mature for the market, the need to engineer and to assess the compliance of SDN solutions with non functional requirements – such as scalability, high availability, fault tolerance and high resilience – becomes more compelling. In such a context, the **traditional software testing techniques appear insufficient to evaluate the resilience and availability of a distributed SDN ecosystems**. Indeed, although these techniques are useful to validate specific system behaviours (e.g. the functional testing), full operational testing may be possible only in production, due to the impossibility to reproduce the entire ecosystem in a testing environment. Ultimately, even if a system can be reproduced in a test context, it is impractical, or even impossible, to fully reproduce all aspects and failure modes that can characterize complex distributed systems during **production hours** [21]. On the other hand, a widely recognized effective way to assess fault-tolerance mechanisms as well as to quantify system availability and/or reliability is *failure injection*. Failure injection allows to assess fault

tolerance mechanisms by reproducing multiple failure scenarios, such as a latent communication, service failure, or hardware transient faults. Furthermore, if applied in a controlled environment while the **system is in production**, the *failure injection* can lead to discover problems in a timely manner, without affecting the customers, and providing helpful insights to build better detection, and mitigation mechanisms to recover the system when real issues arise.

Therefore, along with the "*softwarization*" of network services, it is an important goal in the engineering of such services, e.g. SDNs and NFVs, to be able to test and assess the proper functioning not only in emulated conditions before release and deployment, but also **in-production** [22], when the system is under real operating conditions.

The goal of this thesis is to devise an approach to evaluate not only the performance, but also the effectiveness of the failure detection, and mitigation mechanisms provided by SDN controllers, as well as the capability of the SDNs to ultimately satisfy non functional requirements, especially resiliency, availability, and reliability. The approach consists of exploiting *benchmarking* techniques, such as the *failure injection*, to get **continuously** feedback on the performance as well as capabilities of the SDN services to survive failures, which is of paramount importance to improve the effectiveness of the system internal mechanisms in reacting to anomalous situations potentially occurring in operation, while its services are regularly updated or improved.

To the best of our knowledge, there is no available approach or tool that can be used to provide automation in the analysis of the SDN control

plane. The literature on SDN performance and resilience assessment is still at the beginning. The thesis aims to **contribute to the advancement in testing and evaluation of SDNs**, trying to go beyond what can be achieved by means of "traditional" software analysis and testing techniques.

Within this vision, this dissertation first presents **SCP-CLUB** (**SDN Control Plane CLoUd-based Benchmarking**), a benchmarking framework designed to automate the characterization of SDN control plane performance, resilience and fault tolerance in telco cloud deployments. The idea is to provide the same level of automation available in deploying NFV function, for the testing of different configuration, using idle cycles of the telco cloud infrastructure. Then, the dissertation proposes an extension of the framework with mechanisms to evaluate the runtime behaviour of a Telco Cloud SDN under (possibly unforeseen) failure conditions, by exploiting the **software failure injection**.

Differently from software *fault* injection [23] - a nowadays consolidated form of testing - failure injection focuses on deliberately introducing *failures* in the components of the system under assessment, or in their execution environment, under real or emulated load conditions, to evaluate the ability of the system internal mechanisms to react to anomalous situations potentially occurring in operation.

Overall, the framework provides an approach to implement an automated methodology for characterizing the performance and resilience of the SDNs, and consists of a configurable software infrastructure. The distributed infrastructure encompasses - as main components - a set of *management* tools, a *workload generator*, a *failure injector*, and *data collectors*.

The experimental evaluation of the proposed framework is based on the open source distributed network operating system, ONOS® [10] [24], which is the very heart of the testbed. The ONOS® initiative is supported by several major industrial partners, including AT&T, Cisco, Ericsson, Google, Huawei, NOKIA.

In summary the **main contribution of this thesis** are:

- An automated and configurable distributed infrastructure (named SCP-CLUB framework) for deploying and testing SDN controllers under various configurations. The infrastructure encompasses tools to support the management of the experiments, the load generation, failure injection and data collection tasks;

- An injection methodology, conceived for both development and in-production stage assessment. The methodology envisages the steps of *(i)* definition of the workload (according to the Intent Based Networking model [25]) to emulate actual operating conditions of a controller; *(ii)* workload generation and actual injection of failures, selected from the failure model, in the emulated load conditions; *(iii)* data collection and assessment analysis. Clearly, workload emulation (definition and generation) is not necessary for in-production tests, yet it is important at the current state of the practice given the limited availability of SDN on-field deployments;

- The experimental evaluation, on a distributed testbed based on ONOS® over Nokia AirFrame telco cloud technologies.

1.3 Thesis organization

The dissertation is organized as follows.

Chapter 2 introduces the main concepts of Software-Defined Networking, expected to become the paradigm underlying the next generation of computer networks. It then presents a discussion on the current state of art, and the identified open challenge, namely the resilience assessment of SDN controllers.

Chapter 3 surveys the literature on SDN architectures and platforms, and in particular on their resilience mechanisms, and on failure injection testing techniques and tools for the assessment of software intensive systems. It then discusses the opportunities envisaged in the proposed application of failure injection methods to the problem of assessing the resilience of SDN controllers.

Chapter 4 presents the SCP-CLUB (SDN Control Plane CLoUd-based Benchmarking), a cloud-based benchmarking framework designed for performance analysis of a telco cloud-based SDN control plane. SCP-CLUB provides the automated tools to deploy and test SDN infrastructures, allowing telco operator to perform the assessment of SDNs in controlled as well as in-production environments. The Chapter describes the implementation of the proposed framework based on the ONOS® distributed SDN controller. The results of extensive experiments in an industrial telco cloud infrastructure are presented, showing the effectiveness of SCP-CLUB in automating performance evaluation campaigns.

Chapter 5 presents the vision of continuous and in-production testing, and describes the improvements made to the SCP-CLUB framework for the

resilience assessment of SDNs. The central idea is the use of failure injection to assess the failure detection and mitigation mechanisms of SDN controllers. The failure injection methodology and the support infrastructure are presented along with the underlying failure model, listing the variety of injectable failure types at system, network and service level. Then, the chapter presents the ONOS-based implementation of the proposed failure injection framework, and discusses the experiments by injecting failures into ONOS®.

Chapter 6 summarizes the problem addressed in this thesis and its main contributions.

This page intentionally left blank.

The only joy in the world is to begin.

Cesare Pavese

Chapter 2

Software-Defined Networks

The Chapter introduces the concept of software-defined networking and its logic architecture. It then presents the major dependability requirements for SDNs, preceded by a short introduction of the basic concepts of dependability. Finally, the open challenges in SDN dependability - and in particular, SDN resilience - are identified, which drive the work of this dissertation.

2.1 Abstracting the Network: SDN

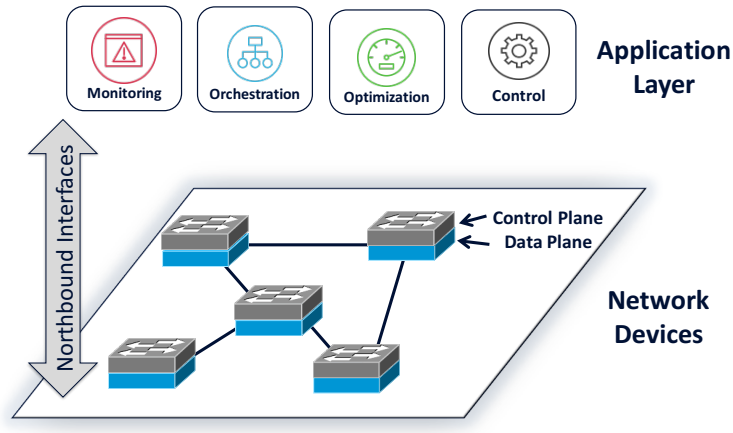
2.1.1 Key Concepts

Software Defined Networking (SDN) is an emerging paradigm to design, build and operate networks. It originated from work started at University of California at Berkeley and Stanford University in 2008, and it has increasingly gained momentum from both the research and industrial viewpoints in the computer networking sector.

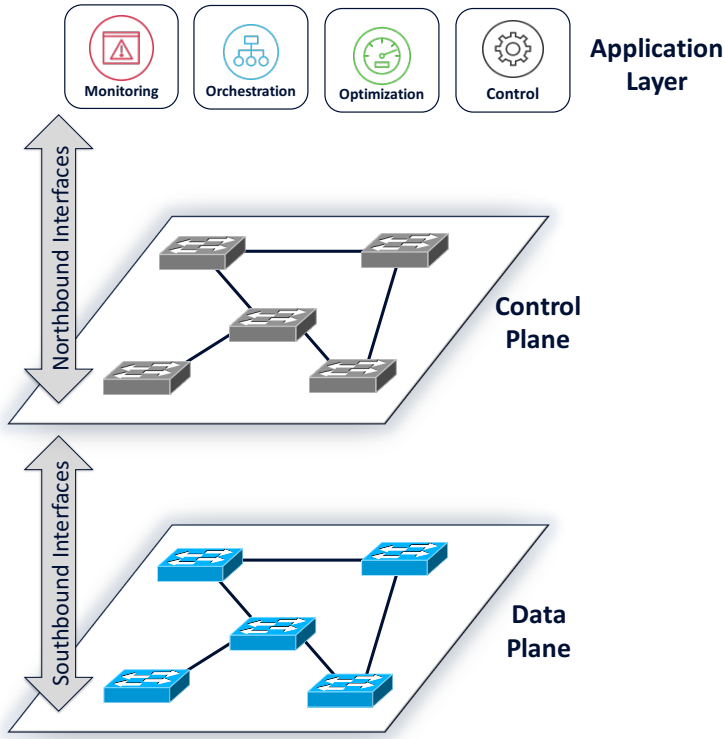
The driving motivation was the need for a major shift in networking technologies in order to support much easier configuration, management,

operation, reconfiguration and evolution than in current computer networks. Indeed, network operation and management are challenging tasks, and telecommunication operators and Internet and cloud service providers face big issues in configuring large networks, enforcing desired policies, and evolving to new technologies [4]. Computer networks are nowadays at the basis of most critical infrastructures, and of the many services we access in our daily activities - be they business, consumer, social or private. Large network's configuration and management is very difficult because enforcing high-level policies requires specifying them in terms of low-level commands of many proprietary, vertically integrated devices of different vendors [4]. These difficulties hamper the development and rapid provisioning of new advanced protocols and services to the highly demanding modern and future fixed and mobile applications. This motivation led to the definition of a new paradigm, envisaging a layered network architecture.

The key concept (Figure 2.1) is the *separation of the network control logic from the network equipments* that forward and transport the traffic [1]. Traditional networks are hardware-centric, and most network equipments (e.g., routers and switches) are *closed*, in the sense they incorporate both the control and data parts (Figure 2.1a), and have their own vendor-specific interfaces. Replacing or simply updating protocols and services is very complex because all equipments have to be replaced/updated [26]. In SDN



(a) Traditional networking



(b) Software-defined networking

Figure 2.1: Separation of data and control in SDNs.

(Figure 2.1b), network devices are simply packet forwarding devices residing in the so-called **data plane**, while the "brain" (the programmable control logic) resides in the **control plane**, distinct and above the data plane; network equipments are programmed via the control layer.

2.1.2 SDN Architecture

As shown in Fig. 2.2, the separation of the layers is realized by means of *open* application programming interfaces (API), called **Northbound Interface** (NBI, towards applications) and **Southbound Interface** (SBI, between the control and data planes). This concept allows co-existence of the new paradigm with the traditional one; indeed, several current commercial network equipments are hybrid, supporting both the new SBI and traditional protocols. This should ease the transition to SDN architectures.

The logical entity hosting software defined core network services in the control plane (e.g. routing, authentication, discovery) is known as **SDN controller** (or simply controller). It is responsible for enacting the policies and the services required by applications, by issuing commands are receiving events and status information from devices in the data plane (referred to as *SDN-enabled switches*) through the SBI.

Much effort in the scientific and technological SDN literature has been put on the Southbound Interface. The main southbound API is OpenFlow

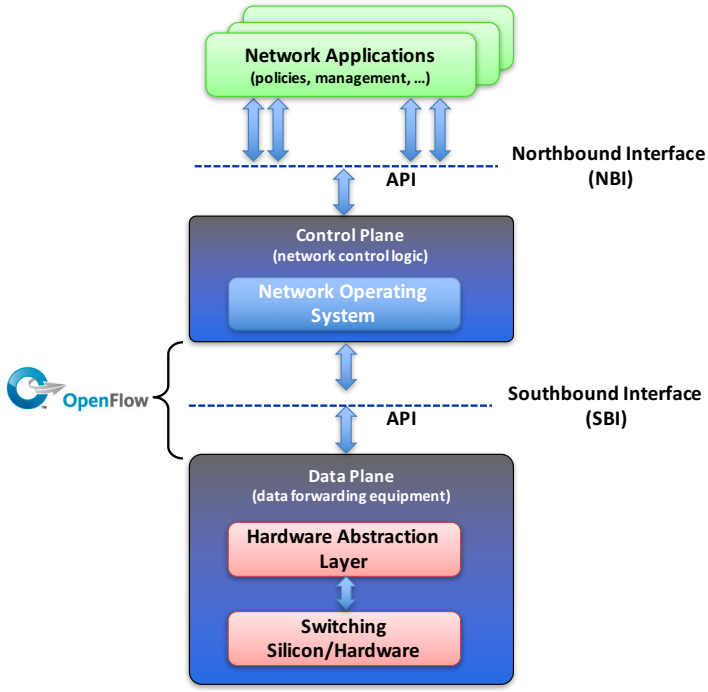


Figure 2.2: SDN architecture and interfaces.

[11], standardized by the Open Networking Foundation [27] (see Section 2.1.3), and supported by many leading network equipment vendors (such as IBM, NetGear, NEC, HP).

Less emphasis has been put so far on the northbound interface and protocols. The NBI is responsible to provide means to specify and request network policies and services in an abstract way, independent from way they are actuated by the controller. A promising proposal for the northbound interface is the **Intent-Based Networking** (IBN) model [28] [25], adopted in the ONOS project; it is described in Subsection 2.1.4.

2.1.3 Standards and technologies

From a technological and industrial viewpoint, several initiatives have started in the recent years to foster SDN development. A major initiative is the **Open Networking Foundation** (ONF), a non-profit organization launched in 2011 by Deutsche Telekom, Facebook, Google, Microsoft, Verizon, and Yahoo!. ONF currently counts tens of partner companies and it has as mission the "promotion and adoption of Software-Defined Networking through open standards development" [27]. Another relevant non-profit initiative is the **Open Networking Lab** (ON.Lab), established by service providers, network operators and network equipment vendors with the main goal of building open SDN tools and platforms. ONF and the ON.Lab announced in late 2016 they will join in 2017 under the ONF name to accelerate the adoption of SDN.

The first ONF achievement is the **OpenFlow™ Standard**, enabling remote programming of the forwarding plane [29]. OpenFlow provides the interface between the control and data planes, enabling a seamless communication between components in the two levels. OpenFlow was initially proposed for technology and application experimentation in a campus network [11]. It then gained momentum, up to be defined as an ONF standard for the southbound interface between the control and the data plane.

The **ONOS project** has been established to develop an open SDN-based vision of the next generation networks, to go beyond current networks, which are "closed, proprietary, complex, operationally expensive, inflexible" [30]. The explicit goal is "to produce the Open Source Network Operating System that will enable service providers to build real Software Defined Networks" [30]. This goal is pursued by a community of partners including many of the major industrial players in the field, be they network operators, Internet service providers, cloud and data center providers, and vendors, including AT&T, Cisco, Ericsson, Google, Huawei, NOKIA, NEC, NTT Communications, Samsung, Verizon.

The project has promoted the development of **ONOS™ (Open Network Operating System)**, claimed to be the first open source SDN network operating system. ONOS is not the first open source SDN controller, yet it is the first targeting scalability, high availability and high performance. It has been conceived to overcome the limitations of previous controllers such as NOX [7] and Beacon [8], which were closely tied to the OpenFlow API and provided applications with direct access to OpenFlow messages - in this sense, they did not provide the proper level of SDN abstraction to applications, hence the need for a real SDN *network operating system*[1].

[1]Note that the term network operating system (NOS) some decades ago referred to operating systems with networking features (such as the one by Novell); this obsolete usage has been changed in [7] "to denote systems that provide an execution environment for programmatic control of the network". This is what is currently still meant with the term in the context of SDN, which is probably why nowadays the terms *SDN controller* and *network operating system* are often used interchangeably.

2.1.4 Intent-Based Networking

In the ONOS view, the Intent-Based Networking model plays an important role in specifying the network needs through a policy-management service [24]. The idea of IBN is that applications should send requests or policies to the control plane in the form of **intents**, specified in terms of *what* and not in terms of actions to be taken in controlling the network, i.e. of *how* they should be actuated (the slogan is: *"tell me what you need not how to do it!"*). A simple example of an intent is the request to establish a point-to-point interconnection between two nodes, complemented by performance or service requirements, such as minimum bandwidth and duration.

An intent can be regarded as an object containing a request to the network operating system to alter the network behavior. It may consist of:

- Network Resources, the parts of the network affected by the intent;
- Constraints, such as bandwidth, optical frequency and link type requested;
- Criteria, describing the slice of traffic affected by the intent;
- Instructions, i.e. actions to apply to the slice of traffic of interest.

The IBN model abstracts the specification of the needs of workloads consuming network services (the "what"), from the way the network infrastructure satisfy those needs (the "how"). This is meant to be achieved

through a policy-management service at the northbound interface of the SDN controller; the latter is in charge of translating the network policies into corresponding control actions, e.g. flow rules installation.

The intents specify at a logical level the actions requested, then the SDN is in charge of satisfying them by directly interacting with the physical devices. By doing so, the IBN framework abstracts the network complexity allowing network operators and applications to describe policies rather than low level instructions for the devices.

2.2 SDN Dependability

2.2.1 Basic Dependability Concepts

Dependability of a system or a service is a concept encompassing several quality attributes (Fig. 2.3), namely *availability*, *reliability*, *safety*, *confidentiality*, *integrity* and *maintainability*, with confidentiality, integrity, and availability being part of the composite attribute *security* [18].

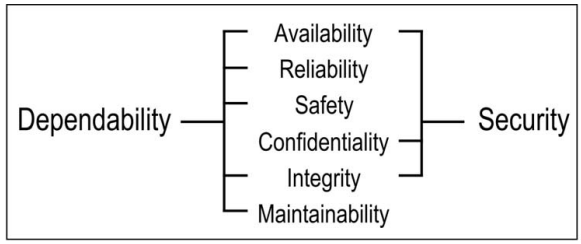


Figure 2.3: Dependability attributes (Avizienis et al., 2004).

Reliability expresses the continuity of correct service. It is the probability that the system functions properly in the time interval $(0, t)$:

$$R(t) = P(!failure\ in\ (0, t)). \tag{2.1}$$

Typically, reliability is evaluated through widely spread metrics such as Mean Time To Failure (MTTF), Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR).

Availability expresses the readiness for correct service. It is the probability that the system functions properly at time t:

$$A(t) = P(!failure\ at\ t)) \tag{2.2}$$

and it is often expressed as uptime divided by total time (uptime plus downtime); often, it is computed as the ratio:

$$A = \frac{MTTF}{MTTF + MTTR} = \frac{1}{1 + \frac{MTTR}{MTTF}} \tag{2.3}$$

which shows that for improving availability, it is important to reduce the ratio between MTTR and MTTF, by increasing the mean time to failure and/or by reducing the mean time to repair.

Safety is the absence of catastrophic consequences for the users and the environment. **Confidentiality** is an information security property; it is the property of a system to be able of not making available or disclosing or making understandable (protected) information to unauthorized individuals, entities or processes. **Integrity** is the property representing the

absence of improper alterations, may they concern - for instance - system, messages or data. **Maintainability** is the ability of a system to undergo modifications and repairs.

In this dissertation, we do not deal with safety, confidentiality, integrity and maintainability, and in the next Section we will focus on the dependability requirements placed on software-defined networks concerning availability, reliability and scalability.

2.2.2 SDN Dependability Requirements

In a software-defined network, although logically centralized, the controller is a physically distributed entity. This is because **dependability requirements** - mainly on scalability, availability and reliability - demand for its engineering in a distributed architecture. If, as claimed, SDN is going to become the technology of future networks, it has to fully address these requirements. Telecommunication operators, for instance, are unlikely to adopt SDN to replace existing carrier-grade networks unless SDN is proved to be able to provide at least the same quality-of-service, while providing greater flexibility and ease of management [16].

Scalability is not strictly a dependability attribute in the current classification, yet it is a major concern for SDNs. While certainly it is a fundamental design consideration for SDN controllers, this concern is often

overlooked. It is a common belief that - differently from current networks, where devices are often realized with specialized application-specific integrated circuits (ASICs) - a logically centralized but physically distributed software defined controller may not scale as the network grows. However, this appears to be a wrong belief. As argued by Yeganeh et al [31], "there is no inherent bottleneck to SDN scalability". The issues of scalability in SDN are similar as in any distributed system, and scalability is not inherently harder to achieve than in traditional networks. That is, if a distributed SDN is required to provide a unified network-wide view, solutions need to incorporate distributed consistency protocols. If there exist limits for distributed systems, they are probably those stated in the famous Brewer's Conjecture [32], claiming that it may not be always possible to achieve strong consistency, high availability and partition tolerance all together.

As for **availability**, the requirements on SDN are very stringent; these are not different from those of current networks. If SDN have to be used for basic yet critical services such as telephony, they are required to provide an end-to-end availability of five "nines" (i.e., 99,999%)[2] as in today carrier-grade networks [33]. In current networks, this is achieved at the cost of manual configuration and long deployment times [16]. The flexibility of SDN is very appealing from the point of view of telecommunication

[2]A five "nines" availability amounts to about five minutes of downtime per year.

networks operators, but they are not going to sacrifice availability for maintainability. Achieving such high levels of availability when the network is software defined may be hard and it requires careful design and accurate implementation, but *it is possible*. However, recent work by Akella and Krishnamurthy [34] has shown that availability issues for SDNs systems are more deeply rooted than those stemming from their complexity (see also Sections 2.4 and 3.1.2.2).

Fault-tolerance is (along with fault prevention, removal and forecasting) a means to increase the dependability of a system. It is clearly of paramount importance for SDN, since in the event of a controller failure the whole network can be compromised, because all applications and services depend on it. All SDN controllers are engineered with mechanisms to tolerate such events, and clearly fault-tolerance is one of the major techniques used to ensure a high level of resilience. This dissertation proposes failure injection as a testing technique to intentionally introduce failures, representative of failure events which can actually occur at various levels in SDN networks, so as to evaluate the controller's fault-tolerance mechanisms, and more in general to assess the extent to which a controller is able to provide the desired level of resilience.

2.3 SDN Resilience

The concept of **resilience** (or resiliency) has multiple definitions, as it has developed in different disciplines, including physics, psychology, ecology, engineering. Very likely, the term originally referred to a property of a physical material or of an entity, but the concept is now applied also to networked systems or organizations. Despite the different definitions, quoting from [35] there are "three elements present across most of them: the ability to change when a force is enacted, [to] perform adequately or minimally while the force is in effect, [to] return to a predefined expected normal state whenever the forces relents or is rendered ineffective".

In engineering, the term somehow intuitively conveys the notion of the ability to survive to unintentional or malicious threats (failures, attacks, etc.) and to resume normal operating conditions; we can say that the *resilience of a system* is often thought as its ability to provide and maintain an acceptable level of performance or service in presence of failures. It is worth to explicitly point out that resilience is a macroscopic-scale property of a system, i.e., a property of the system as a whole [35].

According to [36], computer **network resilience** *is the ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation.* In [37] network resilience is defined as *the ability of a network to defend against and maintain an acceptable level of service in*

the presence of challenges, such as malicious attacks, software and hardware faults, human mistakes (e.g., software and hardware misconfigurations), and large-scale natural disasters threatening its normal operation.

In [38] the resilience of systems has been evaluated by means of a time-dependent indicator, known as figure-of-merit $F(\bullet)$, which is a quantifiable indicator of the performance of the system. As depicted in Figure 2.4, the state of a system is characterized by a value of $F(\bullet)$, directly affected by the two events (the disruptive event and the corresponding recovery action). Multiple indicators can be defined to provide a measurement of resilience, concerning reliability, network connectivity paths, flows, etc.

In Figure 2.4, the system is initially "functional" at time t_0, and this state remains constant until the occurrence of a disruptive event at time t_e, bringing the value of the delivery function of the system from its initial value $F(t_0)$, to the lower value $F(t_d)$. Thus, the system is assumed to function in a degraded mode from t_e to t_d, when it reaches the point where

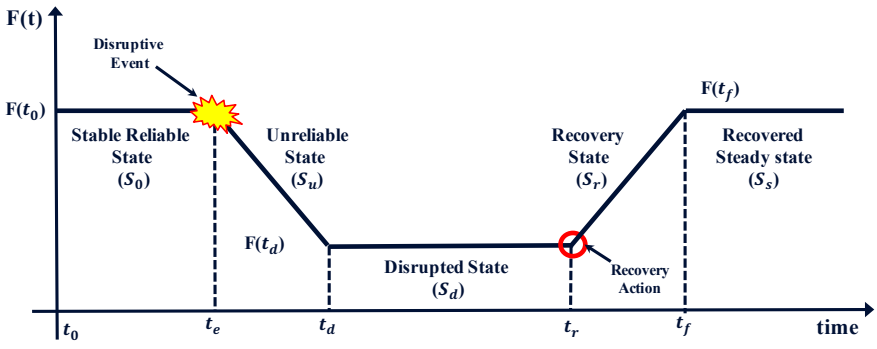


Figure 2.4: System behavior in the occurrence of a disruptive event.

the functionality is considered to be entirely lost. The system remains in such state until a recovery/repair action is initiated at time t_r when the system regains functionality, although in a degraded mode. As a result of the resilience action, the system is considered recovered and fully-functional at time t_f, with delivery function value $F(t_f)$. However, the final state, reached by the system after the recovery action, does not necessarily have to coincide with the original state of the system, i.e., the figure-of-merit $F(t_f)$ can be equal, grather, or smaller than $F(t_0)$. Finally, the value of resilience corresponding to a specific figure-of-merit function can be computed as:

$$\Lambda(t) = F(t) - F(t_d)F(t_0) - F(t_d) \tag{2.4}$$

where $0 \leq \Lambda(t) \geq 1$ for $t \in (t_r, t_f)$ assuming that the recovery action succeeds in restoring the functionality.

In recent years, the concept of resilience has been broadened to incorporate a notion of dependability also with respect to changes; indeed, it is increasingly conceived as *the capability of a system to remain dependable in the present of changes*. This probably comes from the work by Laprie [39], who pointed out the need to address the growth of complexity of today so pervasive computing systems, a need deriving from changes which can be *functional*, *environmental* and *technological*. Laprie introduced *scalable resilience* as a concept of "survivability in direct support of the emerging pervasiveness of computing systems" [39].

For the purpose of this thesis, the following definition - essentially provided in [40] - will be used: **Network resilience is the ability to provide and maintain an acceptable level of service in the face of failures.**

Nowadays, resilience is a major requirement and design objective for computer networks. This is due to the fact that computer networks are at the basis of most critical infrastructures, subject to both unintentional faults/failures and to malicious (cyber-)attacks.

2.4 Open Challenges

Notwithstanding the large literature on SDNs, there are several still open problems related to the fulfillment of dependability and resilience requirements by SDNs. One of the reasons for this is that - quoting from [40] - "there is almost no practical way to experiment with new protocols in sufficiently realistic settings (e.g. at the scale carrying real traffic) to gain the confidence needed for their widespread deployment".

Availability may pose more threats to SDNs than those posed to generic distributed systems. Akella and Krishnamurthy [34] have shown that availability issues for SDNs systems are more deeply rooted than those stemming from the complex and critical inter-dependencies among the various network and distributed systems protocols they use. The authors provide a case for this, which is worth presenting here.

Let us consider the network in Fig. 2.5 (reproduced from [34]), where $C1$ to $C5$ are controller replicas in a distributed control plane, and $S1$ to $S9$ are switches in the data plane. Let us also point out that distributed controllers incorporate consensus protocols such as Paxos [41] to ensure they have a consistent view of network topology and data plane state even in the presence of failures or disconnections. When the links between $S4$ and $S6$ and $S4$ and $S8$ fail, a network partition occurs. In this case, switches in the data plane partition on the right cannot be updated since they can contact only a minority group ($C4$ and $C5$) of the replicated controllers. If the path between $S6$ and $S8$ managed by the control plane prior to the partitioning is not wholly contained in the right partition, $S6$ and $S8$ cannot communicate even if there is a path between them not affected by the failure. Such critical situations undermine high availability. Clearly, legacy network protocols do not suffer from this issue: when partitions happen, routers re-converge to new intra-partition routes. Quoting the authors, the case they provide shows that "current SDN designs fail to provide important fault-tolerance properties, which renders SDNs less available than traditional networks in some situations". While there exist solutions to the problem described (e.g. using partitioned consensus), they may address it only partially; specifically, there may be consistency pitfalls in case of concurrent events. The authors themselves proposed a solution - although they did not prove it - based on

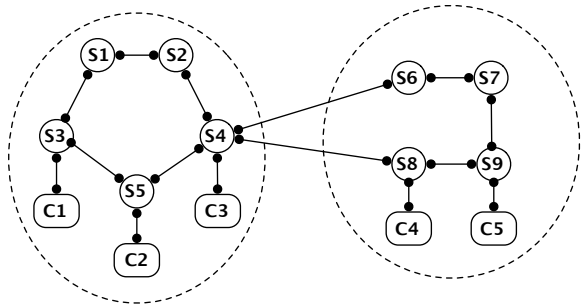


Figure 2.5: A case for availability threat in SDN (Akella, 2014).

a combination of the Chandy-Lamport snapshot algorithm [42], of reliable flooding and of whole quorum consensus. However such mechanisms are expensive, and while they appear to be sufficient, it is not clear if they are all necessary.

Sharma et al. focus on **fault tolerance** and **failure recover** in OpenFlow for deployment it in carrier-grade networks [40], as means to improve SDN resiliency. Carrier-grade networks pose a strict requirement that the network should recover from a failure within a 50 ms interval. They show that OpenFlow may not be able to satisfy this requirement, and they propose a recovery action in the switches without involving the controller. As for **reliability**, they briefly discuss as future work some possible approaches.

Di Martino et al. [16] present the **resiliency challenges** for future carrier-grade networks based on SDN. They build on an analysis of outages of current carrier-grade networks to identify three main factors impacting the effectiveness of their failover mechanisms, namely: *untested operational*

context, *multiple failures during the failover window*, and *human/procedural errors*. Based on this, the implications drawn include:

- The need for new resiliency techniques and for new in-production approaches to testing (the authors mention those used in the Chaos Monkey approach by Netflix [22] [43]). This is because SDNs are expected to reduce the service deployment time down to the order of seconds; this in turn may imply reduced testing and assessment.

- The need for new failover mechanisms. This is because many mechanisms are programmed in today's network devices, while in SDNs failover will demand for interaction between various layers and among different domains.

- Service business models driven by Service Level Agreements (SLA) will require richer resiliency specifications and validation. This is because SDN will allow high flexibility, service provisioning will be more dynamic (for instance, due to migration for resources' optimization), and service level will need to be validated dynamically.

Jain et al. [21] present *B4*, a practical example of a large scale implementation of an SDN-based WAN connecting the Google©'s data centers around the globe. The authors provide implementation details about one of the first and largest SDN deployments, which has shown to be efficient in

meeting both performance, and reliability requirements. However, despite the outstanding performance, *B4* experienced an outage due to a human error. Indeed, during a planned maintenance activity two physical switches were configure with the same ID causing substantial *link-flap* errors, i.e. their network interfaces continuously went up and down, and more protocol processing activities to discover the network topology. This has lead to subsequent failures of the Google's public network, affecting the network connectivity of their customers.

Unpredictable events underlines that, as long as a system has been shown to be robust as a result of testing activities, nevertheless it can fail in a **production environment**, where operating conditions change and evolve over time. Moreover, **traditional software-testing tools are inadequate** to verify the resilience of such complex distributed systems against all potential failure scenarios that can span across the whole system stack.

In summary, new approaches are needed for the **automated verification of SDN resilience** while exercised by real workload conditions, namely in production. Innovative approaches would allow to discover failure points otherwise difficult to detect by means merely of software testing, helping to design better detection techniques, and building the right mitigation means to recover the system when real issues arise.

This page intentionally left blank.

Every journey has a starting point.

-

Chapter 3

Related work

This Chapter surveys the literature on performance and resilience of software defined networks and on software fault injection (SFI). First, it analyzes existing work on the design of mechanisms for SDN controllers dependability, and on metrics and techniques for the evaluation of their performance, reliability, fault tolerance an resilience. It then introduces SFI, the technique proposed in this thesis for resilience assessment of SDN. Rather than providing a Systematic Literature Review, the goal is: i) to investigate the state of research on performance evaluation of SDNs as well as techniques to ensure controllers' resilience; ii) to provide a short background on SFI and on its application to dependability assessment.

3.1 SDN Performance and Resilience

3.1.1 Sources

Despite the fact that Software-Defined Networking is still a relatively young field, the literature is rather large. This is due to the great appeal it has encountered in both the academic and industrial sectors.

From a scientific viewpoint, at its beginning the SDN literature spread

in publication venues traditional of the computer network and software dependability fields, such as the IEEE/IFIP Conference on Dependable Systems and Networks (DSN), the IEEE Symposium on Software Reliability Engineering (ISSRE), and all major journals and conferences on computer networks, and on systems/software dependability. Currently, specific conferences, conference series and journal special issues on SDN are really proliferating. The main ones include:

- The ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (**HotSDN**); the series started in 2012 in Helsinki (FN);
- The IEEE Conference on Network Softwarization (**NetSoft**), whose first edition was held in London (UK) in April 2015;
- The IEEE Conference on Network Function Virtualization and Software Defined Networking (**NFV-SDN**) (from 2015, and in 2017 the SDN-NFV Track in SAC Symposium at the IEEE International Conference on Communication (ICC 2017);
- The Research Track of the Open Networking Summit (**ONS**, since 2014 in conjunction with USENIX, the Advanced Computing Systems Association);
- The IEEE/IFIP Network Operations and Management Symposium (**NOMS**);
- The First International Workshop on Software Defined Networks and Network Function Virtualization (**SDN-NFV**), in conjunction with The Fourth International Conference on Software Defined Systems (SDS-2017);
- The European Workshop on Software Defined Networking (**EWSDN**), whose first edition was held in Darmstadt (D) in April 2012;
- The Special Issue on "Software Defined Networking", September 2015, and the upcoming Special issue on "SDN and NFV based 5G Heterogeneous Networks" of **IET Networks**;

- The Special Issue on "Future network: software-defined networking" of Frontiers of Information Technology & Electronic Engineering, July 2016;
- The Special Issue on "Management of Softwarized Networks", September 2016, and the (upcoming) Special Issue on "Advances in Management of Softwarized Networks" of IEEE Transactions on Network and Service Management (**TNSM**);
- The Special Issue on "Software-Defined Networking and Network Functions Virtualization for flexible network management" of the International Journal on Network Management (**IJNM**), November 2016;
- The Special Issue on "Management of SDN/NFV-based Systems" of the International Journal on Network Management (**IJNM**).
- The (upcoming) Special Section on "Network Virtualization, Network Softwarization, and Fusion Platform of Computing and Networking" of IEICE Transactions on Communications (**IEICE TC**);

The existing literature on the evaluation of performance, reliability and fault tolerance of SDN controllers has been searched in the above sources in the networking and dependability research fields, and more in general in all major scientific computer science and engineering databases, including IEEExplore, ACM Digital Library, Scopus, ISI Web of Science, Google Scholar. It is described in the following subsection.

3.1.2 Related Work

Despite the bulk of work in the scientific literature, the research on SDN performance and dependability can be considered still at the beginning.

Clearly, the decoupling of control plane from the network devices and the implementation of controllers as distributed systems inevitably make software-defined networks inherit the weaknesses associated with reliable distributed systems [34]. As recently stated in [16], "*the specific benefits and risks that SDN may bring to the resilience of carrier-grade networks remain largely unexplored*", and according to [44] "*the dependability of SDN itself is still an open issue*". The related work can be broadly categorized in *SDN benchmarking* (i.e., the evaluation of performance and scalability of SDN platforms), and *SDN dependability* (i.e., the mechanisms to ensure desired dependability features by proper design choices).

3.1.2.1 SDN Benchmarking

SDN performance benchmarking is addressed in several studies [45] [46] [47] [48].

Cbench [45] is a benchmarking tool of this type based on simulating a configurable number of OpenFlow switches; it used in [46] to compute various controller performance metrics (response time, throughput, latency) of four controllers (NOX, NOX-MT, Beacon, and Maestro) based on OpenFlow, probably the most widely known southbound API [11].

Jarschel et al. [47] proposed a flexible OpenFlow controller benchmarking tool (OFCBench) to overcome various limitation of Cbench and previous tool (single threading, use of one controller connection for all emulated switches). They also proposed an analytic (queueing) model for predicting the performance of an OpenFlow architecture, in terms of packet sojourn time and probability of lost packets [49].

Cbench is also used by Zhao et al. [48], who presented an evaluation of five open-source controllers. However, it considers mainly centralized controllers; distributed controllers, which are gaining momentum also for addressing scalability, have more complex performance problems, ranging from the placement of the replicas to their synchronization.

Prior to [48], Tootoonchian et al. [46] presented a study of four publicly-available OpenFlow controllers, based on their Cbench tool [45]. This tool measures the number of flow setups per second that a controller can handle, and it supports two modes of operation: latency and throughput mode. A further experience including the evaluation of performance in a wide-area SDN is described in [21].

A crucial performance aspect for SDNs is **latency**. As pointed out in [50], critical SDN mechanisms such as *fast failover* and *traffic engineering* demand for the ability to program the data plane state at fine time-scales. In their study, He et al. state that "*timeliness is determined by: (1) the speed of*

control programs, (2) the latency to/from the logically central controller, and (3) the responsiveness of network switches in interacting with the controller [50]. While the first two factors are already being overcome by advances in distributed controllers, the authors' measurements show that the third one may be critical even with most modern switches. The *inbound* and *outbound* latencies (those concerning events generated by switches and those in the execution of rules provided by the controller, respectively) are high and variable. The study thus highlights the need for "*careful design of future switch silicon and software in order to fully utilize the power of SDN*".

Scalability metrics for the the SDN control plane are proposed in [51] and [52]. The first metric specific for SDNs appears to be the one by Hu et al. [51], who propose to compute scalability when the network scale varies from N2 to N1 as:

$$Scalability \quad \Psi(N1, N2) = \frac{\phi(N2)\frac{T(N2)}{C(N2)}}{\phi(N1)\frac{T(N1)}{C(N1)}}, \tag{3.1}$$

where $\phi(N)$ is the throughput of the control plane in processing network requests, $T(N)$ is the average response time per request, and $C(N)$ is the cost to deploy the control plane. The authors evaluate the metric with reference to three SDN control plane architectures – centralized, distributed and hierarchical - by building performance models for the response time, based on which they evaluate the scalability of the three structures.

While the previous metric considers throughput, average response time

and cost, the metric proposed in [52] does not consider the deployment cost, and assumes that the average flow processing time has to remain the same when scaling. The metric they propose is:

$$Scalability \quad f(W, O) = \frac{W}{O}, \tag{3.2}$$

where W and O are the workload and overhead, respectively; the former is the number of flows entering the data plane, the latter is the number of messages processed by the controller(s).

3.1.2.2 SDN Resilience and Dependability

Several research groups have coped with the problem of defining proper mechanisms to ensure resilience and dependability for SDN controllers.

Heller et al. [53] have formulated the **Controller Placement Problem**, the one of deciding - given a network topology - how many controllers need to be used and where to place them to satisfy performance and fault tolerance requirements. They are concerned specifically with wide-area networks and with the minimization of propagation delays. Their study shows that the answers to the two questions depend on the topology and on the metric (a tradeoff has to be found between optimizing for worst-case or average-case latency), that for most topologies adding controllers provides almost proportional delays' reduction, but surprisingly, *in medium-size networks*

one controller location can be sufficient to meet current typical real-time requirements. Clearly, one is not enough for fault tolerance.

SDN fault tolerance is addressed by Fonseca et al. [54], who propose a primary-backup mechanism to provide resilience against several types of failures in a centralized OpenFlow-based controlled network. In *primary-backup replication* [55], one or more secondary (backup) replica servers are kept consistent with the state of the primary server, and as the primary server enters in a failure state, one (warm) backup replica is chosen to replace the primary server. Hence, this approach is well suited where there is a centralized control concentrating in one point of the network the information that need to be replicated. The approach has been implemented in the NOX controller, and has been shown to work in several failure scenarios, namely abrupt abort of the controller, failure of a management application (client running atop the controller), Distributed Denial-of-Service (DDoS) attack. The authors conclude that the OpenFlow protocol proved to be appropriate to support ease implementation of primary-backup replication.

Ross et al. [56] build on the previous work by Heller et al. formulating the **Fault Tolerant Controller Placement Problem**, as the problem of deciding how many controllers are needed, where they have to be deployed, and what network devices are under control of each of them, in order to

achieve at least five nines reliability at the southbound interface (the typical reliability level required by carrier-grade networks). They also propose a heuristic to compute placements with such reliability level. Again, the answers depend on the topology (rather than on the network size). However, *it is possible to achieve fault tolerance in SDN by careful selection of the placement of controllers.*

SDN availability design issues are addressed by Akella and Krishnamurthy [34]. They show that there may be situations where link failures can compromise the proper functioning of portions of a SDN. This is due to the fact that controller internal modules - specifically, distributed consensus protocols, mechanisms for switch-controller or controller-controller communication, and transport protocols for reliable message exchange - can have cyclical dependencies. This means that link failures can cause transient disconnections between switches and controllers or controller instances, which in turn undermine high availability. What appears to be particularly critical in SDN is the lack of robustness to failures which partition the topology of controllers. In fact, it has to be noted that - since in SDN the control has been taken out of switches and logically centralized in the control plane - it may happen for two switches to be unable to communicate even if a physical path between them does exist. The authors argue that current SDNs may be unable to offer high availability, and they should be re-architected by

including advanced mechanisms from the distributed systems theory, such as reliable flooding and global snapshots.

3.2 Fault Injection Testing

Fault injection [57] is the technique of introducing faults in a system to assess its behavior and to measure the effectiveness of fault tolerance or other resilience mechanisms. Although much more recent than hardware fault injection, **Software Fault Injection** (SFI) is today widely used too [23]. SFI developed as it became clear that software faults were becoming a major cause of systems' failures. It proved to be effective for fault-tolerance and reliability assessment for several classes of systems, such as distributed systems, operating systems, Data Base Management Systems, and it is nowadays recommended by several standards in critical systems domains [23].

SFI consists of applying small changes in a target program code, in a way similar to *mutation testing* (a well-known software testing technique), with the goal of assessing the system behavior in the presence of (injected) faults, which clearly have to be representative of potentially real faults - or, specifically, of *residual faults*, those which escape testing and debugging before software product release and may be activated in execution on-field [58]. In a SFI experiment, a fault is injected in the program, which is executed under a workload, in turn representative of real operating conditions.

Fault injection has been proposed by Cotroneo et al. [59] for dependability evaluation and benchmarking of Network Function Virtualization Infrastructures (NFVIs). NFV is a field closely related yet different from SDN. SDN and NFV share the goal of fostering innovation in networking by means of a shift to software-based platforms, that it to network programmability. NFV refers to the virtualization of specific in-network functions (e.g., firewalls and VPN gateways) in order to reduce the dependency on underlying hardware; this eases resource management, provides faster service enablement and lowers OPEX (Operating Expenditures) and CAPEX (Capital Expenditures). NFV solutions can operate in SDNs. While SDN is more concerned with control plane programmability, NFV mainly focuses on data plane programmability. As for SDNs, NFV inherits performance and reliability requirements from telecommunication systems. In their paper, the authors define some key performance indicators for Virtualized Network Functions (VNF), namely latency, throughput and *experimental availability*, and propose fault injection for evaluation and benchmarking of VNFs.

Differently from the *fault* injection approach pursued in [59] for NFV, we propose the use of *failure* injection for the assessment of the resilience mechanisms SDN distributed controllers. The aim it to devise a methodology suited for **in-production assessment**. In-production testing is gaining

importance in all those dynamic contexts where traditional software testing techniques *in fabric* (i.e. before deployment) are not deemed to be suited anymore: one famous such case is represented by Netflix [22]. In the framework of this dissertation, it means injecting failures at system, network or service level during executions under a workload (which is not emulated but real, in case of in-production testing), failures which have to be representative of events typically occurring in normal operation. The proposed failure injection methodology is described in the next Chapter.

3.3 Failure Injection Testing: the Netflix approach

In defining a methodology to assess the resilience and the failover mechanisms of SDN platforms we take inspiration from the failure injection approach proposed by *Netflix*®. It is a multinational company providing streaming and on demanded multimedia services to a wide range of users around the world. In doing so, they engineered a very complex ecosystem according to a "micro-services" architecture pattern, i.e. with multiple small and independent services working together to fulfill a specific goal. This leads to a dynamic operational context, where services are updated or added at runtime without ever interrupting the system, making it impractical, or even impossible, to perform testing activities aimed to assess possible system's deficiencies and identify potential failure modes.

Consequently, a methodology has been proposed to find possible weaknesses in a **production system** by observing its behaviour under the deliberate injection of failures. The execution of failure injection experiments within a live production environment has three main advantages:

- It allows a better assessment of the system by verifying its correct behaviour in realistic production deployment and load conditions;
- It helps making the the system immune to possible failures;
- It helps preventing outages that can affect system availability.

The methodology falls under the umbrella of the broader concept of "Chaos Engineering" [22], [60], which is defined as the "discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand faulty conditions in production". Specifically, this discipline provides few practical principles meant to facilitate the testing activities to uncover system weaknesses, namely:

- Definition of what is a normal system's behaviour, i.e. the system "*steady state*", considering some measurable output of the system;
- Build a control system and an experimental one, with the latter used for the failure injection experiment;
- Introduce disruptions on the experimental system to simulate real-world events, such as server crashes, network failures etc.;

- Compare the steady states of the experimental and control systems to find possible deviations from the normal behaviour and build confidence on system resilience.

Along with these chaos principles, Netflix proposes a *Failure Injection Testing* (FIT) platform to automate [61] [62] the injection and monitoring of arbitrary failures scenarios into specific targeted services or system subset, aiming to support the implementation of systems that are resilient to failure.

Though inspired by the Netflix's approach, the failure injection assessment methodology proposed in this dissertation differs in several aspects:

(i) It targets distributed SDN platform to perform a resilience assessment under failure scenarios, aiming to verify if such systems provide suitable failover mechanisms;

(ii) The framework reproduces failure scenarios which are representative for SDN ecosystems, e.g. faulty communications between SDN controllers, or a faulty controller's service;

(iii) It is meant to perform both *offline*, and *in-production* assessment, since the SDN technologies are still in very early stages to be deployed in a real production environment;

(iv) It provides measurements which give valuable insights into the performance and resilience of the SDNs.

What cannot be assessed
cannot be claimed.

-

Chapter 4

SDN Control Plane CLoUd-based Benchmarking

This chapter presents SCP-CLUB, a benchmarking framework for performance analysis of a telco cloud-based SDN control plane. First, an overview of the generic framework and its tool suite is given. Then, a detailed description is provided of the design and implementation of an ONOS-based instance of the framework. Afterwords, an experimental campaign is presented, showing the effectiveness of the proposed framework in automating very long sessions of experiments for benchmarking a telco cloud SDN. The last section presents and discusses the experimental results.

4.1 Introduction

Telco Cloud is an emerging paradigm in the engineering of telecommunication infrastructures and services, which promises to make their management much more agile [12] [15]. Telecommunication (telco) operators are expected not only to be more and more software driven - with the adoption of softwarization technologies, such as SDN and NFV, in replacing the dedicated telco hardware boxes - but also to increasingly adopt a cloud computing approach to automate the management of their infrastructures and services.

The combined use of these softwarization technologies [1] [63], cloud and virtualization technologies has the potential to support in meeting the requirements of future networks in terms of elasticity, scalability, and resiliency to mutable network conditions, unpredictable traffic patterns, and continuous streams of services. However, it is still not clear for telco operators how these technologies can be best leveraged to meet such highly demanding requirements with carrier-grade service-level guarantees [16].

In a virtualized and cloud-based operational context for SDNs, there are several software layers and thousands of different setups, configurations, and parameters to be properly fine tune in order to obtain specific levels of service. For instance, Telco Cloud Orchestrators (such as XAAS, EC2 and VCLOUD) need automated tools and procedures to decide when to scale up or down the resources running the telco-cloud control (e.g., virtual machines and SDN controller instances). The ability of automating the performance analysis of these emerging technologies, combined to the cloud-based infrastructures is fundamental in the telco cloud scenario. No proper techniques and tools are available so far to address this task.

This chapter presents **SCP-CLUB**, an automated benchmarking framework for performance analysis of a telco cloud-based SDN control plane. SCP-CLUB provides the following features:

- a configurable load generator for SDNs benchmarking;

- cloud automation and orchestration tools to enable parallel benchmarking experiments for computing relevant performance assessment metrics using idle cycles of a telco cloud.

The load generator is based on the **Intent-Based Networking** (IBN) approach (see §2.1.4). Cloud automation supports performance evaluators in the many tasks of orchestrating and running benchmarking experiments and collecting data at various software levels for the analysis, with the ultimate aim of extracting actionable intelligence that can be used to manage (e.g., scale up or down) the cloud resources running the control network.

4.2 State-of-the-art Progress

As we have seen in Section 3.1.2.1, several authors have dealt with the problem of benchmarking SDN controllers. However, the literature focuses essentially on measuring the intrinsic performance of single controller instances with faked interactions with switches, or the performance at the southbound interface.

Probably the most widely used tool is Cbench [45], and the most comprehensive evaluation of the major open-source SDN controllers is carried out by Zhao et al. [48]. Their focus is on benchmarking one controller managing a variable number of switches. They consider Cbench suited to provide accurate results, in that it fakes switches just as "kind of traffic

generators able to send `packet_in` messages as fast as possible". Moreover, measurements are taken sending traffic through the local loopback interface, to eliminate also the link bottleneck.

Rather than focusing on the performance of single instances of controllers without significant requests from applications and with faked interactions with switches, SCP-CLUB takes a different perspective. It considers really distributed controllers in a cloud-based and virtualized deployment; by generating and submitting synthetic yet realistic work loads in terms of *intents* to be actually processed, it allows to investigate the performance of the control plane related to the whole workflow of installing and translating application requests, up to sending commands with flow rules to the data plane and checking their successful processing by the controlled switches. To this aim, both intent installation and withdraw requests are generated and submitted to the control plane at the northbound interface. SCP-CLUB allows also to evaluate the impact on performance of the main configuration parameters, so as to derive hints for their proper configurations.

SCP-CLUB has some similarities with the approach used in the Performance and Scale-out Test Plan [64] designed to characterize ONOS latencies, throughput and capacities (again with Cbench): both use a load generator producing self-adjusting intent install and withdraw requests, and both make measurements as the number of controllers instances in a cluster

scales up. There are however substantial differences.

First, the ONOS performance test plan and results described in [64] are based on "a set of Null Device Providers at the adapter level to interact with the ONOS core. The Null Providers act as device, link, host producers as well as a sink of flow rules". Using such dummy stubs, the ONOS performance figures typically reported in the literature intentionally disregard Openflow adapters and interactions with real or emulated switches. From the point of view of telco operators, this means that such figures can not be trusted to assess the suitability of a controller for use in a carrier-grade SDN. Moreover, the ONOS performance metrics available in the literature are made generating a load to the highest rate ONOS can sustain with synchronous requests. SCP-CLUB is designed to assess actual controller performance with concrete intent requests, complete processing of requests by the controller (intent compilation, installation and removal), and real or emulated topologies considering failures in the data plane (which indeed occur in reality). Finally, SCP-CLUB is based on cloud and virtualization technologies, and it is designed to analyze performance figures such as throughput and latency in real industrial SDN cloud testing datacenters.

The next section introduces the generic SCP-CLUB framework, while Section 4.4 presents an implementation based on the ONOS open source SDN controller [10]. Section 4.5 shows the results of experiments with a

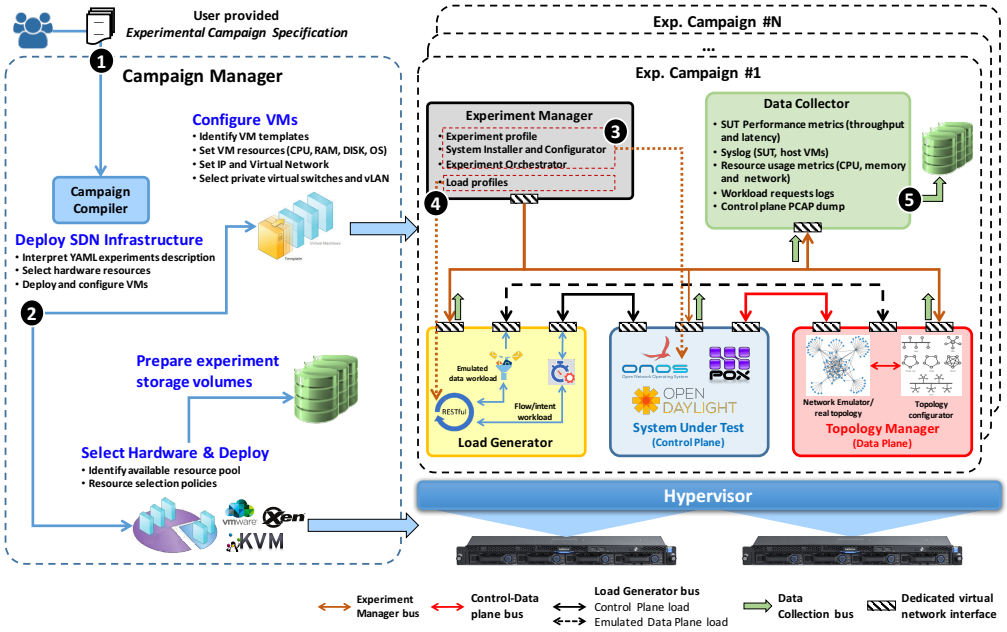


Figure 4.1: The SCP-CLUB framework.

proof-of-concept telco cloud testing datacenter, built to analyze the performance in actual carrier-grade settings. The effectiveness of SCP-CLUB is shown by analyzing the performance of ONOS under a variety of parameters, including: work load, virtual machine size, ONOS cluster size, and number of controlled network devices.

4.3 The SCP-CLUB framework

4.3.1 Overview

The ***SCP-CLUB*** framework is presented in Figure 4.1. It includes five software tools running on different hosting machines. These automated

tools cooperate to configure and deploy a distributed SDN infrastructure in a cloud-based environment, as well as to perform benchmarking experiments for performance evaluation of the SDNs.

For each experiment, an ***Experiment Manager*** (grey box) loads an *experiment profile*, installs and configures all entities in the telco cloud SDN-based infrastructure (controller instances, monitoring and data collection services, topology), then it instructs the ***Load Generator*** (yellow box) to stimulate with a workload the *System under test* (SUT), namely the cluster of controller instances. The load consists of requests from an application at the northbound interface. The ***Topology Manager*** (pink box) configures the topology of the data plane. Run-time data are gathered concerning several entities of various layers (workload, cloud resources, virtual machines, controllers, network topology): this is the task of the ***Data Collector***. Data are collected through a data bus consisting of links logically separated from both those used to communicate among controllers and between controllers and switches.

An *SCP-CLUB* campaign consists of running a set of experiments, each stimulating the control plane of the *SUT* under various operating conditions. The ***Campaign Manager*** (left box in Figure 4.1) orchestrates a session of multiple experiments. A subsequent data analysis allows to assess performance metrics (e.g., throughput and latency), as well as to extract

actionable intelligence for fine-tuning the many configuration parameters of cloud resources running the control network.

Figure 4.1 shows the workflow of a typical *SCP-CLUB* session to accomplish an experimental campaign. At startup, ❶ the user provides a specification of the experiments to the *Campaign Manager*. Then, ❷ the *Campaign Manager* builds the infrastructure for each experimental campaign by selecting the hardware resources, configuring and deploying the VMs. Once the infrastructure is deployed, the *Experiment Manager* ❸ loads an *experiment profile* (built from the user specifications), installs and configures all entities in the telco cloud SDN-based infrastructure, and orchestrates each experiment of the campaign. It also ❹ instructs the *Load Generator* to stimulate with a workload the System Under Test (SUT), namely the cluster of controller instances. Finally, ❺ the *Data Collector* stores experiment log data, raw results and basic metrics in a repository, for subsequent higher-level metrics computation and visualization.

4.3.2 Tool Suite

The automation of the tasks of a campaign of benchmarking experiments is provided by the following SCP-CLUB tools:

- The **Campaign Manager** (CM);

- The **Experiment Manager** (EM);
- The **Load Generator** (LG);
- The **Topology Manager** (TM);
- The **Data Collector** (DC).

The *CM* manages a campaign based on a user-specified description of the experiments, which is interpreted by the *Campaign Compiler* (CC). Before each experiment, the *CM* automatically selects and prepares virtual machines (VMs) from a set of templates, prepares storage area, and selects the hardware from a pool of available cloud resources. Figure 4.2 shows an example of the user-provided specification: it defines experiment parameters, such as number and duration of runs, number of controllers, workload, type of topology.

The *EM* receives an experiment profile from the *CM*, and automatically installs and configures all entities in the cloud SDN infrastructure (controller instances, monitoring and data collection services, topology).

The *LG* emulates the arrival of requests at the northbound interface of the SDN controllers in the SUT. It is decoupled from the SUT, avoiding any dependencies from the several implementations of the northbound interfaces proposed by the today's SDN controllers (e.g., ONOS® and ODL®).

```
experiments:
  -
    id: 5Medium_100
    runs: 5
    cell: "five_controller_cell"
    duration: 5
    warmup: 300
    intentTraffic: 1Kb
    networkTopologyInfo:
      topologyType: real
      topologyName: realTopo
      switches: 17
      hosts: 35
      links: 32
    loadGenerationInfo:
      batchSize: 2000
      requestPerTask: 10
...
  -
    id: 3Large_100
    runs: 5
    cell: "three_controller_cell"
    duration: 5
    warmup: 300
    intentTraffic: 1Kb
    networkTopologyInfo:
      topologyType: emulated
      topologyName: linear
      switches: 10
      hosts: 5
      links: 18
    loadGenerationInfo:
      batchSize: 1000
      requestPerTask: 10
```

Figure 4.2: Sample SCP-CLUB experiments specification.

The load consists of a set of policy-based management requests, the so-called *intents* [28], which are request objects carrying the requirements of an application demand to the SDN [25]. The *LG* feeds the SUT with a mix of intent executions and withdraw. The controller(s) translate intents through a compilation process into instructions specifying how the network devices in the data plane have to be programmed. The *LG* can be configured to stimulate the SUT with various workload types. It also computes some basic performance measurements, such as the *throughput* to measure the number of "Intents operations per unit time" (IPS) and the *latency* with which such operations are accomplished.

The *TM* sets up the topology of the data plane in an experiment; it can be either real or emulated.

Finally, the *DC* gathers experiment data from a number of sources, including VMs and controllers logs, workload metrics, resource usage metrics (e.g., CPU, memory).

4.4 ONOS-based SCP-CLUB

4.4.1 The Open Network Operating System

This section describes the implementation of the SCP-CLUB framework centered around ONOS® [10], the popular open source SDN controller developed in Java by the *Open Networking Lab* [27] and hosted by the *Linux Foundation*. ONOS has been chosen for this study as it is designed targeting service providers' network requirements of scalability, high availability, and policy-driven network programmability. It is built atop *Apache Karaf* [65], a Java ® *OSGi* ™ (Open Service Gateway Initiative) container providing a modular run-time environment.

The control plane is deployed on a cluster of servers, running the same ONOS software. As depicted in Figure 4.3, ONOS provides a three-tiered architecture consisting of three main "subsystems", namely the core and the northbound and southbound protocols. The various instances make up the ONOS *distributed core*, providing applications with a centralized logical view of the network. The distributed core design choice is meant to support scalability (instances can be added incrementally and dynamically, that is

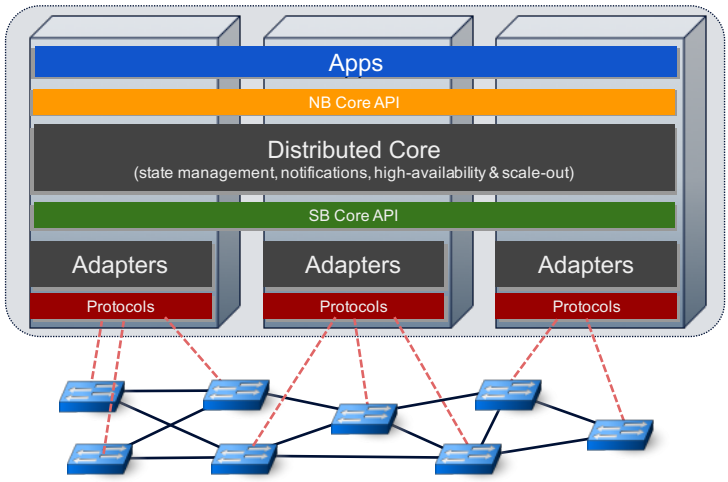


Figure 4.3: ONOS distributed architecture.

without interruptions), as well as availability by enabling rapid failover in case of failure of an ONOS server, through proper recovery protocols.

The presence of cooperating instances is transparent to applications atop, as well as to switches in the data plane. Every device has one *master* and may have multiple *backup* instances. ONOS includes a distributed *leader election* algorithm to assign a single master to each device. Applications program the network at a high level of abstraction through the northbound interface, submitting their *intents* [10] in the form of policy statements or connectivity requests, such as: i) *Set up a connection between host A and host B*; ii) *Do not allow communication between host A and host B*. At the southbound interface, ONOS supports several communication protocols, to allow the interaction with different types of physical network devices and to support legacy equipments.

ONOS adopts two type of consistency models to guarantee the data replication consistency between the replicas, namely:

1. **Eventual Consistency**: This model is used to achieve high availability. It provides a weak consistency guarantee, in the sense that data updates made by a controller are "*eventually*" stored by all the controllers [66]. This implies that for some time the nodes can be in an inconsistent state, but then they converge towards a common state. To support this model, ONOS incorporates an "*anti-entropy*" protocol, which relies on a simple gossiping algorithm [67]. According to this, each controller randomly chooses (every 5 s) another controller to exchange information about any updates of shared data. By doing so, the nodes make sure they gather a fresh copy of the data.

2. **Strong Consistency**: This model ensures high consistency, at the expenses of availability (CAP theorem [68]). Updates are written to all nodes, and reads are guaranteed to return the most recent value regardless of the updating node. This way, controllers are guaranteed to be always in a consistent state, as it is not allowed to read data before they are updated on all nodes. ONOS offers this consistency model by means of *Atomix/Copycat* [69], a fault-tolerant state machine replication framework built on the *Raft* consensus algorithm [70] [71].

ONOS uses the strong and eventual consistency models to implement

two of the main distributed datastore, namely the *mastership store* to keep track of the switch-master node relationship, and the *network topology store* to keep track of the network topology (e.g., switches, links and hosts), respectively. Clearly, state consistency affects the overall ONOS performance.

4.4.2 ONOS-based SCP-CLUB Architecture

Figure 4.4 shows the logical architecture of the ONOS-based SCP-CLUB system. Multiple controllers can be installed and configured to manage the switches in the data plane. Each ONOS instance runs on its own cluster node, in an Apache Karaf container atop the Ubuntu operating system within a virtual machine. VMs use the `vmware`® technology; they are installed by *CM* on the bare metal (i.e., with no operating system on the node). On each cluster node, two data collection daemons monitor and records the events local to the ONOS instance and to the entire VM.

Separate cluster nodes host the Experiment Manager, the Load Generator, the Topology Manager and the Data Collector. The *TM* uses Mininet for emulated topologies [72]. For real topologies, it uses the `centec networks` [73] technology. The *DC* is implemented by a Monitoring Server in a VM running on a server. It uses the InfluxDB (by InfluxData [74]) and Grafana (by Grafana Labs [75]) open source platforms for monitored data storage and visualization, respectively.

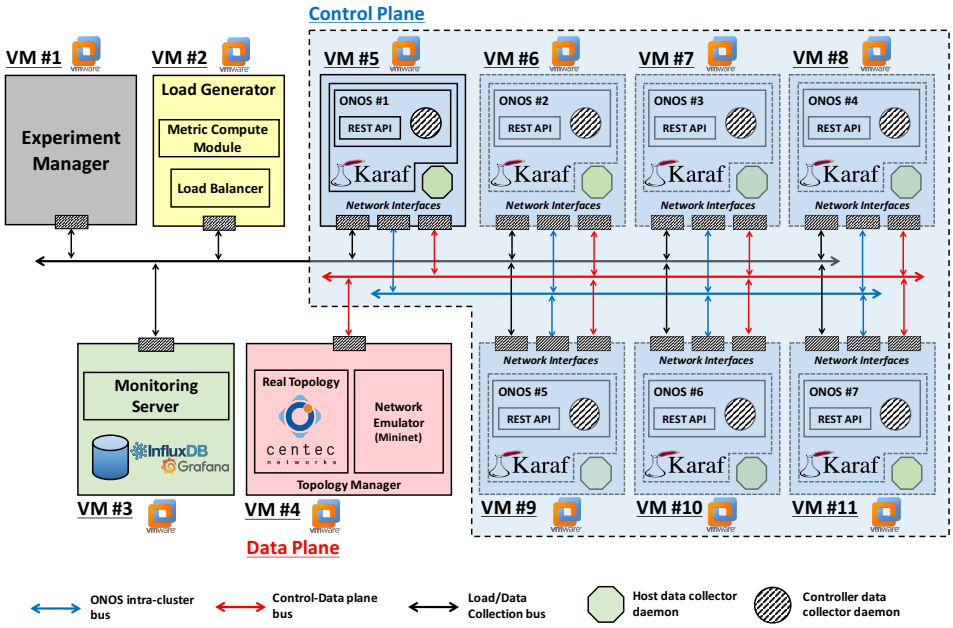


Figure 4.4: Architecture of the ONOS-based SCP-CLUB framework.

4.4.3 Campaign Manager

The *Campaign Manager* (CM), is in charge of creating and configuring the SDN infrastructure (see Figure 4.4). It is a *Python-based* tool capable of interacting with a *VMware*-based virtualization platform. It uses the *VMware vSphere* and *ESXi* API [76] to actually configure and deploy the VMs to host each component of the SCP-CLUB framework.

We designed the *CM* to be as much extensible as possible, supporting different virtualization technologies (e.g.,*KVM* and *Xen*), requiring the user to only provide an implementation of the interaction of the *CM* with such virtualization platform.

The *CM* allows users to provide their own *experimental campaign specification* file. Figure 4.2 shows an example of the user-provided specification: it is a *YAML* file enclosing both deployment and workload specifications, which are also used to provide the *Experiment Manager* with the *experiments profile* file.

The deployment specifications are defined by means of bash snippets (the "cell" parameter in Figure 4.2) including information related to the VMs to deploy, such as *vm_size*, *vm_type*, *instances*, and *servers*. *vm_size* and *vm_type* specify the size (i.e., small, medium, large, or extra large) and type (e.g., SUT VMs, EM, LG, etc.) of the SCP-CLUB VM, respectively, while *instances* defines the number of VMs to deploy for each type, and *servers* includes user preferences concerning the blade servers for the VMs.

According to the *deployment specifications*, the *CM* first identifies the blade servers to deploy the VMs. The servers are selected either according to the available resources, or as specified by the user. In the later case, if no resources are available on the chosen servers, the *CM* automatically selects from the pool the servers having enough resources to accommodate the VMs. Then it creates the VMs from a predefined template deploying them on the selected servers. Finally it configures the network interfaces (e.g., assign static IP addresses) of each VM and checks whether they are able to interact with each other.

4.4.4 Experiment Manager

The *Experiment Manager* (EM) is a lightweight *Java-based* – and *Spring Boot 1.4*® [77] – application aiming to control the experimental campaign. The parameters for each experiment of the campaign are defined by the *experiments profile* file, which is provided by the *CM*, and constructed according to the user specifications.

To actually perform each experiment of the campaign, the *EM* follows a three-phases procedure, namely:

i **start-up phase**: the *EM* deploys the SDN controller instances, and waits until the cluster is correctly formed by continuously checking the status of each instance. Then, the *EM* starts the *DC* server tool, and configures the *LG* with a predefined *load profile*. Once the tools activation is terminated, the *EM* interacts with the *TM* to configure the data-plane topology (e.g., starting the emulated topology with Mininet). Finally, the *EM* checks that the controller instances correctly discovered the data-plane topology and forces the balancing of the mastership of the devices between the deployed controllers. By doing so, it makes sure that each instance is defined as master of at least one device, and as backup controller for other devices. If any of the described operations fails, the *EM* clean-up the SDN infrastructure and repeats (for a predefined number of times) the start-up phase

Algorithm 1 start-up phase

```
Input: ExperimentProfile ep, LoadProfile ld,
    TimesToRetry N
Output: PhaseStatus ps
 1: retry = N
 2: ds = ep.DeploymentSpecification
 3: ps.isComplated = false
 4: repeat
 5:    isSDNs = deploySdnControllers(ds.controllers)
 6:    if isSDNs then
 7:       isLoader = deployLoader(ds.loader, ld)
 8:       isCollector = deployCollector(ds.collector)
 9:       if (isLoader and isCollector) then
10:          startDataPlane(ep.dataPlaneInfo)
11:          if each SDN controller detects the Data Plane then
12:             {balance the mastership of the devices between the SDN con-
                trollers}
13:             balanceDevicesMastership(ep)
14:             ps.isComplated = true
15:             break
16:          end if
17:       end if
18:    end if
19:    retry.decrement(1)
20: until (not ps.isComplated and retry > 0 )
21: if retry == 0 then
22:    print "Experiment with ID ep.experimentID failed!"
23: end if
```

from the beginning. The steps to start-up the SDN infrastructure for the experimental evaluation are summarized in Algorithm 1;

ii **experimental phase**: the *EM* starts the *DC* daemons (host and controller daemons) on the controller instances, and commands the *LG* to actually start the load; then, if specified in the *experiment profile* file, it waits for the experiment duration, otherwise it waits for

Algorithm 2 experimental phase

```
Input: DeploymentSpecification ds, ExperimentProfile ep
vms = getSdnControllers(ds)
for all vm in virtualMachines do
    activateCollectorDaemnos(vm)
end for
startLoadGeneration(ds.loader)
if ep.hasExperimentDuration then
    sleep ep.ExperimentDuration
else
    waitLoaderTermination()
end if
storeCollectedData()
```

the *LG* to stop. Finally the *EM* instructs the *DC* to permanently store the collected data for subsequent offline analysis. These steps are summarized in Algorithm 2.

iii **clean-up phase**: every experiment is followed by a system clean-up phase, accomplishes by the *EM* by reverting the VMs with a clear snapshot. A further clean-up option, which is supported by the *EM*, is to stop all the tools and clean up the VMs (e.g, uninstalling the SDN controller source code, removing the syslog, etc.). The steps of the clean-up phase are summarized in Algorithm 3.

Algorithm 3 clean-up phase

```
Input: DeploymentSpecification ds, RestoreOption ro
virtualMachines = getDeployedVMs(ds)
for all vm in virtualMachines do
    if ro.isSnapshotToRestore then
        pm = getPhysicalHost(vm.id)
        restoreSnapshot(pm, vm.id, vm.snapshotID)
    else
        cleanUp(ds, vm.type)
    end if
end for
```

4.4.5 Topology Manager

The *Topology Manager* (TM) consists of a set of *Python* and *bash* scripts aiming to set up the data plane topology, either emulated or real. It incorporates the *Mininet* tool [72] to reproduce different types of data-plane, varying the shape (e.g., linear, full meshed, etc.) and the size (i.e., number of switches) of the topology. By interacting with the standard management API (such as the *OpenFlow Switch Management API*) typically exposed by OpenFlow switches, the *TM* is also capable of interacting with specific types of physical devices, such as the *Centec* switches, configuring them to operate with the ONOS controllers, thus forming a real data plane topology.

4.4.6 Workload Generator

We designed the *Workload Generator* (*LG*) as an extensible *Java-based* – and *Spring Boot 1.4* – application, aiming to reproduce realistic workload

conditions for the SUT. As anticipated earlier in this chapter, the LG emulates policy-based application requests, namely *intents* specifying high-level requirements which have to be satisfied by the controller by programming network devices, after a *compilation* step.

The Intent is the main concept beyond the **Intent-Based Networking** [25] (IBN) model, where an application can simply specify its own network requirements without worrying about the underlying network details (§2.1.4). According to the IBN model, the application specifies what it requires in terms of action to be taken in controlling the network, ignoring how such needs are actually implemented by the SDNs. Hence, the IBN framework underlying the SDNs abstracts the network complexity allowing network operators and applications to describe global network policy rather than low level instructions for the devices.

More in detail, an Intent is an immutable request model carrying the requirements of an application's demand to the SDN platform, which involve the modification of the network's behavior. As mentioned in §2.1.4, Intents are portrayed at lower level as network resources, constraints, criteria and instructions for specifying the network policies. To enforce such policies, there is a need to configure the physical network infrastructure, and the appliances. To this end, once received, the Intents requests are translated

through the *intent compilation* process into corresponding device instructions specifying how the network must be programmed at low level. A *point-to-point interconnection* request with default network constrains, e.g. minimum network bandwidth, is an example of Intent request handled by the SDNs which require the creation of a connectivity between endpoints. Such a request, is converted, i.e. compiled, by the SDNs into a set of flow rules to install on the corresponding network devices.

In summary, an Intent is submitted, then - providing the request is feasible, i.e. it can be satisfied - it is compiled, installed and executed. Network problems, such as loss of throughput or connectivity, may impact a compiled intent, in which case the controller tries to recompile it looking for an alternate approach to satisfy it. Intents may also be withdrawn intentionally by the applications requesting them.

The *LG* is capable of emulate two type of intent requests:

- **Install intents**, carrying requests to set up new connections; these are further categorized into:
  - *Host-to-host* intents, requesting a bidirectional connection between two hosts in the data-plane. They are compiled into two paths, one per direction;
  - *Host-to-multihosts* intents, for bidirectional connections between a source host and multiple destination hosts. They are compiled

into path pairs.

- **Withdraw intents**, carrying request to cancel a connection previously successfully installed and set up.

Initially, the *LG* interacts with one of the available ONOS instances to discover the network topology. Each request is created by randomly selecting two edge switches (i.e., switches having at least one host connected); for each of them, the *LG* randomly selects a host connected (or multiple hosts for a point-to-multipoint intent). Although it is legitimate for an intent to involve hosts connected to the same switch, the *LG* creates requests involving only hosts connected to different switches, to actually trigger the creation of a network path.

The requests are submitted in batch units. A batch can carry both install and withdraw intents. The *LG* allows requests to be submitted to a target ONOS instance in synchronous or asynchronous mode; in the latter case, a *callback* task is provided to process the reply, thus increasing the throughput of the *LG*. The *LG* manages a pool of TCP connections for each controller instance, and reuses them to send its requests, hence reducing the overhead.

The *LG* incorporates a load balancing algorithm to identify the target ONOS instance for an intent. Specifically, it uses a *Weighted Round Robin* policy, with each server weighted according to its response time. This way, the longer the response time, the lower the weight an instance is assigned.

Then, the policy randomly picks a server according to the weights. An alternative simple Round Robin policy is also provided for further experiments.

For each generated request, the LG reports the corresponding performance measurements, i.e., processing throughput and latency. In computing such measurements, an intent request is considered timed out, i.e., failed, if no response is received after 10 seconds.

The LG provides three operational modes:

1. **Impulse Response mode**. The LG submits a set of $Smax$ subsequent impulses of requests. Each impulse consists of two batches of Rx install, and withdraw requests, respectively. The LG first submits the batch of install requests, waiting until the process terminate. The waiting cycle ends when: $IntentThroughput + IntentFailureRate = Rx$, with $IntentThroughput$ being the rate of requests correctly handled by ONOS, and $IntentFailureRate$ the rate of the failed requests. Then, the LG submits the batch of withdraw requests and, as for the install batch, waits the withdrawal phase terminates. Finally, it pauses for a time interval T before starting a new impulse of requests.

2. **Steady-State Response mode**. The LG constantly generates and submits a batch of Rx requests per unit time. Each batch contains both type of requests, i.e., intent install and withdraw requests. Regardless of the type of submitted requests, the LG ensures that only

Rx requests will be submitted per unit time. It uses a token bucket mechanism to control the load, generating new requests only if the token is non-empty. It is worth noting that by intentionally adopting high request rates we can assess if the system is able to tolerate unpredictable legitimate (e.g. flash crowd), or malicious (e.g. DDoS attack) high level Intent traffic without a significant performance loss.

3. **Probabilistic Mode**. The *LG* submits intent installation and withdraw requests following a specific probability distributions, i.e., an approximation to a *Poisson Markov* process with exponentially distributed inter-arrival times. This is achieved by discretizing the time domain as successive intervals of length *dt* and, in each time interval, performing an independent *Bernoulli* trial with a success probability of $\mathbf{p} = \mathbf{r} * \delta\mathbf{t}$ (a coin flip with probability p), where $\mathbf{r}$ is the *event rate* (requests/sec). In other words, it submits an intent install and withdraw requests according to an *Install Request Rate*, and *Withdraw Request Rate*, respectively. The time between two consecutive submissions of install or withdraw requests is given by the *Install Update Time*, and *Withdraw Update Time*, respectively.

The operational mode to adopt for experiments and the corresponding parameters are defined in the *load profile* file provided by the *EM*. In any

operational mode, the *LG* can be configured to emulate network traffic between two hosts of an intent, be they actual hosts in a real topology, or emulated with Mininet. To this aim, it exploits the *iperf3* [78] tool.

It has to be pointed out that the experiments described in this dissertation have been conducted exclusively with the *LG* configured in *steady-state* working mode, leaving the other options for future in-depth analysis.

4.4.6.1 Data Collector

The *Data Collector* (DC) allows on-line analysis of the *SUT* by continuously collecting data from all the target nodes. These measurements allow the user to conduct in-depth analysis of the system behaviour, helping in identifying the route causes of possible performance degradation. A detailed description for each collected metric is reported in Table 4.1. In achieving his goal, the *DC* relies on two monitoring tools (see Figure 4.4):

(i) **Host Daemon** – It is lightweight *Java-based* application running on the hosting machine. It collects finer-grained measurements at both *(i) system-level*, to check the overall status of the VMs hosting the controller instances, and at *(ii) process-level*, to identify possible bottleneck that can cause performance degradation of the SDN controllers. Examples of measurements collected by the *Host Daemon* are *CPU*, *memory*, *disk* and *network* utilization.

(ii) **Controller Daemon** – Designed as a controller-side application, this agent periodically intercepts the *events* occurring at both *instance-level*, i.e. in the local ONOS instance, and *cluster-level*, i.e. across all the controller instances. Examples of intercepted events are data plane, IBN, and cluster communication events. The classes of events intercepted by the *Controller Agent* are listed in Table 4.1. This agent collects also some basic statistics about the ports of the physical network devices, such as the total packets (bytes) received and sent, the number of transmitted and received packets drops.

The data collected by the two daemons are periodically sent to the **Monitoring Server**, a lightweight *Java-based* application running on a separate VM, to be (permanently) stored for both on-line and off-line analysis.

Table 4.1: *System* and *process* metrics, and *controller*-related events collected during an experimental campaign.

	Metric Name	**Description**	**Additional Information Supplied**
System Metrics	**CPU Usage**	System-wide actively used CPU as a percentage of total available.	Additional finer-grained metrics: user, system, idle, nice, iowait etc.
	Virtual Memory Usage	System-wide actively used memory as a percentage of total available.	Additional finer-grained metrics, expressed in bytes: total, used, free, active, inactive, cache etc.
	Swap Memory Usage	System swap memory usage as percentage of total available.	Additional finer-grained metrics, expressed in bytes: total, used, free, swap in, swap out.
	Disk Statistics	System-wide disk I/O usage statistics.	The collected metrics include: number of read/write, the number of bytes read/write, the disk read/write/busy time.

Continued on next page

Table 4.1 – *Continued from previous page*

	Metric Name	Description	Additional Information Supplied
	Network Statistics	System-wide network I/O statistics.	The collected metrics include: bytes/packets received/sent, total number of error/drop of incoming/outgoing packets.
Process Metrics	**CPU Usage**	Process-wide actively used CPU as a percentage of total available.	
	Heap Memory Usage	Process-wide actively used heap memory as a percentage of total available.	The actual and max number of used bytes are also reported.
	Non-Heap Memory Usage	Process-wide actively used non-heap memory (e.g. stack memory) as a percentage of total available.	The actual and max number of used bytes are also reported.
	Pending Finalizations	Number of object waiting to be finalized, i.e. definitively deallocated from the memory.	This metrics, along with the heap and non-heap memory metrics, is useful to identify possible memory leaks the process may suffer.
	Thread Count	Number of currently active threads started by the process.	
Controller Metrics	**Cluster Event**	Rate at which the SDN cluster related events occur.	Raised when a controller instance joins/leaves the cluster, or when the instance becomes operative/inoperative.
	Cluster Communication Events	Rate at which the communications occur within the cluster.	Raised when a controller sends/receives a message from one or more nodes of the cluster.
	Network Devices Events	Rate at which the network devices related events occur.	Raised when a physical device joins/leaves the data plane, or when a status update of an existing device is detected, e.g. physical ports changes.
	Network Links Events	Rate at which the network links related events occur.	Raised when a physical network link is discovered/removed or when a link status updated is detected.
	Network Hosts Events	Rate at which the network hosts related events occur.	Raised when an end-station joins/leaves the data plane, or when a status or physical location update of an existing end-station is detected.
	Network Topology Events	Rate at which the network topology related events occur.	Raised when the graph view of the network topology changes, e.g. when a new disjoint shortest path is created or updated.
	IBN Events	Rate at which the Intent related events occur.	Raised when an Intent[a] request is processed by the Intent-based networking framework. These events are related to the IBN compilation process, thus example of events are: Intent installed, withdrawn, failed etc.

Continued on next page

Table 4.1 – *Continued from previous page*

Metric Name	Description	Additional Information Supplied
Flow Rule Events	Rate at which the flow rule related events occur.	Raised when the forwarding tables of an OpenFlow device is updated, e.g. when a flow-rule is added, updated or removed.
Leadership Events	Rate at which the leadership related events occur.	Raised when a leadership election for a specific event[a] is started, or when an event's leader has changed.
Mastership Events	Rate at which the mastership related events occur.	Raised when the mastership (backup[b]), i.e. the SDN controller instance that act as a master, of a physical network device changes.
Distributed Consistent Events	Rate at which the distributed consistent related events, i.e., events that require strong consistency guarantee, occur within the cluster.	Raised when an operation needs to be processes with strong consistency guarantee, e.g., network topology events, device events etc.
Control Plane Statistics	Per-device counter statistics of transmitted/received OpenFlow messages.	The collected statistics are related to six types of control messages: PacketIn, PacketOut, FlowMod, FlowRemoved, StatsRequest, StatsReply[c].

[a] Examples of events that raise a leadership election are the mastership of a physical device and the management of Intent requests.
[b] In a cluster of *N* controllers, an OpenFlow physical device is mastered by a single controller while the remaining *N-1* controllers act as failover.
[c] PacketIn: sent from the devices to the controller; PacketOut: injected from the controller into the devices; FlowMod: to modify the state of the switch; FlowRemoved: sent to communicate to the controller when a flow entry in a flow table is removed; StatsRequest: used by the controller to request information about individual flows; StatsReply: used by the devices to respond to StatsRequest packets.

4.4.7 Workflow of an intent

It is useful to look at the processing of an intent by ONOS, to understand how SCP-CLUB differs from the reviewed ONOS benchmarking [45] [46] [49] [48] and testing [64] approaches.

Figure 4.5 shows the ONOS internal steps to elaborate an intent install request. Each request arriving at the northbound interface of an ONOS instance (step 1) is served by the *Intent Service*. The instance adds it to

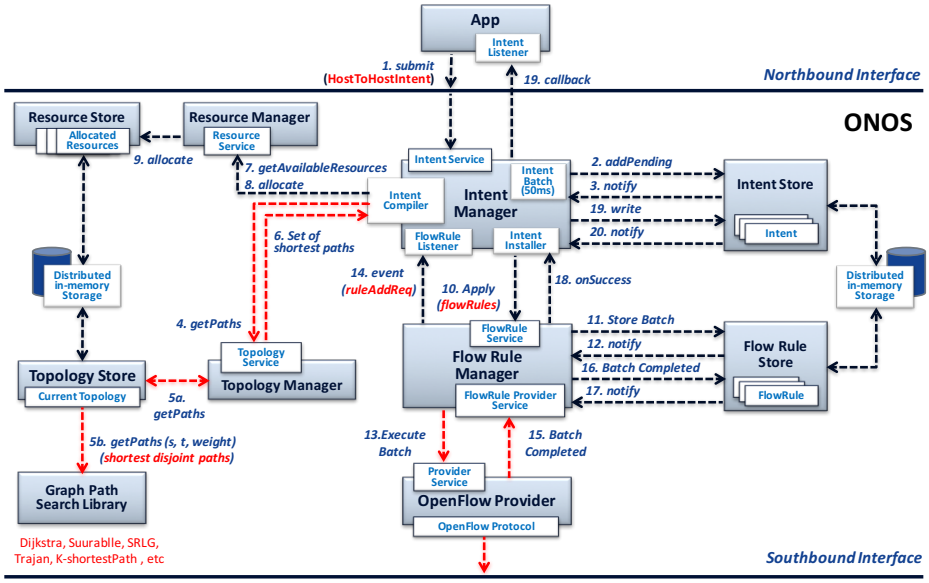


Figure 4.5: The ONOS internal workflow for an intent.

the distributed in-memory storage (step 2). The latter in turn notifies the cluster replicas (step 3) with the new event, triggering the assignment of the request to a cluster instance.

The requests are accumulated in a single batch (with a default max size of 1,000 requests) to reduce the processing overhead. A batch is processed either when it reaches its maximum size, or after a specified timeout (the default is 50 ms). Each intent is compiled by the *Intent Compiler*: this looks for the available network paths (steps 4, 5, and 6) and resources (steps 7 and 8) to generate the set of flow rules for the switches.

The Intent Manager interacts with the *Flow Rule Service* to install the flow rules. This service splits the rules in subsets for each specific device.

The rules are stored (step 10) into the distributed in-memory *Flow Rule Store*, and all cluster replicas are notified (step 11). Since the network devices can only be programmed by their master controller, each ONOS node is responsible to only install the subsets of rules related to the controlled devices. For instance, if the main batch contains flow rules for switches *d1* and *d2*, it is split in two units - *b1* and *b2*. Assuming *d1* and *d2* are mastered by controller $C1$ and $C2$, respectively, the flow rules in *b1* are installed by $C1$, while those in *b2* are sent to the master of *d2*, i.e., $C2$.

The flow rules are actually installed in the devices (step 12) by means of a corresponding southbound provider implementing the device communication protocol (e.g., OpenFlow). Once the devices have been correctly programmed (step 13), the *Flow Rule Manager* updates the distributed store (step 14), which in turn notifies the cluster nodes (step 15) with the new flow rules related events. Then, the *Intent Manager* is notified (step 16) and it updates the store with the new intent related events (step 17), triggering the notification of the cluster nodes (step 18). Furthermore, the events related to an intent request (e.g., intent installed, withdrawn) can also be intercepted by the northbound application (step 19) by means of a corresponding intent event *Listener*.

The processing of a withdraw request differs from the above only for the steps concerning the resource reservation. Indeed, the withdrawal of an

intent requires to return to the pool the network resources allocated to it.

ONOS adopts an *eventual consistency* [66] algorithm among the distributed nodes for intents and flow rules related events. The information are first stored locally, and then eventually propagated within the cluster. Consistency mechanisms clearly influence performance when scaling out controllers; moreover, inconsistencies between the control and the data planes may cause failures in the actual installation of intents [79].

To analyze the actual performance of an SDN deployment in a telco cloud, the ONOS-based SCP-CLUB framework has been designed with several essential differences with respect to the way ONOS performance tests are typically conducted:

- Most controllers performance tests use the so called "Null Providers" [64] or faked devices [48], i.e. services that fake the data plane (devices, hosts, links, topology). This way, the control-data plane communication is totally avoided (steps 13 and 15, including the communication via the southbound interface), and the flow rules are saved locally to the Null Provider. As pointed out in [49], the controller processing time varies with the sojourn time: SCP-CLUB computes performance metrics considering all steps shown in Figure 4.5 (indeed the dotted red lines in Figure 4.5 represent the steps skipped by most of ONOS performance tests);

- When compiling a host-to-host intent, the ONOS Intent Manager identifies a bidirectional shortest path connecting the hosts. Then, the path is compiled in sets of flow rules to be installed on devices in the path. In the ONOS performance tests in [64], intents are compiled in a single flow rule to be installed on one device, regardless of the type and size of the network topology. Consequently, steps 4, 5 and 6 are bypassed. The SCP-CLUB Load Generator produces realistic requests for real or emulated topologies, which traverse the whole intent workflow within controllers, hence the performance metrics computed take into account the actual processing of intents;

- When installing an intent involving devices managed by different ONOS instances, the instance responsible of the intent needs to communicate with the others to confirm the termination of the intent installation process. Again, the metrics computed by SCP-CLUB take into account all steps in the processing of intents, thus including the management of intent and flow rules stores, and the coordination among controller instances.

4.4.8 Capacity Measurement

The definition of a capacity for an SDN controller, such as ONOS, may vary according to the performed functionality, and the type of workload

(e.g., intent or flow rule installation). This experimental analysis is meant to identify the capacity of the ONOS IBN framework under different working conditions (e.g., workload, deploy and data-plane size), and verify its capability in satisfying the Telco Service Level Agreements (SLA). The SLA imposes stricter constraints to the SDNs, asking for low latency operations (e.g. recovering from a failure within a 50 ms interval), while keeping a high processing throughput.

Consequently, the most important metrics considered in this study are the service *throughput* and *latency*, which are computed as follows:

- **IBN Service Throughput** (IST). The throughput is the rate at which requests are successfully processed by the ONOS cluster, e.g., number of intent installed and/or withdrawn per unit time. It represents the maximum workload at which the controller(s) can provide the desired stable service. The service throughput of each experiment is given by:

$$IST = \frac{Intent_{req}}{t_{LI}^{res} - t_{FI}^{req}} \qquad [requests/s], \tag{4.1}$$

  where t_{LI}^{res} and t_{FI}^{req} refer to the time of the *last* Intent response and of the *first* Intent request, respectively, and $Intent_{req}$ is the number of requests correctly processed by the system.

- **IBN Service Latency** (ISL). The latency is the time required by the ONOS controllers to process an intent request, which can be measured

as follows:

$$ISL = t^{res}_{Intent} - t^{req}_{Intent}, \tag{4.2}$$

where t^{req}_{Intent} is the time at which the intent request enters a controller instance, and t^{res}_{Intent} the time at which the request is successfully completed (e.g. the intent is compiled, and the network devices are programmed), or an error is raised, due to timeout expiration or controller failures.

More in detail, the t^{res}_{Intent} provides information about the IBN processing overhead. Let C and S be the number of deployed controller instances and data-plane switches, respectively, and m be the controller which is the master of an Intent request (i.e., the controller that received the request). If $C = 1$, or the request is compiled into flow rules for the only switches managed by m, then the processing overhead is composed by:

$$t^{res}_{overhead} = t^{m}_{overhead} = t_{controller} + t_{switch} + t_{link}, \tag{4.3}$$

where $t_{controller}$, and t_{switch} are the processing delays in controller and switch, respectively, and t_{link} is the transmission delay on the link (i.e., the TCP connections) between the controller and the switches. Further overhead is added when the flow rules refer to switches managed by several controller instances. Indeed, in such a case an intra-cluster

collaboration is required to actually process the intent request. Then, the above eq. (4.3) becomes:

$$t^{res}_{overhead} = t^{m}_{overhead} + \sum_{\substack{i \in C_m \subset C \\ i \neq m}} \sum_{j \in S_i \subset S} \left(t^{i}_{controller} + t^{j,i}_{switch} + t^{i,m}_{link} + t^{i,j}_{link} \right) \tag{4.4}$$

where:

C_m = subset of the controllers involved by m in the request processing
S_i = subset of switches managed by the i-th controller ($i \in C_m$) where installing a subset of flow rules
$t^{m}_{overhead}$ = the processing delay due to the only controller m
$t^{i}_{controller}$ = the processing delay due to i-th controller
$t^{j,i}_{switch}$ = the processing delay due to j-th switch ($j \in S_j$), managed by the i-th controller
$t^{i,m}_{link}$ = the transmission delay on the link connecting the i-th controller with the controller m
$t^{i,j}_{link}$ = the transmission delay on the link connecting the i-th controller with the managed j-th switch.

Figure 4.6 displays a simplified architecture of the SDN ecosystem. The request arriving at the NB interface are first processed by the ONOS core to be compiled in corresponding flow rules, then the latter are programmed in the switches by means of the SB interface. In addition, the OpenFlow switches has a two-layer design, with a user space daemon interacting with the controllers, and a kernel module to actually set the rules in the flow tables.

Each of these steps introduces additional latency, which is further

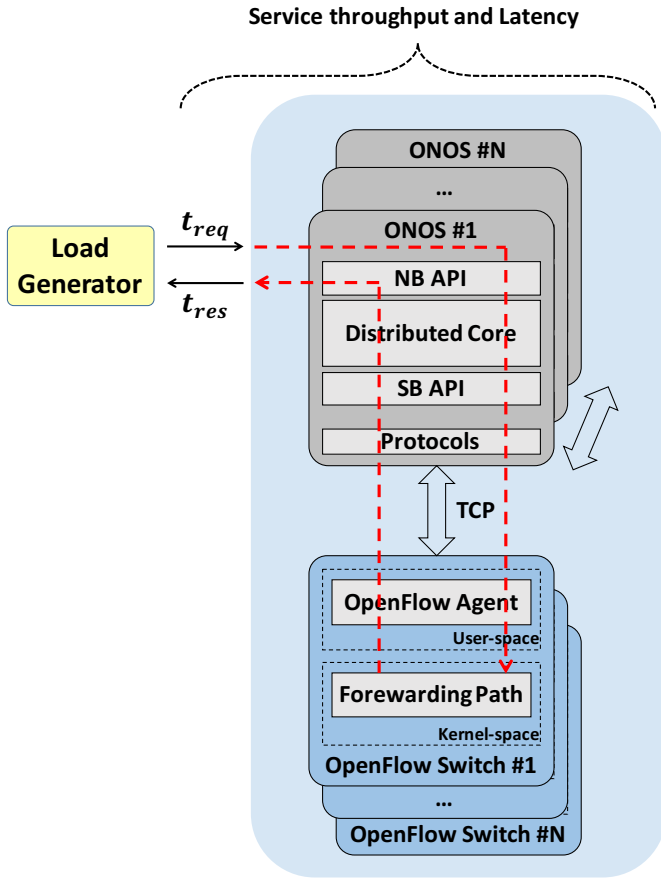


Figure 4.6: The ONOS service performance measurements.

increased when multiple controller are involved in processing the requests. In addition, the dominant latency factor may differ according to the specific working scenario. For example, t_{link} is the dominant factor in a WAN deployment, while it is kept tiny in a data center scenario, where t_{switch} is the main factor. In the proposed test environment, the latency measurements only accounts for controllers and switches aspects, while the influence of the connection is negligible.

The above equations 4.3 and 4.6, can thus be simplified to:

$$t^{res}_{overhead} = t^{m}_{overhead} = t_{controller} + t_{switch} \tag{4.5}$$

and to:

$$t^{res}_{overhead} = t^{m}_{overhead} + \sum_{\substack{i \in C_m \subset C \\ i \neq m}} \sum_{j \in S_i \subset S} \left(t^{i}_{controller} + t^{j,i}_{switch} \right) \tag{4.6}$$

These metrics are the primary indicators of the system's overall heath, allowing to identify if the system is "*working properly*". Indeed, under normal working condition, i.e. without any disruptive condition affecting the system, the indicators will variate within a standard range of values. Viceversa, abrupt and unexpected changes of these indicators will be representative of degraded performance conditions which can be related to a variety of problems, such as resource depletion, bottlenecks, and failures.

4.5 Benchmarking a telco cloud SDN

4.5.1 Experimental Campaign

In order to analyze the capacity of a SDN deployment in a telco cloud infrastructure, we run experiments by varying the following parameters:

- **vertical scale (VSCALE)**: the resources - namely, CPUs and memory - allocated to each controller instance (scaling up/down);

Table 4.2: Levels of the DOE factors. The LOAD factor ranges between 1,000 and 5,000 requests/s with increments of 1,000 requests/s.

		Levels			
Factors	**VSCALE**	SMALL	MEDIUM	LARGE	XLARGE
	HSCALE	1	3	5	7
	LOAD	1,000 req/s [MIN]		5,000 req/s [MAX]	
	TSIZE	SMALL		LARGE	

- **horizontal scale (HSCALE)**: the number of SDN controllers (scaling in/out);
- **load level (LLEVEL)**: the number of requests arriving at the northbound interfaces;
- **topology type (TSIZE)**: the size of the data-plane network topology - namely, the number of devices, links and hosts.

In the following the terminology of the *Design of Experiments* (DOE) method [80] will be used. According to the DOE terminology, the VSCALE, HSCALE, LLEVEL and TSIZE parameters are referred to as *factors*, while the values each factor can take are called *levels*. Table 4.2 shows the levels of each factor. The levels have been used to configure the *CM* and the EM, in the ONOS-based implementation of the SCP-CLUB framework described in Subsection 4.4.

4.5.2 Telco Cloud Experimental Setup

An experimental telco cloud infrastructure has been set up at Nokia Bell Labs (Figure 4.7). It encompasses three racks, each equipped with 20 blade servers, connected to two high-speed subnetworks: *i)* the *management subnetwork*, used exclusively to configure and control the blade servers, and *ii)* the *data network*, interconnecting the servers. Each server is connected to two 1/10GbE Ethernet Nokia Management switches, for the interconnections within a rack, and to two QSFP 10/40GbE Nokia Ethernet switches, for data exchanges within the rack. Communication across racks is ensured by means of two additional QSFP 10/40GbE Nokia Ethernet switches.

Table 4.3 lists the configuration of the testbed machines. The D51BP-1U blade servers are used as compute nodes while the D51BP-2U servers are used as storage nodes. The server virtualization is provided by means of VMware ESXi [81] (ESXi v 6.5), a bare-metal hypervisor, while VMware Vsphere is used to automate the management of virtualized resources.

The deploy consists of ONOS version 1.10 instances installed on Ubuntu 16.04. The SCP-CLUB *CM* has been instructed to scale the ONOS controllers up (VSCALE) by varying the flavour of the VMs from *small* to *medium*, *large* and *extra-large*, and out (HSCALE) by varying the number of instances from to *3*, *5* and *7*. To avoid resource overcommitment, the *CM* has been configured to distribute the VMs on different blade servers.

Figure 4.7: The Nokia AirFrame testbed, running the SCP-CLUB tools and the ONOS cluster.

The *EM* is instructed to vary both the load level (LLEVEL), and size (TSIZE) of the data-plane network for performing various experiments in a campaign. Specifically, the *LG* is configured to vary the request rate from *1,000* up to *7,000* requests/s with increments of *1,000* requests/s. The *TM* is set to emulate two type of linear data-plane topology: *i)* a *small* topology, consisting of 10 switch connected linearly, and 10 hosts (5 hosts connected to the end nodes); *ii)* a *large* topology, consisting of 30 switches and 10 hosts. Mininet 2.2 has been adopted to emulate each type of OpenFlow-based network.

Table 4.3: Telco cloud blade servers and VMs configuration.

	Type	CPU	Cores	RAM	Network
Physical Machines	D51BP-1U D51BP-2U	2x Intel Xeon E5-2680 v3	24 (48 logical)	256GB 516GB	1 Intel I350 2x1GbE 1 Intel X540 2xSFP+ 10GbE 1 Dedicated 1GbE Management
Virtual Machines	ONOS.small	2x vCPU	2	2 GB	3x 10Gb vNIC
	ONOS.medium	4x vCPU	4	4 GB	
	ONOS.large	8x vCPU	8	8 GB	
	ONOS.xlarge	8x vCPU	8	16 GB	
	Experiment Manager	2x vCPU	2	2 GB	2x 10Gb vNIC
	Load Generator	8x vCPU	8	64 GB	
	Topology Manager	8x vCPU	8	8 GB	
	Monitoring Server	2x vCPU	2	4 GB	

Besides the experiments with the emulated topology, several tests have been conducted with a real topology consisting of 16x V350 Centec [82] silicon SDN/Openflow switches. The switches are connected linearly, with a total of 35 real hosts wired to 7 switches (5 hosts per switch).

As described in §4.4.4, an experiment encompasses of three main steps, i.e., the i) *startu-up*, the ii) *experimental*, and iii) *cleanup* phases. The experimental phase lasts 300 seconds and each experiment is repeated 10 times to average the results. For each run a trial phase is performed submitting a total of 2000 requests to warm-up the system and reach a steady state. During the trail phase, no performance measurements have been taken.

Table 4.4: Operating system and ONOS parameters configuration.

	Parameter	Value	Additional information
Operating System	rmem_max wmem_max tcp_rmem tcp_wmem	16777216 16777216 4096 87380 16777216 4096 16384 16777216	Sets the TCP read and write buffer size (16MB) for a 10Gb connections.
	net.core.somaxconn netdev_max_backlog tcp_max_syn_backlog tcp_syncookies	4096 16384 8192 1	Sets the size of the socket connection listening queue and the incoming packet queue for upper-layer (e.g Java) processing.
	ip_local_port_range tcp_tw_recycle	1024 65535 1	Sets the number of usable ports and allow reuse of sockets in TIME_WAIT state.
	nofileSoftLimit nofileHardLimit	1050000 1050000	Sets the number of open file descriptors before a soft/hard error is issued.
ONOS	acceptors minThreads maxThreads maxQueuedRequests	# vmCPU 10 400 6000	Sets the Jetty's acceptor[a] threads, the min and max threads of the thread pool, and the maximum number of requests to queue before blocking the acceptors.
	JVM Heap JVM GC	vmRAM ConcurrentMarkSweep	Sets the JVM's initial and maximum memory sizes with the total VM's RAM, and the Garbage Collection algorithm.
	skipReleaseResources	True	Sets skipReleaseResourcesOnWithdrawal to skip the release of the resources assigned to an Intent when it is withdrawn.

The OS parameters have been applied to both *LG* and ONOS VMs, with the network port configurations that have only been applied to *LG* VM.
[a] The Jetty's acceptor threads are meant to accept new connections from the client (the *LG*).

4.5.3 System Configuration

In order to avoid any pitfalls due to an incorrect system configuration (e.g., a small port rage limiting the outgoing connections), the *LG* and the ONOS

VMs have been configured for high load performance testing. Table 4.4 reports the values used to tune both the Operating System (OS), and the ONOS environment parameters.

Several parameters have been set at OS level. For example, the ports range, the TCP window and queue size of the TCP/IP layer, and the Linux file descriptor limit have been increased to support high connection rates. Furthermore, at ONOS level, the number of threads (e.g., max and min number of threads) used by the Karaf's built-in web server (i.e., the Jetty web server) has been tuned to achieve better performance. Finally, the ONOS JVM is configured to use the maximum available memory (i.e., VM's memory), and the "*skipReleaseResourcesOnWithdrawal*" property of the ONOS's *Intent Manager* is set to reduce the overhead due to the release of network resources on intent withdrawal.

4.6 Results

4.6.1 Experiments with Emulated Data Plane

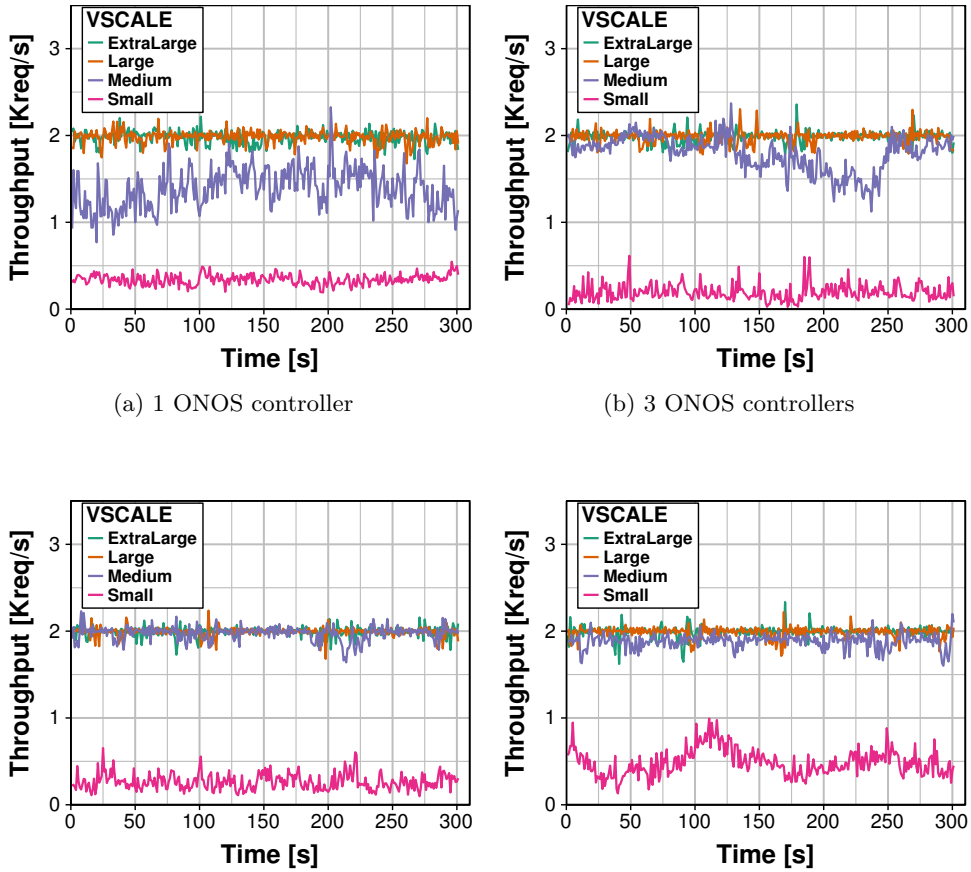
Figure 4.8 shows the throughput over the 5 minutes of duration of experiments with a constant workload of 2,000 intents per second (install/withdraw), for various sizes of the VM, and using from 1 (Fig. 4.8a) to 7 controllers (Fig. 4.8d). Each plot is the average throughput over 10 runs; the total 2,240 experiments lasted about 6 days, including the start-up and

clean-up phases. It can be observed that:

- The system sustains the exact input load (i.e., Throughput ≈ Input Request Rate) only with the LARGE and EXTRALARGE values for the VSCALE factor, independently of the number of controllers;
- For SMALL and MEDIUM deployments, the throughput with one controllers is well below the load rate of 2,000 requests/s, and for the SMALL adding more controllers is insufficient to keep the input pace;
- With 3 controllers and MEDIUM VSCALE factor the load is sustained only for about $100s$, then the throughput drops down.
- Increasing the HSCALE factor may slightly reduce the overall performance, even with the LARGE and EXTRALARGE deployments. This is due to the overhead of the East-West communication, which increases as the number of controllers.

These results suggest that scaling up the VMs vertically seems to be an affordable possible strategy to sustain a constant workload as "high" as 2,000 intents per second.

Table 4.5 shows the maximum IBN System Capacity (ISC) of ONOS when the rate $\eta = \frac{IST}{IRR}$ is grater than a specified threshold, e.g., 0.8 or 0.9, which represents the maximum . The IST is the successful Intent request rate, and IRR is the rate at which the intent requests are submitted to the



(a) 1 ONOS controller

(b) 3 ONOS controllers

(c) 5 ONOS controllers

(d) 7 ONOS controllers

Figure 4.8: Service throughput time series per VM flavour (VSCALE) and deploy size (HSCALE) with a load of 2,000 requests/s.

ONOS cluster, namely, the LOAD factor. The rate η shows how closely the IST follows the IRR specified in our tests, giving an overview of what is the maximum offered workload at which ONOS can provide the desirable stable service. In other words, when the IRR exceeds the system capacity, a certain percentage, say 20 or 10% (0.8 and 0.9 thresholds, respectively), of the intent requests will fail. In such a case, η will be well below the specified threshold

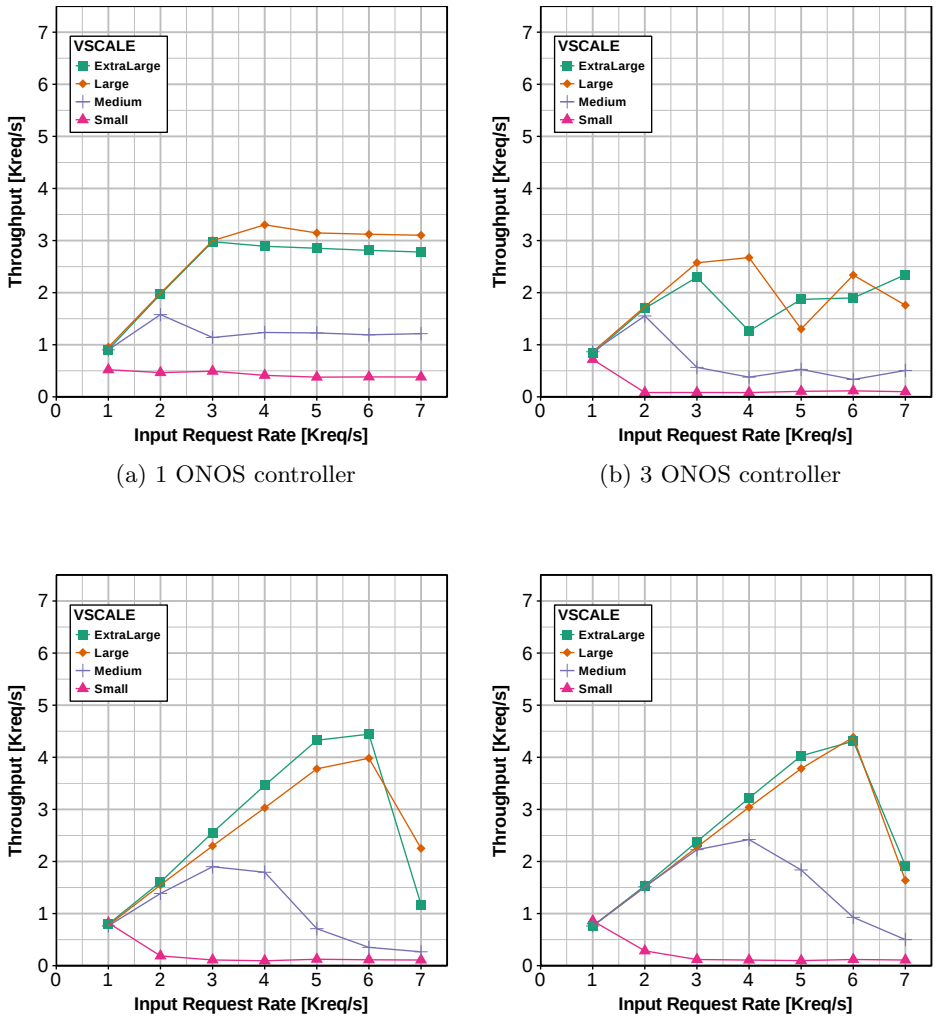
Table 4.5: ONOS Intent Based Networking System Capacity (ISC) using different scaling methods.

ONOS Deploy Size	Resources		ISC max for $\eta \geq 0,8$	ISC max for $\eta \geq 0,9$
	vCPU	RAM		
ONOS–1–Medium	4	4	1,000	1,000
ONOS–1–Large	8	8	3,000	3,000
ONOS–1–Extra	8	16	3,000	3,000
ONOS–3–Medium	12	12	2,000	1,000
ONOS–3–Large	24	24	3,000	3,000
ONOS–3–Extra	24	48	3,000	2,000
ONOS–5–Small	10	10	1,000	1,000
ONOS–5–Medium	20	20	2,000	2,000
ONOS–5–Large	40	40	4,000	4,000
ONOS–5–Extra	40	80	5,000	4,000
ONOS–7–Small	14	14	1,000	1,000
ONOS–7–Medium	28	28	2,000	2,000
ONOS–7–Large	56	56	5,000	3,000
ONOS–7–Extra	56	112	5,000	4,000

suggesting that the system is no longer capable to satisfy the required level of service, e.g., 80% or 90% of the IIR. Hence, the more the η is close to 1 the better the performance.

As for evaluating the scaling ability, Figures 4.9 and 4.12 show the results with emulated topologies with number of switches of 10 (small) and 30 (large), respectively. The graphics plot the throughput (thousands of requests per second) exhibited when soliciting the system with a load from 1,000 to 7,000 requests/s, using an odd number of controllers varying from 1 to 7. The curves represent the average over 10 runs of each experiment.

As for scaling up, a single ONOS instance is used and its size is increased from SMALL to EXTRALARGE.



(a) 1 ONOS controller

(b) 3 ONOS controller

(c) 5 ONOS controllers

(d) 7 ONOS controllers

Figure 4.9: Service throughput while scaling (up/out) ONOS with a small data plane topology ($TSIZE = 10$ switches).

Figure 4.9a shows the throughput changes as the input rate increases. It can be noticed that for the same deployment size, IST changes significantly when the IRR exceeds the system capacity point of $IST = 0.9$, which means that ONOS can no longer guarantee its stable service beyond that point.

For instance, the performance drops significantly in the case of LARGE and EXTRALARGE deployments when the input request rate exceeds 3,000 requests/s. It also shows that scaling ONOS up by doubling the resources (CPU and Memory) almost doubles the doubles the system capacity. For instance, when ONOS scales from SMALL to MEDIUM, the system capacity increases from 500 requests/s to 1,500 requests/s, and scaling from MEDIUM to LARGE bring the system capacity to 3,000 requests/s.

Similar observations also apply for most of the cases with multiple ONOS instances. The 3 controllers deployment seems the only one exhibiting a different behaviour. Indeed, scaling out from 1 to 3 controller instances does not improve the performance of ONOS, even for the LARGE and EXTRALARGE deployments. This was caused by the fact that sometimes the ONOS controllers lost the connectivity with the data plane devices, leading the system to an inconsistent state where the instances kept trying to backup the OpenFlow devices, i.e., restoring the connectivity and flow rules, causing the depletion of the system resources [83] [84].

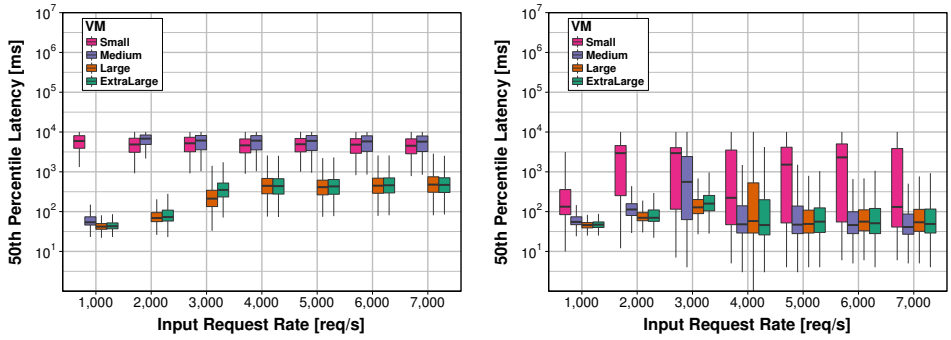
From all results in Figure 4.9 we can observe that:

- The highest input rate of 7,000 requests/s is never sustained;
- For SMALL and MEDIUM VM sizes the input load is not sustained, and scaling out does not necessarily improve the throughput.

- With the LARGE and EXTRALARGE VM sizes the load is sustained up to 3,000 intents/s with one controller;
- The input rate of 5,000 requests/s is sustained only with the EXTRALARGE size, and this demands for at least 5 controllers;
- With the LARGE and EXTRALARGE VM size, scaling out ONOS does not always pay; indeed, scaling out further to 7 controllers does not improve the performance if compared the 5 controllers deployment;

Figure 4.10 shows the average service latency, that is the time required by the ONOS cluster to process an intent request. It can be noticed that the ISL increases approximately linearly with the input request rate and the number of controllers, then it stabilizes when the IIR exceeds the maximum system capacity. For example, Figure 4.10a shows that for both LARGE and EXTRALARGE deployments the latency increases from 90ms up to 300ms when load level increases from 1,000 req/s up to 3,000 req/s. With the request rates that exceed the system capacity (see Table 4.5), i.e., from 4,000 up to 7,000 req/s, the latency is relatively stable at 400ms.

The latency rises from 0,48ms with the LARGE and EXTRALARGE deployments and low request rates, up to 1,000ms with SMALL and MEDIUM deployments. Figure 4.11 shows the tail latencies obtained submitting a



(a) 1 ONOS controller

(b) 3 ONOS controller

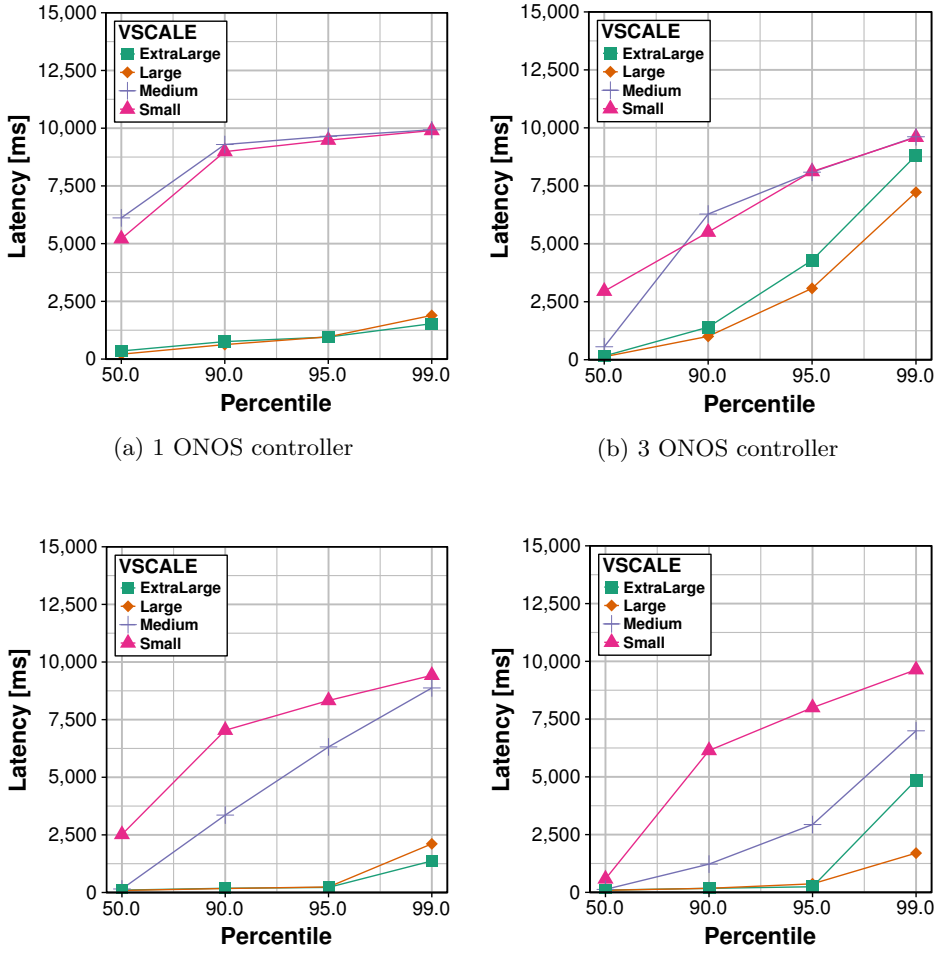
(c) 5 ONOS controllers

(d) 7 ONOS controllers

Figure 4.10: Service latency while scaling (up/out) ONOS with a small data plane topology ($TSIZE = 10$ switches).

request rate of 3,000 requests/s. They exhibit roughly similar characteristics until the 99th percentile for both SMALL and MEDIUM in a single controller scenario, and for LARGE and EXTRALARGE deployments, independently from the deploy size.

These results suggest that scaling out the ONOS instances does not increase the system capacity mainly due to the data synchronisation among the controllers required to maintain a consistent, and always updated, view



(a) 1 ONOS controller

(b) 3 ONOS controller

(c) 5 ONOS controllers

(d) 7 ONOS controllers

Figure 4.11: Tail service latency while scaling (up/out) ONOS with a small data plane topology ($TSIZE = 10$ switches) and 3,000 req/s.

of the network. Hence, This suggest the need to dynamically establish the scale out point, i.e. to find the tradeoff between the benefit of using more controllers and the cost of the ONOS eventual consistency mechanism - even for the highest VM flavours at input rates higher than 3,000 requests/s.

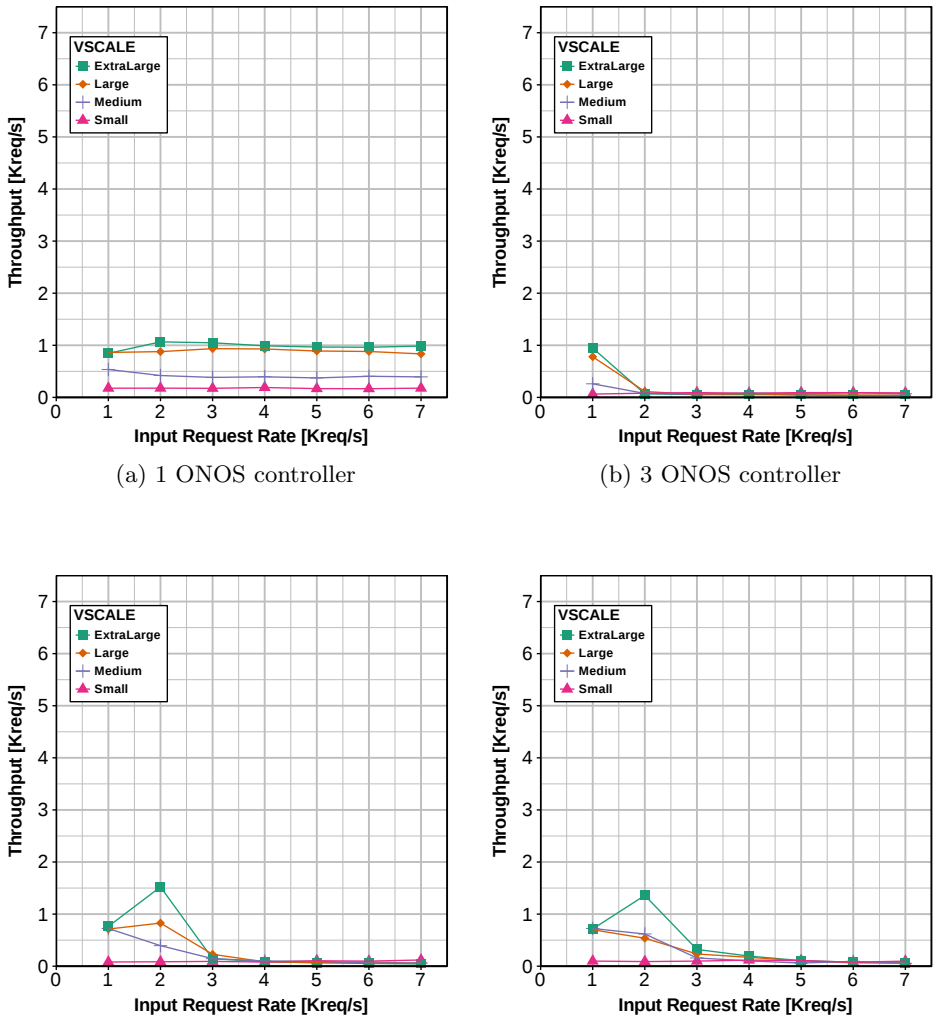
The experimental results in Figure 4.12 show a dramatic loss of performance when the size of the emulated topology is increased to 30 switches (compared to 10 switches of Figure 4.9). In this case, scaling out the SDN adding more controller instances does not provide the expected benefits.

This implies that the maintenance of a large number of connections is expensive. Indeed, during this set of experiments the controller instances have experienced a high memory utilization which caused the stuck of some of the instances, implying that ONOS is not capable to handle a large number of overloaded connections.

Again, the main cause of the performance degradation was related to the overhead of both the East-West, and control-data plane communications. This highlighting the need to adopt more efficient mechanisms, such as the *hierarchical SDNs*, to improve scalability and increase service flexibility of the nowadays SDNs technology.

4.6.2 Experiments with Real Data Plane

A further set of experiments has been conducted with a real topology consisting of 16x V350 Centec Openflow switches connected linearly. The number of ONOS controllers varies from 1, to 3, 5 and 7 instances (HSCALE), while the ONOS VMs is the *large* flavour ($VSCALE = Large$).



(a) 1 ONOS controller

(b) 3 ONOS controller

(c) 5 ONOS controllers

(d) 7 ONOS controllers

Figure 4.12: Service throughput while scaling (up/out) ONOS with a large data plane topology ($TSIZE = 30$ switches).

The first set of tests, conducted with a moderate load of 2,000 request-s/s, showed poor performance in terms of successful Intent operations per unit time, especially when increasing the number of controllers. Hence, to better investigate the possible causes of the performance degradation, we drastically reduced the load, by varying the request rate with a granularity of 200 requests/sec starting from 100 up to 1,400 requests/s.

On the one hand, Figure 4.13 shows that the performance is relatively stable up to 400 requests/s regardless the number of controllers. When the load reaches 600 requests/s, a performance degradation appears on all tested deployment, and adding more controllers entails a further performance loss. Figure 4.14, on the other hand, shows that the average time to process

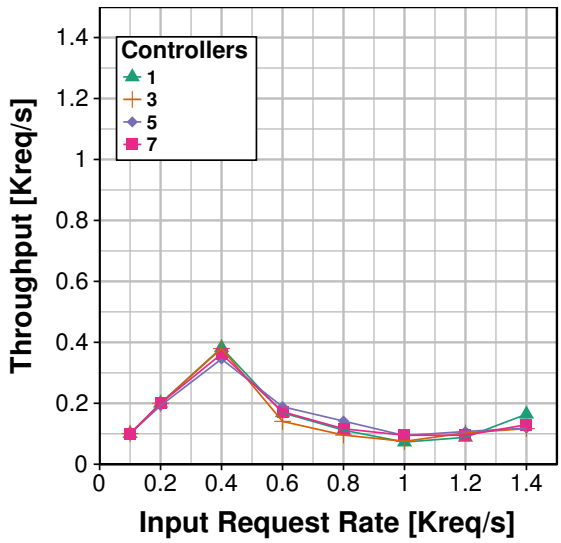


Figure 4.13: Service throughput while scaling out ONOS with a real data plane topology.

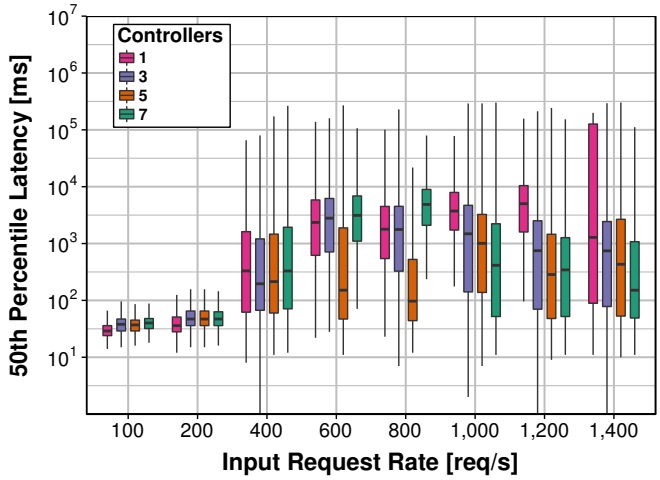


Figure 4.14: Service latency while scaling out ONOS with a real data plane topology.

an intent request is almost doubled when switching from 200 requests/s to 400 requests/s, requiring the system more than a 1,000ms to process each request. However, the cause of these poor performance results is to be found elsewhere, namely in the Centec switches.

We observed that the higher the load, the more the flow rule operations were discarded by the Centec switches due to a hardware limitation of the latter. More in details, the driver queue (a.k.a. ring buffer) of the Ethernet interface (used for management) in the Centec switch was filled out by the high volume of OpenFlow rules submitted by the ONOS instances, resulting in a high throughput degradation. Thus, adding more controllers caused the queue to be filled up faster.

These results point out that the Centec switches perform better under the management of a single controller instance, sustaining up to 400 flow

rule installation per second. At the same time, it has been observed how the OpenFlow SDN switch still needs to be improved in terms of management and programmability, in order to meet the strict performance requirements of the future carrier grade networks (e.g., 50ms constraint in restoring a network path).

4.7 Summary

This Chapter has presented SCP-CLUB, a framework for performance assessment of SDN deployments on a telco cloud infrastructure. It features several tools to automate the deployment of controllers and to orchestrate a campaign of experiments under various operating conditions (number of controller instances, VM size, workload, topology). The user defines the experimental campaign by providing a textual specification with the desired values for configuration parameters. This allows to analyze the telco cloud SDN setup capacity and/or to extract actionable intelligence that can be used to fine-tune or to dynamically adapt (e.g., scale up or down) the cloud resources running the control network.

A campaign of experiments, designed with the DOE methodology, with the ONOS-based implementation of SCP- CLUB has been performed with a setup in a real telco datacenter. The campaign was meant to analyze the performance (and in particular, scalability) under various configurations in

terms of requests rate, VM flavour, and number of ONOS instances.

The results show that: *i)* load levels as high as 3,000 intents per second and higher are sustained only using large VM flavours, regardless of the number of controller instances; *ii)* as the load increases up to 5,000 requests/s, scaling out controllers horizontally (i.e., deploying multiple controllers) provides benefits only for the highest VM flavours; with small- and medium-sized VMs, the performance may worsen at high loads with multiple controllers; *iii)* input rates higher than 5,000 requests/s are not sustained; *iv)* when managing a relatively high number of switches (even in experiments with emulated topologies), the control plane exhibit a dramatic loss of performance; *v)* with real topologies, the data plane suffers from serious scalability problems. These are examples of the indications which can be drawn running experiments with SCP-CLUB to find the desired tradeoff between scaling up and/or out distributed SDN deployments.

Finally, we explicitly point out that all performance figures appear orders of magnitude lower than the those reported in often-cited studies like [11] and [85], which show that controllers can be optimized to handle tens of thousands flow events per second. This is however not surprising, as these are typically measured just as flow initiation events at the southbound interface of a single (possibly multithreaded) controller. Essentially, they measure the maximum flow setup rate that a controller can maintain, while

SCP-CLUB was motivated by the need to assess actual SDN performance under synthetic yet realistic loads, demanding for full intent processing in a really distributed SDN controller deployment on a telco cloud.

This page intentionally left blank.

No amount of experiments can prove that I am right; a single experiment can prove me wrong.

Albert Einstein

Chapter 5

SDN Resilience Assessment: a Failure Injection Tool Suite

The Chapter describes the failure injection methodology and a framework for continuous assessment of the reliability and resilience of SDN technologies. To this aim, the already presented SCP-CLUB framework is extended with a configurable and distributed software infrastructure for failure injection. The Chapter then describes the steps of the methodology, which encompasses the definition of a workload to bring the SDN platform under assessment in a state where to inject failures according to the failure model; the workload is based on the Intent-Based Networking model. A failure model is presented, describing the variety of injectable failure types at system, network and controller level. Then the Chapter describes the logical architecture and components of the distributed software, along with its implementation details. Finally, the Chapter terminates with the experimental campaign, and results, aimed to evaluate the resiliency of the Open Network Operating System (ONOS).

5.1 Assessment Methodology

5.1.1 Overview

The knowledge on how failures may affect software systems is of paramount importance to improve their resilience and reliability. With the complexity of modern distributed systems, the design of effective detection and

mitigation mechanisms can no longer rely exclusively on software testing techniques in controlled environments. Indeed, it is impractical to fully reproduce a complex operational context. In SDNs, for instance, it has been shown that a faulty SDN application can compromise or crash the whole SDN network [86]: while SDN controllers software is likely to stabilize, even the application plane may be a vehicle for dependability threats. Therefore, in the engineering of software network services, it is a key goal to be able to test the proper functioning not only in controlled environments, but also in-production, under real operating conditions.

Figure 5.1 depicts a high-level view of the approach, where failure injection is exploited to **continuously** assess the reliability and resilience of the network services against a wide range of failure scenarios. This will provide continuous feedback on the capabilities of the softwarized network services to survive failures, which is of fundamental importance for improving the system internal mechanisms to react to anomalous situations potentially occurring in operation, while its services are regularly updated or improved. To this end, failures are injected in different layers of the telco cloud infrastructure (Figure 5.1), namely: *(i)* at **data-plane** level, to emulate faulty network appliances, e.g. by injecting Bit Error Rate (BER) or packet latency and corruption at switchs' port level; *(ii)* at **infrastructure** level, to emulate faulty physical nodes or virtualized hosts; and *(iii)* at **control**

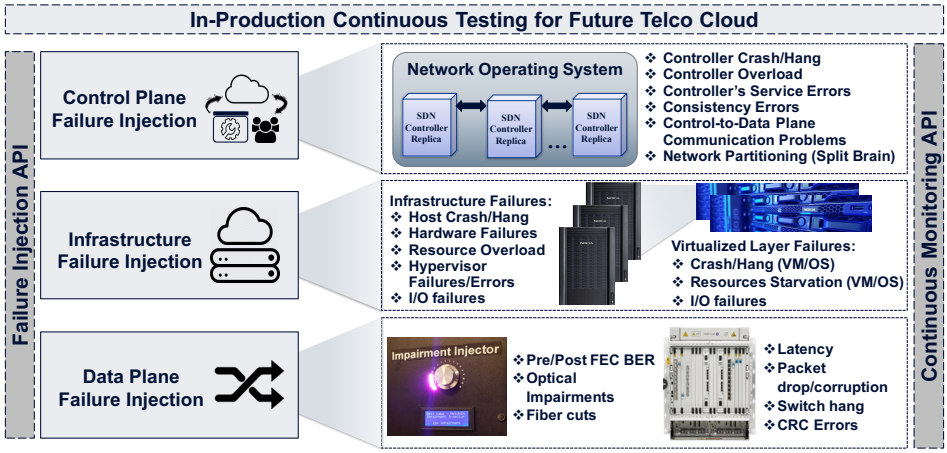


Figure 5.1: In-production continuous testing in Telco Cloud.

plane level, to emulate faulty network controllers.

With this approach in mind, the chapter's aim is pursued through the use of software failure injection to deliberately introduce failures in the components of the system under assessment, or in their execution environment, under real or emulated load scenarios, to evaluate the system behaviour under (possibly unforeseen) disruptive conditions. Specifically, it focuses on the resilience of the control plane layer, and proposes a methodology and a tool suite to validate the reliability and resilience of distributed SDN platforms.

5.1.2 Failure Injection Methodology

The proposed methodology aims to assess the effectiveness of the failure detection and mitigation mechanisms provided by the SDN technology by

reproducing representative failure scenarios [1]. To this end, the SCP-CLUB infrastructure presented in §4.1 has been improved with a *failure injection* tool suite to deliberately inject failures in the SDN ecosystem limiting intrusiveness, as much as possible.

According to the proposed methodology, the system is exercised with a *workload* and a *faultload*. The *workload* reflects a load profile that a distributed SDN system will face in production environment; the *faultload* consists of a set of failure, i.e., system or network misbehaving, which are injected in the system. By executing the system with the workload and subjecting it to the faultload, we aim inducing errors and failures into the SDN ecosystem.

The steps of the methodology are outlined in Figure 5.2. The execution of the experiments is automated and supervised by the *Experiment Manager* program, or *EM* (see §4.4.4), while the *System Under Test* (SUT) is the target SDN infrastructure, i.e., the set of the SDN controller instances distributed across several machines. The *EM* specifies the experimental parameters, sets the SUT, i.e., deploy, starts/stops all the SDN controller instances and the framework's components, and restore the machines to ensure the same initial conditions for each experiment.

[1] It is worth mentioning that, although the failure model described in the next sections is meant to target specifically the control plane ecosystem, some of the proposed failure modes may overlap with those required to emulate faulty condition at infrastructure level (see Figure 5.1), e.g. a controller crash may correspond with the crash of the virtual machine hosting such controller.

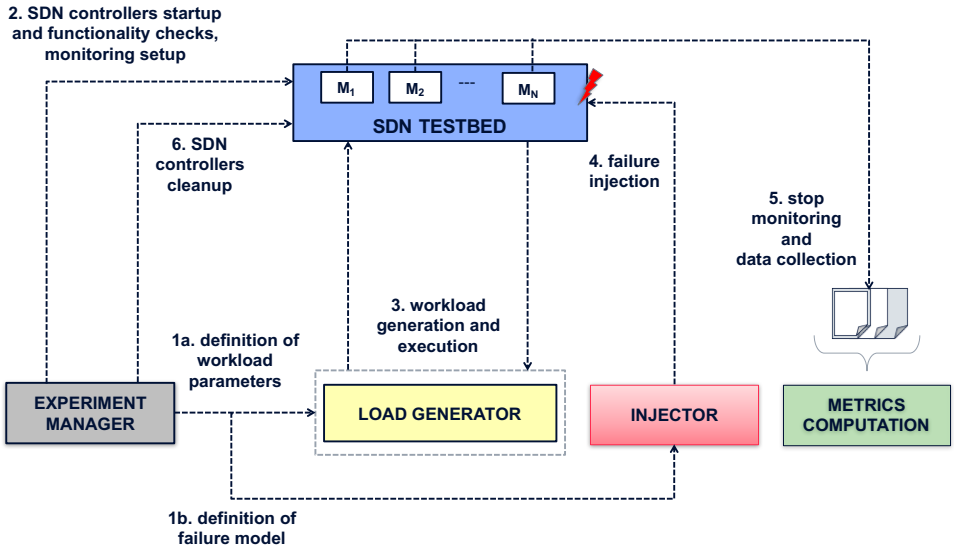


Figure 5.2: Steps of a failure injection experiment.

More in details, the execution of the failure injection experiments encompasses a number of tasks (Fig. 5.2). After the definition of the failure model and the workload parameters (step 1 in Fig. 5.2), the experiment is set up, configuring and deploying the SDN infrastructure under evaluation, along with the data monitors (step 2). Then the workload is generated (step 3), so as to stimulate the SDN to bring it in a state where to inject a failure selected from the failure model. During execution, a failure is injected (step 4), while the system is monitored and data are collected (step 5). After execution, the testbed is cleaned up by restoring the original status of the machines running the SDN controllers, and restoring the controller instances, before starting the next experiment (step 6). This process

is repeated for each execution of a failure injection experiment, and several failures are injected at each experiment of the campaign (while keeping the same workload and collecting the same performance measurements).

It is worth emphasizing that although the analysis reported in this dissertation have been conducted in a controlled test environment, the proposal has been conceived as a framework (methodology and injection infrastructure) to assess the resilience of the SDNs in production. Indeed, the complexity of the environments where SDNs operate can lead to situations that are difficult or even impossible to replicate with the traditional software testing approaches [21].

The support infrastructure has been designed to be easily deployed into the target SDN ecosystem, so as to support the injections of failures even in a production environment, with the aim of continuous testing.

The execution of failure injection experiments is supported by the infrastructures described in the next sections.

5.1.3 Failure Model

Software-Defined Networks are typically engineered as distributed systems given the drawbacks of centralized solutions in terms of scalability, performance, and fault tolerance. This is true also for the newest SDN implementations, such as ONOS® (Open Network Operating System) and ODL®

(OpenDaylight). For such a reason, failures have been chosen among those belonging to the most common failure classes observed in distributed systems [21] [87] [88] [89]. These failures are injected by merely using API calls in a non-intrusive fashion.

Table 5.1 lists the failure[2] classes considered in the proposed assessment methodology. Each class is intended to mimic different types of failure scenarios at different levels of the software stack. According to the level to which they apply, failures are classified in three main categories:

- ***Infrastructure Failures Model***. These failures are further catechorized into:
  - ***System Failures***. The computational power and reliability of hardware equipments, adopted for both dedicated and COTS (Commercial Off-The-Shelf) server, has increased dramatically in the past several decades. Despite these increases, failure still occur in complex and high powerful infrastructures; the adoption of virtualization technologies and cloud computing solutions has introduced further challenges in terms of security and reliability. Therefore, the failure model encompasses failures affecting the

[2]It has to be noted that some of the failures in the proposed model - e.g. memory corruption and saturation - should actually be considered *errors*. However for the sake of simplicity, all the injected failure and error classes have been considered as part of the *failure model* presented here.

computational resources as well as I/O operations (e.g., physical/virtual CPU, memory and disk) of a target node. The goals is to evaluate the resilience of the SDNs to nodes crashes and resources depletion affecting the machines hosting the controller instances. The failure types envisaged in this respect are system *hang*, *starvation*, *outage* and *shutdown* (at single CPU level), as well as disk and memory *saturation*.

Furthermore, a single controller instance might suffer from increased CPU utilization, for instance due to other compute-intensive jobs running on the same target machine. The corresponding failure types to mimic such a scenario are CPU or I/O *burn*. This class of failures are emulated by starting additional jobs that deliberately consume CPU cycles and allocate memory areas aiming to cause resource exhaustion, i.e., CPU and memory "hogs".

– ***Network Failures***. Network problems, such as link failures or latent communication, are among the ones that have always been faced by distributed applications [87] [90] [91]. The most common consequence of these kind of failures is the partitioning of the network that split a system in multiple disjoint, disconnected partitions. As a result, even if a system is designed to be partition-tolerant, there are no guarantee that the modern SDN

distributed systems are able to cope with partitioned, unreliable networks [88]. To reproduce such network failures, message *corruption* or *loss*, partial or total *network partitions* as well as the permanent *unavailability* are introduced into the network interfaces. In addition, even *latent communications* and *bandwidth limitation* are emulated.

- ***SDN Controller Failures Model***: This class of failures aims of mimicking the malfunctioning that may occur in the interaction between the SDN controller services, or malfunctioning of the controller itself. API calls may also be used to emulate a faulty controller instance by shut it down or to mimic an anomalous service behaviour by forcing the termination of specified system process. The corresponding failure types are emulated by process *kill*, and controller or dependency *shutdown*. Furthermore, a *memory corruption* failure is also provided to corrupt the state of the controller, mimicking a hardware fault, or a programming error affecting the controller's memory.

It has to be pointed out that future SDN controllers are likely to be engineered to be deployable also in virtualized platforms, or in container technologies. This means that system failures should in principle to be injectable also at the virtual machine (VM) or container level. However, as this is true for any software system deployed in VM or containers (for

Table 5.1: Failure model.

Failure class	Failure type	Description
System Failures	System Hang	Stuck the system on an infinite loop without releasing the resources. The system's network interfaces are still responding.
	System Starvation	Cause the starvation of all the available system resources. The system first slow down, then crashes.
	System Outage	Cause a fatal error from which the system cannot safely recover (e.g. kernel panic).
	CPU Shutdown	Restrict the number of available CPUs.
	Disk Saturation	Cause the saturation of the disk, mimicking disk full errors.
	Memory Saturation	Cause the saturation of the memory, mimicking memory full errors.
	Burn CPU	Spawn CPU-bound processes, mimicking a faulty CPU and/or noisy process.
	Burn I/O	Spawn I/O-bound processes, mimicking a faulty disk and/or noisy process.
Network Failures	Black-hole	Abruptly drop the network communications towards a specific address or a subset of addresses.
	Packet Reject	Abruptly drop the inbound and/or outbound packets sent to specified address and/or port.
	Packet Drop	Quietly drop the inbound and/or outbound packets sent to specified address and/or port.
	Packet Latency	Induce artificial delays for packets sent to specified address and/or port.
	Packet Loss	Induce artificial losses for the packets sent to specified address and/or port.
	Packet Re-order	Induce a mis-ordering of certain packets sent to specified addresses and/or port.
	Packet Duplication	Induce duplication of certain packets sent to specified addresses and/or port.
	Packet Corruption	Induce a random noise by introducing an error in a random position of certain packets sent to specified addresses and/or port.
	Throttling	Induce a bandwidth limitation to the outgoing network traffic with specified addresses and/or port.
SDN Controller Failures	Kill Process	Quietly terminate the controller process (*SIGTERM* signal), mimicking a faulty service.
	Process Corruption	Randomly corrupt the state of the SDN controller process, mimicking a service misbehaviour.
	Controller Shutdown	Quietly stop of the SDN controller process .
	Controller Restart	Gracefully restart the SDN controller.
	Dependency Shutdown	Quietly stop one or more dependencies, i.e., modules of the SDN controller.

instance, fault injection has been proposed for virtualized network functions [92]), this dissertation intentionally did not include in the proposed failure model failures at the Hypervisor, host or container level, limiting it to failures whose injection allows to test controllers wherever they are in execution.

Each failure in the model is triggered according to a specified *injection time*, that is the exact time when the failure must be injected. In addition, in order to allow the user to design more complex failure-injection scenarios, the failure can be set in three different ways, namely:

(i) ***transient***: the failures are injected only once and removed after a specified amount of time in order to emulate temporary failure;

(ii) ***intermittent***: the failures are periodically injected and left in the system for a specified amount of time to emulate temporary, but recurrent failure conditions;

(iii) ***permanent***: the failures are injected and never removed from the system to emulate persistent failure conditions.

5.1.4 Measurements

To evaluate the performance of an SDN platforms under faulty scenarios, and their capability in detecting and mitigating such disruptive conditions,

two set of metrics have been identified, namely:

- **Service-level Measures**. These are high-level performance measurements representative of the quality of the service provided by the system, which are primary indicators of the system's health. They encompasses the *i)* ***IBN Service throughput*** and *ii)* ***latency*** described in §4.4.8.

- **System-level Resilience Measures**. The goal of a system's resilience mechanisms is to reduce, as much as possible, the disruptions to business operations. The ultimate goal is to avoid any possible downtime. Unfortunately, the provided resilience mechanisms may fail in accomplishing their purpose, proving to be inefficient under certain faulty conditions. Therefore, these set of measurements cope with the characterization of the SDNs in terms of their ability to detect and correctly handle unforeseen faulty conditions. They encompasses the following metrics:

  - ***Failure Detection Coverage and Latency***. The *Failure Detection Coverage* (FDC) is defined as the percentage of failure injection tests in witch the SDN infrastructure raises a notification about the faulty condition, either on a single node, or on all the failure-free nodes belonging to the cluster. It is computed as

follows:

$$FDC = \frac{\#F_{detected}}{\#F_{undetected} + \#F_{detected}} \tag{5.1}$$

where $\#F_{detected}$ and $\#F_{undetected}$ are the number of tests in which the injected failure is detected and reported by the SDN cluster, and the number of tests in which the injected failures is not detected but causes performance degradation, respectively. The *Failure Detection Latency* (FDL) is computed as follows:

$$FDL = t^{e}_{detection} - t^{e}_{injection} \qquad e \in E \tag{5.2}$$

where $t^{e}_{injection}$ refers to the time the failure is actually injected on the target controller, and $t^{e}_{detection}$ refers to the time at which an anomaly is raised by the SDN cluster. In computing this metric we only consider the subset e of all the performed experiments E in which the failures have been correctly detected and reported by the SDN infrastructure.

– ***Failure Recovery Coverage and Latency***. The *Failure Recovery Coverage* (FRC) is defined as the percentage of failure injection tests in witch the SDNs initiated and successfully completed a recovery process. It is computed as follows:

$$FRC = \frac{\#F_{detected}}{\#F_{recovered}} \tag{5.3}$$

where $\#F_{detected}$ and $\#F_{recovered}$ are the number of tests in which the injected failure is detected, and the number of tests in which a corresponding recovery action is *successfully* completed, respectively. For example, upon the crash of a controller instance, the recovery action is considered successful if the remaining controllers correctly redistribute the management of the switches controlled by the failed instance, keeping a consistent view of the data plane network, and without losing performance.

The *Failure Recovery Latency* (FRL) is given by:

$$FRL = t^e_{recovery} - t^e_{detection} \quad where \quad e \in E \tag{5.4}$$

which is the time between the detection of the failure, i.e., $t^e_{detection}$, and the termination of the recovery procedure, i.e., $t^e_{recovery}$. It is computes exclusively for those experiments in which a recovery action has been taken by the failure-free SDN nodes.

It should be noted that only the Service-level Measures have been used for the experimental evaluation described in the next sections; the System-level Resilience Measures are defined in this dissertation, but they will be used for future works.

5.2 Failure Injection Framework

This section introduces the failure injection tool suite designed to complement the proposed assessment methodology, by supporting the execution of failure injection experiments. The tool extends the SCP-CLUB architecture obsessed in the previous chapter with the ***Failure Injector*** (FI), which is a ready-to-use component which can be integrated seamlessly into the target system. Figure 5.3 shows how the *FI* components are integrated with the existing components of the SCP-CLUB framework, and the SUT.

The *FI* is based on the failure model described in §5.1.3. It supports and simplifies the simulation of several failure scenarios, such as misbehaving or crash of one or more SDN controllers. Using those scenarios, a network provider could pick the proper template *failure scenario* to run against their SDN ecosystem. The model's failures are synthetically introduced into SDN's components, allowing them to experience a series of failures that simulate real-world failures. In addition, since error conditions and corner cases occur in a production environment, even if they are not observed during testing activities, the proposed *FI* is conceived to also inject failures into running SDNs, as a mechanism to best mimic real-world dependability challenges.

Figure 5.4 depicts the high-level overview of the FI architecture. It is a distributed application which adopts a *Publish/Subscribe* communication

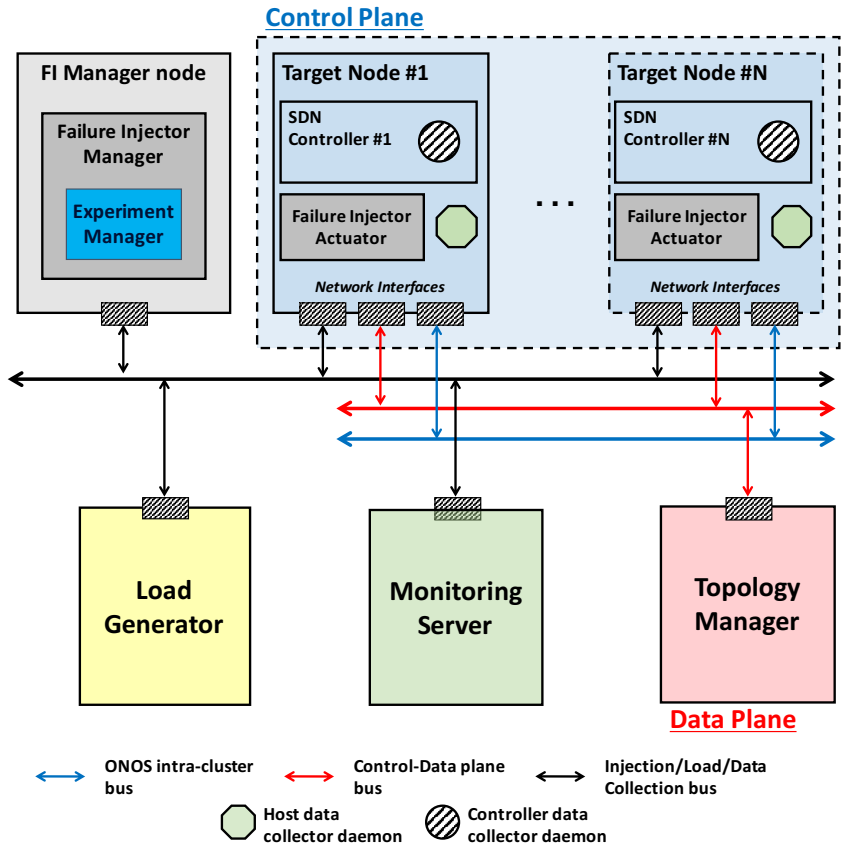


Figure 5.3: Architecture of the SDN failure injection framework.

model (or pub/sub, for short) and consists of two main components:

- ***Failure Injector Actuator***. This program resides on the Target Node, that is the machine running the target SDN controller instance, and actually performs the failure injection;

- ***Failure Injector Manager***. This user space program resides on a separate machine, the *FI Manager* node, and remotely coordinate the injection on all the target nodes.

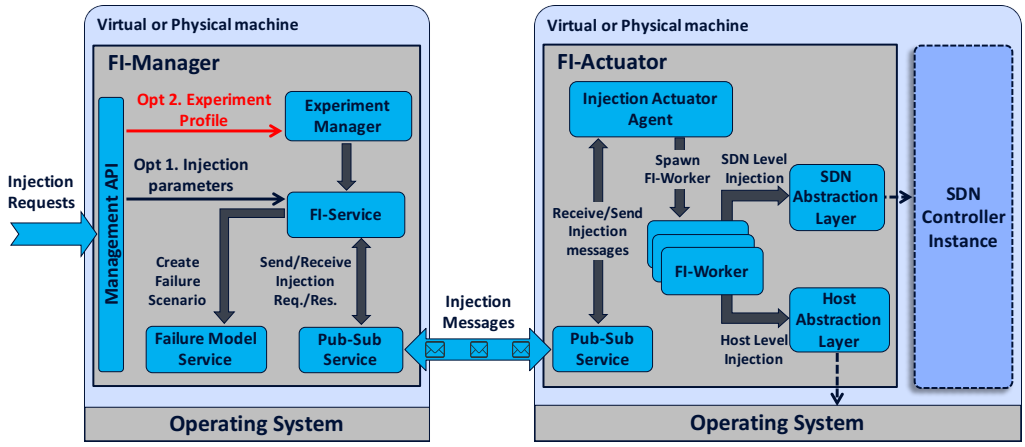


Figure 5.4: The Failure Injector architecture.

The *FI* architecture has been designed with the goal of simplifying and speeding up the extension of the failure scenarios, while the provided user-friendly API makes the proposed tool suite easy-to-use and accessible.

The next sub-sections describe in detail the implementation of the outlined failure injector's components.

5.2.1 Failure Injector Implementation

We have designed a *Java-based – Spring Boot ® 1.4* – implementation of the *FI* framework extending the ONOS-based *SCP-CLUB* implementation proposed in §4.4. Figure 5.5 depicts its implementation design, where the two main components, namely the *FI Manager* and the *FI Actuators*, are deployed on different hosting machines, and communicate through the *Apache ActiveMQ*™ [93] (*ActiveMQ 5.14*) message broker, a Java implementation

of the Publish-Subscriber design patter. The broker resides on the the same machine hosting the *FI Manager*, and provides a reliable *topic-based* messaging infrastructure that the *FI* components can use without being coupled to each other. According to such pattern, the *FI* components act as senders (publishers), and/or receivers (subscribers). The former publish messages (i.e., events) on specific topics, without knowing the subscribers, i.e., without specifying the receivers. Similarly, subscribers receive only messages for the topics to which they subscribe, without knowing the sender.

The Pub/Sub pattern has been chosen as it provides the following key advantages, compared to the traditional client-server communication mechanisms:

- *loose coupling*: publishers and subscribers can work independently, i.e., each one can continue to operate normally regardless to the other;
- *scalability*: by decoupling publishers and subscribers, the pattern supports greater systems scalability as well as a more dynamic network topology.

For the sake of clarity, Figure 5.5 shows a single instance of the *FI Actuator*, deployed on the target node hosting the ONOS controller. However, according to the size of the ONOS deploy, that is the number of ONOS instances, several FI Actuators may exist to be able to perform the injection on several target nodes.

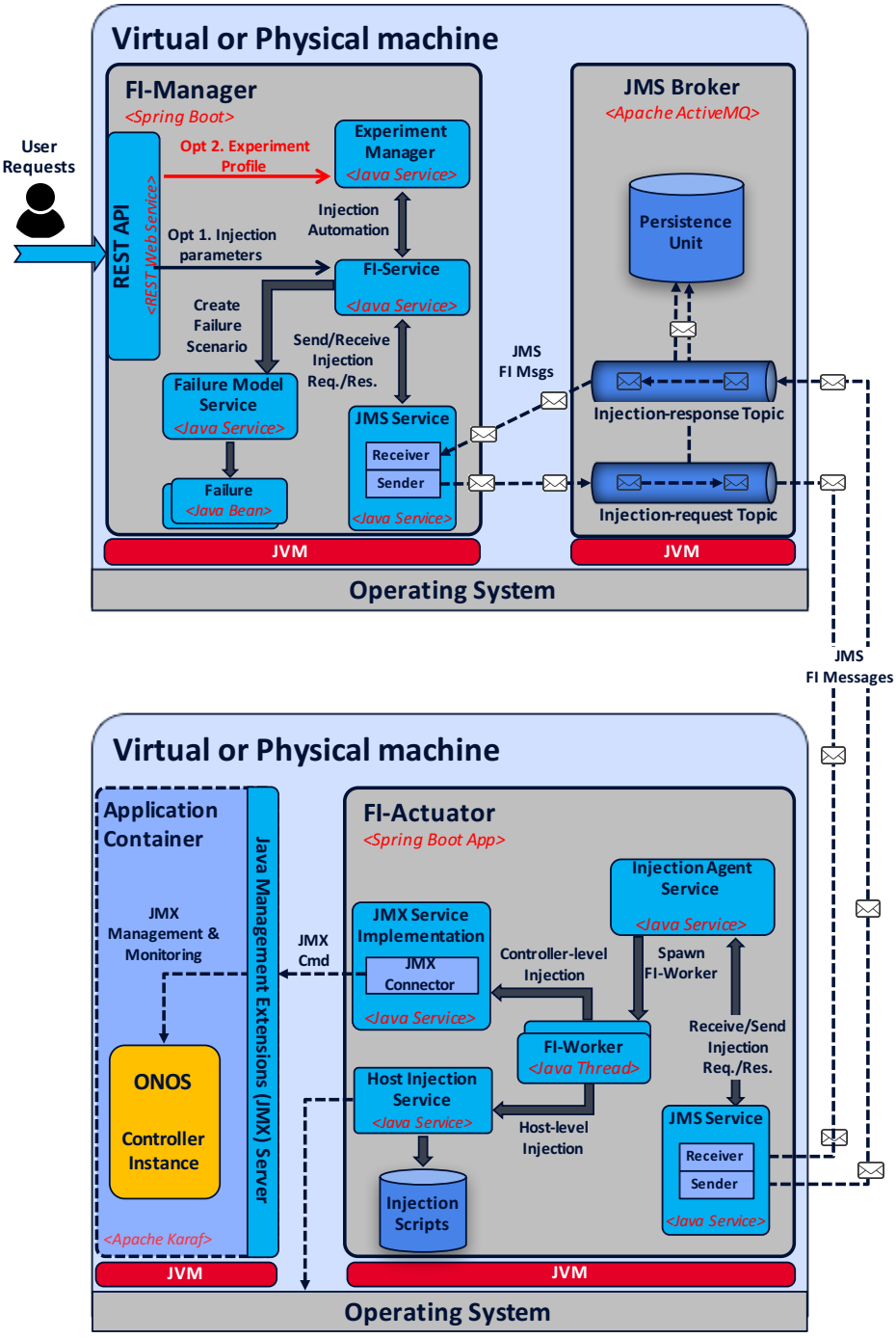


Figure 5.5: Design of the Failure Injector for the ONOS controller.

Failure Injector Manager

The *FI Manager* runs as a system service on the Manager Node. With respect to the adopted Pub-Sub messaging model, the manager acts as: (i) a *publisher*, while sending *failure injection requests* to the actuator(s) as well as (ii) a *durable subscriber*, while receiving the actuator(s) response. Here a failure injection request refers to a message containing all the parameters the *FI-Actuator* requires to actually apply the injection.

The *FI Manager* is quite simple in its infrastructure (see Figure 5.5). It provides an end-user API that adheres to the principles of *REpresentational State Transfer* (REST) paradigm, or *RESTful* web services, simplifying the parametrization of the failure injection experiments. Each experiment is converted in a corresponding failure scenario. To this aim the *Failure Injection Service* interacts with the *Failure Model Service* to retrieve the "*failureload*" scenario(s), i.e., the *Java Bean* object which specifies *what* failure to inject, *when* and *where*. The latter is then translated in a message and sent to the corresponding actuators through the *JMS Service*.

As can be seen in Figure 5.4 and 5.5, the *FI Manager* incorporates the *Experiment Manager* (§4.4.4) of the SCP-CLUB framework to automate the failure injection experiments . To this end, it extends the user-provided specification (§4.3.2) with further parameters to support the failure injection. Figure 5.6 shows an example of such specification file. In addition

```
experiments:
 -
  id: CPUBURN_1
  runs: 5
  cell: "five_controller_cell"
  duration: 5
  networkTopologyInfo:
  ...
  failure:
   hosts:
    - "X.Y.Z.1"
    - "X.Y.Z.2"
   what: burnCpu
   when: now
   where: compute
   mode: transient
   injectionSpanTime: 10
   tasks: 10
```

```
experiments:
 -
  id: NET_LATENCY_1
  runs: 5
  cell: "three_controller_cell"
  duration: 5
  networkTopologyInfo:
  ...
  failure:
   hosts:
    - "X.Y.Z.1"
    - "X.Y.Z.2"
   what: networkLatency
   when: now
   where: compute
   mode: permanent
   destinationIps:
    - "X.Y.Z.3
```

Figure 5.6: A user-provided specification for failure injection experiments.

to the data plane and workload parameters already described in §4.3.2, the specification file defines the target nodes of the failure injection experiment, namely the FI Actuators that must inject the failure (e.g., the node with IP address *X.Y.Z.1* and *X.Y.Z.2*). It also specifies the type of failure to inject on such nodes, e.g. a "*burnCpu*", namely a *System Failure*; and the failure *injection mode*, i.e., a *transient* failure with an *injection time span* of 10*s*. The user can specify a set of failure injection experiments to be performed sequentially, hence making the *EM* a very powerful tool to speed up the experimental analysis.

Failure Injector Actuator

The *FI-Actuator* component is in charge of actually perform the failure injection. It is a lightweight program running as a system service daemon on a target node and, as the *FI-Manager*, acts as both *subscriber* and *publisher* in receiving and sending *failure injection request* and *response*, respectively.

The *Injection Agent Service* (see Figure 5.5) is the main service of the actuator. It coordinates the communication with the message broker and triggers the *FI-Worker* to perform the injection task. According to the injection type and the timing parameters provided along with the injection request, the Agent Service spawns one or more FI-Workers - in the form of Java threads - which can run once, to emulate **transient** and **permanent** failures, or periodically to emulate **intermittent** failures.

Furthermore, according to the failure scenario to assess, failures must be injected in a specific component, e.g. in the SDN controller or in the Operating System (OS). To this end, the FI-Workers use two interfaces (see Figure 5.4), acting as abstraction layers to make the injector cross-platform compatible, namely:

i) ***SDN Abstraction Layer***. This layer provides a common and restricted set of APIs to simulate internal failures of the SDN controller's services or the controller itself. To this end, the *FI Actuator* provides an implementation of the *Java Management Extensions* (JMX), which

is the technology supporting the monitoring and the management of Java application. Through this service, the FI-Workers can intercept and terminate specific ONOS services as well as the ONOS instance itself, by directly interacting with the corresponding JMX server exposed by the *Apache Karaf Container* [94].

ii) ***Host Abstraction Layer***. This layer offers the abstractions required to enable the injection of failure at OS level. To this end, the *FI Actuator* provides the *Host Injection Service* which runs lightweight *bash scripts*, *C programs*, or *kernel modules*, according to the type of failure to inject. The *Host Injection Service* works with several Unix-like operating systems, and it can be simply extended to work with other OSs and to accommodate further failure modes.

5.2.2 Failure Implementation

This section provides a description of the failure models introduced in Section 5.1.3 supported by the proposed Failure Injection tools. More in detail, it discusses the scenario reproduced by the occurrence of each supported failure type, and how this is realized by the *FI*.

As shown in Figure 5.7, failures classes are intended to target different components of a target node. The *System* failures are injected to emulate failures related to the computational resources of the machine hosting an

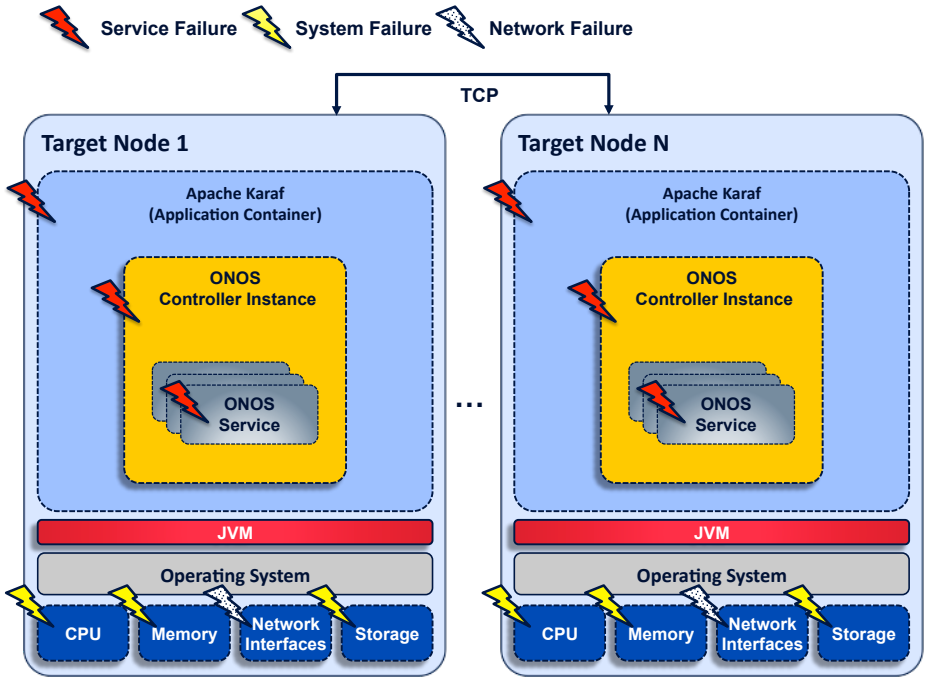


Figure 5.7: Localization of failure injections.

ONOS instance, while the *Network* failures are meant to specifically emulate failures concerning network communications and interfaces. In addition, failures are injected at *Service* level, i.e. at application container or process level, to emulate a faulty controller instance.

We emphasize the fact that each of these failures can be injected, at the same time, into a specific target node or a subset of nodes. Additionally, as already pointed out in this dissertation, these failure models are intended to assess: *(i)* the resilience and performance of the target SDN platform; and *(ii)* if proper failover mechanisms, i.e. fallback logics or degraded operation modes, are provided under faulty conditions.

System Failures

This class of failures reproduces scenarios mimicking resource saturation or crashes of the machines hosting the SDN controller instances. It encompasses several modes such as, system *hang*, *starvation*, *outage* and *shutdown* (at single CPU level), as well as disk and memory *saturation*.

The ***hang*** of the hosting machine is emulated by installing a kernel module in the OS which performs uninterruptible computation, i.e. it takes control of all the available computational resources without ever releasing them. By doing so, the controller process, as well as other system processes, is no longer able to acquire any CPU cycle to perform its operations.

The ***system outage*** may lead to the stop of the machine hosting the ONOS instance. It is implemented as a kernel module which performs inconsistent operations leading to the so called "kernel panic" error.

The ***system starvation*** is a further type of failure which can cause the crash of the hosting machine. It depletes the system resources: if injected and left in the system for enough time, or injected in a permanent manner, it slows down or crashes the system due to resource starvation. This failure scenarios is reproduced by starting a user-space process which continuously replicates itself, exhausting all the system resources.

The ***memory*** and ***disk saturation*** failures are implemented as high priority user-space process which allocate as much memory, or disk, space

as possible, without ever freeing the acquired resources. These classes of failure are meant to assess how the lack of memory, or disk, space, can affect the availability of a controller instance, and if such a condition can lead to an "inconsistent" state of the control plane.

In order to evaluate the resilience of the SDN platform against faulty CPUs or maintenance activities, the Injector also allows the ***shutdown*** of the CPUs. It changes the target CPU state by turning it off. In doing so, the Injector exploits the *CPU hot-plugging* [95] feature supported by the Linux, which is the ability to turn a CPU core on and off dynamically. The Injector changes the state of a CPU by modifying the value in */sys/devices/system/cpu/cpuX/online* (where X is the target CPU to turn on or off), which in turn invokes the corresponding kernel function that updates the CPU state.

Finally, a single target controller instance might suffer from increased CPU utilization, for instance due to resource overcommitment problems in a virtualization environment, or to other compute-intensive jobs running on the same target machine. The corresponding failure types to mimic such a scenario are ***CPU*** and ***I/O burns***. They are emulated by spawning high-priority user-space processes which compute CPU, or I/O, intensive activities causing the performance degradation of other system processes.

Network Failures

In complex distributed environments, such as those of SDNs, network problems can occur at any layers of the infrastructure. For instance, network failures can be due to to physical faults of network devices, or to bad network configurations or to wrong design. This turns out to produce "inconsistencies" among the layers of the SDN stack, e.g., the view of the logical network is misaligned with the physical network, or to compromise the control-to-data plane communication, resulting in a "brainless" network. Therefore, the failures belonging to this class aim to mimic network-related problems typically addressed by distributed systems, such as a corrupted messages, or a latent connection which can lead to *split brain* problems.

In order to reproduce these failure scenarios, the Injector leverages the *Linux Traffic Control* tool, or *tc*, which is a powerful tool for network traffic shaping, scheduling, classification and prioritization. It is part of the Linux framework for controlling and monitoring various aspects of networking, such as routing, bridging and firewalling. As shown in Figure 5.8, *tc* is the last component of the Linux networking environment that packets has to pass through before leaving a specific output interface. The *tc* tool is build atop *qdisc* (for *queue discipline*), which is basically a scheduler for the packets passing through a network interface. The simplest implementation of a qdisc is *first in first out* (*FIFO*), however several schedulers are provided.

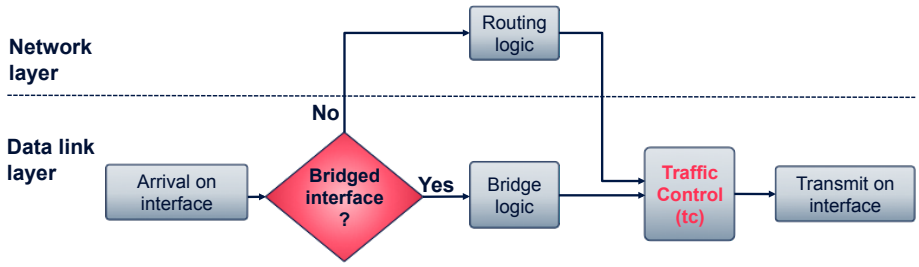


Figure 5.8: The Linux *Traffic Control* tool.

The Injector leverages the built-in *Token Bucket Filter* [96] (*TBF*) queue discipline to slow down, i.e. ***throttle***, the outgoing network traffic flowing from a target controller replica to one or more replicas, aiming to emulate failures due to *network congestion* problems. Vice versa, it uses the *Network Emulation* [97] (*NetEm*) utility, in order to emulate failures scenarios due to packets latency, loss, corruption and so on. By mans of this kernel component, the Injector defines specific queue disciplines to ***fails*** or ***corrupt*** the requests between two or more controllers, as well as to inject ***latency*** or to induce ***miss-ordering*** into such requests.

To mimic failures due to "firewalling" or faulty network interfaces, the Injector adds specific filtering rules into *Netfliter* [98], the Linux kernel's framework for packet filtering and manipulation. Such rules are intended to filter the incoming and/or outgoing packets directed to specified IP addresses and/or ports, and discard them, with a ***drop*** or ***reject*** action.

Finally, the ***black-hole*** failure injection is performed by leveraging the *ip* [99] Linux utility to add entries into the TCP/IP routing table of the

Linux kernel (aka "Forwarding Information Base") aiming to mimic a possible ***split brain*** problem, by dropping all the network packets designated to a specified IP address and/or ports.

SDN Controller Failures

As discussed in §4.4, the ONOS controller - similarly to other SDN controllers (e.g., ODL) - is engineered as set of software modules, or bundles, interacting with each other to provide several type of services at different levels of the software stack, as specified by the *OSGi* framework. It is built atop the *Apache Karaf* feature service, an OSGi container which simplifies the management of the OSGi ONOS services.

In such a complex ecosystem, failures can occur, leading to impairment of the interaction between the ONOS services, or worse, to the unavailability of the ONOS instance itself. Thus, this class of failures aims of reproducing scenarios affecting the ONOS services, as well as scenarios that emulate a faulty controller instance.

The injector uses *Java Management Extensions* [94] (*JMX*) features provided by the Karaf container to emulate problems affecting the ONOS services. Indeed, as specified in §5.2.1, the Injector provides an implementation of a JMX Client to interact with the corresponding server exposed by

Karaf and dynamically manage the Karaf's resources at runtime. Specifically, the Apache Karaf features provide a set of *Managed Beans* (*MBeans*), i.e., an enhanced Java bean representing a manageable resource, which are accessible through the *MBean Server*. Then, the Injector uses the *JMX Connector* to connect to MBean server and to manage the registered resources, such as the installed OSGi services, or the Karaf instance itself.

Figure 5.9 depicts the procedure adopted by the Injector to *stop* the ONOS controller, or one or more of its modules. The steps are the following:

1. The Injector, acting as *JMX Client*, issues a Java Remote Method Invocation (RMI) to the Karaf RMI Registry to obtain the *stub* for the *JMX Connector Server*[3];

2. The client uses the stub to connect to the Karaf *MBean Server* and interacts with the MBeans;

3. The client leverages the services of the Karaf's *MBean* to inject a service failure, e.g. by ***shutting down***, ***restarting*** the ONOS instance or its services, or the Karaf container itself. According to the provided injection parameters, the Injector calls the appropriate methods exposed by the Karaf's MBbean to actually inject the failure.

The ***kill*** of the ONOS instance is performed by sending to its process the

[3]The stub is a local Java class implementing the remote interface of the JMX Connector.

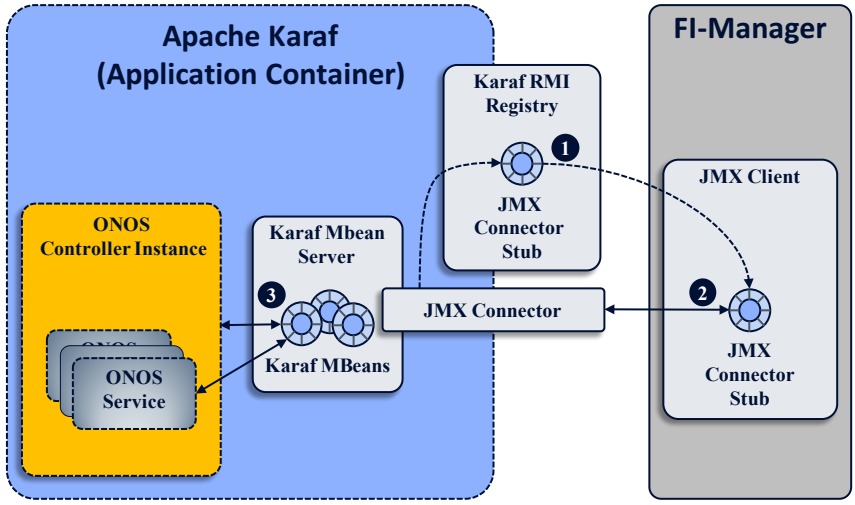


Figure 5.9: Example of JMX-based procedure to inject a service failure.

SIGKILL signal. Unlike the clean stop of the instance, the *SIGKILL* signal is not captured by the process and stop it immediately, thus not allowing the ONOS instance to cleanly close the socket connections.

The Failure Injector framework provides a further injection to ***corrupt*** the state of an ONOS instance. To this end, the Injector corrupts partially, or totally, the Java *heap* or *stack* space of the controller process memory. In the first case, the corruption affects the controller's runtime environment, while in the second case it affects the execution environment of the controller threads. Moreover, both conditions can lead to corruption, or worse, to Java Virtual Machine (JVM) crash.

Therefore, the *corrupt* failure injection is meant to observe how an instance with a corrupted runtime environment can affect the data integrity

of the overall SDN platform, and if the failure-free instances are able to deal with a misbehaving instance. Indeed, even if the instance lies in a corrupted state, for a short amount of time it is still able to interact with its peers and the data plane.

In order to perform a memory corruption, the Injector runs a user-space process which observes and controls the ONOS process, by using of the Linux *Process Trace* system call, or *ptrace* [100, 101, 102]. *ptrace* provides a mechanism to examine and change the core image and registers of a monitored process, and is used primarily to implement breakpoint debugging and system call tracing. By means of *ptrace* the Injector, i.e., the "*tracer*", first attaches to the ONOS process, i.e., the "*tracee*", then it identifies and accesses the memory locations of such process and corrupts the content of the heap, or stack, memory.

5.3 Experimental Evaluation

High availability and reliability are key goals for SDN technologies, that need to meet the six-nines reliability requirements of the carrier-grade networks before being widely adopted in today's network. Therefore, this experimental campaign aimed to verify how the SDN technologies, such as the ONOS platform, perform under disruptive conditions affecting the control plane. Specifically, the goal was to *(a)* characterize the resiliency and reliability of

the ONOS platform by reproducing faulty scenarios, and *(b)* investigate the effectiveness of the detection and mitigation mechanisms of ONOS.

5.3.1 Experimental Campaign

To show the application of the proposed methodology an experimental campaign has been conducting running ONOS (§4.4.1) as SUT. The latter has been deployed in a subset of the the proof-of-concept telco cloud infrastructure described in §4.5.2.

The experimental testbed consists of: (i) 3x 2-Socket server equipped with 2 Intel® Xeon® E5-2680 v3 12-core 2.5GHz CPU, 256 GB of DDR4 hosting both the VMs running the *SCP-CLUB* components as well as the ONOS instances; (ii) a blade server having the same hardware configuration of the previous hosts mounting a shared iSCSI storage of 14x 6TB 7.2k 3.5" SAS disk, and hosting the *SCP-CLUB* data collector machine; (iii) a 1/10Gb Nokia Ethernet *management* switch interconnecting all the servers. The VMware ESXi hypervisor have been used for the virtualization layer.

In order to test how ONOS perform under faulty conditions, two main high-availability set-up have been considered, namely: a set-up with *i)* 3, and *ii)* 5 ONOS controllers belonging to the same cluster and deployed across the servers. Each VM run ONOS 1.10 on Ubuntu 16.04.2 LTS, and was equipped with 8 vCPU and 8GB of RAM (i.e., a *large* VM), while JVM

running the ONOS software was configured to exploit the maximum available RAM. Mininet 2.2 has been adopted to emulate the data plane network, consisting in a *linear topology*, that is a topology of 10 switches connected linearly, and 5 hosts attached to the edge switches. The management of the emulated switches has been distributed between the ONOS instances, so that a East-West communication is triggered whenever a network paths is to be established.

The ONOS cluster is exercised with a workload encompassing Intent installation and withdrawing requests (i.e., host-to-host intents), which are balanced across the controller instances. Specifically, the *Load Generator* (§4.4.6) of the SCP-CLUB framework has been configured in *steady-state* working mode to produce 1,000, and 3,000 requests/s.

Regardless of the number of controllers, the failures are injected in a single controller to make the results more comparable. As discussed in §5.1.3, failures are emulated at both *infrastructure*, i.e., *system* and *network* level, and *SDN controller* level. *Transient* failures are injected after 90s from the end of the warmup phase and, if possible (e.g., system hang cannot be removed once injected), removed after 60s.

Table 5.2 summarizes the values selected for the controllable parameters of the *FI*, which are common to all the experimental settings proposed in this campaign. Each experiment lasts for 300s, and it is repeated for a total

Table 5.2: Experimental parameters adopted for failure injections.

	Failure type	**Configurable Parameters**	**Values**
System Failures	System Hang	n/a	n/a
	System Starvation	n/a	n/a
	System Outage	n/a	n/a
	CPU Shutdown	CPUs to shutdown	6
	Disk Saturation	n/a	n/a
	Memory Saturation	n/a	n/a
	Burn CPU	CPU-bound tasks	100 tasks
	Burn I/O	I/O-bound tasks	100 tasks
Network Failures	Black-hole	Subnet to black hole.	control-plane subnet
	Packet Reject	Reject mode	outgoing packets
	Packet Drop	Drop mode	outgoing packets
	Packet Latency	Latency	200ms
	Packet Loss	Loss percentage	10%
	Packet Reorder[a]	Ordered percentage	30%
		Latency	200ms
	Packet Duplication	Duplication percentage	10%
	Packet Corruption	Corruption percentage	10%
	Throttling	Bandwidth limitation	1 $Mbit$
SDN Controller Failures	Kill Process	Process Name or PID	Controller's PID
	Process Corruption	Process Name or PID	Controller's PID
	Controller Stop	n/a	n/a
	Controller Restart	n/a	n/a
	Dependency Stop	Dependency Name	ONOS core

[a] In order to emulate packets out of order, a percentage of packets are sent in order in a unit time, i.e. to without applying any latency to them, while the remaining packets are sent with a delay.

of 10 runs.

In performing this experimental campaign, the *Experiment Manager* follows the three-phases procedure described in §4.4.4. For reach experiment of the campaign, the VMs hosting the controller instances are first deployed on the blade servers, then the *startup* phase is activated to actually form

the ONOS cluster and start the emulated topology. Then, the *experimental* phase is executed, and the load generator starts by first activating a *warmup* phase to warmup the controller's JVM. The warmup is not accounted in the final results, and the evaluation metrics are computed *from the injection time up to the end of the experiment.* Finally, the experiment ends with the *cleanup* phase by reverting the VMs with a clear snapshot.

5.4 Results

In order to identify if the any of the failure injection test has actually affected the overall system performance, we first computed the throughput (IST) and latency (ISL) of the IBN framework of ONOS (see §4.4.8) in *failure-free* conditions, namely without injecting any failures. Such metrics are then compared with the measurements collected during the failure injection experiments to quantify the possible impact of performance loss of ONOS.

The failure-free performance metrics have been computed as the average over 10 runs. Table 5.3 and Table 5.4 show the throughput and latency results of the failure-free tests for both deploy size, i.e., 3 and 5 controllers scenarios, and for a load level of 1,000 and 3,000 requests/s.

Henceforth, the terms "*target replica*" and "*target controller*" are used

interchangeably to refer to the replica of the ONOS cluster in which the specific failure has been injected.

5.4.1 SDN Service-level Results

5.4.2 System Failures

This section shows the performance results obtained by injecting *system failure* (§5.1.3) on a single instance of the ONOS deploy. Figure 5.10, 5.12, 5.11 and 5.13 show, respectively, the *throughput*, the 50th and 95th percentile *latency* of the ONOS IBN framework, for deploys with 3 and 5 controller instances, computed for failure injection experiments with a load levels of 1,000 and 3,000 requests/s.

Both types of experiments show that most of the emulated failures impact the overall system performance. The performance degradation seems to be more evident as the load increases, e.g, from 1,000 to 3,000 requests/s.

Figure 5.10 shows that the system *hang*, *starvation* and *outage*, as well as *memory* and *disk saturation* failures impact more on the 3 controllers

Table 5.3: Failure free Intent requests' Throughput for 3 and 5 controller scenario under 1,000, and 3,000 Load Levels.

	Throughput [req/s] LoadLevel 1000	**Throughput [req/s] LoadLevel 3000**
3 Controller	997.52	2978.32
5 Controller	997.39	2987.68

Table 5.4: Failure free Intent requests' Latency for 3 and 5 controller scenario under 1,000, and 3,000 Load Levels.

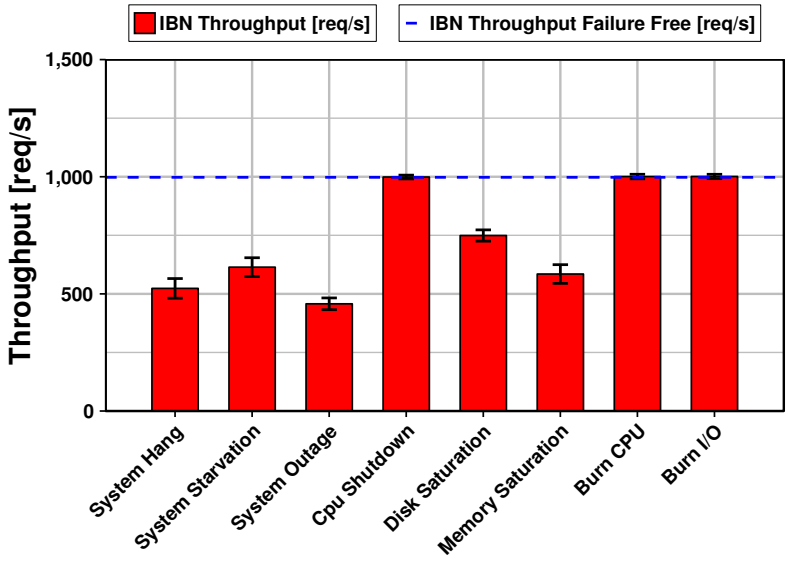
	Latency [ms] LoadLevel 1000		**Latency [ms] LoadLevel 3000**	
	50th %ile	**90th %ile**	**50th %ile**	**90th %ile**
3 Controller	46	84	128	1007
5 Controller	45	81	89	173

scenario, causing a throughput loss between 30% and 60% compared to the failure-free scenario. A similar observation applies to most of the failure injection experiments reported in this chapter. This highlights the criticality of a deploy consisting of 3 controllers, where a single instance affected by failures can lead to the unavailability of the whole ONOS cluster.

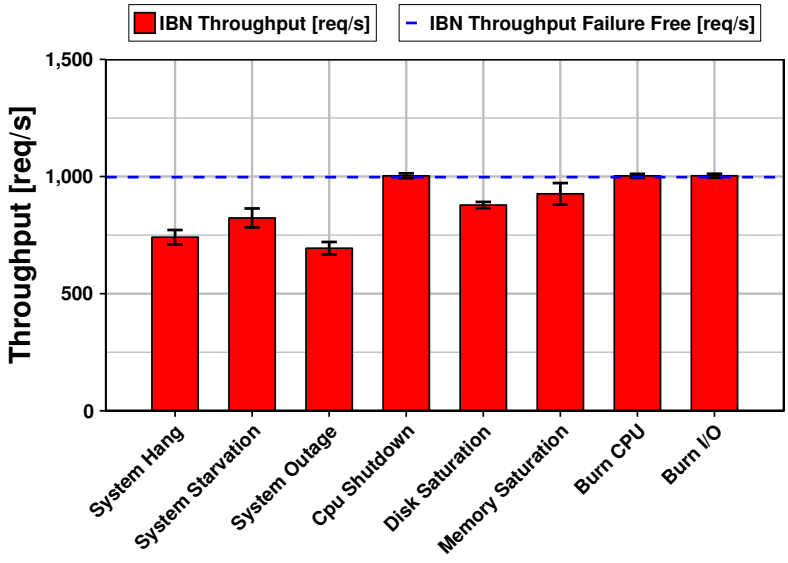
As can be seen in Figure 5.10 and Figure 5.12, the *system hang*, *starvation* and *outage* failures impact severely on the ONOS performance. System *outage* cause the crash of the target instance, while the system *hang* and *starvation* failures drastically reduce its responsiveness. The crash of the target replica is promptly detected by the other replicas. However, as can be seen in Figure 5.10, with a load as high as 1,000 request/s, the ONOS cluster seems unable to properly recover from the injected *outage* failure. This behaviour is explained by the fact that the ONOS replicas not affected by the failure were still considering the target instance as master for specific devices. Therefore, each flow rule operation sent to the target replica failed without triggering any mitigation mechanisms, e.g., a the election of a new

master for the devices previously managed by the target instance; that lead to a high unavailability and performance loss. Similar observations also apply for system *hang* and *starvation* failure.

As can be seen in Figure 5.10 and Figure 5.12, *memory* and *disk saturation* failures shows different effects between the two deployment scenarios, and they both indirectly affect the consistency of the data shared between the replicas. Indeed, ONOS maintains in memory the information related to the network state according to an eventually consistent approach, while the other pieces of information, namely the intents and system configurations, are stored in a persistent manner (on the disc). In particular, ONOS relies on the the *Raft* [71] consensus algorithm implemented by the adopted *Copycat* [69] framework for consistency and data replication. To accomplish this, each server in the cluster maintains a separate copy of the system state machine along with a log of all operations that have been performed on that state machine and their results. Logs are durably stored on disk and are used to restore the state of a machine in the event of a failure. Hence, the *disk* saturation failure lead the target controller to be no longer able to update its local state machine, hence affecting the consistency of the subset of data managed by the target replica. Therefore, the throughput degradation was mainly caused by the fact that the target replica was accepting intent requests without being able neither to process them locally, nor to share

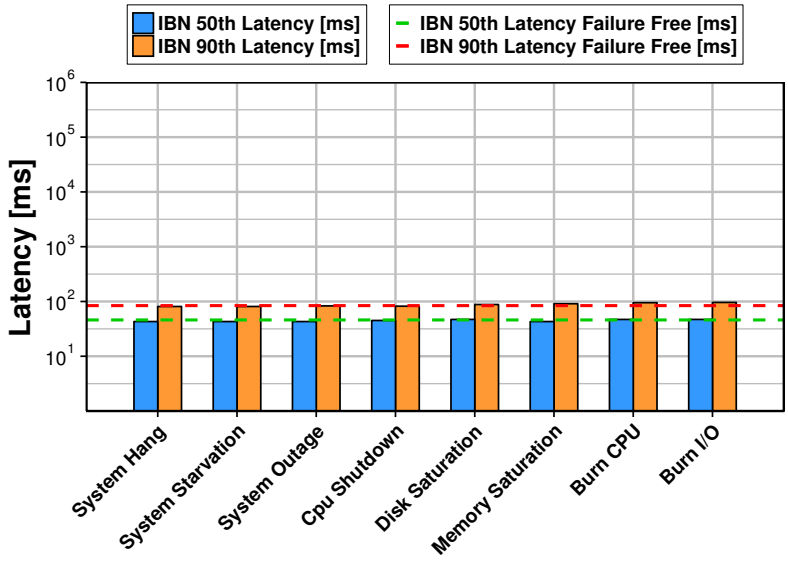


(a) 3 ONOS controllers

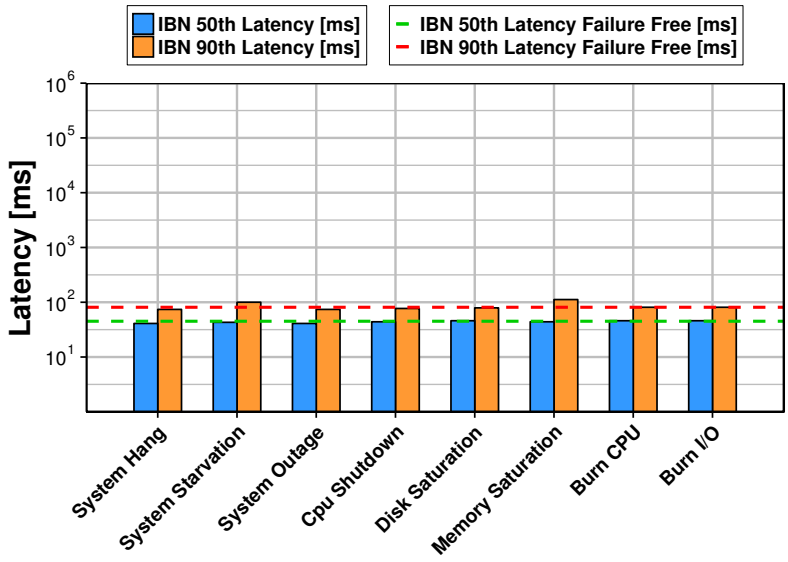


(b) 5 ONOS controllers

Figure 5.10: Service throughput with *system* failures injection; 3 and 5 controllers; workload 1,000 requests/s.



(a) 3 ONOS controllers



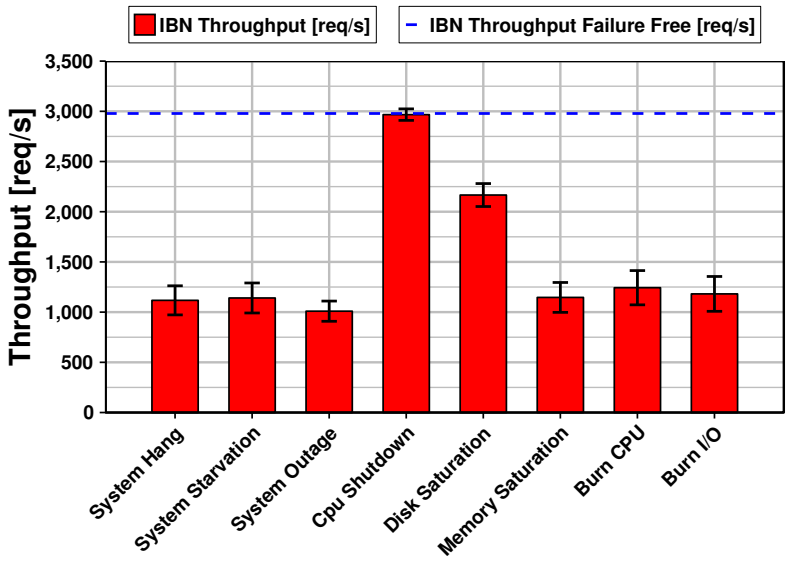
(b) 5 ONOS controllers

Figure 5.11: Service latency with *system* failures injection; 3 and 5 controllers; workload 1,000 requests/s.

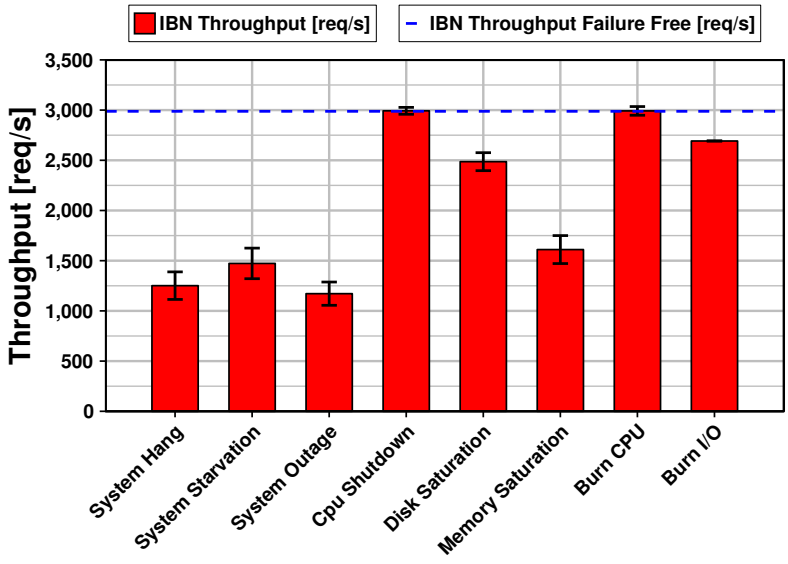
them with the other replicas.

The *memory saturation* failure has caused a higher performance degradation than the *disk saturation* one, for both deployment scenarios, as well as, for both tested load levels. A closer look to the system log files, has shown that after the injection of *memory saturation* failures, all the ONOS replicas were experiencing unstable connections towards the the data plane devices, causing then the disconnection of hosts. This turns out to the failure of the intent compilation, since no valid path can be found anymore.

Other typea of injected failures, such as *CPU* and *I/O burn*, and *CPU shutdown*, show no direct impact on the performance for both deployments with 3 and 5 controllers, and a load of 1,000 requests/s (see Figure 5.10). A different story is dictated by Figure 5.12 and Figure 5.13, showing that most of the injected failures negatively impact both latency and throughput performance of both types of deploy, when a load of 3,000 requests/s is submitted to the system. Figure 5.12a and Figure 5.13a show that *CPU* and *I/O burn* failures drastically affect the performance and latency of the 3 controllers deployment. Both failures show similar effects to those obsoverd by injecting *fill memory* failures. The *I/O burn* failure has slightly affected also the deploy consisting of 5 controllers. Indeed, the injection of such failure delays disk operations, causing the processing of intent requests to slow down.

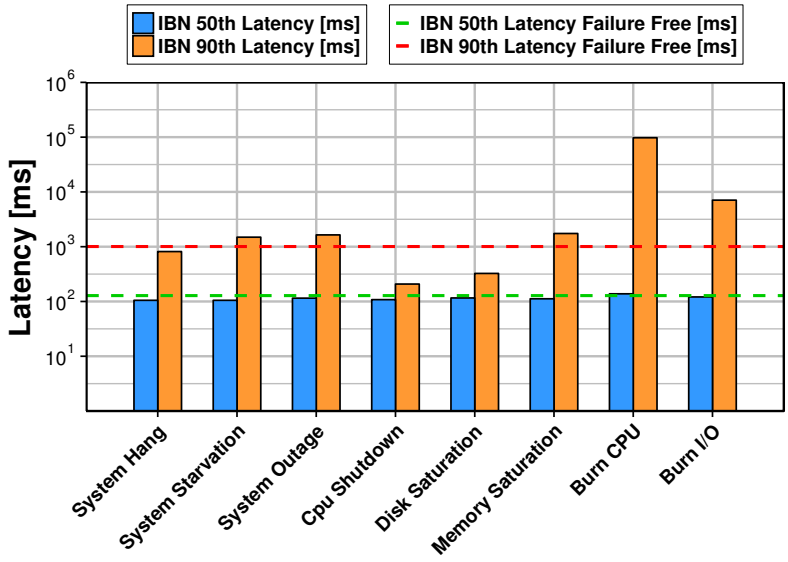


(a) 3 ONOS controllers

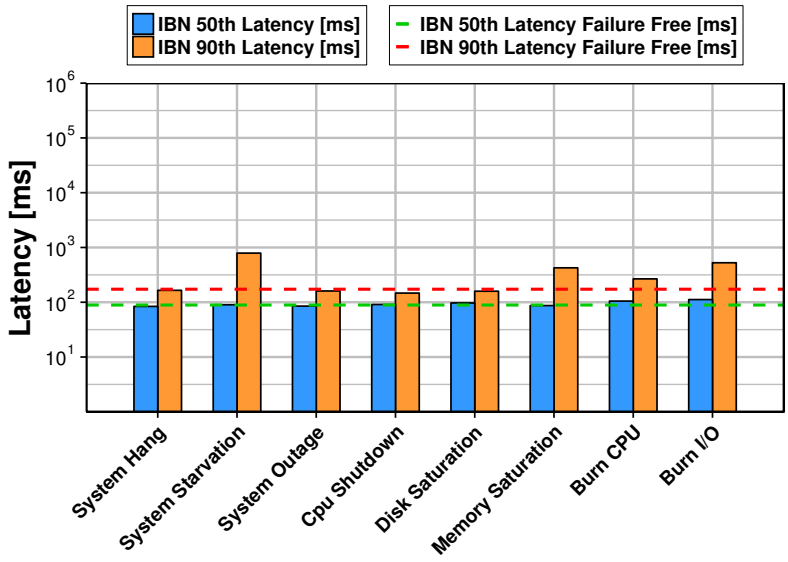


(b) 5 ONOS controllers

Figure 5.12: Service throughput with *system* failures injection; 3 and 5 controllers; workload 3,000 requests/s.



(a) 3 ONOS controllers



(b) 5 ONOS controllers

Figure 5.13: Service latency with *system* failures injection; 3 and 5 controllers; workload 3,000 requests/s.

The *CPU shutdown* failure appear to be the only one not affecting the performance at all. This can be explained by the fact that the operations performed by the ONOS process are mostly I/O bound, requiring continuous access to memory and disk while gather and update the data related to the system state machine, as well as to the data plane status.

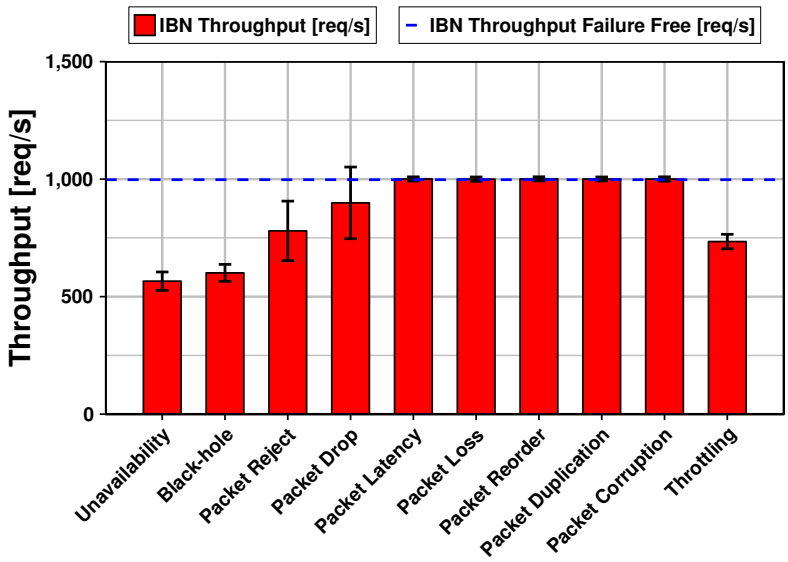
In this set of experiments, it has been observed that no mechanisms are provided by ONOS to detect and mitigate possible resource depletion due to other user – and/or kernel – level tasks running on the same machine hosting the ONOS process. Indeed, most of the experiments involving failures mimicking resource saturation or corruption lead to an inconsistent state of the control plane. Although in such a state the target ONOS instance is still able to interact with its peers participating in the management of the cluster events, under these faulty conditions it is not able to accomplish most of its tasks (e.g. interacting with a managed switch, or satisfy an intent request). Consequently, the target instance continually triggers abnormal events (e.g., timeout exceptions in the cluster communication), as it is slow yet not believed crashed, forcing the other replicas to reply to such issues, taking away useful resources to process the incoming request load.

5.4.3 Network Failures

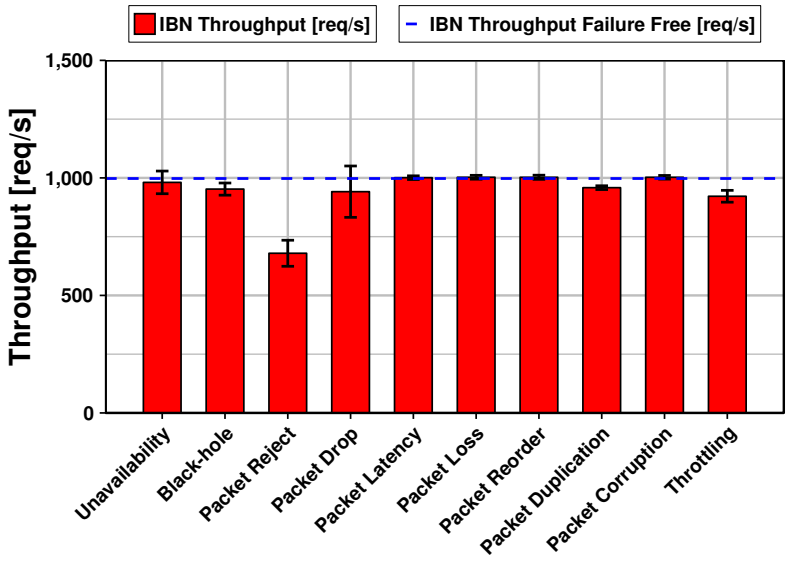
Figure 5.14, 5.16, 5.15 and 5.17 show, respectively, the *throughput*, the 50th and 95th percentile *latency* of the ONOS IBN framework, for deploys with 3 and 5 controller instances, computed during experiments with the injection of *network* failures and load levels of 1,000 and 3,000 requests/s.

The results show that network failures impact very differently in the two deploy scenarios. Figure 5.14a and Figure 5.15a shows again the vulnerability of a deploy consisting of 3 ONOS instances, even under a load as high as 1,000 requests/s.

Both Figure 5.14 and Figure 5.16 show that the ONOS cluster tolerates most of the injected network failures. Indeed, packet *latency*, *loss*, *reorder*, *duplication* and *corruption* do not impact on the overall performance of both deployment scenarios, since most of these failures are detected and mitigated by the TCP protocol stack (e.g., packet corruption and reordering failures) and the heartbeats mechanism adopted by the ONOS Copycat framework (e.g. latency failure). The *throtthling*, *unavailability*, *balck-hole*, *packet reject* and *drop* network failures caused a non-negligible amount of performance loss, for both throughput and latency. With a load as high as 1,000 requests/s, the 5 controllers deploy (see Figure 5.14b and Figure 5.15b) seems to be more resilient to these type of network failures than the case with 3 controllers (see Figure 5.14a and Figure 5.17). Similar observations

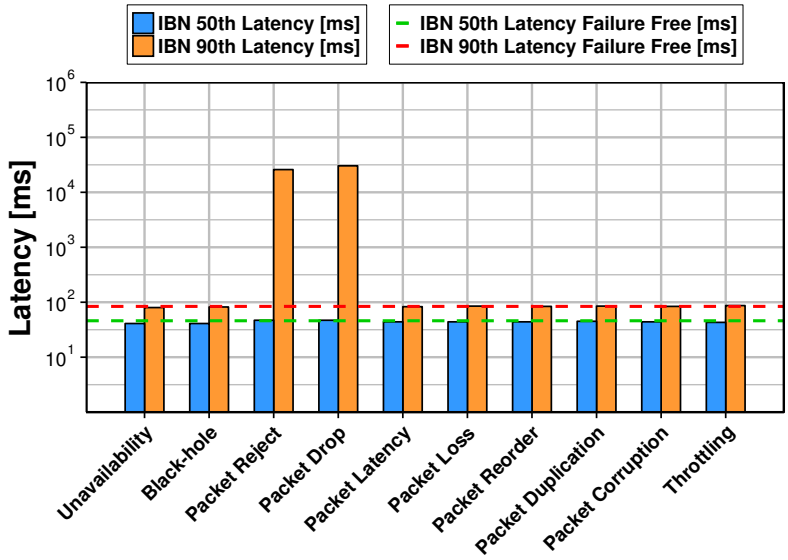


(a) 3 ONOS controllers

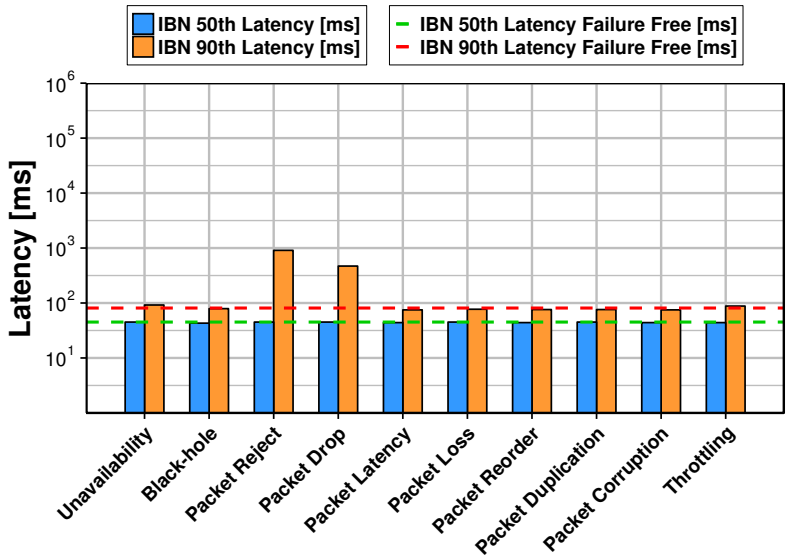


(b) 5 ONOS controllers

Figure 5.14: Service throughput with *network* failures injection; 3 and 5 controllers; workload 1,000 requests/s.



(a) 3 ONOS controllers



(b) 5 ONOS controllers

Figure 5.15: Service latency with *network* failures injection; 3 and 5 controllers; workload 1,000 requests/s.

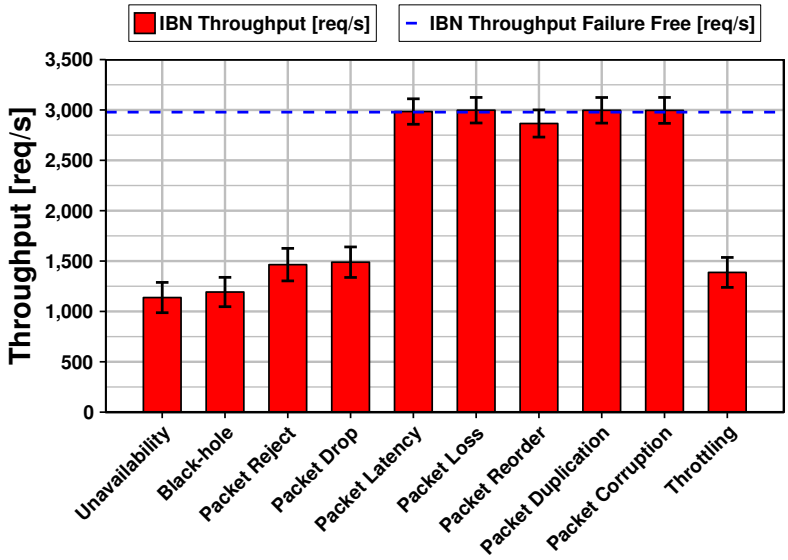
do not apply to the scenario with a higher request rate, i.e., with 3,000 requests/s. Indeed, as can be seen in Figure 5.16 and Figure 5.16b, both deploys show significant loss of performance, both in terms of throughput and latency.

Figure 5.18 shows an example of the system throughput when a *packet reject* failure is injected in the target ONOS instance. As can be noticed, after the injection time (left line of the red area in 5.18) the throughput drastically drops towards zero for the 3 controllers scenario, while undergoing a significant reduction for the scenario with 5 controllers. In the latter case the system shows to be unable to quickly mitigate such failure, taking long time to recover, even after the failure is removed (right line of the red area in 5.18).

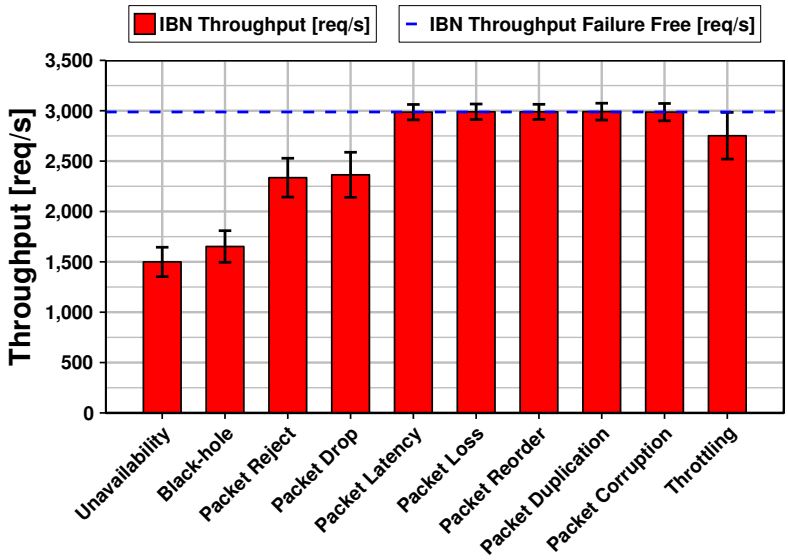
Analyzing system's log, it has been observed that all the network failures affecting the system performance have led to three major faulty situations:

1. The ONOS instances which keep loosing the connectivity with the data plane due to the injection of network failures, try repeatedly, but unsuccessfully, to re-establish the interaction with the data plane, consuming system resources to a remarkable extent;

2. When an ONOS instance is not promptly responsive due to network failures, the other replicas initiate a mastership election for the data

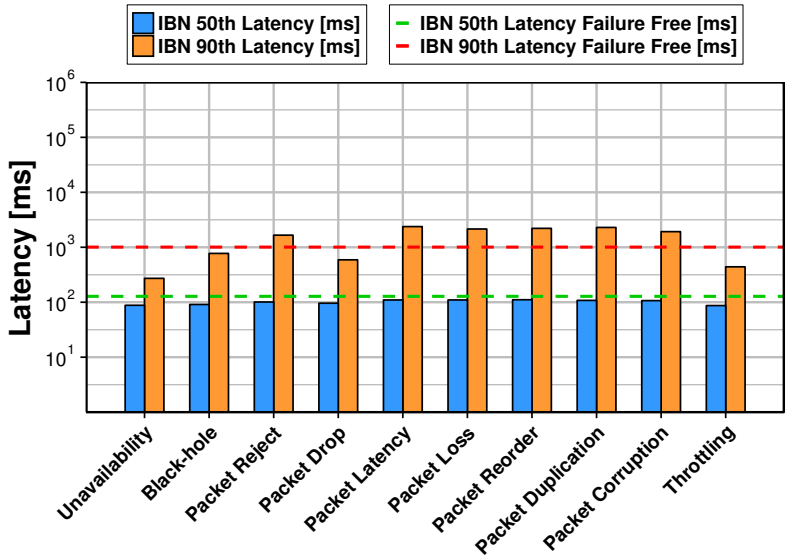


(a) 3 ONOS controllers

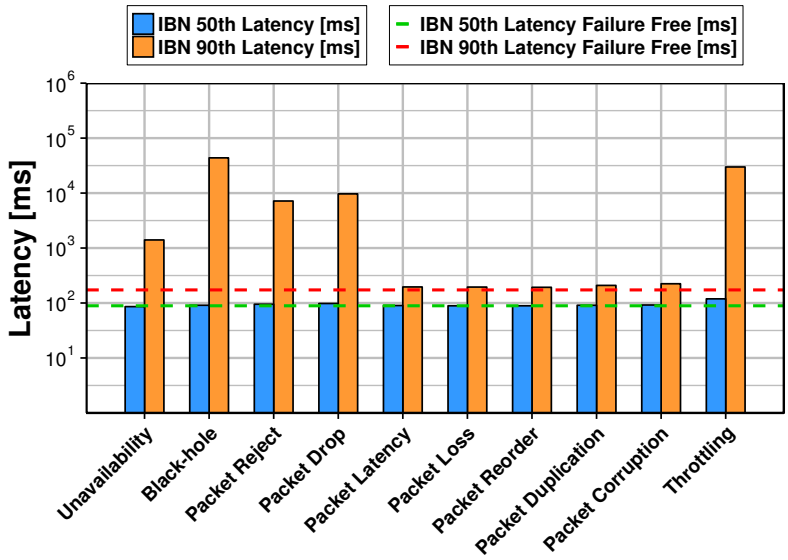


(b) 5 ONOS controllers

Figure 5.16: Service throughput with *network* failures injection; 3 and 5 controllers; workload 3,000 requests/s.

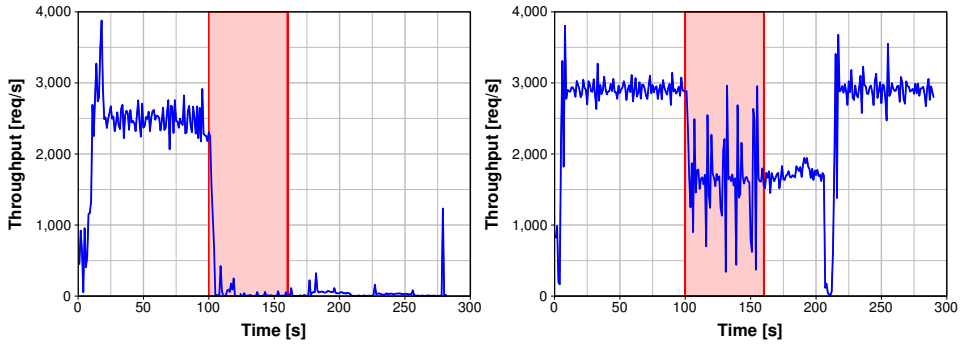


(a) 3 ONOS controllers



(b) 5 ONOS controllers

Figure 5.17: Service latency with *network* failures injection; 3 and 5 controllers; workload 3,000 requests/s.



(a) 3 ONOS controllers (b) 5 ONOS controllers

Figure 5.18: Performance degradation due to *network packet reject* injection; 3 and 5 controllers; workload 3,000 requests/s.

plane devices, and for the new assignee of the intents previously managed by the apparently faulty instance. However, they experience issues in reprogramming the device mastership, which in turn triggers further leadership election sessions. This is due to the fact that the ONOS instance is still considered as master by its switches, as it is slow yet not believed crashed. This is especially the case for *packet drop*, *reject*, and *throttling* failures, which lead the cluster in a state in which the *controller-device* mastership status returns inconsistent results across all instances;

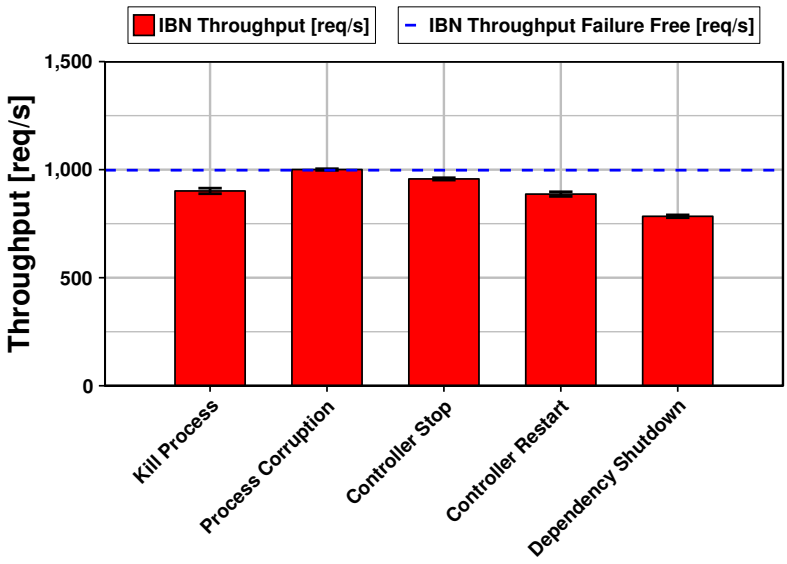
3. As a consequence of the point 2, the ONOS replicas keep recompiling and re-executing the failed intents. However, it has been observed that the recompilation or the re-execution process often fail, causing an excessive resource consumption: this is possibly due to the fact

that ONOS keeps allocation new threads for the compilation and installation of the intents.

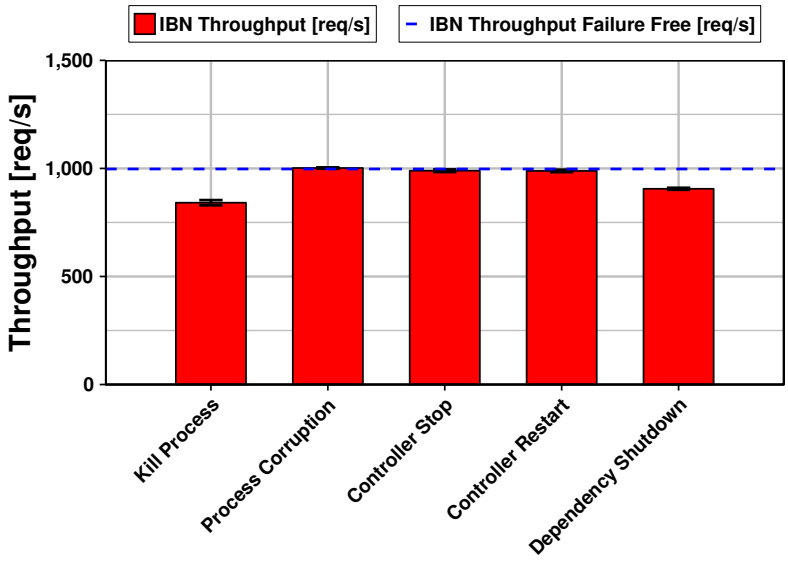
5.4.4 SDN Controller Failures

This section describes the performance results obtained by emulating failures directly affecting the ONOS instance, such as *process* and *dependency* failures. Figure 5.19 and Figure 5.21 show that the *controller stop*, *restart*, and *dependency shutdown* failures, as well as the *kill* of the Java process of the target instance, cause a non negligible performance loss, both in terms of throughput and latency. All these failures lead to the abnormal termination of a target instance, triggering the internal failover mechanism. In particular, the other ONOS instances started a leader election session to elect a new master for all the devices managed by the target instance. This means that for a brief period (seconds) the cluster will be unavailable.

Although the failover seems to work properly in detecting and mitigating the injected failure for the scenario with a low request rate (e.g., 1,000 requests/s), in which it only causes a small degradation of the system throughput, it is not the case of the scenario with a high request rate (e.g., 3,000 requests/s). Indeed, Figure 5.21 and Figure 5.22 show that the system suffers a high performance degradation, in terms of throughput and latency. This behaviour was due to the service managing the flow rule operations,

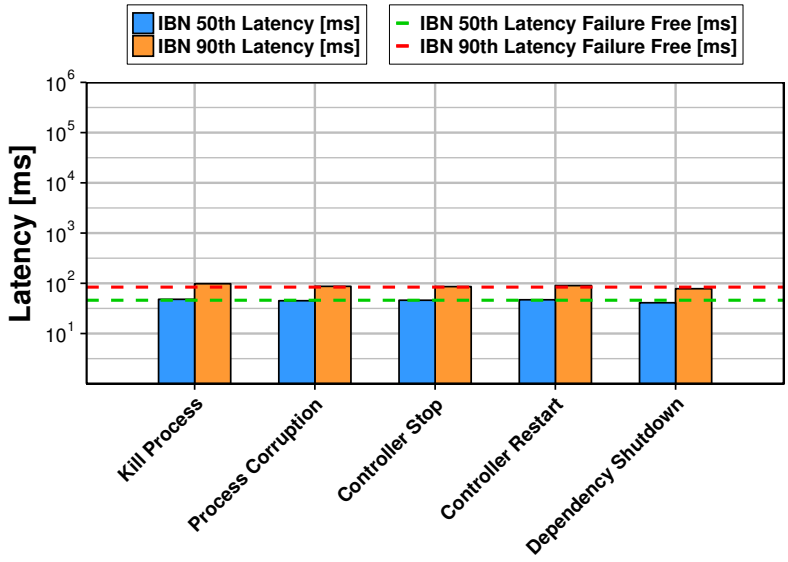


(a) 3 ONOS controllers

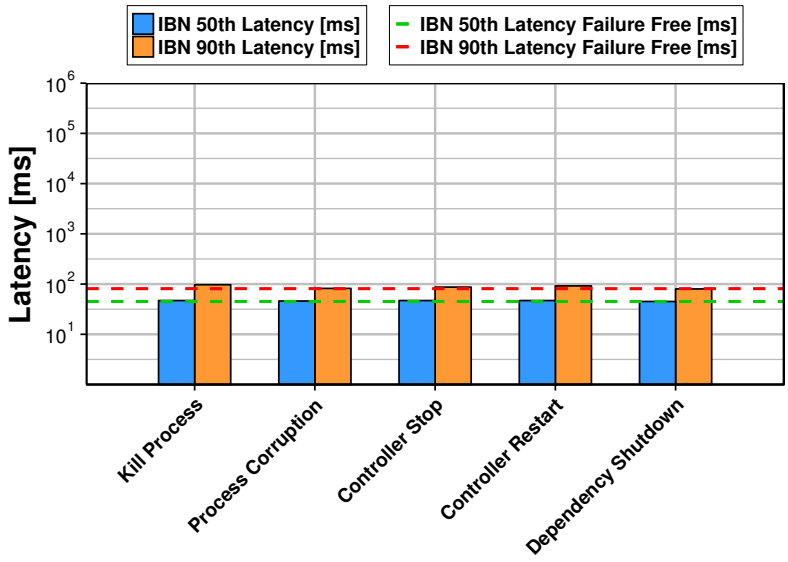


(b) 5 ONOS controllers

Figure 5.19: Throughput with *controller* failure injection; 3 and 5 controllers; workload 1,000 requests/s.



(a) 3 ONOS controllers

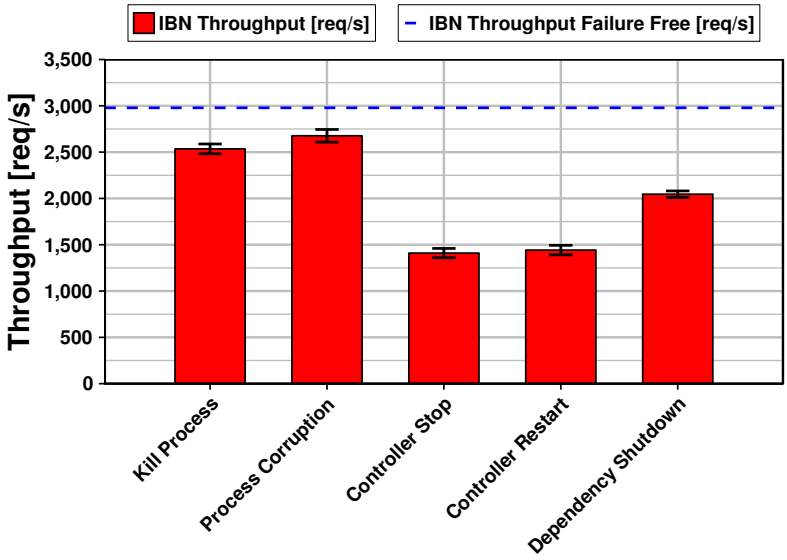


(b) 5 ONOS controllers

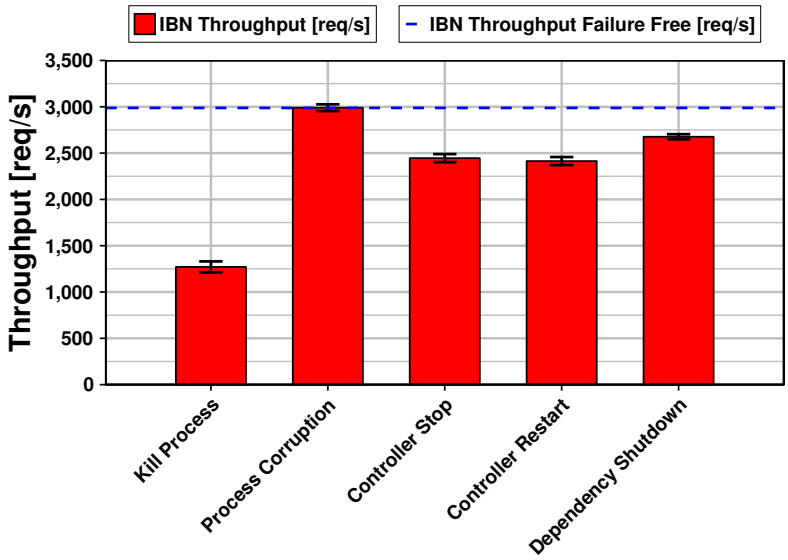
Figure 5.20: Service latency with *controller* failures injection; 3 and 5 controllers; workload 1,000 requests/s.

which repeatedly experienced failure while trying to re-program the data plane devices to restore the intents previously installed by the target node.

Furthermore, the *kill process* failure show that sometimes the provided failover mechanisms prove to be inefficient in mitigating the termination of a single replica. Indeed, such failure has introduced a faulty situation, already observed with the injection of other types of failures, in which the ONOS instances keep loosing the connection with the data plane devices, thus causing the connected hosts to be removed from the network view. This in turn causes the failure of most of the submitted intents, since it is no longer possible to identify a valid path to create a connection.

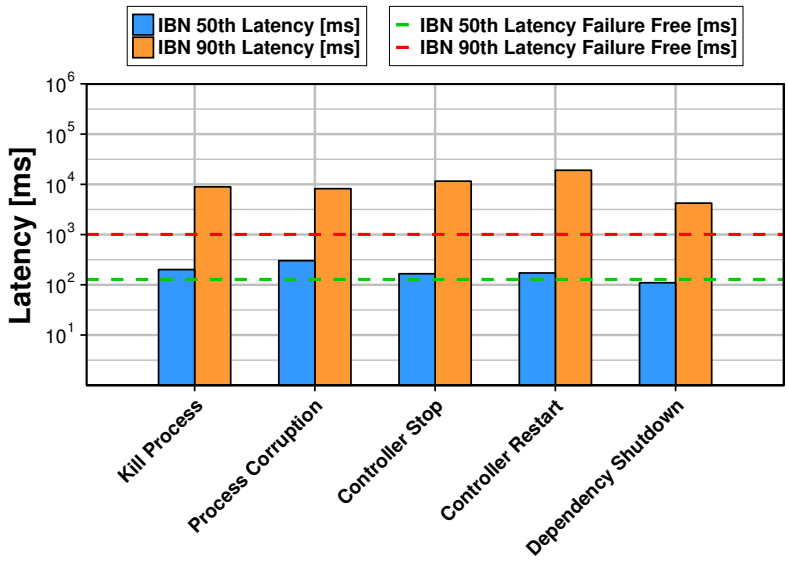


(a) 3 ONOS controllers

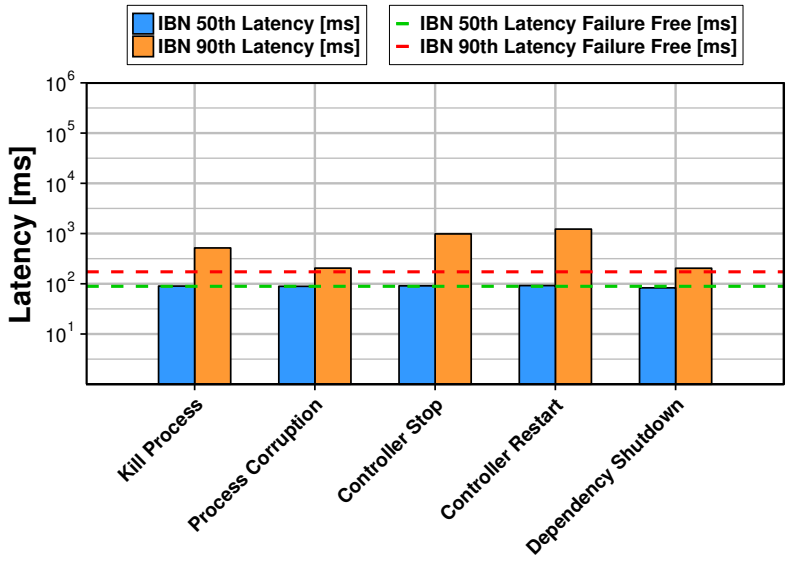


(b) 5 ONOS controllers

Figure 5.21: Throughput with *controller* failure injection; 3 and 5 controllers; workload 3,000 requests/s.



(a) 3 ONOS controllers



(b) 5 ONOS controllers

Figure 5.22: Service latency with *controller* failures injection; 3 and 5 controllers; workload 3,000 requests/s.

That's all folks.

Popular TV cartoons closing

Chapter 6

Conclusions

We summarize the research problem addressed and the main contributions provided by this dissertation.

In the era of the highly and always connected information society, Software-Defined Networking is a very hot research area, and SDN are foreseen to have a huge potential market in the forthcoming years. Among the several open issues, this dissertation has identified two still open research problems: *i)* assessment of SDN performance and *ii)* in production assessment of SDN failure resilience, with reference to real industrial telco cloud data centers. Their relevance is due to the fact that network operating systems (SDN controllers) are very complex distributed systems - subject to performance and dependability requirements as severe as those of current carrier-grade networks - for which current experimental evaluations cannot be trusted by telco operators, and traditional software testing techniques appear insufficient for dependability assessment, given the difficulty to reproduce the

many potential failures which can actually occur in operation and may affect at all levels their many software components.

The thesis has contributed to these research issues with a twofold proposal. The presented SCP-CLUB framework automates experimental campaigns for SDN performance evaluation in real telco cloud data centers. As for the assessment of SDN failure resilience, the proposal extends SCP-CLUB with the use of software failure injection. A methodology has been devised for *in-fabric* test, as well as for *in-production* assessment, with the aim of continuous testing. The methodology in complemented by an infrastructure specifically designed to be integrated with limited intrusiveness into distributed SDN controllers, in order to support the execution of failure-injection experiments.

The resilience assessment methodology and the infrastructure are conceived to be usable in a controlled test environment, as well as in a normal operational environment for future SDN platforms (including virtualized and container-based SDN controllers). For in-fabric test, a workload consisting of *intents* may be generated and then submitted to the platform. Failures are injected during the execution, which is monitored so as to gather data of interest for analysis. Failures belong to a failure model representative of typical classes of events occurring in operation at network, system and service level. For in-production assessment, failures are injected when

the controller under assessment is subject to normal operating conditions (under the real load). A set of metrics to evaluate controllers' resilience to failures have been proposed, too.

The proposed SCP-CLUB framework and the failure injection infrastructure have been implemented and experimented with reference to the open-source ONOS™ network operating system. The workload is automatically generated based on the Intent-Based Networking (IBN) model. The implementation is based on the Linux, Java, Apache ActiveMQ and Apache Karaf technologies.

The experimental evaluation has shown that the framework can be effectively applied to assess the performance of SDN deployments in telco clouds, and the controllers' resilience mechanisms (for failure detection and mitigation), as well as to quantify system availability and reliability. The experiments have been performed at the prestigious Murray Hill NOKIA Bell Labs in New Providence, New Jersey, USA, in the framework of a continuous and in-production testing strategy for the future generation of network solutions.

This page intentionally left blank.

Bibliography

[1] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *IEEE Communications Surveys Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[3] E. Hernandez-Valencia, S. Izzo, and B. Polonsky, "How Will NFV/SDN Transform Service Provider OpEx?," *IEEE Network*, vol. 29, no. 3, pp. 60–67, 2015.

[4] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[5] Allied Market Research, "Global Software-Defined Networking Market: Opportunities and Forecasts, 2015 - 2022." `www.alliedmarketresearch.com/software-defined-networking-market`, 2016. [Online; accessed 21-September-2017].

[6] S. Russo, "Finding a way in the Model Driven jungle," in *Proceedings of the 9th India Software Engineering Conference (ISEC)*, pp. 13–15, ACM, 2016.

[7] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 13, pp. 105–110, 2008.

[8] D. Erickson, "The Beacon OpenFlow Controller," in *Proceeding of the 2nd Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pp. 13–18, ACM, 2013.

[9] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a Model-Driven SDN Controller architecture," in *Proceedings of IEEE 15th International Symposium on a World of Wireless, Mobile and Multimedia Networks*, IEEE, 2014.

[10] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. O. Bob Lantz, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: towards an open, distributed SDN OS," in *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pp. 1–6, ACM, 2014.

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

[12] P. Bosch, A. Duminuco, F. Pianese, and T. L. Wood, "Telco Clouds and Virtual Telco: Consolidation, Convergence, and Beyond," in *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 982–988, IEEE, 2011.

[13] J. Soares, C. Gonçalves, B. Parreira, P. Tavares, J. Carapinha, J. P. Barraca, R. L. Aguiar, and S. Sargento, "Toward a telco cloud environment for service functions," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 98–106, 2015.

[14] X. Zhiqun, C. Duan, H. Zhiyuan, and S. Qunying, "Emerging of telco cloud," *China Communications*, vol. 10, no. 6, pp. 79–85, 2013.

[15] J. Soares, C. Gonçalves, B. Parreira, P. Tavares, J. Carapinha, J. P. Barraca, R. L. Aguiar, and S. Sargento, "Toward a telco cloud environment for service functions," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 98–106, 2015.

[16] C. Di Martino, V. Mendiratta, and M. Thottan, "Resiliency Challenges in Accelerating Carrier-Grade Networks with SDN," in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 242–245, IEEE, 2016.

[17] J.-C. Laprie, "From dependability to resilience," in *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, pp. G8–G9, Citeseer, 2008.

[18] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[19] A. S. Team *et al.*, "Amazon s3 availability event: July 20, 2008," *Retrieved November*, vol. 15, p. 2008, 2008.

[20] E. Bauer and R. Adams, *Reliability and availability of cloud computing*. John Wiley & Sons, 2012.

[21] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, and M. Zhu, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[22] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos Engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[23] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing Dependability with Software Fault Injection: A Survey," *ACM Computing Surveys*, vol. 48, no. 3, pp. 44:1–44:55, 2016.

[24] ONOS Project, "ONOS White Paper." `http://onosproject.org/wp-content/uploads/2014/11/Whitepaper-ONOS-final.pdf`, 2014. [Online; accessed 20-September-2017].

[25] ONOS Project, "Intent-based framework." `http://wiki.onosproject.org/display/ONOS/Intent+Framework`. [Online; accessed 20-September-2017].

[26] H. Farhady, H. Lee, and A. Nakao, "Software-defined networking: A survey," *Computer Networks*, vol. 81, pp. 79–95, 2015.

[27] The Open Networking Foundation (ONF). `http://onlab.us`. [Online; accessed 20-September-2017].

[28] D. Lenrow, "Intent As The Common Interface to Network Resources." Presentation at the Intent Based Network Summit 2015, Palo Alto, CA, USA. Available at `www.ietf.org/mail-archive/web/i2nsf/current/pdfEhAfL7kT9F.pdf(AccessedJanuary2017)`, 2015.

[29] Open Networking Foundation, OpenFlow. `www.opennetworking.org/sdn-resources/openflow`. [Online; accessed January 2017].

[30] ONOS Project, "Onos mission." `http://onosproject.org/mission/`. [Online; accessed 20-September-2017].

[31] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On Scalability of Software-Defined Networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

[32] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM SIGACT News*, vol. 33, no. 2, 2002.

[33] C. R. Johnson, Y. Kogan, Y. Levy, F. Saheban, and P. Tarapore, "Voip Reliability: A Service Provider's Perspective," *IEEE Communications Magazine*, vol. 42, no. 7, pp. 48–54, 2004.

[34] A. Akella and A. Krishnamurthy, "A Highly Available Software Defined Fabric," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets)*, pp. 1–7, ACM, 2014.

[35] J.-F. Castet and J. H. Saleh, "Survivability and Resiliency of Spacecraft and Space-Based Networks: a Framework for Characterization and Analysis," in *AIAA SPACE Conference*, American Institute of Aeronautics and Astronautics, 2008.

[36] The ResiliNets Initiative, "ResiliNets Wiki." `https://wiki.ittc.ku.edu/resilinets_wiki/index.php/Definitions#Resilience`. [Online; accessed 20-September-2017].

[37] P. Smith, D. Hutchison, J. P. Sterbenz, M. Schöller, A. Fessi, M. Karaliopoulos, C. Lac, and B. Plattner, "Network resilience: a systematic approach," *IEEE Communications Magazine*, vol. 49, no. 7, pp. 88–97, 2011.

[38] D. Henry and J. E. Ramirez-Marquez, "Generic metrics and quantitative approaches for system resilience as a function of time," *Reliability Engineering & System Safety*, vol. 99, pp. 114–122, 2012.

[39] J.-C. Laprie, "Resilience for the scalability of dependability," in *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA)*, pp. 5–6, IEEE, 2005.

[40] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "OpenFlow: Meeting carrier-grade recovery requirements," *Computer Communications*, vol. 36, no. 6, pp. 656–665, 2013.

[41] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[42] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[43] The Netflix Tech Blog, "Netflix Chaos Monkey 2.0." `http://techblog.netflix.com/2016/10/netflix-chaos-monkey-upgraded.html`, 2017. [Online; accessed January 2017].

[44] S. Huang, Z. Deng, and S. Fu, "Quantifying entity criticality for fault impact analysis and dependability enhancement in software-defined networks," in *Proceedings of the IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8, IEEE, 2016.

[45] A. Tootoonchian et al., "Cbench: an Open-Flow Controller Benchmarker." `https://github.com/mininet/oflops/tree/master/cbench`.

[46] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. 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)*, USENIX Association, 2012.

[47] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries, "A flexible OpenFlow-controller benchmark," in *Proceedings of the European Workshop on Software Defined Networking (EWSDN)*, pp. 48–53, IEEE, 2012.

[48] Y. Zhao, L. Iannone, and M. Riguidel, "On the performance of SDN controllers: A Reality Check," in *Proceedings of the IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pp. 79–85, IEEE, 2015.

[49] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and Performance Evaluation of an OpenFlow Architecture," in *Proc. of the 23rd Int. Teletraffic Congress (ITC)*, IEEE, 2011.

[50] K. He, J. Khalid, S. Das, A. Gember-Jacobson, C. Prakash, A. Akella, L. E. Li, and M. Thottan, "Latency in Software Defined Networks: Measurements and Mitigation Techniques," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1, pp. 435–436, 2015.

[51] J. Hu, C. Lin, X. Li, and J. Huang, "Scalability of control planes for software defined networks: Modeling and evaluation," in *Proceedings of the IEEE 22nd International Symposium of Quality of Service (IWQoS)*, pp. 147–152, IEEE, 2014.

[52] M. Karakus and A. Durresi, "A Scalability Metric for Control Planes in Software Defined Networks (SDNs)," in *Proceedings of the IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pp. 282–289, IEEE, 2016.

[53] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pp. 7–12, ACM, 2012.

[54] P. Fonseca, R. Bennesby, and E. Mota, "A Replication Component for Resilient OpenFlow-based Networking," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, IEEE, 2012.

[55] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, *Distribyted systems*, ch. The primary-backup approach, pp. 199–216. ACM Press/Addison-Wesley Publishing Co., 2nd ed., 1993.

[56] F. J. Ros and P. M. Ruiz, "Five Nines of Southbound Reliability in Software-Defined Networks," in *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pp. 31–36, ACM, 2014.

[57] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[58] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "On Fault Representativeness of Software Fault Injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[59] D. Cotroneo, L. D. Simone, A. K. Iannillo, A. Lanzaro, and R. Natella, "Dependability Evaluation and Benchmarking of Network Function Virtualization Infrastructures," in *Proceedings of the 1st IEEE Conference on Network Softwarization (NetSoft)*, pp. 1–9, IEEE, 2015.

[60] The Netflix Tech Blog, "Principles of chaos engineering." `http://principlesofchaos.org/`, 2015. [Online; accessed 31-January-2017].

[61] The Netflix Tech Blog, "Fit: Failure injection testing." `http://techblog.netflix.com/2014/10/fit-failure-injection-testing.html`, 2015. [Online; accessed 31-January-2017].

[62] P. Alvaro, K. Andrus, C. Sanden, C. Rosenthal, A. Basiri, and L. Hochstein, "Automating failure testing research at internet scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 17–28, ACM, 2016.

[63] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[64] ONOS Project, "ONOS Test Plan - Performance and Scale-out." `https://wiki.onosproject.org/pages/viewpage.action?pageId=3441823(accessedJuly2017)`, February 2017.

[65] Apache Software Foundation, Apache Karaf. `http://karaf.apache.org`. [Online; accessed 21-September-2017].

[66] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[67] A.-M. Kermarrec and M. Van Steen, "Gossiping in distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 2–7, 2007.

[68] E. Brewer, "CAP twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[69] The Copycat framework. `http://atomix.io/copycat/`. [Online; accessed 20-September-2017].

[70] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference*, pp. 305–319, USENIX Association, 2014.

[71] The Raft Consensus Algorithm. `https://raft.github.io/`, 2014. [Online; accessed 10-September-2017].

[72] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *Proc. of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*, ACM, 2010.

[73] Centec Networks. `http://www.centecnetworks.com`. [Online; accessed 19-September-2017].

[74] InfluxData, InfluxDB Time-Series Data Storage. `http://www.influxdata.com`. [Online; accessed 20-September-2017].

[75] Grafana Labs, Grafana. `http://grafana.com`. [Online; accessed 20-September-2017].

[76] Vmware Inc., Vmware PowerCLI. `https://www.vmware.com/support/developer/PowerCLI/`. [Online; accessed 21-September-2017].

[77] Pivotal Software Inc., Spring Boot. `https://projects.spring.io/spring-boot/`. [Online; accessed 21-September-2017].

[78] Iperf3. `https://iperf.fr/iperf-download.php`. [Online; accessed 21-September-2017].

[79] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, "Mind the Gap: Monitoring the Control-Data Plane Consistency in Software Defined Networks," in *Proc. of the 12th Int. Conference on emerging Networking EXperiments and Technologies*, pp. 19–33, ACM, 2016.

[80] R. Jain, *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley & Sons, 1990.

[81] Vmware Inc., VMware ESXi. `https://my.vmware.com/en/web/vmware/evalcenter?p=free-esxi6`. [Online; accessed 21-September-2017].

[82] Centec Networks, V350 Series OpenFlow/SDN Switch. `http://www.centecnetworks.com/en/SolutionList.asp?ID=43`. [Online; accessed 19-September-2017].

[83] ONOS bug reports. `https://jira.onosproject.org/browse/ONOS-4978`. [Online; accessed 21-September-2017].

[84] ONOS bug reports. `https://jira.onosproject.org/browse/ONOS-6780`. [Online; accessed 21-September-2017].

[85] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying NOX to the datacenter," in *Proceedings of the 8th Workshop on Hot Topics in Networks (HotNets-VIII)*, ACM, 2009.

[86] L. J. Jagadeesan and V. Mendiratta, "Programming the Network: Application Software Faults in Software-Defined Networks," in *Proceedings of the IEEE 27th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 125–131, IEEE, 2016.

[87] R. M. Lefever, M. Cukier, and W. H. Sanders, "An experimental evaluation of correlated network partitions in the Coda distributed file system," in *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS)*, pp. 273–282, IEEE, 2003.

[88] "Network faults in distributed systems." `https://github.com/aphyr/partitions-post`, 2014.

[89] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva, "On fault resilience of OpenStack," in *Proceedings of the 4th annual Symposium on Cloud Computing (SOCC)*, ACM, 2013.

[90] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung, "Testing of fault-tolerant and real-time distributed systems via protocol fault injection," in *Proceedings of Annual Symposium on Fault Tolerant Computing*, pp. 404–414, IEEE, 1996.

[91] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, G. Goel, S. Sarkar, and R. Ganesan, "Characterization of operational failures from a business data processing SaaS platform," in *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pp. 195–204, ACM, 2014.

[92] D. Cotroneo, L. D. Simone, A. K. Iannillo, A. Lanzaro, R. Natella, J. Fan, and W. Ping, "Network Function Virtualization: Challenges and Directions for Reliability Assurance," in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 37–42, IEEE, 2014.

[93] Apache Software Foundation, Apache ActiveMQ. `http://activemq.apache.org/`. [Online; accessed 21-September-2017].

[94] Apache Software Foundation, Apache Karaf Monitoring and Management using JMX. `https://karaf.apache.org/manual/latest-3.0.x/monitoring`. [Online; accessed 21-September-2017].

[95] Z. Mwaikambo, A. Raj, R. Russell, J. Schopp, and S. Vaddagiri, "Linux kernel hotplug CPU support," in *Proceedings of the Linux Symposium - Volume Two*, pp. 467–480, 2004.

[96] Linux Foundation, tbf – Token Bucket Filter. `http://linux.die.net/man/8/tc-tbf`. [Online; accessed 21-September-2017].

[97] Linux Foundation, NetEm - Network Emulator. `http://man7.org/linux/man-pages/man8/tc-netem.8.html`. [Online; accessed 21-September-2017].

[98] Linux Foundation, iptables. `https://linux.die.net/man/8/iptables`. [Online; accessed 21-September-2017].

[99] Linux Foundation, ip. `https://linux.die.net/man/8/ip`. [Online; accessed 21-September-2017].

[100] Linux Foundation, ptrace – process trace. `http://man7.org/linux/man-pages/man2/ptrace.2.html`. [Online; accessed 21-September-2017].

[101] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad, "Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps," in *Proceedings of the 2007 Linux Symposium*, vol. one, pp. 215–224, 2007.

[102] V. Sieh, "Fault-injector using UNIX ptrace interface," Internal Report 11/93, IMMD3, Universität Erlangen Nürnberg, 1993.